LỜI NÓI ĐẦU

"Alorithms + Data Structures = Programs"
N. Wirth

"Computing is an art form. Some programs are elegant,
some are exquisite, some are sparkling.
My claim is it is possible to write grand programs,
noble programs, truly magnifient programs"
D.E.Knuth

Cuốn sách này trình bày các vấn đề cơ bản, quan trọng nhất của Cấu trúc dữ liệu (CTDL) và thuật toán đã được đề xuất trong IEEE/ACM computing curricula, theo quan điểm hiện đại.

Khi thiết kế thuật toán để giải quyết một vấn đề, chúng ta cần phải sử dụng các đối tượng dữ liệu và các phép toán trên các đối tượng dữ liệu ở mức độ trừu tượng. Một trong các nội dung chính của sách này là nghiên cứu các kiểu dữ liệu trừu tượng (KDLTT) và các CTDL để cài đặt các KDLTT. KDLTT quan trọng nhất là tập động (một tập đối tượng dữ liệu với các phép toán tìm kiếm, xen, loại, …), KDLTT này được sử dụng rộng rãi nhất trong các chương trình ứng dụng. Các KDLTT cơ bản khác sẽ được nghiên cứu là : danh sách, ngăn xếp, hàng đợi, hàng ưu tiên, từ điển, …

Chúng ta sẽ cài đặt các KDLTT bởi các lớp C + +. Sự cài đặt các KDLTT bởi các lớp C + + cho phép ta có thể biểu diễn các đối tượng dữ liệu và các phép toán trên các đối tượng dữ liệu trong các chương trình ứng dụng một cách toán học, ngắn gọn và dễ hiểu, tương tự như khi ta sử dụng các số nguyên, số thực trong chương trình. Một ưu điểm quan trọng khác là, nó cho phép khi thiết kế và cài đặt phần mềm, chúng ta có thể làm việc ở mức độ quan niệm cao, có thể thực hành được các nguyên lý lập trình.

Với mỗi KDLTT, chúng ta sẽ nghiên cứu các cách cài đặt bởi các CTDL khác nhau. Hiệu quả của các phép toán trong mỗi cách cài đặt sẽ được đánh giá. Sự đánh giá so sánh các cách cài đặt sẽ giúp cho người sử dụng có sự lựa chọn thích hợp cho từng chương trình ứng dụng. Thông qua sự cài đặt các lớp C + + cho mỗi KDLTT và các chương trình ứng dụng chúng, độc giả sẽ được cung cấp thêm nhiều kỹ thuật lập trình hữu ích.

Sự nghiên cứu mỗi KDLTT sẽ được tiến hành qua các bước sau đây.

- Đặc tả KDLTT. Chúng ta sẽ mô tả các đối tượng dữ liệu bằng cách sử dụng các ký hiệu, các khái niệm toán học và logic. Các phép toán trên các đối tượng dữ liệu sẽ được mô tả bởi các hàm toán học.
- Lựa chọn CTDL thích hợp để cài đặt đối tượng dữ liệu
- Thiết kế và cài đặt lớp C + +.
- Phân tích hiệu quả của các phép toán.
- Các ví dụ ứng dụng.

Tổ chức sách

Nội dung của cuốn sách được tổ chức thành ba phần. Phần 1 sẽ nghiên cứu các CTDL cơ bản được sử dụng để cài đặt các KDLTT, đó là danh sách liên kết (DSLK), cây tìm kiếm nhị phân (TKNP), cây thứ tự bộ phận (heap), bảng băm. Danh sách, ngăn xếp, hàng đợi sẽ được cài đặt bởi mảng hoặc bởi DSLK. Cây TKNP được sử dụng để cài đặt tập động. Hàng ưu tiên được cài đặt hiệu quả bởi heap. Bảng băm là CTDL rất thích hợp để cài đặt từ điển.

Trong phần 2 chúng ta sẽ nghiên cứu các CTDL cao cấp. Các CTDL này có đặc điểm chung là sự tổ chức dữ liệu và các phép toán trên các CTDL này là khá phức tạp, song bù lại thời gian thực hiện các phép toán lại hiệu quả hơn. Chúng ta sẽ nghiên cứu các loại cây tìm kiếm cân bằng, các CTDL tự điều chỉnh, các CTDL đa chiều, … Đặc biệt, chúng ta sẽ đưa vào kỹ thuật phân tích trả góp, đây là kỹ thuật phân tích hoàn toàn mới, được sử dụng để đánh giá thời gian chạy của một dãy phép toán trên các CTDL tự điều chỉnh.

Phần 3 dành để nói về thuật toán. Chúng ta sẽ trình bày phương pháp đánh giá thời gian chạy của thuật toán bằng ký hiệu ô lớn, và các kỹ thuật để phân tích, đánh giá thời gian chạy của thuật toán. Một nội dung quan trọng của phần này là nghiên cứu các chiến lược thiết kế thuật toán. Chúng ta sẽ trình bày các chiến lược thiết kế thuật toán hay được sử dụng là : chia - để - trị, quy hoạch động, quay lui, … Các thuật toán sắp xếp, các thuật toán đồ thị cũng sẽ được nghiên cứu. Cuối cùng chúng ta trình bày một vấn đề có tính chất lý thuyết, đó là các bài toán NP – khó và NP - đầy đủ.

Sử dụng sách

Để đọc cuốn sách này, độc giả cần phải biết lập trình định hướng đối tượng với C + +. Tuy nhiên, chúng tôi đã đưa vào các chương 2 và 3 để trình bày một số vấn đề quan trọng liên quan tới thiết kế lớp C + +, giúp cho độc giả chưa biết C + + cũng có thể hiểu được các chương tiếp theo.

Nội dung của sách này đề cập tới nhiều vấn đề hơn là nội dung của giáo trình Cấu trúc dữ liệu và thuật toán cho sinh viên công nghệ thông tin. Theo quan điểm của chúng tôi, trong giáo trình Cấu trúc dữ liệu và thuật toán cho sinh viên công nghệ thông tin, chỉ nên đưa vào các chương 1, 4, 5, 6, 7, 8, 9 của phần I và các chương 15, 16, 17, 18 của phần II. Nếu sinh viên chưa được làm quen với sự đánh giá thời gian chạy của thuật toán, thì nội dung chương 15 cần được dạy trước.

Lời cảm ơn

Chúng tôi xin chân thành cảm ơn các đồng nghiệp ở bộ môn Khoa học máy tính, Khoa công nghệ thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội, vì những trao đổi bổ ích về các vấn đề được đề cập trong sách, đặc biệt TS. Phạm Hồng Thái, ThS Trần Quốc Long và ThS Ma Thị Châu đã cùng chúng tôi giảng dạy giáo trình Cấu trúc dữ liệu và thuật toán. Chúng tôi cũng xin chân thành cảm ơn Trường Đại học công nghệ, Đại học Quốc gia Hà Nội đã tạo điều kiện tốt nhất cho chúng tôi viết cuốn sách này.

Tháng Giêng, 2007 Đinh Mạnh Tường

MỤC LỤC

PHẦN I

CÁC CẤU TRÚC DỮ LIỆU CƠ BẢN

CHƯƠNG 1

SỰ TRỪU TƯỢNG HOÁ DỮ LIỆU

Khi thiết kế thuật giải cho một vấn đề, chúng ta cần sử dụng sự trừu tượng hoá dữ liệu. Sự trừu tượng hoá dữ liệu được hiểu là chúng ta chỉ quan tâm tới một tập các đối tượng dữ liệu (ở mức độ trừu tượng) và các phép toán (các hành động) có thể thực hiện được trên các đối tượng dữ liệu đó. Với mỗi phép toán chúng ta cũng chỉ quan tâm tới điều kiện có thể sử dụng nó và hiệu quả mà nó mang lại, không cần biết nó được thực hiện như thế nào. Sự trừu tượng hoá dữ liệu được thực hiện bằng cách tạo ra các kiểu dữ liệu trừu tượng. Trong chương này chúng ta sẽ trình bày khái niệm kiểu dữ liệu trừu tượng, các phương pháp đặc tả và cài đặt kiểu dữ liệu trừu tượng.

1.1 BIỂU DIỄN DỮ LIỆU TRONG CÁC NGÔN NGỮ LẬP TRÌNH

Trong khoa học máy tính, dữ liệu được hiểu là bất kỳ thông tin nào được xử lý bởi máy tính. Dữ liệu có thể là số nguyên, số thực, ký tự, … Dữ liệu có thể có cấu trúc phức tạp, gồm nhiều thành phần dữ liệu được liên kết với nhau theo một cách nào đó. Trong bộ nhớ của máy tính, mọi dữ liệu đều được biểu diễn dưới dạng nhị phân (một dãy các ký hiệu 0 và 1). Đó là dạng biểu diễn cụ thể nhất của dữ liệu (dạng biểu diễn vật lý của dữ liệu).

Trong các ngôn ngữ lập trình bậc cao (Pascal, C, C+ +…), dữ liệu được biểu diễn dưới dạng trừu tượng, tức là dạng biểu diễn của dữ liệu xuất phát từ dạng biểu diễn toán học của dữ liệu (sử dụng các khái niệm toán học, các mô hình toán học để biểu diễn dữ liệu). Chẳng hạn, nếu dữ liệu là các điểm trong mặt phẳng, thì chúng ta có thể biểu diễn nó như một cặp số thực (x, y), trong đó số thực x là hoành độ, còn số thực y là tung độ của điểm. Do đó, trong ngôn ngữ C + +, một điểm được biểu diễn bởi cấu trúc:

```
struct   point
{  double    x;
   double    y;
};
```

Trong các ngôn ngữ lập trình bậc cao, các dữ liệu được phân thành các lớp dữ liệu (kiểu dữ liệu). Kiểu dữ liệu của một biến được xác định bởi một tập các giá trị mà biến đó có thể nhận và các phép toán có thể thực hiện trên các giá trị đó. Ví dụ, có lẽ kiểu dữ liệu đơn giản nhất và có trong nhiều ngôn ngữ lập trình là kiểu boolean, miền giá trị của kiểu này chỉ gồm hai giá trị false và true, các phép toán có thể thực hiện trên các giá trị này là các phép toán logic mà chúng ta đã quen biết.

Mỗi ngôn ngữ lập trình cung cấp cho chúng ta một số **kiểu dữ liệu cơ bản (basic data types).** Trong các ngôn ngữ lập trình khác nhau, các kiểu dữ liệu cơ bản có thể khác nhau. Ngôn ngữ lập trình Lisp chỉ có một kiểu cơ bản, đó là các S-biểu thức. Song trong nhiều ngôn ngữ lập trình khác (chẳng hạn Pascal, C / C + +, Ada, …), các kiểu dữ liệu cơ bản rất phong phú. Ví dụ, ngôn ngữ C + + có các kiểu dữ liệu cơ bản sau:

Các kiểu ký tự (char, signed char, unsigned char)

Các kiểu nguyên (int, short int, long int, unsigned)

Các kiểu thực (float, double, long double)

Các kiểu liệt kê (enum)

Kiểu boolean (bool)

Gọi là các kiểu dữ liệu cơ bản, vì các dữ liệu của các kiểu này sẽ được sử dụng như các thành phần cơ sở để kiến tạo nên các dữ liệu có cấu trúc phức tạp. Các kiểu dữ liệu đã cài đặt sẵn (build-in types) mà ngôn ngữ lập trình cung cấp là không đủ cho người sử dụng. Trong nhiều áp dụng, người lập trình cần phải tiến hành các thao tác trên các dữ liệu phức hợp. Vì vậy, mỗi ngôn ngữ lập trình cung cấp cho người sử dụng một số quy tắc cú pháp để tạo ra các kiểu dữ liệu mới từ các kiểu cơ bản hoặc các kiểu khác đã được xây dựng. Chẳng hạn, C + + cung cấp cho người lập trình các luật để xác

định các kiểu mới: kiểu mảng (array), kiểu cấu trúc (struct), kiểu con trỏ, …
Ví dụ. Từ các kiểu đã có $T_1, T_2, ..., T_n$ (có thể khác nhau), khai báo sau

```
struct      S {
   T1       M1 ;
   T2       M2 ;
   ……
   Tn       Mn ;
}
```

xác định một kiểu cấu trúc với tên là S, mỗi dữ liệu của kiểu này gồm n thành phần, thành phần thứ i có tên là M_i và có giá trị thuộc kiểu T_i (i = 1,…, n).

Các kiểu dữ liệu được tạo thành từ nhiều kiểu dữ liệu khác (các kiểu này có thể là kiểu cơ bản hoặc kiểu dữ liệu đã được xây dựng) được gọi là **kiểu dữ liệu có cấu trúc**. Các dữ liệu thuộc kiểu dữ liệu có cấu trúc được gọi là các **cấu trúc dữ liệu** (data structure). Ví dụ, các mảng, các cấu trúc, các danh sách liên kết, … là các cấu trúc dữ liệu (CTDL).

Từ các kiểu cơ bản, bằng cách sử dụng các qui tắc cú pháp kiến tạo các kiểu dữ liệu, người lập trình có thể xây dựng nên các kiểu dữ liệu mới thích hợp cho từng vấn đề. Các kiểu dữ liệu mà người lập trình xây dựng nên được gọi là các kiểu dữ liệu được xác định bởi người sử dụng (user-defined data types).

Như vậy, một CTDL là một dữ liệu phức hợp, gồm nhiều thành phần dữ liệu, mỗi thành phần hoặc là dữ liệu cơ sở (số nguyên, số thực, ký tự,…) hoặc là một CTDL đã được xây dựng. Các thành phần dữ liệu tạo nên một CTDL được liên kết với nhau theo một cách nào đó. Trong các ngôn ngữ lập trình thông dụng (Pascal, C/ C+ +), có ba phương pháp để liên kết các dữ liệu:

1. Liên kết các dữ liệu cùng kiểu tạo thành mảng dữ liệu.
2. Liên kết các dữ liệu (không nhất thiết cùng kiểu) tạo thành cấu trúc trong C/ C+ +, hoặc bản ghi trong Pascal.

3. Sử dụng con trỏ để liên kết dữ liệu. Chẳng hạn, sử dụng con trỏ chúng ta có thể tạo nên các danh sách liên kết, hoặc các CTDL để biểu diễn cây. (Chúng ta sẽ nghiên cứu các CTDL này trong các chương sau)

Ví dụ. Giả sử chúng ta cần xác định CTDL biểu diễn các lớp học. Giả sử mỗi lớp học cần được mô tả bởi các thông tin sau: tên lớp, số tổ của lớp, danh sách sinh viên của mỗi tổ; mỗi sinh viên được mô tả bởi 3 thuộc tính: tên sinh viên, tuổi và giới tính. Việc xây dựng một CTDL cho một đối tượng dữ liệu được tiến hành theo nguyên tắc sau: từ các dữ liệu có kiểu cơ sở tạo ra kiểu dữ liệu mới, rồi từ các kiểu dữ liệu đã xây dựng tạo ra kiểu dữ liệu phức tạp hơn, cho tới khi nhận được kiểu dữ liệu cho đối tượng dữ liệu mong muốn. Trong ví dụ trên, đầu tiên ta xác định cấu trúc Student

```
struct   Student
{
    string     StName;
    int        Age;
    bool       Sex;
}
```

Danh sách sinh viên của mỗi tổ có thể lưu trong mảng, hoặc biểu diễn bởi danh sách liên kết. Ở đây chúng ta dùng danh sách liên kết, mỗi tế bào của nó là cấu trúc sau:

```
struct     Cell
{
      Student       Infor;
      Cell*         Next;
}
```

Chúng ta sử dụng một mảng để biểu diễn các tổ, mỗi thành phần của mảng lưu con trỏ trỏ tới đầu một danh sách liên kết biểu diễn danh sách các sinh viên của một tổ. Giả sử mỗi lớp có nhiều nhất 10 tổ, kiểu mảng GroupArray được xác định như sau:

```
typedef     Cell*     GroupArray[10];
```

Cuối cùng, ta có thể biểu diễn lớp học bởi cấu trúc sau:

```
struct        StudentClass
{
   string     ClassName;
   int        GroupNumber;
   GroupArray Group;
}
```

1.2 SỰ TRỪU TƯỢNG HOÁ DỮ LIỆU

Thiết kế và phát triển một chương trình để giải quyết một vấn đề là một quá trình phức tạp. Thông thường quá trình này cần phải qua các giai đoạn chính sau:

1. Đặc tả vấn đề.
2. Thiết kế thuật toán và cấu trúc dữ liệu.
3. Cài đặt (chuyển dịch thuật toán thành các câu lệnh trong một ngôn ngữ lập trình, chẳng hạn C+ +)
4. Thử nghiệm và sửa lỗi.

Liên quan tới nội dung của sách này, chúng ta chỉ đề cập tới hai giai đoạn đầu. Chúng ta muốn làm sang tỏ vai trò quan trọng của sự trừu tượng hoá (abstraction) trong đặc tả một vấn đề, đặc biệt là sự trừu tương hoá dữ liệu (data abstraction) trong thiết kế thuật toán.

Vấn đề được đặt ra bởi người sử dụng thường được phát biểu không rõ ràng, thiếu chính xác. Do đó, điều đầu tiên chúng ta phải làm là chính xác hoá vấn đề cần giải quyết, hay nói một cách khác là mô tả chính xác vấn đề. Điều đó được gọi là đặc tả vấn đề. Trong giai đoạn này, chúng ta phải trả lời chính xác các câu hỏi sau. Chúng ta được cho trước những gì? Chúng ta cần tìm những gì? Những cái đã biết và những cái cần tìm có quan hệ với nhau như thế nào? Như vậy, trong giai đoạn đặc tả, chúng ta cần mô tả chính xác các dữ liệu vào (inputs) và các dữ liệu ra (outputs) của chương trình. Toán học là một ngành khoa học trừu tượng, chính xác, các khái niệm toán học, các mô hình toán học là sự trừu tượng hoá từ thế giới hiện thực. Sử dụng trừu tượng hoá trong đặc tả một vấn đề đồng nghĩa với việc chúng ta sử dụng các khái niệm toán học, các mô hình toán học và logic để biểu diễn chính xác một vấn đề.

Ví dụ. Giả sử chúng ta cần viết chương trình lập lịch thi. Vấn đề như sau. Mỗi người dự thi đăng kí thi một số môn trong số các môn tổ chức thi. Chúng ta cần xếp lịch thi, mỗi ngày thi một số môn trong cùng một thời gian, sao cho mỗi người dự thi có thể thi tất cả các môn họ đã đăng kí.
Chúng ta có thể đặc tả inputs và outputs của chương trình như sau:

Inputs: danh sách các người dự thi, mỗi người dự thi được biểu diễn bởi danh sách các môn mà anh ta đăng kí.

Outputs: danh sách các ngày thi, mỗi ngày thi được biểu diễn bởi danh sách các môn thi trong ngày đó sao cho hai môn thi bất kì trong danh sách này không thuộc cùng một danh sách các môn đăng kí của một người dự thi.

Trong mô tả trên, chúng ta đã sử dụng khái niệm danh sách (khái niệm dãy trong toán học). Các khái niệm toán học, các mô hình toán học hoặc logic được sử dụng để mô tả các đối tượng dữ liệu tạo thành các **mô hình dữ liệu (data models**). Danh sách là một mô hình dữ liệu. Chú ý rằng, lịch thi cần thoả mãn đòi hỏi: người dự thi có thể thi tất cả các môn mà họ đăng kí. Để dễ dàng đưa ra thuật toán lập lịch, chúng ta sử dụng một mô hình dữ liệu khác: đồ thị. Mỗi môn tổ chức thi là một đỉnh của đồ thị. Hai đỉnh có cạnh nối, nếu có một người dự thi đăng kí thi cả hai môn ứng với hai đỉnh đó. Từ

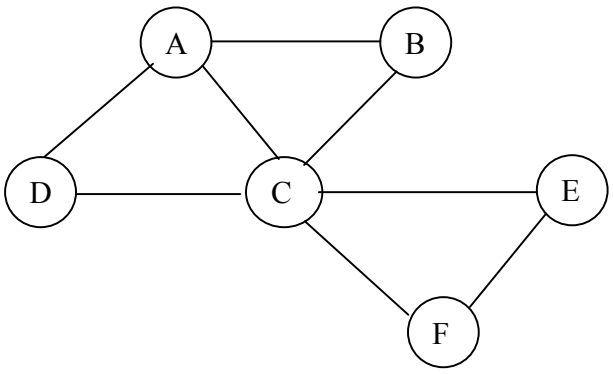
mô hình dữ liệu đồ thị này, chúng ta có thể đưa ra thuật toán lập lịch như sau:

Bước 1: Chọn một đỉnh bất kì, đưa môn thi ứng với đỉnh này vào danh sách các môn thi trong một ngày thi (danh sách này ban đầu rỗng). Đánh dấu đỉnh đã chọn và tất cả các đỉnh kề nó. Trong các đỉnh chưa đánh dấu, lại chọn một đỉnh bất kì và đưa môn thi ứng với đỉnh này vào danh sách các môn thi trong ngày thi. Lại đánh dấu đỉnh vừa chọn và các đỉnh kề nó. Tiếp tục quá trình trên cho tới khi tất cả các đỉnh của đồ thị được đánh dấu, chúng ta nhận được danh sách các môn thi trong một ngày thi.

Bước 2: Loại khỏi đồ thị tất cả các đỉnh đã xếp vào danh sách các môn thi trong một ngày thi ở bước 1 và loại tất cả các cạnh kề các đỉnh đó. Các đỉnh và các cạnh còn lại tạo thành đồ thị mới.

Bước 3: Lặp lại bước 1 và bước 2 cho tới khi đồ thị trở thành rỗng.

Chẳng hạn, giả sử các môn tổ chức thi là A, B, C, D, E, F và đồ thị xây dựng nên từ các dữ liệu vào được cho trong hình sau:



Khi đó lịch thi có thể như sau:

Ngày thi 1: A, F.

Ngày thi 2: D, E, B.

Ngày thi 3: C.

Sau khi đặc tả vấn đề, chúng ta chuyển sang giai đoạn thiết kế thuật toán để giải quyết vấn đề. Ở mức độ cao nhất của sự trừu tượng hoá, thuật toán được thiết kế như là **một dãy các hành động trên các đối tượng dữ**

liệu được thực hiện theo một trình tự logic nào đó. Thuật toán lập lịch thi ở trên là một ví dụ. Các đối tượng dữ liệu có thể là số nguyên, số thực, ký tự; có thể là các điểm trên mặt phẳng; có thể là các hình hình học; có thể là con người, có thể là danh sách các đối tượng (chẳng hạn, danh sách các môn thi trong ví dụ lập lịch thi); có thể là đồ thị, cây, …

Các hành động trên các đối tượng dữ liệu cũng rất đa dạng và tuỳ thuộc vào từng loại đối tượng dữ liệu. Chẳng hạn, nếu đối tượng dữ liệu là điểm trên mặt phẳng, thì các hành động có thể là: quay điểm đi một góc nào đó, tịnh tiến điểm theo một hướng, tính khoảng cách giữa hai điểm, … Khi đối tượng dữ liệu là danh sách, thì các hành động có thể là: loại một đối tượng khỏi danh sách, xen một đối tượng mới vào danh sách, tìm xem một đối tượng đã cho có trong danh sách hay không, …

Khi thiết kế thuật toán như là một dãy các hành động trên các đối tượng dữ liệu, chúng ta cần sử dụng **sự trừu tượng hoá dữ liệu (data abstraction).**

Sự trừu tượng hoá dữ liệu có nghĩa là chúng ta chỉ quan tâm tới một tập các đối tượng dữ liệu (ở mức độ trừu tượng) và các hành động (các phép toán) có thể thực hiện trên các đối tượng dữ liệu đó (với các điều kiện nào thì hành động có thể được thực hiện và sau khi thực hiện hành động cho kết quả gì), chúng ta không quan tâm tới các đối tượng dữ liệu đó được lưu trữ như thế nào trong bộ nhớ của máy tính, chúng ta không quan tâm tới các hành động được thực hiện như thế nào.

Sử dụng sự trừu tượng hoá dữ liệu trong thiết kế thuật toán là phương pháp luận thiết kế rất quan trọng. Nó có các ưu điểm sau:

- Đơn giản hoá quá trình thiết kế, giúp ta tránh được sự phức tạp liên quan tới biểu diễn cụ thể của dữ liệu .
- Chương trình sẽ có tính mođun (modularity). Chẳng hạn, một hành động trên đối tượng dữ liệu phức tạp được cài đặt thành một mođun (một hàm). Chương trình có tính mođun sẽ dễ đọc, dễ phát hiện lỗi, dễ sửa, …

Sự trừu tượng hoá dữ liệu được thực hiện bằng cách xác định các **kiểu dữ liệu trừu tượng (Abstract Data Type).** Kiểu dữ liệu trừu tượng (KDLTT) là một tập các đối tượng dữ liệu cùng với các phép toán có thể thực hiện trên các đối tượng dữ liệu đó. Ví dụ, tập các điểm trên mặt phẳng với các phép toán trên các điểm mà chúng ta đã xác định tạo thành KDLTT điểm.

Chúng ta có thể sử dụng các phép toán của các KDLTT trong thiết kế thuật toán khi chúng ta biết rõ các điều kiện để phép toán có thể thực hiện và hiệu quả mà phép toán mang lại. Trong nhiều trường hợp, các KDLTT mà chúng ta đã biết sẽ gợi cho ta ý tưởng thiết kế thuật toán . Đồng thời trong quá trình thiết kế, khi thuật toán cần đến các hành động trên các loại đối tượng dữ liệu mới chúng ta có thể thiết kế KDLTT mới để sử dụng không chỉ trong chương trình mà ta đang thiết kế mà còn trong các chương trình khác.

Phần lớn nội dung trong sách này là nói về các KDLTT. Chúng ta sẽ nghiên cứu sự thiết kế và cài đặt một số KDLTT quan trọng nhất được sử dụng thường xuyên trong thiết kế thuật toán .

1.3 KIỂU DỮ LIỆU TRỪU TƯỢNG

Mục này trình bày phương pháp đặc tả và cài đặt một KDLTT.

1.3.1 Đặc tả kiểu dữ liệu trừu tượng

Nhớ lại rằng, một KDLTT được định nghĩa là một tập các đối tượng dữ liệu và một tập các phép toán trên các đối tượng dữ liệu đó. Do đó, đặc tả một KDLTT gồm hai phần: đặc tả đối tượng dữ liệu và đặc tả các phép toán.

- **Đặc tả đối tượng dữ liệu**. Mô tả bằng toán học các đối tượng dữ liệu. Thông thường các đối tượng dữ liệu là các đối tượng trong thế giới hiện thực, chúng là các thực thể phức hợp, có cấu trúc nào

đó. Để mô tả chúng, chúng ta cần sử dụng sự trừu tượng hoá (chỉ quan tâm tới các đặc tính quan trọng, bỏ qua các chi tiết thứ yếu). Nói cụ thể hơn, để mô tả chính xác các đối tượng dữ liệu , chúng ta cần sử dụng các khái niệm toán học, các mô hình toán học như tập hợp, dãy, đồ thị, cây, … Chẳng hạn, đối tượng dữ liệu là sinh viên, thì có thể biểu diễn nó bởi một tập các thuộc tính quan trọng như tên, ngày sinh, giới tính, …

- **Đặc tả các phép toán**. Việc mô tả các phép toán phải đủ chặt chẽ, chính xác nhằm xác định đầy đủ kết quả mà các phép toán mang lại, nhưng không cần phải mô tả các phép toán được thực hiện như thế nào để cho kết quả như thế. Cách tiếp cận chính xác để đạt được mục tiêu trên là khi mô tả các phép toán, chúng ta xác định một tập các tiên đề mô tả đầy đủ các tính chất của các phép toán. Chẳng hạn, các phép toán cộng và nhân các số nguyên phải thoả mãn các tiên đề: giao hoán, kết hợp, phân phối, … Tuy nhiên, việc xác định một tập đầy đủ các tiên đề mô tả đầy đủ bản chất của các phép toán là cực kỳ khó khăn, do đó chúng ta mô tả các phép toán một cách không hình thức. Chúng ta sẽ mô tả mỗi phép toán bởi một hàm (hoặc thủ tục), tên hàm là tên của phép toán, theo sau là danh sách các biến. Sau đó chỉ rõ nhiệm vụ mà hàm cần phải thực hiện.

Ví dụ. Sau đây là đặc tả KDLTT số phức. Trong sách này, chúng ta sẽ đặc tả các KDLTT khác theo khuôn mẫu của ví dụ này.

Mỗi số phức là một cặp số thực (x, y), trong đó x được gọi là phần thực (real), y được gọi là phần ảo (image) của số phức.

Trên các số phức, có thể thực hiện các phép toán sau:

1. Create (a, b). Trả về số phức có phần thực là a, phần ảo là b.
2. GetReal (c). Trả về phần thực của số phức c.
3. GetImage (c). Trả về phần ảo của số phức c.
4. Abs (c). Trả về giá trị tuyệt đối (mođun) của số phức c.
5. Add (c_1,c_2). Trả về tổng của số phức c_1 và số phức c_2 .
6. Multiply (c_1 , c_2). Trả về tích của số phức c_1 và số phức c_2 .

7. Print (c). Viết ra số phức c dưới dạng a + i b trong đó a là phần thực, b là phần ảo của số phức c.

Trên đây chỉ là một số ít các phép toán số phức. Còn nhiều các phép toán khác trên số phức, chẳng hạn các phép toán so sánh, các phép toán lượng giác, …, để cho ngắn ngọn chúng ta không liệt kê ra hết.

1.3.2 Cài đặt kiểu dữ liệu trừu tượng

Trong giai đoạn đặc tả, chúng ta chỉ mới mô tả các phép toán trên các đối tượng dữ liệu, chúng ta chưa xác định các phép toán đó thực hiện nhiệm vụ của mình như thế nào. Trong chương trình, để sử dụng được các phép toán của một KDLTT đã đặc tả, chúng ta cần phải cài đặt KDLTT đó trong một ngôn ngữ lập trình.

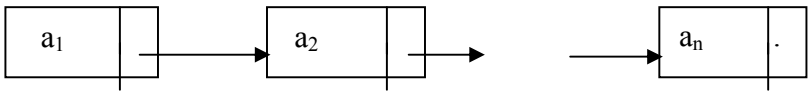
Công việc đầu tiên phải làm khi cài đặt một KDLTT là chọn một CTDL để biểu diễn các đối tượng dữ liệu. Cần lưu ý rằng, một CTDL là một dữ liệu phức hợp được tạo nên từ nhiều dữ liệu thành phần bằng các liên kết nào đó. Chúng ta có thể mô tả các CTDL trong một ngôn ngữ lập trình (chẳng hạn, C/ C + +) bằng cách sử dụng các phương tiện có sẵn trong ngôn ngữ lập trình đó, chẳng hạn sử dụng các qui tắc cú pháp mô tả mảng, cấu trúc, … Một CTDL cũng xác định cho ta cách lưu trữ dữ liệu trong bộ nhớ của máy tính.

Ví dụ. Chúng ta có thể biểu diễn một số phức bởi cấu trúc trong C + +

```
struct      complex
{
        float    real;
        float    imag;
 }
```

Cần chú ý rằng, một đối tượng dữ liệu có thể cài đặt bởi các CTDL khác nhau. Chẳng hạn, một danh sách (a_1, a_2, …, a_n) có thể cài đặt bởi mảng A, các thành phần của danh sách lần lượt được lưu trong các thành phần liên

tiếp của mảng A[0], A[1],…, A[n-1]. Nhưng chúng ta cũng có thể cài đặt danh sách bởi CTDL danh sách liên kết sau:



Sau khi đã chọn CTDL biểu diễn đối tượng dữ liệu, bước tiếp theo chúng ta phải thiết kế và cài đặt các hàm thực hiện các phép toán của KDLTT.

Trong giai đoạn thiết kế một hàm thực hiện nhiệm vụ của một phép toán, chúng ta cần sử dụng **sự trừu tượng hoá hàm (functional abstraction).** Sự trừu tượng hoá hàm có nghĩa là cần mô tả hàm sao cho người sử dụng biết được hàm thực hiện công việc gì, và sao cho họ có thể sử dụng được hàm trong chương trình của mình mà không cần biết đến các chi tiết cài đặt, tức là không cần biết hàm thực hiện công việc đó như thế nào.

Sự trừu tượng hoá hàm được thực hiện bằng cách viết ra **mẫu hàm (function prototype)** kèm theo các chú thích.

Mẫu hàm gồm tên hàm và theo sau là danh sách các tham biến. Tên hàm cần ngắn ngọn, nói lên được nhiệm vụ của hàm. Các tham biến cần phải đầy đủ: các dữ liệu vào cần thiết để hàm có thể thực hiện được công việc của mình và các dữ liệu ra sau khi hàm hoàn thành công việc.

Chú thích đưa ra sau đầu hàm là rất cần thiết (đặc biệt trong các đề án lập trình theo đội). Trong chú thích này, chúng ta cần mô tả đầy đủ, chính xác nhiệm vụ của hàm. Sau đó là hai phần: Preconditions (các điều kiện trước) và Postconditions (các điều kiện sau).

- Preconditions gồm các phát biểu về các điều kiện cần phải thoả mãn trước khi hàm thực hiện.
- Postconditions gồm các phát biểu về các điều kiện cần phải thoả mãn sau khi hàm hoàn thành thực hiện.

Hai phần Preconditions và Postconditions tạo thành **hợp đồng** giữa một bên là người sử dụng hàm và một bên là hàm. Preconditions là trách

nhiệm của người sử dụng, còn Postconditions là trách nhiệm của hàm. Một khi sử dụng hàm (gọi hàm), người sử dụng phải có trách nhiệm cung cấp cho hàm các dữ liệu vào thoả mãn các điều kiện trong Preconditions. Sau khi hoàn thành thực hiện, hàm phải cho ra các kết quả thoả mãn các điều kiện trong Postconditions. Sau đây là ví dụ một mẫu hàm:

```
void        Sort (int  A[ ], int  n)
//  Sắp xếp mảng A theo thứ tự không giảm.
//  Preconditions: A là mảng số nguyên có cỡ Max ≥ n.
//  Postconditions: A[0] ≤ A[1] ≤ …. ≤ A[n-1],
//  n không thay đổi.
```

Bước tiếp theo, chúng ta phải thiết kế thuật toán thực hiện công việc của hàm khi mà đối tượng dữ liệu được biểu diễn bởi CTDL đã chọn. Việc cài đặt hàm bây giờ là chuyển dịch thuật toán thực hiện nhiệm vụ của hàm sang dãy các khai báo biến địa phương cần thiết và các câu lệnh. Tất cả các chi tiết mà hàm cần thực hiện này là công việc riêng tư của hàm, người sử dụng hàm không cần biết đến, và không được can thiệp vào. Làm được như vậy có nghĩa là chúng ta đã thực hành **nguyên lý che dấu thông tin (the principle of information hiding)** - một nguyên lý quan trọng trong phương pháp luận lập trình môđun.

Trên đây chúng ta mới chỉ trình bày các kỹ thuật liên quan đến thiết kế CTDL cho đối tượng dữ liệu, thiết kế và cài đặt các hàm cho các phép toán của KDLTT. Câu hỏi được đặt ra là: Chúng ta phải tổ chức CTDL và các hàm đó như thế nào? Có hai cách: cách cài đặt cổ điển và cách cài đặt định hướng đối tượng. Mục sau sẽ trình bày phương pháp cài đặt KDLTT trong ngôn ngữ C. Cài đặt KDLTT bởi lớp trong C + + sẽ được trình bày trong chương 3.

1.4 CÀI ĐẶT KIỂU DỮ LIỆU TRỪU TƯỢNG TRONG C

Trong mục này chúng ta sẽ trình bày phương pháp cài đặt KDLTT theo cách truyền thống (cài đặt không định hướng đối tượng) trong C. Trong cách cài đặt này, chúng ta sẽ xây dựng nên một thư viện các hàm thực hiện các phép toán của một KDLTT sao cho bạn có thể sử dụng các hàm này trong một chương trình bất kỳ giống như bạn sử dụng các hàm trong thư viện chuẩn. Sự xây dựng một thư viện điển hình được tổ chức thành hai file: file đầu (header file) và file cài đặt (implementation file).

- File đầu chứa các mệnh đề # include, các định nghĩa hằng, … cần thiết và khai báo CTDL. Theo sau là các mẫu hàm cho mỗi phép toán của KDLTT.
- File cài đặt cũng chứa các mệnh đề # include cần thiết và chứa các định nghĩa của các hàm đã được đặc tả trong file đầu.

Tên file đầu có đuôi là .h, file cài đặt có đuôi là .c (hoặc .cpp, .cxx). File cài đặt được dịch và được kết nối vào file thực hiện được mỗi khi cần thiết. Với cách tổ chức này, bạn có thể sử dụng các hàm của một KDLTT giống hệt như bạn sử dụng các hàm trong thư viện chuẩn. Chỉ có một việc bạn cần nhớ là bạn phải include file đầu vào trong chương trình của bạn bởi mệnh đề:

include “tên file đầu”

Người sử dụng các hàm của một KDLTT chỉ cần biết các thông tin trong file đầu, không cần biết các hàm này được cài đặt như thế nào (các thông tin trong file cài đặt).

Ví dụ. Cài đặt KDLTT số phức đã đặc tả trong mục 2.3. File đầu complex.h cho trong hình 2.1. Nội dung của file này nằm giữa các mệnh đề

ifndef … # define … và # endif. Đây là các chỉ định tiền xử lý cần thiết phải có nhằm đảm bảo file đầu chỉ có mặt một lần trong file nguồn chương trình của bạn.

```
// File : complex.h
# ifndef  COMPLEX_H
```

```
# define    COMPLEX_H
          struct  Complex
          {
             double  real;
             double  image;
          };

 Complex     CreateComplex (double a, double b) ;
// Postcondition: Trả về số phức có phần thực là a, phần ảo là b.

double    GetReal (Complex c);
// Postcondition: Trả về phần thực của số phức c.

double    GetImag (Complex c);
// Postcondition: Trả về phần ảo của số phức c.

double    GetAbs (Complex c);
// Postcondition: Trả về giá trị tuyệt đối (mođun) của số phức c.

Complex    Add (Complex c1, Complex c2);
// Postcondition: Trả về số phức là tổng của số phức c1 và số phức c2.

Complex    Multiply (Complex c1, Complex c2);
// Postcondition: Trả về số phức là tích của số phức c1 và số phức c2.

void   Print (Complex c);
// Postcondition: số phức c được viết ra dưới dạng a + ib, trong đó a là
// phần thực, b là phần ảo của số phức c.

// Mẫu hàm của các phép toán khác.
# endif
```

Hình 1.1. File đầu của sự cài đặt không định hướng đối tượng của KDLTT số phức trong C/C + +.

File cài đặt KDLTT số phức được cho trong hình 1.2. Trong file cài đặt, ngoài các mệnh đề # include cần thiết cho sự cài đặt các hàm, nhất thiết phải có mệnh đề

include "tên file đầu"

```
// File: Complex.cxx
# include "complex.h"
# include < math.h >          // cung cấp hàm sqrt
# include < iostream.h >    // cung cấp đối tượng cout

Complex   CreateComplex (double a, double b)
{
    Complex  c ;
    c.real = a ;
    c.imag = b ;
    return c ;
}

double  GetReal (Complex c)
{
   return  c.real ;
}

double  GetImag (Complex c)
{
   return c.imag ;
}
```

```
double  GetAbs (Complex c)
{
   double  result ;
   result = sqrt ( c.real * c.real + c.imag * c.imag);
   return  result ;
}

Complex  Add (Complex c1, Complex c2)
{
   Complex  c ;
   c.real = c1.real + c2.real ;
   c.imag = c1.imag + c2.imag ;
   return c ;
}

Complex  Multiply (Complex c1, Complex c2)
{
   Complex  c ;
   c.real = (c1.real * c2.real) – (c1.imag * c2.imag) ;
   c.imag = (c1.real *c2.imag) + (c1.imag * c2.real) ;
   return  c ;
}

void  Print (Complex c)
{
  cout << c.real << " +i " << r.imag << " \ n" ;
}
```

Hình 1.2. File cài đặt của KDLTT số phức.

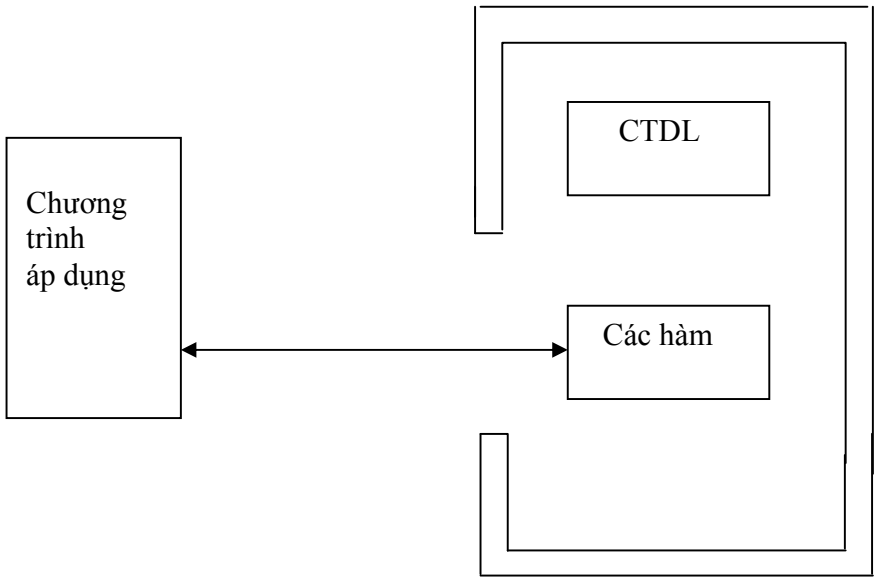
1.5 TRIẾT LÝ CÀI ĐẶT KIỂU DỮ LIỆU TRỪU TƯỢNG

Trong giai đoạn thiết kế chương trình, chúng ta cần xem xét dữ liệu dưới cách nhìn của sự trừu tượng hoá dữ liệu. Cụ thể hơn là, trong giai đoạn thiết kế chương trình, chúng ta chỉ cần quan tâm tới đặc tả của các KDLTT. Cài đặt KDLTT có nghĩa là biểu diễn các đối tượng dữ liệu bởi các CTDL và cài đặt các hàm thực hiện các phép toán trên dữ liệu. Cần nhấn mạnh rằng, chỉ có một cách nhìn logic đối với dữ liệu, nhưng có thể có nhiều cách tiếp cận để cài đặt nó. Nói một cách khác, ứng với mỗi KDLTT có thể có nhiều cách cài đặt.

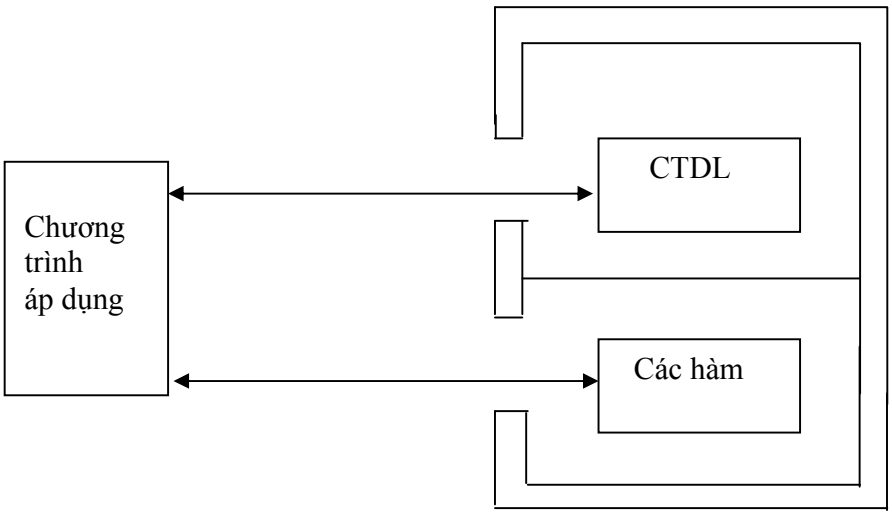
Một chương trình áp dụng cần phải sử dụng các phép toán trên dữ liệu dưới dạng biểu diễn trừu tượng, chứ không phải dưới dạng mà dữ liệu được lưu trữ trong bộ nhớ của máy tính (dạng cài đặt). Chẳng hạn, chúng ta đã sử dụng các số nguyên trong chương trình dưới dạng biểu diễn toán học của các số nguyên, và sử dụng các phép toán +, - , *, / các số nguyên với cách viết và ý nghĩa của chúng trong toán học mà không cần biết các số nguyên và các phép toán +, -, *, / được cài đặt như thế nào trong máy tính. (Cần biết rằng, các số nguyên cùng với các phép toán trên số nguyên: +, - , *, / tạo thành một KDLTT và KDLTT này đã được cài đặt sẵn, chúng ta chỉ việc sử dụng.)

Do vậy, khi cài đặt KDLTT chúng ta cần tạo ra một giao diện (interface) giữa chương trình áp dụng và sự cài đặt KDLTT. Giao diện này bao gồm các phép toán đã xác định trong KDLTT. Người sử dụng chỉ được phép giao tiếp với sự cài đặt KDLTT thông qua giao diện này. Nói một cách hình ảnh thì cách cài đặt KDLTT cần phải sao cho tạo thành bức tường giữa chương trình áp dụng và sự cài đặt KDLTT. Bức tường này che chắn CTDL, chỉ có thể truy cập tới CTDL thông qua các phép toán đã đặc tả trong KDLTT (hình 1.3.a). Nếu thực hiện được sự cài đặt KDLTT như thế, thì khi ta thay đổi CTDL biểu diễn đối tượng dữ liệu và thay đổi cách cài đặt các hàm cũng không ảnh hưởng gì đến chương trình áp dụng sử dụng KDLTT này. Điều này tương tự như khi ta sử dụng máy bán nước giải khát tự động.

Thiết kế bên ngoài và các chỉ dẫn sẽ cho phép người sử dụng mua được loại nước mà mình mong muốn. Chẳng hạn, có ba nút ấn: cam, côca, cà phê. Bỏ 5 xu vào lỗ dưới nút cam và ấn nút cam, người sử dụng sẽ nhận được cốc cam ,… Người sử dụng không cần biết đến cấu tạo bên trong của máy.Chúng ta có thể thay đổi cấu tạo bên trong của máy, miễn là nó vẫn đáp ứng được mong muốn của người sử dụng khi họ thực hiện các thao tác theo chỉ dẫn.



a. Cài đặt định hướng đối tượng



b. Cài đặt không định hướng đối tượng

Hình 1.3. Chương trình áp dụng và sự cài đặt KDLTT.

Cách cài đặt truyền thống (cài đặt không định hướng đối tượng) đã tách biệt CTDL và các hàm. Người sử dụng vẫn có thể truy cập trực tiếp đến CTDL không cần thông qua các hàm (hình 1.3.b). Nói một cách khác, cách cài đặt truyền thống không tạo thành bức tường vững chắc che chắn CTDL. Điều này có thể dẫn tới các lỗi không kiểm soát được, khó phát hiện. Chỉ có cách cài đặt định hướng đối tượng mới đảm bảo yêu cầu tạo ra bức tường giữa chương trình áp dụng và sự cài đặt KDLTT.

Trong cách cài đặt định hướng đối tượng sử dụng C + +, CTDL và các hàm thực hiện các phép toán trên dữ liệu được đóng gói (encapsulation) vào một thực thể được gọi là lớp. Lớp có cơ chế điều khiển sự truy cập đến CTDL. Mỗi lớp cung cấp cho người sử dụng một giao diện. Chương trình áp dụng chỉ có thể truy cập đến CTDL qua giao diện này (bao gồm các hàm trong mục public). Cài đặt KDLTT bởi lớp C + + cho phép ta khi viết các chương trình ứng dụng, có thể biểu diễn các phép toán trên các đối tượng dữ liệu dưới dạng các biểu thức toán học rất sáng sủa, dễ hiểu. Cách cài đặt KDLTT bởi lớp C + + còn cho phép thực hành sử dụng lại phần mềm

(reusability). Chúng ta sẽ nghiên cứu phương pháp cài đặt KDLTT bởi lớp trong chương sau.

BÀI TẬP

Trong các bài tập sau đây, hãy đặc tả các KDLTT bằng cách thực hiện hai phần sau:

- Mô tả đối tương dữ liệu bằng cách sử dụng các ký hiệu và các khái niệm toán học.
- Mô tả các phép toán trên các đối tượng dữ liệu đó. Cần biểu diễn mỗi phép toán bởi hàm, nói rõ mục tiêu của hàm, các điều kiện để hàm thực hiện được mục tiêu đó, và hiệu quả mà hàm mang lại.

1. KDLTT tập hợp với các phép toán: hợp, giao, hiệu, kiểm tra một tập có rỗng không, kiểm tra một đối tượng có là phần tử của một tập, kiểm tra hai tập có bằng nhau không, kiểm tra một tập có là tập con của một tập khác không.
2. KDLTT điểm (điểm trên mặt phẳng). Các phép toán gồm: tịnh tiến điểm, quay điểm đi một góc, tính khoảng cách giữa hai điểm, xác định điểm giữa hai điểm.
3. KDLTT hình cầu.
4. KDLTT ma trận.
5. KDLTT phân số.
6. KDLTT xâu ký tự.

Trong các bài tập 3 → 6, hãy tự xác định các phép toán (càng đầy đủ càng tốt), và hãy đặc tả các phép toán bởi các hàm.

CHƯƠNG 2

KIỂU DỮ LIỆU TRỪU TƯỢNG VÀ CÁC LỚP C + +

Mục đích của chương này là trình bày khái niệm lớp và các thành phần của lớp trong C + +. Sự trình bày sẽ không đi vào chi tiết, mà chỉ đề cập tới các vấn đề quan trọng liên quan tới các thành phần của lớp giúp cho bạn đọc dễ dàng hơn trong việc thiết kế các lớp khi cài đặt các KDLTT. Chương này cũng trình bày khái niệm lớp khuôn, lớp khuôn được sử dụng để cài đặt các lớp côngtơnơ. Cuối chương chúng ta sẽ giới thiệu các KDLTT quan trọng sẽ được nghiên cứu kỹ trong các chương sau.

2.1 LỚP VÀ CÁC THÀNH PHẦN CỦA LỚP

Các ngôn ngữ lập trình định hướng đối tượng, chẳng hạn C + +, cung cấp các phương tiện cho phép đóng gói CTDL và các hàm thao tác trên CTDL thành một đơn vị được gọi là lớp (class). Ví dụ, sau đây là định nghĩa lớp số phức:

```
class       Complex
{
 public :
        (1)  Complex (double a = 0.0 , double b = 0.0) ;
        (2)  Complex (const Complex & c);
        (3)  double GetReal ( )  const ;
        (4)  double GetImag ( )  const ;
        (5)  double GetAbs ( )  const ;
        (6)  friend Complex & operator + (const Complex & c1,
```

```
                                                const   Complex & c2) ;
        (7)   friend Complex & operator - (const Complex & c1,
                                                const Complex & c2) ;
        (8)   friend Complex & operator * (const Complex & c1,
                                                const Complex & c2) ;
        (9)   friend Complex & operator / (const Complex & c1,
                                                const Complex & c2) ;
       (10)   friend ostream & operator << (ostream & os,
                                                const Complex & c);
              // Các mẫu hàm cho các phép toán khác.
    private:
          double  real ;
          double  imag ;
} ;
```

Từ ví dụ đơn giản trên, chúng ta thấy rằng, một lớp bắt đầu bởi **đầu lớp**: đầu lớp gồm từ khoá class, rồi đến tên lớp. Phần còn lại trong định nghĩa lớp (nằm giữa cặp dấu { và }) là **danh sách thành phần**. Danh sách thành phần gồm các **thành phần dữ liệu (data member**), hay còn gọi là **biến thành phần (member variable**), chẳng hạn lớp Complex có hai biến thành phần là real và imag. Các thành phần (1) – (5) trong lớp Complex là các **hàm thành phần (member functions hoặc methods).**

Một lớp là một kiểu dữ liệu, ví dụ khai báo lớp Complex như trên, có nghĩa là người lập trình đã xác định một kiểu dữ liệu Complex. Các đối tượng dữ liệu thuộc một lớp được gọi là các **đối tượng (objects).**

Các thành phần của lớp điển hình được chia thành hai mục: mục public và mục private như trong định nghĩa lớp Complex. Trong chương trình, người lập trình có thể sử dụng trực tiếp các thành phần trong mục public để tiến hành các thao tác trên các đối tượng của lớp. Các thành phần trong mục private chỉ được phép sử dụng trong nội bộ lớp. Mục public (mục private) có thể chứa các hàm thành phần và các biến thành phần. Tuy nhiên,

khi cần thiết kế một lớp cài đặt một KDLTT, chúng ta nên đưa các biến thành phần mô tả CTDL vào mục private, còn các hàm biểu diễn các phép toán vào mục public. Trong định nghĩa lớp Complex cài đặt KDLTT số phức, chúng ta đã làm như thế.

Nên biết rằng, các thành phần của lớp có thể khai báo là tĩnh bằng cách đặt từ khoá static ở trước. Trong một lớp, chúng ta có thể khai báo các hằng tĩnh, các biến thành phần tĩnh, các hàm thành phần tĩnh. Chẳng hạn:

```
static  const  int  CAPACITY = 50; // khai báo hằng tĩnh
static  double  static  Var; // khai báo biến tĩnh
```

Các **thành phần tĩnh** là các thành phần được dùng chung cho tất cả các đối tượng của lớp. Trong lớp Complex không có thành phần nào cần phải là tĩnh.

Nếu khai báo của hàm trong một lớp bắt đầu bởi từ khoá friend, thì hàm được nói là bạn của lớp, chẳng hạn các hàm (6) – (10) trong lớp Complex. Một **hàm bạn (friend function**) không phải là hàm thành phần, song nó được phép truy cập tới các thành phần dữ liệu trong mục private của lớp.

Một hàm thành phần mà khai báo của nó có từ khoá const ở sau cùng được gọi là **hàm thành phần hằng (const member function**). Một hàm thành phần hằng có thể xem xét trạng thái của đối tượng, song không được phép thay đổi nó. Chẳng hạn, các hàm (3), (4), (5) trong lớp Complex. Các hàm này khi áp dụng vào một số phức, không làm thay đổi số phức mà chỉ cho ra phần thực, phần ảo và mođun của số phức, tương ứng.

2.2 CÁC HÀM THÀNH PHẦN

Trong mục này chúng ta sẽ xem xét một số đặc điểm của hàm thành phần.

2.2.1 Hàm kiến tạo và hàm huỷ

Một chương trình áp dụng sử dụng đến các lớp (cần nhớ rằng lớp là một kiểu dữ liệu) sẽ tiến hành một dãy các thao tác trên các đối tượng được khai báo và được tạo ra ban đầu. Do đó, trong một lớp cần có một số hàm thành phần thực hiện công việc khởi tạo ra các đối tượng. Các hàm thành phần này được gọi là **hàm kiến tạo (constructor**). Hàm kiến tạo có đặc điểm là tên của nó trùng với tên lớp và không có kiểu trả về, chẳng hạn hàm (1), (2) trong lớp Complex.

Nếu trong một lớp, bạn không định nghĩa một hàm kiến tạo, thì chương trình dịch sẽ tự động tạo ra một **hàm kiến tạo mặc định tự động (automatic default constructor**). Hàm này chỉ tạo ra đối tượng với tất cả các thành phần dữ liệu đều bằng 0. Nói chung, rất ít khi người ta thiết kế một lớp không có hàm kiến tạo. Đặc biệt khi bạn thiết kế một lớp có chứa thành phần dữ liệu là đối tượng của một lớp khác, thì nhất thiết bạn phải viết hàm kiến tạo.

Một loại hàm kiến tạo đặc biệt có tên gọi là **hàm kiến tạo copy** (**copy constructor**). Nhiệm vụ của hàm kiến tạo copy là khởi tạo ra một đối tượng mới là bản sao của một đối tượng đã có. Ví dụ, hàm (2) trong lớp Complex là hàm kiến tạo copy. Hàm kiến tạo copy chỉ có một tham biến tham chiếu hằng có kiểu là kiểu lớp đang định nghĩa.

Nếu bạn không đưa vào một hàm kiến tạo copy trong định nghĩa lớp, thì chương trình dịch sẽ tự động tạo ra một **hàm kiến tạo copy tự động** (**automatic copy constructor**). Nó thực hiện sao chép tất cả các thành phần dữ liệu của đối tượng đã có sang đối tượng đang khởi tạo. Nói chung, trong nhiều trường hợp chỉ cần sử dụng hàm kiến tạo copy tự động là đủ. Chẳng hạn, trong lớp Complex, thực ra không cần có hàm kiến tạo copy (2). Song trong trường hợp lớp chứa các biến thành phần là biến con trỏ, thì cần thiết phải thiết kế hàm kiến tạo copy cho lớp. (Tại sao?)

Sau đây là một số ví dụ sử dụng hàm kiến tạo trong khai báo các đối tượng thuộc lớp Complex:

Complex c_1; // khởi tạo số phức c_1 với c_1.real = 0.0 và c_1.imag = 0.0

Complex c_2(2.6); // khởi tạo số phức c_2 với c_2.real = 2.6
// và c_2.imag = 0.0
Complex c_3(5.4, 3.7); // khởi tạo số phức c_3 với c_3.real =5.4
// và c_3.imag = 3.7
Complex c_4 = c_2; // khởi tạo số phức c_4 là copy của c_2.

Ngược lại với hàm kiến tạo là **hàm huỷ (destructor**). Hàm huỷ thực hiện nhiệm vụ huỷ đối tượng (thu hồi vùng nhớ cấp phát cho đối tượng và trả lại cho hệ thống), khi đối tượng không cần thiết cho chương trình nữa. Hàm huỷ là hàm thành phần có tên trùng với tên lớp, không có tham biến và phía trước có dấu ngã ~. Hàm huỷ tự động được gọi khi đối tượng ra khỏi phạm vi của nó. Trong một định nghĩa lớp chỉ có thể có một hàm huỷ. Nói chung, trong một lớp không cần thiết phải đưa vào hàm huỷ (chẳng hạn, lớp Complex), trừ trường hợp lớp chứa thành phần dữ liệu là con trỏ trỏ tới vùng nhớ cấp phát động.

2.2.2 Các tham biến của hàm

Các hàm thành phần của một lớp cũng như các hàm thông thường khác có một danh sách các tham biến (danh sách này có thể rỗng) được liệt kê sau tên hàm trong khai báo hàm. Các tham biến này được gọi là **tham biến hình thức (formal parameter**). Khi gọi hàm, các tham biến hình thức được thay thế bởi các **đối số (argument**) hay còn gọi là các **tham biến thực tế (actual parameter).**

Chúng ta xem xét ba loại tham biến:

- **Tham biến giá trị**: Tham biến giá trị (**value parameter**) được khai báo bằng cách viết tên kiểu theo sau là tên tham biến. Chẳng hạn, trong hàm kiến tạo của lớp Complex:

  Complex (double a = 0.0, double b = 0.0) ;

  thì a và b là các tham biến giá trị. Trong khai báo trên chúng ta đã xác định các đối số mặc định (default argument) cho các tham biến a

và b, chúng đều là 0.0. Khi chúng ta gọi hàm kiến tạo không đưa vào đối số, thì có nghĩa là đã gọi hàm kiến tạo với đối số mặc định. Ví dụ, khi ta khai báo Complex c ; thì số phức c được khởi tạo bằng gọi hàm kiến tạo với các đối số mặc định (số phức c có phần thực và phần ảo đều là 0.0).

- **Tham biến tham chiếu**: Tham biến tham chiếu (**reference parameter**) được khai báo bằng cách viết tên kiểu theo sau là dấu & rồi đến tên tham biến. Chẳng hạn, chúng ta có thể thiết kế hàm cộng hai số phức như sau:

void Add (Complex c_1, Complex c_2, Complex & c) ;

Trong hàm Add này, c_1 và c_2 là tham biến giá trị kiểu Complex, còn c là tham biến tham chiếu kiểu Complex.

Để hiểu được sự khác nhau giữa tham biến giá trị và tham biến tham chiếu, bạn cần biết cơ chế thực hiện một lời gọi hàm trong bộ nhớ của máy tính. Mỗi khi một hàm được gọi trong chương trình thì một vùng nhớ dành cho sự thực hiện hàm có tên gọi là **bản ghi hoạt động** (**activation record**) được tạo ra trên vùng nhớ ngăn xếp thời gian chạy (run-time staek). Bản ghi hoạt động ngoài việc chứa bộ nhớ cho các biến địa phương trong hàm, nó còn lưu bản sao của các đối số ứng với các tham biến giá trị và chỉ dẫn tới các đối số ứng với các tham biến tham chiếu. Như vậy, khi thực hiện một lời gọi hàm, các đối số ứng với tham biến giá trị sẽ được copy vào bản ghi hoạt động, còn các đối số ứng với tham biến tham biến thì không cần copy. Khi hoàn thành sự thực hiện hàm, thì bản ghi hoạt động được trả về cho ngăn xếp thời gian chạy. Do đó, sau khi thực hiện hàm, đối số ứng với tham biến giá trị không thay đổi giá trị vốn có của nó, còn đối số ứng với các tham biến tham chiếu vẫn lưu lại kết quả của các tính toán khi thực hiện hàm. Bởi vậy, các tham biến ghi lại kết quả của sự thực hiện hàm cần được khai báo là tham biến tham chiếu.

Trong hàm Add tham biến c ghi lại tổng của số phức c_1 và c_2, nên nó đã được khai báo là tham biến tham chiếu.

Trên đây là một cách sử dụng toán tử tham chiếu (&): nó được sử dụng để khai báo các tham biến tham chiếu. Một cách sử dụng khác của toán tử tham chiếu là để khai báo **kiểu trả về tham chiếu** (**reference return type**) cho một hàm. Ví dụ, chúng ta có thể thiết kế một hàm thực hiện phép cộng số phức một cách khác như sau:

Complex & Add (Complex c_1, Complex c_2) ;

Khai báo kiểu trả về của một hàm là kiểu trả về tham chiếu khi nào? Cần lưu ý rằng, khi thực hiện một hàm, giá trị trả về của hàm được lưu trong một biến địa phương, rồi mệnh đề return sẽ trả về một copy của biến này cho chương trình gọi hàm. Bởi vậy, khi đối tượng trả về của một hàm là lớn, để tránh phải copy từ ngăn xếp thời gian chạy, kiểu trả về của hàm đó nên được khai báo là kiểu trả về tham chiếu.

- **Tham biến tham chiếu hằng**: Như trên đã nói, tham biến tham chiếu ưu việt hơn tham biến giá trị ở chỗ khi thực hiện một hàm, đối số ứng với tham biến tham chiếu không cần phải copy vào ngăn xếp thời gian chạy, nhưng giá trị của nó có thể bị thay đổi, trong khi đó giá trị của đối số ứng với tham biến giá trị không thay đổi khi thực hiện hàm. Kết hợp tính hiệu quả của tham biến tham chiếu và tính an toàn của tham biến giá trị, người ta đưa vào loại tham biến tham chiếu hằng. Để xác định một tham biến tham chiếu hằng (**const reference parameter**), chúng ta chỉ cần đặt từ khoá const trước khai báo tham biến tham chiếu. Đối với tham biến tham chiếu hằng, trong thân hàm chúng ta chỉ có thể tham khảo nó, mọi hành động làm thay đổi giá trị của nó đều không được phép. Khi mà tham biến giá trị có kiểu dữ liệu lớn, để cho hiệu quả chúng ta có thể sử dụng tham biến tham chiếu hằng để thay thế.

Ví dụ, bạn có thể khai báo một hàm tính tổng của hai số phức như sau:

Complex & Add (const Complex & c_1, const Complex & c_2) ;

Trong hàm Add này, c_1 và c_2 là hai tham biến tham chiếu hằng, do đó trong thân của hàm chỉ được phép đọc c_1, c_2, không được phép làm thay đổi chúng.

2.2.3 Định nghĩa lại các phép toán

Giả sử trong định nghĩa lớp Complex, chúng ta xác định các hàm tính tổng và tích của hai số phức như sau:

Complex & Add (const Complex & c_1, const Complex & c_2) ;

Complex & Multiply (const Complex & c_1, const Complex & c_2) ;

Khi đó trong chương trình muốn lấy số phức A cộng với tích của số phức B và số phức C, ta cần viết:

D = Add (A, Multiply (B, C)) ;

Cách viết này rất không sáng sủa, nhất là đối với các tính toán phức tạp hơn trên các số phức.

Chúng ta mong muốn biểu diễn các tính toán trên các số phức trong chương trình bởi các biểu thức toán học. Chẳng hạn, dòng lệnh trên, nếu được viết thành:

D = A + B * C ;

thì chương trình sẽ trở nên sáng sủa hơn, dễ đọc, dễ hiểu hơn. Sử dụng các công cụ mà C + + cung cấp, chúng ta có thể làm được điều đó.

Trong ngôn ngữ lập trình C + + có rất nhiều các phép toán (toán tử). Chẳng hạn, các phép toán số học +, - , * , / , % ; các phép toán so sánh = =, != , < , <= , > , >= , các toán tử gán và rất nhiều các phép toán khác. Các phép toán này có ngữ nghĩa đã được xác định trong ngôn ngữ. Chúng ta muốn sử dụng các ký hiệu phép toán trong C + +, nhưng với ngữ nghĩa hoàn

toàn mới, chẳng hạn chúng ta muốn sử dụng ký hiệu + để chỉ phép cộng số phức hoặc phép cộng vectơ hoặc phép cộng ma trận … Việc xác định lại ngữ nghĩa của các phép toán (toán tử) trên các lớp đối tượng dữ liệu mới sẽ được gọi là **định nghĩa lại các phép toán (operator overloading).**

Các phép toán được định nghĩa lại bởi các hàm có tên hàm bắt đầu bởi từ khoá operator và đi sau là ký hiệu phép toán, chúng ta sẽ gọi các hàm này là **hàm toán tử**. Ví dụ, chúng ta có thể định nghĩa lại phép toán + cho các số phức. Có ba cách định nghĩa phép toán cộng số phức bởi hàm toán tử operator +

- **Hàm toán tử không phải là hàm thành phần của lớp Complex**:

```
Complex & Operator + ( const Complex & c1, const Complex & c2);
{
   double  x , y ;
   x = c1. GetReal ( ) + c2. GetReal ( ) ;
   y = c1. GetImag ( ) + c2. GetImag ( ) ;
   Complex  c(x,y) ;
   return  c ;
}
```

Khi đó, trong chương trình muốn cộng hai số phức, ta có thể viết như sau:

```
   Complex A (3.5, 2.7) ;
   Complex B (-4.3, 5.8) ;
   Complex C ;
   C = A + B ;
```

Cũng có thể viết C = operator + (A, B), nhưng không nên sử dụng cách này.

- **Hàm toán tử là hàm thành phần của lớp Complex**

```
Complex & Complex :: operator + (const Complex & c)
{
  Complex temp ;
  temp.real = real + c.real ;
  temp.imag = imag + c. imag ;
  return temp ;
}
```

Trong cách này, khi ta viết C = A + B, thì toán hạng thứ nhất (số phức A) là đối tượng kích hoạt hàm toán tử, tức là

C = A.operator + (B).

- **Hàm toán tử là hàm bạn của lớp Complex**. Đây là cách mà chúng ta đã lựa chọn trong định nghĩa lớp Complex (xem mục 2.1.). Hàm bạn này được cài đặt như sau:

```
Complex & operator + (const Complex & c1, const Complex & c2);
{
  Complex  sum ;
  sum.real = c1.real + c2.real ;
  sum.imag = c1.imag + c2.imag ;
  return  sum ;
}
```

Sử dụng hàm toán tử là bạn giống như sử dụng hàm toán tử không phải thành phần của lớp.

Có sự khác nhau tinh tế giữa hàm toán tử thành phần và hàm toán tử bạn. Ví dụ, giả sử A và B là hai số phức, và hàm operator + là hàm bạn của lớp Complex. Khi đó, câu lệnh:

A = 1 + B ;

được chương trình dịch xem là:

A = operator+ (1,B) ;

và để thực hiện, 1 được chuyển thành đối tượng Complex với phần thực bằng 1, phần ảo bằng 0 bởi toán tử chuyển kiểu được xác định trong lớp Complex, rồi được cộng với số phức A.

Chúng ta xét xem điều gì sẽ xảy ra khi hàm toán tử operator + là hàm thành phần của lớp Complex. Trong trường hợp này, chương trình dịch sẽ minh họa A = 1 + B như là

A = 1. operator (B) ;

Nhưng 1 không phải là đối tượng của lớp Complex, do đó nó không thể kích hoạt một hàm thành phần của lớp Complex. Điều này dẫn tới lỗi!

Vì những lý do trên, khi thiết kế một lớp cài đặt một KDLTT thì các phép toán hai toán hạng (chẳng hạn, các phép cộng, trừ, nhân, chia số phức) nên được cài đặt bởi hàm toán tử bạn của lớp.

Trong một lớp cài đặt một KDLTT, nói chung ta cần đưa vào một hàm viết ra đối tượng dữ liệu trên các thiết bị ra chuẩn. C + + đã đưa vào toán tử << để viết ra các số nguyên, số thực, ký tự, … Chúng ta có thể định nghĩa lại toán tử << để viết ra các đối tượng dữ liệu phức hợp khác, chẳng hạn để viết ra các số phức trong lớp Complex. Trong lớp Complex, hàm operator << được thiết kế là hàm bạn với khai báo sau:

```
ostream &   operator << (ostream & os, const Complex & c) ;
```

trong đó ostream là lớp các luồng dữ liệu ra (output stream), ostream là thành viên của thư viện iostream.h, và cout (thiết bị ra chuẩn) là một đối tượng của lớp ostream.

Sau đây là cài đặt hàm toán tử bạn operator << trong lớp Complex:

```
ostream &   operator << (ostream & os, const Complex & c)
// Postcondition. Số phức c được viết ra luồng os, dưới dạng a + ib,
// trong đó a là phần thực và b là phần ảo của số  phức c.
// Giá trị trả về là luồng ra os.
{
```

```
  os << c.real << " + i" << c.imag ;
  return os ;
}
```

Trên đây chúng ta đã xét cách cài đặt các hàm toán tử định nghĩa lại các phép toán + và << cho các số phức. Các ví dụ đó cũng cho bạn một phương pháp chung để khi thiết kế một lớp cài đặt một KDLTT, bạn có thể cài đặt một phép toán hai toán hạng bởi một hàm toán tử định nghĩa lại các phép toán trong C + +.

Hầu hết các phép toán, các toán tử trong C + + đều có thể định nghĩa lại. Tuy nhiên, khi thiết kế các lớp cài đặt các KDLTT, thông thường chúng ta chỉ cần đến định nghĩa lại các phép toán số học: +, - , * , / , các phép toán so sánh = = , ! = , < , < = , > , > = , các toán tử gán = , + = , - = , * = , / = .

2.3 PHÁT TRIỂN LỚP C + + CÀI ĐẶT KDLTT

Trong mục này chúng ta sẽ trình bày một ví dụ về lớp Complex, qua đó bạn đọc sẽ thấy cần phải làm gì để phát triển một lớp C + + cài đặt một KDLTT. Phần cuối của mục sẽ trình bày các hướng dẫn cài đặt KDLTT bởi lớp.

Một lớp khi được khai báo sẽ là một kiểu dữ liệu được xác định bởi người sử dụng. Vì vậy, bạn có thể khai báo một lớp trong chương trình và sử dụng nó trong chương trình giống như khai báo và sử dụng các kiểu dữ liệu quen thuộc khác. Hình 2.1 là một chương trình demo cho việc khai báo và sử dụng lớp Complex. Chú ý rằng, khi cài đặt các hàm thành phần của một lớp, bạn cần sử dụng toán tử định phạm vi để chỉ nó thuộc lớp đó ở đây bạn phải đặt Complex :: trước tên hàm.

```
# include  < math.h >
# include  < iostream.h >
```

```
class  Complex
{
public :
Complex (double a = 0, double b = 0) ;
// Tạo ra số phức có phần thực a, phần ảo b
double    GetReal( ) const ;
// Trả về phần thực của số phức.
double    GetImag ( ) const ;
// Trả về phần ảo của số phức.
double    GetAbs ( ) const ;
// Trả về giá trị tuyệt đối của số phức.
friend Complex & operator +(const Complex & c1,const Complex &c2);
// Trả về tổng của số phức c1 và c2.
friend Complex & operator -(const Complex & c1,const Complex & c2);
// Trả về hiệu của số phức c1 và c2.
friend Complex & operator *(const Complex & c1,const Complex & c2);
// Trả về tích của số phức c1 và c2.
friend Complex & operator /(const Complex & c1, const Complex & c2);
// Trả về thương của số phức c1 và c2.
 friend ostream &   operator << (ostream & os, const Complex &c);
 // In số phức c trên luồng ra os.
// Các mẫu hàm khác.
private :
      double real ;
      double imag ;
} ;

int main ( )
{
      Complex   A (3.2, 5.7) ;
      Complex   B (6.3, -4.5) ;
```

```
    cout << "Phần thực của số phức A:" << A.GetReal( ) << endl;
    cout << "Phần ảo của số phức A:" << A.GetImag ( ) << endl ;
    A = A + B ;
    cout << A << endl ;  // In ra số phức A.
    return   0 ;
}

// Sau đây là cài đặt các hàm đã khai báo trong lớp Complex
Complex :: Complex (double a = 0, double b = 0)
{
    real = a ;
    imag = b ;
}
double Complex :: GetReal ( )
{
    return real ;
}
// Cài đặt các hàm còn lại trong lớp Complex.
```

Hình 2.1. Chương trình demo về khai báo và sử dụng lớp.

Tuy nhiên chúng ta thiết kế một KDLTT và cài đặt nó bởi lớp C + + là để sử dụng trong một chương trình bất kỳ cần đến KDLTT đó, do đó khi phát triển một lớp cài đặt một KDLTT, chúng ta cần phải tổ chức thành hai file: file đầu và file cài đặt (tương tự như chúng ta đã làm khi cài đặt không định hướng đối tượng KDLTT, xem mục 1.4).

- **File đầu**: File đầu có tên kết thúc bởi .h. File đầu chứa tất cả các thông tin cần thiết mà người lập trình cần biết để sử dụng KDLTT trong chương trình của mình. Chúng ta sẽ tổ chức file đầu như sau: Đầu tiên là các chú thích về các hàm trong mục public của lớp. Mỗi chú thích về một hàm bao gồm mẫu hàm và các điều kiện trước, điều

kiện sau kèm theo mỗi hàm. Người sử dụng lớp chỉ cần đọc các thông tin trong phần chú thích này. Tiếp theo là định nghĩa lớp. Chú ý rằng, định nghĩa lớp cần đặt giữa các định hướng tiền xử lý # ifndef # define … # endif. Chẳng hạn, định nghĩa lớp Complex như sau:

```
# ifndef  COMPLEX_H
# define  COMPLEX_H
      class    Complex
      {
         // danh sách thành phần
       } ;
# endif
```

File đầu của lớp Complex được cho trong hình 2.2. Cần lưu ý rằng, trong lớp Complex đó, chúng ta mới chỉ đưa vào một số ít phép toán, để thuận lợi cho việc tiến hành các thao tác số phức, lớp Complex thực tế cần phải chứa rất nhiều phép toán khác, song để cho ngắn gọn, chúng ta đã không đưa vào.

```
// File đầu Complex.h
// Các hàm kiến tạo :
// Complex (double a = 0.0, double b = 0.0) ;
// Postcondition: số phức được tạo ra có phần thức là a, phần ảo là b.
// Các hàm thành phần khác:
// double GetReal ( ) const ;
// Trả về phần thực của số phức.
// double GetImag ( ) const ;
// Trả về phần ảo của số phức.
// double GetAbs ( ) const ;
// Trả về giá trị tuyệt đối của số phức.
// Các hàm bạn:
```

```
// friend Complex & operator + (const Complex & c1,
//                                                const Comple  & c2) ;
// Trả về tổng c1 + c2 của số phức c1 và c2.
// friend Complex & operator - (const Complex & c1,
//                                                const Complex  & c2);
// Trả về hiệu c1 – c2 của số phức c1 và c2.
// friend Complex & operator * (const Complex & c1,
//                                                const Complex & c2);
// Trả về tích c1 * c2 của số phức c1 và c2.
// friend Complex & operator / (const Complex & c1,
//                                                 const Complex &c2);
// Trả về thương c1 / c2 của số phức c1 và c2.
// friend ostream & operator << (ostream & os, const Complex &c);
// Postcondition: số phức c được in ra dưới dạng a + ib, trong đó a là
// phần thực, b là phần ảo của c.

# ifndef  COMPLEX_H
# define  COMPLEX_H
        class    Complex
        {
            public :
            Complex (double a = 0.0, double b = 0.0) ;
            double GetReal ( ) const ;
            double GetImag ( ) const ;
            double GetAbs ( ) const ;
            friend Complex & operator + (const Complex & c1, const
                                                Complex & c2) ;
            friend Complex & operator - (const Complex & c1,  const
                                                Complex & c2) ;
            friend Complex & operator * (const Complex & c1,  const
                                                Complex & c2) ;
```

```
        friend Complex & operator / (const Complex & c1,  const
                                     Complex & c2) ;
        friend ostream & operator << (ostream & os, const
                                      Complex & c) ;
        private :
           double   real ;
           double   imag ;
    } ;
# endif
```

Hình 2.2. File đầu của lớp Complex

- **File cài đặt**. Hầu hết các chương trình dịch đòi hỏi file cài đặt có tận cùng là .cpp hoặc .c. Trong file cài đặt trước hết cần có mệnh đề # include "tên file đầu" và các mệnh đề # include khác, chẳng hạn # include <math.h>, …, khi mà các file thư viện chuẩn này cần thiết cho sự cài đặt các hàm trong lớp. Một điều cần lưu ý là, khi viết định nghĩa mỗi hàm thành phần, bạn cần sử dụng toán tử định phạm vi để chỉ nó thuộc lớp nào. Trong ví dụ đang xét, bạn cần đặt Complex :: ngay trứơc tên hàm thành phần. File cài đặt Complex.cpp được cho trong hình 2.3.

```
// File cài đặt Complex.cpp
# include "Complex.h"
# include <math.h>
# include <iostream.h>
  Complex :: Complex (double a = 0.0, double b = 0.0)
  {
        real = a ;
        imag = b ;
  }
```

```
double Complex :: GetReal ( ) const
{
      return  real ;
}
double Complex :: GetImag ( ) const
{
      return  imag ;
}
double Complex :: GetAbs ( ) const
{
      return sqrt (real * real + imag * imag) ;
}
Complex & operator + (const Complex & c1, const Complex & c2)
{
      Complex  c;
      c.real = c1.real + c2.real ;
      c.imag = c1.imag +c2.imag ;
      return  c ;
}
// Các hàm toán tử -, *, /
 ostream & operator << (ostream & os, const Complẽ &c)
 {
       os << c.real << "+i" << c.imag ;
       return  os ;
 }
```

Hình 2.3. File cài đặt Complex.cpp

Hướng dẫn xây dựng lớp cài đặt KDLTT.

Xây dựng một lớp tốt cài đặt một KDLTT là một nhiệm vụ khó khăn. Ứng với một KDLTT có thể có nhiều cách cài đặt khác nhau. Điều đó trước hết là do một loại đối tượng dữ liệu có thể được biểu diễn bởi nhiều CTDL khác nhau. Sự lựa chọn CTDL để cài đặt đối tượng dữ liệu cần phải sao cho các hàm thực hiện các phép toán trên dữ liệu là hiệu quả.

Sau khi đã lựa chọn CTDL, nhiệm vụ tiếp theo là thiết kế lớp: lớp cần chứa các hàm thành phần, hàm bạn nào? Các hàm đó cần được thiết kế như thế nào? (Tức là các hàm đó cần có mẫu hàm như thế nào?). Các hướng dẫn sau đây sẽ giúp bạn dễ dàng hơn khi phát triển một lớp cài đặt KDLTT. Các hướng dẫn này cũng nhằm mục đích để người lập trình có thể sử dụng KDLTT một cách thuận tiện, an toàn và hiệu quả.

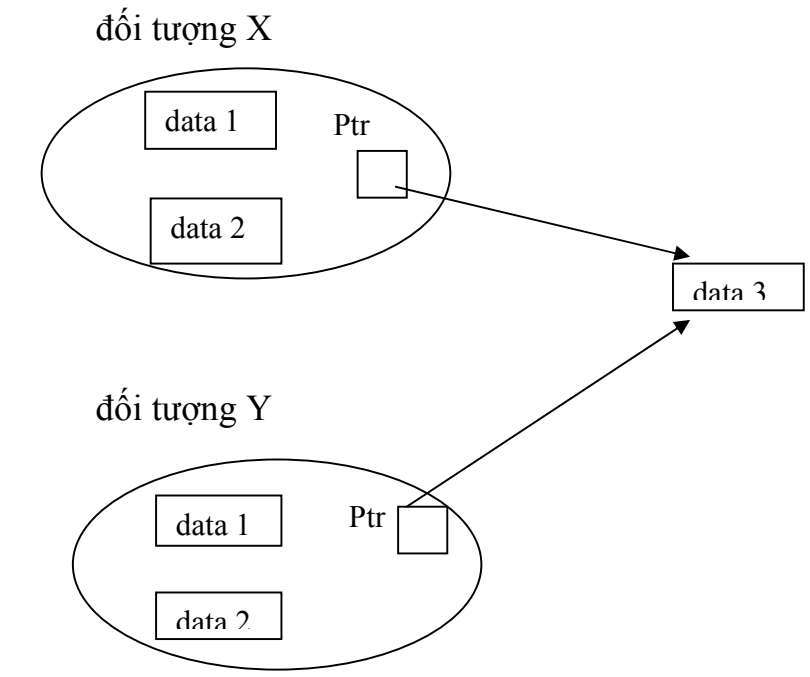
1.Cần nhớ rằng, không phải trong đặc tả KDLTT có bao nhiêu phép toán thì trong lớp chỉ có bấy nhiêu hàm tương ứng với các phép toán đó. Thông thường ngoài các hàm tương ứng với các phép toán, chúng ta cần đưa vào lớp nhiều hàm thành phần (hoặc hàm bạn) khác giúp cho người sử dụng tiến hành dễ dàng các thao tác trên dữ liệu trong chương trình, chẳng hạn các hàm kiến tạo, hàm huỷ, các loại toán tử gán, các hàm đọc dữ liệu, viết dữ liệu, hàm chuyển kiểu, …

2.Cần cung cấp một số hàm kiến tạo thích hợp để khởi tạo ra các đối tượng của lớp (rất ít khi người ta thiết kế một lớp không có hàm kiến tạo). Đặc biệt cần lưu ý đến hàm kiến tạo mặc định (hàm kiến tạo không có tham số) và hàm kiến tạo copy.

Nếu bạn không cung cấp cho lớp hàm kiến tạo mặc định, thì chương trình dịch sẽ tạo ra hàm kiến tạo mặc định tự động. Tuy nhiên hàm kiến tạo mặc định được cung cấp bởi chương trình dịch có hạn chế: nó chỉ khởi tạo các thành phần dữ liệu có hàm kiến tạo mặc định, còn các thành phần dữ liệu khác thì không được khởi tạo.

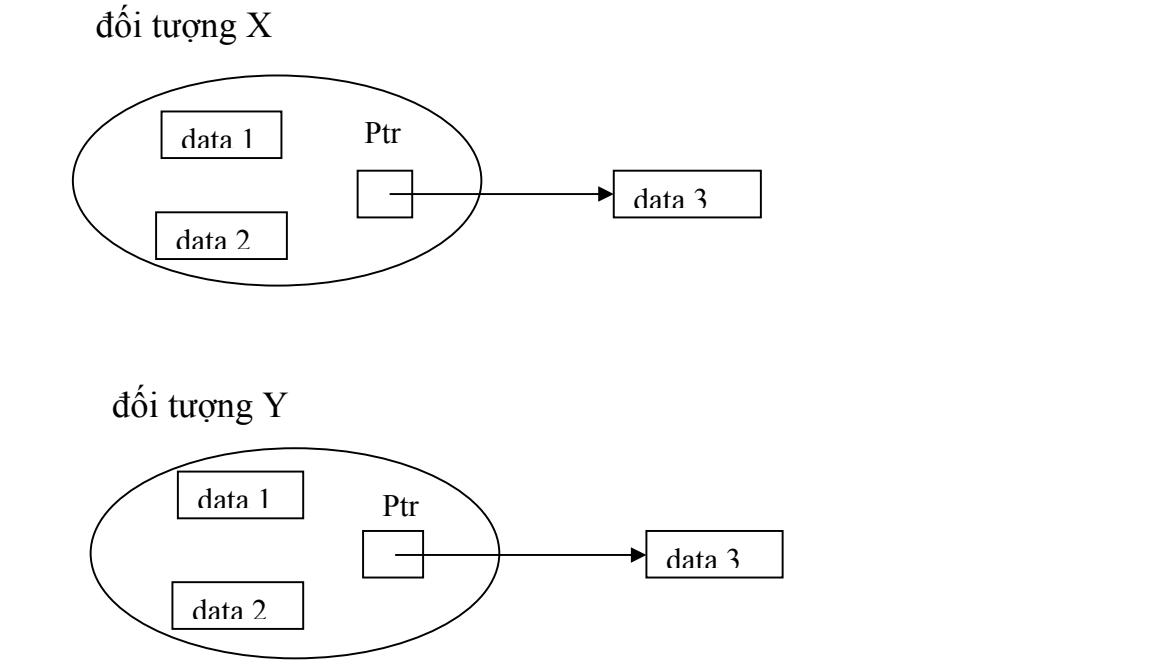
Cần biết rằng, hàm kiến tạo copy sẽ được tự động gọi bởi chương trình dịch khi mà đối tượng được truyền bởi giá trị trong một lời gọi hàm.

Nếu ta không cung cấp cho lớp một hàm kiến tạo copy, thì một hàm kiến tạo copy tự động sẽ được chương trình dịch cung cấp. Hàm này sẽ copy từng thành phần dữ liệu của đối tượng bị copy sang cho đối tượng copy. Chúng ta sẽ xét trường hợp lớp chứa thành phần dữ liệu là biến con trỏ Ptr. Giả sử X là đối tượng đã có của lớp này. Ta muốn khởi tạo ra đối tượng mới Y là bản sao của X bởi khai báo Y(X) (hoặc Y = X). Nếu trong lớp không cung cấp hàm kiến tạo copy, thì kết quả của lệnh trên sẽ được minh hoạ trong hình 2.4. Điều này sẽ kéo theo hậu quả là bất kỳ sự thay đổi dữ liệu data3 của đối tượng Y cũng kéo theo sự thay đổi dữ liệu data3 của đối tượng X và ngược lại. Đó là điều mà chúng ta không muốn có.



Hình 2.4. Đối tượng Y là copy của đối tượng X, khi lớp không được cung cấp hàm kiến tạo copy.

Trong trường hợp lớp có chứa biến thành phần là biến con trỏ, chúng ta cần đưa vào lớp hàm kiến tạo copy. Hàm này cần phải tạo ra đối tượng mới Y từ đối tượng X cũ như trong hình 2.5.



Hình 2.5. Đối tượng Y là copy của đối tượng X, khi trong lớp có hàm kiến tạo copy

3.Nếu bạn không đưa vào lớp hàm huỷ, thì chương trình dịch sẽ tạo ra hàm huỷ tự động. Song hàm huỷ này chỉ thu hồi vùng nhớ đã cấp cho tất cả các biến thành phần của lớp, vùng nhớ cấp phát động mà các biến thành phần kiểu con trỏ trỏ tới thì không bị thu hồi. Vì vậy, trong trường hợp lớp chứa các thành phần dữ liệu là biến con trỏ thì nhất thiết bạn phải thiết kế cho lớp một hàm huỷ, hàm này thực hiện nhiệm vụ thu hồi tất cả các vùng nhớ liên quan tới đối tượng để trả về cho hệ thống, khi mà đối tượng không còn cần thiết nữa.

4.Trong chương trình, toán tử gán được sử dụng thường xuyên trên các đối tượng dữ liệu. Do đó trong lớp, chúng ta cần đưa vào hàm toán tử operator =, nó định nghĩa lại toán tử gán truyền thống trong C + +, nó tương tự như hàm kiến tạo copy, chỉ có điều nó được sử dụng để gán, chứ không phải để khởi tạo ra đối tượng mới. Nếu bạn không cung cấp cho lớp toán tử gán thì toán tử gán tự động sẽ được chương trình dịch cung cấp. Nó chỉ làm được một việc: copy từng thành phần dữ liệu của đối tượng ở bên phải toán tử gán tới các thành phần dữ liệu tương ứng của đối tượng ở bên trái.

5.Các phép toán một toán hạng, hai toán hạng trên các đối tượng dữ liệu của KDLTT cần được thiết kế là các hàm toán tử của lớp (các hàm này định nghĩa lại các phép toán số học +, -, *, /, …tuỳ theo ngữ nghĩa của chúng). Các hàm toán tử có thể thiết kế như là hàm thành phần hoặc như là hàm bạn của lớp.

2.4 LỚP KHUÔN

Trong mục này chúng ta sẽ trình bày khái niệm **lớp khuôn (template class).** Lớp khuôn là một công cụ quan trọng trong C + + được sử dụng để cài đặt các lớp phụ thuộc tham biến kiểu dữ liệu. Các KDLTT quan trọng mà chúng ta nghiên cứu trong các chương sau đều được cài đặt bởi lớp khuôn. Trước hết chúng ta xét một ví dụ về lớp côngtơnơ và cách cài đặt không sử dụng lớp khuôn.

2.4.1 Lớp côngtơnơ

Trong nhiều ứng dụng, chúng ta cần sử dụng các KDLTT mà mỗi đối tượng dữ liệu của nó là một bộ sưu tập các phần tử dữ liệu cùng kiểu nào đó. Lớp cài đặt các KDLTT như thế được gọi là **lớp côngtơnơ (container class**). Như vậy, lớp côngtơnơ là một thuật ngữ để chỉ các lớp mà mỗi đối tượng của lớp là một "côngtơnơ" chứa một bộ sưu tập các dữ liệu cùng kiểu.

Các lớp danh sách, hàng đội, ngăn xếp, … được nghiên cứu sau này đều là lớp côngtơnơ. Trong một chương trình của mình, bạn có thể cần đến lớp côngtơnơ các số nguyên, người khác lại cần sử dụng chính lớp côngtơnơ đó, chỉ khác một điều là các côngtơnơ của anh ta không phải là côngtơnơ các số nguyên mà là côngtơnơ các số thực, hoặc côngtơnơ các ký tự. Do đó, vấn đề đặt ra cho việc thiết kế các lớp côngtơnơ là: lớp cần được thiết kế như lớp phụ thuộc tham biến kiểu dữ liệu Item sao cho trong một chương trình bất kỳ, bạn có thể dễ dàng sử dụng lớp bằng cách chỉ ra Item được thay bởi một kiểu dữ liệu cụ thể, chẳng hạn int, double hoặc char… Sau đây là một ví dụ về một lớp côngtơnơ và cách cài đặt sử dụng mệnh đề định nghĩa kiểu typedef.

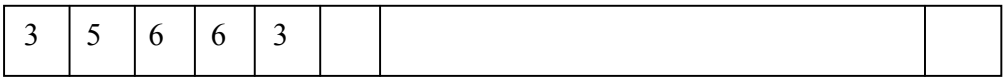
Ví dụ (KDLTT Túi). Trước hết chúng ta đặc tả KDLTT Túi. Mỗi đối tượng dữ liệu là một "túi" chứa một bộ sưu tập các phần tử cùng kiểu. Chẳng hạn, đối tượng dữ liệu có thể là túi bi, một túi có thể chứa 3 bi xanh, 1 bi đỏ, 2 bi vàng, … Một ví dụ khác, đối tượng dữ liệu có thể là côngtơnơ áo sơ mi, mỗi côngtơnơ có thể chứa nhiều áo thuộc cùng một loại áo (mỗi loại áo đặc trưng bởi kiểu dáng, loại vải, cỡ áo).

Sau đây là các phép toán có thể thực hiện trên các túi, Trong các phép toán này, B là ký hiệu một túi, element là một phần tử cùng kiểu với kiểu của các phần tử trong túi.

1. Sum (B). Hàm trả về số phần tử trong túi B.
2. Insert (B, element). Thêm phần tử element vào túi B.
3. Remove (B, element). Lấy ra một phần tử là bản sao của element khỏi túi B.
4. Occurr (B, element). Hàm trả về số phần tử là bản sao của element trong túi B.
5. Union (B_1, B_2). Hợp của túi B_1và túi B_2. Hàm trả về một túi chứa tất cả các phần tử của B_1 và tất cả các phần tử của túi B_2.

Dưới đây chúng ta đưa ra một cách cài đặt KDLTT Túi đã đặc tả ở trên.

Chúng ta sẽ sử dụng một mảng (tĩnh) data có cỡ là MAX để lưu các phần tử của túi. Các phần tử của túi được lưu trong các thành phần của mảng: data[0], data[1], …, data[size-1], trong đó size là số phần tử của túi. Chẳng hạn, một túi chứa 5 số nguyên: 2 số nguyên 3, 1 số nguyên 5, 2 số nguyên 6 có thể lưu trong mảng như sau:



Trong khai báo mảng data, cỡ của mảng MAX cần phải là hằng, MAX là số tối đa các phần tử mà túi có thể chứa, và nó là chung cho tất cả các đối tượng của lớp. Vì vậy, chúng ta sẽ khai báo MAX là hằng tĩnh trong lớp. Chẳng hạn, static const int MAX = 50 ;

Mục tiêu thiết kế lớp túi (class bag) của chúng ta là sao cho người lập trình sử dụng lớp túi trong các hoàn cảnh khác nhau một cách thuận tiện: chỉ cần một vài thay đổi nhỏ, không cần viết lại toàn bộ mã của lớp. Trong một chương trình, người lập trình có thể sử dụng lớp túi số nguyên, trong chương trình khác lại có thể sử dụng lớp túi bi hoặc lớp túi áo sơ mi, … Các phép toán trên các đối tượng của các loại túi khác nhau là hoàn toàn như nhau, không phụ thuộc gì vào kiểu dữ liệu của các phần tử trong túi. Do đó, chúng ta đưa vào lớp mệnh đề định nghĩa kiểu, mệnh đề này xác định kiểu dữ liệu Item, Item là kiểu dữ liệu của các phần tử trong túi . Chẳng hạn, trong định nghĩa lớp túi số nguyên, bạn đưa vào mệnh đề:

typedef int Item ;

Lớp túi chứa hai biến thành phần: mảng data lưu các phần tử của túi và biến size lưu số phần tử trong túi. Thiết kế ban đầu của lớp túi như sau:

```
class Bag
{
  public :
```

```
        static  const  int  MAX = 50 ;
         typedef  int  Item ;
         // Các hàm thành phần
    private :
        Item  data[MAX] ;
        int  size ;
} ;
```

Trong các hàm thành phần của lớp Bag, bất kỳ chỗ nào có mặt Item, chương trình dịch sẽ hiểu đó là int. Với cách thiết kế này, nếu chương trình của bạn cần đến lớp túi mà các đối tượng của nó chứa các ký tự, bạn chỉ cần thay int trong mệnh đề typedef bởi char. Còn nếu bạn muốn sử dụng các túi có cỡ lớn hơn, bạn chỉ cần xác định lại hằng MAX. Cần lưu ý rằng, hằng MAX và kiểu Item được xác định trong phạm vi lớp Bag. Vì vậy, ở ngoài phạm vi lớp Bag, để truy cập tới chúng, bạn cần sử dụng toán tử định phạm vi. Chẳng hạn, trong chương trình để in ra cỡ tối đa của túi, bạn cần viết

```
cout << "cỡ tối đa của túi là:" << Bag :: MAX << endl ;
```

Bây giờ chúng ta thiết kế các hàm thành phần của lớp Bag. Trước hết, cần có hàm kiến tạo mặc định sau

```
Bag ( ) ;
```

Hàm này khởi tạo ra một túi rỗng.

Các phép toán (1)-(4) của KDLTT Túi được cài đặt bởi các hàm thành phần tương ứng của lớp Bag như sau:

```
int  Sum (   )  const ;
void  Insert (const Item & element) ;
void  Remove (const Item & element) ;
int  Occurr (const Item & element) ;
```

Phép toán (5) được cài đặt bởi hàm toán tử bạn:

```
friend  Bag& operator + (const Bag & B1, const Bag & B2);
```

Với hàm toán tử này, chúng ta đã xác định phép + trên các túi. Do đó, để thuận tiện khi cần cộng thêm một túi vào một túi khác, chúng ta đưa vào lớp Bag một hàm toán tử thành phần operator += như sau:

void operator += (const Bag & B1) ;

Không cần đưa vào lớp Bag hàm kiến tạo copy, hàm toán tử gán, vì chỉ cần sử dụng hàm kiến tạo copy tự động, toán tử gán tự động là đủ.

Đến đây chúng ta có thể viết ra file đầu của lớp Bag. File đầu được cho trong hình 2.6.

```
// File đầu bag.h
// Sau đây là các thông tin người lập trình cần biết
// khi sử dụng lớp Bag.
// MAX là số tối đa các phần tử mà một Bag có thể chứa.
// Item là kiểu dữ liệu của các phần tử trong túi.
// Item có thể là một kiểu bất kỳ có sẵn trong C + + (chẳng hạn int,
// double, char, …), hoặc có thể là một lớp trong đó được cung cấp
// hàm kiến tạo mặc định, các hàm toán tử =, == và !=
// Sau đây là các hàm thành phần của lớp Bag:
// Bag( ) ;
// Postcondition: một Bag rỗng được khởi tạo.
// int  Sum (  )  const ;
// Postcondition: Trả về tổng số phần tử của túi.
// void  Insert(const Item& element) ;
// Hàm thực hiện thêm element vào túi B.
// Precondition: B.Sum ( ) < MAX
// Postcondition: một bản sao của element được thêm vào túi B.
// void Remove (const Item& element) ;
// Hàm loại một phần tử khỏi túi B.
// Postcondition: một phần tử là bản sao của element bị loại khỏi túi B,
// nếu trong B có chứa, và B không thay đổi nếu trong B không chứa.
// int  Occurr (const Item& element) ;
```

```
// Postcondition: Trả về số lần xuất hiện của element trong túi B.
// void operator += (const Bag& B1)
// Hàm thực hiện cộng túi B1 vào túi B, B += B1
// Postcondition: tất cả các phần tử của túi B1 được thêm vào túi B.
// Hàm toán tử bạn của lớp Bag:
// friend Bag& operator + (const Bag& B1, const Bag& B2) ;
// Precondition: B1.Sum ( ) + B2.Sum( ) <= MAX
// Postcondition: Trả về túi B là hợp của túi B1 và B2.
// Bạn có thể sử dụng toán tử gán = và hàm kiến tạo copy cho các
// đối tượng của lớp Bag.
# ifdef  BAG_H
# define  BAG_H
    class  Bag
    {
      public:
          static const  int  MAX = 50 ;
          typdef  int  Item ;
          Bag ( ) {size = 0 ;}
          int  Sum( )  const
          {return size ;}
          int  Occurr (const Item& element) const ;
          void  Insert (const Item& element) ;
          void  Remove (const Item& element) ;
          void  operator += (const  Bag & B1) ;
          friend Bag & operator + (const Bag & B1,const Bag & B2);
      private :
          Item  data[MAX];
          int  size ;
     } ;
# endif
```

Hình 2.6. File đầu của lớp Bag.

Bây giờ chúng ta tiến hành cài đặt các hàm đã khai báo trong định nghĩa lớp Bag.

- **Hàm Insert**: Trước hết cần kiểm tra túi không đầy. Các phần tử của túi được lưu trong các thành phần của mảng data: data[0], data[1], …, data[size - 1], do đó nếu túi không đầy, phần tử element được lưu vào thành phần data[size], tức là

```
if (size < MAX)
   {
      data[size] = element ;
      size + + ;
   }
```

- **Hàm Remove**: Để loại được phần tử element khỏi túi, trước hết ta tìm vị trí đầu tiên trong mảng data, tại đó có lưu element. Điều này được thực hiện bằng cách cho biến i chạy từ 0 tới size – 1, nếu gặp một thành phần của mảng mà data[i] == element thì dừng lại, còn nếu i chạy tới size thì điều đó có nghĩa là túi không chứa element. Bước tìm vị trí đầu tiên lưu element được thực hiện bởi vòng lặp:

```
For ( i = 0; ( i < size ) && (data[i] != element ); i + + ) ;
```

  Việc loại element ở vị trí thứ i trong mảng data được thực hiện bằng cách chuyển phần tử lưu ở vị trí cuối cùng data[size - 1] tới vị trí i, và giảm số phần tử của túi i đi 1. Tức là:

```
data[i] = data[size - 1] ;
size -- ;
```

- **Hàm Occurr**: Chúng ta cần đếm số lần xuất hiện của phần tử element trong túi. Muốn vậy, chúng ta sử dụng một biến đếm count, cho biến chỉ số i chạy từ đầu mảng tới vị trí size – 1, cứ mỗi lần gặp phần tử element thì biến đếm count được tăng lên 1.

```
count = 0 ;
for (i = 0; i < size; i + +)
```

```
if ( data[i] = = element)
    count + +;
```

- **Hàm toán tử operator +=** : Nhiệm vụ của hàm là "đổ" nội dung của túi B1 vào túi B. Chỉ có thể thực hiện được điều đó khi mà tổng số phần tử của hai túi không vượt quá dung lượng MAX. Nếu điều kiện đó đúng, chúng ta sao chép lần lượt các phần tử của túi B1 vào các thành phần mảng data của túi B, bắt đầu từ vị trí size. Hành động đó được thực hiện bởi vòng lặp:

```
for (i =0; i < B1.size; i + +)
{
   data[size] = B1.data[i] ;
   size + + ;
 }
```

- **Hàm toán tử bạn operator +**. Muốn gộp hai túi thành một túi, đương nhiên cũng cần điều kiện tổng số phần tử của hai túi không lớn hơn MAX. Chúng ta khởi tạo một túi B rỗng, rồi áp dụng toán tử += đổ lần lượt túi B1, B2 vào túi B, tức là:

```
Bag  B;
B  +=  B1;
B  +=  B2;
```

Sử dụng hàm tiện ích assert.

Trong nhiều hàm, để hàm thực hiện được nhiệm vụ của mình, các dữ liệu vào cần thoả mãn các điều kiện được liệt kê trong phần chú thích Precondition ở ngay sau mẫu hàm. Khi cài đặt các hàm, chúng ta có thể sử dụng mệnh đề if (Precondition) ở đầu thân hàm, như chúng ta đã làm khi nói về hàm Insert. Một cách lựa chọn khác là sử dụng hàm assert trong thư viện chuẩn assert.h. Hàm assert có một đối số, đó là một biểu thức nguyên, hoặc thông thường là một biểu thức logic. Hàm assert sẽ đánh giá biểu thức, nếu biểu thức có giá trị true, các lệnh tiếp theo trong thân hàm sẽ được thực hiện. Nếu biểu thức có giá trị false, assert sẽ in ra một thông báo lỗi và chương

trình sẽ bị treo. Vì vậy, khi cài đặt một hàm, chúng ta nên sử dụng hàm assert thay cho mệnh đề if.

File cài đặt của lớp Bag được cho trong hình 3.7. Chú ý rằng, trong các hàm có sử dụng hàm assert, nên ở đầu file chúng ta cần đưa vào mệnh đề # include < assert.h >

```
// File : bag.cpp
# include  <assert.h>
# include  "bag.h"
int  Bag :: Occurr (const Item & element) const
{
        int  count = 0 ;
        int i ;
        for (i=0; i < size ; i + +)
           if (data[i] == element)
                count + + ;
        return  count ;
}
void  Bag :: Insert (const Item & element)
{
        assert (Sum( ) < MAX) ;
        data[size] = element ;
        size + + ;
}
void  Bag :: Remove (const Item & element)
{
        int  i;
        for (i = 0; (i < size) && (data[i]! =element); i + +) ;
        if (i == size)
          return ;
        data[i] = data[size - 1] ;
```

```
        size -- ;
}
void  Bag :: operator += (const Bag & B1)
{
        int  i;
        int  a = B1.size ;
        assert (Sum( ) + B1.Sum( ) <= MAX) ;
        for (i = 0; i < a; i + +)
        {
           data[size] = B1.data[i] ;
           size + + ;
        }
}
Bag & operator + (const Bag & B1, const Bag & B2)
{
        Bag  B;
        assert (B1.Sum( ) + B2.Sum( ) <= Bag :: MAX) ;
        B += B1 ;
        B += B2 ;
        return  B ;
}
```

Hình 2.7. File cài đặt của lớp Bag.

Nhận xét. Lớp Bag mà chúng ta đã cài đặt có các đặc điểm sau: Trước hết, mỗi đối tượng của lớp là túi chỉ chứa được tối đa 50 phần tử, do khai báo hằng MAX ở trong lớp:

```
        static  const  int  MAX = 50;
```

Mặt khác, mệnh đề

```
        typedef  int  Item;
```

xác định rằng, các phần tử trong túi là các số nguyên. Trong một chương trình, nếu bạn muốn sử dụng Bag với cỡ túi lớn hơn, chẳng hạn 100, và các phần tử trong túi có kiểu khác, chẳng hạn char, bạn chỉ cần thay 50 bằng 100 trong mệnh đề khai báo hằng MAX và thay int bằng char trong mệnh đề định nghĩa Item. Song nếu như trong một chương trình, bạn cần đến cả túi số nguyên, cả túi ký tự thì sao? Bạn không thể định nghĩa hai lần lớp Bag trong một chương trình! **Lớp khuôn (template class**) sẽ giải quyết được khó khăn này. Lớp khuôn trong C + + là một công cụ cho phép ta thiết kế các lớp phụ thuộc tham biến kiểu dữ liệu. Trước khi trình bày khái niệm lớp khuôn, chúng ta hãy nói về các **hàm khuôn (template function**)

2.4.2 Hàm khuôn

Xét vấn đề tìm giá trị lớn nhất (nhỏ nhất) trong một mảng. Thuật toán đơn giản là đọc tuần tự các thành phần của mảng và lưu lại vết của phần tử lớn nhất (nhỏ nhất). Rõ ràng, logic của thuật toán này chẳng phụ thuộc gì vào kiểu dữ liệu của các phần tử trong mảng. Các thuật toán sắp xếp mảng cũng có đặc trưng đó. Một ví dụ khác: vấn đề trao đổi giá trị của hai biến. Thuật toán trao đổi giá trị của hai biến mà chúng ta đã quen biết cũng không phụ thuộc vào kiểu dữ liệu của hai biến đó. Chúng ta mong muốn viết ra các hàm cài đặt các thuật toán đó sao cho hàm có thể sử dụng được cho các loại kiểu dữ liệu khác nhau.

Một cách lựa chọn là sử dụng mệnh đề định nghĩa kiểu typedef. Chẳng hạn, hàm trao đổi giá trị của hai biến có thể viết như sau:

```
typedef  int  Item ;
void  Swap (Item & x, Item & y)
{  Item  temp;
   temp = x ;
   x = y ;
```

```
  y = temp ;
}
```

Hàm trên để trao đổi hai biến nguyên. Nếu cần trao đổi hai biến ký tự, bạn thay int bởi char trong mệnh đề typedef. Nhưng cách tiếp cận này có vấn đề, khi trong cùng một chương trình bạn vừa cần trao đổi hai biến nguyên, vừa cần trao đổi hai biến ký tự, bởi vì bạn không thể hai lần định nghĩa kiểu Item trong cùng một chương trình.

Một cách tiếp cận khác khắc phục được hạn chế của cách tiếp cận trên là sử dụng hàm khuôn. Chẳng hạn, hàm khuôn Swap được viết như sau:

```
template  <class Item>
void  Swap (Item & x, Item & y)
// Item cần phải là kiểu bất kỳ có sẵn trong C + + (int, char, …)
// hoặc là lớp có hàm kiến tạo mặc định và toán tử gán.
{
  Item  temp ;
  temp = x ;
  x = y ;
  y = temp;
}
```

Để xác định một hàm khuôn, bạn chỉ cần đặt biểu thức **template <class Item>** ngay trước mẫu hàm. Biểu thức đó nói rằng, ở mọi chỗ Item xuất hiện trong định nghĩa hàm, Item là một kiểu dữ liệu nào đó chưa được xác định. Nó sẽ được hoàn toàn xác định khi ta sử dụng hàm (gọi hàm). Chương trình đơn giản sau minh họa cách sử dụng hàm khuôn Swap.

```
int main(  )
 {
    int  a = 3;
```

```
    int  b = 5;
    double  x = 4.2;
    double  y = 3.6;
    Swap(a,b); // Item được thay thế bởi int
    Swap(x,y); // Item được thay thế bởi double
    return   0 ;
}
```

Một ví dụ khác về hàm khuôn: hàm tìm giá trị lớn nhất trong mảng

```
template  <class  Item>
Item & FindMax (Item data[  ], int n )
// Item cần phải là kiểu bất kỳ có sẵn trong C + + (int, char, …)
// hoặc là lớp có phép toán so sánh < ,
// Precondition: data là mảng có ít nhất n thành phần, n > 0.
// Postcondition: giá trị trả về là phần tử lớn nhất trong n phần tử
// data[0], …,data[n -1].
{
  int  i ;
  int index = 0; // chỉ số của phần tử lớn nhất.
  assert( n>0)
  for (i = 1; i < n; i + +)
    if (data [index] < data [i])
        index = i;
  return data [index];
}
```

2.4.3 Lớp khuôn

Hàm khuôn là hàm phụ thuộc tham biến kiểu dữ liệu. Tương tự, nhờ lớp khuôn chúng ta xây dựng được lớp phụ thuộc tham biến kiểu dữ liệu.

Định nghĩa lớp khuôn

Xác định một lớp khuôn cũng tương tự như xác định một hàm khuôn. Cần đặt biểu thức template <class Item> ngay trước định nghĩa lớp, Item là tham biến kiểu của lớp. Trong định nghĩa lớp, chỗ nào có mặt Item, chúng ta chỉ cần hiểu Item là một kiểu dữ liệu nào đó. Item là kiểu cụ thể gì (là int hay double, …) sẽ được xác định khi chúng ta sử dụng lớp trong chương trình.

Ví dụ (Lớp khuôn Bag). Từ lớp Bag được xây dựng bằng cách sử dụng mệnh đề định nghĩa kiểu typedef int Item (xem hình 2.6), chúng ta có thể xây dựng lớp khuôn Bag như sau:

```
template <class Item>
class   Bag
{
  // Bỏ mệnh đề typedef int Item;
} ;
```

Trong định nghĩa lớp khuôn Bag, tất cả các hàm thành phần của nó là các hàm thành phần của lớp Bag cũ (lớp trong hình 2.6), không có gì thay đổi. Riêng hàm operator +, nó không phải là hàm thành phần của lớp, nên trong mẫu hàm của nó, Bag (với tư cách là tên kiểu dữ liệu) cần phải được viết là Bag <Item>. Cụ thể là:

```
friend Bag<Item> & operator + (const Bag <Item> & B1,
                               const Bag <Item> & B2);
```

Tổng quát, ở bên ngoài định nghĩa lớp khuôn Bag (chẳng hạn, ở trong file cài đặt lớp khuôn Bag), bất kỳ chỗ nào Bag xuất hiện với tư cách là tên lớp (trong toán tử định phạm vi Bag ::). hoặc với tư cách là tên kiểu dữ liệu, chúng ta cần phải viết là **Bag <Item>** để chỉ rằng, Bag là lớp phụ thuộc tham biến kiểu Item.

Cài đặt các hàm. Tất cả các hàm thao tác trên các đối tượng của lớp khuôn (bao gồm các hàm thành phần, các hàm bạn, các hàm không thành

phần) đều phải được cài đặt là hàm khuôn. Chẳng hạn, hàm toán tử += của lớp Bag cần được cài đặt như sau:

```
template  <class Item>
void  Bag <Item> :: operator += (const Bag <Item> & B1)
{
    int  i;
    int  a = B1. size;
    assert (Sum( ) + B1. Sum( ) < = MAX);
    for (i = 0; i < a; i + +)
    {
      data [size] = B1.data [i];
      size + + ;
    }
}
```

Tổ chức các file. Như khi cài đặt lớp bình thường, cài đặt lớp khuôn cũng được tổ chức thành hai file: file đầu và file cài đặt. Chỉ có một điều cần lưu ý là, hầu hết các chương trình dịch đòi hỏi phải đưa tên file cài đặt vào trong file đầu bởi mệnh đề # include "tên file cài đặt". Hình 2.8 là file đầu của lớp khuôn Bag, file cài đặt ở trong hình 2.9.

```
// File đầu: bagt.h
// Các thông tin cần thiết để sử dụng lớp Bag, các chú thích về các
//  hàm hoàn toàn giống như trong file đầu bag.h (xem hình 2.6).
# ifndef   BAG_H
# define   BAG_H
   template  <class Item>
   class   Bag
   {
     public :
```

```
        static  const  int  MAX = 50;
        Bag( )  {size = 0; ]
        int  Sum( )  const
        {  return  size; }
        int   Occurr (const Item & element) const;
        void   Insert (const Item & element);
        void   Remove (const Item & element);
        void   operator += (const Bag & B1);
        friend   Bag< Item> & operator + (const Bag<Item> & B1,
                                          const Bag<Item> & B2);
    private :
        Item data[MAX];
        int  size;
    };
# include "bag.template"
# endif
```

Hình 2.8. File đầu của lớp khuôn Bag.

```
// File cài đặt : bag.template
# include  <assert.h>
template  <class  Item>
int  Bag<Item> :: Occurr (const Item & element) const
{
  int count = 0;
  int  i ;
  for (i = 0; i < size; i + +)
      if (data[i] == element)
          count + +;
```

```
  return count ;
}

template  <class Item>
void Bag<Item> :: Insert (const Item & element)
{
  assert ( Sum( ) < MAX) ;
  data[size] = element;
  size + +;
}

template  <class Item>
void Bag<Item> :: Remove (const Item & element)
{
  int i;
  for (i = 0; (i < size) && (data[i]!= element); i + +);
    if ( i == size)
        return;
  data[i] = data[size - 1];
  size - - ;
}

template  <class Item>
void  Bag<Item> :: operator += (const Bag<Item> & B1)
{
  int  i;
  int a = B1. size;
  assert ( Sum( ) + B1. Sum( ) <= MAX);
  for (i = 0; i < a; i + +)
  {
     data[size] = B1.data[i];
```

```
      size + +;
   }
}

template  <class Item>
Bag<Item> & operator + (const Bag<Item> & B1,
                        const Bag<Item> & B2)
{
   Bag<Item>  B;
   assert (B1.Sum( ) + B2.Sum( ) <= Bag<Item> :: MAX);
   B += B1;
   B += B2;
   return  B;
}
```

Hình 2.9. File cài đặt lớp khuôn Bag.

Sử dụng lớp khuôn. Trong một chương trình, để sử dụng một lớp khuôn, trước hết bạn cần đưa vào mệnh đề # include "tên file đầu", chẳng hạn # include "bag.h". Khi khai báo các đối tượng của lớp, bạn cần thay thế tham biến kiểu bởi kiểu thực tế. Chẳng hạn, biểu thức Bag<Item> nói rằng, Bag là lớp khuôn với tham biến kiểu Item, nên trong chương trình nếu bạn muốn sử dụng đối tượng A là túi số nguyên, bạn cần khai báo:

```
Bag<int>   A;
```

Chương trình demo sử dụng lớp khuôn Bag được cho trong hình 2.10.

```
# include <iostream.h>
# include "bag.h"
int  main( )
{
    Bag<int> A; // A là túi số nguyên.
```

```
    Bag<char>  B; // B là túi ký tự.
    char c; // ký tự bạn đưa vào.
    cout << "Please type 10 chars \ n"
         << "Press the return key after typing \ n" ;
    for (int i = 1; i <= 10; i + +)
    {
      cin >> c;
      B. Insert(c);
      A. Insert(int(c));
    }
    cout << "The total of integers 65 in the bag A:"
         << A. Occurr(65);
    return 0;
}
```

Hình 2.10. Sử dụng lớp khuôn.

2.5 CÁC KIỂU DỮ LIỆU TRỪU TƯỢNG QUAN TRỌNG

Trong mục này chúng ta sẽ giới thiệu các KDLTT quan trọng nhất. Các KDLTT này được sử dụng thường xuyên trong thiết kế các thuật toán. Các KDLTT này sẽ được lần lượt nghiên cứu trong các chương tiếp theo.

Trong chương 1 chúng ta đã thảo luận về tầm quan trọng của sự trừu tượng hoá dữ liệu trong thiết kế thuật toán. Sự trừu tượng hoá dữ liệu được thực hiện bằng cách xác định các KDLTT. Một KDLTT là một tập các đối tượng dữ liệu cùng với một tập phép toán có thể thực hiện trên các đối tượng dữ liệu đó. Các đối tượng dữ liệu trong thế giới hiện thực rất đa dạng và có thể rất phức tạp. Để mô tả chính xác các đối tượng dữ liệu, chúng ta cần sử dụng các khái niệm toán học, các mô hình toán học.

Tập hợp là khái niệm cơ bản trong toán học, nó là cơ sở để xây dựng nên nhiều khái niệm toán học khác. Tuy nhiên, tập hợp trong toán học là cố định. Còn trong các chương trình, thông thường chúng ta cần phải lưu một tập các phần tử dữ liệu, tập này sẽ thay đổi theo thời gian trong quá trình xử lý, do chúng ta thêm các phần tử dữ liệu mới vào và loại các phần tử dữ liệu nào đó khỏi tập. Chúng ta sẽ gọi các tập như thế là **tập động (dynamic set**). Giả sử rằng, các phần tử của tập động chứa một trường được gọi là khoá (key) và trên các giá trị khoá có quan hệ thứ tự (chẳng hạn khoá là số nguyên, số thực, ký tự,…). Chúng ta cũng giả thiết rằng, các phần tử khác nhau của tập động có khoá khác nhau.

KDLTT tập động (**dynamic set ADT**) được xác định như sau: Mỗi đối tượng dữ liệu của kiểu này là một tập động. Dưới đây là các phép toán của KDLTT tập động, trong các phép toán đó, S ký hiệu một tập, k là một giá trị khoá và x là một phần tử dữ liệu.

1. Insert(S, x). Thêm phần tử x vào tập S.
2. Delete(S, k). Loại khỏi tập S phần tử có khoá k.
3. Search(S, k). Tìm phần tử có khoá k trong tập S.
4. Max(S). Trả về phần tử có khoá lớn nhất trong tập S.
5. Min(S). Trả về phần tử có khoá nhỏ nhất trong tập S.

Ngoài các phép toán chính trên, còn có một số phép toán khác trên tập động.

Trong nhiều áp dụng, chúng ta chỉ cần đến ba phép toán Insert, Delete và Search. Các tập động với ba phép toán này tạo thành **KDLTT từ điển (dictionary ADT).**

Một khái niệm quan trọng khác trong giải tích toán học là khái niệm dãy (a_1, …, a_i, …). Dãy khác tập hợp ở chỗ, các phần tử của dãy được sắp xếp theo một thứ tự xác định: a_1 là phần tử đầu tiên, a_i là phần tử ở vị trí thứ i (i = 1, 2, …), một phần tử có thể xuất hiện ở nhiều vị trí khác nhau trong dãy. Chúng ta sẽ gọi một dãy hữu hạn là một **danh sách.** Sử dụng danh sách với tư cách là đối tượng dữ liệu, chúng ta cũng cần đến các thao tác Insert, Delete. Tuy nhiên, các phép toán Insert và Delete trên danh sách có khác với

các phép toán này trên tập động. Nếu ký hiệu L là danh sách thì phép toán Insert (L, x, i) có nghĩa là xen phần tử x và vị trí thứ i trong danh sách L, còn Delete (L, i) là loại phần tử ở vị trí thứ i khỏi danh sách L. Trên danh sách còn có thể tiến hành nhiều phép toán khác. Các danh sách cùng với các phép toán trên danh sách tạo thành **KDLTT danh sách (list ADT**). KDLTT danh sách sẽ được nghiên cứu trong chương 4 và 5.

Trong nhiều trường hợp, khi thiết kế thuật toán, chúng ta chỉ cần sử dụng hạn chế hai phép toán Insert và Delete trên danh sách. Các danh sách với hai phép toán Insert và Delete chỉ được phép thực hiện ở một đầu danh sách lập ra một KDLTT mới: **KDLTT ngăn xếp (stack ADT**). Nếu chúng ta xét các danh sách với phép toán Insert chỉ được phép thực hiện ở một đầu danh sách, còn phép toán Delete chỉ được phép thực hiện ở một đầu khác của danh sách, chúng ta có **KDLTT hàng đợi (queue ADT**). Ngăn xếp được nghiên cứu trong chương 6, hàng đợi trong chương 7.

Cây là một tập hợp với cấu trúc phân cấp. Một dạng cây đặc biệt là cây nhị phân, trong cây nhị phân mỗi đỉnh chỉ có nhiều nhất hai đỉnh con được phân biệt là đỉnh con trái và đỉnh con phải. **KDLTT cây nhị phân** (**binary tree ADT**) sẽ được nghiên cứu trong chương 9.

KDLTT hàng ưu tiên (priority queue ADT) là một biến thể của KDLTT từ điển. Hàng ưu tiên là một tập động với ba phép toán Insert, FindMin, DeleteMin. Hàng ưu tiên sẽ được trình bày trong chương 11.

Các KDLTT trên đây đóng vai trò cực kỳ quan trọng trong thiết kế chương trình, đặc biệt KDLTT từ điển, bởi vì trong phần lớn các chương trình chúng ta cần lưu một tập dữ liệu rồi thì tiến hành tìm kiếm dữ liệu và cập nhật tập dữ liệu đó (bởi xen, loại dữ liệu).

Như chúng ta đã nhấn mạnh trong chương 2, một KDLTT có thể có nhiều cách cài đặt. Chúng ta sẽ cài đặt các KDLTT bởi các lớp C + +, và sẽ phân tích, đánh gía hiệu quả của các phép toán trong mỗi cách cài đặt. Chúng ta có thể cài đặt danh sách bởi mảng tĩnh, bởi mảng động hoặc bởi CTDL danh sách liên kết (chương 4).

Có nhiều cách cài đặt tập động. Có thể cài đặt tập động bởi danh sách, danh sách được sắp (chương 4). Cài đặt tập động bởi **cây tìm kiếm nhị phân** (**binary search tree**) sẽ được trình bày trong chương 9. Đó là một trong các cách cài đặt tập động đơn giản và hiệu quả.

Bảng băm (hashing table) là một CTDL đặc biệt thích hợp cho cài đặt từ điển. Cài đặt từ điển bởi bảng băm là nội dung của chương 10.

BÀI TẬP

1. Hãy nói rõ sự khác nhau giữa hai mục public và private của một lớp.
2. Hãy mô tả vai trò của hàm kiến tạo và hàm huỷ? Có các loại hàm kiến tạo nào?
3. Sự khác nhau giữa hàm kiến tạo copy và toán tử gán?
4. Nếu ta không cung cấp cho lớp hàm kiến tạo và hàm huỷ thì kết quả sẽ như thế nào?
5. Khi nào trong một lớp nhất thiết phải đưa vào hàm huỷ, hàm kiến tạo copy, toán tử gán?
6. Sự khác nhau giữa hàm thành phần và hàm bạn của một lớp?
7. Hãy cài đặt lớp xâu ký tự theo các chỉ dẫn sau. Xâu ký tự được biểu diễn bởi mảng động. Lớp cần chứa các hàm thực hiện các công việc sau: xác định độ dài của xâu, truy cập tới ký tự thứ i trong xâu, kết nối hai xâu thành một xâu, các phép toán so sánh hai xâu, đọc và viết ra xâu. Tất cả các hàm cần phải cài đặt là hàm toán tử, chỉ trừ hàm xác định độ dài của xâu.

CHƯƠNG 3

SỰ THỪA KẾ

Một trong các đặc trưng quan trọng nhất của C + + và các ngôn ngữ lập trình định hướng đối tượng khác là cho phép chúng ta có thể sử dụng lại các thành phần mềm. Trong mục 2.4 chúng ta đã trình bày một phương pháp thực hiện sử dụng lại các thành phần mềm bằng cách xây dựng các lớp khuôn. Chương này sẽ trình bày một phương pháp khác: sử dụng lại các thành phần mềm thông qua tính **thừa kế (inheritance**). Sử dụng tính thừa kế, chúng ta có thể xây dựng nên các lớp mới từ các lớp đã có, tránh phải viết lại các thành phần mềm đã có.

3.1 CÁC LỚP DẪN XUẤT

Khi xây dựng một lớp mới, trong nhiều trường hợp lớp mới cần xây dựng có nhiều điểm giống một lớp đã có. Khi đó trên cơ sở lớp đã có, bằng cách sử dụng tính thừa kế, chúng ta có thể xây dựng nên lớp mới. Lớp đã có được gọi là **lớp cơ sở (base class**), lớp mới được xây dựng nên từ lớp cơ sở được gọi là **lớp dẫn xuất (derived class**). Một lớp dẫn xuất có thể được thừa kế từ nhiều lớp cơ sở, điều này được gọi là **tính đa thừa kế** (**multiple inheritance**). Song để đơn giản cho trình bày, sau đây chúng ta chỉ đề cập tới sự thiết kế lớp dẫn xuất thừa kế từ một lớp cơ sở.

Tính thừa kế cho phép ta sử dụng lại các thành phần mềm khi chúng ta xây dựng một lớp mới. Lớp dẫn xuất có thể thừa kế các thành phần dữ liệu và các hàm thành phần từ lớp cơ sở, trừ các hàm kiến tạo và hàm huỷ. Lớp dẫn xuất có thể thêm vào các thành phần dữ liệu mới và các hàm thành phần mới cần thiết cho các phép toán của nó. Ngoài ra, lớp dẫn xuất còn có

thể xác định lại bất kỳ hàm thành phần nào của lớp cơ sở cho phù hợp với các đặc điểm của lớp dẫn xuất.

Cú pháp xác định một lớp dẫn xuất như sau: Đầu lớp bắt đầu bởi từ khoá class, sau đó là tên lớp dẫn xuất, rồi đến dấu hai chấm, theo sau là từ khoá chỉ định dạng thừa kế (public, private, protected), và cuối cùng là tên lớp cơ sở. Chẳng hạn, nếu ta muốn xác định một lớp dẫn xuất D từ lớp cơ sở B thì có thể sử dụng một trong ba khai báo sau:

```
class D :  public  B { … } ;
class D :  private  B { … } ;
class D :  protected  B { … } ;
```

Chúng ta sẽ nói tới đặc điểm của các dạng thừa kế ở cuối mục này, còn bây giờ chúng ta sẽ xét một ví dụ minh hoạ. Giả sử chúng ta muốn xây dựng lớp Ball (lớp quả bóng) từ lớp Sphere (lớp hình cầu). Giả sử lớp hình cầu được xác định như sau:

```
class    Sphere
{
 public :
    Sphere (double R = 1) ;
    double   Radius ( ) const ;
    double   Area ( ) const ;
    double   Volume ( ) const ;
    void   WhatIsIt ( ) const ;
 private :
    double  radius ;
}
```

Lớp Sphere chỉ có một thành phần dữ liệu radius là bán kính của hình cầu, và các hàm thành phần: hàm kiến tạo ra hình cầu có bán kính R, hàm

cho biết bán kính hình cầu Radius ( ), hàm tính diện tích hình cầu Area ( ) và hàm tính thể tích hình cầu Volume ( ), cuối cùng là hàm WhatIsIt ( ) cho ta câu trả lời rằng đối tượng được hỏi là hình cầu có bán kính là bao nhiêu. Hàm WhatIsIt ( ) được cài đặt như sau:

```
void   Sphere :: WhatIsIt ( )   const
{
  cout << "It is the sphere with the radius"
       << radius ;
}
```

Bởi vì mỗi quả bóng là một hình cầu, nên chúng ta có thể xây dựng lớp Ball như là lớp dẫn xuất từ lớp Sphere. Lớp Ball thừa kế tất cả các thành phần của lớp Sphere, trừ ra hàm kiến tạo và xác định lại hàm WhatIsIt ( ). Chúng ta sẽ thêm vào lớp Ball một thành phần dữ liệu mới madeof để chỉ quả bóng được làm bằng chất liệu gì: cao su, nhựa hay gỗ. Một hàm thành phần mới cũng được thêm vào lớp Ball, đó là hàm Madeof ( ) trả về chất liệu tạo thành quả bóng. Lớp Ball được khai báo như sau:

```
class  Ball : public  Sphere
{
 public :
    enum Materials {RUBBER, PLASTIC, WOOD};
    Ball (double R = 1, Materials M = RUBBER) ;
    Materials   MadeOf (  )  const ;
    void   WhatIsIt ( )   const ;
 private :
    Materials   madeOf ;
} ;
```

Lớp Ball được định nghĩa như trên sẽ có hai thành phần dữ liệu: radius được thừa kế từ lớp Sphere và madeof mới được đưa vào. Ngoài hàm kiến tạo, lớp Ball có ba hàm thành phần được thừa kế từ lớp Sphere, đó là các hàm Radius ( ), Area ( ) và Volume( ), một hàm thành phần mới là hàm MadeOf( ), và hàm thành phầnWhatIsIt( ) mới, nó định nghĩa lại một hàm cùng tên đã có trong lớp cơ sở Sphere. Hàm WhatIsIt( ) trong lớp Ball được định nghĩa như sau:

```
void   Ball :: WhatIsIt ( )   const
{
  Sphere :: WhatIsIt( ) ;
  cout << " and made of " << madeof ;
}
```

Hàm kiến tạo của lớp dẫn xuất. Nếu chúng ta không cung cấp cho lớp dẫn xuất hàm kiến tạo, thì chương trình dịch sẽ tự động cung cấp hàm kiến tạo mặc định tự động. Nhưng cũng như đối với một lớp bất kỳ, khi thiết kế một lớp dẫn xuất, nói chung chúng ta cần phải cung cấp cho lớp dẫn xuất hàm kiến tạo. Bây giờ chúng ta xét xem hàm kiến tạo của lớp dẫn xuất được cài đặt như thế nào? Chú ý rằng, lớp dẫn xuất chứa các thành phần dữ liệu được thừa kế từ lớp cơ sở, ngoài ra nó còn chứa các thành phần dữ liệu mới, trong các thành phần dữ liệu mới này có thể có thành phần dữ liệu là đối tượng của một lớp khác. Nhưng trong lớp dẫn xuất, chúng ta không được quyền truy cập trực tiếp đến các thành phần dữ liệu của lớp cơ sở (và các thành phần dữ liệu của lớp khác). Vậy làm thế nào để khởi tạo các thành phần dữ liệu của lớp cơ sở (và các thành phần dữ liệu của lớp khác). Vấn đề này được giải quyết bằng cách cung cấp một **danh sách khởi tạo (initiazation list)**. Danh sách khởi tạo là danh sách các lời gọi hàm kiến tạo của lớp cơ sở (và các hàm kiến tạo của các lớp khác). Danh sách này được đặt ngay sau đầu hàm kiến tạo của lớp dẫn xuất. Ví dụ, hàm kiến tạo của lớp dẫn xuất Ball được cài đặt như sau:

```
Ball :: Ball (double R, Materials M)
: Sphere (R)
{ madeof = M; }
```

Lưu ý rằng, ngay trước danh sách khởi tạo phải có dấu hai chấm :, trong ví dụ trên danh sách khởi tạo chỉ có một lời gọi hàm kiến tạo lớp cơ sở Sphere (R), nếu có nhiều lời gọi hàm thì cần có dấu phẩy giữa các lời gọi hàm.

Các mục public, private và protected của một lớp

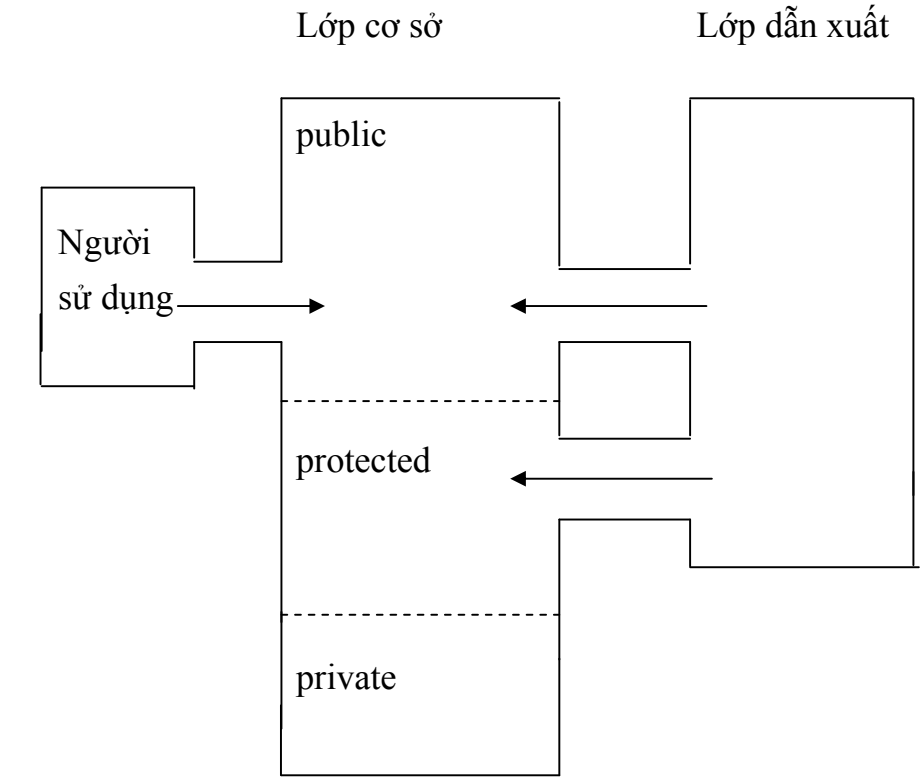
Trong các ví dụ mà chúng ta đưa ra từ trước tới nay, các thành phần của lớp được đưa vào hai mục: public và private. Các thành phần nằm trong mục public là các thành phần công khai, khách hàng của lớp có thể sử dụng trực tiếp các thành phần này. Các thành phần nằm trong mục private là các thành phần cá nhân của lớp, chỉ được phép sử dụng trong phạm vi lớp. Song khi chúng ta thiết kế một lớp làm cơ sở cho các lớp dẫn xuất khác, chúng ta mong muốn rằng một số thành phần của lớp, khách hàng không được quyền sử dụng, nhưng cho phép các lớp dẫn xuất được quyền sử dụng. Muốn vậy chúng ta đưa các thành phần đó vào mục protected. Như vậy các thành phần nằm trong mục protected là các thành phần được bảo vệ đối với khách hàng, nhưng các lớp dẫn xuất được quyền truy cập. Hình 3.1 minh hoạ quyền truy cập đến các mục public, protected và private của một lớp. Đến đây chúng ta có thể đưa ra cấu trúc tổng quát của một định nghĩa lớp:

```
class  tên_lớp
{
 public:
        danh sách các thành phần công khai
 protected :
        danh sách các thành phần được bảo vệ
```

private :

danh sách các thành phần cá nhân

};



Hình 3.1. Quyền truy cập đến các thành phần của lớp

Các dạng thừa kế public, private và protected

Khi xây dựng một lớp dẫn xuất từ một lớp cơ sở, chúng ta có thể sử dụng một trong ba dạng thừa kế: public, private hay protected. Tức là, định nghĩa một lớp dẫn xuất được bắt đầu bởi:

class tên_lớp_dẫn_xuất : dạng_thừa_kế tên_lớp_cơ_sở

Trong đó, dạng_thừa_kế là một trong ba từ khoá chỉ định dạng thừa kế: public, private, protected. Dù dạng thừa kế là gì (là public hoặc private hoặc protected) thì lớp dẫn xuất đều có quyền truy cập đến các thành phần trong mục public và protected của lớp cơ sở. Như chúng ta đã nhấn mạnh, lớp dẫn xuất được thừa kế các thành phần của lớp cơ sở, trừ ra các hàm kiến tạo, hàm huỷ và các hàm được định nghĩa lại. Vấn đề được đặt ra là, quyền truy cập đến các thành phần được thừa kế của lớp dẫn xuất như thế nào? Câu trả lời phụ thuộc vào dạng thừa kế. Sau đây là các quy tắc quy định quyền truy cập đến các thành phần được thừa kế của lớp dẫn xuất.

1. Thừa kế public: các thành phần public và protected của lớp cơ sở trở thành các thành phần public và protected tương ứng của lớp dẫn xuất.
2. Thừa kế protected: các thành phần public và protected của lớp cơ sở trở thành các thành phần protected của lớp xuất.
3. Thừa kế private: các thành phần public và protected của lớp cơ sở trở thành các thành phần private của lớp dẫn xuất.
4. Trong bất kỳ trường hợp nào, các thành phần private của lớp cơ sở, lớp dẫn xuất đều không có quyền truy cập tới mặc dầu nó được thừa kế.

Trong ba dạng thừa kế đã nêu, thì thừa kế public là quan trọng nhất. Nó được sử dụng để mở rộng một định nghĩa lớp, tức là để cài đặt một lớp mới (lớp dẫn xuất) có đầy đủ các tính chất của lớp cơ sở và được bổ sung thêm các tính chất mới. Thừa kế private được sử dụng để cài đặt một lớp mới bằng các phương tiện của lớp cơ sở. Thừa kế protected ít được sử dụng.

Sự tương thích kiểu.

Khi lớp cơ sở là lớp cơ sở public, thì một lớp dẫn xuất có thể xem như lớp con của lớp cơ sở theo nghĩa thuyết tập, tức là mỗi đối tượng của lớp dẫn xuất là một đối tượng của lớp cơ sở, song điều ngược lại thì không đúng. Do đó, khi ta khai báo một con trỏ trỏ tới đối tượng của lớp cơ sở, thì trong thời

gian chạy con trỏ này có thể trỏ tới đối tượng của lớp dẫn xuất. Ví dụ, giả sử chúng ta có các khai báo sau:

```
class   Polygon { … };
class   Rectangle : public Polygon { … };
```

Chú ý rằng, mỗi lớp là một kiểu dữ liệu. Do đó, ta có thể khai báo các biến sau:

```
Polygon   *P ;
Polygon    Pobj;
Rectangle  *R;
Rectangle   Robj;
```

Khi đó:

```
P = new Polygon( ); // hợp lệ
P = new Rectangle ( ); // hợp lệ
R = new Polygon( ); // không hợp lệ
R = new Rectangle( ); // hợp lệ
P = R; // hợp lệ
R = P; // không hợp lệ
```

Chúng ta cũng có thể gán đối tượng lớp dẫn xuất cho đối tượng lớp cơ sở, song ngược lại thì không được phép, chẳng hạn:

```
Pobj = Robj; // hợp lệ
Robj = Pobj; // không hợp lệ
```

3.2 HÀM ẢO VÀ TÍNH ĐA HÌNH

Tính đa hình (polymorphism) để chỉ một hàm khai báo trong một lớp cơ sở có thể có nhiều dạng khác nhau trong các lớp dẫn xuất, tức là hàm có nội dung khác nhau trong các lớp dẫn xuất. Để hiểu được tính đa hình, chúng ta hãy xét một ví dụ đơn giản. Giả sử chúng ta có một lớp cơ sở:

```
class    Alpha
{
   public :
            ……
       void   Hello( )   const
       { cout << “I am Alpha” << endl ; }
            ……
} ;
```

Lớp Alpha chứa hàm Hello( ), hàm này in ra thông báo “I am Alpha”. Chúng ta xây dựng lớp Beta dẫn xuất từ lớp Alpha, lớp Beta cũng chứa hàm Hello( ) nhưng với nội dung khác: nó in ra thông báo “I am Beta”.

```
class   Beta : public   Alpha
{
  public :
            …….
     void   Hello( )   const
     { cout << “I am Beta” << endl ; }
            …….
} ;
```

Bây giờ, ta khai báo

Alpha   Obj;

Khi đó, lời gọi hàm Obj.Hello( ) sẽ cho in ra “I am Alpha”, tức là bản hàm Hello( ) trong lớp Alpha được thực hiện. Còn nếu ta khai báo

Beta   Obj ;

thì lời gọi hàm obj.Hello( ) sẽ sử dụng bản hàm trong lớp Beta và sẽ cho in ra “I am Beta”. Như vậy, bản hàm nào của hàm Hello( ) được sử dụng khi thực hiện một lời gọi hàm được quyết định bởi kiểu đã khai báo của đối tượng, tức là được quyết định trong thời gian dịch. Hoàn cảnh này được gọi

là **sự ràng buộc tĩnh (static binding**). Song sự ràng buộc tĩnh không đáp ứng được mong muốn của chúng ta trong tình huống sau đây:

Giả sử, Aptr là con trỏ trỏ tới đối tượng lớp Alpha:

Alpha *Aptr ;

Kiểu của con trỏ Aptr khi khai báo là kiểu tĩnh. Trong thời gian chạy, con trỏ Aptr có thể trỏ tới một đối tượng của lớp dẫn xuất. Kiểu của con trỏ Aptr lúc đó là kiểu động. Chẳng hạn, khi gặp các dòng lệnh:

Aptr = new Beta ;

Aptr →Hello( );

Aptr trỏ tới đối tượng của lớp Beta. Do đó, chúng ta mong muốn rằng, lời gọi hàm Aptr →Hello( ) sẽ cho in ra "I am Beta". Song đáng tiếc là không phải như vậy, kiểu tĩnh của con trỏ Aptr đã quyết định bản hàm Hello( ) trong lớp Alpha được thực hiện và cho in ra "I am Alpha".

Chúng ta có thể khắc phục được sự bất cập trên bằng cách khai báo hàm Hello( ) trong lớp cơ sở Alpha là **hàm ảo (virtual function**). Để chỉ một hàm là ảo, chúng ta chỉ cần viết từ khoá virtual trước khai báo hàm trong định nghĩa lớp cơ sở. Chẳng hạn, lớp Alpha được khai báo lại như sau:

```
class   Alpha
{
  public :
        ……
        virtual  void  Hello( )  const
        { cout << "I am Alpha << endl; }
        ……
};
```

Khi một hàm được khai báo là ảo trong lớp cơ sở, như hàm Hello( ) trong lớp Alpha, thì nó có thể được định nghĩa lại với các nội dung mới trong các lớp dẫn xuất. Chẳng hạn, trong lớp Beta:

```
class   Beta : public   Alpha
```

```
{
  public :
        ……..
        virtual   void   Hello( )   const
        { cout << “I am Beta” << endl; }
        ……..
};
```

Chúng ta xây dựng một lớp dẫn xuất khác Gama từ lớp cơ sở Alpha. Trong lớp Gama, hàm Hello( ) cũng được định nghĩa lại:

```
class   Gama : public   Alpha
{
  public :
        ……..
        virtual   void   Hello( )   const
        { cout << “I am Gama” << endl; }
        ……..
};
```

Như vậy, hàm ảo Hello( ) có ba dạng khác nhau. Chúng ta thử xem bản hàm nào được sử dụng trong chương trình sau:

```
int   main( )
{
  Alpha   Aobj;
  Beta   Bobj;
  Alpha   *Aptr;
  Aptr = &Aobj; // con trỏ Aptr trỏ tới đối tượng lớp Alpha.
  Aptr → Hello( ); // Bản hàm Hello( ) trong lớp Alpha được sử dụng.
  Aptr = &Bobj; // con trỏ Aptr trỏ tới đối tượng lớp Beta.
```

```
    Aptr → Hello( ); // Bản hàm Hello( ) trong lớp Beta được sử dụng.
    Aptr = new Gama( ); // con trỏ Aptr trỏ tới đối tượng lớp Gama.
    Aptr → Hello( ); // Bản hàm Hello( ) trong lớp Gama được sử dụng.
    return   0;
}
```

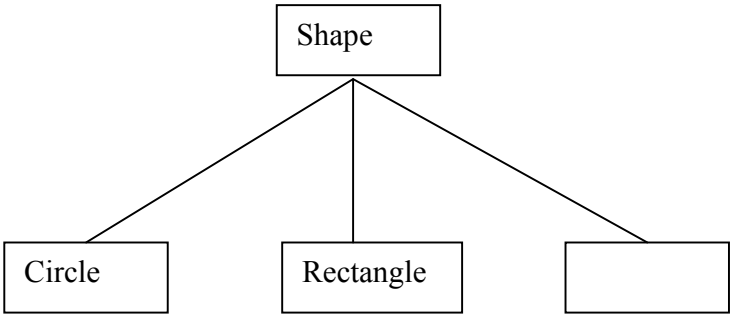
Tóm lại, một hàm được khai báo là ảo trong một lớp cơ sở, nó có thể được định nghĩa lại trong các lớp dẫn xuất và do đó nó có thể có nhiều dạng khác nhau trong các lớp dẫn xuất, chẳng hạn hàm Hello( ) có ba bản hàm khác nhau. Bản hàm nào được sử dụng trong lời gọi hàm (chẳng hạn, Aptr → Hello( )) được quyết định bởi kiểu của đối tượng mà con trỏ lớp cơ sở trỏ tới, tức là được xác định trong thời gian chạy. Điều này được gọi là **sự ràng buộc động (dynamic binding**). Như vậy, một hàm được khai báo là ảo trong lớp cơ sở là hàm có tính đa hình, tức là hàm có nhiều dạng khác nhau. Dạng hàm thích hợp được lựa chọn để thực hiện phụ thuộc vào kiểu động của đối tượng kích hoạt hàm.

Khi thiết kế một lớp làm cơ sở cho các lớp dẫn xuất khác, chúng ta cần chú ý đến các luật tổng quát sau:

- Các hàm cần định nghĩa lại trong các lớp dẫn xuất cần phải là ảo.
- Hàm kiến tạo không thể là ảo, song hàm huỷ cần phải là ảo.

3.3 LỚP CƠ SỞ TRỪU TƯỢNG

Giả sử chúng ta cần thiết kế các lớp sau: lớp các hình tròn (Circle), lớp các hình chữ nhật (Rectangle), và nhiều lớp các hình phẳng có dạng đặc biệt khác. Trong các lớp đó chúng ta cần phải đưa vào các hàm thành phần thực hiện các hành động có đặc điểm chung cho tất cả các loại hình, chẳng hạn tính chu vi, tính diện tích, vẽ hình, … Trong tình huống này, chúng ta cần thiết kế một lớp, lớp các hình (Shape), làm cơ sở để dẫn xuất ra các lớp Circle, Rectangle, …, như được minh hoạ trong hình sau:



Lớp Shape sẽ chứa các khai báo các hàm thực hiện các xử lý có đặc điểm chung cho các lớp dẫn xuất, chẳng hạn các hàm tính chu vi Perimeter( ), hàm tính diện tích Area( ), … Song chúng ta không thể cài đặt được các hàm này trong lớp Shape (bởi vì chúng ta không thể tính được chu vi và diện tích của một hình trừu tượng), các hàm này sẽ được cài đặt trong các lớp dẫn xuất từ lớp Shape, chẳng hạn được cài đặt trong các lớp Circle, Rectangle, …Và do đó, trong lớp Shape chúng được khai báo là **hàm ảo thuần tuý (pure virtual function**) hay còn gọi là **hàm trừu tượng (abstract function**). Một hàm thành phần trong một lớp được gọi là ảo thuần tuý (hay trừu tượng) nếu nó chỉ được khai báo, nhưng không được định nghĩa trong lớp đó. Một hàm ảo thuần tuý được khai báo bằng cách khai báo nó là ảo và đặt = 0 ở sau mẫu hàm. Ví dụ, để khai báo các hàm Permeter( ) và Area( ) là ảo thuần tuý, ta viết:

```
virtual double Perimeter( ) const = 0;
virtual double Area( ) const = 0;
```

Một lớp chứa ít nhất một hàm ảo thuần tuý được gọi là **lớp cơ sở trừu tượng (abstract base class**). Lớp cơ sở trừu tượng không có đối tượng nào

cả, nó chỉ được sử dụng làm cơ sở để xây dựng các lớp khác. Lớp cơ sở trừu tượng chứa các hàm ảo thuần tuý biểu diễn các xử lý có đặc điểm chung cho các lớp đối tượng khác nhau. Các hàm đó sẽ được định nghĩa trong các lớp dẫn xuất. Và như vậy, các hàm ảo thuần tuý trong một lớp cơ sở trừu tượng là các hàm có tính đa hình. Sử dụng tính đa hình, người lập trình có thể viết các phần mềm dễ dàng hơn khi mở rộng, có tính khái quát cao, dễ đọc, dễ hiểu,… Mô hình thiết kế các lớp dẫn xuất từ lớp cơ sở trừu tượng là mô hình thiết kế mà chúng ta cần sử dụng trong các hoàn cảnh tương tự như khi thiết kế các lớp đối tượng hình học phẳng.

Để làm ví dụ minh hoạ cho khái niệm lớp cơ sở trừu tượng, chúng ta xây dựng lớp Shape. Một lớp cơ sở trừu tượng có thể chứa các thành phần dữ liệu là chung cho tất cả các lớp dẫn xuất. Chẳng hạn, lớp Shape chứa một biến thành phần private có tên là name để lưu tên các loại hình. Chúng ta đưa vào lớp Shape một hàm kiến tạo, nó không được gọi trực tiếp để khởi tạo ra đối tượng của lớp Shape (vì Shape là lớp trừu tượng, nên không có đối tượng nào cả), song nó sẽ được gọi để khởi tạo các đối tượng của các lớp dẫn xuất. Lớp Shape chứa hai hàm ảo thuần tuý: hàm Perimeter( ) và hàm Area( ). Ngoài các hàm ảo thuần tuý, lớp cơ sở trừu tượng còn có thể chứa các hàm ảo và các hàm thành phần khác. Chẳng hạn, lớp Shape chứa hàm operator <= để so sánh các hình theo diện tích và hàm ảo print để in ra một số thông tin về các hình. Định nghĩa lớp trừu tượng Shape được cho trong hình 3.2.

```
class   Shape
{
  public:
      Shape (const string & s = “  ”)
          : name (s) {  }
      ~ Shape( ) {  }
      virtual  double  Perimeter( )  const = 0;
```

```
        virtual  double  Area( )  const = 0;
        bool  operator <=(const Shape & sh)  const
        { return   Area( ) <=  sh. Area( ); }
        virtual  void  Print (ostream & out = cout)  const
        { out << name << "of area" << Area( ) ; }
    private:
        string name;
};
```

Hình 3.2. Lớp cơ sở trừu tượng Shape.

Chú ý rằng, hàm Print đã được cài đặt trong lớp Shape, nhưng nó được khai báo là ảo, mục đích là để bạn có thể cài đặt lại hàm này trong các lớp dẫn xuất để in ra các thông tin khác nếu bạn muốn. Sử dụng hàm Print, bạn có thể định nghĩa lại toán tử << để nó in ra các thông tin về các hình như sau:

```
ostream & operator << (ostream & out, const  Shape & sh)
{
    sh.Print (out);
    return   out;
}
```

Bây giờ dựa trên lớp cơ sở trừu tượng Shape, chúng ta sẽ xây dựng một loạt lớp dẫn xuất: lớp các hình có dạng đặc biệt, chẳng hạn các lớp Circle, Rectangle,… Lớp Rectangle được thiết kế như sau: ngoài thành phần dữ liệu name được thừa kế từ lớp Shape, nó chứa hai thành phần dữ liệu khác là length (chỉ chiều dài) và width (chỉ chiều rộng của hình chữ nhật). Lớp chứa hàm kiến tạo để khởi tạo nên hình chữ nhật có chiều dài là l, chiều rộng là w.

```
Rectangle ::  Rectangle (double l = 0.0, double w = 0.0)
: Shape (“rectangle”)
{
  length = l;
  width = r ;
}
```

Chú ý rằng, trong hàm kiến tạo trên, chúng ta đã gọi hàm kiến tạo của lớp Shape để đặt tên cho hình. Trong lớp Rectangle, chúng ta cần cung cấp định nghĩa cho các hàm Perimeter( ) và Area( ) được khai báo là trừu tượng trong lớp Shape. Chúng ta cũng định nghĩa lại hàm ảo Print để nó cho biết thêm một số thông tin khác về hình chữ nhật. Hàm Print được cài đặt như sau:

```
void   Rectangle :: Print (ostream & out = cout)  const
{
  Shape :: Print(out);
  out << “and of length” << length
      << “and of width” << width;
}
```

Lớp Circle được thiết kế một cách tương tự. Định nghĩa lớp Rectangle và lớp Circle được cho trong hình 3.3.

```
class   Rectangle : public  Shape
{
  public:
     Rectangle (double l = 0.0, double w = 0.0)
     : Shape (“rectangle”), length(l), width(w) { }
     double   GetLength( )   const
     { return   length; }
```

```
    double  GetWidth( )  const
    { return   width; }
    double   Perimeter( )  const
    {return   2*(length + width); }
    double   Area( )   const
    { return   length * width; }
    void   Print(ostream & out = cout)
    {
      // như đã trình bày ở trên.
    }
  private :
        double   length;
        double   width;
};

class   Circle : public  Shape
{
   public :
     const  double  PI = 3.14159;
     Circle(double r = 0.0)
     : Shape("circle"), radius (r) { }
     double  GetRadius( ) const
     { return   radius; }
     double   Perimeter( ) const
     { return   2 * radius * PI; }
     double   Area( ) const
     { return   radius * radius * PI; }
     void   Print (ostream & out = cout) const
     {
       Shape :: Print(out);
       out << "of radius" << radius;
```

```
    }
  private :
    double   radius ;
} ;
```

Hình 3.3. Các lớp Rectangle và Circle.

Tóm lại, trong một lớp các hàm ảo thuần tuý (hay các hàm trừu tượng) là các hàm chỉ được khai báo, nhưng không được định nghĩa. Một lớp chứa ít nhất một hàm ảo thuần tuý được gọi là lớp trừu tượng, nó không có đối tượng nào cả, nhưng được sử dụng làm cơ sở để xây dựng các lớp dẫn xuất. Nó cung cấp cho người sử dụng một dao diện chung cho tất cả các lớp dẫn xuất. Trong một lớp dẫn xuất từ lớp cơ sở trừu tượng, chúng ta có thể cung cấp định nghĩa cho các hàm trừu tượng của lớp cơ sở. Một hàm ảo thuần tuý cũng giống như hàm ảo thông thường ở đặc điểm là hàm có tính đa hình, bản hàm nào (trong số các bản hàm được cài đặt ở các lớp dẫn xuất) được thực hiện phụ thuộc vào đối tượng kích hoạt hàm thuộc lớp dẫn xuất nào, đối tượng đó được xác định trong thời gian chạy (sự ràng buộc động).

Mặc dù lớp cơ sở trừu tượng không có đối tượng nào cả, song chúng ta có thể khai báo một biến tham chiếu đến lớp cơ sở trừu tượng, chẳng hạn trong khai báo hàm sau:

```
ostream & operator << (ostream & out, const Shape & sh);
```

Chúng ta cũng có thể khai báo các con trỏ trỏ tới lớp cơ sở trừu tượng, chẳng hạn

```
Shape * aptr, * bptr;
aptr = new Rectangle (2.3, 5.4); // hợp lệ
bptr = new Shape( “circle” ) ; // không hợp lệ
```

BÀI TẬP

1. Giả sử chúng ta xây dựng lớp Beta dẫn xuất từ lớp cơ sở Alpha; ngoài các thành phần dữ liệu được thừa kế từ lớp Alpha, lớp Beta còn chứa thành phần dữ liệu là đối tượng của một lớp khác: lớp Gama và các thành phần dữ liệu khác. Nếu ta không cung cấp cho lớp Beta hàm kiến tạo, hàm huỷ, toán tử gán thì C++ sẽ cung cấp hàm kiến tạo mặc định tự động, hàm kiến tạo copy tự động, hàm huỷ tự động và toán tử gán tự động cho lớp dẫn xuất Beta. Hãy cho biết hiệu quả các hàm tự động đó.

2. Giả sử ta xây dựng lớp dẫn xuất Beta như trong bài tập 1. Nói chung, chúng ta cần đưa vào lớp Beta hàm kiến tạo. Hàm kiến tạo của lớp dẫn xuất Beta thực hiện các nhiệm vụ gì? Nó cần được cài đặt như thế nào? Hãy đưa ra ví dụ minh hoạ.

3. Giả sử trong giao diện của lớp Alpha có chứa hàm foo( ) thực hiện một nhiệm vụ nào đó.

```
class Alpha
{
    public :
        void foo( ) ;
        …
} ;
```

Giả sử ta xây dựng lớp dẫn xuất Beta từ lớp cơ sở Alpha với dạng thừa kế private :

```
class Beta : private Alpha
```

Chúng ta muốn hàm foo( ) là hàm public của lớp Beta. Để có điều đó, ta cần làm gì?

4. Cho các khai báo lớp như sau:

```
class      Alpha
{
  private :
     int   w ;
  protected :
     int   x ;
  public :
     Alpha(  ) { w = 1; x = 2; }
     void   foo( ) {cout << "w =" << w << endl ; }
     virtual   void   bar(  )
     {cout << "x =" << x << endl; }
} ;

class    Beta : public   Alpha
{
   private :
     int   y ;
   protected :
     int   z ;
   public:
     Beta(  ) {y = 3 ; z = 4; }
     void   foo(  ) {cout << "y =" << y << endl; }
     void   bar(  ) {cout << "z =" << z << endl; }
} ;
```

Hãy cho biết chương trình sau in ra cái gì? Giải thích?

```
int main(  )
{
   Alpha   A;
   Beta   B;
```

```
    Alpha* Bptr = &A ;
    A.foo( ) ;
    A.bar( ) ;
    B.foo( ) ;
    B.bar( ) ;
    Bptr → foo( ) ;
    Bptr → bar( ) ;
    Bptr = &B ;
    Bptr → foo( ) ;
    Bptr → bar( ) ;
}
```

CHƯƠNG 4

DANH SÁCH

Danh sách là cấu trúc dữ liệu tuyến tính, danh sách được tạo nên từ các phần tử dữ liệu được sắp xếp theo một thứ tự xác định. Danh sách là một trong các cấu trúc dữ liệu cơ bản được sử dụng thường xuyên nhất trong các thuật toán. Danh sách còn được sử dụng để cài đặt nhiều KDLTT khác. Trong chương này, chúng ta sẽ xác định KDLTT danh sách và nghiên cứu phương pháp cài đặt KDLTT danh sách bởi mảng. Sau đó, chúng ta sẽ sử dụng danh sách để cài đặt KDLTT tập động.

4.1 KIỂU DỮ LIỆU TRỪU TƯỢNG DANH SÁCH

Danh sách là một khái niệm được sử dụng thường xuyên trong thực tiễn. Chẳng hạn, chúng ta thường nói đến danh sách sinh viên của một lớp, danh sách các số điện thoại, danh sách thí sinh trúng tuyển, …

Danh sách được định nghĩa là một dãy hữu hạn các phần tử:

$L = (a_1, a_2, \dots , a_n)$

trong đó a_i (i = 1, …, n) là phần tử thứ i của danh sách. Cần lưu ý rằng, số phần tử của danh sách, được gọi là **độ dài** của danh sách, có thể thay đổi theo thời gian. Và một phần tử dữ liệu có thể xuất hiện nhiều lần trong danh sách ở các vị trí khác nhau. Chẳng hạn, trong danh sách các số nguyên sau:

L = (3, 5, 5, 0, 7, 5)

số nguyên 5 xuất hiện 3 lần ở các vị trí 2, 3, và 6. Một đặc điểm quan trọng khác của danh sách là các phần tử của nó có thứ tự tuyến tính, thứ tự này được xác định bởi vị trí của các phần tử trong danh sách. Khi độ dài của danh sách bằng không (n = 0), ta nói danh sách rỗng. Nếu danh sách không rỗng ($n \geq 1$), thì phần tử đầu tiên a_1 được gọi là **đầu** của danh sách, còn phần tử cuối cùng a_n được gọi là **đuôi** của danh sách.

Không có hạn chế nào trên kiểu dữ liệu của các phần tử trong danh sách. Khi mà tất cả các phần tử của danh sách cùng một kiểu, ta nói danh sách là danh sách thuần nhất. Trong trường hợp tổng quát, một danh sách có thể chứa các phần tử có kiểu khác nhau, đặc biệt một phần tử của danh sách có thể lại là một danh sách. Chẳng hạn

L = (An, (20, 7, 1985), 8321067)

Trong danh sách này, phần tử đầu tiên là một xâu ký tự, phần tử thứ hai là danh sách các số nguyên, phần tử thứ ba là số nguyên. Danh sách này có thể sử dụng để biểu diễn, chẳng hạn, một sinh viên có tên là An, sinh ngày 20/7/1985, có số điện thoại 8321067. Danh sách (tổng quát) là cấu trúc dữ liệu cơ bản trong các ngôn ngữ lập trình chuyên dụng cho các xử lý dữ liệu phi số, chẳng hạn Prolog, Lisp. Trong sách này, chúng ta chỉ quan tâm tới các danh sách thuần nhất, tức là khi nói đến danh sách thì cần được hiểu đó là danh sách mà tất cả các phần tử của nó cùng một kiểu.

Khi sử dụng danh sách trong thiết kế thuật toán, chúng ta cần dùng đến một tập các phép toán rất đa dạng trên danh sách. Sau đây là một số phép toán chính. Trong các phép toán này, L ký hiệu một danh sách bất kỳ có độ dài $n \geq 0$, x ký hiệu một phần tử bất kỳ cùng kiểu với các phần tử của L và i là số nguyên dương chỉ vị trí.

1. Empty(L). Hàm trả về true nếu L rỗng và false nếu L không rỗng.
2. Length(L). Hàm trả về độ dài của danh sách L.
3. Insert(L, x, i). Thêm phần tử x vào danh sách L tại vị trí i. Nếu thành công thì các phần tử $a_i, a_{i+1}, \ldots, a_n$ trở thành các phần tử $a_{i+1}, a_{i+2}, \ldots, a_{n+1}$ tương ứng, và độ dài danh sách là n+1. Điều kiện để phép toán xen có thể thực hiện được là i phải là vị trí hợp lý, tức $1 \leq i \leq n+1$, và không gian nhớ dành để lưu danh sách L còn chỗ.
4. Append(L, x). Thêm x vào đuôi danh sách L, độ dài danh sách tăng lên 1.
5. Delete(L, i). Loại phần tử ở vị trí thứ i trong danh sách L. Nếu thành công, các phần tử $a_{i+1}, a_{i+2}, \ldots, a_n$ trở thành các phần tử $a_i, a_{i+1}, \ldots, a_{n-1}$ tương ứng, và độ dài danh sách là n-1. Phép toán loại

chỉ được thực hiện thành công khi mà danh sách không rỗng và i là vị trí thực sự trong danh sách, tức là $1 \leq i \leq n$.

6. Element(L, i). Tìm biết phần tử ở vị trí thứ i của L. Nếu thành công hàm trả về phần tử ở vị trí i. Điều kiện để phép toán tìm thực hiện thành công cũng giống như đối với phép toán loại.

Chúng ta quan tâm đến các phép toán trên, vì chúng là các phép toán được sử dụng thường xuyên khi xử lý danh sách. Hơn nữa, đó còn là các phép toán sẽ được sử dụng để cài đặt nhiều KDLTT khác như tập động, từ điển, ngăn xếp, hàng đợi, hàng ưu tiên. Nhưng trong các chương trình có sử dụng danh sách, nhiều trường hợp chúng ta cần thực hiện các phép toán đa dạng khác trên danh sách, đặc biệt chúng ta thường phải đi qua danh sách (duyệt danh sách) để xem xét lần lượt từng phần tử của danh sách từ phần tử đầu tiên đến phần tử cuối cùng và tiến hành các xử lý cần thiết với mỗi phần tử của danh sách. Để cho quá trình duyệt danh sách được thực hiện thuận tiện, hiệu quả, chúng ta xác định các phép toán sau đây. Các phép toán này tạo thành bộ công cụ lặp (iteration). Tại mỗi thời điểm, phần tử đang được xem xét của danh sách được gọi là **phần tử hiện thời** và vị trí của nó trong danh sách được gọi là **vị trí hiện thời.**

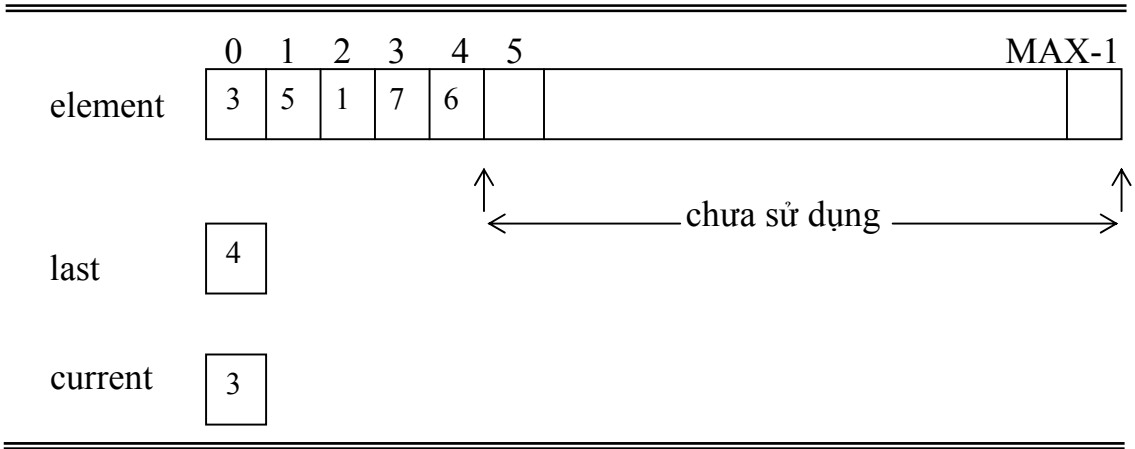
7. Start(L). Đặt vị trí hiện thời là vị trí đầu tiên trong danh sách L.
8. Valid(L). Trả về true, nếu vị trí hiện thời có chứa phần tử của danh sách L, nó trả về false nếu không.
9. Advance(L). Chuyển vị trí hiện thời tới vị trí tiếp theo trong danh sách L.
10. Current(L). Trả về phần tử tại vị trí hiện thời trong L.
11. Add(L, x). Thêm phần tử x vào trước phần tử hiện thời, phần tử hiện thời vẫn còn là phần tử hiện thời.
12. Remove(L). Loại phần tử hiện thời khỏi L. Phần tử đi sau phần bị loại trở thành phần tử hiện thời**.**

Chúng ta đã đặc tả KDLTT danh sách. Bây giờ chuyển sang giai đoạn cài đặt danh sách.

4.2 CÀI ĐẶT DANH SÁCH BỞI MẢNG

Chúng ta sẽ cài đặt KDLTT danh sách bởi các lớp C + +. Có nhiều cách thiết kế lớp cho KDLTT danh sách, điều đó trước hết là do danh sách có thể biểu diễn bởi các cấu trúc dữ liệu khác nhau. Các thiết kế lớp khác nhau cho danh sách sẽ được trình bày trong chương này và chương sau. Trong mục này chúng ta sẽ trình bày cách cài đặt danh sách bởi mảng (mảng tĩnh). Đây là phương pháp cài đặt đơn giản và tự nhiên nhất.

Chúng ta sẽ sử dụng một mảng element có cỡ là MAX để lưu các phần tử của danh sách. Các phần tử của danh sách sẽ được lần lượt lưu trong các thành phần của mảng element[0], element[1], … , element[n-1], nếu danh sách có n phần tử. Tức là, danh sách được lưu trong đoạn đầu element[0…n-1] của mảng, đoạn sau của mảng, element[n.. MAX-1], là không gian chưa được sử dụng. Cần lưu ý rằng, phần tử thứ i của danh sách (i = 1, 2, …) được lưu trong thành phần element[i-1] của mảng. Cần có một biến last ghi lại chỉ số sau cùng mà tại đó mảng có chứa phần tử của danh sách. Vị trí hiện thời được xác định bởi biến current, nó là chỉ số mà element[current] chứa phần tử hiện thời của danh sách. Chẳng hạn, giả sử chúng ta có danh sách các số nguyên L = (3, 5, 1, 7, 6) và phần tử hiện thời đứng ở vị trí thứ 4 trong danh sách, khi đó danh sách L được biểu diễn bởi cấu trúc dữ liệu được mô tả trong hình 4.1.



Hình 4.1.Biểu diễn danh sách bởi mảng.

Chúng ta muốn thiết kế lớp danh sách sao cho người lập trình có thể sử dụng nó để biểu diễn danh sách với các phần tử có kiểu tuỳ ý. Do đó, lớp danh sách được thiết kế là lớp khuôn phụ thuộc tham biến kiểu Item như sau:

```
template   <class Item>
class      List
{
 public :
      static  const  int  MAX = 50; // khai báo cỡ của mảng.
      // các hàm thành phần.
 private :
      Item   element[MAX] ;
      int    last ;
      int    current ;
} ;
```

Bây giờ cần phải thiết kế các hàm thành phần của lớp List. Ngoài các hàm thành phần tương ứng với các phép toán trên danh sách, chúng ta đưa vào một hàm kiến tạo mặc định, hàm này khởi tạo nên danh sách rỗng. Lớp List chứa ba biến thành phần đã khai báo như trên, nên không cần thiết phải đưa vào hàm kiến tạo copy, hàm huỷ và hàm toán tử gán, vì chỉ cần sử dụng các hàm kiến tạo copy tự động, … do chương trình dịch cung cấp là đủ. Định nghĩa đầy đủ của lớp List được cho trong hình 4.2.

```
// File đầu list.h
# ifndef  LIST_H
# define LIST_H
# include <assert.h>
```

```
template   <class, Item>
class    List
{
  public :
     static  const  int   MAX = 50 ;
     List ( )
     // Khởi tạo danh sách rỗng.
     { last = -1; current = -1; }
     bool  Empty(  )  const
     // Kiểm tra danh sách có rỗng không.
     // Postcondition: Hàm trả về true nếu danh sách rỗng và false
     // nếu không
     { return   last < 0; }
     int  Length(  )  const
     // Xác định độ dài danh sách.
     // Postcondition: Trả về số phần tử trong danh sách.
     {return   last+1; }
     void  Insert(const Item&x, int i);
     // Xen phần tử x vào vị trí thứ i trong danh sách.
     // Precondition: Length(  ) < MAX và 1 ≤ i ≤ Length( )
     // Postcondition: các phần tử của danh sách kể từ vị trí thứ i
     // được đẩy ra sau một vị trí, x nằm ở vị trí i.
     void  Append(const  Item&x);
     // Thêm phần tử x vào đuôi danh sách.
     // Precondition : Length( ) < MAX
     // Postcondition : x là đuôi của danh sách.

     void  Delete(int i)
     // Loại khỏi danh sách phần tử ở vị trí i.
     // Precondition: 1 ≤ i ≤ Length( )
     // Postcondition: phần tử ở vị trí i bị loại khỏi danh sách,
```

```
// các phần tử đi sau được đẩy lên trước một vị trí.

Item &   Element(int i)  const
// Tìm phần tử ở vị trí thứ i.
// Precondition: 1 ≤ i ≤ Length( )
// Postcondition: Trả về phần tử ở vị trí i.
   { assert (1<= i && i <= Length( ) ); return element[i - 1]; }
// Các hàm bộ công cụ lặp:

void   start( )
// Postcondition: vị trí hiện thời là vị trí đầu tiên của danh sách.
   {current = 0; }
bool   Valid( ) const
// Postcondition: Trả về true nếu tại vị trí hiện thời có phần tử
// trong danh sách, và trả về false nếu không.
   {return   current >= 0 && current <= last; }
void   Advance( )
// Precondition: Hàm Valid( ) trả về true.
// Postcondition: Vị trí hiện thời là vị trí tiếp theo trong danh
// sách.
   { assert (Valid( )); assert (Valid( )); current + +;}
Item &   Current( )  const
// Precondition: Hàm Valid( ) trả về true.
// Postcondition: Trả về phần tử hiện thời của danh sách.
   { assert (Valid( ));  return element[current]; }
void    Add(const Item& x);
// Precondition: Length( ) < MAX và hàm Valid( ) trả về true
// Postcondition: Phần tử x được xen vào trước phần tử
// hiện thời, phần tử hiện thời vẫn còn là phần tử hiện thời.

void   Remove( );
```

```
        // Precondition: hàm Valid( ) trả về true.
        // Postcondition: Phần tử hiện thời bị loại khỏi danh sách,
        // phần tử đi sau nó trở thành phần tử hiện thời.

    private :
        Item   element[MAX];
        int   last;
        int   current;
  };
# include "list.template"
# endif
```

Hình 4.2. Định nghĩa lớp List.

Bước tiếp theo chúng ta cần cài đặt các hàm thành phần của lớp List. Trước hết, nói về hàm kiến tạo mặc định, hàm này cần tạo ra một danh sách rỗng, do vậy chỉ cần đặt giá trị cho biến last là –1, giá trị của biến current cũng là –1. Hàm này được cài đặt là hàm inline. Với cách khởi tạo này, mỗi khi cần thêm phần tử mới vào đuôi danh sách (kể cả khi danh sách rỗng), ta chỉ cần tăng chỉ số last lên 1 và đặt phần tử cần thêm vào thành phần mảng element[last].

Hàm Append được định nghĩa như sau:

```
template  <class  Item>
void   List<Item> :: Append (const Item& x)
{ assert (Length( ) < MAX);
  last + + ;
  element[last] = x;
}
```

Hàm Insert. Để xen phần tử x vào vị trí thứ i của danh sách, tức là x cần đặt vào thành phần element[i - 1] trong mảng, chúng ta cần dịch chuyển đoạn element[i – 1 … last] ra sau một vị trí để có chỗ cho phần tử x. Hàm Insert có nội dung như sau:

```
template  <class  Item>
void   List<Item> :: Insert(const Item& x, int i)
{
  assert (Length( ) < MAX && 1 <= i && i <= Length( ));
  last + +;
  for (int k = last; k >= i; k - -)
        element[k] = element[k - 1];
  element[i - 1] = x;
}
```

Hàm Add. Hàm này cũng tương tự như hàm Insert, nó có nhiệm vụ xen phần tử mới x vào vị trí của phần tử hiện thời. Vì phần tử hiện thời được đẩy ra sau một vị trí, nên chỉ số hiện thời phải được tăng lên 1.

```
template <class  Item>
void   List<Item> :: Add(const Item& x)
{
  assert (Length( ) < MAX && Valid( ));
  last + +;
  for(int k = last; k > current; k - -)
       element[k] = element[k - 1];
  element[current] = x;
  current + +;
}
```

Hàm Delete. Muốn loại khỏi danh sách phần tử ở vị trí thứ i, chúng ta cần phải đẩy đoạn cuối của danh sách kể từ vị trí i + 1 lên trước 1 vị trí, và giảm chỉ số last đi 1.

```
template  <class  Item>
void   List<Item> :: Delete(int i)
{
  assert (1 <= i && i <= Length( ));
  for (int k = i – 1; k < last; k + +)
        element[k] = element[k + 1];
  last - - ;
}
```

Hàm Remove. Hàm này được cài đặt tương tự như hàm Delete.

```
template  <class  Item>
void   List<Item> :: Remove( )
{
  assert(Valid( ));
  for (int k = current; k < last; k + +)
  element[k] = element[k + 1];
  last - - ;
}
```

Tất cả các hàm còn lại đều rất đơn giản và được cài đặt là hàm inline.

Bây giờ chúng ta đánh giá hiệu quả của các phép toán của KDLTT danh sách, khi mà danh sách được cài đặt bởi mảng. Ưu điểm của cách cài đặt này là nó cho phép ta truy cập trực tiếp tới từng phần tử của danh sách, vì phần tử thứ i của danh sách được lưu trong thành phần mảng element[i - 1]. Nhờ đó mà thời gian của phép toán Element(i) là O(1). Giả sử danh sách có độ dài n, để xen phần tử mới vào vị trí thứ i trong danh sách, chúng ta cần

đẩy các phần tử lưu ở các thành phần mảng từ element[i -1] đến element[n - 1] ra sau một vị trí. Trong trường hợp xấu nhất (khi xen vào vị trí đầu tiên trong danh sách), cần n lần đẩy, vì vậy thời gian của phép toán Insert là O(n). Phân tích một cách tượng tự, chúng ta thấy rằng thời gian chạy của các phép toán Delete, Add và Remove cũng là O(n), trong đó n là độ dài của danh sách. Thời gian của tất cả các phép toán còn lại đều là O(1).

Như chúng ta đã nói, các phép toán trong bộ công cụ lặp cho phép chúng ta có thể tiến hành dễ dàng các xử lý danh sách cần đến duyệt danh sách. Chẳng hạn, giả sử L là danh sách các số nguyên, để loại khỏi danh sách L tất cả các số nguyên bằng 3, chúng ta có thể viết:

```
L. Start( );
while (L.Valid( ))
    if (L.Current( ) = = 3)
                    L.Remove( ) ;
    else    L.Advance( ) ;
```

Chúng ta cũng có thể in ra tất cả các phần tử của danh sách bằng cách sử dụng vòng lặp for như sau:

```
for (L.Start( ); L.Valid( ); L.Advance( ))
        cout << L.Current( ) << endl;
```

Nhận xét. Cài đặt danh sách bởi mảng có ưu điểm cơ bản là nó cho phép truy cập trực tiếp tới từng phần tử của danh sách. Nhờ đó chúng ta có thể cài đặt rất thuận tiện phép toán tìm kiếm trên danh sách, đặc biệt khi danh sách là danh sách được sắp (các phần tử của nó được sắp xếp theo thứ tự không tăng hoặc không giảm), nếu lưu danh sách được sắp trong mảng, chúng ta có thể cài đặt dễ dàng phương pháp tìm kiếm nhị phân. Phương pháp tìm kiếm nhị phân là một kỹ thuật tìm kiếm rất hiệu quả và sẽ được nghiên cứu trong mục 4.4.

Chúng ta đã cài đặt danh sách bởi mảng tĩnh, mảng có cỡ MAX cố định. Khi danh sách phát triển, tới lúc nào đó mảng sẽ đầy, và lúc đó các phép toán Insert, Append, Add sẽ không thể thực hiện được. Đó là nhược điểm chính của cách cài đặt danh sách bởi mảng tĩnh.

Bây giờ nói về cách thiết kế lớp List. Trong lớp List này, tất cả các hàm trong bộ công cụ lặp được đưa vào các hàm thành phần của lớp và biến lưu vị trí hiện thời cũng là một biến thành phần của lớp. Thiết kế này có vấn đề: chỉ có một biến hiện thời, do đó chúng ta không thể tiến hành đồng thời hai hoặc nhiều phép lặp khác nhau trên cùng một danh sách.

Mục sau sẽ trình bày một phương pháp thiết kế lớp List khác, nó khắc phục được các nhược điểm đã nêu trên.

4.3 CÀI ĐẶT DANH SÁCH BỞI MẢNG ĐỘNG

Trong mục này chúng ta sẽ thiết kế một lớp khác cài đặt KDLTT danh sách, lớp này được đặt tên là Dlist (Dynamic List). Lớp Dlist khác với lớp List đã được trình bày trong mục 4.2 ở hai điểm. Thứ nhất, danh sách được lưu trong mảng được cấp phát động. Thứ hai, các hàm trong bộ công cụ lặp được tách ra và đưa vào một lớp riêng: lớp công cụ lặp trên danh sách, chúng ta sẽ gọi lớp này là DlistIterator. Với cách thiết kế này, chúng ta sẽ khắc phục được các nhược điểm của lớp List đã được nêu ra trong nhận xét ở cuối mục 4.2.

Lớp Dlist chưa ba thành phần dữ liệu: Biến con trỏ element trỏ tới mảng được cấp phát động để lưu các phần tử của danh sách. Biến size lưu cỡ của mảng, và biến last lưu chỉ số cuối cùng mà tại đó mảng chứa phần tử của danh sách.

Lớp Dlist chứa tất cả các hàm thành phần thực hiện các phép toán trên danh sách giống như trong lớp List, trừ ra các hàm công cụ lặp (các hàm này sẽ được đưa vào lớp DlistIterator). Chúng ta đưa vào lớp Dlist hai hàm kiến tạo. Hàm kiến tạo một tham biến nguyên là cỡ của mảng được cấp phát động và hàm kiến tạo copy. Chúng ta cần phải đưa vào lớp Dlist hàm huỷ để thu

hồi bộ nhớ đã cấp phát cho mảng element, khi mà đối tượng không còn cần thiết nữa. Lớp Dlist cũng cần có hàm toán tử gán. Định nghĩa lớp Dlist được cho trong hình 4.3.

```
template <class  Item>
class   DlistIterator ;
// Khai báo trước lớp DlistIterator.

template <class  Item>
class    Dlist
{
  public:
     friend class DlistIterator<Item>;
     Dlist( )
     {element = NULL; size = 0; last = -1; }
     DList (int m);
     Dlist (const Dlist & L);
     ~ Dlist( )
     {delete [ ] element; }
     Dlist & operator = (const Dlist & L);
     inline  bool  Empty( ) const;
     inline  int  Length( ) const;
     void   Insert(const Item & x, int i);
     void   Append(const Item & x);
     void   Delete(int i);
     inline  Item & Element(int i);
  private:
     Item*   element;
     Int    size;
     Int    last;
```

```
};
```

Hình 4.3. Định nghĩa lớp Dlist

Chú ý rằng, trước khi định nghĩa lớp Dlist, chúng ta đã khai báo trước lớp DlistIterator. Khai báo này là cần thiết, bởi vì trong định nghĩa lớp Dlist, chúng ta xác định lớp DlistIterator là bạn của nó.

Sau đây chúng ta sẽ xem xét sự cài đặt các hàm thành phần của lớp Dlist. Các hàm Empty, Length, Delete và Retrieve là hàm hoàn toàn giống như trong lớp List.

Các hàm kiến tạo. Trước hết nói đến hàm kiến tạo có một tham biến nguyên m. Nhiệm vụ chính của nó là cấp phát một mảng động có cỡ là m để lưu các phần tử của danh sách.

```
template <class  Item>
Dlist<Item> :: Dlist(int m)
// Precondition. m là số nguyên dương.
// Postcondition. Một danh sách rỗng được khởi tạo, với khả năng tối
// đa chứa được m phần tử.
{
    element = new Item[m];
    assert (element != NULL);
    size = m;
    last = -1;
}
```

Hàm kiến tạo copy có trách nhiệm tạo ra một danh sách mới là bản sao của danh sách đã có L. Trước hết ta cần cấp phát một mảng động có cỡ là cỡ của mảng trong danh sách L, sau đó sao chép từng thành phần của mảng trong danh sách L sang mảng mới.

```
template <class Item>
Dlist<Item> :: Dlist(const Dlist<Item> & L)
{
    element = new Item[L.size];
    size = L.size;
    last = L.last;
    for (int i = 0; i <= last; i + +)
        element[i] = L.element[i];
}
```

Toán tử gán. Toán tử gán được cài đặt gần giống như hàm kiến tạo copy. Chỉ có điều cần lưu ý là, khi cỡ của mảng trong hai danh sách khác nhau, chúng ta mới cần cấp phát một mảng động có cỡ là cỡ của mảng trong danh sách nguồn, rồi thực hiện sao chép giống như trong hàm kiến tạo copy.

```
Dlist<Item> & Dlist<Item> :: operator = (const Dlist<Item> & L)
{
 if (size != L.size)
 {
    delete [ ] element;
    element = new Item[L.size];
    size = L.size;
 }
 last = L.last;
 for(int i = 0; i <= last; i + +)
    element[i] = L.element[i];
 return   *this;
}
```

Hàm Insert. Ưu điểm của cách cài đặt danh sách bởi mảng động là các hành động thêm phần tử mới vào danh sách luôn luôn được thực hiện. Bởi vì khi mà mảng đầy, chúng ta sẽ cấp phát một mảng động mới có cỡ gấp đôi cỡ của mảng cũ. Sau đó sao chép đoạn đầu [0… i-1] của mảng cũ sang mảng mới, đưa phần tử cần xen vào mảng mới, rồi lại chép đoạn còn lại của mảng cũ sang mảng mới. Hàm Insert được định nghĩa như sau:

```
template <class  Item>
void   Dlist<Item> :: Insert(const Item&x, int i)
{
   assert(i >= 0 && i <= last);
   if (Length( ) < size)
   {
     last + +;
     for (int k = last; k >= i; k - -)
       element[k] = element[k -1];
       element[i-1] = x;
   }
   else  // mảng element đầy
   {
      Item*   newArray = new Item[2 * size + 1]
      assert (newArray != NULL);
      for (int k = 0; k < i –1; k + + )
               newArray[k] = element[k];
      newArray[i - 1] = x;
      for (int k = i; k <= last + 1; k + + )
               newArray[k] = element[k - 1];
      delete [ ] element;
      element = newArray;
      last + +;
      size = 2 * size + 1;
```

```
    }
}
```

Hàm Append được cài đặt tương tự như hàm Insert, nhưng đơn giản hơn. Chúng tôi để lại cho độc giả, xem như bài tập.

Hàm Delete được cài đặt như trong lớp List (xem mục 4.2).

Bây giờ chúng ta thiết kế lớp công cụ lặp trên danh sách được cài đặt bởi mảng: lớp DlistIterator. Khai báo lớp công cụ lặp được sử dụng với lớp Dlist được cho trong hình 4.4. Lớp này chứa các phép toán của KDLTT danh sách liên quan tới phép lặp qua các phần tử của danh sách. Lớp DlistIterator chứa hai thành phần dữ liệu: biến current lưu số nguyên biểu diễn vị trí hiện thời, nó là chỉ số mà tại đó mảng lưu phần tử hiện thời; và biến LPtr lưu một con trỏ hằng trỏ tới đối tượng của lớp Dlist mà các phép toán công cụ lặp sẽ thực hiện trên nó.

```
# include  <assert.h>
template   <class Item>
class    DlistIterator
{
  public :
      DlistIterator (const Dlist<Item> & L) // Hàm kiến tạo
      { LPtr = &L; current = -1; }
      void   Start( )
      { current = 0; }
      bool   Valid( )  const
      { return 0 <= current && current <= LPtr ⟶ last; }
      void   Advance( )
      { assert (Valid( )); current + +;}
      Item &   Current( ) const
      { assert(Valid( )); return LPtr  ⟶  element[current]; }
```

```
        void    Add(const Item & x);
        void    Remove( );
    private :
        const   Dlist<Item>*   LPtr;
        int    current;
};
```

Hình 4.4. Định nghĩa lớp công cụ lặp.

Chú ý rằng, chúng ta đã khai báo lớp DlistIterator là bạn của lớp Dlist. Điều này là để khi cài đặt các hàm thành phần của lớp DlistIterator chúng ta có quyền truy cập trực tiếp tới các thành phần dữ liệu của lớp Dlist. Bây giờ chúng ta cài đặt hàm Add, hàm này được cài đặt hoàn toàn tương tự như hàm Insert trong lớp Dlist. Chỉ cần để ý rằng, ở đây chúng ta cần xen một phần tử mới vào vị trí hiện thời trong danh sách mà con trỏ LPtr trỏ tới.

```
template <class Item>
void   DlistIterator<Item> :: Add(const Item & x)
{
  if (LPtr →Length( ) < LPtr → size)
    {
      LPtr → last + +;
      for (int k = LPtr → last; k > current; k - - )
            LPtr → element[k] = LPtr → element[k - 1];
      LPtr  →element[current] = x;
    }
  else {
       Item* newArray = new Item[ 2 * (LPtr → size + 1) ];
       assert(newArray != NULL);
       for(int i = 0; i < current; i + + )
             newArray[i] = LPtr →element[i];
```

```
        newArray[current] = x;
        for(int i = current + 1; i <= LPtr →Length( ); i + +)
                newArray[i] = LPtr →element[i - 1];
        delete  [ ]  LPtr → element;
        LPtr → element = newArray;
        LPtr → size = 2 * (LPtr → size) + 1;
        LPtr → last + +;
        }
        current + +;
}
```

Dễ dàng thấy rằng, mặc dầu các phép toán Insert, Append và Add cài đặt phức tạp hơn các phép toán tương ứng khi danh sách được cài đặt bởi mảng tĩnh, song thời gian thực hiện các phép toán đó vẫn chỉ là O(n).

Chúng ta đưa ra một ví dụ minh hoạ cách sử dụng lớp công cụ lặp. Giả sử L là danh sách các số nguyên. Chúng ta cần thêm số nguyên 4 vào trước số nguyên 5 xuất hiện lần đầu trong danh sách, nếu L không chứa 5 thì thêm 4 vào đuôi danh sách. Trước hết, chúng ta phải tạo ra một đối tượng của lớp công cụ lặp gắn với danh sách L, và cần nhớ rằng, mọi phép lặp cần đến thao tác đầu tiên là Start. Xem đoạn mã sau:

```
Dlist<int>  L(100); // Danh sách L có thể chứa được tối đa 100
                    // số nguyên.
….
DlistIterator<Int>  itr(L); // khởi tạo đối tượng lặp itr gắn với
                            // danh sách L.
itr.Start( );
while(itr.Valid( ))
   if (itr.Current( ) = = 5)
           { itr.Add(4); break; }
   else    itr.Advance( );
if (!itr.Valid( ))
```

L. Append(4);

4.4 CÀI ĐẶT TẬP ĐỘNG BỞI DANH SÁCH. TÌM KIẾM TUẦN TỰ VÀ TÌM KIẾM NHỊ PHÂN

Nhớ lại rằng, mỗi đối tượng của KDLTT tập động là một tập các phần tử dữ liệu , và các phần tử dữ liệu chứa một thành phần dữ liệu được gọi là khoá. Chúng ta giả thiết rằng, trên các giá trị khoá có quan hệ thứ tự và các phần tử dữ liệu khác nhau có giá trị khoá khác nhau. Trên tập các phần tử dữ liệu S, chúng ta xác định các phép toán cơ bản sau:

1. Insert(S, x). Xen phần tử dữ liệu x vào tập S.
2. Delete(S, k). Loại khỏi tập S phần tử dữ liệu có khoá k.
3. Search(S, k). Tìm phần tử dữ liệu có giá trị khoá là k trong tập S. Hàm trả về true nếu tìm thấy và false nếu ngược lại.
4. Max(S). Hàm trả về phần tử dữ liệu có giá trị khoá lớn nhất trong tập S.
5. Min(S). Hàm trả về phần tử dữ liệu có giá trị khoá nhỏ nhất trong tập S.

Để viết cho ngắn gọn, chúng ta giả sử rằng các phần tử dữ liệu có kiểu là Item và có thể truy cập trực tiếp tới thành phần khoá của Item, chẳng hạn trong các áp dụng thông thường Item là một cấu trúc:

```
struct   Item
{
  keyType   key; // khoá của dữ liệu.
  // các thành phần khác.
};
```

4.4.1 Cài đặt bởi danh sách không được sắp. Tìm kiếm tuần tự

Chúng ta có thể sắp xếp các phần tử của tập động (theo một thứ tự tuỳ ý) thành một danh sách, và do đó dễ dàng thấy rằng, ta có thể sử dụng cách

cài đặt danh sách bởi mảng động để cài đặt KDLTT tập động. Lớp tập động Dset (Dynamic Set) được thiết kế bằng cách sử dụng lớp Dlist (xem mục 4.3) làm lớp cơ sở với dạng thừa kế private. Với cách thiết kế này , các hàm thành phần của lớp Dlist trở thành các hàm thành phần private của lớp DSet và chúng ta sẽ sử dụng chúng để cài đặt các phép toán tập động. Lớp Dset chỉ chứa các thành phần dữ liệu được thừa kế từ lớp Dlist. Định nghĩa lớp Dset được cho trong hình 4.5.

```
template  <class  Item>
class   Dset : private   Dlist<Item>
{
  public :
     DSet(int m = 1)
     // khởi tạo ra tập rỗng, có thể chứa được  nhiều nhất là m phần tử
     // dữ liệu.
     : Dlist(m) {  }
     void    DSetInsert(const Item & x);
     // Postcondition: phần tử x trở thành thành viên của tập động.
     void     DSetDelete(keyType  k);
     // Postcondition: phần tử với khoá k không có trong tập động.
     bool    Search(keyType k);
     // Postcondition: Trả về true nếu phần tử với khoá k có trong
     // tập động và trả về false nếu không.
     Item &   Max(  );
     // Precondition: Danh sách không rỗng.
     // Postcondition: Trả về phần tử có khoá lớn nhất trong tập động.
     Item &   Min(  );
     // Precondition: Danh sách không rỗng.
     // Postcondition: Trả về phần tử có khoá nhỏ nhất trong tập động.
};
```

Hình 4.5. Định nghĩa lớp DSet.

Sau đây chúng ta cài đặt các phép toán trên tập động bằng cách sử dụng các hàm được thừa kế từ lớp danh sách.

Hàm DSetInsert. Hàm này được thực hiện bằng cách thêm phần tử mới vào đuôi danh sách:

```
template   <class  Item>
void    DSet<Item> ::  DSetInsert(const Item & x)
{
  if (! Search(x.key))
                Append(x);
}
```

Hàm DSetDelete. Để loại phần tử có khóa k khỏi tập động, chúng ta cần xem xét từng phần tử trong danh sách, khi gặp phần tử cần loại ở vị trí thứ i trong danh sách thì sử dụng hàm Delete(i) thừa kế từ lớp Dlist.

```
template <class  Item>
void   DSet<Item> : DSetDelete(keyType  k)
{
  for(int i = 1; i <= Length(  ); i + +)
         if (Element(i).key = = k)
                    {
                      Delete(i);
                       break;
                     }
}
```

Hàm Search. Cần lưu ý rằng, các phần tử của tập động đã được lưu dưới dạng một danh sách. Do đó để tìm một phần tử với khoá cho trước có

trong tập động hay không, chúng ta xem xét lần lượt từng phần tử của danh sách bắt đầu từ phần tử đầu tiên, cho tới khi gặp phần tử cần tìm hoặc đi đến hết danh sách mà không thấy. Kỹ thuật tìm kiếm này được gọi là **tìm kiếm tuần tự (sequential search)**, đó là một dạng của tìm kiếm vét cạn.

```
template <class  Item>
bool   DSet<Item> :: Search(keyType   k)
{
  for(int i = 1; i <= Length(  ); i + +)
          if (Element(i).key = = k)
                          return   true;
  return     false;
}
```

Phép toán Max, Min cũng được cài đặt rất đơn giản. Chúng ta chỉ cần đi qua danh sách và lưu lại phần tử có khoá lớn nhất (nhỏ nhất).

Thời gian thực hiện các phép toán tập động trong cách cài đặt này. Để thực hiện các phép toán DSetInsert, DSetDelete, Search, Max, Min chúng ta đều cần phải đi qua từng phần tử của danh sách, kể từ phần tử đầu tiên (vòng lặp for). Khi tìm kiếm, trong trường hợp xấu nhất (chẳng hạn phần tử cần tìm không có trong danh sách), cần phải đi hết danh sách, do đó số lần lặp tối đa là n (n là độ dài danh sách). Vì danh sách được lưu trong mảng, nên phép toán Element(i) chỉ tốn thời gian O(1). Do đó thời gian thực hiện tìm kiếm là O(n), trong đó n là độ dài danh sách. Phân tích tương tự ta thấy rằng, thời gian của các phép toán khác cũng là O(n).

4.4.2 Cài đặt bởi danh sách được sắp. Tìm kiếm nhị phân

Bây giờ chúng ta xét một cách cài đặt tập động khác, trong cách này, tập động cũng được biểu diễn dưới dạng danh sách, nhưng các phần tử của danh sách được sắp xếp theo thứ tự tăng của các giá trị khoá. Phần tử có

khoá nhỏ nhất của tập động đứng ở vị trí đầu tiên trong danh sách, còn phần tử có khoá lớn nhất của tập động đứng sau cùng trong danh sách. Do đó, các phép toán Max, Min được cài đặt rất đơn giản và chỉ cần thời gian O(1). Chẳng hạn, phép toán Min được xác định như sau:

```
template <class  Item>
Item &   DSet<Item> :: Min(  )
{
    assert( ! Empty(  ));
    return    Element(1);
}
```

Bây giờ chúng ta viết hàm DSetInsert. Vì tập động được biểu diễn dưới dạng danh sách được sắp, nên khi xen một phần tử mới vào tập động, chúng ta cần phải đặt nó vào vị trí thích hợp trong danh sách, để đảm bảo sau khi xen, danh sách vẫn còn là danh sách được sắp. DSetInsert được cài đặt như sau:

```
template <class  Item>
void   DSet<Item> :: DSetInsert(const Item & x)
{
  int   i;
  for(i = 1; i < = Length(  ); i + +)
     if (x.key < Element(i).key)
        { Insert(x, i); break; }
     else if (x.key = = Element(i).key)
              break;
     if (i > Length(  ))     // danh sách rỗng hoặc x có khóa lớn hơn
            Append(x);    // khoá của mọi phần tử trong danh sách.
}
```

Phép toán DSetDelete được cài đặt giống như trong trường hợp danh sách không được sắp, nhưng cần lưu ý rằng, khi gặp một phần tử trong danh sách có khoá lớn hơn khoá k có nghĩa là trong danh sách không chứa phần tử cần loại và do đó ta có thể dừng lại mà không cần đi hết danh sách.

```
template <class Item>
void DSet<Item> :: DSetDelete(keyType k)
{
  for (int i = 1; i <= Length(  ); i + +)
    if (Element(i).key = = k)
            {
              Delete(i);
              break;
            }
    else if (Element(i).key > k)
              break;
}
```

Ưu điểm lớn nhất của phương pháp cài đặt tập động bởi danh sách được sắp là chúng ta có thể sử dụng kỹ thuật tìm kiếm khác hiệu quả hơn tìm kiếm tuần tự.

Tìm kiếm nhị phân (Binary Search)

Tư tưởng của kỹ thuật tìm kiếm nhị phân là như sau: xét phần tử đứng giữa danh sách, nó chia danh sách thành hai phần: nửa đầu và nửa sau. (Chú ý rằng, vì danh sách được lưu trong mảng, nên ta có thể tìm thấy phần tử ở giữa danh sách chỉ với thời gian O(1)). Do danh sách được sắp xếp theo thứ tự tăng của khoá, nên tất cả các phần tử ở nửa đầu danh sách đều có khoá nhỏ hơn khoá của phần tử đứng giữa danh sách và khoá của phần tử này nhỏ hơn khoá của tất cả các phần tử ở nửa sau danh sách. Nếu khoá k

của phần tử cần tìm bằng khoá của phần tử đứng giữa danh sách có nghĩa là ta đã tìm thấy; còn nếu k khác khoá của phần tử đứng giữa, ta tiếp tục tìm kiếm ở nửa đầu (nửa sau) danh sách tuỳ thuộc khoá k nhỏ hơn (lớn hơn) khóa của phần tử đứng giữa danh sách. Quá trình tìm kiếm ở nửa đầu (hoặc nửa sau) được thực hiện bằng cách lặp lại thủ tục trên. Chúng ta có thể cài đặt thuật toán tìm kiếm nhị phân bởi hàm đệ quy hoặc không đệ quy. Trong hình 4.6 là hàm Search không đệ quy.

```
template <class  Item>
bool   DSet<Item> :: Search(keyType k)
{
   int   bottom, top, mid;
   bottom = 1;
   top = Length(  );
   while (bottom <= top)
   {
      mid = (bottom + top) / 2;
      if (k = = Element(mid).key)
                    return    true;
      else if (k  < Element(mid).key)
                   top = mid – 1;
           else    bottom = mid + 1;
   }
   return    false;
}
```

Hình 4.6.Tìm kiếm nhị phân.

Thời gian tìm kiếm nhị phân. Bây giờ chúng ta đánh giá thời gian chạy của thuật toán tìm kiếm nhị phân theo độ dài n của danh sách. Dễ dàng thấy rằng, thời gian thực hiện thân của vòng lặp while là O(1), bởi vì việc

truy cập tới phần tử đứng giữa danh sách, Element(mid), chỉ cần thời gian O(1). Do đó chúng ta chỉ cần đánh giá số lần lặp. Mỗi lần lặp là một lần chia danh sách đang xét thành hai phần: nửa đầu và nửa sau, và danh sách được xét tới ở lần lặp sau là một trong hai phần đó. Số lần lặp tối đa là số tối đa lần chia danh sách ban đầu cho tới khi nhận được danh sách cần xem xét chỉ gồm một phần tử (bottom = top). Trường hợp xấu nhất này sẽ xảy ra khi mà phần tử cần tìm là phần tử đầu tiên (phần tử cuối cùng) của danh sách, hoặc không có trong danh sách. Nếu n là chẵn, thì khi chia đôi ta nhận được danh sách ở bên trái phần tử đứng giữa có độ dài n/2 –1, còn danh sách con bên phải có độ dài n/2. Xét trường hợp $n = 2^r$. Nếu phần tử cần tìm ở cuối cùng trong danh sách hoặc khoá k lớn hơn khoá của tất cả các phần tử trong danh sách, thì số lần chia là r. Đó là trường hợp xấu nhất. Vì vậy, trong trường hợp $n = 2^r$, thì số lần chia nhiều nhất là r và $r = \log_2 n$.

Còn nếu $n \neq 2^r$ thì sao? Chúng ta có thể tìm được số nguyên r nhỏ nhất sao cho:

$2^{r-1} < n < 2^r$

$r - 1 < \log_2 n < r$

$r < 1 + \log_2 n < r + 1$

Do đó, trong trường hợp $n \neq 2^r$, thì số lần chia tối đa để dẫn đến danh sách gồm một phần tử không vượt quá $r < 1 + \log_2 n$.

Như vậy, thời gian của thuật toán tìm kiếm nhị phân là O(logn).

So sánh hai phương pháp cài đặt.

Nếu chúng ta cài đặt tập động bởi danh sách không được sắp (các phần tử của tập động được xếp thành danh sách theo trật tự tuỳ ý), thì thời gian thực hiện các phép toán trên tập động là O(n). (Cần lưu ý rằng, để xen phần tử mới vào tập động, chúng ta chỉ cần thêm nó vào đuôi danh sách, nhưng chúng ta cần phải kiểm tra nó đã có trong tập động chưa, tức là vẫn cần duyệt danh sách, do đó phép toán DSetInsert cũng đòi hỏi thời gian O(n)). Trong khi đó, nếu tập động được biểu diễn bởi danh sách được sắp

(các phần tử của tập động được sắp xếp thành danh sách theo thứ tự tăng (hoặc giảm) của các giá trị khoá), thì các phép toán DSetInsert, DSetDelete vẫn cần thời gian O(n), phép toán Min, Max chỉ cần thời gian O(1); đặc biệt phép toán Search, do áp dụng kỹ thuật tìm kiếm nhị phân, nên chỉ đòi hỏi thời gian O(logn).

Trên đây chúng ta đã trình bày một cách thiết kế lớp tập động DSet sử dụng lớp Dlist như lớp cơ sở private. Chúng ta có thể đưa ra một cách thiết kế lớp DSet khác, thay cho sử dụng lớp cơ sở Dlist, lớp DSet sẽ chứa một thành phần dữ liệu là đối tượng của lớp Dlist. Lớp DSet này sẽ có dạng như sau:

```
template   <class  Item>
class    DSet1
{
  public :
    // Các hàm thành phần như trong lớp DSet
  private :
       Dlist    L;
};
```

Chúng ta còn có thể thiết kế lớp DSet một cách khác không cần sử dụng đến lớp Dlist, mà trực tiếp sử dụng ba thành phần dữ liệu: biến con trỏ element trỏ tới mảng được cấp phát động để lưu các phần tử của danh sách biểu diễn tập động, biến size lưu cỡ của mảng động và biến last lưu chỉ số của mảng chứa phần tử cuối cùng của danh sách. Trong cách thiết kế này, lớp DSet có dạng sau:

```
template <class   Item>
class    DSet2
{
  public :
```

```
    // Các hàm kiến tạo mặc định, copy
    // Hàm huỷ
    // Toán tử gán, …
    // Các hàm cho các phép toán tập động.
  private :
    Item*   element;
    int   size;
    int   last;
};
```

Độc giả có thể cài đặt dễ dàng các lớp DSet1 và DSet2 (bài tập).

4.5 ỨNG DỤNG

Danh sách là một cấu trúc dữ liệu tuyến tính được sử dụng nhiều trong thiết kế các thuật toán. Nhiều loại đối tượng dữ liệu có thể biểu diễn dưới dạng danh sách, và do đó chúng ta có thể sử dụng danh sách để cài đặt nhiều KDLTT khác. Chẳng hạn, trong mục 4.4, chúng ta đã sử dụng danh sách để cài đặt KDLTT tập động. Trong mục này chúng ta sẽ xét một vài ứng dụng khác: sử dụng danh sách để cài đặt đa thức và các phép toán đa thức, và sử dụng danh sách để cài đặt các ma trận thưa (ma trận chỉ chứa một số ít các thành phần khác không).

Đa thức và các phép toán đa thức.

Một đa thức là một biểu thức có dạng:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

Mỗi hạng thức $a_k x^k$, $a_k \neq 0$ có thể biểu diễn bởi một cặp hai thành phần: hệ số (coef) a_k và số mũ (expo) k. Và do đó, chúng ta có thể biểu diễn đa thức như là một danh sách các cặp hệ số, số mũ. Chẳng hạn, đa thức:

$$17x^5 - 25x^2 + 14x - 32$$

được biểu diễn dưới dạng danh sách sau:

((17, 5), (-25, 2), (14, 1), (-32, 0))

Các đa thức cùng với các phép toán quen biết trên đa thức tạo thành KDLTT đa thức. Chúng ta sẽ cài đặt lớp đa thức (class Poly), bằng cách biểu diễn đa thức dưới dạng danh sách các hạng thức như đã trình bày ở trên. Lớp đa thức chứa một thành phần dữ liệu là danh sách các hạng thức. Do đó, trước khi định nghĩa lớp đa thức, chúng ta cần xác định lớp hạng thức (class Term).

Định nghĩa lớp đa thức được cho trong hình 4.7. Để cho ngắn gọn, trong đó chúng ta chỉ xác định một hàm toán tử thực hiện phép cộng đa thức, các hàm thành phần cần thiết khác độc giả tự xác định lấy.

```
class   Poly; // khai báo trước lớp đa thức.
class   Term  // Lớp hạng thức.
{
  public :
    friend  class  Poly;
    Term(double  a = 0, int  k = 0);
    // Kiến tạo nên hạng thức có hệ số a, số mũ k.
  private:
        double   coef;
        int   expo;
};
class Poly  // Lớp đa thức.
{
  public :
            ……
    Poly& operator + (const Poly & p);
    // Toán tử cộng hai đa thức.
            ……
  private :
```

```
    Dlist<Term>    TermList; // Danh sách các hạng thức.
};
```

Hình 4.7.Lớp đa thức.

Bây giờ chúng ta cài đặt một hàm thành phần: hàm cộng đa thức. Thuật toán cộng hai đa thức như sau: sử dụng hai phép lặp, mỗi phép lặp chạy trên một danh sách các hạng thức của một đa thức. Chúng ta so sánh hai hạng thức hiện thời, nếu chúng có cùng số mũ thì tạo ra một hạng thức mới với số mũ là số mũ đó, còn hệ số là tổng các hệ số của hai hạng thức hiện thời (nếu tổng này khác không), và đưa hạng thức mới vào đuôi danh sách hạng thức của đa thức kết quả, đồng thời dịch chuyển hai vị trí hiện thời đến các vị trí tiếp theo trong hai danh sách. Nếu hạng thức hiện thời của một trong hai danh sách có số mũ lớn hơn số mũ của hạng thức hiện thời kia, thì ta đưa hạng hiện thời có số mũ lớn hơn vào đuôi danh sách của đa thức kết quả. Khi mà một vị trí hiện thời đã chạy hết danh sách, chúng ta tiếp tục cho vị trí hiện thời kia chạy trên danh sách còn lại, để ghi các hạng thức còn lại của danh sách này vào đuôi danh sách của đa thức kết quả. Hàm toán tử cộng đa thức được cho trong hình 4.8.

```
Poly &  Poly :: operator + (const Poly & p)
{
   Term   term1, term2, term3;
   Poly result;
   DListIterator<Term>  itr1(TermList);
   DlistIterator<Term>   itr2(p.TermList);
   itr1.Start(  );
   itr2.Start(  );
   while (itr1.Valid(  ) && itr2.Valid(  ))
   {
     term1 = itr1.Current( );
```

```
  term2 = itr2.Current( 0;
  if (term1. expo = = term2. expo)
  {
     if (term1.coef + term2.coef ! = 0)
     {
         term3.coef = term1.coef + term2.coef;
         term3.expo = term1.expo;
         result.TermList.Append(term3);
         // Thêm hạng thức term3 vào đuôi danh sách các hạng thức của
         // đa thức kết quả.
     }
     itr1.Advance(  );
     itr2.Advance(  );
  }
  else if (term1.expo > term2.expo)
     {
         result.TermList.Append(term1);
         itr1.Advance(  );
     }
  else {
         result.TermList.Append(term2);
         itr2.Advance(  );
         }
}; // Hết while
while (itr1.Valid( ))
{
   term1 = itr1.Current( );
   result.TermList.Append(term1);
   itr1.Advance( );
}
 while (itr2.Valid( ))
```

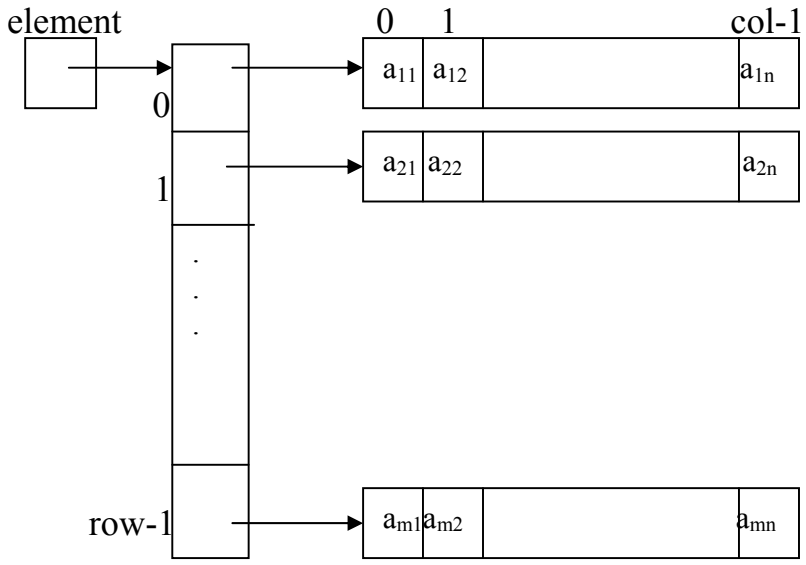
```
    {
      term2 = itr2.Current( );
      result.TermList.Append(term2);
      itr2.Advance( );
    }

    return    result;

}
```

Hình 4.8.Hàm cộng đa thức.

Ma trân thưa. Một ma trận là một bảng hình chữ nhật chứa m x n phần tử được sắp xếp thành m dòng, n cột:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

trong đó a_{ij} là phần tử đứng ở dòng thứ i, cột thứ j. Tập hợp các ma trận cùng với các phép toán quen biết trên ma trận lập thành KDLTT ma trận. Để cài đặt KDLTT ma trận, trước hết chúng ta cần biểu diễn ma trận bởi cấu trúc dữ liệu thích hợp. Cách tự nhiên nhất là cài đặt ma trận bởi mảng hai chiều. Nhưng để có thể biểu diễn được ma trận có cỡ m x n bất kỳ, chúng ta sử dụng mảng được cấp phát động. Do đó, để biểu diễn ma trận chúng ta sử dụng ba biến: biến row lưu số dòng và biến col lưu số cột của ma trận, và con trỏ element trỏ tới mảng động cỡ row, mảng này chứa các con trỏ trỏ tới các mảng động cỡ col để lưu các phần tử của ma trận, như trong hình sau:



Với cách biểu diễn ma trận như trên, lớp ma trận sẽ có nội dung như sau:

```
template <class  Item>
class    Matrix
{
private :
   int   row, col;
   Item**   element;
public :
   Matrix(  )
   { row = 0; col = 0; element = NULL; }
   Matrix (int m, int n);
   // Kiến tạo ma trận không m dòng, n cột.
   Matrix (const Matrix & M); // Kiến tạo copy.
   ~ Matrix(  ); // Hàm huỷ.
   Matrix &  operator = (const Matrix & M); // Toán tử gán.
   int   Row(  )  const
   { return   row; }
   int   Col(  )  const
```

```
    { return   col; }
    friend Matrix<Item> & operator + (const Matrix<Item> & M1
                                      const Matrix<Item> & M2);
    // Các hàm toán tử cho các phép toán khác trên ma trận.
};
```

Bây giờ chúng ta xét các ma trận thưa: ma trận chỉ chứa một số ít các phần tử khác không. Chẳng hạn ma trận sau:

$$M = \begin{bmatrix} 0 & 7 & 0 & 0 & 3 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 9 & 0 \end{bmatrix}$$

Với các ma trận thưa, nếu biểu diễn ma trận bởi mảng sẽ rất lãng phí bộ nhớ. Chẳng hạn, với ma trận cỡ 1000 x 2000 chúng ta cần cấp phát bộ nhớ để lưu 2 triệu phần tử. Song nếu ma trận đó chỉ chứa 2000 phần tử khác không thì chúng ta mong muốn chỉ cần cấp phát bộ nhớ để lưu 2000 phần tử đó. Cách tiếp cận biểu diễn ma trận bởi danh sách cho phép ta làm được điều đó.

Mỗi phần tử của ma trận được mô tả bởi một cặp: chỉ số cột của phần tử đó và giá trị của nó. Mỗi dòng của ma trận sẽ được biểu diễn bởi danh sách các cặp (chỉ số, giá trị). Chẳng hạn, dòng thứ nhất của ma trận M ở trên được biểu diễn bởi danh sách:

((2,7), (5,3))

Một ma trận có thể xem như danh sách các dòng. Và do đó, chúng ta có thể biểu diễn ma trận bởi danh sách của các danh sách các cặp (chỉ số, giá trị). Chẳng hạn, ma trận M ở trên được biểu diễn bởi danh sách sau:

(((2,7), (5,3)), ((3,8)), ((2,5), (4,9)))

Với cách biểu diễn ma trận bởi danh sách, để xây dựng lớp các ma trận thưa (SpMatrix), trước hết chúng ta cần xác định lớp các phần tử của ma trận (Elem).

```
template <class  Item>
class    MaRow; // Lớp dòng của ma trận.
template <class  Item>
class    SpMatrix; // Lớp ma trận thưa.
template <class  Item>
class    Elem // Lớp phần tử của ma trận.
{
   friend   class MaRow<Item>;
   friend   class SpMatrix<Item>;
public :
    Elem (int j, Item a)
    // Kiến tạo một phần tử ở cột j, có giá trị a.
    { colIndex = j; value = a; }
private :
   int    colIndex; // Chỉ số cột.
   Item    value; // giá trị.
};
```

Lớp dòng của ma trận (MaRow) chứa hai thành phần dữ liệu: chỉ số dòng của ma trận (rowIndex), và một thành phần dữ liệu khác là danh sách các phần tử trong dòng đó, danh sách này được ký hiệu là eleList và được cài đặt bởi danh sách động Dlist (xem mục 4.3). Trong lớp MaRow, chúng ta cần đưa vào các hàm phục vụ cho việc thực hiện các phép toán trên ma trận: đó là các hàm toán tử + (cộng hai dòng), - (trừ hai dòng), * (nhân một dòng với một giá trị), * (nhân một dòng với một ma trận), … Lớp MaRow được xác định như sau:

```
template <class  Item>
class     MaRow
{
   friend   class   SpMatrix;
```

```
    public :
        MaRow(  ); // khởi tạo dòng rỗng.
        MaRow & operator + (const MaRow & R);
        MaRow & operator + = (const MaRow & R);
        MaRow & operator – (const MaRow & R);
        MaRow & operator - = (const MaRow & R);
        MaRow & operator * (const Item & a);
        // nhân một dòng với giá trị a.
        MaRow & operator * (const SpMatrix<Item> & M);
        // nhân một dòng với ma trận M.
    private :
        Dlist <Elem<Item>>   eleList; // Danh sách các phần tử.
        int  rowIndex; // chỉ số dòng.
};
```

Tại sao trong lớp MaRow chúng ta cần đưa vào các hàm toán tử trên? Để cộng (trừ) hai ma trận, chúng ta cần cộng (trừ) các dòng tương ứng của hai ma trận, vì vậy trong lớp MaRow chúng ta đã đưa vào các toán tử +, + =, -, - =. Chú ý rằng, mỗi dòng của ma trận được biểu diễn bởi danh sách các phần tử được sắp xếp theo thứ tự tăng dần theo chỉ số cột, vì vậy phép cộng (trừ) các dòng được thực hiện theo thuật toán hoàn toàn giống như khi ta thực hiện cộng hai đa thức. Phép toán nhân dòng ma trận với một giá trị được đưa vào để cài đặt phép nhân ma trận với một giá trị. Bây giờ chúng ta thảo luận về phép nhân một dòng với một ma trận. Tại sao lại cần phép toán này? Giả sử cần tính tích của hai ma trận A và B: A * B = C. Phần tử c_{ij} của ma trận C được tính theo công thức:

$c_{ij} = \sum_k a_{ik} b_{kj}$, trong đó A = (a_{ik}), B = (b_{kj}).

Tính theo công thức này sẽ kém hiệu quả, vì chúng ta phải duyệt dòng thứ i của ma trận A, và với mỗi phần tử ở cột k, chúng ta cần tìm đến dòng

thứ k của ma trận B để tìm phần tử ở cột j. Song phân tích công thức trên, chúng ta sẽ thấy rằng, dòng thứ i của ma trận C là dòng thứ i của ma trận A nhân với ma trận B. Trong đó phép nhân một dòng với ma trận B được định nghĩa như sau: Lấy phần tử ở cột k của dòng nhân với dòng thứ k của ma trận B, rồi cộng tất cả các dòng thu được ta nhận được dòng kết quả. Chẳng hạn:

$$[5, 0, 1, 3]* \begin{bmatrix} 0 & 3 & 2 \\ 1 & 0 & 4 \\ 0 & 6 & 0 \\ 2 & 0 & 0 \end{bmatrix} = 5* [0\ \ 3\ \ 2] + 0* [1\ \ 0\ \ 4] + 1* [0\ \ 6\ \ 0] + 3* [2\ \ 0\ \ 0]$$

$$= [6\ \ 21\ \ 10].$$

Đến đây, chúng ta có thể định nghĩa lớp ma trận thưa SpMatrix. Lớp này chứa ba thành phần dữ liệu: số dòng (row), số cột (col) và danh sách các dòng của ma trận (rowList)

```
template <class  Item>
class    SpMatrix
{
public:
     // Hàm kiến tạo và các hàm cần thiết khác.
     // Các hàm toán tử cho các phép toán ma trận.
private:
     int   row; // Số dòng.
     int   col; // Số cột.
     Dlist<MaRow<Elem<Item>>>    rowList;
     // Danh sách các dòng của ma trận.
};
```

Cài đặt chi tiết các lớp MaRow, SpMatrix để lại cho độc giả xem như bài tập.

BÀI TẬP

1. Trong lớp Dlist, hãy cài đặt hàm truy cập tới phần tử thứ i trong danh sách (hàm Element(i)) bởi hàm toán tử Operator [ ].

2. Hãy cài đặt hàm Advance( ) trong lớp DlistIterator bởi hàm toán tử Operator + +.

3. Giả sử trong lớp Dset, tập động được cài đặt bởi danh sách được sắp theo giá trị khoá tăng dần. Hãy cài đặt hàm Search(k) bởi hàm đệ quy theo kỹ thuật tìm kiếm nhị phân.

4. Viết chương trình sử dụng danh sách các số nguyên L (L là đối tượng của lớp Dlist). Chương trình cần thực hiện các nhiệm vụ sau:
    a. Tạo ra danh sách các số nguyên được đưa vào từ bàn phím.
    b. Loại khỏi danh sách tất cả các số nguyên bằng một số nguyên cho trước.
    c. Thêm vào danh sách số nguyên n sau số nguyên m xuất hiện đầu tiên trong danh sách, nếu m không có trong danh sách thì thêm m vào đuôi danh sách.
    d. In ra các số nguyên trong danh sách.

5. Cho hai danh sách số nguyên L_1 và L_2 là đối tượng của lớp Dlist. Hãy viết hàm toán tử Operator + thực hiện nối hai danh sách đó thành một danh sách, cần giữ nguyên thứ tự của các phần tử và các phần tử của L_2 đi sau các phần tử của L_1.

CHƯƠNG 5

DANH SÁCH LIÊN KẾT

KDLTT danh sách đã được xác định trong chương 4 với các phép toán xen vào, loại bỏ và tìm kiếm một đối tượng khi cho biết vị trí của nó trong danh sách và một loạt các phép toán khác cần thiết cho các xử lý đa dạng khác trên danh sách. Trong chương 4 chúng ta đã cài đặt danh sách bởi mảng. Hạn chế cơ bản của cách cài đặt này là mảng có cỡ cố định, mà danh sách thì luôn phát triển khi ta thực hiện phép toán xen vào, và do đó trong quá trình xử lý danh sách, nó có thể có độ dài vượt quá cỡ của mảng. Một cách lựa chọn khác là cài đặt danh sách bởi cấu trúc dữ liệu **danh sách liên kết** (DSLK). Các thành phần dữ liệu trong CTDL này được liên kết với nhau bởi các con trỏ; mỗi thành phần chứa con trỏ trỏ tới thành phần tiếp theo. DSLK có thể "nối dài ra" khi cần thiết, trong khi mảng chỉ lưu được một số cố định đối tượng dữ liệu.

Nội dung chính của chương này là như sau: Mục 5.1 trình bày những kiến thức cần thiết về con trỏ và cấp phát động bộ nhớ trong C + + được sử dụng thường xuyên sau này. Mục 5.2 sẽ nói về DSLK đơn và các dạng DSLK khác: DSLK vòng tròn, DSLK kép. Trong mục 5.3, chúng ta sẽ cài đặt KDLTT danh sách bởi DSLK. Sau đó, trong mục 5.4, chúng ta sẽ phân tích so sánh hai phương pháp cài đặt KDLTT danh sách: cài đặt bởi mảng và cài đặt bởi DSLK. Cuối cùng, trong mục 5.5, chúng ta sẽ thảo luận về sự cài đặt KDLTT tập động bởi DSLK.

5.1 CON TRỎ VÀ CẤP PHÁT ĐỘNG BỘ NHỚ

Cũng như nhiều ngôn ngữ lập trình khác, C + + có các con trỏ, nó cho phép ta xây dựng nên các DSLK và các CTDL phức tạp khác. **Biến con trỏ** (pointer variable), hay gọi tắt là **con trỏ** (pointer), là biến chứa **địa chỉ** của một tế bào nhớ trong bộ nhớ của máy tính. Nhờ có con trỏ, ta định vị được tế bào nhớ và do đó có thể truy cập được nội dung của nó.

Để khai báo một biến con trỏ P lưu giữ địa chỉ của tế bào nhớ chứa dữ liệu kiểu Item, chúng ta viết

```
Item *  P;
```

Chẳng hạn, khai báo

```
int *   P ;
```

nói rằng P là biến con trỏ nguyên, tức là P chỉ trỏ tới các tế bào nhớ chứa số nguyên.

Muốn khai báo P và Q là hai con trỏ nguyên, ta cần viết

```
int *   P;
```

```
int *   Q;
```

hoặc

```
int * P , * Q;
```

Cần nhớ rằng, nếu viết

```
int * P, Q;
```

thì chỉ có P là biến con trỏ nguyên, còn Q là biến nguyên.

Nếu chúng ta khai báo:

```
int * P;
int    x;
```

thì các biến này được cấp phát bộ nhớ trong thời gian dịch, sự cấp phát bộ nhớ như thế được gọi là **cấp phát tĩnh**, nó xảy ra trước khi chương trình được thực hiện. Nội dung của các tế bào nhớ đã cấp phát cho P và x chưa được xác định, xem minh hoạ trong hình 5.1a.

Bạn có thể đặt địa chỉ của x vào P bằng cách sử dụng toán tử lấy địa chỉ & như sau:

```
P = &x;
```

Khi đó chúng ta có hoàn cảnh được minh hoạ trong hình 5.16, ký hiệu *P biểu diễn tế bào nhớ mà con trỏ P trỏ tới

Đến đây, nếu bạn viết tiếp:

```
*P = 5;
```

thì biến x sẽ có giá trị 5, tức là hiệu quả của lệnh gán *P = 5 tương đương với lệnh gán x = 5, như được minh hoạ trong hình 5.1c.

Sự cấp phát bộ nhớ còn có thể xảy ra trong thời gian thực hiện chương trình, và được gọi là **cấp phát động** (dynamic allocation). Toán tử **new** trong C + + cho phép ta thực hiện sự cấp phát động bộ nhớ.

Bây giờ, nếu bạn viết

```
P = new  int;
```

thì một tế bào nhớ lưu giữ số nguyên được cấp phát cho biến động *P và con trỏ P trỏ tới tế bào nhớ đó, như được minh hoạ trong hình 5.1d. Cần lưu ý rằng biến động *P chỉ tồn tại khi nó đã được cấp phát bộ nhớ, và khi đó, nó được sử dụng như các biến khác.

Tiếp theo, bạn có thể viết

```
*P = 8;
```

khi đó số nguyên 8 được đặt trong tế bào nhớ mà con trỏ P trỏ tới, ta có hoàn cảnh như trong hình 5.1e.

Giả sử, bạn viết tiếp:

```
int *  Q;
Q = P;
```

Khi đó, con trỏ Q sẽ trỏ tới tế bào nhớ mà con trỏ P đã trỏ tới, như được minh hoạ trong hình 5.1f.

Nếu bây giờ bạn không muốn con trỏ trỏ tới bất kỳ tế bào nhớ nào trong bộ nhớ, bạn có thể sử dụng hằng con trỏ NULL. Trong các minh hoạ, chúng ta sẽ biểu diễn hằng NULL bởi dấu chấm. Dòng lệnh:

Q = NULL

sẽ cho hiệu quả như trong hình 5.1g.

Một khi biến động *P không cần thiết nữa trong chương trình, chúng ta có thể thu hồi tế bào nhớ đã cấp phát cho nó và trả về cho hệ thống. Toán tử **delete** thực hiện công việc này.

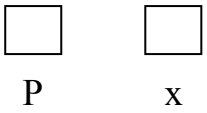
Dòng lệnh

delete P;

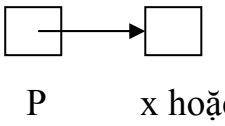
sẽ thu hồi tế bào nhớ đã cấp phát cho biến động *P bởi toán tử new, và chúng ta sẽ có hoàn cảnh được minh hoạ bởi hình 5.1h

---

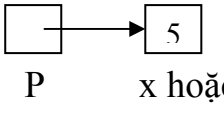
a) int * P;
int x;



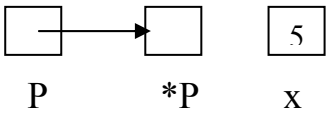
b) P = &x ;



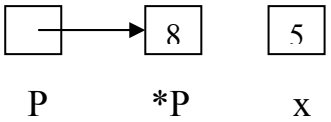
c) *P = 5;
(hoặc x = 5;)



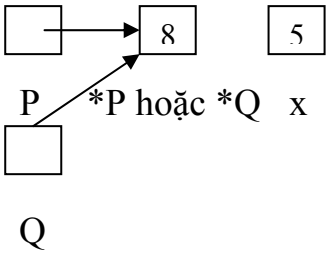
d) P = new int ;



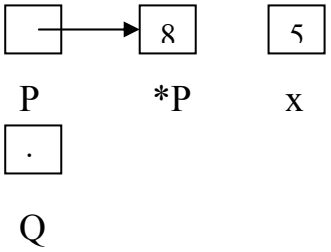
e) *P = 8 ;



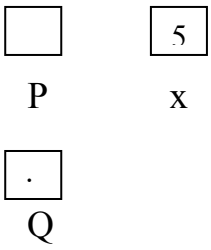
f) int * Q ;
Q = P ;



g) Q = NULL;



h) delete P ;



Hình 5.1. Các thao tác với con trỏ.

Cấp phát mảng động

Khi chúng ta khai báo một mảng, chẳng hạn

int A[30];

thì chương trình dịch sẽ cấp phát 30 tế bào nhớ liên tiếp để lưu các số nguyên, trước khi chương trình được thực hiện. Mảng A là mảng tĩnh, vì bộ nhớ được cấp phát cho nó là cố định và tồn tại trong suốt thời gian chương trình thực hiện, đồng thời A là con trỏ trỏ tới thành phần đầu tiên A[0] của mảng.

Chúng ta có thể sử dụng toán tử **new** để cấp phát cả một mảng. Sự cấp phát này xảy ra trong thời gian thực hiện chương trình, và được gọi là sự cấp phát mảng động.

Giả sử có các khai báo sau:

int size;
double * B;

Sau đó, chúng ta đưa vào lệnh:

size = 5;
B = new double[size]l

Khi đó, toán tử new sẽ cấp phát một mảng động B gồm 5 tế bào nhớ double, và con trỏ B trỏ tới thành phần đầu tiên của mảng này, như trong hình 5.2b.

Chúng ta có thể truy cập tới thành phần thứ i trong mảng động B bởi ký hiệu truyền thống B[i], hoặc *(B + i). Chẳng hạn, nếu bạn viết tiếp

B[3] = 3.14;
hoặc *(B + 3) = 3.14;

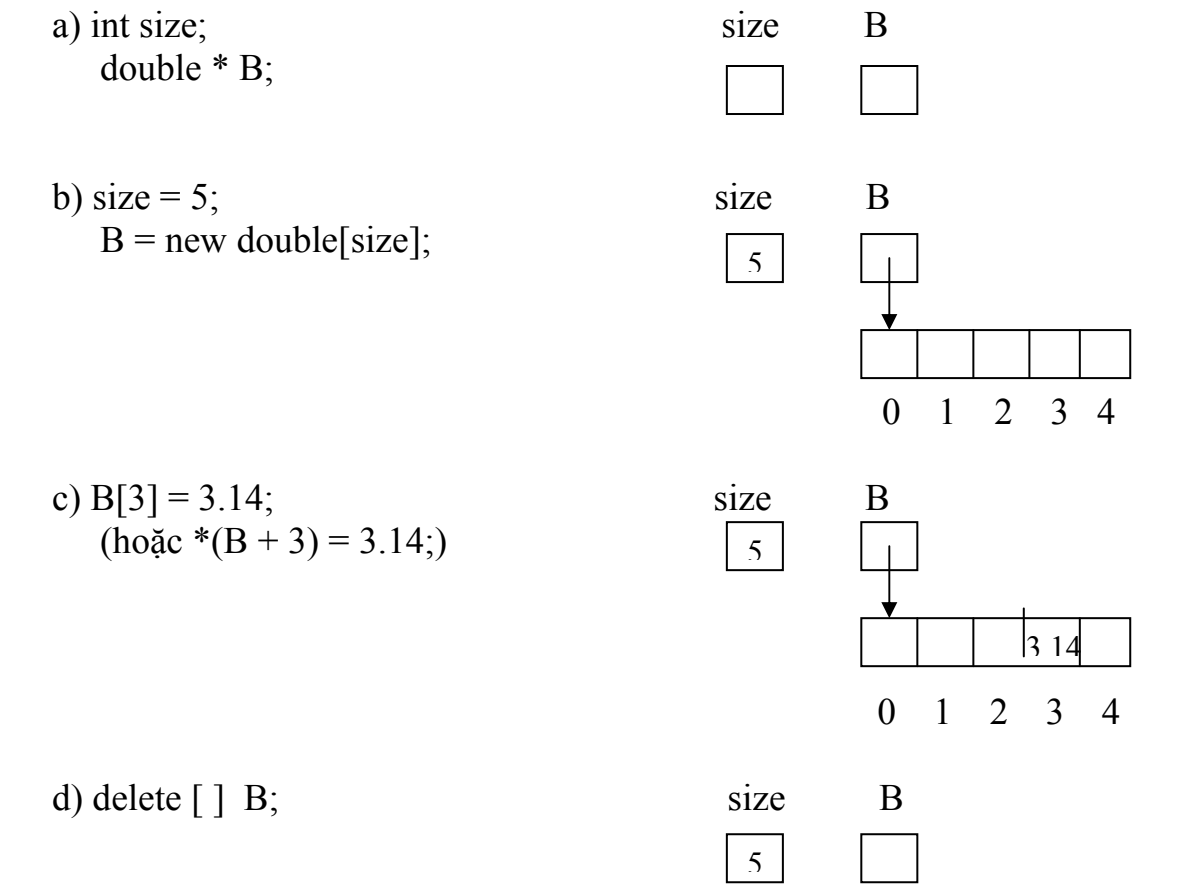
thì thành phần thứ 3 trong mảng B sẽ lưu 3.14, như được minh hoạ trong hình 5.3c. Tóm lại, các thao tác đối với mảng động là giống như đối với mảng tĩnh.

Một khi mảng động không còn được sử dụng nữa, chúng ta có thể thu hồi bộ nhớ đã cấp phát cho nó và trả về cho hệ thống để có thể sử dụng lại cho các công việc khác.

Nếu bạn viết

delete [ ] B;

thì mảng động B được thu hồi trả lại cho hệ thống và chúng ta có hoàn cảnh như trong hình 5.2d



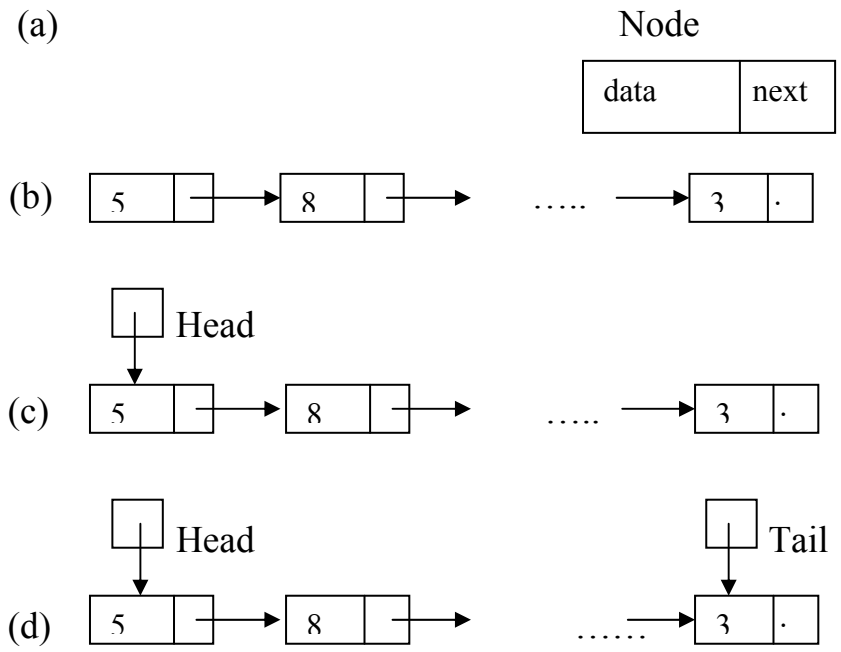
Hình 5.2. Cấp phát và thu hồi mảng động.

5.2 CẤU TRÚC DỮ LIỆU DANH SÁCH LIÊN KẾT

Trong mục này, chúng ta sẽ trình bày cấu trúc dữ liệu DSLK và các phép toán cơ bản trên DSLK.

Danh sách liên kết đơn

Danh sách liên kết đơn, gọi tắt là danh sách liên kết (DSLK) được tạo nên từ các thành phần được liên kết với nhau bởi các con trỏ. Mỗi thành phần trong DSLK chứa một dữ liệu và một con trỏ trỏ tới thành phần tiếp theo. Chúng ta sẽ mô tả mỗi thành phần của DSLK như một hộp gồm hai ngăn: một ngăn chứa dữ liệu data và một ngăn chứa con trỏ next, như trong hình 5.3a. Hình 5.3b biểu diễn một DSLK chứa dữ liệu là các số nguyên.



Hình 5.3. a) Một thành phần của DSLK.
b) DSLK chứa các số nguyên.
c) DSLK với một con trỏ ngoài Head
d) DSLK với hai con trỏ ngoài Head và Tail.

Chúng ta sẽ biểu diễn mỗi thành phần của DSLK bởi một cấu trúc trong C + +, cấu trúc này gồm hai biến thành phần: biến data có kiểu Item nào đó, và biến next là con trỏ trỏ tới cấu trúc này.

```
struct  Node
{
   Item  data ;
   Node*  next ;
} ;
```

Cần lưu ý rằng, trong thành phần cuối cùng của DSLK, giá trị của next là hằng con trỏ NULL, có nghĩa là nó không trỏ tới đâu cả và được biểu diễn bởi dấu chấm.

Để tiến hành các xử lý trên DSLK, chúng ta cần phải có khả năng truy cập tới từng thành phần của DSLK. Nếu biết được thành phần đầu tiên, "đi theo" con trỏ next, ta truy cập tới thành phần thứ hai, rồi từ thành phần thứ hai ta có thể truy cập tới thành phần thứ ba, … Do đó, khi lưu trữ một DSLK, chúng ta cần phải xác định một con trỏ trỏ tới thành phần đầu tiên trong DSLK, con trỏ này sẽ được gọi là con trỏ đầu Head. Như vậy, khi lập trình xử lý DSLK với con trỏ đầu Head, chúng ta cần đưa vào khai báo

Node* Head ;

Khi mà DSLK không chứa thành phần nào cả (ta nói DSLK rỗng), chúng ta lấy hằng con trỏ NULL làm giá trị của biến Head. Do đó, khi sử dụng DSLK với con trỏ đầu Head, để khởi tạo một DSLK rỗng, chúng ta chỉ cần đặt:

Head = NULL ;

Hình 5.3c biểu diễn DSLK với con trỏ đầu Head. Cần phân biệt rằng, con trỏ Head là con trỏ ngoài, trong khi các con trỏ next trong các thành phần là các con trỏ trong của DSLK, chúng làm nhiệm vụ "liên kết" các dữ liệu.

Trong nhiều trường hợp, để các thao tác trên DSLK được thuận lợi, ngoài con trỏ đầu Head người ta sử dụng thêm một con trỏ ngoài khác trỏ tới thành phần cuối cùng của DSLK : con trỏ đuôi Tail. Hình 5.3d biểu diễn một DSLK với hai con trỏ ngoài Head và Tail. Trong trường hợp DSLK rỗng, cả hai con trỏ Head và Tail đều có giá trị là NULL.

Sau đây chúng ta sẽ xét các phép toán cơ bản trên DSLK. Các phép toán này là cơ sở cho nhiều thuật toán trên DSLK. Chúng ta sẽ nghiên cứu các phép toán sau:

- Xen một thành phần mới vào DSLK.
- Loại một thành phần khỏi DSLK.
- Đi qua DSLK (duyệt DSLK).

1. Xen một thành phần mới vào DSLK

Giả sử chúng ta đã có một DSLK với một con trỏ ngoài Head (hình 5.3c), chúng ta cần xen một thành phần mới chứa dữ liệu là value vào sau (trước) một thành phần được trỏ tới bởi con trỏ P (ta sẽ gọi thành phần này là thành phần P).

Việc đầu tiên cần phải làm là tạo ra thành phần mới được trỏ tới bởi con trỏ Q, và đặt dữ liệu value vào thành phần này:

```
Node* Q;
Q = new Node ;
Q → data = value;
```

Các thao tác cần tiến hành để xen một thành phần mới phụ thuộc vào vị trí của thành phần P, nó ở đầu hay giữa DSLK, và chúng ta cần xen vào sau hay trước P.

Xen vào đầu DSLK. Trong trường hợp này, thành phần mới được xen vào trở thành đầu của DSLK, và do đó giá trị của con trỏ Head cần phải được thay đổi. Trước hết ta cần "móc nối" thành phần mới vào đầu DSLK, sau đó cho con trỏ Head trỏ tới thành phần mới, như được chỉ ra trong hình 5.4a. Các thao tác này được thực hiện bởi các lệnh sau:

```
Q → next = Head;
Head = Q;
```

Chúng ta có nhận xét rằng, thủ tục trên cũng đúng cho trường hợp DSLK rỗng, bởi vì khi DSLK rỗng thì giá trị của Head là NULL và do đó giá trị của con trỏ next trong thành phần đầu mới được xen vào cũng là NULL.

Xen vào sau thành phần P. Giả sử DSLK không rỗng và con trỏ P trỏ tới một thành phần bất kỳ của DSLK. Để xen thành phần mới Q vào sau thành phần P, chúng ta cần "móc nối" thành phần Q vào trong "dây chuyền" đã có sẵn, như được chỉ ra trong hình 5.4b. Trước hết ta cho con trỏ next của thành phần mới Q trỏ tới thành phần đi sau P, sau đó cho con trỏ next của thành phần P trỏ tới Q:

Q → next = P → next;

P → next = Q;

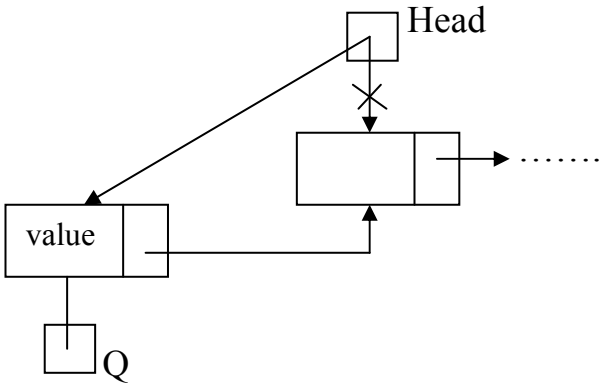
Cũng cần lưu ý rằng, thủ tục trên vẫn làm việc tốt cho trường hợp P là thành phần cuối cùng trong DSLK.

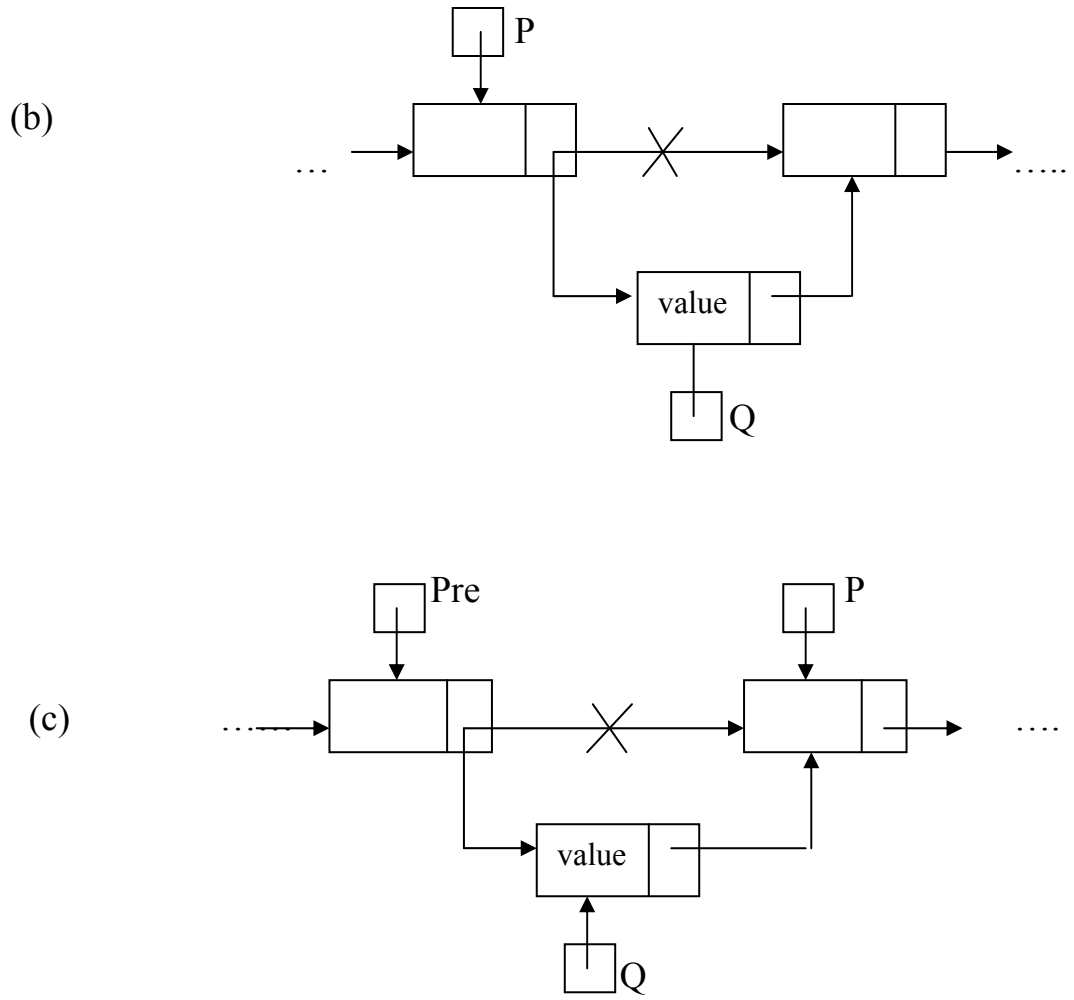
Xen vào trước thành phần P. Giả sử DSLK không rỗng, và con trỏ P trỏ tới thành phần không phải là đầu của DSLK. Trong trường hợp này, để xen thành phần mới vào trước thành phần P, chúng ta cần biết thành phần đi trước P để có thể "móc nối" thành phần mới vào trước P. Giả sử thành phần này được trỏ tới bởi con trỏ Pre. Khi đó việc xen thành phần mới vào trước thành phần P tương đương với việc xen nó vào sau thành phần Pre, xem hình 5.4c. Tức là cần thực hiện các lệnh sau:

Q → next = P; (hoặc Q → next = Pre → next)

Pre → next = Q;

(a)





Hình 5.4. a) Xen vào đầu DSLK.
b) Xen vào sau thành phần P.
c) Xen vào trước thành phần P.

2. Loại một thành phần khỏi DSLK

Chúng ta cần loại khỏi DSLK một thành phần được trỏ tới bởi con trỏ P. Cũng như phép toán xen vào, khi loại một thành phần khỏi DSLK, cần quan tâm tới nó nằm ở đâu trong DSLK. Nếu thành phần cần loại ở giữa DSLK thì giá trị của con trỏ ngoài Head không thay đổi, nhưng nếu ta loại đầu DSLK thì thành phần tiếp theo trở thành đầu của DSLK, và do đó giá trị của con trỏ Head cần thay đổi thích ứng.

Loại đầu DSLK. Đó là trường hợp P trỏ tới đầu DSLK. Để loại đầu DSLK, ta chỉ cần cho con trỏ Head trỏ tới thành phần tiếp theo (xem hình 5.5a). Với thao tác đó thành phần đầu thực sự đã bị loại khỏi DSLK, song nó

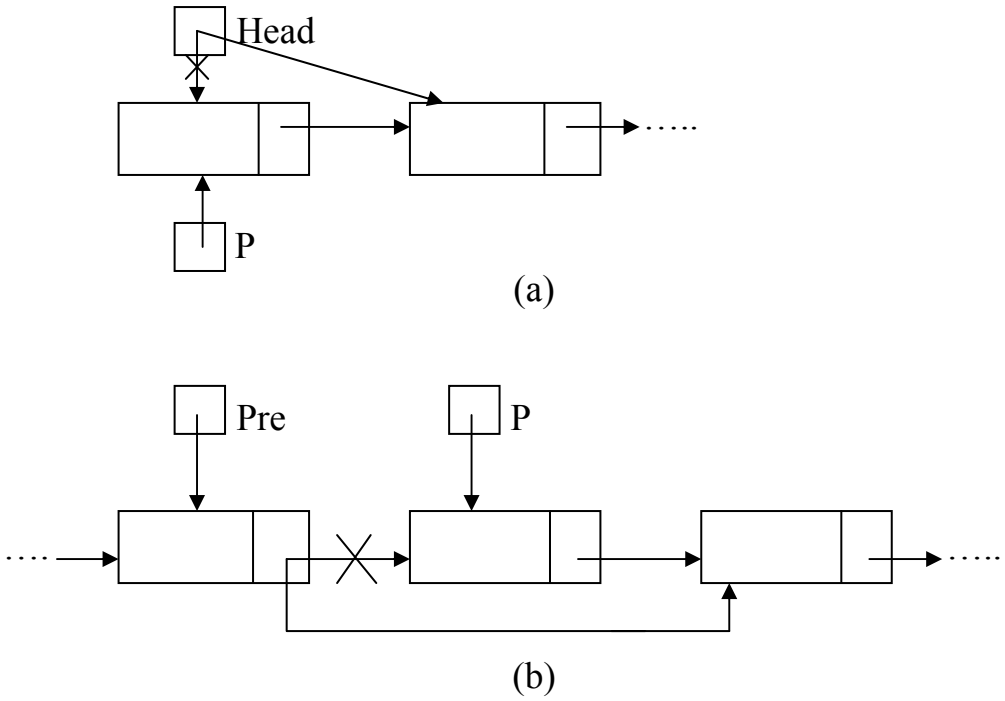
vẫn còn tồn tại trong bộ nhớ. Sẽ là lãng phí bộ nhớ, nếu để nguyên như thế, vì vậy chúng ta cần thu hồi bộ nhớ của thành phần bị loại trả về cho hệ thống. Như vậy việc loại thành phần đầu DSLK được thực hiện bởi các lệnh sau:

Head = Head → next;
delete P;

Loại thành phần không phải đầu DSLK. Trong trường hợp này để "tháo gỡ" thành phần P khỏi "dây chuyền", chúng ta cần móc nối thành phần đi trước P với thành phần đi sau P (xem hình 5.5b). Giả sử thành phần đi trước thành phần P được trỏ tới bởi con trỏ Pre. Chúng ta cần cho con trỏ next của thành phần đi trước P trỏ tới thành phần đi sau P, sau đó thu hồi bộ nhớ của thành phần bị loại. Do đó thủ tục loại thành phần P là như sau:

Pre → next = P → next;
delete P;

Thủ tục loại trên cũng đúng cho trường hợp P là thành phần cuối cùng trong DSLK.



Hình 5.5. (a) Loại đầu DSLK.
(b) Loại thành phần P

3. Đi qua DSLK (Duyệt DSLK)

Giả sử rằng, bạn đã có một DSLK, đi qua DSLK có nghĩa là truy cập tới từng thành phần của DSLK bắt đầu từ thành phần đầu tiên đến thành phần cuối cùng và tiến hành các xử lý cần thiết với mỗi thành phần của DSLK. Chúng ta thường xuyên cần đến duyệt DSLK, chẳng hạn bạn muốn biết một dữ liệu có chứa trong DSLK không, hoặc bạn muốn in ra tất cả các dữ liệu có trong DSLK.

Giải pháp cho vấn đề đặt ra là, chúng ta sử dụng một con trỏ cur trỏ tới thành phần hiện thời (thành phần đang xem xét) của DSLK. Ban đầu con trỏ cur trỏ tới thành phần đầu tiên của DSLK, cur = Head, sau đó ta cho nó lần lượt trỏ tới các thành phần của DSLK, bằng cách gán cur = cur → next, cho tới khi cur chạy qua toàn bộ DSLK, tức là cur = = NULL.

Lược đồ duyệt DSLK được mô tả bởi vòng lặp sau:

Node* cur ;
for (cur = Head; cur ! = NULL; cur = cur → next)
{ các xử lý với dữ liệu trong thành phần được trỏ bởi cur}

Ví dụ. Để in ra tất cả các dữ liệu trong DSLK, bạn có thể viết

for (cur = Head; cur ! = NULL; cur = cur → next)
    cout << cur → data << endl;

Có những xử lý trên DSLK mà để các thao tác được thực hiện dễ dàng, khi duyệt DSLK người ta sử dụng hai con trỏ "chạy" trên DSLK cách nhau một bước, con trỏ cur trỏ tới thành phần hiện thời, con trỏ pre trỏ tới thành phần đứng trước thành phần hiện thời, tức là cur = = pre → next. Ban đầu cur trỏ tới đầu DSLK, cur = Head còn pre = NULL. Sau đó mỗi lần chúng ta cho hai con trỏ tiến lên một bước, tức là đặt pre = cur và cur = cur → next.

Duyệt DSLK với hai con trỏ cur và pre là rất thuận tiện cho những trường hợp mà các xử lý với thành phần hiện thời liên quan tới thành phần đứng trước nó, chẳng hạn khi bạn muốn xen một thành phần mới vào trước thành phần hiện thời, hoặc bạn muốn loại bỏ thành phần hiện thời.

Chúng ta có thể biểu diễn lược đồ duyệt DSLK sử dụng hai con trỏ bởi vòng lặp sau:

```
Node*   cur = Head ;
Node*   pre = NULL;
while (cur! = NULL)
{
  Các xử lý với thành phần được trỏ bởi cur;
  pre = cur;
  cur = cur  → next;
}
```

Ví dụ. Giả sử bạn muốn loại khỏi DSLK tất cả các thành phần chứa dữ liệu là value. Để thực hiện được điều đó, chúng ta đi qua DSLK với hai

con trỏ. Khi thành phần hiện thời chứa dữ liệu value, chúng ta loại nó khỏi DSLK, nhưng trong trường hợp này chỉ cho con trỏ cur tiến lên một bước, còn con trỏ Pre đứng nguyên, và thu hồi bộ nhớ của thành phần bị loại. Chúng ta cũng cần chú ý đến thành phần cần loại là đầu hay ở giữa DSLK để có các hành động thích hợp. Khi thành phần hiện thời không chứa dữ liệu value, chúng ta cho cả hai con trỏ cur và Pre tiến lên một bước. Thuật toán loại khỏi DSLK tất cả các thành phần chứa dữ liệu value là như sau:

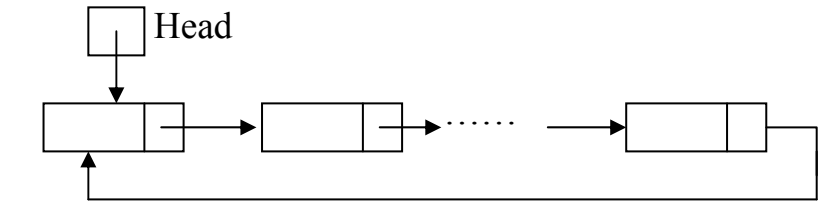
```
Node*   P;
Node*   cur = Head;
Node*   pre = NULL;
while (cur ! = NULL)
{
   if (cur  → data = = value)
   if (cur = = Head)
     {
        Head = Head → next;
        P = cur;
        cur = cur → next;
        delete  P;
      }
   else {
            pre → next = cur → next;
            P = cur;
            cur = cur → next;
            delete P;
         }
   else {
            pre = cur;
            cur = cur → next;
         }
} // hết while
```

5.3 CÁC DẠNG DSLK KHÁC.

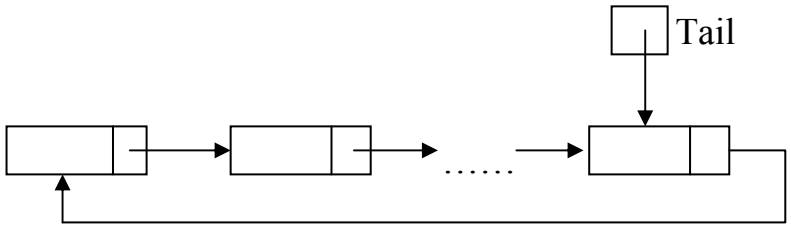
DSLK mà chúng ta đã xét trong mục 5.2 là DSLK đơn, mỗi thành phần của nó chứa một con trỏ trỏ tới thành phần đi sau, thành phần cuối cùng chứa con trỏ NULL. Trong mục này chúng ta sẽ trình bày một số dạng DSLK khác: DSLK vòng tròn, DSLK có đầu giả, DSLK kép. Và như vậy, trong một ứng dụng khi cần sử dụng DSLK, bạn sẽ có nhiều cách lựa chọn, bạn cần chọn dạng DSLK nào phù hợp với ứng dụng của mình.

5.3.1 DSLK vòng tròn

Giả sử trong chương trình bạn sử dụng một DSLK đơn (hình 5.3c) để lưu các dữ liệu. Trong chương trình ngoài các thao tác xen vào dữ liệu mới và loại bỏ các dữ liệu không cần thiết, giả sử bạn thường xuyên phải xử lý các dữ liệu theo trật tự đã lưu trong DSLK từ dữ liệu ở thành phần đầu tiên trong DSLK đến dữ liệu ở thành phần sau cùng, rồi quay lại thành phần đầu tiên và tiếp tục. Từ một thành phần, đi theo con trỏ next, chúng ta truy cập tới dữ liệu ở thành phần tiếp theo, song tới thành phần cuối cùng chúng ta phải cần đến con trỏ Head mới truy cập tới dữ liệu ở thành phần đầu. Trong hoàn cảnh này, sẽ thuận tiện hơn cho lập trình, nếu trong thành phần cuối cùng, ta cho con trỏ next trỏ tới thành phần đầu tiên trong DSLK để tạo thành một DSLK vòng tròn, như được minh hoạ trong hình 5.6a.



(a)



(b)

Hình 5.6. DSLK vòng tròn.

Trong DSLK vòng tròn, mọi thành phần đều bình đẳng, từ một thành phần bất kỳ chúng ta có thể đi qua toàn bộ danh sách. Con trỏ ngoài (có nó ta mới truy cập được DSLK) có thể trỏ tới một thành phần bất kỳ trong DSLK vòng tròn. Tuy nhiên để thuận tiện cho các xử lý, chúng ta vẫn tách biệt ra một thành phần đầu tiên và thành phần cuối cùng như trong DSLK đơn. Nếu chúng ta sử dụng con trỏ ngoài trỏ tới thành phần đầu tiên như trong hình 5.6a, thì để truy cập tới thành phần cuối cùng chúng ta không có cách nào khác là phải đi qua danh sách. Song nếu chúng ta sử dụng một con trỏ ngoài Tail trỏ tới thành phần cuối cùng như hình 5.6b, thì chúng ta có thể truy cập

tới cả thành phần cuối và thành phần đầu tiên, bởi vì con trỏ Tail → next trỏ tới thành phần đầu tiên. Do đó, sau này khi nói tới DSLK vòng tròn ta cần hiểu là DSLK vòng tròn với con trỏ ngoài Tail trỏ tới thành phần cuối cùng, như trong hình 5.6b. Khi DSLK vòng tròn rỗng, giá trị của con trỏ Tail là NULL.

Với DSLK vòng tròn, khi thực hiện các thao tác xen, loại trong các hoàn cảnh đặc biệt, bạn cần lưu ý để khỏi mắc sai sót. Chẳng hạn, từ DSLK vòng tròn rỗng, khi xen vào một thành phần mới được trỏ tới bởi con trỏ Q, bạn cần viết:

Tail = Q ;
Tail → next = Tail;

Thêm vào dòng lệnh Tail → next = Tail để tạo thành vòng tròn.

Bạn cũng cần chú ý đến phép toán duyệt DSLK. Với DSLK đơn, ta cho con trỏ cur chạy trên DSLK bắt đầu từ cur = Head, rồi thì giá trị của con trỏ cur thay đổi bởi cur = cur → next cho tới khi cur có giá trị NULL. Nhưng với DSLK vòng tròn, giá trị của con trỏ next trong các thành phần không bao giờ là NULL, do đó điều kiện kết thúc vòng lặp cần phải thay đổi, chẳng hạn như sau:

```
if (Tail ! = NULL)
  {
    Node*   first = Tail → next;
    Node*   cur = first;
    do  {
          các xử lý với dữ liệu trong thành phần được trỏ bởi cur;
          cur = cur → next;
          }
    while (cur ! = first) ;
  }
```

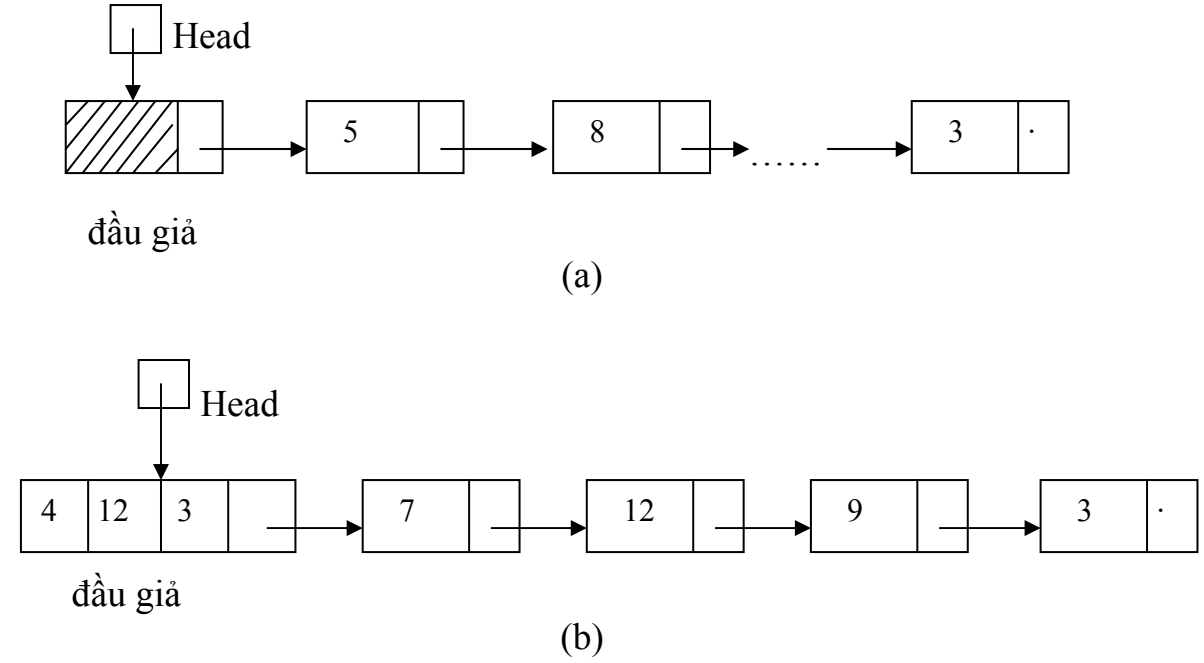
5.3.2 DSLK có đầu giả

Trong DSLK đơn, khi thực hiện phép toán xen, loại bạn cần phải xét riêng trường hợp xen thành phần mới vào đầu DSLK và loại thành phần ở đầu DSLK. Để tránh phải phân biệt các trường hợp đặc biệt này, người ta đưa vào DSLK một thành phần được gọi là đầu giả. Chẳng hạn, DSLK đơn trong hình 5.3c, nếu đưa vào đầu giả chúng ta có DSLK như trong hình 5.7a. Cần chú ý rằng, trong DSLK có đầu giả, các thành phần thực sự của danh sách bắt đầu từ thành phần thứ hai, tức là được trỏ bởi Head → next. DSLK có đầu giả không bao giờ rỗng, vì ít nhất nó cũng chứa đầu giả.

Khi đi qua DSLK có đầu giả sử dụng hai con trỏ: con trỏ trước Pre và con trỏ hiện thời cur, thì ban đầu Pre = Head và cur = Head → next. Để loại

thành phần được trỏ bởi cur ta không cần lưu ý thành phần đó có là đầu DSLK không, trong mọi hoàn cảnh, ta chỉ cần đặt:

Pre → next = cur → next;



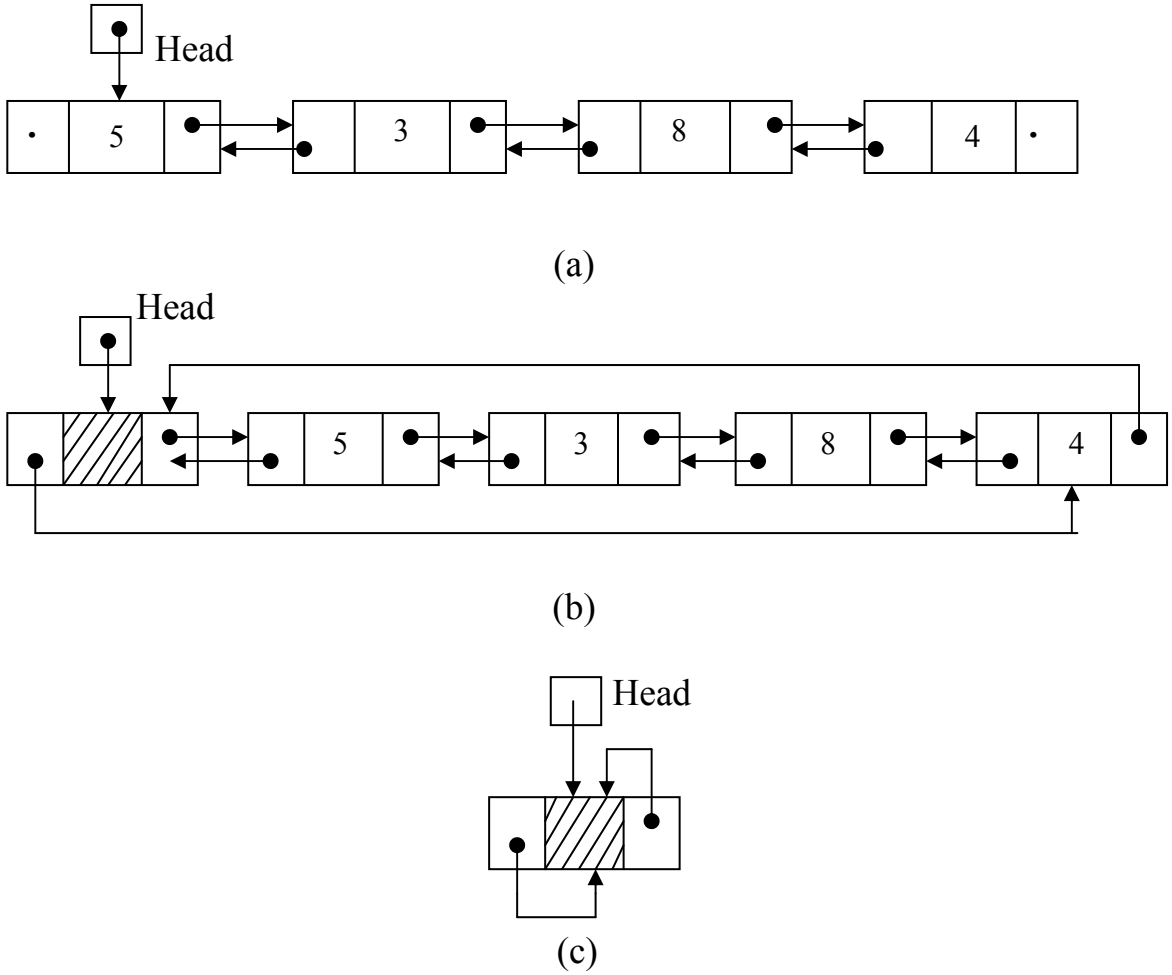
Hình 5.7. (a) DSLK có đầu giả
(b) DSLK với đầu giả lưu thông tin

Đôi khi, người ta sử dụng đầu giả để lưu một số thông tin về danh sách, chẳng hạn như độ dài, giá trị lớn nhất, giá trị nhỏ nhất trong danh sách, như trong hình 5.7b. Nhưng khi đó, đầu giả có thể có kiểu khác với các thành phần thực sự của DSLK. Và do đó các phép toán xen, loại vẫn cần các thao tác riêng cho các trường hợp xen một thành phần mới vào đầu DSLK và loại thành phần đầu của DSLK.

5.3.3 DSLK kép

Trong DSLK đơn, giả sử bạn cần loại một thành phần được định vị bởi con trỏ P, bạn cần phải biết thành phần đứng trước, nếu không biết bạn không có cách nào khác là phải đi từ đầu DSLK. Một hoàn cảnh khác, các xử lý với dữ liệu trong một thành phần lại liên quan tới các dữ liệu trong thành phần đi sau và cả thành phần đi trước. Trong các hoàn cảnh như thế, để thuận lợi người ta thêm vào mỗi thành phần của DSLK một con trỏ mới: con trỏ trước precede, nó trỏ tới thành phần đứng trước. Và khi đó ta có một DSLK kép, như trong hình 5.8a. Mỗi thành phần của DSLK kép chứa một dữ liệu và hai con trỏ: con trỏ next trỏ tới thành phần đi sau, con trỏ precede

trỏ tới thành phần đi trước. Giá trị của con trỏ precede ở thành phần đầu tiên và giá trị của con trỏ next ở thành phần sau cùng là hằng NULL.



Hình 5.8. (a) DSLK kép.
(b) DSLK kép vòng tròn có đầu giả.
(c) DSLK kép rỗng vòng tròn với đầu giả.

Để thuận tiện cho việc thực hiện các phép toán xen, loại trên DSLK kép, người ta thường sử dụng **DSLK kép vòng tròn có đầu giả** như được minh hoạ trong hình 5.8b. Cần lưu ý rằng một DSLK kép rỗng vòng tròn với đầu giả sẽ có dạng như trong hình 5.8c. Bạn có thể khởi tạo ra nó bởi các dòng lệnh:

Head = new Node;
Head → next = Head;
Head → precede = Head;

Với DSLK kép vòng tròn có đầu giả, bạn có thể thực hiện các phép toán xen, loại mà không cần quan tâm tới vị trí đặc biệt. Xen, loại ở vị trí đầu tiên hay cuối cùng cũng giống như ở vị trí bất kỳ khác.

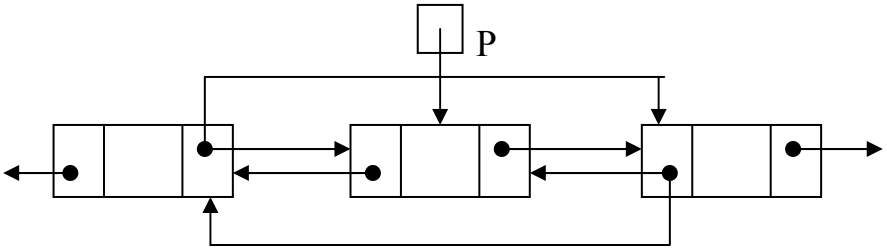
Giả sử bạn cần loại thành phần được trỏ bởi con trỏ P, bạn chỉ cần tiến hành các thao tác được chỉ ra trong hình 5.9a, tức là:

1. Cho con trỏ next của thành phần đi trước P trỏ tới thành phần đi sau P.
2. Cho con trỏ precede của thành phần đi sau P trỏ tới thành phần đi trước P.

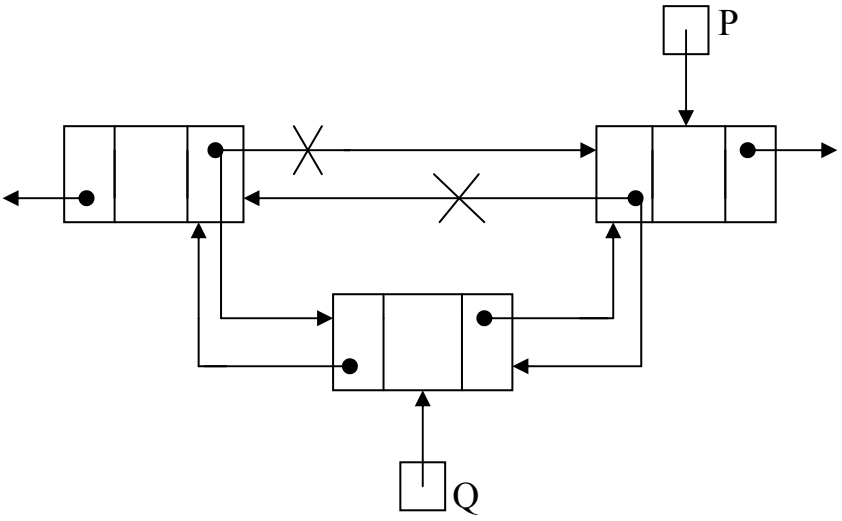
Các thao tác trên được thực hiện bởi các dòng lệnh sau:

P → precede → next = P → next;

P → next → precede = P → precede;



(a)



(b)

Hình 5.9. (a) Loại khỏi DSLK kép thành phần P.
(b) Xen thành phần Q vào trước thành phần P.

Chúng ta có thể xen vào DSLK kép một thành phần mới được trỏ bởi con trỏ Q vào trước thành phần được trỏ bởi con trỏ P bằng các thao tác được chỉ ra trong hình 5.9b

1. Đặt con trỏ next của thành phần mới trỏ tới thành phần P.
2. Đặt con trỏ precede của thành phần mới trỏ tới thành phần đi trước P.
3. Đặt con trỏ next của thành phần đi trước P trỏ tới thành phần mới.
4. Đặt con trỏ precede của thành phần P trỏ tới thành phần mới.

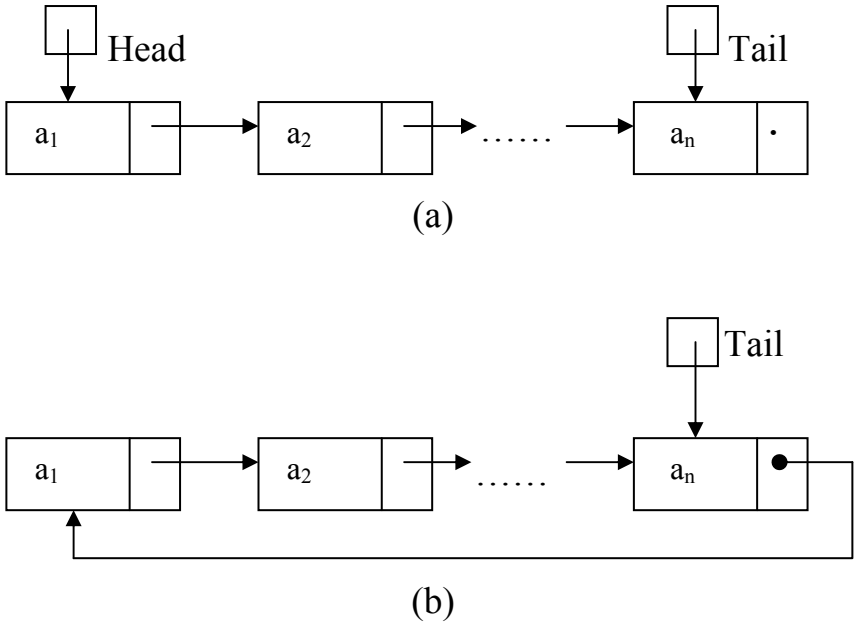
Các dòng lệnh sau thực hiện các hành động trên:

```
Q → next = P;
Q → precede = P → precede;
P → precede → next = Q;
P → precede = Q;
```

Việc xen một thành phần mới vào sau một thành phần đã định vị trong DSLK kép cũng được thực hiện tương tự.

5.4 CÀI ĐẶT DANH SÁCH BỞI DSLK

Trong chương 4, chúng ta đã nghiên cứu phương pháp cài đặt KDLTT danh sách bởi mảng, tức là một danh sách (a_1, a_2, … , a_n) sẽ được lưu ở đoạn đầu của mảng. Mục này sẽ trình bày một cách cài đặt khác: cài đặt bởi DSLK, các phần tử của danh sách sẽ được lần lượt được lưu trong các thành phần của DSLK. Để cho phép toán Append (thêm một phần tử mới vào đuôi danh sách) được thực hiện dễ dàng, chúng ta có thể sử dụng DSLK với hai con trỏ ngoài Head và Tail (hình 5.10a) hoặc DSLK vòng tròn với một con trỏ ngoài Tail (hình 5.10b). Nếu lựa chọn cách thứ hai, tức là cài đặt danh sách bởi DSLK vòng tròn, thì chỉ cần một con trỏ ngoài Tail, ta có thể truy cập tới cả đuôi và đầu (bởi Tail → next) và thuận tiện cho các xử lý mang tính "lần lượt từng thành phần và quay vòng" trên danh sách. Sau đây chúng ta sẽ cài đặt danh sách bởi DSLK với hai con trỏ ngoài Head và Tail như trong hình 5.10a. Việc cài đặt danh sách bởi DSLK vòng tròn với một con trỏ ngoài Tail (hình 5.10b) để lại cho độc giả xem như bài tập.



Hình 5.10. Cài đặt danh sách (a_1, a_2, … , a_n) bởi DSLK

Cần lưu ý rằng, nếu cài đặt DSLK chỉ với một con trỏ ngoài Head, thì khi cần xen vào đuôi danh sách chúng ta phải đi từ đầu lần lượt qua các thành phần mới đạt tới thành phần cuối cùng.

Chúng ta sẽ cài đặt KDLTT danh sách bởi lớp khuôn phụ thuộc tham biến kiểu Item, với Item là kiểu của các phần tử trong danh sách (như chúng ta đã làm trong các mục 4.2, 4.3). Danh sách sẽ được cài đặt bởi ba lớp: lớp các thành phần của DSLK (được đặt tên là lớp LNode), lớp danh sách liên kết (lớp LList) và lớp công cụ lặp (lớp LListIterator). Sau đây chúng ta sẽ lần lượt mô tả ba lớp này.

Lớp LNode sẽ chứa hai thành phần dữ liệu: biến data để lưu dữ liệu có kiểu Item và con trỏ next trỏ tới thành phần đi sau. Lớp này chỉ chứa một hàm kiến tạo để tạo ra một thành phần của DSLK chứa dữ liệu value, và giá trị của con trỏ next là NULL. Mọi thành phần của lớp này đều là private. Tuy nhiên, các lớp LList và LListIterator cần có khả năng truy cập trực tiếp đến các thành phần của lớp LNode. Do đó, chúng ta khai bào các lớp LList và LListIterator là bạn của lớp này. Định nghĩa lớp LNode được cho trong hình 5.11.

```
template <class Item>
class    LList ; // khai báo trước lớp LList.
```

```
template <class Item>
class    LListIterator ;  // khai báo trước.

template <class  Item>
class    LNode
{
     LNode (const Item & value)
     {  data = value ; next = NULL ; }
     Item  data ;
     LNode *  next ;
     friend  class   LList<Item> ;
     friend  class   LListIterator<Item> ;
} ;
```

Hình 5.11. Lớp LNode.

Lớp LList. Lớp này chứa ba thành phần dữ liệu: con trỏ Head trỏ tới đầu DSLK, con trỏ Tail trỏ tới đuôi của DSLK, biến length lưu độ dài của danh sách. Lớp LList chứa các hàm thành phần cũng giống như trong lớp Dlist (xem hình 4.3) với một vài thay đổi nhỏ. Chúng ta cũng khai báo lớp LListIterator là bạn của LList, để nó có thể truy cập trực tiếp tới các thành phần dữ liệu của lớp LList. Định nghĩa là LList được cho trong hình 5.12.

```
template <class Item>
class    LListIterator ;

template <class Item>
class     LList
{
  friend  class  LListIterator <Item> ;
  public :
       LList( )  // khởi tạo danh sách rỗng
       { Head = NULL; Tail = NULL ; length = 0 ; }
       LList (const  LList & L) ; // Hàm kiến tạo copy
       ~ LList( ) ; // Hàm huỷ
       LList & operator = (const  LList & L); // Toán tử gán.
       bool  Empty( )  const
       { return  length = = 0; }
       int  Length( )  const
       { return  length ; }
       void  Insert(const Item & x, int i);
       // xen phần tử x vào vị trí thứ i trong danh sách.
```

```
        void   Append(const Item & x);
        // xen phần tử x vào đuôi danh sách.
        void   Delete(int i);
        // loại phần tử ở vị trí thứ i trong danh sách.
        Item &   Element(int i);
        // trả về phần tử ở vị trí thứ i trong danh sách.
    private :
        LNode <Item> *  Head ;
        LNode <Item> *  Tail ;
        int   length ;
} ;
```

Hình 5.12. Định nghĩa lớp LList.

Bây giờ chúng ta xét sự cài đặt các hàm thành phần của lớp LList. Lớp LList chứa các thành phần dữ liệu được cấp phát động, vì vậy trong lớp này, ngoài hàm kiến tạo mặc định khởi tạo ra danh sách rỗng, được cài đặt inline, chúng ta cần phải đưa vào hàm kiến tạo copy, hàm huỷ và toán tử gán. Sau đây chúng ta xét lần lượt các hàm này.

Hàm kiến tạo copy. Hàm này thực hiện nhiệm vụ tạo ra một DSLK mới với các con trỏ ngoài Head và Tail chứa các dữ liệu như trong DSLK đã cho với con trỏ ngoài L.Head và L.Tail. Nếu DSLK đã cho không rỗng, đầu tiên ta tạo ra DSLK mới gồm một thành phần chứa dữ liệu như trong thành phần đầu tiên của DSLK đã cho, rồi sau đó lần lượt xen vào DSLK mới các thành phần chứa dữ liệu như trong các thành phần tiếp theo của DSLK đã cho. Có thể cài đặt hàm kiến tạo copy như sau:

```
LList <Item> :: LList(const  LList<Item> & L)
{
  if (L.Empty( ))
        { Head = Tail = NULL ; length = 0 ; }
  else {
        Head = new LNode <Item> (L.Head → data);
        Tail = Head ;
        LNode <Item> * P ;
        for (P = L.Head  → next; P! = NULL ; P = P → next)
              Append (P → data) ;
        length = L.length ;
        }
}
```

Hàm huỷ. Hàm này cần thu hồi tất cả bộ nhớ đã cấp phát cho các thành phần của DSLK trả về cho hệ thống và đặt con trỏ Head và Tail là NULL. Muốn vậy, chúng ta thu hồi từng thành phần kể từ thành phần đầu tiên của DSLK. Hàm huỷ được cài đặt như sau:

```
LList <Item> :: ~ LList(  )
{
  if (Head ! = NULL)
     {
       LNode <Item> * P ;
       while (Head ! = NULL)
           {
             P = Head ;
             Head = Head  → next ;
             delete  P;
           }
        Tail = NULL ;
         length = 0 ;
     }
 }
```

Toán tử gán. Nhiệm vụ của toán tử gán như sau: ta có một đối tượng (được trỏ bởi con trỏ this) chứa DSLK với các con trỏ ngoài Head và Tail và một đối tượng khác L chứa DSLK với các con trỏ ngoài L.Head và L.Tail, phép gán cần phải làm cho đối tượng *this chứa DSLK là bản sao của DSLK trong đối tượng L. Để thực hiện được điều đó, đầu tiên chúng ta cần làm cho DSLK trong đối tượng *this trở thành rỗng (với các hành động như trong hàm huỷ), sau đó copy DSLK trong đối tượng L (giống như hàm kiến tạo copy). Hàm operator = trả về đối tượng *this. Cài đặt cụ thể hàm toán tử gán để lại cho độc giả, xem như bài tập.

Sau đây chúng ta cài đặt các hàm thực hiện các phép toán trên danh sách.

Hàm Append (xen một phần tử mới x vào đuôi danh sách). Hàm này rất đơn giản, chỉ cần sử dụng thao tác xen một thành phần mới chứa dữ liệu x vào đuôi DSLK

```
void  LList <Item> :: Append (const Item & x)
{
  LNode <Item> * Q = new  LNode <Item> (x) ;
  if (Empty(  ))
        { Head = Tail = Q ; }
  else {
```

```
            Tail → next = Q ;
            Tail = Q ;
        }
    length + + ;
}
```

Các hàm xen phần tử x vào vị trí thứ i trong danh sách, loại phần tử ở vị trí thứ i khỏi danh sách và tìm phần tử ở vị trí thứ i của danh sách đều có một điểm chung là cần phải định vị được thành phần của DSLK chứa phần tử thứ i của danh sách. Chúng ta sẽ sử dụng con trỏ P chạy trên DSLK bắt đầu từ đầu, đi theo con trỏ next trong các thành phần của DSLK, đến vị trí mong muốn thì dừng lại. Một khi con trỏ P trỏ tới thành phần của DSLK chứa phần tử thứ i – 1 của danh sách, thì việc xen vào vị trí thứ i của danh sách phần tử mới x tương đương với việc xen vào DSLK một thành phần mới chứa dữ liệu x sau thành phần được trỏ bởi con trỏ P. Việc loại khỏi danh sách phần tử ở vị trí thứ i có nghĩa là loại khỏi DSLK thành phần đi sau thành phần được trỏ bởi P. Các hàm xen và loại được cài đặt như sau:

Hàm Insert

```
template  <class Item>
void   LList<Item> :: Insert(const  Item & x, int  i)
{
  assert ( i > = 1 && i < = length) ;
  LNode <Item> * Q = new LNode<Item> (x) ;
  if (i = = 1)  // xen vào đầu DSLK.
        {
          Q → next = Head ;
           Head = Q ;
        }
  else  {
           LNode <Item> * P = Head ;
           for (int k = 1 ; k < i – 1 ; k + + )
                    P = P → next ;
           Q → next = P → next ;
           P → next = Q ;
         }
   length + + ;
}
```

Hàm Delete

```
template <class Item>
void  LList <Item> :: Delete (int i)
```

```
{
   assert ( i > = 1 && i <= length) ;
   LNode<Item> * P ;
   If ( k = = 1)  // Loại đầu DSLK
         {
            P = Head ;
            Head = Head → next ;
            Delete P ;
            if (Head = = NULL)  Tail = NULL ;
         }
   else {
            P = Head ;
            for (int k = q ; k < i – 1; k + +)
                  P = P → next ;
            LNode <Item> * Q ;
            Q = P → next ;
            P →next = Q → next ;
            delete Q;
            if ( P → next = = NULL)  Tail = P ;
         }
   length - - ;
}
```

Hàm tìm phần tử ở vị trí thứ i

```
template <class Item>
Item & LList <Item> :: Element(int i)
{
  assert ( i > = 1 && i < = length) ;
  LNode <Item> * P = Head;
  for (int k = 1 ; k < i ; k + +)
        P = P → next ;
  return    P → data ;
}
```

Bây giờ chúng ta cài đặt lớp công cụ lặp LListIterator. Lớp này chứa các hàm thành phần giống như các hàm thành phần trong lớp DlistIterator (xem hình 4.4). Lớp LListIterator chứa một con trỏ hằng LlistPtr trỏ tới đối tượng của lớp LList, ngoài ra để cho các phép toán liên quan tới duyệt DSLK được thực hiện thuận tiện, chúng ta đưa vào lớp LListIterator hai biến con trỏ: con trỏ current trỏ tới thành phần hiện thời, con trỏ pre trỏ tới thành phần đứng trước thành phần hiện thời trong DSLK thuộc đối tượng mà con trỏ LListPtr trỏ tới. Định nghĩa lớp LListIterator được cho trong hình 5.12.

```
template <class  Item>
class      LListIterator
{
  public :
       LListIterator (const LList<Item> & L) // Hàm kiến tạo.
          { LListPtr = & L; current = NULL ; pre = NULL; }
       void   Start( )
          { current = LListPtr →Head; pre = NULL: }
       void   Advance( )
         { assert (current ! = NULL); pre = current; current =
                                    current → next; }
       bool    Valid( )
         {return    current ! = NULL; }
      Item &    Current( )
         { assert (current! = NULL); return current → data; }
      void    Add(const Item & x);
      void     Remove( );
   private :
      const  LList<Item> *   LlistPtr;
      LNode<Item> *   current;
      LNode<Item> *   pre ;
}
```

Hình 5.12. Lớp công cụ lặp LListIterator.

Các hàm thành phần của lớp LListIterator đều rất đơn giản và được cài đặt inline, trừ ra hai hàm Add và Remove. Cài đặt hai hàm này cũng chẳng có gì khó khăn cả, chúng ta chỉ cần sử dụng các phép toán xen, loại trên DSLK đã trình bày trong mục 5.2.1. Các hàm này được cài đặt như sau:

Hàm Add

```
template  <class Item>
void  LListIterator<Item> :: Add (const Item & x)
{
  assert (current ! = NULL);
  LNode <Item> * Q = new LNode<Item> (x);
  if  (current = = LListPtr → Head)
  {
      Q → next = LListPtr → Head;
      LListPtr →Head = Q;
```

```
        pre = Q;
    }
    else {
            Q →next = current;
            pre → next = Q;
            pre = Q ;
        }
    LListPtr → length + + ;
}
```

Hàm Remove

```
template <class Item>
void LListIterator<Item> :: Remove( )
{
  assert (current ! = NULL);
  if (current = = LListPtr → Head)
  {
      LListPtr → Head = current →next;
      delete current ;
      current = LListPtr →Head ;
      if (LListPtr →Head = = NULL)
                LListPtr → Tail = NULL;
  }
  else {
          pre → next = current → next;
          delete current;
          current = pre → next;
          if (current = = NULL)
                  LListPtr →Tail = pre;
        }
        LListPtr →length - - ;
}
```

5.5 SO SÁNH HAI PHƯƠNG PHÁP CÀI ĐẶT DANH SÁCH

Một KDLTT có thể cài đặt bởi các phương pháp khác nhau. Trong chương 4, chúng ta đã nghiên cứu cách cài đặt KDLTT danh sách bởi mảng (mảng tĩnh hoặc mảng động). Mục 5.4 đã trình bày cách cài đặt danh sách bởi DSLK. Trong mục này chúng ta sẽ phân tích đánh giá ưu khuyết điểm của mỗi phương pháp. Dựa vào sự phân tích này, bạn có thể đưa ra quyết định lựa chọn cách cài đặt nào cho phù hợp với ứng dụng của mình.

Cài đặt danh sách bởi mảng là cách lựa chọn tự nhiên hợp lý: các phần tử của danh sách lần lượt được lưu trong các thành phần liên tiếp của mảng, kể từ đầu mảng. Giả sử bạn cài đặt danh sách bởi mảng tĩnh có cỡ là MAX. Nếu MAX là số rất lớn, khi danh sách còn ít phần tử, thì cả một không gian nhớ rộng lớn trong mảng không sử dụng đến, và do đó sẽ lãng phí bộ nhớ. Khi danh sách phát triển, tới một lúc nào đó nó có thể có số phần tử vượt quá cỡ của mảng. Đó là hạn chế cơ bản của cách cài đặt danh sách bởi mảng tĩnh. Bây giờ giả sử bạn lưu danh sách trong mảng động. Mỗi khi mảng đầy, bạn có thể cấp phát một mảng động mới có cỡ gấp đôi mảng động cũ. Nhưng khi đó bạn lại phải mất thời gian để sao chép dữ liệu từ mảng cũ sang mảng mới. Sự lãng phí bộ nhớ vẫn xảy ra khi mà danh sách thì ngắn mà cỡ mảng thì lớn.

Trong cách cài đặt danh sách bởi DSLK, các phần tử của danh sách được lưu trong các thành phần của DSLK, các thành phần này được cấp phát động. DSLK có thể móc nối thêm các thành phần mới hoặc loại bỏ các thành phần trả về cho hệ thống mỗi khi cần thiết. Do đó cài đặt danh sách bởi DSLK sẽ tiết kiệm được bộ nhớ.

Một ưu điểm của cài đặt danh sách bởi mảng là ta có thể truy cập trực tiếp tới mỗi phần tử của danh sách. Nếu ta cài đặt danh sách bởi mảng A, thì phần tử thứ i trong danh sách được lưu trong thành phần A[i - 1] của mảng, do đó thời gian truy cập tới phần tử bất kỳ của danh sách là O(1). Vì vậy thời gian của phép toán tìm phần tử thứ i trong danh sách Element(i) là O(1), với i bất kỳ.

Song nếu chúng ta sử dụng DSLK để cài đặt danh sách, thì chúng ta không có cách nào truy cập trực tiếp tới thành phần của DSLK chứa phần tử thứ i của danh sách. Chúng ta phải sử dụng con trỏ P chạy trên DSLK bắt đầu từ đầu, lần lượt qua các thành phần kế tiếp để đạt tới thành phần chứa phần tử thứ i của danh sách. Do vậy, thời gian để thực hiện phép toán tìm phần tử thứ i của danh sách khi danh sách được cài đặt bởi DSLK là phụ thuộc vào i và là O(i).

Nếu danh sách được cài đặt bởi mảng A, thì để xen một phần tử mới vào vị trí thứ i trong danh sách (hoặc loại khỏi danh sách phần tử ở vị trí thứ i), chúng ta phải "đẩy" các phần tử của danh sách chứa trong các thành phần của mảng kể từ A[i] ra phía sau một vị trí (hoặc đẩy lên trước một vị trí các phần tử chứa trong các thành phần kể từ A[i + 1]). Do đó, các phép toán Insert(x,i) và Delete(i) đòi hỏi thời gian O(n – i), trong đó n là độ dài của danh sách.

Mặt khác, nếu cài đặt danh sách bởi DSLK thì để thực hiện phép toán xen, loại ở vị trí thứ i của danh sách, chúng ta lại phải mất thời gian để định vị thành phần của DSLK chứa phần tử thứ i của danh sách (bằng cách cho con trỏ P chạy từ đầu DSLK). Do đó, thời gian thực hiện các phép toán xen, loại là O(i).

Thời gian thực hiện các phép toán danh sách trong hai cách cài đặt bởi mảng và bởi DSLK được cho trong bảng sau, trong bảng này n là độ dài của danh sách.

Phép toán	Danh sách cài đặt bởi mảng	Danh sách cài đặt bởi DSLK
Insert (x, i)	O(n – i)	O(i)
Delete (i)	O(n – i)	O(i)
Append (x)	O(1)	O(1)
Element (i)	O(1)	O(i)
Add (x)	O(n)	O(1)
Remove ( )	O(n)	O(1)
Current ( )	O(1)	O(1)

5.6 CÀI ĐẶT TẬP ĐỘNG BỞI DSLK

Trong mục 4.4, chúng ta đã nghiên cứu phương pháp cài đặt tập động bởi mảng. Ở đó lớp tập động DSet đã được cài đặt bằng cách sử dụng lớp danh sách động Dlist như lớp cơ sở private. Đương nhiên chúng ta cũng có thể sử dụng lớp danh sách liên kết LList như lớp cơ sở private để cài đặt lớp DSet. Song cài đặt như thế thì phép toán tìm kiếm trên tập động sẽ đòi hỏi thời gian không phải là O(n) như khi sử dụng lớp DList. Hàm tìm kiếm tuần tự trong lớp DSet sử dụng lớp cơ sở DList (xem mục 4.4) chứa dòng lệnh:

for (int i = 1; i < = length( ); i + +)
if (Element (i). key = = k) return true;

Trong lớp DList, thời gian của phép toán Element(i) là O(1) với mọi i, do đó thời gian của phép toán tìm kiếm là O(n), với n là độ dài của danh sách. Tuy nhiên trong lớp LList, thời gian của Element(i) là O(i), do đó thời gian của phép toán tìm kiếm trên tập động nếu chúng ta cài đặt lớp DSet bằng cách sử dụng lớp LList làm lớp cơ sở private sẽ là $O(n^2)$.

Một cách tiếp cận khác để cài đặt tập động bởi DSLK là chúng ta biểu diễn tập động bởi một List với List là một đối tượng của lớp LList. Lớp DSet sẽ chứa một thành phần dữ liệu là List.

Cũng như trong mục 4.4, chúng ta giả thiết rằng tập động chứa các dữ liệu có kiểu Item, và Item là một cấu trúc chứa một thành phần là khoá (key) với kiểu là keyType. Lớp DSet được định nghĩa như sau:

```
template  <class Item>
class     DSet
{
  public:
       void  DsetInsert (const  Item & x);
       void  DsetDelete (keyType k) ;
```

```
        bool  Search (keyType k)  const ;
        Item & Max( )  const ;
        Item & Min( )  const ;
    private :
         LList <Item>   List;
};
```

Chú ý rằng, lớp DSet có các hàm sau đây được tự động thừa hưởng từ lớp LList:

- Hàm kiến tạo mặc định tự động (hàm kiến tạo copy tự động), nó kích hoạt hàm kiến tạo mặc định (hàm copy, tương ứng) của lớp LList để khởi tạo đối tượng List.
- Toán tử gán tự động, nó kích hoạt toán tử gán của lớp LList.
- Hàm huỷ tự động, nó kích hoạt hàm huỷ của lớp LList.

Các phép toán tập động sẽ được cài đặt bằng cách sử dụng các phép toán bộ công cụ lặp để duyệt DSLK List. Chẳng hạn, hàm tìm kiếm được cài đặt như sau:

```
template  <class Item>
bool  DSet<Item> :: Search(keyType k)
{
  LListIterator<Item>   It<List> ;   // Khởi tạo It là đối tượng của lớp
                                     // công cụ lặp, It gắn với List.
  for (It.Start( ) ; It.Valid( ); It.Advance( ))
       if (It.Current( ).key = = k)
                    return     true;
  return     false;
}
```

Hàm loại khỏi tập động dữ liệu với khoá k được cài đặt tương tự: duyệt DSLK List, khi gặp dữ liệu cần loại thì sử dụng hàm Remove( ) trong bộ công cụ lặp.

```
template <class  Item>
void    DSet<Item> :: DsetDelete (keyType   k)
{
  LListIterator<Item>    It(List);
  for  (It.Start( ); It.Valid( ); It.Advance( ))
       if (It.Current( ).key = = k)
           { It.Remove( ) ; break); }
}
```

Hàm xen một dữ liệu mới x vào tập động được thực hiện bằng cách gọi hàm Append trong lớp LList để xen x vào đuôi DSLK.

```
template  <class Item>
void  DSet<Item> :: DsetInsert(const Item & x)
{
  if (! Search (x.key))
          List.Append(x);
}
```

Các phép toán tìm phần tử có khoá lớn nhất (hàm Max), tìm phần tử có khoá nhỏ nhất (hàm Min) để lại cho độc giả , xem như bài tập.

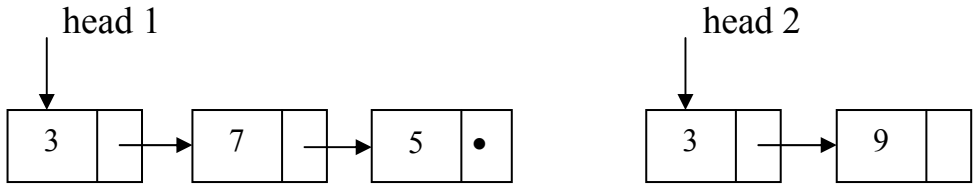
Các phép toán trong bộ công cụ lặp chỉ cần thời gian O(1), do đó với cách cài đặt lớp DSet như trên, tất cả các phép toán trong tập động chỉ đòi hỏi thời gian O(n), với n là số dữ liệu trong tập động.

Chú ý. Khi cài đặt tập động bởi DSLK chúng ta chỉ có thể tìm kiếm tuần tự. Cho dù các dữ liệu của tập động được lưu trong DSLK lần lượt theo giá trị khoá tăng dần, chúng ta cũng không thể áp dụng kỹ thuật tìm kiếm nhị phân, lý do đơn giản là trong DSLK chúng ta không thể truy cập trực tiếp tới thành phần ở giữa DSLK.

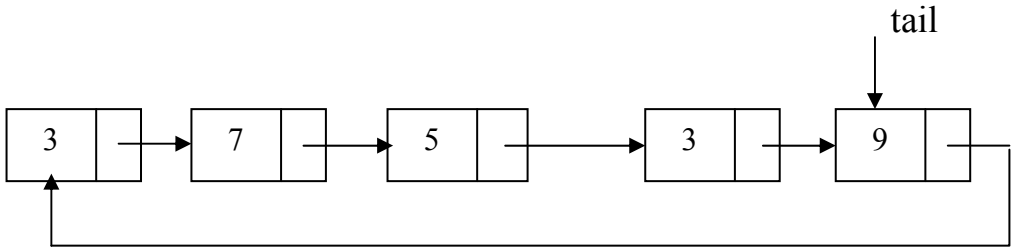
BÀI TẬP.

1. Cho DSLK đơn với con trỏ ngoài head trỏ tới đầu DSLK, và P là con trỏ trỏ tới một thành phần của DSLK đó. Hãy viết ra các mẫu hàm và cài đặt các hàm thực hiện các nhiệm vụ sau:
   a. Xen thành phần mới chứa dữ liệu d vào trước P.
   b. Loại thành phần P.
   c. In ra tất cả các dữ liệu trong DSLK.
   d. Loại khỏi DSLK tất cả các thành phần chứa dữ liệu d.

2. Cho hai DSLK, hãy viết hàm kết nối hai DSLK đó thành một DSLK vòng tròn, chẳng hạn với hai DSLK:



ta nhận được DSLK vòng tròn sau:



Chú ý rằng, một trong hoặc cả hai DSLK đã cho có thể rỗng.

3. Cho DSLK vòng tròn với con trỏ ngoài tail trỏ tới đuôi DSLK. Hãy cài đặt các hàm sau:
    a. Xen thành phần mới chứa dữ liệu d vào đuôi DSLK.
    b. Xen thành phần mới chứa dữ liệu d vào đầu DSLK.
    c. Loại thành phần ở đầu DSLK.

4. Hãy cài đặt các hàm kiến tạo copy, hàm huỷ, toán tử gán trong lớp Dlist bởi các hàm đệ quy.

5. Hãy cài đặt lớp Llist, trong đó danh sách được cài đặt bởi DSLK vòng tròn với một con trỏ ngoài tail.

CHƯƠNG 6

NGĂN XẾP

Trong chương này, chúng ta sẽ trình bày KDLTT ngăn xếp. Cũng giống như danh sách, ngăn xếp là CTDL tuyến tính, nó gồm các đối tượng dữ liệu được sắp thứ tự. Nhưng đối với danh sách, các phép toán xen, loại và truy cập có thể thực hiện ở vị trí bất kỳ của danh sách, còn đối với ngăn xếp các phép toán đó chỉ được thực hiện ở một đầu. Mặc dù các phép toán trên ngăn xếp là rất đơn giản, song ngăn xếp là một trong các CTDL quan trọng nhất. Trong chương này chúng ta sẽ đặc tả KDLTT ngăn xếp, sau đó sẽ nghiên cứu các phương pháp cài đặt ngăn xếp. Cuối cùng chúng ta sẽ trình bày một số ứng dụng của ngăn xếp.

6.1 KIỂU DỮ LIỆU TRỪU TƯỢNG NGĂN XẾP

Chúng ta có thể xem một chồng sách là một ngăn xếp. Trong chồng sách, các quyển sách đã được sắp xếp theo thứ tự trên - dưới, quyển sách nằm trên cùng được xem là ở đỉnh của chồng sách. Chúng ta có thể dễ dàng đặt một quyển sách mới lên đỉnh chồng sách và lấy quyển sách ở đỉnh ra khỏi chồng sách. Và như thế quyển sách được lấy ra khỏi chồng là quyển sách được đặt vào chồng sau cùng.

Ngăn xếp (**stack** hoặc đôi khi **pushdown store**) là một cấu trúc dữ liệu bao gồm các đối tượng dữ liệu được sắp xếp theo thứ tự tuyến tính, một trong hai đầu được gọi là **đỉnh** của ngăn xếp. Chúng ta chỉ có thể truy cập tới đối tượng dữ liệu ở đỉnh của ngăn xếp, thêm một đối tượng dữ liệu mới vào đỉnh của ngăn xếp và loại đối tượng dữ liệu ở đỉnh ra khỏi ngăn xếp.

Sau đây chúng ta sẽ đặc tả chính xác hơn các phép toán ngăn xếp. Chúng ta sẽ mô tả các phép toán ngăn xếp bởi các hàm, trong đó S ký hiệu một ngăn xếp và x là một đối tượng dữ liệu cùng kiểu với các đối tượng trong ngăn xếp S.

Các phép toán ngăn xếp:

1. Empty(S): Hàm trả về true nếu ngăn xếp S rỗng và false nếu ngược lại.
2. Push(S,x): Đẩy x vào đỉnh của ngăn xếp S. Chẳng hạn, S là ngăn xếp các số nguyên, S = [5, 3, 6, 4, 7], và đầu bên phải là đỉnh ngăn xếp, tức là 7 ở đỉnh của ngăn xếp. Nếu chúng ta đẩy số nguyên x = 2 vào đỉnh ngăn xếp, thì ngăn xếp trở thành S = [5, 3, 6, 4, 7, 2].
3. Pop(S): Loại đối tượng ở đỉnh của ngăn xếp S. Ví dụ, nếu S là ngăn xếp S = [5, 3, 6, 4, 7], thì khi loại phần tử ở đỉnh ngăn xếp, ngăn xếp trở thành S = [5, 3, 6, 4].

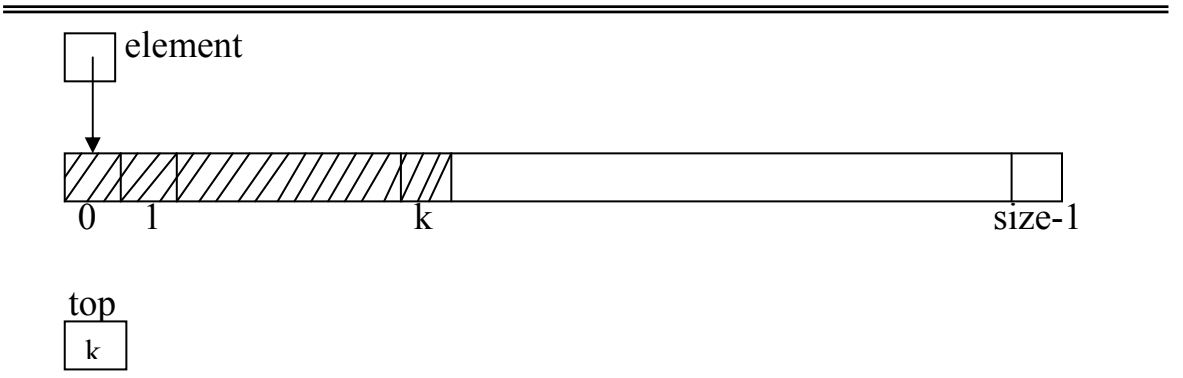
4. GetTop(S): Hàm trả về đối tượng ở đỉnh của S, ngăn xếp S không thay đổi.

Ngăn xếp được gọi là cấu trúc dữ liệu LIFO (viết tắt của cụm từ Last-In- First- Out), có nghĩa là đối tượng sau cùng được đưa vào ngăn xếp sẽ là đối tượng đầu tiên được lấy ra khỏi ngăn xếp.

Cũng giống như danh sách, chúng ta có thể cài đặt ngăn xếp bởi mảng hoặc bởi DSLK. Sau đây chúng ta sẽ nghiên cứu mỗi cách cài đặt đó.

6.2 CÀI ĐẶT NGĂN XẾP BỞI MẢNG

Chúng ta sử dụng một mảng element (mảng tĩnh hoặc động) để lưu giữ các đối tượng dữ liệu của ngăn xếp. Các thành phần mảng element[0], element[1], …, element[k] sẽ lần lượt lưu giữ các đối tượng của ngăn xếp. Đỉnh của ngăn xếp được lưu ở element[0] hay element[k] ? Nếu đỉnh của ngăn xếp ở element[0], thì khi thực hiện phép toán Push (Pop) chúng ta sẽ phải đẩy (dồn) các đối tượng được lưu trong đoạn mảng element[1…k] ra sau (lên trên) một vị trí. Do đó, hợp lý hơn, chúng ta chọn đỉnh ngăn xếp ở vị trí k trong mảng. Hình 6.1 mô tả cấu trúc dữ liệu cài đặt ngăn xếp sử dụng một mảng động element.



Hình 6.1. Ngăn xếp cài đặt bởi mảng động

Nếu cài đặt ngăn xếp bởi mảng như trên, thì các phép toán ngăn xếp là các trường hợp riêng của các phép toán trên danh sách: việc đẩy đối tượng x vào đỉnh của ngăn xếp là xen x vào đuôi danh sách (phép toán Append trên danh sách), còn việc loại (hoặc truy cập) phần tử ở đỉnh của ngăn xếp là loại (truy cập) phần tử ở vị trí thứ k + 1 của danh sách. Do đó chúng ta có thể sử dụng lớp DList (xem mục 4.3) để cài đặt lớp ngăn xếp Stack. Sẽ có hai cách lựa chọn:

- Xây dựng lớp Stack là lớp dẫn xuất từ lớp cơ sở DList (tương tự như chúng ta xây dựng lớp tập động DSet sử dụng lớp DList như lớp cơ sở private, xem mục 4.4).

- Thay cho sử dụng lớp DList làm lớp cơ sở private, chúng ta có thể thiết kế lớp Stack một cách khác: lớp Stack chứa một thành phần dữ liệu là đối tượng của lớp DList.

Cả hai cách trên cho phép ta sử dụng các hàm thành phần của lớp DList (các hàm Append, Delete, Element) để cài đặt các hàm thực hiện các phép toán ngăn xếp. Cài đặt lớp Stack theo các phương án trên để lại cho độc giả, xem như bài tập.

Bởi vì các phép toán ngăn xếp là rất đơn giản, cho nên chúng ta sẽ cài đặt lớp Stack trực tiếp, không sử dụng lớp DList.

Lớp Stack được thiết kế sau đây là lớp khuôn phụ thuộc tham biến kiểu Item, Item là kiểu của các phần tử trong ngăn xếp. Lớp Stack chứa ba thành phần dữ liệu: biến con trỏ element trỏ tới mảng được cấp phát động để lưu các phần tử của ngăn xếp, biến size lưu cỡ của mảng động, và biến top lưu chỉ số mảng là đỉnh ngăn xếp. Định nghĩa lớp Stack được cho trong hình 6.2.

```
template <class Item>
class    Stack
{
  public :
     Stack (int m = 1);
     //khởi tạo ngăn xếp rỗng với dung lượng m, m là số nguyên dương
     Stack (const Stack & S) ;
     // Hàm kiến tạo copy.
     ~ Stack( ) ; // Hàm huỷ.
     Stack &  operator = (const Stack & S);
     // Toán tử gán.
     // Các phép toán ngăn xếp:
     bool   Empty( )  const;
     // Xác định ngăn xếp có rỗng không.
     // Postcondition : Hàm trả về true nếu ngăn xếp rỗng, và trả về
     // false nếu không.
     void   Push(const Item & x);
     // Đẩy phần tử x vào đỉnh của ngăn xếp.
     // Postcondition: phần tử x trở thành đỉnh của ngăn xếp.
     Item &   Pop( );
     // Loại phần tử ở đỉnh của ngăn xếp.
     // Precondition: ngăn xếp không rỗng.
     // Postcondition: phần tử ở đỉnh ngăn xếp bị loại khỏi ngăn xếp,
     // và hàm trả về phần tử này.
     Item &   GetTop( )  const ;
     // Truy cập đỉnh ngăn xếp.
```

```
    // Precondition : ngăn xếp không rỗng.
    // Postcondition: Hàm trả về phần tử ở đỉnh ngăn xếp, ngăn xếp
    // không thay đổi.
  private:
    Item *   element ;
    int    size ;
    int    top ;
} ;
```

Hình 6.2. Lớp Stack.

Sau đây chúng ta cài đặt các hàm thành phần của lớp Stack. Trước hết nói về hàm kiến tạo Stack(int m), hàm này làm nhiệm vụ cấp phát một mảng động có cỡ m để lưu các phần tử của ngăn xếp, vì ngăn xếp rỗng mảng không chứa phần tử nào cả, biến top được đặt bằng –1.

```
template  <class   Item>
Stack<Item>   Stack(int m)
{
  element = new Item[m] ;
  size = m ;
  top = -1 ;
}
```

Các hàm kiến tạo copy, hàm huỷ, toán tử gán được cài đặt tương tự như trong lớp DList (xem mục 4.3). Các hàm thực hiện các phép toán ngăn xếp được cài đặt rất đơn giản như sau:

```
template <class  Item>
bool    Stack<Item> :: Empty(  )
{
 return    top = = -1 ;
}

template <class Item>
void   Stack<Item> : : Push(const Item & x)
{
 if  (top = = size –1) // mảng đầy
 {
     Item * A = new Item[2 * size] ;
     Assert (A! = NULL) ;
     for (int i = 0 ; i < = top ; i + +)
         A[i] = element[i] ;
```

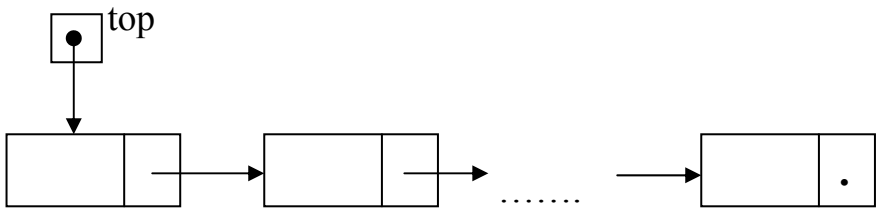
```
        size = 2 * size ;
        delete [  ] element ;
        element = A ;
    } ;
    element [+ + top] = x ;
}

template <class Item>
Item & Stack<Item> : : Pop(  )
{
    assert (top > = 0) ;
    return   element[top - -] ;
}

template <class Item>
Item & Stack<Item> : : GetTop(  )
{
    assert (top >= 0) ;
    return    element[top];
}
```

Chúng ta có nhận xét rằng, tất cả các phép toán ngăn xếp chỉ đòi hỏi thời gian O(1), trừ khi chúng ta thực hiện phép toán Push và mảng đã đầy. Khi đó chúng ta phải cấp phát một mảng động mới với cỡ gấp đôi mảng cũ và sao chép dữ liệu từ mảng cũ sang mảng mới. Do đó trong trường hợp mảng đầy, phép toán Push đòi hỏi thời gian O(n), với n là số phần tử trong ngăn xếp.

6.3 CÀI ĐẶT NGĂN XẾP BỞI DSLK

Ngăn xếp cũng có thể cài đặt bởi DSLK, và chúng ta có thể sử dụng lớp LList (xem mục 5.4) làm lớp cơ sở private để xây dựng lớp Stack. Tuy nhiên, khi lưu giữ các phần tử của ngăn xếp trong các thành phần của DSLK thì các phép toán ngăn xếp cũng được thực hiện rất đơn giản. Do đó chúng ta sẽ cài đặt trực tiếp lớp Stack, không sử dụng tới lớp LList.

Chúng ta sẽ cài đặt ngăn xếp bởi DSLK, đỉnh của ngăn xếp được lưu trong thành phần đầu tiên của DSLK, một con trỏ ngoài top trỏ tới thành phần này, xem hình 6.3.



Hình 6.3. Cài đặt ngăn xếp bởi DSLK.

Lớp Stack cài đặt KDLTT ngăn xếp sử dụng DSLK được mô tả trong hình 6.4. Lớp Stack này chỉ chứa một thành phần dữ liệu là con trỏ top trỏ tới thành phần đầu tiên của DSLK (như trong hình 6.3). Các thành phần của DSLK là cấu trúc Node gồm biến data có kiểu Item (Item là kiểu của các phần tử trong ngăn xếp), và biến con trỏ next trỏ tới thành phần đi sau. Lớp Stack ở đây chứa các hàm thành phần với khai báo và chú thích giống hệt như các hàm thành phần trong lớp Stack ở mục 6.2 (xem hình 6.2), trừ hàm kiến tạo.

```
template  <class Item>
class     Stack
{
  public :
     Stack(  ) // Hàm kiến tạo mặc định khởi tạo ngăn xếp rỗng.
     { top = NULL; }
     Stack (const  Stack & S) ;
     ~ Stack(  ) ;
     Stack &   operator = (const Stack & S) ;
     bool    Empty(  )   const
              { return  top = = NULL; }
     void    Push (const Item & x) ;
     Item &    Pop(  ) ;
     Item &    GetTop(  )   const ;
  private :
     struct  Node
     {
        Item    data ;
        Node *   next;
        Node (const Item & x)
        : data(x), next (NULL) {  }
     } ;
```

```
    Node *   top ;
} ;
```

Hình 6.4. Lớp Stack cài đặt bởi DSLK.

Bây giờ chúng ta cài đặt các hàm thành phần của lớp Stack. Hàm kiến tạo mặc định nhằm khởi tạo nên một ngăn xếp rỗng, muốn vậy chỉ cần đặt giá trị của con trỏ top là hằng NULL. Hàm này đã được cài đặt inline.

Hàm kiến tạo copy. Cho trước một ngăn xếp L, công việc của hàm kiến tạo copy là tạo ra một ngăn xếp mới là bản sao của L. Cụ thể hơn, phải tạo ra một DSLK với con trỏ ngoài top là bản sao của DSLK với con trỏ ngoài L.top. Nếu ngăn xếp L không rỗng, đầu tiên ta khởi tạo ra DSLK top chỉ có một thành phần chứa dữ liệu như trong thành phần đầu tiên của DSLK L.top, tức là:

top = new Node (L.top → data) ;

Sau đó, ta nối dài DSLK top bằng cách thêm vào các thành phần chứa dữ liệu như trong các thành phần tương ứng ở DSLK L.top. Điều đó được thực hiện bởi vòng lặp với con trỏ P chạy trên DSLK L.top và con trỏ Q chạy trên DSLK top. Ở mỗi bước lặp, cần thực hiện:

Q = Q → next = new Node (P → data) ;

Hàm kiến tạo copy được viết như sau:

```
template <class Item>
Stack <Item> : : Stack (const Stack & S)
{
  if (S.Empty( ))
          top = NULL ;
  else {
          Node * P = S.top ;
          top = new Node (P → data) ;
          Node * Q = top ;
          for  (P = P → next ; P! = NULL ; P = P → next)
                Q = Q → next = new Node (P →data) ;
        }
}
```

Hàm huỷ. Hàm cần thực hiện nhiệm vụ thu hồi bộ nhớ đã cấp phát cho từng thành phần của DSLK, lần lượt từ thành phần đầu tiên.

```
template <class Item>
Stack<Item> : : ~ Stack( )
{
  if (top! = NULL)
```

```
    {
        Node * P = top ;
        while (top ! = NULL)
        {
          P = top ;
          top = top → next ;
          delete P ;
        }
    }
}
```

Toán tử gán. Hàm này gồm hai phần. Đầu tiên ta huỷ DSLK top với các dòng lệnh như trong hàm ~Stack( ), sau đó tạo ra DSLK top là bản sao của DSLK S.top với các dòng lệnh như trong hàm kiến tạo copy.

Các hàm thực hiện các phép toán ngăn xếp được cài đặt rất đơn giản: việc đẩy một phần tử mới x vào đỉnh ngăn xếp chẳng qua là xen một thành phần chứa dữ liệu x vào đầu DSLK, còn việc loại (truy cập) phần tử ở đỉnh ngăn xếp đơn giản là loại (truy cập) thành phần đầu tiên của DSLK. Các phép toán ngăn xếp được cài đặt như sau:

```
template  <class Item>
void    Stack<Item> : : Push (const Item & x)
{
  Node* P = new Node(x) ;
  if  (top = = NULL)
          top = P ;
  else  {
          P → next = top ;
          top = P ;
        }
}

template  <class Item>
Item &   Stack<Item> : : Pop( )
{
   assert (top! = NULL) ;
   Item object = top → data ;
   Node* P = top ;
   top = top → next ;
   delete P ;
   return   object ;
}
```

```
template <class Item>
Item &  Stack<Item> : : GetTop( )
{
  assert( top! = NULL) ;
  return   top → data ;
}
```

Rõ ràng là khi cài đặt ngăn xếp bởi DSLK thì các phép toán ngăn xếp Push, Pop, GetTop chỉ cần thời gian O(1).

Sau đây chúng ta sẽ trình bày một số ứng dụng của ngăn xếp.

6.4 BIỂU THỨC DẤU NGOẶC CÂN XỨNG

Ngăn xếp được sử dụng nhiều trong các chương trình dịch (compiler). Trong mục này chúng ta sẽ trình bày vấn đề: sử dụng ngăn xếp để kiểm tra tính cân xứng của các dấu ngoặc trong chương trình nguồn.

Trong chương trình ở dạng mã nguồn, chẳng hạn chương trình viết bằng ngôn ngữ C + +, chúng ta sử dụng nhiều các dấu mở ngoặc "{" và đóng ngoặc "}", mỗi dấu "{" cần phải có một dấu "}" tương ứng đi sau. Tương tự, trong các biểu thức số học (hoặc logic) chúng ta cũng sử dụng các dấu mở ngoặc "(" và đóng ngoặc ")", mỗi dấu "(" cần phải tương ứng với một dấu ")". Nếu chúng ta loại bỏ tất cả các ký hiệu khác, chỉ giữ lại các dấu mở ngoặc và đóng ngoặc thì chúng ta sẽ có một dãy các dấu mở ngoặc và đóng ngoặc mà ta gọi là **biểu thức dấu ngoặc** và nó cần phải cân xứng. Độc giả có thể đưa ra định nghĩa chính xác thế nào là biểu thức dấu ngoặc cân xứng. Để minh hoạ, ta xét biểu thức số học

(((a – b) * (5 + c) + x) / (x + y))

Loại bỏ đi tất cả các toán hạng và các dấu phép toán ta nhận được biểu thức dấu ngoặc:

```
( (  ( )  ( ) )  ( ) )
1 2 3 4  5 6 7 8 9 10
```

Biểu thức dấu ngoặc trên là cân xứng: các cặp dấu ngoặc tương ứng là 1 – 10, 2 – 7, 3 – 4, 5 – 6 và 8 – 9. Biểu thức dấu ngoặc (( ) ( ) là không cân xứng. Vấn đề đặt ra là làm thế nào để cho biết một biểu thức dấu ngoặc là cân xứng hay không cân xứng.

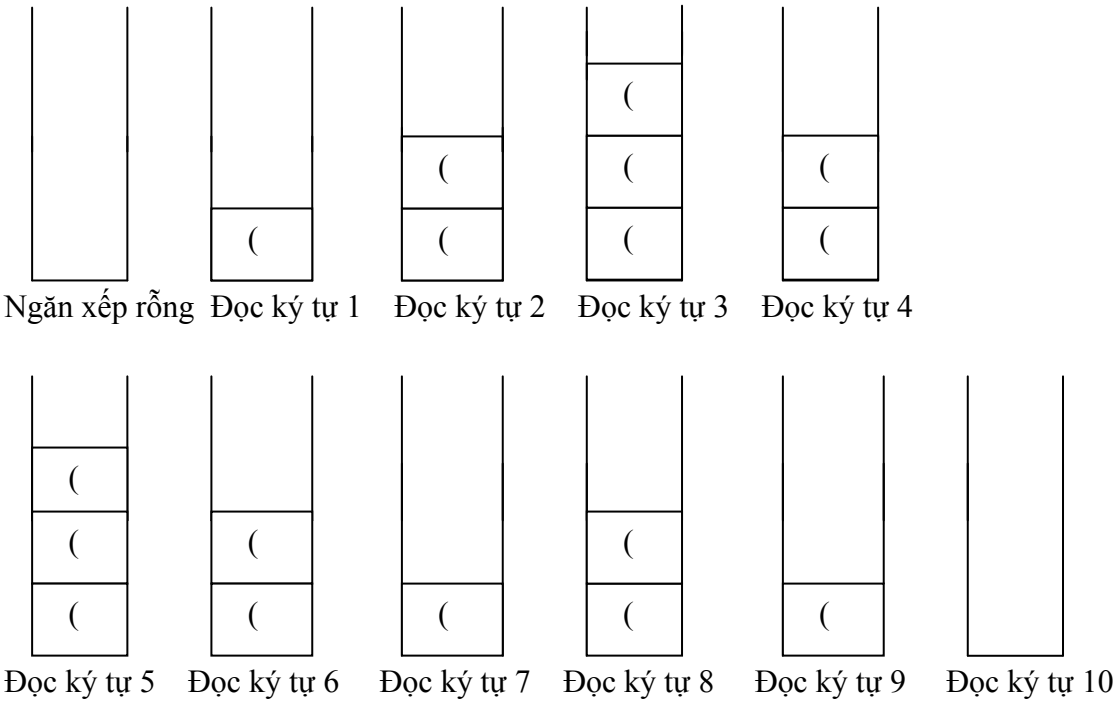
Sử dụng ngăn xếp, chúng ta dễ dàng thiết kế được thuật toán kiểm tra tính cân xứng của biểu thức dấu ngoặc. Một biểu thức dấu ngoặc được xem như một xâu ký tự được tạo thành từ hai ký tự mở ngoặc và đóng ngoặc. Ngăn xếp được sử dụng để lưu các dấu mở ngoặc. Thuật toán gồm các bước sau:

1. Khởi tạo một ngăn xếp rỗng.
2. Đọc lần lượt các ký tự trong biểu thức dấu ngoặc
    a. Nếu ký tự là dấu mở ngoặc thì đẩy nó vào ngăn xếp.
    b. Nếu ký tự là dấu đóng ngoặc thì:

- Nếu ngăn xếp rỗng thì thông báo biểu thức dấu ngoặc không cân xứng và dừng.
- Nếu ngăn xếp không rỗng thì loại dấu mở ngoặc ở đỉnh ngăn xếp.

3. Sau khi ký tự cuối cùng trong biểu thức dấu ngoặc đã được đọc, nếu ngăn xếp rỗng thì thông báo biểu thức dấu ngoặc cân xứng.

Để thấy được thuật toán trên làm việc như thế nào, ta xét biểu thức dấu ngoặc đã đưa ra ở trên: ((( ) ( )) ( )). Biểu thức dấu ngoặc này là một xâu gồm 10 ký tự. Ban đầu ngăn xếp rỗng. Đọc ký tự đầu tiên, nó là dấu mở ngoặc và được đẩy vào ngăn xếp. Ký tự thứ hai và ba cũng là dấu mở ngoặc, nên cũng được đẩy vào ngăn xếp, và như vậy đến đây ngăn xếp chứa ba dấu mở ngoặc. Ký tự thứ tư là dấu đóng ngoặc, do đó dấu mở ngoặc ở đỉnh ngăn xếp bị loại. Ký tự thứ năm là dấu mở ngoặc, nó lại được đẩy vào ngăn xếp. Tiếp tục, sau khi ký tự cuối cùng được đọc, ta thấy ngăn xếp rỗng, do đó biểu thức dấu ngoặc đã xét là cân xứng. Hình 6.4 minh hoạ các trạng thái của ngăn xếp tương ứng với mỗi ký tự được đọc.



Hình 6.4. Các trạng thái của ngăn xếp khi đọc biểu thức ((( ) ( )) ( ))

6.5 ĐÁNH GIÁ BIỂU THỨC SỐ HỌC

Trong các chương trình viết bằng ngôn ngữ bậc cao, chẳng hạn Pascal, C/ C + +, thường có mặt các biểu thức số học (hoặc logic). Khi chương trình được thực hiện, các biểu thức sẽ được đánh giá. Trong mục này chúng ta sẽ trình bày thuật toán tính giá trị của biểu thức, thuật toán này được cài đặt trong module thực hiện nhiệm vụ đánh giá biểu thức trong các chương trình dịch.

Biểu thức dạng postfix (biểu thức Balan). Để đơn giản cho trình bày, chúng ta giả thiết rằng biểu thức chỉ chứa các phép toán có hai toán hạng, chẳng hạn các phép toán +, -, *, /, … Trong các chương trình, các biểu thức được viết theo cách truyền thống như trong toán học, tức là ký hiệu phép toán đứng giữa hai toán hạng, chẳng hạn biểu thức:

(a + b) * c

Các biểu thức được viết theo cách truyền thống được gọi là **biểu thức dạng infix**. Trong các biểu thức dạng infix có thể có các dấu ngoặc, chẳng hạn biểu thức trên. Các dấu ngoặc được đưa vào biểu thức để thay đổi thứ tự tính toán. Các biểu thức còn có thể được viết dưới một dạng khác: **dạng postfix**. Trong các biểu thức dạng postfix (biểu thức Balan) ký hiệu phép toán đứng sau hai toán hạng, chẳng hạn dạng postfix của biểu thức (a + b) * c là biểu thức:

a b + c *

Cần lưu ý rằng, trong các biểu thức postfix không có các dấu ngoặc. Sau đây chúng ta sẽ thấy việc đánh giá các biểu thức postfix là rất dễ dàng. Vì vậy, việc đánh giá các biểu thức infix được thực hiện qua hai giai đoạn:

- Chuyển biểu thức infix thành biểu thức postfix.
- Đánh giá biểu thức postfix.

Trước hết chúng ta trình bày thuật toán đánh giá biểu thức postfix.

6.5.1 Đánh giá biểu thức postfix

Trong biểu thức postfix, các phép toán được thực hiện theo thứ tự mà chúng xuất hiện trong biểu thức kể từ trái sang phải. Chẳng hạn, xét biểu thức postfix:

5 8 3 1 - / 3 * + (1)

Phép toán được thực hiện đầu tiên là -, rồi đến /, tiếp theo là * và sau cùng là +. Mỗi phép toán được thực hiện với hai toán hạng đứng ngay trước nó. Chẳng hạn, phép – được thực hiện với các toán hạng là 1 và 3: 3 – 1 = 2. Sau khi thực hiện phép - biểu thức (1) trở thành 5 8 2 / 3 * +. Phép / được thực hiện với hai toán hạng đứng ngay trước nó là 2 và 8: 8 / 2 = 4. Tiếp tục, ta có 5 8 2 / 3 * + = 5 4 3 * + = 5 12 + = 17. Như vậy, mỗi khi thực hiện một phép toán trong biểu thức postfix chúng ta cần có khả năng tìm được các toán hạng của nó. Cách dễ nhất để làm được điều đó là sử dụng một ngăn xếp.

Chúng ta sẽ sử dụng một ngăn xếp S để lưu các toán hạng, ban đầu ngăn xếp rỗng.

Thuật toán đánh giá biểu thức postfix là như sau. Xem biểu thức postfix như một dãy các thành phần (mỗi thành phần hoặc là toán hạng, hoặc là ký hiệu phép toán). Đọc lần lượt các thành phần của biểu thức từ trái sang phải, với mỗi thành phần được đọc thực hiện các bước sau:

1. Nếu thành phần được đọc là toán hạng od thì đẩy nó vào ngăn xếp, tức là S. Push (od).
2. Nếu thành phần được đọc là phép toán op thì lấy ra các toán hạng ở đỉnh ngăn xếp:

   od 2 = S.Pop( )

   od 1 = S.Pop( )

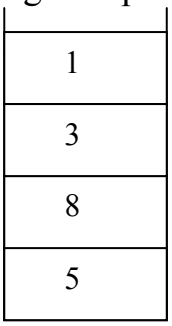
   Thực hiện phép toán op với các toán hạng là od 1 và od 2, kết quả được đẩy vào ngăn xếp:

   r = od 1 op od 2

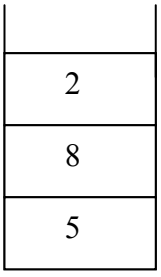
   S. Push(r)

   Lặp lại hai bước trên cho tới khi thành phần cuối cùng của biểu thức được đọc qua. Khi đó ngăn xếp chứa kết quả của biểu thức.

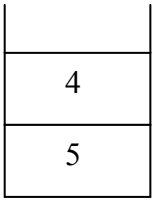
   Để thấy được thuật toán trên làm việc như thế nào, chúng ta lại xét biểu thức postfix (1). Khi bốn toán hạng đầu tiên được đọc, chúng được đẩy vào ngăn xếp và ngăn xếp có nội dung như sau:



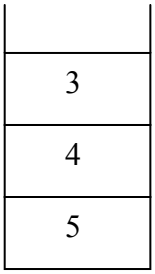
Tiếp theo, thành phần được đọc là phép " – ", hai toán hạng ở đỉnh của ngăn xếp là 1 và 3 được lấy ra khỏi ngăn xếp và thực hiện phép " – " với các toán hạng đó, ta thu được kết quả là 2, đẩy 2 vào ngăn xếp:



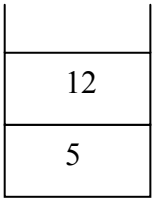
Thành phần được đọc tiếp theo là phép toán /, do đó các toán hạng 2 và 8 được lấy ra khỏi ngăn xếp, và thương của chúng là 4 lại được đẩy vào ngăn xếp:



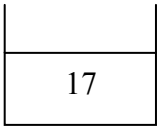
Toán hạng tiếp theo 3 được đẩy vào ngăn xếp:



Thành phần tiếp theo là phép x, do đó 3 và 4 được lấy ra khỏi ngăn xếp, và 4 * 3 = 12 được đẩy vào ngăn xếp:



Cuối cùng là phép + được đọc, nên 12 và 5 được lấy ra khỏi ngăn xếp và kết quả của biểu thức là 5 + 12 = 17 được đẩy vào ngăn xếp:



6.5.2 Chuyển biểu thức infix thành biểu thức postfix

Để chuyển biểu thức infix thành biểu thức postfix, trước hết chúng ta cần chú ý tới thứ tự thực hiện các phép toán trong biểu thức infix. Nhớ lại rằng, trong biểu thức infix, các phép toán * và / cần được thực hiện trước các phép toán + và - , chẳng hạn 1 + 2 * 3 = 1 + 6 = 7. Dó đó chúng ta nói rằng các phép toán * và / có quyền ưu tiên cao hơn các phép toán + và -, các phép toán * và / cùng mức ưu tiên, các phép toán + và – cùng mức ưu tiên. Mặt khác nếu các phép toán cùng mức ưu tiên đứng kề nhau, thì thứ tự thực hiện chúng là từ trái qua phải, chẳng hạn 8 – 2 + 3 = 6 + 3 = 9. Các cặp dấu ngoặc

được đưa vào biểu thức infix để thay đổi thứ tự thực hiện các phép toán, các phép toán trong cặp dấu ngoặc cần được thực hiện trước, chẳng hạn 8 – (2 + 3) = 8 – 5 = 3. Vì vậy vấn đề then chốt trong việc chuyển biểu thức infix thành biểu thức postfix là xác định được thứ tự thực hiện các phép toán trong biểu thức infix là đặt các phép toán đó vào đúng vị trí của chúng trong biểu thức postfix.

Ý tưởng của thuật toán chuyển biểu thức infix thành biểu thức postfix là như sau: Chúng ta xem biểu thức infix như dãy các thành phần (mỗi thành phần (mỗi thành phần là toán hạng hoặc ký hiệu phép toán hoặc dấu mở ngoặc hoặc dấu đóng ngoặc). Thuật toán có đầu vào là biểu thức infix đã cho và đầu ra là biểu thức postfix. Chúng ta đã sử dụng một ngăn xếp S để lưu các phép toán và dấu mở ngoặc. Đọc lần lượt từng thành phần của biểu thức infix từ trái sang phải, cứ mỗi lần gặp toán hạng, chúng ta viết nó vào biểu thức postfix. Khi đọc tới một phép toán (phép toán hiện thời), chúng ta xét các phép toán ở đỉnh ngăn xếp. Nếu các phép toán ở đỉnh ngăn xếp có quyền ưu tiên cao hơn hoặc bằng phép toán hiện thời, thì chúng được kéo ra khỏi ngăn xếp và được viết vào biểu thức postfix. Khi mà phép toán ở đỉnh ngăn xếp có quyền ưu tiên thấp hơn phép toán hiện thời, thì phép toán hiện thời được đẩy vào ngăn xếp. Bởi vì các phép toán trong cặp dấu ngoặc cần được thực hiện trước tiên, do đó chúng ta xem dấu mở ngoặc như phép toán có quyền ưu tiên cao hơn mọi phép toán khác. Vì vậy, mỗi khi gặp dấu mở ngoặc thì nó được đẩy vào ngăn xếp, và nó chỉ bị loại khỏi ngăn xếp khi ta đọc tới dấu đóng ngoặc tương ứng và tất cả các phép toán trong cặp dấu ngoặc đó đã được viết vào biểu thức postfix.

1. Nếu thành phần được đọc là toán hạng thì viết nó vào biểu thức postfix.
2. Nếu thành phần được đọc là phép toán (phép toán hiện thời), thì thực hiện các bước sau:

Nếu ngăn xếp không rỗng thì nếu phần tử ở đỉnh ngăn xếp là phép toán có quyền ưu tiên cao hơn hay bằng phép toán hiện thời, thì phép toán đó được kéo ra khỏi ngăn xếp và viết vào biểu thức postfix. Lặp lại bước này.

Nếu ngăn xếp rỗng hoặc phần tử ở đỉnh ngăn xếp là dấu mở ngoặc hoặc phép toán ở đỉnh ngăn xếp có quyền ưu tiên thấp hơn phép toán hiện thời, thì phép toán hiện thời được đẩy vào ngăn xếp.

3. Nếu thành phần được đọc là dấu mở ngoặc thì nó được đẩy vào ngăn xếp.
4. Nếu thành phần được đọc là dấu đóng ngoặc thì thực hiện các bước sau:

(Bước lặp) Loại các phép toán ở đỉnh ngăn xếp và viết vào biểu thức postfix cho tới khi đỉnh ngăn xếp là dấu mở ngoặc.

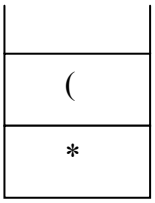
Loại dấu mở ngoặc khỏi ngăn xếp.

5. Sau khi toàn bộ biểu thức infix được đọc, loại các phép toán ở đỉnh ngăn xếp và viết vào biểu thức postfix cho tới khi ngăn xếp rỗng.

Ví dụ. Xét biểu thức infix:

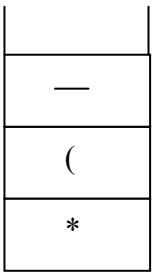
a * (b – c + d) + e / f

Đầu tiên ký hiệu được đọc là toán hạng a, nó được viết vào biểu thức postfix. Tiếp theo ký hiệu phép toán * được đọc, và ngăn xếp rỗng, nên nó được đẩy vào ngăn xếp. Đọc thành phần tiếp theo, đó là dấu mở ngoặc nên nó được đẩy vào ngăn xếp. Tới đây ta có trạng thái ngăn xếp và biểu thức postfix như sau:



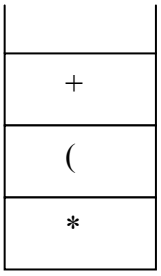
Biểu thức postfix: a

Đọc tiếp toán hạng b, nó được viết vào biểu thức postfix. Thành phần tiếp theo là phép -, lúc này đỉnh ngăn xếp là dấu mở ngoặc, nên ký hiệu - được đẩy vào ngăn xếp. Ta có:



Biểu thức postfix: a b

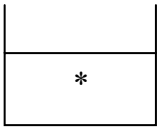
Thành phần được đọc tiếp theo là toán hạng c, nó được viết vào biểu thức postfix. Ký hiệu được đọc tiếp là phép +. Phép toán ở đỉnh ngăn xếp là phép – cùng quyền ưu tiên với phép +, nên nó được kéo ra khỏi ngăn xếp và viết vào đầu ra. Lúc này, đỉnh ngăn xếp là dấu mở ngoặc, nên phép + được đẩy vào ngăn xếp và ta có:



Biểu thức postfix
a b c -

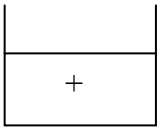
Đọc đến toán hạng d, nó được viết vào biểu thức postfix. Đọc tiếp ký hiệu tiếp theo: dấu đóng ngoặc. Lúc này ta cần loại lần lượt các phép toán ở đỉnh

ngăn xếp và viết chúng vào biểu thức postfix, cho tới khi đỉnh ngăn xếp là dấu mở ngoặc thì loại nó. Ta có:



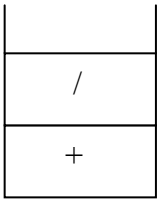
Biểu thức postfix
a b c – d +

Ký hiệu được đọc tiếp theo là phép +. Đỉnh ngăn xếp là phép *, nó có quyền ưu tiên cao hơn phép +, do đó nó được loại khỏi ngăn xếp và viết vào biểu thức postfix. Đến đây ngăn xếp rỗng, nên phép toán hiện thời + được đẩy vào ngăn xếp và ta có:



Biểu thức postfix
a b c – d + *

Đọc tiếp toán hạng e, nó được viết vào đầu ra. Ký hiệu được đọc tiếp là phép chia / , nó có quyền ưu tiên cao hơn phép + ở đỉnh ngăn xếp, nên phép / được đẩy vào ngăn xếp. Ký hiệu cuối cùng được đọc là toán hạng f, nó được viết vào đầu ra, ta có:



Biểu thức postfx
a b c – d + * e f

Loại các phép toán ở đỉnh ngăn xếp và viết vào đầu ra cho tới khi ngăn xếp rỗng, chúng ta nhận được biểu thức postfix kết quả là:

a b c – d + * e f / +

6.6 NGĂN XẾP VÀ ĐỆ QUY

Một trong các ứng dụng quan trọng của ngăn xếp là sử dụng ngăn xếp để chuyển thuật toán đệ quy thành thuật toán không đệ quy. Với nhiều vấn đề, có thể tồn tại cả thuật toán đệ quy và thuật toán không đệ quy để giải quyết. Trong nhiều trường hợp, thuật toán đệ quy kém hiệu quả cả về thời gian và bộ nhớ. Chúng ta sẽ nghiên cứu kỹ hơn các thuật toán đệ quy trong mục 16.2. Việc loại bỏ các lời gọi đệ quy trong các hàm đệ quy có thể cải thiện đáng kể thời gian chạy và bộ nhớ đòi hỏi. Trong mục này chúng ta sẽ trình bày các kỹ thuật sử dụng ngăn xếp để chuyển đổi các hàm đệ quy thành hàm không đệ quy.

Trước hết cần biết răng, trong một hàm đệ quy thì lời gọi đệ quy ở dòng cuối cùng của hàm được gọi là lời gọi **đệ quy đuôi** (tail recursion). Lời

gọi đệ quy đuôi dễ dàng được thay thế bởi một vòng lặp, mà không cần sử dụng ngăn xếp. Chẳng hạn, xét hàm đệ quy sắp xếp mảng QuickSort, hàm này sẽ được nghiên cứu trong mục 17.3. Thuật toán sắp xếp nhanh được thiết kế theo chiến lược chia - để - trị: để sắp xếp mảng A[a..b] theo thứ tự tăng dần, chúng ta phân hoạch đoạn [a ..b] thành hai đoạn con bởi một chỉ số k, a ≤ k ≤ b, sao cho với mọi chỉ số i trong đoạn con dưới [a, k - 1] và với mọi chỉ số j trong đoạn con trên [k + 1….b] ta có A[i] ≤ A[k] ≤ A[j]. Nếu thực hiện được sự phân hoạch như thế, thì việc sắp xếp mảng A[a…b] được quy về việc sắp hai mảng con A[a... k - 1] và A[k+1 …b]. Do đó, hàm đệ quy QuickSort như sau:

```
void     QuickSort(item  A[ ], int a, int b)
{
  if (a < b)
  {
    k = Partition(A, a, b); // hàm phân hoạch
    QuickSort(A, a, k – 1) ;
    QuickSort(A, k+1, b); // lời gọi đệ quy đuôi
  }
}
```

Thực hiện lời gọi đệ quy đuôi (ở dòng lệnh cuối cùng) có nghĩa là chúng ta cần thực hiện lặp lại các dòng lệnh từ đầu hàm đến dòng lệnh đứng trên kề lời gọi đệ quy đuôi. Vì vậy đưa dãy các lệnh đó vào một vòng lặp, chúng ta loại bỏ được lời gọi đệ quy đuôi. Hàm QuickSort có thể viết lại như sau:

```
void    QuickSort(item  A[  ], int a, int b)
{
  k = Partition(A, a, b);
  while (a < b)
  {
    QuickSort(A, a, k – 1);
    a = k + 1;
    k = b + 1;
  }
}
```

Việc loại bỏ các lời gọi đệ quy không phải là đuôi không đơn giản như đệ quy đuôi. Chúng ta cần sử dụng ngăn xếp để lưu lại các đối số trong các lời gọi đệ quy. Phân tích sự thực hiện hàm đệ quy QuickSort, chúng ta sẽ thấy quá trình thực hiện hàm sẽ diễn ra như sau: phân hoạch đoạn [a …b] bởi chỉ số k, rồi phân hoạch đoạn con dưới [a ..k - 1] bởi chỉ số k_1, rồi lại phân hoạch đoạn con dưới [a .. k_1 - 1]… cho tới khi đoạn con dưới chỉ gồm một

chỉ số, lúc đó ta mới phân hoạch đoạn con trên ứng với đoạn con dưới đó. Việc phân hoạch một đoạn bất kỳ lại diễn ra như trên. Vì vậy, chúng ta sẽ sử dụng một ngăn xếp để lưu các chỉ số đầu và chỉ số cuối của các đoạn con trên khi phân hoạch. Cụ thể là ngăn xếp sẽ lưu k + 1, b, rồi k_1 + 1, k – 1, … Hàm đệ quy QuickSort được chuyển thành hàm không đệ quy sau:

```
void QuickSort2 (item  A[  ], int a, int b)
{
  Stack S;
  S.Push(a) ;
  S.Push(b) ;
  do
      b = S.Pop( ) ;
      a = S.Pop( ) ;
      while (a < b)
          {
             k = Partition(A, a, b) ;
             S.Push(k + 1) ;
             S.Push(b) ;
             b = k – 1 ;
           }
  while (! S.Empty( ) ) ;
}
```

Sau này chúng ta sẽ thấy, ngăn xếp được sử dụng để thiết kế các hàm không đệ quy thực hiện các phép toán trên cây.

BÀI TẬP

1. Hãy cài đặt lớp Stack bằng cách sử dụng lớp Dlist (hoặc lớp Llist) làm lớp cơ sở với dạng thừ kế private.

2. Cho ngăn xếp S chứa các phần tử. Sử dụng ngăn xếp T rỗng, hãy đưa ra thuật toán (chỉ được sử dụng các phép toán ngăn xếp) thực hiện các nhiệm vụ sau:
   a. Đếm số phần tử trong ngăn xếp S, ngăn xếp S không thay đổi.
   b. Loại bỏ tất cả các phần tử trong ngăn xếp S bằng một phần tử cho trước, thứ tự các phần tử còn lại trong S không thay đổi.

3. Giả sử biểu thức dấu ngoặc chứa ba loại dấu ngoặc ( ), [ ], { }. Biểu thức [( ) ( )] { } được xem là cân xứng, còn biểu thức {[( ]) ( ) là không cân xứng. Hãy đưa ra định nghĩa biểu thức dấu ngoặc cân xứng

và đưa ra thuật toán cho biết biểu thức dấu ngoặc có cân xứng hay không.

4. Áp dụng thuật toán chuyển biểu thức dạng infix thành biểu thức dạng postfix (xem mục 6.5.2), hãy chuyển các biểu thức infix sau thành biểu thức postfix, cần chỉ ra nội dung của ngăn xếp sau mỗi bước của thuật toán.
   a. A / B /C – (D + E) * F
   b. A – (B + C * D) / E
   c. A * (B / C / D) + E

5. Thiết kế thuật toán đoán nhận các xâu ký tự có dạng w $ w', trong đó w' là đảo ngược của xâu w, chẳng hạn nếu w = a c d b thì w' = b d c a.

6. Cho đỉnh A (đỉnh xuất phát) và đỉnh B (đỉnh đích) trong đồ thị định hướng G (đồ thị có thể có chu trình). Sử dụng ngăn xếp, hãy thiết kế thuật toán tìm đường đi từ A đến B. (Sử dụng ngăn xếp để lưu vết của đường đi từ A đến B).

7. Sử dụng các ngăn xếp, hãy đưa ra thuật toán không đệ quy cho bài toán tháp Hà Nội.

CHƯƠNG 7

HÀNG ĐỢI

Cũng như ngăn xếp, hàng đợi là CTDL tuyến tính. Hàng đợi là một danh sách các đối tượng, một đầu của danh sách được xem là đầu hàng đợi, còn đầu kia của danh sách được xem là đuôi hàng đợi. Với hàng đợi, chúng ta chỉ có thể xen một đối tượng mới vào đuôi hàng và loại đối tượng ở đầu hàng ra khỏi hàng. Trong chương này chúng ta sẽ nghiên cứu các phương pháp cài đặt hàng đợi và trình bày một số ứng dụng của hàng đợi.

7.1 KIỂU DỮ LIỆU TRỪU TƯỢNG HÀNG ĐỢI

Trong mục này chúng ta sẽ đặc tả KDLTT hàng đợi. Chúng ta có thể xem một hàng người đứng xếp hàng chờ được phục vụ (chẳng hạn, xếp hàng chờ mua vé tàu, xếp hàng chờ giao dịch ở ngân hàng, …) là một hàng đợi, bởi vì người ra khỏi hàng và được phục vụ là người đứng ở đầu hàng, còn người mới đến sẽ đứng vào đuôi hàng.

Hàng đợi là một danh sách các đối tượng dữ liệu, một trong hai đầu danh sách được xem là đầu hàng, còn đầu kia là đuôi hàng. Chẳng hạn, hàng đợi có thể là danh sách các ký tự (a, b, c, d), trong đó a đứng ở đầu hàng, còn d đứng ở đuôi hàng. Chúng ta có thể thực hiện các phép toán sau đây trên hàng đợi, trong các phép toán đó Q là một hàng đợi, còn x là một đối tượng cùng kiểu với các đối tượng trong hàng Q.

1. Empty(Q). Hàm trả về true nếu hàng Q rỗng và false nếu không.
2. Enqueue(x, Q). Thêm đối tượng x vào đuôi hàng Q.
3. Dequeue(Q). Loại đối tượng đứng ở đầu hàng Q.
4. GetHead(Q). Hàm trả về đối tượng đứng ở đầu hàng Q, còn hàng Q thì không thay đổi.

Ví dụ. Nếu Q = (a, b, c, d) và a ở đầu hàng, d ở đuôi hàng, thì khi thực hiện phép toán Enqueue(e, Q) ta nhận được Q = (a, b, c, d, e), với e đứng ở đuôi hàng, nếu sau đó thực hiện phép toán Dequeue(Q), ta sẽ có Q = (b, c, d, e) và b trở thành phần tử đứng ở đầu hàng.

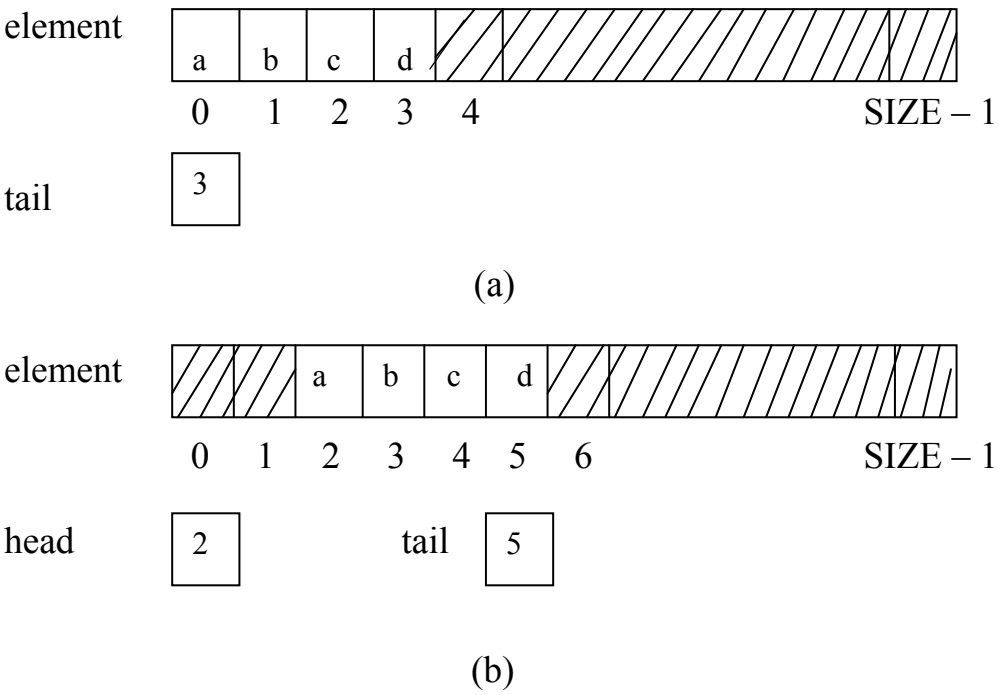
Với các phép toán Enqueue và Dequeue xác định như trên thì đối tượng vào hàng trước sẽ ra khỏi hàng trước. Vì lý do đó mà hàng đợi được gọi là cấu trúc dữ liệu FIFO (viết tắt của cụm từ First- In First- Out). Điều này đối lập với ngăn xếp, trong ngăn xếp đối tượng ra khỏi ngăn xếp là đối tượng sau cùng được đặt vào ngăn xếp.

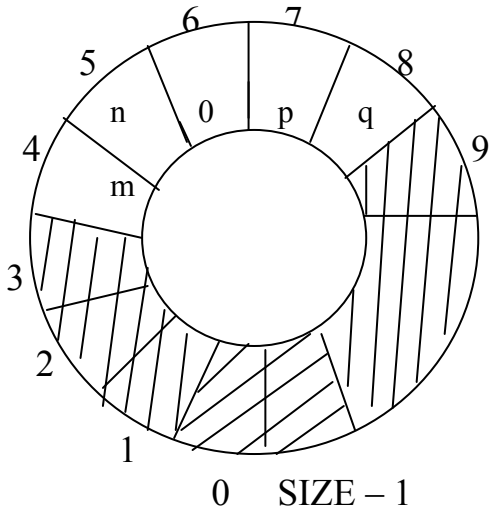
Hàng đợi sẽ được sử dụng trong bất kỳ hoàn cảnh nào mà chúng ta cần xử lý các đối tượng theo trình tự FIFO. Cuối chương này chúng ta sẽ trình bày một ứng dụng của hàng đợi trong mô phỏng một hệ phục vụ. Nhưng

trước hết chúng ta cần nghiên cứu các phương pháp cài đặt hàng đợi. Cũng như ngăn xếp, chúng ta có thể cài đặt hàng đợi bởi mảng hoặc bởi DSLK.

7.2 CÀI ĐẶT HÀNG ĐỢI BỞI MẢNG

Cũng như ngăn xếp, chúng ta có thể cài đặt hàng đợi bởi mảng. Song cài đặt hàng đợi bởi mảng sẽ phức tạp hơn ngăn xếp. Nhớ lại rằng, khi cài đặt danh sách (hoặc ngăn xếp) bởi mảng thì các phần tử của danh sách (hoặc ngăn xếp) sẽ được lưu trong đoạn đầu của mảng, còn đoạn sau của mảng là không gian chưa sử dụng đến. Chúng ta có thể làm như thế với hàng đợi được không? Câu trả lời là có, nhưng không hiệu quả. Giả sử chúng ta sử dụng mảng element để lưu các phần tử của hàng đợi, các phần tử của hàng đợi được lưu trong các thành phần mảng element[0], element[1], … , element[k] như trong hình 7.1a. Với cách này, phần tử ở đầu hàng luôn luôn được lưu trong thành phần mảng element[0], còn phần tử ở đuôi hàng được lưu trong element[k], và do đó ngoài mảng element ta chỉ cần một biến tail ghi lại chỉ số k. Để thêm phần tử mới vào đuôi hàng, ta chỉ cần tăng chỉ số tail lên 1 và đặt phần tử mới vào thành phần mảng element[tail]. Song nếu muốn loại phần tử ở đầu hàng, chúng ta cần chuyển lên trên một vị trí tất cả các phần tử được lưu trong element[1],…, element[tail] và giảm chỉ số tail đi 1, và như vậy tiêu tốn nhiều thời gian.





head | 4 |

tail | 8 |

(c)

Hình 7.1. Các phương pháp cài đặt hàng đợi bởi mảng

Để khắc phục nhược điểm của phương pháp trên, chúng ta lưu các phần tử của hàng đợi trong một đoạn con của mảng từ chỉ số đầu head tới chỉ số đuôi tail, tức là các phần tử của hàng đợi được lưu trong các thành phần mảng element[head], element[head + 1], … , element[tail], như trong hình 7.1b. Với cách cài đặt này, ngoài mảng element, chúng ta cần sử dụng hai biến: biến chỉ số đầu head và biến chỉ số đuôi tail. Để loại phần tử ở đầu hàng chúng ta chỉ cần tăng chỉ số head lên 1. Chúng ta có nhận xét rằng, khi thêm phần tử mới vào đuôi hàng thì chỉ số tail tăng, khi loại phần tử ở đầu hàng thì chỉ số head tăng, tới một thời điểm nào đó hàng đợi sẽ chiếm đoạn cuối của mảng, tức là biến tail nhận giá trị SIZE – 1 (SIZE là cỡ của mảng). Lúc đó ta không thể thêm phần tử mới vào đuôi hàng, mặc dầu không gian chưa sử dụng trong mảng, đoạn đầu của mảng từ chỉ số 0 tới head – 1, có thể còn rất lớn! Để giải quyết vấn đề này, chúng ta "cuộn" mảng thành vòng tròn, như được minh hoạ trong hình 7.1c. Trong **mảng vòng tròn**, chúng ta xem thành phần tiếp theo thành phần element[SIZE - 1] là thành phần element[0]. Với mảng vòng tròn, khi thêm phần tử mới vào đuôi hàng, nếu chỉ số tail có giá trị là SIZE – 1, ta đặt tail = 0 và lưu phần tử mới trong thành phần mảng element[0]. Tương tự khi phần tử ở đầu hàng được lưu

trong element[SIZE - 1], thì để loại nó, ta chỉ cần đặt head = 0. Do đó, nếu cài đặt hàng đợi bởi mảng vòng tròn, thì phép toán thêm phần tử mới vào đuôi hàng không thực hiện được chỉ khi mảng thực sự đầy, tức là khi trong mảng không còn thành phần nào chưa sử dụng.

Sau đây chúng ta sẽ nghiên cứu sự cài đặt hàng đợi bởi mảng vòng tròn. KDLTT hàng đợi sẽ được cài đặt bởi lớp Queue, đây là lớp khuôn phụ thuộc tham biến kiểu Item, trong đó Item là kiểu của các phần tử trong hàng đợi. Lớp Queue chứa các biến thành phần nào? Các phần tử của hàng đợi được lưu trong mảng vòng tròn được cấp phát động, do đó cần có biến con trỏ element trỏ tới thành phần đầu tiên của mảng đó, biến size lưu cỡ của mảng, biến chỉ số đầu head và biến chỉ số đuôi tail. Ngoài các biến trên, chúng ta thêm vào một biến length để lưu độ dài của hàng đợi (tức là số phần tử trong hàng đợi). Lớp Queue được định nghĩa như trong hình 7.2.

```
template <class Item>
class     Queue
{
  public :
      Queue (int m = 1);
      // Hàm kiến tạo hàng đợi rỗng với dung lượng là m,
      // m nguyên dương (tức là cỡ mảng động là m).
      Queue (const Queue & Q) ;
      // Hàm kiến tạo copy.
      ~ Queue( ) // Hàm huỷ.
      { delete [ ] element ; }
      Queue & operator = (const Queue & Q); // Toán tử gán.
      // Các hàm thực hiện các phép toán hàng đợi :
      bool    Empty( ) const ;
      // Kiểm tra hàng có rỗng không.
      // Postcondition: hàm trả về true nếu hàng rỗng và false nếu không rỗng
      { return length = = 0 ; }
      void    Enqueue (const Item & x)
      // Thêm phần tử mới x vào đuôi hàng.
      // Postcondition: x là phần tử ở đuôi hàng.
      Item &   Dequeue( );
      // Loại phần tử ở đầu hàng.
      // Precondition: hàng không rỗng.
      // Postcondition: phần tử ở đầu hàng bị loại khỏi hàng và hàm trả về
      // phần tử đó.
      Item &   GetHead( ) const ;
      // Precondition: hàng không rỗng.
      // Postcondition: hàm trả về phần tử ở đầu hàng, nhưng hàng vẫn
```

```
    // giữ nguyên.
  private:
    Item *   element ;
    int    size ;
    int    head ;
    int    tail ;
    int    length ;
} ;
```

Hình 7.2. Lớp hàng đợi được cài đặt bởi mảng.

Bây giờ chúng ta nghiên cứu sự cài đặt các hàm thành phần của lớp Queue.

Hàm kiến tạo hàng rỗng với sức chứa là m. Chúng ta cần cấp phát một mảng động element có cỡ là m. Vì hàng rỗng, nên giá trị của biến length là 0. Các chỉ số đầu head và chỉ số đuôi tail cần có giá trị nào? Nhớ lại rằng khi cần thêm phần tử x vào đuôi hàng, ta tăng biến tail lên 1 và đặt element[tail] = x. Để cho thao tác này làm việc trong mọi hoàn cảnh, kể cả trường hợp hàng rỗng, chúng ta đặt head = 0 và tail = -1 khi kiến tạo hàng rỗng. Do đó, hàm kiến tạo được cài đặt như sau:

```
template <class Item>
Queue<Item> :: Queue (int m)
{
   element = new Item[m] ;
   size = m ;
   head = 0 ;
   tail = -1 ;
   length = 0 ;
}
```

Hàm kiến tạo copy. Hàm này thực hiện nhiệm vụ kiến tạo ra một hàng đợi mới là bản sao của một hàng đợi đã có Q. Do đó chúng ta cần cấp phát một mảng động mới có cỡ bằng cỡ của mảng trong Q và tiến hành sao chép các dữ liệu từ đối tượng Q sang đối tượng mới.

Hàm kiến tạo copy được viết như sau:

```
template <class Item>
Queue<Item> :: Queue (const Queue<Item> & Q)
{
   element = new Item[Q.size] ;
   size = Q.size ;
   head = Q.head ;
```

```
        tail = Q.tail ;
        length = Q.length ;
        for (int i = 0 ; i < length ; i + + )
            element [(head + i) % size] = Q.element [(head + i) % size] ;
    }
```

Trong hàm trên, vòng lặp for thực hiện sao chép mảng trong Q sang mảng mới kể từ chỉ số head tới chỉ số tail, tức là sao chép length thành phần mảng kể từ chỉ số head. Nhưng cần lưu ý rằng, trong mảng vòng tròn, thành phần tiếp theo thành phần với chỉ số k, trong các trường hợp k ≠ size – 1, là thành phần với chỉ số k + 1. Song nếu k = size – 1 thì thành phần tiếp theo có chỉ số là 0. Vì vậy, trong mọi trường hợp thành phần tiếp theo thành phần với chỉ số k là thành phần với chỉ số (k + 1) % size.

Toán tử gán được cài đặt tương tự như hàm kiến tạo copy:

```
template <class Item>
Queue<Item> & Queue<Item> :: operator = (const Queue<Item> & Q)
{
   if ( this != Q )
      {
        delete [  ] element ;
        element = new Item[Q.size] ;
        size = Q.size ;
        head = Q.head ;
        tail = Q.tail ;
        length = Q.length ;
        for (int i = 0 ; i < length ; i + + )
            element [(head + i) % size] = Q.element[(head + i) % size] ;
      }
   return    * this ;
}
```

Hàm xen phần tử mới vào đuôi hàng được cài đặt bằng cách đặt phần tử mới vào mảng vòng tròn tại thành phần đứng ngay sau thành phần element[tail], nếu mảng chưa đầy. Nếu mảng đầy thì chúng ta cấp phát một mảng mới với cỡ lớn hơn (chẳng hạn, với cỡ gấp đôi cỡ mảng cũ), và sao chép dữ liệu từ mảng cũ sang mảng mới, rồi huỷ mảng cũ, sau đó đặt phần tử mới vào mảng mới.

```
template <class Item>
void    Queue<Item> :: Enqueue (const Item & x)
{
  if (length <size) // mảng chưa đầy.
```

```
    {
      tail = (tail + 1) % size ;
      element[tail] = x ;
    }
  else  // mảng đầy
    {
      Item * array = new Item[2 * size] ;
      for (int i = 0 ; i < size ; i + +)
      array[i] = element[(head + i) % size] ;
      array[size] = x ;
      delete [  ] element ;
      element = array ;
      head = 0 ;
      tail = size ;
      size = 2 * size ;
    }
  length + + ;
}
```

Các hàm loại phần tử ở đầu hàng và truy cập phần tử ở đầu hàng được cài đặt rất đơn giản như sau:

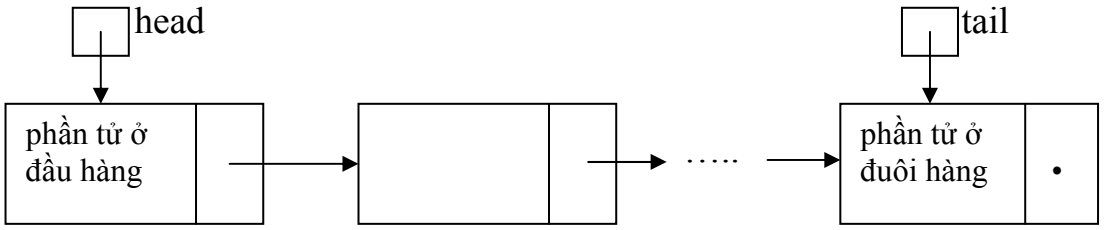
```
template <class Item>
Item &    Queue<Item> :: Dequeue(  )
{
   assert (length > 0) ; // Kiểm tra hàng không rỗng.
   Item   data = element[head] ;
   if (length = = 1) // Hàng có một phần tử.
     { head = 0 ; tail = -1 ; }
   else    head = (head + 1) % size ;
   length - - ;
   return   data ;
}

template  <class Item>
Item &   Queue<Item> :: GetHead( )
{
  assert (length > 0) ;
  return   element[head] ;
}
```

Dễ dàng thấy rằng, khi cài đặt hàng đợi bởi mảng vòng tròn thì các phép toán hàng đợi: xen phần tử mới vào đuôi hàng, loại phần tử ở đầu hàng

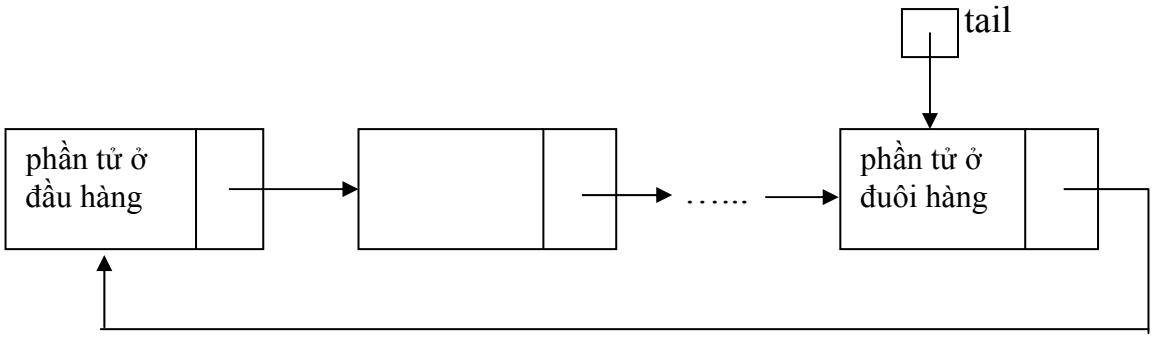
và truy cập phần tử ở đầu hàng chỉ đòi hỏi thời gian O(1). Chỉ trừ trường hợp mảng đầy, nếu mảng đầy thì để xen phần tử mới vào đuôi hàng chúng ta mất thời gian sao chép dữ liệu từ mảng cũ sang mảng mới với cỡ lớn hơn, do đó thời gian thực hiện phép toán thêm vào đuôi hàng trong trường hợp này là O(n), n là số phần tử trong hàng. Tuy nhiên trong các ứng dụng, nếu ta đánh giá được số tối đa các phần tử ở trong hàng và lựa chọn số đó làm dung lượng m của hàng đợi khi khởi tạo hàng đợi, thì có thể đảm bảo rằng tất cả các phép toán hàng đợi chỉ cần thời gian O(1).

7.3 CÀI ĐẶT HÀNG ĐỢI BỞI DSLK

Cũng như ngăn xếp, chúng ta có thể cài đặt hàng đợi bởi DSLK. Với ngăn xếp, chúng ta chỉ cần truy cập tới phần tử ở đỉnh ngăn xếp, nên chỉ cần một con trỏ ngoài top trỏ tới đầu DSLK (xem hình 6.3). Nhưng với hàng đợi, chúng ta cần phải truy cập tới cả phần tử ở đầu hàng và phần tử ở đuôi hàng, vì vậy chúng ta cần sử dụng hai con trỏ ngoài: con trỏ head trỏ tới thành phần đầu DSLK, tại đó lưu phần tử ở đầu hàng, và con trỏ tail trỏ tới thành phần cuối cùng của DSLK, tại đó lưu phần tử ở đuôi hàng, như trong hình 7.3a. Một cách tiếp cận khác, chúng ta có thể cài đặt hàng đợi bởi DSLK vòng tròn với một con trỏ ngoài tail như trong hình 7.3b.



(a)



(b)

Hình 7.3. (a) Cài đặt hàng đợi bởi DSLK với hai con trỏ ngoài.
(b) Cài đặt hàng đợi bởi DSLK vòng tròn.

Trong mục 5.4, chúng ta đã nghiên cứu cách cài đặt danh sách bởi DSLK, ở đó KDLTT danh sách đã được cài đặt bởi lớp LList. Các phép toán hàng đợi chỉ là các trường hợp riêng của các phép toán danh sách: xen phần tử mới vào đuôi hàng có nghĩa là xen nó vào đuôi danh sách, còn loại phần tử ở đầu hàng là loại phần tử ở vị trí đầu tiên trong danh sách. Do đó chúng ta có thể sử dụng lớp LList (xem hình 5.12) để cài đặt lớp hàng đợi Queue. Chúng ta có thể xây dựng lớp Queue bằng cách sử dụng lớp LList làm lớp cơ sở với dạng thừa kế private, hoặc cũng có thể xây dựng lớp Queue là lớp chỉ chứa một thành phần dữ liệu là đối tượng của lớp LList. Độc giả nên cài đặt lớp Queue theo các phương án trên, xem như bài tập. Các phép toán hàng đợi là rất đơn giản, nên chúng ta sẽ cài đặt lớp Queue trực tiếp, không thông qua lớp LList.

Sau đây chúng ta sẽ cài đặt hàng đợi bởi DSLK vòng tròn với con trỏ ngoài tail (hình 7.3b). KDLTT hàng đợi được cài đặt bởi lớp khuôn phụ thuộc tham biến kiểu Item (Item là kiểu của các phần tử trong hàng đợi). Lớp Queue này được định nghĩa trong hình 7.4. Lớp này chứa các hàm thành phần được khai báo hoàn toàn giống như lớp trong hình 7.2, chỉ trừ hàm kiến tạo mặc định làm nhiệm vụ khởi tạo hàng rỗng. Lớp chỉ chứa một thành phần dữ liệu là con trỏ tail.

```
template <class Item>
class    Queue
{
  public :
     Queue( ) // Hàm kiến tạo hàng đợi rỗng.
     { tail = NULL ; }
     Queue (const Queue & Q) ;
     ~ Queue( ) ;
     Queue &   operator = (const Queue & Q) ;
     bool    Empty( ) const
     {return tail = = NULL ; }
     void    Enqueue (const Item & x) ;
     Item &   Dequeue( )
     Item &   GetHead( ) const ;
  private :
     struct    Node
     {
         Item   data ;
```

```
        Node *   next ;
        Node (const Item & x)
        {data = x; next = NULL ; }
    }
    Node *   tail ;
    void    MakeEmpty( ) ; // Hàm huỷ DSLK vòng tròn tail.
} ;
```

Hình 7.4. Lớp hàng đợi được cài đặt bởi DSLK vòng tròn.

Bây giờ chúng ta cài đặt các hàm thành phần của lớp Queue trong hình 7.4. Chúng ta đã đưa vào lớp Queue hàm MakeEmpty, nhiệm vụ của nó là thu hồi vùng nhớ đã cấp phát cho các thành phần của DSLK tail làm cho DSLK này trở thành rỗng. Hàm MakeEmpty là hàm thành phần private, nó được sử dụng để cài đặt hàm huỷ và toán tử gán. Hàm MakeEmpty được cài đặt như sau:

```
template <class Item>
void    Queue<Item> :: MakeEmpty( )
{
  while (tail ! = NULL)
    {
       Node*  Ptr = tail →next ;
       if  (Ptr = = tail) // DSLK chỉ có một thành phần.
                tail = NULL ;
       else
                tail → next = Ptr → next ;
       delete   Ptr ;
     }
}
```

Hàm huỷ được cài đặt bằng cách gọi hàm MakeEmpty:

```
template <class Item>
Queue <Item> :: ~ Queue( )
{
   MakeEmpty( ) ;
}
```

Hàm kiến tạo copy.

```
template <class Item>
```

```
Queue <Item> :: Queue (const Queue<Item> & Q)
{
  tail = NULL ;
  * this = Q ;
}
```

Toán tử gán.

```
template <class Item>
Queue<Item> & Queue<Item>:: operator =(const Queue <Item> & Q)
{
  if (this ! = & Q)
  {
      MakeEmpty( ) ;
      if (Q.Empty( ))
      return *this ;
      Node * Ptr = Q.tail → next ;
      do {
            Enqueue (Ptr → data) ;
            Ptr = Ptr → next ;
          }
      while (Ptr ! = Q.tail → next) ;
  }
  return    *this ;
}
```

Hàm thêm phần tử mới vào đuôi hàng:

```
template <class Item>
void    Queue<Item> :: Enqueue (const Item & x)
{
  if (Empty( ))
        {
           tail = new Node(x) ;
           tail → next = tail ;
        }
  else {
          Node *  Ptr = tail → next ;
          tail = tail →next = new Node(x) ;
          tail → next = Ptr ;
        }
}
```

Hàm loại phần tử ở đầu hàng:

```
template <class Item>
Item &   Queue<Item> :: Dequeue( )
{
   Item   headElement = GetHead( ) ;
   Node *   Ptr = tail → next ;
   if (Ptr ! = tail)
          tail → next = Ptr → next ;
   else    tail = NULL ;
   delete   Ptr ;
   return   headElement ;
}
```

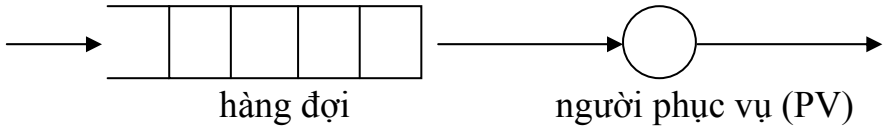
Hàm tìm phần tử ở đầu hàng:

```
template <class Item>
Item &   Queue<Item> :: GetHead( )
{
   assert (tail ! = NULL) ;
   return    tail → next → data ;
}
```

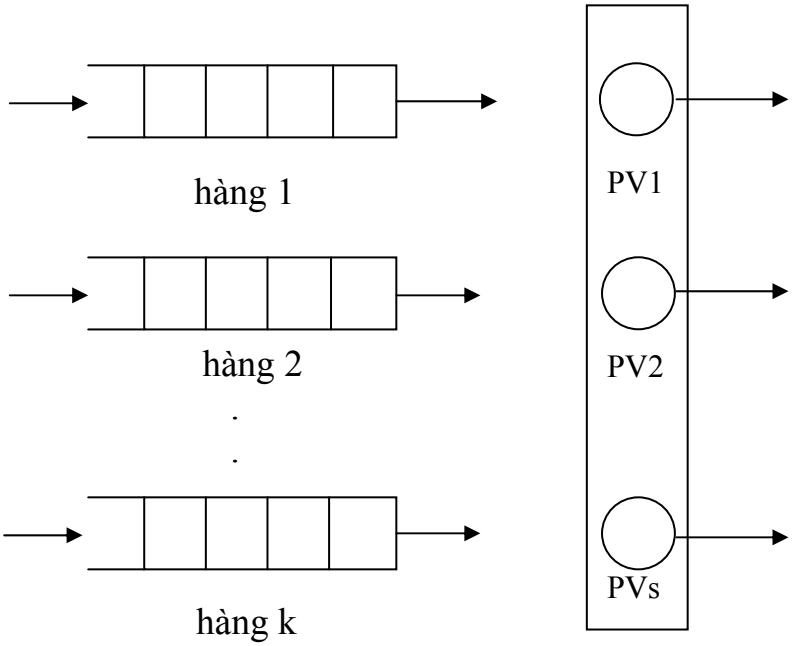
7.4 MÔ PHỎNG HỆ SẮP HÀNG

Mô phỏng (simulation) là một trong các lĩnh vực áp dụng quan trọng của máy tính. Mục đích của một chương trình mô phỏng là mô hình hoá sự hoạt động của một hệ hiện thực (hệ tồn tại trong tự nhiên hoặc hệ do con người sáng tạo ra) nhằm phân tích đánh giá hiệu năng của hệ hoặc đưa ra các tiên đoán để có thể cải tiến (đối với các hệ do con người làm ra) hoặc có thể đưa ra các biện pháp phòng ngừa, chế ngự (đối với các hệ tồn tại trong tự nhiên). Chúng ta có thể quan niệm một hệ hiện thực bao gồm các thực thể (các thành phần) phụ thuộc lẫn nhau, chúng hoạt động và tương tác với nhau để thực hiện một nhiệm vụ nào đó. Vì vậy, lập chương trình định hướng đối tượng là cách tiếp cận thích hợp nhất để xây dựng các chương trình mô phỏng. Chúng ta có thể biểu diễn mỗi thành phần trong hệ bởi một lớp đối tượng chứa các biến mô tả trạng thái của thực thể, sự tương tác giữa các thành phần của hệ được mô phỏng bởi sự truyền thông báo giữa các đối tượng. Dưới đây chúng ta sẽ xét một ví dụ: xây dựng chương trình mô phỏng một hệ sắp hàng (hệ phục vụ), tức là hệ với các hàng đợi được phục vụ theo nguyên tắc ai đến trước người đó được phục vụ trước, chẳng hạn, hệ các quầy giao dịch ở ngân hàng, hệ các cửa bán vé tàu ở nhà ga.

Trường hợp đơn giản nhất, hệ sắp hàng chỉ có một người phục vụ và một hàng đợi, như được chỉ ra trong hình 7.5a. Tổng quát hơn, một hệ sắp hàng có thể gồm k hàng đợi và s người phục vụ, như trong 7.5b.



(a)



(b)

Hình 7.5 (a) Hệ sắp hàng đơn giản
(b) Hệ sắp hàng với k hàng đợi, s người phục vụ.

Sau đây chúng ta xét trường hợp đơn giản nhất: hệ chỉ có một hàng đợi các khách hàng và một người phục vụ. Chúng ta sẽ mô phỏng sự hoạt động của hệ này trong khoảng thời gian T (tính bằng phút, chẳng hạn), kể từ thời điểm ban đầu t = 0. Trong khoảng thời gian 1 phút có thể có khách hàng đến hoặc không, chúng ta không thể biết trước được. Song giả thiết rằng, chúng ta được cho biết xác suất p: xác suất trong khoảng thời gian 1 phút có khách hàng đến. Trong trường hợp tổng quát, thời gian phục vụ dành cho mỗi khách hàng chúng ta cũng không biết trước được, và thời gian phục vụ các khách hàng khác nhau cũng khác nhau, chẳng hạn thời gian phục vụ các khách hàng ở ngân hàng. Để cho đơn giản, chúng ta giả thiết rằng thời gian

phục vụ mỗi khách hàng là như nhau và là s (phút) (chẳng hạn như một trạm rửa xe tự động, nó rửa sạch mỗi xe hết 5 phút). Như vậy, chúng ta đã biết các thông tin sau đây:

1. Xác suất trong thời gian 1 phút có khách đến là p ($0 \le p \le 1$), (chúng ta giả thiết rằng trong 1 phút chỉ có nhiều nhất một khách hàng đến)
2. Thời gian phục vụ mỗi khách hàng là s phút. Chương trình mô phỏng cần thực hiện các nhiệm vụ sau:
   - Đánh giá số khách hàng được phục vụ.
   - Đánh giá thời gian trung bình mà mỗi khách hàng phải chờ đợi.

Để xây dựng được chương trình mô phỏng hệ sắp hàng có một hàng đợi và một người phục vụ với các giả thiết đã nêu trên, chúng ta cần tạo ra hai lớp: lớp các khách hàng, và lớp người phục vụ.

Lớp các khách hàng đương nhiên là lớp hàng đợi Queue mà chúng ta đã cài đặt (lớp trong hình 7.2, hoặc 7.4). Vấn đề mà chúng ta cần giải quyết ở đây là biểu diễn các khách hàng như thế nào? Chúng ta không cần quan tâm tới các thông tin về khách hàng như tên, tuổi, giới tính, … ; với các nhiệm vụ của chương trình mô phỏng đã nêu ở trên, chúng ta chỉ cần quan tâm tới thời điểm mà khách hàng đến và vào hàng đợi. Vì vậy, chúng ta sẽ biểu diễn mỗi khách hàng bởi thời điểm t mà khách hàng vào hàng đợi, t là số nguyên dương. Và do đó, trong chương trình, chúng ta chỉ cần khai báo:

```
Queue<int>  customer ;
```

Bây giờ chúng ta thiết kế lớp người phục vụ ServerClass. Tại mỗi thời điểm, người phục vụ có thể đang bận phục vụ một khách hàng, hoặc không. Vì vậy để chỉ trạng thái (bận hay không bận) của người phục vụ, chúng ta đưa vào lớp một biến time-left, biến này ghi lại thời gian còn lại mà người phục vụ cần làm việc với khách hàng đang được phục vụ. Khi bắt đầu phục vụ một khách hàng biến time-left được gán giá trị là thời gian phục vụ. Người phục vụ bận có nghĩa là time-left > 0. Tại mỗi thời điểm t khi người phục vụ tiếp tục làm việc với một khách hàng, biến time-left sẽ giảm đi 1. Khi mà time-left nhận giá trị 0 có nghĩa là người phục vụ rỗi, có thể phục vụ khách hàng tiếp theo. Lớp người phục vụ được khai báo như sau:

```
class      ServerClass
{
  public:
     ServerClass (int s) // Khởi tạo người phục vụ rỗi với thời gian
     // phục vụ mỗi khách hàng là s.
          { server-time = s; time-left = 0 ; }
     bool    Busy( )  const // Người phục vụ có bận không?
          { return   time-left > 0 ; }
     void     Start( ) // Bắt đầu phục vụ một khách hàng.
```

```
        { time-left = server-time ; }
    void    Continue( ) // Tiếp tục làm việc với khách hang.
        { if (Busy( ))   time-left - - ; }
  private :
    int   server-time ; // Thời gian phục vụ mỗi khách hàng.
    int   time-left ;
}
```

Tới đây chúng ta có thể thiết kế chương trình mô phỏng. Hàm mô phỏng chứa các biến đầu vào sau:

- Thời gian phục vụ mỗi khách hàng: s
- Xác suất khách hàng đến: p
- Thời gian mô phỏng: T

Các biến đầu ra:

- Số khách hàng được phục vụ: count
- Thời gian chờ đợi trung bình của khách hàng: wait-time.

Hàm mô phỏng sẽ mô tả sự hoạt động của hệ sắp hàng bởi một vòng lặp: tại mỗi thời điểm t (t = 1, 2, … , T), nếu có khách hàng đến thì đưa nó vào hàng đợi, nếu người phục vụ bận thì cho họ tiếp tục làm việc; rồi sau đó kiểm tra xem, nếu hàng đợi không rỗng và người phục vụ rỗi thì cho khách hàng ở đầu hàng ra khỏi hàng và người phục vụ bắt đầu phục vụ khách hàng đó. Hàm mô phỏng có nội dung như sau:

```
void    Queueing_System_Simulation
(int s, double p, int T, int & count, int & wait-time)
{
  Queue <int>  customer ; // Hàng đợi các khách hàng.
  ServerClass   server(s) ; // Người phục vụ.
  void   QueueEntry( int t) ;
  // Hàm làm nhiệm vụ đưa khách hàng đến (nếu có) tại thời điểm t
  // vào hàng đợi.
  int   t ; // Biến chỉ thời điểm, t = 1, 2, … , T
  int   sum = 0 ; // Tổng thời gian chờ đợi của khách hàng.
  count = 0 ;
  for (t = 1; t < = T ; t + +)
  {
    QueueEntry(t) ; // Đưa khách hàng (nếu có) vào hàng đợi.
    if (server.Busy( ))
            server.Continue( ) ;
    else  if ( ! customer.Empty( ))
            {
              int   t1 = customer.Dequeue( );
              server.Start( ) ;
```

```
                    sum  + =  t – t1 ;
                    count + + ;
                }
    }
    wait-time = sum / count ;
}
```

Bây giờ chúng ta cần cài đặt hàm QueueEntry. Hàm này có nhiệm vụ đưa ra quyết định tại thời điểm t có khách hàng đến hay không và nếu có thì đưa khách hàng đó vào đuôi hàng đợi. Chúng ta đã biết trước xác suất để trong khoảng thời gian 1 phút có khách hàng đến là p. Vì vậy chúng ta sử dụng hàm rand( ) để sinh ra số nguyên ngẫu nhiên trong khoảng từ 0 tới RAND- MAX. (Hàm rand( ) và hằng số RAND-MAX có trong thư viện chuẩn stdlib.h.). Tại mỗi thời điểm t (t = 1, 2, …), chúng ta gọi hàm rand( ) để sinh ra một số nguyên ngẫu nhiên, nếu số nguyên này nhỏ hơn p * RAND-MAX thì chúng ta cho rằng tại thời điểm đó có khách hàng đến.

```
void QueueEntry(int t)
{
  if (rand( ) < p * RAND-MAX)
        customers.Enqueue(t) ;
}
```

BÀI TẬP.

1. Hãy cài đặt lớp Queue như là lớp dẫn xuất từ lớp cơ sở Llist với dạng thừa kế private. Làm thế nào để các hàm Empty( ), Length( ) của lớp Llist trở thành hàm thành phần public của lớp Queue ?

2. Hàng hai đầu (double-ended queue, hoặc deque) được định nghĩa là một danh sách với các phép toán xen, loại được phép thực hiện ở cả hai đầu. Hãy đặc tả KDLTT này và đưa ra các cách cài đặt thích hợp bởi mảng và bởi DSLK.

3. Cho bàn cờ tướng với một số quân nằm ở các vị trí tuỳ ý và hai vị trí A và B bất kỳ trên bàn cờ. Sử dụng hàng đợi, hãy thiết kế và cài đặt thuật toán tìm đường đi ngắn nhất (nếu có) của con mã từ vị trí A đến vị trí B.

CHƯƠNG 8

CÂY

Các CTDL được nghiên cứu trong các chương trước (danh sách, ngăn xếp, hàng đợi) là các CTDL tuyến tính (các thành phần dữ liệu được sắp xếp tuyến tính). Trong chương này chúng ta sẽ nghiên cứu một loại CTDL không tuyến tính: CTDL cây. Cây là một trong các CTDL quan trọng nhất trong khoa học máy tính. Hầu hết các hệ điều hành đều tổ chức các file dưới dạng cây. Cây được sử dụng trong thiết kế chương trình dịch, xử lý ngôn ngữ tự nhiên, đặc biệt trong các vấn đề tổ chức dữ liệu để tìm kiếm và cập nhật dữ liệu hiệu quả. Trong chương này, chúng ta sẽ trình bày các vấn đề sau đây:

- Các khái niệm cơ bản về cây và các CTDL biểu diễn cây.
- Nghiên cứu một lớp cây đặc biệt: Cây nhị phân.

- Nghiên cứu CTDL cây tìm kiếm nhị phân và cài đặt KDLTT tập động bởi cây tìm kiếm nhị phân.

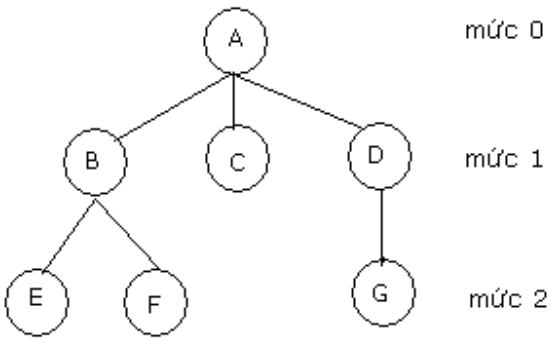
8.1 CÁC KHÁI NIỆM CƠ BẢN

Chúng ta có thể xác định khái niệm cây bằng hai cách: đệ quy và không đệ quy. Trước hết chúng ta đưa ra định nghĩa cây thông qua các khái niệm trong đồ thị định hướng. Một ví dụ điển hình về cây là tập hợp các thành viên trong một dòng họ với quan hệ cha – con. Trừ ông tổ của dòng họ này, mỗi một người trong dòng họ là con của một người cha nào đó trong dòng họ. Biểu diễn dòng họ dưới dạng đồ thị hướng: quan hệ cha – con được biểu diễn bởi các cung của đồ thị, nếu A là cha của B, thì trong đồ thị có cung đi từ đỉnh A tới đỉnh B. Xem xét các đặc điểm của đồ thị định hướng này, chúng ta đưa ra định nghĩa cây như sau:

Cây là một đồ thị định hướng thỏa mãn các tính chất sau:

- Có một đỉnh đặc biệt được gọi là **gốc** cây
- Mỗi đỉnh C bất kỳ không phải là gốc, tồn tại duy nhất một đỉnh P có cung đi từ P đến C. Đỉnh P được gọi là **cha** của đỉnh C, và C là con của P
- Có đường đi duy nhất từ gốc tới mỗi đỉnh của cây.

Người ta quy ước biểu diễn cây như trong hình 8.1: gốc ở trên cùng, hàng dưới là các đỉnh con của gốc, dưới một hàng là các đỉnh con của các đỉnh trong hàng đó, có đoạn nối từ đỉnh cha tới đỉnh con (cần lưu ý cung luôn luôn đi từ trên xuống dưới)



Hình 8.1. Biểu diễn hình học một cây

Sau đây chúng ta sẽ đưa ra một số thuật ngữ hay được dùng đến sau này.

Mở rộng của quan hệ cha – con, là quan hệ tổ tiên – con cháu. Trong cây nếu có đường đi từ đỉnh A tới đỉnh B thì A được gọi là **tổ tiên** của B, hay B là **con cháu** của A. Chẳng hạn, gốc cây là tổ tiên của các đỉnh còn lại trong cây.

Các đỉnh cùng cha được xem là **anh em**. Chẳng hạn, trong cây ở hình 8.1 các đỉnh B, C, D là anh em.

Các đỉnh không có con được gọi là **lá**. Trong hình 8.1, các đỉnh lá là E, F, C, G. Một đỉnh không phải là lá được gọi là **đỉnh trong**.

Một đỉnh bất kỳ A cùng với tất cả các con cháu của nó lập thành một cây gốc là A. Cây này được gọi là **cây con** của cây đã cho. Nếu đỉnh A là con của gốc, thì cây con gốc A được gọi là **cây con của gốc.**

Độ cao của cây là số đỉnh nằm trên đường đi dài nhất từ gốc tới một lá. Chẳng hạn, cây trong hình 8.1 có độ cao là 3. Dễ dàng thấy rằng, độ cao của cây là độ cao lớn nhất của cây con của gốc cộng thêm 1.

Độ sâu của một đỉnh là độ dài đường đi từ gốc tới đỉnh đó. Chẳng hạn, trong hình 8.1, đỉnh G có độ sâu là 2.

Cây là một CTDL **phân cấp**: Các đỉnh của cây được phân thành các mức. Mức của mỗi đỉnh được xác định đệ quy như sau:

- Gốc ở mức 1
- Mức của một đỉnh = mức của đỉnh cha + 1

Như vậy, các đỉnh trong cùng một mức là đỉnh con của một đỉnh nào đó ở mức trên. Độ cao của cây chính là mức lớn nhất của cây. Ví dụ, cây trong hình 8.1 được phân thành 3 mức: mức 1 chỉ gồm có gốc, mức 2 gồm các đỉnh A, B, C, D, mức 3 gồm các đỉnh E, F, G.

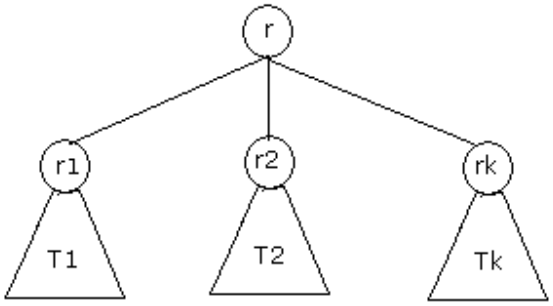
Sau này chúng ta chỉ quan tâm đến các cây được sắp. **Cây được sắp** là cây mà các đỉnh con của mối đỉnh được sắp sếp theo một thứ thứ tự xác định. Giả sử a là một đỉnh và các con của nó được sắp xếp theo thứ tự b_1, b_2, ..., b_k ($k \geq 1$), khi đó đỉnh b_1 được gọi là **con cả** của a, còn đỉnh b_{i+1} (i=1, 2, ...k-1) được gọi là **em liền kề** của đỉnh b_i.

Trong biểu diễn hình học, đỉnh con cả là đỉnh ngoài cùng bên trái, đỉnh b_k là đỉnh ngoài cùng bên phải.

Trên đây chúng ta đã định nghĩa cây như một đồ thị định hướng có một số tính chất đặc biệt. Khái niệm cây còn có thể định nghĩa một cách khác: định nghĩa đệ quy.

Định nghĩa đệ quy. Cây là một tập hợp không rỗng T các phần tử (được gọi là các **đỉnh**) được xác định đệ quy như sau:

- Tập chỉ có một đỉnh a là cây, cây này có gốc là a.
- Giả sử T_1, T_2, ..., T_k ($k \geq 1$) là các cây có gốc tương ứng là r_1, r_2, ..., r_k , trong đó hai cây bất kỳ không có đỉnh chung. Giả sử r là một đỉnh mới không có trong các cây đó. Khi đó tập T gồm đỉnh r và tất cả các đỉnh trong các cây T_i (i=1, ..., k) lập thành một cây có gốc là đỉnh r, và r là đỉnh cha của đỉnh r_i hay r_i là đỉnh con của r (i=1, ..., k). Các cây T_i (i=1, ..., k) được gọi là các cây con của gốc r. Cây T được biểu diễn hình học như sau:



Sử dụng định nghĩa cây đệ quy, chúng ta có thể dễ dàng đưa ra các thuật toán đệ quy cho các nhiệm vụ xử lý trên cây.

Sau đây chúng ta xét xem có thể biểu diễn cây bởi các CTDL như thế nào.

Cài đặt cây

Cây có thể cài đặt bởi các CTDL khác nhau. Chúng ta có thể sử dụng mảng để cài đặt cây. Song cách này không thuận tiện, ít được sử dụng. Sau đây, chúng ta trình bày hai phương pháp cài đặt cây thông dụng nhất.

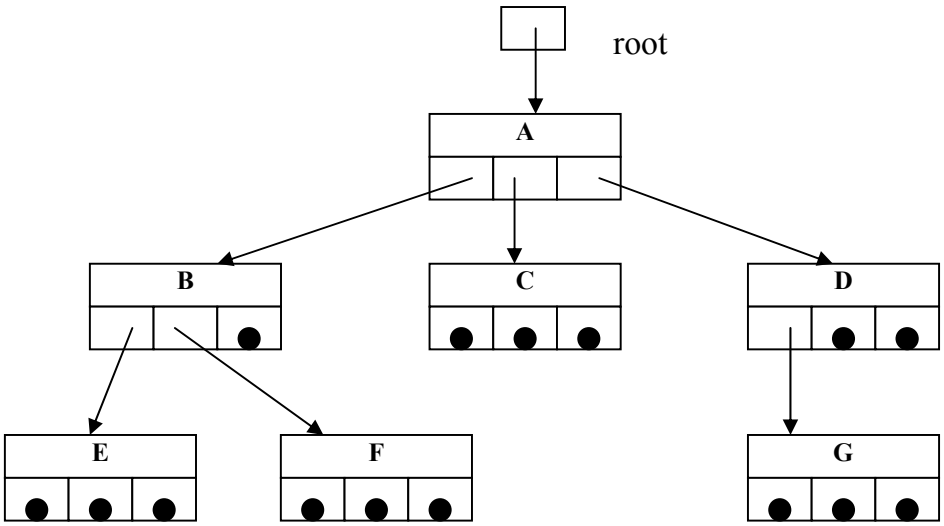
Phương pháp 1 (chỉ ra danh sách các đỉnh con của mỗi đỉnh). Với mỗi đỉnh của cây, ta sử dụng một con trỏ trỏ tới một đỉnh con của nó. Và như vậy, mỗi đỉnh của cây được biểu diễn bởi một cấu trúc gồm hai thành phần: một biến **data** lưu dữ liệu chứa trong đỉnh đó và một mảng **child** các con trỏ trỏ tới các đỉnh con. Giả sử, mỗi đỉnh chỉ có nhiều nhất K đỉnh con, khi đó ta có thể mô tả mỗi đỉnh bởi cấu trúc sau:

```
const  int  K = 10;
template <class Item>
{
  Item   data;
  Node*   child [K];
};
```

Chúng ta có thể truy cập tới một đỉnh bất kỳ trong cây bằng cách đi theo các con trỏ bắt đầu từ gốc cây. Vì vậy, ta cần có một con trỏ ngoài trỏ tới gốc cây, con trỏ root:

Node <Item>* root;

với cách cài đặt này, cây trong hình 8.1 được cài đặt bởi CTDT được biểu diễn hình học trong hình 8.2.



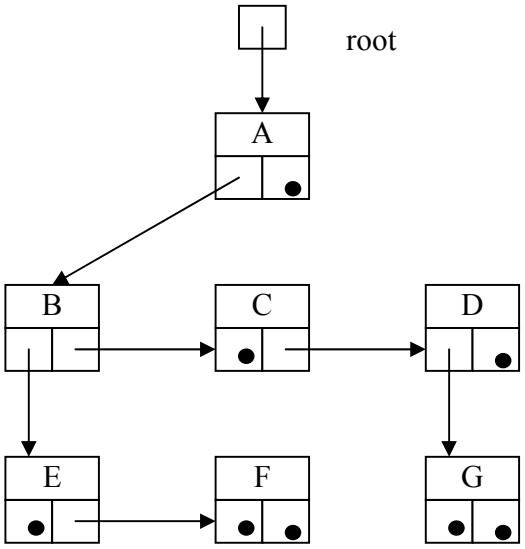
Hình 8.2. Cài đặt cây bởi mảng con trỏ.

Phương pháp 2 (chỉ ra con cả và em liền kề của mỗi đỉnh). Trong một cây, số đỉnh con của các đỉnh có thể rất khác nhau. Trong trường hợp đó, nếu sử dụng mảng con trỏ, sẽ lãng phí bộ nhớ. Thay vì sử dụng mảng con trỏ, ta chỉ sử dụng hai con trỏ: con trỏ firstChild trỏ tới đỉnh con cả và con trỏ nextSibling trỏ tới em liền kề. Mỗi đỉnh của cây được biểu diễn bởi cấu trúc sau:

```
template  <class Item>
struct    Node
{
```

```
    Item    data;
    Node*;  firstChild;
    Node*   nextSibling;
};
```

Chúng ta cũng cần có một con trỏ ngoài root trỏ tới gốc cây như trong phương pháp 1. Với cách này, cây trong hình 8.1 được cài đặt bởi CTDL như trong hình 8.3. Dễ dàng thấy rằng, xuất phát từ gốc đi theo con trỏ firstChild hoặc con trỏ nextSibling, ta có thể truy cập tới đỉnh bất kỳ trong cây. Ta có nhận xét rằng, các con trỏ nextSibling liên kết các đỉnh tạo thành một danh sách liên kết biểu diễn danh sách các đỉnh con của mỗi đỉnh.



Hình 8.3. Cài đặt cây sử dụng hai con trỏ.

Cần chú ý rằng, trong một số trường hợp, để thuận tiện cho các xử lý, ta có thể đưa thêm vào cấu trúc Node một con trỏ **parent** trỏ tới đỉnh cha.

8.2 DUYỆT CÂY

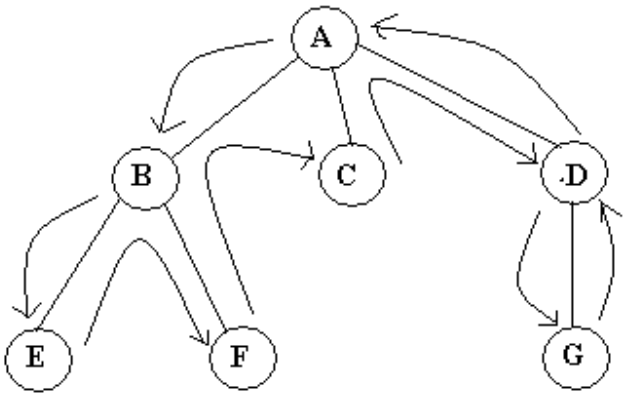
Người ta thường sử dụng cây để tổ chức dữ liệu. Khi dữ liệu được tổ chức dưới dạng cây, thì hành động hay được sử dụng là duyệt cây. **Duyệt**

cây có nghĩa là lần lượt thăm các đỉnh của cây theo một trật tự nào đó và tiến hành các xử lý cần thiết với các dữ liệu trong mỗi đỉnh của cây, chẳng hạn như in ra các dữ liệu đó. Có ba phương pháp duyệt cây hay được sử dụng nhất trong các ứng dụng là: duyệt cây theo **thứ tự trước** (preorder), theo **thứ tự trong** (inorder) và theo **thứ tự sau** (postorder). Chúng ta xác định các phương pháp duyệt cây này. Các phương pháp duyệt cây được mô tả rất đơn giản bằng đệ quy. Giả sử T là cây có gốc r và các cây con của gốc là T_1, T_2, ..., T_k (k>=0).

Duyệt cây T theo thứ tự trước có nghĩa là:

- Thăm gốc r.
- Duyệt lần lượt các cây con T_1,..., T_k theo thứ tự trước.

Chẳng hạn, xét cây trong hình 8.1. Thứ tự các đỉnh được thăm theo phương pháp này là A, B, E, F, C, D, G. Phân tích kỹ thuật duyệt cây theo thứ tự trước, ta rút ra quy luật đi thăm các đỉnh của cây như sau: đầu tiên thăm gốc r , sau đó đi xuống thăm con cả r_1 của gốc (r_1 là gốc của cây con T_1), rồi lại tiếp tục đi xuống thăm con cả của r_1... Khi không đi sâu xuống được, tức là đạt tới một đỉnh không có con (lá), ta quay lên cha của nó và đi xuống thăm một đỉnh con tiếp theo (nếu có) của đỉnh cha đó, rồi lại tiếp tục đi xuống... Quá trình đi thăm các đỉnh của cây trong hình 8.1 được biểu diễn bởi hình 8.4. Như vậy, duyệt cây theo thứ tự trước có nghĩa là xuất phát thăm từ gốc, luôn luôn đi sâu xuống thăm các đỉnh con, chỉ trừ khi nào không xuống dưới được nữa mới quay lên đỉnh cha để rồi lại tiếp tục đi xuống. Do đó, kỹ thuật duyệt cây theo thứ tự trước còn được gọi là kỹ thuật **tìm kiếm theo độ sâu**.



Hình 8.4. Thăm các đỉnh của cây theo thứ tự trước

Duyệt cây T theo thứ tự trong được quy định như sau:

- Duyệt cây con T_1 theo thứ tự trong
- Thăm gốc r
- Duyệt lần lượt các cây con T_2, ..., T_k theo thứ tự trong.

Ví dụ, thứ tự các đỉnh của cây trong hình 8.1 được thăm theo thứ tự là E, B, F, A, C, G, D.

Duyệt cây T theo thứ tự sau được tiến hành như sau:

- Duyệt lần lượt các cây con T_1, ...T_k theo thứ tự sau
- Thăm gốc r

Chẳng hạn, cũng với cây trong hình 8.1, các đỉnh được thăm theo phương pháp này lần lượt là E, F, B, C, G, D, A.

Bây giờ chúng ta cài đặt các hàm duyệt cây. Các kỹ thuật duyệt cây được xác định đệ quy, vì vậy dễ dàng cài đặt các kỹ thuật duyệt cây bởi các hàm đệ quy. Sau đây ta viết hàm đệ quy duyệt cây theo thứ tự trước: hàm Preorder. Hàm này chứa một tham biến là con trỏ root trỏ tới gốc cây. Lưu ý

rằng, C++ cho phép tham biến của một hàm có thể là hàm. Chúng ta đưa vào hàm Preorder một tham biến khác, đó là hàm với khai báo sau:

```
void   f(Item&);
```

Hàm f là hàm bất kỳ thực hiện các xử lý nào đó với dữ liệu có kiểu Item, trong đó Item là kiểu của dữ liệu chứa trong đỉnh của cây. Giả sử cây được cài đặt bằng phương pháp sử dụng hai con trỏ: firstChild và nextSibling. Khi đó hàm đệ quy Preorder được cài đặt như sau:

```
template <class Item>
void   Preorder (Node<Item>*   root, void   f(Item&))
{
       if (root ! = NULL)
       {
            f (root → data);
            node <Item>*    P = root → firstChild;
            while (P ! = NULL)
            {
                Preorder (P, f);
                P = P → nextSibling;
            }
}
```

Chúng ta cũng có thể cài đặt hàm Preorder không đệ quy. Muốn vậy chúng ta cần sử dụng một ngăn xếp để lưu các đỉnh nằm trên đường đi từ gốc tới đỉnh đang được thăm để khi tới một đỉnh mà không đi sâu xuống được thì biết được đỉnh cha của nó mà quay lên. Thay vì lưu các đỉnh, ngăn xếp sẽ lưu các con trỏ trỏ tới các đỉnh.

Hàm Preorder không đệ quy:

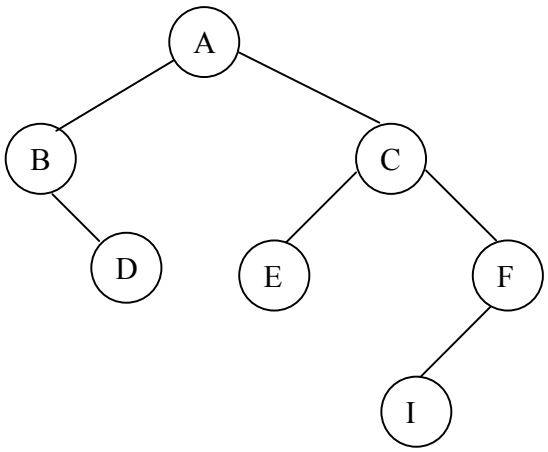
```
template <class Item>;
void  Preorder (Node <Item>*   root, void  f (Item))
{
        Stack <Node<Item>*> S; // Khởi tạo ngăn xếp rỗng S lưu
                                // các con trỏ.
        Node <Item>*  P = root
        while (P ! = NULL)
```

```
        {
                f (P → data);
                S. Push (P); // Đẩy con trỏ P vào ngăn xếp S
                P = P → firstChild;
        }
        while (! S. Empty ())
        {
                P = S. Pop (); // Loại con trỏ P ở đỉnh ngăn xếp.
                P = P → nextSibling;
                while (P ! = NULL)
                {
                        f (P → data);
                        S. Push (P);
                        P = P → firstChild;
                }
        }
}
```

Một cách tương tự, bạn đọc có thể viết ra các hàm đệ quy và không đệ quy thực hiện duyệt cây theo thứ tự trong: hàm Inorder, và duyệt cây theo thứ tự sau: hàm Postorder.

8.3 CÂY NHỊ PHÂN

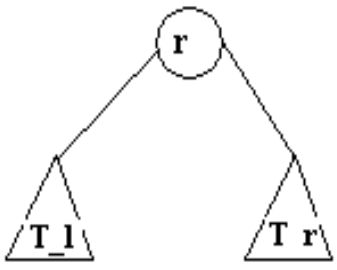
Trong mục này, chúng ta xét một lớp cây đặc biệt: **Cây nhị phân (Binary tree)**. Cây nhị phân có thể rỗng (không có đỉnh nào), nếu không rỗng thì mỗi đỉnh của cây nhị phân chỉ có nhiều nhất hai con được phân biệt là con trái và con phải. Điều đó có nghĩa rằng, trong cây nhị phân không rỗng, một đỉnh có thể không có con, có thể có đầy đủ cả con trái và con phải, có thể có con trái không có con phải hoặc có con phải nhưng không có con trái. Hình 8.5 biểu diễn một cây nhị phân: các đỉnh A, C có hai con, đỉnh B có con phải, đỉnh F có con trái.



Hình 8.5. Một cây nhị phân

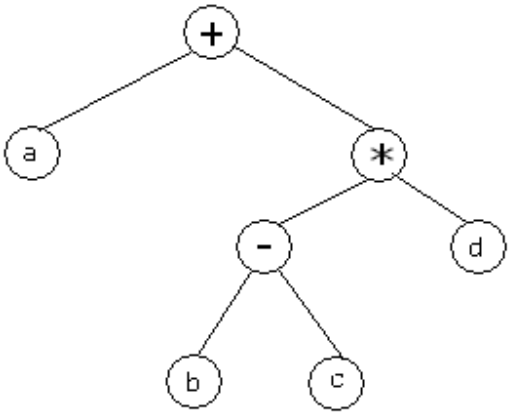
Chúng ta có thể định nghĩa cây nhị phân một cách đệ quy như sau:

- Một tập rỗng là cây nhị phân. Ta gọi là cây nhị phân rỗng
- Giả sử T_L và T_R là hai cây nhị phân không có đỉnh chung và r là một đỉnh không có trong các cây T_L và T_R. Khi đó một cây nhị phân T được tạo thành với gốc là r, có T_L là cây con trái của gốc và T_R là cây con phải của gốc. Cây nhị phân T được biểu diễn hình học như sau:



Trong định nghĩa trên, nếu T_L không rỗng thì gốc của nó được gọi là đỉnh con trái của r. Tương tự, nếu T_R không rỗng thì gốc của nó được gọi là đỉnh con phải của r.

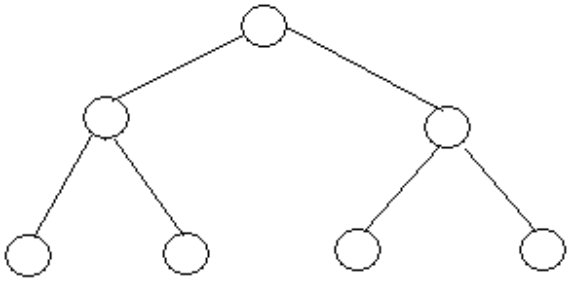
Một ví dụ về cây nhị phân là cây biến thức, nó biểu diễn các biểu thức số học. Trong cây biểu thức, các đỉnh trong biểu diễn các phép toán + , - , *, / ; cây con trái (cây con phải) của một đỉnh trong biểu diễn biểu thức con là toán hạng bên trái (toán hạng bên phải) của phép toán chứa ở đỉnh trong đó; các lá của cây biểu diễn các toán hạng có trong biểu thức. Chẳng hạn, biểu thức a + (b – c) * d được biểu diễn bởi cây nhị phân như trong hình 8.6. Chú ý rằng, nếu viết ra các đỉnh của cây biểu thức theo thứ tự sau, chúng ta nhận được biểu thức số học dạng postfix. Chẳng hạn với cây trong hình 8.6, ta có biểu thức dạng postfix: a b c – d * +.



Hình 8.6. Cây nhị phân biểu diễn biểu thức a + (b – c) * d

Sau đây chúng ta xác định một số dạng cây nhị phân đặc biệt sẽ được sử dụng đến sau này.

Cây nhị phân đầy đủ (full binary tree). Cây nhị phân rỗng (có độ cao 0), hoặc cây chỉ có đỉnh gốc (độ cao 1) được xem là cây nhị phân đầy đủ. Cây nhị phân có độ cao h >= 2 được xem là đầy đủ, nếu tất cả các đỉnh ở các mức trên (mức 1, 2, ..., h-1) đều có đầy đủ cả hai con. Hình 8.7 minh họa một cây nhị phân đầy đủ.

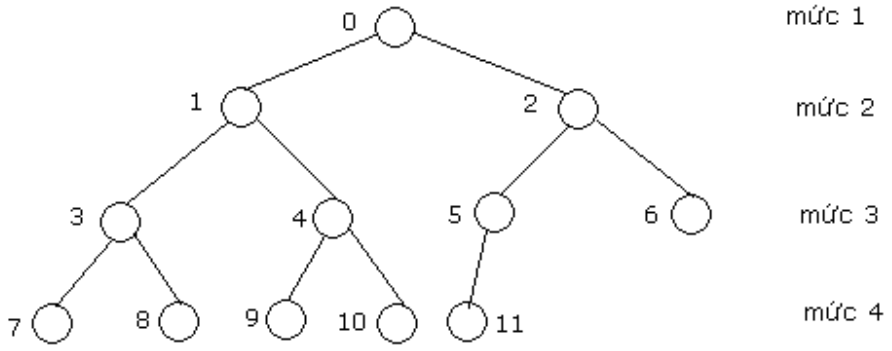


Hình 8.7. Cây nhị phân đầy đủ

Cây nhị phân hoàn toàn (complete binary tree). Các cây nhị phân có độ cao h<=1 (tức là cây rỗng hoặc chỉ có một đỉnh) là cây nhị phân hoàn toàn. Cây nhị phân có độ cao h>= 2 được xem là hoàn toàn, nếu

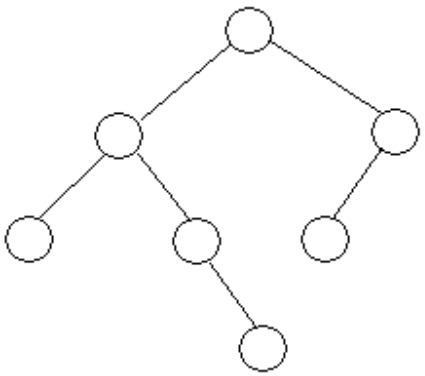
- Cây kể từ mức h – 1 trở lên là cây nhị phân đầy đủ.
- Ở mức h - 1, nếu một đỉnh chỉ có một con thì các đỉnh đứng trước nó (nếu có) có đầy đủ hai con, nếu một đỉnh chỉ có một con thì nó phải là đỉnh con trái.

Ví dụ, cây trong hình 8.8 là cây nhị phân hoàn toàn.



Hình 8.8. Cây nhị phân hoàn toàn

Cây nhị phân cân bằng (balanced binary tree). Trong một cây nhị phân, nếu độ cao của cây con trái của một đỉnh bất kỳ và độ cao của cây con phải của đỉnh đó khác nhau không quá 1 thì cây được gọi là cây nhị phân cân bằng. Hình 8.9 minh họa một cây nhị phân cân bằng.



Hình 8.9. Cây nhị phân cân bằng

Cấu trúc dữ liệu biểu diễn cây nhị phân

Do mỗi đỉnh của cây nhị phân chỉ có hai cây con: cây con trái và cây con phải, nên cách biểu diễn thông dụng nhất là sử dụng hai con trỏ: con trỏ **left** trỏ tới gốc của cây con trái, con trỏ **right** trỏ tới gốc của cây con phải, khi mà cây con trái (cây con phải) rỗng thì con trỏ left (right) có giá trị là NULL. Bằng cách này, mỗi đỉnh của cây nhị phân có cấu trúc sau:

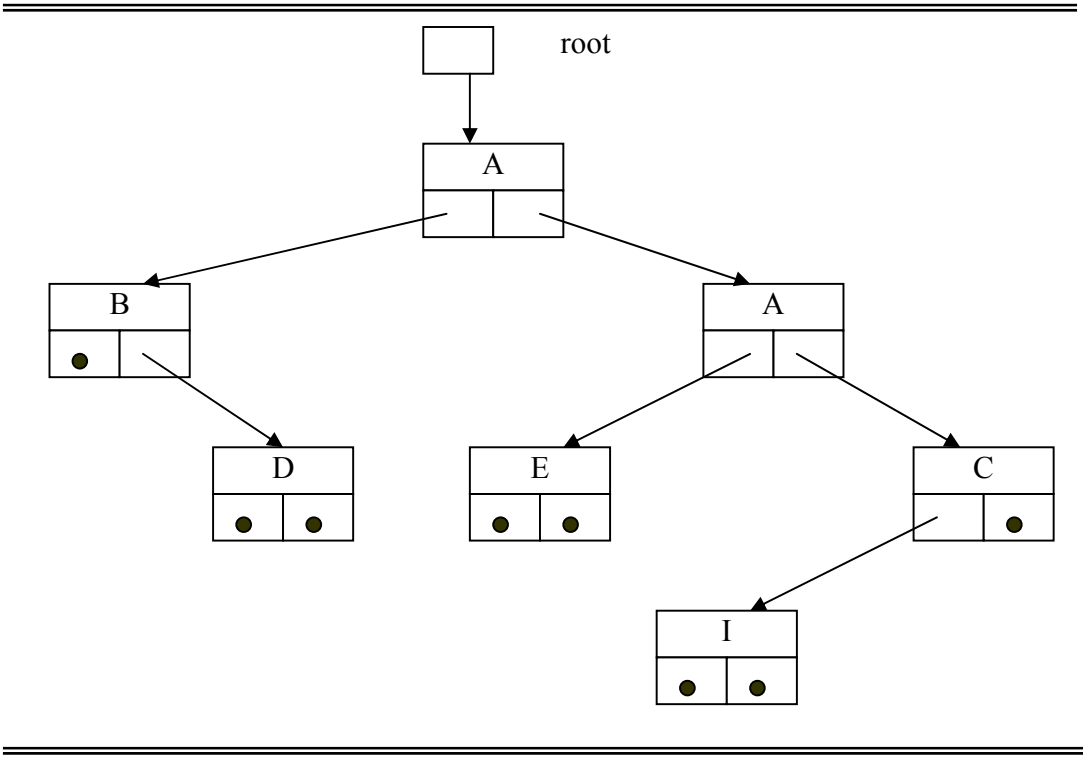
```
template <class Item>
struct    Node
{
     Item    data; // Dữ liệu chứa trong mỗi đỉnh
     Node*   left;
     Node*   right;
};
```

Mỗi cây nhị phân được biểu diễn bởi một con trỏ ngoài trỏ tới đỉnh gốc của cây: con trỏ root.

Node* root;

Khi mà cây nhị phân rỗng, thì con trỏ root có giá trị là NULL. Chẳng hạn, cây nhị phân trong hình 8.5 được biểu diễn bởi CTDL được minh họa trong hình 8.10. Có thể cài đặt cây nhị phân bởi mảng, bạn đọc hãy đưa ra cách cài đặt này (bài tập).

Biểu diễn cây nhị phân hoàn toàn bởi mảng. Từ các tính chất đặc biệt của cây nhị phân hoàn toàn, chúng ta có thể đưa ra cách cài đặt rất đơn giản và hiệu quả. Chúng ta đánh số các đỉnh cây nhị phân hoàn toàn theo thứ tự các mức từ trên xuống dưới, trong cùng một mức thì theo thứ tự từ trái qua phải, bắt đầu từ gốc được đánh số là 0, như trong hình 8.8. Với cách đánh số này, ta có nhận xét rằng, nếu một đỉnh được đánh số là i thì đỉnh con trái (nếu có) là 2*i + 1, đỉnh con phải (nếu có) là 2*i + 2, còn cha của i là đỉnh (i - 1)/2. Do đó chúng ta có thể sử dụng một mảng T để lưu các đỉnh của cây nhị phân hoàn toàn, đỉnh i (i = 0, 1, 2,...) được lưu trong thành phần T[i] của mảng.



Hình 8.10. Cài đặt cây nhị phân sử dụng con trỏ.

Duyệt cây nhị phân. Cũng như đối với cây tổng quát, người ta thường tiến hành duyệt cây nhị phân theo thứ tự trước, trong và sau. Duyệt cây nhị phân theo thứ tự trước (Preorder) có nghĩa là: Nếu cây không rỗng thì thăm gốc trước, sau đó duyệt cây con trái của gốc theo thứ tự trước, rồi duyệt cây con phải của gốc theo thứ tự trước. Chúng ta có thể dễ dàng cài đặt phương pháp duyệt cây nhị phân theo thứ tự trước bởi hàm đệ quy như sau:

```
template <class Item>
void   Preorder (Node <Item> *  root, void  f (Item &))
// Thăm các đỉnh của cây nhị phân theo thứ tự trước
// và tiến hành các xử lý (được mô tả bởi hàm f) với dữ
// liệu chứa trong mỗi đỉnh của cây.
{if (root! = NULL)
      {
      f (root → data);
      Preorder (root → left, f);
```

```
        Preorder (root → right, f);
      }
}
```

Bạn đọc hãy tự mình đưa ra định nghĩa duyệt cây nhị phân theo thứ tự trong và theo thứ tự sau, và viết các hàm đệ quy cài đặt các phương pháp duyệt cây đó (bài tập).

8.4 CÂY TÌM KIẾM NHỊ PHÂN

Một trong các ứng dụng quan trọng nhất của cây nhị phân là sử dụng cây nhị phân để tổ chức dữ liệu. Trong các chương trình, thông thường chúng ta cần phải lưu một tập các dữ liệu, rồi thường xuyên phải thực hiện cá phép toán: tìm kiếm dữ liệu, cập nhật dữ liệu... Trong các chương 4 và 5, chúng ta đã nghiên cứu sự cài đặt **KDLTT** tập động (một tập dữ liệu với các phép toán tìm kiếm, xen, loại...) bởi danh sách. Nếu tập dữ liệu được lưu trong **DSLK** thì các phép toán tìm kiếm, xen, loại, ... đòi hỏi thời gian O(n), trong đó n là số dữ liệu. Nếu tập dữ liệu được sắp xếp thành một danh sách theo thứ tự tăng (giảm) theo khóa tìm kiếm, và danh sách này được lưu trong mảng, thì phép toán tìm kiếm chỉ đòi hỏi thời gian O(logn) nếu sử dụng kỹ thuật tìm kiếm nhị phân (mục 4.4), nhưng các phép toán xen, loại vẫn cần thời gian O(n). Trong mục này, chúng ta sẽ nghiên cứu cách tổ chức một tập dữ liệu dưới dạng cây nhị phân, các dữ liệu được chứa trong các đỉnh của cây nhị phân theo một trật tự xác định, cấu trúc dữ liệu này cho phép ta cài đặt các phép toán tìm kiếm, xen, loại,... chỉ trong thời gian O(h), trong đó h là độ cao của cây nhị phân.

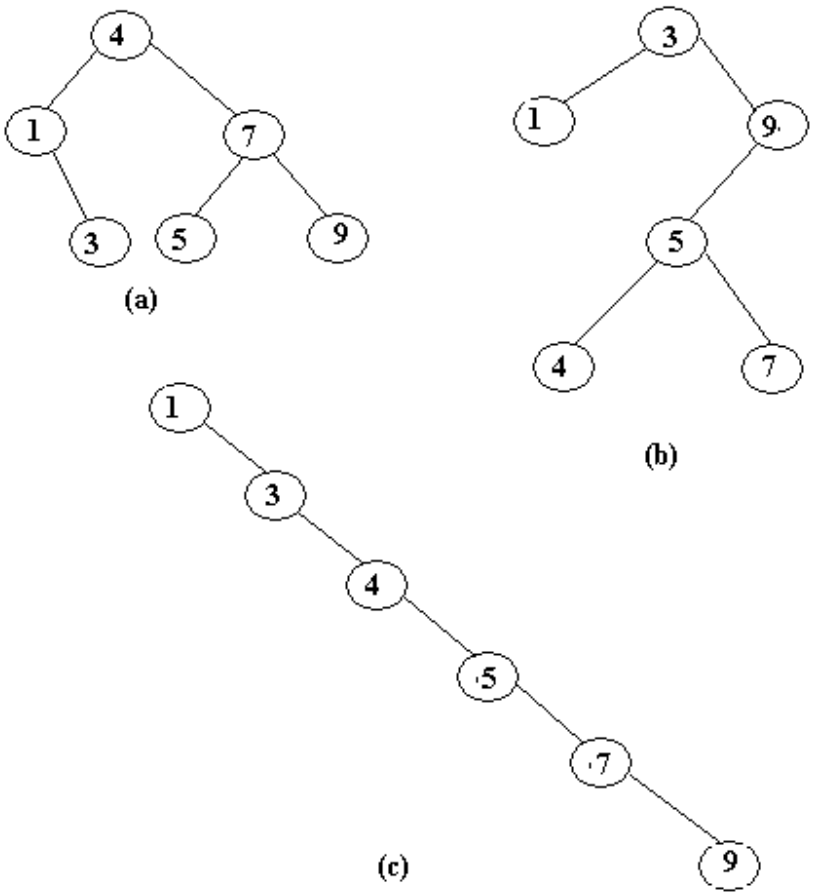
8.4.1 Cây tìm kiếm nhị phân

Giả sử chúng ta có một tập dữ liệu, các dữ liệu có kiểu Item nào đó chứa một thành phần được lấy làm khóa tìm kiếm. Chúng ta giả thiết rằng các giá trị khóa có thể sắp thứ tự tuyến tính, thông thường các giá trị khóa là các số nguyên, các số thực, các ký tự hoặc xâu ký tự. Chúng ta sẽ lưu tập dữ liệu đó trong một cây nhị phân (khóa của dữ liệu được nói tới như là khóa

của một đỉnh) theo trật tự như sau: giá trị khóa của một đỉnh bất kỳ lớn hơn các giá trị khóa của tất cả các đỉnh ở cây con trái của đỉnh đó và nhỏ hơn các giá trị khóa của tất cả các đỉnh ở cây con phải của đỉnh đó. Do đó, chúng ta có định nghĩa sau:

Cây tìm kiếm nhị phân (binary search tree) là cây nhị phân thỏa mãn tính chất sau: đối với mỗi đỉnh x trong cây, nếu y là đỉnh bất kỳ ở cây con trái của x thì khóa của x lớn hơn khóa của y, còn nếu y là đỉnh bất kỳ ở cây con phải của x thì khóa của x nhỏ hơn khóa của y.

Ví dụ. Chúng ta xét các cây nhị phân với các giá trị khoá của các đỉnh là các số nguyên. Các cây nhị phân trong hình 8.11 là các cây tìm kiếm nhị phân. Chúng ta có nhận xét rằng, các cây tìm kiếm nhị phân trong hình 8.11 biểu diễn cùng một tập hợp dữ liệu, nhưng cây trong hình 8.11a có độ cao là 3, cây trong hình 8.11b có độ cao là 4, còn cây trong hình 8.11c có tất cả các cây con trái của các đỉnh đều rỗng và nó có độ cao là 6. Một nhận xét quan trọng khác là, nếu chúng ta duyệt cây tìm kiếm nhị phân theo thứ tự trong, chúng ta sẽ nhận được một dãy dữ liệu được sắp xếp theo thứ tự khóa tăng dần. Chẳng hạn, với các cây tìm kiếm nhị phân trong hình 8.11, ta có dãy các giá trị khóa là 1, 3, 4, 5,7, 9.



Hình 11. Các cây tìm kiếm nhị phân

Chúng ta cũng có thể định nghĩa cây tìm kiếm nhị phân bởi đệ quy như sau:

- Cây nhị phân rỗng là cây tìm kiếm nhị phân
- Cây nhị phân không rỗng T là cây tìm kiếm nhị phân nếu:
    1. Khóa của gốc lớn hơn khóa của tất cả các đỉnh ở cây con trái T_L và nhỏ hơn khóa của tất cả các đỉnh ở cây con phải T_R.
    2. Cây con trái T_L và cây con phải T_R là các cây tìm kiếm nhị phân.

Sử dụng định nghĩa đệ quy này, chúng ta dễ dàng đưa ra các thuật toán đệ quy thực hiện các phép toán trên cây tìm kiếm nhị phân, như chúng ta sẽ thấy trong mục sau đây.

8.4.2 Các phép toán tập động trên cây tìm kiếm nhị phân

Bây giờ chúng ta xét xem các phép toán tập động (tìm kiếm, xen, loại, ...) sẽ được thực hiện như thế nào khi mà tập dữ liệu được cài đặt bởi cây tìm kiếm nhị phân. Chúng ta sẽ chỉ ra rằng, các phép toán tập động trên cây tìm kiếm nhị phân chỉ đòi hỏi thời gian O(h), trong đó h là độ cao của cây. Như độc giả đã thấy trong hình 8.12, một tập dữ liệu có thể lưu trong các cây tìm kiếm nhị phân có độ cao của cây có thể là n, trong đó n là số dữ liệu. Như vậy, trong trường hợp xấu nhất thời gian thực hiện các phép toán tập động trên cây tìm kiếm nhị phân là O(n). Tuy nhiên, trong mục 8.5 chúng ta sẽ chứng tỏ rằng, nếu cây tìm kiếm nhị phân được tạo thành bằng cách xen vào các dữ liệu được lấy ra từ tập dữ liệu một cách ngẫu nhiên, thì thời gian trung bình của các phép toán tập động là O(logn). Hơn nữa, bằng cách áp dụng các kỹ thuật hạn chế độ cao của cây, chúng ta có thể đảm bảo thời gian logarit cho các phép toán tập động trên cây tìm kiếm nhị phân (xem chương 11)

Dưới đây chúng ta sẽ đưa ra các thuật toán thực hiện các phép toán tập động trên cây tìm kiếm nhị phân. Chúng ta sẽ mô tả các thuật toán bởi các hàm dưới dạng giả mã . Trong các thuật toán, chúng ta sẽ sử dụng các ký hiệu sau đây: T là cây tìm kiếm nhị phân có gốc là root, dữ liệu chứa ở gốc được ký hiệu là rootData, cây con trái của gốc là T_L, cây con phải của gốc là T_R; v là một đỉnh, dữ liệu chứa trong đỉnh v được ký hiệu là data(v), đỉnh con trái của v là leftChild(v), đỉnh con phải là rightChild(v); dữ liệu trong các đỉnh có kiểu Item, khóa của dữ liệu d được ký hiệu là d.key.

Phép toán tìm kiếm. Cho cây tìm kiếm nhị phân T, để tìm xem cây T có chứa dữ liệu với khóa k cho trước hay không, chúng ta kiểm tra xem gốc có chứa dữ liệu với khóa k hay không. Nếu không, giả sử k < rootData.key, khi đó do tính chất của cây tìm kiếm nhị phân, dữ liệu với khóa k chỉ có thể

chứa trong cây con trái của gốc, và do đó, ta chỉ cần tiếp tục tìm kiếm trong cây con trái của gốc. Tương tự, nếu k > rootData.key, sự tìm kiếm được hạn chế trong phạm vi cây con phải của gốc. Từ đó, ta có thuật toán tìm kiếm đệ quy sau:

```
bool   Search (T, k)
// Tìm dữ liệu với khóa k trong cây tìm kiếm nhị phân
// Hàm trả về true (false) nếu tìm thấy (không tìm thấy)
{
if (T rỗng)
        return   false;
else if (rootData.key = = k)
        return   true;
      else if (k <rootData.key)
                        Search (T_L, k);
           else         Search (T_R, k)'
}
```

Phân tích thuật toán đệ quy trên, chúng ta dễ dàng đưa ra thuật toán tìm kiếm không đệ quy. Sử dụng biến v chạy trên các đỉnh của cây T bắt đầu từ gốc. Khi v là một đỉnh nào đó của cây T, chúng ta kiểm tra xem đỉnh v có chứa dữ liệu với khóa k hay không. Nếu không, tùy theo khóa k nhỏ hơn (lớn hơn) khóa của dữ liệu trong đỉnh v mà chúng ta đi xuống đỉnh con trái (con phải) của v. Thuật toán tìm kiếm không đệ quy là như sau:

```
bool       Search (T, k)
{
        if (T rỗng)
                return false;
        else {
                v = root;
                do {
                        if (data (v).key = = k)
                                return true;
                        else    if (k < data (v).key)
                                        if (v có con trái)
                                                v = leftchild (v);
                                        else    return false;
```

```
                else    if (v có con phải)
                                v = rightchild (v);
                        else    return false;
                }
        while (1);
}
```

Các đỉnh mà biến v chạy qua tạo thành một đường đi từ gốc hướng tới một lá của cây. Trong trường hợp xấu nhất, biến v sẽ dừng lại ở một đỉnh lá. Bởi vì độ cao của cây là độ dài của đường đi dài nhất từ gốc tới lá, do đó thời gian của phép toán Search là O(h). Chúng ta nhận thấy có sự tương tự giữa kỹ thuật tìm kiếm nhị phân (xem 4.4) và kỹ thuật tìm kiếm trên cây tìm kiếm nhị phân. Trong quá trình tìm kiếm trên cây tìm kiếm nhị phân , tại mỗi thời điểm chúng ta hạn chế tìm kiếm ở cây con trái hoặc ở cây con phải; còn trong tìm kiếm nhị phân chúng ta tiếp tục tìm kiếm ở nửa bên trái hay nửa bên phải của mảng. Tuy nhiên trong tìm kiếm nhị phân, tại mỗi thời điểm không gian tìm kiếm (mảng) được chia đôi, nửa bên trái và nửa bên phải bằng nhau; điều đó đảm bảo thời gian trong tìm kiếm nhị phân là O(logn). Nhưng trong cây tìm kiếm nhị phân, cây con trái và cây con phải có thể có số đỉnh rất khác nhau, do đó nói chung thời gian tìm kiếm trên cây tìm kiếm nhị phân không phải là O(logn), chỉ có thời gian này khi mà cây tìm kiếm nhị phân được xây dựng "cân bằng" tại mọi đỉnh.

Các phép toán tìm dữ liệu có khóa nhỏ nhất (phép toán Min) và tìm dữ liệu có khóa lớn nhất (phép toán Max)

Do tính chất của cây tìm kiếm nhị phân, nếu cây con phải không rỗng thì dữ liệu có khóa lớn nhất phải nằm ở cây con phải; tương tự, nếu cây con trái không rỗng thì dữ liệu có khóa nhỏ nhất phải nằm ở cây con trái. Từ đó, chúng ta dễ dàng đưa ra thuật toán đệ quy tìm dữ liệu có khóa lớn nhất (nhỏ nhất). Sau đây là hàm Min đệ quy:

```
Item   Min (T)
// Cây T không rỗng
// Hàm trả về dữ liệu có khóa nhỏ nhất.
```

```
{
        if (cây con trái T_L rỗng)
                        return    rootData;
        else            return    Min (T_L);
}
```

Chúng ta có nhận xét rằng, đỉnh chứa dữ liệu có khóa lớn nhất (nhỏ nhất) là đỉnh ngoài cùng bên phải (ngoài cùng bên trái). Chúng ta có thể đạt tớ đỉnh ngoài cùng bên phải (ngoài cùng bên trái) bằng cách xuất phát từ gốc và luôn luôn đi xuống nhánh bên phải (luôn luôn đi xuống nhánh bên trái). Do đó, chúng ta có thể đưa ra thuật toán không đệ quy cho phép toán Min như sau:

```
Item   Min (T)
{
  v = root;
  while (v có đỉnh con trái)
        v = leftChild (v);
  return        data (v);
}
```

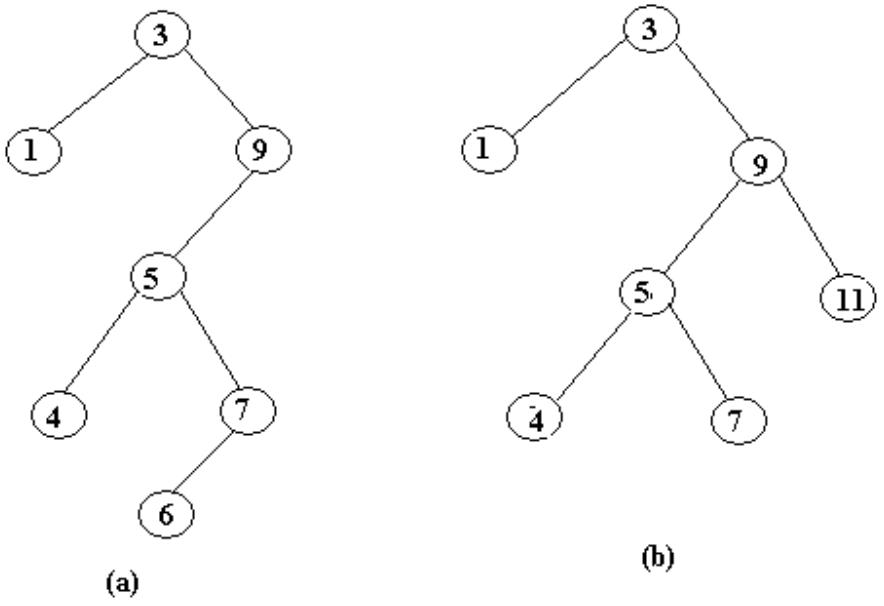
Phép toán xen (Insert). Chúng ta cần xen vào cây tìm kiếm nhị phân T một đỉnh mới chứa dữ liệu d, nhưng cần phải đảm bảo cây nhận được sau khi xen vẫn còn là cây tìm kiếm nhị phân. Nếu cây T rỗng thì ta chỉ cần tạo ra cây T chỉ có một đỉnh gốc chứa dữ liệu d. Trong trường hợp cây T không rỗng, nếu d.key < rootData.key (d.key > rootData.key) thì để đảm bảo tính chất của cây tìm kiếm nhị phân, chúng ta cần phải xen đỉnh mới chứa dữ liệu d vào cây con trái của gốc (cây con phải của gốc).

Hàm Insert đệ quy như sau:

```
void   Insert (T, d)
// xen vào cây T một đỉnh mới chứa dữ liệu d
{
  if (T rỗng)
        Tạo ra cây T chỉ có gốc chứa dữ liệu d;
  else  if (d.key < rootData.key)
```

```
                    insert (T_L, d);
            else  if (d.key > rootData.key)
                    insert (T_R, d);
}
```

Thuật toán Insert không đệ quy cũng tương tự như thuật toán Search không đệ quy. Nếu cây T không rỗng, ta xuất phát từ gốc và khi ở một đỉnh thì ta đi tiếp xuống đỉnh con trái, hoặc đỉnh con phải hoàn toàn giống như khi tìm kiếm, và dừng lại tại đỉnh v khi mà d.key < data (v).key nhưng đỉnh v không có con trái, hoặc khi mà d.key > data (v).key nhưng v không có con phải. Sau đó, ta gắn đỉnh mới chứa dữ liệu d làm đỉnh con trái của đỉnh v khi mà d.key < data (v).key, hoặc gắn đỉnh mới làm đỉnh con phải của v trong trường hợp kia. Chẳng hạn, nếu chúng ta xen vào cây trong hình 8.11b đỉnh mới chứa dữ liệu với khóa 6, ta nhận được cây trong hình 8.12a, còn nếu xen vào đỉnh có khóa 11, ta nhận được cây trong hình 8.12b.

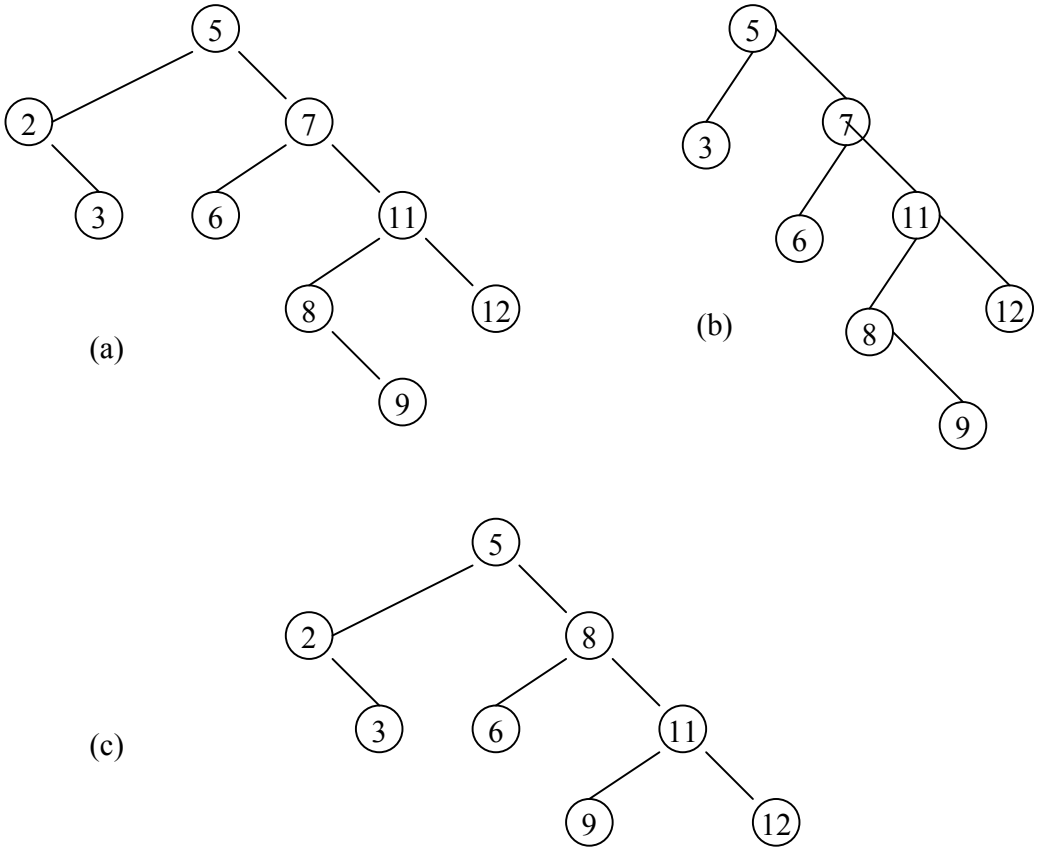


Hình 8.12. (a) Xen vào cây 8.11b đỉnh mới có khóa 6
(b) Xen vào cây 8.11b đỉnh mới có khóa 11

Phép toán loại (Delete). Chúng ta cần phải loại khỏi cây T một đỉnh chứa dữ liệu với khóa là k. Phép toán Delete là phép toán phức tạp nhất trong các phép toán trên cây tìm kiếm nhị phân. Trước hết, chúng ta cần phải áp dụng thủ tục tìm kiếm để định vị đỉnh cần loại trong cây. Giả sử đỉnh cần loại là đỉnh v. Các bước cần tiến hành tiếp theo phụ thuộc vào đỉnh v chỉ có nhiều nhất là một con hay có đầy đủ cả hai con. Chúng ta xét từng trường hợp:

1. Đỉnh v chỉ có nhiều nhất một con (tức là v chỉ có con phải, hoặc chỉ có con trái, hoặc v là lá). Trường hợp này rất đơn giản, chẳng hạn giả sử đỉnh v chỉ có con phải, khi đó nếu cha của v là đỉnh p và v là con trái của p thì ta chỉ cần đặt con trái của p là con phải của v. Chẳng hạn, xét cây trong hình 8.13a, đỉnh cần loại là đỉnh 2, đỉnh này chỉ có con phải là đỉnh 3, đỉnh 2 là con trái của đỉnh 5. Để loại đỉnh 2, ta chỉ cần đặt con trái của đỉnh 5 là đỉnh 3, ta nhận được cây trong hình 8.13b.

2. Chúng ta có nhận xét rằng, phép toán loại khỏi cây đỉnh có khóa nhỏ nhất (Delete Min) là trường hợp riêng của trường hợp này, bởi vì đỉnh có khóa nhỏ nhất là đỉnh ngoài cùng bên trái của cây, nó là đỉnh không có con trái.



Hình 8.13. (a) Một cây nhị phân
(b) Cây nhị phân (a) sau khi loại đỉnh 2
(c) Cây nhị phân (a) sau khi loại đỉnh 7

3. Đỉnh v có đầy đủ cả hai con, chẳng hạn đỉnh 7 trong cây hình 8.13. Vấn đề đặt ra là sau khi bỏ đi đỉnh v, chúng ta cần phải bố trí các đỉnh còn lại của cây như thế nào để nó vẫn còn là cây tìm kiếm nhị phân. Cách giải quyết vấn đề là quy về trường hợp đã xét, tức là chúng ta cần biến đổi cây thế nào để dẫn đến chỉ cần loại đỉnh có nhiều nhất một con. Trong cây con bên phải của đỉnh v, đỉnh có khóa nhỏ nhất là đỉnh ngoài cùng bên trái,

giả sử đó là đỉnh u. Nếu chúng ta thay dữ liệu chứa trong v bởi dữ liệu chứa trong đỉnh u thì thay vì loại đỉnh v ta chỉ cần loại đỉnh u, mà đỉnh u thì không có con trái. Chẳng hạn, trong cây hình 8.13a, giả sử đỉnh v là đỉnh chứa khóa 7, khi đó đỉnh u là đỉnh chứa khóa 8. Sau khi sao chép dữ liệu từ đỉnh u sang đỉnh v và loại đỉnh u ta nhận được cây trong hình 8.13c. Bạn đọc dễ dàng chứng minh được rằng các hành động như trên sẽ đảm bảo cây kết quả vẫn còn là cây tìm kiếm nhị phân (bài tập).

Tổng kết các bước đã trình bày trên, chúng ta đưa ra thuật toán loại đệ quy như sau:

```
void   Delete (T, k)
// Cây T không rỗng
// Loại khỏi cây T đỉnh chứa dữ liệu với khóa k
{
   if (k < rootData.key)
            Delete (TL, k); //Loại ở cây con trái
   else if (k > rootData.key)
            Delete (TR, k); // Loại ở cây con phải
        else if (cả hai TL và TR không rỗng)
            {
               rootData = Min (TR);
               DeleteMin (TR);
            }
           else if ( TL rỗng )
                        T = TR;
               else       T = TL;
}
```

Chúng ta cũng có thể đưa ra thuật toán loại không đệ quy. Để xác định đỉnh cần loại, ta chỉ cần một biến v chạy trên các đỉnh của cây bắt đầu từ gốc và dừng lại tại đỉnh cần loại (như trong thuật toán tìm kiếm không đệ quy). Tiếp theo để xác định đỉnh ngoài cùng bên trái trong cây con phải của đỉnh v, chúng ta cần sử dụng hai biến: biên u ghi lại đỉnh ngoài cùng bên trái trong cây con phải của đỉnh v và biến p ghi lại cha của đỉnh u. Sao chép dữ

liệu từ đỉnh u lên đỉnh v. Rồi sau đó gắn con mới cho đỉnh p khi bỏ đi đỉnh u. Độc giả hãy viết ra thuật toán loại không đệ quy này (bài tập).

Chúng ta có nhận xét rằng, các phép toán Insert, Delete cũng chỉ đòi hỏi thời gian O(h), bởi vì quá trình xác định vị trí để xen hoặc vị trí để loại là quá trình đi từ gốc xuống lá và xem xét dữ liệu chứa trong các đỉnh trên đường đi đó.

8.5 CÀI ĐẶT TẬP ĐỘNG BỞI CÂY TÌM KIẾM NHỊ PHÂN

Trong mục trước chúng ta đã nghiên cứu các thuật toán thực hiện các phép toán tập động (Search, Insert, Delete, Min, Max, DeleteMin) khi tập dữ liệu được cài đặt bởi CTDL cây tìm kiếm nhị phân. Trong mục này, chúng ta sẽ nghiên cứu sự cài đặt KDLTT tập động. Trước hết, chúng ta sẽ giả thiết rằng, dữ liệu được lưu trong đỉnh của cây tìm kiếm nhị phân có kiểu Item, Item là kiểu bất kỳ chứa một trường **key** (khóa tìm kiếm) có kiểu là **keyType**, keyType là kiểu có thứ tự bất kỳ (chẳng hạn int, double, char, string). Để đơn giản cho cách viết, ta giả thiết rằng, chúng ta có thể truy cập trực tiếp tới trường key, chẳng hạn khi Item là một struct.

KDLTT tập động sẽ được cài đặt bởi lớp khuôn phụ thuộc tham biến kiểu Item, lớp này được đặt tên là lớp BSTree (lớp cây tìm kiếm nhị phân). Trước hết, chúng ta xây dựng lớp BSNode (lớp đỉnh cây tìm kiếm nhị phân). Lớp BSNode cùng là lớp khuôn phụ thuộc tham biến kiểu Item và được mô tả trong hình 8.14. Cần lưu ý rằng, tất cả các thành phần của lớp BSNode đều là private, nhưng khai báo lớp BSTree là bạn để các hàm thành phần của lớp BSTree có thể truy cập trực tiếp đến các thành phần của lớp BSNode, lớp BSNode chỉ chứa một hàm kiến tạo.

```
template <class Item>
class    BSTree; //khai thác trước
template <class Item>
class     BSNode
{
```

```
        Item   data;
        BSNode*   left;
        BSNode*    right;
        Node (const Item & element)
        // Khởi tạo một đỉnh chứa dữ liệu là element
        : data (element), left (NULL), right (NULL) {}
       friend    class   BSTree <Item>;
};
```

Hình 8.14. Lớp đỉnh cây tìm kiếm nhị phân

Bây giờ chúng ta thiết kế lớp BSTree. Lớp này chỉ chứa một thành phần dữ liệu là con trỏ root trỏ tới gốc cây. Chúng ta sẽ cài đặt các hàm thực hiện các phép toán tập động (các hàm Search, Insert, Delete...) theo các thuật toán đệ quy đã trình bày trong mục 8.4.2. Nhưng các hàm này nằm trong giao diện của lớp BSTree, chúng không chứa tham biến root. Vậy làm thế nào để có thể viết các lời gọi đệ quy? Kỹ thuật được sử dụng ở đây là, chúng ta đưa vào các hàm ẩn tương ứng, các hàm này chứa một tham biến là con trỏ trỏ tới gốc các cây con. Sử dụng các hàm ẩn, việc cài đặt các hàm trong giao diện của lớp sẽ rất đơn giản, chỉ cần gọi các hàm ẩn tương ứng với tham số là con trỏ root. Lớp BSTree được khai báo trong hình 8.15.

```
template <class  Item>
class      BSTree
{
 public:
       BSTree () // Khởi tạo cây rỗng
           {root = NULL;}
       BSTree (const BSTree & T); // Hàm kiến tạo copy
       BSTree & Operator = (const BSTree & T); //Toán tử gán.
       virtual ~ BSTree () // Hàm hủy
           { MakeEmpty (root); }
        bool    Empty () const
           {return root = = NULL;}
```

```
        void     Insert (const Item & element);
        // Xen vào dữ liệu mới element
        void     Delete (keyType k);
        // Loại dữ liệu có khóa là k
        Item &   DeleteMin ()
        // Loại dữ liệu có khóa nhỏ nhất; hàm trả về
        // dữ liệu này, nếu cây không rỗng
        bool     Search (keyType k, Item & I)   const;
        // Tìm dữ liệu có khóa k, biến I lưu lại dữ liệu đó.
        // Hàm trả về true nếu tìm thấy, false nếu không.
        Item &    Min ()   const;
        // Hàm trả về dữ liệu có khóa nhỏ nhất, nếu cây không rỗng.
        Item &    Max ()   const;
        // Hàm trả về dữ liệu có khóa lớn nhất, nếu cây không rỗng.
        typedef    BSNode <Item>   Node;
private:
        Node *   root;
        // Các hàm ẩn phục vụ cho cài đặt các hàm public:
        void    MakeEmpty (Node * & P);
        // Làm cho cây gốc trỏ bởi P trở thành rỗng, thu hồi các
        // tế bào nhớ đã cấp phát cho các đỉnh của cây.
        void     Insert (const Item & element, Node* & P);
        // Xen dữ liệu mới element vòa cây gốc trỏ bởi P.
        void    Delete (keyType k, Node* & P);
        // Loại dữ liệu có khóa k khỏi cây gốc trỏ bởi P
        Item &   DeleteMin (Node* & P);
        // Loại dữ liệu có khóa nhỏ nhất khỏi cây gốc trỏ bởi P
        Item &    Min(Node * P) const;
        // Hàm trả về dữ liệu có khóa nhỏ nhất
        // trên cây gốc trỏ bởi P.
        Item &    Max( Node * P)  const;
        // Hàm trả về dư liệu có khoá lớn nhất.
        Item & CopyTree (Node* Q, Node* & P);
        // Copy cây gốc trỏ bởi Q thành cây gốc trỏ bởi P
};
```

Hình 8.15. Lớp cây tìm kiếm nhị phân

Sau đây chúng ta cài đặt các hàm thành phần của lớp BSTree. Các hàm trong mục public được cài đặt rất đơn giản, chỉ cần gọi các hàm ẩn tương ứng và thay tham biến P bởi con trỏ root. Chúng ta viết ra một số hàm

```
template <class Item>
BSTree <Item> : : BSTree (const BSTree <Item> & T)
{
     root = NULL;
     *this = T;
}

template <class Item>
BSTree<Item> & BSTree <Item>: : Operator = (const BSTree <Item>
& T)
{
    if (this ! = T)
    {
          MakeEmpty (root);
          if (T.root ! = NULL)
          CopyTree (&T, root);
     }
    return    * this;
}

template <clas Item>
void BSTree <Item> : : Insert (const Item & element)
{
    Insert (element, root);
}
```

Các hàm thành phần public còn lại được cài đặt tương tự, chỉ chứa một lời gọi hàm ẩn tương ứng, bạn đọc hãy tự viết ra các hàm đó.

Vấn đề còn lại của chúng ta là cài đặt các hàm ẩn. Tất cả các hàm ẩn này đều có một điểm chung là các hàm đệ quy, chứa một tham biến để gọi đệ quy, đó là biến con trỏ P trỏ tới gốc cây. Chúng ta sẽ cài đặt các hàm đệ quy này theo các thuật toán đệ quy đã được đưa ra trong mục 8.4.2.

```
template <class Item>
void    BSTree <Item> : : MakeEmpty (Node* & P)
```

```
{
    if (P ! = NULL)
      {
        MakeEmpty (P → left);
       MakeEmpty (P → right);
       delete P;
       P = NULL;
     }
}

template <class Item>
void   BSTree <Item> : : Insert (const Item & element, Node * & P)
{
    if (P = = NULL)
          P = new  Node<Item> (element);
    else if (element.key < P → data.key)
                Insert (element, P → left);
          else if (element.key > P → data.key)
                Insert (element, P → right);
}

template <class Item>
Item &   BSTree <Item> : : DeleteMin (Node* & P)
{
    assert (P ! = NULL); // Kiểm tra cây không rỗng
    if (P → left ! = NULL)
            return    DeleteMin (P → left);
    else {
            Item   element = P → data;
             Node <Item>*   Ptr = P;
             P = P → right;
            delete    Ptr;
             return    element;
          }
}

template <class Item>
void   BSTree <Item> : : Delete (keyType k, Node* & P)
{
   if (P! = NULL)
```

```
    {
      if (k < P → data.key)
            Delete (k, P → left);
      else if (k> P → data.key)
            Delete (k, P → right);
      else
      if ((P → left != NULL) & & (P → right != NULL))
      // Đỉnh P có đầy đủ 2 con.
            P → data = DeleteMin (P → right);
      else {
            Node <Item> *   Ptr = P;
            P = (P → left ! = NULL)? P → left : P → right;
            delete   Ptr;
          }
      }
}

template <class Item>
bool BSTree <Item>: : Search (keyType k, Item & I, Node* P)
{
   if (P ! = NULL)
      if (k = = P → data.key
        {
            I = P → data;
            return   true;
        }
       else if (k < P → data.key)
                  return   Search (k, I, P → left);
           else     return   Search (k, I, P → right);
  else {
          I = *(new Item);
          return   false;
       }
}
```

Khi sử dụng hàm Search, cần chú ý rằng nếu không tìm thấy (hàm trả về false) thì giá trị lưu trong biến I chỉ là giá trị giả tạo!

```
template <class Item>
```

```
Item & BSTree <Item> : : Min(Node*  P)
{
    assert (P! = NULL);
    if (P → left ! = NULL)
            return    Min(P → left);
   else     return    P → data;
}
template <class Item>
void BSTree <Item> : : CopyTree (Node*Q, Node* & P)
{
   if (Q = = NULL)
        P = NULL;
   else {
            P = new Node <Item> (Q → data);
            CopyTree (Q → left, P → left);
            CopyTree (Q → right, P → right);
         }
}
```

Trên đây chúng ta đã trình bày một cách thiết kế lớp BSTree, trong đó chúng ta đã cài đặt các phép toán tập động (Search, Insert, Delete, Min, Max, DeleteMin) sử dụng các thuật toán đệ quy. Đương nhiên, bạn cũng có thể cài đặt các phép toán tập động không đệ quy, và do đó bạn không cần đưa vào các hàm ẩn. Bạn hãy thiết kế và cài đặt lớp BSTree theo cách đó (bài tập).

8.6 THỜI GIAN THỰC HIỆN CÁC PHÉP TOÁN TẬP ĐỘNG TRÊN CÂY TÌM KIẾM NHỊ PHÂN

Trong mục 8.4.2 chúng ta đã chỉ ra rằng, thời gian thực hiện các phép toán Search, Insert và Delete trên cây tìm kiếm nhị phân là O(h), trong đó h là độ cao của cây. Tuy nhiên, cùng một tập dữ liệu chúng ta có thể lưu trong các cây tìm kiếm nhị phân có độ cao rất khác nhau. Chúng ta đã chỉ ra điều đó trong hình 8.11, ở đó chúng ta đã đưa ra ba cây tìm kiếm nhị phân cùng biểu diễn một tập gồm 6 dữ liệu với các giá trị khóa là 4, 1, 3, 7, 5, 9. Xuất phát từ cây tìm kiếm nhị phân rỗng, bằng cách sử dụng phép toán xen vào

cây một dãy các dữ liệu, ta sẽ nhận được một cây tìm kiếm nhị phân biểu diễn tập dữ liệu đó. Chẳng hạn, chúng ta có cây trong hình 8.11a, khi chúng ta xen vào cây rỗng lần lượt các dữ liệu với các khóa 4, 7, 5, 1, 3, 9; cây này có độ cao là 3. Nhưng nếu chúng ta xen vào cây rỗng lần lượt các dữ liệu với các giá trị khóa được sắp xếp theo thứ tự tăng dần, chúng ta sẽ nhận được cây trong hình 8.11c. Cây 8.11c có đặc điểm là tất cả các cây con trái của mỗi đỉnh đều rỗng, nó có độ cao là 6, cây này thực sự là một DSLK, việc tìm kiếm, xen, loại trên cây này là tìm kiếm, xen, loại trên DSLK. Như vậy, cây tìm kiếm nhị phân biểu diễn tập N dữ liệu trong trường hợp xấu nhất (khi cây suy biến thành DSLK), thời gian thực hiện các phép toán Search, Insert, Delete trên cây tìm kiếm nhị phân là O(N).

Một câu hỏi được đặt ra là, có thể xây dựng được cây tìm kiếm nhị phân biểu diễn tập N dữ liệu với độ cao nhỏ nhất có thể được bằng bao nhiêu? Có thể chứng minh được khẳng định sau đây:

Độ cao nhỏ nhất của cây nhị phân N đỉnh là

$h = [\log_2(N + 1)]$

trong đó, [x] ký hiệu số nguyên nhỏ nhất $\geq$ x, chẳng hạn [3, 41] = 4.

Cây nhị phân hoàn toàn, hoặc cây nhị phân cân bằng sẽ là các cây có độ cao ngắn nhất như trên. Như vậy trong trường hợp tốt nhất thì thời gian thực hiện các phép toán Search, Insert, Delete là O(logN).

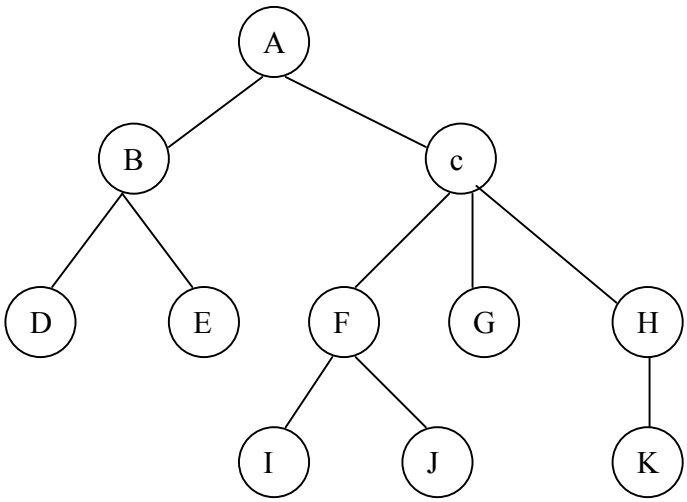
Cây tìm kiếm nhị phân ngẫu nhiên. Từ cây tìm kiếm nhị phân ban đầu rỗng, chúng ta xen vào cây N dữ liệu theo một thứ tự ngẫu nhiên; chúng ta sẽ gọi cây tìm kiếm nhị phân nhận được bằng cách đó là **cây tìm kiếm nhị phân được xây dựng một cách ngẫu nhiên**, hay gọn hơn: **cây tìm kiếm nhị phân ngẫu nhiên**.

Người ta đã chứng minh được rằng, độ cao trung bình của các cây tìm kiếm nhị phân ngẫu nhiêu với N đỉnh là O(logN). Từ kết quả này chúng ta suy ra rằng, thời gian trung bình của các phép toán Search, Insert, Delete, Min, Max,… trên cây tìm kiếm nhị phân ngẫu nhiên là O(logN).

BAI TẬP.

1. Hãy đưa ra cách biểu diễn cây bởi mảng. Mô tả CTDL biểu diễn cây theo cách đó bằng các khai báo trong C + +.

2. Cho cây:



   Hãy viết ra danh sách các đỉnh khi duyệt cây theo các thứ tự preorder, inorder và postorder.

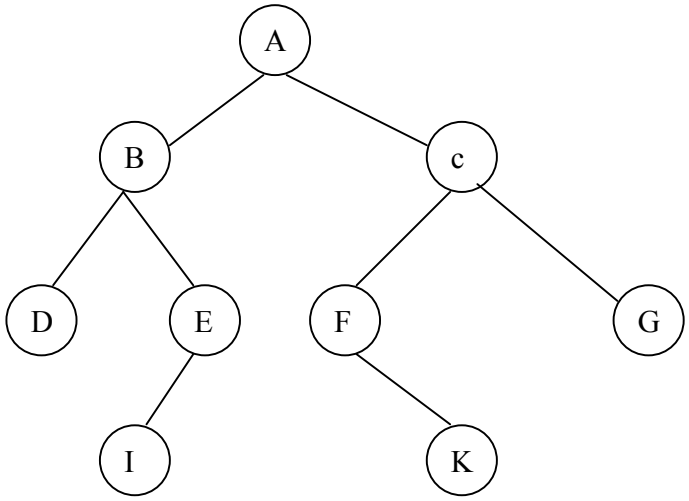
3. Giả sử cây được biểu diễn bằng cách sử dụng hai con trỏ firstChild và nextSibling. Bằng cách sử dụng ngăn xếp, hãy viết các hàm không đệ quy duyệt cây theo thứ tự inorder và postorder.

4. Mô tả và cài đặt lớp đỉnh cây nhị phân (class BinaryNode). Lớp này cần thoả mãn các đòi hỏi sau:
   - Chứa ba biến: data (lưu dữ liệu chứa trong đỉnh), hai con trỏ left và right.
   - Hàm kiến tạo để tạo ra một đỉnh chứa dữ liệu d cho trước, hai con trỏ left và right đều là NULL.
   - Chứa các hàm printPreorder( ),printInorder( ) và printPostorder( ) để in ra các dữ liệu lưu trong các đỉnh của cây con với gốc tại đỉnh đang xét theo các thứ tự trước, trong và sau.

5. Sử dụng lớp đỉnh cây nhị phân (bài tập 4), hãy đặc tả và cài đặt lớp cây nhị phân (class BinaryTree) theo các đòi hỏi và chỉ dẫn sau đây:

- Mục private chứa con trỏ root trỏ tới đỉnh gốc cây và các hàm ẩn cần thiết cho sự cài đặt các hàm trong mục public.
- Mục public chứa các hàm sau:
  - Hàm kiến tạo mặc định tạo ra cây rỗng.
  - Hàm kiến tạo ra cây chỉ có một đỉnh gốc chứa dữ liệu d cho trước.
  - Hàm kiến tạo copy.
  - Hàm huỷ.
  - Toán tử gán.
  - Hàm kiểm tra cây có rỗng không.
  - Hàm cho biết độ cao của cây.
  - Hàm cho biết số đỉnh trong cây.
  - Các hàm in ra các dữ liệu chứa trong các đỉnh của cây theo thứ tự preorder, inorder và postorder.

6. Hãy đặc tả và cài đặt ba lớp công cụ lặp trên cây nhị phân: PreIterator, InIterator và PostIterator, chứa các hàm công cụ giúp ta duyệt cây nhị phân theo các thứ tự trước, trong và sau. Mỗi lớp cần chứa hàm kiến tạo và bốn hàm sau:
   1) Hàm start( ) xác định đỉnh bắt đầu duyệt. Chẳng hạn, với cây nhị phân sau đây, nếu duyệt theo thứ tự trước thì bắt đầu là đỉnh A, còn nếu duyệt theo thứ tự trong và sau thì bắt đầu là đỉnh D.



   2) Hàm Valid( ) tương tự như trong lớp công cụ lặp trên danh sách liên kết.
   3) Hàm Current( ) trả về dữ liệu chứa trong đỉnh hiện thời.

4) Hàm Advance( ). Chẳng hạn, nếu đỉnh hiện thời là B, thì trong lớp PreIterator hàm Advance( ) cho ra đỉnh tiếp theo là D, còn trong lớp InIterator nó cho ra đỉnh tiếp theo là I, trong lớp PostIterator nó lại cho ra đỉnh tiếp theo là K.

Tương tự như trong lớp công cụ lặp trên danh sách liên kết, mỗi một trong ba lớp công cụ lặp trên cây nhị phân, cần chứa một con trỏ hằng trỏ tới gốc cây nhị phân, một con trỏ trỏ tới đỉnh hiện thời. Ngoài ra nó cần chứa một ngăn xếp để lưu các đỉnh trong quá trình duyệt. Chú ý rằng, cả ba lớp công cụ lặp trên cây nhị phân đều có giao diện và các biến thành phần giống nhau, chỉ có các hàm Start( ) và Advance( ) được cài đặt khác nhau để đảm bảo các đỉnh cây được thăm theo đúng thứ tự trước, trong và sau tương ứng.

7. Hãy vẽ ra cây tìm kiếm nhị phân được tạo thành bằng cách xen lần lượt các dữ liệu với các giá trị khoá là 5, 9, 2, 4, 1, 6, 10, 8, 3, 7, xuất phát từ cây rỗng. Sau đó hãy đưa ra cây kết quả khi loại gốc cây.

8. Trong lớp BSTree chúng ta đã cài đặt các hàm thực hiện các phép toán tập động bằng cách sử dụng các hàm ẩn đệ quy tương ứng. Hãy cài đặt các hàm đó (Search(k, I), Min( ), Max( ), Insert(element), Delete(k), DeleteMin( )) không đệ quy, và do đó không cần đưa vào các hàm ẩn.

9. Sử dụng cây tìm kiếm nhị phân, hãy đưa ra thuật toán sắp xếp mảng theo thứ tự khoá tăng dần, bằng cách sử dụng các phép toán Insert và DeleteMin.

10. Giả sử chúng ta có một tập dữ liệu, trong đó các dữ liệu khác nhau có thể có khoá bằng nhau. Hãy đưa ra cách cài đặt tập dữ liệu đó bởi cây tìm kiếm nhị phân. (Gợi ý: mỗi đỉnh cây chứa một danh sách các dữ liệu cùng khoá). Với CTDL cài đặt tập dữ liệu đã đưa ra đó, hãy cài đặt các hàm Search(d) (tìm dữ liệu d), Insert(d) (xen vào dữ liệu d) và Delete(d) (loại dữ liệu d).

11. Giả sử tập dữ liệu được lưu giữ dưới dạng cây tìm kiếm nhị phân. Bài toán tìm kiếm phạm vị được xác định như sau: Cho hai giá trị khoá $k_1 < k_2$, ta cần tìm tất cả các dữ liệu d mà $k_1 \le d.key \le k_2$. Hãy thiết kế và cài đặt thuật toán cho bài toán tìm kiếm phạm vi.

CHƯƠNG 9
BẢNG BĂM

Trong chương này, chúng ta sẽ nghiên cứu bảng băm. Bảng băm là cấu trúc dữ liệu được sử dụng để cài đặt KDLTT từ điển. Nhớ lại rằng, KDLTT từ điển là một tập các đối tượng dữ liệu được xem xét đến chỉ với ba phép toán tìm kiếm, xen vào và loại bỏ. Đương nhiên là chúng ta có thể cài đặt từ điển bởi danh sách, hoặc bởi cây tìm kiếm nhị phân. Tuy nhiên bảng băm là một trong các phương tiện hiệu quả nhất để cài đặt từ điển. Trong chương này, chúng ta sẽ đề cập tới các vấn đề sau đây:

- Phương pháp băm và hàm băm.
- Các chiến lược giải quyết sự va chạm.
- Cài đặt KDLTT từ điển bởi bảng băm.

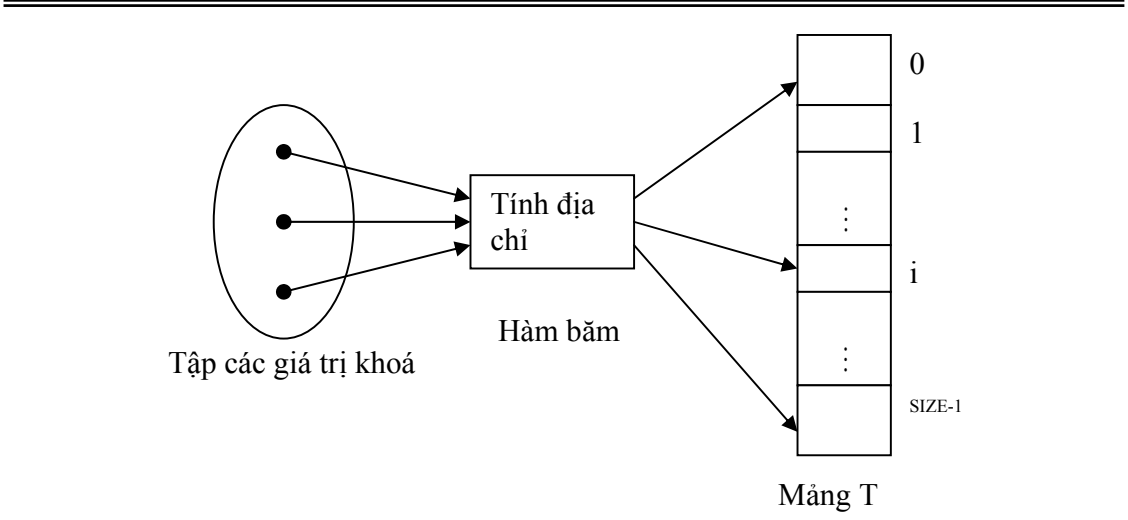
9.1 PHƯƠNG PHÁP BĂM

Vấn đề được đặt ra là, chúng ta có một tập dữ liệu, chúng ta cần đưa ra một CTDL cài đặt tập dữ liệu này sao cho các phép toán tìm kiếm, xen, loại được thực hiện hiệu quả. Trong các chương trước, chúng ta đã trình bày các phương pháp cài đặt KDLTT tập động (từ điển là trường hợp riêng của tập động khi mà chúng ta chỉ quan tâm tới ba phép toán tìm kiếm, xen, loại). Sau đây chúng ta trình bày một kỹ thuật mới để lưu giữ một tập dữ liệu, đó là phương pháp băm.

Nếu như các giá trị khoá của các dữ liệu là số nguyên không âm và nằm trong khoảng [0..SIZE-1], chúng ta có thể sử dụng một mảng data có cỡ SIZE để lưu tập dữ liệu đó. Dữ liệu có khoá là k sẽ được lưu trong thành phần data[k] của mảng. Bởi vì mảng cho phép ta truy cập trực tiếp tới từng thành phần của mảng theo chỉ số, do đó các phép toán tìm kiếm, xen, loại

được thực hiện trong thời gian O(1). Song đáng tiếc là, khoá có thể không phải là số nguyên, thông thường khoá còn có thể là số thực, là ký tự hoặc xâu ký tự. Ngay cả khoá là số nguyên, thì các giá trị khoá nói chung không chạy trong khoảng [0..SIZE-1].

Trong trường hợp tổng quát, khi khoá không phải là các số nguyên trong khoảng [0..SIZE-1], chúng ta cũng mong muốn lưu tập dữ liệu bởi mảng, để lợi dụng tính ưu việt cho phép truy cập trực tiếp của mảng. Giả sử chúng ta muốn lưu tập dữ liệu trong mảng T với cỡ là SIZE. Để làm được điều đó, với mỗi dữ liệu chúng ta cần định vị được vị trí trong mảng tại đó dữ liệu được lưu giữ. Nếu chúng ta đưa ra được cách tính chỉ số mảng tại đó lưu dữ liệu thì chúng ta có thể lưu tập dữ liệu trong mảng theo sơ đồ hình 9.1.



Hình 9.1. Lược đồ phương pháp băm.

Trong lược đồ hình 9.1, khi cho một dữ liệu có khoá là k, nếu tính địa chỉ theo k ta thu được chỉ số i, 0 <= i <= SIZE-1, thì dữ liệu sẽ được lưu trong thành phần mảng T[i].

Một hàm ứng với mỗi giá trị khoá của dữ liệu với một địa chỉ (chỉ số) của dữ liệu trong mảng được gọi là **hàm băm** (hash function). Phương pháp lưu tập dữ liệu theo lược đồ trên được gọi là phương pháp băm (hashing). Trong lược đồ 9.1, mảng T được gọi là **bảng băm** (hash table).

Như vậy, hàm băm là một ánh xạ h từ tập các giá trị khoá của dữ liệu vào tập các số nguyên {0,1,…, SIZE-1}, trong đó SIZE là cỡ của mảng dùng để lưu tập dữ liệu, tức là:

h : K → {0,1,…,SIZE-1}

với K là tập các giá trị khoá. Cho một dữ liệu có khoá là k, thì h(k) được gọi là **giá trị băm** của khoá k, và dữ liệu được lưu trong T[h(k)].

Nếu hàm băm cho phép ứng các giá trị khoá khác nhau với các chỉ số khác nhau, tức là nếu $k_1 \neq k_2$ thì $h(k_1) \neq h(k_2)$, và việc tính chỉ số h(k) ứng với mỗi khoá k chỉ đòi hỏi thời gian hằng, thì các phép toán tìm kiếm, xen, loại cũng chỉ cần thời gian O(1). Tuy nhiên, trong thực tế một hàm băm có thể ánh xạ hai hay nhiều giá trị khoá tới cùng một chỉ số nào đó. Điều đó có nghĩa là chúng ta phải lưu các dữ liệu đó trong cùng một thành phần mảng, mà mỗi thành phần mảng chỉ cho phép lưu một dữ liệu ! Hiện tượng này được gọi là **sự va chạm** (collision). Vấn đề đặt ra là, giải quyết sự va chạm như thế nào? Chẳng hạn, giả sử dữ liệu d_1 với khoá k_1 đã được lưu trong T[i], $i = h(k_1)$; bây giờ chúng ta cần xen vào dữ liệu d_2 với khoá k_2, nếu $h(k_2) = i$ thì dữ liệu d_2 cần được đặt vào vị trí nào trong mảng?

Như vậy, một hàm băm như thế nào thì được xem là tốt. Từ những điều đã nêu trên, chúng ta đưa ra các tiêu chuẩn để thiết kế một hàm băm tốt như sau:

1. Tính được dễ dàng và nhanh địa chỉ ứng với mỗi khoá.
2. Đảm bảo ít xảy ra va chạm.

9.2 CÁC HÀM BĂM

Trong các hàm băm được đưa ra dưới đây, chúng ta sẽ ký hiệu k là một giá trị khoá bất kỳ và SIZE là cỡ của bảng băm. Trước hết chúng ta sẽ xét trường hợp các giá trị khoá là các số nguyên không âm. Nếu không phải là trường hợp này (chẳng hạn, khi các giá trị khoá là các xâu ký tự), chúng ta chỉ cần chuyển đổi các giá trị khoá thành các số nguyên không âm, sau đó băm chúng bằng một phương pháp cho trường hợp khoá là số nguyên.

Có nhiều phương pháp thiết kế hàm băm đã được đề xuất, nhưng được sử dụng nhiều nhất trong thực tế là các phương pháp được trình bày sau đây:

9.2.1 Phương pháp chia

Phương pháp này đơn giản là lấy phần dư của phép chia khoá k cho cỡ bảng băm SIZE làm giá trị băm:

h(k) = k mod SIZE

Bằng cách này, giá trị băm h(k) là một trong các số 0,1,…, SIZE-1. Hàm băm này được cài đặt trong C++ như sau:

```
unsigned int hash(int k, int SIZE)
{
    return k % SIZE;
}
```

Trong phương pháp này, để băm một khoá k chỉ cần một phép chia, nhưng hạn chế cơ bản của phương pháp này là để hạn chế xảy ra va chạm, chúng ta cần phải biết cách lựa chọn cỡ của bảng băm. Các phân tích lý thuyết đã chỉ ra rằng, để hạn chế va chạm, khi sử dụng phương pháp băm này chúng ta nên lựa chọn SIZE là số nguyên tố, tốt hơn là số nguyên tố có dạng đặc biệt, chẳng hạn có dạng 4k+3. Ví dụ, có thể chọn SIZE = 811, vì 811 là số nguyên tố và 811 = 4 . 202 + 3.

9.2.2 Phương pháp nhân

Phương pháp chia có ưu điểm là rất đơn giản và dễ dàng tính được giá trị băm, song đối với sự va chạm nó lại rất nhạy cảm với cỡ của bảng băm. Để hạn chế sự va chạm, chúng ta có thể sử dụng phương pháp nhân, phương pháp này có ưu điểm là ít phụ thuộc vào cỡ của bảng băm.

Phương pháp nhân tính giá trị băm của khoá k như sau. Đầu tiên, ta tính tích của khoá k với một hằng số thực α, $0 < \alpha < 1$. Sau đó lấy phần thập phân của tích αk nhân với SIZE, phần nguyên của tích này được lấy làm giá trị băm của khoá k. Tức là:

$$h(k) = \lfloor (\alpha k - \lfloor \alpha k \rfloor) \, . \, SIZE \rfloor$$

(Ký hiệu $\lfloor x \rfloor$ chỉ phần nguyên của số thực x, tức là số nguyên lớn nhất <=x, chẳng hạn $\lfloor 3 \rfloor = 3, \lfloor 3.407 \rfloor = 3$).

Chú ý rằng, phần thập phân của tích αk, tức là $\alpha k - \lfloor \alpha k \rfloor$, là số thực dương nhỏ hơn 1. Do đó tích của phần thập phân với SIZE là số dương nhỏ hơn SIZE. Từ đó, giá trị băm h(k) là một trong các số nguyên 0,1,…, SIZE-1.

Để có thể phân phối đều các giá trị khoá vào các vị trí trong bảng băm, trong thực tế người ta thường chọn hằng số α như sau:

$$\alpha = \Phi^{-1} \approx 0{,}61803399$$

Chẳng hạn, nếu cỡ bảng băm là SIZE = 1024 và hằng số α được chọn như trên, thì với k = 1849970, ta có

$$h(k) = \lfloor (1024.(\alpha.1849970 - \lfloor \alpha.1849970 \rfloor)) \rfloor = 348.$$

9.2.3 Hàm băm cho các giá trị khoá là xâu ký tự

Để băm các xâu ký tự, trước hết chúng ta chuyển đổi các xâu ký tự thành các số nguyên. Các ký tự trong bảng mã ASCII gồm 128 ký tự được đánh số từ 0 đến 127, đo đó một xâu ký tự có thể xem như một số trong hệ đếm cơ số 128. Áp dụng phương pháp chuyển đổi một số trong hệ đếm bất kỳ sang một số trong hệ đếm cơ số 10, chúng ta sẽ chuyển đổi được một xâu ký tự

thành một số nguyên. Chẳng hạn, xâu "NOTE" được chuyển thành một số nguyên như sau:

"NOTE" → 'N'.128^3 + 'O'.128^2 + 'T'.128 + 'E' =

= 78.128^3 + 79.128^2 + 84.128 + 69

Vấn đề nảy sinh với cách chuyển đổi này là, chúng ta cần tính các luỹ thừa của 128, với các xâu ký tự tương đối dài, kết quả nhận được sẽ là một số nguyên cực lớn vượt quá khả năng biểu diễn của máy tính.

Trong thực tế, thông thường một xâu ký tự được tạo thành từ 26 chữ cái và 10 chữ số, và một vài ký tự khác. Do đó chúng ta thay 128 bởi 37 và tính số nguyên ứng với xâu ký tự theo luật Horner. Chẳng hạn, số nguyên ứng với xâu ký tự "NOTE" được tính như sau:

"NOTE" → 78.37^3 + 79.37^2 + 84.37 + 69=

= ((78.37 + 79).37 +84).37 +69

Sau khi chuyển đổi xâu ký tự thành số nguyên bằng phương pháp trên, chúng ta sẽ áp dụng phương pháp chia để tính giá trị băm. Hàm băm các xâu ký tự được cài đặt như sau:

```
unsigned   int   hash(const string &k, int SIZE)
{
        unsigned  int  value = 0;
        for (int i=0; i< k.length(); i++)
                value = 37 * value + k[i];
        return    value % SIZE;
}
```

9.3 CÁC PHƯƠNG PHÁP GIẢI QUYẾT VA CHẠM

Trong mục 9.2 chúng ta đã trình bày các phương pháp thiết kế hàm băm nhằm hạn chế xẩy ra va chạm. Tuy nhiên trong các ứng dụng, sự va chạm là không tránh khỏi. Chúng ta sẽ thấy rằng, cách giải quyết va chạm ảnh hưởng trực tiếp đến hiệu quả của các phép toán từ điển trên bảng băm. Trong mục này chúng ta sẽ trình bày hai phương pháp giải quyết va chạm. Trong phương pháp thứ nhất, mỗi khi xảy ra va chạm, chúng ta tiến hành thăm dò để tìm một vị trí còn trống trong bảng và đặt dữ liệu mới vào đó. Một phương pháp khác là, chúng ta tạo ra một cấu trúc dữ liệu lưu giữ tất cả các dữ liệu được băm vào cùng một vị trí trong bảng và "gắn" cấu trúc dữ liệu này vào vị trí đó trong bảng.

9.3.1 Phương pháp định địa chỉ mở

Trong phương pháp này, các dữ liệu được lưu trong các thành phần của mảng, mỗi thành phần chỉ chứa được một dữ liệu. Vì thế, mỗi khi cần xen một dữ liệu mới với khoá k vào mảng, nhưng tại vị trí h(k) đã chứa dữ liệu, chúng ta sẽ tiến hành thăm dò một số vị trí khác trong mảng để tìm ra một vị trí còn trống và đặt dữ liệu mới vào vị trí đó. Phương pháp tiến hành thăm dò để phát hiện ra vị trí trống được gọi là phương pháp định địa chỉ mở (open addressing).

Giả sử vị trí mà hàm băm xác định ứng với khoá k là i, i=h(k). Từ vị trí này chúng ta lần lượt xem xét các vị trí

$i_0, i_1, i_2, \ldots, i_m, \ldots$

Trong đó $i_0 = i$, $i_m (m=0,1,2,\ldots)$ là vị trí thăm dò ở lần thứ m. Dãy các vị trí này sẽ được gọi là dãy thăm dò. Vấn đề đặt ra là, xác định dãy thăm dò như thế nào? Sau đây chúng ta sẽ trình bày một số phương pháp thăm dò và phân tích ưu khuyết điểm của mỗi phương pháp.

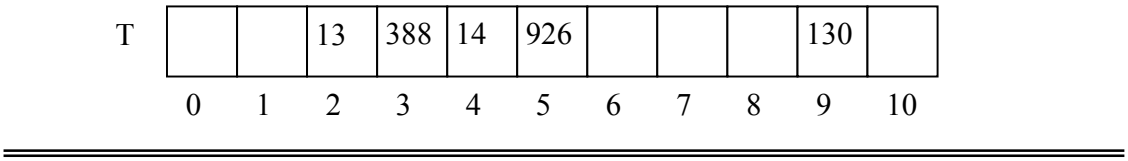
Thăm dò tuyến tính.

Đây là phương pháp thăm dò đơn giản và dễ cài đặt nhất. Với khoá k, giả sử vị trí được xác định bởi hàm băm là i=h(k), khi đó dãy thăm dò là

i , i+1, i+2 , …

Như vậy thăm dò tuyến tính có nghĩa là chúng ta xem xét các vị trí tiếp liền nhau kể từ vị trí ban đầu được xác định bởi hàm băm. Khi cần xen vào một dữ liệu mới với khoá k, nếu vị trí i = h(k) đã bị chiếm thì ta tìm đến các vị trí đi liền sau đó, gặp vị trí còn trống thì đặt dữ liệu mới vào đó.

Ví dụ. Giả sử cỡ của mảng SIZE = 11. Ban đầu mảng T rỗng, và ta cần xen lần lượt các dữ liệu với khoá là 388, 130, 13, 14, 926 vào mảng. Băm khoá 388, h(388) = 3, vì vậy 388 được đặt vào T[3]; h(130) = 9, đặt 130 vào T[9]; h(13) = 2, đặt 13 trong T[2]. Xét tiếp dữ liệu với khoá 14, h(14) = 3, xẩy ra va chạm (vì T[3] đã bị chiếm bởi 388), ta tìm đến vị trí tiếp theo là 4, vị trí này trống và 14 được đặt vào T[4]. Tương tự, khi xen vào 926 cũng xảy ra va chạm, h(926) = 2, tìm đến các vị trí tiếp theo 3, 4, 5 và 92 được đặt vào T[5]. Kết quả là chúng ta nhận được mảng T như trong hình 9.2.



Hình 9.2. Bảng băm sau khi xen vào các dữ liệu 38, 130, 13, 14 và 926

Bây giờ chúng ta xét xem, nếu lưu tập dữ liệu trong mảng bằng phương pháp định địa chỉ mở thì các phép toán tìm kiếm, xen, loại được tiến hành như thế nào. Các kỹ thuật tìm kiếm, xen, loại được trình bày dưới đây có thể sử dụng cho bất kỳ phương pháp thăm dò nào. Trước hết cần lưu ý rằng, để tìm, xen, loại chúng ta phải sử dụng cùng một phương pháp thăm dò, chẳng hạn thăm dò tuyến tính. Giả sử chúng ta cần tìm dữ liệu với khoá là k. Đầu tiên cần băm khoá k, giả sử h(k)=i. Nếu trong bảng ta chưa một lần nào thực hiện phép toán loại, thì chúng ta xem xét các dữ liệu chứa trong mảng tại vị trí i và các vị trí tiếp theo trong dãy thăm dò, chúng ta sẽ phát

hiện ra dữ liệu cần tìm tại một vị trí nào đó trong dãy thăm dò, hoặc nếu gặp một vị trí trống trong dãy thăm dò thì có thể dừng lại và kết luận dữ liệu cần tìm không có trong mảng. Chẳng hạn chúng ta muốn tìm xem mảng trong hình 9.2 có chứa dữ liệu với khoá là 47? Bởi vì h(47) = 3, và dữ liệu được lưu theo phương pháp thăm dò tuyến tính, nên chúng ta lần lượt xem xét các vị trí 3, 4, 5. Các vị trí này đều chứa dữ liệu khác với 47. Đến vị trí 6, mảng trống. Vậy ta kết luận 47 không có trong mảng.

Để loại dữ liệu với khoá k, trước hết chúng ta cần áp dụng thủ tục tìm kiếm đã trình bày ở trên để định vị dữ liệu ở trong mảng. Giả sử dữ liệu được lưu trong mảng tại vị trí p. Loại dữ liệu ở vị trí p bằng cách nào? Nếu đặt vị trí p là vị trí trống, thì khi tìm kiếm nếu thăm dò gặp vị trí trống ta không thể dừng và đưa ra kết luận dữ liệu không có trong mảng. Chẳng hạn, trong mảng hình 9.2, ta loại dữ liệu 388 bằng cách xem vị trí 3 là trống, sau đó ta tìm dữ liệu 926, vì h (926) = 2 và T[2] không chứa 926, tìm đến vị trí 3 là trống, nhưng ta không thể kết luận 926 không có trong mảng. Thực tế 926 ở vị trí 5, vì lúc đưa 926 vào mảng các vị trí 2, 3, 4 đã bị chiếm. Vì vậy để đảm bảo thủ tục tìm kiếm đã trình bày ở trên vẫn còn đúng cho trường hợp đã thực hiện phép toán loại, khi loại dữ liệu ở vị trí p chúng ta đặt vị trí p là vị trí đã loại bỏ. Như vậy, chúng ta quan niệm mỗi vị trí i trong mảng (0 <= i <= SIZE-1) có thể là vị trí trống (EMPTY), vị trí đã loại bỏ (DELETED), hoặc vị trí chứa dữ liệu (ACTIVE). Đương nhiên là khi xen vào dữ liệu mới, chúng ta có thể đặt nó vào vị trí đã loại bỏ.

Việc xen vào mảng một dữ liệu mới được tiến hành bằng cách lần lượt xem xét các vị trí trong dãy thăm dò ứng với mỗi khoá của dữ liệu, khi gặp một vị trí trống hoặc vị trí đã được loại bỏ thì đặt dữ liệu vào đó.

Sau đây là hàm thăm dò tuyến tính

```
int     Probing (int i, int m, int SIZE)
// SIZE là cỡ của mảng
// i là vị trí ban đầu được xác định bởi băm khoá k, i = h(k)
```

```
// hàm trả về vị trí thăm dò ở lần thứ  m= 0, 1, 2,…
{
        return    (i + m)  %   SIZE;
}
```

Phương pháp thăm dò tuyến tính có ưu điểm là cho phép ta xem xét tất cả các vị trí trong mảng, và do đó phép toán xen vào luôn luôn thực hiện được, trừ khi mảng đầy. Song nhược điểm của phương pháp này là các dữ liệu tập trung thành từng đoạn, trong quá trình xen các dữ liệu mới vào, các đoạn có thể gộp thành đoạn dài hơn. Điều đó làm cho các phép toán kém hiệu quả, chẳng hạn nếu i = h(k) ở đầu một đoạn, để tìm dữ liệu với khoá k chúng ta cần xem xét cả một đoạn dài.

Thăm dò bình phương

Để khắc phục tình trạng dữ liệu tích tụ thành từng cụm trong phương pháp thăm dò tuyến tính, chúng ta không thăm dò các vị trí kế tiếp liền nhau, mà thăm dò bỏ chỗ theo một quy luật nào đó.

Trong thăm dò bình phương, nếu vị trí ứng với khoá k là i = h(k), thì dãy thăm dò là

$i , i + 1^2, i + 2^2,\ldots , i + m^2,\ldots$

Ví dụ. Nếu cỡ của mảng SIZE = 11, và i = h(k) = 3, thì thăm dò bình phương cho phép ta tìm đến các địa chỉ 3, 4, 7, 1, 8 và 6.

Phương pháp thăm dò bình phương tránh được sự tích tụ dữ liệu thành từng đoạn và tránh được sự tìm kiếm tuần tự trong các đoạn. Tuy nhiên nhược điểm của nó là không cho phép ta tìm đến tất cả các vị trí trong mảng, chẳng hạn trong ví dụ trên, trong số 11 vị trí từ 0, 1, 2, …, 10, ta chỉ tìm đến các vị trí 3, 4, 7, 1, 8 và 6. Hậu quả của điều đó là, phép toán xen vào có thể không thực hiện được, mặc dầu trong mảng vẫn còn các vị trí không chứa dữ liệu. Chúng ta có thể dễ dàng chứng minh được khẳng định sau đây:

Nếu cỡ của mảng là số nguyên tố, thì thăm dò bình phương cho phép ta tìm đến một nửa số vị trí trong mảng. Cụ thể hơn là, các vị trí thăm dò $h(k) + m^2$ (mode SIZE) với $m = 0, 1, \ldots, \lfloor SIZE/2 \rfloor$ là khác nhau.

Từ khẳng định trên chúng ta suy ra rằng, nếu cỡ của mảng là số nguyên tố và mảng không đầy quá 50% thì phép toán xen vào luôn luôn thực hiện được.

Băm kép

Phương pháp băm kép (double hashing) có ưu điểm như thăm dò bình phương là hạn chế được sự tích tụ dữ liệu thành cụm; ngoài ra nếu chúng ta chọn cỡ của mảng là số nguyên tố, thì băm kép còn cho phép ta thăm dò tới tất cả các vị trí trong mảng.

Trong thăm dò tuyến tính hoặc thăm dò bình phương, các vị trí thăm dò cách vị trí xuất phát một khoảng cách hoàn toàn xác định trước và các khoảng cách này không phụ thuộc vào khoá. Trong băm kép, chúng ta sử dụng hai hàm băm h_1 và h_2:

- Hàm băm h_1 đóng vai trò như hàm băm h trong các phương pháp trước, nó xác định vị trí thăm dò đầu tiên
- Hàm băm h_2 xác định bước thăm dò.

Điều đó có nghĩa là, ứng với mỗi khoá k, dãy thăm dò là:

$$h_1(k) + m\, h_2(k), \text{ với } m = 0, 1, 2, \ldots$$

Bởi vì $h_2(k)$ là bước thăm dò, nên hàm băm h_2 phải thoả mãn điều kiện $h_2(k) \neq 0$ với mọi k.

Có thể chứng minh được rằng, nếu cỡ của mảng và bước thăm dò $h_2(k)$ nguyên tố cùng nhau thì phương pháp băm kép cho phép ta tìm đến tất cả các vị trí trong mảng. Khẳng định trên sẽ đúng nếu chúng ta lựa chọn cỡ của mảng là số nguyên tố.

Ví dụ. Giả sử SIZE = 11, và các hàm băm được xác định như sau:

$h_1(k) = k \% 11$

$h_2(k) = 1 + (k \% 7)$

với k = 58, thì bước thăm dò là $h_2(58) = 1 + 2 = 3$, do đó dãy thăm dò là: $h_1(58) = 3, 6, 9, 1, 4, 7, 10, 2, 5, 8, 0$. còn với k = 36, thì bước thăm dò là $h_2(36) = 1 + 1 = 2$, và dãy thăm dò là 3, 5, 7, 9, 0, 2, 4, 6, 8, 10.

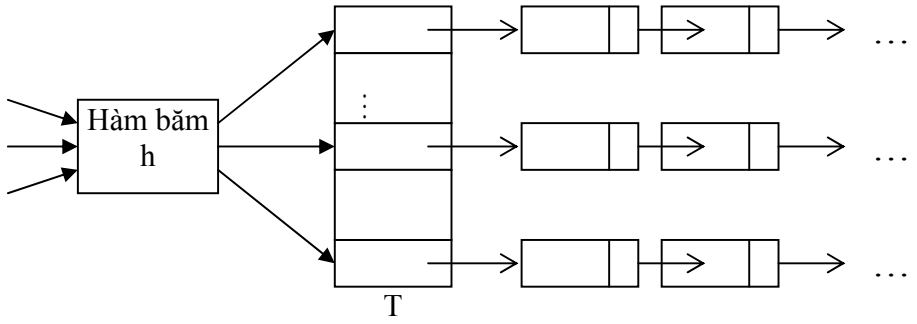
Trong các ứng dụng, chúng ta có thể chọn cỡ mảng SIZE là số nguyên tố và chọn M là số nguyên tố, M < SIZE, rồi sử dụng các hàm băm

$h_1(k) = k \% SIZE$

$h_2(k) = 1 + (k \% M)$

9.3.2 Phương pháp tạo dây chuyền

Một cách tiếp cận khác để giải quyết sự va chạm là chúng ta tạo một cấu trúc dữ liệu để lưu tất cả các dữ liệu được băm vào cùng một vị trí trong mảng. Cấu trúc dữ liệu thích hợp nhất là danh sách liên kết (dây chuyền). Khi đó mỗi thành phần trong bảng băm T[i], với i = 0, 1, …, SIZE – 1, sẽ chứa con trỏ trỏ tới đầu một DSLK. Cách giải quyết va chạm như trên được gọi là phương pháp **tạo dây chuyền** (separated chaining). Lược đồ lưu tập dữ liệu trong bảng băm sử dụng phương pháp tạo dây chuyền được mô tả trong hình 9.3.



Hình 9.3. Phương pháp tạo dây chuyền.

Ưu điểm của phương pháp giải quyết va chạm này là số dữ liệu được lưu không phụ thuộc vào cỡ của mảng, nó chỉ hạn chế bởi bộ nhớ cấp phát động cho các dây chuyền.

Bây giờ chúng ta xét xem các phép toán từ điển (tìm kiếm, xen, loại) được thực hiện như thế nào. Các phép toán được thực hiện rất dễ dàng, để xen vào bảng băm dữ liệu khoá k, chúng ta chỉ cần xen dữ liệu này vào đầu DSLK được trỏ tới bởi con trỏ T[h(k)]. Phép toán xen vào chỉ đòi hỏi thời gian O(1), nếu thời gian tính giá trị băm h(k) là O(1). Việc tìm kiếm hoặc loại bỏ một dữ liệu với khoá k được quy về tìm kiếm hoặc loại bỏ trên DSLK T[h(k)]. Thời gian tìm kiếm hoặc loại bỏ đương nhiên là phụ thuộc vào độ dài của DSLK.

Chúng ta có nhận xét rằng, dù giải quyết va chạm bằng cách thăm dò, hay giải quyết va chạm bằng cách tạo dây chuyền, thì bảng băm đều không thuận tiện cho sự thực hiện các phép toán tập động khác, chẳng hạn phép toán Min (tìm dữ liệu có khoá nhỏ nhất), phép toán DeleteMin (loại dữ liệu có khoá nhỏ nhất), hoặc phép duyệt dữ liệu.

Sau này chúng ta sẽ gọi bảng băm với giải quyết va chạm bằng phương pháp định địa chỉ mở là **bảng băm địa chỉ mở**, còn bảng băm giải quyết va chạm bằng cách tạo dây chuyền là **bảng băm dây chuyền**.

9.4 CÀI ĐẶT BẢNG BĂM ĐỊA CHỈ MỞ

Trong mục này chúng ta sẽ nghiên cứu sự cài đặt KDLTT từ điển bởi bảng băm địa chỉ mở. Chúng ta sẽ giả thiết rằng, các dữ liệu trong từ điển có kiểu Item nào đó, và chúng chứa một trường dùng làm khoá tìm kiếm (trường key), các giá trị khoá có kiểu keyType. Ngoài ra để đơn giản cho viết ta giả thiết rằng, có thể truy cập trực tiếp trường key. Như đã thảo luận trong mục 9.3.1, trong bảng băm T, mỗi thành phần T[i], 0 <= i <= SIZE -1 , sẽ chứa hai biến: biến data để lưu dữ liệu và biến state để lưu trạng thái của vị trí i, trạng thái của vị trí i có thể là rỗng (EMPTY), có thể chứa dữ liệu (ACTIVE), hoặc có thể đã loại bỏ (DELETED). Chúng ta sẽ cài đặt KDLTT bởi lớp OpenHash phụ thuộc tham biến kiểu Item, lớp này sử dụng một hàm băm Hash và một hàm thăm dò Probing đã được cung cấp. Lớp OpenHash được khai báo trong hình 9.4.

```
typedef   int   keyType;
const   int   SIZE = 811;
template <class  Item>
class       OpenHash
{
public:
        OpenHash(); // khởi tạo bảng băm rỗng.
        bool     Search(keyType   k, Item & I)  const;
        // Tìm dữ liệu có khoá là k.
        // Hàm trả về true (false) nếu tìm thấy (không tìm thấy).
        // Nếu tìm kiếm thành công, biến I ghi lại dữ liệu cần tìm.
        void     Insert(const Item &  object, bool &  Suc)
        // Xen vào dữ liệu object. biến Suc nhận giá trị true
        // nếu phép xen thành công, và false nếu thất bại.
        void     Delete(keyType   k);
        // Loại khỏi bảng băm dữ liệu có khoá k.
        enum   stateType {ACTIVE, EMPTY, DELETED};
private:
        struct     Entry
        {
           Item     data;
           stateType    state;
        }
```

```
    Entry   T[SIZE];
    bool    Find(keyType k, int & index, int & index1)   const;
    // Hàm thực hiện thăm dò tìm dữ liệu có khoá k.
    // Nếu thành công, hàm trả về true và biến index ghi lại chỉ
    // số tại đó chứa dữ liệu.
    // Nếu thất bại, hàm trả về false và biến index1 ghi lại
    // chỉ số ở trạng thái EMPTY hoặc DELETED nếu thăm dò
    // phát hiện ra.
};
```

Hình 9.4. Định nghĩa lớp OpenHash.

Bây giờ chúng ta cài đặt các hàm thành phần của lớp OpenHash. Hàm kiến tạo bảng băm rỗng được cài đặt như sau:

```
template <class Item>
OpenHash<Item>::OpenHash()
{
    for ( int i = 0 ; i < SIZE ; i++ )
        T[i].state = EMPTY;
}
```

Chú ý rằng, các phép toán tìm kiếm, xen, loại đều cần phải thực hiện thăm dò để phát hiện ra dữ liệu cần tìm hoặc để phát hiện ra vị trí rỗng (hoặc bị trí đã loại bỏ) để đưa vào dữ liệu mới. Vì vậy, trong lớp OpenHash chúng ta đã đưa vào hàm ẩn Find. Sử dụng hàm Find ta dễ dàng cài đặt được các hàm Search, Insert và Delete. Trước hết chúng ta cài đặt hàm Find. Trong hàm Find khi mà quá trình thăm dò phát hiện ra vị trí rỗng thì có nghĩa là bảng không chứa dữ liệu cần tìm, song trước khi đạt tới vị trí rỗng có thể ta

đã phát hiện ra các vị trí đã loại bỏ, biến index1 sẽ ghi lại vị trí đã loại bỏ đầu tiên đã phát hiện ra . Còn nếu phát hiện ra vị trí rỗng, nhưng trước đó ta không gặp vị trí đã loại bỏ nào, thì biến index1 sẽ ghi lại vị trí rỗng. Hàm Find được cài đặt như sau:

```
template <class Item>
bool OpenHash<Item>::Find(keyType k, int & index, int & index1)
{
        int i = Hash(k);
        index = 0;
        index1 = i;
        for (int m = 0 ; m <  SIZE ; m++)
        {
                int   n = Probing(i,m); // vị trí thăm dò ở lần thứ m.
                if (T[n].state = = ACTIVE && T[n].data.key = = k )
                  {
                     index = n;
                     return   true;
                   }
                else if (T[n].state = = EMPTY)
                     {
                       if (T[index1].state != DELETED)
                              index1 = n;
                       return   false;
                     }
```

```
            else if (T[n].state = = DELETED && T[index1].state
                                        != DELETED)
                index1 = n;
        }
        return   false; // Dừng thăm dò mà vẫn không tìm ra dữ liệu
                        //  và cũng không phát hiện ra vị trí rỗng.
    }
```

Sử dụng hàm Find, các hàm tìm kiếm, xen, loại được cài đặt như sau:

```
template<class Item>
bool   OpenHash<Item>:: Search(keyType k, Item &I)
{
        int   ind, ind1;
        if (Find(k,ind,ind1))
            {
               I  =  T[ind].data;
               return    true;
            }
        else
            {
               I  = *(new Item); // giá trị của I là giả
               return   false;
            }
}
```

```
template <class Item>
void   OpenHash<Item>::  Insert(const Item & object, bool &  Suc)
{
        int    ind, ind1;
        if (!Find(object.key, ind, ind1))
        if (T[ind1].state = = DELETED || T[ind1].state = = EMPTY)
            {
                T[ind1].data  =  object;
                T[ind1].state  = ACTIVE;
                Suc  =  true;
             }
        else    Suc  =  false;
}

template <class Item>
void   OpenHash<Item>:: Delete(keyType   k)
{
        int   ind, ind1;
        if (Find(k, ind, ind1))
                T[ind].state = DELETED;
}
```

Trên đây chúng ta đã cài đặt bảng băm địa chỉ mở bởi mảng có cỡ cố định. Hạn chế của cách này là, phép toán Insert có thể không thực hiện được do mảng đầy hoặc có thể mảng không đầy nhưng thăm dò không phát hiện ra vị trí rỗng hoặc vị trí đã loại bỏ để đặt dữ liệu vào. Câu hỏi đặt ra là,

chúng ta có thể cài đặt bởi mảng động như chúng ta đã làm khi cài đặt KDLTT tập động (xem 4.4). Câu trả lời là có, tuy nhiên cài đặt bảng băm bởi mảng động sẽ phức tạp hơn, vì các lý do sau:

- Cỡ của mảng cần là số nguyên tố, do đó chúng ta cần tìm số nguyên tố tiếp theo SIZE làm cỡ của mảng mới.
- Hàm băm phụ thuộc vào cỡ của mảng, chúng ta không thể sao chép một cách đơn giản mảng cũ sang mảng mới như chúng ta đã làm trước đây, mà cần phải sử dụng hàm Insert để xen từng dữ liệu của bảng cũ sang bảng mới

9.5 CÀI ĐẶT BẢNG BĂM DÂY CHUYỀN

Trong mục này chúng ta sẽ cài đặt KDLTT từ điển bởi bảng băm dây chuyền. Lớp ChainHash phụ thuộc tham biến kiểu Item với các giả thiết như trong mục 9.4. Lớp này được định nghĩa trong hình 9.5.

```
template<class Item>
class       ChainHash
{
        public:
             static   const  int  SIZE =  811;
             ChainHash(); // Hàm kiến tạo mặc định.
             ChainHash(const ChainHash  &  Table);//Kiến tạo copy.
             ~ChainHash();//Hàm huỷ
             void Operator=(const ChainHash & Table);//Toán tử gán
             // Các phép toán từ điển:
             bool   Search(keyType k, Item & I)  const;
```

```
        void    Insert(const Item & object, bool & Suc);
        void    Delete(keyType   k);
    private:
        struct      Cell
        {
            Item    data;
            Cell*   next;
        }; // Cấu trúc tế bào trong dây chuyền.
        Cell*   T[SIZE];// Mảng các con trỏ trỏ  đầu các dây chuyền
};
```

Hình 9.5. Lớp ChainHash.

Sau đây chúng ta cài đặt các hàm thành phần của lớp ChainHash. Để khởi tạo ra bảng băm rỗng, chúng ta chỉ cần đặt các thành phần trong mảng T là con trỏ NULL.

Hàm kiến tạo mặc định như sau:

```
template<class Item>
ChainHash<Item>::ChainHash()
{
    for ( int i = 0 ; i < SIZE ; i++ )
        T[i] = NULL;
}
```

Các hàm tìm kiếm, xen, loại được cài đặt rất đơn giản, sau khi băm chúng ta chỉ cần áp dụng các kỹ thuật tìm kiếm, xen, loại trên các DSLK. Các hàm Search, Insert và Delete được xác định dưới đây:

```
template<class Item>
bool ChainHash<Item>::Search(keyType k, Item & I)
{
        int  i = Hash(k);
        Cell*  P = T[i];
        while (P ! = NULL)
                if (P →data.key = = k)
                        {
                           I = P →data;
                            return   true;
                        }
                else P= P →next;
        I = *(new Item); // giá trị của biến I là giả.
        return    false;
}
template<class Item>
void   ChainHash<Item>::Insert(const Item &  object, bool &  Suc)
{
        int   i = Hash(k);
        Cell*   P = new Cell;
        If (P != NULL)
        {
                P→data = object;
                P→next = T[i];
```

```
            T[i] = P;  //Xen vào đầu dây chuyền.
            Suc = true;
        }
        else    Suc  = false;
}

template <class Item>
void   ChainHash<Item>::Delete(keyType  k)
{
        int   i = Hash(k);
        Cell*   P;
        If (T[i] != NULL)
        If (T[i]→data.key = = k)
            {
                P = T[i];
                T[i] = T[i]→next;
                delete   P;
            }
        else
            {
                P = T[i];
                Cell*  Q = P→next;
                while (Q != NULL)
                    if (Q→data.key = = k)
```

```
                {
                    P→next = Q→next;
                    delete   Q;
                Q = NULL;
                }
            else
                {
                    P  =  Q;
                    Q  =  Q→next;
                }
        }
    }
```

Ưu điểm lớn nhất của bảng băm dây chuyền là, phép toán Insert luôn luôn được thực hiện, chỉ trừ khi bộ nhớ để cấp phát động đã cạn kiệt. Ngoài ra, các phép toán tìm kiếm, xen, loại, trên bảng băm dây chuyền cũng rất đơn giản. Tuy nhiên, phương pháp này tiêu tốn bộ nhớ giành cho các con trỏ trong các dây chuyền.

9.6 HIỆU QUẢ CỦA PHƯƠNG PHÁP BĂM

Trong mục này, chúng ta sẽ phân tích thời gian thực hiện các phép toán từ điển (tìm kiếm, xen, loại) khi sử dụng phương pháp băm. Trong trường hợp xấu nhất, khi mà hàm băm băm tất cả các giá trị khoá vào cùng một chỉ số mảng để tìm kiếm chẳng hạn, chúng ta cần xem xét từng dữ liệu giống như tìm kiếm tuần tự, vì vậy thời gian các phép toán đòi hỏi là O(N), trong đó N là số dữ liệu.

Sau đây chúng ta sẽ đánh giá thời gian trung bình cho các phép toán từ điển. Đánh giá này dựa trên giả thiết hàm băm phân phối đều các khoá

vào các vị trí trong bảng băm (uniform hashing). Chúng ta sẽ sử dụng một tham số α, được gọi là **mức độ đầy** (load factor). Mức độ đầy α là tỷ số giữa số dữ liệu hiện có trong bảng băm và cỡ của bảng, tức là

$$\alpha = \frac{N}{SIZE}$$

trong đó, N là số dữ liệu trong bảng. Rõ ràng là, khi α tăng thì khả năng xẩy ra va chạm sẽ tăng, điều này kéo theo thời gian tìm kiếm sẽ tăng. Như vậy hiệu quả của các phép toán phụ thuộc vào mức độ đầy α. Khi cỡ mảng cố định, hiệu quả sẽ giảm nếu số dữ liệu N tăng lên. Vì vậy, trong thực hành thiết kế bảng băm, chúng ta cần đánh giá số tối đa các dữ liệu cần lưu để lựa chọn cỡ SIZE sao cho α đủ nhỏ. Mức độ đầy α không nên vượt quá 2/3.

Thời gian tìm kiếm cũng phụ thuộc sự tìm kiếm là thành công hay thất bại. Tìm kiếm thất bại đòi hỏi nhiều thời gian hơn tìm kiếm thành công, chẳng hạn trong bảng băm dây chuyền chúng ta phải xem xét toàn bộ một dây chuyền mới biết không có dữ liệu trong bảng.

D.E. Knuth (trong The art of computer programming, vol3) đã phân tích và đưa ra các công thức đánh giá hiệu quả cho từng phương pháp giải quyết va chạm như sau.

Thời gian tìm kiếm trung bình trên bảng băm địa chỉ mở sử dụng thăm dò tuyến tính. Số trung bình các lần thăm dò cho tìm kiếm xấp xỉ là:

Tìm kiếm thành công $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$

Tìm kiếm thất bại $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Trong đó α là mức độ đầy và $\alpha < 1$.

Ví dụ. Nếu cỡ bảng băm SIZE = 811, bảng chứa N = 649 dữ liệu, thì mức độ đầy là $\alpha = \frac{649}{811} \approx 80\%$. Khi đó, để tìm kiếm thành công một dữ liệu, trung bình chỉ đòi hỏi xem xét 3 vị trí mảng, vì

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) = \frac{1}{2}\left(1+\frac{1}{1-0,8}\right) = 3$$

Thời gian tìm kiếm trung bình trên bảng băm địa chỉ mở sử dụng thăm dò bình phương (hoặc băm kép). Số trung bình các lần thăm dò cho tìm kiếm được đánh giá là

Tìm kiếm thành công $\quad \frac{-\ln(1-\alpha)}{\alpha}$

Tìm kiếm thất bại $\quad \frac{1}{1-\alpha}$

Phương pháp thăm dò này đòi hỏi số lần thăm dò ít hơn phương pháp thăm dò tuyến tính. Chẳng hạn, giả sử bảng đầy tới 80%, để tìm kiếm thành công trung bình chỉ đòi hỏi xem xét 2 vị trí mảng,

$$\frac{-\ln(1-\alpha)}{\alpha} = \frac{-\ln(1-0,8)}{0,8} = \frac{1,6}{0,8} = 2$$

Thời gian tìm kiếm trung bình trên bảng băm dây chuyền. Trong bảng băm dây chuyền, để xen vào một dữ liệu mới, ta chỉ cần đặt dữ liệu vào đầu một dây chuyền được định vị bởi hàm băm. Do đó, thời gian xen vào là O(1).

Để tìm kiếm (hay loại bỏ) một dữ liệu, ta cần xem xét các tế bào trong một dây chuyền. Đương nhiên là dây chuyền càng ngắn thì tìm kiếm càng nhanh. Độ dài trung bình của một dây chuyền là $\frac{N}{SIZE} = \alpha$ (với giả thiết hàm băm phân phối đều).

Khi tìm kiếm thành công, chúng ta cần biết dây chuyền có rỗng không, rồi cần xem xét trung bình là một nửa dây chuyền. Do đó, số trung bình các vị trí cần xem xét khi tìm kiếm thành công là

$1+\frac{\alpha}{2}$

Nếu tìm kiếm thất bại, có nghĩa là ta đã xem xét tất cả các tế bào trong một dây chuyền nhưng không thấy dữ liệu cần tìm, do đó số trung bình các vị trí cần xem xét khi tìm kiếm thất bại là α.

Tóm lại, hiệu quả của phép toán tìm kiếm trên bảng băm dây chuyền là:

Tìm kiếm thành công $1+\frac{1}{\alpha}$

Tìm kiếm thất bại α

Mức độ đầy α	Bảng băm địa chỉ mở với thăm dò tuyến tính	Bảng băm địa chỉ mở với thăm dò bình phương	Bảng băm dây chuyền
0,5	1,50	1,39	1,25
0,6	1,75	1,53	1,30
0,7	2,17	1,72	1,35
0,8	3,00	2,01	1,40
0,9	5,50	2,56	1,45
1,0			1,50
2,0			2,00
3,0			3,00

Hình 9.6. Số trung bình các vị trí cần xem xét trong tìm kiếm thành công.

Các con số trong bảng ở hình 9.6, và thực tiễn cũng chứng tỏ rằng, phương pháp băm là phương pháp rất hiệu quả để cài đặt từ điển.

BÀI TẬP.

1. Hãy cài đặt hàm băm sử dụng phương pháp nhân (mục 9.2.2).

2. Hãy cài đặt hàm thăm dò sử dụng phương pháp băm kép.

3. Giả sử cỡ của bảng băm là SIZE = s và $d_1, d_2, \ldots, d_{s-1}$ là hoán vị ngẫu nhiên của các số 1, 2, …, s-1. Dãy thăm dò ứng với khoá k được xác định như sau:
   $i_0 = i = h(k)$
   $i_m = (i + d_i) \% SIZE , 1 \le m \le s - 1$
   Hãy cài đặt hàm thăm dò theo phương pháp trên.

4. Cho cỡ bảng băm SIZE = 11. Từ bảng băm rỗng, sử dụng hàm băm chia lấy dư, hãy đưa lần lượt các dữ liệu với khoá:
   32 , 15 , 25 , 44 , 36 , 21
   vào bảng băm và đưa ra bảng băm kết quả trong các trường hợp sau:
   a. Bảng băm được chỉ mở với thăm dò tuyến tính.
   b. Bảng băm được chỉ mở với thăm dò bình phương.
   c. Bảng băm dây chuyền.

5. Từ các bảng băm kết quả trong bài tập 4, hãy loại bỏ dữ liệu với khoá là 44 rồi sau đó xen vào dữ liệu với khoá là 65.

6. Bảng băm chỉ cho phép thực hiện hiệu quả các phép toán tập động nào? Không thích hợp cho các phép toán tập động nào? Hãy giải thích tại sao?

7. Giả sử khoá tìm kiếm là từ tiếng Anh. Hãy đưa ra ít nhất 3 cách thiết kế hàm băm. Bình luận về các cách thiết kế đó theo các tiêu chuẩn hàm băm tốt.

CHƯƠNG 10

HÀNG ƯU TIÊN

Trong các chương trước chúng ta đã nghiên cứu KDLTT từ điển. Từ điển là một tập đối tượng dữ liệu, mỗi đối tượng được gắn với một giá trị khóa, và các phép toán tìm kiếm, xen, loại trên từ điển được tiến hành khi được cung cấp một giá trị khóa. Trong chương này chúng ta sẽ đưa ra KDLTT hàng ưu tiên. Hàng ưu tiên khác với từ điển ở chỗ, thay cho giá trị khóa, mỗi đối tượng dữ liệu trong hàng ưu tiên được gắn với một giá trị ưu tiên, và chúng ta chỉ quan tâm tới việc tìm kiếm và loại bỏ đối tượng có giá trị ưu tiên nhỏ nhất. Nội dung chính của chương này là:

- Đặc tả KDLTT hàng ưu tiên.
- Trình bày phương pháp cài đặt hàng ưu tiên bởi cây thứ tự bộ phận (heap).
- Đưa ra một ứng dụng của hàng ưu tiên trong nén dữ liệu và xây dựng mã Huffman.

10.1 KIỂU DỮ LIỆU TRỪU TƯỢNG HÀNG ƯU TIÊN

Giả sử chúng ta cần bảo lưu một cơ sở dữ liệu gồm các bản ghi về các bệnh nhân đến khám và chữa bệnh tại một bệnh viện. Các bệnh nhân khi đến bệnh viện sẽ được đưa vào cơ sở dữ liệu này. Nhưng các bác sĩ khám cho bệnh nhân sẽ không phục vụ người bệnh theo thứ tự ai đến trước sẽ được khám trước (như cách tổ chức dữ liệu theo hàng đợi). Mỗi bệnh nhân sẽ được cấp một giá trị ưu tiên và các bác sĩ sẽ gọi vào phong khám bệnh nhân có giá trị ưu tiên nhỏ nhất (người được ưu tiên trước hết tại thời điểm đó). Nhiều hoàn cảnh khác (chẳng hạn, hệ máy tính phục vụ nhiều người sử dụng) cũng đòi hỏi chúng ta cần phải tổ chức một tập đối tượng dữ liệu theo giá trị ưu tiên sao cho các thao tác tìm đối tượng và loại đối tượng có giá trị

ưu tiên nhỏ nhất được thực hiện hiệu quả. Điều đó dẫn đến sự hình thành KDLTT hàng ưu tiên.

Hàng ưu tiên (priority queue) được xem là một tập các đối tượng dữ liệu, mỗi đối tượng có một giá trị ưu tiên. Thông thường các giá trị ưu tiên có thể là các số nguyên, các số thực, các ký tự…; điều quan trọng là chúng ta có thể so sánh được các giá trị ưu tiên . Trên hàng ưu tiên chúng ta chi quan tâm tới các phép toán sau đây:

1. Insert (P,x). Xen vào hàng ưu tiên P đối tượng x.
2. FindMin(P). Hàm trả về đối tượng trong P có giá trị ưu tiên nhỏ nhất (đối tượng được ưu tiên nhất). Phép toán này đòi hỏi P không rỗng
3. DeleteMin(P). Loại bỏ và trả về đối tượng có giá trị ưu tiên nhỏ nhất trong P. P cũng cần phải không rỗng.

Hàng ưu tiên được sử dụng trong các hoàn cảnh tương tự như hoàn cảnh đã nêu trên, tức là khi ta cần quản lý sự phục vụ theo mức độ ưu tiên. Hàng ưu tiên còn được sử dụng để thiết kế các thuật toán trong nhiều ứng dụng. Cuối chương này chúng ta sẽ đưa ra một ứng dụng: sử dụng hàng ưu tiên để thiết kế thuật toán xây dựng mã Huffman.

Trong các mục tiếp theo chúng ta sẽ đề cập tới các phương pháp cài đặt hàng ưu tiên.

10.2 CÁC PHƯƠNG PHÁP ĐƠN GIẢN CÀI ĐẶT HÀNG ƯU TIÊN

Trong mục này chúng ta sẽ thảo luận cách sự dụng các cấu trúc dữ liệu đã quen biết: danh sách, cây tìm kiếm nhị phân để cài đặt hàng ưu tiên và thảo luận về hiệu quả của các phép toán hàng ưu tiên trong các cách cài đặt đơn giản đó.

10.2.1 Cài đặt hàng ưu tiên bởi danh sách

Cài đặt hàng ưu tiên đơn giản nhất là biểu diễn hàng ưu tiên dưới dạng một danh sách được sắp hoặc không được sắp theo giá trị ưu tiên. Đương nhiên là danh sách đó có thể được lưu trong mảng hay DSLK.

Nếu chung ta cài đặt hàng ưu tiên bởi danh sách các phần tử được sắp xếp theo thứ tự ưu tiên tăng dần (hoặc giảm dần) thì phần tử có giá trị ưu tiên nhỏ nhất nằm ở một đầu danh sách, và do đó các phép toán FindMin và DeleteMin chỉ cần thời gian O(1). Nhưng để thực hiện phép toán Insert chúng ta cần tìm vị trí thích hợp trong danh sách để đặt phần tử mới sao cho tính chất được sắp của danh sách được bảo tồn. Vì vậy phép toán Insert đòi hỏi thời gian O(n), trong đó n là độ dài của danh sách.

Nếu chúng ta cài đặt hàng ưu tiên bởi danh sách các phần tử theo một thứ tự tùy ý, thì khi cần xen vào hàng ưu tiên một phần tử mới chúng ta chỉ cần đưa nó vào đuôi danh sách, do đó phép toán Insert chỉ cần thời gian O(1). Song để tìm và loại phần tử có giá trị ưu tiên nhỏ nhất chúng ta cần phải duyệt toàn bộ danh sách, và vì vậy các phép toán FindMin và DeleteMin cần thời gian O(n).

10.2.2 Cài đặt hàng ưu tiên bởi cây tìm kiếm nhị phân

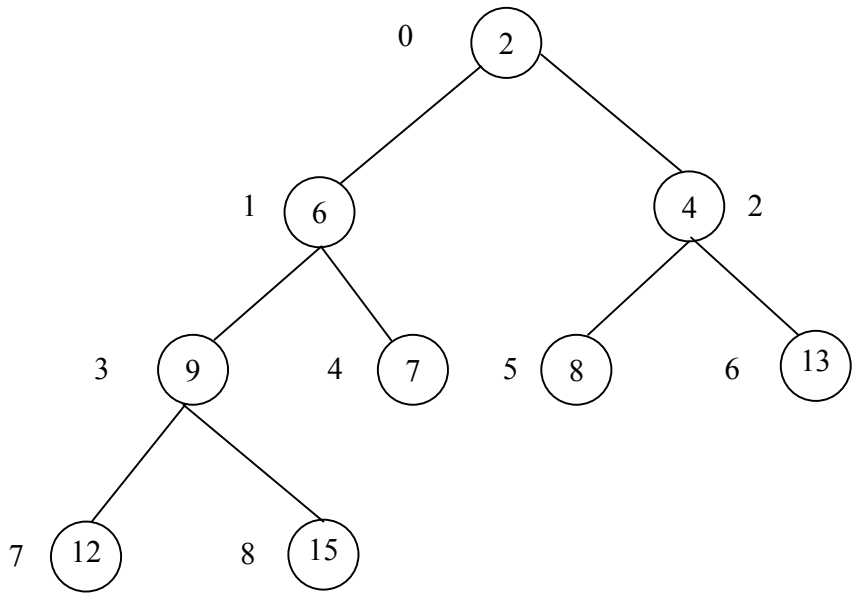
Trong mục 8.4.3 chúng ta đã cài đặt KDLTT tập động bởi cây tìm kiếm nhị phân. Cần lưu ý rằng, có thể xem hàng ưu tiên như là một dạng đặc biệt của tập động khi mà ta lấy giá trị ưu tiên của mỗi phần tử làm khóa và chi quan tâm tới các phép toán Insert, FindMin, DeleteMin. Vì vậy đương nhiên chúng ta có thể cài đặt hàng ưu tiên bởi cây tìm kiếm nhị phân. Từ lớp BSTree (trong hình 8.16), bằng thừa kế bạn có thể đưa ra lớp cài đặt hàng ưu tiên. Hiệu quả của các phép toán hàng ưu tiên trong cách cài đặt này đã được thảo luận trong mục 8.5. Trong trường hợp tốt nhất thời gian thực hiện các phép toán hàng ưu tiên là O(log n), trường hợp xấu nhất là O(n).

Trong mục sau đây chúng ta sẽ đưa ra một cách cài đặt mới: cài đặt hàng ưu tiên bởi cây thứ tự bộ phận. Với cách cài đặt này, thời gian thực hiện của các phép toán hàng ưu tiên luôn luôn là O(log n).

10.3 CÂY THỨ TỰ BỘ PHẬN

Trong mục 8.4 chúng ta đã nghiên cứu CTDL cây tìm kiếm nhị phân. Trong cây tìm kiếm nhị phân, các khóa của dữ liệu chứa trong các đỉnh của cây cần phải thỏa mãn tính chất thứ tự: khóa của một đỉnh bất kỳ lớn hơn khóa của các đỉnh trong cây con trái và nhỏ hơn khóa của các đỉnh trong cây con phải. Trong cây thứ tự bộ phận, chỉ đòi hỏi các khóa chứa trong các đỉnh thỏa mãn tính chất thứ tự bộ phận: Khóa của một đỉnh bất kỳ nhỏ hơn hoặc bằng khóa của các đỉnh con của nó. Mặc dù vậy, CTDL cây thứ tự bộ phận cho phép ta thực hiện hiệu quả các phép toán hàng ưu tiên.

Cây thứ tự bộ phận (partially ordered tree, hoặc heap) là một cây nhị phân hoàn toàn và thỏa mãn tính chất thứ tự bộ phận. Ví dụ, cây nhị phân trong hình 10.1 là cây thứ tự bộ phận.



Hình 10.1 Cây thứ tự bộ phận

Chúng ta cần lưu ý đến một số đặc điểm của cây thứ tự bộ phận. Trước hết, nó cần phải là cây nhị phân hoàn toàn (xem định nghĩa trong mục 8.3), tức là tất cả các mức của cây đều không thiếu đỉnh nào, trừ mức thấp nhất được lấp đầy kể từ bên trái. Tính chất này cho phép ta cài đặt cây thứ tự bộ phận bởi mảng. Mặc khác, từ tính chất thứ tự bộ phận ta rút ra đặc điểm sau: Các giá trị khóa nằm trên đường đi từ gốc tới các nút lá tạo thành một dãy không giảm. Chẳng hạn, dãy các giái trị khóa từ gốc (đỉnh 0) tới đỉnh 8 là 3, 6, 9, 15. Điều đó có nghĩa là, dữ liệu có khóa nhỏ nhất được lưu trong gốc của cây thứ tự bộ phận.

Độ cao của cây thứ tự bộ phận. Giả sử n là số đỉnh của cây nhị phân hoàn toàn có độ cao h. Dễ dàng thấy rằng, $2^{h-1} < n \leq 2^h - 1$ và do đó $2^{h-1} < n+1 \leq 2^h$. Từ đó ta suy ra rằng, độ cao h của cây thứ tự bộ phận có n đỉnh bằng [log(n+1)].

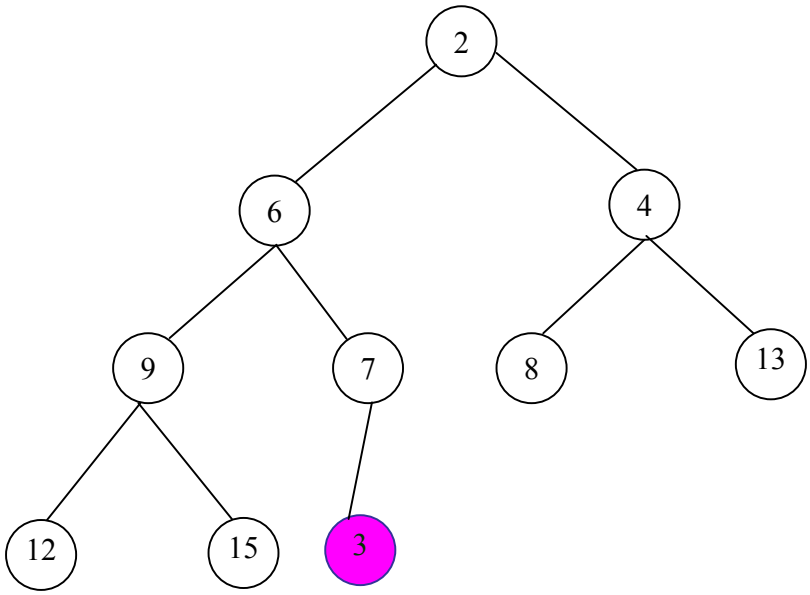
Sau đây chúng ta xét xem các phép toán hàng ưu tiên được thực hiện như thế nào khi hàng ưu tiên biểu diễn bởi cây thứ tự bộ phận (với giá trị khóa của dữ liệu chứa trong mỗi đỉnh được lấy là giá trị ưu tiên).

10.3.1 Các phép toán hàng ưu tiên trên cây thứ tự bộ phận

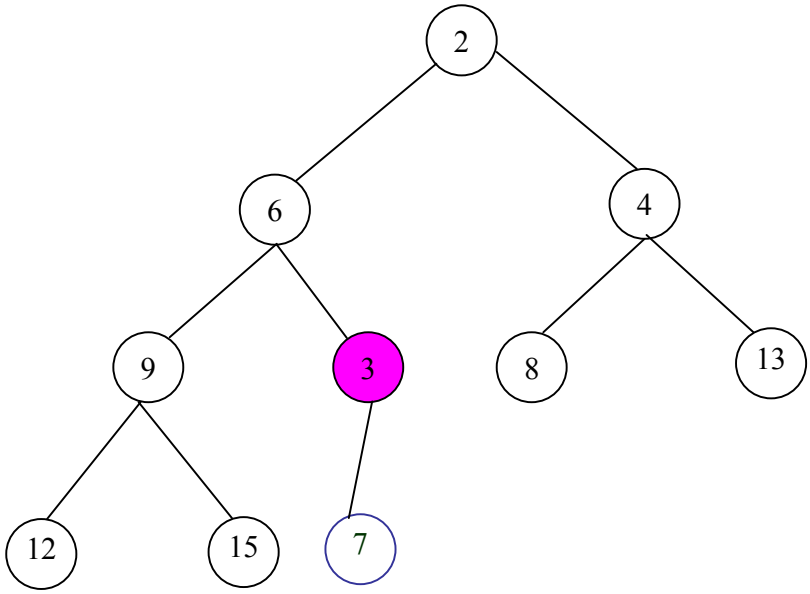
Như trên đã nhận xét, đối tượng dữ liệu có khóa nhỏ nhất được lưu trong gốc của cây thứ tự bộ phận, do đó phép toán FindMin được thực hiện dễ dàng và chỉ đòi hỏi thời gian O(1). Khó khăn khi thực hiện phép toán Insert và DeleteMin là ở chỗ chúng ta phải biến đổi cây như thế nào để sau khi xen vào (hoặc loại bỏ) cây vẫn còn thỏa mãn tính chất thứ tự bộ phận.

Phép toán Insert. Giả sử chúng ta cần xen vào cây một đỉnh mới chứa dữ liệu x. Để đảm bảo tính chất là cây nhị phân hoàn toàn, đỉnh mới được đặt là đỉnh ngoài cùng bên phải ở mức thấp nhất. Chẳng hạn, từ cây trong hình 10.1, khi thêm vào đỉnh mới với khóa là 3 ta nhận được cây trong hình 10.2 a. Nhưng cây nhận được có thể không thỏa mãn tính chất thứ tự bộ phận, vì đỉnh mới thêm vào có thể có khóa nhỏ hơn khóa của đỉnh cha nó,

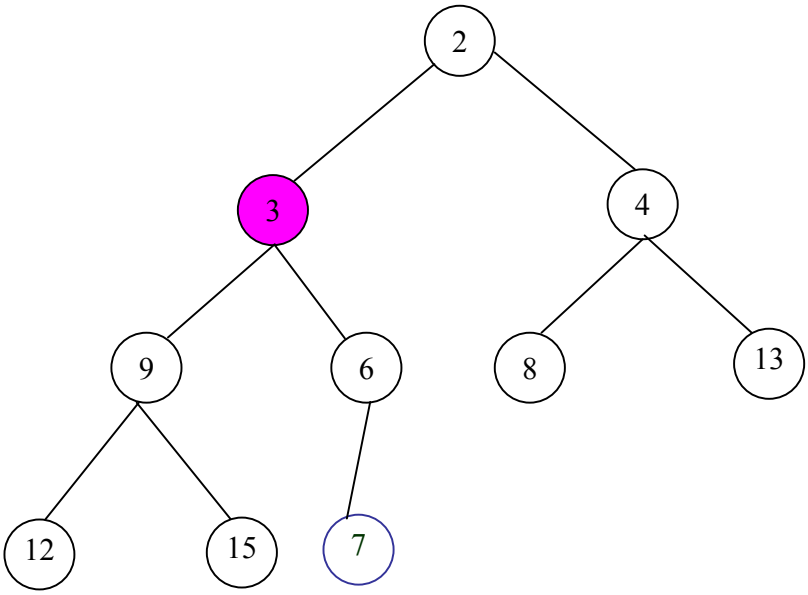
chẳng hạn cây trong hình 10.2 a, đỉnh mới thêm vào có khóa là 3, đỉnh cha nó có khóa là 7. Chúng ta có thể sắp xếp lại dữ liệu chứa trong các đỉnh để cây thỏa mãn tính chất thứ tự bộ phận bằng thuật toán đơn giản sau. Đi từ đỉnh mới thêm vào lên gốc cây, mỗi khi ta đang ở một đỉnh có khóa nhỏ hơn khóa của đỉnh cha nó thì ta hoán vị dữ liệu chuá trong hai đinh đó và đi lên đỉnh cha. Quá trình đi lên sẽ dừng lại khi ta đạt tới một đỉnh có khóa lớn hơn khóa của đỉnh cha nó, hoặc cùng lắm là khi đạt tới gốc cây. Ví dụ, với cây trong hình 10.2 a, sự đổi chỗ các dữ liệu được minh họa trong hình 10.2 a-c. Dễ dàng chứng minh được rằng, quá trình đi từ đỉnh mới thêm vào lên gốc và tiến hành đổi chỗ các dữ liệu như trên sẽ cho cây kết quả là cây thứ tự bộ phận (bài tập).



(a)



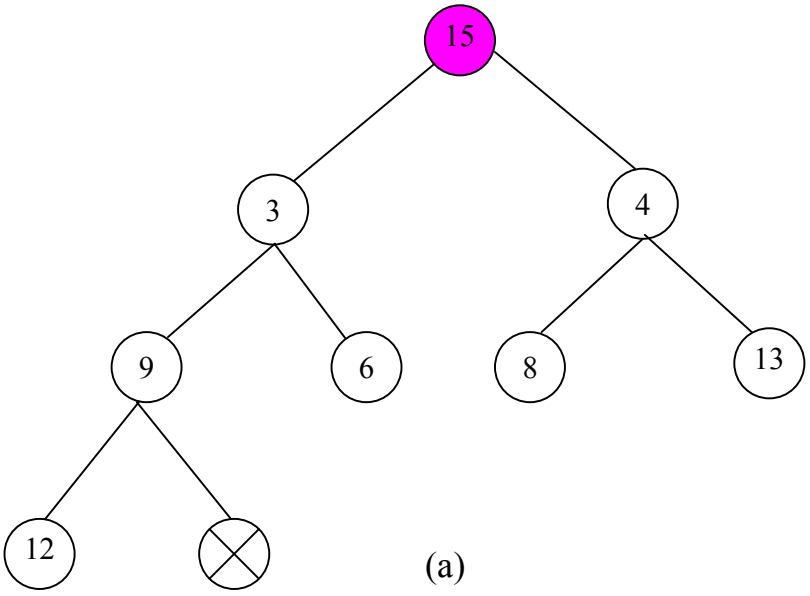
(b)



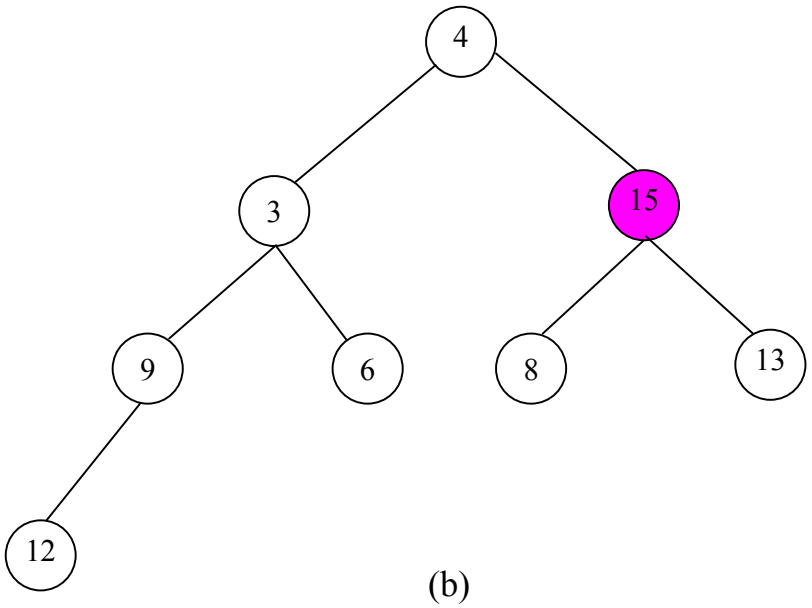
(c)

Hình 10.2. Xen một đỉnh mới vào cây thứ tự bộ phận.

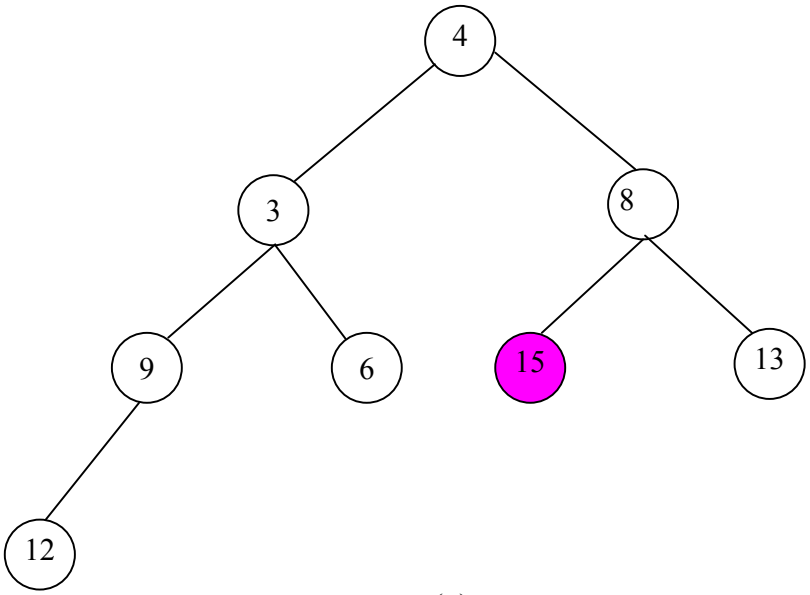
Phép toán DeleteMin. Chúng ta cần loại bỏ khỏi cây thứ tự bộ phận đỉnh có khóa nhỏ nhất, như trên đã nhận xét, đó là gốc cây. Thuật toán loại gốc cây được tiến hành qua hai bước. Đầu tiên, ta đem dữ liệu ở đỉnh ngoài cùng bên phải ở mức thấp nhất lưu vào gốc cây và loại đỉnh đó. Chẳng han, từ cây trong hình 10.1, ta nhận được cây trong hình 10.3a . Hành động trên mới chỉ đảm bảo đỉnh chứa dữ liệu có khóa nhỏ nhất đã bị loại và cây vẫn là cây nhị phân hoàn toàn. Song tính chất thứ tự bộ phận có thể bị vi phạm, vì khóa của đỉnh gốc bây giờ có thể lớn hơn khóa của các đỉnh con của nó, chẳng hạn như cây trong hình 10.3 a. Bước tiếp theo, ta đi từ gốc cây xuống lá và tiến hành hoán vị các dữ liệu chứa trong các đỉnh cha và con khi cần thiết. Giả sử ta đang ở đỉnh p, con trái của p (nếu có) là v_l, con phải (nếu có) là v_r . Giả sử đỉnh p có khóa lớn hơn khóa của ít nhất một trong hai đỉnh con là v_l và v_r. Khi đó, nếu khóa của đỉnh v_l nhỏ hơn khóa của đỉnh v_r thì ta hoán vị các dữ liệu trong p và v_l và đi xuống đỉnh v_l; nếu ngược lại, ta hoán vị các dữ liệu trong p và v_r, rồi đi xuống đỉnh v_r. Quá trình đi xuống sẽ dừng lại khi ta đạt tới một đỉnh có khóa nhỏ hơn khóa của các đỉnh con, hoặc cùng lắm là khi đạt tới một đỉnh lá. Ví dụ, sự đổi chỗ các dữ liệu chứa trong các đỉnh của cây trong hình 10.3a được thể hiện trong các hình 10.3 a-c.



(a)



(b)



(c)

Hình 10.3. Loại đỉnh chứa khóa nhỏ nhất.

Chúng ta đã chỉ ra rằng, độ cao của cây hoàn toàn xấp xỉ bằng log(n), trong đó n là số đỉnh của cây. Các phép toán Insert và DeleteMin chỉ tiêu tốn thời gian để hoán vị các dữ liệu chứa trong các đỉnh nằm trên đường đi từ lá lên gốc (đối với phép toán Insert) hoặc trên đường đi từ gốc xuống lá (phép toán DeleteMin). Vì vậy, thời gian thực hiện các phép toán Insert và DeleteMin trên cây thứ tự bộ phận là O(log n), trong đó n lá số dữ liệu trong hàng ưu tiên.

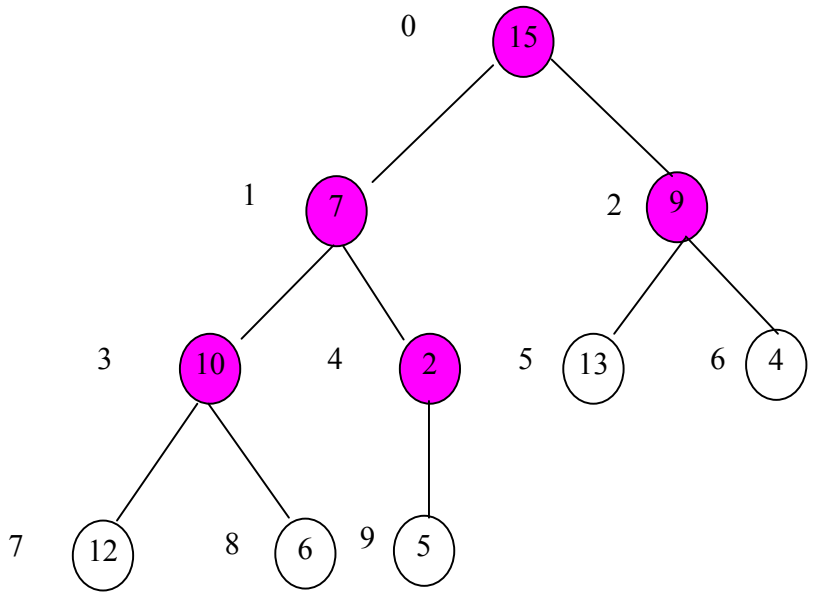
10.3.2 Xây dựng cây thứ tự bộ phận

Trong các ứng dụng, thông thường chúng ta cần phải xây dựng một cây thứ tự bộ phận từ n đối tượng dữ liệu đã có. Một cách đơn giản là, xuất phát từ cây rỗng, áp dụng phép toán Insert để xen vào các đỉnh mới chứa các dữ liệu đã cho. Mỗi phép toán xen vào đòi hỏi thời gian O(log n) và do đó toàn bộ quá trình trên cần thời gian O(n log n). Sau đây chúng ta đưa ra một cách xây dựng khác, chỉ đòi hỏi thời gian là O(n).

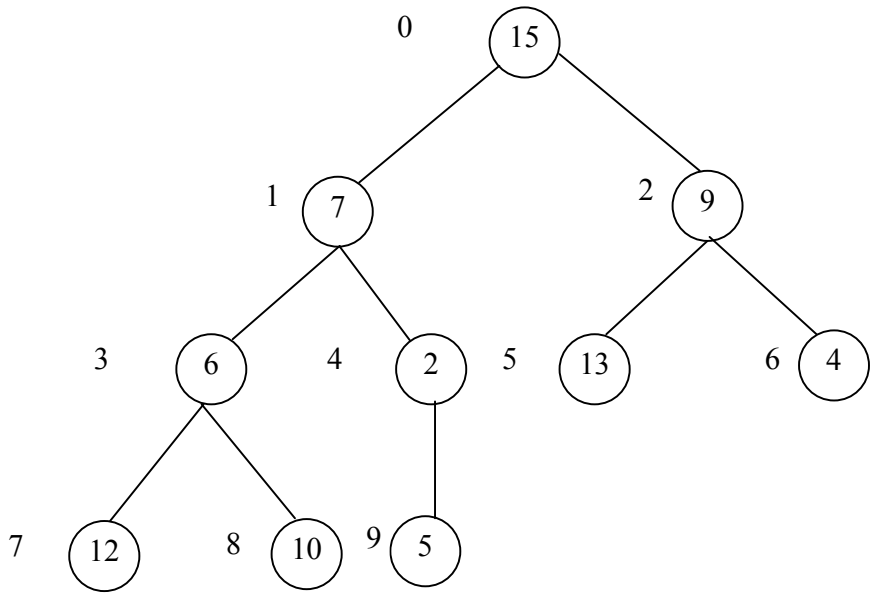
Như đã trình bày ở trên, thuật toán DeleteMin gồm hai bước. Sau bước thứ nhất, ta nhận được cây (chẳng hạn, cây trong hình 10.3 a), cây này có hai cây con trái và phải của gốc đã là cây thứ tự bộ phận, chỉ trừ tại gốc có thể tính chất thứ tự bộ phận không thỏa mãn. Trong bước thứ hai, ta đi từ gốc xuống và tiến hành hoán vị các dữ liệu trong cặp đỉnh cha, con khi cần thiết để cho cây thỏa mãn tính chất thứ tự bộ phận. Chúng ta sẽ nói tới bước này như là bước đẩy dữ liệu chứa trong gốc cây xuống vị trí thích hợp (siftdown). Điều đó là cơ sở của thuật toán xây dựng cây sau đây. Thuật toán này gồm hai giai đoạn:

1. Từ dãy n dữ liệu, ta xây dựng cậy nhị phân hoàn toàn, các dữ liệu được lần lượt đưa vào các đỉnh của cây theo thứ tự từ trên xuống dưới và trong cùng một mức thì từ trái qua phải, bắt đầu từ gốc được đánh số là 0. Chẳng hạn, từ 10 dữ liệu với các khóa là 15 , 7 , 9 , 10 , 2 , 13 , 4 , 12 , 6 và 5, ta xây dựng được cây nhị phân hoàn toàn như trong hình 10.4 a.

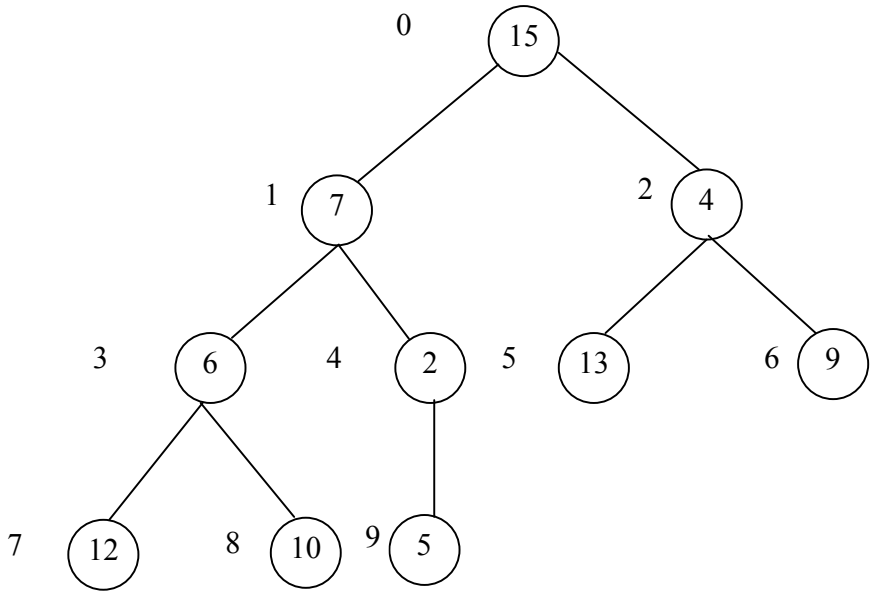
2. Xử lý các đỉnh i của cây bắt đầu từ đỉnh được đánh số là $\lfloor n/2-1 \rfloor$, sau mỗi lần i được giảm đi 1 cho tới khi i bằng 0. Chẳng hạn, trong hình 10.4 a, chúng ta lần lượt xử lý các đỉnh i = 4, 3, 2, 1, 0. Mỗi khi xử lý tới đỉnh i, thì đỉnh i đã là gốc của cây nhị phân hoàn toàn với hai cây con đã là cây thứ thứ tự bộ phận. Vì vậy, chúng ta chỉ cần tiến hành đẩy dữ liệu chứa trong đỉnh i xuống vị trí thích hợp, như chúng ta đã làm trong bước hai của thuật toán DeleteMin. Ví dụ, xét cây trong hình 10.4 a. Bắt đầu từ đỉnh i=4, đỉnh này có khóa nhỏ hơn khóa của đỉnh con nó, nên không cần làm gì cả. Với i=3, ta đẩy dữ liệu trong đỉnh 3 xuống đỉnh 8 để có cây trong hình 10.4 c. Với i = 2, đẩy dữ liệu ở đỉnh 2 xuống đỉnh 6, ta có cây trong hình 10.4c. Với i=1, đẩy dữ liệu ở đỉnh 1 xuống đỉnh 9, nhận được cây trong hình 10.4d. Với i=0, ta cần đẩy dữ liệu ở gốc cây xuống đỉnh 9 và nhận được cây thứ tự bộ phận như trong hình 10.4 e.



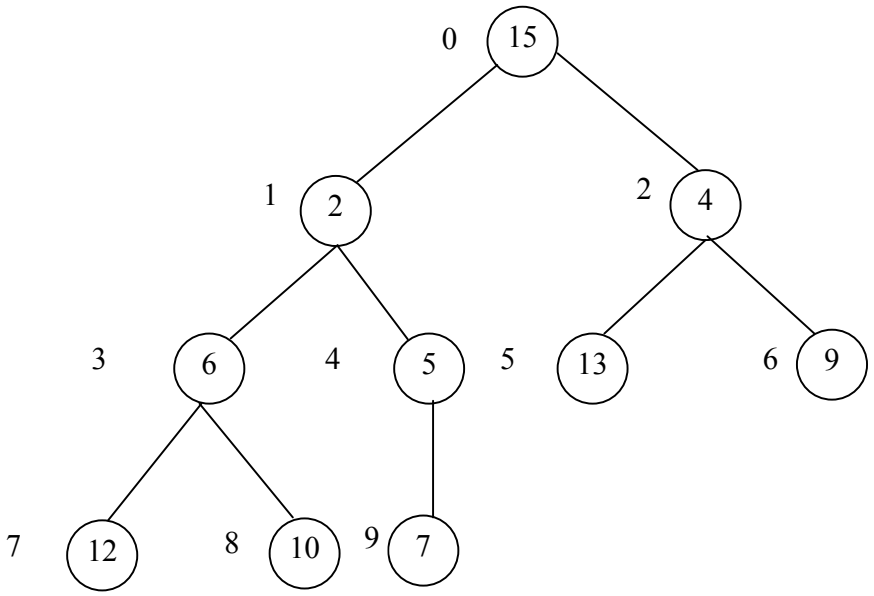
(a)



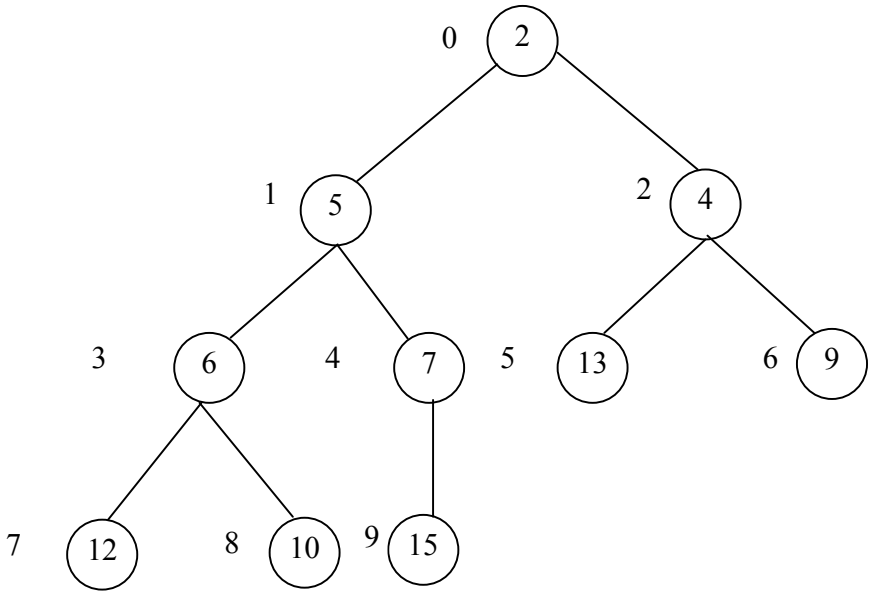
(b)



(c)



(d)



(e)

Hình 10.4. Xây dựng cây thứ tự bộ phận.

Bây giờ chúng ta chứng tỏ rằng, thuật toán xây dựng cây thứ tự bộ phận trên chỉ đòi hỏi thời gian O(n). Rõ ràng rằng, giai đoạn xây dựng cây hoàn toàn từ n dữ liệu chỉ cần thời gian O(n). Vì vậy chúng ta chỉ cần chỉ ra rằng giai đoạn hai cũng chỉ cần thời gian O(n). Trong giai đoạn hai, chúng ta tiến hành xử lý các đỉnh của cây theo thứ tự ngược lại từ đỉnh được đánh số là n-1 tới đỉnh 0. Với mỗi đỉnh ta tiến hành đẩy dữ liệu chứa trong đỉnh đó tới vị trí thích hợp, vị trí này cùng lắm là một lá của cây. Vì vậy thời gian cho xử lý mỗi đỉnh là độ cao của đỉnh đó. Chúng ta tính tổng độ cao của tất cả các đỉnh trong cây. Đỉnh gốc có độ cao h, ở mức 1 có 2 đỉnh với độ cao là h-1,…. Ở mức i có 2^i đỉnh với độ cao là h-i, do đó tổng độ cao của các đỉnh là:

$$f(n) = 2^0.h + 2^1.(h - 1) + \ldots + 2^{h-2}.2 + 2^{h-1}.1$$

$$= \sum_{i=0}^{h-1} 2^i (h-i)$$

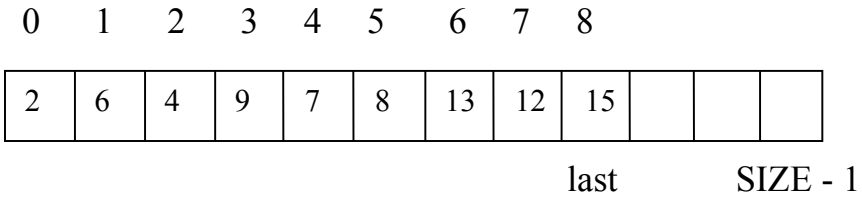
$$= h\sum_{i=0}^{h-1} 2^i - \sum_{i=0}^{h-1} i2^i$$

$$= h(2^h - 1) - [(h - 2)2^h + 2] = 2^{h+1} - h - 2$$

Chúng ta đã biết rằng, độ cao h của cây hoàn toàn n đỉnh là $h \approx \log(n+1)$. Từ đó, $f(n) \approx 2n - \log(n+1)$ hay f(n) = O(n). Như vậy giai đoạn hai chỉ đòi hỏi thời gian O(n).

10.4 CÀI ĐẶT HÀNG ƯU TIÊN BỞI CÂY THỨ TỰ BỘ PHẬN

Trong mục 8.3 chúng ta đã đưa ra cách cài đặt cây nhị phân hoàn toàn bởi mảng. Cây thứ tự bộ phận là cây nhị phân hoàn toàn, do đó có thể cài đặt cấy thứ tự bộ phận bởi mảng. Chẳng hạn, cây thứ tự bộ phận trong hình 10.1 có thể được lưu trong mảng như sau:



Chúng ta sẽ sử dụng một mảng data với cỡ là SIZE để lưu các dữ liệu chứa trong các đỉnh của cây thứ tự bộ phận. Dữ liệu chứa trong đỉnh được đánh số i sẽ được lưu trong thành phần data[i] của mảng, i sẽ chạy từ 0 đến chỉ số sau cùng là last, và không gian trong mảng từ chỉ số last+1 tới SIZE-1 là không gian chưa sử dụng. Nhớ lại rằng, với cách lưu này, nếu một đỉnh được lưu trong data[i] thì đỉnh con trái của nó được lưu trong data[2*i+1], còn đỉnh con phải được lưu trong data[2*i+2] và đỉnh cha của nó được lưu trong data[(i-1)/2].

Sử dụng phương pháp biểu diễn hàng ưu tiên bởi cây thứ tự bộ phận (khóa của các đỉnh là giá trị ưu tiên), chúng ta cài đặt KDLTT hàng ưu tiên bởi lớp PriQueue được mô tả trong hình 10.5. Lớp PriQueue là lớp phụ thuộc tham biến kiểu Item, trong đó Item là kiểu của các phần tử trong hàng ưu tiên. Các phần tử trong hàng ưu tiên chứa một thành phần là giá trị ưu tiên (priority), và để đơn giản cho cách viết ta giả thiết rằng, có thể truy cập trực tiếp tới giá trị ưu tiên của các phần tử. Chúng ta có thể cài đặt hàng ưu tiên bởi mảng động, như chúng ta đã làm khi cài đặt danh sách (xem mục 4.3). Song để tập trung sự chú ý tới các kỹ thuật được sử dụng trong các phép toán hàng ưu tiên, chúng ta cài đặt hàng ưu tiên bởi mảng tĩnh data. Cài đặt bởi mảng động để lại cho độc giả, xem như bài tập. Lớp PriQueue chứa hai hàm kiến tạo, hàm kiến tạo mặc định làm nhiệm vụ tạo ra một hàng ưu tiên rỗng, và một hàm kiến tạo khác xây dựng nên hàng ưu tiên từ các dữ liệu đã được lưu trong một mảng theo phương pháp trong mục 10.3.2. Các hàm thành phần còn lại là các hàm thực hiện các phép toán trên hàng ưu tiên. Một điều cần lưu ý là, chúng ta đưa vào lớp PriQueue một hàm ẩn ShiftDown(int i), hàm này thực hiện bước hai trong thuật toán DeleteMin. Hàm này được sử dụng để cài đặt hàm kiến tạo thứ hai và hàm DeleteMin.

```
template <class Item>
class       PriQueue
{
  public:
    static const int SIZE = 1000;
    PriQueue()  // Hàm kiến tạo mặc định
         {last = -1}
    PriQueue(Item* element, int n);
    //Xây dựng hàm ưu tiên từ n phần tử được lưu trong mảng element.
    bool   Empty() const
          { return last < 0;}
    Item &   FindMin() const;
    // Trả về phần tử có giá trị ưu tiên nhỏ nhất
    void     Insert(const Item &  object, bool &  suc);
    // Xen object vào hàng ưu tiên. Nếu thành công biến suc nhận giá
    // trị true, nếu không giá trị của nó là false.
    Item &   DeleteMin();
    // Loại và trả về đối tượng có giá trị ưu tiên nhỏ nhất.
  private:
     Item     data[SIZE];
     int      last;
     void     ShiftDown(int i);
     // Đẩy dữ liệu trong đỉnh i xuống vị trí thích hợp
};
```

Hình 10.5. Lớp PriQueue.

Sau đây chúng ta cài đặt các hàm thành phần của lớp PriQueue. Trước hết là hàm tìm đối tượng có giá trị ưu tiên nhỏ nhất.

```
template <class Item>
Item &   PriQueue <Item> ::  FindMin()
{
      assert( last  >=  0);
      return      data[0];
}
```

Hàm ẩn ShiftDown(int i) thực hiện bước hai trong thuật toán DeleteMin(). Chúng ta sử dụng biến parent ghi lại chỉ số của đỉnh cha và biến child ghi lại chỉ số của đỉnh con có giá trị ưu tiên nhỏ hơn giá trị ưu tiên của đỉnh con kia (nếu có). Biến x lưu lại phần tử chứa trong đỉnh i. Ban đầu giá trị của parent là i, mỗi khi giá trị ưu tiên của x lớn hơn giá trị ưu tiên của đỉnh child thì phần tử chứa trong đỉnh child được đẩy lên đỉnh parent và ta đi xuống đỉnh child. Đến khi giá trị ưu tiên của x nhỏ hơn hoặc bằng giá trị ưu tiên của đỉnh child hoặc khi đỉnh parent là đỉnh lá thì phần tử x được đặt vào đỉnh parent. Hàm ShiftDown được cài đặt như sau:

```
template <class Item>
void  PriQueue <Item> ::  ShiftDown(int i)
{
        Item   x = data[i];
        int      parent = i;
        int      child  =  2*parent+1; //đỉnh con trái.
while (child <= last)
  {
        int     right  = child+1; //đỉnh con phải
        if (right <= last  && data[right].priority < data[child].priority)
                child = right;
        if (x.priority > data[child].priority)
                {
                   data[parent] = data[child];
                   child = 2*parent+1;
                }
        else    break;
  }
        data[parent] = x;
}
```

Sử dụng hàm ShiftDown, chúng ta dễ dàng cài đặt được các hàm DeleteMin và hàm kiến tạo.

```
template <class Item>
Item &  PriQueue<Item> ::  DeteleMin()
{
```

```
    assert(last >= 0);
    data[0] = data[last--];  //chuyển phần tử trong đỉnh lá ngoài cùng
                             //bên phải ở mức thấp nhất lên gốc.
    ShiftDown(0);
}

template <class Item>
PriQueue<Item> ::  PriQueue(Item* element, int n)
{
     int i;
     for (i = 0 ; i < n ; i++)
              data[i]  =  element[i];
     last  =  n-1;
     for (i = n/2 - 1; i >= 0 ; i--)
              ShiftDown(i);
}
```

Bây giờ chúng ta sẽ cài đặt hàm Insert. Hàm Insert được cài đặt theo kỹ thuật tương tự như hàm ShiftDown. Biến child ban đầu nhận giá trị là last+1. Biến parent được tính theo biến child, parent=(child-1)/2. Mỗi khi giá trị ưu tiên của đỉnh cha lớn hơn giá trị ưu tiên của đối tượng cần xen vào thì phần tử trong đỉnh cha được đẩy xuống đỉnh con và ta đi lên đỉnh cha. Khi đạt tới đỉnh cha có giá trị ưu tiên nhỏ hơn hay bằng giá trị ưu tiên của đối tượng cần xen vào thì đối tượng được đặt vào đỉnh con. Hàm Insert được cài đặt như sau:

```
template <class Item>
void  PriQueue<Item>:: Insert(const Item & object, bool & suc)
{
  if (last = = SIZE-1)
             suc =  false;
  else {
          suc =  true;
          int   child = ++last;
          while (child > 0)
            {
              int   parent = (child-1)/2;
              if (data[parent].priority > object.priority)
```

```
                {
                    data[child] =  data[parent];
                    child =  parent;
                }
            else    break;
        }
    data[child] = object;
  }
}
```

10.5 NÉN DỮ LIỆU VÀ MÃ HUFFMAN

Nén dữ liệu (data compression) hay còn gọi là nén file là kỹ thuật rút ngắn cỡ của file được lưu trữ trên đĩa. Nén dữ liệu không chỉ nhằm tăng khả năng lưu trữ của đĩa mà còn để tăng hiệu quả của việc truyền dữ liệu qua các đường truyền bởi modem. Chúng ta luôn luôn giả thiết rằng, file dữ liệu chứa các dữ liệu được biểu diễn như là một xâu ký tự được tạo thành từ các ký tự trong bảng ký tự ASCII. Để lưu các dữ liệu dưới dạng file, chúng ta cần phải **mã hoá** các dữ liệu dưới dạng một xâu nhị phân (xâu bit), trong đó mỗi ký tự được biểu diễn bởi một dãy bit nào đó được gọi là từ mã (code word). Quá trình sử dụng các từ mã để biến đổi xâu bit trở về xâu ký tự nguồn được gọi là **giải mã**.

Trước hết ta nói tới cách mã hoá đơn giản: các từ mã có đội dài bằng nhau. Bởi vì bảng ASCII chứa 128 ký tự, nên chỉ cần các từ mã 8 bit là đủ để biểu diễn 128 ký tự. Cần chú ý rằng, nếu xâu ký tự được tạo thành từ p ký tự thì mỗi từ mã cần ít nhất $\lceil \log p \rceil$ bit. Ví dụ, nếu xâu ký tự nguồn chỉ chứa các ký tự a,b,c,d,e và f thì chỉ cần 3 bit cho mỗi từ mã, chẳng han ta có thể mã a = 000, b = 001, c = 010, d = 011, e = 100, và f = 101. Phương pháp mã hoá sử dụng các từ mã có độ dài bằng nhau có ưu điểm là việc giải mã rất đơn giản, nhưng không tiết kiệm được không gian trên đĩa, chẳng hạn trong ví dụ trên, nếu xâu nguồn có độ dài n thì

số bít cần thiết là 3.n bất kể số lần xuất hiện của các ký tự trong xâu nhiều hay ít.

Chúng ta có thể giảm số bít cần thiết để mã xâu ký tự nguồn bằng cách sử dụng các từ mã có độ dài thay đổi, tức là các ký tự được mã hoá bởi các từ mã có độ dài khác nhau. Chúng ta sẽ biểu diễn ký tự có tần xuất xuất hiện cao bởi từ mã ngắn, còn ký tự có tần xuất xuất hiện thấp với các từ mã dài hơn. Nhưng khi mà các từ mã có độ dài thay đổi, để giải mã chúng ta cần phải có cách xác định bit bắt đầu và bit kết thúc một từ mã trong xâu bit. Nếu một từ mã có thể là đoạn đầu (tiền tố) của một từ mã khác thì không thể giải mã được duy nhất. Chẳng hạn, khi xâu nguồn chứa các ký tự a, b, c, d, e và f và ta sử dụng các từ mã: a = 00, b = 111, c = 0011, d = 01, e = 0 và f = 1 (từ mã 00 là tiền tố của từ mã 0011, từ mã 0 là tiền tố của từ mã 01, 00, 0011, từ mã 1 là tiền tố của từ mã 111); khi đó, xâu bit 010001 có thể giải mã là các từ dad, deed, efad,… Do đó, để đảm bảo một xâu bit chỉ có thể là mã của một xâu nguồn duy nhất, chúng ta có thể sử dụng các từ mã sao cho không có từ mã nào là tiền tố của một từ mã khác.

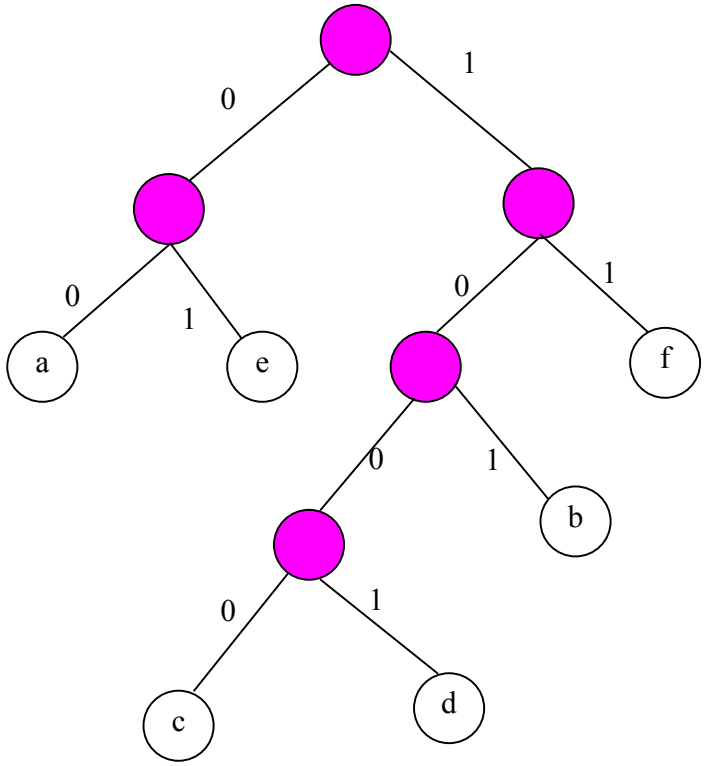
Một hệ từ mã mà không có từ mã nào là tiền tố của từ mã khác sẽ được gọi là mã tiền tố (prefix code). Người ta xây dựng mã tiền tố dựa trên tần xuất của các ký tự trong xâu nguồn. Phương pháp mã này cho phép rút bớt đáng kể số bit cần thiết. Ví dụ, chúng ta lại giả thiết xâu nguồn được tạo thành từ các ký tự a, b, c, d, e và f, xâu có độ dài 10000 ký tự, trong đó ký tự a xuất hiện 3000 lần, b xuất hiện 1000 lần, c xuất hiện 500 lần, d xuất hiện 500 lần, e xuất hiện 3000 lần và f xuất hiện 2000 lần. Chúng ta có thể sử dụng mã tiền tố như sau, a = 00, b = 101, c = 1000, d = 1001, e = 01 và f = 11, khi đó số bit cần thiết là:

3000.2+1000.3+500.4+500.4+3000.2+2000.2 = 23000 bit

Trong khi đó, nếu sử dụng các từ mã cùng độ dài là 3 bit thì số bit đòi hỏi là 10000.3 = 30000 bit.

Như vậy, vấn đề được đặt ra bây giờ là, làm thế nào từ một xâu ký tự nguồn, ta xây dựng được mã tiền tố sao cho số bit cần thiết trong xâu mã là ít nhất có thể được. Mã tiền tố thoả mãn tính chất đó được gọi là mã tiền tố tối ưu. Dưới đây chúng ta sẽ trình bày phương pháp mã được đề xuất bởi Huffman.

Chúng ta sẽ biểu diễn mã tiền tố dưới dạng cây nhị phân, tại mỗi đỉnh của cây, nhánh bên trái được gắn nhãn là 0, nhánh bên phải được gắn nhãn là 1, mỗi ký tự được lưu trong một đỉnh lá của cây. Khi đó, từ mã của mỗi ký tự là xâu bit tạo thành từ các nhãn trên đường đi từ gốc tới đỉnh lá chứa ký tự đó. Chẳng hạn, mã tiền tố đã đưa ra ở trên (a=00, b=101, c=1000, d=1001, e=01 và f=11) được biểu diễn bởi cây nhị phân trong hình 10.6.



Hình 10.6. Cây nhị phân biểu diễn mã tiền tố.

Chúng ta có nhận xét rằng, khi một mã có thể biểu diễn bởi cây nhị phân như đã mô tả ở trên thì mã sẽ là mã tiền tố, điều đó được suy ra từ tính chất các ký tự chỉ được chứa trong các đỉnh lá của cây. Mặt khác, chúng ta có thể giải mã rất đơn giản nếu chúng ta biểu diễn mã tiền tố bởi cây nhị phân. Thuật toán giải mã như sau: đọc xâu bit bắt đầu từ bit đầu tiên, và bắt đầu từ gốc cây ta đi theo nhánh trái nếu bit được đọc là 0 và đi theo nhánh phải nếu bit là 1. Tiếp tục đọc bit và đi như thế cho tới khi gặp một đỉnh lá, ký tự được lưu trong đỉnh lá đó được viết ra xâu kết quả. Lại đọc ký tự tiếp theo và lại bắt đầu từ gốc đi xuống, lặp lại quá trình đó cho tới khi toàn bộ xâu bit được đọc qua.

Thuật toán Huffman sử dụng hàng ưu tiên để xây dựng mã tiền tố dưới dạng cây nhị phân. Cây kết quả được gọi là **cây Huffman** và mã tiền tố ứng với cây đó được gọi là **mã Huffman.** Trước hết xâu ký tự nguồn cần được đọc qua để tính tần xuất của mỗi ký tự trong xâu. Ta ký hiệu tần xuất của ký tự c là f(c). Với mỗi ký tự xuất hiện trong xâu nguồn, ta tạo ra một đỉnh chứa ký tự đó, đỉnh này được xem như gốc của cây nhị phân chỉ có một đỉnh. Chúng ta ký hiệu tập đỉnh đó là Γ. Tập Γ được lấy làm đầu vào của thuật toán Huffman. Ý tưởng của thuật toán Huffman là từ tập các cây chỉ có một đỉnh, tại mỗi bước ta kết hợp hai cây thành một cây, và lặp lại cho đến khi nhận được một cây nhị phân kết quả. Chú ý rằng, chúng ta cần xây dựng được cây nhị phân sao cho đường đi từ gốc tới đỉnh lá chứa ký tự có tần xuất cao cần phải ngắn hơn đường đi từ gốc tới đỉnh lá chứa ký tự có tần xuất thấp hơn. Vì vậy, trong quá trình xây dựng, mỗi đỉnh gốc của cây nhị phân được gắn với một giá trị ưu tiên được tính bằng tổng các tần xuất của các ký tự chứa trong các đỉnh lá của nó, và tại mỗi bước ta cần chọn hai cây nhị phân có mức ưu tiên nhỏ nhất để kết hợp thành một. Do đó trong thuật toán, ta sẽ sử dụng hàng ưu tiên P để lưu các đỉnh gốc của các cây nhị phân, giá trị ưu tiên của đỉnh v sẽ được ký hiệu là f(v), đỉnh con trái của đỉnh v được ký

hiệu là leftchild(v), đỉnh con phải là rightchild(v). ban đâu hàng ưu tiên P được tạo thành từ các tập đỉnh Γ đã nói ở trên, ta giả sử Γ chứa n đỉnh.

Thuật toán Huffman gồm các bước sau:

1. Khởi tạo hàng ưu tiên P chứa các đỉnh trong tập Γ
2. Bước lặp

For (i = 1 ; i <= n – 1 ; i++)

v_1 = DeleteMin(P);

v_2 = DeleteMin(P);

Tạo ra đỉnh v mới với:

leftchild(v) = v_1;

leftchild(v) = v_2;

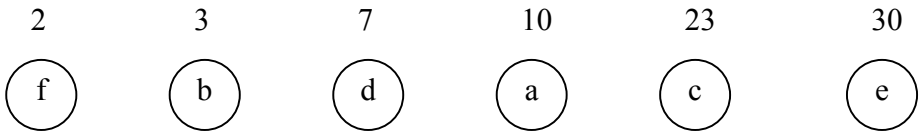
f(v) = $f(v_1) + f(v_2)$;

Insert(P,v);

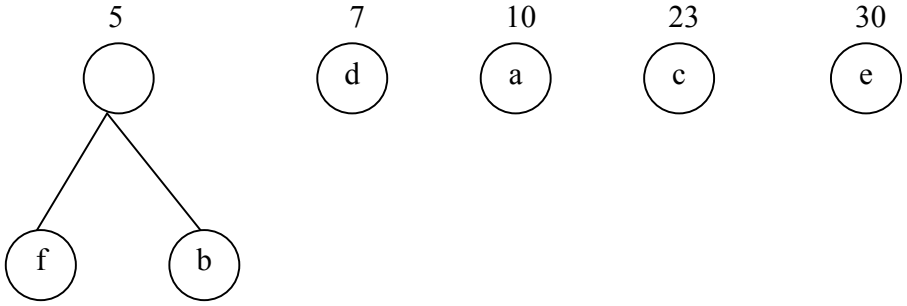
Có thể chứng minh được rằng, mã Huffman là mã tiền tố tối ưu. Một điều nữa cần lưu ý là mã Huffman không phải là duy nhất, bởi vì trong bước 2.3 chúng ta cũng có thể đặt rightchild(v) = v_1, leftchild(v) = v_2.

Để hiểu rõ hơn thuật toán Huffman, ta xét ví dụ sau. Giả sử xâu ký tự nguồn chứa các ký tự với tần xuất như sau: a xuất hiện 10 lần, b xuất hiện 3 lần, c xuất hiện 23 lần, d xuất hiện 7 lần, e xuất hiện 30 lần và f xuất hiện 2 lần. Như vậy ban đầu hàng ưu tiên P chứa các đỉnh được chỉ ra trong hình 10.7a, trong đó giá trị ưu tiên(tần xuất) được ghi trên mỗi đỉnh. Tại mỗi bước ta kết hợp hai cây có giá trị ưu tiên của gốc nhỏ nhất thành một cây mới. Vì có 6 ký tự, nên số bước là 5, kết quả của mỗi bước là một rừng cây được cho trong các hình từ 10.6 b đến 10.6 f

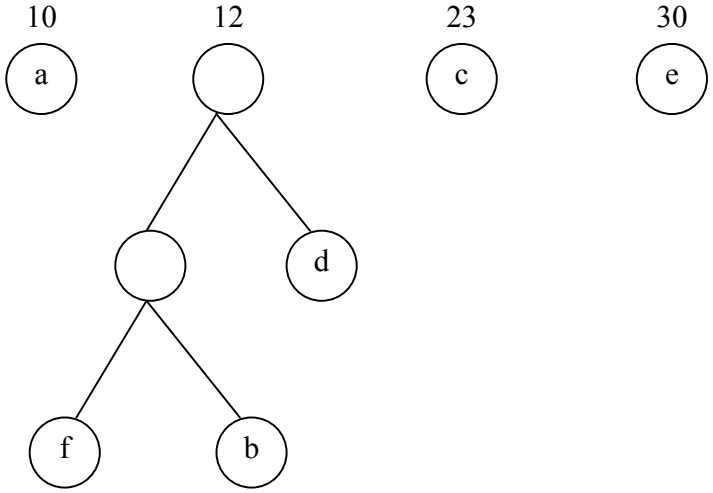
(a)


2
3
7
10
23
30
f
b
d
a
c
e

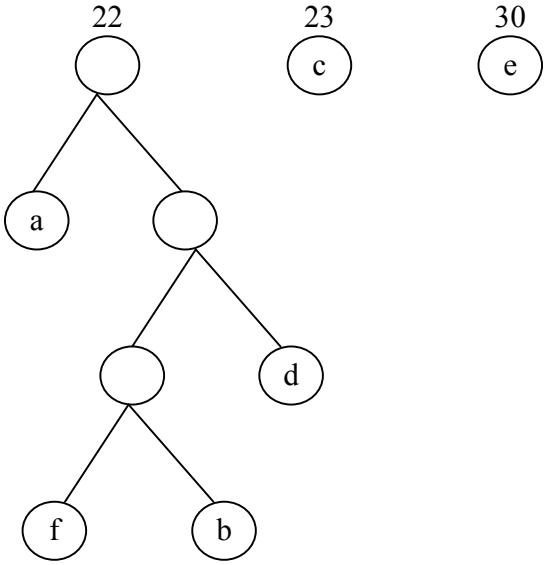
(b)


5
7
10
23
30
d
a
c
e
f
b

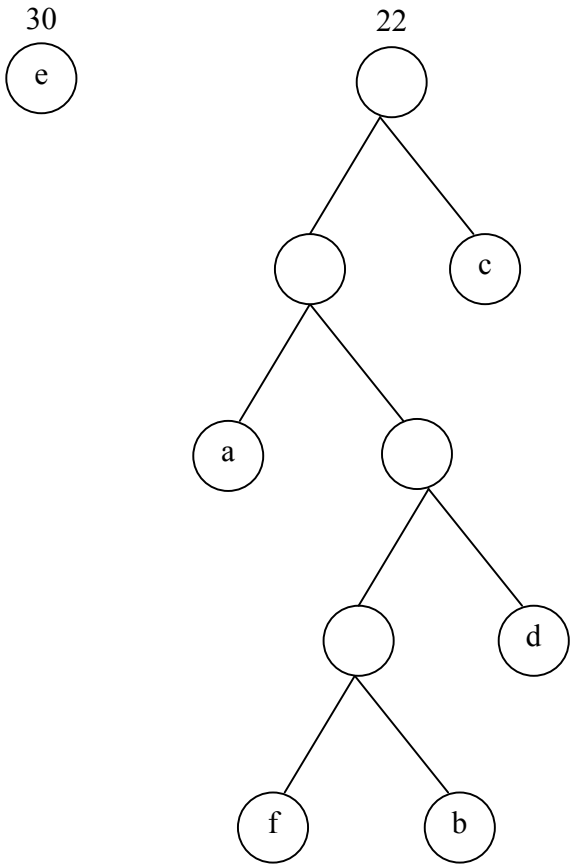
(c)


10
12
23
30
a
c
e
d
f
b

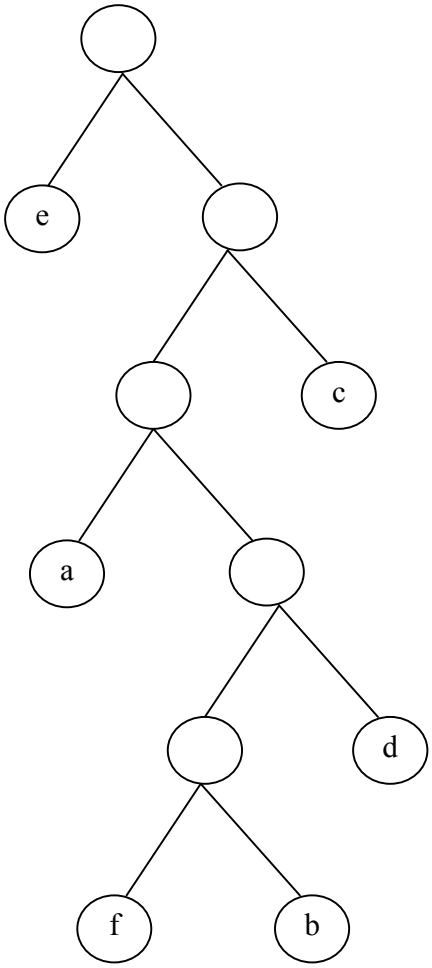
(d)


22
23
30
c
e
a
d
f
b

(e)



(f)



Hình 10.6. Quá trình xây dựng cây Huffman.

Từ cây Huffman trong hình 10.6, ta thành lập được các từ mã như sau: e = 0, c = 11, a = 100, d = 1011, f = 10100, và b = 10101.

BÀI TẬP

1. Ta có thể biểu diễn hàng ưu tiên bởi danh sách theo thứ tự bất kỳ hoặc bởi danh sách được sắp theo giá trị ưu tiên tăng dần (hoặc giảm dần). Hãy cài đặt lớp hàng ưu tiên bằng cách thừa kế private từ lớp cơ sở Dlist (hoặc Llist) trong các trường hợp sau:

a. Hàng ưu tiên được biểu diễn bởi danh sách theo thứ tự bất kỳ.
b. Hàng ưu tiên được biểu diễn bởi danh sách được sắp theo giá trị ưu tiên giảm dần (tăng dần).

2. Giả sử rằng, các đối tượng khác nhau của hàng ưu tiên có thể có cùng một giá trị ưu tiên. Lấy giá trị ưu tiên làm khoá, hãy đưa ra cách biểu diễn hàng ưu tiên dưới dạng cây tìm kiếm nhị phân. Hãy thiết kế và cài đặt lớp hàng ưu tiên khi hàng ưu tiên được biểu diễn bởi cây tìm kiếm nhị phân theo cách đã đưa ra.

3. Cho một dãy đối tượng với các giá trị ưu tiên là 10, 12, 1, 14, 6, 5, 8, 15 và 9.
   a. Từ cây thứ tự bộ phận rỗng, hãy xen vào từng đối tượng theo thứ tự đã liệt kê. Vẽ ra cây kết quả.
   b. Hãy sử dụng thuật toán xây dựng cây thứ tự bộ phận cho dãy đối tượng trên. So sánh cây này với cây nhận được bằng cách xen vào từng đối tượng ở câu a

PHẦN II

CÁC CẤU TRÚC DỮ LIỆU CAO CẤP

CHƯƠNG 11

CÁC CÂY TÌM KIẾM CÂN BẰNG

Trong mục 8.4 chúng ta đã nghiên cứu CTDL cây tìm kiếm nhị phân và sử dụng CTDL này để cài đặt KDLTT tập động. Chúng ta đã chỉ ra rằng, các phép toán tập động trên cây tìm kiếm nhị phân, trong trường hợp xấu nhất, sẽ đòi hỏi thời gian O(n), trong đó n là số đỉnh của cây. Đó là trường hợp cây suy biến thành danh sách liên kết, tức là tất cả các nhánh trái (phải) của mọi đỉnh đều rỗng. Trường hợp này sẽ xảy ra khi chúng ta xen vào cây một dãy dữ liệu đã được sắp xếp theo thứ tự tăng (giảm), một hoàn cảnh thường gặp trong thực tiễn. Trong chương này chúng ta sẽ nghiên cứu một số loại cây tìm kiếm cân bằng, khắc phục được sự chênh lệch nhiều về số đỉnh giữa nhánh trái và nhánh phải tại mọi đỉnh, và do đó thời gian thực hiện các phép toán tập động, trong trường hợp xấu nhất, cũng chỉ là O(logn). Các cây tìm kiếm cân bằng mà chúng ta sẽ đưa ra là cây AVL và cấy đỏ - đen.

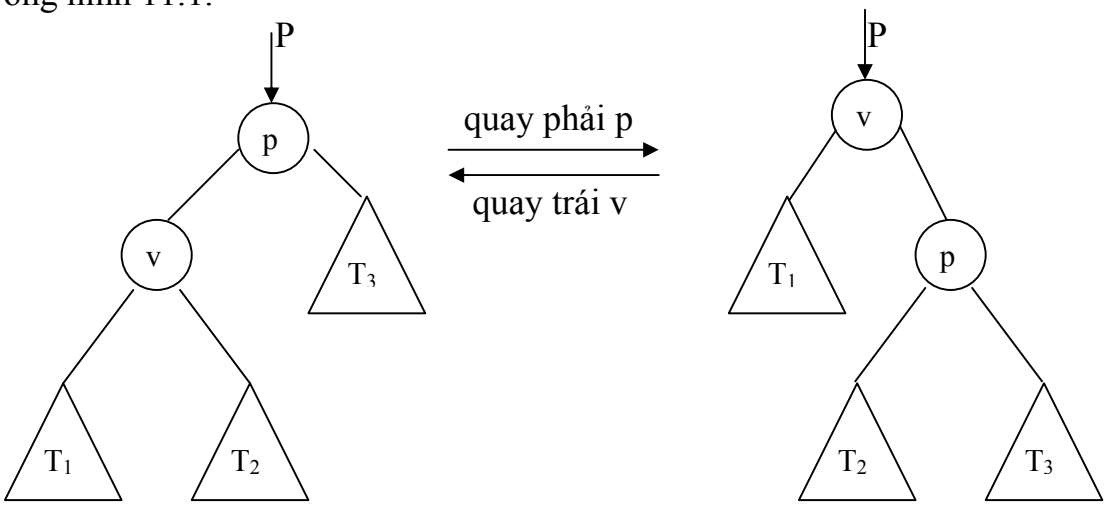
Chúng ta sẽ đưa vào chương này phương pháp phân tích mới, từ trước đến nay chúng ta chưa bao giờ sử dụng tới, đó là phân tích trả góp. Phương pháp này cho phép ta đánh giá cận trên chặt của thời gian thực hiện một dãy phép toán trên CTDL tự điều chỉnh. Cuối chương này chúng ta sẽ nghiên cứu CTDL cây tán loe: một dạng CTDL tự điều chỉnh, và sử dụng phương pháp phân tích trả góp để đánh giá thời gian thực hiện một dãy phép toán tập động trên cây tán loe.

Trước hết chúng ta đưa vào các phép toán quay trên cây nhị phân. Các phép quay này sẽ được sử dụng đến trong các mục tiếp theo.

11.1 CÁC PHÉP QUAY

Các cây AVL, cây đỏ - đen, cây tán loe mà chúng ta sẽ lần lượt xét trong chương này đều là cây tìm kiếm nhị phân, tức là chúng đều phải thoả mãn tính chất tìm kiếm nhị phân (hay tính chất được sắp): dữ liệu chứa trong một đỉnh bất kỳ có khoá lớn hơn khoá của mọi dữ liệu chứa trong cây con trái và nhỏ hơn khoá của mọi dữ liệu chứa trong cây con phải. Chúng chỉ khác nhau bởi các điều kiện áp đặt nhằm giảm độ cao của cây hoặc không làm mất cân bằng giữa nhánh trái và nhánh phải tại mọi đỉnh. Mỗi khi thực hiện một phép toán (xen hoặc loại) làm cho cây không còn thoả mãn các điều kiện áp đặt, chúng ta sẽ cấu tạo lại cây bằng cách sử dụng các phép quay.

Có hai phép quay cơ bản là quay trái và quay phải được chỉ ra trong hình 11.1. Giả sử p là một đỉnh trong cây nhị phân, và đỉnh con trái của nó là v. Phép quay phải đỉnh p sẽ đặt v vào vị trí của đỉnh p, p trở thành con phải của v và cây con phải của đỉnh v trở thành cây con trái của đỉnh p. Trong hình 11.1, phép quay phải đỉnh p sẽ biến cây ở vế trái thành cây ở vế phải. Đối xứng qua gương của phép quay phải là phép quay trái, như được chỉ ra trong hình 11.1.



Hình 11.1. Các phép quay cơ bản.

Bây giờ chúng ta chứng minh một khẳng định quan trọng: các phép quay cơ bản (trái hoặc phải) không phá vỡ tính chất tìm kiếm nhị phân. Chúng ta chứng minh cho phép quay phải tại đỉnh p. Phép quay này chỉ tác động đến đỉnh p và v, và do đó ta chỉ cần chỉ ra rằng, sau khi quay đỉnh p và v vẫn còn thoả mãn tính chất tìm kiếm nhị phân. Ta có nhận xét rằng, phép quay không ảnh hưởng gì đến cây con trái của v và cây con phải của đỉnh p.Trước khi quay, cây con phải của v là T_2 thuộc cây con trái của đỉnh p,do đó, khoá của mọi đỉnh trong cây con T_2 lớn hơn khoá của đỉnh v và nhỏ hơn khoá của đỉnh p; mặt khác, khoá của mọi đỉnh trong cây con T_3 lớn hơn khoá của đỉnh p, và do đó lớn hơn khoá của đỉnh v, vì khóa của đỉnh p lớn hơn khoá của đỉnh v. Từ các kết luận đó, ta thấy ngay rằng, các đỉnh p và v sau phép quay phải vẫn còn thoả mãn tính chất tìm kiếm nhị phân.

11.2 CÂY AVL

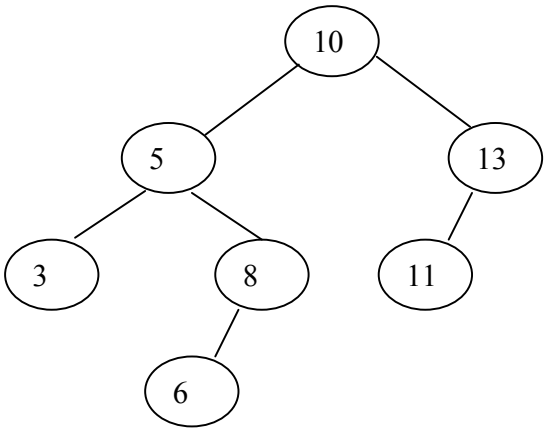
Trong mục này chúng ta nghiên cứu CTDL cây AVL, do các nhà toán học Nga Adelson- Velskii và Landis đề xuất. Nhớ lại rằng, chúng ta xác định

độ cao của cây nhị phân là số đỉnh trên đường đi dài nhất từ gốc tới lá, và độ cao của cây rỗng bằng 0. Cây AVL được định nghĩa như sau:

Định nghĩa 11.1. Cây AVL là cây tìm kiếm nhị phân, trong đó độ cao của cây con trái và độ cao của cây con phải của mỗi đỉnh khác nhau không quá 1.

Như vậy, mỗi đỉnh của cây AVL sẽ ở một trong ba trạng thái: cây con trái và phải có độ cao bằng nhau (ta ký hiệu trạng thái này là EH), cây con trái cao hơn cây con phải 1 (trạng thái LH) và cây con phải cao hơn cây con trái 1 (trạng thái RH). Sau này chúng ta sẽ nói đỉnh ở một trong ba trạng thái trên là đỉnh cân bằng. Còn nếu một đỉnh có độ cao cây con trái lớn hơn độ cao cây con phải 2 thì nó được xem là đỉnh lệch bên trái. Tương tự, ta có khái niệm đỉnh lệch bên phải.

Hình 11.1 cho ta một ví dụ về cây AVL



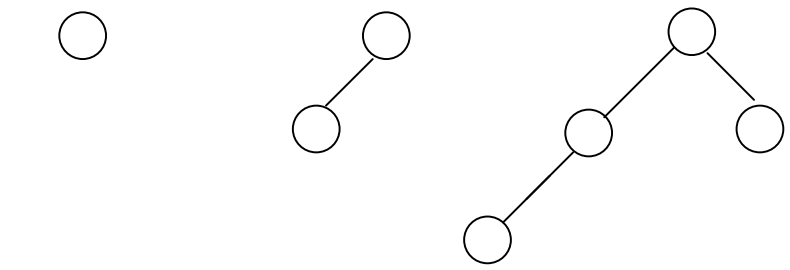
Hình 11.1. Một cây AVL

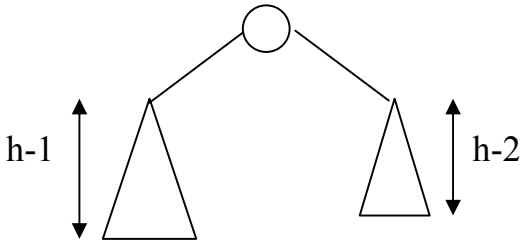
Bây giờ chúng ta chứng minh một tính chất quan trọng về độ cao của cây AVL.

Định lý 11.1. Độ cao của cây AVL có n đỉnh là O(logn).

Thay cho đánh giá độ cao lớn nhất của cây AVL chứa n đỉnh, ta đánh giá số đỉnh ít nhất của cây AVL với độ cao h. Ta ký hiệu N(h) là số đỉnh ít nhất của cây AVL có độ cao h. Các cây AVL có số đỉnh ít nhất với độ cao h = 1, 2, 3 được chỉ ra trong hình 11.2a. Như vậy N(0) = 0, N(1) = 1, N(2) = 2, N(3) = 3. Cây AVL có số đỉnh ít nhất với độ cao h là cây trong hình 11.2b, nó có cây con trái là cây AVL có số đỉnh ít nhất với độ cao h-1, và cây con phải là cây AVL có số đỉnh ít nhất với độ cao h-2. Do đó

$$N(h) = N(h-1) + N(h-2) + 1 \qquad (1)$$





Hình 11.2. Các cây AVL có số đỉnh ít nhất.

Từ đẳng thức (1) và tính chất của dãy só Fibonacci, bằng quy nạp ta chứng minh được

$N(h) = F(h + 1) - 1$ (2)

trong đó F(m) là số thứ m trong dãy số Fibonacci.

Nhưng $F(m) = \frac{1}{\sqrt{5}}(\phi^m - \hat{\phi}^m)$, trong đó

$\phi = \frac{1}{2}(1+\sqrt{5})$ và $\hat{\phi} = \frac{1}{2}(1-\sqrt{5})$, do đó

$N(h) = \frac{1}{\sqrt{5}}(\phi^{h+1} - \hat{\phi}^{h+1}) - 1$

Chú ý rằng $|\hat{\phi}| = \left|\frac{1}{2}(1-\sqrt{5})\right| < 1$. Do đó hạng thức $\hat{\phi}^{h+1}$ sẽ gần với 0 khi h lớn. Do đó, ta có đánh giá

$N(h) \approx \frac{1}{\sqrt{5}}\phi^{h+1}$

Từ đó ta có

$$h = O(\log N(h))$$

Vì N(h) là số đỉnh ít nhất của cây AVL có độ cao h, do đó độ cao h của cây AVL có n đỉnh là h = O(logn).

Chúng ta sẽ sử dụng tính chất về độ cao của cây AVL để đánh giá thời gian thực hiện các phép toán tập động trên cây AVL.

12.2.1 Các phép toán tập động trên cây AVL

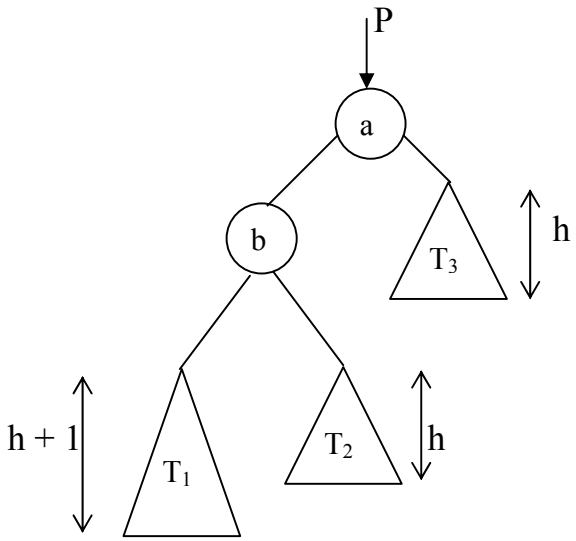
Bởi vì cây AVL là cây tìm kiếm nhị phân, nên các phép toán hỏi: tìm kiếm dữ liệu có khoá k cho trước, tìm dữ liệu có khoá nhỏ nhất (lớn nhất) được thực hiện theo các thuật toán như trên cây tìm kiếm nhị phân. Chúng ta đã chỉ ra rằng, độ cao của cây AVL chứa n đỉnh là O(logn), do đó thời gian thực hiện các phép toán hỏi trên cây AVL là O(logn). Nhưng nếu chúng ta thực hiện phép toán xen (hoặc phép loại) trên cây AVL thì chúng ta không thể chỉ tiến hành như trên cây tìm kiếm nhị phân, bởi vì sau khi xen (loại) tính cân bằng tại một số đỉnh có thể bị phá vỡ, tức là độ cao của cây con trái (phải) lớn hơn độ cao cây con phải (trái) 2. Trong các trường hợp đó, chúng ta cần phải cấu trúc lại cây bằng cách sử dụng các phép quay để thiết lập lại sự cân bằng của các đỉnh.

Phép toán xen. Việc xen vào cây AVL một đỉnh mới trước hết được thực hiện theo thuật toán xen vào cây tìm kiếm nhị phân. Khi đó các đỉnh nằm trên đường đi từ đỉnh mới xen vào lên gốc có thể không còn cân bằng nữa. Vì vậy chúng ta cần phải đi từ đỉnh mới lên gốc, gặp đỉnh nào mất cân bằng thì sử dụng các phép quay để làm cho nó trở thành cân bằng. Một đỉnh mất cân bằng chỉ có thể là lệch bên trái hoặc lệch bên phải. Chúng ta xét trường hợp đỉnh lệch bên trái.

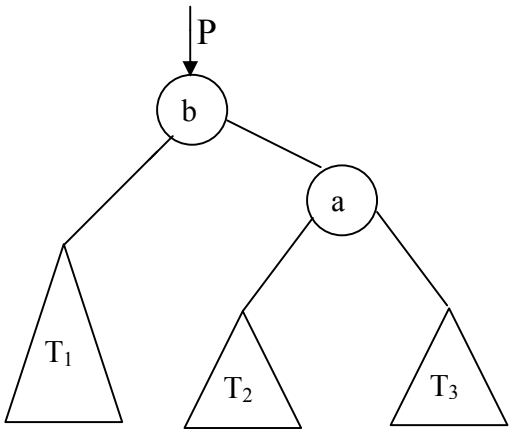
Gải sử a là đỉnh đầu tiên trên đường đi từ đỉnh mới lên gốc bị lệch bên trái. Giả sử P là con trỏ liên kết trong cây trỏ tới đỉnh a. Giả sử đỉnh con trái của a là đỉnh b, cây con trái của b là T_1, cây con phải của b là T_2, cây con phải của đỉnh a là T_3. Trường hợp đỉnh a bị lệch trái chỉ xảy ra khi mà trước khi xen vào đỉnh mới, độ cao cây con trái của đỉnh a lớn hơn độ cao cây con phải 1 và đỉnh mới được xen vào cây con trái của a, làm tăng độ cao cây con trái của a lên 1. Do đó nếu cây con T_3 có độ cao h, thì một trong hai cây con T_1, T_2 phải có độ cao h + 1, còn cây kia có độ cao là h. Chúng ta xét từng trường hợp.

- **Mẫu quay phải**. Trường hợp cây con T_1 có độ cao h + 1 và cây con T_2 có độ cao h ($h \geq 0$), như trong hình 11.3a, chúng ta chỉ cần thực hiện phép quay phải đỉnh a. Kết quả ta nhận được cây trong hình 11.3b. Dễ dàng thấy rằng sau phép quay phải, cả hai đỉnh a và b đều ở

trạng thái cân bằng EH và độ cao của cây P giảm đi 1 so với trước khi quay.



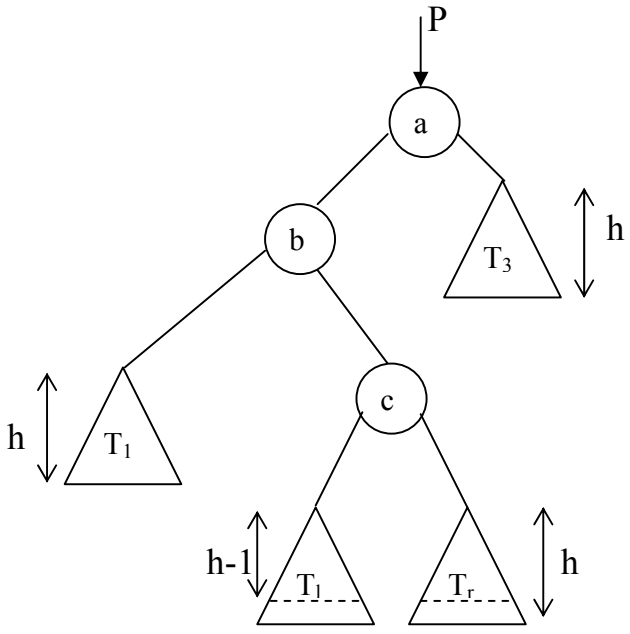
(a)



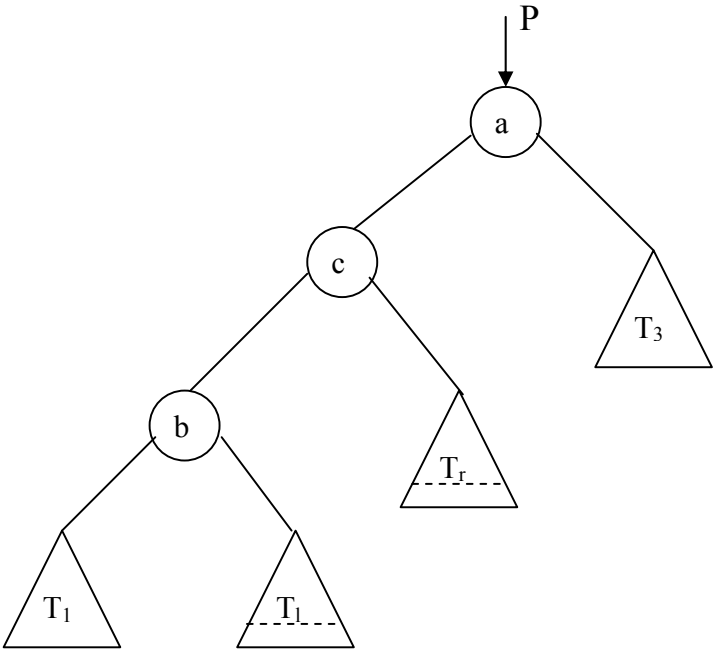
(b)

Hình 11.3. Mẫu quay phải

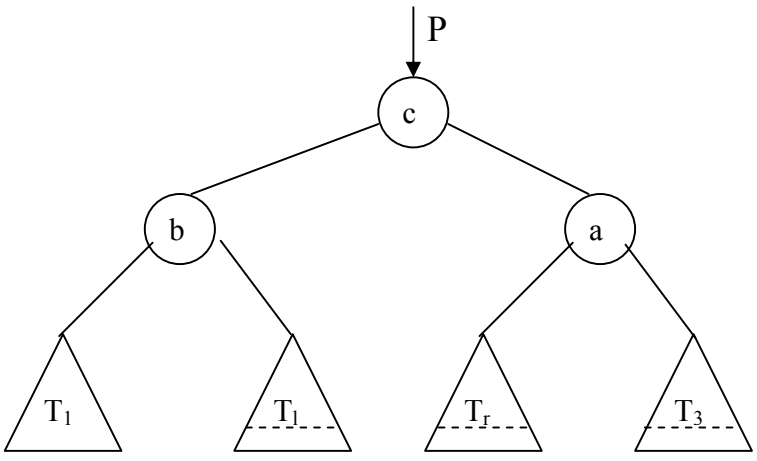
- **Mẫu quay kép trái - phải**. Trường hợp cây con T_1 có độ cao h, và cây con T_2 có độ cao h + 1, nếu chúng ta thực hiện phép quay phải đỉnh a thì đỉnh P lại trở thành lệch phải (hãy thử xem). Trong trường hợp này chúng ta phải tiến hành phép quay kép. Giả sử cây con T_2 có gốc là đỉnh c, cây con trái của c là T_l, cây con phải là T_r. Để cho T_2 có độ cao h + 1, thì ít nhất một trong hai cây con T_l và T_r phải có độ cao h, còn cây kia có thể có độ cao h hoặc h – 1. Chúng ta có cây như trong hình 11.4a. Đầu tiên ta quay trái đỉnh b để nhận được cây trong hình 11.4b Sau đó ta quay phải đỉnh a, cây kết quả là cây trong hình 11.4c. Dễ dàng thấy sau hai phép quay liên tiếp trái, phải này thì cả ba đỉnh a, b, c đều trở thành cân bằng, đỉnh c ở trạng thái EH, đỉnh b ở trạng thái EH hoặc EH, còn đỉnh a ở trạng thái EH hoặc RH. Chúng ta thấy rằng, sau phép quay kép trái - phải này thì độ cao của cây P cũng giảm đi 1 so với trước khi thực hiện phép quay.



(a)



(b)



(c)

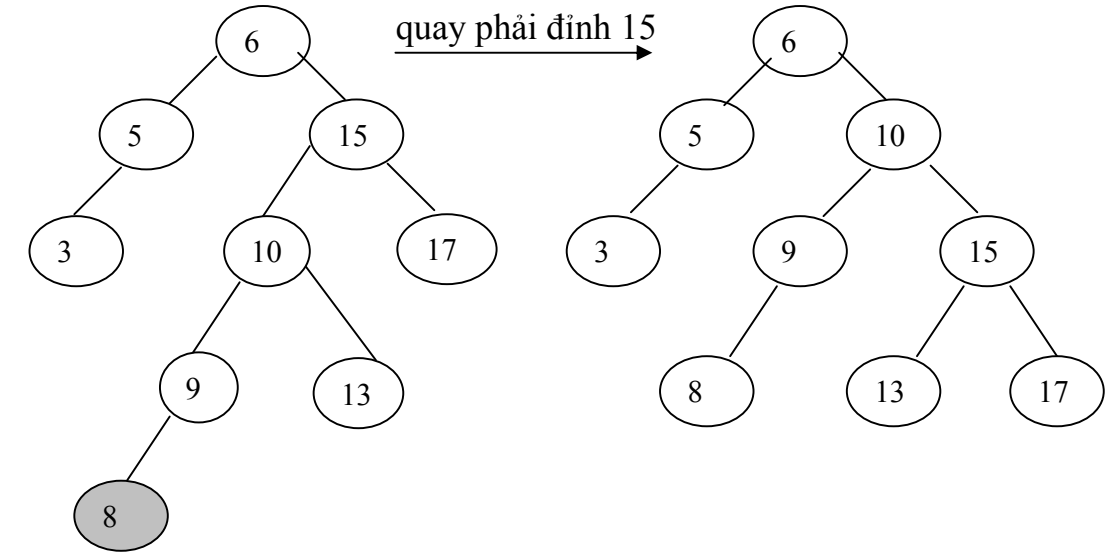
Hình 11.4. Mẫu quay kép trái - phải

Chúng ta có một nhận xét quan trọng: cả hai mẫu quay, quay phải hoặc quay kép trái - phải, đều làm giảm độ cao của cây P đi 1, tức là trở lại độ cao của cây P trước khi ta thực hiện phép xen vào.

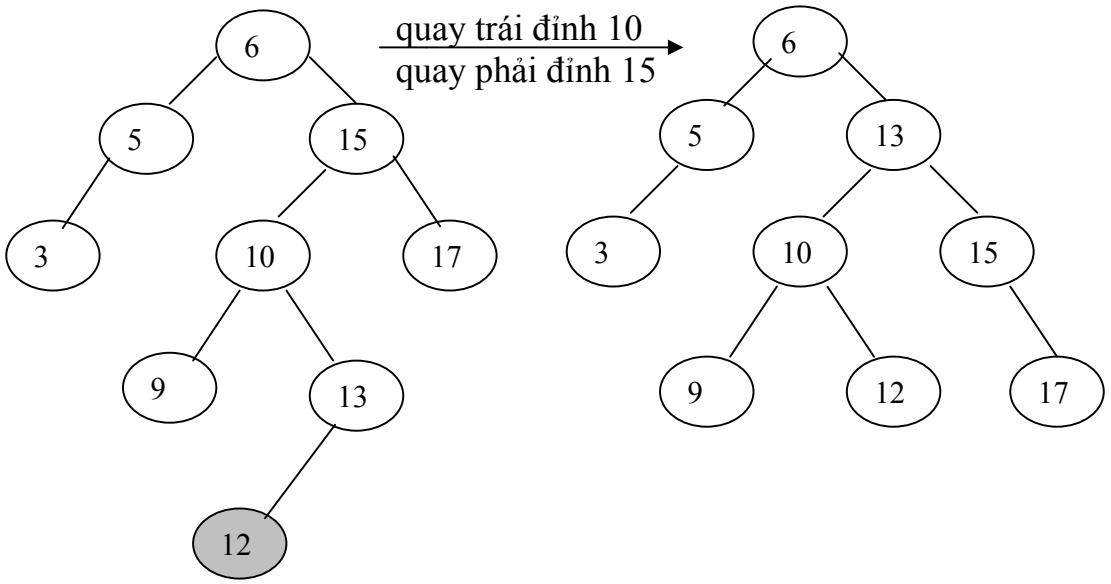
Trường hợp đỉnh a bị lệch bên phải, ta cũng có hai trường hợp riêng. Đó là các cây là đối xứng qua gương của cây trong hình 11.3a và 11.4a. Nếu đỉnh a lệch phải và có dạng đối xứng qua gương cúa cây trong hình 11.3a, ta chỉ cần thực hiện phép quay trái đỉnh a. Nếu đỉnh a lệch phải và có dạng đối xứng qua gương của cây trong hình 11.4a, ta thực hiện phép quay phải – trái (đầu tiên quay phải đỉnh b, sau đó quay trái đỉnh a).

Từ nhận xét ở trên, ta suy ra rằng, khi xen vào một đỉnh mới, ta đi từ đỉnh mới lên gốc cây, gặp đỉnh đầu tiên mất cân bằng ta chỉ cần thực hiện một phép quay đơn (phải hoặc trái) hoặc một phép quay kép (trái - phải hoặc phải – trái) là cây trở thành cân bằng.

Ví dụ. Khi xen vào cây AVL đỉnh mới 8, ta có cây ở vế trái trong hình 11.5a. Đi từ đỉnh 8 lên, ta thấy đỉnh đầu tiên mất cân bằng là đỉnh 15. Quay phải đỉnh 15 ta nhận được cây AVL ở vế phải hình 11.5a. Bây giờ thay cho đỉnh 8 ta xen vào đỉnh mới 12, ta có cây ở vế trái hình 11.5b. Trường hợp này, đỉnh lệch bên trái cũng là đỉnh 15. Nhưng trong hoàn cảnh này, chúng ta phải sử dụng phép quay kép trái - phải. Đầu tiên quay trái đỉnh 10, sau đó quay phải đỉnh 15 ta nhận được cây AVL ở vế phải hình 11.5b.



(a)



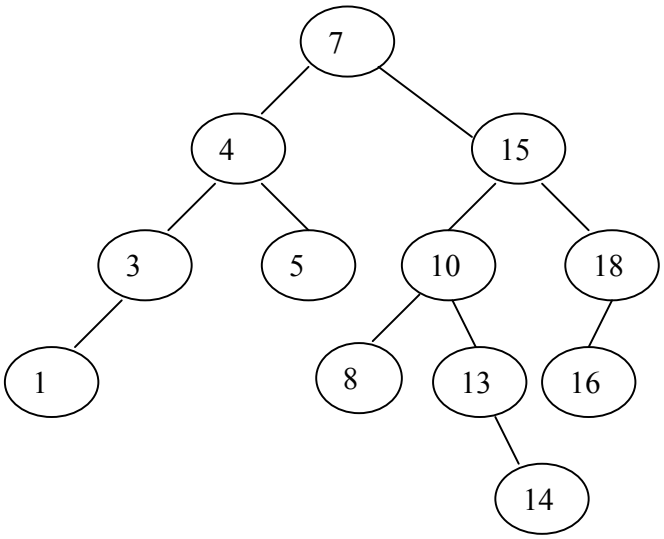
(b)

Hình 11.5. Xen vào cây AVL một đỉnh mới

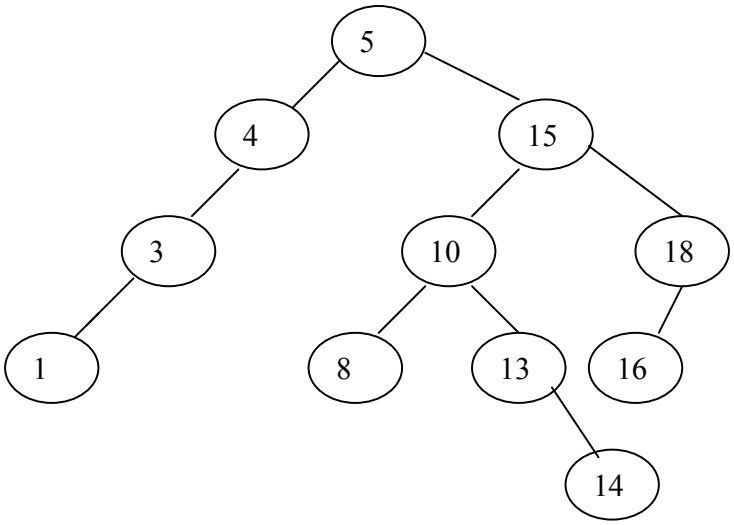
Phép toán loại. Giả sử chúng ta cần loại khỏi cây AVL một đỉnh chứa dữ liệu với khoá k. Đầu tiên ta sử dụng thuật toán loại trên cây tìm kiếm nhị phân để loại đỉnh chứa khoá k. Nhớ lại rằng, nếu đỉnh p chứa khoá k có đầy đủ cả hai con thì ta tìm đến đỉnh ngoài cùng bên phải v của cây con trái của đỉnh p. Lưu dữ liệu trong đỉnh v vào đỉnh p và cắt bỏ đỉnh v (đỉnh v là lá hoặc chỉ có một đỉnh con trái). Trong mọi trường hợp, đỉnh thực sự bị cắt bỏ

là đỉnh v, nó là lá hoặc chỉ có một đỉnh con trái hoặc phải. Giả sử cha của đỉnh v là đỉnh u. Sau khi cắt bỏ đỉnh v thì độ cao cây con trái (phải) của u sẽ giảm đi 1, nếu v là đỉnh con trái (phải) của u. Do đó đỉnh u có thể mất cân bằng. Bởi vậy, cũng giống như khi thực hiện phép xen, chúng ta đi từ đỉnh bị cắt bỏ lên gốc cây, gặp đỉnh nào mất cân bằng chúng ta cần áp dụng phép quay đơn (quay trái hoặc phải) hoặc phép quay kép (quay trái - phải hoặc quay phải – trái) để lập lại sự cân bằng cho đỉnh đó. Nhưng lưu ý rằng, phép quay đơn hoặc quay kép tại u làm giảm độ cao của cây con gốc u đi 1, và do đó khi ta thực hiện phép quay đơn (kép) làm cho đỉnh u trở lại cân bằng thì đỉnh cha nó có thể lại mất cân bằng. Do đó, không giống như khi thực hiện phép xen, chỉ cần một lần quay đơn (hoặc quay kép) tại một đỉnh là cây trở lại cân bằng, khi thực hiện phép loại sự mất cân bằng được truyền từ đỉnh bị cắt bỏ lên gốc. Trong trường hợp xấu nhất, tất cả các đỉnh trên đường đi từ đỉnh bị cắt bỏ lên gốc đều lần lượt bị mất cân bằng.

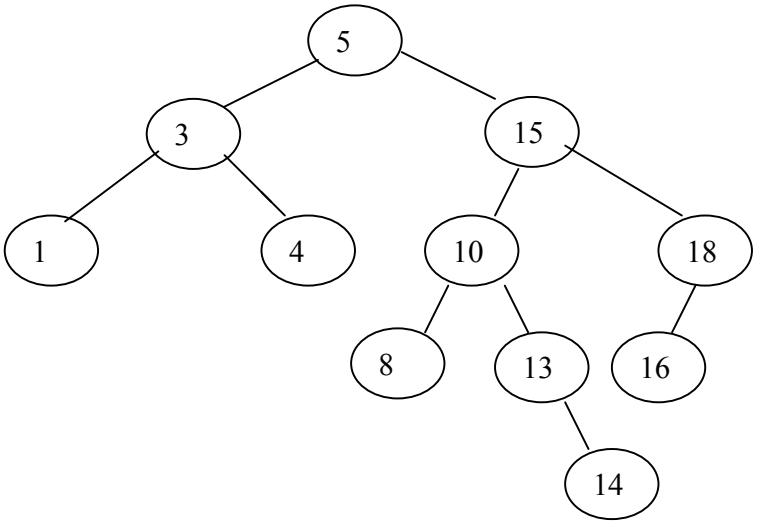
Ví dụ. Giả sử chúng ta cần loại khỏi cây AVL trong hình 11.5a đỉnh chứa khoá 7. Tìm đến đỉnh ngoài cùng bên phải của cây con trái đỉnh 7 là đỉnh 5. Chép dữ liệu từ đỉnh 5 lên đỉnh 7 và cắt bỏ đỉnh 5 ta nhận được cây tìm kiếm nhị phân như trong hình 11.5b. Trong cây hình 11.5b, đỉnh 4 mất cân bằng, thực hiện phép quay phải tại đỉnh 4, ta nhận được cây trong hình 11.5c. Đến đây đỉnh 5 lại mất cân bằng, thực hiện phép quay kép phải – trái tại đỉnh 5 chúng ta nhận được cây AVL trong hình 11.5d.



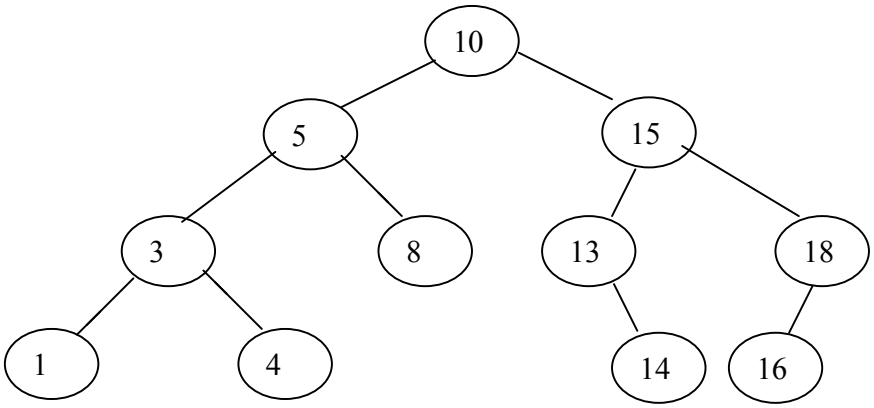
(a)



(b)



(c)



(d)

Hình 11.5. Loại khỏi cây AVL

Thời gian thực hiện phép toán xen, loại trên cây AVL. Các phép toán xen và loại trên cây AVL đều cần phải đi từ đỉnh mới xen vào hoặc đỉnh bị cắt bỏ ngược lên gốc, và tiến hành khôi phục lại sự cân bằng của các đỉnh bởi các phép quay. Nhưng thời gian thực hiện phép quay đơn hoặc quay kép là O(1). Do đó, thời gian thực hiện phép xen, loại là tỷ lệ với độ cao của cây AVL. Nhưng độ cao của cây AVL với n đỉnh, theo định lý 11.1, là O(logn), do đó các phép xen, loại trên cây AVL chỉ đòi hỏi thời gian O(logn).

11.2.2 Cài đặt tập động bởi cây AVL

CTDL biểu diễn đỉnh của cây AVL là cấu trúc biểu diễn đỉnh của cây tìm kiếm nhị phân được bổ sung thêm một trường lưu giữ trạng thái cân bằng của đỉnh.

```
enum  stateType  { LH, EH, RH};
struct  Node
{
  Item   data ;
  Node*  left, right ;
  StateType   balance ;
}
```

Trong đó, Item là một cấu trúc chứa một trường key là khoá của dữ liệu.

Trước hết chúng ta cài đặt các hàm thực hiện các mẫu quay đơn và các mẫu quay kép.

Hàm quay phải (RightRotation):

```
Void    RightRotation (Node* & P)
// Quay phải đỉnh P, P là con trỏ liên kết trong cây.
{
  Node* Q = P → left ;
  P → left = Q → right ;
  Q → right = P ;
  P = Q ;
}
```

Tương tự, bạn đọc hãy viết ra hàm quay trái (LeftRotation). Các hàm quay kép trái - phải (LR_ Rotation) và quay kép phải – trái (RL_ Rotation) chứa các lời gọi hàm quay đơn. Cụ thể là như sau:

```
void    LR_Rotation (Node* & P)
{
  LeftRotation (P → left):
  RightRotation (P) ;
}

void    RL_Rotation (Node* & P)
{
  RightRotation ( P → right) ;
  LeftRotation (P) ;
}
```

Sau đây chúng ta cài đặt hàm xen vào cây AVL một đỉnh mới chứa dữ liệu là d.

Hàm xen. Hàm Insert được cài đặt là hàm đệ quy

```
void    Insert (Node* & P, const Item & d, bool & taller)
// Xen vào cây P một đỉnh mới chứa dữ liệu d.
// Biến taller nhận giá trị true nếu sau khi xen độ cao của cây P tăng
// lên 1, nó nhận giá trị false nếu sau khi xen độ cao cây P không thay
// đổi
```

Giả sử khi chúng ta xen đỉnh mới vào cây con trái của đỉnh P, đỉnh P ở trạng thái LH và sau khi xen độ cao của cây con trái tăng lên 1. Rõ ràng trong trường hợp này đỉnh P bị lệch trái, chúng ta cần phải sử dụng phép quay phải hoặc phép quay kép trái - phải để lập lại sự cân bằng cho đỉnh P. Hành động này được cài đặt bởi hàm cân bằng trái (LeftBalance).

```
void    LeftBalance (Node* & P)
{
  switch (P → left → balance)
  {
    case   LH : // Trường hợp hình 11.3a
          RightRotation(P) ;
          P → balance = EH ;
          P → right → balance = EH ;
          break ;
    case  RH : // Trường hợp hình 11.4t
          LR_Rotation(P) ;
          switch (P → balance)
          {
            case  EH :
              P →left → balance = EH ;
              P → right → balance = RH ;
              break ;
            case  LH :
              P →left → balance = EH ;
              P → right → balance = RH ;
              break ;
            case  RH :
              P → left → balance = LH ;
              P →right → balance = EH ;
          }
          P → balance = EH ;
  }
}
```

Tương tự, nếu đỉnh P ở trạng thái RH, và đỉnh mới được xen vào cây con phải của đỉnh P và làm tăng độ cao cây con phải lên 1, thì đỉnh P trở thành lệch phải và chúng ta cần sử dụng hàm cân bằng phải (RightBalance). Hàm này được cài đặt tương tự hàm LeftBalance (bài tập)

Đến đây chúng ta có thể viết được hàm Insert.

```
void    Insert (Node* &  P, const Item &  d, bool &  taller)
{
  if (P = = NULL)
    {
        P = new Node ;
        P → data = d ;
        P → left = P → right = NULL ;
        P → balance = EH;
        taller = true ;
    }
  else if (d.key < P → data.key)
    {
        Insert (P → left, d, taller)
        if (taller)
          switch (P → balance)
          {
           case  LH :
               LeftBalance (P) ;
               taller = false ;
               break ;
           case  EH :
               P → balance = LH ;
               taller = true ;
               break ;
           case  RH :
               P → balance = EH ;
               taller = false ;
          }
    }
  else if (d.key > P → data.key)
    {
        Insert (P → right, d, taller)
        // Các dòng lệnh còn lại tương tự như các dòng lệnh đi sau
        // Insert (P → left, d, taller)
    }
}
```

Hàm loại. Hàm loại cũng được cài đặt là hàm đệ quy.

```
void    Delete (Node* &  P, keyType  k, bool  shorter)
// Loại khỏi cây P đỉnh chứa dữ liệu có khoá là k.
// Biến shorter nhận giá trị true nếu sau khi loại độ cao cây P ngắn đi 1
// và nhận giá trị false nếu sau khi loại độ cao của cây P không thay
// đổi.
```

Giả sử đỉnh bị loại ở cây con phải của đỉnh P và sau khi loại, độ cao cây con phải của P ngắn đi 1. Khi đó, nếu đỉnh P ở trạng thái LH thì sau khi loại, đỉnh P sẽ bị lệch bên trái, trong trường hợp này ta cần sử dụng hàm LeftBalance (P) để lập lại sự cân bằng cho đỉnh P. Nếu đỉnh P không ở trạng thái LH thì sau khi loại, ta cũng cần phải xác định lại trạng thái cân bằng của đỉnh P. Chúng ta cài đặt các hành động cần thực hiện đó trong môt hàm BalanceLeft (P, shorter) như sau:

```
void   BalanceLeft (Node* &  P, bool  shorter)
// Xác định lại trạng thái cân bằng cho đỉnh P
// Biến shorter nhận giá trị true nếu cây P ngắn đi 1 và nhận giá trị
// false nếu độ cao cây P không đổi.
// Hàm này được sử dụng khi đỉnh bị loại thuộc cây con phải của P và
// phép loại làm cho cây con phải của P ngắn đi 1.
{
  switch (P → balance)
  {
    case  LH :
       LeftBalance(P) ;
       shorter = true;
       break ;
    case   EH :
       P → balance = LH ;
       shorter = false ;
       break ;
     case   RH :
       P → balance = EH ;
       shorter = true :
  }
}
```

Tương tự, khi đỉnh bị loại thuộc cây con trái của đỉnh P, và phép loại làm cho cây con trái của P ngắn đi 1, chúng ta cần sử dụng hàm

BalanceRight (P, shorter). Hàm này được cài đặt tương tự như hàm BalanceLeft.

Chúng ta cần một hàm thực hiện nhiệm vụ cắt bỏ đỉnh ngoài cùng bên phải của cây R, dữ liệu chứa trong đỉnh bị cắt bỏ sẽ được lưu trong đỉnh Q.

```
void   Remove (Node* &  R, Node* Q, bool  shorter)
{
  if (R → right = = NULL)
     {
        Q → data = R → data ;
        Node* Ptr = R ;
        R = R → left ;
        shorter = true ;
        delete   Ptr ;
     }
  else  {
          Remove (R → right, Q, shorter) ;
          if (shorter)
                      BalanceLeft (R, shorter) ;
         }
}
```

Khi đã tìm ra đỉnh chứa dữ liệu với khoá k là đỉnh Q, chúng ta sẽ sử dụng hàm Del (Q, shorter) để loại bỏ đỉnh Q khỏi cây.

```
void   Del (Node* &  Q, bool &  shorter)
{
  if (Q → right = = NULL)
     {
       Q = Q → left ;
        shorter = true ;
     }
  else if (Q → left = = NULL)
     {
       Q = Q → right ;
       shorter = true ;
     }
        else  {
                  Remove (Q → left, Q, shorter)
                  if (shorter)
```

```
                    BalanceRight (Q, shorter) ;
            }
    }
```

Bây giờ, sử dụng các hàm BalanceLeft, BalanceRight và hàm Del, chúng ta có thể cài đặt hàm đệ quy Delete như sau:

```
void   Delete (Node* & P, keyType  k, bool  shorter)
{
  if  (P! = NULL)
  if  (k < P → data.key)
      {
        Delete (P → left, k, shorter) ;
        if (shorter)
                BalanceRight (P, shorter) ;
      }
  else if (k > P → data.key)
        {
          Delete (P → right, k, shorter);
          if (shorter)
                  BalanceLeft (P, shorter) ;
        }
       else  Del (P, shorter) ;
}
```

Trên đây chúng ta đã cài đặt phép xen và phép loại trên cây AVL bởi các hàm đệ quy. Có thể cài đặt các phép toán xen, loại bởi các hàm không đệ quy được không? Đương nhiên là có, nhưng chúng ta cần phải sử dụng một ngăn xếp để lưu lại các đỉnh trên đường đi từ gốc tới đỉnh mới xen vào, hoặc đường đi từ gốc tới đỉnh bị cắt bỏ.

11.3 CÂY ĐỎ - ĐEN

Trong mục này chúng ta trình bày một dạng cây cân bằng khác: cây đỏ - đen. Trong cây đỏ - đen, các đỉnh của cây sẽ được sơn đỏ hoặc đen, và áp đặt các điều kiện về màu của các đỉnh nhằm hạn chế sự chênh lệch nhiều về số đỉnh giữa hai nhánh của mọi đỉnh. Cây đỏ - đen cũng cho phép ta thực hiện các phép toán tập động trong thời gian O(logn).

Trong cây nhị phân một đỉnh được xem là **đỉnh không đầy đủ**, nếu nó có ít hơn 2 đỉnh con (tức là nó là đỉnh lá hoặc chỉ có một đỉnh con trái hoặc con phải).

Định nghĩa 11.2. Cây đỏ - đen là cây tìm kiếm nhị phân, các đỉnh của cây được sơn đỏ hoặc đen và thoả mãn các điều kiện sau:

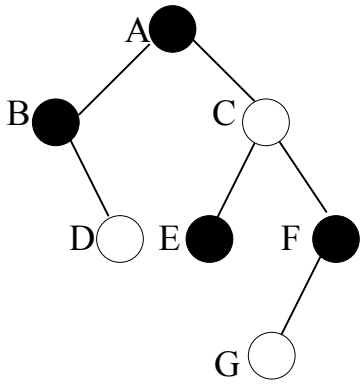
1.(Luật đỏ) Nếu một đỉnh được sơn đỏ thì các đỉnh con (nếu có) của nó được sơn đen.

2.(Luật đường) Mọi đường đi từ gốc tới đỉnh không đầy đủ có cùng số đỉnh đen.

Từ định nghĩa trên, chúng ta suy ra một số tính chất sau đây của cây đỏ - đen:

- Mọi đường đi từ một đỉnh bất kỳ tới đỉnh không đầy đủ có cùng số đỉnh đen. Do đó cây con của cây đỏ - đen là cây đỏ - đen.
- Nếu một đỉnh không đầy đủ là lá thì nó có thể được sơn đỏ hoặc đen. Song nếu đỉnh không đầy đủ có một con thì nó phải được sơn đen và con của nó phải được sơn đỏ (tại sao?)
- Nếu một đỉnh được sơn đỏ thì các đỉnh con (nếu có) và đỉnh cha (nếu có) của nó phải được sơn đen.

Ví dụ. Cây trong hình 11.6 là cây đỏ đen, trong đó các đỉnh tô đen là đỉnh đen, các đỉnh không tô là đỉnh đỏ. Các đỉnh không đầy đủ là B, D, E, F, G, đường đi từ gốc tới các đỉnh này đều có số đỉnh đen là 2.



Hình 11.6. Một cây đỏ đen

Sau đây chúng ta chứng minh một định lý quan trọng về độ cao của cây đỏ đen.

Định lý 11.2. Cây đỏ đen n đỉnh có độ cao nhiều nhất là 2log(n + 1).

Chúng ta cần đến định nghĩa sau. Số đỉnh đen trên đường đi từ đỉnh v, không kể đỉnh v, đến đỉnh không đầy đủ được gọi là **độ cao đen của đỉnh v**, và được ký hiệu là bh(v).

Trước hết ta chứng minh bằng quy nạp rằng, cây con gốc v chứa ít nhất $2^{bh(v)} - 1$ đỉnh . Thật vậy, nếu v là đỉnh không đầy đủ, từ nhận xét sau định nghĩa cây đỏ đen ta có bh(v) = 0, và do đó $2^{bh(v)} - 1 = 2^0 - 1 = 0$. Khẳng định đương nhiên được thoả mãn. Bây giờ giả sử v có đầy đủ cả hai con, u là đỉnh con của v và khẳng định đã đúng cho cây con gốc u. Rõ ràng là, nếu u là đỉnh đỏ thì bh(u) = bh(v), còn nếu u là đen thì bh(u) = bh(v) – 1. Theo giả thiết quy nạp, cây con gốc v chứa ít nhất

$$2(2^{bh(u)} - 1) + 1 \text{ đỉnh.}$$

Thay bh(u) = bh(v) – 1, ta có cây con gốc v chứa ít nhất

$$2(2^{bh(v)-1} - 1) + 1 = 2^{bh(v)} - 1 \text{ đỉnh.}$$

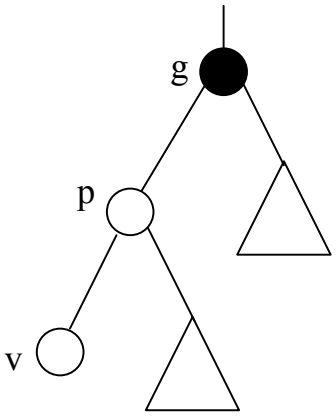
Mặt khác, từ luật đỏ ta suy ra rằng, có ít nhất một nửa số đỉnh trên đường đi từ gốc tới đỉnh không đầy đủ là đen. Do đó, nếu cây đỏ - đen có độ cao h thì độ cao đen của gốc ít nhất là h / 2. Vậy nếu cây đỏ - đen có n đỉnh thì:

$$n \geq 2^{h/2} - 1 \text{ hay } h \leq 2\log(n+1).$$

Sau đây chúng ta xét xem, nếu tập dữ liệu được cài đặt dưới dạng cây đỏ - đen thì các phép toán tập động sẽ được thực hiện như thế nào.

Bởi vì cây đỏ - đen là cây tìm kiếm nhị phân, nên các phép toán hỏi được tiến hành như trên cây tìm kiếm nhị phân. Từ định lý 11.2 ta suy ra rằng, các phép toán hỏi trên cây đỏ - đen chỉ cần thời gian O(logn). Cũng như trên cây AVL, các phép toán xen, loại trên cây đỏ - đen khá phức tạp. Điều đó là do mỗi khi thực hiện phép xen, loại như trên cây tìm kiếm nhị phân thì luật đỏ và luật đường có thể bị vi phạm, và lúc đó chúng ta cần phải biến đổi cây để cho cây tuân thủ hai luật đó.

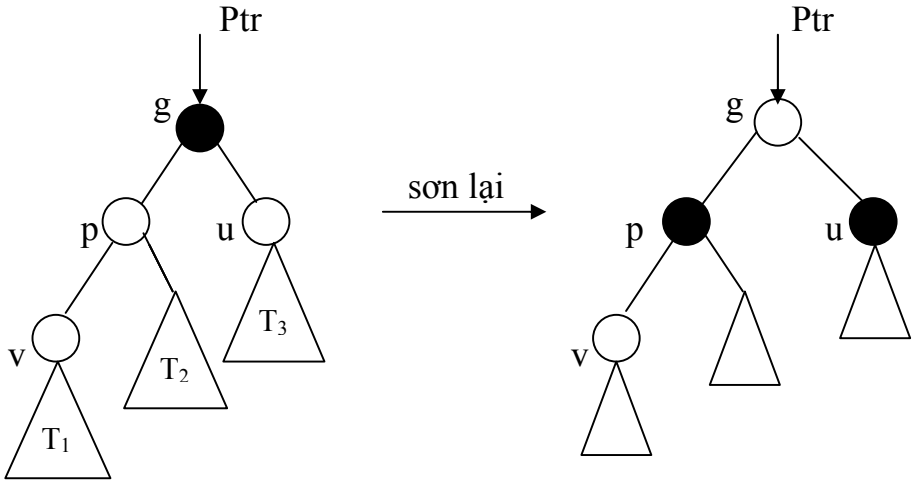
Phép toán xen. Giả sử chúng ta cần xen vào cây đỏ - đen một đỉnh mới v chứa dữ liệu là d. Đầu tiên ta áp dụng thuật toán xen vào cây tìm kiếm nhị phân để xen vào cây đỏ - đen đỉnh v và sơn đỏ đỉnh v. Như vậy, luật đường vẫn được thoả mãn. Nếu ta xen vào cây rỗng, thì cây trở thành cây chỉ có một đỉnh gốc v và đương nhiên cả hai luật đỏ và luật đường đều được thoả mãn. Giả sử đỉnh v được xen vào cây không rỗng, và cha của v là đỉnh p. Nếu p là đen, thì cây vẫn còn là cây đỏ - đen. Nhưng nếu p là đỏ thì luật đỏ bị vi phạm. Trường hợp p là gốc cây, ta chỉ cần sơn lại p thành đen. Nếu p không phải là gốc cây, ta giả sử đỉnh cha của p là đỉnh g. Vì p là đỏ, nên g phải là đen. Chúng ta có hoàn cảnh, chẳng hạn như sau:



Đây là một trong các tình huống vi phạm luật đỏ: đỉnh p là con trái của g. Sau đây chúng ta sẽ xét từng tình huống mà luật đỏ bị vi phạm (đỉnh p đỏ và đỉnh con nó là v cũng đỏ) và các phép biến đổi cây nhằm khôi phục lại luật đỏ đồng thời vẫn bảo tồn luật đường.

Trước hết ta xét trường hợp đỉnh p là con trái của g. Khi p và v là hai đỉnh cha – con cùng được sơn đỏ và p là con trái của g, chúng ta có ba mẫu biến đổi bằng các phép quay và sơn lại nhằm làm cho cây con gốc g trở thành cây đỏ - đen.

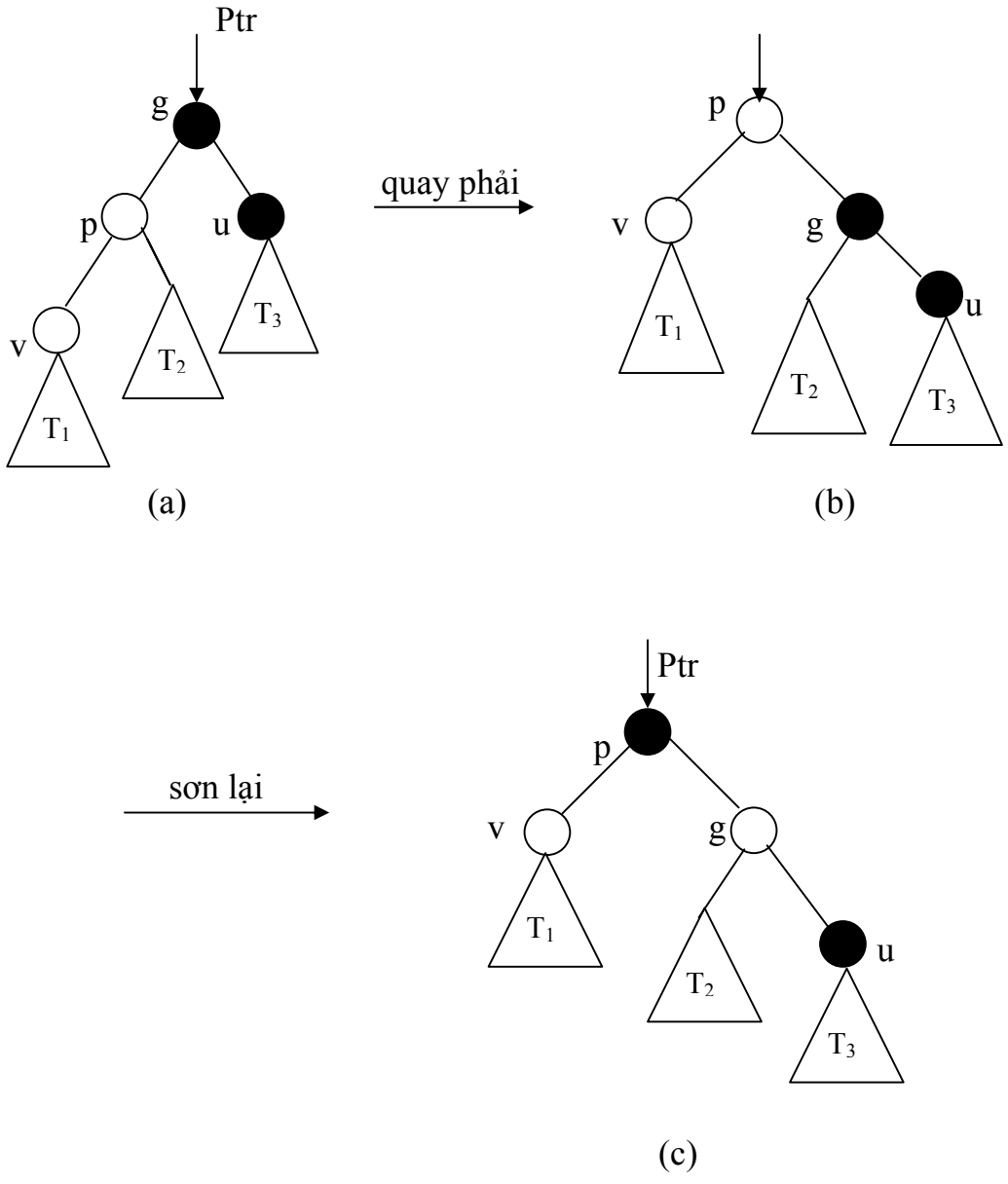
- **Mẫu 1**. Đỉnh g có con phải là đỉnh u được sơn đỏ. Trường hợp này ta chỉ cần sơn lại: sơn p, u đen và sơn g đỏ (xem hình 11.7)



Hình 11.7. Mẫu biến đổi 1

Chú ý rằng, các cây T_1, T_2, T_3 là cây đỏ - đen, cây T_2 có thể rỗng. Sau khi sơn lại, nếu cha của đỉnh g lại là đỏ, thì luật đỏ lại bị vi phạm.

- **Mẫu 2**. Đỉnh g không có con phải hoặc có con phải là đỉnh u được sơn đen và v là con trái của p (hình 11.8a). Trong trường hợp này, ta biến đổi cây như sau: đầu tiên quay phải đỉnh g để nhận được cây trong hình 11.8b, sau đó sơn lại p thành đen, g thành đỏ, ta nhận được cây đỏ đen như trong hình 11.8c.

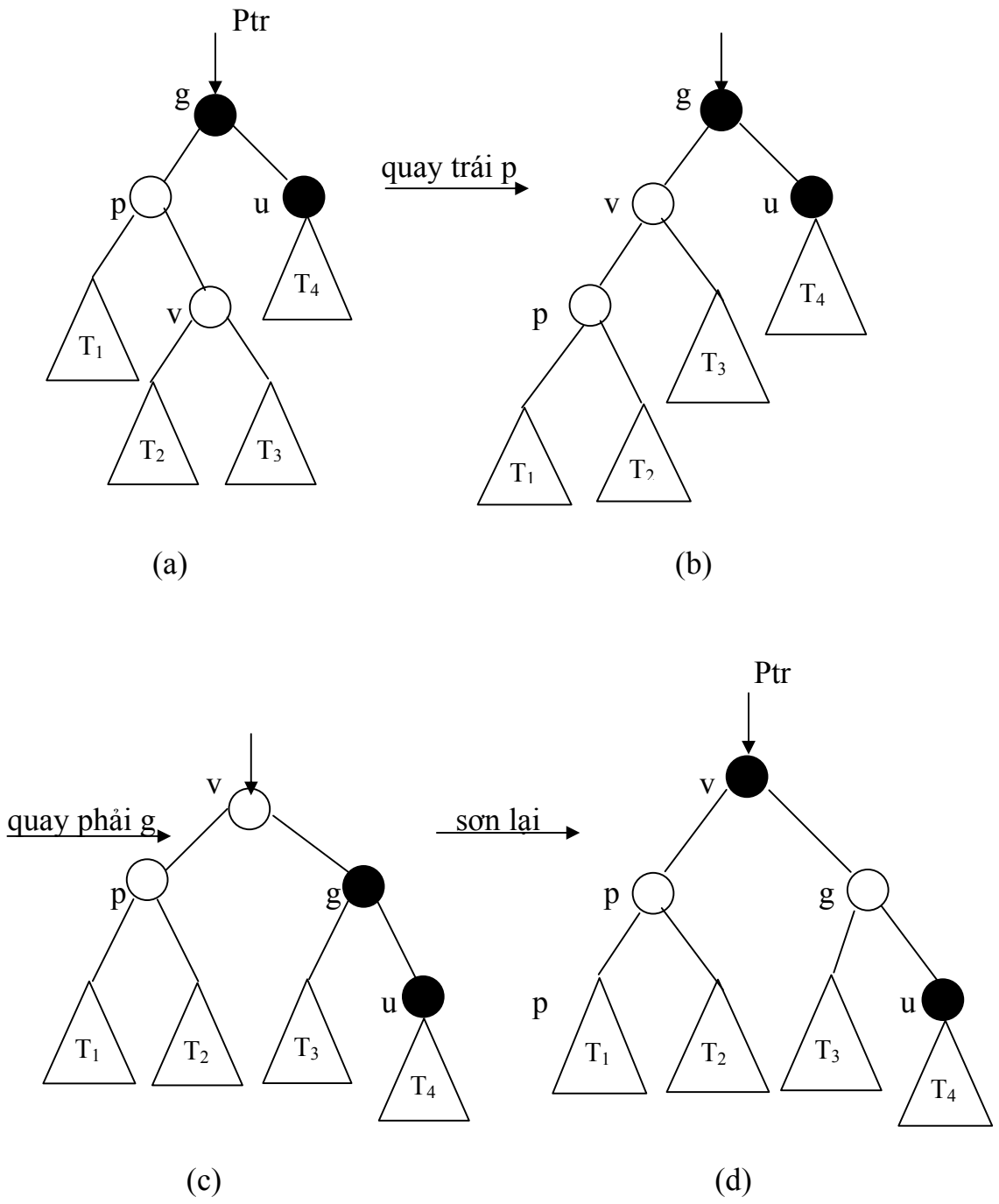


Hình 11.8. Mẫu biến đổi 2

Cần lưu ý rằng, trong hoàn cảnh 11.8a, cây con T_2 sẽ không rỗng và gốc s của nó là con phải của p, mà p là đỏ nên s là đen, và do đó trong cây 11.8c, đỉnh g thoả mãn luật đỏ.

- **Mẫu 3**. Đỉnh g không có con phải hoặc có con phải là đỉnh u được sơn đen, và v là con phải của p (xem hình 11.9a). Gặp hoàn cảnh này, đầu tiên ta quay trái đỉnh p để nhận được cây trong hình 11.9b, rồi

quay phải đỉnh g để có cây trong hình 11.9c, sau đó sơn lại đỉnh v thành đen, đỉnh g thành đỏ, ta có cây hình 11.9d.



Hình 11.9. Mẫu biến đổi 3

Chúng ta có nhận xét rằng, sau các mẫu biến đổi 2 và 3 thì cây con Ptr trở thành cây đỏ - đen và gốc của nó được sơn đen, do đó toàn bộ cây trở thành cây đỏ - đen.

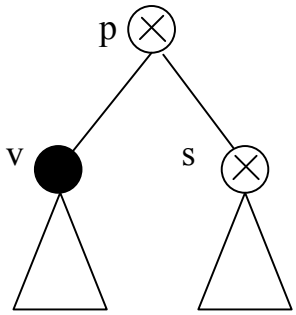
Trường hợp p là con phải của g, chúng ta sẽ có thêm hai mẫu biến đổi nữa là mẫu 4 (đối xứng qua gương của mẫu 2) và mẫu 5 (đối xứng qua gương của mẫu 3)

Tổng kết lại, thuật toán xen vào cây đỏ - đen là như sau. Đầu tiên áp dụng thuật toán xen vào cây tìm kiếm nhị phân đỉnh mới v, đỉnh v được sơn đỏ. Sau đó đi từ đỉnh v lên gốc, nếu hai đỉnh cha con cùng là đỏ thì ta áp dụng một trong 5 mẫu biến đổi đã trình bày, Nếu một trong 4 mẫu 2, 3, 4, 5 được áp dụng thì cây đã trở lại là cây đỏ - đen và dừng lại. Nếu mẫu 1 được áp dụng thì luật đỏ lại có thể bị vi phạm, song hai đỉnh cha – con cùng đỏ đã được đẩy lên trên. Trong trường hợp xấu nhất, hai đỉnh cha – con cùng đỏ được đẩy lên trên cùng, đỉnh cha là gốc cây. Lúc này ta chỉ cần sơn lại đỉnh cha đen.

Phép toán loại. Giả sử chúng ta cần loại khỏi cây đỏ - đen đỉnh chứa dữ liệu với khoá là k. Đầu tiên ta cũng áp dụng thuật toán loại trên cây tìm kiếm nhị phân. Nhớ lại thuật toán này, đỉnh thực sự bị cắt bỏ là đỉnh không đầy đủ, bởi vì trường hợp đỉnh chứa khoá k có đầy đủ cả hai con được đưa về cắt bỏ đỉnh ngoài cùng bên phải của cây con trái của đỉnh chứa khoá k. Giả sử đỉnh bị cắt bỏ là đỉnh v. Nếu v là đỏ thì sau khi cắt bỏ nó, cây vẫn còn là cây đỏ - đen, chẳng hạn khi ta cắt bỏ đỉnh D hoặc G trong cây hình 11.6. Nếu v có một con là u, trường hợp này v phải đen và u phải là đỏ, sau khi cắt bỏ v, đỉnh u ở vị trí của v, ta sơn lại u thành đen, và cây vẫn còn là cây đỏ - đen, chẳng hạn khi ta cắt bỏ đỉnh B hoặc F trong cây hình 11.6. Trường hợp còn lại là khi đỉnh bị cắt bỏ v là đen và là lá, chẳng hạn đỉnh E trong hình 11.6.

Chúng ta sẽ gọi **trọng số đen** của cây đỏ - đen là số đỉnh đen trên đường đi từ gốc tới đỉnh không đầy đủ. Hiển nhiên là , trong cây đỏ - đen thì trọng số đen của cây con trái và cây con phải của mọi đỉnh là bằng nhau, hay nói cách khác, mọi đỉnh của cây đỏ - đen đều ở trạng thái cân bằng (theo trọng số đen). Bây giờ nếu ta cắt bỏ đỉnh lá đen v, và gọi cha của v là p, thì đỉnh p bị mất cân bằng, luật đường đi từ đỉnh p bị vi phạm.

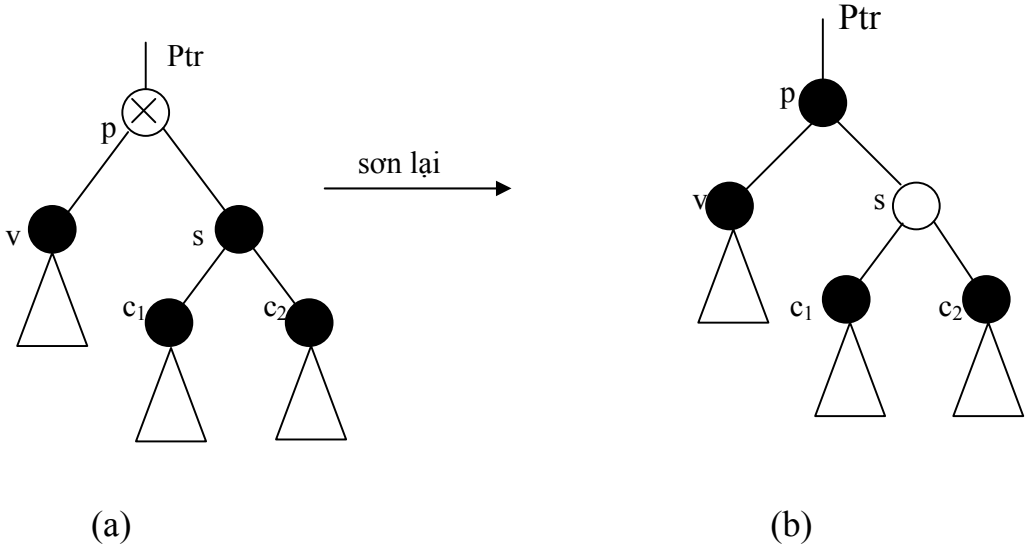
Tổng quát, luật đường bị vi phạm khi thực hiện phép loại là như sau. Đỉnh p có hai con trái phải đều là cây đỏ - đen, một cây con có trọng số đen ít hơn cây con kia 1. Cây con nặng hơn có gốc là s (nó không thể rỗng). Cây con nhẹ hơn có gốc là v và được sơn đen, nếu nó không rỗng (nó có thể rỗng, chẳng hạn như khi ta cắt bỏ đỉnh E trong cây hình 11.6)



đỉnh p và đỉnh s có thể là đỏ hoặc đen, chúng được đánh dấu chéo. Có hai khả năng, cây con nhẹ hơn là cây con trái hoặc phải của p.

Trước hết ta xét trường hợp đỉnh p nhẹ bên trái. Trong trường hợp này chúng ta có bốn mẫu điều chỉnh để khôi phục lại luật đường tại p đồng thời vẫn bảo tồn luật đỏ.

Mẫu điều chỉnh 1. Đỉnh s là đen và nó có thể là lá hoặc có cả hai con c_1 và c_2 đều là đen như trong hình 11.10a. Khi đó ta chỉ cần sơn lại đỉnh s thành đỏ, và nếu p đỏ thì sơn nó thành đen, ta thu được cây đỏ - đen trong hình 11.10b.



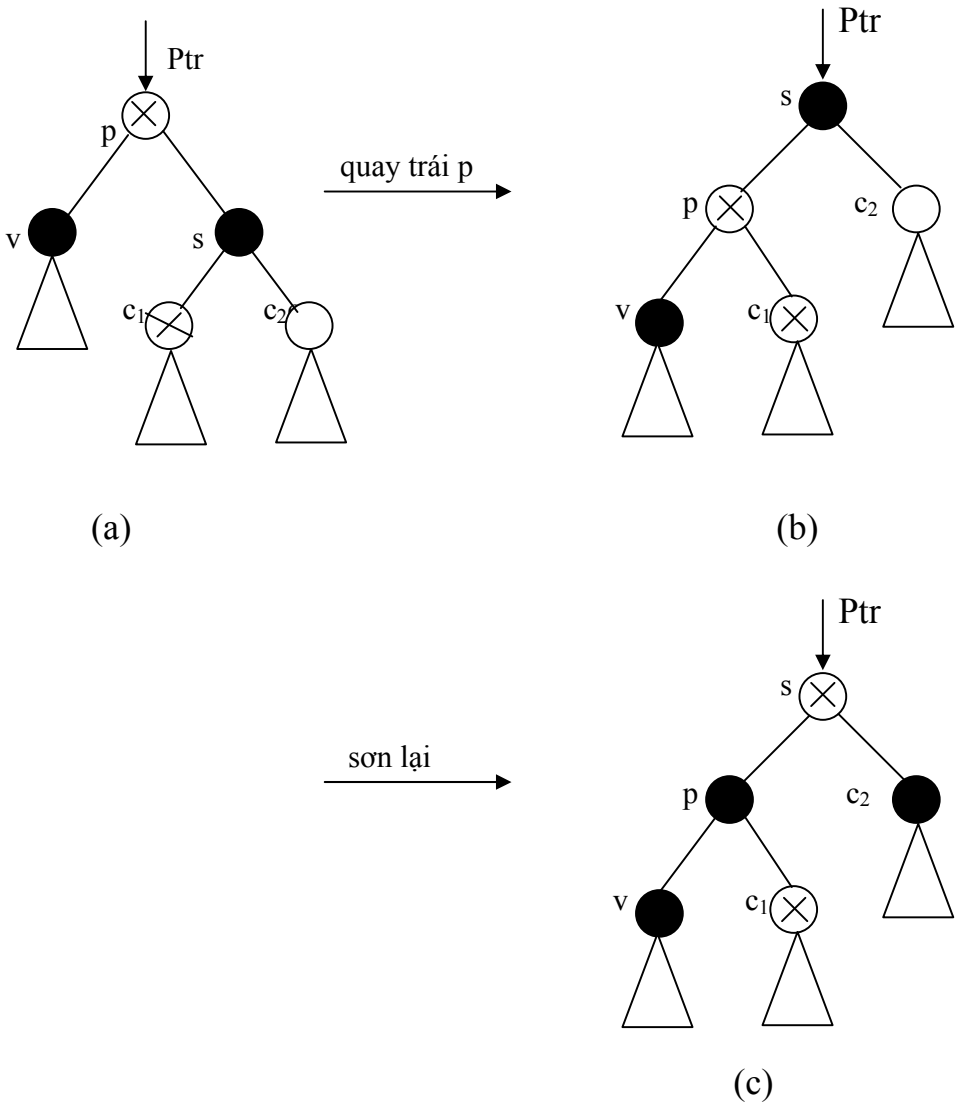
(a) (b)

Hình 11.10. Mẫu điều chỉnh 1.

Một nhận xét quan trọng. Nếu đỉnh p vốn dĩ là đỏ thì cây nhận được sau khi sơn lại có trọng số đen như trước khi thực hiện phép loại, và do đó sau khi sơn lại, tất cả các đỉnh từ p lên gốc đều cân bằng. Song nếu p vốn đã

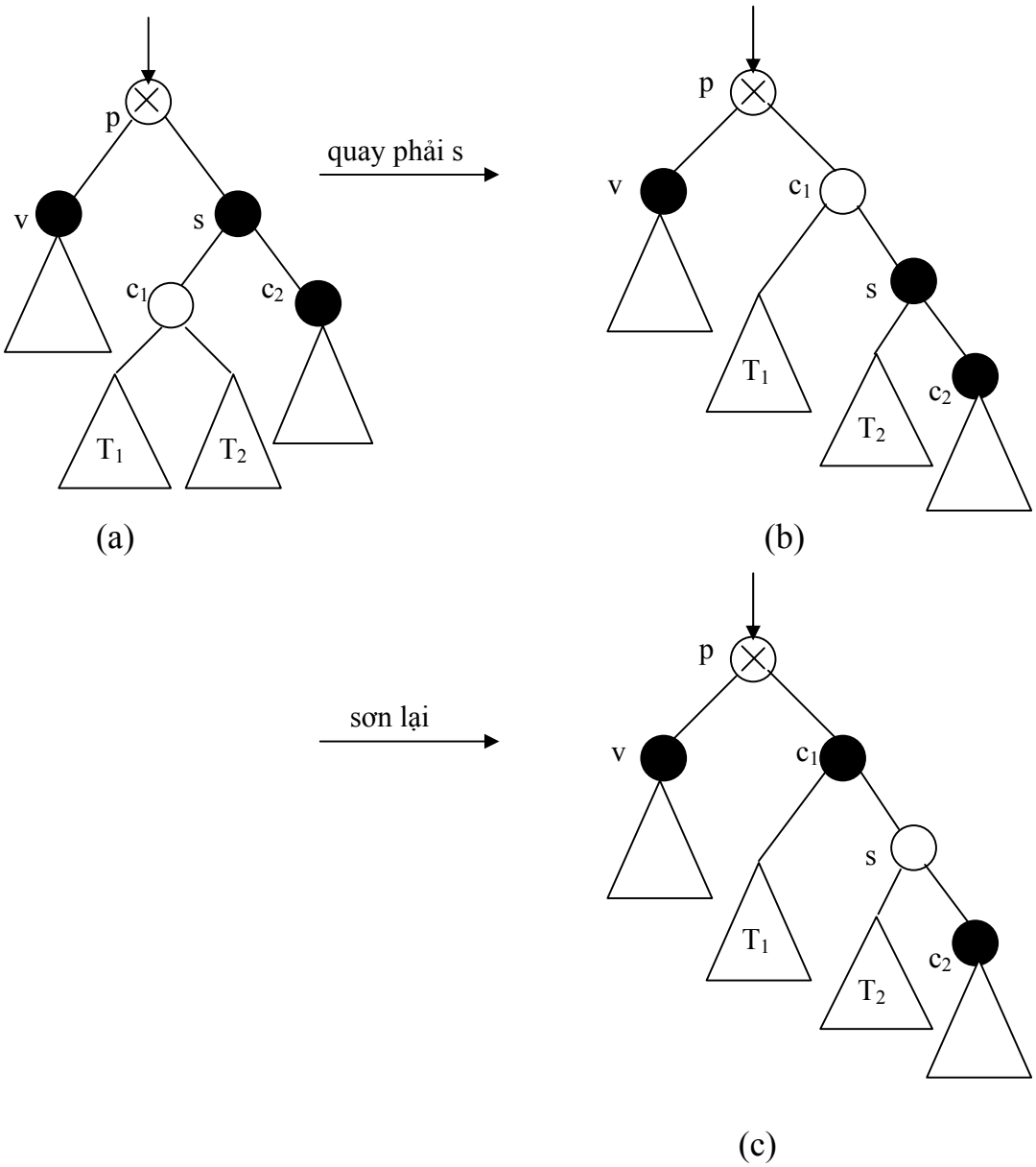
là đen thì trọng số đen của cây con gốc p giảm đi 1 so với trước khi thực hiện phép loại, và do đó đỉnh cha (nếu có) của đỉnh p lại mất cân bằng.

Mẫu điều chỉnh 2. Đỉnh s là đen và đỉnh con phải c_2 của nó là đỏ, như hình 11.11a. Trường hợp này, trước hết ta quay trái đỉnh p để có cây trong hình 11.11b. Sau đó sơn lại đỉnh c_2 thành đen và trao đổi màu của hai đỉnh s và p, ta thu được cây trong hình 11.11c. Dễ dàng thấy rằng, cây ở hình 11.11c là cây đỏ - đen có trọng số đen như trước khi thực hiện phép loại. Do đó, sau khi thực hiện mẫu điều chỉnh 2, ta có thể dừng lại.



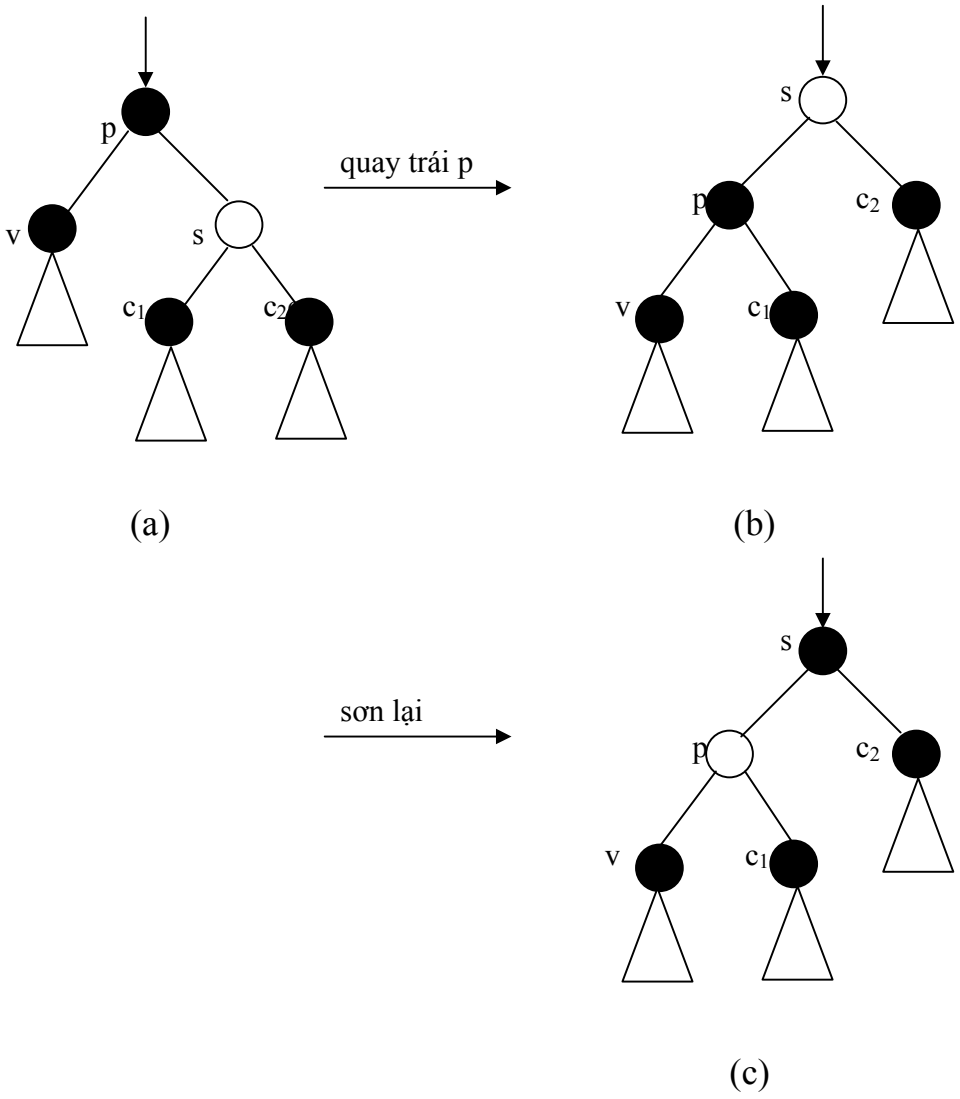
Hình 11.11. Mẫu điều chỉnh 2.

Mẫu điều chỉnh 3. Đỉnh s là đen, đỉnh con trái của nó là c_1 đỏ và đỉnh con phải c_2 đen (hình 11.12a). Ta thực hiện điều chỉnh như sau. Đầu tiên quay phải đỉnh s, thu được cây hình 11.12b. Sau đó sơn lại s thành đỏ, c_1 thành đen, ta có cây hình 11.12c. Không khó khăn thấy rằng, làm như vậy cây con phải của p vẫn còn là cây đỏ - đen và có trọng số đen không thay đổi, tức là đỉnh p vẫn nhẹ bên trái, nhưng cây hình 11.12c có dạng như hình 11.11a và do đó ta chỉ cần thực hiện tiếp mẫu điều chỉnh 2.



Hình 11.12. Mẫu điều chỉnh 3.

Mẫu điều chỉnh 4. Đỉnh s có màu đỏ, và do đó nó phải có hai con đều là đen và cha nó là p cũng đen, như hình 11.13a. Trong trường hợp này, ta quay trái đỉnh p để có cây hình 11.13b, sau đó sơn lại s thành đen, và p thành đỏ, ta có cây hình 11.13c. Chúng ta có nhận xét rằng, trong cây hình 11.13c, đỉnh p vẫn nhẹ bên trái, song đỉnh con phải của nó là c_1 có màu đen. Tức là ta đã quy về một trong các mẫu điều chỉnh 1, 2, 3.



Hình 11.13. Mẫu điều chỉnh 4.

Trong trường hợp đỉnh p nhẹ bên phải ta có 4 mẫu điều chỉnh là đối xứng qua gương của 4 mẫu điều chỉnh trên.

Tóm lại, thuật toán loại trên cây đỏ - đen như sau. Trước hết áp dụng thuật toán loại trên cây tìm kiếm nhị phân. Sau đó, đi từ đỉnh bị cắt bỏ lên gốc cây, gặp đỉnh mất cân bằng thì áp dụng các mẫu điều chỉnh để làm cho đỉnh đó trở thành cân bằng. Trong trường hợp xấu nhất, sự mất cân bằng có thể truyền lên tận gốc. Chú ý rằng, thời gian thực hiện mỗi mẫu điều chỉnh là O(1), và do đó thời gian thực hiện phép loại là tỉ lệ với độ cao của cây đỏ - đen. Do đó, theo định lý 11.2, thời gian thực hiện phép loại trên cây đỏ - đen là O(logn).

11.4 CẤU TRÚC DỮ LIỆU TỰ ĐIỀU CHỈNH

Các CTDL mà chúng ta đã đưa ra từ trước tới nay, điển hình là cây tìm kiếm nhị phân, cây AVL, cây đỏ - đen, đều có các điểm chung sau. Các phép toán hỏi không làm thay đổi CTDL, chỉ khi chúng ta thực hiện các phép toán, như phép xen hoặc phép loại, bắt buộc chúng ta phải điều chỉnh CTDL cho nó thỏa mãn các điều kiện đã áp đặt CTDL. Chẳng hạn, khi thực hiện phép xen (loại) trên cây đỏ - đen làm cho luật đỏ hoặc luật đường bị vi phạm, chúng ta mới phải biến đổi cây để nó trở lại là cây đỏ đen. Điều đó có nghĩa là khi tổ chức dữ liệu cũng như sau này khi thực hiện các phép toán trên tập dữ liệu đã được cấu trúc, chúng ta không hề quan tâm tới các dữ liệu nào được truy cập thường xuyên hơn các dữ liệu khác. Nhưng thực tế người ta đã kiểm nghiệm rằng, 90 phần trăm các lần truy cập dữ liệu là truy cập tới 10 phần trăm dữ liệu. Mặt khác, chúng ta mong muốn rằng, khi chúng ta truy cập tới một dữ liệu nào đó thì các lần truy cập sau này tới dữ liệu đó sẽ nhanh hơn. Một thực tế nữa là, trong các chương trình áp dụng, rất ít khi chúng ta sử dụng một CTDL để biểu diễn một tập dữ liệu mà chỉ thực hiện một vài phép toán trên CTDL đó, thông thường cần phải thực hiện một dãy rất lớn các phép toán. Bởi vậy người ta đưa ra ý tưởng: khi thực hiện các phép toán trên một CTDL đã lựa chọn, người ta tiến hành điều chỉnh CTDL ngay cả đối với các phép toán không đòi hỏi phải điều chỉnh (chẳng hạn, các phép toán hỏi), nhằm làm cho các phép toán thực hiện sau đó sẽ nhanh hơn và do đó có thể giảm thời gian thực hiện một dãy dài các phép toán. Các CTDL như thế được gọi là **CTDL tự điều chỉnh (self – adjusting data structure)**. Trong thực tế, người ta thường sử dụng các chiến lược heuristic để điều chỉnh CTDL. Để làm ví dụ, chúng ta nói đến các **danh sách tự điều chỉnh**.

Giả sử chúng ta có một tập dữ liệu được lưu dưới dạng danh sách. Nếu chúng ta biết được xác suất truy cập tới mỗi phần tử của danh sách, và

các lần truy cập dữ liệu là độc lập với nhau, khi đó với quan điểm thống kê, cách tốt nhất là chúng ta lưu danh sách theo thứ tự giảm dần của xác suất truy cập và không bao giờ sắp xếp lại chúng. Tuy nhiên trong thực tế, rất ít khi chúng ta có thể biết được các xác suất đó. Do đó, chúng ta có thể sử dụng các chiến lược heuristic sau đây để tổ chức lại danh sách mỗi khi ta thực hiện một phép toán trên danh sách.

- **Chiến lược chuyển lên phía trước**. Mỗi khi tìm kiếm thành công một phần tử thì chuyển phần tử đó lên đầu danh sách và giữ nguyên thứ tự của các phần tử còn lại. Khi xen một phần tử mới vào danh sách thì ta đặt nó ở đầu danh sách.
- **Chiến lược hoán vị**. Mỗi khi tìm kiếm thành công một phần tử trong danh sách thì ta hoán vị nó với phần tử đứng ngay trước nó. Khi xen một phần tử mới vào danh sách, thì nó được đặt ở đầu danh sách.

Với các chiến lược này, thời gian tìm kiếm một phần tử trong trường hợp xấu nhất vẫn là O(n) tức là vẫn như khi ta tổ chức danh sách theo trật tự bất kỳ và không tiến hành điều chỉnh. Tuy nhiên thực tế chứng tỏ rằng, thời gian thực hiện một dãy tìm kiếm là được cải thiện.

Trong mục 11.6 chúng ta sẽ trình bày một CTDL tự điều chỉnh đặc biệt, được gọi là cây tán loe, và áp dụng một phương pháp phân tích hoàn toàn mới để đánh giá thời gian thực hiện một dãy phép toán trên cây tán loe.

11.5 PHÂN TÍCH TRẢ GÓP

Khi một KDLTT được cài đặt và được sử dụng trong một chương trình áp dụng thì thông thường là một dãy các phép toán của kiểu dữ liệu đó sẽ được thực hiện, chứ ít khi chỉ thực hiện một vài phép toán . Nhưng từ trước đến nay, ta mới chỉ quan tâm đánh giá thời gian chạy của mỗi phép toán riêng biệt, và cụ thể là đánh giá thời gian chạy trong trường hợp xấu nhất và thời gian chạy trung bình của mỗi phép toán. Chúng ta có thể đánh giá thời gian chạy trong trường hợp xấu nhất của một dãy phép toán bằng phương pháp đơn giản sau. Đánh giá thời gian chạy trong trường hợp xấu nhất của mỗi phép toán trong dãy, sau đó lấy tổng để nhận được cận trên của thời gian chạy trong trường hợp xấu nhất của cả dãy phép toán. Tuy nhiên cách đánh giá này là quá thô, bởi vì khi một phép toán được thực hiện, nó có thể làm thay đổi vị trí của các dữ liệu trong cấu trúc và do đó có thể làm cho trường hợp xấu nhất của các phép toán được thực hiện sau không xảy ra. Như vậy thời gian chạy thực tế trong trường hợp xấu nhất của một dãy phép toán có thể thấp hơn nhiều so với tổng thời gian chạy trong trường hợp xấu nhất của mỗi phép toán trong dãy.

Trong mục này chúng ta sẽ đưa ra một phương pháp đánh giá thời gian chạy của một dãy phép toán, được gọi là **phương pháp phân tích trả góp (amortized analysis)**. Phương pháp phân tích trả góp cho phép ta có thể đánh giá cận trên chặt của thời gian chạy của một dãy phép toán. Phương pháp này thường được sử dụng để đánh giá thời gian chạy của một dãy phép toán trên các cấu trúc dữ liệu tự điều chỉnh.

Tư tưởng của phân tích trả góp là chúng ta xem xét các phép toán của dãy trong một mối liên quan, mỗi phép toán khi được thực hiện sẽ ảnh hưởng đến thời gian chạy của các phép toán đứng ở sau trong dãy. Với mỗi phép toán thứ i trong dãy, ta ký hiệu c_i là thời gian chạy thực tế và $\hat{c}_i$ là **thời gian chạy trả góp (amortized running time)**. Chúng ta mong muốn rằng, thời gian chạy thực tế của một dãy n phép toán bị chặn trên bởi thời gian chạy trả góp của dãy phép toán đó, tức là

$$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$$

Một cách tiếp cận hay được sử dụng nhất để xác định thời gian chạy trả góp là **phương pháp tiềm năng (potential method)**.

Tư tưởng của phương pháp này là, chúng ta quan niệm một cấu trúc dữ liệu như một hệ vật lý, mỗi trạng thái của hệ này có một năng lượng tiềm ẩn nào đó. Khi một phép toán được thực hiện, nó có thể làm tăng hoặc giảm tiềm năng của hệ. Hơn nữa, một phép toán khi được thực hiện có thể đưa vào hệ một năng lượng nhiều hơn là năng lượng mà nó tiêu tốn. Năng lượng dư ra này sẽ được sử dụng cho các phép toán đứng sau.

Giả sử D là một cấu trúc dữ liệu, trạng thái ban đầu của nó được ký hiệu là D_o. Trạng thái thứ i của nó (trạng thái của CTDL sau khi ta thực hiện phép toán thứ i trong dãy phép toán) được ký hiệu là D_i. Một **hàm tiềm năng (potential function)** ϕ là hàm xác định trên CTDL D và nhận giá trị là các số thực:

$$\phi : D \rightarrow R$$

Thời gian chạy trả góp của một phép toán trên cấu trúc dữ liệu D (đối với hàm tiềm năng ϕ) được xác định như sau:

$$\hat{c} = c + \Delta\phi(D)$$

tức là thời gian chạy trả góp là thời gian chạy thực tế cộng với số gia năng lượng mà phép toán tạo ra. Như vậy, thời gian chạy trả góp của một dãy n phép toán sẽ là:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_o)$$

Từ công thức này ta suy ra rằng, nếu $\phi(D_n) \geq \phi(D_0)$, tức là năng lượng của hệ sau khi thực hiện một dãy phép toán lớn hơn hoặc bằng năng lượng ban đầu của hệ, thì thời gian chạy trả góp của một dãy phép toán là cận trên thời gian chạy thực tế của dãy phép toán đó. Đặc biệt, chúng ta sẽ có điều đó nếu $\phi(D_0) = 0$ và $\phi(D_i) \geq 0$ với mọi i = 1, 2, .., n.

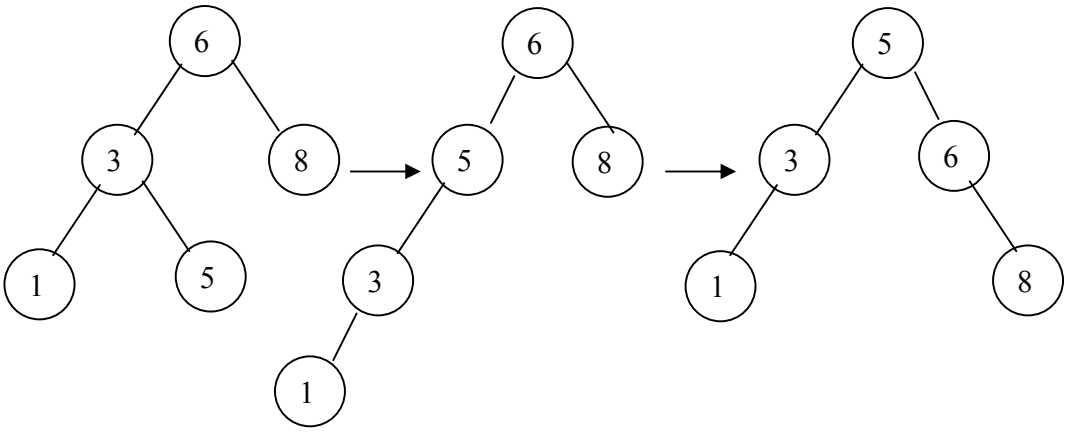
Khó khăn khi áp dụng phương pháp tiềm năng để xác định thời gian chạy trả góp của các phép toán trên một cấu trúc dữ liệu là làm thế nào xác định được hàm tiềm năng trên cấu trúc dữ liệu đó, nó cần phản ánh được hiệu ứng của các phép toán làm thay đổi trạng thái của cấu trúc dữ liệu.

11.6 CÂY TÁN LOE

Với cây AVL hoặc cây đỏ - đen, chúng ta không quan tâm tới tần suất truy cập của các phần tử dữ liệu, mà thay cho điều đó chúng ta luôn luôn đảm bảo cây không bao giờ mất cân bằng tại mọi đỉnh, và do đó thời gian thực hiện các phép toán trên cây là O(logn). Để cài đặt cây AVL hoặc cây đỏ - đen, ta cần phải đưa vào mỗi đỉnh thông tin về sự cân bằng hoặc về màu của đỉnh đó. Trong mục này, chúng ta sẽ đưa vào một cấu trúc dữ liệu rất đặc biệt được sử dụng để cài đặt KDLTT tập động, đó là **cây tán loe (splay tree).**

Cây tán loe là cây tìm kiếm nhị phân, song mỗi phép toán trên cây đi kèm theo thao tác cấu trúc lại cây, được gọi là làm loe cây. Làm loe cây nhằm mục đích giảm bớt tổng thời gian truy cập dữ liệu bằng cách dịch chuyển các dữ liệu được thường xuyên truy cập lên gần gốc cây, và vì vậy sự truy cập tới các dữ liệu đó sẽ nhanh hơn. Ưu điểm của cây tán loe là chúng ta không cần lưu thông tin về sự cân bằng của các đỉnh, và do đó, tiết kiệm được bộ nhớ và sự cài đặt cũng đơn giản hơn.

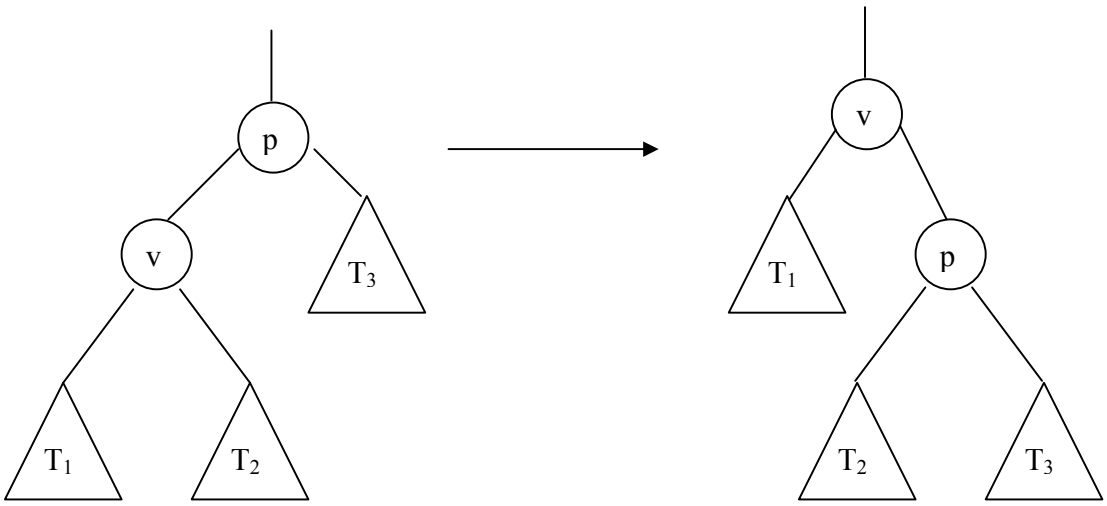
Việc chuyển một đỉnh v bất kỳ lên gốc cây là rất đơn giản bằng cách sử dụng các phép quay cây (trái hoặc phải), mỗi lần quay đỉnh v được chuyển lên 1 mức. Chẳng hạn, chuyển đỉnh 5 lên gốc cây được thực hiện bằng hai phép quay như trong hình 11.14.



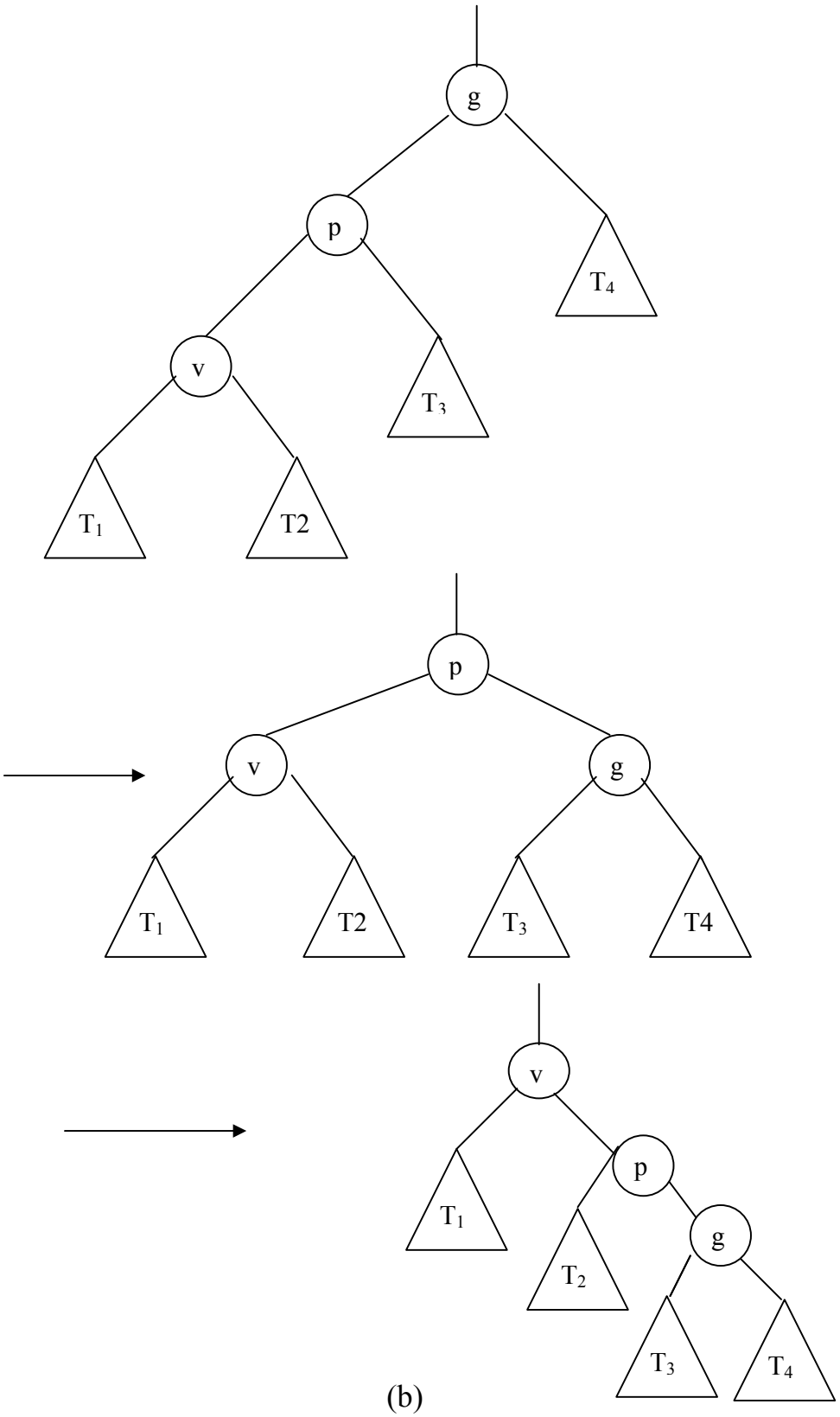
Hình 11.14. Chuyển đỉnh 5 lên gốc.

Tuy nhiên chiến lược quay lên gốc như trên là không tốt, bởi vì trong nhiều hoàn cảnh đỉnh v được đặt lên gốc thì nhiều đỉnh trên v lại bị đưa xuống xa gốc hơn. Do đó, ta làm loe cây bằng chiến lược sau. Trong quá trình làm loe cây, tại mỗi bước ta sẽ sử dụng một trong ba mẫu biến đổi cây sau đây. Giả sử rằng, ta cần làm loe cây tại đỉnh v, đỉnh v có cha là đỉnh p và đỉnh cha của p là g. Mỗi mẫu biến đổi sau đây có hai trường hợp, ta đưa ra một trường hợp, vì trường hợp kia là đối xứng qua gương của trường hợp đưa ra.

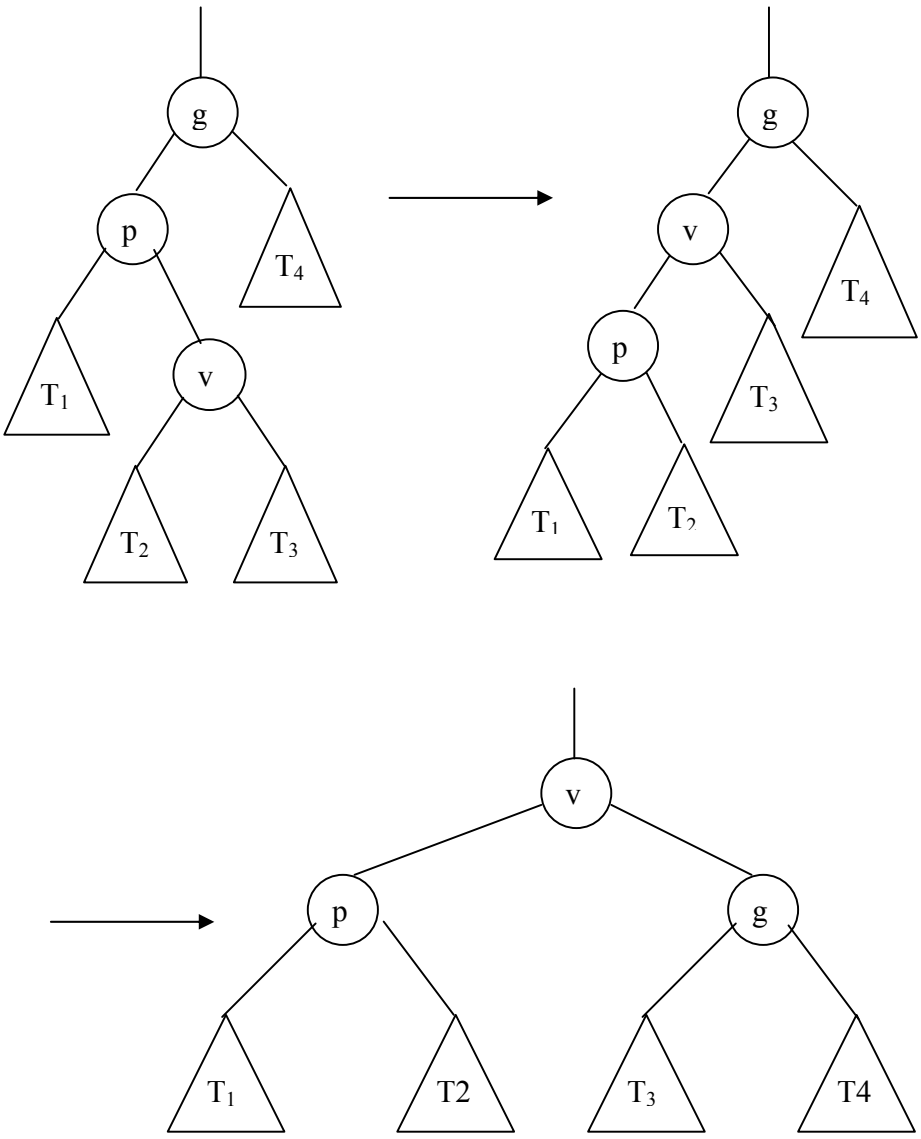
1. **Mẫu zig**. Giả sử p là gốc cây và v là con trái của p. (Trường hợp khác: v là con phải của p). Quay phải đỉnh p, khi đó v sẽ ở gốc cây, như trong hình 11.15a
2. **Mẫu zig – zig**. Đỉnh p không phải là gốc cây, cả p và v đều là con trái. Đầu tiên quay phải đỉnh g, sau đó quay phải đỉnh p, như trong hình 11.15b.
3. **Mẫu zig – zag**. Đỉnh p không phải là gốc cây, p là con trái của g, và v là con phải của p. Đầu tiên quay trái đỉnh p sau đó quay phải đỉnh g, như trong hình 11.15c.



(a)



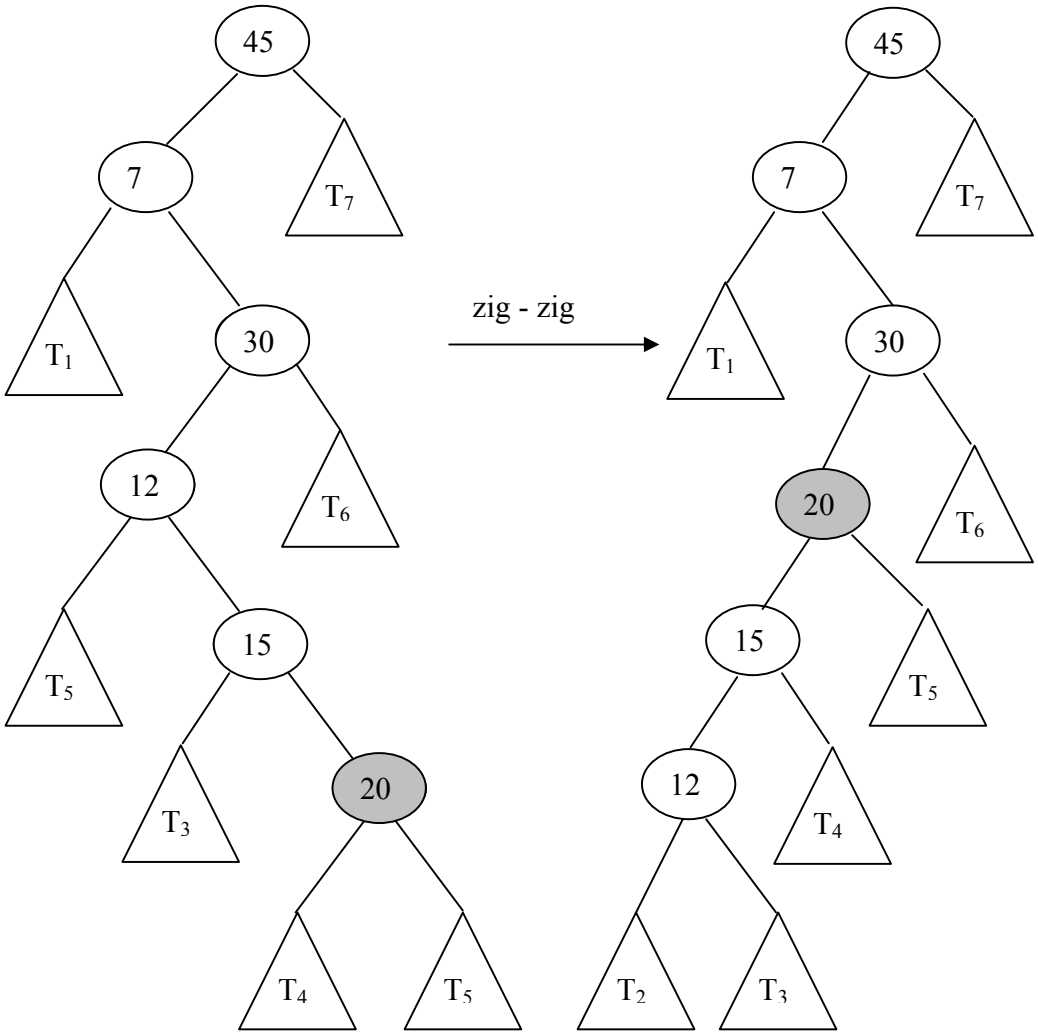
(b)



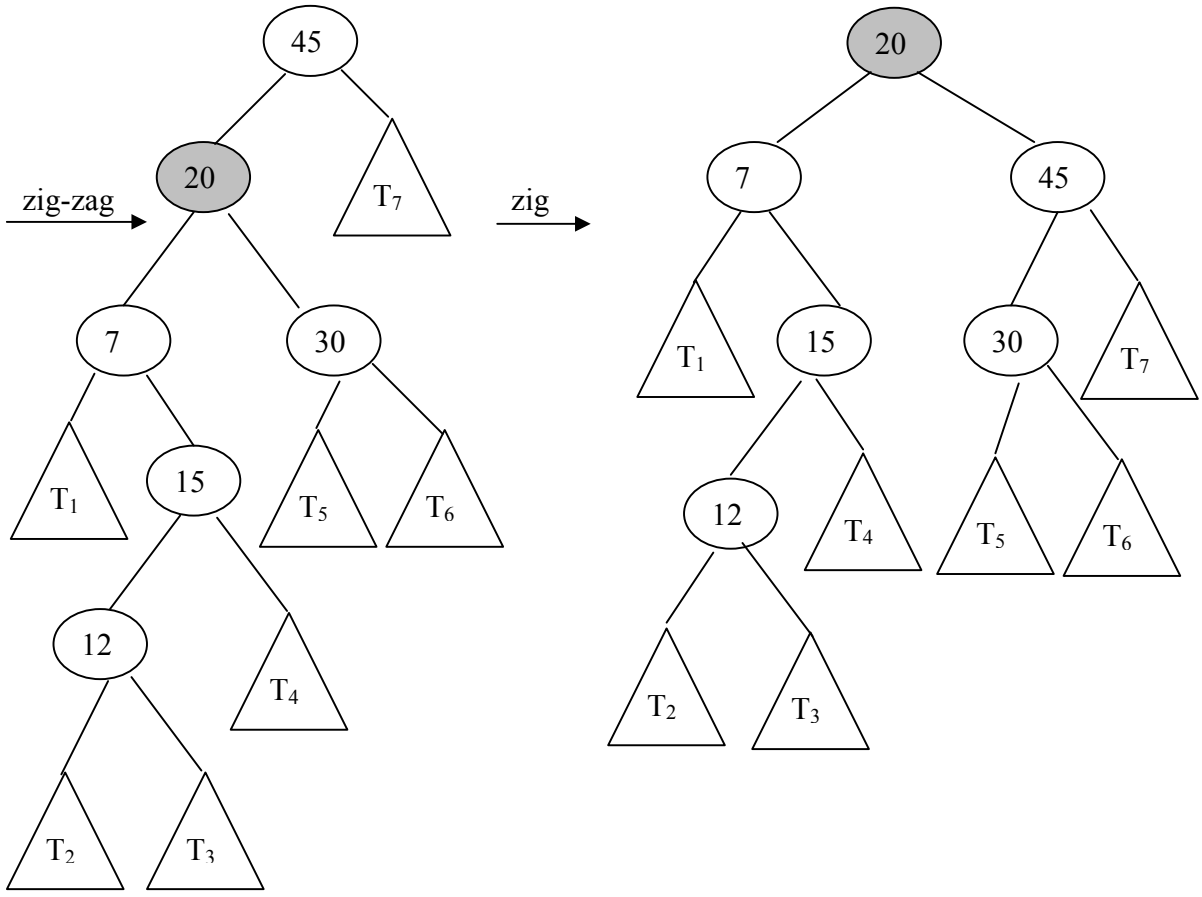
(c)

Hình 11.15. Các mẫu biến đổi cây
(a) Mẫu zig. (b)Mẫu zig – zig. (c)Mẫu zig – zag.

Ví dụ. Giả sử ta cần làm loe cây tại đỉnh 20 của cây trong hình 11.16a. Đầu tiên ta thực hiện mẫu biến đổi zig – zig, ta nhận được cây hình 11.16b. Sau đó thực hiện mẫu zig – zag, ta có cây hình 11.16c, và cuối cùng thực hiện mẫu zig ta đưa được đỉnh 20 lên gốc cây như hình 11.16d.


45
7
T7
T1
30
12
T6
T5
15
T3
20
T4
T5
zig - zig
45
7
T7
T1
30
20
T6
15
T5
12
T4
T2
T3

(a) (b)



Hình 11.16. Làm loe cây tại đỉnh 20

11.6.1 Các phép toán tập động trên cây tán loe

Tất cả các phép toán trên cây tán loe đều được tiến hành theo cách sau. Đầu tiên ta thực hiện phép toán đó như trên cây tìm kiếm nhị phân, rồi sau đó ta điều chỉnh cây bằng các phép làm loe cây.

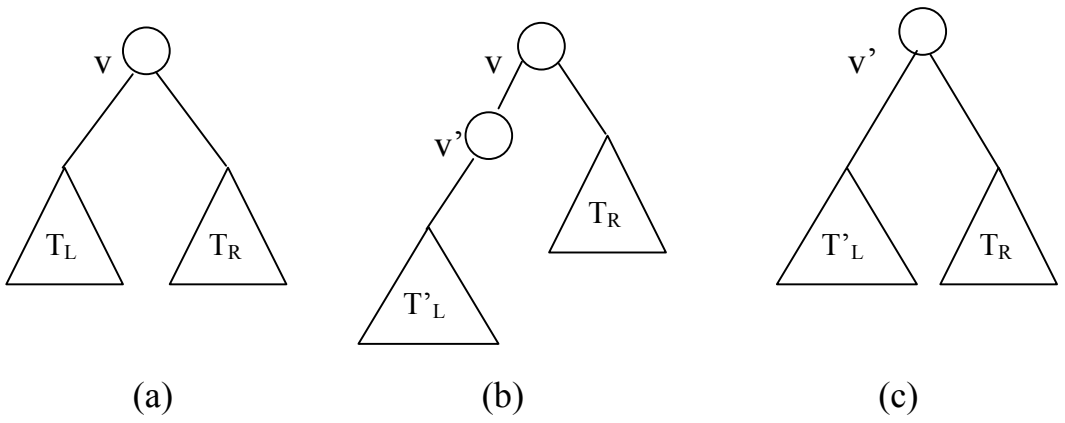
Các phép toán hỏi. Giả sử chúng ta cần tìm dữ liệu có khoá k. Bởi vì cây tán loe là cây tìm kiếm nhị phân, nên ta sử dụng thuật toán tìm kiếm trên cây tìm kiếm nhị phân. Giả sử tìm kiếm thành công và đỉnh chứa dữ liệu với khoá k là đỉnh v. Ta thực hiện làm loe cây tại đỉnh v. Nếu tìm kiếm không thành công và quá trình tìm kiếm dừng lại tại đỉnh u thì ta làm loe cây tại đỉnh u. Trong trường hợp này đỉnh u là đỉnh có khoá lớn nhất (hoặc nhỏ nhất) nhỏ hơn k (lớn hơn k). Bởi vì các phép quay trong quá trình làm loe

cây được tiến hành trên đường đi từ đỉnh v (hoặc đỉnh u) lên gốc, đường đi này là ngược lại của đường đi trong quá trình tìm kiếm, và do đó thời gian của toàn bộ phép tìm kiếm trên cây tán loe là tỉ lệ với thời gian làm loe cây.

Các phép toán hỏi khác: tìm dữ liệu có khoá nhỏ nhất (phép toán Min), tìm dữ liệu có khoá lớn nhất (phép toán Max) … được cài đặt hoàn toàn tương tự.

Phép toán xen. Giả sử ta cần xen vào cây tán loe một đỉnh mới chứa dữ liệu d. Áp dụng thuật toán xen trên cây tìm kiếm nhị phân để xen đỉnh v chứa dữ liệu d vào cây, đỉnh v trở thành lá của cây, rồi làm loe cây tại đỉnh v. Như vậy đỉnh v chứa dữ liệu d trở thành gốc của cây. Trong quá trình tìm kiếm nếu ta phát hiện ra dữ liệu d đã có sẵn trong cây thì ta làm loe cây tại đỉnh chứa dữ liệu d.

Phép toán loại. Giả sử ta cần loại khỏi cây đỉnh chứa dữ liệu có khoá k. Đầu tiên cần tìm đỉnh chứa dữ liệu có khoá k. Giả sử đó là đỉnh v. Làm loe cây tại v để chuyển v lên gốc cây, ta nhận được cây như trong hình 11.17a. Tìm đỉnh ngoài cùng bên phải của cây con trái T_L của đỉnh v. Giả sử đó là đỉnh v'. Làm loe cây con T_L tại đỉnh v'. Chú ý rằng, vì v' là đỉnh ngoài cùng bên phải của cây con T_L, nên v' không có con phải, và trong quá trình làm loe cây T_L tại v' chỉ có các mẫu zig-zig hoặc mẫu zig được sử dụng, do đó khi v' được đưa lên thành gốc của cây con trái của đỉnh v, thì đỉnh v' vẫn không có con phải. Như vậy sau khi làm loe cây con trái T_L tại đỉnh v', cây trở thành cây như trong hình 11.17b. Gắn cây con phải T_R của đỉnh v thành cây con phải của đỉnh v', loại bỏ đỉnh v ta nhận được cây kết quả, như trong hình 11.17c.



Hình 11.17. Các giai đoạn của phép toán loại

Như vậy khi thực hiện phép loại, ta cần thực hiện hai lần làm loe cây. Đầu tiên là làm loe cây tại đỉnh v cần loại. Khi v trở thành gốc cây thì làm loe cây con trái của v tại đỉnh ngoài cùng bên phải của cây con trái đó.

Không mấy khó khăn thấy rằng, thời gian thực hiện phép xen hoặc phép loại trên cây tán loe là tỷ lệ với thời gian làm loe cây (chỉ cần một hoặc hai lân làm loe cây).

11.6.2 Phân tích trả góp

Trong mục này chúng ta sẽ áp dụng kỹ thuật phân tích trả góp để đánh giá thời gian trong trường hợp xấu nhất của một dãy phép toán trên cây tán loe. Chúng ta sẽ chỉ ra rằng, một dãy m phép toán được thực hiện bắt đầu từ cây rỗng và làm cho cây phát triển tới cây n đỉnh đòi hỏi thời gian O(mlogn), và do đó thời gian trung bình của mỗi phép toán trên cây tán loe là O(logn), mặc dầu rằng thời gian thực hiện một phép toán nào đó có thể là O(n).

Chúng ta ký hiệu W(v) là trọng số của đỉnh v, đó là số đỉnh của cây con gốc v. Giả sử T là cây tán loe, hàm tiềm năng được xác định là

$$\phi(T) = \sum_{v \in T} \log W(v)$$

Ta đặt R(v) = logW(v), số R(v) được gọi là hạng của đỉnh v. Như vậy, tiềm năng của cây T là tổng các hạng của tất cả các đỉnh trong cây T.

$$\phi(T) = \sum_{v \in T} R(v) \qquad (1)$$

Sau đây chúng ta đánh giá thời gian trả góp của các mẫu biến đổi cây (mẫu zig, zig-zig và zig-zag).

Giả sử c là thời gian chạy thực tế của một mẫu biến đổi. Với mẫu zig, chỉ thực hiện 1 phép quay cây, nên c = 1; với các mẫu zig – zig, zig – zag thì c = 2, vì phải thực hiện 2 phép quay. Thời gian chạy trả góp của một mẫu biến đổi $\hat{c}$ được xác định là

$$\hat{c} = c + \Delta\phi(T) \qquad (2)$$

Định lý 11.3. Giả sử R(v), R'(v) là hạng của đỉnh v trước, sau khi thực hiện một mẫu biến đổi. Khi đó thời gian chạy trả góp của một mẫu biến đổi được đánh giá là:

1. Đối với mẫu zig
$$\hat{c} < 1 + R'(v) - R(v)$$
2. Đối với mẫu zig – zig
$$\hat{c} < 3(R'(v) - R(v))$$
3. Đối với mẫu zig – zag

$$\hat{c} < 2\,(R'(v) - R(v))$$

Chúng ta lần lượt xét từng mẫu biến đổi.

1. **Mẫu zig**. Khi thực hiện mẫu zig, chỉ có 2 đỉnh v và p là thay đổi hạng,do đó từ (2) ta có :

$$\hat{c} = 1 + R'(p) - R(p) + R'(v) - R(v)$$

Nhưng từ các cây ở vế trái và phải của hình 11.15a, ta có $R'(p) - R(p) \leq 0$ và $R'(v) - R(v) \geq 0$. Vậy

$$\hat{c} < 1 + R'(v) - R(v)$$

2.Mẫu zig – zig. Nếu thực hiện mẫu zig – zig thì chỉ làm thay đổi hạng của 3 đỉnh v, p, g. Do đó

$$\hat{c} = 2 + R'(g) - R(g) + R'(p) - R(p) + R'(v) - R(v)$$

Từ các cây trước và sau khi thực hiện mẫu zig – zig ở hình 11.15b, ta thấy rằng: $R'(v) = R(g)$, $R(p) > R(v)$ và $R'(v) > R'(p)$. Do đó

$$\hat{c} = 2 + R'(g) + R'(p) - R(p) - R(v)$$

$$\hat{c} < 2 + R'(g) + R'(v) - 2R(v) \qquad (3)$$

Chú ý rằng, trọng số của v trước khi thực hiện mẫu zig – zig là $W(v) = 1 + |T_1| + |T_2|$, trong đó $|T|$ là số đỉnh của cây T, còn trọng số của g sau khi thực hiện mẫu zig – zig là $W'(g) = 1 + |T_3| + |T_4|$. Do đó

$$W'(g) + W(v) < W'(v) \qquad (4)$$

Sử dụng sự kiện rằng, với các số nguyên dương x, y, z nếu $x + y < z$ thì $\log x + \log y < 2\log z - 2$, từ bất đẳng thức (4) ta có

$$R'(g) + R(v) < 2R'(v) - 2$$

Từ bất đẳng thức này và từ (3) , ta có

$$\hat{c} < 2 + R'(g) + R'(v) - 2R(v) =$$
$$2 + R'(g) + R(v) + R'(v) - 3R(v) <$$
$$3(R'(v) - R(v))$$

3.**Mẫu zig – zag**. Mẫu biến đổi này chỉ làm cho 3 đỉnh v, p, g thaydổi hạng. Do đó

$$\hat{c} = 2 + R'(g) - R(g) + R'(p) - R(p) + R'(v) - R(v)$$
$$= 2 + R'(g) + R'(p) - R(p) - R(v)$$

(vì $R(g) = R'(v)$).

Nhưng $R(p) > R(v)$, nên

$$\hat{c} < 2 + R'(g) + R'(p) - 2R(v) \qquad (5)$$

Ta có

$$W'(g) + W'(p) < W'(v)$$

Do đó

$$R'(g) + R'(p) < 2R'(v) - 2$$

Từ đánh giá này và (5), ta suy ra

$$\hat{c} < 2\,(R'(v) - R(v))$$

Định lý được chứng minh.

Nhớ lại rằng, ta gọi phép làm loe cây tại đỉnh v là một dãy các mẫu biến đổi để đưa đỉnh v lên gốc cây. Bây giờ chúng ta đánh giá thời gian trả góp của một phép làm loe cây. Giả sử chúng ta sử dụng k mẫu biến đổi để đưa đỉnh v lên gốc cây và thời gian trả góp của mẫu biến đổi thứ i là $\hat{c}_i$ hạng của đỉnh v sau mẫu biến đổi thứ i là $R^i(v)$, i = 1, 2, …, k. Khi đó từ định lý 11.3, ta có

$$\sum_{i=1}^{k} \hat{c}_i < 3(R'(v) - R(v)) + 3(R''(v) - R'(v)) + ... + 3(R^k(v) - R^{k-1}(v)) + 1$$

$$= 3(R^k(v) \_ R(v)) + 1$$

Lưu ý rằng, sau khi thực hiện mẫu biến đổi thứ k thì đỉnh v là gốc cây, do đó $R^k(v) = \log n$, trong đó n là số đỉnh của cây. Vậy

$$\sum_{i=1}^{k} \hat{c}_i < 3(\log n - R(v)) + 1$$

Từ đó ta suy ra rằng, thời gian trả góp của một phép làm loe cây là O(logn).

Mỗi phép toán trên cây tán loe kèm theo 1 hoặc 2 phép làm loe cây (với phép xen ta cần 2 lần làm loe cây). Thời gian thực hiện mỗi phép toán là tỷ lệ với thời gian thực hiện phép làm loe cây tương ứng. Giả sử ta thực hiện một dãy m phép toán trên cây tán loe, xuất phát từ cây rỗng. Khi đó ta cần thực hiện một dãy phép làm loe cây tương ứng. Ban đầu, cây T_o rỗng, nên $\phi(T_o) = 0$, sau phép làm loe cây thứ i, ta có cây T_i và $\phi(T_i) \geq 0$. Do đó thời gian chạy thực tế của một dãy phép làm loe cây tương ứng với dãy phép toán bị chặn trên bởi thời gian trả góp của dãy phép làm loe cây đó. Ở trên ta đã chứng minh thời gian trả góp của một phép làm loe cây là O(logn). Vậy thời gian thực hiện một dãy phép làm loe cây tương ứng với một dãy m phép toán là O(mlogn). Điều đó cũng có nghĩa là thời gian thực hiện một dãy m phép toán trên cây tán loe xuất phát từ cây rỗng là O(mlogn).

CHƯƠNG 12

HÀNG ƯU TIÊN VỚI PHÉP TOÁN HỢP NHẤT

Trong chương này chúng ta mở rộng KDLTT hàng ưu tiên bằng cách thêm vào hai phép toán: phép toán hợp nhất (Merg) và phép toán giảm khoá (Decreasekey). Các phép toán này là rất cần thiết trong thiết kế thuật toán cho các bài toán tối ưu, chẳng hạn các thuật toán đồ thị như tìm đường đi ngắn nhất (thuật toán Dijkstra), tìm cây bao trùm ngắn nhất (thuật toán Prim). Đầu tiên chúng ta sẽ cài đặt KDLTT hàng ưu tiên với phép toán hợp nhất bởi cây thứ tự bộ phận (binary heap), sau đó chúng ta sẽ xây dựng một CTDL tự điều chỉnh, đó là cây nghiêng (skew heap), để cài đặt hàng ưu tiên với phép toán hợp nhất và sử dụng kỹ thuật phân tích trả góp để đánh giá thời gian chạy của các phép toán hàng ưu tiên trên CTDL này.

12.1 HÀNG ƯU TIÊN VỚI PHÉP TOÁN HỢP NHẤT

Từ đây về sau chúng ta sẽ xem giá trị khoá của một phần tử là giá trị ưu tiên của phần tử đó. KDLTT hàng ưu tiên với phép toán hợp nhất là một họ các hàng ưu tiên với các phép toán sau:

- Các phép toán trên mỗi hàng ưu tiên (xem 10.1): xen một phần tử vào hàng ưu tiên (Insert), tìm phần tử có khoá nhỏ nhất (FindMin), loại khỏi hàng ưu tiên phần tử có khoá nhỏ nhất (DeleteMin).
- Phép toán hợp nhất Merg (P_1, P_2). Hợp nhất hai hàng ưu tiên P_1 và P_2 thành một hàng ưu tiên và trả về hàng ưu tiên này, các hàng ưu tiên P_1 và P_2 bị huỷ bỏ.
- Phép toán giảm khoá Decreasekey (P, x, k). Thay đổi giá trị khoá của phần tử x trong hàng ưu tiên P bởi k, ở đây k là giá trị khoá nhỏ hơn giá trị khoá hiện thời của x.

Cần lưu ý rằng, trong phép toán giảm khoá, vị trí của phần tử x trong hàng ưu tiên P được xem là đã biết, không cần phải thực hiện thao tác tìm. Trong các ứng dụng, phép toán giảm khoá thường được sử dụng trong vòng lặp sau: loại phần tử có khoá nhỏ nhất khỏi hàng ưu tiên P, rồi xem xét từng phần tử còn lại và tiến hành giảm khoá (nếu cần thiết); lặp lại quá trình đó cho tới khi hàng ưu tiên P trở thành rỗng. Phép toán giảm khoá là đặc biệt hữu ích trong việc thiết kế các thuật toán cho các vấn đề tối ưu. Chúng ta sẽ

đưa ra các ví dụ ứng dụng hàng ưu tiên với phép toán hợp nhất trong chương nói về các thuật toán đồ thị.

Người ta đã nghiên cứu và đề xuất nhiều CTDL khác nhau để cài đặt hàng ưu tiên với phép toán hợp nhất, chẳng hạn như binary heap, leftist heap, binomial heap, fibonacci heap, skew heap. Tất cả các CTDL này đều có đặc điểm chung, đó là cấu trúc cây thoả mãn tính chất heap (khoá của mỗi đỉnh không lớn hơn khoá của các đỉnh con của nó).

12.2 CÁC PHÉP TOÁN HỢP NHẤT VÀ GIẢM KHOÁ TRÊN CÂY THỨ TỰ BỘ PHẬN

Trong mục 10.3 chúng ta đã cài đặt hàng ưu tiên bởi cây thứ tự bộ phận (binary heap). Nhớ lại rằng với cách cài đặt đó, phép toán FindMin chỉ cần thời gian O(1), vì phần tử có khoá nhỏ nhất nằm ở gốc cây; các phép toán Insert và DeleteMin chỉ đòi hỏi thời gian O(logn), bởi vì để thực hiện các phép toán này chúng ta chỉ cần đi từ lá lên gốc (đối với Insert) hoặc đi từ gốc xuống lá (đối với DeleteMin) và tiến hành hoán vị các dữ liệu chứa trong các đỉnh của cây, mà độ cao của cây thứ tự bộ phận n đỉnh là O(logn). Bây giờ chúng ta xét xem các phép toán hợp nhất và giảm khoá được thực hiện như thế nào trên cây thứ tự bộ phận.

Phép toán hợp nhất Merg(P_1, P_2). Ở đây chúng ta cần phải kết hợp hai cây thứ tự bộ phận P_1 và P_2 thành một cây thứ tự bộ phận. Cách tốt nhất chúng ta có thể làm là xen từng đỉnh của cây P_2 vào cây P_1. Giả sử cây P_1 có n_1 đỉnh, cây P_2 có n_2 đỉnh. Chúng ta cần sử dụng n_2 phép toán Insert, mỗi phép toán này cần thời gian logarit theo số đỉnh trong cây P_1. Do đó, phép toán Merg(P_1, P_2) đòi hỏi thời gian $n_2\log(n_1 + n_2)$.

Phép toán giảm khoá Decreasekey (P, x, k). Trên cây thứ tự bộ phận, phép toán giảm khoá được tiến hành rất thuận tiện. Đi từ đỉnh chứa x lên gốc (giống như khi ta thực hiện phép toán Insert) nếu khoá k nhỏ hơn khoá của dữ liệu trong đỉnh cha thì ta hoán vị dữ liệu x và dữ liệu đó. Như vậy phép toán giảm khoá trên cây thứ tự bộ phận chỉ cần thời gian O(logn).

Tóm lại trên cây thứ tự bộ phận (binary heap), phép toán FindMin chỉ cần thời gian hằng, các phép toán Insert, DeleteMin, Decreasekey chỉ cần thời gian logarit. Riêng phép toán Merg, cây thứ tự bộ phận không cho phép ta thực hiện hiệu quả phép toán này.

12.3 CÂY NGHIÊNG

Trong mục này, chúng ta sẽ nghiên cứu sự cài đặt hàng ưu tiên với phép toán hợp nhất bởi CTDL tự điều chỉnh được gọi là **cây nghiêng** (skew

heap). Cây nghiêng là cây nhị phân thoả mãn tính chất thứ tự bộ phận (hay còn được gọi là tính chất heap), tức là khoá của dữ liệu trong mỗi đỉnh không lớn hơn khoá của dữ liệu trong các đỉnh con của nó. Chú ý rằng, trong cây nghiêng không có điều kiện áp đặt nào nhằm hạn chế độ cao của cây. Tuy nhiên mỗi khi tiến hành một phép toán hàng ưu tiên trên cây nghiêng, ta thực hiện một phép điều chỉnh cây với mục đích để các phép toán thực hiện sau đó sẽ hiệu quả hơn. Kết quả là thời gian thực hiện một phép toán riêng biệt trên cây nghiêng có thể là O(n), nhưng thời gian chạy trả góp của mỗi phép toán hàng ưu tiên trên cây nghiêng chỉ là O(logn).

12.3.1 Các phép toán hàng ưu tiên trên cây nghiêng

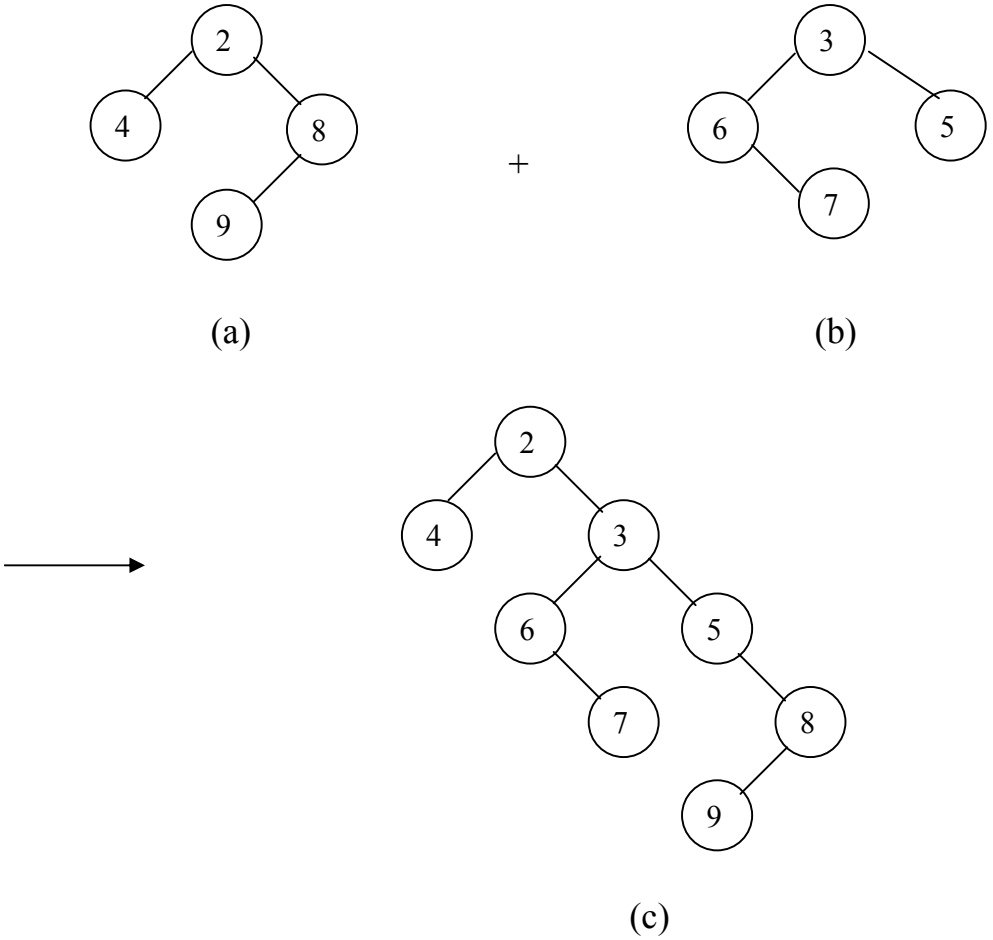
Khi hàng ưu tiên được biểu diễn dưới dạng cây nhị phân thoả mãn tính chất thứ tự bộ phận, chúng ta có thể cài đặt các phép toán khác thông qua phép toán hợp nhất. Trong mô tả các phép toán sau đây, ta ký hiệu S, S_1, S_2 là các cây nghiêng biểu diễn các hàng ưu tiên, x là một phần tử dữ liệu, k là một giá trị khoá, và p là con trỏ liên kết trong cây trỏ tới đỉnh chứa phần tử cần giảm khoá. Các phép toán được thực hiện như sau:

- FindMin(S): Trả về phần tử chứa trong gốc cây S.
- Insert(S, x): Tạo ra cây chỉ có một đỉnh chứa x và hợp nhất cây này với cây S.
- DeleteMin(S): Loại bỏ gốc cây, rồi hợp nhất cây con trái và cây con phải của S.
- Decreasekey(S, p, k): Phép toán này có nghĩa là chúng ta cần giảm khoá của phần tử chứa trong đỉnh p của cây nghiêng S với giá trị khoá mới là k. Giả sử S_1 là cây con của S có gốc là p, phần tử chứa trong gốc cây S_1 bây giờ có khoá là k, S_2 là cây nhận được từ cây S bằng cách cắt bỏ nhánh p. Phép toán giảm khoá được thực hiện bằng cách hợp nhất cây S_1 và S_2.

Như vậy vấn đề còn lại là cài đặt phép toán hợp nhất: hợp nhất hai cây nhị phân thoả mãn tính chất thứ tự bộ phận thành một cây nhị phân cũng thoả mãn tính chất đó. Điều này là không có gì khó khăn.

Trong cây nhị phân, chúng ta gọi đường bên phải là đường đi xuất phát từ gốc cây và luôn luôn đi theo nhánh bên phải. Chẳng hạn, đường bên phải trong cây hình 12.1c là đường 2 – 3 – 5 - 8 . Chú ý rằng, trong cây nghiêng, các dữ liệu chứa trong các đỉnh trên đường đi bất kỳ từ gốc tới lá được sắp xếp theo thứ tự tăng dần theo khoá. Từ đó, ta có thể đưa ra thuật toán đơn giản sau để hợp nhất hai cây nghiêng S_1 và S_2:

Xét các đường bên phải của cây nghiêng S_1 và S_2 như các DSLK được sắp theo thứ tự khoá tăng dần. Hợp nhất hai cây nghiêng S_1 và S_2 được thực hiện bằng cách hợp nhất hai DSLK này thành một DSLK được sắp theo thứ tự khoá tăng dần, trong khi vẫn giữ nguyên các nhánh trái của các đỉnh nằm trên hai DSLK đó. Ví dụ, hợp nhất hai cây nghiêng trong hình 12.1a và 12.1b, chúng ta nhận được cây nghiêng trong hình 12.1c.



Hình 12.1. Phương pháp hợp nhất đơn giản.

Chúng ta cũng có thể thực hiện phương pháp hợp nhất trên bằng thuật toán đệ quy sau.

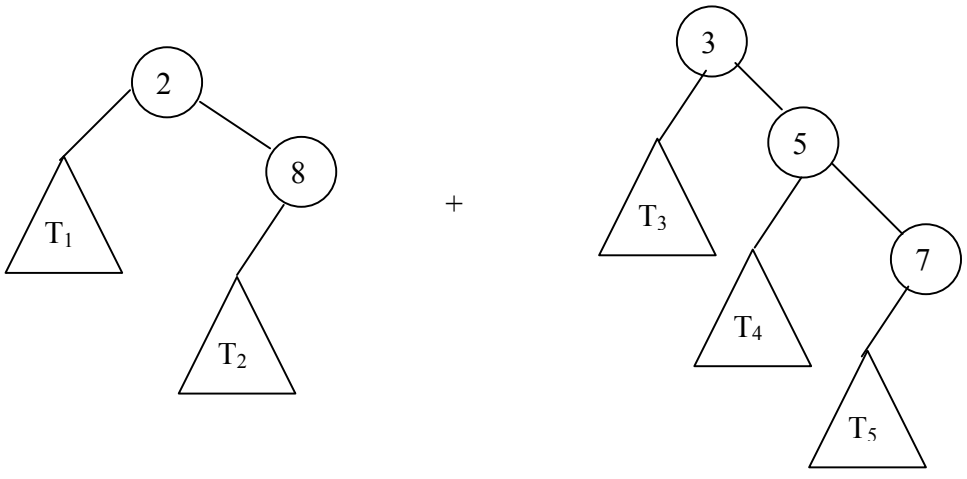
Giả sử khoá của dữ liệu trong gốc của cây S_1 nhỏ hơn khoá của dữ liệu trong gốc cây S_2. (Nếu ngược lại, ta chỉ cần hoán đổi hai cây S_1 và S_2). Khi đó hợp nhất của cây S_1 với cây S_2 là cây S có nhánh trái là nhánh trái

của cây S_1, còn nhánh phải của S là hợp nhất của cây là nhánh phải của cây S_1 với cây S_2 (lời gọi đệ quy).

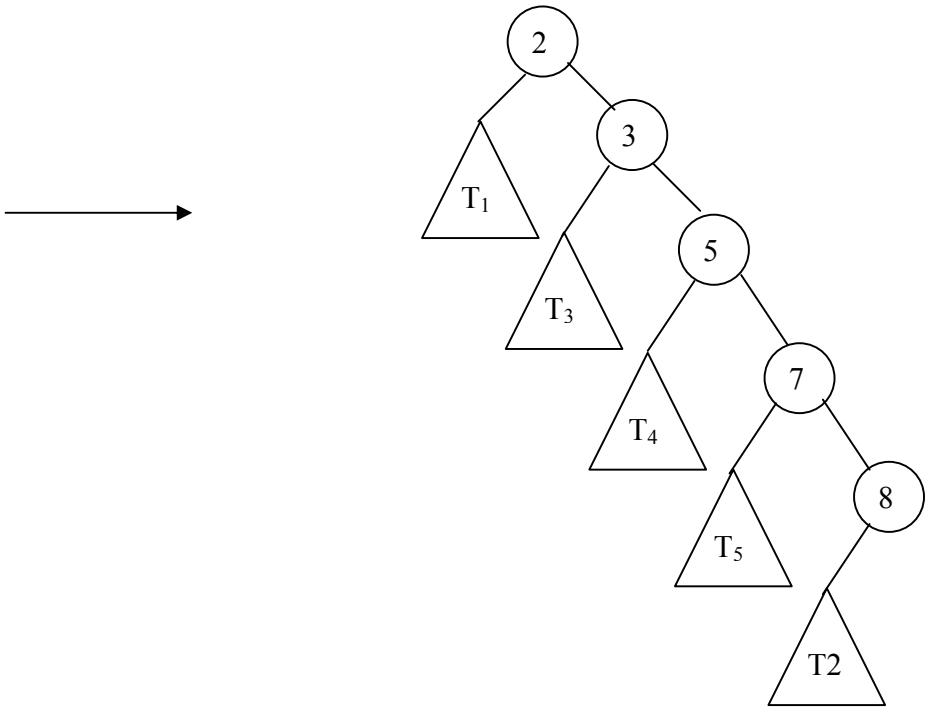
Chúng ta có các nhận xét sau về phương pháp hợp nhất trên. Thời gian thực hiện phép hợp nhất phụ thuộc vào độ dài của các đường bên phải của các cây S_1 và S_2. Nhớ lại rằng, không có điều kiện áp đặt nào nhằm hạn chế độ cao của các cây nghiêng, trong trường hợp xấu nhất các cây nghiêng S_1 và S_2 có thể chỉ gồm đường bên phải. Do đó trong trường hợp xấu nhất, phương pháp hợp nhất trên đòi hỏi thời gian $O(n_1 + n_2)$, trong đó n_1, n_2 là số đỉnh trong cây S_1, S_2 tương ứng. Cây nghiêng kết quả S có đường bên phải dài hơn các đường bên phải của S_1 và S_2, tức là khi thực hiện các phép hợp nhất, đường bên phải của cây càng ngày càng dài ra. Nhằm khắc phục nhược điểm này, mỗi khi thực hiện hợp nhất theo phương pháp trên, ta tiến hành kèm theo một phép biến đổi cây. Như vậy, thuật toán hợp nhất hai cây nghiêng gồm hai bước sau:

- Bước 1. Hợp nhất cây S_1 với S_2 bằng cách hợp nhất đường bên phải của S_1 với đường bên của S_2 để nhận được cây S.
- Bước 2. Trao đổi cây con trái với cây con phải của tất cả các đỉnh nằm trên đường bên phải của cây S nhận được ở bước 1 trừ đỉnh cuối cùng (nó không có nhánh phải).

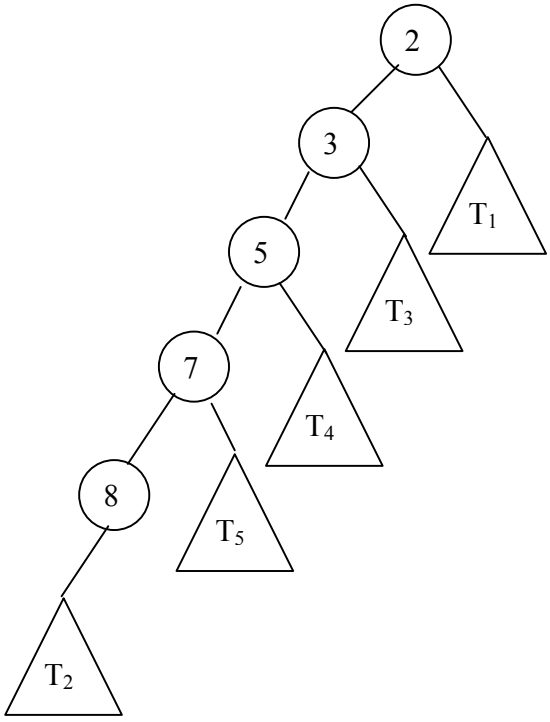
Ví dụ. Giả sử ta cần hợp nhất hai cây nghiêng trong hình 12.2a. Thực hiện bước 1 của thuật toán, ta nhận được cây trong hình 12.2b, thực hiện bước 2 trên cây này, ta nhận được cây kết quả trong hình 12.2c.



(a)



(b)



(c)

Hình 12.2. Hợp nhất hai cây nghiêng.
(a) Hai cây nghiêng cần hợp nhất
(b) Cây nhận được bằng cách thực hiện phương pháp hợp nhất đơn giản: hợp nhất hai đường bên phải.
(c) Cây kết quả sau khi trao đổi hai nhánh trái, phải của các đỉnh trên đường bên phải của cây (b)

Nhận xét. Bước 2 của thuật toán hợp nhất là phép biến đổi cây mang tính heuristic nhằm hạn chế chiều dài đường bên phải của cây kết quả.

Bây giờ chúng ta nói tới sự cài đặt thuật toán hợp nhất hai cây nghiêng gồm hai bước đã trình bày trên. Có thể cài đặt thuật toán này bởi hàm không đệ quy, chúng tôi để lại cách này cho độc giả xem như bài tập. Sau đây chúng ta sẽ cài đặt phép toán hợp nhất bởi hàm đệ quy. Chúng ta giả sử rằng, các đỉnh của cây nghiêng có cấu trúc sau:

```
struct   Node
{
  keyType  key ;
  // Các trường dữ liệu khác.
  Node*  left ; // con trỏ tới đỉnh trái.
  Node*  right ; // con trỏ tới đỉnh phải.
} ;
```

Giả sử root1, root2 là các con trỏ trỏ tới gốc hai cây nghiêng cần hợp nhất. Hàm hợp nhất đệ quy là như sau:

```
Node*  Merg(Node* & root1, Node* & root2)
{
  if ((root1 = = NULL) || ((root2 ! = NULL) &&
      (root2 → key) < (root1 → key)))
       swap (root1, root2) ;
  // Sau lệnh này, nếu một trong hai cây rỗng thì cây rỗng là root2,
  // nếu cả hai cây đều khác rỗng thì root1 → key <= root2→ key
  if (root1! = NULL)
      root1 → right = Merg(root1 → right, root2) ;
  if (root1 → right! = NULL)
      swap (root1 → left, root1 → right) ;
  return   root1 ;
}
```

Ở đầu mục này, chúng ta đã chỉ ra rằng, các phép toán hàng ưu tiên khác Insert, DeleteMin, Decreasekey được thực hiện như thế nào thông qua phép hợp nhất. Từ đó, sử dụng hàm Merg, bạn đọc dễ dàng cài đặt được các hàm cho các phép toán còn lại trên hàng ưu tiên.

Chú ý rằng, bước 2 của thuật toán hợp nhất là bước điều chỉnh cây với mong muốn hạn chế chiều dài của đường bên phải của cây kết quả. Thời gian thực hiện phép hợp nhất trong trường hợp xấu nhất vẫn là O(n), trong đó n là tổng số đỉnh của hai cây cần hợp nhất. Nhưng chúng ta sẽ chỉ ra rằng, nhờ có phép biến đổi cây trong bước 2 mà thời gian chạy trả góp của phép hợp nhất sẽ là O(logn).

12.3.2 Phân tích trả góp

Trong mục này chúng ta sẽ đánh giá thời gian chạy trả góp của phép toán Merg trên cây nghiêng. Trước hết, chúng ta cần đến khái niệm đỉnh nặng, đỉnh nhẹ.

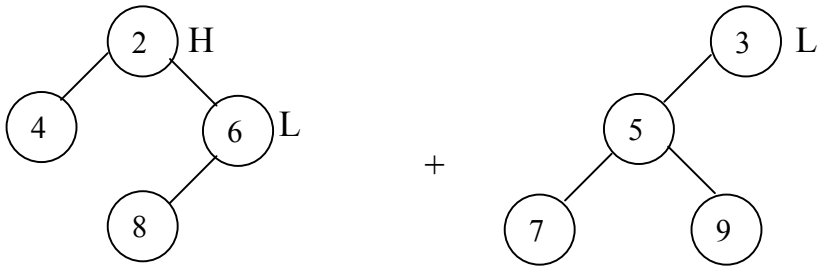
Định nghĩa 12.1. Trong cây nhị phân, một đỉnh được gọi là đỉnh nặng nếu số đỉnh của cây con phải của nó lớn hơn số đỉnh của cây con trái. Một đỉnh không phải là đỉnh nặng sẽ được gọi là đỉnh nhẹ.

Từ định nghĩa trên, chúng ta có các nhận xét sau:

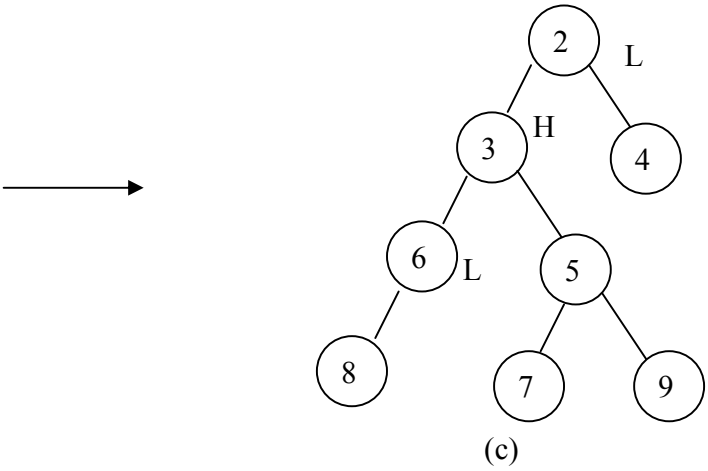
- Các đỉnh lá của cây là đỉnh nhẹ.
- Khi thực hiện hợp nhất hai cây nghiêng S_1 và S_2 thì chỉ có các đỉnh nằm trên đường bên phải của hai cây đó là thay đổi trạng thái nặng, nhẹ. Hơn nữa, các đỉnh nặng nằm trên đường bên phải của hai cây đó sẽ trở thành đỉnh nhẹ trong cây hợp nhất (tại sao?)

Chẳng hạn, khi ta hợp nhất hai cây nghiêng trong hình 12.3a và 12.3b, các đỉnh nằm trên đường bên phải của hai cây đó là 2, 6 và 3, trong đó 2 là đỉnh nặng, còn 6 và 3 là đỉnh nhẹ.

Trong cây hợp nhất 12.3c, đỉnh nặng 2 trở thành đỉnh nhẹ, và đỉnh nhẹ 3 trở thành đỉnh nặng, đỉnh 6 vẫn còn là đỉnh nhẹ.



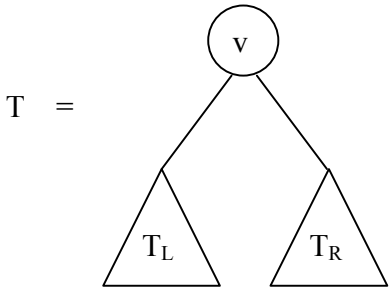
(a) (b)



Hình 12.3. Sự thay đổi trạng thái nặng, nhẹ của các đỉnh khi thực hiện hợp nhất

Định lý 12.1. Số đỉnh nhẹ trên đường bên phải của cây nghiêng n đỉnh nhiều nhất là $\lfloor \log n \rfloor + 1$.

Thật vậy, giả sử cây nghiêng T có gốc là đỉnh v, cây con trái là T_L và cây con phải là T_R.



Giả sử số đỉnh của cây T là n, của cây con trái T_L là n_L, của cây con phải T_R là n_R. Ta có $n = n_L + n_R + 1$. Nếu v là đỉnh nhẹ thì $n_L \geq n_R$. Do đó

$$n \geq 2n_R + 1 > 2n_R$$

hay $n_R < \frac{n}{2}$

Từ đó ta suy ra rằng, số đỉnh nhẹ trên đường bên phải của cây T nhiều nhất là k, trong đó k là số nguyên lớn nhất sao cho

$$\mathbf{1} < \frac{n}{2^k}$$

hay logn > k

Cây con phải cuối cùng có thể chỉ gồm một đỉnh (lá), mà đỉnh lá là đỉnh nhẹ. Vậy số đỉnh nhẹ trên đường bên phải của cây nghiêng nhiều nhất là $\lfloor \log n \rfloor + 1$.

Các nhận xét đã đưa ra và định lý 12.1 sẽ được sử dụng để đánh giá thời gian chạy trả góp của phép hợp nhất.

Bây giờ chúng ta xác định hàm tiềm năng ϕ trên một họ các cây nghiêng $D = \{S_i\}$, trong đó S_i là cây nghiêng. Hàm ϕ được xác định như sau

$$\phi(D) = \sum_i r_i$$

trong đó r_i là số đỉnh nặng trong cây S_i.

Định lý 12.2. Giả sử các cây nghiêng S_1, S_2 có số đỉnh là n_1, n_2 tương ứng. Thời gian chạy trả góp của phép hợp nhất $Merg(S_1, S_2)$ là $O(\log n)$, trong đó $n = n_1 + n_2$.

Giả sử đường bên phải của cây S_1 chứa l_1 đỉnh nhẹ và h_1 đỉnh nặng, còn đường bên phải của cây S_2 chứa l_2 đỉnh nhẹ và h_2 đỉnh nặng. Như vậy, số đỉnh trên đường bên phải của cây S_1 là $l_1 + h_1$, của cây S_2 là $l_2 + h_2$. Do đó, thời gian thực tế để thực hiện phép hợp nhất là

$c = l_1 + l_2 + h_1 + h_2$.

Chúng ta đánh giá sự thay đổi tiềm năng khi thực hiện phép hợp nhất. Như nhận xét đã đưa ra, khi thực hiện phép hợp nhất, chỉ có các đỉnh nằm trên đường bên phải của hai cây là thay đổi trạng thái nặng, nhẹ. Tất cả các đỉnh nặng trên đường bên phải của hai cây S_1 và S_2 trở thành đỉnh nhẹ. Mặt khác khả năng nhiều nhất là tất cả các đỉnh nhẹ trên đường bên phải của hai cây đó trở thành đỉnh nặng. Do đó sự thay đổi tiềm năng nhiều nhất là $l_1 + l_2 - h_1 - h_2$.

Từ các kết luận trên, ta có thể đánh giá thời gian trả góp của phép hợp nhất như sau:

$$\begin{aligned} \hat{c} &= c + (\text{thay đổi tiềm năng}) \\ &= l_1 + l_2 + h_1 + h_2 + (l_1 + l_2 - h_1 - h_2) \\ &= 2(l_1 + l_2) \end{aligned}$$

Nhưng theo định lý 12.1, ta có;

$l_1 \le \log n_1 + 1$ và $l_2 \le \log n_2 + 1$

Vậy

$$\begin{aligned} \hat{c} &\le 2(\log n_1 + \log n_2 + 2) \\ &\le 4 \log n \end{aligned}$$

trong đó $n = n_1 + n_2$. (Bất đẳng thức sau cùng được suy ra từ bổ đề sau: Nếu $a + b \leq c$, trong đó a, b, c là các số thực dương, thì $\log a + \log b \leq 2\log c - 2$) Bất đẳng thức cuối cùng đã hoàn thành chứng minh định lý.

Từ định lý 12.2 và thủ tục thực hiện các phép toán Insert, DeleteMin, Decreasekey bằng cách sử dụng phép hợp nhất, chúng ta suy ra kết luận sau: Thời gian trả góp của các phép toán Merg, Insert, DeleteMin, Decreasekey trên cây nghiêng n đỉnh là O(logn), còn thời gian của phép toán FindMin là O(1).

Chúng ta còn phải chứng tỏ rằng, thời gian trả góp của một dãy phép toán hàng ưu tiên trên các cây nghiêng là cận trên của thời gian chạy thực tế của dãy phép toán đó. Chúng ta thực hiện dãy phép toán xuất phát từ một họ các hàng ưu tiên chỉ có một phần tử. Như vậy trạng thái ban đầu D_o là một họ các cây nghiêng chỉ có một đỉnh gốc. Theo định nghĩa hàm tiềm năng, ta có $\phi(D_o) = 0$. Giả sử D_i là họ các cây nghiêng nhận được sau khi ta thực hiện phép toán thứ i trong dãy. Từ định nghĩa hàm tiềm năng, ta có $\phi(D_i) \geq 0$ với mọi i. Vận dụng các kết luận đã đưa ra ở cuối mục 11.4, chúng ta suy ra khẳng định cần chứng minh.

CHƯƠNG 14

CÁC CẤU TRÚC DỮ LIỆU ĐA CHIỀU

Từ trước tới nay chúng ta mới chỉ nghiên cứu các CTDL để biểu diễn tập dữ liệu, trong đó dữ liệu được hoàn toàn xác định bởi một thuộc tính được gọi là khoá của dữ liệu, và khoá của dữ liệu được sử dụng trong các phép toán tìm kiếm, xen, loại. Chúng ta sẽ nói đến các dữ liệu được xác định chỉ bởi một thuộc tính khoá như là các dữ liệu một chiều. Tuy nhiên trong rất nhiều lĩnh vực áp dụng, chẳng hạn như đồ hoạ máy tính, xử lý ảnh, các hệ thông tin địa lý, các hệ cơ sở dữ liệu đa phương tiện, các áp dụng của hình học tính toán …, chúng ta cần phải làm việc với các dữ liệu không phải một chiều. Đó là các dữ liệu hình ảnh (image data), dữ liệu video, dữ liệu audio, dữ liệu văn bản (document data), dữ liệu viết tay (handwritten data)… Các dữ liệu này có thể được biểu diễn bởi, chẳng hạn, các thuộc tính không gian, thời gian. Nói chung, các dữ liệu này được biểu diễn bởi vectơ các giá trị thuộc tính (x_1,…, x_k), tức là mỗi dữ liệu được mô tả bởi một điểm trong không gian k - chiều. Các dữ liệu này được gọi là các **dữ liệu điểm k chiều (k – dimensional point data)**. Trong chương này, chúng ta sẽ nghiên cứu các CTDL để biểu diễn tập dữ liệu điểm k - chiều. Các CTDL này được gọi là các **CTDL đa chiều (multidimensional data structures**) hay còn được gọi là các **CTDL không gian (spatial data structures**). Kỹ thuật chung được sử dụng là biểu diễn tập dữ liệu điểm k - chiều bởi các loại cây khác nhau. Cây biểu diễn sự phân hoạch không gian thành các miền con. Gốc cây biểu diễn toàn bộ miền chứa dữ liệu. Mỗi đỉnh của cây biểu diễn môt miền nào đó, các con của nó biểu diễn sự phân hoạch miền này thành các miền con. Nhiều CTDL đã được đề xuất thực hiện ý tưởng phân hoạch miền chứa dữ liệu thành các miền con sao cho việc tìm kiếm dữ liệu và cập nhật tập dữ liệu được thực hiện hiệu quả. Chúng ta sẽ lần lượt nghiên cứu các CTDL: **cây k - chiều (k – dimensional tree), cây tứ phân (quadtree), cây tứ phân MX (MX – Quadtree**). Các loại cây này khác nhau ở chiến lược phân hoạch một miền thành các miền con.

14.1 CÁC PHÉP TOÁN TRÊN CÁC DỮ LIỆU ĐA CHIỀU

Dữ liệu điểm k - chiều là dữ liệu được hoàn toàn xác định bởi vectơ các giá trị thuộc tính (x_1, …, x_k) trong không gian k - chiều, hay nói cách khác dữ liệu được hoàn toàn xác định bởi k giá trị khoá x_1, …, x_k. Trong các ứng dụng, khi có một tập dữ liệu điểm k - chiều, trong quá trình xử lý dữ

liệu chúng ta thường xuyên phải sử dụng **các phép toán từ điển: tìm kiếm, xen, loại,** giống như đối với các dữ liệu một chiều (một khoá). Chỉ có điều khác là ở đây chúng ta cần tiến hành tìm kiếm, xen, loại dựa vào k giá trị khoá đã cho. Chẳng hạn, phép tìm kiếm ở đây có nghĩa là: tìm trong tập dữ liệu điểm k - chiều đã cho một dữ liệu khi biết vectơ các giá trị khoá của nó là $(x_1, \ldots, x_k)$.

Ngoài các phép toán từ điển, trên các dữ liệu đa chiều, chúng ta còn cần đến một phép toán đặc biệt: **tìm kiếm phạm vi (Range Search**). Phép toán tìm kiếm phạm vi được phát biểu như sau: cho trước một điểm dữ liệu $(x_1, \ldots, x_k)$ và một số thực dương r, cần tìm trong tập dữ liệu điểm k - chiều đã cho tất cả các điểm dữ liệu cách $(x_1, \ldots, x_k)$ một khoảng cách không lớn hơn r. Trong không gian 2 - chiều, phép toán tìm kiếm phạm vi, nói theo ngôn ngữ hình học có nghĩa là, cho trước hình tròn tâm (x, y) bán kính r, cần tìm tất cả các điểm dữ liệu nằm trong hình tròn này. Sau đây chúng ta đưa ra một ví dụ minh hoạ tầm quan trọng của phép toán tìm kiếm phạm vi.

Giả sử D là một tập các văn bản. Trước hết chúng ta cần đưa ra một cách biểu diễn văn bản. Giả sử T là danh sách các từ "quan trọng" xuất hiện trong các văn bản của tập D, $T = (t_1, \ldots, t_k)$ .Với mỗi văn bản d thuộc D, ta biểu diễn d bởi vectơ các số thực không âm, $d = (x_1, \ldots, x_k)$ trong đó x_i (i = 1, …, k) là tỷ số giữa số lần xuất hiện từ t_i trong văn bản d trên số từ trong văn bản d. Chúng ta cần xác định độ đo "sự liên quan" hay độ đo "sự tương tự", hay còn gọi là "khoảng cách" giữa hai văn bản. Có nhiều cách xác định độ đo đó, chúng ta không nêu ra ở đây. Trong các hệ tìm kiếm thông tin, người sử dụng mong muốn tìm ra một danh sách p văn bản "liên quan" nhiều nhất tới một câu hỏi Q từ hệ cơ sở dữ liệu văn bản D. Chúng ta quan niệm câu hỏi Q như một văn bản và biểu diễn nó bởi một vectơ $Q = (q_1, \ldots, q_k)$ theo cách biểu diễn đã nêu. Vấn đề tìm câu trả lời cho câu hỏi Q bây giờ được quy về tìm p láng giềng gần nhất với Q (theo độ đo tương tự đã xác định), tức là được quy về thực hiện phép toán tìm kiếm phạm vi.

14.2 CÂY K - CHIỀU

Cây 2 - chiều được sử dụng để lưu giữ các dữ liệu điểm 2 - chiều, cây 3 - chiều để lưu giữ các dữ liệu điểm 3 - chiều, … Tổng quát, cây k - chiều để biểu diễn tập dữ liệu điểm k - chiều. Trong mục này, trước hết chúng ta sẽ trình bày CTDL cây 2 - chiều và chỉ ra các phép toán từ điển và phép toán tìm kiếm phạm vị được thực hiện như thế nào trên cây 2 - chiều. Sau đó chúng ta sẽ trình bày cách tổng quát hoá cây 2 - chiều để có cây k - chiều.

14.2.1 Cây 2 - chiều

Cây 2 - chiều là cây nhị phân. Mỗi dữ liệu điểm 2 - chiều gồm các giá trị toạ độ của điểm và các thông tin khác gắn với điểm này mà ta quan tâm. Vì vậy, mỗi đỉnh của cây 2 - chiều là một cấu trúc có dạng sau:

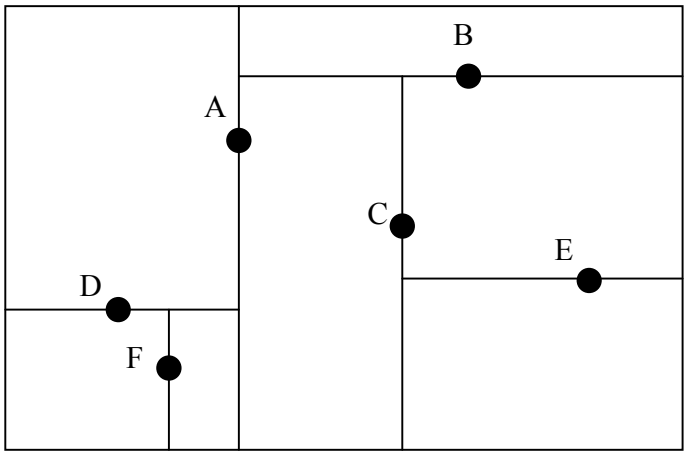
```
struct   Node
{
    infoType  info ;
    double  Xval ;
    double  Yval ;
    Node*  left ;
    Node*  right ;
} ;
```

Trong đó, các trường Xval và Yval ký hiệu hoành độ và tung độ của điểm, các con trỏ left và right trỏ tới đỉnh con trái và phải, còn trường info lưu các thông tin khác về điểm. Nội dung của trường info được xác định cụ thể tuỳ từng ứng dụng. Chẳng hạn trong các hệ thông tin địa lý (geographic information system), các dữ liệu điểm được lưu giữ có thể là các điểm biểu diễn các vị trí mà chúng ta quan tâm trên bản đồ một vùng lãnh thổ nào đó. Nếu các vị trí là các thành phố thì thông tin về vị trí có thể là tên thành phố, số dân của thành phố, …

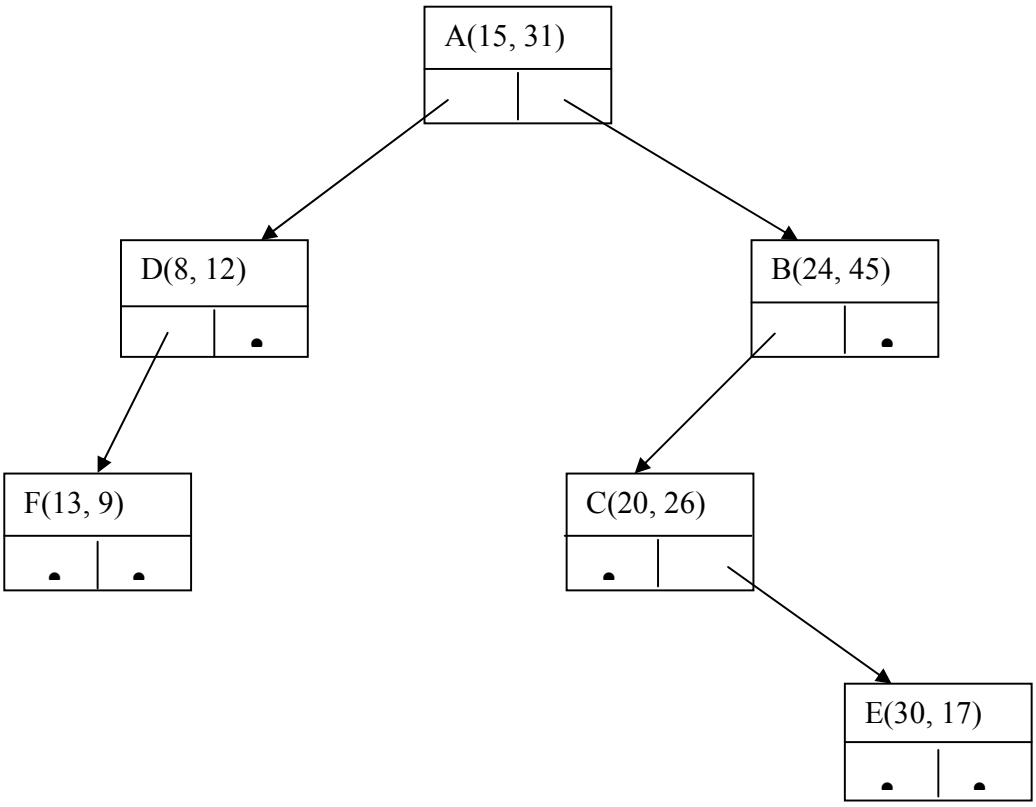
Cây 2 - chiều là sự tổng quát hoá của cây tìm kiếm nhị phân. Mỗi đỉnh trong của cây biểu diễn một miền và phân hoạch miền dữ liệu này theo một chiều (theo một toạ độ). Các đỉnh trên cùng một mức sẽ phân hoạch các miền dữ liệu tương ứng theo đường thẳng đứng (theo hoành độ) hoặc theo đường nằm ngang (theo tung độ). Nếu các đỉnh trên một mức phân hoạch dữ liệu theo đường thẳng đứng, thì ở mức tiếp theo các đỉnh sẽ phân hoạch dữ liệu theo đường nằm ngang, các đỉnh ở mức tiếp theo nữa lại phân hoạch dữ liệu theo đường thẳng đứng, và cứ thế tiếp tục. Chẳng hạn, giả sử chúng ta có tập các điểm như sau:

Điểm	Toạ độ
A	(15, 31)
B	(24, 45)
C	(20, 26)
D	(8, 12)
E	(30, 17)
F	(13, 9)

Giả sử ta chọn điểm A đại biểu cho toàn bộ miền chứa các điểm, khi đó A là gốc cây và phân hoạch miền thành hai miền con theo đường thẳng đứng. Chọn điểm B đại biểu cho miền bên phải đường thẳng đứng đi qua A, chọn điểm D đại biểu cho miền bên trái đường thẳng đứng đi qua A, khi đó B là đỉnh con phải, còn D là đỉnh con trái của A, đồng thời đỉnh B và D phân hoạch các miền con tương ứng theo đường nằm ngang. Tiếp theo chọn C đại biểu cho miền con nằm dưới đường nằm ngang đi qua B, ta có C sẽ là đỉnh con trái của B và đỉnh C phân hoạch miền con đó theo đường thẳng đứng … Cuối cùng, chúng ta nhận được sự phân hoạch miền chứa các điểm thành các miền con như trong hình 14.1a và cây biểu diễn tập điểm tương ứng với phân hoạch này được cho trong hình 14.1b.



(a)



(b)

Hình 14.1. (a) Một cách phân hoạch tập điểm
(b) Cây 2- chiều tương ứng

Cần lưu ý rằng, có nhiều cách phân hoạch miền chứa dữ liệu và do đó chúng ta có thể biểu diễn tập dữ liệu điểm 2 - chiều bởi các cây 2 chiều khác nhau.

Mức của các đỉnh trong cây được xác định như sau. Gốc ở mức 0, nếu một đỉnh ở mức l, thì các con (nếu có) của nó ở mức l + 1. Bây giờ chúng ta có thể đưa ra định nghĩa cây 2-chiều.

Cây 2-chiều là cây nhị phân thoả mãn các điều kiện sau:

1. Nếu P là đỉnh ở mức chẵn và Q là đỉnh bất kỳ thuộc cây con trái của đỉnh P thì Q → Xval < P → Xval, còn nếu Q là đỉnh bất kỳ thuộc cây con phải của đỉnh P thì Q → Xval ≥ P → Xval.

2. Nếu P là đỉnh ở mức lẻ và Q là đỉnh bất kỳ thuộc cây con trái của P thì Q → Yval < P → Yval, và nếu Q là đỉnh bất kỳ thuộc cây con phải của P thì Q → Yval ≥ P → Yval.

Chú ý rằng, trong định nghĩa trên, các điều kiện đã nêu có nghĩa là các đỉnh ở mức chẵn (lẻ) sẽ phân hoạch miền mà chúng đại diện thành hai miền con theo đường thẳng đứng (theo đường nằm ngang).

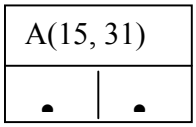
Sau đây chúng ta sẽ xét các phép toán từ điển: tìm kiếm, xen, loại và phép toán tìm kiếm phạm vi trên cây 2-chiều.

Phép toán tìm kiếm. Giả sử T là con trỏ trỏ tới gốc cây 2-chiều, chúng ta cần tìm xem điểm (x, y) có được lưu trong một đỉnh của cây T hay không. Thuật toán tìm kiếm trên cây 2-chiều cũng tương tự như thuật toán tìm kiếm trên cây tìm kiếm nhị phân. Chúng ta cho con trỏ P chạy trên các đỉnh của cây T, ban đầu P trỏ tới gốc cây. Nếu P ≠ NULL, ta kiểm tra xem (P → Xval, P → Yval) có trùng với (x, y) không. Nếu (P → Xval, P →Yval) = (x, y) thì sự tìm kiếm đã thành công và dừng lại. Giả sử chúng khác nhau và P ở mức chẵn, khi đó nếu P → Xval > x thì ta cho con trỏ P trỏ tới đỉnh con trái của nó, còn nếu P → Xval ≤ x thì cho P trỏ tới đỉnh con phải của nó. Tương tự, nếu P ở mức lẻ và P → Yval > y, thì cho P trỏ tới đỉnh con trái của nó còn nếu P → Yval ≤ y thì cho P trỏ tới đỉnh con phải của nó. Quá trình trên được lặp lại. Nếu tới một thời điểm nào đó P = NULL thì có nghĩa là sự tìm kiếm thất bại và dừng lại.

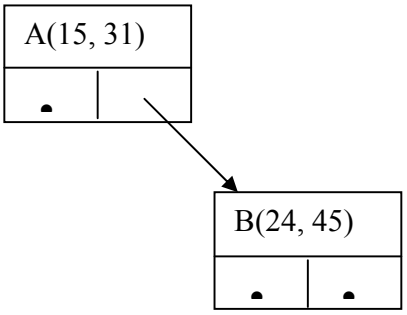
Phép toán xen. Giả sử chúng ta cần xen vào cây 2-chiều T một đỉnh mới chứa điểm (x, y). Nếu T là cây rỗng, ta tạo ra một đỉnh chứa điểm (x, y), các trường left và right chứa hằng NULL, và cho con trỏ T trỏ tới đỉnh này. Nếu cây T không rỗng, ta tiến hành giống như khi tìm kiếm, cho con trỏ P chạy trên các đỉnh của cây, bắt đầu từ gốc cây. Nếu P ≠NULL và (P → Xval, P → Yval) = (x, y) thì điều đó có nghĩa là điểm (x, y) đã có sẵn trong cây T, ta dừng lại. Nếu P ≠ NULL và ta cần cho P trỏ tới đỉnh con trái của nó, nhưng P → left là NULL, thì ta tạo ra một đỉnh mới chứa điểm (x, y) và cho con trỏ P → left trỏ tới đỉnh mới này. Còn nếu P ≠ NULL và ta cần cho P trỏ tới đỉnh con phải của P, nhưng P không có đỉnh con phải, thì đỉnh mới cần xen vào là đỉnh con phải của P.

Ví dụ, giả sử chúng ta cần tạo ra cây 2-chiều biểu diễn tập điểm A(15, 31), B(24, 45), C(20, 26), D(8, 12), E(30, 17) và F(13, 9) bằng cách xen vào cây ban đầu rỗng lần lượt các điểm theo thứ tự đã liệt kê. Đầu tiên, xen đỉnh A(15, 31) vào cây rỗng, ta có cây chỉ có một đỉnh gốc A, như trong hình

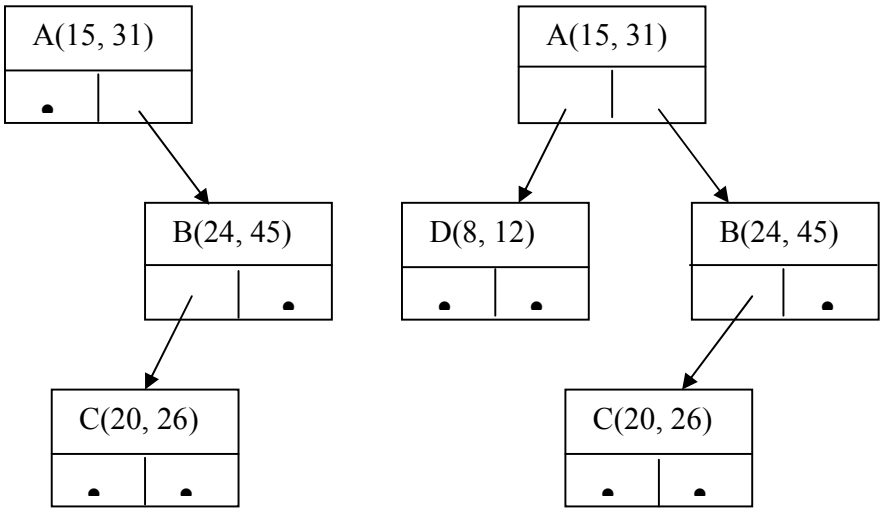
14.2a. Xen vào cây này đỉnh B(24, 25), vì hoành độ của điểm B là 24, lớn hơn hoành độ 15 của điểm A, nên đỉnh B(24, 45) được thêm vào là đỉnh con phải của đỉnh A, ta có cây trong hình 14.2b. Xen đỉnh C(20, 26) vào cây 14.2b. Ta có đỉnh A ở mức chẵn và hoành độ 20 của điểm C lớn hơn hoành độ 15 của điểm A, nên đỉnh C(20, 26) phải được xen vào cây con phải của đỉnh A. Đỉnh con phải của A là đỉnh B ở mức lẻ, và tung độ 26 của C nhỏ hơn tung độ 45 của B, do đó đỉnh C(20, 26) được xen vào như là đỉnh con trái của B và ta có cây hình 14.2c. Tiếp tục, xen đỉnh D(8, 12) vào cây hình 14.2c, vì 8 <15 nên đỉnh D(8, 12) trở thành đỉnh con trái của A, như trong hình 14.2d. Tương tự, xen nốt các đỉnh E(30, 17) và F(13, 9), ta nhận được cây kết quả như trong hình 14.1b.



(a)



(b)



(c) (d)

Hình 14.2. Xen vào cây 2-chiều lần lượt các điểm A(15, 31), B(24, 45), C(20, 26), D(8, 12)

Phép toán loại. Giả sử T là cây 2-chiều, và chúng ta cần loại khỏi cây này đỉnh chứa điểm (x, y). Phép toán loại là phép toán phức tạp nhất trong các phép toán trên cây 2-chiều. Tư tưởng của thuật toán loại trên cây 2-chiều cũng tương tự như trên cây tìm kiếm nhị phân.

Trước hết ta cần tìm đỉnh cần loại, giả sử đó là đỉnh P. Nếu P là đỉnh lá, việc loại P rất đơn giản, ta chỉ cần đặt con trỏ liên kết từ đỉnh cha của P tới P bằng NULL và thu hồi bộ nhớ đã cấp phát cho P. Giả sử P không phải là đỉnh lá, tức là ít nhất một trong hai cây con của P không rỗng. Ta ký hiệu T_l là cây con trái của P, T_r là cây con phải của P. Các hành động tiếp theo phụ thuộc vào cây con phải T_r là rỗng hay không rỗng. Ta xét từng khả năng:

- **Cây con phải T_r không rỗng**. Trong trường hợp này ta thực hiện các bước cơ bản sau:

Bước 1. Tìm đỉnh Q thuộc T_r có thể thay thế cho đỉnh P. Đỉnh Q cần phải phân hoạch miền dữ liệu giống như P.
Bước 2. Thay thế đỉnh P bởi đỉnh Q. Điều này được thực hiện bằng cách chuyển dữ liệu chứa trong đỉnh Q lên đỉnh P, tức là đặt:
P → info = Q → info
P → Xval = Q → Xval
P → Yval = Q → Yval
Bước 3. Loại đệ quy đỉnh Q khỏi cây con T_r.

Chú ý rằng, vì cây con T_r có độ cao thấp hơn cây T nên các lời gọi đệ quy sẽ dẫn tới việc loại đỉnh lá, một việc đơn giản đã nói.

Bây giờ chúng ta làm sáng tỏ bước 1. Chúng ta cần tìm trong cây con T_r một đỉnh Q cho ta sự phân hoạch miền dữ liệu giống như đỉnh P. Điều này có nghĩa là, nếu P ở mức chẵn thì với mọi đỉnh L thuộc cây con trái của P, ta có L → Xval < Q → Xval, và với mọi đỉnh R thuộc cây con phải của P, ta có R → Xval ≥ Q → Xval. Còn nếu P ở mức lẻ, thì L → Yval < Q → Yval và R → Yval ≥ Q → Yval. Việc tìm đỉnh Q thuộc cây con phải T_r thoả mãn các tính chất trên là đơn giản. Nếu P ở mức chẵn, chúng ta chỉ cần tìm đỉnh Q là đỉnh có giá trị Xval là nhỏ nhất trong cây T_r; còn nếu P ở mức lẻ thì Q là đỉnh có giá trị Yval nhỏ nhất trong cây T_r. Chẳng hạn, trong cây hình 14.1b, giả sử ta cần loại đỉnh A(15, 31), khi đó đỉnh thay thế là đỉnh C(20, 26) trong cây con phải của đỉnh A. Từ hình 14.1a, ta thấy khi xoá đi điểm A thì điểm C cho ta sự phân hoạch dữ liệu giống như điểm A.

- **Cây con phải T_r rỗng và cây con trái T_r không rỗng**. Trong trường hợp này, ta thực hiện các bước sau:

  Bước 1. Tìm đỉnh Q thuộc cây con trái T_r có giá trị Xval nhỏ nhất (nếu P ở mức chẵn) hoặc Q có giá trị Yval nhỏ nhất (nếu P ở mức lẻ).
  Bước 2. Thay thế đỉnh P bởi đỉnh Q. Chuyển cây con trái của P thành cây con phải của P, tức là đặt:
  P → right = P → left
  P → left = NULL
  Bước 3. Loại đệ quy đỉnh Q khỏi cây con phải của P.

Phép toán tìm kiếm phạm vi. Vấn đề được đặt ra là, cho trước một điểm (x, y) và một số thực dương r, chúng ta cần tìm tất cả các điểm chứa trong các đỉnh của cây 2-chiều, nằm trong hình tròn tâm (x,y) với bán kính r. Để giải quyết vấn đề này, cần lưu ý rằng, mỗi đỉnh của cây 2-chiều biểu diễn một miền. Miền tương ứng với mỗi đỉnh là miền hình chữ nhật [X_1, X_2, Y_1,

Y_2], trong đó (X_1, Y_1) là toạ độ của góc dưới bên trái và (X_2, Y_2) là toạ độ của góc trên bên phải của hình chữ nhật, miền này gồm tất cả các điểm (x, y) mà $X_1 \le x < X_2$ và $Y_1 \le y < Y_2$. Chẳng hạn, trong cây 2-chiều hình 14.1b, đỉnh A biểu diễn toàn bộ không gian, tức là miền [-∞, +∞; -∞, +∞]. Đỉnh B biểu diễn miền [15, +∞; -∞, +∞], đỉnh D biểu diễn miền [-∞, 15; -∞, +∞]. Miền tương ứng với đỉnh C sẽ là [15, +∞; -∞,45], … Miền được biểu diễn bởi gốc cây 2-chiều có thể xác định được ngay, khi mà không đánh giá được hình chữ nhật chứa toàn bộ dữ liệu, chúng ta có thể xem miền này là toàn bộ không gian. Nếu biết được miền hình chữ nhật tương ứng với một đỉnh, ta có thể xác định được miền hình chữ nhật tương ứng với các đỉnh con của nó. Và do đó, ta có thể xác định được miền hình chữ nhật tương ứng với mọi đỉnh của cây 2-chiều. (Bạn đọc hãy đưa ra thuật toán). Chúng ta sẽ ký hiệu hình tròn tâm (x,y), bán kính r là H, ký hiệu miền hình chữ nhật tương ứng với đỉnh P là R_P. Trong quá trình tìm kiếm, nếu chúng ta thấy miền được biểu diễn bởi đỉnh P không cắt hình tròn H, thì ta bỏ qua toàn bộ cây con gốc P, không cần xem xét cây con đó. Thuật toán tìm các điểm trên cây 2-chiều T (T là con trỏ trỏ tới gốc cây) nằm trong hình tròn H là thuật toán đệ quy như sau:

```
            RangeSearch (T, H)
{
    if  (R_T ∩ H = ϕ)
        return ;
    else  {
          if (( T → Xval , T → Yval) ∈ H )
                In ra điểm (T → Xval, T → Yval) ;
          RangeSearch  (T → left, H );
          RangeSearch  (T → right, H );
          }
}
```

Ví dụ. Giả sử chúng ta cần tìm trên cây 2-chiều hình 14.1b các điểm nằm trong phạm vi hình tròn H với tâm (25, 22), bán kính 8. Đương nhiên là hình tròn H giao với miền được biểu diễn bởi đỉnh A. Kiểm tra ta thấy điểm (15, 31) không nằm trong hình tròn H. Xét đỉnh D(8, 12) là gốc của cây con trái của đỉnh A, miền được biểu diễn bởi đỉnh D không cắt hình tròn H, do đó ta không cần xem xét cây con trái của đỉnh A. Xét đỉnh B(24, 45) là đỉnh con phải của A, miền biểu diễn bởi đỉnh B giao với hình tròn H. Kiểm tra, ta thấy điểm (24, 45) không nằm trong hình tròn H. Lại xét đỉnh C(20, 26),

miền được biểu diễn bởi đỉnh C giao với hình tròn H, và kiểm tra ta thấy điểm (20, 26) nằm trong hình tròn H. Tiếp tục xét đỉnh E(30, 17), và ta cũng thấy điểm (30, 17) nằm trong hình tròn H. Như vậy, ta tìm được hai điểm (20, 26) và (30, 17) nằm trong hình tròn tâm (25, 22), bán kính 8.

14.2.2 Cây k-chiều

Cây 2-chiều để biểu diễn các dữ liệu điểm 2 chiều. Chúng ta cần cây 3-chiều để biểu diễn các điểm (x, y, z) trong không gian 3 chiều, cây 4-chiều để biểu diễn các điểm (x, y, z, t), với t là chiều thời gian chẳng hạn. Tổng quát, các dữ liệu điểm k-chiều (x_o, x_1, …, x_{k-1}) (k > 2) có thể được biểu diễn bởi CTDL cây k-chiều. Cây k-chiều là sự tổng quát hoá tự nhiên của cây 2-chiều. Cây k-chiều cũng là cây nhị phân. Mỗi đỉnh của cây k-chiều là một cấu trúc giống như cấu trúc biểu diễn đỉnh cây 2-chiều, chỉ khác là thay cho các trường Xval và Yval chúng ta sử dụng mảng Xarray[k] để lưu điểm (x_o, x_1, …, x_{k-1}).

```
struct   Node
{
   infoType   info ;
   double   Xarray[k] ;
   Node*   left ;
   Node*   right ;
} ;
```

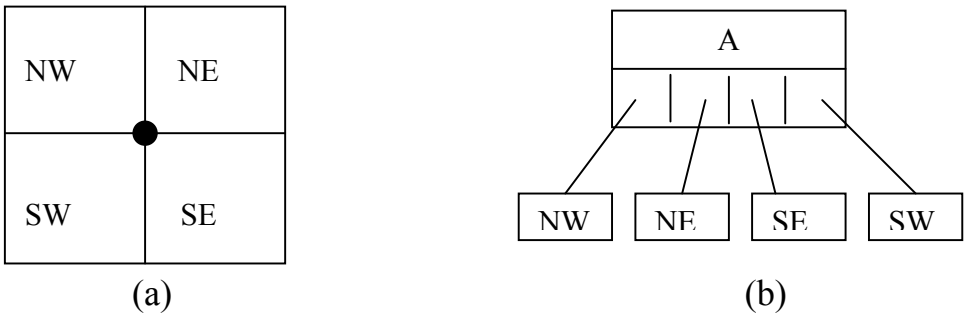
Giống như trong cây 2-chiều, các đỉnh nằm trên cùng một mức của cây k-chiều sẽ phân hoạch các miền mà chúng biểu diễn thành hai nửa theo một chiều không gian nào đó. Cụ thể là các đỉnh nằm ở mức m sẽ phân hoạch các miền tương ứng thành hai phần theo chiều không gian thứ i với i = m mod k. Các thuật toán tìm kiếm, xen, loại, tìm kiếm phạm vi trên cây k-chiều là sự tổng quát hoá tự nhiên của các thuật toán tương ứng trên cây 2-chiều.

Nhận xét. Ưu điểm của cây k-chiều là dễ cài đặt. Cũng như trên cây tìm kiếm nhị phân, dễ dàng thấy rằng, thời gian thực hiện các phép toán tìm kiếm, xen, loại trên cây k-chiều là O(h), trong đó h là độ cao của cây. Trong trường hợp xấu nhất, cây k-chiều với n đỉnh có thể có độ cao n, và do đó thời gian thực hiện các phép toán tìm kiếm, xen, loại là O(n). Người ta đã chứng minh được rằng, thời gian thực hiện phép toán tìm kiếm phạm vị trên

cây k-chiều với n đỉnh là O($kn^{1-1/k}$); nói riêng với k = 2, tìm kiếm phạm vi trên cây 2-chiều đòi hỏi thời gian O($2\sqrt{n}$)

14.3 CÂY TỨ PHÂN

CTDL cây tứ phân (quadtree) được sử dụng để biểu diễncác điểm 2-chiều. Trong cây 2-chiều, mỗi đỉnh của cây phân hoạch miền mà nó biểu diễn thành hai phần hoặc là theo đường thẳng đứng (theo chiều x), hoặc theo đường nằm ngang (theo chiều y), và do đó cây 2-chiều là cây nhị phân. Trong cây tứ phân, mỗi điểm sẽ phân hoạch miền mà nó đại diện thành bốn phần theo cả hai chiều: phần tây-bắc (NW), phần đông-bắc (NE), phần đông- nam (SE) và phần tây-nam (SW), như trong hình 14.3a. Do đó, mỗi đỉnh trong cây tứ phân sẽ có bốn con tương ứng với một trong bốn phần đó như trong hình 14.3b.



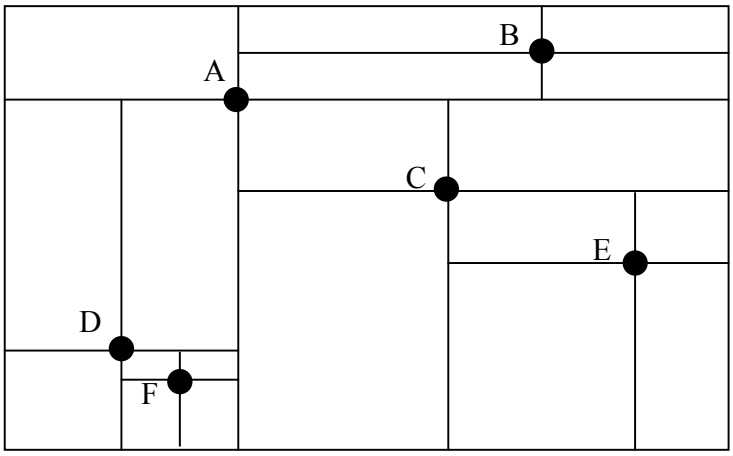
(a) (b)

Hình 14.3. (a) Điểm A phân hoạch một miền thành bốn phần
(b) Điểm A được biểu diễn bởi một đỉnh có bốn con.

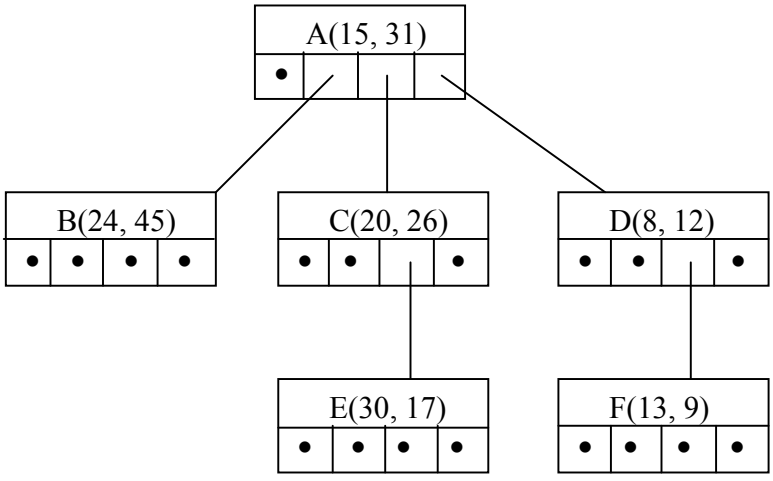
Cấu trúc đỉnh của cây tứ phân tương tự như cấu trúc đỉnh của cây 2-chiều, chỉ khác là ta cần đưa vào bốn con trỏ trỏ tới bốn đỉnh con.

```
struct   Node
{
  infoType    info ;
  double     Xval ;
  double     Yval ;
  Node*    NW-pointer ;
  Node*    NE-pointer ;
  Node*    SE-pointer ;
  Node*    SW-pointer ;
} ;
```

Ví dụ. Giả sử chúng ta cần biểu diễn tập điểm A(15, 31), B(24, 45), C(20, 26), D(8, 12), E(30, 17), F(13, 9), (tập điểm này đã được biểu diễn bởi cây 2-chiều hình 14.1b). Giả sử chúng ta sử dụng các điểm này để phân hoạch mặt phẳng thành các miền con như trong hình 14.4a. Tương ứng với phân hoạch này, tập điểm A, B, C, D, E, F được biểu diễn dưới dạng cây tứ phân hình 14.4b.



(a)



(b)

Hình 14.4. (a) Một cách phân hoạch mặt phẳng bởi các điểm A, B, C, D, E, F
(b) Cây tứ phân tương ứng với phân hoạch

Chú ý rằng, cũng như trong trường hợp xây dựng cây 2-chiều, bởi vì có nhiều cách chọn điểm đại biểu cho một miền, và do đó có nhiều cách phân hoạch một miền thành các miền con, nên có nhiều cây tứ phân khác nhau biểu diễn cùng một tập điểm.

Bây giờ chúng ta xét xem các phép toán tìm kiếm, xen, loại và phép toán tìm kiếm phạm vi được thực hiện như thế nào trên cây tứ phân.

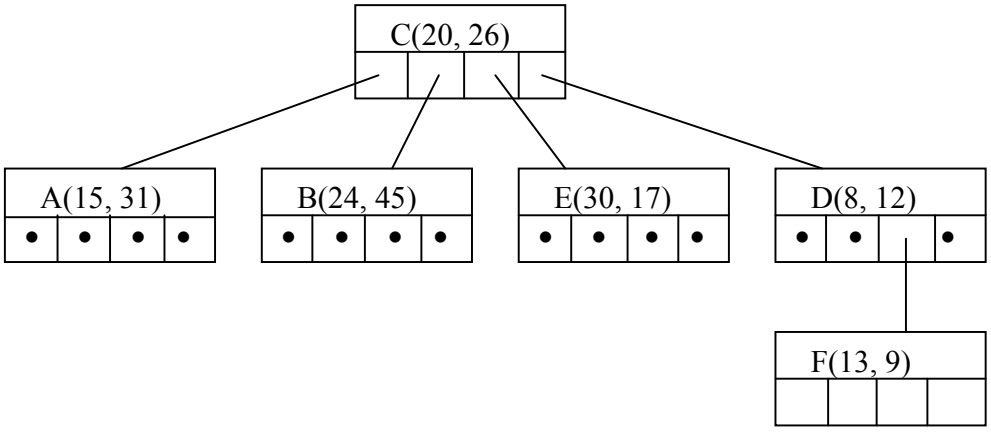
Phép toán tìm kiếm và phép toán xen. Hai phép toán này được tiến hành theo cùng một phương pháp như trên cây 2-chiều. Để tìm kiếm (hoặc để xen vào) một điểm, chúng ta cần tìm miền chứa điểm đó. Mỗi điểm đại diện cho một miền hình chữ nhật. Nếu điểm P có toạ độ (x_P, y_P) biểu diễn miền hình chữ nhật [X_1, X_2; Y_1, Y_2] thì điểm P sẽ chia miền này thành bốn miền con như sau:

Miền con NW = [X_1, x_P; y_P, Y_2]
Miền con NE = [x_P, X_2; y_P, Y_2]
Miền con SW = [X_1, x_P; Y_1, y_P]
Miền con SE = [x_P, X_2; Y_1, y_P]

Muốn biết cây tứ phân có chứa điểm (x, y) hay không, chúng ta xem xét các đỉnh của cây, kể từ gốc. Nếu đỉnh đang xem xét là đỉnh P và điểm (x_P, y_P) là (x, y) thì có nghĩa là ta đã tìm kiếm thành công. Nếu không, tuỳ thuộc điểm (x, y) nằm trong miền con nào trong bốn miền con NW, NE, SE, SW mà ta xem xét đỉnh con của P tương ứng với miền con đó. Nếu đỉnh con chúng ta cần xem xét không có, thì ta kết luận điểm (x, y) không có trong cây tứ phân. Chẳng hạn, giả sử ta cần tìm xem điểm (x, y) = (18, 23) có chứa trong cây tứ phân hình 14.4b hay không. Điểm ở gốc cây là (15, 31) ≠ (18, 23). Điểm (18, 23) nằm trong miền con SE đối với điểm (15, 31). Đi theo con trỏ SE tới đỉnh C(20, 26). Điểm (20, 26) ≠ (18, 23) và (18, 23) nằm trong miền con SW đối với (20, 26). Nhưng SW-pointer của đỉnh C(20, 26) là NULL, do đó ta kết luận điểm (18, 23) không có trong cây tứ phân hình 14.4b.

Bây giờ chúng ta xét phép toán xen. Giả sử ta xen tập điểm trong hình 14.4a vào cây ban đầu rỗng. Ta xen các điểm đó vào cây theo thứ tự C, B, A, D, E, F. Xen điểm C(20, 26) vào cây rỗng ta có cây chỉ có một đỉnh gốc là đỉnh C(20, 26). Các điểm B(24, 45), A(15, 31), D(8, 12), E(30, 17) lần lượt nằm ở miền NE, NW, SW, SE đối với điểm C, và do đó chúng lần lượt được thêm vào cây như các đỉnh con NE, NW, SW, SE của đỉnh C. Đến đây, ta có cây tứ phân gốc là đỉnh C, đỉnh C có bốn đỉnh con theo thứ tự NW, NE, SE SW là A, B, E, D. Bây giờ xen vào cây này điểm F(13, 9). Điểm F nằm trong miền SW đối với đỉnh C. Đi theo con trỏ SW tới đỉnh D. Điểm F nằm trong miền SE đối với đỉnh D, và do đó nó được thêm vào cây như là đỉnh

con SE của đỉnh D. Ta thu được cây hình 14.5, cây này khác với cây hình 14.4b, mặc dầu chúng biểu diễn cùng một tập điểm.



Hình 14.5. Cây tứ phân được tạo thành bằng cách xen vào cây rỗng lần lượt các điểm C, B, A, D, E, F

Phép toán loại. Loại một đỉnh P(x, y) khỏi cây tứ phân là rất phức tạp. Nhớ lại rằng, để loại đỉnh P khỏi cây 2-chiều khi P không phải là đỉnh lá, ta tìm đỉnh Q trong cây con phải T_r của P sao cho Q cho phân hoạch miền dữ liệu giống như P, rồi thay P bởi Q và loại đệ quy Q khỏi cây con T_r. Đối với cây tứ phân ta có thể áp dụng kỹ thuật đó được không? Tức là, để loại đỉnh P khỏi cây tứ phân khi P không phải là đỉnh lá (nếu P là đỉnh lá thì việc loại nó là tầm thường), ta có thể tìm một đỉnh Q thuộc một trong các cây con NW, NE, SE, SW của đỉnh P, có thể thay thế cho đỉnh P, tức là Q phân hoạch miền dữ liệu giống như P? Nếu tìm được đỉnh Q như thế, ta thay P bởi Q và loại đệ quy Q khỏi cây con chứa nó. Song đáng tiếc là không phải lúc nào ta cũng tìm được đỉnh Q thay thế cho đỉnh P. Chẳng hạn, xét cây tứ phân hình 14.4b, nếu ta loại đỉnh A thì có thể tìm được đỉnh thay thế là đỉnh C. Nhưng nếu ta thêm vào cây hình 14.4b một điểm mới G(18, 23), thì trên cây này muốn loại đỉnh A ta không thể tìm được đỉnh thay thế.

Chúng ta có thể loại đỉnh P (không phải là lá) khỏi cây tứ phân theo thuật toán đơn giản sau. Cắt khỏi cây tứ phân cây con gốc P, rồi xen vào cây tứ phân còn lại các đỉnh nằm trong cây con gốc P, trừ đỉnh P. Thuật toán này đương nhiên là kém hiệu quả. Người ta đã đưa ra các thuật toán loại hiệu quả hơn, nhưng phức tạp nên chúng ta không trình bày.

Phép toán tìm kiếm phạm vi trên cây tứ phân được thực hiện theo cùng phương pháp như trên cây 2-chiều.

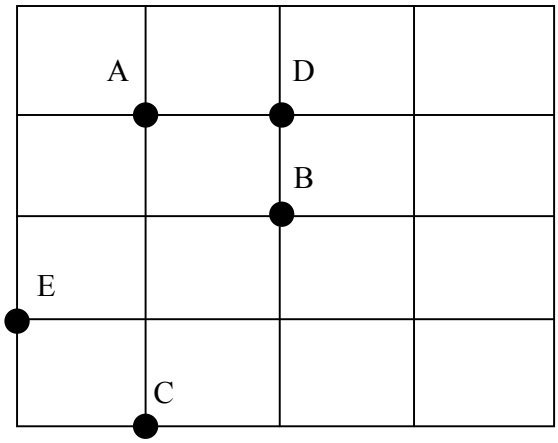
Cây tứ phân cũng rất dễ cài đặt. Thời gian thực hiện các phép toán tìm kiếm và xen vào là O(h), trong đó h là độ cao của cây, tức là cũng như trên cây 2-chiều. Tuy nhiên, cùng biểu diễn một tập điểm thì nói chung cây tứ phân ngắn hơn cây 2-chiều, bởi vì mỗi đỉnh trong cây tứ phân có bốn con, trong cây 2-chiều mỗi đỉnh chỉ có hai con. Phép toán tìm kiếm phạm vi trên cây tứ phân cần thời gian $O(2\sqrt{n})$, với n là số đỉnh trong cây, thời gian này cũng giống như trên cây 2-chiều. Nhược điểm của cây tứ phân là phép toán loại phức tạp. Trong mục sau đây, chúng ta sẽ nghiên cứu một loại cây tứ phân khác được gọi là cây tứ phân MX, trong đó phép loại được thực hiện dễ dàng hơn nhiều.

14.4 CÂY TỨ PHÂN MX

Trong cây 2-chiều cũng như trong cây tứ phân, "hình dạng" của cây phụ thuộc vào thứ tự các điểm được xen vào cây. Chẳng hạn, cây tứ phân hình 14.4b và cây tứ phân hình 14.5 có hình dạng khác nhau, cây hình 14.4b được tạo thành khi ta xen vào cây rỗng các điểm theo thứ tự A, B,C, D, E, F, còn cây hình 14.5 được tạo thành khi ta xen các điểm đó theo thứ tự C, B, A, D, E, F. Do đó trật tự các điểm được xen vào cây ảnh hưởng đến độ cao của cây. Hiệu quả của các phép toán trên cây lại phụ thuộc vào độ cao của cây.

Trong mục này chúng ta sẽ nghiên cứu một dạng cây tứ phân được gọi là ***cây tứ phân MX (MX- QuadTree***). Cây tứ phân MX có ưu điểm là hình dạng của cây không phụ thuộc vào thứ tự các điểm được xen vào cây, hay nói cách khác, một tập điểm được biểu diễn bởi duy nhất một cây tứ phân MX. Một ưu điểm khác là, các phép toán trên cây tứ phân MX như tìm kiếm tìm kiếm phạm vi, xen vào cũng dễ dàng như trên cây tứ phân, còn phép toán loại thì đơn giản và hiệu quả hơn.

Ý tưởng biểu diễn tập điểm trên mặt phẳng bởi cây tứ phân MX là như sau. Chúng ta giả sử rằng tập điểm mà chúng ta quan tâm nằm trong một miền hình vuông. Chia hình vuông này thành một lưới 2^k x 2^k ô vuông (với k nào đó), điểm ở góc dưới bên trái được xem là có toạ độ (0, 0), điểm ở góc trên bên phải có toạ độ (2^k, 2^k). Trong các ứng dụng ta cần chọn k thích hợp sao cho các điểm mà ta quan tâm tương ứng với các điểm (i, j) trên lưới với $0 \le i, j \le 2^k - 1$. Chẳng hạn, hình 14.6 là lưới 2^2 x 2^2 ô vuông (k = 2) với các điểm được quan tâm là A(1, 2), B(2, 2), C(1, 0), D(2, 3) và E(0, 1).



Hình 14.6. Một lưới ô vuông với k = 2

Cấu trúc đỉnh của cây tứ phân MX giống như cấu trúc đỉnh của cây tứ phân ta đã đưa ra trong mục 14.3. Gốc cây tứ phân MX biểu diễn miền hình vuông $[0, 2^k; 0, 2^k]$, hay nói cách khác, hình vuông với điểm dưới bên trái là (0, 0), cạnh là 2^k. Mỗi đỉnh của cây tứ phân MX biểu diễn miền hình vuông với điểm dưới bên trái (i, j) cạnh a, ta ký hiệu hình vuông này là [(i, j), a], và sẽ phân chia hình vuông này thành bốn hình vuông con bằng nhau (xem hình 14.7).

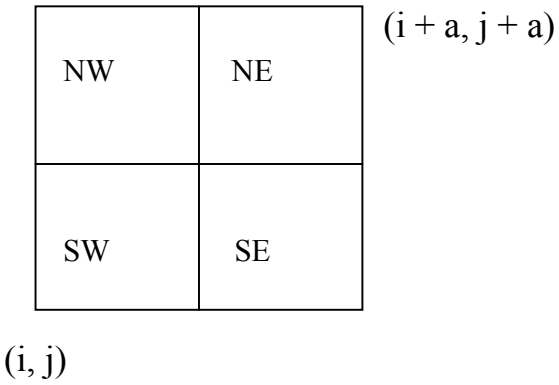
Hình vuông tây-bắc (hình vuông NW): [(i, j + a/2), a/2]

Hình vuông đông-bắc (hình vuông NE): [(i + a/2, j + a/2), a/2]

Hình vuông đông-nam (hình vuông SE): [(i + a/2, j), a/2]

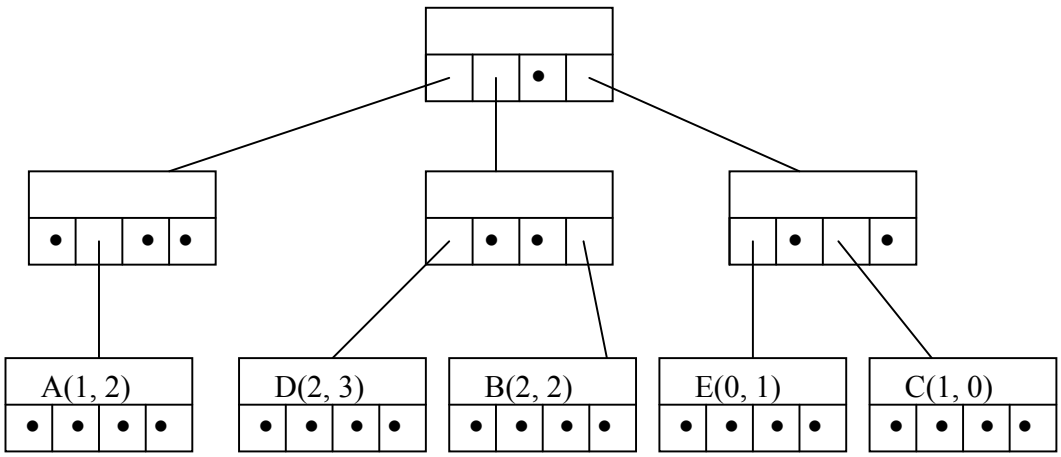
Hình vuông tây-nam (hình vuông SW): [(i, j), a/2]

Mỗi đỉnh chứa con trỏ trỏ tới đỉnh con biểu diễn hình vuông con tương ứng.



Hình 14.7. Đỉnh cây tứ phân MX phân chia hình vuông mà nó biểu diễn thành bốn hình vuông con

Gốc cây (đỉnh ở mức 0) biểu diễn miền hình vuông với cạnh là 2^k. Các đỉnh ở mức 1 biểu diễn các miền hình vuông con với cạnh là 2^{k-1} , … Các đỉnh ở mức k biểu diễn các hình vuông với cạnh là 1. Các điểm dữ liệu (x, y), chẳng hạn các điểm A, B, C, D, E trong hình 14.6, được xem là biểu diễn các ô vuông [(x, y), 1] và do đó được lưu giữ trong các đỉnh ở mức k (các đỉnh lá). Ví dụ, các điểm A(1, 2), B(2, 2), C(1, 0), D(2, 3), E(0, 1) trên lưới ô vuông 2^2 x 2^2 (k = 2) hình 14.6 được biểu diễn bởi cây tứ phân MX hình 14.8, trong đó các con trỏ theo thứ tự từ trái sang phải là NW, NE, SE, SW.

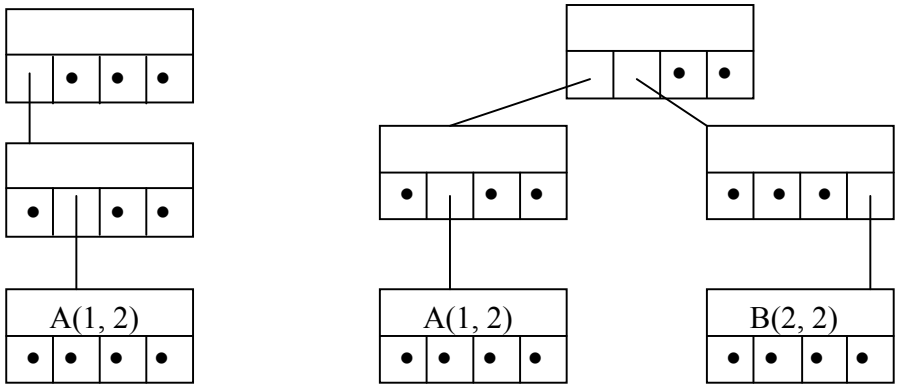


Hình 14.8. Cây tứ phân MX biểu diễn các điểm A, B, C, D, E trên lưới hình 14.6

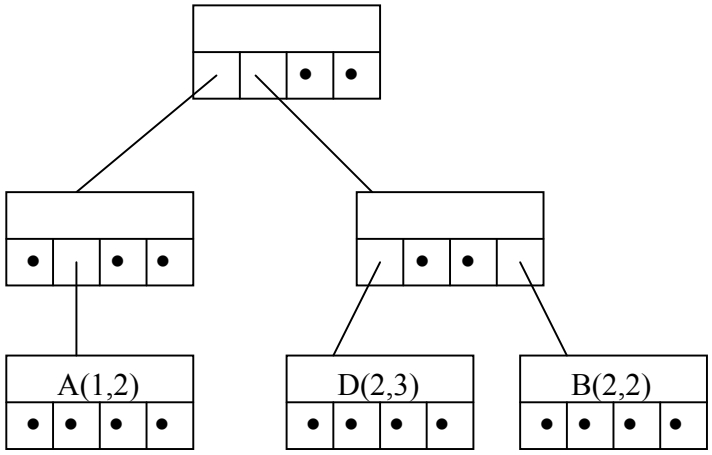
Như vậy, đặc điểm của cây tứ phân MX là: cây tứ phân MX biểu diễn tập điểm dữ liệu trên lưới 2^k x 2^k là cây tứ phân có độ cao k, tất cả các đỉnh lá đều nằm trên cùng một mức (mức k) và các điểm dữ liệu chỉ chứa trong các đỉnh lá.

Phép toán tìm kiếm và xen trên cây tứ phân MX được tiến hành theo cùng một phương pháp. Chúng ta xét phép toán xen. Giả sử ta cần xen vào cây ban đầu rỗng các điểm A, B, C, D, E trên lưới hình 14.6. Trước hết ta xen vào cây điểm A(1, 2). Tạo ra gốc cây biểu diễn hình vuông [(0, 0), 2^2], nó chia hình vuông này thành bốn hình vuông con, điểm A nằm trong hình vuông NW. Tạo ra đỉnh con NW của gốc biểu diễn hình vuông [(0, 1), 2],

đỉnh này lại chia hình vuông thành bống hình vuông con với cạnh là 1. Điểm A nằm ở hình vuông con NE, ta cho con trỏ NE từ đỉnh đó trỏ tới một đỉnh lá chứa điểm A, ta có cây hình 14.9a. Bây giờ ta xen vào cây đó điểm B(2, 2). Điểm B nằm trong hình vuông con NE của hình vuông được biểu diễn bởi gốc. Tạo ra đỉnh con NE của gốc biểu diễn hình vuông [(1, 1), 2], đỉnh này lại cia hình vuông thành bốn hình vuông con với cạnh là 1. Điểm B nằm trong hình vuông con SW, và do đó con trỏ SW của đỉnh đó sẽ trỏ tới một đỉnh lá chứa điểm B, ta nhận được cây hình 14.9b. Tiếp tục, ta xen vào cây hình 14.9b điểm D(2, 3). Điểm D nằm trong hình vuông con NE của hình vuông ứng với gốc. Đi theo con trỏ NE tới đỉnh biểu diễn hình vuông con đó, tức hình vuông [(1, 1), 2]. Điểm D lại nằm trong hình vuông NW, tức hình vuông [(2, 3), 1], và do đó ta cho con trỏ NW của đỉnh đó trỏ tới một đỉnh lá chứa điểm D, ta nhận được cây hình 14.9c. Tương tự xen tiếp các đỉnh còn lại C, E ta có cây kết quả như trong hình 14.8. Ta có nhận xét rằng, nếu ta xen các điểm A, B, C, D, E theo một thứ tự khác bất kỳ thì ta cũng nhận được cây hình 14.8.



(a) (b)



(c)

Hình 14.9. (a) Xen vào cây rỗng điểm A.
(b) Xen vào cây (a) điểm B.
(c) Xen vào cây (b) điểm D.

Phép toán loại. Phép toán loại trên cây tứ phân MX được thực hiện khá đơn giản. Trước hết ta cần lưu ý đến tính chất của cây tứ phân MX: tất cả các đỉnh lá đều nằm trên cùng một mức (mức k) và chỉ các đỉnh lá mới chứa các điểm dữ liệu. Khi thực hiện phép loại, chúng ta phải đảm bảo cây sau khi loại vẫn còn thoả mãn tính chất đó. Thủ tục loại khỏi cây tứ phân MX một đỉnh chứa điểm (x, y) là như sau. Tìm đỉnh lá chứa điểm (x, y), giả sử đó là đỉnh P. Giả sử cha của đỉnh P là đỉnh Q. Để loại đỉnh P, ta chỉ cần đặt con trỏ trong đỉnh Q trỏ tới P bằng NULL và thu hội vùng nhớ của đỉnh P. Sau đó kiểm tra, nếu đỉnh Q còn chứa con trỏ khác NULL thì không cần làm gì nữa, nếu bốn con trỏ trong Q đều là NULL, tức Q trở thành đỉnh lá, thì ta lại loại đỉnh Q. Lặp lại quá trình trên, trường hợp xấu nhất quá trình trên dẫn ta tới xem xét gốc cây có là đỉnh lá hay không.

Ví dụ, xét cây hình 14.8. Để loại đỉnh B, ta chỉ cần đặt con trỏ SW trong đỉnh cha của B bằng NULL. Nhưng nếu loại đỉnh A thì cha của đỉnh A trở thành lá, và do đó ta phải loại tiếp đỉnh đó bằng cách đặt con trỏ NW trong đỉnh gốc bằng NULL.

Phép toán tìm kiếm phạm vi trên cây tứ phân MX được tiến hành tương tự như trên cây tứ phân. Cần chú ý rằng, cây tứ phân và cây tứ phân MX khác nhau ở chỗ, trong cây tứ phân tất cả các đỉnh đều chứa dữ liệu, còn trong cây tứ phân MX thì chỉ các đỉnh lá (các đỉnh ở mức k) mới chứa dữ

liệu. Vì vậy, việc tìm các điểm dữ liệu trên cây tứ phân MX nằm trong phạm vi hình tròn C được tiến hành như sau. Ta xem xét các đỉnh của cây bắt đầu từ đỉnh gốc, giả sử đỉnh đang xem xét là P. Nếu hình vuông mà P biểu diễn không cắt hình tròn C thì ta không cần xem xét tiếp các đỉnh thuộc cây con gốc P. Còn nếu hình vuông ứng với đỉnh P cắt hình tròn C thì ta xem xét tiếp bốn đỉnh con (nếu có) của P, và trong trường hợp các đỉnh con của P là đỉnh lá thì ta kiểm tra xem các điểm chứa trong các đỉnh lá đó có nằm trong hình tròn C hay không. Thuật toán tìm kiếm phạm vi trên cây tứ pân MX được mô tả dưới dạng hàm đệ quy như sau, trong đó T là con trỏ trỏ tới gốc cây.

```
        RangeSearch (T, C)
{
  if (Hình vuông được biểu diễn bởi đỉnh T không cắt C)
                    return ;
  else  if (Đỉnh T ở mức < k – 1)
            {
                RangeSearch (T → NW-pointer, C) ;
                RangeSearch (T → NE-pointer, C) ;
                RangeSearch ( T → SE-pointer, C) ;
                RangeSearch (T → SW-pointer, C) ;
            }
   else if (Đỉnh T ở mức k – 1)
            {
                Kiểm tra xem các điểm nằm trong các đỉnh con của T có
                nằm trong hình tròn C hay không, nếu có thì in ra.
            }
}
```

Dễ dàng thấy rằng, các phép toán tìm kiếm, xen, loại trên cây tứ phân MX biểu diễn tập điểm trên lưới 2^k x 2^k đòi hỏi thời gian O(k), trong đó k là độ cao của cây, và có thể chỉ ra rằng phép toán tìm kiếm phạm vi cần thời gian $O(2^k + m)$, trong đó m là số điểm nằm trong hình tròn.

PHẦN 3

THUẬT TOÁN

CHƯƠNG 15

PHÂN TÍCH THUẬT TOÁN

Với một vấn đề đặt ra có thể có nhiều thuật toán giải, chẳng hạn người ta đã tìm ra rất nhiều thuật toán sắp xếp một mảng dữ liệu (chúng ta sẽ nghiên cứu các thuật toán sắp xếp này trong chương 17). Trong các trường hợp như thế, khi cần sử dụng thuật toán người ta thường chọn thuật toán có thời gian thực hiện ít hơn các thuật toán khác. Mặt khác, khi bạn đưa ra một thuật toán để giải quyết một vấn đề thì một câu hỏi đặt ra là thuật toán đó có ý nghĩa thực tế không? Nếu thuật toán đó có thời gian thực hiện quá lớn chẳng hạn hàng năm, hàng thế kỷ thì đương nhiên không thể áp dụng thuật toán này trong thực tế. Như vậy chúng ta cần đánh giá thời gian thực hiện thuật toán. Phân tích thuật toán, đánh giá thời gian chạy của thuật toán là một lĩnh vực nghiên cứu quan trong của khoa học máy tính. Trong chương này, chúng ta sẽ nghiên cứu phương pháp đánh giá thời gian chạy của thuật toán bằng cách sử dụng ký hiệu ô lớn, và chỉ ra cách đánh gía thời gian chạy thuật toán bằng ký hiệu ô lớn. Trước khi đi tới mục tiêu trên, chúng ta sẽ thảo luận ngắn gọn một số vấn đề liên quan đến thuật toán và tính hiệu quả của thuật toán.

15.1 THUẬT TOÁN VÀ CÁC VẤN ĐỀ LIÊN QUAN

Thuật toán được hiểu là **sự đặc tả chính xác một dãy các bước có thể thực hiện được một cách máy móc** để giải quyết một vấn đề. Cần nhấn mạnh rằng, mỗi thuật toán có một **dữ liệu vào (Input)** và một **dữ liệu ra (Output)**; khi thực hiện thuật toán (thực hiện các bước đã mô tả), thuật toán cần cho ra các dữ liệu ra tương ứng với các dữ liệu vào.

Biểu diễn thuật toán. Để đảm bảo tính chính xác, chỉ có thể hiểu một cách duy nhất, thuật toán cần được mô tả trong một ngôn ngữ lập trình thành

một chương trình (hoặc một hàm, một thủ tục), tức là thuật toán cần được mô tả dưới **dạng mã (code)**. Tuy nhiên, khi trình bày một thuật toán để cho ngắn gọn nhưng vẫn đảm bảo đủ chính xác, người ta thường biểu diễn thuật toán dưới dạng **giả mã (pseudo code)**. Trong cách biểu diễn này, người ta sử dụng các câu lệnh trong một ngôn ngữ lập trình (pascal hoặc C++) và cả các ký hiệu toán học, các mệnh đề trong ngôn ngữ tự nhiên (tiếng Anh hoặc tiếng Việt chẳng hạn). Tất cả các thuật toán được đưa ra trong sách này đều được trình bày theo cách này. Trong một số trường hợp, để người đọc hiểu được ý tưởng khái quát của thuật toán, người ta có thể biểu diễn thuật toán dưới dạng sơ đồ (thường được gọi là sơ đồ khối).

Tính đúng đắn (correctness) của thuật toán. Đòi hỏi trước hết đối với thuật toán là nó phải đúng đắn, tức là khi thực hiện nó phải cho ra các dữ liệu mà ta mong muốn tương ứng với các dữ liệu vào. Chẳng hạn nếu thuật toán được thiết kế để tìm ước chung lớn nhất của 2 số nguyên dương, thì khi đưa vào 2 số nguyên dương (dữ liệu vào) và thực hiện thuật toán phải cho ra một số nguyên dương (dữ liệu ra) là ước chung lớn nhất của 2 số nguyên đó. Chứng minh một cách chặt chẽ (bằng toán học) tính đúng đắn của thuật toán là một công việc rất khó khăn. Tuy nhiên đối với phần lớn các thuật toán được trình bày trong sách này, chúng ta có thể thấy (bằng cách lập luận không hoàn toàn chặt chẽ) các thuật toán đó là đúng đắn, và do đó chúng ta không đưa ra chứng minh chặt chẽ bằng toán học.

Một tính chất quan trong khác của thuật toán là **tính hiệu quả (efficiency)**, chúng ta sẽ thảo luận về tính hiệu quả của thuật toán trong mục tiếp theo.

Đến đây chúng ta có thể đặt câu hỏi: có phải đối với bất kỳ vấn đề nào cũng có thuật toán giải (có thể tìm ra lời giải bằng thuật toán)? câu trả lời là không. Người ta đã phát hiện ra một số vấn đề không thể đưa ra thuật toán để giải quyết nó. Các vấn đề đó được gọi là các **vấn đề không giải được** bằng thuật toán.

15.2 TÍNH HIỆU QUẢ CỦA THUẬT TOÁN

Người ta thường xem xét thuật toán, lựa chọn thuật toán để áp dụng dựa vào các tiêu chí sau:

1. Thuật toán đơn giản, dễ hiểu.
2. Thuật toán dễ cài đặt (dễ viết chương trình)
3. Thuật toán cần ít bộ nhớ
4. Thuật toán chạy nhanh

Khi cài đặt thuật toán chỉ để sử dụng một số ít lần, người ta thường lựa chọn thuật toán theo tiêu chí 1 và 2. Tuy nhiên, có những thuật toán được sử dụng rất nhiều lần, trong nhiều chương trình, chẳng hạn các thuật toán sắp xếp, các thuật toán tìm kiếm, các thuật toán đồ thị… Trong các trường hợp như thế người ta lựa chọn thuật toán để sử dụng theo tiêu chí 3 và 4. Hai tiêu chí này được nói tới như là tính hiệu quả của thuật toán. **Tính hiệu quả** của thuật toán gồm hai yếu tố: dung lượng bộ nhớ mà thuật toán đòi hỏi và thời gian thực hiện thuật toán. Dung lượng bộ nhớ gồm bộ nhớ dùng để lưu dữ liệu vào, dữ liệu ra, và các kết quả trung gian khi thực hiện thuật toán; dung lượng bộ nhớ mà thuật toán đòi hỏi còn được gọi là **độ phức tạp không gian** của thuật toán. Thời gian thực hiện thuật toán được nói tới như là **thời gian chạy** (running time) hoặc **độ phức tạp thời gian** của thuật toán. Sau này chúng ta chỉ quan tâm tới đánh giá thời gian chạy của thuật toán.

Đánh giá thời gian chạy của thuật toán bằng cách nào? Với cách tiếp cận thực nghiệm chúng ta có thể cài đặt thuật toán và cho chạy chương trình trên một máy tính nào đó với một số dữ liệu vào. Thời gian chạy mà ta thu được sẽ phụ thuộc vào nhiều nhân tố:

- Kỹ năng của người lập trình
- Chương trình dịch

- Tốc độ thực hiện các phép toán của máy tính
- Dữ liệu vào

Vì vậy, trong cách tiếp cận thực nghiệm, ta không thể nói thời gian chạy của thuật toán là bao nhiêu đơn vị thời gian. Chẳng hạn câu nói "thời gian chạy của thuật toán là 30 giây" là không thể chấp nhận được. Nếu có hai thuật toán A và B giải quyết cùng một vấn đề, ta cũng không thể dùng phương pháp thực nghiệm để kết luận thuật toán nào chạy nhanh hơn, bởi vì ta mới chỉ chạy chương trình với một số dữ liệu vào.

Một cách tiếp cận khác để đánh giá thời gian chạy của thuật toán là phương pháp phân tích sử dụng các công cụ toán học. Chúng ta mong muốn có kết luận về thời gian chạy của một thuật toán mà nó không phụ thuộc vào sự cài đặt của thuật toán, không phụ thuộc vào máy tính mà trên đó thuật toán được thực hiện.

Để phân tích thuật toán chúng ta cần sử dụng khái niệm **cỡ** (size) của dữ liệu vào. Cỡ của dữ liệu vào được xác định phụ thuộc vào từng thuật toán. Ví dụ, trong thuật toán tính định thức của ma trận vuông cấp n, ta có thể chọn cỡ của dữ liệu vào là cấp n của ma trận; còn đối với thuật toán sắp xếp mảng cỡ n thì cỡ của dữ liệu vào chính là cỡ n của mảng. Đương nhiên là có vô số dữ liệu vào cùng một cỡ. Nói chung trong phần lớn các thuật toán, cỡ của dữ liệu vào là một số nguyên dương n. Thời gian chạy của thuật toán phụ thuộc vào cỡ của dữ liệu vào; chẳng hạn tính định thức của ma trận cấp 20 đòi hỏi thời gian chạy nhiều hơn tính định thức của ma trận cấp 10. Nói chung, cỡ của dữ liệu càng lớn thì thời gian thực hiện thuật toán càng lớn. Nhưng thời gian thực hiện thuật toán không chỉ phụ thuộc vào cỡ của dữ liệu vào mà còn phụ thuộc vào chính dữ liệu vào.

Trong số các dữ liệu vào cùng một cỡ, thời gian chạy của thuật toán cũng thay đổi. Chẳng hạn, xét bài toán tìm xem đối tượng a có mặt trong danh sách ($a_{1,...}$, $a_{i,...,}a_n$) hay không. Thuật toán được sử dụng là thuật toán tìm kiếm tuần tự: Xem xét lần lượt từng phần tử của danh sách cho tới khi

phát hiện ra đối tượng cần tìm thì dừng lại, hoặc đi hết danh sách mà không gặp phần tử nào bằng a. Ở đây cỡ của dữ liệu vào là n, nếu một danh sách với a là phần tử đầu tiên, ta chỉ cần một lần so sánh và đây là trường hợp tốt nhất, nhưng nếu một danh sách mà a xuất hiện ở vị trí cuối cùng hoặc a không có trong danh sách, ta cần n lần so sánh a với từng a_i (i=1,2,…,n), trường hợp này là trường hợp xấu nhất. Vì vậy, chúng ta cần đưa vào khái niệm thời gian chạy trong trường hợp xấu nhất và thời gian chạy trung bình.

Thời gian chạy trong trường hợp xấu nhất (worst-case running time) của một thuật toán là thời gian chạy lớn nhất của thuật toán đó trên tất cả các dữ liệu vào cùng cỡ . Chúng ta sẽ ký hiệu thời gian chạy trong trường hợp xấu nhất là T(n), trong đó n là cỡ của dữ liệu vào. Sau này khi nói tới thời gian chạy của thuật toán chúng ta cần hiểu đó là thời gian chạy trong trường hợp xấu nhất. Sử dụng thời gian chạy trong trường hợp xấu nhất để biểu thị thời gian chạy của thuật toán có nhiều ưu điểm. Trước hết, nó đảm bảo rằng, thuật toán không khi nào tiêu tốn nhiều thời gian hơn thời gian chạy đó. Hơn nữa, trong các áp dụng, trường hợp xấu nhất cũng thường xuyên xảy ra.

Chúng ta xác định **thời gian chạy trung bình** (average running time) của thuật toán là số trung bình cộng của thời gian chạy của thuật toán đó trên tất cả các dữ liệu vào cùng cỡ n. Thời gian chạy trung bình của thuật toán sẽ được ký hiệu là $T_{tb}(n)$. Đánh giá thời gian chạy trung bình của thuật toán là công việc rất khó khăn, cần phải sử dụng các công cụ của xác suất, thống kê và cần phải biết được phân phối xác suất của các dữ liệu vào. Rất khó biết được phân phối xác suất của các dữ liệu vào. Các phân tích thường phải dựa trên giả thiết các dữ liệu vào có phân phối xác suất đều. Do đó, sau này ít khi ta đánh giá thời gian chạy trung bình.

Để có thể phân tích đưa ra kết luận về thời gian chạy của thuật toán độc lập với sự cài đặt thuật toán trong một ngôn ngữ lập trình, độc lập với máy tính được sử dụng để thực hiện thuật toán, chúng ta đo **thời gian chạy của thuật toán bởi số phép toán sơ cấp cần phải thực hiện** khi ta thực

hiện thuật toán. Cần chú ý rằng, các phép toán sơ cấp là các phép toán số học, các phép toán logic, các phép toán so sánh,…, nói chung, các phép toán sơ cấp cần được hiểu là các phép toán mà khi thực hiện chỉ đòi hỏi một thời gian cố định nào đó (thời gian này nhiều hay ít là phụ thuộc vào tốc độ của máy tính). Như vậy chúng ta xác định thời gian chạy T(n) là số phép toán sơ cấp mà thuật toán đòi hỏi, khi thực hiện thuật toán trên dữ liệu vào cỡ n. Tính ra biểu thức mô tả hàm T(n) được xác định như trên là không đơn giản, và biểu thức thu được có thể rất phức tạp. Do đó, chúng ta sẽ chỉ quan tâm tới **tốc độ tăng (rate of growth)** của hàm T(n), tức là tốc độ tăng của thời gian chạy khi cỡ dữ liệu vào tăng. Ví dụ, giả sử thời gian chạy của thuật toán là $T(n) = 3n^2 + 7n + 5$ (phép toán sơ cấp). Khi cỡ n tăng, hạng thức $3n^2$ quyết định tốc độ tăng của hàm T(n), nên ta có thể bỏ qua các hạng thức khác và có thể nói rằng thời gian chạy của thuật toán tỉ lệ với bình phương của cỡ dữ liệu vào. Trong mục tiếp theo chúng ta sẽ định nghĩa ký hiệu ô lớn và sử dụng ký hiệu ô lớn để biểu diễn thời gian chạy của thuật toán.

15.3 KÝ HIỆU Ô LỚN VÀ BIỂU DIỄN THỜI GIAN CHẠY BỞI KÝ HIỆU Ô LỚN

15.3.1 Định nghĩa ký hiệu ô lớn

Bây giờ chúng ta đưa ra định nghĩa khái niệm một hàm là "ô lớn" của một hàm khác.

Định nghĩa. Giả sử f(n) và g(n) là các hàm thực không âm của đối số nguyên không âm n. Ta nói "f(n) là ô lớn của g(n)" và viết là

f(n) = O(g(n))

nếu tồn tại các hằng số dương c và n_0 sao cho f(n) <= cg(n) với mọi n >= n_0.

Như vậy, f(n) = O(g(n)) có nghĩa là hàm f(n) bị chặn trên bởi hàm g(n) với một nhân tử hằng nào đó khi n đủ lớn. Muốn chứng minh được f(n) = O(g(n)), chúng ta cần chỉ ra nhân tử hằng c , số nguyên dương n_0 và chứng minh được f(n) <= cg(n) với mọi n >= n_o .

Ví dụ. Giả sử $f(n) = 5n^3 + 2n^2 + 13n + 6$, ta có:

$$f(n) = 5n^3 + 2n^2 + 13n + 6 <= 5n^3 + 2n^3 + 13n^3 + 6n^3 = 26n^3$$

Bất đẳng thức trên đúng với mọi n >= 1, và ta có $n_0 = 1$, c = 26. Do đó, ta có thể nói $f(n) = O(n^3)$. Tổng quát nếu f(n) là một đa thức bậc k của n:

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0 \text{ thì } f(n) = O(n^k)$$

Sau đây chúng ta đưa ra một số hệ quả từ định nghĩa ký hiệu ô lớn, nó giúp chúng ta hiểu rõ bản chất ký hiệu ô lớn. (Lưu ý, các hàm mà ta nói tới đều là các hàm thực không âm của đối số nguyên dương)

- Nếu $f(n) = g(n) + g_1(n) + ... + g_k(n)$, trong đó các hàm $g_i(n)$ (i=1,...,k) tăng chậm hơn hàm g(n) (tức là $g_i(n)/g(n) \dashrightarrow 0$, khi $n \dashrightarrow 0$) thì $f(n) = O(g(n))$
- Nếu $f(n) = O(g(n))$ thì $f(n) = O(d.g(n))$, trong đó d là hằng số dương bất kỳ
- Nếu $f(n) = O(g(n))$ và $g(n) = O(h(n))$ thì $f(n) = O(h(n))$ (tính bắc cầu)

Các kết luận trên dễ dàng được chứng minh dựa vào định nghĩa của ký hiệu ô lớn. Đến đây, ta thấy rằng, chẳng hạn nếu $f(n) = O(n^2)$ thì $f(n) = O(75n^2)$, $f(n) = O(0,01n^2)$, $f(n) = O(n^2 + 7n + \log n)$, $f(n) = O(n^3)$,..., tức là có vô số hàm là cận trên (với một nhân tử hằng nào đó) của hàm f(n).

Một nhận xét quan trọng nữa là, ký hiệu O(g(n)) xác định một tập hợp vô hạn các hàm bị chặn trên bởi hàm g(n), cho nên ta viết f(n) = O(g(n)) chỉ có nghĩa f(n) là một trong các hàm đó.

15.3.2 Biểu diễn thời gian chạy của thuật toán

Thời gian chạy của thuật toán là một hàm của cỡ dữ liệu vào: hàm T(n). Chúng ta sẽ biểu diễn thời gian chạy của thuật toán bởi ký hiệu ô lớn: T(n) = O(f(n)), biểu diễn này có nghĩa là thời gian chạy T(n) bị chặn trên bởi hàm f(n). Thế nhưng như ta đã nhận xét, một hàm có vô số cận trên. Trong

số các cận trên của thời gian chạy, chúng ta sẽ lấy **cận trên chặt** (tight bound) để biểu diễn thời gian chạy của thuật toán.

Định nghĩa. Ta nói f(n) là cận trên chặt của T(n) nếu

- T(n) = O(f(n)), và
- Nếu T(n) = O(g(n)) thì f(n) = O(g(n)).

Nói một cách khác, f(n) là cận trên chặt của T(n) nếu nó là cận trên của T(n) và ta không thể tìm được một hàm g(n) là cận trên của T(n) mà lại tăng chậm hơn hàm f(n).

Sau này khi nói thời gian chạy của thuật toán là O(f(n)), chúng ta cần hiểu f(n) là cận trên chặt của thời gian chạy.

Nếu T(n) = O(1) thì điều này có nghĩa là thời gian chạy của thuật toán bị chặn trên bởi một hằng số nào đó, và ta thường nói thuật toán có thời gian chạy hằng. Nếu T(n) = O(n), thì thời gian chạy của thuật toán bị chặn trên bởi hàm tuyến tính, và do đó ta nói thời gian chạy của thuật toán là tuyến tính. Các cấp độ thời gian chạy của thuật toán và tên gọi của chúng được liệt kê trong bảng sau:

Ký hiệu ô lớn	Tên gọi
O(1)	hằng
O(logn)	logarit
O(n)	tuyến tính
O(nlogn)	nlogn
$O(n^2)$	bình phương
$O(n^3)$	lập phương
$O(2^n)$	mũ

Đối với một thuật toán, chúng ta sẽ đánh giá thời gian chạy của nó thuộc cấp độ nào trong các cấp độ đã liệt kê trên. Trong bảng trên, chúng ta đã sắp xếp các cấp độ thời gian chạy theo thứ tự tăng dần, chẳng hạn thuật

toán có thời gian chạy là O(logn) chạy nhanh hơn thuật toán có thời gian chạy là O(n),... Các thuật toán có thời gian chạy là $O(n^k)$, với k = 1,2,3,..., được gọi là các thuật toán **thời gian chạy đa thức** (polynimial-time algorithm). Để so sánh thời gian chạy của các thuật toán thời gian đa thức và các thuật toán thời gian mũ, chúng ta hãy xem xét bảng sau:

Thời gian chạy	Cỡ dữ liệu vào					
	10	20	30	40	50	60
n	0,00001 giây	0,00002 giây	0,00003 giây	0,00004 giây	0,00005 giây	0,00006 giây
n^2	0,0001 giây	0,0004 giây	0,0009 giây	0,0016 giây	0,0025 giây	0,0036 giây
n^3	0,001 giây	0,008 giây	0,027 giây	0,064 giây	0,125 giây	0,216 giây
n^5	0,1 giây	3,2 giây	24,3 giây	1,7 phút	5,2 phút	13 phút
2^n	0,001 giây	1,0 giây	17,9 phút	12,7 ngày	35,7 năm	366 thế kỷ
3^n	0,059 giây	58 phút	6,5 năm	3855 thế kỷ	2.10^8 thế kỷ	1,3. 10^{13} thế kỷ

Trong bảng trên, ta giả thiết rằng mỗi phép toán sơ cấp cần 1 micro giây để thực hiện. Thuật toán có thời gian chạy n^2, với cỡ dữ liệu vào n = 20, nó đòi hỏi thời gian chạy là $20^2 x 10^{-6} = 0,004$ giây. Đối với các thuật toán thời gian mũ, ta thấy rằng thời gian chạy của thuật toán là chấp nhận được chỉ với các dữ liệu vào có cỡ rất khiêm tốn, n < 30; khi cỡ dữ liệu vào tăng, thời gian chạy của thuật toán tăng lên rất nhanh và trở thành con số khổng lồ. Chẳng hạn, thuật toán với thời gian chạy 3^n, để tính ra kết quả với dữ liệu vào cỡ 60, nó đòi hỏi thời gian là $1,3x10^{13}$ thế kỷ! Để thấy con số này khổng lồ đến mức nào, ta hãy liên tưởng tới vụ nổ "big-bang", "big-bang" được ước tính là xảy ra cách đây $1,5x10^8$ thế kỷ. Chúng ta không hy vọng có thể áp dụng các thuật toán có thời gian chạy mũ trong tương lai nhờ tăng tốc độ

máy tính, bởi vì không thể tăng tốc độ máy tính lên mãi được, do sự hạn chế của các quy luật vật lý. Vì vậy nghiên cứu tìm ra các thuật toán hiệu quả (chạy nhanh) cho các vấn đề có nhiều ứng dụng trong thực tiễn luôn luôn là sự mong muốn của các nhà tin học.

15.4 ĐÁNH GIÁ THỜI GIAN CHẠY CỦA THUẬT TOÁN

Mục này trình bày các kỹ thuật để đánh giá thời gian chạy của thuật toán bởi ký hiệu ô lớn. Cần lưu ý rằng, đánh giá thời gian chạy của thuật toán là công việc rất khó khăn, đặc biệt là đối với các thuật toán đệ quy. Tuy nhiên các kỹ thuật đưa ra trong mục này cho phép đanh giá được thời gian chạy của hầu hết các thuật toán mà ta gặp trong thực tế. Trước hết chúng ta cần biết cách thao tác trên các ký hiệu ô lớn. Quy tắc "cộng các ký hiệu ô lớn" sau đây được sử dụng thường xuyên nhất.

15.4.1 Luật tổng

Giả sử thuật toán gồm hai phần (hoặc nhiều phần), thời gian chạy của phần đầu là $T_1(n)$, phần sau là $T_2(n)$. Khi đó thời gian chạy của thuật toán là $T_1(n) + T_2(n)$ sẽ được suy ra từ sự đánh giá của $T_1(n)$ và $T_2(n)$ theo luật sau:

Luật tổng. Giả sử $T_1(n) = O(f(n))$ và $T_2(n) = O(g(n))$. Nếu hàm f(n) tăng nhanh hơn hàm g(n), tức là $g(n) = O(f(n))$, thì $T_1(n) + T_2(n) = O(f(n))$.

Luật này được chứng minh như sau. Theo định nghĩa ký hiệu ô lớn, ta tìm được các hằng số c_1, c_2, c_3 và n_1, n_2, n_3 sao cho

$T_1(n) <= c_1 f(n)$ với $n >= n_1$

$T_2(n) <= c_2 g(n)$ với $n >= n_2$

$g(n) <= c_3 f(n)$ với $n >= n_3$

Đặt $n_0 = \max(n_1, n_2, n_3)$. Khi đó với mọi $n >= n_0$, ta có

$$T_1(n) + T_2(n) <= c_1 f(n) + c_2 g(n)$$

$$<= c_1 f(n) + c_2 c_3 f(n) = (c_{1+} c_2 c_3) f(n)$$

Như vậy với $c = c_1 + c_2\ c_3$ thì

$T_1(n) + T_2(n) <= cf(n)$ với mọi $n >= n_0$

Ví dụ. Giả sử thuật toán gồm ba phần, thời gian chạy của từng phần được đánh giá là $T_1(n) = O(nlogn)$, $T_2(n) = O(n^2)$ và $T_3(n) = O(n)$. Khi đó thời gian chạy của toàn bộ thuật toán là $T(n) = T_1(n) + T_2(n) + T_3(n) = O(n^2)$, vì hàm n^2 tăng nhanh hơn các hàm nlogn và n.

15.4.2 Thời gian chạy của các lệnh

Các thuật toán được đưa ra trong sách này sẽ được trình bày dưới dạng giả mã sử dụng các câu lệnh trong C/C++. Dựa vào luật tổng, đánh giá thời gian chạy của thuật toán được quy về đánh giá thời gian chạy của từng câu lệnh.

Thời gian thực hiện các phép toán sơ cấp là O(1).

1. Lệnh gán

Lệnh gán có dạng

X = <biểu thức>

Thời gian chạy của lệnh gán là thời gian thực hiện biểu thức. Trường hợp hay gặp nhất là biểu thức chỉ chứa các phép toán sơ cấp, và thời gian thực hiện nó là O(1). Nếu biểu thức chứa các lời gọi hàm thì ta phải tính đến thời gian thực hiện hàm, và do đó trong trường hợp này thời gian thực hiện biểu thức có thể không là O(1).

2. Lệnh lựa chọn

Lệnh lựa chọn **if-else** có dạng

if (<điều kiện>)

    lệnh 1

else

    lệnh 2

Trong đó, điều kiện là một biểu thức cần được đánh giá, nếu điều kiện đúng thì lệnh 1 được thực hiện, nếu không thì lệnh 2 được thực hiện. Giả sử thời gian đánh giá điều kiện là $T_0(n)$, thời gian thực hiện lệnh 1 là $T_1(n)$, thời gian thực hiện lệnh 2 là $T_2(n)$. Thời gian thực hiện lệnh lựa chọn if-else sẽ là thời gian lớn nhất trong các thời gian $T_0(n) + T_1(n)$ và $T_0(n) + T_1(n)$.

Trường hợp hay gặp là kiểm tra điều kiện chỉ cần O(1). Khi đó nếu $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ và f(n) tăng nhanh hơn g(n) thì thời gian chạy của lệnh if-else là O(f(n)); còn nếu g(n) tăng nhanh hơn f(n) thì lệnh if-else cần thời gian O(g(n)).

Thời gian chạy của lệnh lựa chọn switch được đánh giá tương tự như lệnh if-else, chỉ cần lưu ý rằng, lệnh if-else có hai khả năng lựa chọn, còn lệnh switch có thể có nhiều hơn hai khả năng lựa chọn.

3. Các lệnh lặp

Các lệnh lặp:

for, while, do-while

Để đánh giá thời gian thực hiện một lệnh lặp, trước hết ta cần đánh giá số tối đa các lần lặp, giả sử đó là L(n). Sau đó đánh giá thời gian chạy của mỗi lần lặp, chú ý rằng thời gian thực hiện thân của một lệnh lặp ở các lần lặp khác nhau có thể khác nhau, giả sử thời gian thực hiện thân lệnh lặp ở lần thứ i (i=1,2,..., L(n)) là $T_i(n)$. Mỗi lần lặp, chúng ta cần kiểm tra điều kiện lặp, giả sử thời gian kiểm tra là $T_0(n)$. Như vậy thời gian chạy của lệnh lặp là:

$$\sum_{i=1}^{L(n)} \left(T_0(n) + T_i(n)\right)$$

Công đoạn khó nhất trong đánh giá thời gian chạy của một lệnh lặp là đánh giá số lần lặp. Trong nhiều lệnh lặp, đặc biệt là trong các lệnh lặp for, ta có thể thấy ngay số lần lặp tối đa là bao nhiêu. Nhưng cũng không ít các

lệnh lặp, từ điều kiện lặp để suy ra số tối đa các lần lặp, cần phải tiến hành các suy diễn không đơn giản.

Trường hợp hay gặp là: kiểm tra điều kiện lặp (thông thường là đánh giá một biểu thức) chỉ cần thời gian O(1), thời gian thực hiện các lần lặp là như nhau và giả sử ta đánh giá được là O(f(n)); khi đó, nếu đánh giá được số lần lặp là O(g(n)), thì thời gian chạy của lệnh lặp là O(g(n)f(n)).

Ví dụ 1. Giả sử ta có mảng A các số thực, cỡ n và ta cần tìm xem mảng có chứa số thực x không. Điều đó có thể thực hiện bởi thuật toán tìm kiếm tuần tự như sau:

```
(1)   i = 0;
(2)   while (i < n &&  x != A[i])
(3)   i++;
```

Lệnh gán (1) có thời gian chạy là O(1). Lệnh lặp (2)-(3) có số tối đa các lần lặp là n, đó là trường hợp x chỉ xuất hiện ở thành phần cuối cùng của mảng A[n-1] hoặc x không có trong mảng. Thân của lệnh lặp là lệnh (3) có thời gian chạy O(1). Do đó, lệnh lặp có thời gian chạy là O(n). Thuật toán gồm lệnh gán và lệnh lặp với thời gian là O(1) và O(n), nên thời gian chạy của nó là O(n).

Ví dụ 2. Thuật toán tạo ra ma trận đơn vị A cấp n;

```
(1)  for (i = 0 ; i < n ;  i++)
(2)    for (j = 0 ; j < n ; j++)
(3)              A[i][j] = 0;
(4)  for (i = 0 ; i < n ; i++)
(5)              A[i][i] = 1;
```

Thuật toán gồm hai lệnh lặp for. Lệnh lặp for đầu tiên (các dòng (1)-(3)) có thân lại là một lệnh lặp for ((2)-(3)). Số lần lặp của lệnh for ((2)-(3)) là n, thân của nó là lệnh (3) có thời gian chạy là O(1), do đó thời gian chạy

của lệnh lặp for này là O(n). Lệnh lặp for ((1)-(3)) cũng có số lần lặp là n, thân của nó có thời gian đã đánh giá là O(n), nên thời gian của lệnh lặp for ((1)-(3)) là $O(n^2)$. Tương tự lệnh for ((4)-(5)) có thời gian chạy là O(n). Sử dụng luật tổng, ta suy ra thời gian chạy của thuật toán là $O(n^2)$.

15.5 PHÂN TÍCH CÁC HÀM ĐỆ QUY

Các hàm đệ quy là các hàm có chứa lời gọi hàm đến chính nó. Trong mục này, chúng ta sẽ trình bầy phương pháp chung để phân tích các hàm đệ quy, sau đó sẽ đưa ra một số kỹ thuật phân tích một số lớp hàm đệ quy hay gặp.

Giả sử ta có hàm đệ quy F, thời gian chạy của hàm này là T(n), với n là cỡ dữ liệu vào. Khi đó thời gian chạy của các lời gọi hàm ở trong hàm F sẽ là T(m) với $m < n$. Trước hết ta cần đánh giá thời gian chạy của hàm F trên dữ liệu cỡ nhỏ nhất n = 1, giả sử T(1) = a với a là một hằng số nào đó. Sau đó bằng cách đánh giá thời gian chạy của các câu lệnh trong thân của hàm F, chúng ta sẽ tìm ra quan hệ đệ quy biểu diễn thời gian chạy của hàm F thông qua lời gọi hàm, tức là biểu diễn T(n) thông qua các T(m), với $m < n$. Chẳng hạn, giả sử hàm đệ quy F chứa hai lời gọi hàm với thời gian chạy tương ứng là $T(m_1)$ và $T(m_2)$, trong đó $m_1, m_2 < n$, khi đó ta thu được quan hệ đệ quy có dạng như sau:

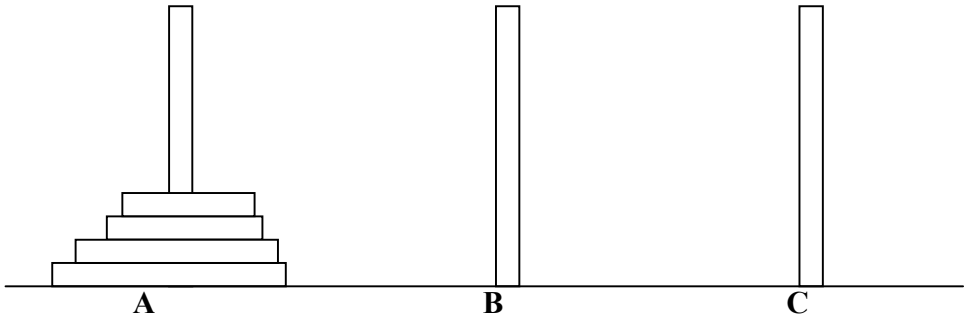
$$T(1) = 1$$

$$T(n) = f(T(m_1),T(m_2))$$

Trong đó, f là một biểu thức nào đó của $T(m_1)$ và $T(m_2)$. Giải quan hệ đệ quy trên, chúng ta sẽ đánh giá được thời gian chạy T(n). Nhưng cần lưu ý rằng, giải các quan hệ đệ quy là rất khó khăn, chúng ta sẽ đưa ra kỹ thuật giải cho một số trường hợp đặc biệt.

Ví dụ (Bài toán tháp Hà Nội). Có ba vị trí A, B, C. Ban đầu ở vị trí A có n đĩa khác nhau được đặt chồng lên nhau theo thứ tự nhỏ dần, tức là đĩa lớn nhất ở dưới cùng, đĩa nhỏ nhất ở trên cùng. Đòi hỏi phải chuyển n đĩa từ

vị trí A sang vị trí B, được sử dụng vị trí C làm vị trí trung gian, mỗi lần chỉ được phép chuyển đĩa trên cùng ở một vị trí đặt lên đỉnh tháp ở vị trí khác, nhưng không được đặt đĩa to lên trên đĩa nhỏ hơn.



Hình 15.1. Trạng thái ban đầu của bài toán tháp Hà Nội

Để chuyển n đĩa từ vị trí A sang vị trí B ta làm như sau: đầu tiên chuyển n-1 đĩa bên trên ở vị trí A sang vị trí C, rồi chuyển đĩa lớn nhất ở vị trí A sang vị trí B, sau đó chuyển n-1 đĩa ở vị trí C sang vị trí B. Việc chuyển n-1 đĩa ở vị trí này sang vị trí khác được thực hiện bằng áp dụng đệ quy thủ trục trên

```
        HanoiTower(n, A, B, C)
        // chuyển n đĩa ở A sang B.
{
   if (n = =1)
         chuyển một đĩa ở A sang B;
  else {
                HanoiTower(n-1,A, C, B);
                chuyển một đĩa ở A sang B;
                HanoiTower(n-1, C, B, A);
         }
}
```

Chúng ta phân tích hàm đệ quy HanoiTower. Chuyển một đĩa ở vị trí này sang vị trí khác là phép toán sơ cấp, ký hiệu T(n) là số lần chuyển (số phép toán sơ cấp) cần thực hiện để chuyển n đĩa ở một vị trí sang vị trí khác. Xem xét thân của hàm HanoiTower, ta có quan hệ đệ quy sau:

$T(1) = 1$

$T(n) = 2T(n-1) + 1$

Có thể tìm ra nghiệm thoả mãn quan hệ đệ quy trên bằng cách suy diễn quy nạp như sau. Với n = 1, 2, 3 ta có $T(1) = 1 = 2^1-1$, $T(2) = 2T(1) + 1 = 3 = 2^2 - 1$, $T(3) = 2T(2) + 1 = 7 = 2^3-1$. Bằng cách quy nạp, ta chứng minh được $T(n) = 2^n - 1$. Như vậy thời gian chạy của hàm HanoiTower là $O(2^n)$.

Một trường hợp hay gặp là: hàm đệ quy giải bài toán với cỡ dữ liệu vào n chứa một lời gọi hàm giải bài toán đó với cỡ dữ liệu vào n-1. Trường hợp này dẫn đến quan hệ đệ quy dạng:

$T(1) = a$

$T(n) = T(n-1) + g(n)$ với $n > 1$

Trong đó, a là một hằng số nào đó, còn g(n) là số phép toán sơ cấp cần thực hiện để đưa bài toán cỡ n về bài toán cỡ n - 1 và các phép toán sơ cấp cần thực hiện để nhận được nghiệm của bài toán cỡ n từ nghiệm của bài toán cỡ n-1.

Ta có thể giải quan hệ đệ quy trên bằng phương pháp thế lặp như sau:

$T(n) = T(n-1) + g(n)$

$= T(n-2) + g(n-1) + g(n)$

$= T(n-3) + g(n-2) + g(n-1) + g(n)$

....

$= T(1) + g(2) + g(3) +...+ g(n)$

$= a + g(2) + g(3) +...+ g(n)$

Đến đây ta chỉ cần đánh giá tổng $a + g(2) + g(3) +...+ g(n)$ bởi ký hiệu ô lớn.

Ví dụ 2 (Hàm tính giai thừa của số nguyên dương n).

```
int   Fact(int  n)
```

```
{
  if (n = = 1)
          return  1;
  else    return   n * Fact(n-1);
}
```

Giả sử thời gian chạy của hàm là T(n), với n = 1 ta có T(1) = O(1). Với n > 1, ta cần kiểm tra điều kiện của lệnh if-else và thực hiện phép nhân n với kết quả của lời gọi hàm, do đó T(n) = T(n-1) + O(1). Như vậy ta có quan hệ đệ quy sau:

$T(1) = O(1)$

$T(n) = T(n-1) + O(1)$ với $n > 1$

Thay các ký hiệu O(1) bởi các hằng số dương a và b tương ứng, ta có

$T(1) = a$

$T(n) = T(n-1) + b$ với $n > 1$

Sử dụng các phép thế T(n-1) = T(n-2) + b, T(n-2) = T(n-3) + b,..., ta có

$T(n) = T(n-1) + b$

$= T(n-2) + 2b$

$= T(n-3) + 3b$

...

$= T(1) + (n-1)b$

$= a + (n-1)b$

Từ đó, ta suy ra T(n) = O(n).

Kỹ thuật thế lặp còn có thể được sử dụng để giải một số dạng quan hệ đệ quy khác, chẳng hạn quan hệ đệ quy sau

$T(1) = a$

$T(n) = 2\ T(n/2) + g(n)$

Quan hệ đệ quy này được dẫn ra từ các thuật toán đệ quy được thiết kế theo ý tưởng: giải quyết bài toán cỡ n được quy về giải quyết hai bài toán con cỡ n/2. Ở đây g(n) là các tính toán để chuyển bài toán về hai bài toán con và các tính toán cần thiết khác để kết hợp nghiệm của hai bài toán con thành nghiệm của bài toán đã cho. Một ví dụ điển hình của các thuật toán được thiết kế theo cách này là thuật toán sắp xếp hoà nhập (MergeSort).

Chúng ta đã xem xét một vài dạng quan hệ đệ quy đơn giản. Thực tế, các hàm đệ quy có thể dẫn tới các quan hệ đệ quy phức tạp hơn nhiều; và có những quan hệ đệ quy rất đơn giản nhưng tìm ra nghiệm của nó cũng rất khó khăn. Chúng ta không đi sâu vào vấn đề này.

BÀI TẬP

1. Sử dụng định nghĩa ký hiệu ô lớn, chứng minh các khẳng định sau:
   a. $n^3 = O(0{,}001n^3)$
   b. $18n^4 - 3n^3 + 25n^2 - 17n + 5 = O(n^4)$
   c. $2^{n+10} = O(2^n)$
   d. $2^n + n^3 = O(2^n)$
   e. $n^{10} = O(3^n)$
   f. $\log_2 n = O(\sqrt{n})$

2. Chứng minh các khẳng định sau:
   a. $n^a = O(n^b)$ nếu $a \le b$.
   b. n^a không là $O(n^b)$ nếu $a > b$.
   c. $(\log n)^a = O(n^b)$ với a và b là các số dương.
   d. n^a không là $O((\log n)^b)$ với $a > b > 0$.

3. Cho a và b là các hằng số dương. Hãy chứng minh rằng $f(n) = O(\log_a n)$ nếu và chỉ nếu $f(n) = O(\log_b n)$. Do đó ta có thể bỏ qua cơ số khi viết O(logn).

4. Giả sử f(n) và g(n) là cận trện chặt của T(n). Hãy chỉ ra rằng, $f(n) = O(g(n))$ và $g(n) = O(f(n))$.

5. Hãy cho biết có bao nhiêu phép so sánh các dữ liệu trong mảng trong lệnh lặp sau:

```
for (g = 1; j < = n-1; j + +)
  {
     a = j + 1;
     do {  if (A[i] < A[j])
              swap (A[i], A[j]);
              i + +;
           }
      while (i <= n)
  } ;
```

6. Hãy tính số lần lặp các lệnh trong {…} trong lệnh sau:

```
for ( i = 0; i < n; i + +)
  for ( j = i + 1; i < = n; j + +)
    for ( k = 1; k < 10; k + +)
        {  các lệnh  };
```

7. Đánh giá thời gian chạy của các đoạn chương trình sau:

   a.
```
sum = 0;
for ( int i = 0; i < n; i + +)
  for ( int j = 0; j < n; j + +)
    sum + +;
```

   b.
```
sum = 0;
for ( int i = 0; i < n; i + +)
  for ( int j = 0; j < n*n; j + +)
    for ( int k = 0; k < j; k + +)
      sum + + ;
```

8. Đánh giá thời gian chạy của hàm đệ quy sau:

```
int     Bart(int n)
// n nguyên dương
{
  if ( n = = 1)
            return 1;
  else {
            result = 0;
            for ( int i = 2; i < = n; i + +)
                  result  + =  Bart(i – 1);
```

```
            return   result ;
        }
}
```

9. Chúng ta có thể tính ước chung lớn nhất của hai số nguyên dương bởi hàm đệ quy UCLN(n, m):

```
int    UCLN( int  n, int  m)
// n và m là nguyên dương và n > m
{
  if ( n % m = = 0)
          return   m;
  else {
          int   k = n % m ;
          return   UCLN(m, k);
        }
}
```

   Cỡ của dữ liệu vào trong hàm trên là n. Hãy đánh giá thời gian chạy của hàm đệ quy trên.

CHƯƠNG 16

CÁC CHIẾN LƯỢC THIẾT KẾ THUẬT TOÁN

Với một vấn đề đặt ra, làm thế nào chúng ta có thể đưa ra thuật toán giải quyết nó? Trong chương này, chúng ta sẽ trình bày các chiến lược thiết kế thuật toán, còn được gọi là các kỹ thuật thiết kế thuật toán. Mỗi chiến lược này có thể áp dụng để giải quyết một phạm vi khá rộng các bài toán. Mỗi chiến lược có các tính chất riêng và chỉ thích hợp cho một số dạng bài toán nào đó. Chúng ta sẽ lần lượt trình bày các chiến lược sau: chia-để-trị (divide-and-conquer), quy hoạch động (dynamic programming), quay lui (backtracking) và tham ăn (greedy method). Trong mỗi chiến lược chúng ta sẽ trình bày ý tưởng chung của phương pháp và sau đó đưa ra một số ví dụ minh họa.

Cần nhấn mạnh rằng, ta không thể áp dụng máy móc một chiến lược cho một vấn đề, mà ta phải phân tích kỹ vấn đề. Cấu trúc của vấn đề, các đặc điểm của vấn đề sẽ quyết định chiến lược có khả năng áp dụng.

16.1 CHIA - ĐỂ - TRỊ

16.1.1 Phương pháp chung

Chiến lược thiết kế thuật toán được sử dụng rộng rãi nhất là chiến lược chia-để-trị. Ý tưởng chung của kỹ thuật này là như sau: Chia vấn đề cần giải thành một số vấn đề con cùng dạng với vấn đề đã cho, chỉ khác là cỡ của chúng nhỏ hơn. Mỗi vấn đề con được giải quyết độc lập. Sau đó, ta kết hợp nghiệm của các vấn đề con để nhận được nghiệm của vấn đề đã cho. Nếu vấn đề con là đủ nhỏ có thể dễ dàng tính được nghiệm, thì ta giải quyết nó, nếu không vấn đề con được giải quyết bằng cách áp dụng đệ quy thủ tục trên (tức là lại tiếp tục chia nó thành các vấn đề con nhỏ hơn,…). Do đó, các

thuật toán được thiết kế bằng chiến lược chia-để-trị sẽ là các thuật toán đệ quy.

Sau đây là lược đồ của kỹ thuật chia-để-trị:

```
                DivideConquer (A,x)
                // tìm nghiệm x của bài toán A.
{
  if (A đủ nhỏ)
        Solve (A);
  else {
        Chia bài toán A thành các bài toán con
                A1, A2,…, Am;
        for (i = 1; i <= m ; i ++)
                DivideConquer (Ai , xi);
        Kết hợp các nghiệm xi của các bài toán con Ai (i=1, …, m) để
        nhận được nghiệm x của bài toán A;
        }
}
```

"Chia một bài toán thành các bài toán con" cần được hiểu là ta thực hiện các phép biến đổi, các tính toán cần thiết để đưa việc giải quyết bài toán đã cho về việc giải quyết các bài toán con cỡ nhỏ hơn.

Thuật toán tìm kiếm nhị phân (xem mục 4.4.2) là thuật toán được thiết kế dựa trên chiến lược chia-để-trị. Cho mảng A cỡ n được sắp xếp theo thứ tự tăng dần: $A[0] \leq \ldots \leq A[n-1]$. Với x cho trước, ta cần tìm xem x có chứa trong mảng A hay không, tức là có hay không chỉ số $0 \leq i \leq n-1$ sao cho $A[i] = x$. Kỹ thuật chia-để-trị gợi ý ta chia mảng A[0…n-1] thành 2 mảng con cỡ n/2 là A[0…k-1] và A[k+1…n-1], trong đó k là chỉ số đứng giữa mảng. So sánh x với A[k]. Nếu x = A[k] thì mảng A chứa x và i = k. Nếu không, do tính được sắp của mảng A, nếu x < A[k] ta tìm x trong mảng A[0…k-1], còn nếu x > A[k] ta tìm x trong mảng A[k+1…n-1].

Thuật toán Tháp Hà Nội (xem mục 15.5), thuật toán sắp xếp nhanh (QuickSort) và thuật toán sắp xếp hoà nhập (MergeSort) sẽ được trình bày

trong chương sau cũng là các thuật toán được thiết kế bởi kỹ thuật chia-để-trị. Sau đây chúng ta đưa ra một ví dụ đơn giản minh hoạ cho kỹ thuật chia-để-trị.

16.1.2 Tìm max và min

Cho mảng A cỡ n, chúng ta cần tìm giá trị lớn nhất (max) và nhỏ nhất (min) của mảng này. Bài toán đơn giản này có thể giải quyết bằng các thuật toán khác nhau.

Một thuật toán rất tự nhiên và đơn giản là như nhau. Đầu tiên ta lấy max, min là giá trị đầu tiên A[0] của mảng. Sau đó so sánh max, min với từng giá trị A[i], $1 \leq i \leq n-1$, và cập nhật max, min một cách thích ứng. Thuật toán này được mô tả bởi hàm sau:

```
SiMaxMin (A, max, min)
{
        max = min = A[0];
        for ( i = 1 ;  i < n , i ++)
                if (A[i] > max)
                        max = A[i];
                else if (A[i] < min)
                        min = A[i];
}
```

Thời gian thực hiện thuật toán này được quyết định bởi số phép so sánh x với các thành phần A[i]. Số lần lặp trong lệnh lặp for là n-1. Trong trường hợp xấu nhất (mảng A được sắp theo thứ tự giảm dần), mỗi lần lặp ta cần thực hiện 2 phép so sánh. Như vậy, trong trường hợp xấu nhất, ta cần thực hiện 2(n-1) phép so sánh, tức là thời gian chạy của thuật toán là O(n).

Bây giờ ta áp dụng kỹ thuật chia-để-trị để đưa ra một thuật toán khác. Ta chia mảng A[0..n-1] thành các mảng con A[0..k] và A[k+1..n-1] với k = [n/2]. Nếu tìm được max, min của các mảng con A[0..k] và A[k+1..n-1], ta dễ dàng xác định được max, min trên mảng A[0..n-1]. Để tìm max, min trên

mảng con ta tiếp tục chia đôi chúng. Quá trình sẽ dừng lại khi ta nhận được mảng con chỉ có một hoặc hai phần tử. Trong các trường hợp này ta xác định được dễ dàng max, min. Do đó, ta có thể đưa ra thuật toán sau:

```
                MaxMin (i, j, max, min)
// Biến max, min ghi lại giá trị lớn nhất, nhỏ nhất trong mảng A[i..j]
{
        if (i = = j)
                max = min = A[i];
        else if (i = = j-1)
                if (A[i] < A[j])
                        {
                                max = A[j];
                                min = A[i];
                        }
                else    {
                                max = A[i];
                                min = A[j];
                        }
        else {
                mid = (i+j) / 2;
                MaxMin (i, mid, max1, min1);
                MaxMin (mid + 1, j, max2, min2);
                if (max 1< max2)
                                max = max2;
                else            max = max1;
                if (min1 < min2)
                                min = min1;
                else            min = min2;
               }
        }
```

Bây giờ ta đánh giá thời gian chạy của thuật toán này. Gọi T(n) là số phép so sánh cần thực hiện. Không khó khăn thấy rằng, T(n) được xác định bởi quan hệ đệ quy sau.

T(1) = 0
T(2) = 1

$T(n) = 2T(n/2) + 2$ với $n > 2$

Áp dụng phương pháp thế lặp, ta tính được T(n) như sau:

$$\begin{aligned} T(n) &= 2\,T(n/2) + 2 \\ &= 2^2T(n/2^2) + 2^2 + 2 \\ &= 2^3T(n/2^3) + 2^3 + 2^2 + 2 \\ &\cdots\cdots \\ &= 2^kT(n/2^k) + 2^k + 2^{k-1} + \ldots + 2 \end{aligned}$$

Với k là số nguyên dương sao cho $2^k \le n < 2^{k+1}$, ta có

$$T(n) = 2^kT(1) + 2^{k+1} - 2 = 2^{k+1} - 2 \le 2(n-1)$$

Như vậy, T(n) = O(n).

16.2 THUẬT TOÁN ĐỆ QUY

Khi thiết kế thuật toán giải quyết một vấn đề bằng kỹ thuật chia-để-trị thì thuật toán thu được là thuật toán đệ quy. Thuật toán đệ quy được biểu diễn trong các ngôn ngữ lập trình bậc cao (chẳng hạn Pascal, C/C++) bởi các hàm đệ quy: đó là các hàm chứa các lời gọi hàm đến chính nó. Trong mục này chúng ta sẽ nêu lên các đặc điểm của thuật toán đệ quy và phân tích hiệu quả (về không gian và thời gian) của thuật toán đệ quy.

Đệ quy là một kỹ thuật đặc biệt quan trọng để giải quyết vấn đề. Có những vấn đề rất phức tạp, nhưng chúng ta có thể đưa ra thuật toán đệ quy rất đơn giản, sáng sủa và dễ hiểu. Cần phải hiểu rõ các đặc điểm của thuật toán đệ quy để có thể đưa ra các thuật toán đệ quy đúng đắn.

Giải thuật đệ quy cho một vấn đề cần phải thoả mãn các đòi hỏi sau:

1. Chứa lời giải cho các trường hợp đơn giản nhất của vấn đề. Các trường hợp này được gọi là các trường hợp cơ sở hay các trường hợp dừng.
2. Chứa các lời gọi đệ quy giải quyết các vấn đề con với cỡ nhỏ hơn.
3. Các lời gọi đệ quy sinh ra các lời gọi đệ quy khác và về tiềm năng các lời gọi đệ quy phải dẫn tới các trường hợp cơ sở.

Tính chất 3 là đặc biệt quan trọng, nếu không thoả mãn, hàm đệ quy sẽ chạy mãi không dừng. Ta xét hàm đệ quy tính giai thừa:

```
int     Fact(int  n)
{
        if (n = 0)
                return   1;
        else
                return   n * Fact(n-1);  // gọi đệ quy.
        }
```

Trong hàm đệ quy trên, trường hợp cơ sở là n = 0. Để tính Fact(n) cần thực hiện lời gọi Fact(n-1), lời gọi này lại dẫn đến lời gọi F(n-2),…, và cuối cùng dẫn tới lời gọi F(0), tức là dẫn tới trường hợp cơ sở.

Đệ quy và phép lặp. Đối với một vấn đề, có thể có hai cách giải: giải thuật đệ quy và giải thuật dùng phép lặp. Giải thuật đệ quy được mô tả bởi hàm đệ quy, còn giải thuật dùng phép lặp được mô tả bởi hàm chứa các lệnh lặp, để phân biệt với hàm đệ quy ta sẽ gọi là hàm lặp. Chẳng hạn, để tính giai thừa, ngoài hàm đệ quy ta có thể sử dụng hàm lặp sau:

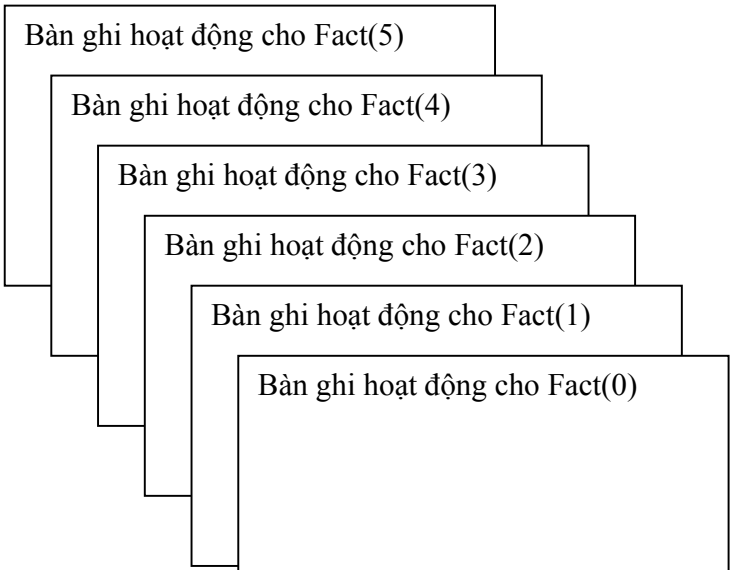
```
int     Fact(int n)
{
        if (n = = 0)
                return   1;
        else    {
                        int      F= 1;
                        for (int i = 1; i <= n ; i + +)
                                F = F * i;

                return   F;
                }
}
```

Ưu điểm nổi bật của đệ quy so với phép lặp là đệ quy cho phép ta đưa ra giải thuật rất đơn giản, dễ hiểu ngay cả đối với những vấn đề phức tạp.

Trong khi đó, nếu không sử dụng đệ quy mà dùng phép lặp thì thuật toán thu được thường là phức tạp hơn, khó hiểu hơn. Ta có thể thấy điều đó trong ví dụ tính giai thừa, hoặc các thuật toán tìm kiếm, xem, loại trên cây tìm kiếm nhị phân (xem mục 8.4). Tuy nhiên, trong nhiều trường hợp, các thuật toán lặp lại hiệu quả hơn thuật toán đệ quy.

Bây giờ chúng ta phân tích các nhân tố có thể làm cho thuật toán đệ quy kém hiệu quả. Trước hết, ta cần biết cơ chế máy tính thực hiện một lời gọi hàm. Khi gặp một lời gọi hàm, máy tính tạo ra một bản ghi hoạt động (activation record) ở ngăn xếp thời gian chạy (run-time stack) trong bộ nhớ của máy tính. Bản ghi hoạt động chứa vùng nhớ cấp cho các tham biến và các biến địa phương của hàm. Ngoài ra, nó còn chứa các thông tin để máy tính trở lại tiếp tục hiện chương trình đúng vị trí sau khi nó đã thực hiện xong lời gọi hàm. Khi hoàn thành thực hiện lời gọi hàm thì bản ghi hoạt động sẽ bị loại bỏ khỏi ngăn xếp thời gian chạy.

Khi thực hiện một hàm đệ quy, một dãy các lời gọi hàm được sinh ra. Hậu quả là một dãy bản ghi hoạt động được tạo ra trong ngăn xếp thời gian chạy. Cần chú ý rằng, một lời gọi hàm chỉ được thực hiện xong khi mà các lời gọi hàm mà nó sinh ra đã được thực hiện xong và do đó rất nhiều bản ghi hoạt động đồng thời tồn tại trong ngăn xếp thời gian chạy, chỉ khi một lời gọi hàm được thực hiện xong thì bản ghi hoạt động cấp cho nó mới được loại ngăn xếp thời gian chạy. Chẳng hạn, xét hàm đệ quy tính giai thừa, nếu thực hiện lời gọi hàm Fact(5) sẽ dẫn đến phải thực hiện các lời họi hàm Fact(4), Fact(3), Fact(2), Fact(1), Fact(0). Chỉ khi Fact(4) đã được tính thì Fact(5) mới được tính, … Do đó trong ngăn xếp thời gian chạy sẽ chứa các bản ghi hoạt động như sau:



Trong đó, bản ghi hoạt động cấp cho lời gọi hàm Fact(0) ở đỉnh ngăn xếp thời gian chạy. Khi thực hiện xong Fact(0) thì bản ghi hoạt động cấp cho nó bị loại, rồi bản ghi hoạt động cho Fact(1) bị loại,…

Vì vậy, việc thực hiện hàm đệ quy có thể đòi hỏi rất nhiều không gian nhớ trong ngăn xếp thời gian chạy, thậm chí có thể vượt quá khả năng của ngăn xếp thời gian chạy trong bộ nhớ của máy tính.

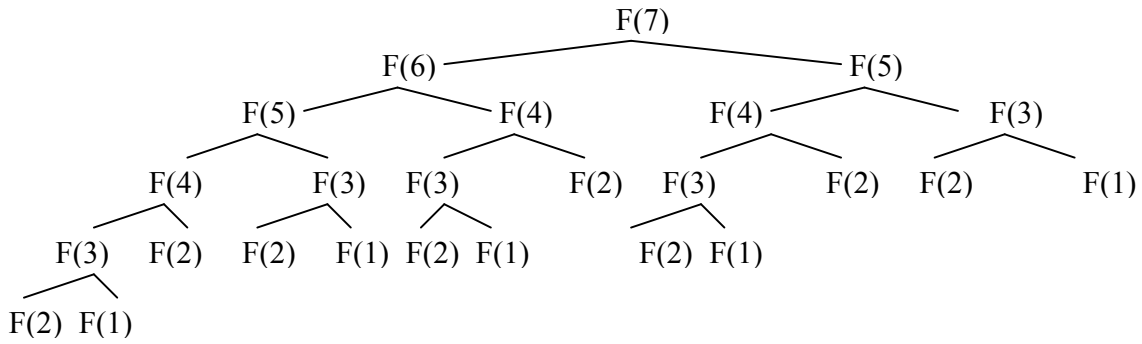
Một nhân tố khác làm cho các thuật toán đệ quy kém hiệu quả là các lời gọi đệ quy có thể dẫn đến phải tính nghiệm của cùng một bài toán con rất nhiều lần. Số Fibonacci thứ n, ký hiệu là F(n), được xác định đệ quy như sau:

F(1) = 1
F(2) = 1
F(n) = F(n-1) + F(n-2) với n>2

Do đó, ta có thể tính F(n) bởi hàm đệ quy sau.

```
int      Fibo(int n)
{
        if ((n = = 1) // (n = = 2))
                return   1;
        else
                return   Fibo (n-1) + Fibo(n-2);
}
```

Để tính F(7), các lời gọi trong hàm đệ quy Fibo dẫn ta đến phải tính các F(k) với k<7, như được biểu diễn bởi cây trong hình dưới đây; chẳng hạn để tính F(7) cần tính F(6) và F(5), để tính F(6) cần tính F(5) và F(4), …



Từ hình vẽ trên ta thấy rằng, để tính được F(7) ta phải tính F(5) 2 lần, tính F(4) 3 lần, tính F(3) 5 lần, tính F(2) 8 lần và tính F(1) 5 lần. Chính sự kiện để tính F(n) ta phải tính các F(k), với k<n, rất nhiều lần đã làm cho hàm đệ quy Fibo kém hiệu quả. Có thể đánh giá thời gian chạy của nó là $O(\phi^n)$, trong đó = $(1 + \sqrt{5})/2$.

Chúng ta có thể đưa ra thuật toán lặp để tính dãy số Fibonacci. Ý tưởng của thuật toán là ta tính lần lượt các F(1), F(2), F(3), …, F(n -2), F(n-1), F(n) và sử dụng hai biến để lưu lại hai giá trị vừa tính. Hàm lặp tính dãy số Fibonacci như sau:

```
int       Fibo1(int n)
{
        if ((n= = 1)//(n= = 2)
                return 1;
        else {
                int   previous = 1;
                int   current   = 1;
        for (int k = 3 ; k <=  n ; k ++)
                {
                        current + = previous;
                        previous = current – previous;
                }
        return   current;
        }
}
```

Dễ dàng thấy rằng, thời gian chạy của hàm lặp Fibo1 là O(n). Để tính F(50) thuật toán lặp Fibo1 cần 1 micro giây, thuật toán đệ quy Fibo đòi hỏi 20 ngày, còn để tính F(100) thuật toán lặp cần 1,5 micro giây, trong khi thuật toán đệ quy cần 10^9 năm!

Tuy nhiên, có rất nhiều thuật toán đệ quy cũng hiệu quả như thuật toán lặp, chẳng hạn các thuật toán đệ quy tìm, xem, loại trên cây tìm kiếm nhị phân (xem mục 8.4). Các thuật toán đệ quy: sắp xếp nhanh (QuickSort)

và sắp xếp hoà nhập (MergeSort) mà chúng ta sẽ nghiên cứu trong chương 17 cũng là các thuật toán rất hiệu quả.

Trong mục 6.6 chúng ta đã nghiên cứu kỹ thuật sử dụng ngăn xếp để chuyển thuật toán đệ quy thành thuật toán lặp. Nói chung, chỉ nên sử dụng thuật toán đệ quy khi mà không có thuật toán lặp hiệu quả hơn.

16.3 QUY HOẠCH ĐỘNG

16.3.1 Phương pháp chung

Kỹ thuật quy hoạch động giống kỹ thuật chia-để-trị ở chỗ cả hai đều giải quyết vấn đề bằng cách chia vấn đề thành các vấn đề con. Nhưng chia-để-trị là kỹ thuật top-down, nó tính nghiệm của các vấn đề con từ lớn tới nhỏ, nghiệm của các vấn đề con được tính độc lập bằng đệ quy. Đối lập, quy hoạch động là kỹ thuật bottom-up, tính nghiệm của các bài toán từ nhỏ đến lớn và ghi lại các kết quả đã tính được. Khi tính nghiệm của bài toán lớn thông qua nghiệm của các bài toán con, ta chỉ việc sử dụng các kết quả đã được ghi lại. Điều đó giúp ta tránh được phải tính nhiều lần nghiệm của cùng một bài toán con. Thuật toán được thiết kế bằng kỹ thuật quy hoạch động sẽ là thuật toán lặp, trong khi thuật toán được thiết kế bằng kỹ thuật chia-để-trị là thuật toán đệ quy. Để thuận tiện cho việc sử dụng lại nghiệm của các bài toán con, chúng ta lưu lại các nghiệm đã tính vào một bảng (thông thường là mảng 1 chiều hoặc 2 chiều).

Tóm lại, để giải một bài toán bằng quy hoạch động, chúng ta cần thực hiện các bước sau:

- Đưa ra cách tính nghiệm của các bài toán con đơn giản nhất.
- Tìm ra các công thức (hoặc các quy tắc) xây dựng nghiệm của bài toán thông qua nghiệm của các bài toán con.
- Thiết kế bảng để lưu nghiệm của các bài toán con.

- Tính nghiệm của các bài toán con từ nhỏ đến lớn và lưu vào bảng.
- Xây dựng nghiệm của bài toán từ bảng.

Một ví dụ đơn giản của thuật toán được thiết kế bằng quy hoạch động là thuật toán lặp tính dãy số Fibonacci mà ta đã đưa ra trong mục 16.2. Trong hàm lặp Fibo1, ta đã tính tuần tự F(1), F(2),…, đến F(n). Và bởi vì để tính F(k) chỉ cần biết F(k-1) và F(k-2), nên ta chỉ cần lưu lại F(k-1) và F(k-2).

Kỹ thuật quy hoạch động thường được áp dụng để giải quyết các **bài toán tối ưu** (optimization problems). Các bài toán tối ưu thường là có một số lớn nghiệm, mỗi nghiệm được gắn với một giá, và mục tiêu của chúng ta là tìm ra nghiệm có giá nhỏ nhất : **nghiệm tối ưu** (optimization solution). Chẳng hạn, bài toán tìm đường đi từ thành phố A đến thành phố B trong bản đồ giao thông, có nhiều đường đi từ A đến B, giá của một đường đi đó là độ dài của nó, nghiệm tối ưu là đường đi ngắn nhất từ A đến B. Nếu nghiệm tối ưu của bài toán được tạo thành từ nghiệm tối ưu của các bài toán con thì ta có thể sử dụng kỹ thuật quy hoạch động.

Sau đây, chúng ta sẽ đưa ra một số thuật toán được thiết kế bằng kỹ thuật quy hoạch động.

16.3.2 Bài toán sắp xếp các đồ vật vào ba lô

Giả sử ta có chiếc ba lô có thể chứa được một khối lượng w, chúng ta có n loại đồ vật được đánh số i,…, n. Mỗi đồ vật loại i (i = 1,…, n) có khối lượng a_i và có giá trị c_i. Chúng ta muốn sắp xếp các đồ vật vào ba lô để nhận được ba lô có gía trị lớn nhất có thể được. Giả sử mỗi loại đồ vật có đủ nhiều đề xếp vào ba lô.

Bài toán ba lô được mô tả chính xác như sau. Cho trước các số nguyên dương w, a_i, và c_i (i = 1,…,n). Chúng ta cần tìm các số nguyên không âm x_i (i = 1,…, n) sao cho

$$\sum_{n=1}^{n} x_i a_i \le w \quad \text{và}$$

$$\sum_{n=1}^{n} x_i c_i \quad \text{đạt giá trị lớn nhất.}$$

Xét trường hợp đơn giản nhất: chỉ có một loại đồ vật (n = 1). Trong trường hợp này ta tìm được ngay lời giải: xếp đồ vật vào ba lô cho tới khi nào không xếp được nữa thì thôi, tức là ta tìm được ngay nghiệm $x_i = w/a_i$.

Bây giờ ta đi tìm cách tính nghiệm của bài toán "xếp n loại đồ vật vào ba lô khối lượng w" thông qua nghiệm của các bài toán con "xếp k loại đồ vật ($1 \le k \le n$) vào ba lô khối lượng v ($1 \le v \le w$)" Ta gọi tắt là bài toán con (k,w), gọi cost (k,v) là giá trị lớn nhất của ba lô khối lượng v ($1 \le v \le w$) và chỉ chứa các loại đồ vật 1, 2,….,k. Ta tìm công thức tính cost (k,v).Với k = 1 và $1 \le v \le w$, ta có

$$x_i = v / a_i \quad \text{và}$$

$$\text{cost}(1,v) = x_i c_i \qquad (1)$$

Giả sử ta đã tính được cost (s,u) với $1 \le s < k$ và $1 \le u \le v$, ta cần tính cost (k,v) theo các cost (s,u) đã biết đó. Gọi $y_k = v / a_k$, ta có

$$\text{cost}(k,v) = \max[\text{cost}(k-1,u) + x_k c_k] \qquad (2)$$

Trong đó, max được lấy với tất cả $x_k = 0, 1,\ldots, y_k$ và $u = v - x_k a_k$ (tức là được lấy với tất cả các khả năng xếp đồ vật thứ k). Như vậy, tính cost (k,v) được quy về tính cost (k-1,u) với $u \le v$. Giá trị của x_k trong (2) mà cost (k-1,u) + $x_k c_k$ đạt max chính là số đồ vật loại k cần xếp. Giá trị lớn nhất của ba lô sẽ là cost(n, w).

Chúng ta sẽ tính nghiệm của bài toán từ cỡ nhỏ đến cỡ lớn theo các công thức (1) và (2). Nghiệm của các bài toán con sẽ được lưu trong mảng 2

chiều A[0..n-1][0..w-1], cần lưu ý là nghiệm của bài toán con (k,v) được lưu giữ trong A[k-1][v-1], vì các chỉ số của mảng được đánh số từ 0.

Mỗi thành phần A[k-1][v-1] sẽ chứa cost(k,v) và số đồ vật loại k cần xếp. Từ các công thức (1) và (2) ta có thể tính được các thành phần của mảng A lần lượt theo dòng 0, 1,…n-1.

Từ bảng A đã làm đầy, làm thế nào xác định được nghiệm của bài toán, tức là xác định được số đồ vật loại i (i = 1,2,…,n) cần xếp vào ba lô? Ô A[n-1][w-1] chứa giá trị lớn nhất của ba lô cost (n,w) và số đồ vật loại n cần xếp x_n. Tính $v = w – x_n a_n$. Tìm đến ô A[n-2][v-1] ta biết được cost(n-1,v) và số đồ vật loại n-1 cần xếp x_{n-1}. Tiếp tục quá trình trên, ta tìm được $x_{n-2},..,x_2$ và cuối cùng là x_1.

16.3.3 Tìm dãy con chung của hai dãy số

Xét bài toán sau: Cho hai dãy số nguyên $a = (a_1,…, a_m)$ và $b = (b_1,…b_n)$, ta cần tìm dãy số nguyên $c = (c_1,…, c_k)$ sao cho c là dãy con của cả a và b, và c là dài nhất có thể được. Ví dụ, nếu a = (3, 5, 1, 3, 5, 5, 3) và b = (1,5,3,5,3,1) thì dãy con chung dài nhất là c = (5,3,5,3) hoặc c = (1,3,5,3) hoặc c = (1,5,5,3).

Trường hợp đơn giản nhất khi một trong hai dãy a và b rỗng (m = 0 hoặc n = 0), ta thấy ngay dãy con chung dài nhất là dãy rỗng.

Ta xét các đoạt đầu của hai dãy a và b, đó là các dãy $(a_1,a_2,…,a_i)$ và $(b_1,b_2,…,a_j)$ với $0 \le i \le m$ và $0 \le j \le n$. Gọi L(i,j) là độ dài lớn nhất của dãy con chung của hai dãy $(a_1,a_2,…,a_i)$ và $(b_1,b_2,…,a_j)$. Do đó L(n,m) là độ dài lớn nhất của dãy con chung của a và b. Bây giờ ta đi tìm cách tính L(i,j) thông qua các L(s,t) với $0 \le s \le i$ và $0 \le t \le j$. Dễ dàng thấy rằng:

L(0,j) = 0 với mọi j

L(i,0) = 0 với mọi i (1)

Nếu i > 0 và j > 0 và $a_i \# b_j$ thì

L(i,j) = max [L(i,j-1), L(i-1,j)] (2)

Nếu i > 0 và j > 0 và $a_i = b_j$ thì

L(i,j) = 1 + L(i-1,j-1) (3)

Sử dụng các công thức đệ quy (1), (2), (3) để tính các L(i,j) lần lượt với i = 0,1,…,m và j = 0,1,…,n. Chúng ta sẽ lưu các giá trị L(i,j) vào mảng L[0..m][0..n].

Công việc tiếp theo là từ mảng L ta xây dựng dãy con chung dài nhất của a và b. Giả sử k = L[m][n] và dãy con chung dài nhất là c = (c_1,…c_{k-1}, c_k). Ta xác định các thành phần của dãy c lần lượt từ phải sang trái, tức là xác định c_k, rồi c_{k-1},…,c_1. Ta xem xét các thành phần của mảng L bắt từ L[m,n]. Giả sử ta đang ở ô L[i][j] và ta đang cần xác định c_r, (1 <= r <= k). Nếu $a_i = b_j$ thì theo (3) ta lấy $c_r = a_i$, giảm r đi 1 và đi đến ô L[i-1][j-1]. Còn nếu $a_i \# b_j$ thì theo (2) hoặc L[i][j] = L[i][j-1], hoặc L[i][j] = L[i-1][j]. Trong trường hợp L[i][j] = L[i][j-1] ta đi tới ô L[i][j-1], còn nếu L[i][j] = L[i-1][j] ta đi tới ô L[i-1][j]. Tiếp tục quá trình trên ta xác định được tất cả các thành phần của dãy con dài nhất.

16.4 QUAY LUI

16.4.1 Tìm kiếm vét cạn

Trong thực tế chúng ta thường gặp các câu hỏi chẳng hạn như "có bao nhiêu khả năng...?", "hãy cho biết tất cả các khả năng...?", hoặc "có tồn tại hay không một khả năng...?". Ví dụ, có hay không một cách đặt 8 con hậu vào bàn cờ sao cho chúng không tấn công nhau. Các vấn đề như thế thông thường đòi hỏi ta phải xem xét tất cả các khả năng có thể có. Tìm **kiếm vét cạn** (exhaustive search) là xem xét tất cả các ứng cử viên nhằm phát hiện ra đối tượng mong muốn. Các thuật toán được thiết kế bằng tìm kiếm vét cạn thường được gọi là **brute-force algorithms**. Ý tưởng của các thuật toán này là sinh-kiểm, tức là sinh ra tất cả các khả năng có thể có và kiểm tra mỗi khả năng xem nó có thoả mãn các điều kiện của bài toán không. Trong nhiều vấn đề, tất cả các khả năng mà ta cần xem xét có thể quy về các đối tượng tổ hợp

(các tập con của một tập), hoặc các hoán vị của n đối tượng, hoặc các tổ hợp k đối tượng từ n đối tượng. Trong các trường hợp như thế, ta cần phải sinh ra, chẳng hạn, tất cả các hoán vị, rồi kiểm tra xem mỗi hoán vị có là nghiệm của bài toán không. Tìm kiếm vét cạn đương nhiên là kém hiệu quả, đòi hỏi rất nhiều thời gian. Nhưng cũng có vấn đề ta không có cách giải quyết nào khác tìm kiếm vét cạn.

Ví dụ 1(Bài toán 8 con hậu). Chúng ta cần đặt 8 con hậu vào bàn cờ 8x8 sao cho chúng không tấn công nhau, tức là không có hai con hậu nào nằm cùng hàng, hoặc cùng cột, hoặc cùng đường chéo.

Vì các con hậu phải nằm trên các hàng khác nhau, ta có thể đánh số các con hậu từ 1 đến 8, con hậu i là con hậu đứng ở hàng thứ i (i=1,...,8). Gọi x_i là cột mà con hậu thứ i đứng. Vì các con hậu phải đứng ở các cột khác nhau, nên (x_1, x_2, ...,x_8) là một hoán vị của 8 số 1, 2,..., 8. Như vậy tất cả các ứng cử viên cho nghiệm của bài toán 8 con hậu là tất cả các hoán vị của 8 số 1, 2,..., 8. Đến đây ta có thể đưa ra thuật toán như sau: sinh ra tất cả các hoán vị của (x_1, x_2, ...,x_8), với mỗi hoán vị ta kiểm tra xem hai ô bất kì (i,x_i) và (j,x_j) có cùng đường chéo hay không.

Đối với bài toán tổng quát: đặt n con hậu vào bàn cờ nxn, số các hoán vị cần xem xét là n!, và do dó thuật toán đặt n con hậu bằng tìm kiếm vét cạn đòi hỏi thời gian O(n!). Trong mục sau, chúng ta sẽ đưa ra thuật toán hiệu quả hơn được thiết kế bằng kỹ thuật quay lui.

Ví dụ 2(Bài toán người bán hàng).

Bài toán người bán hàng (saleperson problem) được phát biểu như sau. Một người bán hàng, hàng ngày phải đi giao hàng từ một thành phố đến một số thành phố khác rồi quay lại thành phố xuất phát. Anh ta muốn tìm một tua qua mỗi thành phố cần đến đúng một lần với độ dài của tua là ngắn nhất có thể được. Chúng ta phát biểu chính xác bài toán như sau. Cho đồ thị định hướng gồm n đỉnh được đánh số 0,1,...,n-1. Độ dài của cung (i,j) được kí hiệu là d_{ij} và là một số không âm. Nếu đồ thị không có cung (i,j) thì ta

xem $d_{ij} = +\infty$. Chúng ta cần tìm một đường đi xuất phát từ một đỉnh qua tất cả các đỉnh khác của đồ thị đúng một lần rồi lại trở về đỉnh xuất phát (tức là tìm một chu trình Hamilton) sao cho độ dài của tua là nhỏ nhất có thể được. Mỗi tua như tế là một dãy các đỉnh (a_0, a_1,..., a_{n-1}, a_0), trong đó các a_0, a_1,..., a_{n-1} là khác nhau. Không mất tính tổng quat, ta có thể xem đỉnh xuất phát là đỉnh 0, $a_0 = 0$. Như vậy, mỗi tua tương ứng với một hoán vị (a_1,..., a_{n-1}) của các đỉnh 1, 2, ..., n-1. Từ đó ta có thuật toán sau: sinh ra tất cả các hoán vị của n-1 đỉnh 1, 2, ..., n-1; với mỗi hoán vị ta tính độ dài của tua tương ứng với hoán vị đó và so sánh các độ dài ta sẽ tìm được tua ngắn nhất. Lưu ý rằng, có tất cả (n-1)! hoán vị và mỗi tua cần n phép toán để tính độ dài, do đó thuật toán giải bài toán người bán hàng với n thành phố bằng tìm kiếm vét cạn cần thời gian O(n!).

Bài toán người bán hàng là bài toán kinh điển và nổi tiếng. Ngoài cách giải bằng tìm kiếm vét cạn, người ta đã đưa ra nhiều thuật toán khác cho bài toán này. Thuật toán quy hoạch động cho bài toán người bán hàng đòi hỏi thời gian $O(n^2 2^n)$. Cho tới nay người ta vẫn chưa tìm ra thuật toán có thời gian đa thức cho bài toán người bán hàng.

16.4.2 Quay lui

Quay lui (backtracking) là kỹ thuật thiết kế thuật toán có thể sử dụng để giải quyết rất nhiều vấn đề khác nhau. Ưu điểm của quay lui so với tìm kiếm vét cạn là ở chỗ có thể cho phép ta hạn chế các khả năng cần xem xét.

Trong nhiều vấn đề, việc tìm nghiệm của vấn đề được quy về tìm một dãy các trạng thái (a_1, a_2,…, a_k,…), trong đó mỗi a_i (i = 1,2,…) là một trạng thái được chọn ra từ một tập hữu hạn A_i các trạng thái, thoả mãn các điều kiện nào đó. Tìm kiếm vét cạn đòi hỏi ta phải xem xét tất cả các dãy trạng thái đó để tìm ra dãy trạng thái thoả mãn các yêu cầu của bài toán.

Chúng ta sẽ gọi dãy các trạng thái (a_1, a_2,…, a_n) thoả mãn các yêu cầu của bài toán là vectơ nghiệm. Ý tưởng của kỹ thuật quay lui là ta xây dựng

vectơ nghiệm xuất phát từ vectơ rỗng, mỗi bước ta bổ xung thêm một thành phần của vectơ nghiệm, lần lượt $a_1,a_2,\ldots$

Đầu tiên, tập S_1 các ứng cử viên có thể là thành phần đầu tiên của vectơ nghiệm chính là A_1.

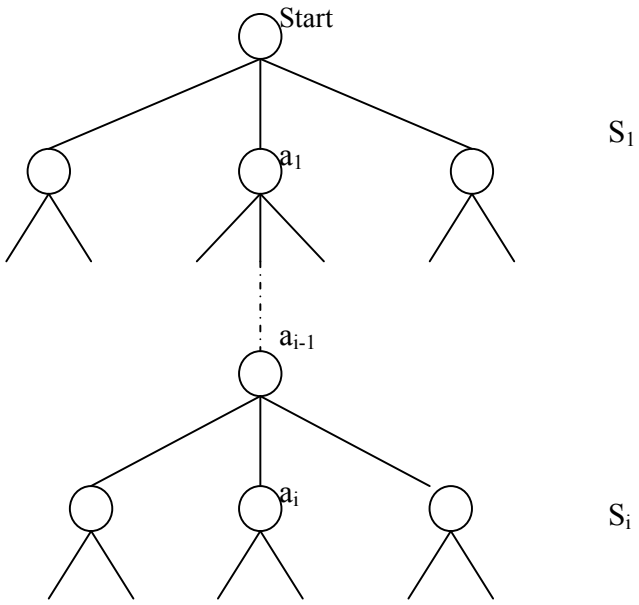
Chọn $a_1 \in S_1$, ta có vectơ (a_1). Giả sử sau bước thứ i-1, ta đã tìm được vectơ $(a_1,a_2,\ldots,a_{i-1})$. Ta sẽ gọi các vectơ như thế là nghiệm một phần (nó thoả mãn các đòi hỏi của bài toán, những chưa "đầy đủ"). Bây giờ ta mở rộng nghiệm một phần $(a_1,a_2,\ldots,a_{i-1})$ bằng cách bổ xung thêm thành phần thứ i. Muốn vậy, ta cần xác định tập S_i các ứng cử viên cho thành phần thứ i của vectơ nghiệm. Cần lưu ý rằng, tập S_i được xác định theo các yêu cầu của bài toán và các thành phần $a_1,a_2,\ldots,a_{i-1}$ đã chọn trước, và do đó S_i là tập con của tập A_i các trạng thái. Có hai khả năng

- Nếu S_i không rỗng, ta chọn $a_i \in S_i$ và thu được nghiệm một phần $(a_1,a_2,\ldots,a_{i-1},a_i)$, đồng thời loại a_i đã chọn khỏi S_i. Sau đó ta lại tiếp tục mở rộng nghiệm một phần $(a_1,a_2,\ldots,a_i)$ bằng cách áp dụng đệ quy thủ tục mở rộng nghiệm.
- Nếu S_i rỗng, điều này có nghĩa là ta không thể mở rộng nghiệm một phần $(a_1,a_2,\ldots,a_{i-2},a_{i-1})$, thì ta quay lại chọn phần tử mới a'_{i-1} trong S_{i-1} làm thành phần thứ i-1 của vectơ nghiệm. Nếu thành công (khi S_{i-1} không rỗng) ta nhận được vectơ $(a_1,a_2,\ldots,a_{i-2},a'_{i-1})$ rồi tiếp tục mở rộng nghiệm một phần này. Nếu không chọn được a'_{i-1} thì ta quay lui tiếp để chọn $a'_{i-2}\ldots$ Khi quay lui để chọn a'_1 mà S_1 đã trở thành rỗng thì thuật toán dừng.

Trong quá trình mở rộng nghiệm một phần, ta cần kiểm tra xem nó có là nghiệm không. Nếu là nghiệm, ta ghi lại hoặc in ra nghiệm này. Kỹ thuật quay lui cho phép ta tìm ra tất cả các nghiệm của bài toán.

Kỹ thuật quay lui mà ta đã trình bày thực chất là kỹ thuật đi qua cây tìm kiếm theo độ sâu (đi qua cây theo thứ tự preorder). Cây tìm kiếm được xây dựng như sau

- Các đỉnh con của gốc là các trạng thái của S_1
- Giả sử a_{i-1} là một đỉnh ở mức thứ i-1 của cây. Khi đó các đỉnh con của a_{i-1} sẽ là các trạng thái thuộc tập ứng cử viên S_i. Cây tìm kiếm được thể hiện trong hình 16.1.



Hình 16.1. Cây tìm kiếm vectơ nghiệm

Trong cây tìm kiếm, mỗi đường đi từ gốc tới một đỉnh tương ứng với một nghiệm một phần.

Khi áp dụng kỹ thuật quay lui để giải quyết một vấn đề, thuật toán được thiết kế có thể là đệ quy hoặc lặp. Sau đây ta sẽ đưa ra lược đồ tổng quát của thuật toán quay lui.

Lược đồ thuật toán quay lui đệ quy. Giả sử vector là nghiệm một phần $(a_1,a_2,\ldots,a_{i-1})$. Hàm đệ quy chọn thành phần thứ i của vector nghiệm là như sau:

Backtrack(vector , i)

```
// Chọn thành phần thứ i của vector.
{
        if (vector là nghiệm)
                viết ra nghiệm;
        Tính Si;
        for (mỗi ai∈Si)
                Backtrack(vector + (ai) , i+1);
}
```

Trong hàm trên, nếu vector là nghiệm một phần $(a_1,\ldots,a_{i-1})$ thì vector + (a_i) là nghiệm một phần $(a_1,a_2,\ldots,a_{i-1},a_i)$. Để tìm ra tất cả các nghiệm, ta chỉ cần gọi Backtrack(vector,1), với vector là vector rỗng.

Lược đồ thuật toán quay lui không đệ quy

```
Backtrack
{
   k = 1;
   Tính   S1;
   while (k>0)
     {
        if (Sk không rỗng)
           {
                chọn ak ∈ Sk;
                Loại ak khỏi Sk;
                if ((a1,…,ak) là nghiệm)
                        viết ra nghiệm;
                k++;
                Tính   Sk;
           }
        else   k-- ;    //Quay lui
      }
}
```

Chú ý rằng, khi cài đặt thuật toán theo lược đồ không đệ quy, chúng ta cần biết cách lưu lại vết của các tập ứng viên S_1, S_2,…,S_k để khi quay lui ta có thể chọn được thành phần mới cho vectơ nghiệm.

Ví dụ 3. Thuật toán quay lui cho bài toán 8 con hậu. Hình 16.2. mô tả một nghiệm của bài toán 8 con hậu.

	0	1	2	3	4	5	6	7
0	x							
1							x	
2					x			
3								x
4		x						
5				x				
6						x		
7			x					

Hình 16.2. Một nghiệm của bài toán 8 con hậu

Như trong ví dụ 1, ta gọi cột của con hậu ở dòng i (i = 0,1,..,7) là x_i. Nghiệm của bài toán là vectơ $(x_0,x_{1,...,}x_7)$, chẳng hạn nghiệm trong hình 16.2 là (0,6,4,7,1,3,5,2). Con hậu 0 (ở dòng 0) có thể được đặt ở một trong tám cột. Do đó S_0={0,1,…,7}. Khi ta đã đặt con hậu 0 ở cột 0 (x_0=0), con hậu 1 ở cột 6 (x_1=6), như trong hình 16.2, thì con hậu 2 chỉ có thể đặt ở một trong các cột 1,3,4. Tổng quát, khi ta đã đặt các con hậu 0,1,2,…,k-1 thì con hậu k (con hậu ở dòng k) chỉ có thể đặt ở một trong các cột khác với các cột mà các con hậu 0,1,2,…,k-1 đã chiếm và không cùng đường chéo với chúng. Điều đó có nghiã là khi đã chọn được nghiệm một phần $(x_0,x_1,…,x_{k-1})$ thì x_k chỉ có thể lấy trong tập ứng viên S_k được xác định như sau

$$S_k = \{x_k \in \{0,1,…,7\} \mid x_k \neq x_i \text{ và } | i-k | \neq | x_k-x_i | \text{ với mọi } i < k\}$$

Từ đó ta có thể đưa ra thuật toán sau đây cho bài toán 8 hậu

```
void      Queen(int   x[8])
{
   int   k = 0;
   x[0] = -1;
   while (k>0)
     {
```

```
        x[k]++;
        if (x[k]<=7)
          {
              int i;
              for (i = 0 ; i < k ; i++)
              if ((x[k] == x[i]) | | (fabs(i-k) == fabs(x[k] - x[i])))
                    break;     // kiểm tra xem x[k] có thuộc Sk
              if (i == k)        // chỉ khi x[k] ∈Sk
                  if (k == 7)
                        viết ra mảng x;
                  else
                      {
                         k++;
                         x[k] = -1;
                       }
          }
         else    k--;     //quay lui
    }     // Hết vòng lặp while
}
```

Ví dụ 4. Các dãy con có tổng cho trước

Cho một dãy số nguyên dương ($a_0,a_1,\ldots,a_{n-1}$) và một số nguyên dương M. Ta cần tìm các dãy con của dãy sao cho tổng của các phần tử trong dãy con đó bằng M. Chẳng hạn, với dãy số (7,1,4,3,5,6) và M=11, thì các dãy con cần tìm là (7,1,3), (7,4), (1,4,6) và (5,6).

Sử dụng kỹ thuật quay lui, ta xác định dãy con ($a_{i0},a_{i1},\ldots,a_{ik}$) sao cho $a_{i0}+a_{i1}+\ldots+a_{ik} = M$ bằng cách chọn lần lượt $a_{i0},a_{i1},\ldots$Ta có thể chọn a_{i0} là một trong $a_0,a_1,\ldots,a_{n-1}$ mà nó <= M, tức là có thể chọn a_{i0} với i_0 thuộc tập ứng viên $S_0 = \{i \in \{0,1,\ldots,n-1\} \mid a_i <= M\}$. Khi đã chọn được ($a_{i0},a_{i1},\ldots,a_{ik-1}$) với $S = a_{i0} + a_{i1} + \ldots + a_{ik-1} < M$ thì ta có thể chọn a_{ik} với i_k là một trong các chỉ số bắt đầu từ $i_{k-1}+1$ tới n-1 và sao cho $S+a_{ik} <= M$. Tức là, ta có thể chọn a_{ik} với i_k thuộc tập $S_k = \{i \in \{i_{k-1}+1,\ldots, n-1\} \mid S+a_i <= M\}$. Giả sử dãy số đã cho được lưu trong mảng A. Lưu dãy chỉ số $\{i_0,i_1,\ldots,i_k\}$ của dãy con cần tìm vào mảng I, ta có thuật toán sau

```
void    SubSequences(int A[n], int M, int I[n])
{
        k  =  0;
        I[0]  =  -1;
        int   S  =  0;
        while (k > 0)
        {
            I[k]++;
            If (I[k] < n)
                {
                  if (S + A[i[k]] <= M)
                        if (S + A[i[k]] == M)
                                viết ra mảng I[0..k];
                        else
                        {
                                S  =  S + A[i[k]];
                                I[k+1]  =  I[k];
                                k++;
                        }

                }
             else
                  {
                        k --;
                        S = S - A[i[k]];
                  }
        }
}
```

16.4.3 Kỹ thuật quay lui để giải bài toán tối ưu

Trong mục này chúng ta sẽ áp dụng kỹ thuật quay lui để tìm nghiệm của bài toán tối ưu.

Giả sử nghiệm của bài toán có thể biểu diễn dưới dạng $(a_1,..,a_n)$, trong đó mỗi thành phần a_i (i = 1,…,n) được chọn ra từ tập S_i các ứng viên. Mỗi nghiệm $(a_1,..,a_n)$ của bài toán có một giá cost$(a_1,..,a_n)$ >= 0, và ta cần tìm nghiệm có giá thấp nhất (nghiệm tối ưu).

Giả sử rằng, giá của các nghiệm một phần là không giảm, tức là nếu $(a_1,..,a_{k-1})$ là nghiệm một phần và $(a_1,..,a_{k-1},a_k)$ là nghiệm mở rộng của nó thì

$cost(a_1,..,a_{k-1}) <= cost(a_1,..,a_{k-1},a_k)$

Trong quá trình mở rộng nghiệm một phần (bằng kỹ thuật quay lui), khi tìm được nghiệm một phần $(a_1,..,a_k)$, nếu biết rằng tất cả các nghiệm mở rộng của nó $(a_1,..,a_k,a_{k+1},...)$ đều có giá lớn hơn giá của nghiệm tốt nhất đã biết ở thời điểm đó, thì ta không cần mở rộng nghiệm một phần $(a_1,..,a_k)$ đó.

Giả sử $cost^*(a_1,..,a_k)$ là cận dưới của giá của tất cả các nghiệm $(a_1,..,a_k,a_{k+1},...)$ mà nó là mở rộng của nghiệm một phần $(a_1,..,a_k)$. Giả sử giá của nghiệm tốt nhất mà ta đã tìm ra trong quá trình tìm kiếm là lowcost. (Ban đầu lowcost được khởi tạo là $+\infty$ và giá trị của nó được cập nhật trong quá trình tìm kiếm). Khi ta đạt tới nghiệm một phần $(a_1,..,a_k)$ mà $cost^*(a_1,..,a_k) > lowcost$ thì ta không cần mở rộng nghiệm một phần $(a_1,..,a_k)$ nữa; điều đó có nghĩa là, trong cây tìm kiếm hình 16.1 ta cắt bỏ đi tất cả các nhánh từ đỉnh a_k.

Từ các điều trình bày trên, ta đưa ra lược đồ thuật toán tìm nghiệm tối ưu sau. Thuật toán này thường được gọi là **thuật toán nhánh–và--cận** (branch – and – bound).

```
                    BranchBound
{
   lowcost  =  +∞;
   cost*  =  0;
   k  =  1;
   tính  S1;
   while (k > 0)
     {
        if (Sk không rỗng và cost* <= lowcost)
        {
              chọn ak ∈ Sk;
              Loại ak ra khỏi Sk;
              cost* = cost*(a1,..,ak);
              if ((a1,..,ak) là nghiệm)
```

```
                    if (cost(a1,..,ak) <  lowcost)
                            lowcost  = cost(a1,..,ak);
                k++;
                tính   Sk;
            }
        else
            {
                k--;
                cost* = cost(a1,..,ak);
            }
        }
    }
```

Ưu điểm của thuật toán nhánh – và - cận là ở chỗ nó cho phép ta không cần phải xem xét tất cả các nghiệm vẫn có thể tìm được nghiệm tối ưu. Cái khó nhất trong việc áp dụng kỹ thuật nhánh và cận là xây dựng hàm đánh giá cận dưới cost* của các nghiệm là mở rộng của nghiệm một phần. Đánh giá cận dưới có chặt mới giúp ta cắt bỏ được nhiều nhánh không cần thiết phải xem xét tiếp, và do đó thuật toán nhận được mới nhanh hơn đáng kể so với thuật toán tìm kiếm vét cạn.

16.5 CHIẾN LƯỢC THAM ĂN

16.5.1 Phương pháp chung

Các bài toán tối ưu thường là có một số rất lớn nghiệm, việc tìm ra nghiệm tối ưu (nghiệm có giá thấp nhất) đòi hỏi rất nhiều thời gian. Điển hình là bài toán người bán hàng, thuật toán quy hoạch động cũng đòi hỏi thời gian $O(n^2 2^n)$, và cho tới nay người ta vẫn chưa tìm ra thuật toán có thời gian đa thức cho bài toán này.

Một cách tiếp cận khác để giải quyết các bài toán tối ưu là chiến lược tham ăn (greedy strategy).

Trong hầu hết các bài toán tối ưu, để nhận được nghiệm tối ưu chúng ta có thể đưa về sự thực hiện một dãy quyết định. Ý tưởng của chiến lược

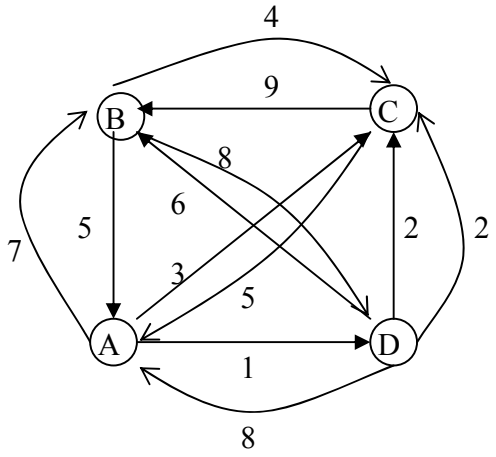
tham ăn là, tại mỗi bước ta sẽ lựa chọn quyết định để thực hiện là quyết định được xem là tốt nhất trong ngữ cảnh nào đó được xác định bởi bài toán. Tức là, quyết định được lựa chọn ở mỗi bước là quyết định tối ưu địa phương. Tùy theo từng bài toán mà ta đưa ra tiêu chuẩn lựa chọn quyết định cho thích hợp.

Các thuật toán tham ăn (greedy algorithm) nói chung là đơn giản và hiệu quả (vì các tính toán để tìm ra quyết định tối ưu địa phương thường là đơn giản). Tuy nhiên, các thuật toán tham ăn có thể không tìm được nghiệm tối ưu, nói chung nó chỉ cho ra nghiệm gần tối ưu, nghiệm tương đối tốt. Nhưng cũng có nhiều thuật toán được thiết kế theo kỹ thuật tham ăn cho ta nghiệm tối ưu, chẳng hạn thuật toán Dijkstra tìm đường đi ngắn nhất từ một đỉnh tới các đỉnh còn lại trong đồ thị định hướng, các thuật toán Prim và Kruskal tìm cây bao chùm ngắn nhất trong đồ thị vô hướng, chúng ta sẽ trình bày các thuật toán này trong chương 18.

16.5.2 Thuật toán tham ăn cho bài toán người bán hàng

Giả sử đồ thị mà ta xét là đồ thị định hướng n đỉnh được đánh số 0,1,2,…,n-1, và là đồ thị đầy đủ, tức là với mọi $0 \le i , j \le$ n-1 đều có cung đi từ i đến j với độ dài là số thực không âm d(i,j). Giả sử đỉnh xuất phát là đỉnh 0, và đường đi ngắn nhất mà ta cần tìm là (0, a_1, a_1,… a_{n-1}, 0) trong đó $a_k \in \{1,2,\ldots,n-1\}$. Để nhận được nghiệm tối ưu trên, tại mỗi bước k (k = 1,…, n-1) chúng ta cần chọn một đỉnh a_k để đi tới trong số các đỉnh chưa thăm (tức là $a_k \neq a_i$, i = 1,…, k-1). Với mong muốn đường đi nhận được là ngắn nhất, ta đưa ra tiêu chuẩn chọn đỉnh a_k ở mỗi bước là đỉnh gần nhất trong số các đỉnh chưa thăm.

Ví dụ. Xét đồ thị định hướng trong hình 16.3



Hình 16.3. Một đồ thị định hướng

Giả sử ta cần tìm tua ngắn nhất xuất phát từ A. Vì d(A,B) = 7, d(A,C) = 3 và d(A,D) = 1, nên ta chọn đỉnh D để đi tới, ta có đường đi (A, D). Từ D, các đỉnh chưa thăm là B và C, ta chọn C để đi tới vì C gần D hơn là B. Ta thu được đường đi (A, D, C). Từ C ta chỉ có một khả năng là đi tới B. Do đó ta nhận được tua (A, D, C, B, A). Độ dài của nó là 1 + 2 + 9 + 5 = 17. Đây không phải là đường đi ngắn nhất, vì đường đi ngắn nhất là (A, C, D, B, A) có độ dài 3 + 2 + 6 + 5 = 16.

16.5.3 Thuật toán tham ăn cho bài toán ba lô

Chúng ta trở lại bài toán ba lô đã đưa ra trong mục 16.3.2. Chúng ta cần nhận được chiếc ba lô chứa đồ vật có giá trị lớn nhất. Một cách tiếp cận khác để có chiếc ba lô đó là mỗi bước xếp một loại đồ vật vào ba lô. Vấn đề đặt ra là tại bước k (k = 1,2,…) ta cần chọn loại đồ vật nào để xếp và xếp bao nhiêu đồ vật loại đó. Từ các đòi hỏi của bài toán, ta đưa ra tiêu chuẩn chọn như sau: tại mỗi bước ta sẽ chọn loại đồ vật có giá trị lớn nhất trên một đơn vị khối lượng (gọi tắt là tỷ giá) trong số các loại đồ vật chưa được xếp vào ba lô. Khi đã chọn một loại đồ vật thì ta xếp tối đa có thể được.

Ví dụ, giả sử ta có ba lô chứa được khối lượng w = 20. Chúng ta có 4 loại đồ vật có khối lượng a_i và giá trị c_i (i = 1,2,3,4) được cho trong bảng sau:

Loại	1	2	3	4
Khối lượng a_i	5	7	8	3
Giá trị c_i	21	42	20	9
Tỷ giá c_i/a_i	4,2	6	2,5	3

Đầu tiên trong 4 loại đồ vật thì loại có tỷ giá lớn nhất là loại 2. Ta có thể xếp được tối đa 2 đồ vật loại 2 vào ba lô, và khối lượng còn lại của ba lô là 20 – 2 . 7 = 6. Đến đây số loại đồ vật chưa xếp: 1 , 3 , 4 , loại có tỷ giá lớn nhất là loại 1. Khối lượng còn lại của ba lô là 6, nên chỉ xếp được 1 đồ vật loại 1. Bước tiếp theo ta chọn loại 4, nhưng khối lượng còn lại của ba lô là 6-1.5 = 1, nên không xếp được đồ vật loại 4 (vì đồ vật loại 4 có khối lượng 3 > 1). Chọn đồ vật loại 3, cũng không xếp được, và dừng lại. Như vậy ta thu được ba lô chứa 2 đồ vật loại 2 và 1 đồ vật loại 1, với giá trị là 2.42 + 1.21 = 105.

Thuật toán tham ăn xếp các đồ vật vào ba lô như đã trình bày cũng không tìm ra nghiệm tối ưu, mà chỉ cho ra nghiệm tốt. Thế thì tại sao ta lại cần đến các thuật toán tham ăn mà nó chỉ cho ra nghiệm tốt, gần đúng với nghiệm tối ưu. Vấn đề là ở chỗ, đối với nhiều bài toán, chẳng hạn bài toán người bán hàng, bài toán sơn đồ thị,…, các thuật toán tìm ra nghiệm chính xác đòi hỏi thời gian mũ, không sử dụng được trong thực tế khi cỡ bài toán khá lớn. Còn có những bài toán để tìm ra nghiệm tối ưu ta chỉ còn có cách là tìm kiếm vét cạn. Trong các trường hợp như thế, sử dụng các thuật toán tham ăn là cần thiết, bởi vì các thuật toán tham ăn thường là đơn giản, rất hiệu quả, và thực tế nhiều khi có được một nghiệm tốt cũng là đủ.

16.6 THUẬT TOÁN NGẪU NHIÊN

Khi trong một bước nào đó của thuật toán, ta cần phải lựa chọn một trong nhiều khả năng, thay vì phải tiêu tốn thời gian xem xét tất cả các khả năng để có sự lựa chọn tối ưu, người ta có thể chọn ngẫu nhiên một khả năng. Sự lựa chọn ngẫu nhiên lại càng thích hợp cho các trường hợp khi mà hầu hết các khả năng đều "tốt" ngang nhau. Các thuật toán chứa sự lựa chọn ngẫu nhiên được gọi là các thuật toán ngẫu nhiên (randomized algorithm hay probabilistic algorithm). Đặc trưng của thuật toán ngẫu nhiên là, kết quả của thuật toán không chỉ phụ thuộc vào giá trị đầu vào của thuật toán mà còn phụ thuộc vào giá trị ngẫu nhiên được sinh ra bởi hàm sinh số ngẫu nhiên. Nếu ta cho chạy thuật toán ngẫu nhiên hai lần trên cùng một dữ liệu vào, thuật toán có thể cho ra kết quả khác nhau.

Trong các thuật toán ngẫu nhiên, ta cần sử dụng các hàm sinh số ngẫu nhiên (random number generator). Trong các thuật toán ngẫu nhiên sẽ đưa ra sau này, ta giả sử đã có sẵn các hàm sinh số ngẫu nhiên sau. Hàm RandInt(i,j), trong đó i, j là các số nguyên và 0 <= i <= j, trả về một số nguyên ngẫu nhiên k, i <= k <= j. Hàm RandReal(a,b), trong đó a, b là các số thực và a < b, trả về một số thực ngẫu nhiên x , a <= x <= b.

Các thuật toán ngẫu nhiên hay gặp thường là thuộc một trong các lớp sau:

* Các thuật toán tính nghiệm gần đúng của các bài toán số.

* Các thuật toán Monte Carlo. Đặc điểm của các thuật toán này là nó luôn cho ra câu trả lời , song câu trả lời có thể không đúng. Xác suất thành công (tức là nhận được câu trả lời đúng) sẽ tăng, khi ta thực hiện lặp lại thuật toán.

* Các thuật toán Las Vegas. Các thuật toán này không bao giờ cho ra câu trả lời sai, song có thể nó không tìm ra câu trả lời. Xác suất thất bại (không tìm ra câu trả lời) có thể là nhỏ tùy ý, khi ta lặp lại thuật toán một số lần đủ lớn với cùng một dữ liệu vào.

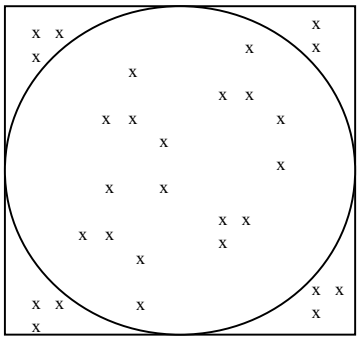
Các thuật toán ngẫu nhiên rất đa dạng và phong phú, và có trong nhiều lĩnh vực khác nhau. Sau đây ta đưa ra một số ví dụ minh họa.

Ví dụ 1. Tính gần đúng số $\prod$

Ta có một hình vuông ngoại tiếp một hình tròn bán kính r (xem hình 16). Ta tiến hành thực nghiệm sau. Ném n hạt vào hình vuông này, giả sử rằng, mọi điểm trong hình vuông này "là điểm rơi khi ta ném một hạt vào hình vuông" với xác suất là như nhau. Diện tích của hình tròn là $\prod r^2$, và diện tích của hình vuông là $4r^2$, do đó $\prod r^2 / 4r^2 = \prod/4$

Giả sử số hạt rơi vào trong hình tròn là k, ta có thể đánh giá $\prod = 4k/n$. Thực nghiệm trên được mô tả bởi thuật toán sau:

```
k = 0;
for (i = 0 ; i < n ; i++)
{
    x  =  RandReal(-r,r);
    y  =  RandReal(-r,r);
    if ( điểm (x,y) nằm trong hình tròn )
                    k++
}
∏ = 4k/n;
```



Hình 16. Ném các hạt để tính $\prod$

Ví dụ 2. Tính gần đúng tính phân xác định

Giả sử ta cần tính tích phân xác định

$$\int_a^b f(x)dx$$

Giả sử tích phân này tồn tại. Ta chọn ngẫu nhiên n điểm trên đoạn [a,b]. Khi đó giá trị của tích phân có thể đánh giá là trung bình cộng các giá trị của hàm f(x) trên các điểm đã chọn nhân với độ dài của đoạn lấy tích phân. Ta có thuật toán sau:

```
Integral(f, a, b)
{
    sum = 0;
    for ( i = 0 ; i < n ; i++)
      {
        x  = RandReal(a, b);
        sum  =  sum+f(x);
      }
    return     (b-a) * (sum / n);

}
```

Ví dụ 3. Phần tử đa số.

Chúng ta gọi phần tử đa số trong một mảng n phần tử A[0..n-1] là phần tử mà số phần tử bằng nó trong mảng A lớn hơn n/2. Với mảng A cho trước ta cần biết mảng A có chứa phần tử đa số hay không. Ta đưa ra thuật toán đơn giản sau. Chọn ngẫu nhiên một phần tử bất kỳ trong mảng A và kiểm tra xem nó có là phần tử đa số hay không.

```
bool      Majority( A[0..n-1] )
{
     i  =  RandInt(0,n-1);
    x  =  A[i];
    k  =  0;
    for ( j = 0 ; j < n ; j++)
       if (A[j] = = x)
              k++;
    return   (k > n / 2);
```

}

Nếu mảng A không chứa phần tử đa số, thuật toán trên luôn trả về false (tức là luôn luôn cho câu trả lời đúng). Giả sử mảng A chứa phần tử đa số. Khi đó thuật toán có thể cho câu trả lời sai. Nhưng vì phần tử đa số chiếm quá nửa số phần tử trong mảng, nên xác suất chọn ngẫu nhiên được phần tử đa số là $p > 1/2$, tức là xác suất thuật toán cho câu trả lời đúng là $p > 1/2$. Bây giờ cho chạy thuật toán trên hai lần và thử tính xem xác suất để "lần đầu nhận được câu trả lời đúng hoặc lần đầu nhận được câu trả lời sai và lần hai nhận được câu trả lời đúng" là bao nhiêu. Xác suất này bằng

$$p + (1 – p)\, p = 1- (1 - p)^2 > 3 / 4$$

Như vậy có thể kết luận rằng, nếu mảng A chứa phần tử đa số, thì thực hiện lặp lại thuật toán trên một số lần đủ lớn, ta sẽ tìm được phần tử đa số. Thuật toán trên là thuật toán Monte Carlo.

Ví dụ 4. Thuật toán Las Vegas cho bài toán 8 con hậu

Chúng ta nhìn lại bài toán 8 con hậu đã đưa ra trong mục 16.4. Nhớ lại rằng, nghiệm của bài toán là vectơ $(x_0, x_1, \ldots, x_7)$, trong đó x_i là cột của con hậu ở dòng thứ i ($i = 0, 1, \ldots, 7$). Trong thuật toán quay lui, x_i được tìm bằng cách xem xét lần lượt các cột 0, 1, …,7 và quan tâm tới điều kiện các con hậu không tấn công nhau. Nhưng quan sát các nghiệm tìm được ta thấy rằng không có một quy luật nào về các vị trí của các con hậu. Điều đó gợi ý ta đưa ra thuật toán ngẫu nhiên sau. Để đặt con hậu thứ i, ta đặt nó ngẫu nhiên vào một trong các cột có thể đặt (tức là khi đặt con hậu thứ i vào cột đó, thì nó không tấn công các con hậu đã đặt). Việc đặt ngẫu nhiên như thế có thể không dẫn tới nghiệm, bởi vì các con hậu đã đặt có thể khống chế mọi vị trí và do đó không thể đặt con hậu tiếp theo.

BÀI TẬP

Thiết kế thuật toán bằng kỹ thuật chia - để - trị cho các bài toán sau:

1. (Trao đổi hai phần của một mảng). Cho mảng A[0…n-1], ta cần trao đổi k phần tử đầu tiên của mảng ($1 \le k < n$) với n – k phần tử còn lại, nhưng không được sử dụng mảng phụ. Chẳng hạn, với k = 3 và A là mảng như sau:

A :	a	b	c	d	e	f	g

Sau khi trao đổi ta cần nhận được mảng:

A :	d	e	f	g	a	b	c

2. (Dãy con không giảm dài nhất). Cho một dãy số nguyên được lưu trong mảng A[0 … n-1], ta cần tìm dãy chỉ số $0 \le i_1 < i_2 < \ldots < i_k \le n - 1$ sao cho $A[i_0] \le A[i_1] \le \ldots \le A[i_k]$ và k là lớn nhất có thể được. Ví dụ, nếu a = (8, 3, 7, 4, 2, 5, 3, 6) thì dãy con không giảm dài nhất là (3, 4, 5, 6).

3. Cho hai dãy không giảm $A = (a_1, a_2, \ldots, a_m)$ trong đó $a_1 \le a_2 \le \ldots \le a_m$ và $B = (b_1, b_2, \ldots, b_n)$ trong đó $b_1 \le b_2 \le \ldots \le b_n$. Dãy hoà nhập của hai dãy không giảm A và B là dãy không giảm $C = (c_1, c_2, \ldots, c_{m+n})$, trong đó mỗi phần tử của dãy A hoặc dãy B xuất hiện trong dãy C đúng một lần.
   Hãy tìm phần tử thứ k của dãy C. Chẳng hạn, nếu A = (1, 3, 5, 9) và B = (3, 6, 8) thì dãy C = (1, 3, 5, 6, 8, 9).

   Thiết kế thuật toán bằng kỹ thuật quy hoạch động cho các bài toán sau:

4. Bài toán tìm dãy con không giảm dài nhất đã nói trong bài toán 2.

5. (Đổi tiền). Cho một tập A các loại tiền $A = \{a_1, a_2, \ldots, a_n\}$, trong đó mỗi a_i là mệnh giá của một loại tiền, a_i là số nguyên dương. Vấn đề đổi tiền được xác định như sau. Cho một số nguyên dương c (số tiền cần đổi), hãy tìm số ít nhất các tờ tiền với các mệnh giá trong A sao cho tổng của chứng bằng c. Giả thiết rằng, mỗi loại tiền có đủ nhiều, và có một loại tiền có mệnh giá là 1.

6. Cho u và v là hai xâu ký tự bất kỳ. Ta muốn biến đổi xâu u thành xâu v bằng cách sử dụng các phép toán sau:
   - Xoá một ký tự.
   - Thêm một ký tự.
   - Thay đổi một ký tự.

   Chẳng hạn, ta có thể biến đổi xâu abbac thành xâu abcbc bằng 3 phép toán như sau:

   abbac → abac (xoá b)
   → ababc (thêm b)
   → abcbc (thay a bằng c)

   Có thể thấy rằng, cách trên không tối ưu, vì chỉ cần 2 phép toán. Vấn đề đặt ra là: hãy tìm số ít nhất các phép toán cần thực hiện để biến xâu u thành xâu v, và cho biết đó là các phép toán nào.

7. Cho n đối tượng, ta muốn sắp xếp n đối tượng đó theo thứ tự được xác định bởi các quan hệ " < " và " = ". Chẳng hạn, với 3 đối tượng A, B, C chúng ta có 13 cách sắp xếp như sau:
   A = B = C, A = B < C, A < B = C, A < B < C, A < C < B, A = C < B, B < A = C, B < A < C, B < C < A, B = C < A, C < A = B, C < A < B, C < B < A.
   Hãy tính số cách sắp xếp n đối tượng.

   Trong các bài toán sau, hãy đưa ra thuật toán được thiết kế bằng kỹ thuật quy lui:

8. Mê lộ là một lưới ô vuông gồm n dòng và n cột, các dòng và các cột được đánh số từ 0 đến n-1. Một ô vuông có thể bị cấm đi vào hoặc không. Từ một ô vuông có thể đi đến ô vuông kề nó theo dòng hoặc theo cột, nếu ô đó không bị cấm đi vào. Cần tìm đường đi từ ô vuông ở góc trên bên trái tới ô ở góc dưới bên phải.

9. Cho số tự nhiên n, hãy cho biết tất cả các dãy số tự nhiên tăng, có tổng bằng n. Chẳng hạn, với n = 6, ta có các dãy sau:
   1 , 2 , 3
   1 , 5
   2 , 4
   6

10. (Bài toán cặp đôi). Cho n đối tượng được đánh số 0, 1, …, n-1. Cho P[i] [j] là số đo sự ưa thích của đối tượng i với đối tượng j, P[i] [j] là

số không âm. Trọng số của cặp đôi (i, j) là tích P[i] [j] * P[j] [i]. Chúng ta cần tìm một cách cặp đôi sao cho mỗi đối tượng phải được cặp đôi với một đối tượng khác (giả sử n chẵn) và hai đối tượng khác nhau cần phải cặp đôi với hai đối tượng khác nhau, và sao cho tổng các trọng số cặp đôi là lớn nhất.

11. Cho bàn cờ n x n và một vị trí xuất phát bất kỳ trên bàn cờ. Tìm đường đi của con mã từ vị trí xuất phát sao cho nó thăm tất cả các vị trí của bàn cờ đúng một lần.

Thiết kế thuật toán giải các bài toán sau đây bằng **kỹ thuật tham ăn**:

12. Quay lại bài toán đổi tiền trong bài tập 5. Hãy đưa ra một thuật toán khác dựa vào ý tưởng sau. Tại mỗi bước, với số tiền còn lại ta sử dụng loại tiền có mệnh giá lớn nhất trong các loại tiền còn lại, và sử dụng số tờ tối đa với mệnh giá đó. Hãy chỉ ra rằng, thuật toán có thể không cho ra cách đổi với số tờ tiền là ít nhất.

13. Cho đồ thị vô hướng G = (V, E), trong đó V là tập đỉnh, còn E là tập cạnh. Một tập U các đỉnh được gọi là một phủ, nếu cạnh (u, v) $\in$ E thì hoặc đỉnh u hoặc đỉnh v phải thuộc U. Phủ có số đỉnh ít nhất được gọi là phủ nhỏ nhất. Có thể xây dựng tập U dần từng bước xuất phát từ U rỗng, tại mỗi bước ta thêm vào U một đỉnh v là đỉnh có bậc lớn nhất trong các đỉnh không có trong U. Hãy viết ra thuật toán dựa theo ý tưởng trên. Thuật toán có cho ra phủ nhỏ nhất không?

14. Hãy đưa ra một thuật toán ngẫu nhiên để tạo ra một mê lộ (xem định nghĩa mê lộ trong bài tập 8).

CHƯƠNG 17

SẮP XẾP

Sắp xếp là một quá trình biến đổi một danh sách các đối tượng thành một danh sách thoả mãn một thứ tự xác định nào đó. Sắp xếp đóng vai trò quan trọng trong tìm kiếm dữ liệu. Chẳng hạn, nếu danh sách đã được sắp xếp theo thứ tự tăng dần (hoặc giảm dần), ta có thể sử dụng kỹ thuật tìm kiếm nhị phân hiệu quả hơn nhiều tìm kiếm tuần tự… Trong thiết kế thuật toán, ta cũng thường xuyên cần đến sắp xếp, nhiều thuật toán được thiết kế dựa trên ý tưởng xử lý các đối tượng theo một thứ tự xác định.

Các thuật toán sắp xếp được chia làm 2 loại: sắp xếp trong và sắp xếp ngoài. Sắp xếp trong được thực hiện khi mà các đối tượng cần sắp xếp được lưu ở bộ nhớ trong của máy tính dưới dạng mảng. Do đó sắp xếp trong còn được gọi là sắp xếp mảng. Khi các đối tượng cần sắp xếp quá lớn cần lưu ở bộ nhớ ngoài dưới dạng file, ta cần sử dụng các phương pháp sắp xếp ngoài, hay còn gọi là sắp xếp file. Trong chương này, chúng ta trình bày các thuật toán sắp xếp đơn giản, các thuật toán này dòi hỏi thời gian $O(n^2)$ để sắp xếp mảng n đối tượng. Sau đó chúng ta đưa ra các thuật toán phức tạp và tinh vi hơn, nhưng hiệu quả hơn, chỉ cần thời gian O(nlogn).

Mảng cần được sắp xếp có thể là mảng số nguyên, mảng các số thực, hoặc mảng các xâu ký tự. Trong trường hợp tổng quát, các đối tượng cần được sắp xếp chứa một số thành phần dữ liệu, và ta cần sắp xếp mảng các đối tượng đó theo một thành phần dữ liệu nào đó. Thành phần dữ liệu đó được gọi là khoá sắp xếp. Chẳng hạn, ta có một mảng các đối tượng sinh viên, mỗi sinh viên gồm các thành phần dữ liệu: tên, tuổi, chiều cao,…, và ta muốn sắp xếp các sinh viên theo thứ tự chiều cao tăng, khi đó chiều cao là khoá sắp xếp.

Từ đây về sau, ta giả thiết rằng, mảng cần được sắp xếp là mảng các đối tượng có kiểu Item, trong đó Item là cấu trúc sau:

```
struct    Item
{
    keyType    key;  // Khoá sắp xếp
    // Các trường dữ liệu khác
};
```

Vấn đề sắp xếp bây giờ được phát biểu chính xác như sau. Cho mảng A[0..n-1] chứa n Item, chúng ta cần sắp xếp lại các thành phần của mảng A sao cho:

A[0].key <= A[1].key <= .. <= A[n-1].key

17.1 CÁC THUẬT TOÁN SẮP XẾP ĐƠN GIẢN

Mục này trình bày các thuật toán sắp xếp đơn giản: sắp xếp lựa chọn (selection sort), sắp xếp xen vào (insertion sort), và sắp xếp nổi bọt (bubble sort). Thời gian chạy của các thuật toán này là $O(n^2)$, trong đó n là cỡ của mảng.

17.1.1 Sắp xếp lựa chọn

Ý tưởng của phương pháp sắp xếp lựa chọn là như sau: Ta tìm thành phần có khóa nhỏ nhất trên toàn mảng, giả sử đó là A[k]. Trao đổi A[0] với A[k]. Khi đó A[0] là thành phần có khoá nhỏ nhất trong mảng. Giả sử đến bước thứ i ta đã có A[0].key <= A[1].key <= … <= A[i-1]. Bây giờ ta tìm thành phần có khóa nhỏ nhất trong các thành phần từ A[i] tới A[n-1]. Giả thành phần tìm được là A[k], i <= k <= n-1. Lại trao đổi A[i] với A[k], ta có A[0].key <=…<= A[i].key. Lặp lại cho tới khi i = n-1, ta có mảng A được sắp xếp.

Ví dụ. Xét mảng A[0…5] các số nguyên. Kết quả thực hiện các bước đã mô tả được cho trong bảng sau

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	I	k
5	9	1	8	3	7	0	2
1	9	5	8	3	7	1	4
1	3	5	8	9	7	2	2
1	3	5	8	9	7	3	5
1	3	5	7	9	8	4	5
1	3	5	7	8	9		

Sau đây là hàm sắp xếp lựa chọn:

```
void    SelectionSort(Item  A[] , int  n)
// Sắp xếp mảng A[0..n-1] với n > 0
{
 (1)    for (int i = 0 ; i < n-1 ; i++)
            {
 (2)            int  k = i;
 (3)            for (int j = i + 1 ;  j < n ; j++)
 (4)                if (A[j].key < A[k].key)
                                k = j;
 (5)            swap(A[i],A[k]);
            }
}
```

Trong hàm trên, swap là hàm thực hiện trao đổi giá trị của hai biến.

Phân tích sắp xếp lựa chọn.

Thân của lệnh lặp (1) là các lệnh (2), (3) và (5). Các lệnh (2) và (5) có thời gian chạy là O(1). Ta đánh giá thời gian chạy của lệnh lặp (3). Số lần lặp là (n-1-i), thời gian thực hiện lệnh (4) là O(1), do đó thời gian chạy của lệnh (3) là (n-1-i)O(1). Như vậy, thân của lệnh lặp (1) có thời gian chạy ở lần lặp thứ i là (n-1-i)O(1). Do đó lệnh lặp (1) đòi hỏi thời gian

$$\sum_{i=0}^{n-2} (n-1-i)O(1) = O(1)(1 + 2 + \ldots + n-1)$$

$$= O(1)n(n-1)/2 = O(n^2)$$

Vậy thời gian chạy của hàm sắp xếp lựa chọn là $O(n^2)$.

17.1.2 Sắp xếp xen vào

Phương pháp sắp xếp xen vào là như sau. Giả sử đoạn đầu của mảng A[0..i-1] (với i >= 1) đã được sắp xếp, tức là ta đã có A[0].key <= … <= A[i-1].key. Ta xen A[i] vào vị trí thích hợp trong đoạn đầu A[0..i-1] để nhận được đoạn A[0..i] được sắp xếp. Với i = 1, đoạn đầu chỉ có một thành phần, đương nhiên là đã được sắp. Lặp lại quá trình đã mô tả với i = 2,…,n-1 ta có mảng được sắp.

Việc xen A[i] vào vị trí thích hợp trong đoạn đầu A[o..i-1] được tiến hành như sau. Cho chỉ số k chạy từ i, nếu A[k].key < A[k-1].key thì ta trao đổi giá trị của A[k] và A[k-1], rồi giảm k đi 1.

Ví dụ. Giả sử ta ta có mảng số nguyên A[0..5] và đoạn đầu A[0..2] đã được sắp

0	1	2	3	4	5
1	4	5	2	9	7

Lúc này i = 3 và k = 3 vì A[3] < A[2], trao đổi A[3] và A[2], ta có

0	1	2	3	4	5
1	4	2	5	9	7

Đến đây k=2, và A[2] < A[1], lại trao đổi A[2] và A[1], ta có

0	1	2	3	4	5
1	2	4	5	9	7

Lúc này k = 1 và A[1] >= A[0] nên ta dừng lại và có đoạn đầu A[0..3] đã được sắp

Hàm sắp xếp xen vào được viết như sau:

```
void    InsertionSort (Item   A[], int  n)
{
```

```
(1)     for ( int i = 1 ; i < n ; i++)
(2)             for ( int k = i ; k > 0 ; k--)
(3)                     if (A[k].key < A[k-1].key)
                                swap(A[k],A[k-1]);
                        else    break;


}
```

Phân tích sắp xếp xen vào

Số lần lặp tối đa của lệnh lặp (2) là i, thân của lệnh lặp (2) là lệnh (3) cần thời gian O(1). Do đó thời gian chạy của lệnh (2) là O(1)i. Thời gian thực hiện lệnh lặp (1) là

$$\sum_{i=1}^{n-1} O(1)i = O(1)(1+2+...+n-1) = O(n^2)$$

17.1.3 Sắp xếp nổi bọt

Ý tưởng của sắp xếp nổi bọt là như sau. Cho chỉ số k chạy từ 0, 1 , …, n-1, nếu hai thành phần kề nhau không đúng trật tự, tức là A[k].key >A[k+1].key thì ta trao đổi hai thành phần A[k] và A[k+1]. Làm như vậy ta đẩy được dữ liệu có khoá lớn nhất lên vị trí sau cùng A[n-1].

Ví dụ. Giả sử ta có mảng số nguyên A[0..4]= (6,1,7,3,5).Kết quả thực hiện quá trình trên được cho trong bảng sau:

A[0]	A[1]	A[2]	A[3]	A[4]	
6	1	7	3	5	Trao đổi A[0] và A[1]
1	6	7	3	5	Trao đổi A[2] và A[3]
1	6	3	7	5	Trao đổi A[3] và A[4]
1	6	3	5	7	

Lặp lại quá trình trên đối với mảng A[0,…, n-2] để đẩy dữ liệu có khoá lớn nhất lên vị trí A[n-2]. Khi đó ta có A[n-2].key ≤ A[n-1].key. Tiếp tục lặp lại quá trình đã mô tả trên các đoạn đầu A[0..i], với i = n-3, …,1, ta sẽ thu được mảng được sắp . Ta có hàm sắp xếp nổi bọt như sau:

```
void   BubbleSort( Item   A[] , int  n)
{
(1)             for (int i = n-1 ;  i > 0 ; i--)
(2)             for (int k = 0 ; k < i ; k++)
(3)                     if  ( A[k].key > A[k+1].key)
                                Swap(A[k],A[k+1]);
}
```

Tương tự như hàm sắp xếp xen vào ,ta có thể đánh giá thời gian chạy của hàm sắp xếp nổi bọt là $O(n^2)$.

Trong hàm BubbleSort khi thực hiện lệnh lặp (1), nếu đến chỉ số i nào đó, $n-1 \geq i > 1$, mà đoạn đầu A[0..i] đã được sắp, thì ta có thể dừng. Do đó ta có thể cải tiến hàm BubbleSort bằng cách đưa vào biến sorted, biến này nhận giá trị true nếu A[0..i] đã được sắp và nhận giá trị false nếu ngược lại. Khi sorted nhận giá trị true thì lệnh lặp (1) sẽ dừng lại.

```
void    BubbleSort (Item   A[] , int  n)
 {
    for (int i = n-1 ; i > 0 ; i -- )
        {
            bool     sorted = true;
            for( int k = 0 ; k < i ; k++)
              if (A[k].key > A[k+1].key)
                 {
                   swap (A[k], A[k+1]);
                    sorted   =  false;
                 }
             if  (sorted)     break;
        }
 }
```

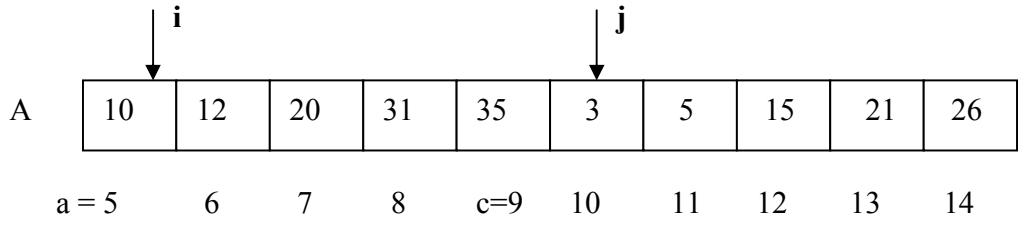
17.2 SẮP XẾP HOÀ NHẬP

Thuật toán sắp xếp hoà nhập (MergeSort) là một thuật toán được thết kế bằng kỹ thuật chia - để - trị. Giả sử ta cần sắp xếp mảng A[a..b], trong đó a, b là các số nguyên không âm, a b, a là chỉ số đầu và b là chỉ số

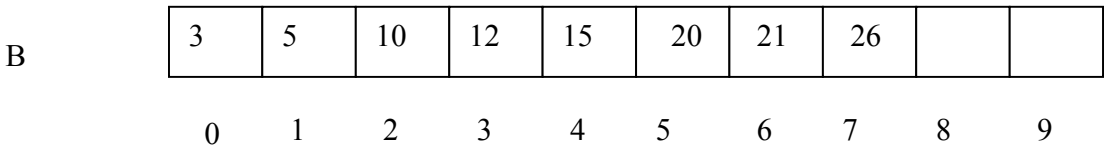
cuối của mảng. Ta chia mảng thành hai mảng con bởi chỉ số c nằm giữa a và b (c = (a + b) / 2). Các mảng con A[a..c] và A[c+1…b] được sắp xếp bằng cách gọi đệ quy thủ tục sắp xếp hoà nhập. Sau đó ta hoà nhập hai mảng con A[a…c] và A[c+1…b] đã được sắp thành mảng A[a…b] được sắp. Giả sử Merge(A,a,c,b) là hàm kết hợp hai mảng con đã được sắp A[a..c] và A[c+ 1..b] thành mảng A[a..b] được sắp. Thuật toán sắp xếp hoà nhập được biểu diễn bởi hàm đệ quy sau.

```
void    MergeSort( Item  A[ ], int a, int b)
{
        if (a < b)
                {
                        int c = (a + b)/2;
                        MergeSort ( A, a, c );
                        MergeSort ( A, c+1, b);
                        Merge ( A, a, c, b);
                }
}
```

Công việc còn lại của ta là thiết kế hàm hoà nhập Merge (A, a, c, b), nhiệm vụ của nó là kết hợp hai nửa mảng đã được sắp A[a…c] và A[c+1…b] thành mảng được sắp. Ý tưởng của thuật toán hoà nhập là ta đọc lần lượt các thành phần của hai nửa mảng và chép vào mảng phụ B[0..b-a] theo đúng thứ tự tăng dần. Giả sử i là chỉ số chạy trên mảng con A[a…c], i được khởi tạo là a ; j là chỉ số chạy trên mảng con A[c+1..b], j được khởi tạo là c + 1. So sánh A[i] và A[j], nếu A[i].key < A[j].key thì ta chép A[i] vào mảng B và tăng i lên 1, còn nếu ngược lại thì ta chép A[j] vào mảng B va tăng j lên 1. Lặp lại hành động đó cho đến khi i vượt quá c hoặc j vượt quá b. Nếu chỉ số i chưa vượt quá c nhưng j đã vượt quá b thì ta cần phải chép phần còn lại A[i…c] vào mảng B. Tương tự, nếu i > c, nhưng j ≤ b thì ta cần chép phần còn lại A[j…b] vào mảng B. Chẳng hạn, xét mảng số nguyên A[5…14], trong đó A[5…9] và A[10…14] đã được sắp như sau:



Bắt đầu i = 5 , j = 10. Vì A[5] > A[10] nên A[10] = 3 được chép vào mảng B và j = 11. Ta lại có A[5] > A[11], nên A[11] = 5 được chép vào mảng B và j = 12. Đến dây A[5] < A[12], ta chép A[5] = 10 vào mảng B và i = 6. Tiếp tục như thế ta nhận được mảng B như sau:



Đến đây j = 15 > b = 14, còn i = 8 < c = 9, do đó ta chép nốt A[8] = 31 và A[9] = 35 sang B để nhận được mảng B được sắp. Bây giờ chỉ cần chép lại mảng B sang mảng A. Hàm Merge được viết như sau:

```
void    Merge( Item  A[] , int  a , int  c , int b)
// a, c, b là các số nguyên không âm, a ≤ c ≤ b.
// Các mảng con A[a…c] và A[c+1…b] đã được sắp.
{
        int i = a;
        int j = c + 1;
        int k = 0;
        int n = b – a + 1;
        Item * B = new Item[n];
 (1)    while ((  i < c +1 )  &&  ( j < b +1 ))
           if ( A [i].key < A[j].key)
                        B[k ++]  =  A[i ++];
```

```
            else        B[k ++]  =  A[j ++];
(2)     while  ( i < c + 1)
                B[k ++] = A[i++];
(3)     while ( j < b +1)
                B[k ++] = A[ j ++];
         i  =  a;
(4)     for ( k = 0 ; k < n ; k ++)
                A[i ++] = B [k];
         delete  [ ]   B;
}
```

Phân tích sắp xếp hoà nhập.

Giả sử mảng cần sắp xếp A[a…b] có độ dài n, n = b – a +1, và T(n) là thời gian chạy của hàm MergeSort (A, a, b). Khi đó thời gian thực hiện mỗi lời gọi đệ quy MergeSort (A, a, c) và MergeSort (A, c + 1, b) là T(n/2). Chúng ta cần đánh gía thời gian chạy của hàm Merge(A, a, c, b). Xem xét hàm Merge ta thấy rằng, các lệnh lặp (1), (2), (3) cần thực hiện tất cả là n lần lặp, mỗi lần lặp chỉ cần thực hiện một số cố định các phép toán. Do đó tổng thời gian của ba lệnh lặp (1), (2), (3) là O(n). Lệnh lặp (4) cần thời gian O(n). Khi thực hiện hàm MergeSort(A, a, b) với a = b, chỉ một phép so sánh phải thực hiện, do đó T(1) = O(1). Từ hàm đệ quy MergeSort và các đánh giá trên, ta có quan hệ đệ quy sau

$$T(1) = O(1)$$

$$T(n) = 2T(n/2) + O(n) \text{ với } n>1$$

Giả sử thời gian thực hiện các phép toán trong mỗi lần lặp ở hàm Merge là hằng số d nào đó, ta có :

$$T(1) \leq d$$

$$T(n) \leq 2T(n/2) + nd$$

Áp dụng phương pháp thế lặp vào bất đẳng thức trên ta nhận được

$$T(n) \leq 2T(n/2) + n\,d$$

$$\leq 2^2 T(n/2^2) + 2\,(n/2)d + n\,d$$

……

$$\leq 2^k T(n/2^k) + n\,d + \ldots + n\,d \quad (k \text{ lần } nd)$$

Giả sử k là số nguyên dương lớn nhất sao cho $1 \leq n / 2^k$. Khi đó, ta có

$$T(n) \leq 2^k T(1) + n\,d + \ldots + n\,d \quad (k \text{ lần } n\,d)$$

$$T(n) \leq (k + 1)\,n\,d$$

$$T(n) \leq (1 + \log n)\,n\,d$$

Vậy T(n) = O (n log n).

17.3 SẮP XẾP NHANH

Trong mục này chúng ta trình bày thuật toán sắp xếp được đưa ra bởi Hoare, nổi tiếng với tên gọi là sắp xếp nhanh (QuickSort). Thời gian chạy của thuật toán này trong trường hợp xấu nhất là $O(n^2)$. Tuy nhiên thời gian chạy trung bình là O(n logn).

Thuật toán sắp xếp nhanh được thiết kế bởi kỹ thuật chia-để-trị như thuật toán sắp xếp hòa nhập. Nhưng trong thuật toán sắp xếp hòa nhập, mảng A[a…b] cần sắp được chia đơn giản thành hai mảng con A[a..c] và A[c+1..b] bởi điểm chia ở giữa mảng, c = (a+b)/2. Còn trong thuật toán sắp xếp nhanh, việc "chia mảng thành hai mảng con" là một quá trình biến đổi phức tạp để từ mảng A[a..b] ta thu được hai mảng con A[a..k-1] và A[k+1..b] thỏa mãn các tính chất sau :

A[i].key $\leq$ A[k].key với mọi i, $a \leq i \leq k-1$.

A[j].key $>$ A[k].key với mọi j, $k+1 \leq j \leq b$.

Nếu thực hiện được sự phân hoạch mảng A[a..b] thành hai mảng con A[a..k-1] và A[k+1..b] thỏa mãn các tính chất trên, thì nếu sắp xếp được các mảng con đó ta sẽ có toàn bộ mảng A[a..b] được sắp xếp. Giả sử Partition(A, a, b, k) là hàm phân hoạch mảng A[a..b] thành hai mảng con A[a..k-1] và

A[k+1..b]. Thuật toán sắp xếp nhanh là thuật toán đệ quy được biểu diễn bởi hàm đệ quy như sau :

```
void    QuickSort(Item   A[] , int a , int b)
//Sắp xếp mảng A[a..b] với a ≤ b.
{
    if (a < b)
        {
            int   k;
            Partition(A, a, b, k);
            if (a <= k – 1)
                QuickSort(A, a, k – 1);
            if (k + 1 <= b)
                QuickSort(A, k + 1, b);
        }
}
```

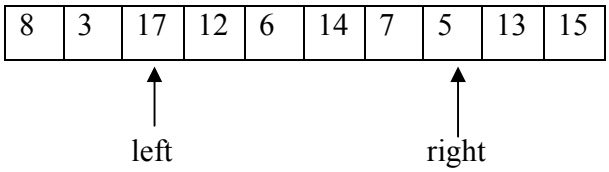
Hàm phân hoạch Partition là thành phần chính của thuật toán sắp xếp nhanh. Vấn đề còn lại là xây dựng hàm phân hoạch. Ý tưởng của thuật toán phân hoạch là như sau. Đầu tiên ta chọn một thành phần trong mảng A[a..b] làm mốc (pivot). Sau đó ta chuyển tất cả các thành phần có khóa nhỏ hơn hoặc bằng khóa của mốc sang bên trái mốc, chuyển tất cả các thành phần có khóa lớn hơn khóa của mốc sang bên phải mốc. Kết quả là, ta có mốc đứng ở vị trí k, bên trái là mảng con A[a..k – 1], và bên phải là mảng con A[k + 1..b], các mảng con này có tính chất mong muốn, tức là mọi thành phần trong mảng con A[a..k - 1] có khỏa nhỏ hơn hay bằng khóa của A[k] và mọi thành phần trong mảng con A[k + 1..b] có khóa lớn hơn khóa của A[k].

Chọn mốc phân hoạch như thế nào? Đương nhiên là, ta mong muốn chọn được phần tử làm mốc sao cho kết quả phân hoạch cho ta hai mảng con bằng nhau. Điều này là có thể làm được, tuy nhiên nó đòi hỏi nhiều thời gian hơn sự cần thiết. Vì vậy, ta sẽ chọn ngay thành phần đầu tiên của mảng làm mốc, tức là pivot = A[a].key. Sau đó ta sử dụng hai chỉ số, chỉ số left chạy từ trái sang phải, ban đầu left = a + 1, chỉ số right chạy từ phải sang trái, ban đầu right = b. Biến left sẽ tăng và dừng tại vị trí mà A[left].key > pivot, còn

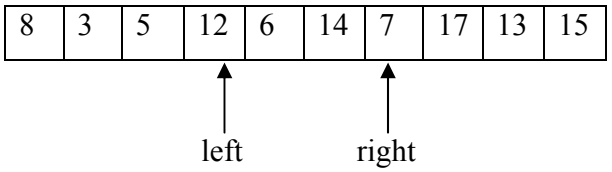
biến right sẽ giảm và dừng lại tại vị trí mà A[right].key ≤ pivot. Khi đó nếu left < right thì ta trao đổi giá trị của A[left] với A[right]. Quá trình trên được lặp lại cho tới khi left > right. Lúc này ta dễ thấy rằng, mọi thành phần trong mảng A[a..right] có khóa nhỏ hơn hay bằng mốc, còn mọi thành phần trong mảng A[left..b] có khóa lớn hơn mốc. Cuối cùng ta trao đổi A[a] và A[right] để đặt mốc vào vị trí k = right. Hàm phân hoạch được viết như sau :

```
void   Partition( Item   A[] , int a , int b , int & k)
   {
       keyType   pivot = A[a].key;
       int   left = a + 1;
       int   right =   b;
       do {
            while (( left <= right ) & (A[left].key <= pivot ))
                    left ++;
            while (( left <= right ) & (A[right].key > pivot ))
                    right --;
            if (left < right)
               {
                    swap(A[left], A[right]);
                    left ++;
                    right --;
               }
            }
        while (left <= right);
        swap (A[a], A[right]) ;
        k = right ;
    }
```

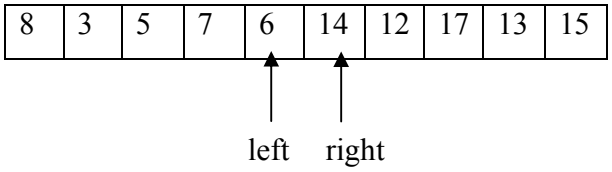
Để thấy được hàm phân hoạch làm việc như thế nào, ta hãy xét ví dụ sau. Giả sử ta cần phân hoạch mảng số nguyên A[0..9] như sau :



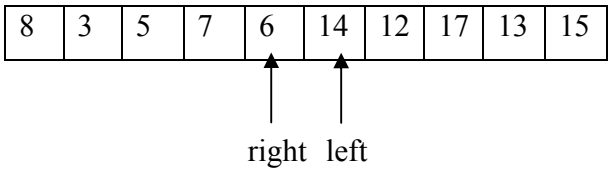
Lấy mốc pivot = A[0] = 8, ban đầu left = 1, right = 9. Chỉ số left tăng và dừng lại tại vị trí left = 2, vì A[2] = 17 > 8, chỉ số right giảm và dừng lại tại vị trí right = 7, vì A[7] = 5 < 8. Trao đổi A[2] với A[7], đồng thời tăng left lên 1, giảm right đi 1, ta có :



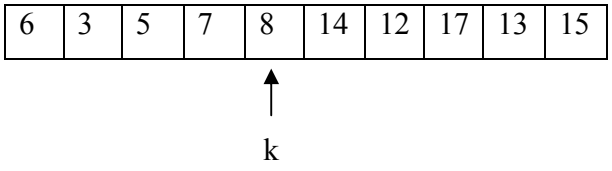
Đến đây A[left] = 12 > 8 và A[right] = 7 < 8. Lại trao đổi A[left] với A[right], và tăng left lên 1, giảm right đi 1, ta có :



Tiếp tục, A[left] = 6 < 8 nên left được tăng lên và dừng lại tại left = 5 vì A[5] > 8. A[right] = 14 > 8 nên right được giảm đi và dừng lại tại right = 4, vì A[4] < 8. Ta có hoàn cảnh sau :



Đến đây right < left, ta dừng lại, trao đổi A[0] với A[4] ta thu được phân hoạch với k = right = 4.



Phân tích sắp xếp nhanh

Chúng ta cần đánh giá thời gian chạy T(n) của thuật toán sắp xếp nhanh trên mảng A[a..b] có n phần tử, n = b – a + 1. Trước hết ta cần đánh giá thời gian thực hiện hàm phân hoạch. Thời gian phân hoạch là thời gian đi qua mảng (hai biến left và right chạy từ hai đầu mảng cho tới khi chúng gặp nhau), tại mỗi vị trí mà left và right chạy qua ta cần so sánh thành phần ở vị trí đó với mốc và các trao đổi khi cần thiết. Do đó khi phân hoạch một mảng n phần tử ta chỉ cần thời gian O(n).

Thời gian trong trường hợp tốt nhất. Trường hợp tốt nhất xảy ra khi mà sau mỗi lần phân hoạch ta nhận được hai mảng con bằng nhau. Trong trường hợp này, từ hàm đệ quy QuickSort, ta suy ra quan hệ đệ quy sau :

$T(1) = O(1)$

$T(n) = 2\ T(n/2) + O(n)$ với $n > 1$.

Đây là quan hệ đệ quy mà ta đã gặp khi phân tích sắp xếp hòa nhập. Như vậy trong trường hợp tốt nhất thời gian chạy của QuickSort là O(n logn).

Thời gian trong trường hợp xấu nhất. Trường hợp xấu nhất là trường hợp mà sau mỗi lần phân hoạch mảng n phần tử ta nhận được mảng con n – 1 phần tử ở một phía của mốc, còn phía kia không có phần tử nào. (Dễ thấy rằng trường hợp này xẩy ra khi ta phân hoạch một mảng đã được sắp). Khi đó ta có quan hệ đệ quy sau :

$T(1) = O(1)$

$T(n) = T(n – 1) + O(n)$ với $n > 1$

Ta có :

$T(1) = C$

$T(n) = T(n – 1) + nC$ với $n > 1$

Trong đó C là hằng số nào đó. Bằng cách thế lặp ta có :

$T(n) = T(1) + 2C + 3C + \ldots + nC$

$$= C \sum_{i=1}^{n} i = Cn(n+1)/2$$

Do đó trong trường hợp xấu nhất, thời gian chạy của sắp xếp nhanh là $O(n^2)$.

Thời gian trung bình. Bây giờ ta đánh giá thời gian trung bình $T_{tb}(n)$ mà QuickSort đòi hỏi để sắp xếp một mảng có n phần tử. Giả sử mảng A[a..b] chứa n phần tử được đưa vào mảng một cách ngẫu nhiên. Khi đó hàm phân hoạch Partition(A, a, b, k) sẽ cho ra hai mảng con A[a..k – 1] và A[k + 1..b] với k là một trong các chỉ số từ a đến b với xác suất như nhau và bằng 1/n. Vì thời gian thực hiện hàm phân hoạch là O(n), từ hàm QuickSort ta suy ra quan hệ đệ quy sau :

$$T_{tb}(n) = \frac{1}{n} \sum_{k=1}^{n} [\, T_{tb}(k - 1) + T_{tb}(n - k)] + O(n)$$

Hay

$$T_{tb}(n) = \frac{1}{n} \sum_{k=1}^{n} [\, T_{tb}(k - 1) + T_{tb}(n - k)] + nC \qquad (1)$$

Trong đó C là hằng số nào đó. Chú ý rằng

$$\sum_{k=1}^{n} T_{tb}(k - 1) = \sum_{k=1}^{n} T_{tb}(n - k)$$

Do đó có thể viết lại (1) như sau :

$$T_{tb}(n) = \frac{2}{n} \sum_{k=1}^{n} T_{tb}(k - 1) + nC \qquad (2)$$

Trong (2) thay n bởi n – 1 ta có :

$$T_{tb}(n - 1) = \frac{2}{n-1} \sum_{k=1}^{n-1} T_{tb}(k - 1) + (n - 1)C \qquad (3)$$

Nhân (2) với n, nhân (3) với n – 1 và trừ cho nhau ta nhận được

$$n\, T_{tb}(n) = (n + 1)\, T_{tb}(n - 1) + (2n - 1)C$$

Chia đẳng thức trên cho n(n + 1) ta nhận được quan hệ đệ quy sau :

$$\frac{T_{tb}(n)}{n+1} = \frac{T_{tb}(n - 1)}{n} + \frac{2n - 1}{n(n+1)} C \qquad (4)$$

Sử dụng phép thế lặp, từ (4) ta có

$$\frac{T_{tb}(n)}{n+1} = \frac{T_{tb}(n - 1)}{n} + \frac{2n - 1}{n(n+1)} C$$

$$= \frac{T_{tb}(n - 2)}{n-1} + \frac{2n - 3}{(n-1)n} + \frac{2n - 1}{n(n+1)} C$$

. . .

$$\frac{T_{tb}(n)}{n+1} = \frac{T_{tb}(1)}{2} + c \sum_{i=1}^{n} \frac{2i-1}{i(i+1)} \qquad (5)$$

Ta có đánh giá

$$\sum_{k=1}^{n} \frac{2i-1}{i(i+1)} \le \sum_{i=1}^{n} \frac{2}{i} \le 2 \int_{1}^{n} \frac{dx}{x} \le 2\log n$$

Do đó từ (5) ta suy ra

$$\frac{T_{tb}(n)}{n+1} = O(\log n)$$

hay $T_{tb}(n) = O(n \log n)$.

Trong trường hợp xấu nhất, QuickSort đòi hỏi thời gian $O(n^2)$, nhưng trường hợp này rất ít khi xảy ra. Thời gian trung bình của QuickSort là O(n logn), và thời gian trong trường hợp xấu nhất của MergeSort cũng là O(n

logn). Tuy nhiên thực tiễn cho thấy rằng, trong phần lớn các trường hợp QuickSort chạy nhanh hơn các thuật toán sắp xếp khác.

17.4 SẮP XẾP SỬ DỤNG CÂY THỨ TỰ BỘ PHẬN

Trong mục này chúng ta trình bày phương pháp sắp xếp sử dụng cây thứ tự bộ phận (heapsort). Trong mục 10.3, chúng ta biết rằng một cây thứ tự bộ phận n đỉnh có thể biểu diễn bởi mảng A[0..n-1], trong đó gốc cây được lưu trong A[0], và nếu một đỉnh được lưu trong A[i], thì đỉnh con trái (nếu có) của nó được lưu trong A[2*i + 1], còn đỉnh con phải nếu có của nó được lưu trong A[2*i + 2]. Mảng A thoả mãn tính chất sau (ta sẽ gọi là tính chất heap):

A[i].key <= A[2*i+1].key và

A[i].key <= A[2*i+2].key

với mọi chỉ số i, 0 <= i <= n/2-1.

Với mảng thoả mãn tính chất heap thì A[0] là phần tử có khoá nhỏ nhất. Do đó ta có thể đưa ra thuật toán sắp xếp mảng như sau.

Giả sử mảng cần được sắp là mảng A[0..n-1]. Đầu tiên ta biến đổi mảng A thành mảng thoả mãn tính chất heap. Sau đó ta trao đổi A[0] và A[n-1]. Mảng A[0..n-2] bây giờ thoả mãn tính chất heap với mọi i >= 1, trừ i = 0. Biến đổi mảng A[0..n-2] để nó thoả mãn tính chất heap. Lại trao đổi A[0] và A[n-2]. Rồi lại biến đổi mảng A[0..n-3] trở thành mảng thoả mãn tính chất heap. Lặp lại quá trình trên, cuối cùng ta sẽ nhận được mảng A[0..n-1] được sắp theo thứ tự giảm dần:

A[0].key >= A[1].key >= … >= A[n-1].key

Trong quá trình trên, sau mỗi lần trao đổi A[0] với A[m] (với m=n-1,…,1), ta sẽ nhận được mảng A[0…m-1] thoả mãn tính chất heap với mọi i >= 1, trừ i = 0. Điều này có nghĩa là cây nhị phân được biểu diễn bởi mảng A[0..m-1] đã thoả mãn tính chất thứ tự bộ phận, chỉ trừ gốc. Để nó trở thành

cây thứ tự bộ phận, ta chỉ cần đẩy dữ liệu lưu ở gốc xuống vị trí thích hợp trong cây, bằng cách sử dụng hàm ShiftDown (Xem mục 10.3.3).

Còn một vấn đề cần giải quyết, đó là biến đổi mảng cần sắp xếp A[0..n-1] thành mảng thoả mãn tính chất heap. Điều này có nghĩa là ta phải biến đổi cây nhị phân được biểu diễn bởi mảng A[0..n-1] thành cây thứ tự bộ phận. Muốn vậy, với i chạy từ n/2-1 giảm xuống 0, ta chỉ cần sử dụng hàm SiftDown để đẩy dữ liệu lưu ở đỉnh i xuống vị trí thíc hợp trong cây. Đây là cách xây dựng cây thứ tự bộ phận mà chúng ta đã trình bày trong 10.3.2.

Bây giờ ta viết lại hàm ShiftDown cho thích hợp với sự sử dụng nó trong thuật toán. Giả sử mảng A[a..b] (a < b) đã thoả mãn tính chất heap với mọi i >= a+1. Hàm ShiftDown(a,b) sau đây thực hiện việc đẩy A[a] xuống vị trí thích hợp trong mảng A[a..b] để mảng thoả mãn tính chất heap với mọi i >= a.

```
void    ShiftDown(int a, int b)
{
   int  i = a;
   int  j = 2 * i + 1;
   while (j <= b)
     {
        int  k = j + 1;
        if (k <= b && A[k].key < A[j].key)
              j = k;
        if (A[i].key > A[j].key)
          {
              swap(A[i],A[j]);
              i  =  j;
              j  =  2 * i + 1;
          }
        else     break;
     }
}
```

Sử dụng hàm ShiftDown, ta đưa ra thuật toán sắp xếp HeapSort sau đây. Cần lưu ý rằng, kết quả của thuật toán là mảng A[0..n-1] được sắp xếp theo thứ tự giảm dần.

```
void      HeapSort(Item   A[] , int  n)
//Sắp xếp mảng A[0..n-1] với n > 1
{
    for (int  i = n / 2 – 1 ; i >= 0 ; i--)
                ShiftDown(i,n-1);       //Biến đổi mảng A[0..n-1]
                                        // thành mảng thoả mãn tính chất heap
    for (int  i = n – 1 ; i >= 1 ; i--)
        {
            swap(A[0],A[i]);
            ShiftDown(0,i - 1);
        }

}
```

Phân tích HeapSort.

Thời gian thực hiện lệnh lặp (1) là thời gian xây dựng cây thứ tự bộ phận mà chúng ta đã xét trong mục 10.3.2. Theo chứng minh đã đưa ra trong 10.3.2, lệnh lặp (1) chỉ đòi hỏi thời gian O(n). Trong lệnh lặp (2), số lần lặp là n-1. Thân vòng lặp (2), với i = n-1 là

```
swap(A[0],A[n - 1]);

ShiftDown(0,n - 2);
```

Đây là các lệnh thực hiện DeleteMin trên cây thứ tự bộ phận được biểu diễn bởi mảng A[0..n-1], và dữ liệu có khoá nhỏ nhất được lưu vào A[n-1]. Trong mục 10.3.1, ta đã chứng tỏ rằng DeleteMin chỉ cần thời gian O(logn). Như vậy thân của lệnh lặp (2) cần thời gian nhiều nhất là O(logn). Do đó lệnh (2) cần thời gian O(nlogn). Vì vậy, thời gian thực hiện HeapSort là O(nlogn).

BÀI TẬP.

1. Cho mảng các số nguyên (8, 1, 4, 1, 5, 2, 6, 5). Hãy sắp xếp mảng này bằng cách sử dụng:
    a. Sắp xếp lựa chọn.
    b. Sắp xếp xen vào.
    c. Sắp xếp nổi bọt.
    d. Sắp xếp nhanh.
    e. Sắp xếp hoà nhập.
    f. Sắp xếp sử dụng cây thứ tự bộ phận.

    Cần đưa ra kết quả thực hiện mỗi bước của thuật toán.

2. Hãy đánh giá thời gian chạy của các thuật toán sắp xếp:
    a. Sắp xếp lựa chọn.
    b. Sắp xếp xen vào.
    c. Sắp xếp nhanh.
    d. Sắp xếp hoà nhập.

    Trong các trường hợp sau:
    a. Mảng đầu vào có tất cả các phần tử có khoá bằng nhau.
    b. Mảng đầu vào đã được sắp.

3. Viết hàm phân hoạch mảng A[a …b] với phần tử được chọn làm mốc là phần tử đứng giữa mảng, tức là phần tử A[(a + b) / 2].

4. Một thuật toán sắp xếp được xem là ổn định, nếu trật tự của các phần tử có khoá bằng nhau trong mảng đầu vào và trong mảng kết quả là như nhau. Trong các thuật toán sắp xếp, thuật toán nào là ổn định, thuật toán nào là không ổn định?

5. Cho mảng chứa n số nguyên và một số nguyên k. Ta muốn biết trong mảng có chứa 2 số nguyên có tổng bằng k hay không. Chẳng hạn, với mảng (8, 4, 1, 5, 6, 3) và k = 10 thì cầu trả lời là có, vì 4 + 6 = 10.
    a. Hãy đưa ra thuật toán không sử dụng sắp xếp mảng. Đánh giá thời gian chạy của thuật toán.
    b. Hãy đưa ra thuật toán sử dụng sắp xếp mảng, thuật toán chỉ đòi hỏi thời gian O(nlogn).

6. Phần tử nhỏ nhất thứ k (k = 1, 2, …) trong mảng chứa n phần tử A[1…n] là phần tử p trong mảng sao cho số phần tử < p là < k, và số

phần tử ≤ p là ≥ k. Ví dụ, trong mảng A = (5, 7, 5, 9, 3, 5) thì phần tử nhỏ nhất là 3, còn phần tử nhỏ nhất thứ 2, 3, 4 đều là 5, vì số phần tử < 5 là 1, còn số phần tử ≤ 5 là 4. Hãy đưa ra thuật toán tìm phần tử nhỏ nhất thứ k trong mảng. (Gợi ý: sử dụng phương pháp phân hoạch như trong QuickSort để biến đổi mảng đã cho thành mảng A[1 …n] sao cho phần tử nhỏ nhất thứ k là A[k], tức là với 1 ≤ i < k thì A[i] < A[k], còn với k < j ≤ n thì A[j] ≥ A[k])

CHƯƠNG 18

CÁC THUẬT TOÁN ĐỒ THỊ

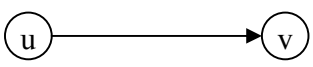
Đồ thị là một mô hình toán học được sử dụng để biểu diễn một tập đối tượng có quan hệ với nhau theo một cách nào đó. Chẳng hạn trong khoa học máy tính, đồ thị được sử dụng để mô hình hoá một mạng truyền thông, kiến trúc của các máy tính song song,... Rất nhiều vấn đề trong các lĩnh vực khác như công nghệ điện, hoá học, chính trị, kinh tế,... cũng có thể biểu diễn bởi đồ thị. Khi một vấn đề được mô hình hoá bởi đồ thị, thì vấn đề sẽ được giải quyết bằng cách sử dụng các thuật toán trên đồ thị. Vì vậy các thuật toán đồ thị có phạm vi áp dụng rộng lớn và có tầm quan trọng đặc biệt. Trong chương này chúng ta sẽ nghiên cứu một số thuật toán quan trọng nhất trên đồ thị: các thuật toán đi qua đồ thị, các thuật toán tìm đường đi ngắn nhất, tìm cây bao trùm ngắn nhất... Nghiên cứu các thuật toán đồ thị còn giúp ta hiểu rõ hơn cách vận dụng các kỹ thuật thiết kế thuật toán (đã được trình bày trong chương 16) để giải quyết các vấn đề cụ thể.

18.1 MỘT SỐ KHÁI NIỆM CƠ BẢN

Trong mục này, chúng ta trình bày một số khái niệm cơ bản về đồ thị.

Một **đồ thị định hướng** G = (V,E) gồm một tập hữu hạn V các đỉnh và một tập E các cung. Mỗi cung là một cặp có thứ tự các đỉnh khác nhau (u,v), tức là (u,v) và (v,u) là hai cung khác nhau.

Cung (u,v) sẽ được gọi là cung đi từ đỉnh u tới đỉnh v và được ký hiệu là u→v. Trong biểu diễn hình học cung (u,v) sẽ được biểu diễn bởi mũi tên như sau

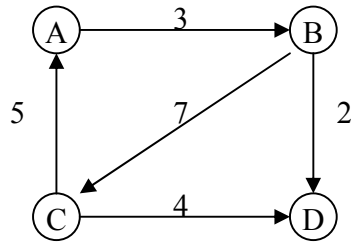


Nếu có cung u→v, thì ta nói v là đỉnh kề với đỉnh u. Trong các ứng dụng thực tế, khi chúng ta quan tâm đến một tập các đối tượng với một quan hệ nào đó, thì ta có thể sử dụng đồ thị để biểu diễn: Các đỉnh của đồ thị là các đối tượng đó và nếu đối tượng A có quan hệ với đối tượng B thì trong đồ thị có cung đi từ A đến đỉnh B.

Để mô hình hoá nhiều vấn đề xuất phát từ các lĩnh vực khác nhau, chúng ta cần phải sử dụng đồ thị có trọng số. Đó là đồ thị mà mỗi cung (u,v) được gắn với một số c(u,v). Số c(u,v) được gọi là **trọng số** của cung (u,v), hay còn được gọi là **giá** hoặc **độ dài** của cung đó.

Một đường đi trên đồ thị G = (V,E) là một dãy hữu hạn các đỉnh (v_0, v_1, …,v_k), trong đó các đỉnh v_0, v_1, …,v_k là khác nhau, trừ ra có thể $v_0 = v_k$, và có cung $v_i \rightarrow v_{i+1}$ với i = 0, 1, …, k-1. Chúng ta sẽ nói đường đi (v_0, v_1, …,v_k) là đường đi từ đỉnh v_0 đều đỉnh v_k. Nếu đồ thị không có trọng số thì độ dài của đường đi (v_0, v_1, …,v_k) được xem là bằng k, còn nếu đồ thị có trọng số thì độ dài của đường đi đó là tổng độ dài của các cung trên đường đi.

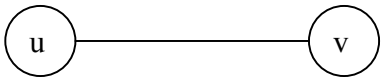
Một đường đi khép kín được gọi là một **chu trình**, hay nói cách khác, chu trình là đường đi từ một đỉnh đến chính nó. Hình 18.1 biểu diễn một đồ thị có trọng số, đồ thị này có một chu trình (A, B, C, A), từ đỉnh A đến đỉnh D có hai đường đi, đường đi (A, B, D) có độ dài 5 và đường đi (A, B, C, D) có độ dài 14.



Hình 18.1. Một đồ thị định hướng có trọng số

Một **đồ thị vô hướng** G = (V, E) gồm một tập hữu hạn V các đỉnh và một tập các cạnh E. Cần lưu ý rằng, mỗi cạnh của đồ thị vô hướng là một

cặp không có thứ tự các đỉnh khác nhau, tức là cạnh (u,v) và cạnh (v,u) là một. Trong biểu diễn hình học, cạnh (u,v) được biểu diễn bởi đoạn thẳng nối hai đỉnh u và v:



Chú ý rằng, mỗi đồ thị vô hướng đều có thể xem như đồ thị định hướng, trong đó mỗi cạnh (u,v) của đồ thị vô hướng được xem như hai cung u→v và v→u trong đồ thị định hướng. Sau này khi không nói rõ mà chỉ nói đồ thị thì bạn đọc cần hiểu đó là đồ thị định hướng. Một số khái niệm quan trọng khác về đồ thị sẽ được đưa ra sau này khi cần thiết.

18.2 BIỂU DIỄN ĐỒ THỊ

Để giải quyết các vấn đề của đồ thị bằng máy tính chúng ta cần lưu giữ đồ thị trong bộ nhớ của máy tính. Do đó chúng ta cần đưa ra các phương pháp biểu diễn đồ thị bởi các cấu trúc dữ liệu. Có nhiều phương pháp biểu diễn đồ thị, nhưng được sử dụng nhiều nhất là hai cách biểu diễn sau: biểu diễn đồ thị bằng ma trận kề và bằng danh sách kề.

18.2.1 Biểu diễn đồ thị bởi ma trận kề

Trong các thuật toán đồ thị sẽ trình bày sau này, chúng ta không quan tâm tới các thông tin về các đỉnh, vì vậy chỉ cần cho mỗi đỉnh một tên gọi để phân biệt nó với các đỉnh khác. Do đó, với một đồ thị N đỉnh ta luôn luôn xem tập các đỉnh của nó V = {0, 1, 2, …, N-1}.

Trong cách biểu diễn đồ thị bởi ma trận kề, đồ thị N đỉnh được lưu trong mảng A hai chiều cỡ N, trong đó

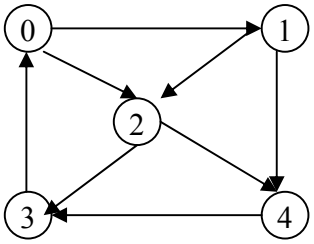
A[u][v] = 1 nếu có cung (u,v)

A[u][v] = 0 nếu không có cung (u,v)

Chẳng hạn, đồ thị trong hình 18.2.a được biểu diễn bởi ma trận kề trong hình 18.2.b. Nếu đồ thị là đồ thị có trọng số thì thay cho mảng bool ta sử dụng mảng các số, trong đó A[u][v] sẽ lưu trọng số của cung u→v.

Như vậy, ta có thể biểu diễn đồ thị N đỉnh bởi mảng Graph được xác định như sau:

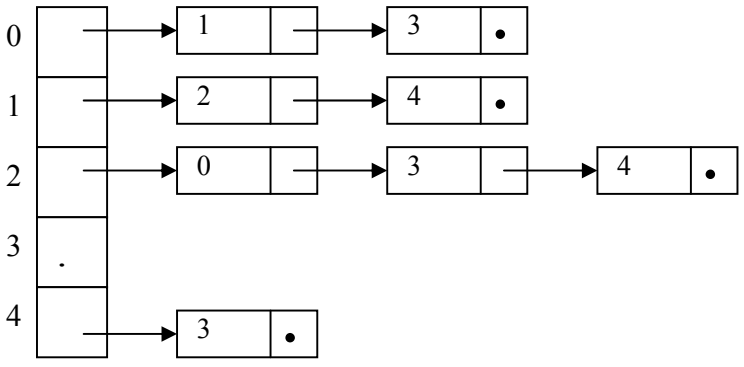
```
const   int   N =…;
typedef   bool   Graph[N][N];
```



(a)

	0	1	2	3	4
0	0	1	0	1	0
1	0	0	1	0	1
2	1	0	0	1	1
3	0	0	0	0	0
4	0	0	0	1	0

(b)



(c)

Hình 18.2. Biểu diễn đồ thị bởi ma trận kề và danh sánh kề.

18.2.2 Biểu diễn đồ thị bởi danh sách kề

Trong cách biểu diễn này, với mỗi đỉnh ta lập một danh sách các đỉnh kề đỉnh đó. Các danh sách này có thể có độ dài rất khác nhau, vì vậy ta tổ chức danh sách này dưới dạng danh sách liên kết, mỗi thành phần của danh sách này sẽ chứa số hiệu của một đỉnh kề và con trỏ trỏ tới thành phần đi sau. Chúng ta sẽ sử dụng một mảng A lưu các con trỏ trỏ tới đầu mỗi danh sách, trong đó A[i] lưu con trỏ trỏ tới đầu danh sách các đỉnh kề với đỉnh i. Chẳng hạn, đồ thị trong hình 18.2.a. được biểu diễn bởi cấu trúc dữ liệu trong hình 18.2.c.

Cấu trúc dữ liệu biểu diễn đồ thị bằng danh sách kề được mô tả như sau:

```
struct     Cell
{
    int    vertex;
    Cell *   next;
};
const   int  N =…;
typedef   Cell*   Graph[N];
```

Chú ý rằng, nếu đồ thị là đồ thị có trọng số thì trong cấu trúc Cell ta cần thêm vào một biến để lưu trọng số của cung.

So sánh hai phương pháp biểu diễn đồ thị

Ưu điểm của phương pháp biểu diễn đồ thị bởi ma trận kề là, bằng cách truy cập tới thành phần A[i][j] của mảng ta biết ngay được có cung (i,j) hay không và độ dài của cung đó (nếu là đồ thị có trọng số). Nhưng phương pháp này đòi hỏi mảng cần có N x N thành phần nếu đồ thị có N đỉnh. Do đó sẽ lãng phí bộ nhớ khi mà số đỉnh N lớn, nhưng đồ thị chỉ có ít cung. Trong trường hợp này, nếu biểu diễn đồ thị bằng danh sách kề ta sẽ tiết kiệm được bộ nhớ. Tuy nhiên, trong cách biểu diễn đồ thị bởi danh sách kề, muốn biết

có cung (i,j) hay không và độ dài của nó bằng bao nhiêu, ta lại phải tiêu tốn thời gian để duyệt danh sách các đỉnh kề của đỉnh i.

18.3 ĐI QUA ĐỒ THỊ

Đi qua đồ thị (hay còn gọi là duyệt đồ thị) có nghĩa là ta cần "thăm" tất cả các đỉnh và cung của đồ thị theo một trật tự nào đó. Giải quyết nhiều vấn đề của lý thuyết đồ thị đòi hỏi ta cần phải duyệt đồ thị. Vì vậy, các kỹ thuật đi qua đồ thị đóng vai trò quan trọng trong việc thiết kế các thuật toán đồ thị. Chẳng hạn, bằng cách duyệt đồ thị, ta có thể đưa ra thuật giải cho các vấn đề: đồ thị có chu trình hay không? Đồ thị có liên thông không? Từ đỉnh u bất kỳ ta có thể đi tới đỉnh v bất kỳ khác hay không?

Có hai kỹ thuật đi qua đồ thị: đi qua đồ thị theo bề rộng và đi qua đồ thị theo độ sâu.

18.3.1 Đi qua đồ thị theo bề rộng

Việc đi qua đồ thị theo bề rộng được thực hiện bằng cách sử dụng kỹ thuật tìm kiếm theo bề rộng (Breadth-First Search). Ý tưởng của tìm kiếm theo bề rộng xuất phát từ đỉnh v là như sau. Từ đỉnh v ta lần lượt đi thăm tất cả các đỉnh u kề đỉnh v mà u chưa được thăm. Sau đó, đỉnh nào được thăm trước thì các đỉnh kề nó cũng sẽ được thăm trước. Quá trình trên sẽ được tiếp tục cho tới khi ta không thể thăm đỉnh nào nữa. Ta cần quan tâm tới các đặc điểm sau của kỹ thuật này:

Tại mỗi bước, từ một đỉnh đã được thăm, ta đi thăm tất cả các đỉnh kề đỉnh đó (tức là thăm theo bề rộng).

Trật tự các đỉnh được thăm là: đỉnh nào được thăm trước thì các đỉnh kề của nó cũng phải được thăm trước.

Để lưu lại vết của các đỉnh đã được thăm, chúng ta sử dụng một hàng đợi Q. Mỗi khi đến thăm một đỉnh thì đỉnh đó được xen vào đuôi hàng đợi Q. Thuật toán tìm kiếm theo bề rộng xuất phát từ đỉnh v được biểu diễn bởi hàm BFS(v) (viết tắt của cụm từ Breadth-First Search)

```
            BFS(v)
            //Tìm kiếm theo bề rộng xuất phát từ v.
{
 (1)  Khởi tạo hàng đợi Q rỗng;
 (2)  Đánh dấu đỉnh v đã được thăm;
 (3)  Xen v vào hàng đợi Q;
 (4)  while (hàng đợi Q không rỗng)
      {
 (5)       Loại đỉnh w ở đầu hàng đợi Q;
 (6)       for (mỗi đỉnh u kề w)
 (7)            if ( u chưa được thăm)
                 {
 (8)                 Đánh dấu u đã được thăm;
 (9)                 Xen u vào đuôi hàng đợi Q;
                 }
      }  // hết vòng lặp while.
}
```

Sử dụng hàm BFS ta có thể dễ dàng đi qua đồ thị. Đầu tiên, tất cả các đỉnh của đồ thị được đánh dấu chưa được thăm. Lấy đỉnh v bất kỳ làm đỉnh xuất phát, sử dụng BFS(v) để thăm các đỉnh. Sau đó nếu còn có đỉnh chưa được thăm, ta lại chọn một đỉnh bất kỳ trong số các đỉnh đó làm đỉnh xuất phát để đi thăm. Tiếp tục cho tới khi tất cả các đỉnh của đồ thị đã được thăm. Sau đây là thuật toán đi qua đồ thị G theo bề rộng.

```
          BFS-Traversal (G)
          // Đi qua đồ thị G=(V, E) theo bề rộng
{
 (10)  for (mỗi v ∈V)
 (11)       Đánh dấu v chưa được thăm;
 (12)  for (mỗi v ∈V)
 (13)       if (v chưa được thăm)
 (14)            BFS(v);
}
```

Đánh dấu các đỉnh chưa thăm, đã thăm bằng cách nào? Giả sử đồ thị có N đỉnh và các đỉnh của đồ thị được đánh số từ 0 đến N-1. Khi đó ta chỉ

cần sử dụng mảng bool d cỡ N, để đánh dấu đỉnh v chưa thăm (đã thăm) ta chỉ cần đặt d[v] = false (d[v] = true). Tuy nhiên, trong các ứng dụng cụ thể, ta cần sử dụng mảng d để ghi lại các thông tin ích lợi hơn.

Phân tích thuật toán đi qua đồ thị theo bề rộng.

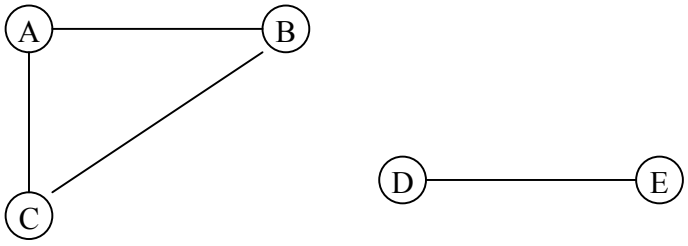
Thời gian thực hiện các dòng lệnh (10), (11) là O(|V|). Thời gian thực hiện các dòng lệnh (12) – (14) là tổng thời gian thực hiện các lời gọi hàm BFS(v). Thời gian chạy của BFS(v) là thời gian thực hiện vòng lặp (4). Chú ý rằng, mỗi đỉnh được đưa vào hàng đợi (dòng lệnh (3) và (9)) và bị loại khỏi hàng đợi (dòng lệnh (5)) đúng một lần. Với mỗi đỉnh w khi bị loại khỏi hàng đợi, ta cần thực hiện lệnh (6), tức là cần xem xét tất cả các cung (w,u). Nếu đồ thị được cài đặt bởi danh sách kề, thì khi thực hiện các lời gọi hàm BFS(v), thời gian truy cập tới các cung của đồ thị là O(|E|). Tóm lại, thực hiện các lời gọi hàm BFS(v) ta cần thực hiện một số hành động với tất cả các đỉnh và cung của đồ thị. Với mỗi đỉnh, ta cần thực hiện các hành động (5), (8), (9) với thời gian O(1). Với mỗi cung (w,u), ta chỉ cần kiểm tra xem u đã thăm hay chưa (dòng (13)). Do đó tổng thời gian thực hiện các lời gọi hàm BFS(v) trong vòng lặp (12) là O(|V| + |E|). Như vậy, thuật toán đi qua đồ thị G = (V,E) có thời gian chạy là O(|V| + |E|) trong đó |V| là số đỉnh, còn |E| là số cung của đồ thị.

Bây giờ, chúng ta đưa ra một vài ứng dụng của kỹ thuật đi qua đồ thị theo bề rộng.

Vấn đề đạt tới. Giả sử v và w là hai đỉnh bất kỳ, ta muốn biết từ đỉnh v có đường đi tới đỉnh w hay không? Nếu có đường đi từ v tới w thì đỉnh w được gọi là đỉnh đạt tới từ v. Dễ dàng thấy rằng, khi xuất phát từ đỉnh v thì sử dụng hàm BFS(v) có thể đến thăm tất cả các đỉnh đạt tới từ v. Ban đầu tất cả các đỉnh được đánh dấu là chưa thăm, rồi gọi hàm BFS(v). Nếu w được đánh dấu đã thăm thì ta kết luận w đạt tới từ v. Bằng cách này, nếu đồ thị không có trọng số thì không những ta có thể biết được đỉnh w có đạt tới từ đỉnh v không, mà trong trường hợp w là đỉnh đạt tới, ta còn tìm được đường đi ngắn nhất từ v tới w (bài tập)

Tính liên thông và thành phần liên thông của đồ thị vô hướng.

Một đồ thị vô hướng được gọi là liên thông nếu có đường đi giữa hai đỉnh bất kì. Nếu đồ thị vô hướng không liên thông, thì mỗi đồ thị con liên thông cực đại là một thành phần liên thông. Chẳng hạn, đồ thị vô hướng trong hình 18.3. có hai thành phần liên thông, một thành phần liên thông là các đỉnh {A,B,C}, và một thành phần liên thông khác là {D,E}.



Hình 18.3. Thành phần liên thông của đồ thị vô hướng.

Không khó khăn thấy rằng, lời gọi hàm BFS(v) cho phép ta xác định thành phần liên thông chứa đỉnh v. Do đó, sử dụng tìm kiếm theo bề rộng, bạn đọc dễ dàng đưa ra thuật toán cho phép xác định một đồ thị vô hướng có liên thông hay không, nếu không thì đồ thị có mấy thành phần liên thông, và mỗi thành phần liên thông gồm các đỉnh nào (Bài tập).

18.3.2 Đi qua đồ thị theo độ sâu

Để đi qua đồ thị theo độ sâu chúng ta cần đến kỹ thuật tìm kiếm theo độ sâu (Depth-First Search). Ý tưởng của tìm kiếm theo độ sâu xuất phát từ đỉnh u bất kỳ của đồ thị là như sau. Từ đỉnh u ta đến thăm một đỉnh v kề đỉnh u, rồi lại từ đỉnh v ta đến thăm đỉnh w kề v, và cứ thế tiếp tục chừng nào có thể được (tức là luôn luôn đi sâu xuống thăm). Khi đạt tới đỉnh v mà tại v ta không đi thăm tiếp được thì ta quay lại đỉnh u và từ đỉnh u ta đi thăm đỉnh v' khác kề u (nếu có), rồi từ v' lại đi thăm tiếp đỉnh kề v',… Quá trình trên sẽ tiếp diễn cho tới khi ta không thể tới thăm đỉnh nào nữa. Quá trình trên sẽ đảm bảo rằng, đỉnh nào được thăm sau thì các đỉnh kề của nó sẽ được thăm trước.

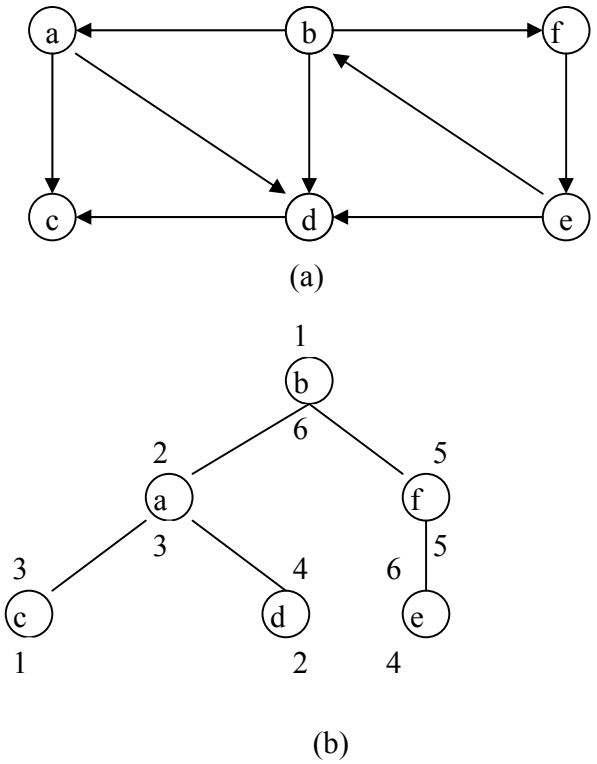
Thuật toán tìm kiếm theo độ sâu xuất phát từ đỉnh u được mô tả bởi hàm DFS(u) (viết tắt của cụm từ Depth-First Search). Có thể biểu diễn hàm DFS(u) bởi hàm không đệ quy bằng cách sử dụng một ngăn xếp để lưu vết của các đỉnh trong quá trình đi thăm. Cụ thể là, nếu ta đang ở thăm đỉnh v thì ngăn xếp sẽ lưu các đỉnh trên đường đi từ đỉnh xuất phát u đã dẫn ta đến đỉnh v. Hàm không đệ quy DFS(u) được viết tương tự như hàm tìm kiếm theo độ sâu không đệ quy trên cây (bài tập). Thay cho sử dụng ngăn xếp, để đảm bảo đỉnh nào được thăm sau thì các đỉnh kề của nó phải được thăm trước, ta có thể sử dụng các lời gọi đệ quy. Hàm đệ quy DFS(u) sẽ chứa các dòng lệnh sau:

```
for (mỗi đỉnh v kề u)
   if (v chưa được thăm)
            DFS(v);      // Gọi đệ quy thăm theo độ sâu  xuất phát từ v
```

Chúng ta sẽ sử dụng mảng T để đánh dấu các đỉnh chưa thăm hoặc đã thăm. Để đánh dấu đỉnh v chưa thăm, ta đặt T[v] = 0, và nếu v đã được thăm thì T[v] sẽ lưu một giá trị nào đó > 0. Chúng ta sẽ dùng T[v] để lưu thời điểm mà v được đến thăm (thời điểm được kể từ 1, 2, …). Bên cạnh mảng T, chúng ta sử dụng mảng S, trong đó S[v] sẽ lưu thời điểm mà ta đã hoàn thành thăm tất cả các đỉnh đạt tới từ đỉnh v (thời điểm này cũng kể từ 1, 2, …).

Ví dụ. Giả sử ta tìm kiếm theo độ sâu trên đồ thị hình 18.4.a. xuất phát từ đỉnh b. Khi đó T[b] = 1. Đi theo cung (b,a) để thăm đỉnh a, nên T[a] = 2. Đi theo cung (a,c) để thăm đỉnh c, T[c] = 3. Lúc này không thể từ c đi thăm tiếp, nên S[c] = 1. Quay lại đỉnh a, theo cung (a,d) đến thăm d, T[d] = 4. Từ d không đi thăm tiếp được đỉnh nào nữa, do đó S[d] = 2…Khi thực hiện tìm kiếm theo độ sâu từ đỉnh v thì một cây gốc v được tạo thành. Trong cây này, nếu ta đi theo cung (a,b) để tới thăm đỉnh b, thì đỉnh b là con của đỉnh a trong cây. Một điều cần lưu ý là, trong cây này T[v] chính là số thứ tự trước của đỉnh v khi ta đi qua cây theo thứ tự trước, còn S[v] là số thứ tự sau của v khi ta đi qua cây theo thứ tự sau. Chẳng hạn, khi tìm kiếm theo độ sâu

trên đồ thị 18.4.a. ta có cây trong hình 18.4.b, trong đó T[v] được ghi trên đỉnh v, còn S[v] được ghi dưới v.



Hình 18.4. **Cây tạo thành khi tìm kiếm theo độ sâu**.

Thuật toán đi qua đồ thị theo độ sâu bắt đầu bằng việc đánh dấu tất cả các đỉnh chưa được thăm. Sử dụng biến i để đếm thời điểm đến thăm mỗi đỉnh và biến k để đếm thời điểm đã thăm hết các đỉnh kề của mỗi đỉnh.

Thuật toán lựa chọn đỉnh u bất kỳ làm đỉnh xuất phát, và gọi hàm DFS(u) để thực hiện tìm kiếm theo độ sâu từ đỉnh u. Sau khi hoàn thành DFS(u), nếu còn có đỉnh chưa được thăm, thì một đỉnh xuất phát mới được lựa chọn và tiếp tục tìm kiếm theo độ sâu từ đỉnh đó. Việc đánh số thứ tự trước (bởi mảng T) và đanh số thứ tự sau (bởi mảng S) được thực hiện trong hàm DFS().

```
DFS-Traversal(G)
//Đi qua đồ thị G = (V,E) theo độ sâu
```

```
{
  for (mỗi đỉnh u ∈ V)
      {
           T[u]  =  0;   // Đánh dấu u chưa thăm.
           S[u] = 0;
      }
   int  i  =  0;
   int  k  =  0;
   for (mỗi đỉnh u ∈ V)
        if ( T[u] = = 0)    // Đỉnh u chưa được thăm.
             DFS(u);
}

           DFS(u)
           // Tìm kiếm theo độ sâu từ đỉnh v
{
    i++;
    T[u]  =  i;
    for (mỗi đỉnh v kề u)
         if (T[v] == 0)
              DFS(v);
     k++;
     S[u]  =  k;
}
```

Dễ dàng thấy rằng, thời gian chạy của thuật toán đi qua đồ thị theo độ sâu cũng là O(|V|+|E|). Bởi vì với mỗi đỉnh u ∈ V, hàm DFS(u) được gọi đúng một lần, và khi gọi DFS(u) ta cần xem xét tất cả các cung (u,v) (vòng lặp for (mỗi đỉnh v kề u)).

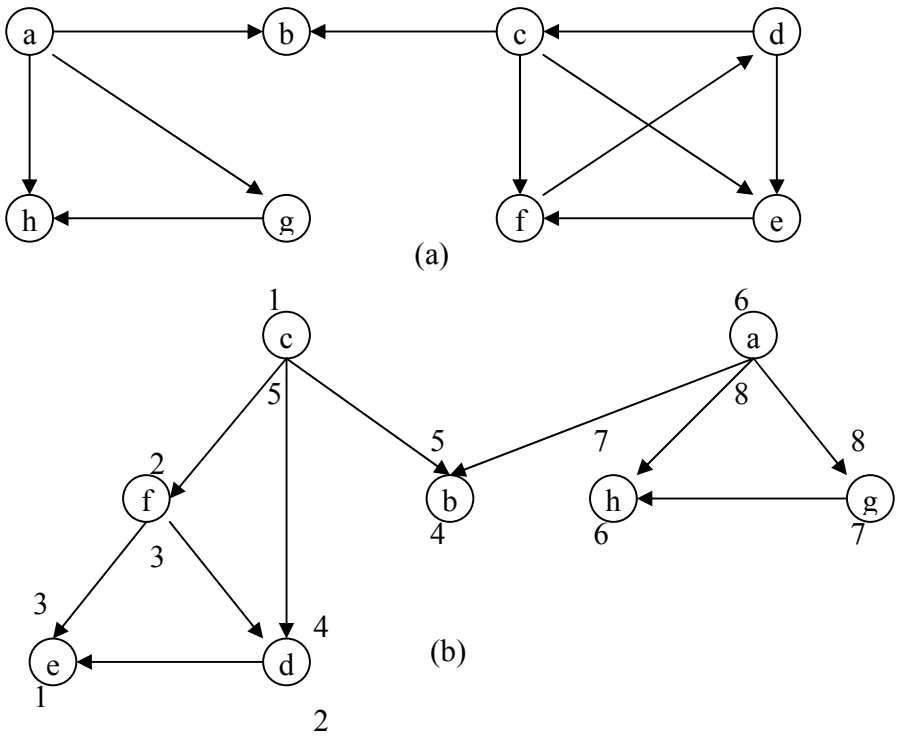
Tại sao khi đi qua đồ thị theo độ sâu chúng ta đã sử dụng hai cách đánh số các đỉnh: đánh số theo thứ tự trước (mảng T) và đánh số theo thứ tự sau (mảng S)? Lý do là các cách đánh số này sẽ giúp ta phân lớp các cung của đồ thị. Sử dụng sự phân lớp các cung của đồ thị sẽ giúp ta phát hiện ra nhiều tính chất quan trọng của đồ thị, chẳng hạn, phát hiện ra đồ thị có chu trình hay không.

Phân lớp các cung

Khi tìm kiếm theo độ sâu xuất phát từ đỉnh v thì một cây gốc v được tạo thành. Do đó khi ta đi qua đồ thị theo độ sâu thì một rừng cây được tạo thành. Trong rừng cây này, các cung của đồ thị được phân thành bốn lớp sau:

- Các cung cây: Đó là các cung liên kết các đỉnh trong một cây
- Các cung tiến: Đó là các cung (u,v) trong đó u và v nằm trong cùng một cây và u là tổ tiên của v.
- Các cung ngược: Đó là các cung (u,v), trong đó u và v nằm trong cùng một cây và u là con cháu của v.
- Các cung xiên: Đó là các cung (u,v), trong đó u và v nằm trong hai cây khác nhau, hoặc chúng nằm trong cùng một cây nhưng u không phải là tổ tiên cũng không phải là con cháu của v.

Ví dụ. Xét đồ thị trong hình 18.5.a. Đầu tiên ta tìm kiếm theo độ sâu xuất phát từ đỉnh c, sau đó trong số các đỉnh không đạt tới từ c, ta chọn đỉnh a làm đỉnh xuất phát để đi thăm tiếp. Kết quả ta thu được hai cây trong hình 18.5.b. Với hai cây này, các cung (c,f), (f,c), (f,d), (c,b), (a,h), (a,g) là các cung cây. Cung (c,e) là cung tiến, cung (d,e) là cung ngược. Các cung (d,e), (a,b), (g,h) là các cung xiên. Cần lưu ý rằng, rừng cây được tạo thành khi đi qua đồ thị không phải là duy nhất, vì nó phụ thuộc vào sự lựa chọn các đỉnh xuất phát, và do đó sự phân lớp các cung cũng không phải là duy nhất.



Hình 18.5. Đi qua đồ thị theo độ sâu và phân lớp các cung.

Chúng ta dễ dàng bổ xung thêm vào hàm DFS() các lệnh cần thiết để gắn nhãn các cung của đồ thị, bằng cách sử dụng các luật sau đây. Giả sử ta đang ở đỉnh u (khi đó T[u] ≠ 0) và đi theo cung (u,v) để đến v, khi đó ta có các luật sau:

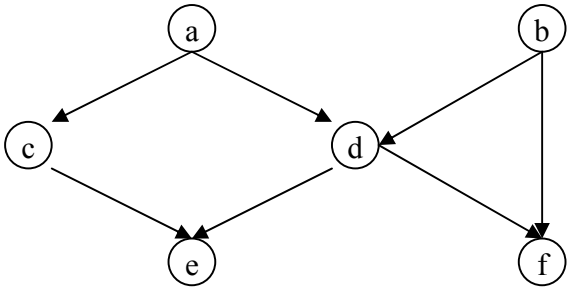
- Nếu T[v] = 0 (tức v chưa được thăm) thì (u,v) là cung cây.
- Nếu T[v] ≠ 0 (tức v đã được thăm) và S[v] = 0 (chưa hoàn thành thăm các đỉnh kề v) thì (u,v) là cung ngược.
- Nếu T[v] ≠ 0 và S[v] ≠ 0 và T[u] < T[v] thì (u,v) là cung tiến.
- Nếu T[v] ≠ 0 và S[v] ≠ 0 và T[u] > T[v] thì (u,v) là cung xiên.

Chúng ta có nhận xét rằng, nếu (u,v) là cung cây, cung tiến hoặc cung xiên thì S[u] > S[v]. Có thể thấy điều này trong sự phân lớp các cung trong hình 18.5.b, ở đó S[u] được ghi dưới mỗi đỉnh u.

Có thể chứng minh được rằng, đồ thị không có chu trình nếu và chỉ nếu nó không có cung ngược. Vì vậy, bằng cách đi qua đồ thị theo độ sâu và phân lớp các cung, nếu không phát hiện ra cung ngược thì đồ thị không có chu trình.

18.4 ĐỒ THỊ ĐỊNH HƯỚNG KHÔNG CÓ CHU TRÌNH VÀ SẮP XẾP TOPO

Một lớp đồ thị quan trọng là các đồ thị định hướng không có chu trình. Hình 18.6 là một ví dụ của đồ thị định hướng không có chu trình. Nó được gọi tắt là DAG (viết tắt của cụm từ Directed Acylic Graph). DAG là trường hợp riêng của đồ thị định hướng, nhưng tổng quát hơn khái niệm cây.



Hình 18.6. Đồ thị định hướng không có chu trình.

Nhiều dạng quan hệ trên một tập đối tượng có thể biểu diễn bởi DAG. Chẳng hạn, quan hệ thứ tự bộ phận trên một tập A có thể biểu diễn bởi DAG, trong đó mỗi phần tử của A là một đỉnh của đồ thị, và nếu a < b thì trong đồ thị sẽ có cung từ đỉnh a đến b. Do tính chất của quan hệ thứ tự bộ phận, đồ thị này không có chu trình, do đó nó là một DAG.

Giả sử chúng ta có một đề án bao gồm nhiều nhiệm vụ. Trong quá trình thực hiện, một nhiệm vụ có thể chỉ được bắt đầu thực hiện khi một số nhiệm vụ khác đã hoàn thành (dễ thấy điều này ở các đề án thi công). Khi đó ta có thể sử dụg DAG để biểu diễn đề án. Mỗi nhiệm vụ là một đỉnh của đồ thị. Nếu nhiệm vụ A cần phải được hoàn thành trước khi nhiệm vụ B bắt đầu

thực hiện, thì trong đồ thị sẽ có cung đi từ đỉnh A đến đỉnh B. Giả sử tại mỗi thời điểm ta chỉ thực hiện được một nhiệm vụ, làm xong một nhiệm vụ mới có thể bắt đầu làm nhiệm vụ khác. Như vậy ta phải sắp xếp các nhiệm vụ để thực hiện sao cho thoả mãn các đòi hỏi về thời gian giữa các nhiệm vụ.

Vấn đề sắp xếp topo (topological sort) được đặt ra như sau. Cho G = (V,E) là một DAG, ta cần sắp xếp các đỉnh của đồ thị thành một danh sách (chúng ta sẽ gọi là danh sách topo), sao cho nếu có cung (u,v) thì u cần phải đứng trước v trong danh sách đó. Ví dụ, với DAG trong hình 18.6 thì danh sách topo là (A, C, B, D, E, F). Cần lưu ý rằng, danh sách topo không phải là duy nhất. Chẳng hạn, một danh sách topo khác của DAG hình 18.6 là (A, B, D, C, E, F)

Trong mục 18.5.2, chúng ta đã chỉ ra rằng, có thể sử dụng kỹ thuật đi qua đồ thị theo độ sâu để phát hiện ra đồ thị là có chu trình hay không. Sau đây ta sẽ sử dụng kỹ thuật đi qua đồ thị theo độ sâu để sinh ra một danh sách topo của đồ thị định hướng không có chu trình.

Nhớ lại rằng đồ thị không có chu trình, thì trong rừng cây được tạo thành khi đi qua đồ thị theo độ sâu chỉ có ba loại cung: cung cây, cung tiến và cung xiên. Mặt khác, nếu (u,v) là một trong ba loại cung đó, thì S[u] > S[v] (trong đó, S[u] là số thứ tự sau của đỉnh u khi ta đi qua đồ thị theo độ sâu). Như vậy, S[u] là cách đánh số các đỉnh trong danh sách topo theo thứ tự ngược lại, Từ đó ta dễ dàng đưa ra thuật toán sắp xếp topo.

Thuật toán sắp xếp topo (TopoSort) sau đây sẽ sử dụng hàm đệ quy TPS(u), hàm này thực chất là hàm tìm kiếm theo độ sâu DFS(u), chỉ khác là thay cho việc đánh số thứ tự sau S[u], ta ghi u vào đầu danh sách topo

```
        TopoSort(G)
//Sắp xếp các đỉnh của đồ thị định hướng
//không có chu trình G =(V,E) thành danh sách topo.
{
    for (mỗi đỉnh u ∈ V)
            Đánh dấu u chưa được thăm;
    Khởi tạo danh sách topo L rỗng;
```

```
    for (mỗi đỉnh u ∈ V)
            if (u chưa thăm)
                    TPS(u);
}

        TPS(u)
{
    Đánh dấu u đã thăm;
    for (mỗi đỉnh v kề u)
            if ( v chưa thăm)
                    TPS(v);
    Xen  u  vào đầu danh sách L;
}
```

18.5 ĐƯỜNG ĐI NGẮN NHẤT

Chúng ta đã chỉ ra trong mục 18.3.1 rằng, đối với đồ thị không có trọng số, ta có thể sử dụng kỹ thuật tìm kiếm theo bề rộng để tìm đường đi ngắn nhất từ một đỉnh tới các đỉnh khác. Đối với đồ thị có trọng số, vấn đề tìm đường đi ngắn nhất sẽ khó khăn, phức tạp hơn.

Trong mục này chúng ta sẽ trình bày các thuật toán tìm đường đi ngắn nhất trong đồ thị có trọng số, với trọng số của các cung là các số không âm, đó cũng là trường hợp hay gặp nhất trong các ứng dụng. Chúng ta sẽ giả thiết rằng, G = (V,E) là đồ thị có trọng số, tập đỉnh V = { 0,1, …, n-1} và độ dài của cung (u, v) là số c(u,v) >= 0, nếu không có cung (u,v) thì c(u,v) = ∞. Nhắc lại rằng, nếu (v_0, v_1,…, v_k), k >= 1, là đường đi từ đỉnh v_0 tới đỉnh v_k thì độ dài của đường đi này là tổng độ dài của các cung trên đường đi.

Chúng ta xét hai vấn đề sau:

- Tìm đường đi ngắn nhất từ một đỉnh nguồn tới các đỉnh còn lại.
- Tìm đường đi ngắn nhất giữa mọi cặp đỉnh của đồ thị.

18.5.1 Đường đi ngắn nhất từ một đỉnh nguồn

Thuật toán được trình bày sau đây là thuật toán Dijkstra (mang tên E. Dijkstra, người phát minh ra thuật toán). Thuật toán này được thiết kế dựa vào kỹ thuật tham ăn.

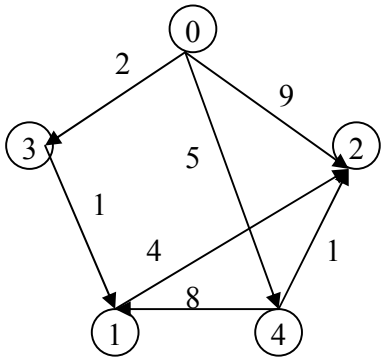
Ta xác định đường đi ngắn nhất từ đỉnh nguồn s tới các đỉnh còn lại qua các bước, mỗi bước ta xác định đường đi ngắn nhất từ nguồn tới một đỉnh. Ta lưu các đỉnh đã xác định đường đi ngắn nhất từ nguồn tới chúng vào tập S. Ban đầu tập S chỉ chứa một đỉnh nguồn s. Chúng ta sẽ gọi đường đi từ nguồn s tới đỉnh v là đường đi đặc biệt, nếu đường đi đó chỉ đi qua các đỉnh trong S, tức là các đường đi ($s = v_0, v_1,\ldots,v_{k-1},v_k = v$), trong đó $v_0, v_1, \ldots v_{k-1} \in S$. Một mảng D được sử dụng để lưu độ dài của đường đi đặc biệt, D[v] là độ dài đường đi đặc biệt từ nguồn tới v. Ban đầu vì S chỉ chứa một đỉnh nguồn s, nên ta lấy D[s] = 0, và D[v] = c(s,v) với mọi $v \neq s$. Tại mỗi bước ta sẽ chọn một đỉnh u không thuộc S mà D[u] nhỏ nhất và thêm u vào S, ta xem D[u] là độ dài đường đi ngắn nhất từ nguồn tới u (sau này ta sẽ chứng minh D[u] đúng là độ dài đường đi ngắn nhất từ nguồn tới u). Sau khi thêm u vào S, ta xác định lại các D[v] với v ở ngoài S. nếu độ dài đường đi đặc biệt qua đỉnh u (vừa được chọn) để tới v nhỏ hơn D[v] thì ta lấy D[v] là độ dài đường đi đó. Bước trên đây được lặp lại cho tới khi S gồm tất cả các đỉnh của đồ thị, và lúc đó mảng D[u] sẽ lưu độ dài đường đi ngắn nhất từ nguồn tới u, với mọi $u \in V$.

```
                Dijktra (G,s)
//Tìm đường đi ngắn nhất trong đồ thị G = (V,E) từ đỉnh nguồn s
{
        Khởi tạo tập S chỉ chứa đỉnh nguồn s;
(1)     for (mỗi đỉnh v∈V)
                D[v] = c(s,v);
        D[s] = 0;
        while (V – S ≠ ∅ )
           {
(2)            chọn đỉnh u ∈ V - S mà D[u] nhỏ nhất;
               S = S ∪ {u};      // bổ sung u vào S
```

```
                for ( mỗi v ∈ V - S)     // xác định lại D[v]
(3)                     if (D[u] + c(u,v) < D[v])
                                D[v] = D[u] + c(u,v);
        }
}
```

Trong thuật toán trên đây, chúng ta mới sử dụng mảng D để ghi lại độ dài đường đi ngắn nhất từ nguồn tới các đỉnh khác. Muốn lưu lại vết của đường đi, ta sử dụng thêm mảng P, trong đó P[v] = u nếu cung (u,v) nằm trên đường đi đặc biệt. Khi khởi tạo, trong vòng lặp (1) ta cần thêm lệnh P[v] = s (vì ta đã đi tới v từ nguồn s). Khi xác định lại D[v], ta cần xác định lại P[v], trong lệnh (3) cần thêm lệnh P[v] = u.

Ví dụ. Xét đồ thị định hướng trong hình 18.7a. Chúng ta cần tìm đường đi ngắn nhất từ đỉnh nguồn là đỉnh 0. Kết quả các bước của thuật toán Dijkstra áp dụng cho đồ thị đó được cho trong 18.7b. Trong đó, bước khởi tạo được cho trong dòng đầu tiên. Thực hiện bước 1, trong D[v], với v=1, 2, 3, 4 ở dòng đầu tiên, thì D[3] = 2 là nhỏ nhất nên đỉnh 3 được chọn. Ta xác định lại các D[v] cho các đỉnh còn lại 1, 2 và 4. Chỉ có D[1] là nhỏ đi và D[1] = 3, vì D[3] + c(3,1) = 2 + 1 = 3 < ∞. Tiếp tục thực hiện các bước 2, 3, 4 ta thu được độ dài đường đi ngắn nhất từ đỉnh 0 tới các đỉnh 1, 2, 3, 4 được cho ở dòng cuối cùng trong bảng, chẳng hạn độ dài đường đi ngắn nhất từ 0 tới 2 là D[2] = 6 và đó là độ dài của đường đi 0→4→2.



(a)

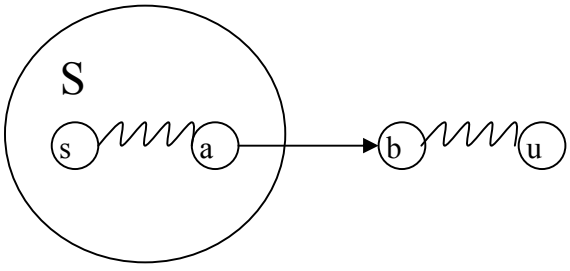
Bước	Đỉnh u được chọn	V-S	D[1]	D[2]	D[3]	D[4]
Khởi tạo	--	1, 2, 3, 4	∞	9	2	5
1	3	1, 2, 4	3	9	2	5
2	1	2, 4	3	7	2	5
3	4	2	3	6	2	5
4	2					

(b)

Hình 18.7. Minh hoạ các bước của thuật toán Dijkstra

Tính đúng đắn của thuật toán Dijkstra.

Chúng ta sẽ chứng minh rằng, khi kết thúc thuật toán, tức là khi S = V, thì D[u] sẽ là độ dài đướng đi ngắn nhất từ đỉnh nguồn tới u với mọi u ∈ S = V. Điều này được chứng minh bằng quy nạp theo cỡ của tập S. Khi S chỉ chứa đỉnh nguồn s thì D[s] = 0, đương nhiên giả thiết quy nạp đúng. Giả sử rằng tại một thời điểm nào đó ta đã có D[a] là độ dài đường đi ngắn nhất từ nguồn tới a, với mọi đỉnh a ∈ S, và u là đỉnh được chọn bởi lệnh (2) trong thuật toán để bổ sung vào S. Ta cần chứng minh rằng khi đó D[u] là độ dài đường đi ngắn nhất từ nguồn tới u. Mỗi khi bổ xung thêm vào tập S một đỉnh mới (lệnh(2)), thì các đỉnh v còn lại không nằm trong S được xác định lại D[v] bởi lệnh (3). Từ đó bằng quy nạp, dễ dàng chứng minh được nhận xét sau: Nếu a là đỉnh bất kỳ trong S và b là đỉnh bất kỳ ngoài S thì D[b] <= D[a] + c(a,b). Giả sử ta có một đường đi bất kỳ từ nguồn s tới u, độ dài của nó được ký hiệu là d(s,u). Giả sử trên đường đi đó a là đỉnh sau cùng ở trong S và b là đỉnh đầu tiên ở ngoài S như trong hình vẽ sau:



Theo cách chọn đỉnh u (lệnh(2)), D[u] là nhỏ nhất trong số các đỉnh ở ngoài S. Vì vậy, D[u] <= D[b]. Theo nhận xét đã đưa ra ở trên, D[b] <= D[a] + c(a,b). Mặt khác, theo giả thiết quy nạp, D[a] là độ dài đường đi ngắn nhất từ nguồn tới a. Do đó, nếu ký hiệu d(s,a) là độ dài đoạn đường từ s tới a, còn d(b,u) là độ dài đoạn đường từ b tới u, ta có

D[u] <= D[b]

<= D[a] + c(a,b)

<= d(s,a) + c(a,b)

<= d(s,a) + c(a,b) + d(b,s)

= d(s,u)

Như vậy ta đã chứng minh được D[u] nhỏ hơn hoặc bằng độ dài d(s,u) của đường đi bất kỳ từ nguồn s tới u.

Trong thuật toán Dijkstra, tại mỗi bước ta cần chọn một đỉnh u không nằm trong S mà D[u] là nhỏ nhất (lệnh (2)), và sau đó với các đỉnh v còn lại không nằm trong S, D[v] có thể bị giảm đi bởi lệnh (3). Vì vậy để cho lệnh (2) được thực hiện hiệu quả, ta có thể sử dụng hàng ưu tiên P (xem chương 12) để cài đặt tập đỉnh V - S với khoá của đỉnh v ∈ V - S là D[v]. Chúng ta có thuật toán sau:

```
                    Dijkstra(G,s)
{
 (1)   for (mỗi đỉnh v ∈ V)
             D[v] = c(s,v);
       D[s] = 0;
 (2)   Khởi tạo hàng ưu tiên P chứa các đỉnh v ≠ s với khoá là D[v];
 (3)   while (P không rỗng)
         {
 (4)       u = DeleteMin(P);
 (5)       for (mỗi đỉnh v kề u)
```

```
                    if (D[u] + c(u,v) < D[v])
                        {
                            D[v] = D[u] + c(u,v);
(6)                         DecreaseKey(P,v,D[v]);
                        }
                }
}
```

Thời gian chạy của thuật toán Dijkstra

Lệnh lặp (1) cần thời gian O(|V|), trong đó |V| ký hiệu số đỉnh của đồ thị. Ta có thể khởi tạo ra hàng ưu tiên P (lệnh(2)) với thời gian là O(|V|) (xem mục 10.3.2). Số lần lặp trong lệnh lặp (3) là |V|. Trong mỗi lần lặp, phép toán DeleteMin đòi hỏi thời gian O(log|V|); do đó, tổng thời gian thực hiện các lệnh (4) là O(|V|log|V|). Tổng số lần lặp trong các lệnh lặp (5) trong tất cả các lần lặp của lệnh lặp (3) tối đa là số cung của đồ thị |E|. Trong mỗi lần lặp đó, nhiều nhất là một phép toán DecreaseKey (lệnh(6)) được thực hiện. Phép toán DecreaseKey chỉ cần thời gian O(log(|V|) khi ta cài đặt hàng ưu tiên P bởi cây thứ tự bộ phận (xem 12.2). Vì vậy, thời gian thực hiện các lệnh lặp (5) trong các lần lặp của lệnh lặp (3) là O(|E|log|V|). Tổng kết lại, thời gian chạy của thuật toán Dijkstra, nếu ta sử dụng hàng ưu tiên được cài đặt bởi cây thứ tự bộ phận, là O(|V|log|V| + |E|log|V|).

18.5.2 Đường đi ngắn nhất giữa mọi cặp đỉnh

Để tìm đường đi ngắn nhất giữa mọi cặp đỉnh chúng ta có thể cho chạy thuật toán Dijkstra với đỉnh nguồn trong mỗi lần chạy là một đỉnh của đồ thị. Do đó thời gian tìm đường đi ngắn nhất giữa mọi cặp đỉnh của đồ thị bằng sử dụng thuật toán Dijkstra sẽ là $O(|V|^2\log|V| + |V||E|\log|V|)$.

Bây giờ chúng ta trình bày thuật toán Floyd, thuật toán này được thiết kế dựa trên kỹ thuật quy hoạch động. Giả sử đồ thị có n đỉnh được đánh số từ 0 đến n-1, V = {0,1,..,n-1}, và độ dài cung (i,j) là c(i,j). Ta ký hiệu S_k là

tập các đỉnh từ 0 đến k, S_k = {0,1, …,k}, k <= n-1. Gọi $A_k(i,j)$ là độ dài đường đi ngắn nhất từ đỉnh i tới đỉnh j nhưng chỉ đi qua các đỉnh trong tập S_k. Khi k = n-1, thì S_{k-1} = V và do đó $A_{n-1}(i,j)$ chính là đường đi ngắn nhất từ i tới j trong đồ thị đã cho.

Trong trường hợp đơn giản nhất, khi S_k rỗng (k = -1), $A_{-1}(i,j)$ là độ dài đường đi ngắn nhất từ i đến j nhưng không qua đỉnh nào cả, do đó $A_{-1}(i,j)$ là độ dài của cung (i,j), tức là $A_{-1}(i,j) = c(i,j)$. Giả sử ta đã tính được các $A_0(i,j)$, $A_1(i,j)$,…, $A_{k-1}(i,j)$, ta tìm cách tính các $A_k(i,j)$ thông qua các $A_{k-1}(i,j)$. Trước hết ta có nhận xét rằng, nếu đỉnh k nằm trên đường đi ngắn nhất từ đỉnh i tới đỉnh j, thì đoạn đường từ i tới k và đoạn đường từ k tới j phải là đường đi ngắn nhất từ i tới k và từ k tới j tương ứng. Nếu $A_k(i,j)$ là độ dài đường đi không qua đỉnh k, tức là đường đi này chỉ đi qua các đỉnh trong S_{k-1} thì $A_k(i,j) = A_{k-1}(i,j)$. Nếu $A_k(i,j)$ là độ dài của đường đi qua đỉnh k, thì trên đường đi này đoạn từ i tới k có độ dài là $A_{k-1}(i,k)$, còn đoạn đường từ k tới j có độ dài là $A_{k-1}(k,j)$ (do nhận xét đã đưa ra). Do đó, ta có thể tính $A_k(i,j)$ bởi công thức sau:

$$A_k(i,j) = \min(A_{k-1}(i,j) , A_{k-1}(i,k) + A_{k-1}(k,j))$$

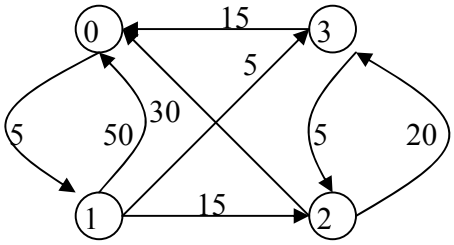
Sử dụng mảng A[i][j] để lưu độ dài đường đi ngắn nhất từ i đến j, ban đầu được khởi tạo là c(i,j). Sau đó với mỗi k = 0, 1, .., n-1 ta cập nhật A[i][j] theo công thức tính $A_k(i,j)$, ta có thuật toán sau

```
                    Floyd (G)
// Thuật toán tìm đường đi ngắn nhất giữa mọi cặp
// đỉnh trên đồ thị G = (V,E), V = {0,1,…,n-1}
{
(1)   for ( i = 0 ; i < n ; i++)
          for ( j = 0 ; j < n ; j++)
(2)             A[i][j] = c[i][j];
(3)   for ( k = 0 ; k < n ; k++)
          for ( i = 0 ; i < n ; i++)
              for ( j = 0 ; j < n ; j++)
(4)               if (A[i][k] + A[k][j] < A[i][j] )
                      A[i][j] = A[i][k] + A[k][j];
```

}

Chúng ta dễ dàng đánh giá được thời gian chạy của thuật toán Floyd. Số lần lặp của lệnh gán (2) trong lệnh lặp (1) là $O(|V|^2)$. Số lần lặp của lệnh (4) trong lệnh lặp (3) là n^3, và do đó thời gian chạy của lệnh lặp (3) là $O(|V|^3)$. Vì vậy thuật toán Floyd đòi hỏi thời gian $O(|V|^3)$

Ví dụ. Áp dụng thuật toán Floyd cho đồ thị sau



Sau khi thực hiện lệnh lặp (1), mảng A lưu độ dài các cung (i,j)

A=

	0	1	2	3
0	0	5	∞	∞
1	50	0	15	5
2	30	∞	0	15
3	15	∞	5	0

Sau đó lần lượt với k = 0, 1, 2, 3 ta cập nhật lại các giá trị trong mảng A theo lệnh (4) chẳng hạn với k = 0, A[2][0] + A[0][1] = 30 + 5 = 35 < A[2][1] = ∞, vì vậy giá trị mới của A[2][1] là 35.

Với k=0

A=

0	5	∞	∞
50	0	15	5
30	35	0	15
15	20	5	0

Với k=0

A=

0	5	∞	∞
50	0	15	5
30	35	0	15
15	20	5	0

Với k=1

A=

0	5	20	10
50	0	15	5
30	35	0	15
15	20	5	0

Với k=2

A=

0	5	20	10
45	0	15	5
30	35	0	15
15	20	5	0

Với k=3

A=

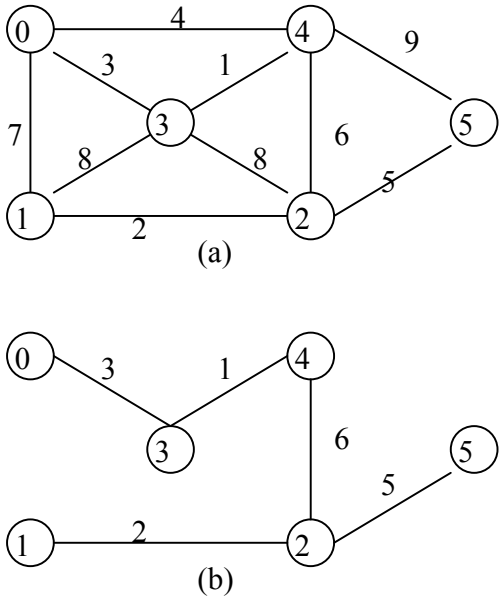
0	5	20	10
20	0	15	5
30	35	0	15
15	20	5	0

Muốn lưu lại vết của đường đi ngắn nhất giữa mọi cặp đỉnh, trong thuật toán Floyd ta sử dụng thêm mảng P, trong đó P[i][j] lưu đỉnh k nếu đường đi ngắn nhất từ i tới j tìm được bởi thuật toán Floyd đi qua đỉnh k. Khi khởi tạo trong lệnh lặp (1), ta đặt P[i][j] = -1 vì ban đầu A[i][j] = k. Bằng cách đó ta tìm được, chẳng hạn đường đi ngắn nhất từ đỉnh 1 tới đỉnh 0 là đường đi 1→3→0.

18.6 CÂY BAO TRÙM NGẮN NHẤT

Trong mục này chúng ta sẽ nghiên cứu các thuật toán tìm cây bao trùm ngắn nhất. Giả sử G = (V,E) là đồ thị vô hướng liên thông. Một tập T các cạnh của đồ thị G sao cho chúng không tạo thành chu tình và nối tất cả các đỉnh của đồ thị được gọi là cây bao trùm của đồ thị. Giả sử G là đồ thị có trọng số, trọng số (độ dài) của cạnh (u,v) $\in$ E là c(u,v) >= 0 và nếu không có cạnh (u,v) thì ta xem c(u,v) = ∞. Trong đồ thị vô hướng liên thông có trọng số, chúng ta quan tâm tới tìm cây bao trùm ngắn nhất , tức là cây bao trùm T mà tổng độ dài các cạnh trong cây T là nhỏ nhất.

Ví dụ. Với đồ thị trong hình 18.8a, cây bao trùm ngắn nhất của đồ thị này được cho trong hình 18.8b. Cây này gồm các cạnh {(0,3), (3,4), (1,2), (2,4), (2,5)} và độ dài của nó là 17.



Hình 18.8. Một cây bao trùm ngắn nhất

Cây bao trùm ngắn nhất có nhiều ứng dụng trong thực tế. Chẳng hạn, trong việc thiết kế mạng truyền thông. Khi đó, có thể mô hình hoá các địa điểm là các đỉnh của đồ thị, các cạnh của đồ thị biểu diễn các đường truyền nối các địa điểm. Giá của các cạnh là giá thi công các đường nối. Việc tìm các đường nối với tổng giá thấp nhất chính là tìm cây bao trùm ngắn nhất.

Cần lưu ý rằng, nếu đồ thị G có n đỉnh, thì cây bao trùm T có đúng n-1 cạnh.

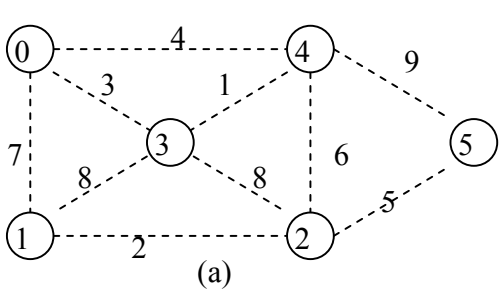
Các thuật toán tìm cây bao trùm ngắn nhất sẽ được trình bày dưới đây đều được thiết kế theo kỹ thuật tham ăn. Ý tưởng chung của các thuật toán này là: ta xây dựng tập T các cạnh dần từng bước xuất phát từ T rỗng. Trong mỗi bước lặp, ta sẽ chọn cạnh (u,v) ngắn nhất trong các cạnh còn lại để đưa vào tập T. Có hai phương pháp chọn trong mỗi bước lặp. Trong phương

pháp thứ nhất (thuật toán Prim), cạnh được chọn ở mỗi bước là cạnh ngắn nhất trong các cạnh còn lại, sao cho nó cùng với các cạnh đã chọn tạo thành đồ thị liên thông, không có chu trình (tức là tạo thành một cây). Còn trong thuật toán Kruskal, cạnh được chọn ở mỗi bước là cạnh ngắn nhất không tạo thành chu trình với các cạnh đã chọn.

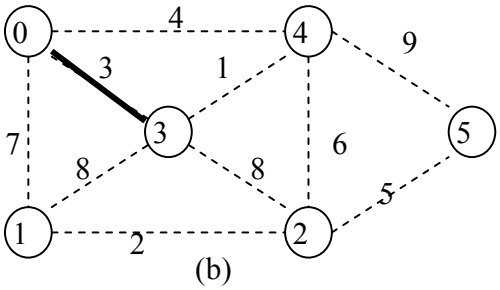
18.6.1 Thuật toán Prim

Giả sử U là tập các đỉnh kề các cạnh trong tập cạnh T. Ban đầu tập U chứa một đỉnh tuỳ chọn của đồ thị G, còn T rỗng. Tại mỗi bước lặp, ta sẽ chọn cạnh (u,v) ngắn nhất mà u ∈ U và v ∈ V - U, rồi thêm đỉnh v vào tập đỉnh U và thêm cạnh (u,v) vào tập cạnh T. Điều đó đảm bảo rằng, sau mỗi bước T luôn luôn là một cây. Tiếp tục phát triển cây T cho tới khi U = V, ta nhận được T là cây bao trùm. Thuật toán Prim là như sau

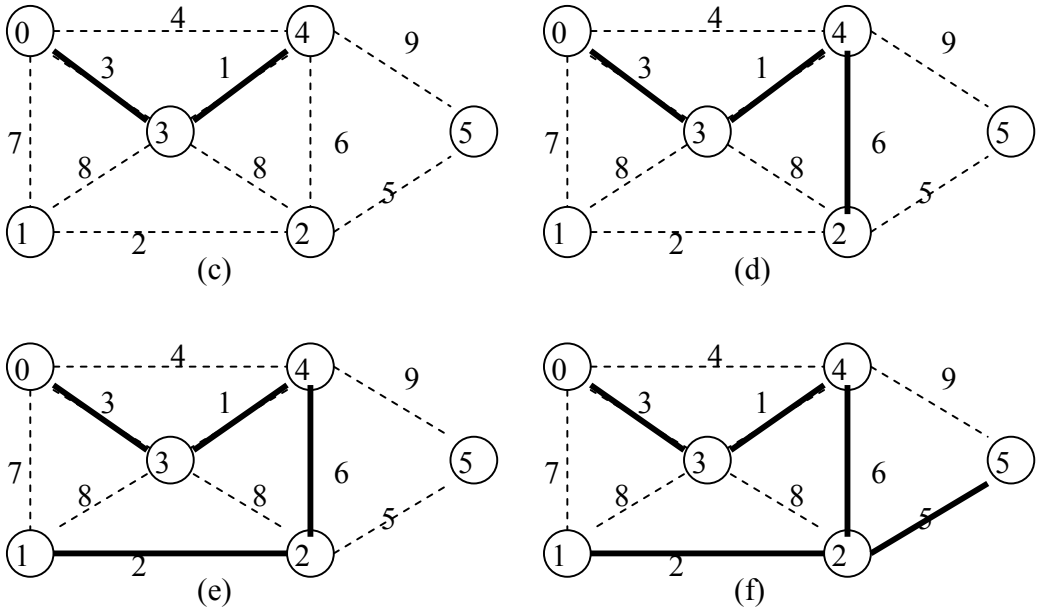
```
                Prim(G,T)
//Xây dựng cây bao trùm ngắn nhất T của đồ thị G
{
    U = {s};    //Khởi tạo tập U chỉ chưa một đỉnh s
    T = ∅;      // Khởi tạo tập cạnh T rỗng.
    while ( U ≠ V )
        {
            chọn (u,v) là cạnh ngắn nhất với u ∈ U và v ∈ V - U;
            U = U ∪ {v};
             T = T ∪ {(u,v};
         }
}
```



(a)



(b)



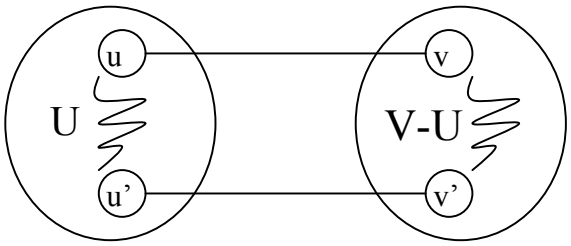
Hình 18.9. Phát triển cây T theo thuật toán Prim.

Để thấy thuật toán Prim làm việc như thế nào, ta hãy xét đồ thị trong hình 18.8a. Giả sử ban đầu U chứa đỉnh 0 và T rỗng như trong hình 18.9a. Trong số các cạnh (0,1), (0,3) và (0,4) thì cạnh (0,3) là ngắn nhất, ta chọn (0,3) đưa vào tập T. Lúc này T = {(0,3)}, U = {0,3}, như trong hình 18.9b. Trong số các cạnh (0,1), (0,4), (3,1), (3,2) và (3,4), cạnh ngắn nhất (3,4) được chọn. Đến đây, T = {(0,3), (3,4)} và U = {0, 3, 4} như trong hình 18.9c. Ở các bước tiếp theo , các cạnh (4,2), (2,1), (2,5) sẽ lần lượt được chọn để thêm vào T và ta nhận được T tương ứng trong các hình 18.9d-18.9f. Kết quả ta nhận được cây bao trùm T trong hình 18.9f.

Tính đúng đắn của thuật toán Prim.

Chúng ta sẽ chứng minh rằng, cây bao trùm T được xây dựng bởi thuật toán Prim là cây bao trùm ngắn nhất. Giả sử T' là một cây bao trùm ngắn nhất và T' không chứa cạnh (u,v) của cây T. Chúng ta sẽ biến đổi cây T' thành cây T_1 chứa cạnh (u,v) sao cho T_1 cũng là cây bao trùm ngắn nhất.

Vì (u,v) là một cạnh trong cây T nên (u,v) là cạnh được chọn để đưa vào T theo lệnh (1) trong thuật toán Prim, tức là (u,v) là cạnh ngắn nhất với u ∈ U và v ∈ V - U. Nhưng (u,v) không thuộc cây T', nên nếu ta thêm cạnh (u,v) vào cây T' thì cạnh (u,v) cùng với đường đi từ đỉnh u tới đỉnh v trong cây T' sẽ tạo thành một chu trình. Trên đường đi từ u tới v trong cây T' ta gọi u' là đỉnh sau cùng ở trong U và v' là đỉnh đầu tiên ở trong V - U, như trong hình sau:



Trong cây T' nếu ta thay cạnh (u',v') bởi cạnh (u,v) thì ta sẽ nhận được cây T_1. Nhưng c(u,v) <= c(u',v'), nên cây T_1 cũng là cây bao trùm ngắn nhất và T_1 chứa (u,v). Nếu cây T_1 còn không chứa cạnh (u_1,v_1) của cây T thì ta lại biến đổi cây T_1 thành cây T_2 chứa (u_1,v_1) và T_2 cũng là cây bao trùm ngắn nhất. Tiếp tục, cùng lắm là sau m phép biến đổi (m <= n - 1, n là số đỉnh của đồ thị), ta sẽ nhận được cây T_m chứa tất cả các cạnh của cây T, tức là T_m = T:

$T' = T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_m = T$

Tất cả các cây T_i (i=1,2,…,m) đều là cây bao trùm ngắn nhất, và do đó $T=T_m$ cũng là cây bao trùm ngắn nhất.

Bây giờ chúng ta xét xem cần phải cài đặt tập V - U như thế nào để có thể thực hiện hiệu quả hành động (1) trong thuật toán Prim. Dễ dàng thấy rằng, có thể cài đặt tập V - U bởi hàng ưu tiên với khoá của đỉnh v, key[v], là độ dài của cạnh (u,v) ngắn nhất với u ∈ U.

Sử dụng mảng near[v] để ghi lại đỉnh u ∈ U, u là đỉnh gần v nhất. Ban đầu vì U chỉ chứa một đỉnh s, nên với mọi v ≠ s ta khởi tạo key[v] = c(s,v), và near[v] = s. Sau đó trong mỗi bước lặp của thuật toán Prim, ta loại đỉnh v có khoá nhỏ nhất khỏi hàng ưu tiên và thêm cạnh (near[v],v) vào tập T. Sau khi đỉnh v được bổ xung vào U thì với các đỉnh còn lại w trong hàng ưu tiên, khoá của nó có thể giảm, cần phải cập nhật. Chúng ta có thể mô tả thuật toán Prim một cách cụ thể hơn như sau.

```
                Prim(G,T)
{
    T = ∅;   //Khởi tạo tập cạnh T rỗng
    for (mỗi đỉnh v ∈ V)
        {
            key[v] = c(s,v);      //s là đỉnh bất kỳ
            near[v] = s;
        }
    Khởi tạo hàng ưu tiên P chứa tất cả các đỉnh v ≠ s
                                  với khoá của v là key[v];
    while (P không rỗng)
      {
          v = DeleteMin(P);
          T = T ∪ {(near[v],v)};
          for ( mỗi w kề v và w ∈ P )
              if ( c(v,w) < key[w] )
                {
                    DecreaseKey (P,w,c(v,w));
                    near[w] = v;
                }
        }
}
```

Chúng ta có nhận xét rằng, dòng điều khiển của thuật toán Prim trên là giống hệt thuật toán Dijkstra, do đó ta có thể kết luận rằng, nếu sử dụng hàng ưu tiên trên được cài đặt bởi cây thứ tự bộ phận, thì thời gian chạy của thuật toán Prim cũng là O(|V|log|V| + |E|log|V|). Nhưng G = (V,E) là đồ thị

vô hướng liên thông, nên |E| >= |V| - 1. Do đó, thời gian chạy của thuật toán Prim là O(|E|log|V|).

18.6.2 Thuật toán Kruskal

Thuật toán Kruskal cũng được thiết kế theo kỹ thuật tham ăn. Tập T các cạnh được xây dựng dần từng bước xuất phát từ T rỗng. Nhưng khác với thuật toán Prim, tại mỗi bước trong thuật toán Kruskal, cạnh (u,v) được chọn thêm vào T là cạnh ngắn nhất trong các cạnh còn lại và không tạo thành chu trình với các cạnh đã có trong T. Vấn đề đặt ra là, tại mỗi bước khi xét cạnh (u,v) ngắn nhất trong các cạnh còn lại, làm thế nào để biết được (u,v) có tạo thành chu trình với các cạnh đã có trong T hay không?

Chú ý rằng, một tập T các cạnh không tạo thành chu trình sẽ phân hoạch tập đỉnh V của đồ thị thành một họ các tập con không cắt nhau, mỗi tập con gồm các đỉnh được nối với nhau bởi các cạnh trong T và hai đỉnh nằm trong hai tập con khác nhau thì không được nối với nhau. Khi ta xét cạnh (u,v) ngắn nhất trong các cạnh còn lại, nếu cả hai đỉnh u và v cùng nằm trong một tập con, thì vì tất cả các đỉnh nằm trong tập con này đã được nối với nhau, nên một chu trình sẽ được tạo ra khi ta thêm vào cạnh (u,v); còn nếu u, v nằm trong hai tập con khác nhau, thì khi thêm (u,v) và T sẽ không có chu trình nào được tạo ra. Thuật toán Kruskal sử dụng CTDL họ các tập con không cắt nhau (xem chương 13) để bảo trì họ các tập con các đỉnh được phân hoạch bởi tập T. Khi xét cạnh (u,v), ta áp dụng phép Find(u), Find(v) để tìm tập con chứa u và tập con chứa v. Nếu Find(u) ≠ Find(v) thì ta thêm (u,v) vào T, rồi áp dụng phép toán Union(u,v) để hợp nhất tập con chứa u và tập con chứa v thành một. Thuật toán Kruskal được mô tả như sau

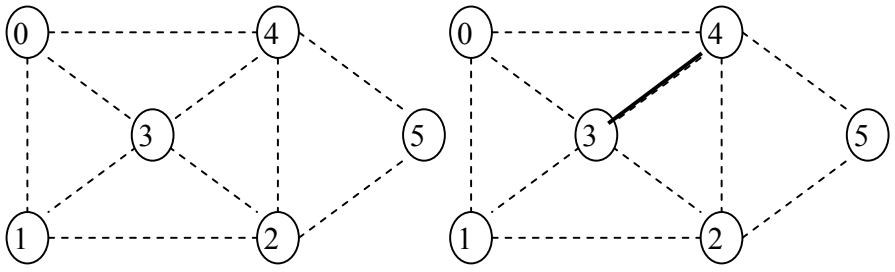
```
Kruskal(G,T)
//Xây dựng cây bao trùm ngắn nhất T của đồ thị G = (V,E).
{
```

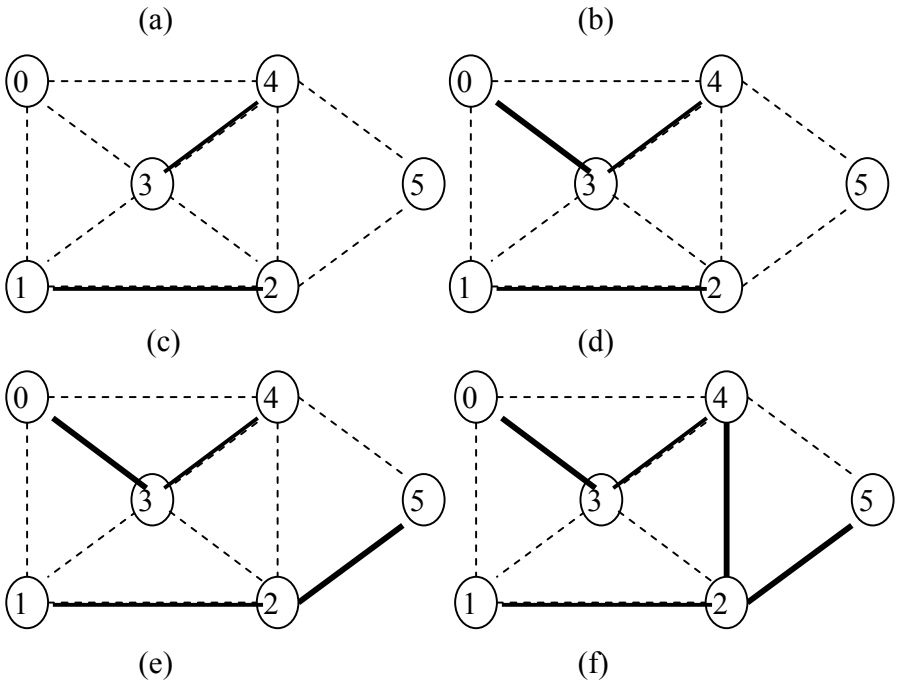
```
    T = ∅;
    Sắp xếp các cạnh (u,v)∈E thành một danh sách L không giảm
                                                    theo độ dài;
    Khởi tạo một họ tập con, mỗi tập con chứa một đỉnh v∈V;
    k = 0;   //Biến đếm số cạnh được đưa vào T
    do {
        Loại cạnh (u,v) ở đầu danh sách L (tức là cạnh ngắn nhất
                                            trong các cạnh còn lại);
        i = Find(u);   // i là đại biểu của tập con chứa u
        j = Find(v);
        if ( i ≠ j )
           {
              T = T ∪ {(u,v)};
              Union(u,v);   //Lấy hợp của các tập con chứa u và v.
               k++;
           }
       }
     while ( k < |V| - 1);
}
```

Ví dụ. Lại xét đồ thị hình 18.8a. Các cạnh của đồ thị được sắp xếp thành danh sách tăng dần theo độ dài như sau :

L = ((3,4), (1,2), (0,3), (0,4), (2,5), (2,4), (0,1), (1,3), (2,3), (4,5))

Ban đầu T rỗng, mỗi đỉnh của đồ thị ở trong một tập con như trong hình 18.10a. Lần lượt các cạnh (3,4), (1,2), (0,3) được thêm vào T, ta có T như trong các hình 18.10b-18.10d. Trong hình 18.10d, họ các tập con là {0,3,4}, {1,2} và {5}. Cạnh tiếp theo được xét là cạnh (0,4). Cả hai đỉnh 0 và 4 thuộc cùng tập con {0,3,4}. Xét cạnh tiếp theo trong danh sách L, cạnh (2,5). Đỉnh 2 thuộc tập con {1,2}, đỉnh 5 thuộc tập con {5}, vì vậy (2,5) được thêm vào T, ta có T như trong hình 18.10e. Cạnh tiếp theo (2,4) cũng được thêm vào T, lúc này T chứa đủ 5 cạnh, ta dừng lại. Cây bao trùm ngắn nhất là cây T trong hình 18.10f.





Hình 18.10 Phát triển tập T theo thuật toán Kruskal .

Thời gian chạy của thuật toán Kruskal.

Thời gian chạy của thuật toán này phụ thuộc vào cách cài đặt họ các tập con không cắt nhau bởi các cây hướng lên (up-tree) (xem mục 13.3). Thời gian sắp xếp các cạnh (lệnh (1)) là O(|E|log|E|). Chú ý rằng, vì G = (V,E) là đồ thị vô hướng liên thông, nên

$|V|-1 <= |E| <= 1/2(|V|^2 - |V|)$

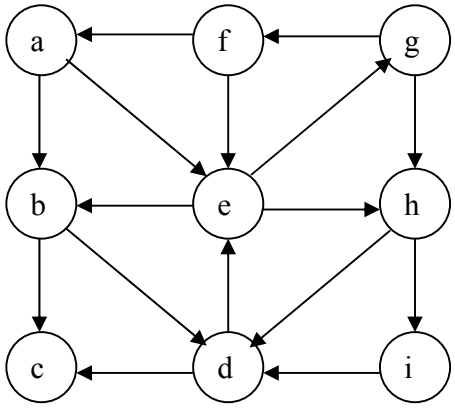
Từ đó ta suy ra, $|E| = O(|V|^2)$ và $\log |E| = O(\log|V|)$. Vì vậy thời gian sắp xếp trong lệnh (1) là O(|E|log|V|). Thời gian thực hiện lệnh (2) là O(|V|). Bây giờ ta đánh giá thời gian của lệnh lặp (3). Số tối đa các lần lặp là O(|E|). Trong mỗi lần lặp ta cần thực hiện các phép tìm (4), (5) và phép hợp (6). Theo định lý 13.2 (mục 13.3), thời gian thực hiện lệnh lặp (3) là O(|E|log*|V|). Do đó thời gian chạy của thuật toán Kruskal là O(|E|log|V| + |E|log*|V|) = O(|E|log|V|).

Tính đúng đắn của thuật toán Kruskal.

Ta cần chứng minh rằng, cây T được xây dựng bởi thuật toán Kruskal là cây bao trùm ngắn nhất. Giả sử T = {e_1, e_2,…,e_{n-1}}, n = |V|, trong đó e_k là cạnh được chọn để thêm vào T ở bước thứ k (k = 1,…, n - 1). Giả sử T' là một cây bao trùm ngắn nhất của đồ thị. Nếu T' chứa tất cả các cạnh trong T, thì T = T'. Giả sử e_i là cạnh đầu tiên trong T mà T' không chứa e_i. Nếu ta bổ xung cạnh e_i vào T' thì một chu trình sẽ được tạo ra (e_i, e'_1,…, e'_k). Trong các cạnh e'_1,…, e'_k thuộc T' phải có một cạnh không thuộc T, giả sử cạnh đó là e'_j. Theo giả thiết về e_i, thì các cạnh e_1,…,e_{i-1} thuộc T', nếu e'_j không tạo thành chu trình e_1,…,e_{i-1}. Do đó, theo cách chọn cạnh e_i ở bước i của thuật toán, $c(e_i) <= c(e'_j)$ (ở đây ta ký hiệu c(.) là độ dài của cạnh). Bây giờ trong cây T' ta thay cạnh e'_j bởi cạnh e_j để nhận được cây T_1. Và vì $c(e_j) <= c(e'_j)$, nên T_1 cũng là cây bao trùm ngắn nhất. Nếu cây T còn có cạnh không thuộc cây T_1, thì bằng cách trên ta lại biến đổi T_1 thành T_2. Cùng lắm là sau m phép biến đổi (m <= n - 1), ta sẽ nhận được cây T_m chứa tất cả các cạnh của cây T, do đó T_m = T. Trong quá trình biến đổi cây, tất cả các cây T_1, T_2,..,T_m đều là cây bao trùm ngắn nhất, do đó T = T_m là cây bao trùm ngắn nhất.

BÀI TẬP.

1. Cho đồ thị định hướng trong hình 18.11. Áp dụng thuật toán đi qua đồ thị theo bề rộng và theo độ sâu cho đồ thị này khi xuất phát từ đỉnh a, đưa ra thứ tự các đỉnh được thăm cho mỗi cách duyệt. Vẽ ra cây tạo thành cho mỗi cách duyệt.



Hình 18.11. Đồ thị cho các bài tập 1. và 5.

2. Cho đồ thị vô hướng. Sử dụng kỹ thuật đi qua đồ thị theo bề rộng, hãy đưa ra thuật toán để trả lời cho câu hỏi: đồ thị có liên thông không, nếu không thì đồ thị có mấy thành phần liên thôn và mỗi thành phần gồm các đỉnh nào?

3. Giả sử v và w là hai đỉnh bất kỳ của đồ thị không có trọng số. Sử dụng kỹ thuật đi qua đồ thị theo độ sâu, hãy đưa ra thuật toán để cho biết đỉnh w có đạt tới từ đỉnh v không, và nếu có thì cho biết đường đi ngắn nhất từ v tới w.

4. Hãy viết hàm tìm kiếm theo độ sâu DFS( ) không đệ quy.

5. Hãy đi qua đồ thị thình 18.11 theo độ sâu xuất phát từ đỉnh f. Vẽ ra cây được tạo thành và cho biết cung nào là cung cây, cung tiến, cung ngược, cung xiên?

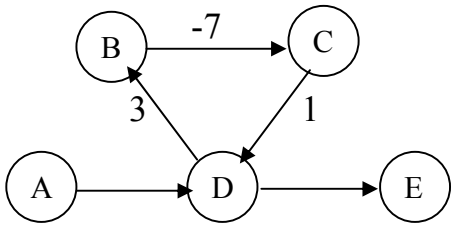
6. Hãy đưa vào hàm DFS( ) các lệnh để gắn nhãn cho các cung (mỗi cung có thể là cung cây, cung tiến, cung ngược hoặc cung xiên).

7. Cho đồ thị định hướng. Sử dụng kỹ thuật đi qua đồ thị theo độ sâu, hãy viết thuật toán để cho biết đồ thị có chu trình không, nếu có thì cần cho biết đó là các chu trình nào?

8. Thiết kế một thuật toán sắp xếp topo không cần phải duyệt cây theo độ sâu. Đánh giá thời gian chạy của thuật toán đưa ra.

9. Cho đồ thị và một đỉnh đích v trong đồ thị. Hãy đưa ra thuật toán tìm đường đi ngắn nhấy từ tất cả các đỉnh khác tới đỉnh đích v.

10. (Đồ thị có trọng số âm). Thuật toán tìm đừong đi ngắn nhất Dijkstra chỉ áp dụng cho đồ thị có trọng số không âm. Xét đồ thị có cung với trọng số âm sau.



5 8

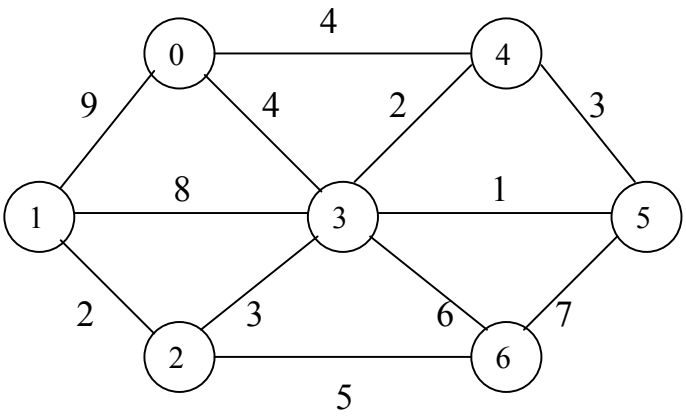
Đồ thị này chứa chu trình (D, B, C, D) có trọng số -7 + 1+ 3 = -3. Đồ thị không thể có đường đi ngắn nhất từ A đến E, vì mỗi lần qua chu trình độ dài đường đi sẽ giảm đi –3.

Hãy đưa ra một đồ thị có chứa cung với trọng số âm, nhưng không chứa chu trình âm mà thuật toán Dijkstra cho ra kết quả sai.

11. Giả sử ta cần tìm đường đi dài nhất từ một đỉnh nguồn tới các đỉnh còn lại của đồ thị. Có thể sử dụng thuật toán Dijkstra với các thay đổi sau: thay cho chọn đỉnh $u \in V - S$ với D[u] ngắn nhất ở mỗi bước, ta chọn đỉnh $u \in V - S$ với D[u] dài nhất, rồi thêm u vào tập S, sau đó cập nhật các giá trị D[v] với $v \in V - S$ bằng cách lấy D[v] = D[u] + C(u, v) nếu D[u] + C(u, v) > D[v] ? Nếu có thể, hãy chứng minh; nếu không, hãy đưa ra một đồ thị mà thuật toán cho ra đường đi không phải là dài nhất.

12. Cho đưa ra một đồ thị vô hướng có trọng số, sau đó hãy xây dựng cây bao trùm ngắn nhất của đồ thị đó bằng cách sử dụng:
    a. Thuật toán Prim.
    b. Thuật toán Kruskal.

    Cần đưa ra kết quả của từng bước trong quá trình phát triển cây.



Hình 18.12. Đồ thị cho bài tập 12

13. Giả sử độ dài của các cung của đồ thị được lưu trong mảng C[i] [j]. Hãy viết chương trình cài đặt thuật toán Dijkstra bằng cách cài đặt tập V – S như sau: sử dụng mảng A[0 … n-1], trong đó nếu u ∈ S thì A[u] = -D[u], còn nếu v ∈ V – S thì A[v] = D[v]. (Giả thiết độ dài các cung là số dương)
14. Viết chương trình cài đặt thuật toán Prim bằng cách cài đặt tập V – U như sau: sử dụng mảng S[0 … n-1], trong đóA[u] = -1 nếu u ∈ U, còn nếu v ∈ V – U thì A[v] = u, trong đó u ∈ U và độ dài cạnh (u, v) là ngắn nhất.
15. Viết chương trình cài đặt thuật toán Kruskal bằng cách cài đặt họ tập con các đỉnh bởi mảng như sau. Chẳng hạn, nếu V = {0, 1, 2, 3, 4, 5} và họ tập con các đỉnh là {0, 2, 5} , {1, 4}, {3} thì họ này được biểu diễn bởi mảng sau:

0	1	0	3	1	0
0	1	2	3	4	5

CHƯƠNG 19

CÁC BÀI TOÁN NP – KHÓ VÀ NP - ĐẦY ĐỦ

Chúng ta đã biết nhiều bài toán, điển hình là bài toán người bán hàng, bài toán balô, bài toán chu trình Hamilton, … Các thuật toán tốt nhất để giải quyết các bài toán này đều có thời gian chạy không phải là thời gian đa thức, chẳng hạn thuật toán quy hoạch động cho bài toán người bán hàng có thời gian chạy $O(n^2 2^n)$, … Cho tới nay chưa có ai tìm ra được thuật toán thời gian đa thức cho bất kỳ bài toán nào trong lớp các bài toán trên. Vấn đề đặt ra là có tồn tại hay không thuật toán thời gian chạy đa thức cho các bài toán trên?

Các vấn đề lý thuyết được trình bày trong chương này không khẳng định rằng, không tồn tại thuật toán thời gian đa thức cho các bài toán trên. Điều mà các nhà nghiên cứu đã làm được là chỉ ra rằng, nhiều bài toán chưa có thuật toán giải trong thời gian đa thức là có liên quan với nhau về mặt tính toán. Cụ thể là, chúng ta sẽ thiết lập hai lớp: lớp các bài toán NP – khó (NP-hard) và lớp các bài toán NP- đầy đủ (NP – complete). Một bài toán là NP-đầy đủ có tính chất rằng, nó có thể giải được trong thời gian đa thức nếu và chỉ nếu tất cả các bài toán NP-đầy đủ khác giải được trong thời gian đa thức. Nếu một bài toán NP- khó giải được trong thời gian đa thức thì tất cả các bài toàn NP-đầy đủ giải được trong thời gian đa thức. Người ta đã chỉ ra rằng, lớp NP-đầy đủ là lớp con của lớp NP – khó, nhưng có bài toán là NP – khó nhưng không phải là NP-đầy đủ.

Các lớp bài toán NP–khó và NP - đầy đủ là rất giàu có. Tất cả các bài toán nổi tiếng mà ta đã quen biết (bài toán người bán hàng, bài toán chiếc balô, và rất nhiều bài toán tổ hợp, các bài toán của lý thuyết đồ thị …) đều là NP – khó.

Các lớp NP – khó và NP - đầy đủ liên quan tới sự tính toán không đơn định (nondeterministic computations). Không thể thực hiện được sự tính

toán không ổn định trong thực tế, bởi vậy về mặt trực quan, chúng ta có thể phỏng đoán (chưa chứng minh được) rằng, các bài toán NP - đầy đủ hoặc NP – khó không giải được trong thời gian đa thức.

Chúng ta sẽ cố gắng trình bày các vấn đề đã nêu trên một cách đơn giản, không hình thức, không đi sâu vào các chứng minh phức tạp.

19.1 THUẬT TOÁN KHÔNG ĐƠN ĐỊNH

Khái niệm thuật toán mà chúng ta sử dụng cho đến nay có tính chất rằng, kết quả thực hiện mỗi phép toán là được xác định duy nhất. Thuật toán với tính chất này được gọi là **thuật toán đơn định** (deterministic algorithm).

Định nghĩa 1. Lớp **P** bao gồm tất cả các bài toán giải được bởi thuật toán đơn định trong thời gian đa thức (tức là tồn tại thuật toán giải quyết nó với thời gian chạy đa thức).

Để xác định lớp NP các bài toán, chúng ta cần phải đưa vào khái niệm **thuật toán không đơn định** (nondeterministic algorithm). Sự mở rộng khái niệm thuật toán đơn định thành khái niệm thuật toán không đơn định cũng hoàn toàn tương tự như khi ta mở rộng khái niêm otomat hữu hạn đơn định thành otomat hữu hạn không đơn định. Trong các thuật toán không đơn định chúng ta được phép đưa vào các phép toán mà kết quả của nó không phải là một giá trị được xác định duy nhất mà là một tập hữu hạn các giá trị. Các phép toán đó được biểu diễn bởi hàm lựa chọn:

choice(S)

Đây là hàm đa trị, giá trị của nó là các phần tử của tập hữu hạn S. Ngoài hàm **choice**, để mô tả các thuật toán không đơn định chúng ta đưa vào hai lệnh:

success
failure

Các lệnh này là các lệnh dừng, hiệu quả của chúng sẽ được mô tả dưới đây.

Các thuật toán không đơn định được thực hiện như thế nào? Để thực hiện thuật toán không đơn định, chúng ta cần thực hiện các tính toán theo dòng điều khiển được quy định bởi các câu lệnh như khi ta thực hiện thuật toán đơn định, chỉ có một điều khác biệt so với thuật toán đơn định là, khi gặp hàm lựa chọn choice(S) thì sự tính toán được phân thành nhiều nhánh, mỗi nhánh tương ứng với một giá trị được chọn ra từ tập S, tất cả các nhánh này sẽ làm việc đồng thời và độc lập với nhau. Mỗi nhánh tính toán đó khi gặp một hàm lựa chọn khác choice(S') lại được phân thành nhiều nhánh tính toán khác. Và như vậy, khi thực hiện thuật toán không đơn định, chúng ta cần phải thực hiện đồng thời và độc lập nhiều đường tính toán (được tạo thành từ các nhánh tương ứng với các lựa chọn). Do đó, ta có thể quan niệm rằng, thuật toán không đơn định mô tả sự tính toán song song không hạn chế. Khi mà một đường tính toán gặp lệnh success, thì có nghĩa là đường tính toán đó đã được tạo thành từ các lựa chọn đúng đắn và đã thành công cho ra nghiệm của bài toán, khi đó tất cả các đường tính toán đều dừng làm việc. Còn nếu một đường tính toán gặp lệnh failure, thì có nghĩa là đường tính toán này đã thất bại, không cho ra nghiệm của bài toán, và chỉ riêng đường này dừng làm việc. Máy có khả năng thực hiện thuật toán không đơn định sẽ được gọi là **máy không đơn định** (nondeterministic machine). Không tồn tại các máy không đơn định trong thực tế. Sau đây chúng ta sẽ đưa ra một vài ví dụ về thuật toán không đơn định.

Ví dụ 1. Xét bài toán tìm xem phần tử x có trong mảng A[0 … n - 1], n ≥ 1, hay không ? Nếu có ta cần in ra chỉ số i mà A[i] = x, nếu không thì in ra n. Thuật toán không đơn định là như sau:

```
        Search (x, A, n)
        // Tìm x trong mảng A[0… n - 1].
{
   i  =  choice( 0: n – 1);
   if (A[i] = = x)
```

```
        {
          print(i);
          success;
        }
      else {
             print(n);
             failure ;
           }
}
```

Chú ý rằng, lệnh gán i = choice(0 : n – 1) cho kết quả là biến i được gán một trong các giá trị trong đoạn [0 … n - 1].

Ví dụ 2. Chúng ta đưa ra thuật toán không đơn định để sắp xếp mảng A[0 … n – 1] theo thứ tự không giảm. Trong thuật toán ta sử dụng mảng phụ B[0 … n - 1].

```
        Sort (A, n)
        // Sắp xếp mảng A[0 … n - 1], n ≥ 1.
{
(1)   for ( i = 0 ; i < n ; i + + )
            B[i] = ∞ ;
(2)   for ( i = 0 ; i < n ; i + +)
          {
(3)         k = choice( 0 : n – 1);
(4)         if (B[k] ≠ ∞ )
                  failure ;
            else
                  B[k] = A[i] ;
          }
(5)   for ( i = 0 ; i < n-1 ; i + +)
```

```
            if (B[i] > B[i + 1])
                failure;
(6)     print (B);
        // in ra mảng B đã được sắp theo thứ tự không giảm.
(7) success;
}
```

Lệnh lặp (1) khởi tạo mảng B chứa một giá trị khác với tất cả các giá trị trong mảng A. Lệnh lặp (2) thực hiện đặt mỗi giá trị A[i] trong mảng A vào mảng B tại chỉ số k, với k được lựa chọn không đơn định bởi lệnh (3). Lệnh (4) kiểm tra xem B[k] đã chứa một giá trị trong mảng A hay chưa. Sau khi thực hiện lệnh lặp (2), nhánh tính toán không gặp lệnh failure sẽ cho kết quả mảng B chứa các giá trị là hoán vị của các giá trị trong mảng A. Lệnh (5) kiểm tra các giá trị trong mảng B có thoả mãn thứ tự không giảm hay không.

Bây giờ chúng ta xác định thời gian thực hiện thuật toán không đơn định. Thời gian này được xác định là thời gian thực hiện đường tính toán ngắn nhất dẫn tới sự kết thúc thành công (lệnh success). Khi đánh giá thời gian chạy của thuật toán không đơn định, chúng ta có thể sử dụng các kỹ thuật đánh giá thời gian chạy của thuật toán đơn định đã trình bày trong chương 15, chỉ cần lưu ý rằng, thời gian thực hiện hàm choice(S) và các lệnh success và failure là O(1). Chẳng hạn, dễ dàng thấy rằng, thuật toán không đơn định tìm phần tử x trong mảng A[0 … n - 1] (ví dụ 1) có thời gian chạy là O(1), trong khi đó thuật toán đơn định (tìm kiếm tuần tự) cần thời gian O(n).

Bây giờ ta đánh giá thời gian chạy của thuật toán không đơn định sắp xếp mảng A[0 … n - 1] (ví dụ 2). Trong thuật toán này, mỗi lệnh lặp (1), (2), (5) và lệnh (6) cần thời gian O(n), và do đó thời gian chạy của thuật toán không đơn định này là O(n). Trong chương 17 chúng ta đã thấy rằng, tất cả các thuật toán đơn định tốt nhất để sắp xếp mảng đều đòi hỏi thời gian O(nlogn).

Khả năng tính toán của thuật toán không đơn định là rất mạnh. Nhiều bài toán cho tới nay người ta mới chỉ tìm ra thuật toán đơn định với thời gian mũ để giải quyết, nhưng chúng ta có thể dễ dàng đưa ra thuật toán không đơn định với thời gian đa thức cho các bài toán đó.

Định nghĩa 2. Lớp **NP** bao gồm tất cả các bài toán có thể giải được bởi thuật toán không đơn định trong thời gian đa thức.

Bởi vì thuật toán đơn định là trường hợp đặc biệt của thuật toán không đơn định, do đó lớp P là lớp con của lớp NP. Nhưng cho tới nay người ta vẫn chưa biết P = NP hay P ≠ NP? Tức là, có phải tất cả các bài toán giải được bởi thuật toán không đơn định trong thời gian đa thức đều có thể giải được bởi thuật toán đơn định trong thời gian đa thức, hay là có bài toán giải được bởi thuật toán không đơn định trong thời gian đa thức, nhưng không tồn tại thuật toán đơn định với thời gian đa thức để giải quyết nó? Đây là vấn đề chưa giải quyết được, nổi tiếng nhất của khoa học máy tính.

19.2 CÁC BÀI TOÁN NP–KHÓ VÀ NP- ĐẦY ĐỦ

Trong quá trình nghiên cứu để tìm câu trả lời cho vấn đề đã nêu trên, S. Cook đã phát hiện ra một bài toán thuộc lớp NP mà nếu ta tìm được thuật toán đơn định với thời gian đa thức để giải quyết nó thì P = NP.

Bài toán thoả được trong logic mệnh đề

Giả sử x_1, x_2, … là các biến boolean (giá trị của chúng chỉ có thể là true hoặc false). Ta ký hiệu $\neg x_i$ là phủ định của x_i. Litaral là một biến hoặc phủ định của một biến. Một câu tuyển (clause) là tuyển của các literal, chẳng hạn $x_1 \vee \neg x_2 \vee x_3$ là một câu tuyển. Một công thức dạng chuẩn hội là hội của các câu tuyển. Chẳng hạn, các công thức sau là các công thức dạng chuẩn hội:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \qquad (1)$$

$$(x_1 \vee x_2) \wedge (\neg x_1) \wedge (\neg x_2) \qquad (2)$$

Chúng ta đã biết rằng, mọi công thức trong logic mệnh đề đều có thể biến đổi đưa về công thức dạng chuẩn hội. Một công thức được gọi là thoả được (satisfiable) nếu có một cách gán giá trị chân lý cho các biến sao cho công thức nhận giá trị true. Nếu với mọi cách gán giá trị chân lý cho các biến, công thức đều nhận giá trị false thì nó được xem là không thoả được. Ví dụ, công thức (1) là thoả được, bởi vì với x_1 = true, x_2 = false, x_3 = false công thức sẽ đúng; công thức (2) là không thoả được.

Bài toán thoả được (satisfiability problem) là như sau: xác định công thức dạng chuẩn hội có thoả được hay không?

Giả sử E = E(x_1 , … , x_n) là công thức dạng chuẩn hội của các biến x_1, …, x_n. Thuật toán không đơn định để kiểm tra E có thoả được hay không là rất đơn giản: ta chỉ cần lựa chọn (không đơn định) một trong 2^n khả năng gán giá trị chân lý cho n biến và tính giá trị của công thức E tương ứng với mỗi cách gán. Thuật toán không đơn định là như sau:

```
        Eval (E, n)
// Xác định xem công thức E chứa n biến có thoả được không.
{
  for ( i = 1 ; i <= n ; i + +)
     xi = choice (true, false);
  if ( giá trị của E là true )
          success ; // thoả được
  else
          failure ;
}
```

Chúng ta đánh giá thời gian thực hiện thuật toán không đơn định Eval. Thời gian để lựa chọn giá trị chân lý cho các biến x_i (i = 1, …, n) (lệnh lặp for) là O(n). Thời gian để tính giá trị của E tương ứng với các giá trị đã lựa chọn của các biến là O(kn), trong đó k là số phép hội trong công thức E. Do

đó thời gian chạy của Eval là O(kn). Như vậy, bài toán thoả được thuộc lớp NP.

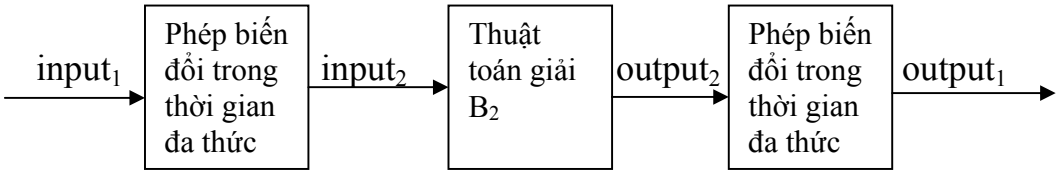
Định lý (S. Cook). Bài toán thoả được thuộc lớp P nếu và chỉ nếu P = NP.

Chứng minh định lý này rất phức tạp, chúng ta không đưa ra. Bây giờ chúng ta xác định hai lớp bài toán: lớp NP – khó và lớp NP - đầy đủ.

Với ý tưởng sử dụng thuật toán giải một bài toán để nhận được thuật toán giải các bài toán khác, chúng ta có định nghĩa sau:

Định nghĩa 3. Ta nói bài toán B_1 quy về bài toán B_2 (ký hiệu $B_1 << B_2$) nếu ta có thể biến đổi trong thời gian đa thức đầu vào của bài toán B_1 ($input_1$) thành đầu vào của bài toán B_2 ($input_2$) sao cho ta có thể nhận được nghiệm của bài toán B_1 ($output_1$) từ nghiệm của bài toán B_2 ($output_2$) bởi phép biến đổi trong thời gian đa thức.

Từ định nghĩa trên, ta suy ra lược đồ chứng minh bài toán B_1 quy về bài toán B_2 là như sau:



Dễ dàng thấy rằng, quan hệ "quy về" có tính bắc cầu, tức là nếu $B_1 << B_2$ và $B_2 << B_3$ thì $B_1 << B_3$. Cũng rõ ràng rằng, nếu B_1 quy về B_2 và B_2 là giải được trong thời gian đa thức thì B_1 cũng giải được trong thời gian đa thức.

Định nghĩa 4. Một bài toán là NP – khó nếu bài toán thoả được quy về bài toán đó. Một bài toán là NP - đầy đủ nếu nó là NP- khó và nó thuộc lớp NP.

Theo định nghĩa trên, bài toán thoả được là NP - đầy đủ. Làm thế nào để chứng minh một bài toán là NP- khó ? Do quan hệ << có tính bắc cầu, để

chứng minh bài toán B là NP- khó, ta chỉ cần chứng minh một bài toán A nào đó mà ta đã biết là NP – khó qui về bài toán B, A << B.

19.3 MỘT SỐ BÀI TOÁN NP – KHÓ

Mục này trình bày một số bài toán NP – khó trong lý thuyết đồ thị. Nhắc lại rằng, để chứng minh một bài toán B là NP – khó, chúng ta có thể chứng minh bài toán thoả được quy về bài toán B, hoặc chứng minh rằng, một bài toán A nào đó mà ta đã biết là NP – khó quy về bài toán B.

Bài toán đồ thị con đầy đủ

Giả sử G = (V, E) là đồ thị vô hướng, đồ thị G' = (V', E') được gọi là đồ thị con đầy đủ của đồ thị G nếu V' $\subseteq$ V và E' $\subseteq$ E, và với mọi cặp đỉnh u, v $\in$V' tồn tại cạnh (u, v) $\in$ E'.

Định lý 1. Bài toán "xác định một đồ thị vô hướng G = (V, E) có đồ thị con đầy đủ k đỉnh hay không" là bài toán NP - đầy đủ.

Chứng minh. Trước hết ta chứng minh bài toán thuộc lớp NP. Thuật toán không đơn định với thời gian đa thức để xác định một đồ thị vô hướng n đỉnh G có chứa đồ thị con đầy đủ k đỉnh ($k \leq n$) là như sau. Đầu tiên ta chọn (không đơn định) ra k đỉnh từ n đỉnh, sau đó kiểm tra xem k đỉnh đã chọn ra có tạo thành đồ thị con đầy đủ không. Việc kiểm tra này chỉ đòi hỏi thời gian $O(k^2)$. Điều đó chứng tỏ rằng, bài toán thuộc lớp NP.

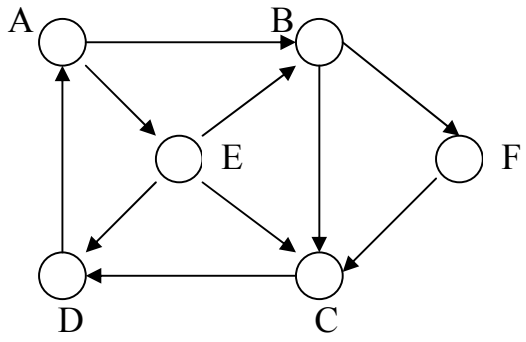
Để chứng minh bài toán là NP – khó, chúng ta sẽ chứng minh bài toán thoả được quy về nó.

Giả sử $F = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ là công thức logic mệnh đề dưới dạng chuẩn hội, F là hội của k câu tuyển C_i(i = 1, …, n). Từ công thức này ta xây dựng đồ thị vô hướng G = (V, E), với tập đỉnh V và tập cạnh E được xác định như sau:

V = { (α , i) , trong đó α là một literal trong câu tuyển C_i }

E = { ((α , i) , (β , j)) , trong đó i $\neq$ j và $\alpha \neq \neg \beta$ }

Chúng ta sẽ chứng minh rằng, công thức F là thoả được nếu và chỉ nếu đồ thị G có đồ thị con đầy đủ k đỉnh. Thật vậy, nếu công thức F thoả được thì tồn tại cách gán giá trị chân lý sao cho mỗi câu tuyển C_i chứa một literal α được gán true với i = 1, 2, …, k. Trong đồ thị G, ta chọn ra k đỉnh (α, i) với i = 1, 2, …, k và α là literal được gán true. Với hai đỉnh bất kỳ được chọn ra (α, i) và (β, j), vì α và β cùng được gán true, nên $\alpha \neq \neg \beta$. Do đó k đỉnh đã chọn ra đó tạo thành đồ thị con đầy đủ của đồ thị G. Ngược lại, giả sử đồ thị G chứa đồ thị con đầy đủ k đỉnh. Ta đưa ra cách gán giá trị chân lý cho các biến trong công thức F như sau. Nếu (α, i) là một đỉnh của đồ thị con đầy đủ và α là biến x thì x được gán true, còn nếu α là phủ đỉnh của biến y thì y được gán false. Với cách gán này tất cả các câu tuyển C_i đều chứa một literal có giá trị true và do đó công thức F nhận giá trị true, tức là F thoả được. Mặt khác, việc xây dựng đồ thị G từ công thức F và việc xây dựng cách gán giá trị chân lý cho các biến của công thức F từ đồ thị con đầy đủ của đồ thị G, dễ dàng thấy, chỉ đòi hỏi thời gian đa thức. Do đó bài toán là NP - đầy đủ.



Hình 19.1. Đồ thị và chu trình Hamilton

Bài toán chu trình Hamilton

Một chu trình Hamilton trong đồ thị (định hướng hoặc vô hướng) G = (V, E) là một chu trình đi qua tất cả các đỉnh của đồ thị đúng một lần và trở về đỉnh xuất phát. Chẳng hạn, đồ thị định hướng trong hình 19.1 có chu trình Hamilton là E B F C D A E.

Định lý 2. Bài toán "xác định một đồ thị có chu trình Hamilton hay không" là bài toán NP - đầy đủ.

Chúng ta dễ dàng đưa ra thuật toán không đơn định với thời gian chạy đa thức để xác định một đồ thị là có chu trình Hamilton hay không (bài tập). Để chứng minh bài toán là NP – khó, chúng ta có thể chứng minh bài toán thoả được quy về nó. Chứng minh điều này là khá phức tạp, chúng ta không đưa ra.

Bài toán người bán hàng

Bài toán người bán hàng được phát biểu như sau: Cho đồ thị đầy đủ có trọng số, chúng ta cần tìm một chu trình Hamilton có độ dài ngắn nhất.

Định lý 3. Bài toán người bán hàng là bài toán NP – khó.

Chứng minh. Chúng ta sẽ chứng minh rằng, bài toán chu trình Hamilton quy về bài toán người bán hàng. Giả sử, G = (V, E) là một đồ thị định hướng có n đỉnh, $|V| = n$. Từ đồ thị G, chúng ta xây dựng một đồ thị định hướng đầy đủ có trọng số G' như sau. Các đỉnh của G' là các đỉnh của G. Với hai đỉnh bất kỳ u, v, thì độ dài của cung (u, v) trong G' được xác định là c(u, v) như sau:

c(u, v) = 1 nếu (u, v) thuộc E

c(u, v) = 2 nếu (u, v) không thuộc E.

Rõ ràng là, nếu đồ thị G có chu trình Hamilton thì chu trình Hamilton ngắn nhất trong đồ thị G' có độ dài là n. Còn nếu G không có chu trình Hamilton, thì chu trình Hamilton ngắn nhất trong G' sẽ có độ dài ≥ n + 1.

Lớp các bài toán NP - đầy đủ và NP – khó là rất rộng lớn. Chúng ta có thể tìm thấy các bài toán NP – khó trong rất nhiều lĩnh vực khác nhau của khoa học, công nghệ và trong đời sống. Khi phải giải quyết một bài toán là NP – khó, ta có thể tin tưởng (mặc dầu chưa chứng minh được) rằng, không tồn tại thuật toán với thời gian đa thức cho bài toán đó. Vì vậy, các nhà nghiên cứu chú ý tới các phương pháp giải gần đúng các bài toán NP – khó. Đặc biệt, trong các năm gần đây, người ta tập trung nghiên cứu các thuật toán heurictic, chẳng hạn các thuật toán di truyền, để giải quyết các bài toán NP – khó.

TÀI LIỆU THAM KHẢO

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison – Wesley, Reeding. Mass., 1974.
2. A. V. Aho, J. E. Hopcroft , and J. D. Ullman.*Data Strutures and Algorithms*. Addison - Wesley, Reeding, Mass., 1983.
3. G. Barassard and P. Bratley. *Algorithms: Theory and Practice*. Prentice – Hall, Englewood Cliffs, NJ, 1988.
4. F. M. Carrano, P. Helman and R. Veroff. *Data Abstraction and Problem Solving with C ++, Walls and Mirrors*. Addison - Wesley, Reeding, Mass., 1998.
5. J. O. Coplien. *Advanced C ++ Programming Styles and Idioms*. Addison - Wesley, Reeding, Mass., 1992.
6. B. Flamig. *Practical Data Structures in C ++*. John Wiley & Sons, New York, 1993.
7. G. H. Gonnet and R. Baeza – Yates. *Handbook of Algorithms and Data Structures*. Addison - Wesley, Reeding, Mass., 2d ed. , 1991.
8. G. L. Heileman. *Data Structures, Algorithms, and Object-Oriented Programming*. McGraw – Hill, New York, 1996.
9. E. Horowitz and S.Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, Rockwille, MD, 1978.
10. D. E. Knuth. *The Art of Computer Programming, vol 1, Fundermental Algorithms*. Addison - Wesley, Reeding, Mass., 3d ed.,1997
11. D. E. Knuth. *The Art of Computer Programming, vol 2, Seminumerial Algorithms*. Addison - Wesley, Reeding, Mass., 3d ed. ,1997.
12. D. E. Knuth. *The Art of Computer Programming, vol 3, Sorting and Searching.* Addison - Wesley, Reeding, Mass., 2d ed., 1998.
13. M.Main and W.Savitch. *Data Structures & Other Objects using C++*. Addison - Wesley, Reeding, Mass., 1998.

14. S. Meyers. *Effective C++*. Addison - Wesley, Reeding, Mass., 2d ed., 1998.
15. E. M. Reingole, J.Nievergelt and N. Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice – Hall, Englewood Cliffs, NJ, 1977.
16. H.Samet. *The Design and Analysis of Spatial Data Structures*. Addison - Wesley, Reeding, Mass., 1989.
17. R.Sedgewick. *Algorithms*. Addison - Wesley, Reeding, Mass., 2d ed., 1988.
18. R.Sedgewick. *Algorithms in C++*. Addison - Wesley, Reeding, Mass., 1992.
19. Đinh Mạnh Tường. *Cấu Trúc Dữ Liệu và Thuật Toán*. Nhà xuất bản Khoa Học và Kỹ Thuật, Hà nội, in lần thứ 3, 2003.
20. M.A. Weiss. *Data Structures and Algorithm Analysis in C ++*. Addison - Wesley, Reeding, Mass., 2d ed., 1999.
21. M.A. Weiss. *Data Structures and Problem Solving using C ++*. Addison - Wesley, Reeding, Mass., 2d ed., 2000.

CHƯƠNG 13

HỌ CÁC TẬP KHÔNG CẮT NHAU

Trong chương này chúng ta sẽ nghiên cứu KDLTT họ các tập không cắt nhau (the disjoint set ADT). Chúng ta có một họ các tập con không cắt nhau của một tập đã cho. Trên họ các tập con rời nhau đó, chúng ta chỉ quan tâm tới hai phép toán: phép hợp (union) và phép tìm (find) (các phép toán này sẽ được xác định trong mục 13.1). Chúng ta sẽ nghiên cứu cách cài đặt họ các tập con rời nhau sao cho hai phép toán hợp và tìm được thực hiện hiệu quả. KDLTT họ các tập không cắt nhau là đặc biệt hữu ích trong thiết kế thuật toán, chẳng hạn các thuật toán đồ thị, thuật toán tìm tổ tiên chung gần nhất của hai đỉnh bất kỳ trong một cây… Cuối chương này, chúng ta sẽ trình bày một vài ứng dụng: sử dụng các phép toán hợp và tìm để giải quyết vấn đề tương đương, để tạo ra một mê lộ.

13.1 KIỂU DỮ LIỆU TRỪU TƯỢNG HỌ CÁC TẬP KHÔNG CẮT NHAU

Giả sử chúng ta có một tập hữu hạn S gồm n đối tượng. Giả sử S_1, S_2, …, S_n là các tập con không rỗng của S sao cho:

$$S_1 \cup S_2 \cup \ldots \cup S_k = S$$

và $S_i \cap S_j = \phi$ với $i \neq j$

Nói một cách khác, các tập con S_1, …, S_k là một phân hoạch của tập S. Mỗi tập con S_i (i = 1, …, k) được gắn nhãn. Chúng ta chọn một phần tử trong tập con S_i làm nhãn cho S_i, phần tử được chọn làm nhãn đó đươc gọi là **phần tử đại biểu** của tập con S_i. Cần lưu ý rằng, chọn phần tử nào trong một tập con làm phần tử đại biểu cho tập đó cũng được, tuy nhiên trong các ứng dụng, các phần tử đại biểu cho các tập con thường được chọn theo một quy luật nào đó, chẳng hạn phần tử đại biểu là phần tử nhỏ nhất trong tập.

Các phép toán trên họ các tập con không cắt nhau của tập S được xác định như sau:

1. Khởi tạo. Tạo ra một phân hoạch của tập S gồm $|S|$ tập con, mỗi tập con chỉ chứa một phần tử của tập S.
2. Union(x, y). Hợp nhất hai tập con chứa đại biểu là x và y thành một tập con.
3. Find(x). Giả sử x là một phần tử của tập S, phép toán này cần trả về phần tử đại biểu của tập con chứa x.

Ví dụ. Giả sử S = {0, 1, 2, …, 9}. Phép toán khởi tạo sẽ tạo ra một phân hoạch của S gồm 10 tập con: {0}, {1}, …, {9}. Giả sử ta có họ các tập con không cắt nhau:

{0, 3, 5}, {1, 7}, {4, 8}, {2, 6, 9}

trong đó đại biểu của mỗi tập con là phần tử nhỏ nhất, chẳng hạn đại biểu của {2, 6, 9} là 2. Khi đó phép Union(1, 4) cho ta tập con {1, 7, 4, 8} với đại biểu là 1. Sau phép hợp này, ta có phân hoạch:

{0, 3, 5}, {1, 4, 7, 8} và {2, 6, 9}

Đến đây, nếu thực hiện phép toán Find(7), phép toán này sẽ trả về 1, vì 1 là đại biểu của tập con chứa 7.

Chú ý rằng, trong các ứng dụng, phép toán tìm thường được sử dụng để tìm câu trả lời cho câu hỏi: Với hai phần tử bất kỳ a và b của tập S, chúng có nằm trong cùng một tập con của phân hoạch hay không? Chúng sẽ thuộc cùng một tập con nếu Find(a) = Find(b) và không nếu Find(a) ≠ Find(b).

13.2 CÀI ĐẶT ĐƠN GIẢN

Giả sử S là một tập gồm n phần tử được đánh số từ 0 đến n – 1, S = {0, 1,…, n - 1}, mỗi i, 0 ≤ i ≤ n – 1, được xem như tên gọi của một phần tử trong tập S. Chúng ta có thể giả thiết như vậy, bởi vì chúng ta chỉ quan tâm tới một phần tử là thuộc hay không thuộc một tập con trong một phân hoạch, và không cần thực hiện một tính toán nào trên các dữ liệu về các phần tử của tập S.

Một cách đơn giản nhất để cài đặt một phân hoạch của tập S là sử dụng một mảng A[0 … n - 1], trong đó A[i] lưu đại biểu của tập con chứa i. Chẳng hạn, giả sử S = {0, 1, …, 9} và họ các tập con là {0, 3, 6}, {1, 2, 4, 9} và {5, 7, 8}, trong đó phần tử đại biểu của một tập con được chọn là phần tử nhỏ nhất trong tập con đó. Khi đó họ các tập con không cắt nhau trên được biểu diễn bởi mảng sau:

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	5	0	5	5	1

Với cách cài đặt này, phép toán Find(i) (tìm đại biểu của tập con chứa phần tử i) chỉ cần thời gian O(1), bởi vì chúng ta chỉ cần truy cập tới thành phần A[i] của mảng.

Bây giờ chúng ta xét xem phép hợp được thực hiện như thế nào. Giả sử ta cần lấy hợp của tập con có đại biểu là i với tập con có đại biểu là j. Để thực hiện phép toán Union(i, j), ta chỉ cần duyệt mảng, và tại mọi thành phần mảng chứa j ta thay j bởi i. Như vậy, phép hợp được thực hiện rất đơn giản,

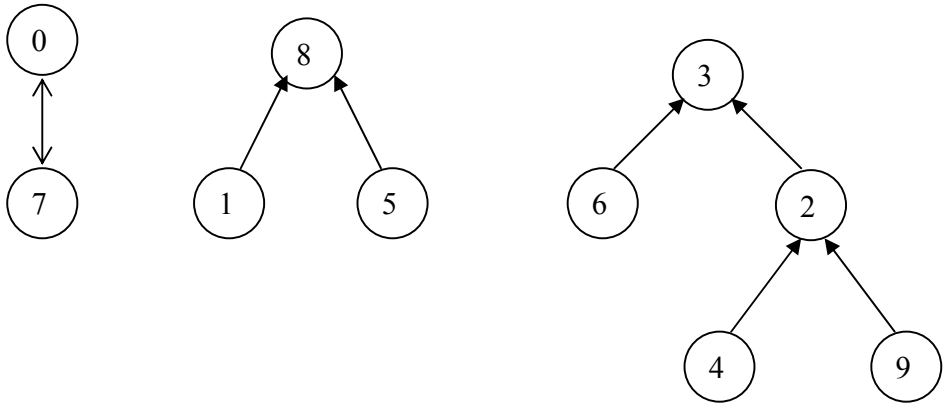
nhưng nó đòi hỏi thời gian O(n), vì chúng ta cần xem xét tất cả n thành phần của mảng.

Tóm lại, với cách cài đặt họ các tập rời nhau bởi mảng như đã trình bày, phép tìm chỉ cần thời gian hằng, song thời gian thực hiện phép hợp là tuyến tính theo cỡ của tập vũ trụ S.

Trong mục sau đây, chúng ta sẽ đưa ra cách cài đặt khác, trong cách cài đặt này phép hợp được thực hiện trong thời gian hằng; còn phép tìm mặc dầu không thể thực hiện trong thời gian hằng, song sự phân tích trả góp đã chỉ ra rằng, thời gian chạy trả góp của phép tìm là xấp xỉ với thời gian hằng.

13.3 CÀI ĐẶT BỞI CÂY

Chúng ta biểu diễn mỗi tập bởi một cây, gốc của cây chứa phần tử đại biểu của tập đó, mỗi đỉnh của cây không phải là gốc sẽ chứa một phần tử của tập đó và chứa con trỏ trỏ tới đỉnh cha. (Vì thế, cây được gọi là cây hướng lên (up – tree)). Như vậy, họ các tập không cắt nhau sẽ được biểu diễn bởi một rừng cây. Với giả thiết các phần tử của tập vũ trụ S được đánh số từ 0, 1, … đến n – 1, chúng ta có thể cài đặt rừng cây đã mô tả trên bởi một mảng A[0 … n-1], trong đó nếu cha của đỉnh i là đỉnh j thì j được lưu trong A[i], còn nếu i là gốc của một cây thì A[i] chứa –1. Ví dụ, giả sử chúng ta có họ tập con {0, 7}, {8, 1, 5} và {3, 6, 2, 4, 9} được biểu diễn bởi các cây trong hình 13.1a, khi đó rừng các cây này được cài đặt bởi mảng trong hình 13.1b.



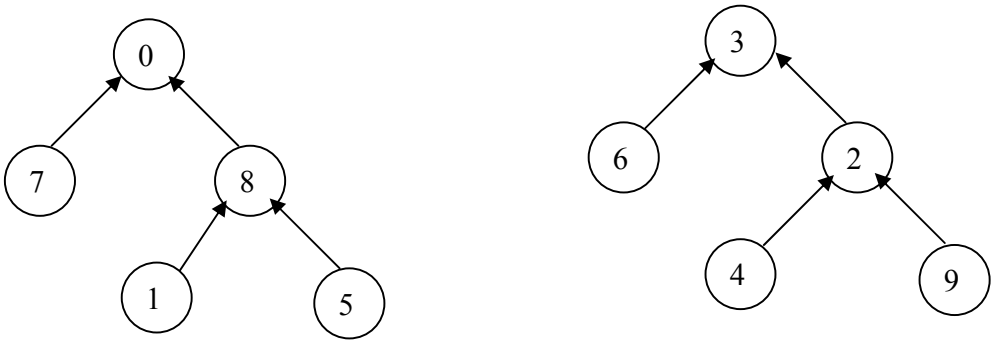
(a)

0	1	2	3	4	5	6	7	8	9
-1	8	3	-1	2	8	3	0	-1	2

(b)

Hình 13.1. (a) Rừng cây biểu diễn ba tập con {0, 7}, {8, 1, 5} và {3, 6, 2, 4, 9}.
(b) Mảng cài đặt rừng cây trong hình (a)

Nếu chúng ta biểu diễn các tập con bởi các cây hướng lên, thì phép hợp được thực hiện rất đơn giản. Chúng ta chỉ cần làm cho một trong hai cây trở thành cây con của gốc của cây kia. Chẳng hạn, nếu chúng ta thực hiện phép Union(0, 8), tức là lấy hợp của hai tập con được biểu diễn bởi hai cây đầu tiên trong hình 13.1a, ta có thể gắn cây thứ hai thành cây con của gốc của cây đầu tiên. Kết quả là từ ba cây trong hình 13.1a ta nhận được hai cây như trong hình 13.2a. Trên mảng, phép hợp Union(0, 8) được thực hiện chỉ bởi một phép gán A[8] = 0, mảng sau khi thực hiện phép hợp Union(0, 8)) được cho trong hình 13.2b. Rõ ràng là phép hợp theo phương pháp đã trình bày trên chỉ cần thời gian O(1).



(a)

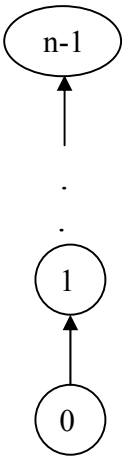
0	1	2	3	4	5	6	7	8	9
-1	8	3	-1	2	8	3	0	0	2

(b)

Hình 13.2. (a) Rừng cây sau khi thực hiện phép hợp Union(0, 8) từ rừng cây trong hình 13.1a.
(b) Mảng nhận được từ mảng 13.1b sau khi thực hiện phép hợp Union (0, 8)

Phép tìm phần tử đại biểu của tập con chứa i, Find(i), được thực hiện bằng cách đi từ đỉnh chứa i lên gốc cây. Thời gian thực hiện phép tìm đương nhiên là tỷ lệ với độ dài của đường đi trong cây. Giả sử chúng ta xuất phát từ họ n tập con một phần tử {0}, {1}, …, {n - 1}. Nếu chúng ta thực hiện liên tiếp n – 1 phép hợp hai tập con đầu tiên trong họ, và luôn luôn cho cây biểu diễn tập đầu tiên thành cây con của gốc của cây biểu diễn tập thứ hai, thì từ rừng cây trong hình 13.3a ta thu được một cây trong hình 13.3b. Cây này thực chất là một danh sách liên kết. Thời gian thực hiện phép tìm phần tử ở mức k trong cây này là O(k), và trong trường hợp xấu nhất là O(n), đó là trường hợp khi ta thực hiện Find(0).

(a)



(b)

Hình 13.3. (a) Rừng cây ban đầu.
(b) Cây kết quả của các phép hợp trong trường hợp xấu nhất

Tóm lại, nếu thực hiện phép hợp bằng cách cho một cây thành cây con của gốc của cây kia, thì phép hợp chỉ cần thời gian O(1), song thời gian thực hiện phép tìm trong trường hợp xấu nhất là O(n). Vấn đề được đặt ra là, sử dụng CTDL cây hướng lên để biểu diễn tập hợp, chúng ta có thể thực hiện cả phép hợp và phép tìm trong thời gian hằng được không? Các nghiên cứu sau đây sẽ trả lời câu hỏi này.

13.3.1 Phép hợp theo trọng số

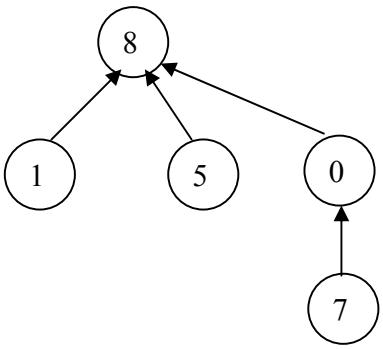
Chúng ta sẽ cải tiến phép hợp nhằm hạn chế độ cao của cây kết quả và do đó phép tìm sẽ nhanh hơn. Ý tưởng là, khi hợp nhất hai cây thành một cây, ta sẽ làm cho cây "nhỏ hơn" trở thành cây con của gốc của cây kia.

Chúng ta gọi **trọng số** của cây là số đỉnh của cây, trọng số của cây T được ký hiệu là w**eight(T)**. Phép hợp bây giờ được thực hiện như sau. Cho hai cây T_1 và T_2 cần hợp nhất, khi đó nếu weight(T_1) ≤ weight(T_2) thì ta gắn cây T_1 thành cây con của gốc của cây T_2, nếu ngược lại thì ta gắn cây T_2 thành cây con của gốc của cây T_1. Phép hợp được thực hiện theo quy tắc này được gọi là **phép hợp theo trọng số (union-by-weight)**.

Để cài đặt phép hợp theo trọng số, chúng ta cần lưu trọng số của các cây. Điều này không có gì là khó khăn. Trong mảng cài đặt rừng cây đã nói trước kia, nếu i là gốc của một cây, thì thay vì A[i] lưu –1, ta cho A[i] lưu –m, trong đó m là số đỉnh của cây gốc i. Chẳng hạn, các cây trong hình 13.1a bây giờ được cài đặt bởi mảng sau

0	1	2	3	4	5	6	7	8	9
-2	8	3	-5	2	8	3	0	-3	2

Nếu chúng ta lấy hợp theo trọng số của hai cây đầu tiên trong hình 13.1a, thì ta cần cho đỉnh gốc của cây thứ nhất trỏ tới cha nó là đỉnh gốc trong cây thứ hai, tức là

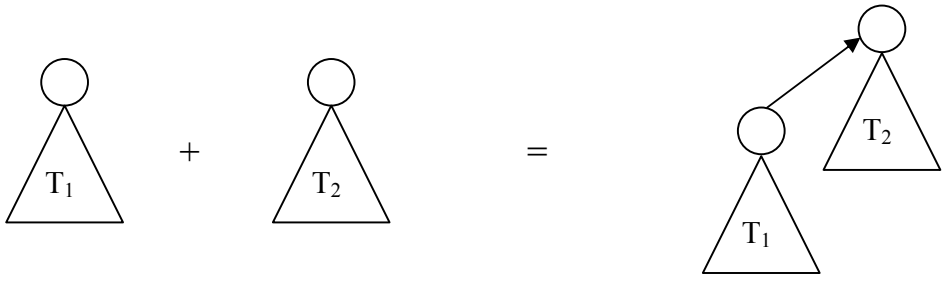


Trên mảng A, ta chỉ cần các lệnh gán A[0] = 8, và A[8] = -5, bởi vì trọng số của cây gốc 8 bây giờ là 5.

Rõ ràng là phép hợp theo trọng số chỉ cần thời gian O(1). Nếu thực hiện các phép hợp theo trọng số thì thời gian thực hiện phép tìm trên một cây bất kỳ là kết quả của một dãy phép hợp theo trọng số xuất phát từ các cây chỉ có một đỉnh sẽ được cải thiện ra sao? Chúng ta chứng minh định lý sau đây

Định lý 13.1. Độ cao của cây n đỉnh được tạo thành từ kết quả của một dãy phép hợp theo trọng số xuất phát từ các cây chỉ có một đỉnh không lớn hơn $\lfloor \log n \rfloor + 1$.

Nhớ lại rằng, độ cao của cây là số đỉnh nằm trên đường đi dài nhất từ gốc tới lá, độ cao của cây T sẽ được ký hiệu là height(T). Giả sử T là cây n đỉnh được tạo thành bởi một dãy phép hợp theo trọng số. Ta cần chứng minh $height(T) \le \lfloor \log n \rfloor + 1$. Chúng ta chứng minh bất đẳng thức này bằng quy nạp theo số đỉnh trong cây. Với cây chỉ có một đỉnh, khẳng định là đúng, vì $1 \le \lfloor \log 1 \rfloor + 1$. Giả sử khẳng định đã đúng cho tất cả các cây có số đỉnh ≤ n-1, và T là cây có n đỉnh được hình thành từ phép hợp hai cây T_1 và T_2. Giả sử cây T_1 có m đỉnh, cây T_2 có n – m đỉnh. Không mất tính tổng quát, ta có thể giả thiết rằng $1 \le m \le n/2$. Khi đó lấy hợp của hai cây T_1 và T_2 theo trọng số, cây T sẽ được tạo thành như sau:



cây T

Rõ ràng là, độ cao của cây T hoặc bằng độ cao của cây T_2 hoặc bằng độ cao của cây T_1 cộng thêm 1.

Trong trường hợp thứ nhất, ta có:

$$height(T) = height(T_2) \le \lfloor \log(n - m) \rfloor + 1$$
$$\le \lfloor \log n \rfloor + 1$$

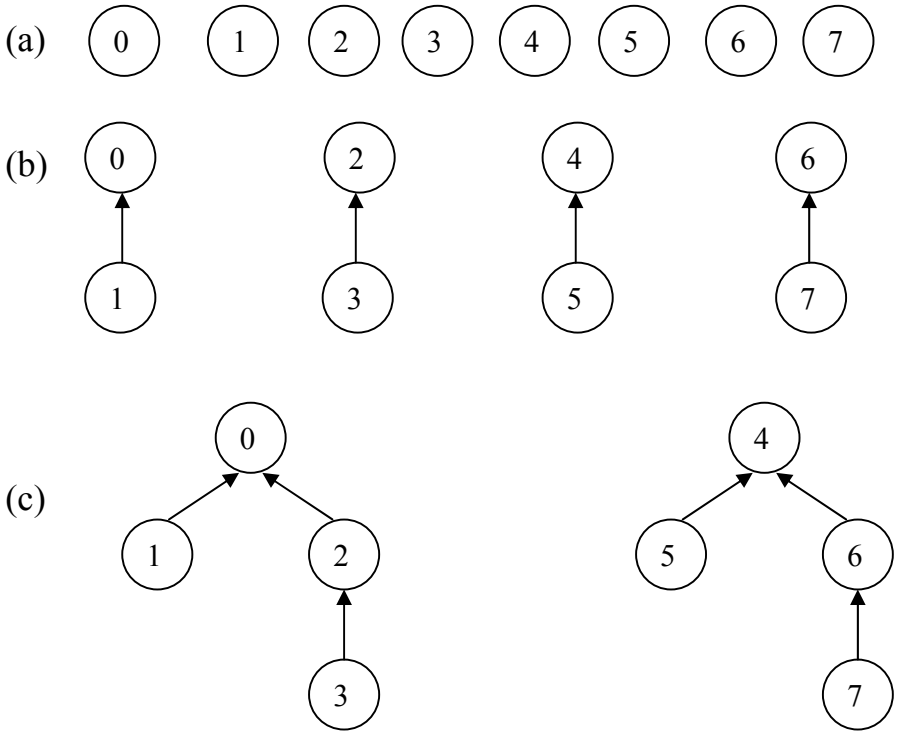
Nếu trường hợp thứ hai xảy ra thì:

$$height(T) = height(T_1) + 1$$
$$\le \lfloor \log m \rfloor + 2$$

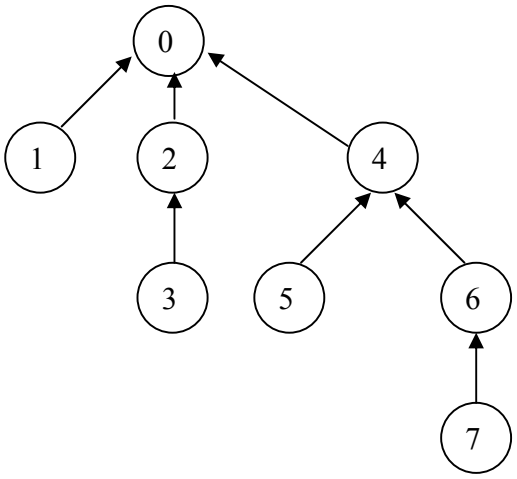
$\leq \lfloor \log(n/2) \rfloor + 2$

$\leq \lfloor \log n \rfloor + 1.$

Chúng ta đưa ra một ví dụ chứng tỏ rằng, cận trên $\lfloor \log n \rfloor + 1$ của độ cao của cây trong định lý 13.1 là không thể hạ thấp. Giả sử $n = 2^3$. Ban đầu ta có 8 cây 1 đỉnh, hình 13.4a. Thực hiện các phép hợp Union(0, 1), Union(2, 3), Union(4, 5), Union(6, 7) ta thu được các cây trong hình 13.4b. Thực hiện tiếp các phép hợp Union(0, 2), Union(4, 6) ta nhận được các cây trong hình 13.4c. Cuối cùng thực hiện phép hợp Union(0, 4) ta nhận được cây hình 13.4d. Cây này có độ cao $4 = \lfloor \log 8 \rfloor + 1$.



(d)



Hình 13.4. Cây trong trường hợp xấu nhất được tạo thành từ dãy phép hợp theo trọng số

Từ định lý 13.1 ta suy ra rằng, nếu sử dụng phép hợp theo trọng số, thì thời gian thực hiện phép tìm trong trường hợp xấu nhất là O(logn), trong đó n là số phần tử của tập vũ trụ. Nếu chúng ta tiến hành một dãy phép toán gồm n phép hợp và m phép tìm, thì thời gian trong trường hợp xấu nhất là O(mlogn).

Thay cho việc lưu trọng số của các cây, chúng ta có thể lưu độ cao của các cây và thực hiện phép hợp theo độ cao (union-by-height). Bạn đọc hãy đưa ra quy tắc thực hiện phép hợp theo độ cao và chứng minh khẳng định tương tự như trong định lý 13.1.

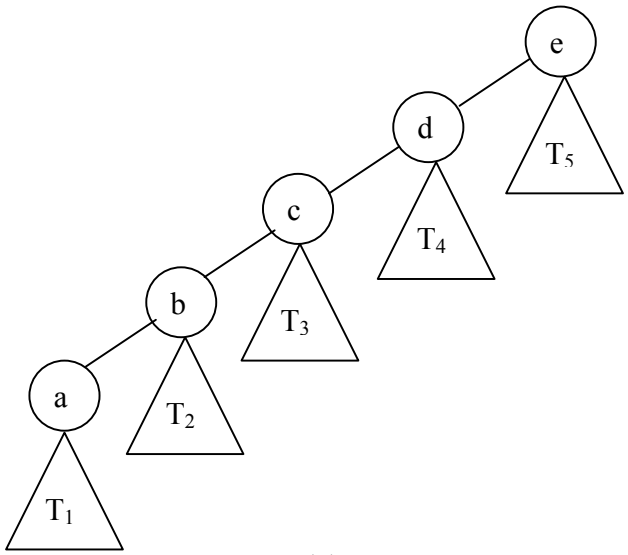
Một câu hỏi được đặt ra là, liệu có thể cải thiện thời gian thực hiện phép tìm được nữa không? Thật đáng ngạc nhiên, câu trả lời là có.

13.3.2 Phép tìm với nén đường

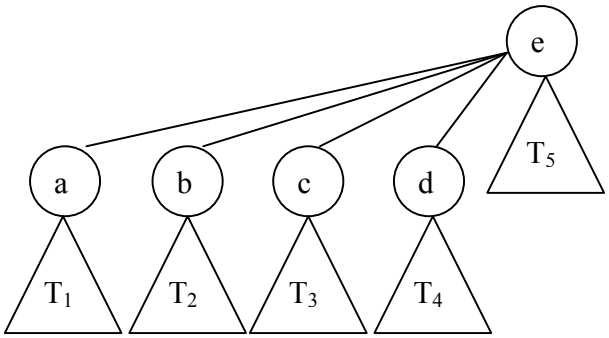
Nhằm cải thiện hơn nữa thời gian thực hiện phép tìm, khi thực hiện một phép tìm chúng ta sẽ thực hiện kèm theo một hành động được gọi là **nén đường (path-compression)**. Khi biểu diễn một tập con bởi cây với phần tử đại biểu của tập được chứa trong gốc của cây, để thực hiện phép tìm Find(a) chúng ta cần phải đi từ đỉnh a theo các con trỏ lên gốc cây. Đường từ a lên gốc cây được gọi là **đường tìm**, chẳng hạn, nếu thực hiện Find(7) trong cây hình 13.4d thì đường tìm là 7 – 6 – 4 – 0.

Nén đường có nghĩa là, khi thực hiện phép tìm Find(a), chúng ta thay đổi tất cả các con trỏ nằm trên đường tìm từ a lên gốc, cho chúng trỏ tới gốc

cây. Nén đường được minh hoạ trong hình 13.5, đường tìm được tìm ra khi thực hiện Find(a) được chỉ ra trong hình 13.5a, cây thu được sau khi nén đường được cho trong hình 13.5b.



(a)



(b)

Hình 13.5. (a) Cây trong đó Find(a) sinh ra một đường tìm. (b) Cây kết quả sau khi nén đường.

Cần lưu ý rằng, việc thực hiện nén đường kèm theo phép tìm không làm tăng thời gian thực hiện phép tìm, phép tìm vẫn chỉ đòi hỏi thời gian tỷ lệ với độ dài của đường tìm. Trực quan chúng ta thấy rằng, nén đường làm cho tất cả các đỉnh nằm trong các cây con gốc tại các đỉnh trên đường tìm trở nên gần gốc hơn. Chẳng hạn, các đỉnh trong các cây con T_1, T_2, T_3 của cây

trong hình 13.5b gần gốc hơn khi chúng ở trong cây hình 13.5a. Điều đó tạo điều kiện cho các phép tìm thực hiện sau sẽ hiệu quả hơn.

Rõ ràng là việc thực hiện phép tìm với nén đường không ảnh hưởng gì đến thời gian thực hiện phép hợp. Thời gian chạy của phép hợp vẫn còn là O(1). Nếu phép hợp là phép hợp theo trọng số (hoặc phép hợp theo độ cao), phép tìm là phép tìm với nén đường, thì thời gian thực hiện phép tìm sẽ ra sao? Người ta đã chứng minh được rằng, thời gian chạy trả góp của phép tìm là rất gần O(1). Kết luận này được suy ra từ định lý sau đây.

Định lý 13.2. Giả sử phép hợp theo trọng số và phép tìm với nén đường được thực hiện trên họ các tập con ban đầu gồm n tập một phần tử. Khi đó thời gian thực hiện một dãy m phép hợp và phép tìm, với m > n, là O(mlog*n), trong đó log*n là hàm được xác định như sau:

$$\log^* n = \min\{i \geq 0 \mid \log^{(i)} n \leq 1\}$$

với $\log^{(i)} n$ là

$$\log^{(i)} n = \underbrace{\log(\log(\ldots(\log n)))}_{i \text{ lần}}$$

Chứng minh định lý 13.2. là cực kỳ phức tạp, chúng ta không đưa ra ở đây. Để hiểu rõ bản chất của khẳng định trong định lý 13.2, chúng ta cần biết đặc điểm của hàm log*n. Hàm này tăng cực kỳ chậm. Bạn sẽ thấy được điều này qua một số giá trị của hàm:

$\log^* 2 = 1$

$\log^* 4 = 2$

$\log^* 16 = 3$

$\log^* 65536 = 4$

$\log^* 2^{65536} = 5$

Tức là để giá trị của hàm vượt quá 5 thì n cần phải lớn hơn 2^{65536}. Đây là con số cực kỳ khổng lồ, nó lớn hơn số các phần tử trong vũ trụ mà ta quan sát được! Đó là cơ sở để ta có thể xem rằng O(mlog*n) là O(m).

13.4 ỨNG DỤNG

Các phép toán hợp và tìm trên họ các tập không cắt nhau được sử dụng trong thiết kế và cài đặt nhiều thuật toán cho các vấn đề khác nhau, chẳng hạn thuật toán tìm cây bao trùm ngắn nhất của đồ thị vô hướng (thuật toán Kruskal), thuật toán tìm tổ tiên chung gần nhất của hai đỉnh bất kỳ trong một cây (thuật toán này rất quan trọng trong các áp dụng của lý thuyết

đồ thị, trong Biomformatics). Trong mục này chúng ta sẽ minh hoạ cách sử dụng phép hợp và phép tìm để thiết kế thuật toán thông qua hai áp dụng.

13.4.1 Vấn đề tương đương

Một quan hệ R trên tập S là một họ nào đó các cặp phần tử của S. Nếu cặp (a, b) $\in$ R ta sẽ nói a có quan hệ R với b và ký hiệu aRb. Quan hệ R được gọi là **quan hệ tương đương** nếu nó thoả mãn các tính chất sau:

- Tính phản xạ: aRa với mọi a $\in$ S.
- Tính đối xứng: aRb nếu và chỉ nếu bRa.
- Tính bắc cầu: aRb và bRc kéo theo aRc.

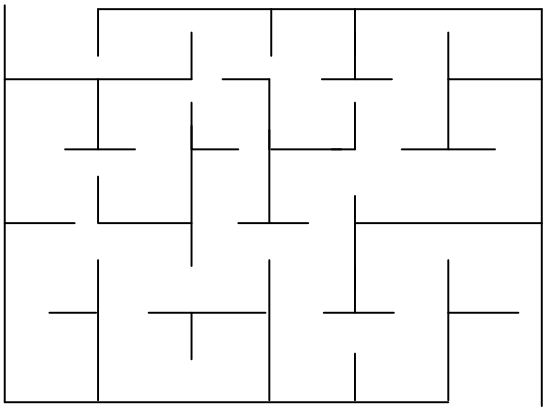
Chúng ta sẽ ký hiệu quan hệ tương đương bởi ~. Lớp tương đương của phần tử x $\in$ S, ký hiệu là [x], gồm tất cả a $\in$ S mà a ~ x (a tương đương với x). Dễ dàng chứng minh được rằng, các lớp tương đương tạo thành một phân hoạch của tập S.

Đối với một quan hệ tương đương, vấn đề được đặt ra là: với hai phần tử bất kỳ a và b của S, chúng ta cần biết a có tương đương với b hay không? Nếu quan hệ tương đương đã hoàn toàn xác định trước, vấn đề được giải quyết rất đơn giản: đánh số của phần tử của S từ 0, 1, … đến n – 1, lưu quan hệ tương đương trong mảng boolean A[0… n-1], [0 … n-1]; truy cập thành phần A[i], [j] ta sẽ biết i và j là tương đương hay không. Tuy nhiên, thông thường chúng ta chỉ biết trước môt tập nào đó các cặp tương đương (i, j). Chúng ta cần đưa ra câu trả lời nhanh cho câu hỏi: với cặp phần tử mới (a, b), chúng có tương đương hay không?

Từ các cặp tương đương (i, j) đã cho, chúng ta sẽ tạo ra các lớp tương đương. Điều này được tiến hành như sau. Ban đầu mỗi phần tử của tập S là một lớp tương đương chỉ chứa một phần tử. Với mỗi cặp tương đương (i, j), ta sử dụng các phép tìm Find(i) và Find(j). Nếu chúng trả về các đại biểu x và y khác nhau, thì ta sử dụng phép hợp Union(x, y) để kết hợp hai lớp tương đương [x] và [y] thành một lớp mới và huỷ bỏ hai lớp đó. Còn nếu Find(i) và Find(j) trả về cùng một đại biểu thì có nghĩa là i và j đã thuộc cùng một lớp tương đương, và ta không cần phải làm gì với cặp (i, j) này. Sau khi đã tạo ra các lớp tương đương từ các cặp tương đương (i, j) đã cho, để có câu trả lời cho cặp (a, b) được hỏi, chúng ta chỉ cần sử dụng các phép tìm Find(a) và Find(b).

13.4.2 Tạo ra mê lộ

Hình 13.5 biểu diễn một mê lộ 5 x 6. Chúng ta quan niệm một mê lộ như một lâu đài hình chữ nhật gồm m x m phòng, mỗi phòng hoặc là có cửa thông sang phòng bên cạnh, hoặc là bị ngăn cách với phòng bên cạnh bởi bức tường (mỗi phòng kề với 4 phòng lân cận, trừ các phòng ở cạnh tường bao ngoài có số phòng kề ít hơn). Lâu đài có một cửa vào ở phòng góc trên bên trái và một lối ra ở phòng góc dưới bên phải (xem hình 13.5). Từ một phòng bất kỳ ta có thể đi tới một phòng bất kỳ khác trong lâu đài. Mê lộ cần được thiết kế sao cho người đi vào lâu đài tìm được lối ra khỏi lâu đài là hết sức khó khăn.



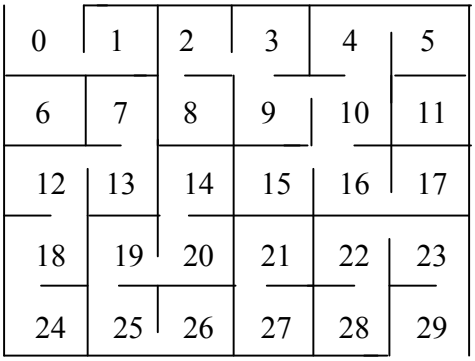
Hình 13.5. Một mê lộ

Sau đây chúng ta sẽ trình bày một thuật toán thiết kế mê lộ. Chúng ta đánh số các phòng từ 0 đến p. Ban đầu tất cả các phòng đều ngăn cách bởi bức tường với các phòng kề như trong hình 13.6. Điều này có nghĩa là ban đầu ta xem mỗi phòng ở trong một tập con riêng biệt. Tới một lúc nào đó, mỗi một tập con sẽ gồm tất cả các phòng liên thông với nhau, tức là từ một phòng bất kỳ trong tập ta có thể đi tới phòng bất kỳ khác trong tập. Tại mỗi bước chúng ta sẽ chọn ngẫu nhiên một bức tường ngăn cách 2 phòng a và b. Sử dụng phép tìm, nếu Find(a) = x, Find(b) = y và $x \neq y$, thì điều đó có nghĩa là a và b không liên thông với nhau, và ta phá bức tường để cửa cho 2 phòng a và b thông với nhau. Điều này được thực hiện bằng cách sử dụng phép hợp Union(x, y) để hợp nhất tập con chứa a với tập con chứa b. Còn nếu a và b đã liên thông với nhau, tức Find(a) = Find(b), thì ta để nguyên bức tường đó. Lặp lại quá trình trên cho tới khi tất cả các phòng là liên thông với nhau.

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

Hình 13.6. Khởi tạo mỗi phòng ở trong một tập con riêng (mỗi phòng không có cửa thông sang phòng bên cạnh)

Để thấy được thuật toán trên làm việc như thế nào, giả sử tới một giai đoạn nào đó chúng ta có trạng thái của các phòng như trong hình 13.7. Ở tình huống này, chúng ta có họ các tập các phòng liên thông với nhau như sau: {0, 1}, {2, 3, 4, 5, 8, 9, 10, 15, 16, 17}, {6}, {7, 12, 13, 18, 24}, {11}, {14, 19, 20, 25, 26}, {21, 27}, {22, 23, 28, 29}. Giả sử đại biểu của một tập con là phòng có số hiệu nhỏ nhất. Bây giờ chúng ta chọn ngẫu nhiên một bức chưa đặt cửa, chẳng hạn đó là bức tường ngăn cách phòng 9 và phòng 15. Chúng ta thấy rằng phòng 9 và phòng 15 đã liên thông với nhau, bởi vì Find(9) = Find(12) = 2, tức là chúng ở trong cùng một tập, và do đó ta để nguyên bức tường này. Chọn ngẫu nhiên một bức tường khác, chẳng hạn bức tường ngăn cách phòng 18 và phòng 19. Phép tìm Find(18) cho ta đại biểu của tập con chứa 18 là 7, phép tìm Find(19) cho ta đại biểu của tập con chứa 19 là 14, như vậy 18 và 19 thuộc các tập con khác nhau, và điều này có nghĩa là chúng không liên thông với nhau. Do đó ta cần đục cửa ở bức tường ngăn cách phòng 18 và 19. Sử dụng phép hợp để hợp nhất tập con chứa 18 và tập con chứa 19 thành một tập con.



Hình 13.6. Một tình huống trong quá trình thực hiện thuật toán xây dựng mê lộ