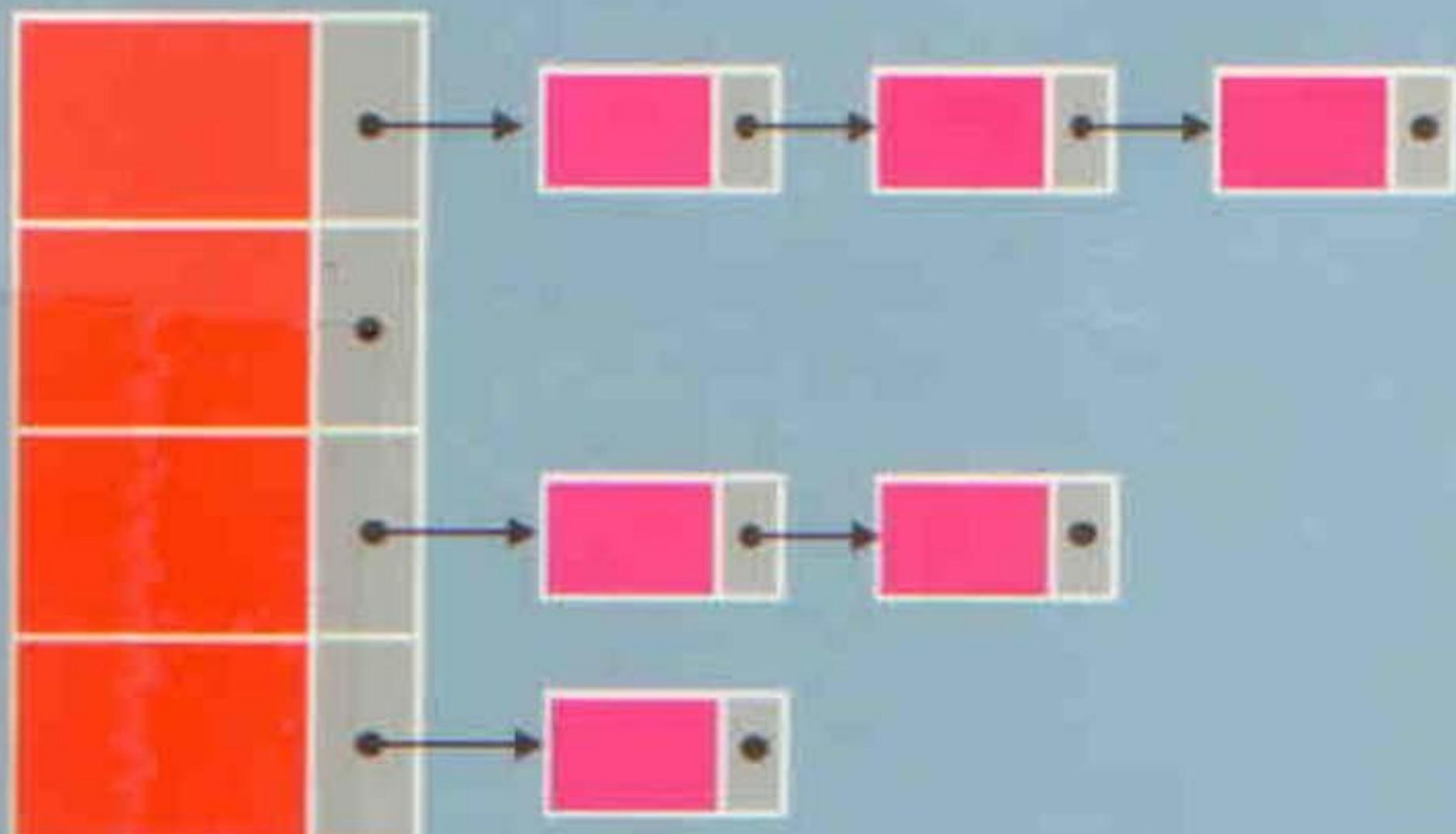


ĐINH MẠNH TƯỜNG

# CẤU TRÚC DỮ LIỆU & THUẬT TOÁN



NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

TS. ĐINH MẠNH TƯỜNG

**CẤU TRÚC DỮ LIỆU  
và  
THUẬT TOÁN**

(In lần thứ hai)



**NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT  
HÀ NỘI - 2001**

QUỐC HỘ KHẨU

NGHỆ TÂM

THỦ TỤC THIẾU THIẾT  
KHÔNG

1. KẾT QUẢ

5 - 591

84 - 290 - 00

KHKT - 01

## LỜI NÓI ĐẦU

Cấu trúc dữ liệu (*CTDL*) và thuật toán là môn học đóng vai trò quan trọng trong quá trình đào tạo cử nhân các ngành khoa học máy tính và công nghệ thông tin. Cuốn sách này được hình thành trên cơ sở giáo trình *CTDL* và thuật toán mà chúng tôi đã giảng dạy trong nhiều năm tại khoa Công nghệ thông tin, Đại học KHTN, Đại học quốc gia Hà Nội.

Trong quá trình phát triển chương trình, nhất là khi phát triển các hệ mềm lớn, chúng ta cần đến hai dạng biểu diễn dữ liệu: biểu diễn trừu tượng và biểu diễn cụ thể. Trong giai đoạn thiết kế chương trình ta cần sử dụng biểu diễn trừu tượng của dữ liệu, tới giai đoạn cài đặt chương trình ta cần sử dụng biểu diễn cụ thể của dữ liệu. Biểu diễn trừu tượng của dữ liệu được xác định bởi mô hình dữ liệu (*MHDL*), đó là mô hình toán học của các đối tượng dữ liệu cùng với các phép toán thực hiện trên các đối tượng đó. Chẳng hạn, các *MHDL* quan trọng là danh sách, cây, tập hợp,... Khi chúng ta sử dụng *MHDL* với một số xác định các phép toán nào đó, chúng ta sẽ có một kiểu dữ liệu trừu tượng (Abstract Data Type). Chẳng hạn, một danh sách với hai phép toán xen vào và loại bỏ chỉ được phép thực hiện ở một đầu của danh sách lập thành kiểu dữ liệu trừu tượng (*KDLTT*) ngăn xếp (Stack).

Biểu diễn cụ thể của dữ liệu là biểu diễn xác định cách lưu giữ vật lý của dữ liệu trong bộ nhớ của máy tính. Biểu diễn cụ thể của dữ liệu được xác định bởi các *CTDL*. Các *CTDL* được mô tả trong ngôn ngữ lập trình mà ta sử dụng. Từ biểu diễn trừu tượng, ta có thể chuyển dịch thành các biểu diễn cụ thể khác nhau. Hay nói cách khác, từ các *MHDL*, hoặc từ các *KDLTT*, ta có thể chuyển đổi thành các *CTDL* khác nhau. Chẳng hạn, ta có thể cài đặt danh sách bởi *CTDL* mảng hoặc bởi *CTDL* danh sách liên kết. Khi cài đặt *MHDL* bởi *CTDL* nào đó, thì các phép toán trên mô hình được thực hiện bởi các thao tác cần thiết trên *CTDL* đó.

Vì vậy, trong phần nói về *CTDL*, phương pháp trình bày của chúng tôi như sau: Đầu tiên chúng tôi đặc tả các *MHDL* (hoặc *KDLTT*), sau đó đưa ra các *CTDL* biểu diễn chúng. Với mỗi *CTDL* được sử dụng để biểu diễn *MHDL* (hoặc *KDLTT*), chúng ta đưa ra các thủ tục hoặc hàm thực hiện các phép toán cần thiết. Chúng ta mô tả các *CTDL* và các thủ tục (hoặc hàm) trong ngôn ngữ Pascal. Tuy nhiên, bạn đọc sẽ không khó khăn gì khi chuyển dịch các mô tả đó sang các ngôn ngữ lập trình khác, chẳng hạn C.

Trong phần thuật toán, chúng tôi tập trung vào các kỹ thuật thiết kế thuật toán, chẳng hạn các chiến lược chia để trị, tham ăn, quay lui,..., bởi vì có nắm được các kỹ thuật này chúng ta mới có thể tìm ra thuật toán cho các vấn đề được đặt ra. Các lớp thuật toán giải quyết các vấn đề trong các lĩnh vực khác nhau: khoa học, kỹ thuật, công nghệ, kinh tế,... là hết sức phong phú. Chúng ta

chỉ trình bày hai lớp thuật toán: các thuật toán sắp xếp và các thuật toán đồ thị. Sở dĩ như vậy, vì đó là các thuật toán được sử dụng rộng rãi nhất, và hơn nữa, các thuật toán này chứa đựng nhiều kỹ thuật quan trọng mà chúng ta cần biết để sử dụng.

Chúng tôi không chú ý nhiều đến phân tích thuật toán. Song trong phần lớn các thuật toán được trình bày chúng tôi cũng đưa ra đánh giá cận trên của thời gian chạy.

Nội dung cuốn sách gồm mười chương:

- Chương 1 trình bày khái niệm thuật toán, đưa ra ký hiệu ô lớn và các phương pháp đánh giá thời gian thực hiện thuật toán bởi ký hiệu ô lớn.
- Chương 2 trình bày các khái niệm cơ bản: kiểu dữ liệu, *CTDL*, *MHDL*, *KDLTT*. Chương này cũng trình bày hệ kiểu của ngôn ngữ Pascal để qua đó chúng ta thấy được phương pháp kiến tạo nên các kiểu dữ liệu của người sử dụng trong các ngôn ngữ lập trình bậc cao.
- Chương 3 trình bày *MHDL* danh sách, các *CTDL* cài đặt danh sách, thủ tục thực hiện các phép toán trên danh sách trong mỗi cách cài đặt. Hai *KDLTT* quan trọng là hàng đợi và ngăn xếp sẽ được nghiên cứu.
- Chương 4 trình bày các *CTDL* biểu diễn cây, và nghiên cứu một lớp cây đặc biệt là cây nhị phân.
- Chương 5 trình bày *MHDL* tập hợp và hai *KDLTT* là từ điển và hàng ưu tiên. Chương này chú trọng đến phương pháp băm, một kỹ thuật quan trọng để cài đặt tập hợp.
- Chương 6 nghiên cứu *MHDL* bảng và các kỹ thuật cài đặt bảng chữ nhặt.
- Chương 7 trình bày các *CTDL* ở bộ nhớ ngoài : file băm, file có chỉ số, B-cây.
- Chương 8 trình bày các kỹ thuật thiết kế thuật toán.
- Chương 9 nghiên cứu các thuật toán sắp xếp.
- Chương 10 nghiên cứu các thuật toán đồ thị.

Để đọc cuốn sách này độc giả chỉ cần biết lập trình trên ngôn ngữ Pascal, ngoài ra không đòi hỏi ở độc giả kiến thức chuyên môn nào khác. Ngoài đối tượng chính là sinh viên các ngành Khoa học máy tính và Công nghệ thông tin, cuốn sách còn cần cho tất cả mọi người (kể cả học sinh các lớp chuyên chọn về Tin học) muốn nâng cao kiến thức cơ bản về Tin học.

Cuốn sách chắc chắn không tránh khỏi những thiếu sót, chúng tôi chân thành mong nhận được ý kiến phê bình của độc giả. Thư góp ý xin gửi về Nhà xuất bản Khoa học và Kỹ thuật - 70 Trần Hưng Đạo Hà Nội.

Tác giả

# *Chương 1*

## **THUẬT TOÁN VÀ PHÂN TÍCH THUẬT TOÁN**

### **1.1. THUẬT TOÁN**

#### **1.1.1. Khái niệm thuật toán**

Thuật toán (algorithm) là một trong những khái niệm quan trọng nhất trong tin học. Thuật ngữ thuật toán xuất phát từ nhà toán học Arập Abu Ja'far Mohammed ibn Musa al Khowarizmi (khoảng năm 825). Tuy nhiên lúc bấy giờ và trong nhiều thế kỷ sau, nó không mang nội dung như ngày nay chúng ta quan niệm. Thuật toán nổi tiếng nhất, có từ thời cổ Hy lạp là thuật toán Euclid, thuật toán tìm ước chung lớn nhất của hai số nguyên. Có thể mô tả thuật toán này như sau :

#### ***Thuật toán Euclid***

Input : m, n nguyên dương

Output : g, ước chung lớn nhất của m và n.

#### ***Phương pháp :***

Bước 1 : Tìm r, phần dư của phép chia m cho n

Bước 2 : Nếu  $r = 0$ , thì  $g \leftarrow n$  (gán giá trị của n cho g) và dừng lại.  
Trong trường hợp ngược lại ( $r \neq 0$ ), thì  $m \leftarrow n$ ,  $n \leftarrow r$  và quay lại bước 1.

Chúng ta có thể quan niệm các bước cần thực hiện để làm một món ăn, được mô tả trong các sách dạy chế biến món ăn, là một thuật toán. Cũng có thể xem các bước cần tiến hành để gấp đồ chơi bằng giấy, được trình bày trong sách dạy gấp đồ chơi bằng giấy, là thuật toán. Phương pháp thực hiện phép cộng, nhân các số nguyên, chúng ta đã học ở cấp I cũng là các thuật toán.

Trong sách này chúng ta chỉ cần đến định nghĩa không hình thức về thuật toán :

Thuật toán là một dãy hữu hạn các bước, mỗi bước mô tả chính xác các phép toán hoặc hành động cần thực hiện, để giải quyết một vấn đề.

(Từ điển Oxford Dictionary định nghĩa, Algorithm: set of well-defined rules for solving a problem in a finite number of steps.)

Định nghĩa này, tất nhiên, còn chứa đựng nhiều điều chưa rõ ràng. Để hiểu đầy đủ ý nghĩa của khái niệm thuật toán, chúng ta nêu ra 5 đặc trưng sau đây của thuật toán (Xem D.E. Knuth [1968]. The Art of Computer Programming, vol. I. Fundamental Algorithms).

**1. Input.** Mỗi thuật toán cần có một số (có thể bằng không) dữ liệu vào (input). Đó là các giá trị cần đưa vào khi thuật toán bắt đầu làm việc. Các dữ liệu này cần được lấy từ các tập hợp giá trị cụ thể nào đó. Chẳng hạn, trong thuật toán Euclid trên, m và n là các dữ liệu vào lấy từ tập các số nguyên dương.

**2. Output.** Mỗi thuật toán cần có một hoặc nhiều dữ liệu ra (output). Đó là các giá trị có quan hệ hoàn toàn xác định với các dữ liệu vào và là kết quả của sự thực hiện thuật toán. Trong thuật toán Euclid có một dữ liệu ra, đó là g, khi thực hiện đến bước 2 và phải dừng lại (trường hợp  $r = 0$ ), giá trị của g là ước chung lớn nhất của m và n.

**3. Tính xác định.** Mỗi bước của thuật toán cần phải được mô tả một cách chính xác, chỉ có một cách hiểu duy nhất. Hiển nhiên, đây là một đòi hỏi rất quan trọng. Bởi vì, nếu một bước có thể hiểu theo nhiều cách khác nhau, thì cùng một dữ liệu vào, những người thực hiện thuật toán khác nhau có thể dẫn đến các kết quả khác nhau. Nếu ta mô tả thuật toán bằng ngôn ngữ thông thường, không có gì đảm bảo người đọc hiểu đúng ý của người viết thuật toán. Để đảm bảo đòi hỏi này, thuật toán cần được mô tả trong các ngôn ngữ lập trình (ngôn ngữ máy, hợp ngữ hoặc ngôn ngữ bậc cao như Pascal, Fortran, C, ...). Trong các ngôn ngữ này, các mệnh đề được tạo thành theo các qui tắc cú pháp nghiêm ngặt và chỉ có một ý nghĩa duy nhất.

**4. Tính khả thi.** Tất cả các phép toán có mặt trong các bước của thuật toán phải đủ đơn giản. Điều đó có nghĩa là, các phép toán phải sao cho, ít nhất về nguyên tắc có thể thực hiện được bởi con người chỉ bằng

giấy trắng và bút chì trong một khoảng thời gian hữu hạn. Chẳng hạn trong thuật toán Euclid, ta chỉ cần thực hiện các phép chia các số nguyên, các phép gán và các phép so sánh để biết  $r = 0$  hay  $r \neq 0$ .

**5. Tính dừng.** Với mọi bộ dữ liệu vào thỏa mãn các điều kiện của dữ liệu vào (tức là được lấy ra từ các tập giá trị của các dữ liệu vào), thuật toán phải dừng lại sau một số hữu hạn bước thực hiện. Chẳng hạn, thuật toán Euclid thỏa mãn điều kiện này. Bởi vì giá trị của  $r$  luôn nhỏ hơn  $n$  (khi thực hiện bước 1), nếu  $r \neq 0$  thì giá trị của  $n$  ở bước 2 là giá trị của  $r$  ở bước trước, ta có  $n > r = n_1 > r_1 = n_2 > r_2 \dots$ . Dãy số nguyên dương giảm dần cần phải kết thúc ở 0, do đó sau một số bước nào đó giá trị của  $r$  phải bằng 0, thuật toán dừng.

Với một vấn đề đặt ra, có thể có một hoặc nhiều thuật toán giải. Một vấn đề có thuật toán giải gọi là vấn đề giải được (bằng thuật toán). Chẳng hạn, vấn đề tìm nghiệm của hệ phương trình tuyến tính là vấn đề giải được. Một vấn đề không tồn tại thuật toán giải gọi là vấn đề không giải được (bằng thuật toán). Một trong những thành tựu xuất sắc nhất của toán học thế kỷ 20 là đã tìm ra những vấn đề không giải được bằng thuật toán.

Trên đây chúng ta đã trình bày định nghĩa không hình thức về thuật toán. Có thể xác định khái niệm thuật toán một cách chính xác bằng cách sử dụng các hệ hình thức. Có nhiều hệ hình thức mô tả thuật toán : máy Turing, hệ thuật toán Markop, văn phạm Chomsky dạng 0, ... Song vấn đề này không thuộc phạm vi những vấn đề mà chúng ta quan tâm. Đối với chúng ta, chỉ sự hiểu biết trực quan, không hình thức về khái niệm thuật toán là đủ.

### 1.1.2. Biểu diễn thuật toán

Có nhiều phương pháp biểu diễn thuật toán. Có thể biểu diễn thuật toán bằng danh sách các bước, các bước được diễn đạt bằng ngôn ngữ thông thường và các ký hiệu toán học. Có thể biểu diễn thuật toán bằng sơ đồ khối. Tuy nhiên, như đã nói, để đảm bảo tính xác định của thuật toán, thuật toán cần được viết trong các ngôn ngữ lập trình. Một chương trình là sự biểu diễn của một thuật toán trong ngôn ngữ lập trình đã chọn. Để đọc dễ dàng các phần tiếp theo, độc giả cần làm quen với ngôn

ngữ lập trình Pascal. Đó là ngôn ngữ thường được chọn để trình bày các thuật toán trong sách báo.

Trong sách này chúng ta sẽ trình bày các thuật toán bởi các thủ tục hoặc hàm trong ngôn ngữ tựa Pascal. Nói là tựa Pascal, bởi vì trong nhiều trường hợp, để cho ngắn gọn, chúng ta không hoàn toàn tuân theo các qui định của Pascal. Ngoài ra, có trường hợp, chúng ta sử dụng cả các ký hiệu toán học và các mệnh đề trong ngôn ngữ tự nhiên (tiếng Anh hoặc tiếng Việt). Sau đây là một số ví dụ.

**Ví dụ 1 :** Thuật toán kiểm tra số nguyên  $n$  ( $n > 2$ ) có là số nguyên tố hay không.

```

function Ngto (n : integer) : boolean ;
    var a : integer ;
begin   NGTO := true ;
        a := 2 ;
        while a <= sqrt (n) do
            begin
                if n mod a = 0 then NGTO := false ;
                a := a + 1 ;
            end
        end ;

```

**Ví dụ 2 :** Bài toán tháp Hà Nội. Có ba cọc A, B, C. Lúc đầu, ở cọc A có  $n$  đĩa được lồng vào theo thứ tự nhỏ dần từ thấp lên cao. Đòi hỏi phải chuyển  $n$  đĩa từ cọc A sang cọc B, được quyền sử dụng cọc C làm vị trí trung gian, nhưng không được phép đặt đĩa lớn lên trên đĩa nhỏ.

Để chuyển  $n$  đĩa từ cọc A sang cọc B, ta thực hiện thủ tục sau : đầu tiên là chuyển  $n - 1$  đĩa bên trên từ cọc A sang cọc C, sau đó chuyển đĩa lớn nhất từ cọc A sang cọc B. Đến đây, chỉ cần chuyển  $n - 1$  đĩa từ cọc C sang cọc B. Việc chuyển  $n - 1$  đĩa từ cọc này sang cọc kia được thực hiện bằng cách áp dụng đệ quy thủ tục trên.

```

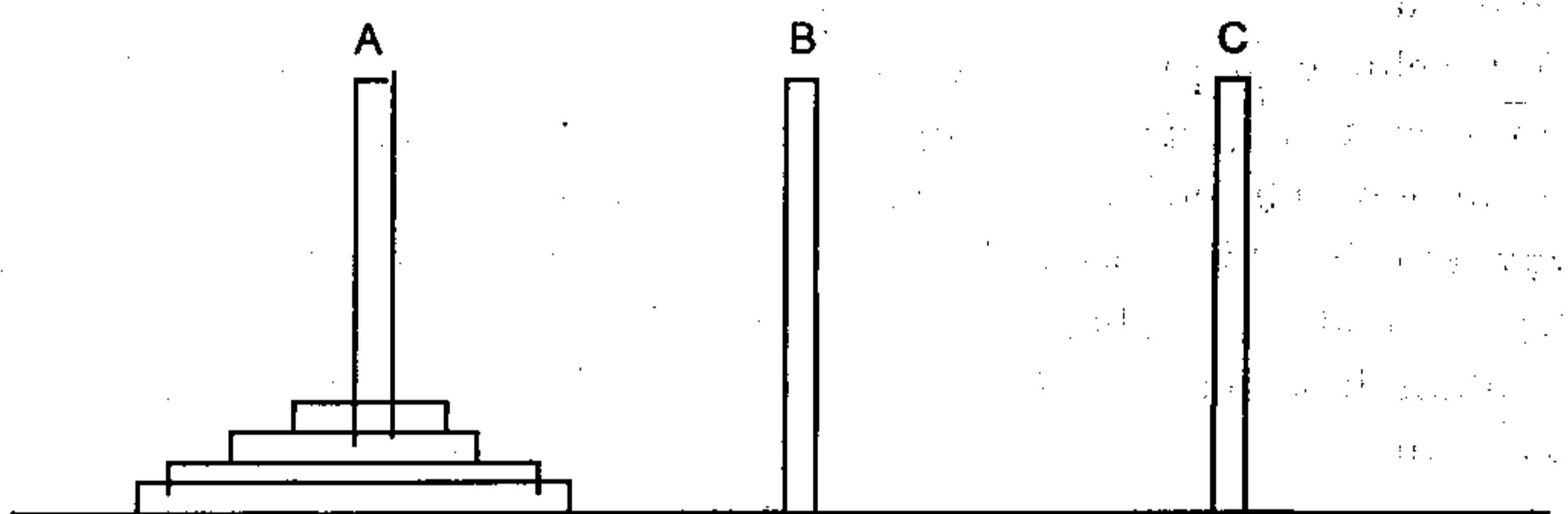
Procedure Move (n, A, B, C) ;
    {thủ tục chuyển n đĩa từ cọc A sang cọc B}
begin
    if n=1 then chuyển một đĩa từ cọc A sang cọc B
    else

```

```

begin
    MOVE (n-1, A, C, B) ;
    Chuyển một đĩa từ cọc A sang cọc B ;
    MOVE (n-1, C, B, A) ;
end;
end ;

```



### 1.1.3. Các vấn đề liên quan đến thuật toán

#### *Thiết kế thuật toán*

Để giải một bài toán trên MTĐT, điều trước tiên là chúng ta phải có thuật toán. Một câu hỏi đặt ra là, làm thế nào để tìm ra thuật toán cho một bài toán đã đặt ra? Lớp các bài toán được đặt ra từ các ngành khoa học kỹ thuật, từ các lĩnh vực hoạt động của con người là hết sức phong phú và đa dạng. Các thuật toán giải các lớp bài toán khác nhau cũng rất khác nhau. Tuy nhiên, có một số kỹ thuật thiết kế thuật toán chung như chia-để-trị (divide-and-conquer), phương pháp tham lam (greedy method), qui hoạch động (dynamic programming), ... Việc nắm được các chiến lược thiết kế thuật toán này là hết sức cần thiết, nó giúp cho bạn dễ tìm ra các thuật toán mới cho các bài toán của bạn. Các đề tài này sẽ được đề cập đến trong chương 8 của sách này.

#### *Tính đúng đắn của thuật toán*

Khi một thuật toán được làm ra, ta cần phải chứng minh rằng, thuật toán khi được thực hiện sẽ cho ta kết quả đúng với mọi dữ liệu vào hợp lệ. Điều này gọi là chứng minh tính đúng đắn của thuật toán. Việc chứng minh một thuật toán đúng đắn là một công việc không dễ dàng.

Trong nhiều trường hợp, nó đòi hỏi ta phải có trình độ và khả năng tư duy toán học tốt.

Sau đây ta sẽ chỉ ra rằng, khi thực hiện thuật toán Euclid, g sẽ là ước chung lớn nhất của hai số nguyên dương m và n bất kỳ. Thật vậy, khi thực hiện bước 1, ta có  $m = qn + r$ , trong đó q là số nguyên nào đó. Nếu  $r = 0$  thì n là ước của m và hiển nhiên n (do đó g) là ước chung lớn nhất của m và n. Nếu  $r \neq 0$ , thì một ước chung bất kỳ của m và n cũng là ước chung của n và r ( $vì r = m - qn$ ). Ngược lại một ước chung bất kỳ của n và r cũng là ước chung của m và n ( $vì m = qn + r$ ). Do đó ước chung lớn nhất của n và r cũng là ước chung lớn nhất của m và n. Vì vậy, khi thực hiện lặp lại bước 1 với sự thay đổi giá trị của m bởi n giá trị của n bởi r (các phép gán  $m \leftarrow n$ ,  $n \leftarrow r$  ở bước 2) cho tới khi  $r = 0$ , ta sẽ nhận được giá trị của g là ước chung lớn nhất của các giá trị m và n ban đầu.

### ***Phân tích thuật toán***

Giả sử đối với một bài toán nào đó chúng ta có một số thuật toán giải. Một câu hỏi mới xuất hiện là, chúng ta cần chọn thuật toán nào trong số các thuật toán đó để áp dụng. Việc phân tích thuật toán, đánh giá độ phức tạp của nó là nội dung của phần sau.

## **1.2. PHÂN TÍCH THUẬT TOÁN**

### **1.2.1. Tính hiệu quả của thuật toán**

Khi giải một vấn đề, chúng ta cần chọn trong số các thuật toán, một thuật toán mà chúng ta cho là “tốt” nhất. Vậy ta cần lựa chọn thuật toán dựa trên cơ sở nào? Thông thường ta dựa trên hai tiêu chuẩn sau đây :

1. Thuật toán đơn giản, dễ hiểu, dễ cài đặt (dễ viết chương trình)
2. Thuật toán sử dụng tiết kiệm nhất các nguồn tài nguyên của máy tính, và đặc biệt, chạy nhanh nhất có thể được.

Khi ta viết một chương trình chỉ để sử dụng một số ít lần, và cái giá của thời gian viết chương trình vượt xa cái giá của chạy chương trình thì tiêu chuẩn (1) là quan trọng nhất. Nhưng có trường hợp ta cần viết các

chương trình (hoặc thủ tục, hàm) để sử dụng nhiều lần, cho nhiều người sử dụng, khi đó giá của thời gian chạy chương trình sẽ vượt xa giá viết nó. Chẳng hạn, các thủ tục sắp xếp, tìm kiếm được sử dụng rất nhiều lần, bởi rất nhiều người trong các bài toán khác nhau. Trong trường hợp này ta cần dựa trên tiêu chuẩn (2). Ta sẽ cài đặt thuật toán có thể rất phức tạp, miễn là chương trình nhận được chạy nhanh hơn các chương trình khác.

Tiêu chuẩn (2) được xem là *tính hiệu quả* của thuật toán. Tính hiệu quả của thuật toán bao gồm hai nhân tố cơ bản.

1. Dung lượng không gian nhớ cần thiết để lưu giữ các dữ liệu vào, các kết quả tính toán trung gian và các kết quả của thuật toán.
2. Thời gian cần thiết để thực hiện thuật toán (ta gọi là thời gian chạy).

Chúng ta sẽ chỉ quan tâm đến thời gian thực hiện thuật toán. Vì vậy, khi nói đến đánh giá độ phức tạp của thuật toán, có nghĩa là ta nói đến đánh giá thời gian thực hiện. Một thuật toán có hiệu quả được xem là thuật toán có thời gian chạy ít hơn các thuật toán khác.

### 1.2.2. Tại sao lại cần thuật toán có hiệu quả?

Kỹ thuật máy tính tiến bộ rất nhanh, ngày nay các máy tính lớn có thể đạt tốc độ tính toán hàng trăm triệu phép tính một giây. Vậy thì có bõ công phải tiêu tốn thời gian để thiết kế các thuật toán hiệu quả không? Một số ví dụ sau đây sẽ trả lời cho câu hỏi này.

**Ví dụ 1.** Tính định thức.

Giả sử  $M$  là một ma trận vuông cấp  $n$ :

$$M = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}$$

Định thức của ma trận  $M$  ký hiệu là  $\det(M)$  được xác định đệ quy như sau

Nếu  $n = 1$ ,  $\det(M) = a_{11}$ . Nếu  $n > 1$ , ta gọi  $M_{ij}$  là ma trận con cấp  $n - 1$ , nhận được từ ma trận  $M$  bằng cách loại bỏ dòng thứ  $i$  và cột thứ  $j$ , và

$$\det(M) = \sum_{j=1}^n (-1)^{1+j} a_{1j} \det(M_{1j})$$

Dễ dàng thấy rằng, nếu ta tính định thức trực tiếp dựa vào công thức đệ qui này, cần thực hiện  $n!$  phép nhân. Một con số khổng lồ với  $n$  không lấy gì làm lớn. Ngay cả với tốc độ của máy tính lớn hiện đại, để tính định thức của ma trận cấp  $n = 25$ , cũng cần hàng triệu năm !

Một thuật toán cổ điển khác, đó là thuật toán Gauss - Jordan, thuật toán này tính định thức cấp  $n$  trong thời gian  $n^3$ .

Để tính định thức cấp  $n = 100$  bằng thuật toán này trên máy tính lớn ta chỉ cần đến 1 giây.

### Ví dụ 2 : Bài toán tháp Hà Nội.

Trong ví dụ 2, mục 1.1 ta đã đưa ra một thuật toán để chuyển  $n$  đĩa từ cọc A sang cọc B. Ta thử tính xem, cần thực hiện bao nhiêu lần chuyển đĩa từ cọc này sang cọc khác (không đặt đĩa to lên trên đĩa nhỏ) để chuyển được  $n$  đĩa từ cọc A sang cọc B. Gọi số đó là  $F(n)$ . Từ thuật toán, ta có :

$$F(1) = 1,$$

$$F(n) = 2F(n-1) + 1 \quad \text{với } n > 1.$$

$$\text{với } n = 1, 2, 3 \text{ ta có } F(1) = 1, F(2) = 3, F(3) = 7.$$

Bằng cách qui nạp, ta chứng minh được  $F(n) = 2^n - 1$ .

Với  $n = 64$ , ta có  $F(64) = 2^{64} - 1$  lần chuyển. Giả sử mỗi lần chuyển một đĩa từ cọc này sang cọc khác, cần 1 giây. Khi đó để thực hiện  $2^{64} - 1$  lần chuyển, ta cần  $5 \times 10^{11}$  năm. Nếu tuổi của vũ trụ là 10 tỉ năm, ta cần 50 lần tuổi của vũ trụ để chuyển 64 đĩa !.

Đối với một vấn đề có thể có nhiều thuật toán, trong số đó có thể thuật toán này hiệu quả hơn (chạy nhanh hơn) thuật toán kia. Tuy nhiên, cũng có những vấn đề không tồn tại thuật toán hiệu quả, tức là có thuật toán, song thời gian thực hiện nó là quá lớn, trong thực tế không thể thực hiện được, dù là trên các máy tính lớn hiện đại nhất.

### 1.2.3. Đánh giá thời gian thực hiện thuật toán như thế nào?

Có hai cách tiếp cận để đánh giá thời gian thực hiện của một thuật toán. Trong phương pháp thử nghiệm, chúng ta viết chương trình và cho chạy chương trình với các dữ liệu vào khác nhau trên một máy tính nào đó. Thời gian chạy chương trình phụ thuộc vào các nhân tố chính sau đây:

1. Các dữ liệu vào.
2. Chương trình dịch để chuyển chương trình nguồn thành mã máy.
3. Tốc độ thực hiện các phép toán của máy tính được sử dụng để chạy chương trình.

Vì thời gian chạy chương trình phụ thuộc vào nhiều nhân tố, nên ta không thể biểu diễn chính xác thời gian chạy là bao nhiêu đơn vị thời gian chuẩn, chẳng hạn nó là bao nhiêu giây.

Trong phương pháp lý thuyết (đó là phương pháp được sử dụng trong sách này), ta sẽ coi thời gian thực hiện thuật toán như là hàm số của *cõ dữ liệu vào*. Cõ của dữ liệu vào là một tham số đặc trưng cho dữ liệu vào, nó có ảnh hưởng quyết định đến thời gian thực hiện chương trình. Cái mà chúng ta chọn làm cõ của dữ liệu vào phụ thuộc vào các thuật toán cụ thể. Đối với các thuật toán sắp xếp mảng, cõ của dữ liệu là số thành phần của mảng. Đối với thuật toán giải hệ n phương trình tuyến tính với n ẩn, ta chọn n là cõ. Thông thường cõ của dữ liệu vào là một số nguyên dương n. Ta sẽ sử dụng hàm số  $T(n)$ , trong đó n là cõ dữ liệu vào, để biểu diễn thời gian thực hiện của một thuật toán.

Thời gian thực hiện thuật toán  $T(n)$  nói chung không chỉ phụ thuộc vào cõ của dữ liệu vào, mà còn phụ thuộc vào dữ liệu vào cá biệt. Chẳng hạn, ta xét bài toán xác định một đối tượng a có mặt trong danh sách n phần tử ( $a_1, a_2, \dots, a_n$ ) hay không. Thuật toán ở đây là, so sánh a với từng phần tử của danh sách đi từ đầu đến cuối danh sách, khi gặp phần tử  $a_i$  đầu tiên  $a_i = a$  thì dừng lại, hoặc đi đến hết danh sách mà không gặp  $a_i$  nào bằng a, trong trường hợp này a không có trong danh sách. Các dữ liệu vào là a và danh sách ( $a_1, a_2, \dots, a_n$ ) (có thể biểu diễn danh sách bằng mảng, chẳng hạn). Cõ của dữ liệu vào là n. Nếu  $a_1 = a$  chỉ cần một phép so sánh. Nếu  $a_1 \neq a$ ,  $a_2 = a$ , cần 2 phép so sánh. Còn nếu  $a_i \neq a$ ,  $i = 1, \dots, n-1$  và  $a_n = a$ , hoặc a không có trong danh sách, ta cần

n phép so sánh. Nếu xem thời gian thực hiện  $T(n)$  là số phép toán so sánh, ta có  $T(n) \leq n$ , trong trường hợp xấu nhất  $T(n) = n$ . Trong các trường hợp như thế, ta nói đến thời gian thực hiện thuật toán trong trường hợp xấu nhất.

Ngoài ra, ta còn sử dụng khái niệm thời gian thực hiện trung bình. Đó là thời gian trung bình  $T_{tb}(n)$  trên tất cả các dữ liệu vào có cỡ  $n$ . Nói chung thời gian thực hiện trung bình khó xác định hơn thời gian thực hiện trong trường hợp xấu nhất.

Chúng ta có thể xác định thời gian thực hiện  $T(n)$  là số *phép toán sơ cấp* cần phải tiến hành khi thực hiện thuật toán. Các phép toán sơ cấp là các phép toán mà thời gian thực hiện bị chặn trên bởi một hằng số chỉ phụ thuộc vào cách cài đặt được sử dụng (ngôn ngữ lập trình, máy tính ...). Chẳng hạn các phép toán số học  $+$ ,  $-$ ,  $*$ ,  $/$ , các phép toán so sánh  $=$ ,  $<$ ,  $<=$ ,  $>$ ,  $>=$  là các phép toán sơ cấp. Phép toán so sánh hai xâu ký tự không thể xem là phép toán sơ cấp, vì thời gian thực hiện nó phụ thuộc vào độ dài của xâu.

#### 1.2.4. Ký hiệu ô lớn và đánh giá thời gian thực hiện thuật toán bằng ký hiệu ô lớn

Khi đánh giá thời gian thực hiện bằng phương pháp toán học, chúng ta sẽ bỏ qua nhân tố phụ thuộc vào cách cài đặt, chỉ tập trung vào xác định độ lớn của thời gian thực hiện  $T(n)$ . Ký hiệu toán học ô lớn được sử dụng để mô tả độ lớn của hàm  $T(n)$ .

Giả sử  $n$  là số nguyên không âm,  $T(n)$  và  $f(n)$  là các hàm thực không âm. Ta viết  $T(n) = O(f(n))$  (đọc :  $T(n)$  là ô lớn của  $f(n)$ ), nếu và chỉ nếu tồn tại các hằng số dương  $c$  và  $n_0$  sao cho  $T(n) \leq c f(n)$ , với mọi  $n \geq n_0$ .

Nếu một thuật toán có thời gian thực hiện  $T(n) = O(f(n))$ , chúng ta sẽ nói rằng thuật toán có thời gian thực hiện cấp  $f(n)$ . Từ định nghĩa ký hiệu ô lớn, ta có thể xem rằng hàm  $f(n)$  là cận trên của  $T(n)$ .

**Ví dụ.** Giả sử  $T(n) = 3n^2 + 5n + 4$ . Ta có

$$3n^2 + 5n + 4 \leq 3n^2 + 5n^2 + 4n^2 = 12n^2, \text{ với mọi } n \geq 1.$$

Vậy  $T(n) = O(n^2)$ . Trong trường hợp này ta nói thuật toán có thời gian thực hiện cấp  $n^2$ , hoặc gọn hơn, thuật toán có thời gian thực hiện bình phương.

Dễ dàng thấy rằng, nếu  $T(n) = O(f(n))$  và  $f(n) = O(f_1(n))$ , thì  $T(n) = O(f_1(n))$ . Thật vậy, vì  $T(n)$  là ô lớn của  $f(n)$  và  $f(n)$  là ô lớn của  $f_1(n)$ , do đó tồn tại các hằng số  $c_0, n_0, c_1, n_1$  sao cho  $T(n) \leq c_0 f(n)$  với mọi  $n \geq n_0$  và  $f(n) \leq c_1 f_1(n)$  với mọi  $n \geq n_1$ . Từ đó ta có  $T(n) \leq c_0 c_1 f_1(n)$  với mọi  $n \geq \max(n_0, n_1)$ .

Khi biểu diễn cấp của thời gian thực hiện thuật toán bởi hàm  $f(n)$ , chúng ta sẽ chọn  $f(n)$  là hàm số nhỏ nhất, đơn giản nhất có thể được sao cho  $T(n) = O(f(n))$ . Thông thường  $f(n)$  là các hàm số sau đây:  $f(n) = 1$ ;  $f(n) = \log n$ ;  $f(n) = n$ ;  $f(n) = n \log n$ ;  $f(n) = n^2, n^3, \dots$  và  $f(n) = 2^n$ .

Nếu  $T(n) = O(1)$  điều này có nghĩa là thời gian thực hiện bị chặn trên bởi một hằng số nào đó, trong trường hợp này ta nói thuật toán có thời gian thực hiện hằng.

Nếu  $T(n) = O(n)$ , tức là bắt đầu từ một  $n_0$  nào đó trở đi ta có  $T(n) \leq cn$  với một hằng số  $c$  nào đó, thì ta nói thuật toán có thời gian thực hiện tuyến tính.

Bảng sau đây cho ta các cấp thời gian thực hiện thuật toán được sử dụng rộng rãi nhất và tên gọi thông thường của chúng.

Ký hiệu ô lớn	Tên gọi thông thường
$O(1)$	hằng
$O(\log n)$	logarit
$O(n)$	tuyến tính
$O(n \log n)$	$n \log n$
$O(n^2)$	bình phương
$O(n^3)$	lập phương
$O(2^n)$	mũ

Danh sách trên sắp xếp theo thứ tự tăng dần của cấp thời gian thực hiện.

Để thấy rõ sự khác nhau của các cấp thời gian thực hiện thuật toán, ta xét ví dụ sau. Giả sử đối với một vấn đề nào đó, ta có hai thuật toán

giải A và B. Thuật toán A có thời gian thực hiện  $T_A(n) = O(n^2)$ , còn thuật toán B có thời gian thực hiện  $T_B(n) = O(n \log n)$ . Với  $n = 1024$ , thuật toán A đòi hỏi khoảng 1048.576 phép toán sơ cấp, còn thuật toán B đòi hỏi khoảng 10.240 phép toán sơ cấp. Nếu cần một micrô-giây cho một phép toán sơ cấp thì thuật toán A cần khoảng 1,05 giây, trong khi thuật toán B chỉ cần khoảng 0,01 giây. Nếu  $n = 1024 \times 2$ , thì thuật toán A đòi hỏi khoảng 4,2 giây, trong khi thuật toán B chỉ đòi hỏi khoảng 0,02 giây. Với  $n$  càng lớn thì thời gian thực hiện thuật toán B càng ít hơn so với thời gian thực hiện thuật toán A. Vì vậy, nếu một vấn đề nào đó đã có một thuật toán giải với thời gian thực hiện cấp  $n^2$ , bạn tìm ra thuật toán mới với thời gian thực hiện cấp  $n \log n$ , thì đó là một kết quả rất có ý nghĩa.

Những thuật toán có thời gian thực hiện cấp  $n^k$ , với  $k$  là số nguyên nào đó  $\geq 1$ , được gọi là các thuật toán có thời gian thực hiện đa thức.

### 1.2.5. Các qui tắc để đánh giá thời gian thực hiện thuật toán

Sau đây chúng ta đưa ra một qui tắc cần thiết về ô lớn để đánh giá thời gian thực hiện một thuật toán.

Qui tắc tổng : Nếu  $T_1(n) = O(f_1(n))$  và  $T_2(n) = O(f_2(n))$  thì

$$T_1(n) + T_2(n) = O(\max(f_1(n), f_2(n))).$$

Thật vậy, vì  $T_1(n)$ ,  $T_2(n)$  là ô lớn của  $f_1(n)$ ,  $f_2(n)$  tương ứng do đó tồn tại hằng số  $c_1$ ,  $c_2$ ,  $n_1$ ,  $n_2$  sao cho  $T_1(n) \leq c_1 f_1(n)$  với mọi  $n \geq n_1$  và  $T_2(n) \leq c_2 f_2(n)$  với mọi  $n \geq n_2$ . Đặt  $n_o = \max(n_1, n_2)$ . Khi đó với mọi  $n \geq n_o$ , ta có  $T_1(n) + T_2(n) \leq (c_1 + c_2) \max(f_1(n), f_2(n))$ .

Qui tắc này thường được áp dụng như sau. Giả sử thuật toán của ta được phân thành ba phần tuần tự. Phần một có thời gian thực hiện  $T_1(n)$  được đánh giá là  $O(1)$ , phần hai có thời gian  $T_2(n)$  là  $O(n^2)$ , phần ba có thời gian  $T_3(n)$  là  $O(n)$ . Khi đó thời gian thực hiện thuật toán  $T(n) = T_1(n) + T_2(n) + T_3(n)$  sẽ là  $O(n^2)$ , vì  $n^2 = \max(1, n^2, n)$ .

Trong sách báo quốc tế các thuật toán thường được trình bày dưới dạng các thủ tục hoặc hàm trong ngôn ngữ tựa Pascal. Để đánh giá thời gian thực hiện thuật toán, ta cần biết cách đánh giá thời gian thực hiện

các câu lệnh của Pascal. Trước hết, chúng ta hãy xác định các câu lệnh trong Pascal. Các câu lệnh trong Pascal được định nghĩa đê qui như sau:

1. Các phép gán, đọc, viết, **goto** là câu lệnh. Các lệnh này được gọi là các lệnh đơn.

2. Nếu  $S_1, S_2, \dots, S_n$  là câu lệnh thì

**begin**  $S_1, S_2, \dots, S_n$  **end**

là câu lệnh.h. Lệnh này được gọi là lệnh hợp thành (hoặc khối).

3. Nếu  $S_1$  và  $S_2$  là các câu lệnh và E là biểu thức logic thì

**if** E **then**  $S_1$  **else**  $S_2$

là câu lệnh, và

**if** E **then**  $S_1$

là câu lệnh.i. Các lệnh này được gọi là lệnh if.

4. Nếu  $S_1, S_2, \dots, S_{n+1}$  là các câu lệnh, E là biểu thức có kiểu thứ tự đếm được, và  $v_1, v_2, \dots, v_n$  là các giá trị cùng kiểu với E thì

**case** E **of**

$v_1 : S_1 ;$

$v_2 : S_2 ;$

.....

$v_n : S_n ;$

**[else]       $S_{n+1}$**

**end**

**THƯ VIỆN ĐH. DÂN LẬP HN  
PHÒNG ĐỌC**

**01.ĐVL.972**

là câu lệnh. Lệnh này được gọi là lệnh case

5. Nếu S là câu lệnh và E là biểu thức logic thì

**while** E **do** S

là câu lệnh. Lệnh này được gọi là lệnh while.

6. Nếu  $S_1, S_2, \dots, S_n$  là các câu lệnh, và E là biểu thức logic thì

**repeat**  $S_1, S_2, \dots, S_n$  **until** E

là câu lệnh. Lệnh này được gọi là lệnh repeat.

7. Với S là câu lệnh, E<sub>1</sub> và E<sub>2</sub> là các biểu cùng một kiểu thứ tự đếm được, thì

**for i := E<sub>1</sub> to E<sub>2</sub> do** S

là câu lệnh, và

**for i := E<sub>2</sub> downto E<sub>1</sub> do S**

là câu lệnh. Các câu lệnh này được gọi là **lệnh for**.

Nhờ định nghĩa đệ qui của các lệnh, chúng ta có thể phân tích một chương trình xuất phát từ các lệnh đơn, rồi từng bước đánh giá các lệnh phức tạp hơn, cuối cùng đánh giá được thời gian thực hiện chương trình.

Giả sử rằng, các lệnh gán không chứa các lời gọi hàm. Khi đó để đánh giá thời gian thực hiện một chương trình, ta có thể áp dụng phương pháp đệ qui sau đây :

1. Thời gian thực hiện các lệnh đơn : gán, đọc, viết, **goto** là O(1).
2. Lệnh hợp thành. Thời gian thực hiện lệnh hợp thành được xác định bởi luật tổng.
3. Lệnh **if**. Giả sử thời gian thực hiện các lệnh S<sub>1</sub>, S<sub>2</sub> là O(f<sub>1</sub>(n)) và O(f<sub>2</sub>(n)) tương ứng. Khi đó thời gian thực hiện lệnh **if** là O(max(f<sub>1</sub>(n), f<sub>2</sub>(n))).
4. Lệnh **case**. Được đánh giá như lệnh **if**.
5. Lệnh **while**. Giả sử thời gian thực hiện lệnh S (thân của lệnh **while**) là O(f(n)). Giả sử g(n) là số tối đa các lần thực hiện lệnh S, khi thực hiện lệnh **while**. Khi đó thời gian thực hiện lệnh **while** là O(f(n)g(n)).
6. Lệnh **repeat**. Giả sử thời gian thực hiện khối **begin** S<sub>1</sub>, S<sub>2</sub>, ... S<sub>n</sub> **end** là O(f(n)). Giả sử g(n) là số tối đa các lần lặp. Khi đó thời gian thực hiện lệnh **repeat** là O(f(n)g(n)).
7. Lệnh **for**. Được đánh giá tương tự lệnh **while** và **repeat**.

Nếu lệnh gán có chứa các lời gọi hàm, thì thời gian thực hiện nó không thể xem là O(1) được, vì khi đó thời gian thực hiện lệnh gán còn phụ thuộc vào thời gian thực hiện các hàm có trong lệnh gán. Việc đánh giá thời gian thực hiện các thủ tục (hoặc hàm) không đệ qui được tiến hành bằng cách áp dụng các qui tắc trên. Việc đánh giá thời gian thực hiện các thủ tục (hoặc hàm) đệ quy sẽ khó khăn hơn nhiều.

**Đánh giá thủ tục (hoặc hàm) đệ qui**

Trước hết chúng ta xét một ví dụ cụ thể. Ta sẽ đánh giá thời gian thực hiện của hàm đệ qui sau (hàm này tính n!).

```
function Fact(n : integer) : integer;
begin
    if n <=1 then fact :=1
    else fact := n * fact(n-1);
end;
```

Trong hàm này cõi của dữ liệu vào là n. Giả sử thời gian thực hiện hàm là  $T(n)$ . Với  $n=1$ , chỉ cần thực hiện lệnh gán  $fact := 1$ , do đó  $T(1) = O(1)$ . Với  $n > 1$ , cần thực hiện lệnh gán  $fact := n * fact(n-1)$ . Do đó, thời gian  $T(n)$  là  $O(1)$  (để thực hiện phép nhân và phép gán) cộng với  $T(n-1)$  (để thực hiện lời gọi đệ qui  $fact(n-1)$ ). Tóm lại, ta có quan hệ đệ qui sau :

$$T(1) = O(1);$$

$$T(n) = O(1) + T(n-1).$$

Thay các  $O(1)$  bởi các hằng nào đó, ta nhận được quan hệ đệ qui sau

$$T(1) = C_1$$

$$T(n) = C_2 + T(n-1)$$

Để giải phương trình đệ qui, tìm  $T(n)$ , chúng ta áp dụng phương pháp thế lặp. Ta có phương trình đệ qui

$$T(m) = C_2 + T(m-1), \text{ với } m > 1$$

Thay m lần lượt bởi 2,3,...,  $n - 1$ ,  $n$ , ta nhận được các quan hệ sau.

$$T(2) = C_2 + T(1)$$

$$T(3) = C_2 + T(2)$$

.....

$$T(n-1) = C_2 + T(n-2)$$

$$T(n) = C_2 + T(n-1)$$

Bằng các phép thế liên tiếp, ta nhận được

$$T(n) = (n-1)C_2 + T(1)$$

hay  $T(n) = (n-1) C_2 + C_1$ , trong đó  $C_1$  và  $C_2$  là các hằng nào đó. Do đó,  $T(n) = O(n)$ .

Từ ví dụ trên, ta suy ra phương pháp tổng quát sau đây để đánh giá thời gian thực hiện thủ tục (hàm) đệ qui. Để đơn giản, ta giả thiết rằng các thủ tục (hàm) là đệ qui trực tiếp. Điều đó có nghĩa là các thủ tục (hàm) chỉ chứa các lời gọi đệ qui đến chính nó (không qua một thủ tục (hàm) khác nào cả). Giả sử thời gian thực hiện thủ tục là  $T(n)$ , với  $n$  là cõ dữ liệu vào. Khi đó thời gian thực hiện các lời gọi đệ qui thủ tục sẽ là  $T(m)$ , với  $m < n$ . Đánh giá thời gian  $T(n_0)$ , với  $n_0$  là cõ dữ liệu vào nhỏ nhất có thể được (trong ví dụ trên, đó là  $T(1)$ ). Sau đó đánh giá thân của thủ tục theo các qui tắc 1-7, ta sẽ nhận được quan hệ đệ qui sau đây.

$$T(n) = F(T(m_1), T(m_2), \dots, T(m_k))$$

trong đó  $m_1, m_2, \dots, m_k < n$ . Giải phương trình đệ qui này, ta sẽ nhận được sự đánh giá của  $T(n)$ . Tuy nhiên, cần biết rằng, việc giải phương trình đệ qui, trong nhiều trường hợp, là rất khó khăn, không đơn giản như trong ví dụ đã trình bày.

### 1.2.6. Phân tích một số thuật toán

Sau đây chúng ta sẽ áp dụng các phương pháp đã trình bày để phân tích độ phức tạp của một số thuật toán.

**Ví dụ 1:** Phân tích thuật toán Euclid. Chúng ta biểu diễn thuật toán Euclid bởi hạm sau.

```

function Euclid (m, n : integer) : integer ;
  var r : integer ;
  begin
    (1)   r := m mod n ;
    (2)   while r < > 0 do
      begin
        (3)           m := n ;
        (4)           n := r ;
        (5)           r := m mod n ;
      end ;
    (6)   Euclid := n ;
  end ;

```

Thời gian thực hiện thuật toán phụ thuộc vào số nhỏ nhất trong hai số  $m$  và  $n$ . Giả sử  $m \geq n > 0$ , do đó cỡ của dữ liệu vào là  $n$ . Các lệnh (1) và (6) có thời gian thực hiện là  $O(1)$ . Vì vậy thời gian thực hiện thuật toán là thời gian thực hiện lệnh while ta đánh giá thời gian thực hiện lệnh while (2). Thân của lệnh này, là khối gồm ba lệnh (3), (4) và (5). Mỗi lệnh có thời gian thực hiện là  $O(1)$ , do đó khối có thời gian thực hiện là  $O(1)$ . Còn phải đánh giá số lớn nhất các lần thực hiện lặp khối.

Ta có

$$m = n \cdot q_1 + r_1, \quad 0 \leq r_1 < n$$

$$n = r_1 \cdot q_2 + r_2, \quad 0 \leq r_2 < r_1$$

Nếu  $r_1 \leq n/2$  thì  $r_2 < r_1 \leq n/2$ , do đó  $r_2 < n/2$ . Nếu  $r_1 > n/2$  thì  $q_2 = 1$ , tức là  $n = r_1 + r_2$ , do đó  $r_2 < n/2$ . Tóm lại, ta luôn có  $r_2 < n/2$ .

Như vậy, cứ hai lần thực hiện khối thì phần dư  $r$  giảm đi một nửa của  $n$ . Gọi  $k$  là số nguyên lớn nhất sao cho  $2^k \leq n$ . Số lần lặp khối tối đa là  $2k+1 \leq 2\log_2 n + 1$ . Do đó thời gian thực hiện lệnh **while** là  $O(\log_2 n)$ . Đó cũng là thời gian thực hiện thuật toán.

**Ví dụ 2.** Dãy số Fibonacci được xác định một cách đệ quy như sau

$$f_0 = 0;$$

$$f_1 = 1;$$

$$f_n = f_{n-1} + f_{n-2} \text{ với } n \geq 2$$

Các thành phần đầu tiên của dãy là 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Dãy này có nhiều ứng dụng trong toán học, tin học và lý thuyết trò chơi.

Thuật toán đệ quy

```
function Fibo1(n : integer) : integer;
begin
  if n < 2 then Fibo1 := n
  else Fibo1 := Fibo1(n-1) + Fibo1(n-2);
end;
```

Bây giờ ta đánh giá thời gian thực hiện hàm Fibo1. Gọi  $T(n)$  là thời gian đó. Từ hàm Fibo1, ta có quan hệ đệ quy sau

$$T(n) = c \text{ với } n < 2 \quad (c \text{ là hằng số})$$

$$T(n) = T(n-1) + T(n-2) \text{ với } n \geq 2$$

Quan hệ đệ quy này rất đơn giản, nhưng giải nó cũng không dễ dàng. Bằng cách giải phương trình đặc trưng của nó (chúng ta không trình bày phương pháp này ở đây), chúng ta có thể tìm ra được thời gian thực hiện  $T(n)$  của hàm Fibolà  $O(\phi^n)$ , với  $\phi=(1+\sqrt{5})/2$ . Tức là, thuật toán Fibol có thời gian thực hiện mũ, nó không có ý nghĩa thực tiễn với  $n$  lớn. Sau đây là một thuật toán khác, có thời gian thực hiện chỉ là  $O(n)$ .

```

funciton Fibo2 (n: integer) : integer;
var i, j, k : integer;
begin
    (1) i := 1 ;
    (2) j := 0 ;
    (3) for k := 1 to n do
        begin
            j := i + j ;
            i := j - i ;
        end ;
    (4) Fibo2 := j ;
end ;

```

Ta phân tích hàm Fibo2. Các lệnh gán (1), (2) và (4) có thời gian thực hiện  $O(1)$ . Thân của lệnh for(3) có thời gian thực hiện là  $O(1)$ , số lần lặp là  $n$ . Do đó lệnh for(3) có thời gian thực hiện là  $O(n)$ . Kết hợp lại, ta có thời gian thực hiện hàm Fibo2 là  $O(n)$ .

Với  $n = 50$ , thuật toán Fibol cần khoảng 20 ngày trên máy tính lớn, trong khi đó thuật toán Fibo2 chỉ cần khoảng 1 micrô giây. Với  $n = 100$ , thời gian chạy của thuật toán Fibol là  $10^9$  năm! còn thuật toán Fibo2 chỉ cần khoảng 1.5 micro giây. Thuật toán Fibo2 chưa phải là thuật toán hiệu quả nhất. Bạn thử tìm một thuật toán hiệu quả hơn.

## **Chương 2**

# **KIỂU DỮ LIỆU, CẤU TRÚC DỮ LIỆU VÀ MÔ HÌNH DỮ LIỆU**

### **2.1. BIỂU ĐIỂN DỮ LIỆU**

Trong máy tính điện tử (MTĐT), các dữ liệu dù có bản chất khác nhau như thế nào (số nguyên, số thực, hay xâu ký tự, ...), đều được biểu diễn dưới dạng nhị phân. Mỗi dữ liệu được biểu diễn dưới dạng một dãy các số nhị phân 0 hoặc 1. Về mặt kỹ thuật đây là cách biểu diễn thích hợp nhất, vì các giá trị 0 và 1 dễ dàng được mã hoá bởi các phần tử vật lý chỉ có hai trạng thái. Chúng ta sẽ không quan tâm đến cách biểu diễn này của dữ liệu, cũng như các cách tiến hành các thao tác, các phép toán trên các dữ liệu được biểu diễn dưới dạng nhị phân.

Cách biểu diễn nhị phân của dữ liệu rất không thuận tiện đối với con người. Việc xuất hiện các ngôn ngữ lập trình bậc cao (FORTRAN, BASIC, PASCAL, C ...) đã giải phóng con người khỏi những khó khăn khi làm việc với cách biểu diễn trong máy của dữ liệu. Trong các ngôn ngữ lập trình bậc cao, các dữ liệu, hiểu theo một nghĩa nào đó, là sự trùu tượng hoá các tính chất của các đối tượng trong thế giới hiện thực. Nói dữ liệu là sự trùu tượng hoá từ thế giới hiện thực, vì ta đã bỏ qua những nhân tố, tính chất mà ta cho là không cơ bản, chỉ giữ lại những tính chất đặc trưng cho các đối tượng thuộc phạm vi bài toán đang xét. Chẳng hạn, vị trí của một đối tượng trong thực tiễn, được đặc trưng bởi cặp số thực ( $x, y$ ) (đó là tọa độa đê-các của một điểm trong mặt phẳng). Do đó, trong ngôn ngữ Pascal, vị trí một đối tượng được biểu diễn bởi bản ghi gồm hai trường tương ứng với hoành độ và tung độ của một điểm. Trong toán học có các khái niệm biểu diễn đặc trưng về mặt số

lượng của các đối tượng trong thế giới hiện thực, đó là các khái niệm số nguyên, số thực, số phức, ... Trên cơ sở các khái niệm toán học này, người ta đã đưa vào trong các ngôn ngữ lập trình bậc cao các dữ liệu kiểu nguyên, thực, phức, ... Tuy nhiên do tính đa dạng của các bài toán cần xử lý bằng MTĐT, chỉ sử dụng các kiểu dữ liệu có sẵn trong các ngôn ngữ lập trình bậc cao là chưa đủ để giải quyết các bài toán. Chúng ta phải cần đến các *cấu trúc dữ liệu*. Đó là các dữ liệu phức tạp, được xây dựng nên từ các dữ liệu đã có, đơn giản hơn bằng các phương pháp liên kết nào đó.

Để giải quyết một bài toán bằng MTĐT, ta cần xây dựng các *mô hình dữ liệu* mô tả bài toán. Đó là sự triết lượng hóa các tính chất của các đối tượng thuộc phạm vi vấn đề mà ta quan tâm và các mối quan hệ giữa các đối tượng đó. Dùng làm các mô hình dữ liệu trong tin học, chúng ta sẽ sử dụng các mô hình toán học như danh sách, cây, tập hợp, ánh xạ, quan hệ, đồ thị, ... Mô hình dữ liệu sẽ được biểu diễn bởi các cấu trúc dữ liệu. Thông thường một mô hình dữ liệu có thể được biểu hiện bởi nhiều cấu trúc dữ liệu khác nhau. Tuỳ từng ứng dụng, ta sẽ chọn cấu trúc dữ liệu nào mà các thao tác cần thực hiện là hiệu quả nhất có thể được.

## 2.2. KIỂU DỮ LIỆU VÀ CẤU TRÚC DỮ LIỆU

Trong các ngôn ngữ lập trình bậc cao, các dữ liệu được phân lớp thành các lớp dữ liệu dựa vào bản chất của dữ liệu. Mỗi một lớp dữ liệu được gọi là một *kiểu dữ liệu*. Như vậy, một kiểu T là một tập hợp nào đó, các phần tử của tập được gọi là các *giá trị* của kiểu. Chẳng hạn, kiểu *integer* là tập hợp các số nguyên, kiểu *char* là một tập hữu hạn các ký hiệu. Các ngôn ngữ lập trình khác nhau có thể có các kiểu dữ liệu khác nhau. Fortran có các kiểu dữ liệu là *integer*, *real*, *logical*, *complex* và *double complex*. Các kiểu dữ liệu trong ngôn ngữ C là *int*, *float*, *char*, *con trỏ*, *struct*, ... Một cách tổng quát, mỗi ngôn ngữ lập trình có một *hệ kiểu* của riêng mình. Hệ kiểu của một ngôn ngữ bao gồm các kiểu dữ liệu cơ sở và các phương pháp cho phép ta từ các kiểu dữ liệu đã có xây dựng nên các kiểu dữ liệu mới.

Khi nói đến một kiểu dữ liệu, chúng ta cần phải đề cập đến hai đặc trưng sau đây:

1. Tập hợp các giá trị thuộc kiểu. Chẳng hạn, kiểu *integer* trong ngôn ngữ Pascal gồm tất cả các số nguyên được biểu diễn bởi hai byte, tức là gồm các số nguyên từ -32768 đến + 32767. Trong các ngôn ngữ lập trình bậc cao mỗi hằng, biến, biểu thức hoặc hàm cần phải được gắn với một kiểu dữ liệu xác định. Khi đó, mỗi biến (biểu thức, hàm) chỉ có thể nhận các giá trị thuộc kiểu của biến (biểu thức, hàm) đó.

Ví dụ, nếu X là biến có kiểu *boolean* trong Pascal (*var X : boolean*) thì X chỉ có thể nhận một trong hai giá trị *true*, *false*.

2. Với mỗi kiểu dữ liệu, cần phải xác định một tập hợp nào đó các phép toán có thể thực hiện được trên các dữ liệu của kiểu. Chẳng hạn, với kiểu *real*, các phép toán có thể thực hiện được là các phép toán số học thông thường +, -, \*, /, và các phép toán so sánh =, < >, < , < = , > , > =.

Thông thường trong một hệ kiểu của một ngôn ngữ lập trình sẽ có một số kiểu dữ liệu được gọi là *kiểu dữ liệu đơn* hay *kiểu dữ liệu phân tử* (atomic).

Chẳng hạn, trong ngôn ngữ Pascal, các kiểu dữ liệu *integer*, *real*, *boolean*, *char* và các kiểu liệt kê được gọi là các kiểu dữ liệu đơn. Sở dĩ gọi là đơn, vì các giá trị của các kiểu này được xem là các đơn thể đơn giản nhất không thể phân tích thành các thành phần đơn giản hơn được nữa.

Như đã nói, khi giải quyết các bài toán phức tạp, chỉ sử dụng các dữ liệu đơn là không đủ, ta phải cần đến các *cấu trúc dữ liệu*. Một cấu trúc dữ liệu bao gồm một tập hợp nào đó các *dữ liệu thành phần*, các dữ liệu thành phần này được liên kết với nhau bởi một phương pháp nào đó. Các dữ liệu thành phần có thể là dữ liệu đơn, hoặc cũng có thể là một cấu trúc dữ liệu đã được xây dựng. Có thể hình dung một cấu trúc dữ liệu được tạo nên từ các *tế bào* (khối xây dựng), mỗi tế bào có thể xem như một cái hộp chứa dữ liệu thành phần.

Trong Pascal và trong nhiều ngôn ngữ thông dụng khác có một cách đơn giản nhất để liên kết các tế bào, đó là sắp xếp các tế bào chứa các dữ liệu cùng một kiểu thành một dãy. Khi đó ta có một cấu trúc dữ liệu được gọi là *mảng* (array). Như vậy, có thể nói, một mảng là một cấu trúc

dữ liệu gồm một dãy xác định các dữ liệu thành phần cùng một kiểu. Ta vẫn thường nói đến mảng các số nguyên, mảng các số thực, mảng các bản ghi, ... Mỗi một dữ liệu thành phần của mảng được gắn với một chỉ số từ một tập chỉ số nào đó. Ta có thể truy cập đến một thành phần nào đó của mảng bằng cách chỉ ra tên mảng và chỉ số của thành phần đó.

Một phương pháp khác để tạo nên các cấu trúc dữ liệu mới, là kết hợp một số tế bào (có thể chứa các dữ liệu có kiểu khác nhau) thành một *bản ghi* (record). Các tế bào thành phần của bản ghi được gọi là các *trường* của bản ghi. Các bản ghi đến lượt lại được sử dụng làm các tế bào để tạo nên các cấu trúc dữ liệu khác. Chẳng hạn, một trong các cấu trúc dữ liệu hay được sử dụng nhất là mảng các bản ghi.

Còn một phương pháp quan trọng nữa để kiến tạo các cấu trúc dữ liệu, đó là sử dụng con trỏ. Trong phương pháp này, mỗi tế bào là một bản ghi gồm hai phần INFOR và LINK, phần INFOR có thể có một hay nhiều trường dữ liệu, còn phần LINK có thể chứa một hay nhiều con trỏ trỏ đến các tế bào khác có quan hệ với tế bào đó. Chẳng hạn, ta có thể cài đặt một danh sách bởi cấu trúc dữ liệu danh sách liên kết, trong đó mỗi thành phần của danh sách liên kết là bản ghi gồm hai trường

```
type Cell = record
    element : Item ;
    next : ^Cell ;
end ;
```

Ở đây, trường *element* có kiểu dữ liệu *Item*, một kiểu dữ liệu nào đó của các phần tử của danh sách. Trường *next* là con trỏ trỏ tới phần tử tiếp theo trong danh sách. Cấu trúc dữ liệu danh sách liên kết biểu diễn danh sách ( $a_1, a_2, \dots, a_n$ ) có thể được biểu diễn như trong hình 2.1.



Hình 2.1. Cấu trúc dữ liệu danh sách liên kết.

Sử dụng con trỏ để liên kết các tế bào là một trong các phương pháp kiến tạo các cấu trúc dữ liệu được áp dụng nhiều nhất. Ngoài danh sách

liên kết, người ta còn dùng các con trỏ để tạo ra các cấu trúc dữ liệu biểu diễn cây, một mô hình dữ liệu quan trọng bậc nhất.

Trên đây chúng ta đã nêu ba phương pháp chính để kiến tạo các cấu trúc dữ liệu. (Ở đây chúng ta chỉ nói đến các cấu trúc dữ liệu trong bộ nhớ trong, các cấu trúc dữ liệu ở bộ nhớ ngoài như file chỉ số, B-cây sẽ được đề cập riêng.)

Một kiểu dữ liệu mà các giá trị thuộc kiểu không phải là các dữ liệu đơn mà là các cấu trúc dữ liệu được gọi là *kiểu dữ liệu có cấu trúc*. Trong ngôn ngữ Pascal, các kiểu dữ liệu mảng, bản ghi, tập hợp, file đều là các kiểu dữ liệu có cấu trúc.

### 2.3. HỆ KIỂU CỦA NGÔN NGỮ PASCAL

Pascal là một trong các ngôn ngữ có hệ kiểu phong phú nhất. Hệ kiểu của Pascal chứa các kiểu cơ sở *integer*, *real*, *boolean*, *char* và các quy tắc, dựa vào đó ta có thể xây dựng nên các kiểu phức tạp hơn từ các kiểu đã có. Từ các kiểu cơ sở và áp dụng các quy tắc, ta có thể xây dựng nên một tập hợp vô hạn các kiểu. Hệ kiểu của Pascal có thể được định nghĩa đê quy như sau:

#### A. Các kiểu cơ sở

1. *Kiểu integer*
2. *Kiểu real*
3. *Kiểu boolean*
4. *Kiểu char*
5. *Kiểu liệt kê*

Giả sử  $\text{obj}_1, \text{obj}_2, \dots, \text{obj}_n$  là các đối tượng nào đó. khi đó ta có thể tạo nên kiểu liệt kê T bằng cách liệt kê ra tất cả các đối tượng đó.

**type T = (obj<sub>1</sub>, obj<sub>2</sub>, ..., obj<sub>n</sub>)**

**Chú ý:** Tất cả các kiểu đơn đều là các *kiểu có thứ tự*, tức là với hai giá trị bất kỳ a và b thuộc cùng một kiểu, ta luôn có  $a \leq b$  hoặc  $a \geq b$ . trừ kiểu *real*, các kiểu còn lại đều là kiểu có thứ tự đếm được.

### 6. Kiểu đoạn con

```
type T = min .. max
```

Trong đó min và max là cận dưới và cận trên của khoảng ; min và max là các giá trị thuộc cùng một kiểu *integer*, *char*, hoặc các kiểu liệt kê, đồng thời  $\text{min} < \text{max}$ . Kiểu T gồm tất cả các giá trị từ min đến max.

### B. Các quy tắc đê quy

Trong Pascal, chúng ta có thể tạo ra các kiểu mới bằng cách sử dụng các quy tắc sau.

### 7. Kiểu array (mảng)

Giả sử  $T_0$  là một kiểu đã cho, ta sẽ gọi  $T_0$  là **kiểu thành phần** mảng. Giả sử I là kiểu đoạn con hoặc kiểu liệt kê, ta sẽ gọi I là **kiểu chỉ số** mảng. Khi đó ta có thể tạo nên kiểu mảng T với các thành phần của mảng là các giá trị thuộc  $T_0$  và được chỉ số hoá bởi tập hữu hạn, có thứ tự I.

```
type T = array [I] of T0
```

### 8. Kiểu record (bản ghi)

Giả sử  $T_1, T_2, \dots, T_n$  là các kiểu đã cho, và  $F_1, F_2, \dots, F_n$  là các tên (tên trường). Khi đó ta có thể thành lập một kiểu bản ghi T với n trường, trường thứ i có tên là  $F_i$  và các giá trị của nó có kiểu  $T_i$  với  $i = 1, 2, \dots, n$ .

```
type T = record
```

```
  F1 : T1;
```

```
  F2 : T2;
```

```
  ....
```

```
  Fn : Tn;
```

```
end;
```

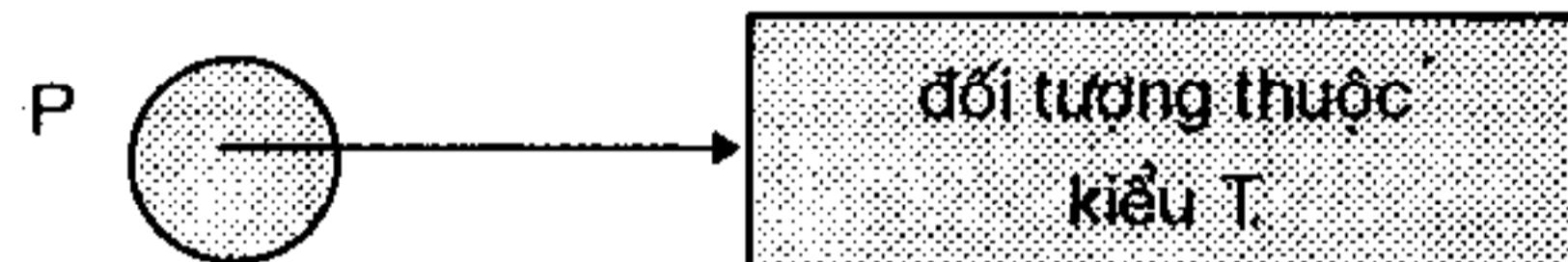
Mỗi giá trị của kiểu bản ghi T là một bộ n giá trị ( $t_1, t_2, \dots, t_n$ ), trong đó  $t_i$  thuộc  $T_i$  với  $i = 1, 2, \dots, n$ .

### 9. Kiểu con trỏ

Giả sử  $T$  là một kiểu đã cho. Khi đó ta có thể tạo nên kiểu con trỏ  $T_p$ .

type  $T_p = ^\wedge T$

Các giá trị của  $T_p$  là địa chỉ (vị trí) trong bộ nhớ của máy tính để lưu giữ các đối tượng thuộc kiểu  $T$ . Chúng ta sẽ biểu diễn một con trỏ  $p$  (var  $p: ^\wedge T$ ) bởi vòng tròn nhỏ có mũi tên chỉ đến đối tượng thuộc kiểu  $T$  (hình 2.2)



Hình 2. 2. Biểu diễn con trỏ

### 10. Kiểu set (tập hợp)

Giả sử  $T_0$  là một kiểu đã cho.  $T_0$  phải là kiểu có thứ tự đếm được "đủ nhỏ", chẳng hạn kiểu đoạn con (giới hạn phụ thuộc vào chương trình dịch). Khi đó, ta có thể xác định kiểu tập  $T$

type  $T = \text{set of } T_0$

Mỗi đối tượng của kiểu  $T$  sẽ là một tập con của tập  $T_0$ .

### 11. Kiểu file (tệp)

Giả sử  $T_0$  là một kiểu nào đó (trừ kiểu file).

Khi đó,

type  $T = \text{file of } T_0$

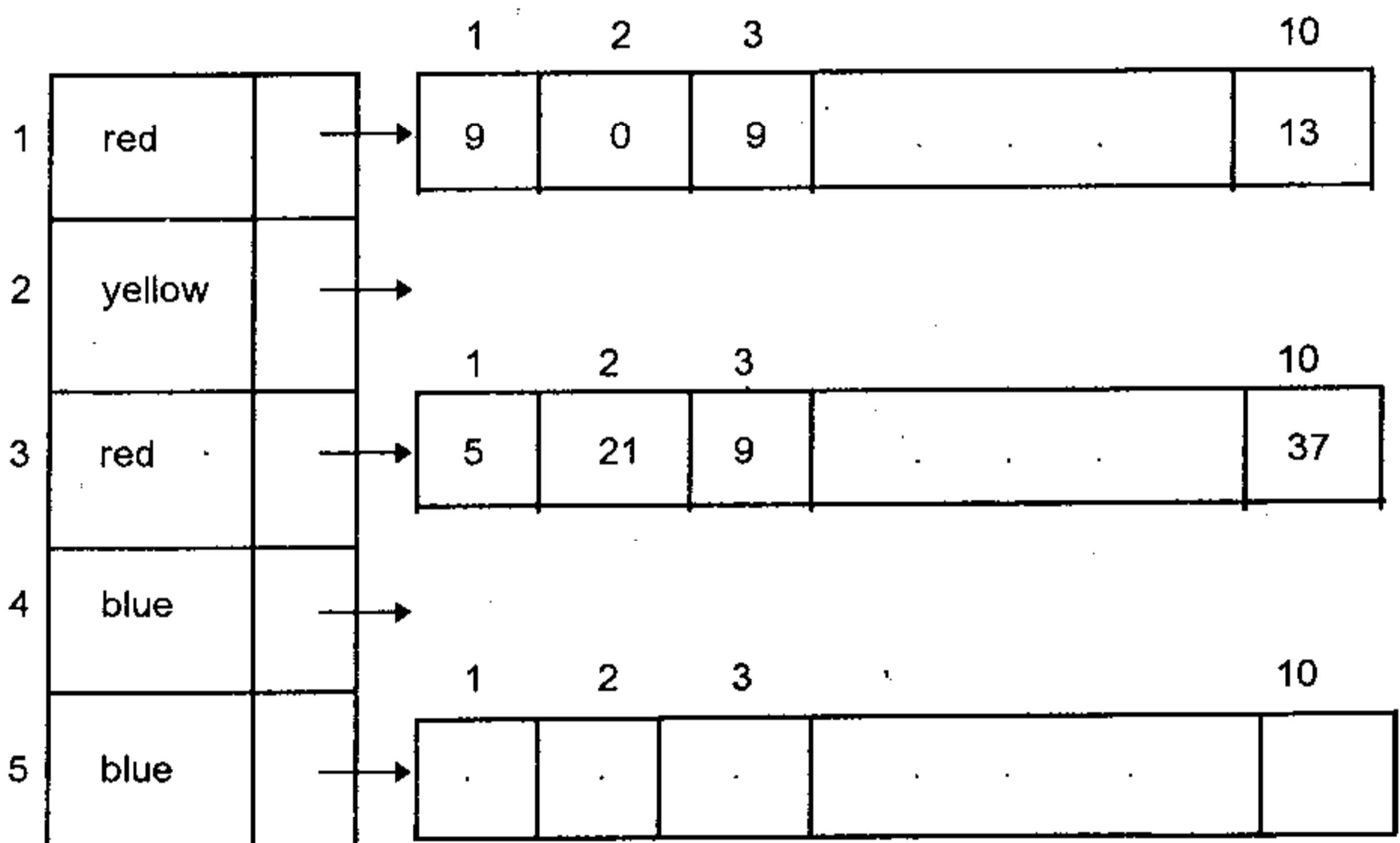
sẽ xác định một kiểu file với các phần tử là các đối tượng thuộc kiểu  $T_0$ .

**Ví dụ:** Sau đây là định nghĩa một số kiểu dữ liệu

```

type     Color = (white, red, blue, yellow, green) ;
          Intarr = array [1 ...10] of integer;
          Rec = record
            Infor : color;
            ptr : ^Intarr;
          end;
          Recarr = array [1 ... 5] of Rec ;
    
```

Biểu diễn hình học của một đối tượng thuộc kiểu Recarr được cho trong hình 2.3.



Hình 2.3. Cấu trúc dữ liệu Recarr

### Các phép toán trong hệ kiểu Pascal

Như đã nói với mỗi kiểu dữ liệu ta chỉ có thể thực hiện một số phép toán nhất định trên các dữ liệu của kiểu. Ta không thể áp dụng một phép toán trên các dữ liệu thuộc kiểu này cho các dữ liệu có kiểu khác. Ta có thể phân tách hợp các phép toán trên các kiểu dữ liệu của Pascal thành hai lớp sau :

**A. Các phép toán truy cập** đến các thành phần của một đối tượng dữ liệu, chẳng hạn truy cập đến các thành phần của một mảng, đến các trường của một bản ghi.

Giả sử A là một mảng với tập chỉ số I, khi đó A[i] cho phép ta truy cập đến thành phần thứ i của mảng. Còn nếu X là một bản ghi thì việc truy cập đến trường F của nó được thực hiện bởi phép toán X.F.

### **B. Các phép toán kết hợp dữ liệu**

Pascal có một tập hợp phong phú các phép toán kết hợp một hoặc nhiều dữ liệu đã cho thành một dữ liệu mới. Sau đây là một số nhóm các phép toán chính.

1. Các phép toán số học. Đó là các phép toán +, -, \*, / trên các số thực; các phép toán +, -, \*, /, div, mod trên các số nguyên.

2. Các phép toán so sánh. Trên các đối tượng thuộc các kiểu có thứ tự (đó là các kiểu cơ sở và kiểu tập), ta có thể thực hiện các phép toán so sánh =, <>, <, <=, >, >=. Cần lưu ý rằng, kết quả của các phép toán này là một giá trị kiểu boolean (tức là true hoặc false).

3. Các phép toán logic. Đó là các phép toán and, or, not được thực hiện trên hai giá trị false và true của kiểu boolean.

4. Các phép toán tập hợp. Các phép toán hợp, giao, hiệu của các tập hợp trong pascal được biểu diễn bởi +, \*, - tương ứng. Việc kiểm tra một đối tượng x có là phần tử của tập A hay không được thực hiện bởi phép toán x in A. Kết quả của phép toán này là một giá boolean.

## **2.4. MÔ HÌNH DỮ LIỆU VÀ KIỂU DỮ LIỆU TRÙU TƯỢNG**

Để giải quyết một vấn đề trên MTĐT thông thường chúng ta cần phải qua một số giai đoạn chính sau đây :

1. Đặt bài toán
2. Xây dựng mô hình
3. Thiết kế thuật toán và phân tích thuật toán
4. Viết chương trình
5. Thủ nghiệm

Chúng ta sẽ không đi sâu phân tích từng giai đoạn. Chúng ta chỉ muốn làm sáng tỏ vai trò của mô hình dữ liệu trong việc thiết kế chương trình. Xét ví dụ sau.

**Ví dụ:** Một người giao hàng, hàng ngày anh ta phải chuyển hàng từ một thành phố đến một số thành phố khác rồi lại quay về thành phố xuất phát. Vấn đề của anh ta là làm thế nào có được một hành trình chỉ qua mỗi thành phố một lần với đường đi ngắn nhất có thể được.

Chúng ta thử giúp người giao hàng mô tả chính xác bài toán. Trước hết, ta cần trả lời câu hỏi, những thông tin đã biết trong bài toán người giao hàng là gì? Đó là tên của các thành phố anh ta phải ghé qua và độ dài các con đường có thể có giữa hai thành phố. Chúng ta cần tìm cái gì? Một hành trình mà người giao hàng mong muốn là một danh sách các thành phố  $A_1, A_2, \dots, A_{n+1}$  (giả sử có  $n$  thành phố), trong đó các  $A_i$  ( $i = 1, 2, \dots, n+1$ ) đều khác nhau, trừ  $A_{n+1} = A_1$ .

Với một vấn đề đặt ra từ thực tiễn, ta có thể mô tả chính xác vấn đề đó hoặc các bộ phận của nó (vấn đề con) bởi một mô hình toán học nào đó. Chẳng hạn, mô hình toán học thích hợp nhất để mô tả bài toán người giao hàng là đồ thị. Các đỉnh của đồ thị là các thành phố, các cạnh của đồ thị là các đường nối hai thành phố. Trọng số của các cạnh là độ dài các đường nối hai thành phố. Trong thuật ngữ của lý thuyết đồ thị, danh sách các thành phố biểu diễn hành trình của người giao hàng, là một chu trình qua tất cả các đỉnh của đồ thị. Như vậy, bài toán người giao hàng được qui về bài toán trong lý thuyết đồ thị. Tìm một chu trình xuất phát từ một đỉnh qua tất cả các đỉnh còn lại với độ dài ngắn nhất.

Bài toán người giao hàng là một trong các bài toán đã trở thành kinh điển. Nó dễ mô hình hóa, song cũng rất khó giải. Chúng ta sẽ quay lại bài toán này.

Cần lưu ý rằng, để tìm ra cấu trúc toán học thích hợp với một bài toán đã cho, chúng ta phải phân tích kỹ bài toán để tìm ra câu trả lời cho các câu hỏi sau.

Các thông tin quan trọng của bài toán có thể biểu diễn bởi các đối tượng toán học nào?

Có các mối quan hệ nào giữa các đối tượng?

Các kết quả phải tìm của bài toán có thể biểu diễn bởi các khái niệm toán học nào.

Sau khi đã có mô hình toán học mô tả bài toán, một câu hỏi đặt ra là, ta phải làm việc với mô hình như thế nào để tìm ra lời giải của bài toán? Chúng ta sẽ thiết kế thuật toán thông qua các hành động, các phép toán thực hiện trên các đối tượng của mô hình.

Một mô hình toán học cùng với các phép toán có thể thực hiện trên các đối tượng của mô hình được gọi là *mô hình dữ liệu*. Chẳng hạn, trong mô hình dữ liệu đồ thị, trong số rất nhiều các phép toán, ta có thể kể ra một số phép toán sau : tìm các đỉnh kề của mỗi đỉnh, xác định đường đi ngắn nhất nối hai đỉnh bất kỳ, tìm các thành phần liên thông, tìm các đỉnh treo... Về mặt toán học, *danh sách* là một dãy hữu hạn n phần tử ( $a_1, a_2, \dots, a_n$ ). Trong mô hình dữ liệu danh sách, chúng ta cũng có thể thực hiện một tập hợp rất đa dạng các phép toán, chẳng hạn như, xác định độ dài của danh sách, xen một phần tử mới vào danh sách, loại một phần tử nào đó khỏi danh sách, sắp xếp lại danh sách theo một trật tự nào đó, gộp hai danh sách thành một danh sách.

Trở lại bài toán người giao hàng. Có nhiều thuật toán giải bài toán này. Chẳng hạn, ta có thể giải bằng phương pháp vét cạn: giả sử có n thành phố, khi đó mỗi hành trình là một hoán vị của n-1 thành phố (trừ thành phố xuất phát), thành lập  $(n-1)!$  hoán vị, tính độ dài của hành trình ứng với mỗi hoán vị và so sánh, ta sẽ tìm được hành trình ngắn nhất. Ta cũng có thể giải bài toán bằng các phương pháp khác: Chẳng hạn, ta có thể giải bài toán này bằng phương pháp nhánh-và-cận hoặc bằng phương pháp qui hoạch động (các phương pháp này sẽ được trình bày trong chương 8). Sau đây ta đưa ra một thuật toán đơn giản. Thuật toán này tìm ra rất nhanh nghiệm gần đúng, trong trường hợp có đường đi nối hai thành phố bất kỳ. Giả sử G là một đồ thị (Graph), V là tập hợp các đỉnh (Node), E là tập hợp các cạnh của nó. Giả sử  $c(u,v)$  là độ dài (nguyên dương) của cạnh  $(u,v)$ . Hành trình (Tour) của người giao hàng có thể xem như một tập hợp nào đó các cạnh. Cost là độ dài của hành trình. Thuật toán được biểu diễn bởi thủ tục Salesperson.

```

Procedure Salesperson (G : Graph; var Tour : set of E; var cost : integer);
    var
        v, w : Node;
        U : set of V;
    begin
        Tour := [];
        Cost := 0;
        v := v0; {v0 - đỉnh xuất phát}
        U := V - [v0];
        while U < > [ ] do
            begin
                Chọn (v, w) là cạnh ngắn nhất với w thuộc U ;
                Tour := Tour + [(v, w)];
                Cost := Cost + c(v,w);
                v := w;
                U := U - [w];
            end ;
            Tour := Tour + [(v,v0)];
            Cost := Cost + c(v,v0);
        end;
    
```

Thuật toán Salesperson được xây dựng trên cơ sở mô hình dữ liệu đồ thị và mô hình dữ liệu tập hợp. Nó chứa các thao tác trên đồ thị, các phép toán tập hợp. Tư tưởng của thuật toán như sau. Xuất phát từ Tour là tập rỗng. Giả sử ta xây dựng được đường đi từ đỉnh xuất phát  $v_0$  tới đỉnh  $v$ . Bước tiếp theo, ta sẽ thêm vào Tour cạnh  $(v, w)$ , đó là cạnh ngắn nhất từ  $v$  tới các đỉnh  $w$  không nằm trên đường đi từ  $v_0$  tới  $v$ . Để có được chương trình, chúng ta phải biểu diễn đồ thị, tập hợp bởi các cấu trúc dữ liệu. Sau đó viết các thủ tục (hoặc hàm) thực hiện các thao tác, các phép toán trên đồ thị, tập hợp có trong thuật toán.

Tóm lại, quá trình giải một bài toán có thể quy về hai giai đoạn kế tiếp như sau

1. Xây dựng các mô hình dữ liệu mô tả bài toán. Thiết kế thuật toán bằng cách sử dụng các thao tác, các phép toán trên các mô hình dữ liệu.

2. Biểu diễn các mô hình dữ liệu bởi các cấu trúc dữ liệu. Với các cấu trúc dữ liệu đã lựa chọn, các phép toán trên các mô hình dữ liệu được thể hiện bởi các thủ tục (hàm) trong ngôn ngữ lập trình nào đó.

Toán học đã cung cấp cho Tin học rất nhiều cấu trúc toán học có thể dùng làm mô hình dữ liệu. Chẳng hạn, các khái niệm toán học như dãy, tập hợp, ánh xạ, cây, đồ thị, quan hệ, nửa nhóm, nhóm, otomat,... Trong các chương sau chúng ta sẽ lần lượt nghiên cứu một số mô hình dữ liệu quan trọng nhất, được sử dụng thường xuyên trong các thuật toán. Đó là các mô hình dữ liệu danh sách, cây, tập hợp. Với mỗi mô hình dữ liệu chúng ta nghiên cứu các cách cài đặt nó bởi các cấu trúc dữ liệu khác nhau. Trong mỗi cách cài đặt, một số phép toán trên mô hình có thể được thực hiện dễ dàng, nhưng các phép toán khác có thể lại không thuận tiện. Việc lựa chọn cấu trúc dữ liệu nào để biểu diễn mô hình phụ thuộc vào từng ứng dụng.

Như đã nói, với mỗi mô hình dữ liệu, chúng ta có thể thực hiện một tập hợp các phép toán rất đa dạng, phong phú. Song trong nhiều ứng dụng, chúng ta chỉ sử dụng mô hình với một số xác định các phép toán nào đó. Khi đó chúng ta sẽ có một kiểu dữ liệu trừu tượng.

Như vậy, một *kiểu dữ liệu trừu tượng* (abstract data type) là một mô hình dữ liệu được xét cùng với một số xác định các phép toán nào đó. Chẳng hạn, các tập hợp được xét với các phép toán : thêm một phần tử vào một tập đã cho, loại một phần tử khỏi một tập hợp đã cho, tìm xem một phần tử đã cho có nằm trong một tập hợp hay không, lập thành kiểu dữ liệu trừu tượng (KDLTT) từ điển (dictionnaire).

Còn KDLTT hàng (hàng đợi) là mô hình dữ liệu danh sách cùng với hai phép toán chính là : thêm một phần tử mới vào một đầu danh sách, và loại một phần tử ở một đầu khác của danh sách. Chúng ta sẽ nghiên cứu kỹ một số kiểu dữ liệu trừu tượng quan trọng nhất : hàng, ngăn xếp (stack), từ điển, hàng ưu tiên. Với mỗi KDLTT, các cấu trúc dữ liệu để biểu diễn nó sẽ được nghiên cứu. Chúng ta cũng sẽ đánh giá hiệu quả của các phép toán trong từng cách cài đặt.

## **Chương 3**

# **DANH SÁCH**

Trong chương này, chúng ta sẽ nghiên cứu danh sách, một trong các mô hình dữ liệu quan trọng nhất, được sử dụng thường xuyên trong các thuật toán. Các phương pháp khác nhau để cài đặt danh sách sẽ được xét. Chúng ta sẽ phân tích hiệu quả của các phép toán trên danh sách trong mỗi cách cài đặt. Hai kiểu dữ liệu trừu tượng đặc biệt quan trọng là ngăn xếp (stack) và hàng (hàng đợi) sẽ được nghiên cứu. Chúng ta cũng sẽ trình bày một số ứng dụng của danh sách.

### **3.1. DANH SÁCH**

Về mặt toán học, danh sách là một dãy hữu hạn các phần tử thuộc cùng một lớp các đối tượng nào đó. Chẳng hạn, ta có thể nói đến danh sách các sinh viên của một lớp, danh sách các số nguyên nào đó, danh sách các báo xuất bản hàng ngày ở thủ đô, ...

Giả sử  $L$  là danh sách có  $n$  ( $n \geq 0$ ) phần tử

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

Ta gọi số  $n$  là *độ dài* của danh sách. Nếu  $n \geq 1$  thì  $a_1$  được gọi là *phần tử đầu tiên* của danh sách, còn  $a_n$  là *phần tử cuối cùng* của danh sách. Nếu  $n = 0$  tức danh sách không có phần tử nào, thì danh sách được gọi là rỗng.

Một tính chất quan trọng của danh sách là các phần tử của nó được sắp tuyến tính: nếu  $n > 1$  thì phần tử  $a_i$  “đi trước” phần tử  $a_{i+1}$  hay  $a_{i+1}$  “đi sau” phần tử  $a_i$  với  $i = 1, 2, \dots, n-1$ . Ta sẽ nói  $a_i$  ( $i = 1, 2, \dots, n$ ) là phần tử ở vị trí thứ  $i$  của danh sách.

Cần chú ý rằng, một đối tượng có thể xuất hiện nhiều lần trong một danh sách. Chẳng hạn như trong danh sách các số ngày của các tháng trong một năm

(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

### **Danh sách con**

Nếu  $L = (a_1, a_2, \dots, a_n)$  là một danh sách và  $i, j$  là các vị trí,  $1 \leq i \leq j \leq n$  thì danh sách  $L' = (b_1, b_2, \dots, b_{j-i+1})$  trong đó  $b_1 = a_i, b_2 = a_{i+1}, \dots, b_{j-i+1} = a_j$ . Như vậy, danh sách con  $L'$  gồm tất cả các phần tử từ  $a_i$  đến  $a_j$  của danh sách  $L$ . Danh sách rỗng được xem là danh sách con của một danh sách bất kỳ.

Danh sách con bất kỳ gồm các phần tử bắt đầu từ phần tử đầu tiên của danh sách  $L$  được gọi là *phân đầu* (prefix) của danh sách  $L$ . *Phân cuối* (postfix) của danh sách  $L$  là một danh sách con bất kỳ kết thúc ở phần tử cuối cùng của danh sách  $L$ .

### **Dãy con**

Một danh sách được tạo thành bằng cách loại bỏ một số (có thể bằng không) phần tử của danh sách  $L$  được gọi là *dãy con* của danh sách  $L$ .

#### **Ví dụ. Xét danh sách**

$L = (\text{black}, \text{blue}, \text{green}, \text{cyan}, \text{red}, \text{brown}, \text{yellow})$

Khi đó danh sách ( $\text{blue}, \text{green}, \text{cyan}, \text{red}$ ) là danh sách con của  $L$ . Danh sách ( $\text{black}, \text{green}, \text{brown}$ ) là dãy con của  $L$ . Danh sách ( $\text{black}, \text{blue}, \text{green}$ ) là phần đầu, còn danh sách ( $\text{red}, \text{brown}, \text{yellow}$ ) là phần cuối của danh sách  $L$ .

### **Các phép toán trên danh sách**

Chúng ta đã trình bày khái niệm toán học danh sách. Khi mô tả một mô hình dữ liệu, chúng ta cần xác định các phép toán có thể thực hiện trên mô hình toán học được dùng làm cơ sở cho mô hình dữ liệu. Có rất nhiều phép toán trên danh sách. Trong các ứng dụng, thông thường chúng ta chỉ sử dụng một nhóm các phép toán nào đó. Sau đây là một số phép toán chính trên danh sách.

Giả sử  $L$  là một danh sách (List), các phần tử của nó có kiểu dữ liệu Item nào đó,  $p$  là một vị trí (position) trong danh sách. Các phép toán sẽ được mô tả bởi các thủ tục hoặc hàm.

1. Khởi tạo danh sách rỗng

**procedure** Initialize (**var** L : List) ;

2. Xác định độ dài của danh sách.

**function** Length (L : List) : **integer**;

3. Loại phần tử ở vị trí thứ p của danh sách

**procedure** Delete (p : position ; **var** L : List) ;

4. Xen phần tử x vào danh sách sau vị trí thứ p

**procedure** Insert After (p : position ; x : Item ; **var** L : List) ;

5. Xen phần tử x vào danh sách trước vị trí thứ p

**procedure** Insert Before (p : position ; x : Item ; **var** L : List) ;

6. Tìm xem trong danh sách có chứa phần tử x hay không?

**procedure** Search (x : Item ; L : List ; **var** found : **boolean**) ;

7. Kiểm tra danh sách có rỗng không?

**function** Empty (L : List) : **boolean** ;

8. Kiểm tra danh sách có đầy không?

**function** Full (L : List) : **boolean** ;

9. Đi qua danh sách. Trong nhiều áp dụng chúng ta cần phải đi qua danh sách, từ đầu đến hết danh sách, và thực hiện một nhóm hành động nào đó với mỗi phần tử của danh sách.

**procedure** Traverse (**var** L : List) ;

10. Các phép toán khác. Còn có thể kể ra nhiều phép toán khác. Chẳng hạn truy cập đến phần tử ở vị trí thứ i của danh sách (để tham khảo hoặc thay thế), kết hợp hai danh sách thành một danh sách, phân tích một danh sách thành nhiều danh sách, ...

**Ví dụ :** Giả sử L là danh sách L = (3, 2, 1, 5). Khi đó, thực hiện Delete (3, L) ta được danh sách (3, 2, 5). Kết quả của InsertBefore (1, 6, L) là danh sách (6, 3, 2, 1, 5).

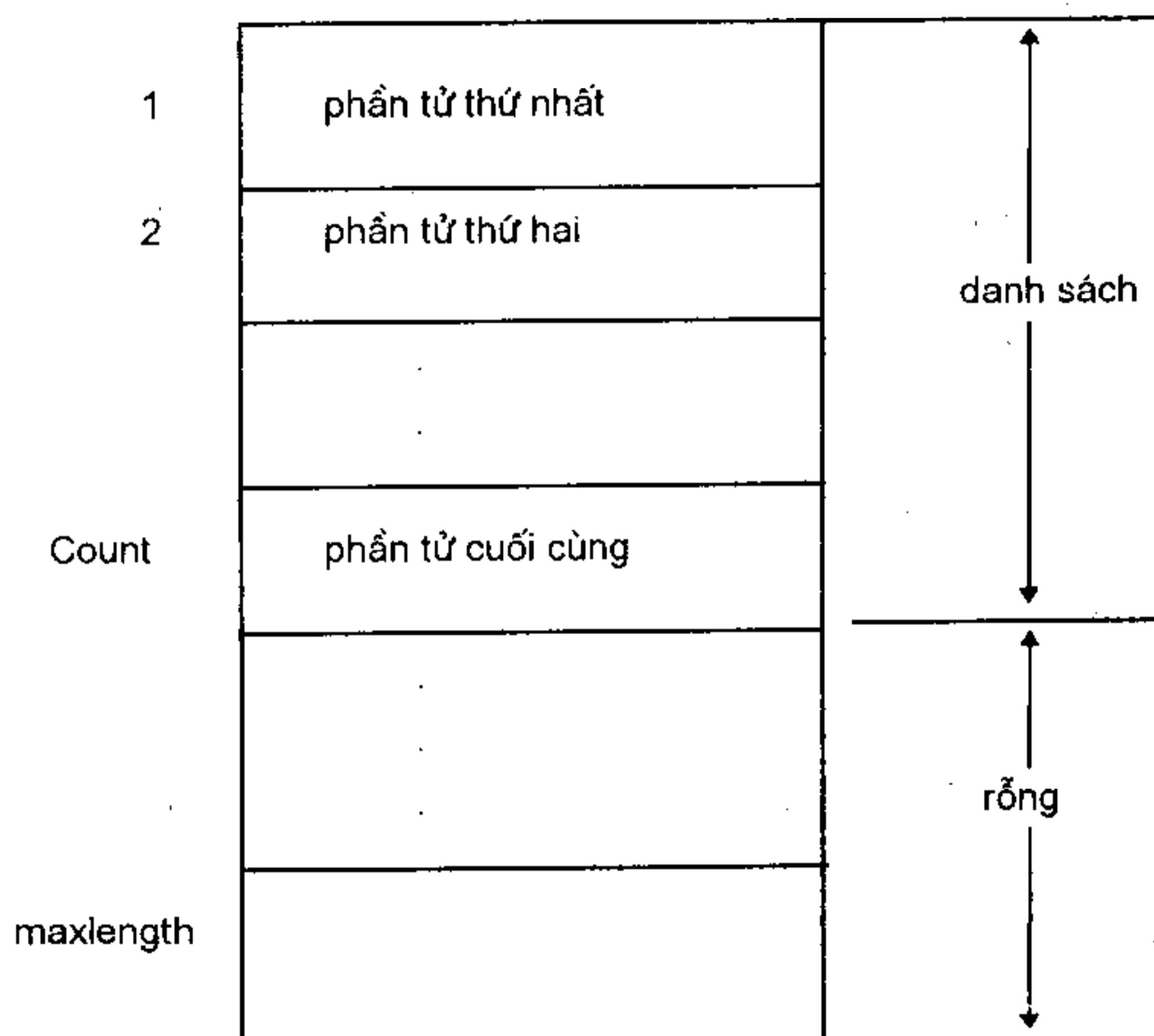
### 3.2. CÀI ĐẶT DANH SÁCH BỞI MẢNG

Phương pháp tự nhiên nhất để cài đặt một danh sách là sử dụng mảng, trong đó mỗi thành phần của mảng sẽ lưu giữ một phần tử nào

đó của danh sách, và các phần tử kế nhau của danh sách được lưu giữ trong các thành phần kế nhau của mảng.

Giả sử độ dài tối đa của danh sách (maxlength) là một số N nào đó, các phần tử của danh sách có kiểu dữ liệu là Item. Item có thể là các kiểu dữ liệu đơn, hoặc các dữ liệu có cấu trúc, thông thường Item là bản ghi. Chúng ta biểu diễn danh sách (List) bởi bản ghi gồm hai trường. Trường thứ nhất là mảng các Item phần tử thứ i của danh sách được lưu giữ trong thành phần thứ i của mảng. Trường thứ hai ghi chỉ số của thành phần mảng lưu giữ phần tử cuối cùng của danh sách (xem hình 3.1). Chúng ta có các khai báo như sau :

```
const maxlength = N ;
type List = record
    element : array [1 ... maxlength] of Item ;
    count : 0 ... maxlength ;
end ;
var L : List ;
```



Hình 3.1. Mảng biểu diễn danh sách

Trong cách cài đặt danh sách bởi mảng, các phép toán trên danh sách được thực hiện rất dễ dàng. Để khởi tạo một danh sách rỗng, chỉ cần một lệnh gán

```
L.count := 0;
```

Độ dài của danh sách là L.count. Danh sách đầy, nếu L.count = maxlen.

Sau đây là các thủ tục thực hiện các phép toán xen một phần tử mới vào danh sách và loại một phần tử khỏi danh sách.

### *Thủ tục loại bỏ*

```
procedure Delete (p : 1 ... maxlen; var L : List; var OK : boolean);
  var i : 1 ... maxlen;
begin
  OK := false;
  with L do
    if p <= count then
      begin
        i := p;
        while i < count do
          begin
            element [i] := element [i + 1];
            i := i + 1
          end;
        count := count - 1;
        OK := true;
      end;
end;
```

Thủ tục trên thực hiện phép loại bỏ phần tử ở vị trí p khỏi danh sách. Phép toán chỉ được thực hiện khi danh sách không rỗng và p chỉ vào một phần tử trong danh sách. Tham biến OK ghi lại phép toán có được thực hiện thành công hay không. Khi loại bỏ, chúng ta phải dồn các phần tử các vị trí p+1, p + 2, ... lên trên một vị trí.

**Thủ tục xen vào**

```

procedure InsertBefore (p : 1 .. maxlen ; x : Item ;
                        var L : List ; var OK : boolean) ;
    var      i : 1.. maxlen ;
begin
    OK := false ;
    with L do
        if (count < maxlen) and (p <= count) then
            begin
                i := count + 1 ;
                while i > p do
                    begin
                        element[i]:= element[i-1] ;
                        i:=i-1 ;
                    end ;
                element [p] := x ;
                count := count + 1 ;
                OK := true ;
            end ;
    end ;
end ;

```

Thủ tục trên thực hiện việc xen phần tử mới x vào trước phần tử ở vị trí p trong danh sách. Phép toán này chỉ được thực hiện khi danh sách chưa đầy ( $count < maxlen$ ) và p chỉ vào một phần tử trong danh sách ( $p \leq count$ ). Chúng ta phải dồn các phần tử ở các vị trí p, p+1, ... xuống dưới một vị trí để lấy chỗ cho x.

Nếu n là độ dài của danh sách ; dễ dàng thấy rằng, cả hai phép toán loại bỏ và xen vào được thực hiện trong thời gian  $O(n)$ .

Việc tìm kiếm trong danh sách là một phép toán được sử dụng thường xuyên trong các ứng dụng. Chúng ta sẽ xét riêng phép toán này trong mục sau.

**Nhận xét về phương pháp biểu diễn danh sách bởi mảng**

Chúng ta đã cài đặt danh sách bởi mảng, tức là dùng mảng để lưu giữ các phần tử của danh sách. Do tính chất của mảng, phương pháp

này cho phép ta truy cập trực tiếp đến phần tử ở vị trí bất kỳ trong danh sách. Các phép toán khác đều được thực hiện rất dễ dàng. Tuy nhiên phương pháp này không thuận tiện để thực hiện phép toán xen vào và loại bỏ. Như đã chỉ ra ở trên, mỗi lần cần xen phần tử mới vào danh sách ở vị trí  $p$  (hoặc loại bỏ phần tử ở vị trí  $p$ ) ta phải đẩy xuống dưới (hoặc lên trên) một vị trí tất cả các phần tử đi sau phần tử thứ  $p$ . Nhưng hạn chế chủ yếu của cách cài đặt này là ở không gian nhớ cố định giành để lưu giữ các phần tử của danh sách. Không gian nhớ này bị quy định bởi cỡ của mảng. Do đó danh sách không thể phát triển quá cỡ của mảng, phép toán xen vào sẽ không được thực hiện khi mảng đã đầy.

### 3.3. TÌM KIẾM TRÊN DANH SÁCH

#### 3.3.1. Vấn đề tìm kiếm

Tìm kiếm thông tin là một trong các vấn đề quan trọng nhất trong tin học. Cho trước một số điện thoại, chúng ta cần tìm biết người có số điện thoại đó, địa chỉ của anh ta, và những thông tin khác gắn với số điện thoại đó. Thông thường các thông tin về một đối tượng được biểu diễn dưới dạng một bản ghi, các thuộc tính của đối tượng là các trường của bản ghi. Trong bài toán tìm kiếm, chúng ta sẽ tiến hành tìm kiếm các đối tượng dựa trên một số thuộc tính đã biết về đối tượng, chúng ta sẽ gọi các thuộc tính này là *khoá tìm kiếm*. Như vậy, *khoá tìm kiếm* được hiểu là một hoặc một số trường nào đó của bản ghi. Với một giá trị cho trước của khoá, có thể có nhiều bản ghi có khoá đó. Cũng có thể xảy ra, không có bản ghi nào có giá trị khoá đã cho.

Thời gian tìm kiếm phụ thuộc vào cách chúng ta tổ chức thông tin và phương pháp tìm kiếm được sử dụng. Chúng ta có thể tổ chức các đối tượng để tìm kiếm dưới dạng danh sách, hoặc cây tìm kiếm nhị phân, ... Với mỗi cách cài đặt (Chẳng hạn, có thể cài đặt danh sách bởi mảng, hoặc danh sách liên kết), chúng ta sẽ có phương pháp tìm kiếm thích hợp.

Người ta phân biệt hai loại tìm kiếm : tìm kiếm trong và tìm kiếm ngoài. Nếu khối lượng thông tin lớn cần lưu giữ dưới dạng các file ở bộ

nhớ ngoài, như đĩa từ hoặc băng từ, thì sự tìm kiếm được gọi là tìm kiếm ngoài. Trong trường hợp thông tin được lưu giữ ở bộ nhớ trong, ta nói đến tìm kiếm trong. Trong chương này và các chương sau, chúng ta chỉ đề cập tìm kiếm trong.

Sau đây chúng ta sẽ nghiên cứu các phương pháp tìm kiếm trên danh sách được biểu diễn bởi mảng.

### 3.3.2. Tìm kiếm tuần tự

Giả sử keytype là kiểu của khoá. Trong nhiều trường hợp keytype là **integer**, **real**, hoặc **string**. Các phần tử của danh sách có kiểu Item - bản ghi có chứa trường key kiểu keytype.

```

type keytype = ... ;
Item  = record
        key : keytype ;
        [các trường khác]
        ....
    end ;
List   = record
        element : array [1..max] of Item ;
        count : 0 .. max ;
    end ;

```

Tìm kiếm tuần tự (hay tìm kiếm tuyến tính) là phương pháp tìm kiếm đơn giản nhất: xuất phát từ đầu danh sách, chúng ta tuần tự đi trên danh sách cho tới khi tìm ra phần tử có khoá đã cho thì dừng lại, hoặc đi đến hết danh sách mà không tìm thấy. Ta có thủ tục tìm kiếm sau.

```

procedure SeqSearch (var L : List ; x : keytype ;
                      var found : boolean ; var p : 1..max) ;
begin
    found := false ;

```

```

p := 1 ;
with L do
  while (not found) and ( p <= count) do
    if element [p]. key = x then found := true
    else p := p + 1 ;
end ;

```

Thủ tục trên tìm xem trong danh sách L có chứa phần tử với khoá là x hay không. Nếu có thì giá trị của tham biến found là true. Trong trường hợp có, biến p sẽ ghi lại vị trí của phần tử đầu tiên có khoá là x.

### **Phân tích tìm kiếm tuần tự**

Giả sử độ dài của danh sách là n (count = n). Dễ dàng thấy rằng, thời gian thực hiện tìm kiếm tuần tự là thời gian thực hiện lệnh *while*. Mỗi lần lặp cần thực hiện phép so sánh khoá x với khoá của một phần tử trong danh sách, số lớn nhất các lần lặp là n, do đó thời gian tìm kiếm tuần tự là O(n).

### **3.3.3. Tìm kiếm nhị phân**

Giả sử L là một danh sách có độ dài n và được biểu diễn bởi mảng, các phần tử của nó có kiểu Item được mô tả như trong mục 3.3.2. Giả sử kiểu của khoá keytype là kiểu có thứ tự, tức là với hai giá trị bất kỳ của nó  $v_1$  và  $v_2$ , ta luôn luôn có  $v_1 \leq v_2$ , hoặc  $v_1 \geq v_2$ ; trong đó  $\leq$  là một quan hệ thứ tự nào đó được xác định trên keytype. Giả sử các phần tử của danh sách L được sắp xếp theo thứ tự khoá không giảm:

$$\begin{aligned}
& L.\text{element}[1].\text{key} \leq L.\text{element}[2].\text{key} \leq \dots \\
& \quad \leq L.\text{element}[n].\text{key}
\end{aligned}$$

Trong trường hợp này, chúng ta có thể áp dụng phương pháp tìm kiếm khác, hiệu quả hơn phương pháp tìm kiếm tuần tự. Đó là kỹ thuật *tìm kiếm nhị phân*. Tư tưởng của tìm kiếm nhị phân như sau: Đầu tiên ta so sánh khoá x với khoá của phần tử ở giữa danh sách, tức phần tử ở

vị trí  $m = \lfloor (1+n)/2 \rfloor^1$ . Nếu chúng bằng nhau,  $x = L.element[m].key$ , ta đã tìm thấy. Nếu  $x < L.element[m].key$ , ta tiếp tục tìm kiếm trong nửa đầu danh sách từ vị trí 1 đến vị trí  $m-1$ . Còn nếu  $x > L.element[m].key$ , ta tiếp tục tìm kiếm trong nửa cuối danh sách từ vị trí  $m + 1$  đến vị trí  $n$ . Nếu đến một thời điểm nào đó, ta phải tìm  $x$  trong một danh sách con rỗng, thì điều đó có nghĩa là trong danh sách không có phần tử nào với khoá  $x$ .

Chúng ta có thể mô tả phương pháp tìm kiếm nhị phân bởi thủ tục sau:

```

procedure BinarySearch (var L : List ; x : keytype ;
                                var found : boolean ; p : 1 .. max) ;

var
    mid, bottom, top : integer ;

begin
    (1) found := false ;
    (2) bottom := 1,
    (3) top := L.count ;
    (4) while (not found) and (bottom <= top) do
        begin
            (5) mid := (bottom + top) div 2 ;
            (6) if x = L.element[mid].key then found := true
                else if x < L.element[mid].key then top := mid - 1
                else bottom := mid + 1 ;
        end ;
    (7) p := mid ;
end ;

```

<sup>1</sup> Ký hiệu  $\lfloor a \rfloor$  chỉ phần nguyên của a, tức là số nguyên lớn nhất nhỏ hơn hoặc bằng a ; chẳng hạn  $\lfloor 5 \rfloor = 5$ ,  $\lfloor 5.2 \rfloor = 5$  còn  $\lceil a \rceil$  chỉ số nguyên nhỏ nhất lớn hơn hoặc bằng chẳng hạn  $\lceil 6.3 \rceil = 7$ ,  $\lceil 6 \rceil = 6$ .

Trong thủ tục trên, ta đã đưa vào hai biến bottom và top để ghi lại vị trí đầu và vị trí cuối của danh sách con mà ta cần tiếp tục tìm kiếm. Biến mid ghi lại vị trí giữa của mỗi danh sách con. Quá trình tìm kiếm được thực hiện bởi vòng lặp **while**. Mỗi lần lặp khoá x được so sánh với khoá của phần tử ở giữa danh sách. Nếu bằng nhau, found := true và dừng lại. Nếu x nhỏ hơn, ta tiếp tục tìm ở nửa đầu của danh sách con đang xét (đặt lại top := mid - 1). Nếu x lớn hơn, ta sẽ tìm tiếp ở nửa cuối danh sách (đặt lại bottom := mid + 1).

### *Phân tích tìm kiếm nhị phân*

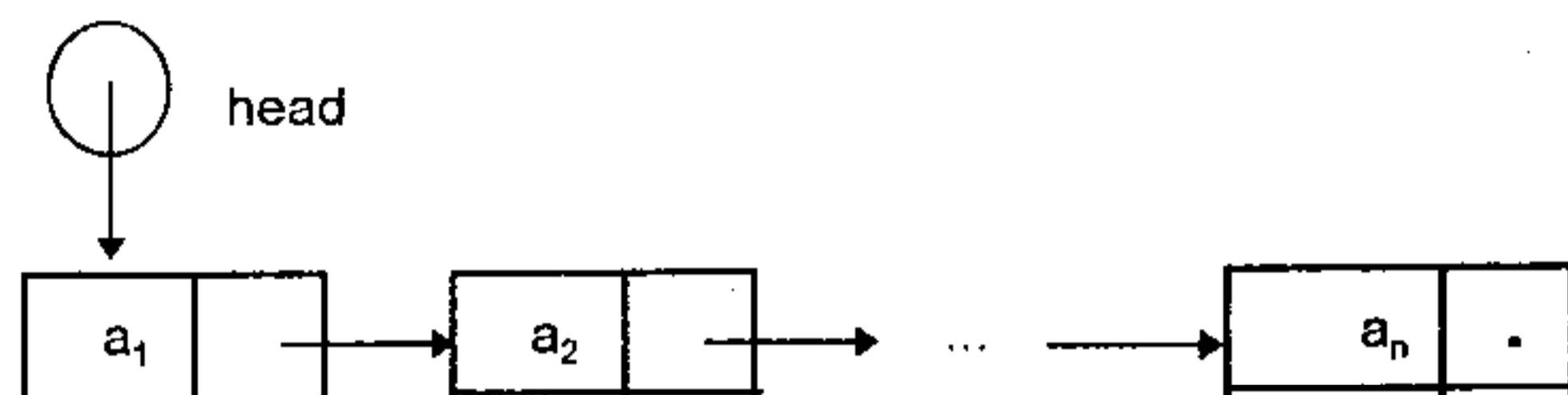
Trực quan, ta thấy ngay tìm kiếm nhị phân hiệu quả hơn tìm kiếm tuần tự, bởi vì trong tìm kiếm tuần tự ta phải lần lượt xét từng phần tử của danh sách, bắt đầu từ phần tử đầu tiên cho tới khi phát hiện ra phần tử cần tìm hoặc không. Còn trong tìm kiếm nhị phân, mỗi bước ta chỉ cần xét phần tử ở giữa danh sách, nếu chưa phát hiện ra ta lại xét tiếp phần tử ở giữa nửa đầu hoặc nửa cuối danh sách. Sau đây, ta đánh giá thời gian thực hiện tìm kiếm nhị phân. Giả sử độ dài danh sách là n. Thời gian thực hiện các lệnh (1) - (3) và (7) là O(1). Vì vậy thời gian thực hiện thủ tục là thời gian thực hiện lệnh **while** (4). Thân của lệnh lặp này (các lệnh (4) và (5) có thời gian thực hiện là O(1). Gọi t là số lần lặp tối đa cần thực hiện. Sau mỗi lần lặp độ dài của danh sách giảm đi một nửa, từ điều kiện bottom  $\leq$  top, ta suy ra t là số nguyên dương lớn nhất sao cho  $2^t \leq n$ , tức là  $t \leq \log_2 n$ . Như vậy, thời gian tìm kiếm nhị phân trong một danh sách có n phần tử là O( $\log_2 n$ ), trong khi đó thời gian tìm kiếm tuần tự là O(n).

## 3.4. CẤU TRÚC DỮ LIỆU DANH SÁCH LIÊN KẾT

### 3.4.1. Danh sách liên kết

Trong mục này chúng ta sẽ biểu diễn danh sách bởi cấu trúc dữ liệu khác, đó là danh sách liên kết. Trong cách cài đặt này, danh sách liên kết được tạo nên từ các tế bào mỗi tế bào là một bản ghi gồm hai trường, trường infor "chứa" phần tử của danh sách, trường next là con trỏ trỏ đến phần tử đi sau trong danh sách. Chúng ta sẽ sử dụng con trỏ head trỏ tới đầu danh sách. Như vậy một danh sách ( $a_1, a_2, \dots, a_n$ ) có thể biểu

điễn bởi cấu trúc dữ liệu danh sách liên kết được minh họa trong hình 3.2.



**Hình 3.2.** Danh sách liên kết biểu diễn danh sách ( $a_1, a_2, \dots, a_n$ )

Một danh sách liên kết được hoàn toàn xác định bởi con trỏ head trỏ tới đầu danh sách, do đó, ta có thể khai báo như sau.

```

type pointer = ^ cell;
      cell = record
        infor : Item ;
        next : pointer;
      end ;
var   head : pointer ;
  
```

Chú ý : Không nên nhầm lẫn danh sách và danh sách liên kết. Danh sách và danh sách liên kết là hai khái niệm hoàn toàn khác nhau. Danh sách là một mô hình dữ liệu, nó có thể được cài đặt bởi các cấu trúc dữ liệu khác nhau. Còn danh sách liên kết là một cấu trúc dữ liệu, ở đây nó được sử dụng để biểu diễn danh sách.

### 3.4.2. Các phép toán trên danh sách liên kết

Sau đây chúng ta sẽ xét xem các phép toán trên danh sách được thực hiện như thế nào khi mà danh sách được cài đặt bởi danh sách liên kết.

Điều kiện để một danh sách liên kết rỗng là

**head = nil**

Do đó, muốn khởi tạo một danh sách rỗng, ta chỉ cần lệnh gán :

**head := nil**

Danh sách liên kết chỉ đầy khi không còn không gian nhớ để cấp phát cho các thành phần mới của danh sách. Chúng ta sẽ giả thiết điều này không xảy ra, tức là danh sách liên kết không khi nào đầy. Do đó phép toán xen một phần tử mới vào danh sách sẽ luôn luôn được thực hiện.

### *Phép toán xen vào*

Giả sử  $Q$  là một con trỏ trỏ vào một thành phần của danh sách liên kết, và trong trường hợp danh sách rỗng ( $\text{head} = \text{nil}$ ) thì  $Q = \text{nil}$ . Chúng ta cần xen một thành phần mới với infor là  $x$  vào sau thành phần của danh sách được trỏ bởi  $Q$ . Phép toán này được thực hiện bởi thủ tục sau:

```

procedure InsertAfter (x : Item ; Q : pointer ; var head : pointer) ;
var P : pointer ;
begin
    new (P) ;
    P^.infor := x ;
    if head = nil then
        begin
            P^.next := nil ;
            head := P ;
        end
    else
        begin
            P^.next := Q^.next ;
            Q^.next := P ;
        end ;
    end ;

```

Các hành động trong thủ tục *InsertAfter* được minh họa trong hình 3.3

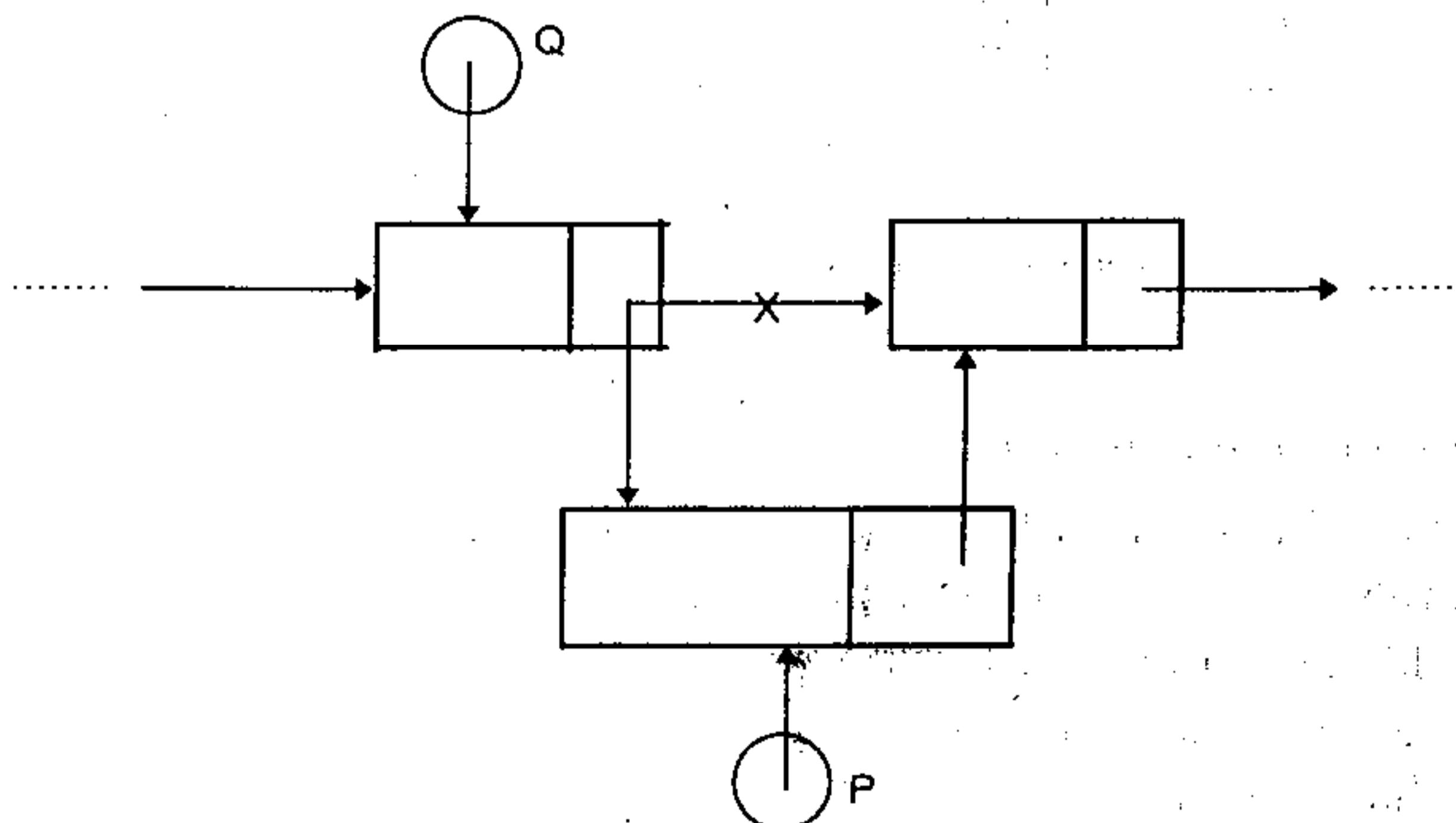
Giả sử bây giờ ta cần xen thành phần mới với infor là  $x$  vào trước thành phần của danh sách được trỏ bởi  $Q$ . Phép toán này (*InsertBefore*) phức tạp hơn. Khó khăn ở đây là, nếu  $Q$  không là thành phần đầu tiên của danh sách (tức là  $Q \neq \text{head}$ ) thì ta không định vị được thành phần đi trước thành phần  $Q$  để kết nối với thành phần sẽ được xen vào. Có thể

giải quyết khó khăn này bằng cách, đầu tiên ta vẫn xen thành phần mới vào sau thành phần Q, sau đó trao đổi giá trị chứa trong phần infor giữa thành phần mới và thành phần Q.

```

procedure InsertBefore (x : Item | Q : pointer ; var head : pointer);
var      P : pointer;
begin
  new (P);
  if Q = head then
    begin
      P^.infor := x;
      P^.next := Q;
      head := P;
    end
  else
    begin
      P^.next := Q^.next;
      Q^.next := P;
      P^.infor := Q^.infor;
      Q^.infor := x;
    end;
end;

```



Hình 3.3: Xen thành phần mới vào danh sách sau Q.

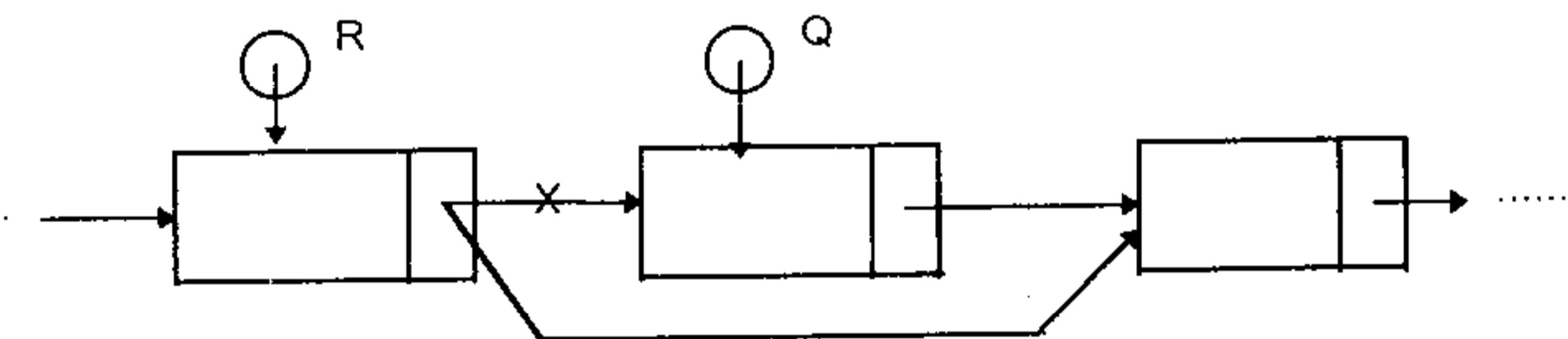
### *Phép toán loại bỏ*

Giả sử ta có một danh sách liên kết không rỗng ( $\text{head} \neq \text{nil}$ ),  $Q$  là một con trỏ trỏ vào một thành phần trong danh sách. Giả sử ta cần loại bỏ thành phần  $Q$  khỏi danh sách. Ở đây ta cũng gặp khó khăn như khi muốn xen một thành phần mới vào trước thành phần  $Q$ . Do đó, ta cần đưa vào một con trỏ  $R$  đi trước con trỏ  $Q$  một bước, tức là nếu  $Q$  không phải là thành phần đầu tiên, thì  $Q = R^{\wedge}.\text{next}$ . Khi đó phép toán loại bỏ thành phần  $Q$  khỏi danh sách được thực hiện rất dễ dàng. Ta có thủ tục sau:

```

procedure Delete (Q,R : pointer ; var head : pointer ; var x : Item);
begin
  x := Q^ .Infor ;
  if Q = head then head := Q^ .next
  else R^ .next := Q^ .next ;
end ;

```



Hình 3.4. Xoá thành phần  $Q$  khỏi danh sách.

### *Phép toán tìm kiếm*

Đối với danh sách liên kết, ta chỉ có thể áp dụng phương pháp tìm kiếm tuần tự. Cho dù danh sách đã được sắp xếp theo thứ tự không tăng (hoặc không giảm) của khoá tìm kiếm, ta cũng không thể áp dụng được phương pháp tìm kiếm nhị phân. Lý do là, ta không dễ dàng xác định được thành phần ở giữa của danh sách liên kết.

Giả sử chúng ta cần tìm trong danh sách thành phần với infor là  $x$  cho trước. Trong thủ tục tìm kiếm sau đây, ta sẽ cho con trỏ  $P$  chạy từ

đầu danh sách, lần lượt qua các thành phần của danh sách và dừng lại ở thành phần với  $\text{infor} = x$ . Biến  $\text{found}$  được sử dụng để ghi lại sự tìm kiếm thành công hay không.

```
procedure Search (x : Item ; head : pointer ; var P : pointer
                  var found : boolean) ;
begin
    P := head ;
    found := false ;
    while (P < > nil) and (not found) do
        if P^.infor = x then found := true
        else P := P^.next;
    end ;
```

Thông thường ta cần tìm kiếm để thực hiện các thao tác khác với danh sách. Chẳng hạn, ta cần loại bỏ khỏi danh sách thành phần với  $\text{infor} = x$  hoặc xen một thành phần mới vào trước (hoặc sau) thành phần với  $\text{infor} = x$ . Muốn thế, trước hết ta phải tìm trong danh sách thành phần với  $\text{infor} = x$  cho trước. Để cho phép loại bỏ và xen vào có thể thực hiện dễ dàng, ta đưa vào thủ tục tìm kiếm hai con trỏ đi cách nhau một bước. Con trỏ  $Q$  trỏ vào thành phần cần tìm, còn  $R$  trỏ vào thành phần đi trước. Ta có thủ tục sau :

```
procedure Search (x : Item ; head : pointer ; var Q, R : pointer;
                  var found : boolean) ;
begin
    R := nil ;
    Q := head ;
    found := false ;
    while (Q < > nil) and (not found) do
        if Q^.infor = x then found := true
        else begin
            R := Q ;
            Q := Q^.next ;
        end ;
    end ;
```

### **Phép toán đi qua danh sách**

Trong nhiều áp dụng, ta phải đi qua danh sách, 'thăm' tất cả các thành phần của danh sách. Với mỗi thành phần, ta cần thực hiện một số phép toán nào đó với các dữ liệu chứa trong phần infor. Các phép toán này, giả sử được mô tả trong thủ tục Visit. Ta có thủ tục sau.

```

procedure traverse (var head : pointer) ;
var
    P : pointer ;
begin
    P := head ;
    while P < > nil do
        begin
            Visit (P^) ;
            P := P^.next;
        end ;
    end ;

```

#### **3.4.3. So sánh hai phương pháp**

Chúng ta đã trình bày hai phương pháp cài đặt danh sách: cài đặt danh sách bởi mảng và cài đặt danh sách bởi danh sách liên kết. Một câu hỏi đặt ra là, phương pháp nào tốt hơn? Chúng ta chỉ có thể nói rằng, mỗi phương pháp đều có ưu điểm và hạn chế, việc lựa chọn phương pháp nào, mảng hay danh sách liên kết để biểu diễn danh sách, tuỳ thuộc vào từng ứng dụng. Sau đây là các nhận xét so sánh hai phương pháp.

1. Khi biểu diễn danh sách bởi mảng, chúng ta phải ước lượng độ dài tối đa của danh sách để khai báo cỡ của mảng. Sẽ xảy ra lãng phí bộ nhớ khi danh sách còn nhỏ. Nhưng trong thời gian chạy chương trình, nếu phép toán xen vào được thực hiện thường xuyên, sẽ có khả năng dẫn đến danh sách đầy. Trong khi đó nếu biểu diễn danh sách bởi danh sách liên kết, ta chỉ cần một lượng không gian nhớ cần thiết cho các phần tử hiện có của danh sách. Với cách biểu diễn này, sẽ không xảy ra

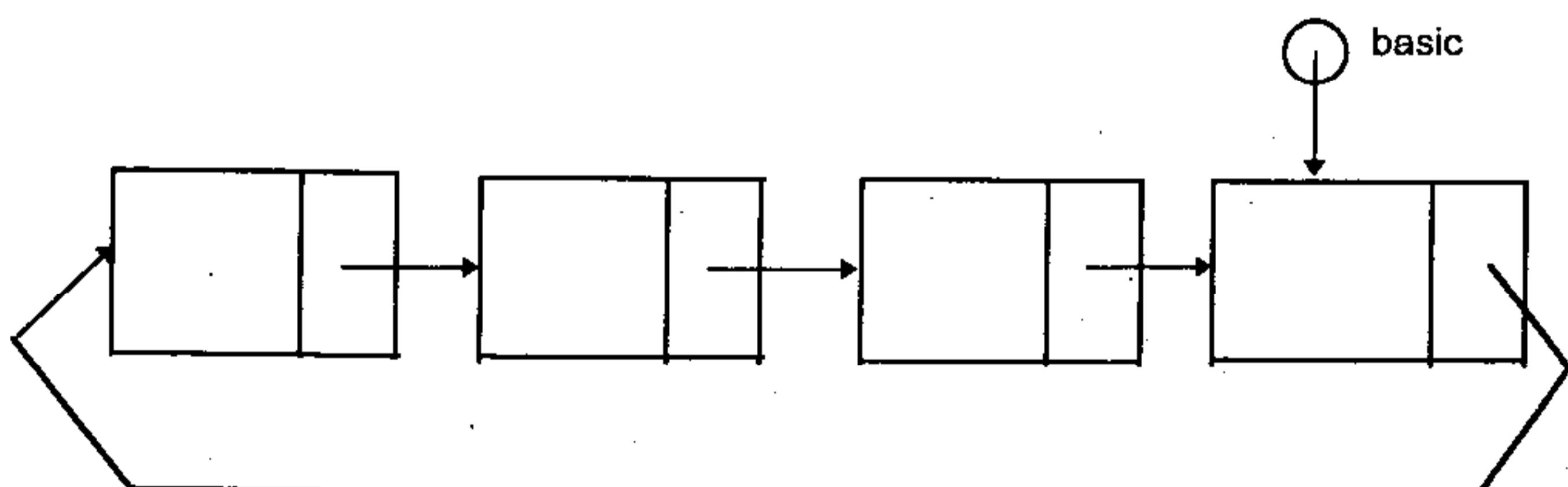
tình trạng danh sách đầy, trừ khi không gian nhớ để cấp phát không còn nữa. Tuy nhiên nó cũng tiêu tốn bộ nhớ cho các con trỏ ở mỗi tế bào.

2. Trong cách biểu diễn danh sách bối mảng, các phép toán truy cập đến mỗi phần tử của danh sách, xác định độ dài của danh sách ... được thực hiện trong thời gian hằng. Trong khi đó các phép toán xen vào và loại bỏ đòi hỏi thời gian tỉ lệ với độ dài của danh sách. Đối với danh sách liên kết, các phép toán xen vào và loại bỏ lại được thực hiện trong thời gian hằng, còn các phép toán khác lại cần thời gian tuyến tính. Do đó, trong áp dụng của mình, ta cần xét xem phép toán nào trên danh sách được sử dụng nhiều nhất, để lựa chọn phương pháp biểu diễn cho thích hợp.

### 3.5. CÁC DẠNG DANH SÁCH LIÊN KẾT KHÁC

#### 3.5.1. Danh sách vòng tròn

Danh sách liên kết vòng tròn (gọi tắt là danh sách vòng tròn) là danh sách mà con trỏ của thành phần cuối cùng của danh sách không bằng nil mà trỏ đến thành phần đầu tiên của danh sách, tạo thành một vòng tròn (xem hình 3.5). Đặc điểm của danh sách vòng tròn là các thành phần trong danh sách đều bình đẳng, mỗi thành phần đều có thành phần đi sau. Xuất phát từ một thành phần bất kỳ ta có thể truy cập đến thành phần bất kỳ khác trong danh sách.



Hình 3.5. Danh sách vòng tròn.

Tế bào tạo nên danh sách vòng tròn có cấu trúc như trong danh sách liên kết. Chúng ta sử dụng một con trỏ basic trỏ tới một thành phần bất kỳ trong danh sách.

```

type pointer = ^Cell ;
Cell = record
    infor : Item ;
    next : pointer ;
end ;
var basic : pointer ;

```

Trong các áp dụng, chúng ta thường sử dụng danh sách vòng tròn có dạng như trong hình 3.5. Ở đó, ta phân biệt một thành phần bên phải (được trả bởi basic) và một thành phần bên trái của danh sách (được trả bởi basic^.next). Đối với danh sách vòng tròn, ta thường sử dụng ba phép toán quan trọng nhất sau đây :

1. Xen vào bên trái danh sách (InsertLeft) một thành phần mới.
2. Xen vào bên phải danh sách (InsertRight) một thành phần mới.
3. Loại bỏ thành phần bên trái danh sách (DeleteLeft).

Sau đây ta sẽ mô tả các thủ tục thực hiện các phép toán trên. Việc xen vào bên trái danh sách một thành phần mới với infor là x được thực hiện bởi thủ tục sau:

```

procedure InsertLeft(var basic : pointer ; x : Item ) ;
var
    P : pointer ;
begin
    new (P) ;
    P^.infor := x ;
    if basic = nil then
        begin
            basic := P ;
            basic^.next := basic ;
        end
    else
        begin
            P^.next := basic^.next ;
            basic^.next := P ;
        end ;
    end ;

```

Việc xen vào bên phải danh sách được tiến hành như sau. Ta thêm thành phần mới vào bên trái, sau đó cho con trỏ basic trở vào thành phần mới được thêm vào này.

```
procedure InsertRight (var basic : pointer ; x : Item) ;
begin
    InsertLeft (basic, x) ;
    basic := basic^.next ;
end ;
```

Sau đây là thủ tục loại bỏ thành phần bên trái danh sách, tham biến x ghi lại các thông tin của thành phần bị loại bỏ.

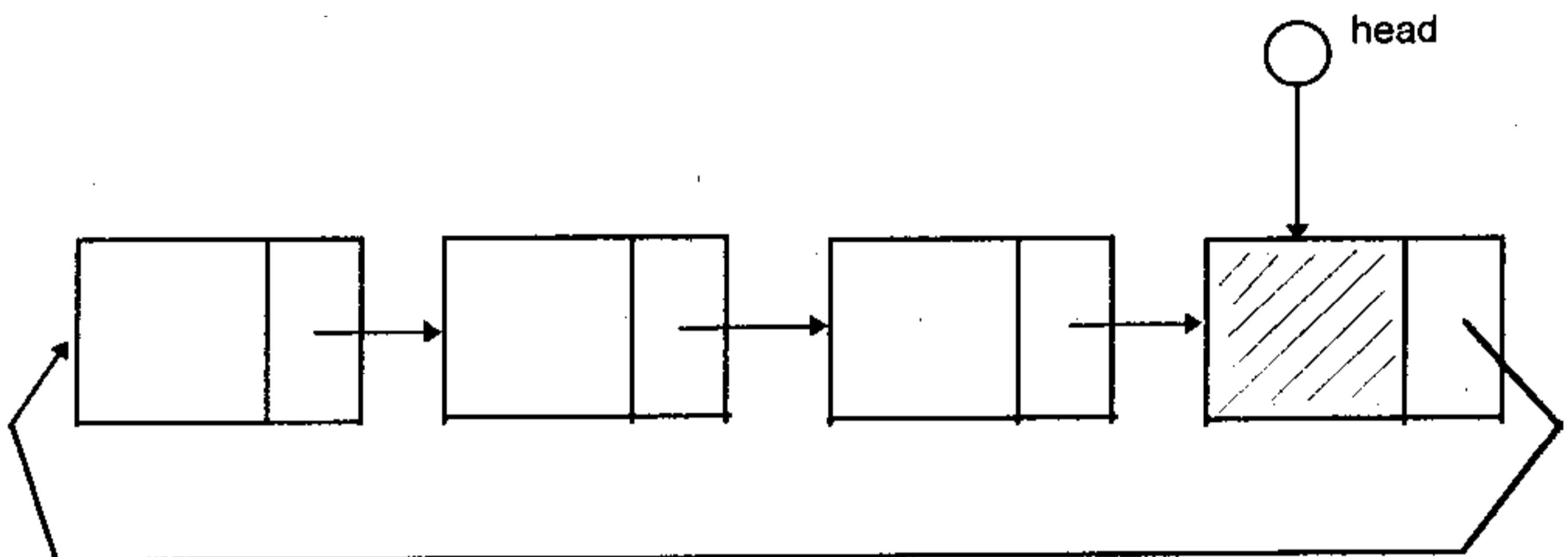
```
procedure DeleteLeft (var basic : pointer ; var x : Item) ;
var
    P : pointer ;
begin
    if basic < > nil then
        begin
            P := basic^.next ;
            x := P^.infor ;
            if basic^.next = basic then basic := nil
            else basic^.next := P^.next ;
            dispose (P);
        end ;
    end ;
```

Một điều đặc biệt nữa của danh sách vòng tròn là ở chỗ, ta có thể sử dụng nó như một stack (với các phép toán InsertLeft và DeleteLeft), hoặc có thể sử dụng nó như một hàng (với các phép toán InsertRight và DeleteLeft). Stack và hàng sẽ được nghiên cứu kỹ trong các mục sau.

Đối với danh sách vòng tròn, một số phép toán khác trên danh sách được thực hiện rất dễ dàng. Chẳng hạn, để nối hai danh sách vòng tròn

base1 và base2 thành một danh sách base1, ta chỉ cần trao đổi các con trỏ  $base1^.next$  và  $base2^.next$ .

Trong nhiều áp dụng, để thuận tiện cho các thao tác với danh sách vòng tròn, ta đưa thêm vào danh sách một thành phần đặc biệt (được gọi là đầu của danh sách). Đầu danh sách chứa các giá trị đặc biệt để phân biệt với các thành phần khác của danh sách (xem hình 3.6). Một ưu điểm của danh sách vòng tròn có đầu, là nó không bao giờ rỗng.



**Hình 3.6.** Danh sách vòng tròn có đầu.

Trong mục 3.5, chúng ta sẽ đưa ra một ứng dụng của danh sách vòng tròn có đầu, ở đó nó được sử dụng để biểu diễn các đa thức.

### 3.5.2. Danh sách hai liên kết

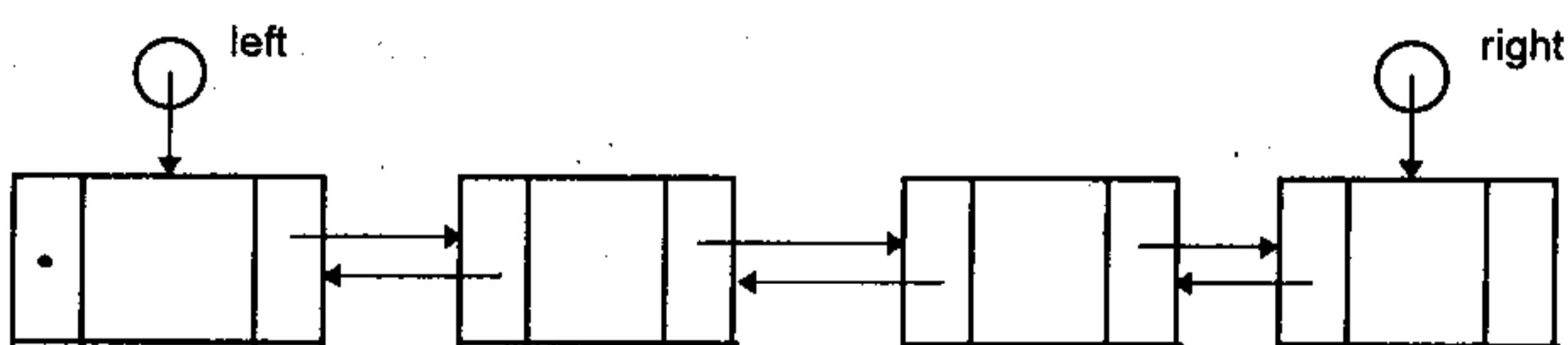
Khi làm việc với danh sách, có những xử lý trên mỗi thành phần của danh sách lại liên quan đến cả thành phần đi trước và thành phần đi sau. Trong các trường hợp như thế, để thuận tiện, người ta đưa vào mỗi thành phần của danh sách hai con trỏ : `nextleft` trỏ đến thành phần bên trái và `nextright` trỏ đến thành phần bên phải. Khi đó chúng ta có một danh sách hai liên kết. Chúng ta cần đến hai con trỏ : `left` trỏ đến thành phần ngoài cùng bên trái và `right` trỏ đến thành phần ngoài cùng bên bên phải danh sách (xem hình 3.7).

Ta có thể khai báo cấu trúc dữ liệu danh sách hai liên kết như sau :

```

type pointer = ^Cell ;
Cell = record
    infor : Item ;
    nextleft, nextright : pointer ;
end ;
List = record
    left, right : pointer ;
end ;

```



**Hình 3.7.** Danh sách hai liên kết.

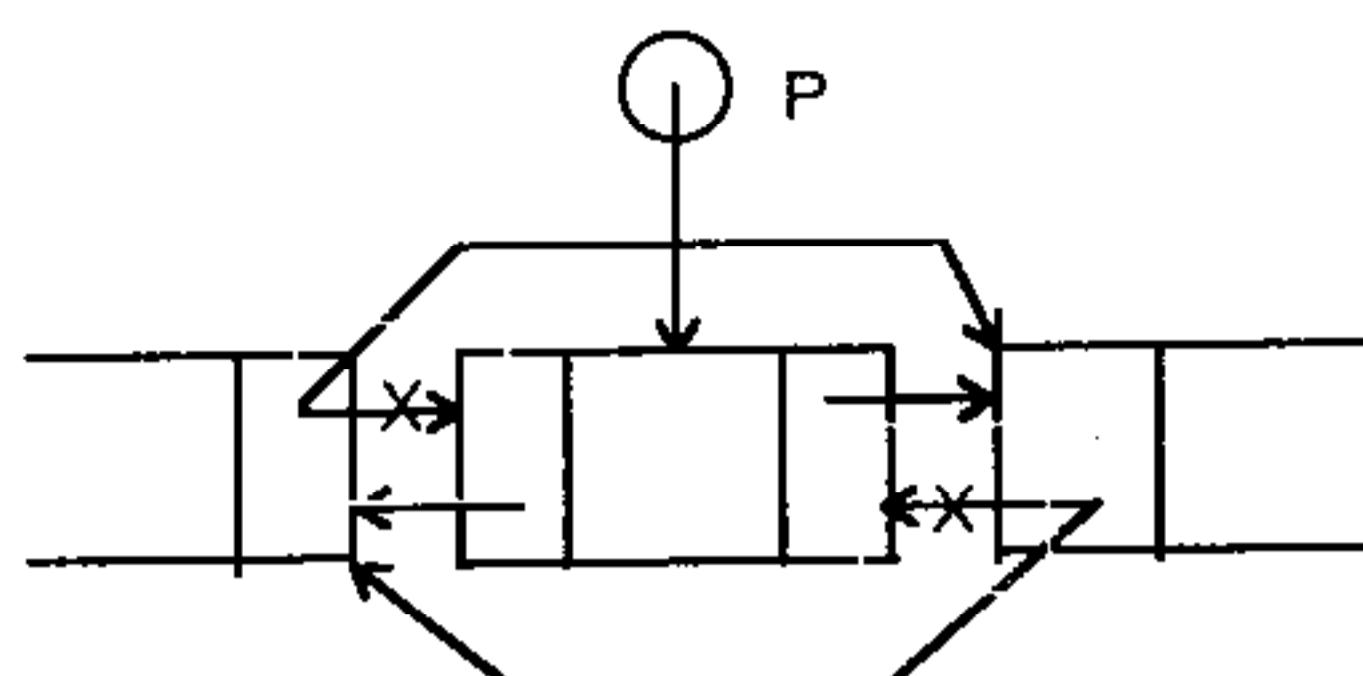
Việc cài đặt danh sách hai liên kết, tất nhiên tiêu tốn nhiều bộ nhớ hơn danh sách một liên kết. Song bù lại, danh sách hai liên kết có những ưu điểm mà danh sách một liên kết không có: khi xem xét một danh sách hai liên kết ta có thể tiến lên trước hoặc lùi lại sau.

Các phép toán trên danh sách hai liên kết được thực hiện dễ dàng hơn danh sách một liên kết. Chẳng hạn, khi thực hiện phép toán loại bỏ, với danh sách một liên kết, ta không thể thực hiện được nếu không biết thành phần đi trước thành phần cần loại bỏ. Trong khi đó, ta có thể tiến hành dễ dàng phép loại bỏ trên danh sách hai liên kết. Hình 3.8 minh họa các thao tác để loại bỏ thành phần P trong danh sách hai liên kết. Ta chỉ cần thực hiện các phép gán sau.

```

Q := P^.nextleft ;
Q^.nextright := P^.nextright ;
Q := P^.nextright ;
Q^.nextleft := P^.nextleft ;
dispose (P) ;

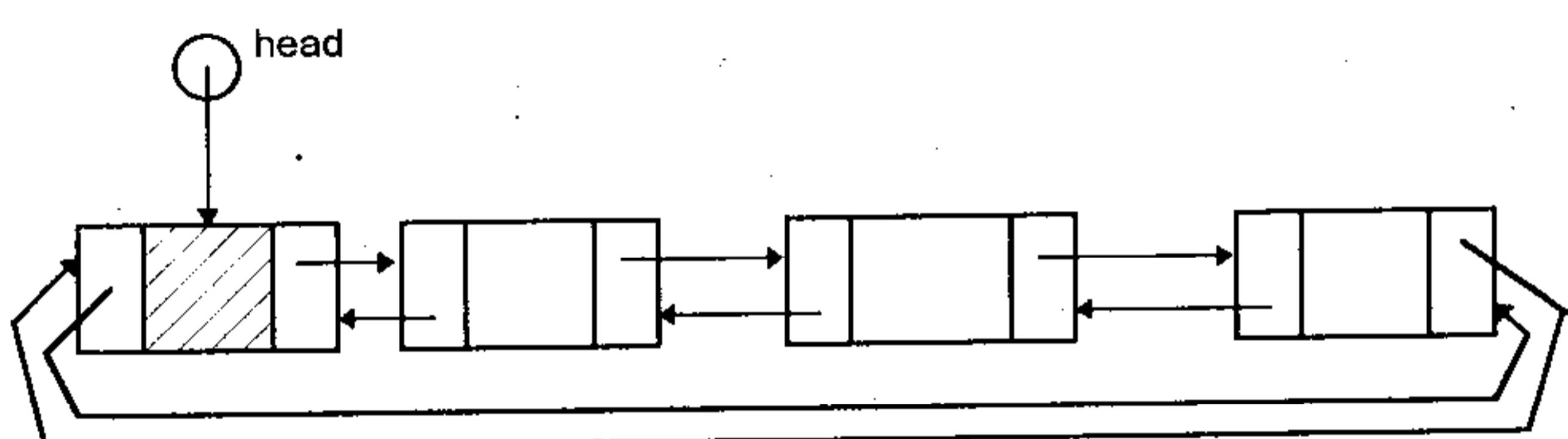
```



**Hình 3.8.** Loại thành phần P của danh sách hai liên kết.

Bạn đọc có thể tự mình viết các thủ tục thực hiện các phép toán trên danh sách hai liên kết (bài tập).

Trong các ứng dụng, người ta ưa dùng các danh sách hai liên kết vòng tròn có đầu (xem hình 3.9). Với danh sách loại này, ta có tất cả các ưu điểm của danh sách vòng tròn và danh sách hai liên kết.



**Hình 3.9.** Danh sách hai liên kết vòng tròn

### 3.6. ỨNG DỤNG DANH SÁCH: CÁC PHÉP TÍNH SỐ HỌC TRÊN ĐA THỨC

Trong mục này ta sẽ xét các phép tính số học (cộng, trừ, nhân, chia) đa thức một ẩn. Các đa thức một ẩn là các biểu thức dạng

$$3x^5 - x^3 + 5x^2 + 6 \quad (1)$$

Mỗi hạng thức của đa thức được đặc trưng bởi hệ số (coef) và số mũ của x (exp). Giả sử các hạng thức trong đa thức được sắp xếp theo thứ tự giảm dần của số mũ, như trong đa thức (1). Rõ ràng ta có thể nhìn nhận đa thức như một danh sách tuyến tính các hạng thức. Khi ta thực hiện các phép toán trên các đa thức ta sẽ nhận được các đa thức có bậc không thể đoán trước được. (bậc của đa thức là số mũ cao nhất của các hạng thức trong đa thức). Ngay cả với các đa thức có bậc xác định thì số các

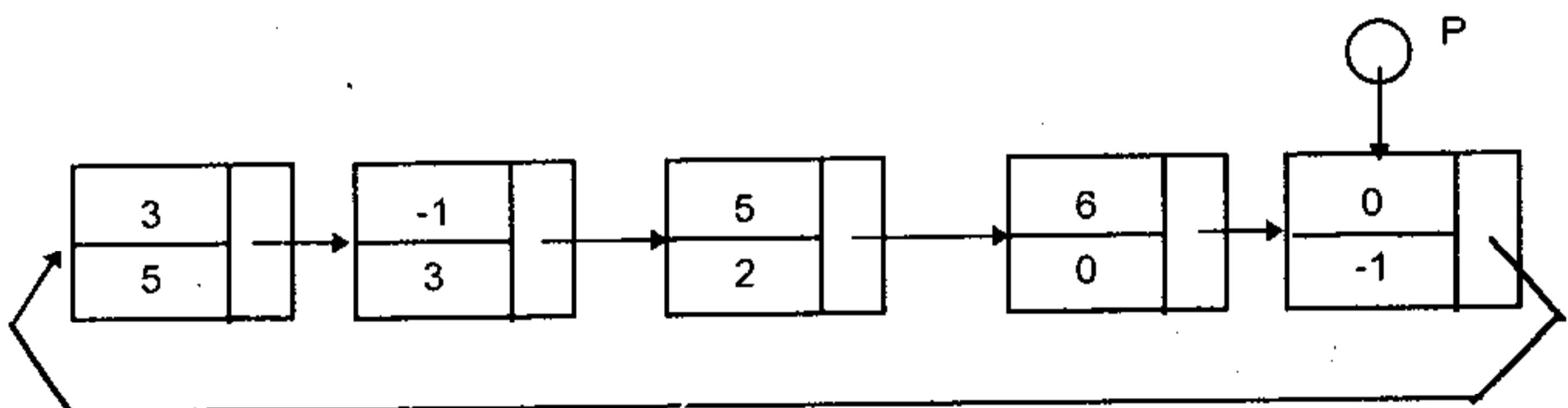
hạng thức của nó cũng biến đổi rất nhiều từ một đa thức này đến một đa thức khác. Do đó phương pháp tốt nhất là biểu diễn đa thức dưới dạng một danh sách liên kết. Thành phần của danh sách này là bản ghi gồm ba trường : coef chỉ hệ số, exp chỉ số mũ của x và con trỏ để trỏ tới thành phần đi sau. Ta có thể mô tả cấu trúc dữ liệu biểu diễn một hạng thức của đa thức như sau:

```

type pointer = ^Term;
Term = record
    coef: real;
    exp: integer;
    next: pointer;
end;

```

Vì những ưu điểm của danh sách vòng tròn có đầu (không phải kiểm tra danh sách rỗng, mọi thành phần đều có thành phần đi sau), ta sẽ chọn danh sách vòng tròn có đầu để biểu diễn đa thức. Với cách chọn này việc thực hiện các phép toán đa thức sẽ rất gọn. Đầu của danh sách là thành phần đặc biệt có  $\text{exp} = -1$ . Chẳng hạn, hình 3.10 minh họa danh sách biểu diễn đa thức (1).



Hình 3.10. Cấu trúc dữ liệu biểu diễn đa thức (1).

Sau đây ta sẽ xét phép cộng đa thức P với đa thức Q. Con trỏ P (Q) trỏ đến đầu danh sách vòng tròn biểu diễn đa thức P (Q). Để thực hiện phép cộng đa thức P với đa thức Q, ta sẽ giữ nguyên danh sách P và thay đổi danh sách Q (xen vào, loại bỏ hay thay đổi trường coef) để nó trở thành danh sách biểu diễn tổng của hai đa thức. Một con trỏ P1 chạy

trên danh sách P, hai con trỏ Q1, Q2 chạy cách nhau một bước ( $Q2 = Q1^. Next$ ) trên danh sách Q. So sánh số mũ của P1 với số mũ của Q2. Có ba khả năng:

1. Nếu  $P1^. exp > Q2^. exp$  thì ta xen thành phần P1 vào danh sách Q trước thành phần Q2. Cho P1 chạy tới thành phần sau trong danh sách.

2. Nếu  $P1^. exp = Q2^. exp$  thì ta thêm  $P1^. coef$  vào  $Q2^. coef$ . Sau khi thêm  $Q2^. coef = 0$  thì ta loại bỏ thành phần Q2 khỏi danh sách Q. Sau đó ta cho P1 và Q1, Q2 chạy tới các thành phần tiếp theo trong danh sách P và Q.

3. Nếu  $P1^. exp < Q2^. exp$  thì ta cho Q1, Q2 chạy lên một bước.

Quá trình trên sẽ lặp lại cho tới khi P1 hoặc Q2 đi hết danh sách. Trong trường hợp Q2 đi hết danh sách Q còn P1 còn ở giữa danh sách P thì chuyển các thành phần còn lại của danh sách P vào danh sách Q. Ta có chương trình sau.

```

program AddPoly;
type pointer = ^ Term;
    Term = record
        coef : real;
        exp : integer;
        next : pointer;
    end;
var
    P, P1: pointer;
    Q, Q1, Q2 : pointer;

procedure ReadPoly (var P : pointer);
    { Tạo ra danh sách vòng tròn biểu diễn đa thức}
var
    P1, P2 : pointer;
    Traloi : char;
begin
    new (P);
    P^. coef := 0;

```

```

P^. exp := -1; {Tạo ra đầu danh sách}
P1 := P;
readln (Traloi);
while Traloi = 'C' do
begin
    new (P2);
    readln (P2^. coef, P2^. exp);
    P1^. next := P2;
    P1 := P2;
    readln (traloi);
end;
P1^. next := P;
end;

procedure Insert (P1: pointer; var Q1, Q2 : pointer);
{Xen thành phần P1^ vào giữa hai thành phần Q1^, Q2^ trong danh sách Q}
begin
    P1^. next := Q2;
    Q1^. next := P1;
    Q1 := P1;
end;

procedure Delete (var Q1, Q2 : pointer);
{Xoá thành phần Q2^ khỏi danh sách Q, Q2 = Q1^. next}
begin
    Q1^. next := Q2^. next;
    Q2 := Q1^. next;
end;

procedure WritePoly ( Q : pointer);
begin
    Q := Q^. next;
    if Q^. exp = -1 then writeln ('Q = 0')
    else
        while Q^. exp > -1 do
begin

```

```

Write (QA. coef, 'xA', QA. exp) ;
QA := QA. next ;
if QA. exp > -1 then write ('+');
end ;
end ;
begin {chương trình chính}
  ReadPoly (P) ;
  ReadPoly (Q) ;
  PA := PA. next ;
  Q1A := QA ;
  Q2A := Q1A. next ;
  while (PA. exp > -1) and ( Q2A. exp > -1) do
    begin
      if PA. exp > Q2A. exp then
        begin
          P1A := PA ;
          PA := PA. next ;
          Insert (P1A, Q1A, Q2A) ;
        end
      else if PA. exp = Q2A. exp then
        begin
          Q2A. coef := Q2A. coef + PA. coef ;
          if Q2A. coef = 0 then Delete (Q1A, Q2A)
          else begin
            Q1A := Q2A ;
            Q2A := Q1A. next ;
          end ;
          PA := PA. next ;
        end
      else begin
        Q1A := Q2A ;
        Q2A := Q1A. next ;
      end ;
    end ;
  { hết vòng lặp while. Nếu Q2A. exp = -1 còn PA. exp > -1 thì chuyển các hạng thức
  còn lại của đa thức P vào cuối đa thức Q}

```

```

while P^.exp > -1 do
  begin
    P1 := P ;
    P := P^.next ;
    Insert (P1, Q1, Q2);
  end ;
  WritePoly (Q) ;
end ;

```

## 3.7. NGĂN XẾP

### 3.7.1. Ngăn xếp

Trong mục này chúng ta sẽ xét ngăn xếp ( stack ), một dạng hạn chế của danh sách, trong đó phép toán xen một phần tử mới vào danh sách và loại bỏ một phần tử khỏi danh sách, chỉ được phép thực hiện ở một đầu của danh sách. Đầu này được gọi là *dỉnh* của ngăn xếp. Ta có thể hình dung ngăn xếp như một chồng đĩa, ta chỉ có thể đặt thêm đĩa mới lên trên đĩa trên cùng, hoặc lấy đĩa trên cùng ra khỏi chồng. Như vậy chiếc đĩa đặt vào chồng sau cùng, khi lấy ra sẽ được lấy ra đầu tiên. Vì thế, ngăn xếp còn được gọi là danh sách LIFO (viết tắt của Last In First Out, nghĩa là, cái vào sau cùng ra đầu tiên).

Nói chung, với một mô hình dữ liệu (chẳng hạn, mô hình dữ liệu danh sách, cây, tập hợp, ...), lớp các phép toán có thể thực hiện trên mô hình rất đa dạng và phong phú. Song trong các ứng dụng chỉ có một số nhóm phép toán được sử dụng thường xuyên. Khi xét một mô hình dữ liệu với một tập hợp xác định các phép toán được phép thực hiện, ta có một *kiểu dữ liệu trừu tượng* (abstract data type). Như vậy ngăn xếp là một kiểu dữ liệu trừu tượng dựa trên mô hình dữ liệu danh sách, với các phép toán sau đây.

Giả sử S là ngăn xếp các phần tử của nó có kiểu Item và x là một phần tử cùng kiểu với các phần tử của ngăn xếp.

1. Khởi tạo ngăn xếp rỗng (ngăn xếp không chứa phần tử nào)

**procedure** Initialize (**var** S : stack);

## 2. Kiểm tra ngăn xếp rỗng

**function** Empty (**var** S : stack) : **boolean** ;

Emty nhận giá trị true nếu S rỗng và false nếu S không rỗng.

## 3. Kiểm tra ngăn xếp đầy

**function** Full (**var** S : stack) : **boolean** ;

Full nhận giá trị true nếu S đầy và false nếu không.

## 4. Thêm một phần tử mới x vào đỉnh của ngăn xếp

**procedure** Push (x : Item, **var** S : stack);

5. Loại phần tử ở đỉnh của ngăn xếp và gán giá trị của phần tử này cho x.

**procedure** Pop (**var** S : stack ; **var** x : Item) ;

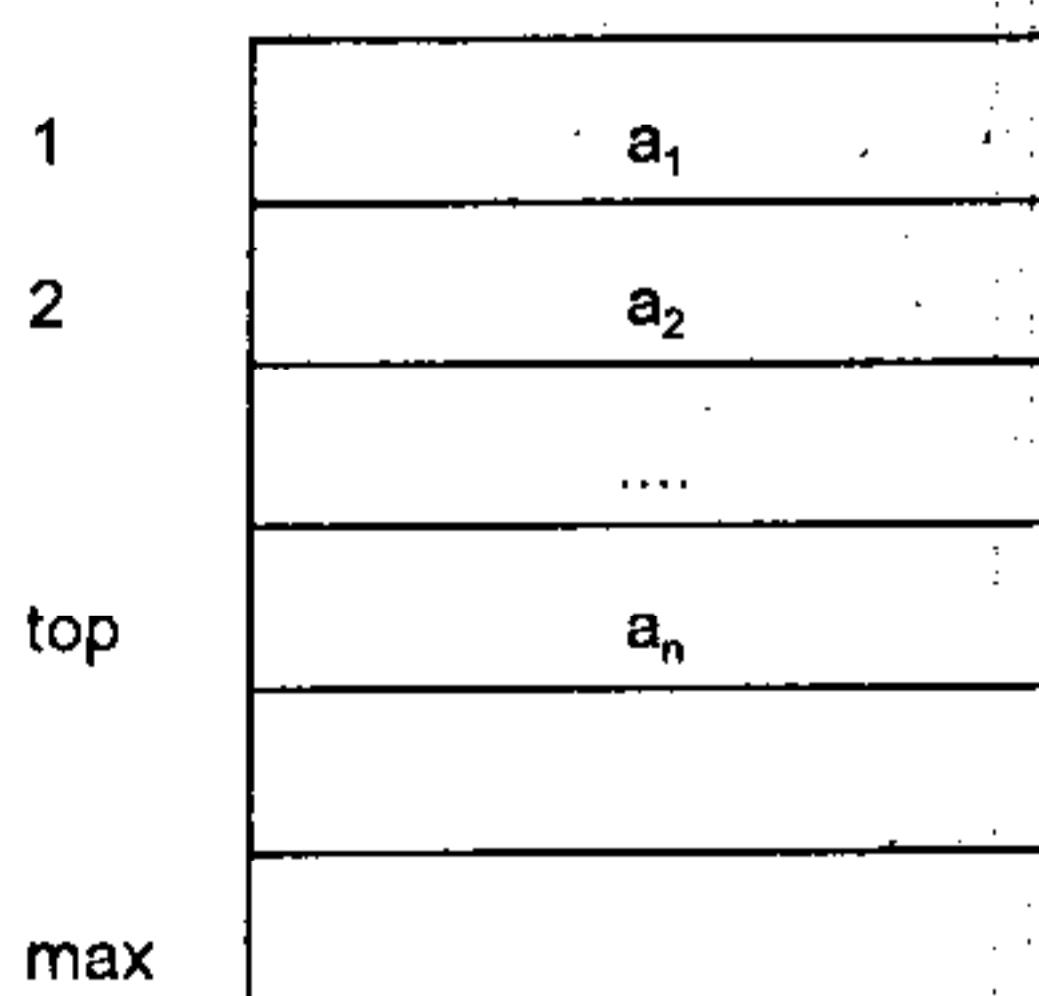
Chú ý rằng, phép toán Push chỉ được thực hiện nếu ngăn xếp không đầy, còn phép toán Pop chỉ được thực hiện nếu ngăn xếp không rỗng.

Ví dụ : Nếu S là ngăn xếp,  $S = (a_1, a_2, \dots, a_n)$  và đỉnh của ngăn xếp là đầu bên phải. Khi đó thực hiện Push (x, S) ta được  $S = (a_1, \dots, a_n, x)$ . Nếu  $n \geq 1$  thì khi thực hiện Pop (S, x) ta được  $s = (a_1, a_2, \dots, a_{n-1})$  và  $x = a_n$ .

### 3.7.2. Cài đặt ngăn xếp bởi mảng

Chúng ta có thể sử dụng các phương pháp cài đặt danh sách để cài đặt ngăn xếp. Trước hết ta cài đặt ngăn xếp bởi mảng.

Giả sử độ dài tối đa của ngăn xếp là max, các phần tử của ngăn xếp có kiểu dữ liệu là Item, đỉnh của ngăn xếp được chỉ bởi biến top. Khi đó ngăn xếp  $S = (a_1, a_2, \dots, a_n)$  được biểu diễn bởi mảng như trong hình 3.11.



Hình 3.11. Mảng biểu diễn ngăn xếp.

Cấu trúc dữ liệu để biểu diễn ngăn xếp có thể được khai báo như sau

```
const max = N;
type Stack = record
    top : 0 .. max;
    element : array [1..max] of Item ;
end;
var S : Stack;
```

Với cách cài đặt này, S là ngăn xếp rỗng, nếu S.top = 0, và nó sẽ đầy nếu S.top = max.

Sau đây là các thủ tục và hàm thực hiện các phép toán trên ngăn xếp.

```
procedure Initialize ( S : Stack );
begin
    S.top := 0;
end ;

function Empty (var S : Stack) : boolean ;
begin
    Empty := (S.top = 0);
end ;

function Full (var S : Stack) : boolean ;
begin
    Full := (S.top = max);
end ;

procedure Push (x : Item ; var S : Stack ; var OK : boolean) ;
begin
    with S do
        if Full(S) then OK := false
        else begin
            top := top + 1;
```

```

    element [top] := x ;
    OK := true;
end ;
end ;

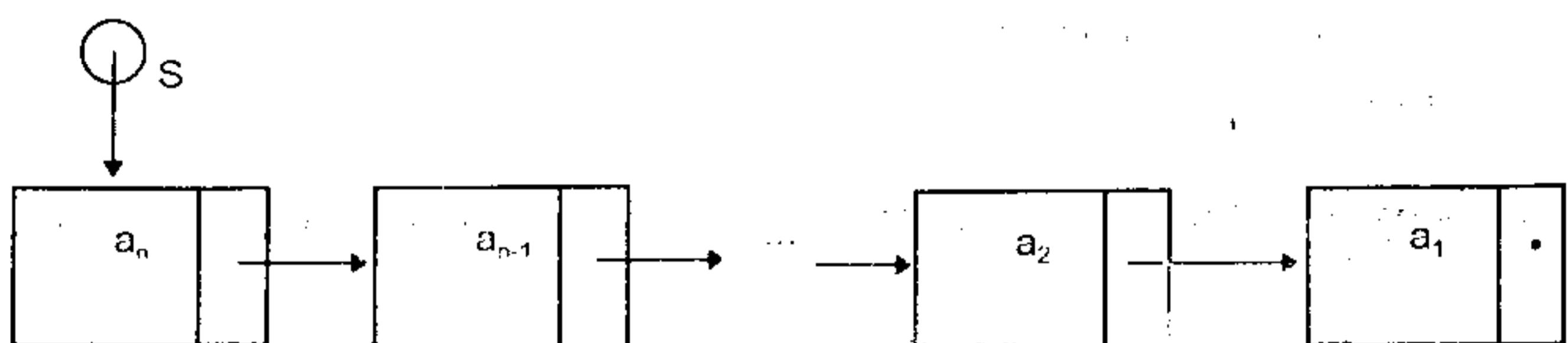
procedure Pop (var S : Stack ; var x : Item ; var OK : boolean);
begin
  with S do
    if Empty (S) then OK := false
    else begin
      x := element [top];
      top := top - 1;
      OK := true;
    end ;
end ;

```

Trong các thủ tục Push và Pop, chúng ta đã đưa vào tham biến OK để ghi lại tình trạng khi thực hiện phép toán, nó nhận giá trị true khi phép toán thực hiện thành công và false nếu thất bại.

### 3.7.3. Cài đặt ngăn xếp bởi danh sách liên kết

Chúng ta có thể cài đặt ngăn xếp bởi danh sách liên kết như chúng ta đã làm đối với danh sách. Đỉnh của ngăn xếp là đầu của danh sách liên kết. Ta sử dụng con trỏ S trỏ đến đỉnh ngăn xếp. Hình 3.12 minh họa danh sách liên kết biểu diễn ngăn xếp ( $a_1, a_2, \dots, a_n$ ) với đỉnh là  $a_n$ .



*Hint: 3 / 2*

Ta có thể khai báo cấu trúc dữ liệu danh sách liên kết biểu diễn ngắn xếp như sau

```

type     Stack = ^Cell;
          Cell = record
                    Infor : Item ;
                    next : Stack ;
            end ;
var      S : Stack ;

```

Trong cách cài đặt này, S là ngắn xếp rỗng, nếu S = nil. Chúng ta giả thiết rằng, việc cấp phát bộ nhớ cho các biến động (thủ tục new trong Pascal) luôn luôn được thực hiện. Do đó hàm Full luôn luôn có giá trị false, và phép toán Push không bao giờ thất bại.

Sau đây là các thủ tục và hàm thực hiện các phép toán trên ngắn xếp được cài đặt bởi danh sách liên kết.

```

procedure   Initialize (var S : Stack) ;
begin
    S := nil;
end ;

function   Empty (var S : Stack) : boolean ;
begin
    if S = nil then Empty := true
        else Empty := false ;
end ;

function   Full (var S : stack) : boolean ;
begin
    Full := false;
end ;

procedure   Push (x : Item ; var S : Stack ; var OK : boolean);
var         P : ^Cell ;
begin

```

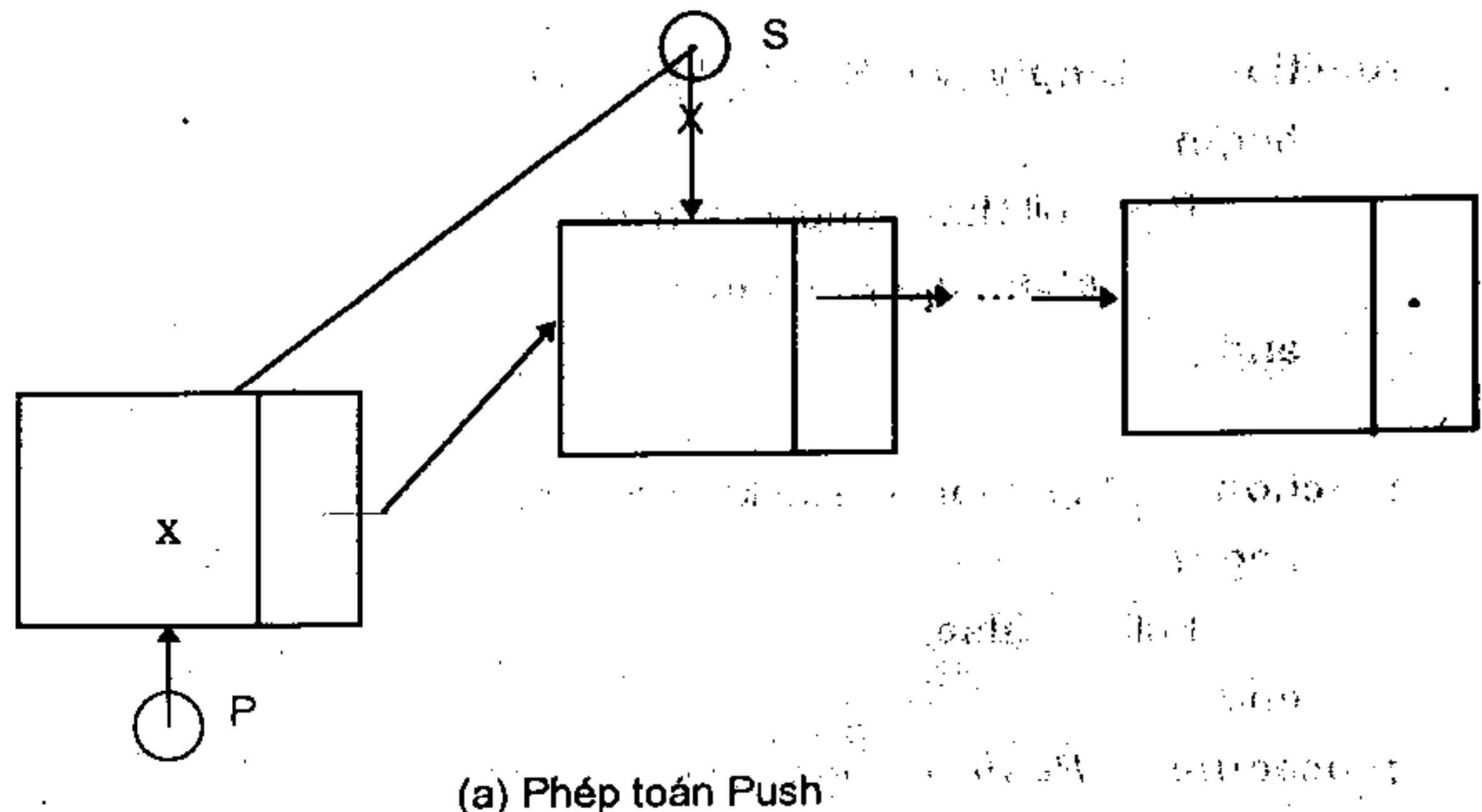
```

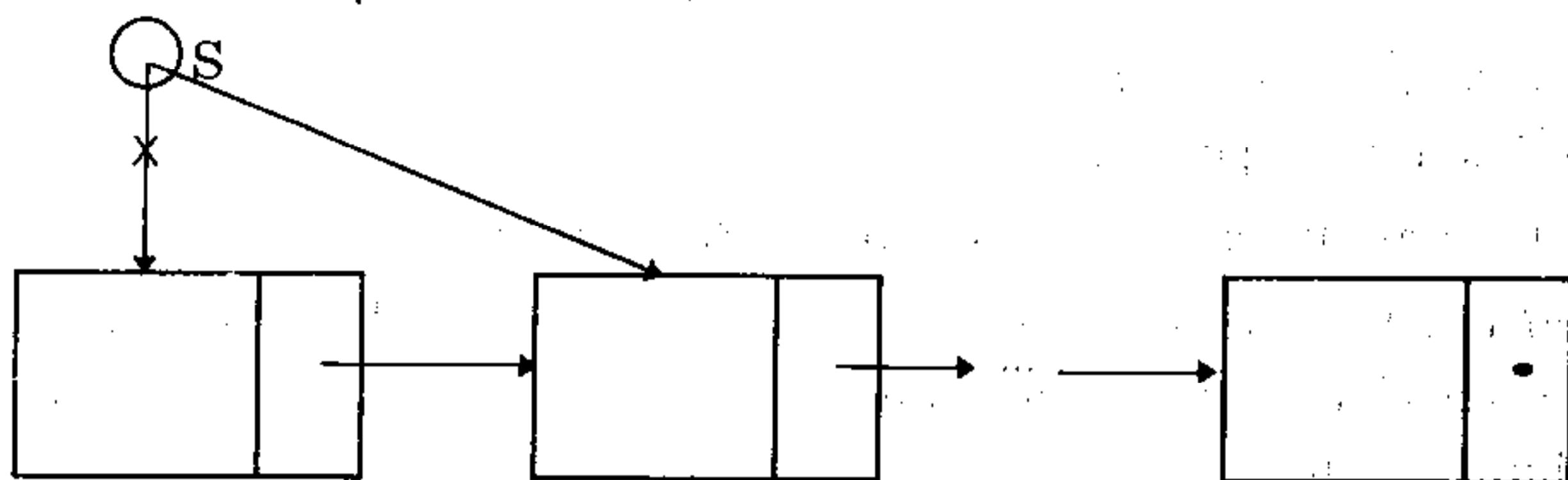
new (P) ;
P^.infor := x ;
P^.next := S ;
S := P ;
OK := true;
end ;

procedure Pop (var S : Stack ; var x: Item ; var OK: boolean) ;
var P : ^Cell ;
begin
  if S = nil then OK := false
  else begin
    P := S ;
    x := S^.infor ;
    S := S^.next ;
    OK := true ;
    dispose (P) ;
  end ;
end ;

```

Các thao tác thực hiện các phép toán Push và Pop trên ngăn xếp được minh họa trong hình 3.13a và 3.13b.





(b) Phép toán Pop

**Hình 3.13.** Các phép toán Push và Pop trên ngăn xếp.

### 3.8. GIÁ TRỊ CỦA MỘT BIỂU THỨC

Trong mục này, chúng ta sẽ trình bày một ứng dụng của ngăn xếp: xác định giá trị của một biểu thức.

Trong các chương trình ta thường viết các lệnh gán

$$X := <\text{biểu thức}>$$

trong đó,  $X$  phải là một biến (số học hoặc logic). Khi thực hiện chương trình, gặp các lệnh gán, máy tính cần phải xác định giá trị của biến và gán kết quả cho biến  $X$ . Do đó vấn đề đặt ra là, làm thế nào thiết kế được thuật toán xác định giá trị của biến  $X$ .

Ta sẽ xét các biểu thức số học. Một cách không hình thức, biểu thức số học là một dãy các toán hạng (hằng, biến hoặc hàm) nối với nhau bởi các phép toán số học. Trong các biểu thức có thể chứa các dấu ngoặc tròn. Để đơn giản ta chỉ xét các biểu thức số học chứa các phép toán hai toán hạng  $+$ ,  $-$ ,  $*$  và  $/$ . Khi tính giá trị của biểu thức, các phép toán trong ngoặc được thực hiện trước, rồi đến các phép toán  $*$  và  $/$ , sau đó đến các phép toán  $+$  và  $-$ . Trong cùng mức ưu tiên, các phép toán được thực hiện từ trái sang phải. Chẳng hạn, xét biểu thức

$$5 + 8 / (3 + 1) * 3$$

Giá trị của biểu thức này được tính như sau:

$$5 + 8/(3+1)*3 = 5+8/4 * 3 = 5+2 * 3 = 5+6 = 11$$

Sau đây ta đưa ra thuật toán xác định giá trị của một biểu thức số học. Thuật toán này gồm hai giai đoạn.

1. Chuyển biểu thức số học thông thường (dạng infix) sang biểu thức số học Ba lan postfix.

2. Tính giá trị của biểu thức số học Balan postfix

Trước hết ta cần xác định thế nào là biểu thức số học Balan postfix. Trong cách viết thông thường, phép toán được đặt giữa hai toán hạng, chẳng hạn,  $a + b$ ,  $a * b$ . Còn trong cách viết Balan, phép toán được đặt sau các toán hạng. Chẳng hạn, các biểu thức  $a + b$ ,  $a * b$  trong cách viết Balan được viết là  $ab +$ ,  $ab *$ . Một số ví dụ khác.

Biểu thức thông thường

$a * b / c$

$a * (b + c) - d / e$

Biểu thức Balan

$ab * c /$

$abc + * de / -$

Cần lưu ý rằng, biểu thức số học Balan không chứa các dấu ngoặc, nó chỉ gồm các toán hạng và các dấu phép toán.

Sau đây ta sẽ trình bày thuật toán xác định giá trị của biểu thức số học Balan. Trong thuật toán này, ta sẽ sử dụng một ngăn xếp S để lưu giữ các toán hạng và các kết quả tính toán trung gian. Thuật toán như sau.

1. Đọc lần lượt các thành phần của biểu thức Balan từ trái sang phải. Nếu gặp toán hạng thì đẩy nó vào ngăn xếp. Nếu gặp phép toán, thì rút hai toán hạng ở đỉnh ngăn xếp ra và thực hiện phép toán này. Kết quả nhận được lại đẩy vào ngăn xếp.

2. Lặp lại quá trình trên cho tới khi toàn bộ biểu thức được đọc qua. Lúc đó đỉnh của ngăn xếp chứa giá trị của biểu thức.

Giả sử E là biểu thức số học Balan nào đó. Ta đưa thêm vào cuối biểu thức ký hiệu # để đánh dấu hết biểu thức. Trong thuật toán tính giá trị của biểu thức E, ta sẽ sử dụng các thủ tục sau.

Thủ tục Read (E,z). Đọc một thành phần của biểu thức E và gán nó cho z. Đầu đọc được chuyển sang phải một vị trí.

Thủ tục Push (x,S). Đẩy x vào đỉnh ngăn xếp S.

Thủ tục Pop(S,x). Loại phần tử ở đỉnh của ngăn xếp và gán nó cho x.

Ta có thể mô tả thuật toán xác định giá trị của biểu thức số học Balan bởi thủ tục sau.

```

procedure      Eval (E : biểu thức) ;
begin
    Read (E,z) ;
    while z < > # do
        begin
            if z là toán hạng then Push (z, S)
            else begin
                Pop (S,y) ; {Rút các toán hạng ở đỉnh stack}
                Pop (S,x) ;
                w := x z y; {Thực hiện phép toán z với các
                                toán hạng x và y }
                Push (w,S);
            end ;
            Read (E,z);
        end ;
    end ;

```

Sau đây chúng ta sẽ thiết kế thuật toán chuyển biểu thức số học thông thường sang biểu thức số học Balan. Khác với thuật toán tính giá trị của biểu thức số học Balan, trong thuật toán này, chúng ta sẽ sử dụng ngăn xếp S để lưu các dấu mở ngoặc (và các dấu phép toán +, -, \*, và /). Ta đưa vào ký hiệu \$ để đánh dấu đáy của ngăn xếp. Khi đỉnh ngăn xếp chứa \$, có nghĩa là ngăn xếp rỗng.

Trên tập hợp các ký hiệu \$, (, +, -, \*, / ta xác định hàm Pri (hàm ưu tiên) như sau :  $\text{Pri} (\$) < \text{Pri} (()) < \text{Pri} (+) = \text{Pri} (-) < \text{Pri} (*) = \text{Pri}(/)$ .

Giả sử ta cần chuyển biểu thức số học thông thường E sang biểu thức số học Balan E1. Ta thêm vào bên phải biểu thức E ký hiệu # để đánh dấu hết biểu thức.

Thuật toán gồm các bước sau :

1. Đọc một thành phần của biểu thức E (Đọc lần lượt từ trái sang phải) Giả sử thành phần được đọc là x.

- 1.1. Nếu x là toán hạng thì viết nó vào bên phải biểu thức E1.
- 1.2. Nếu x là dấu mở ngoặc ( thì đẩy nó vào ngăn xếp
- 1.3. Nếu x là một trong các dấu phép toán +, -, \*, / thì
  - a. Xét phần tử y ở đỉnh ngăn xếp
  - b. Nếu  $Fri(y) \geq Fri(x)$  thì loại y khỏi ngăn xếp, viết y vào bên phải E1 và quay lại bước a)
  - c. Nếu  $Fri(y) < Fri(x)$  thì đẩy x vào ngăn xếp
- 1.4. Nếu x là dấu đóng ngoặc ) thì
  - a. Xét phần tử y ở đỉnh của ngăn xếp
  - b. Nếu y là dấu phép toán thì loại y khỏi ngăn xếp, viết y vào bên phải E1 và quay lại a)
  - c. Nếu y là dấu mở ngoặc ( thì loại nó khỏi ngăn xếp
2. Lặp lại bước 1 cho tới khi toàn bộ biểu thức E được đọc qua.
3. Loại phần tử ở đỉnh ngăn xếp và viết nó vào bên phải E1. Lặp lại bước này cho tới khi ngăn xếp rỗng.

Trong thuật toán ta sử dụng các thủ tục sau.

**Read (E,x).** Đọc một thành phần của biểu thức E và gán cho x

**Write (x,E1).** Viết x vào bên phải biểu thức Balan E1.

**Push (x,S).** Đẩy x vào ngăn xếp

**Pop (S,x).** Loại phần tử ở đỉnh ngăn xếp và gán cho x

Gọi phần tử ở đỉnh của ngăn xếp là top

Chúng ta mô tả thuật toán chuyển biểu thức số học thông thường E sang biểu thức số học Balan E1 bởi thủ tục sau.

**procedure Postfix (E: biểu-thức; var E1 : biểu-thức);**

**begin**

**Push(\$,S);**

**Read (E,x);**

```

while x < > # do

    begin

        if x là toán hạng then Write (x,E1)

        else if x = ( then Push (x,S)

        else if x = ) then

            begin

                while top < > ( do

                    begin

                        Pop(S,y) ;

                        Write (y, E1);

                    end ;

                    Pop (S,y) ;

                end

            else begin

                while Pri(top) >= Pri(x) do

                    begin

                        Pop (S,y) ;

                        Write (y, E1);

                    end ;

                    Push (x,S);

                end ;

                Read (E,x);

            end ; { hết lệnh while x < > # }

            write (#, E1);

        end ;
    
```

Ví dụ : Xét biểu thức :

$$E = a * (b + c) - d \#$$

Kết quả các bước thực hiện thuật toán được cho trong bảng sau

Thành phần trong biểu thức E	Ngăn xếp	Biểu thức Balan E1
	\$	
a	\$	a
*	\$*	a
(	\$*(	a
b	\$*(	ab
+	\$*(+	ab
c	\$*(+	abc
)	\$*(	abc+
-	\$*	abc+
	\$	abc+*
	\$-	abc + *
d	\$-.	abc+*d
#	\$	abc+*d-
		abc + * d- #

### 3.9. HÀNG

#### 3.9.1. Hàng

Một kiểu dữ liệu trùu tượng quan trọng khác được xây dựng trên cơ sở mô hình dữ liệu danh sách là hàng. Hàng là một danh sách với hai phép toán quan trọng nhất là thêm một phần tử mới vào một đầu danh sách (đầu này được gọi là *cuối hàng*) và loại phần tử khỏi danh sách ở một đầu khác (đầu này gọi là *đầu hàng*). Trong đời sống hàng ngày, ta thường xuyên gặp hàng. Chẳng hạn, hàng người chờ đợi được phục vụ (chờ mua vé tàu, chặng hạn). Người ta chỉ có thể đi vào hàng ở cuối hàng, người được phục vụ và ra khỏi hàng là người ở đầu hàng, tức là ai vào hàng trước sẽ được phục vụ trước. Vì vậy, hàng còn được gọi là danh

sách FIFO (viết tắt của First In First Out, nghĩa là, ai vào đầu tiên ra đầu tiên).

Sau đây là tập hợp đầy đủ các phép toán mà ta có thể thực hiện trên hàng.

Giả sử Q là một hàng các đối tượng nào đó có kiểu dữ liệu Item và x là một phần tử cùng kiểu với các đối tượng của hàng.

1. Khởi tạo hàng rỗng.

**procedure** Initialize (**var** Q : Queue) ;

2. Kiểm tra hàng rỗng

**function** Empty (**var** Q : Queue) : **boolean** ;

Empty nhận giá trị true nếu Q rỗng và false nếu không

3. Kiểm tra hàng đầy

**function** Full (**var** Q : Queue) : **boolean** ;

Full nhận giá trị true nếu Q đầy và false nếu không

4. Thêm một phần tử mới x vào cuối hàng

**procedure** AddQueue ( x : Item ; **var** Q : Queue) ;

5. Loại phần tử ở đầu hàng, giá trị của phần tử này được lưu vào x.

**procedure** DeleteQueue (**var** Q : Queue, **var** x : Item)

### 3.9.2. Cài đặt hàng bởi mảng

Ta có thể biểu diễn hàng bởi mảng và sử dụng hai chỉ số front chỉ vị trí đầu hàng và rear chỉ vị trí cuối hàng.

Có thể khai báo cấu trúc dữ liệu biểu diễn hàng như sau

```
const max = N;
```

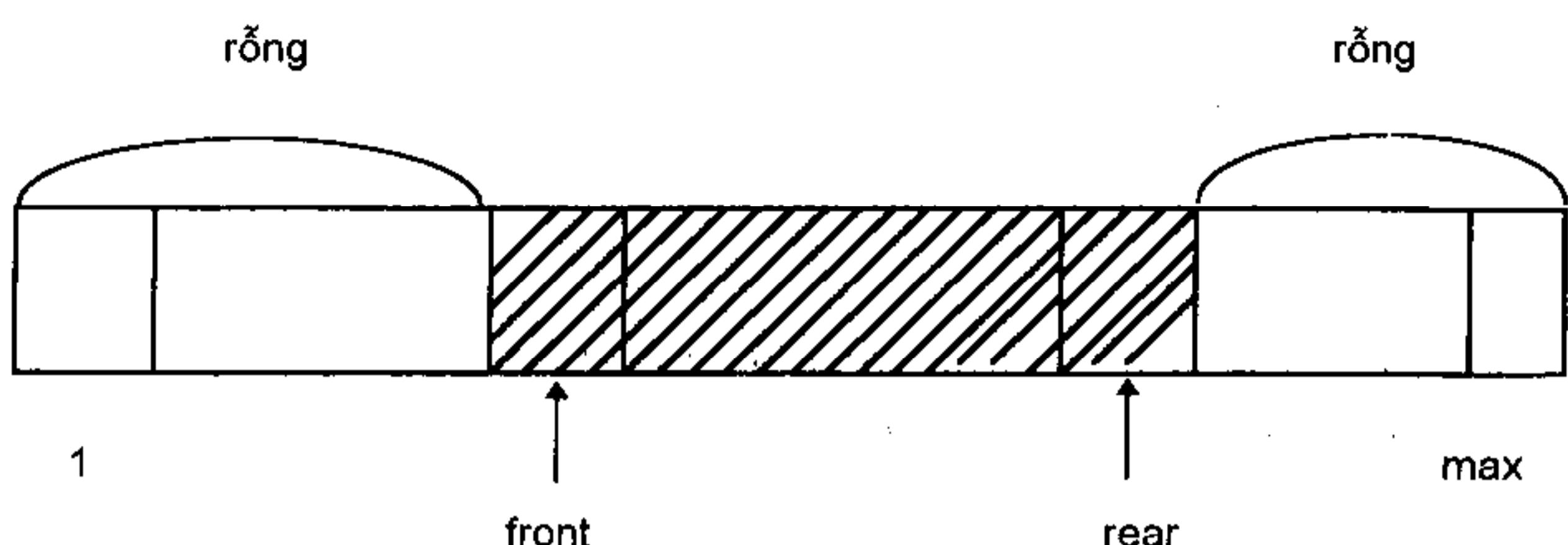
```
type Queue = record
```

```
    front, rear : 0 .. max ;
```

```
    element : array [1..max] of Item ;
```

```
end ;
```

```
var Q : Queue ;
```



**Hình 3.14.** Mảng biểu diễn hàng

Trong cách cài đặt này, hàng rỗng nếu  $\text{rear} = 0$  và hàng đầy nếu  $\text{rear} = \text{max}$ .

Sau đây là các thủ tục và hàm thực hiện các phép toán trên hàng

```

procedure Initialize (var Q : Queue) ;
begin
  with Q do
    begin
      front := 1 ;
      rear := 0 ;
    end ;
end ;

function Empty (var Q : Queue) : boolean ;
begin
  if Q.rear = 0 then Empty := true
  else Empty := false;
end ;

function Full (var Q : Queue) : boolean ;
begin
  if Q.rear = max then Full := true
  else Full := false;
end ;

```

```
procedure AddQueue (x : Item ; var Q:Queue ; var OK : boolean);
```

```
begin
```

```
with Q do
```

```
if rear = max then OK := false
```

```
else begin
```

```
    rear := rear + 1;
```

```
    element [rear] := x;
```

```
    OK := true;
```

```
end ;
```

```
end ;
```

```
procedure DeleteQueue (var Q : Queue; var x : Item; var OK : boolean);
```

```
begin
```

```
with Q do
```

```
if rear = 0 then OK := false
```

```
else begin
```

```
    x := element [front];
```

```
    if front = rear then
```

```
begin
```

```
        front := 1;
```

```
        rear := 0;
```

```
end else front := front + 1 ;
```

```
    OK := true;
```

```
end ;
```

```
end ;
```

Phương pháp cài đặt hàng bởi mảng với hai chỉ số (front chỉ đầu hàng, rear chỉ cuối hàng) có nhược điểm lớn. Nếu phép loại bỏ không thường xuyên làm cho hàng rỗng, thì các chỉ số front và rear sẽ tăng liên tục, nhanh chóng vượt quá cỡ của mảng. Hàng sẽ trở thành đầy, mặc dù các vị trí trống trong mảng có thể vẫn còn nhiều !

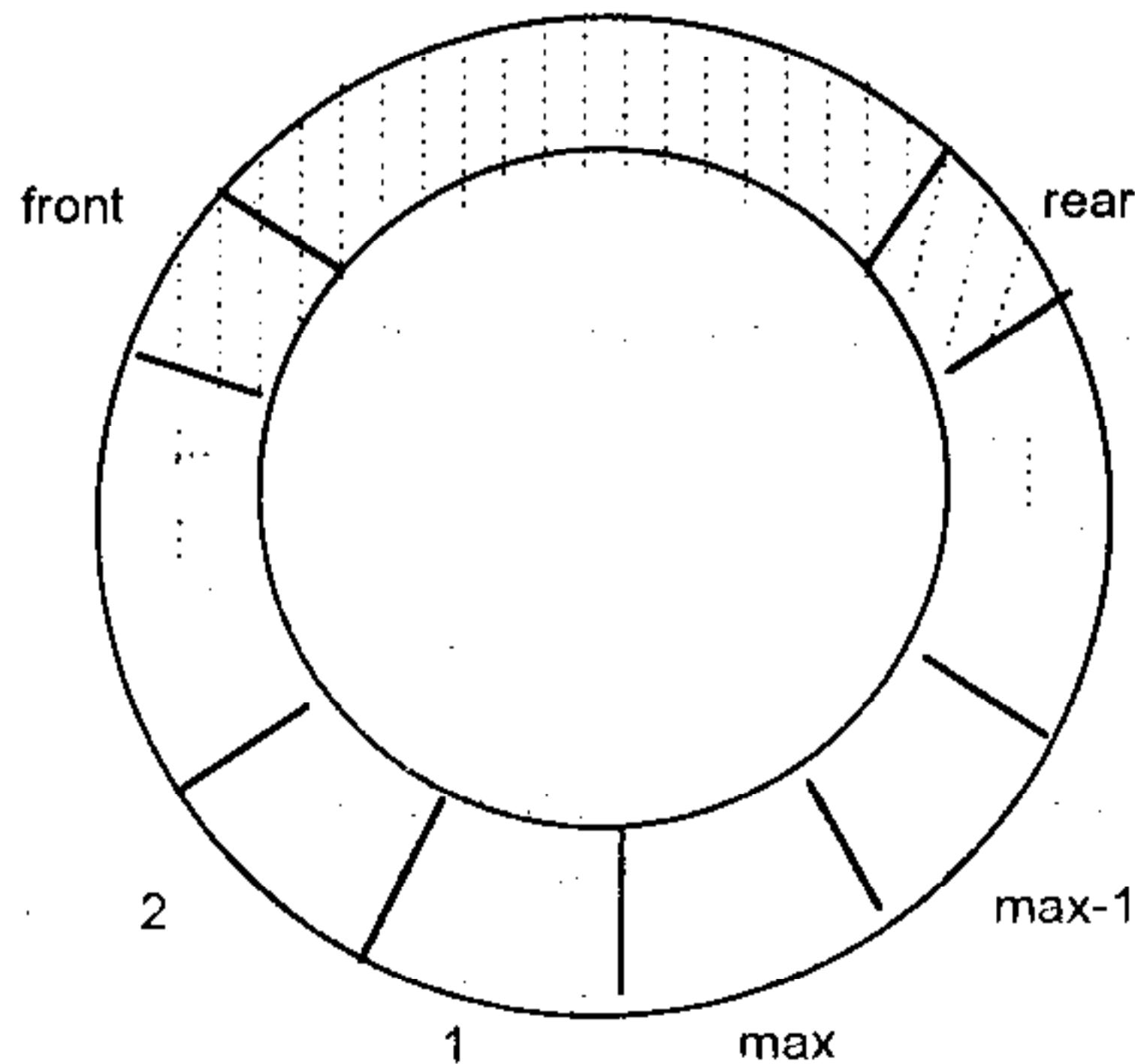
Có thể khắc phục nhược điểm trên bằng cách sau. Ta chỉ sử dụng một chỉ số rear để chỉ cuối hàng, còn phần tử đầu hàng luôn luôn được xem là thành phần đầu tiên của mảng, tức là ta luôn luôn coi front = 1.

```
type Queue = record
    rear : 0 .. max ;
    element : array[1..max] of Item ;
end ;
```

Tuy nhiên cách cài đặt này lại không thuận tiện cho phép toán loại phần tử đầu hàng, bởi vì mỗi lần loại bỏ ta phải dồn tất cả các phần tử còn lại của hàng lên một vị trí.

### 3.9.3. Cài đặt hàng bởi mảng vòng tròn

Trong trường hợp số phần tử trong hàng không bao giờ vượt quá một số cố định N nào đó, phương pháp tốt nhất là biểu diễn hàng bởi mảng vòng tròn. Đó là mảng với chỉ số chạy trong miền 1.. max, với mọi  $i = 1, 2, \dots, max - 1$ , thành phần thứ  $i$  của mảng đi trước thành phần thứ  $i + 1$  còn thành phần thứ  $max$  đi trước thành phần đầu tiên, tức là các thành phần của mảng được xếp thành vòng tròn (xem hình 3.15)



Hình 3.15. Mảng vòng tròn biểu diễn hàng.

Khi biểu diễn hàng bởi mảng vòng tròn, để biết khi nào hàng đầy, khi nào hàng rỗng ta cần đưa thêm vào biến count để đếm số phần tử trong hàng. Chúng ta có khai báo sau.

```

const max = N;
type Queue = record
    count : 0 .. max;
    front, rear : 0 .. max;
    element : array [1..max] of Item;
end;
var Q : Queue;

```

Trong cách cài đặt này, điều kiện để hàng Q rỗng là Q.count = 0 và nó đầy nếu Q.count = max.

Khi làm việc với mảng vòng tròn, ta cần lưu ý rằng, thành phần đầu tiên của mảng đi sau thành phần thứ max. Sau đây chúng ta sẽ viết các thủ tục AddQueue và DeleteQueue, các thủ tục khác giành lại cho bạn đọc.

```

procedure AddQueue (x : Item; var Q : Queue; ok : boolean);
begin
    with Q do
        if count = max then ok := false
        else begin
            if rear = max then rear := 1
            else rear := rear + 1;
            element [rear] := x;
            count := count + 1;
            Ok := true
        end;
    end;

```

```

procedure DeleteQueue (var Q : Queue ; var x : Item ; var Ok : boolean) ;
begin
with Q do
  if count = 0 then Ok := false
  else begin
    x := element [front] ;
    if front = rear then
      begin
        front := 1 ;
        rear := 0;
      end else
        if front = max then front := 1
        else front := front + 1 ;
    count := count - 1 ;
    Ok := true ;
  end;
end ;

```

### 3.9.4. Cài đặt hàng bởi danh sách liên kết

Cũng như stack, hàng có cấu trúc toán học như danh sách, nhưng chỉ được phép thực hiện được một số phép toán đặc biệt. Do đó, ta cũng có thể sử dụng danh sách liên kết để biểu diễn hàng. Ta cần đưa vào hai con trỏ, một con trỏ trỏ đến đầu hàng một con trỏ trỏ đến cuối hàng. Ta có thể mô tả cấu trúc dữ liệu danh sách liên kết để biểu diễn hàng như sau.

```

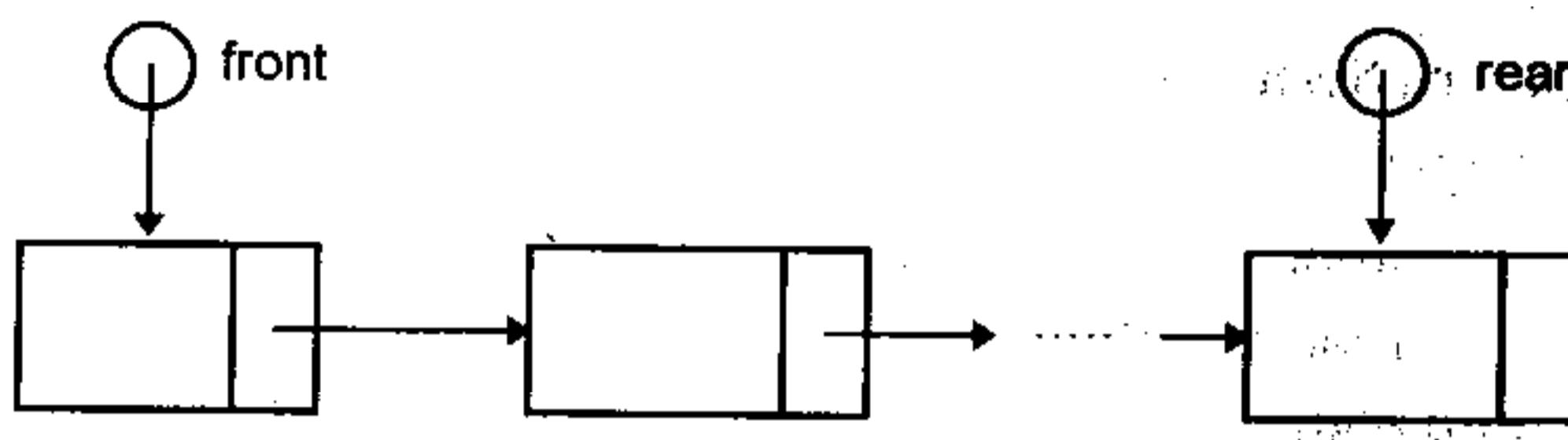
type pointer = ^ Cell;
Cell = record
  infor : Item ;
  next : pointer ;
end ;

```

```

Queue = record
    front : pointer;
    rear : pointer;
end;
var Q : Queue ;

```



**Hình 3.16.** Danh sách liên kết biểu diễn hàng

Trong cách cài đặt này, hàng được xem là không **khi** nào đầy (ta giả thiết là, luôn luôn có không gian nhớ để cấp phát cho các thành mới cần được đưa vào hàng). Điều kiện để hàng rỗng là **Q.front = nil**

Sau đây là các thủ tục và hàm thực hiện các phép toán trên hàng.

```

procedure Initialize (var Q : Queue) ;
begin
    Q.front := nil;
end ;

function Empty (var Q : queue) : boolean ;
begin
    if Q.front = nil then Empty := true
        else Empty := false;
end ;

function Full (var Q : Queue ) boolean ;
begin
    Full := false;
end ;

```

```
procedure AddQueue (x : Item; var Q : Queue ; var Ok : boolean);
```

```
var P : pointer ;
```

```
begin
```

```
    new (P) ;
```

```
    P^.infor := x ;
```

```
    P^.next := nil ;
```

```
with Q do
```

```
    if front = nil then
```

```
        begin
```

```
            front := P ;
```

```
            rear := P ;
```

```
        end else
```

```
        begin
```

```
            rear^.next := P ;
```

```
            rear := P ;
```

```
        end ;
```

```
    Ok := true ;
```

```
end ;
```

```
procedure DeleteQueue (var Q : Queue ; var x : Item ; var Ok : boolean);
```

```
var P : pointer ;
```

```
begin
```

```
with Q do
```

```
    if front = nil then Ok := false
```

```
    else begin
```

```
        P := front ;
```

```
        x := front^.infor ;
```

```
        front := front^.next ;
```

```
        Ok := true ;
```

```
        dispose (P) ;
```

```
    end ;
```

```
end ;
```

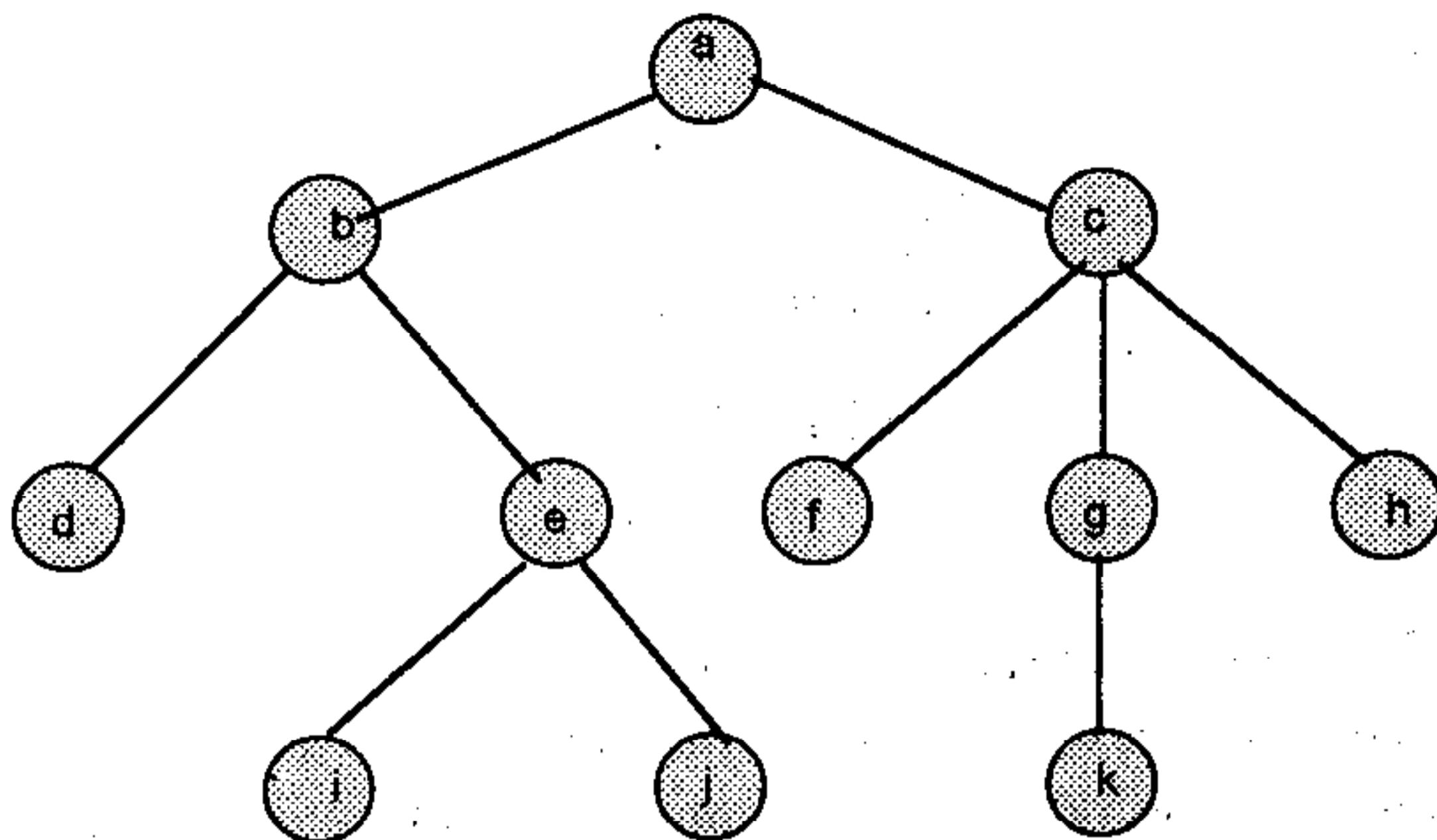
## **Chương 4**

### **CÂY**

Trong chương này chúng ta sẽ nghiên cứu mô hình dữ liệu cây. Cây là một cấu trúc phân cấp trên một tập hợp nào đó các đối tượng. Một ví dụ quen thuộc về cây, đó là cây thư mục. Cây được sử dụng rộng rãi trong rất nhiều vấn đề khác nhau. Chẳng hạn, nó được áp dụng để tổ chức thông tin trong các hệ cơ sở dữ liệu, để mô tả cấu trúc cú pháp của các chương trình nguồn khi xây dựng các chương trình dịch. Rất nhiều các bài toán mà ta gặp trong các lĩnh vực khác nhau được quy về việc thực hiện các phép toán trên cây. Trong chương này chúng ta sẽ trình bày định nghĩa và các khái niệm cơ bản về cây. Chúng ta cũng sẽ xét các phương pháp cài đặt cây và sự thực hiện các phép toán cơ bản trên cây. Sau đó chúng ta sẽ nghiên cứu kỹ một dạng cây đặc biệt, đó là cây tìm kiếm nhị phân.

#### **4.1. CÂY VÀ CÁC KHÁI NIỆM VỀ CÂY**

Hình 4.1 minh họa một cây T. Đó là một tập hợp T gồm 11 phần tử,  $T=\{a, b, c, d, e, f, g, h, i, j, k\}$ . Các phần tử của T được gọi là các đỉnh của cây T. Tập T có cấu trúc như sau. Các đỉnh của T được phân thành các lớp không cắt nhau : lớp thứ nhất gồm một đỉnh duy nhất a, đỉnh này gọi là gốc của cây; lớp thứ hai gồm các đỉnh b, c ; lớp thứ ba gồm các đỉnh d, e, f, g, h và lớp cuối cùng gồm các đỉnh i, j, k, mỗi đỉnh thuộc một lớp (trừ gốc), có một cung duy nhất nối với một đỉnh nào đó thuộc lớp kề trên. (Cung này biểu diễn mối quan hệ nào đó).

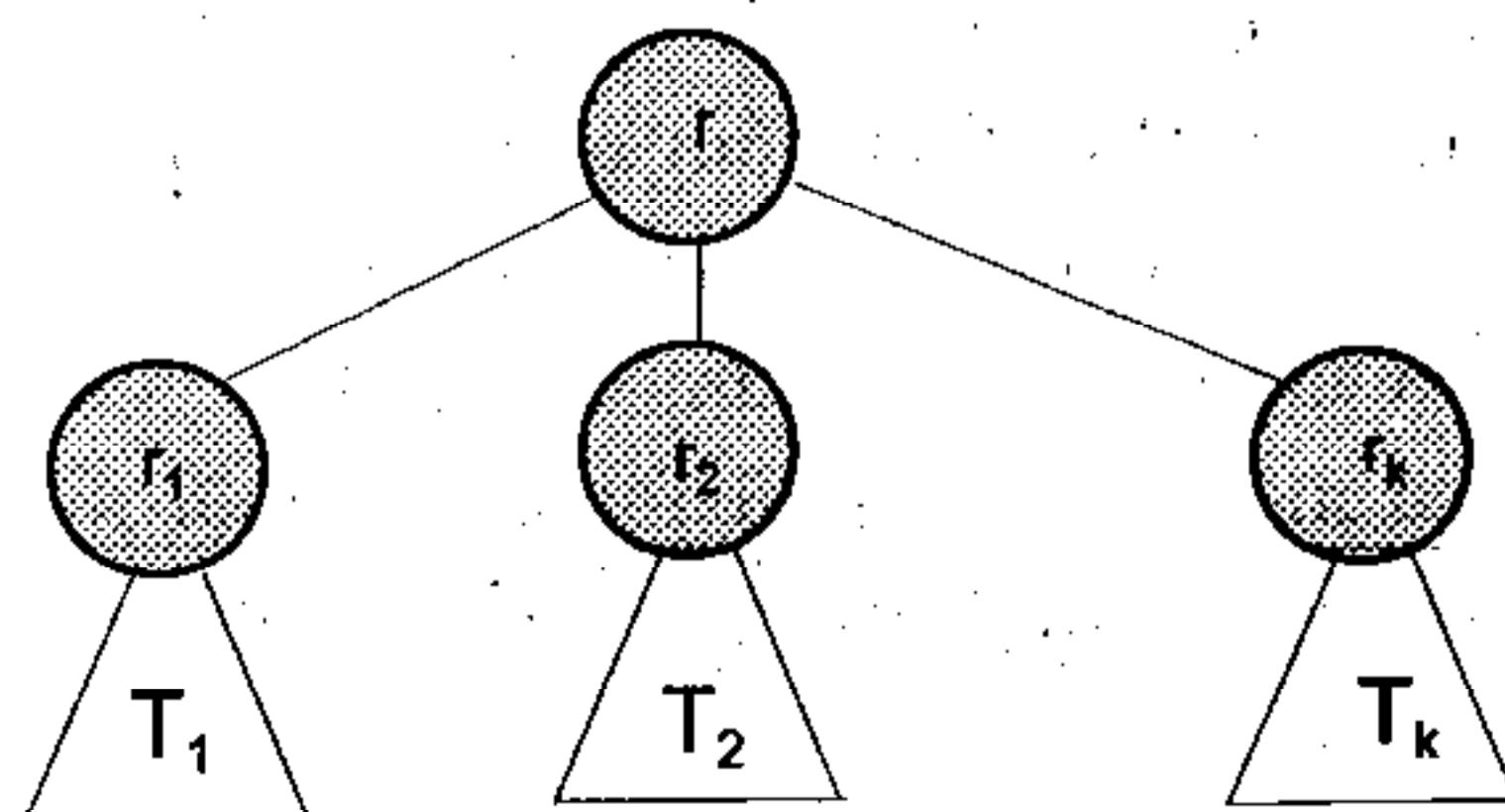
**Hình 4.1.** Biểu diễn hình học một cây

Trong toán học có nhiều cách định nghĩa cây. Ở đây chúng ta đưa ra định nghĩa đệ quy về cây. Định nghĩa này cho phép ta xuất phát từ các cây đơn giản nhất (cây chỉ có một đỉnh) xây dựng nên các cây lớn hơn.

Cây (cây có gốc) được xác định đệ quy như sau.

1. Tập hợp gồm một đỉnh là cây. Cây này có gốc là đỉnh duy nhất của nó.

2. Giả sử  $T_1, T_2, \dots, T_k$  ( $k \geq 1$ ) là các cây có gốc tương ứng là  $r_1, r_2, \dots, r_k$ . Các cây  $T_i$  ( $i = 1, 2, \dots, k$ ), không khống cắt nhau tức là  $T_i \cap T_j = \emptyset$  với  $i \neq j$ . Giả sử  $r$  là một đỉnh mới không thuộc các cây  $T_i$  ( $i = 1, 2, \dots, k$ ). Khi đó, tập hợp  $T$  gồm đỉnh  $r$  và tất cả các đỉnh của cây  $T_i$  ( $i = 1, 2, \dots, k$ ) lập thành một cây mới với gốc  $r$ . Các cây  $T_i$  ( $i = 1, 2, \dots, k$ ) được gọi là cây con của gốc  $r$ . Trong biểu diễn hình học của cây  $T$ , mỗi đỉnh  $r_i$  ( $i = 1, 2, \dots, k$ ) có cung nối với gốc  $r$  (xem hình 4.2).

**Hình 4.2.** Cây có gốc r và các cây con của gốc T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>k</sub>

### **Cha, con, đường đi**

Từ định nghĩa cây ta suy ra rằng, mỗi đỉnh của cây là gốc của các cây con của nó. Số các cây con của một đỉnh gọi là bậc của đỉnh đó. Các đỉnh có bậc không được gọi là lá của cây.

Nếu đỉnh b là gốc của một cây con của đỉnh a thì ta nói đỉnh b là con của đỉnh a và a là cha của b. Như vậy, bậc của một đỉnh là số các đỉnh con của nó, còn lá là đỉnh không có con. Các đỉnh có ít nhất một con được gọi là đỉnh trong. Các đỉnh của cây hoặc là lá hoặc là đỉnh trong.

Các đỉnh có cùng một cha được gọi là anh em. Một dãy các đỉnh  $a_1, a_2, \dots, a_n$  ( $n \geq 1$ ), sao cho  $a_i$  ( $i = 1, 2, \dots, n-1$ ) là cha của  $a_{i+1}$  được gọi là đường đi từ  $a_1$  đến  $a_n$ . Độ dài của đường đi này là  $n-1$ . Ta có nhận xét rằng, luôn luôn tồn tại một đường đi duy nhất từ gốc tới một đỉnh bất kỳ trong cây.

Nếu có một đường đi từ đỉnh a đến đỉnh b có độ dài  $k \geq 1$ , thì ta nói a là tiền thân của b và b là hậu thế của a.

**Ví dụ.** Trong cây ở hình 4.1, đỉnh c là cha của đỉnh f, g, h. Các đỉnh d, i, j, k và h là lá, các đỉnh còn lại là đỉnh trong. a, c, g, k là đường đi có độ dài 3 từ a đến k. Đỉnh b là tiền thân của các đỉnh d, e, i, j.

### **Cây con**

Từ định nghĩa cây ta có, mỗi đỉnh a bất kỳ của cây T là gốc của một cây nào đó, ta gọi cây này là cây con của cây T. Nó gồm đỉnh a và tất cả các đỉnh là hậu thế của a. Chẳng hạn, với cây T trong hình 4.1,  $T_1 = \{c, f, g, h, k\}$  là một cây con

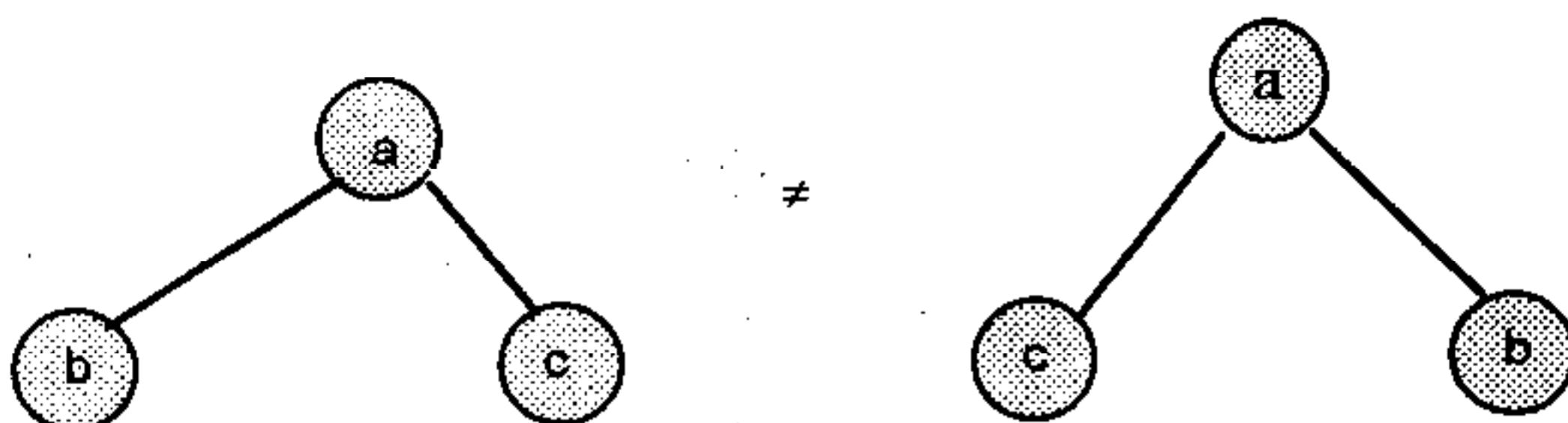
### **Độ cao, mức**

Trong một cây, độ cao của một đỉnh a là độ dài của đường đi dài nhất từ a đến một lá. Độ cao của gốc được gọi là độ cao của cây. Mức của đỉnh a là độ dài của đường đi từ gốc đến a. Như vậy gốc có mức 0.

**Ví dụ.** Trong cây ở hình 4.1, đỉnh b có độ cao là 2, cây có độ cao là 3. Các đỉnh b, c có mức 1 ; các đỉnh d, e, f, g, h có mức 2, còn mức của các đỉnh i, j, k là 3.

### **Cây được sắp**

Trong một cây, nếu các cây con của mỗi đỉnh được sắp theo một thứ tự nhất định, thì cây được gọi là cây được sắp. Chẳng hạn, hình 4.3 minh họa hai cây được sắp khác nhau,

**Hình 4.3.** Hai cây được sắp khác nhau

Sau này chúng ta chỉ quan tâm đến các cây được sắp. Do đó khi nói đến cây thì cần được hiểu là cây được sắp.

Giả sử trong một cây được sắp  $T$ , đỉnh  $a$  có các con được sắp theo thứ tự:  $b_1, b_2, \dots, b_k$  ( $k \geq 1$ ). Khi đó ta nói  $b_1$  là con trưởng của  $a$ , và  $b_i$  là anh liền kề của  $b_{i+1}$  ( $b_{i+1}$  là em liền kề của  $b_i$ ),  $i = 1, 2, \dots, k-1$ . Ta còn nói, với  $i < j$  thì  $b_i$  ở bên trái  $b_j$  ( $b_j$  ở bên phải  $b_i$ ). Quan hệ này được mở rộng như sau. Nếu  $a$  ở bên trái  $b$  thì mọi hậu thế của  $a$  ở bên trái mọi hậu thế của  $b$ .

**Ví dụ.** Trong hình 4.1,  $f$  là con trưởng của  $c$ , và là anh liền kề của đỉnh  $g$ . Đỉnh  $i$  ở bên trái đỉnh  $g$ .

### Cây gắn nhãn

Cây gắn nhãn là cây mà mỗi đỉnh của nó được gắn với một giá trị (nhãn) nào đó. Nói một cách khác, cây gắn nhãn là một cây cùng với một ánh xạ từ tập hợp các đỉnh của cây vào tập hợp nào đó các giá trị (các nhãn). Chúng ta có thể xem nhãn như thông tin liên kết với mỗi đỉnh của cây. Nhãn có thể là các dữ liệu đơn như số nguyên, số thực, hoặc cũng có thể là các dữ liệu phức tạp như bản ghi. Cần biết rằng, các đỉnh khác nhau của cây có thể có cùng một nhãn.

### Rừng

Một rừng  $F$  là một danh sách các cây

$$F = (T_1, T_2, \dots, T_n)$$

trong đó  $T_i$  ( $i = 1, \dots, n$ ) là cây (cây được sắp)

Chúng ta có tương ứng một - một giữa tập hợp các cây và tập hợp các rừng. Thật vậy, một cây  $T$  với gốc  $r$  và các cây con của gốc theo thứ tự từ trái sang phải là  $T_1, T_2, \dots, T_n$ ,  $T = (r, T_1, T_2, \dots, T_n)$  tương ứng với rừng  $F = (T_1, T_2, \dots, T_n)$  và ngược lại.

## 4.2. CÁC PHÉP TOÁN TRÊN CÂY

Trong mục 1 chúng ta đã trình bày cấu trúc toán học cây. Để có một mô hình dữ liệu cây, ta cần phải xác định các phép toán có thể thực hiện được trên cây. Cũng như với danh sách, các phép toán có thể thực hiện được trên cây rất đa dạng và phong phú. Trong số đó, có một số phép toán cơ bản được sử dụng thường xuyên để thực hiện các phép toán khác và thiết kế các thuật toán trên cây.

### 4.2.1. Các phép toán cơ bản trên cây

#### *1. Tìm cha của mỗi đỉnh*

Giả sử  $x$  là đỉnh bất kỳ trong cây  $T$ . Hàm  $\text{Parent}(x)$  xác định cha của đỉnh  $x$ . Trong trường hợp đỉnh  $x$  không có cha ( $x$  là gốc) thì giá trị của hàm  $\text{Parent}(x)$  là một ký hiệu đặc biệt nào đó khác với tất cả các đỉnh của cây, chẳng hạn  $\$$ . Như vậy nếu  $\text{parent}(x) = \$$  thì  $x$  là gốc của cây.

#### *2. Tìm con bên trái ngoài cùng (con trưởng) của mỗi đỉnh*

Hàm  $\text{EldestChild}(x)$  cho ta con trưởng của đỉnh  $x$ . Trong trường hợp  $x$  là lá ( $x$  không có con) thì  $\text{EldestChild}(x) = \$$ .

#### *3. Tìm em liền kề của mỗi đỉnh*

Hàm  $\text{NextSibling}(x)$  xác định em liền kề của đỉnh  $x$ . Trong trường hợp  $x$  không có em liền kề (tức  $x$  là con ngoài cùng bên phải của một đỉnh nào đó) thì  $\text{NextSibling}(x) = \$$ .

**Ví dụ.** Giả sử  $T$  là cây đã cho trong hình 4.1. Khi đó  $\text{Parent}(e) = b$ ,  $\text{Parent}(a) = \$$ ,  $\text{EldestChild}(c) = f$ ,  $\text{EldestChild}(k) = \$$ ,  $\text{NextSibling}(g) = h$ ,  $\text{NextSibling}(h) = \$$ .

### 4.2.2. Đi qua cây (duyệt cây)

Trong thực tiễn chúng ta gặp rất nhiều bài toán mà việc giải quyết nó được qui về việc đi qua cây (còn gọi là duyệt cây), "thăm" tất cả các đỉnh của cây một cách hệ thống.

Có nhiều phương pháp đi qua cây. Chẳng hạn, ta có thể đi qua cây lần lượt từ mức 0, mức 1, ... cho tới mức thấp nhất. Trọng cùng một mức ta sẽ thăm các đỉnh từ trái sang phải. Ví dụ, với cây trong hình 4.1, danh sách các đỉnh lần lượt được thăm là (a, b, c, d, e, f, g, h, i, j, k). Đó là phương pháp đi qua cây theo bề rộng.

Tuy nhiên, ba phương pháp đi qua cây theo các hệ thống sau đây là quan trọng nhất : đi qua cây theo thứ tự Preorder, Inorder và Postorder. Danh sách các đỉnh của cây theo thứ tự Preorder, Inorder, và Postorder (gọi tắt là danh sách Preorder, Inorder, và Postorder) được xác định đệ qui như sau

1. Nếu T là cây gồm một đỉnh duy nhất thì các danh sách Preorder, Inorder và Postorder chỉ chứa một đỉnh đó.

2. Nếu T là cây có gốc r và các cây con của gốc là  $T_1, T_2, \dots, T_k$  (hình 4.2) thì

2a. Danh sách Preorder các đỉnh của cây T bắt đầu là r, sau là các đỉnh của cây con  $T_1$  theo thứ tự Preorder, rồi đến các đỉnh của cây con  $T_2$  theo thứ tự Preorder, ..., cuối cùng là các đỉnh của cây con  $T_k$  theo thứ tự Preorder.

2b. Danh sách Inorder các đỉnh của cây T bắt đầu là các đỉnh của cây con  $T_1$  theo thứ tự Inorder, rồi đến gốc r, sau là các đỉnh của các cây con  $T_2, \dots, T_k$  theo thứ tự Inorder.

2c. Danh sách Postorder các đỉnh của cây T lần lượt là các đỉnh của các cây con  $T_1, T_2, \dots, T_k$ , sau cùng là gốc r.

**Ví dụ.** Khi đi qua cây trong hình 4.1 theo thứ tự Preorder ta được danh sách các đỉnh là (a, b, d, e, i, j, c, f, g, k, h). Nếu đi qua cây theo thứ tự Inorder, ta có danh sách (d, b, i, e, j, a, f, c, k, g, h). Còn danh sách Postorder là (d, i, j, e, b, f, k, g, h, c, a).

Phương pháp đi qua cây theo thứ tự Preorder còn được gọi là kỹ thuật *đi qua cây theo độ sâu*. Đó là một kỹ thuật quan trọng thường được áp dụng để tìm kiếm nghiệm của các bài toán. Gọi là đi qua cây theo độ sâu, bởi vì khi ta đang ở một đỉnh x nào đó của cây (chẳng hạn, đỉnh b trong cây ở hình 4.1), ta cố gắng đi sâu xuống đỉnh còn chưa được thăm ngoài cùng bên trái chừng nào có thể được (chẳng hạn, đỉnh d trong cây ở hình 4.1) để thăm đỉnh đó. Nếu tất cả các đỉnh con của x đã

được thăm (tức là từ x không thể đi sâu xuống được) ta quay lên tìm đến cha của x. Tại đây ta lại cố gắng đi sâu xuống đỉnh con chưa được thăm. Chẳng hạn, trong cây ở hình 4.1, ta đang ở đỉnh f, tại đây không thể đi sâu xuống, ta quay lên cha của f là đỉnh c. Tại c có thể đi sâu xuống thăm đỉnh g, từ g lại có thể đi sâu xuống thăm đỉnh k. Quá trình trên cứ tiếp tục cho tới khi nào toàn bộ các đỉnh của cây đã được thăm.

Đối lập với kỹ thuật đi qua cây theo độ sâu là kỹ thuật *đi qua cây theo bề rộng* mà chúng ta đã trình bày. Trong kỹ thuật này, khi đang ở thăm đỉnh x nào đó của cây, ta đi theo bề ngang sang bên phải tìm đến em liền kề của x để thăm. Nếu x là đỉnh ngoài cùng bên phải, ta đi xuống mức sau thăm đỉnh ngoài cùng bên trái, rồi lại tiếp tục đi theo bề ngang sang bên phải.

Sau đây chúng ta sẽ trình bày các thủ tục đi qua cây theo các thứ tự Preorder, Inorder, Postorder và đi qua cây theo bề rộng.

Sử dụng các phép toán cơ bản trên cây và định nghĩa đệ qui của thứ tự Preorder, chúng ta dễ dàng viết được thủ tục đệ qui đi qua cây theo thứ tự Preorder. Trong thủ tục, chúng ta sẽ sử dụng thủ tục Visit (x) (thăm đỉnh x) nó được cài đặt tùy theo từng ứng dụng. Các biến A, B trong thủ tục là các đỉnh (Node) của cây.

```

procedure Preorder ( A : Node );
  {Thủ tục đệ qui đi qua cây gốc A theo thứ tự Preorder}
  var B : Node;
  begin
    Visit (A);
    B := EldestChild (A);
    while B < > $ do
      begin
        Preorder ( B );
        B := NextSibling (B);
      end ;
  end ;

```

Một cách tương tự, ta có thể viết được các thủ tục đệ qui đi qua cây theo thứ tự Inorder và Postorder.

```
procedure Inorder ( A : Node ) ;
```

{Thủ tục đệ qui đi qua cây gốc A theo thứ tự Inorder }

```
var B : Node ;
```

```
begin
```

```
    B := EldestChild (A) ;
```

```
    if B < > $ then
```

```
        begin
```

```
            Inorder (B) ;
```

```
            B := NextSibling (B);
```

```
        end ;
```

```
        Visit (A) ;
```

```
        while B < > $ do
```

```
            begin
```

```
                Inorder (B) ;
```

```
                B := NextSibling (B);
```

```
            end ;
```

```
        end ;
```

```
procedure Postorder (A : Node) ;
```

{Thủ tục đệ qui đi qua cây gốc A theo thứ tự Postorder}

```
var B : Node ;
```

```
begin
```

```
    B := EldestChild (A) ;
```

```
    while B < > $ do
```

```
        begin
```

```
            Postorder (B) ;
```

```
            B := NextSibling (B);
```

```
        end ;
```

```
        Visit (A);
```

```
end ;
```

Chúng ta cũng có thể viết được các thủ tục không đệ qui đi qua cây theo các thứ tự Preorder, Inorder và Postorder. Chúng ta sẽ viết một trong ba thủ tục đó (các thủ tục khác giành lại cho độc giả). Tư tưởng cơ bản của thuật toán không đệ qui đi qua cây theo thứ tự Preorder là như sau. Chúng ta sẽ sử dụng một ngăn xếp S để lưu giữ các đỉnh của cây. Nếu ở một thời điểm nào đó ta đang ở thăm đỉnh x thì ngăn xếp sẽ lưu giữ đường đi từ gốc đến x, gốc ở đáy của ngăn xếp còn x ở đỉnh ngăn xếp. Chẳng hạn, với cây trong hình 4.1, nếu ta đang ở thăm đỉnh i, thì ngăn xếp sẽ lưu (a, b, e, i) và i ở đỉnh ngăn xếp

```

procedure Preorder ( A : Node ) ;
    {Thủ tục không đệ qui đi qua cây theo thứ tự Preorder}
var B : Node ;
    S : Stack ;
begin
    Initialize (S) ; {khởi tạo stack rỗng}
    B := A ;
    while B < > $ do
        begin
            Visit (B) ;
            Push (B, S) ; {đẩy B vào stack}
            B := EldestChild (B);
        end ;
    while not Empty (S) do
        begin
            Pop (S,B) ;{loại phần tử ở đỉnh stack và gán cho B}
            B := NextSibling (B) ;
            if B < > $ then
                while B < > $ do
                    begin
                        Visit (B) ;
                        Push (B, S) ;
                        B := EldestChild (B);
                    end ;
            end ;
    end ;

```

Sau đây chúng ta sẽ trình bày thuật toán đi qua cây theo bề rộng, chúng ta sẽ sử dụng hàng Q để lưu giữ các đỉnh theo thứ tự đã được thăm, đầu hàng là đỉnh ngoài cùng bên trái mà ta chưa thăm các con của nó, còn cuối hàng là đỉnh ta đang ở thăm. Chẳng hạn, với cây trong hình 4.1, nếu ta đang ở thăm đỉnh i thì trong hàng sẽ chứa các đỉnh (f, g, h, i) trong đó f ở đầu hàng và i ở cuối hàng. Khi loại một phần tử ở đầu hàng, chúng ta sẽ lần lượt thăm các con của nó (nếu có) và khi thăm đỉnh nào thì đưa đỉnh đó vào cuối hàng. Chúng ta có thủ tục sau

```

procedure BreadthTraverse ( A : Node ) ;
    {Thủ tục đi qua cây gốc A theo bề rộng }

var B : node ;
    Q : Queue ;

begin
    Initialize (Q) ; {khởi tạo hàng rỗng}
    Visit (A) ;
    Add (A, Q) ; {đưa gốc A vào hàng Q}
    while not Empty (Q) do
        begin
            Delete (Q, B) ; {loại phần tử đầu hàng và gán cho B}
            B := EldestChild (B) ;
            while B < > $ do
                begin
                    Visit (B) ;
                    Add (B, Q) ;
                    B := NextSibling (B);
                end ;
        end ;
    end ;

```

### 4.3. CÀI ĐẶT CÂY

Trong mục này chúng ta sẽ trình bày các phương pháp cơ bản cài đặt cây và nghiên cứu khả năng thực hiện các phép toán cơ bản trên cây trong mỗi cách cài đặt.

#### 4.3.1. Biểu diễn cây bằng danh sách các con của mỗi đỉnh

Phương pháp thông dụng để biểu diễn cây là, với mỗi đỉnh của cây ta thành lập một danh sách các đỉnh con của nó theo thứ tự từ trái sang phải.

##### 1. Cài đặt bởi mảng

Trong cách cài đặt này, ta sẽ sử dụng một mảng để lưu giữ các đỉnh của cây. Mỗi thành phần của mảng là một tế bào chứa thông tin gắn với mỗi đỉnh và danh sách các đỉnh con của nó. Danh sách các đỉnh con của một đỉnh có thể biểu diễn bởi mảng hoặc bởi danh sách liên kết. Tuy nhiên, vì số con của mỗi đỉnh có thể thay đổi nhiều, cho nên ta sẽ sử dụng danh sách liên kết. Như vậy mỗi tế bào mô tả đỉnh của cây là một bản ghi gồm hai trường : trường infor chứa thông tin gắn với đỉnh, trường Child là con trỏ trỏ tới danh sách các con của đỉnh đó. Giả sử các đỉnh của cây được đánh số từ 1 đến N. Với cách cài đặt này, ta có thể khai báo cấu trúc dữ liệu biểu diễn cây như sau

```

const N = ... ;
type pointer = ^ Member ;
    Member = record
        id : 1..N ;
        next : pointer;
    end ;
    Node = record
        infor : item ;
        child : pointer;
    end ;
Tree = array [1.. N] of Node

```

Trong khai báo trên, Member biểu diễn các thành phần của danh sách các con, còn Node biểu diễn các đỉnh của cây. Với cách cài đặt này, cấu trúc dữ liệu biểu diễn cây trong hình 4.4a được minh họa trong hình 4.4b.

Ta có nhận xét rằng, trong cách cài đặt này, với mỗi đỉnh k ta xác định được ngay con trưởng của nó. Chẳng hạn, với cây trong hình 4.4b, con trưởng của đỉnh 3 là đỉnh 6, con trưởng của đỉnh 5 là 9, còn đỉnh 6 không có con. Phép toán tìm con trưởng EldestChild (k) có thể được mô tả bởi hàm sau.

```
function EldestChild ( k : 1.. N ; T : Tree ) : 0..N ;
var P : pointer ;
begin
    if T[k] < > nil then
        begin
            P := T[k]. child ;
            EldestChild := P^. id ;
        end else EldestChild := 0
    end ;
```

Tuy nhiên trong cách cài đặt này, việc tìm cha và em liền kề của mỗi đỉnh lại không đơn giản. Chẳng hạn, để tìm cha của đỉnh k, ta phải duyệt các danh sách các con của mỗi đỉnh. Nếu phát hiện ra trong danh sách các con của đỉnh m có chứa k thì Parent (k) = m. Hàm Parent (k) được xác định như sau

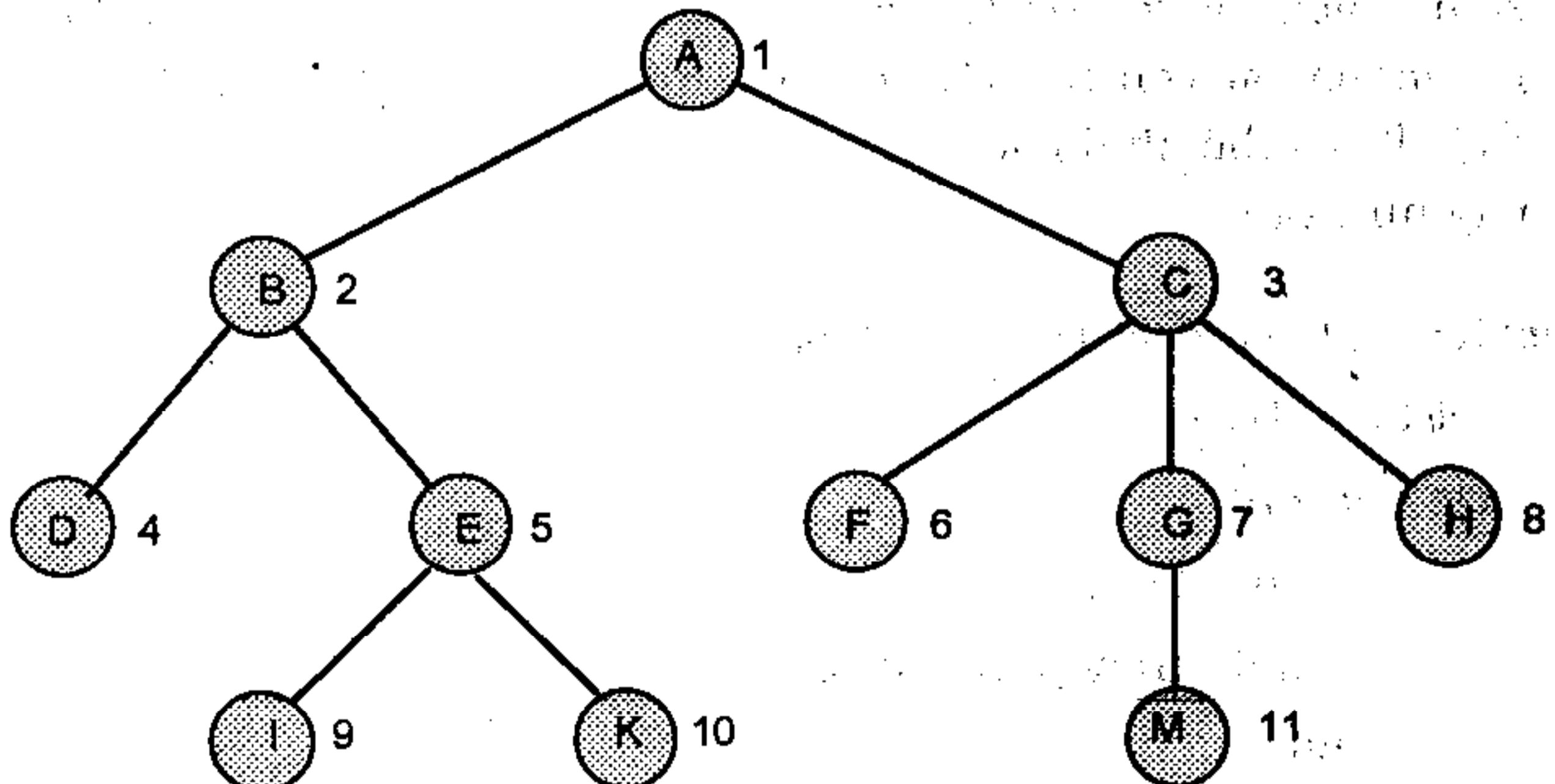
```
function Parent (k : 1..N ; T : Tree) : 0..N ;
var P : pointer ;
    i : 1..N ;
    found : boolean ;
begin
    i := 1 ;
    found := false ;
```

```

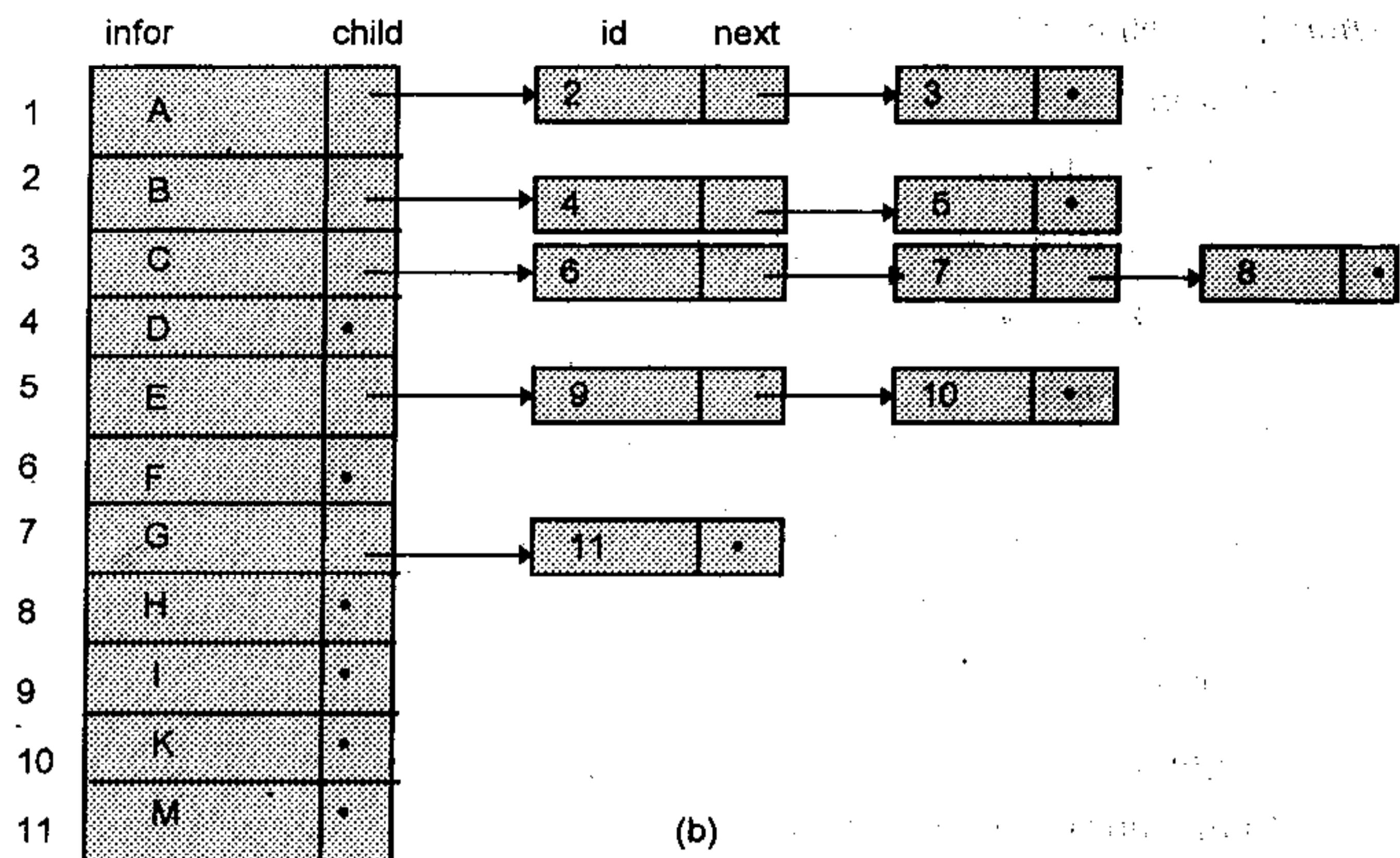
while ( i <= N) and (not found) do
    begin
        P := T[i].child ;
        while (P <> nil) and (not found) do
            if P^.id = k then
                begin
                    Parent := i ;
                    found := true ;
                end else P := P^.next ;
            i := i + 1 ;
        end ;
        if not found then Parent := 0 ;
    end ;

```

Một cách tương tự (duyệt các danh sách các con), ta cũng có thể tìm được em liền kề của mỗi đỉnh. Mô tả chi tiết hàm `NextSibling` được để lại xem như bài tập.



Hình 4.4 (a)



Hình 4.4. Cấu trúc dữ liệu biểu diễn cây

## 2. Cài đặt bởi con trỏ

Nếu không dùng mảng để lưu giữ các đỉnh của cây, ta có thể sử dụng các con trỏ trỏ tới các đỉnh của cây. Tại mỗi đỉnh, ta sẽ sử dụng một danh sách các con trỏ trỏ tới các con của nó, danh sách này được cài đặt bởi mảng các con trỏ. Một con trỏ Root được sử dụng để trỏ tới gốc của cây. Ta có thể khai báo cấu trúc dữ liệu biểu diễn cây trong cách cài đặt này như sau.

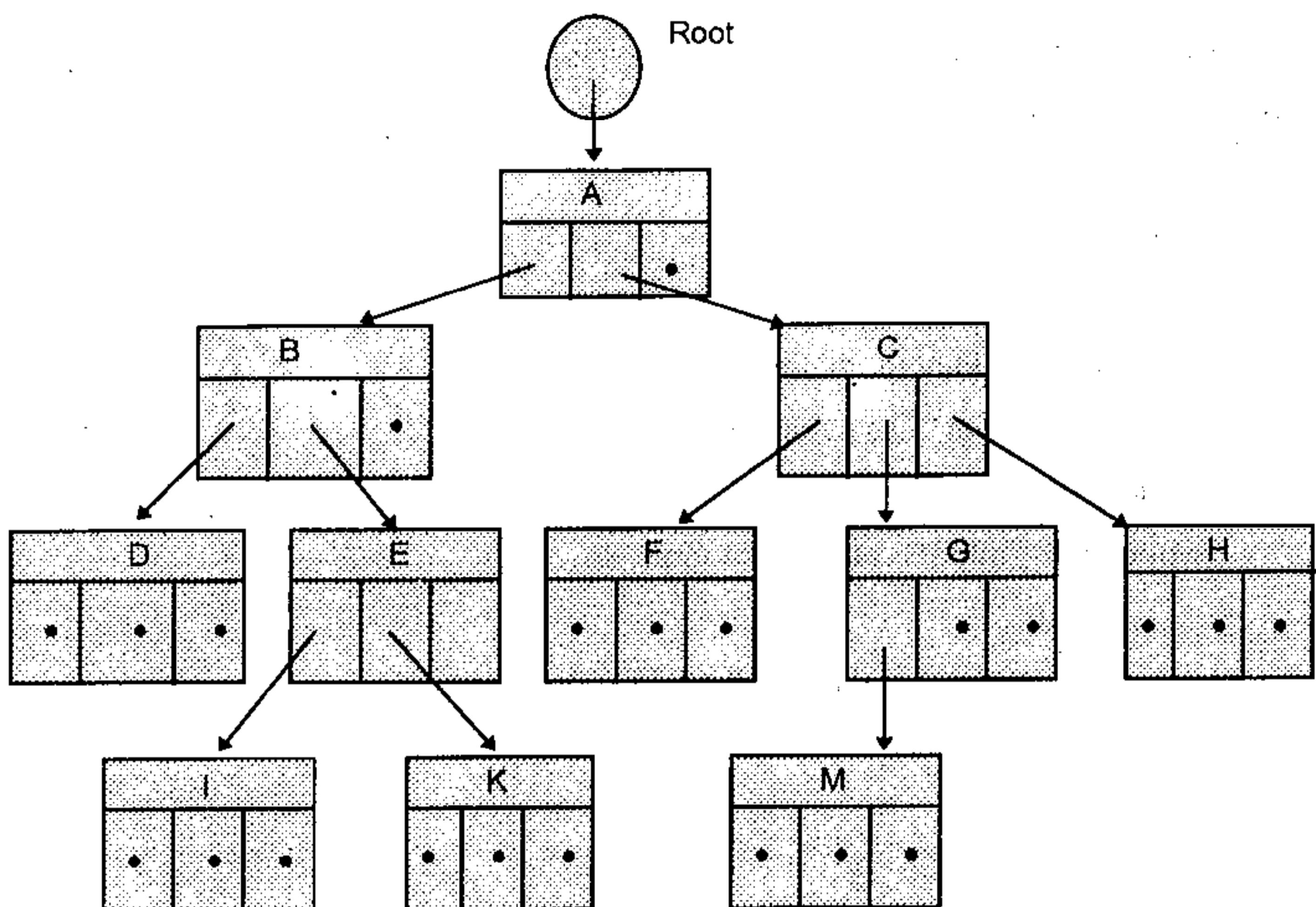
```

const K = ... ; {K là số tối đa các con của mỗi đỉnh}
type pointer = ^Node ;

Node = record
    infor : item ;
    child : array [1..K] of pointer ;
end ;
var Root : pointer ;

```

Với cách cài đặt này, cấu trúc dữ liệu biểu diễn cây trong hình 4.4a được minh họa trong hình 4.5.



**Hình 4.5.** Biểu diễn cây bởi con trỏ

Giả sử P là một con trỏ trỏ tới một đỉnh nào đó trong cây, ta sẽ gọi đỉnh này là đỉnh P. Sau đây ta sẽ xét xem các phép toán tìm con trưởng của nó EldestChild (P), tìm cha của nó Parent (P) và tìm em liên kề NextSibling (P) được thực hiện như thế nào. Dễ dàng thấy rằng, cũng như trong cách cài đặt bởi mảng, ta có thể xác định được ngay con trưởng của một đỉnh. Bạn đọc tự viết lấy hàm EldestChild. Tư tưởng của thuật toán tìm cha của đỉnh P cũng không có gì khác trước, khi cây cài đặt bởi mảng, tức là ta cũng phải duyệt các đỉnh con của mỗi đỉnh. Song trước kia, khi các đỉnh của cây được lưu trong mảng, việc đi lần lượt qua các đỉnh của cây để xét các con của nó được thực hiện rất dễ dàng. Còn ở đây ta phải sử dụng một hàng H để lưu các đỉnh đã được xét. Đầu tiên hàng chứa gốc Root của cây. Tại mỗi thời điểm ta sẽ loại đỉnh Q ở đầu hàng ra khỏi hàng và xét các con của nó. Nếu một trong các đỉnh con

của Q là P thì Parent (P) = Q, trong trường hợp ngược lại ta sẽ đưa các đính con của Q vào cuối hàng. Hàm Parent được xác định như sau.

```

function      Parent (P : pointer ; Root : pointer) : pointer ;
var    Q, R : pointer ;
        H : Queue ;
        found : boolean ;
begin
    Initialize (H) ; {khởi tạo hàng rỗng H}
    AddQueue (Root, H) ; {Đưa Root vào hàng}
    found := false ;
    while (not Emty (H) and (not found)) do
        begin
            DeleteQueue (H, Q) ; {Loại Q khỏi đầu hàng}
            i := 0 ;
            repeat
                i := i + 1 ;
                R := Q^.child [i] ;
                if R < > nil then
                    if R = P then
                        begin
                            Parent := Q;
                            found := true;
                        end else AddQueue (R, H)
                until found or (R = nil) or (i = N)
        end ;
        if not found then Parent := nil;
    end ;

```

Một cách hoàn toàn tương tự, ta có thể viết được hàm tìm em liền kề NextSibling.

### 4.3.2. Biểu diễn cây bằng con trưởng và em liền kề của mỗi đỉnh

Một phương pháp thông dụng khác để biểu diễn cây là, với mỗi đỉnh của cây ta chỉ ra con trưởng và em liền kề của nó.

#### 1. Cài đặt bởi mảng

Giả sử các đỉnh của cây được đánh số từ 1 đến N. Dùng mảng để lưu giữ các đỉnh của cây, mỗi đỉnh được biểu diễn bởi bản ghi gồm ba trường, ngoài trường infor, các trường EldestChild và Nexsibling sẽ lưu con trưởng và em liền kề của mỗi đỉnh. Ta có thể khai báo như sau

```
type Node = record
    infor : item;
    EldestChild : 0..N;
    NextSibling : 0..N;
end;
Tree = array[1..N] of Node;
```

Hình 4.6 Minh họa cấu trúc dữ liệu biểu diễn cây trong hình 4.4a.

Trong cách biểu diễn này, EldestChild và NexSibling được đưa vào làm các trường của bản ghi biểu diễn mỗi đỉnh của cây. Do đó, ta chỉ còn phải xét phép toán tìm cha của mỗi đỉnh. Cũng như trước kia, để tìm cha của một đỉnh k nào đó, ta sẽ lần lượt đi qua các đỉnh của cây, với mỗi đỉnh ta tìm đến các con của nó, cho tới khi tìm thấy đỉnh k. Cụ thể ta có thể mô tả hàm Parent(k) như sau

```
function Parent (k : 1..N; T : Tree) : 0..N;
var i, j : 0..N;
    found : boolean;
begin
    i := 1;
    found := false;
    while (i <= N) and (not found) do
        begin
            j := T[i].EldestChild;
```

```

if j = k then
    begin
        Parent := i;
        found := true;
    end
else begin
    j := T[j].NextSibling;
    while (j < > 0) and (not found) do
        if j = k then
            begin
                Parent := i;
                found := true
            end
        else j := T[j].NextSibling;
    end;
    i := i+1
end;
if not found then Parent := 0
end;

```

	infor	EldestChild	NextSibling
1	A	2	0
2	B	4	3
3	C	6	0
4	D	0	5
5	E	9	0
6	F	0	7
7	G	11	8
8	H	0	0
9	I	0	10
10	K	0	0
11	M	0	0

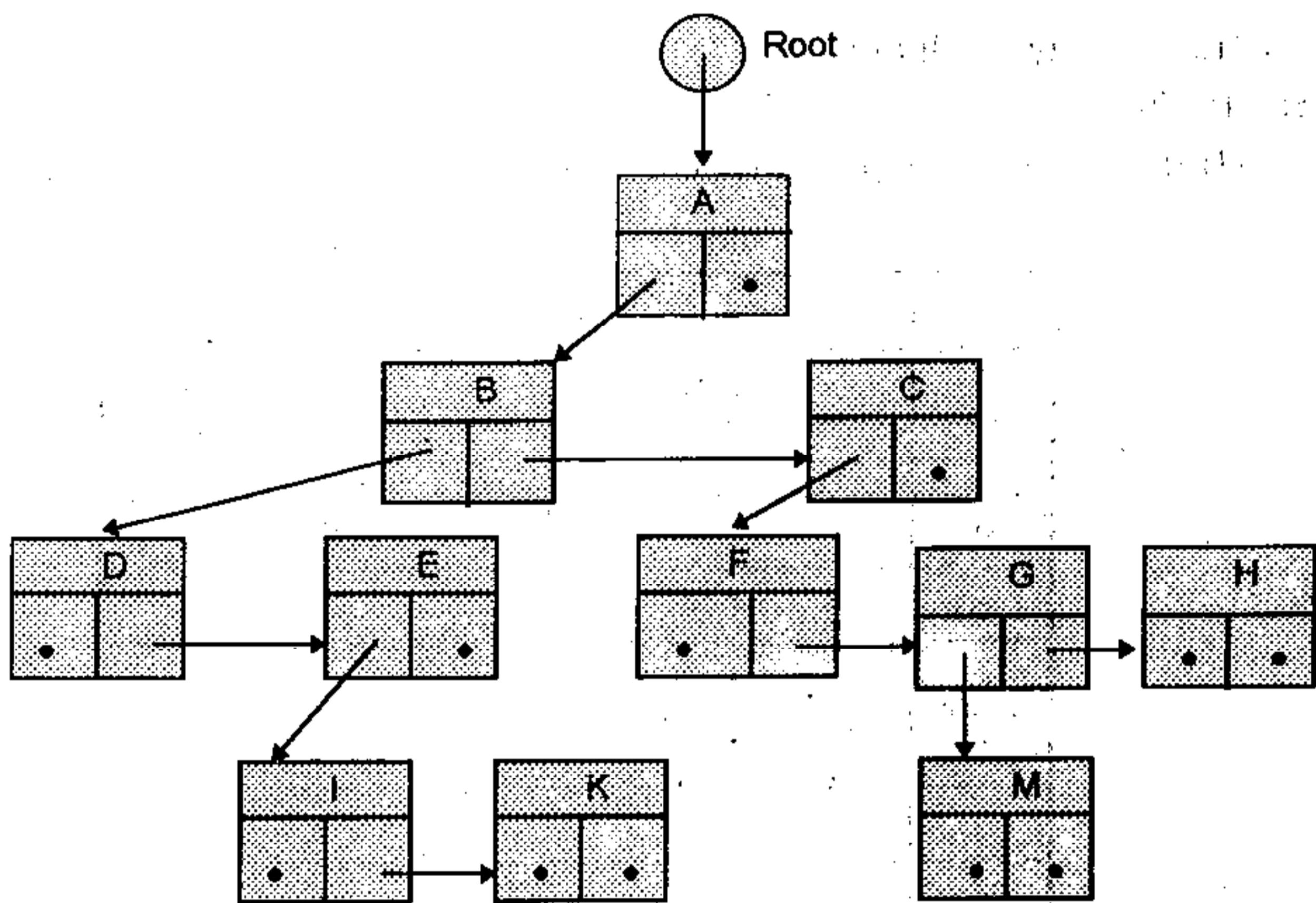
Hình 4.6

## 2. Cài đặt bởi con trỏ

Thay cho dùng mảng, ta có thể sử dụng các con trỏ để cài đặt. Khi đó trong bản ghi Node, các trường EldestChild và NextSibling sẽ là các con trỏ. Cây sẽ được biểu diễn bởi cấu trúc dữ liệu sau.

```
Type pointer = ^Node ;
Node = record
    infor : item ;
    EldestChild : pointer ;
    NextSibling : pointer ;
end ;
var Root : pointer ;
```

Trong cách cài đặt này, cây trong hình 4.4a được biểu diễn bởi cấu trúc dữ liệu trong hình 4.7.



Hình 4.7. Cấu trúc dữ liệu biểu diễn cây

Độc giả hãy tự viết lối thủ tục tìm cha của đỉnh P, Parent (P), trong đó P là con trỏ, trong cách cài đặt này.

### 4.3.3. Biểu diễn cây bởi cha của mỗi đỉnh

Trong một số áp dụng, người ta còn có thể sử dụng cách biểu diễn cây đơn giản sau đây. Giả sử các đỉnh của cây được đánh số từ 1 đến N. Dựa vào tính chất, mỗi đỉnh của cây (trừ gốc) đều có một cha, ta sẽ dùng một mảng A[1...N] để biểu diễn cây, trong đó A[k] = m nếu đỉnh m là cha của đỉnh k. Trong trường hợp cần quan tâm đến các thông tin gắn với mỗi đỉnh, ta cần phải đưa vào mỗi thành phần của mảng trường infor mô tả thông tin ở mỗi đỉnh. Cây được biểu diễn bởi cấu trúc dữ liệu sau.

```

const N = ... ;
type Node = record
    infor : item ;
    parent : 0..N ;
end ;
Tree = array [1..N] of Node ;
var T : Tree ;

```

Hình 4.8. minh họa cấu trúc dữ liệu biểu diễn cây trong hình 4.4a.

	infor	parent
1	A	0
2	B	1
3	C	1
4	D	2
5	E	2
6	F	3
7	G	3
8	H	3
9	I	5
10	K	5
11	M	5

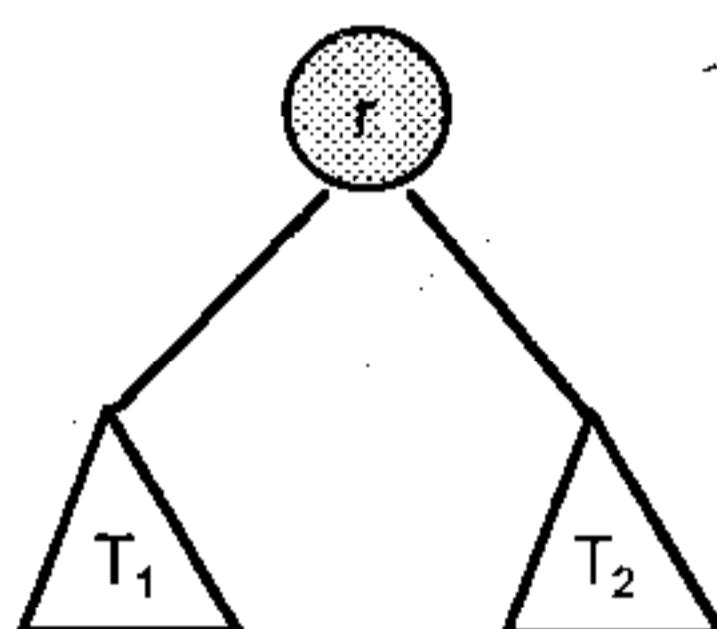
Hình 4.8

## 4.4. CÂY NHỊ PHÂN

Bắt đầu từ mục này chúng ta sẽ xét một dạng cây đặc biệt : cây nhị phân.

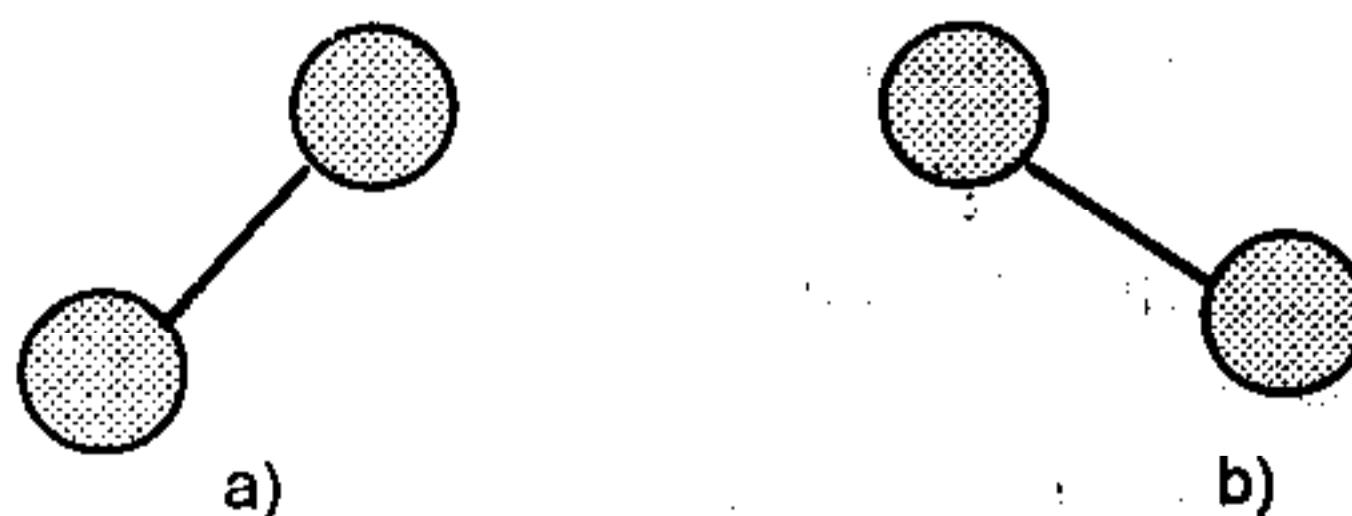
Cây nhị phân là một tập hợp hữu hạn các đỉnh được xác định đệ qui như sau.

1. Một tập trống là cây nhị phân
2. Giả sử  $T_1$  và  $T_2$  là hai cây nhị phân không cắt nhau ( $T_1 \cap T_2 = \emptyset$ ) và  $r$  là một đỉnh mới không thuộc  $T_1$ ,  $T_2$ . Khi đó ta có thể thành lập một cây nhị phân mới  $T$  với gốc  $r$  có  $T_1$  là cây con bên trái,  $T_2$  là cây con bên phải của gốc. Cây nhị phân  $T$  được biểu diễn bởi hình 4.9.



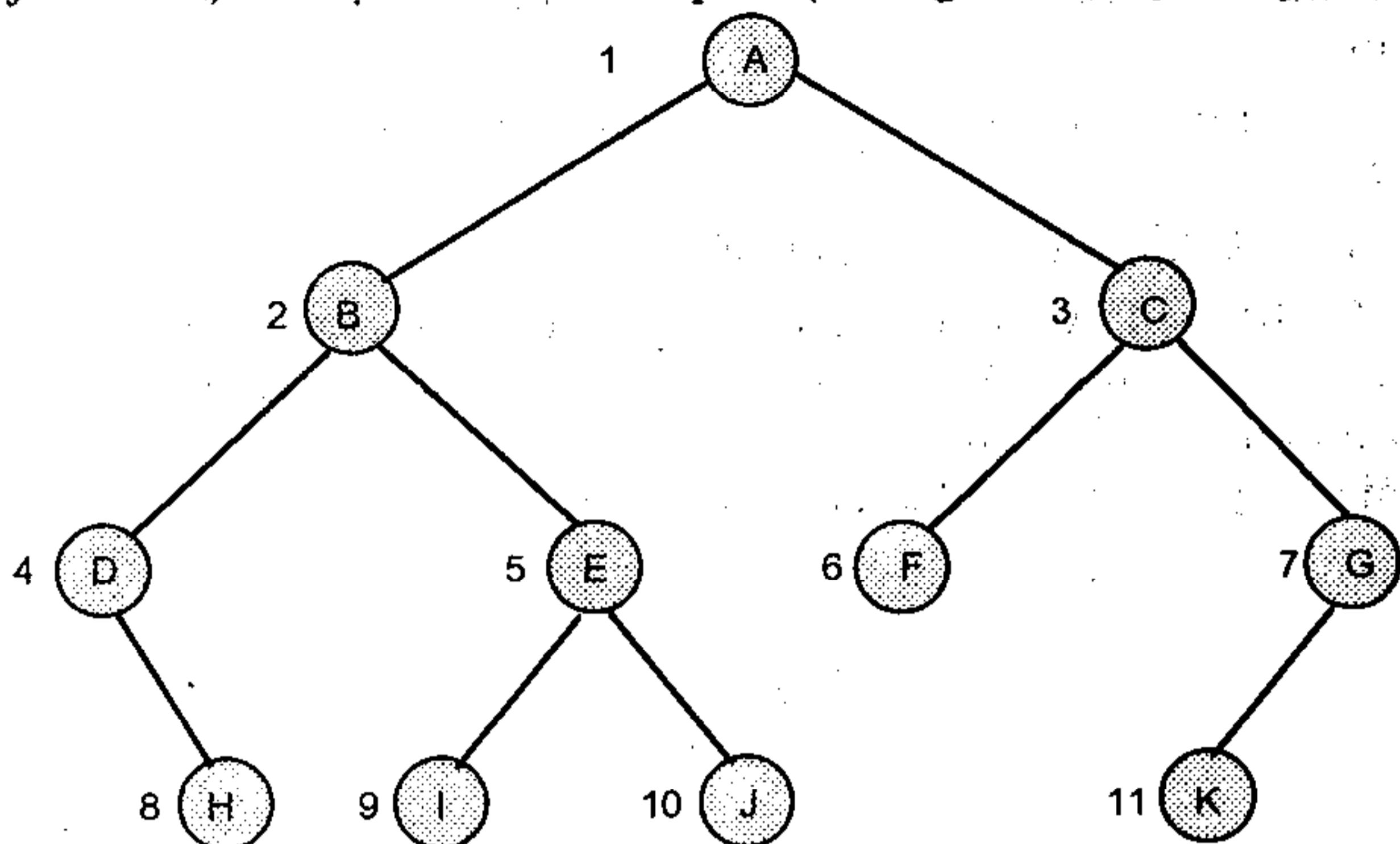
**Hình 4.9.** Cây nhị phân có gốc  $r$ , cây con trái  $T_1$ , cây con phải  $T_2$ .

Cần lưu ý rằng, cây (cây có gốc) và cây nhị phân là hai khái niệm khác nhau. Cây không bao giờ trống, nó luôn luôn chứa ít nhất một đỉnh, mỗi đỉnh có thể không có, có thể có một hay nhiều cây con. Còn cây nhị phân có thể trống, mỗi đỉnh của nó luôn luôn có hai cây con được phân biệt là cây con bên trái và cây con bên phải. Chẳng hạn, hình 4.10 minh họa hai cây nhị phân khác nhau. Cây nhị phân trong hình 4.10a có cây con trái của gốc gồm một đỉnh, còn cây con phải trống. Cây nhị phân trong hình 4.10b có cây con trái của gốc trống, còn cây con phải gồm một đỉnh. Song ở đây ta chỉ có một cây : đó là cây mà gốc của nó chỉ có một cây con gồm một đỉnh.



**Hình 4.10.** Hai cây nhị phân khác nhau

Từ định nghĩa cây nhị phân, ta suy ra rằng, **mỗi đỉnh của cây nhị phân chỉ có nhiều nhất là hai đỉnh con, một đỉnh con bên trái (đó là gốc của cây con trái) và một đỉnh con bên phải (đó là gốc của cây con phải).**



**Hình 4.11.** Một cây nhị phân

### Cài đặt cây nhị phân

Phương pháp tự nhiên nhất để biểu diễn cây nhị phân là chỉ ra đỉnh con trái và đỉnh con phải của mỗi đỉnh.

Ta có thể sử dụng một mảng để lưu giữ các đỉnh của cây nhị phân. Mỗi đỉnh của cây được biểu diễn bởi bản ghi gồm ba trường : trường infor mô tả thông tin gắn với mỗi đỉnh, trường left chỉ đỉnh con trái, trường right chỉ đỉnh con phải. Giả sử các đỉnh của cây được đánh số từ 1 đến max, khi đó cấu trúc dữ liệu biểu diễn cây nhị phân được khai báo như sau.

```

const max = ... ;
type Node = record
    infor : Item ;
    left : 0..max ;
    right : 0..max
  end ;
Tree = array [1.. max] of Node ;
  
```

Hình 4.12 minh họa cấu trúc dữ liệu biểu diễn cây nhị phân trong hình 4.11.

	infor	left	right
1	A	2	3
2	B	4	5
3	C	6	7
4	D	0	8
5	E	9	10
6	F	0	0
7	G	11	0
8	H	0	0
9	I	0	0
10	J	0	0
11	J	0	0
12	K	0	0

Hình 4.12. Cấu trúc dữ liệu biểu diễn cây

Ngoài cách cài đặt cây nhị phân bởi mảng, chúng ta còn có thể sử dụng con trỏ để cài đặt cây nhị phân. Trong cách này mỗi bản ghi biểu diễn một đỉnh của cây chứa hai con trỏ : con trỏ left trỏ tới đỉnh con trái, con trỏ right trỏ tới đỉnh con phải. Tức là ta có khai báo sau.

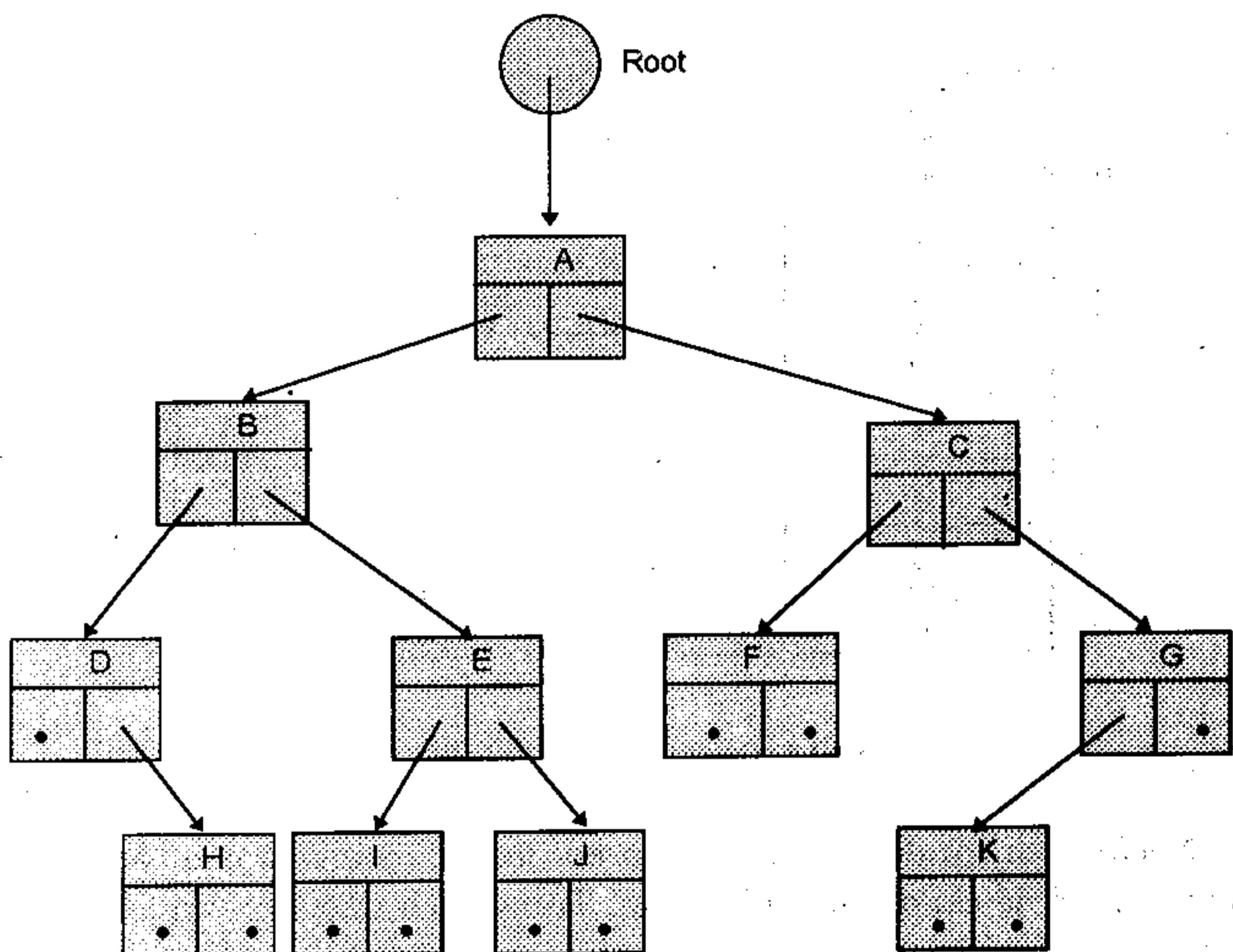
```

type Pointer = ^ Node ;
Node = record
    infor : Item ;
    left : Pointer ;
    right : Pointer ;
end ;
var Root : Pointer ;

```

Biến con trỏ Root trỏ tới gốc của cây. Với cách cài đặt này, cấu trúc dữ liệu biểu diễn cây nhị phân trong hình 4.11 được minh họa bởi hình 4.13.

Từ nay về sau chúng ta sẽ chỉ sử dụng cách biểu diễn cây nhị phân bằng con trỏ. Các phép toán đối với cây nhị phân sau này đều được thể hiện trong cách biểu diễn bằng con trỏ.



Hình 4.13. Cấu trúc dữ liệu biểu diễn cây

### *Đi qua cây nhị phân*

Cũng như đối với cây, trong nhiều ứng dụng ta cần phải đi qua cây nhị phân, thăm tất cả các đỉnh của cây một cách hệ thống, với mỗi đỉnh của cây ta cần thực hiện một nhóm hành động nào đó được mô tả trong thủ tục Visit. Chúng ta thường đi qua cây nhị phân theo một trong ba thứ tự Preorder, Inorder và Postorder. Sau đây là thủ tục để quy trình đi qua cây theo thứ tự Preorder.

```

procedure Preorder (Root : Pointer) ;
begin
  if Root < > nil then
    begin
      Visit (Root) ;
      Preorder (Root^.left) ;
      Preorder (Root^.right) ;
    end ;
  end ;

```

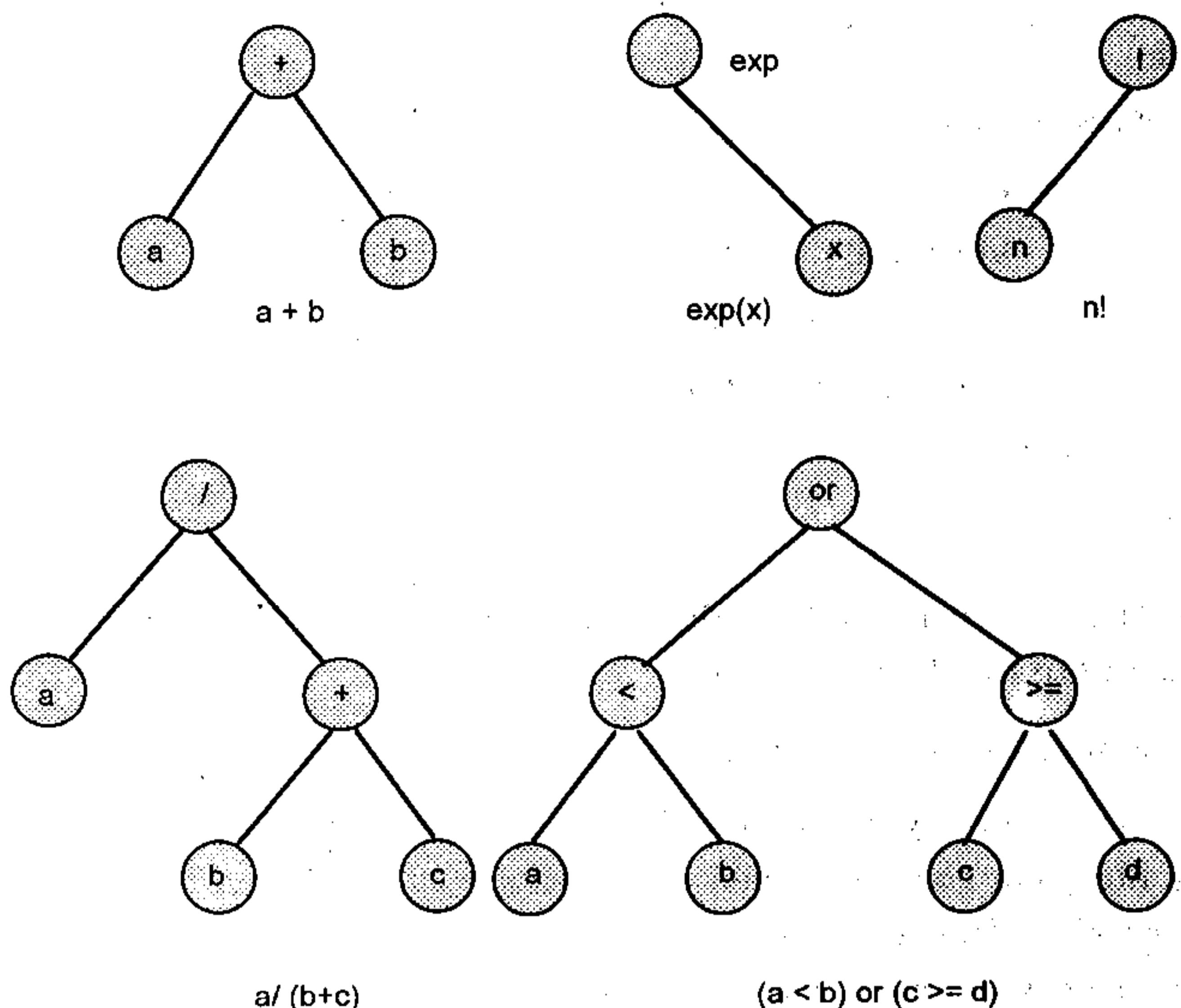
Một cách tương tự, ta có thể viết được các thủ tục để quy đi qua cây theo thứ tự Inorder và Postorder.

#### *Một ví dụ cây nhị phân : cây biểu thức*

Một ví dụ hay về cây nhị phân là cây biểu thức. Cây biểu thức là cây nhị phân gắn nhãn, biểu diễn cấu trúc của một biểu thức (số học hoặc logic). Mỗi phép toán hai toán hạng (chẳng hạn, +, -, \*, /) được biểu diễn bởi cây nhị phân, gốc của nó chứa ký hiệu phép toán, cây con trái biểu diễn toán hạng bên trái, còn cây con phải biểu diễn toán hạng bên phải. Với các phép toán một toán hạng như 'phủ định' hoặc 'lấy giá trị đối', hoặc các hàm chuẩn như exp( ) hoặc cos( ) thì cây con bên trái rỗng. Còn với các phép toán một toán hạng như phép lấy đạo hàm ( ') hoặc hàm giao thừa ( !) thì cây con bên phải rỗng.

Hình 4.14 minh họa một số cây biểu thức.

Ta có nhận xét rằng, nếu đi qua cây biểu thức theo thứ tự Preorder ta sẽ được biểu thức Balan dạng prefix (ký hiệu phép toán đứng trước các toán hạng). Nếu đi qua cây biểu thức theo thứ tự Postorder, ta có biểu thức Balan dạng postfix (ký hiệu phép toán đứng sau các toán hạng); còn theo thứ tự Inorder ta nhận được cách viết thông thường của biểu thức (ký hiệu phép toán đứng giữa hai toán hạng).



Hình 4.14. Một số cây biểu thức

#### 4.5. CÂY TÌM KIẾM NHỊ PHÂN

Cây nhị phân được sử dụng trong nhiều mục đích khác nhau. Tuy nhiên việc sử dụng cây nhị phân để lưu giữ và tìm kiếm thông tin vẫn là một trong những áp dụng quan trọng nhất của cây nhị phân. Trong mục này chúng ta sẽ xét một lớp cây nhị phân đặc biệt, phục vụ cho việc tìm kiếm thông tin, đó là cây tìm kiếm nhị phân.

Trong thực tiễn, một lớp đối tượng nào đó có thể được mô tả bởi một kiểu bản ghi, các trường của bản ghi biểu diễn các thuộc tính của đối tượng. Trong bài toán tìm kiếm thông tin, chúng ta thường quan tâm đến một nhóm thuộc tính nào đó của đối tượng hoàn toàn xác định được

đối tượng. Chúng ta sẽ gọi các thuộc tính này là khoá. Như vậy, khoá là một nhóm thuộc tính của một lớp đối tượng sao cho hai đối tượng khác nhau cần phải có các giá trị khác nhau trên nhóm thuộc tính đó. Từ nay về sau ta giả thiết rằng, thông tin gắn với mỗi đỉnh của cây nhị phân là khoá của đối tượng nào đó. Do đó mỗi đỉnh của cây nhị phân được biểu diễn bởi bản ghi kiểu Node có cấu trúc như sau.

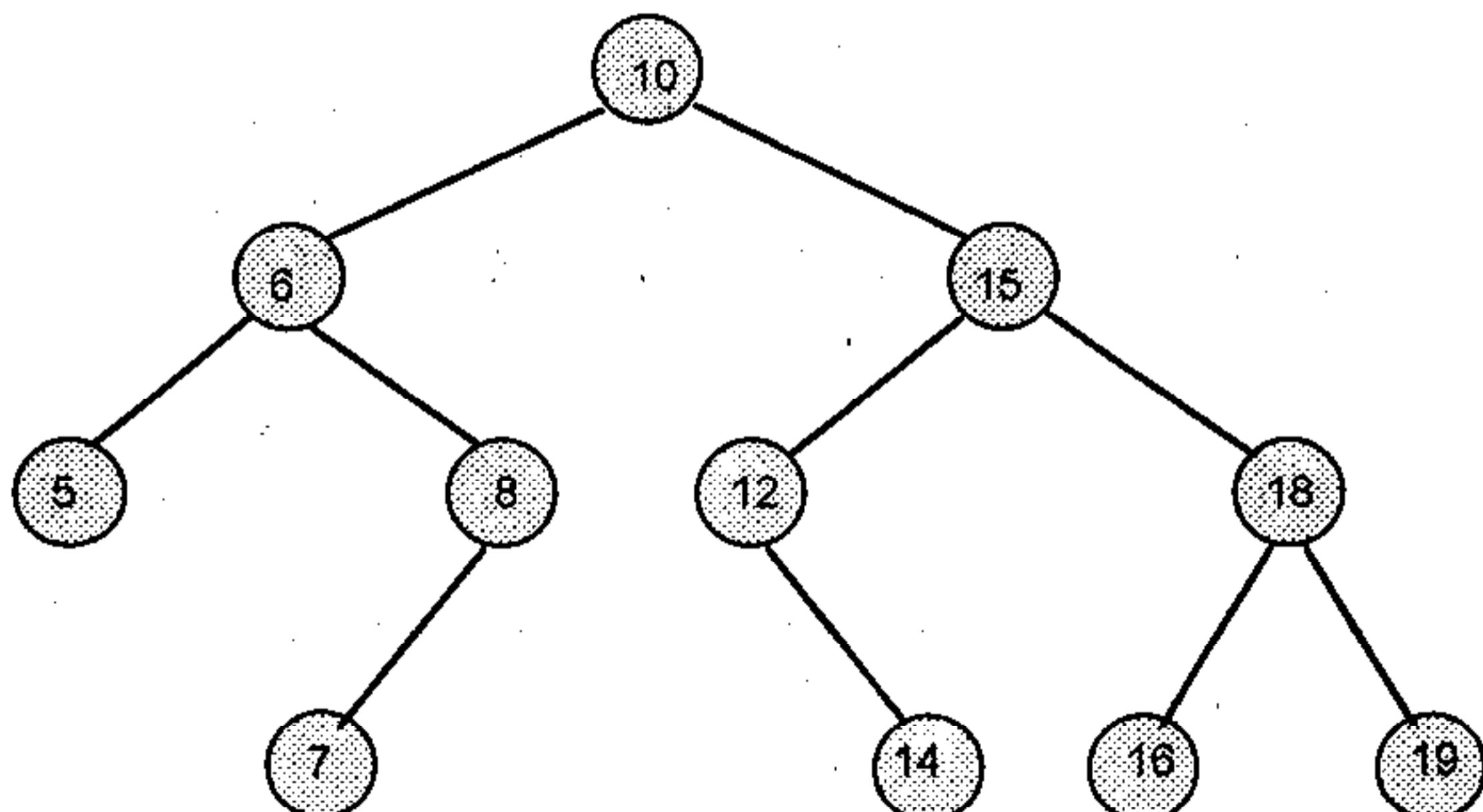
```
type pointer = ^Node;
Node = record
    key : keytype;
    left : pointer;
    right : pointer;
end;
```

Giả sử kiểu của khoá (keytype) là một kiểu có thứ tự, chẳng hạn kiểu nguyên, thực, ký tự, xâu ký tự. Khi đó cây tìm kiếm nhị phân được định nghĩa như sau.

Cây tìm kiếm nhị phân là cây nhị phân hoặc trống, hoặc thoả mãn các điều kiện sau:

1. Khoá của các đỉnh thuộc cây con trái nhỏ hơn khoá của gốc
2. Khoá của gốc nhỏ hơn khoá của các đỉnh thuộc cây con phải của gốc.
3. Cây con trái và cây con phải của gốc cũng là cây tìm kiếm nhị phân.

Hình 4.15 biểu diễn một cây tìm kiếm nhị phân, trong đó khoá của các đỉnh là các số nguyên.



**Hình 4.15.** Một cây tìm kiếm nhị phân

## Các phép toán trên cây tìm kiếm nhị phân

### 1. Tìm kiếm

Tìm kiếm trên cây là một trong các phép toán quan trọng nhất đối với cây tìm kiếm nhị phân. Giả sử mỗi đỉnh của cây được biểu diễn bởi bản ghi có kiểu Node đã xác định ở trên, biến con trỏ Root trỏ tới gốc cây và x là một giá trị khoá cho trước. Vấn đề là, tìm xem trên cây có chứa đỉnh với khoá là x hay không. Sau đây chúng ta sẽ viết các thủ tục tìm kiếm.

Trong thủ tục tìm kiếm đệ qui dưới đây, chúng ta sẽ sử dụng tham biến P. Đó là con trỏ chạy trên các đỉnh của cây, bắt đầu từ gốc, nếu tìm kiếm thành công thì P sẽ trỏ vào đỉnh với khoá x, còn nếu tìm kiếm không kết quả (tức là, trong cây không có đỉnh nào với khoá x) thì P = nil.

```
procedure Search (x : Key Type ; Root : pointer ; var P : pointer) ;
begin
    P := Root ;
    if P < > nil then
        if x < P^.key then Search (x, P^.left, P)
        else if x > P^.key then Search(x,P^.right, P)
    end ;
```

Sau đây ta sẽ trình bày thủ tục tìm kiếm không đệ qui. Trong thủ tục này, ta sẽ sử dụng biến địa phương found có kiểu boolean để điều khiển vòng lặp, nó có giá trị ban đầu là false. Nếu tìm kiếm thành công thì found nhận giá trị true, vòng lặp kết thúc và P trỏ vào đỉnh tìm thấy. Còn nếu tìm kiếm không kết quả thì giá trị của found vẫn là false và giá trị của P là nil.

```
procedure Search (x : keytype, Root : pointer ; var P : pointer) ;
var found : boolean ;
begin
    P := Root ;
```

```

found : = false ;
while (P < > nil) and (not found) do
    if P^.key = x then found : = true
    else if x < P^.key then P : = P^.left
        else P : = P^.right ;
end ;

```

## 2. Xen vào cây tìm kiếm

Khi cập nhật thông tin được tổ chức dưới dạng cây tìm kiếm nhị phân, ta thường xuyên phải thực hiện các phép toán xen vào và loại bỏ khỏi cây tìm kiếm. Chúng ta cần phải xen vào cây tìm kiếm nhị phân một đỉnh mới có khoá x cho trước, sao cho sau khi xen vào cây vẫn còn là cây tìm kiếm nhị phân.

Đầu tiên ta viết thủ tục đệ qui xen vào cây tìm kiếm.

```

procedure Insert (var Root : pointer ; x : keytype) ;
    {xen vào.cây gốc Root đỉnh mới với khoá là x}
var Q : pointer ;
begin
    if Root = nil then
        begin
            new (Q) ; {tạo ra đỉnh mới }
            Q^.key : = x ;
            Q^.left : = nil ;
            Q^.right : = nil ;
            Root : = Q ;
        end
    else with Root^ do
        if x < key then Insert (left, x)
        else if x > key then Insert (right, x)
end ;

```

Sau đây ta sẽ viết thủ tục không đệ qui xen vào cây tìm kiếm. Trong thủ tục này ta sẽ sử dụng biến con trỏ địa phương P chạy trên các đỉnh của cây bắt đầu từ gốc. Khi đang ở một đỉnh nào đó, P sẽ chạy xuống đỉnh con trái (phải) tùy theo khoá của đỉnh đó lớn hơn (nhỏ hơn) khoá x.

Nếu tại một đỉnh nào đó, P cần phải chạy xuống đỉnh con trái (phải), nhưng đỉnh đó không có con trái (phải). thì ta 'treo' đỉnh mới vào bên trái (phải) đỉnh đó. Điều kiện  $P = \text{nil}$  sẽ kết thúc vòng lặp.

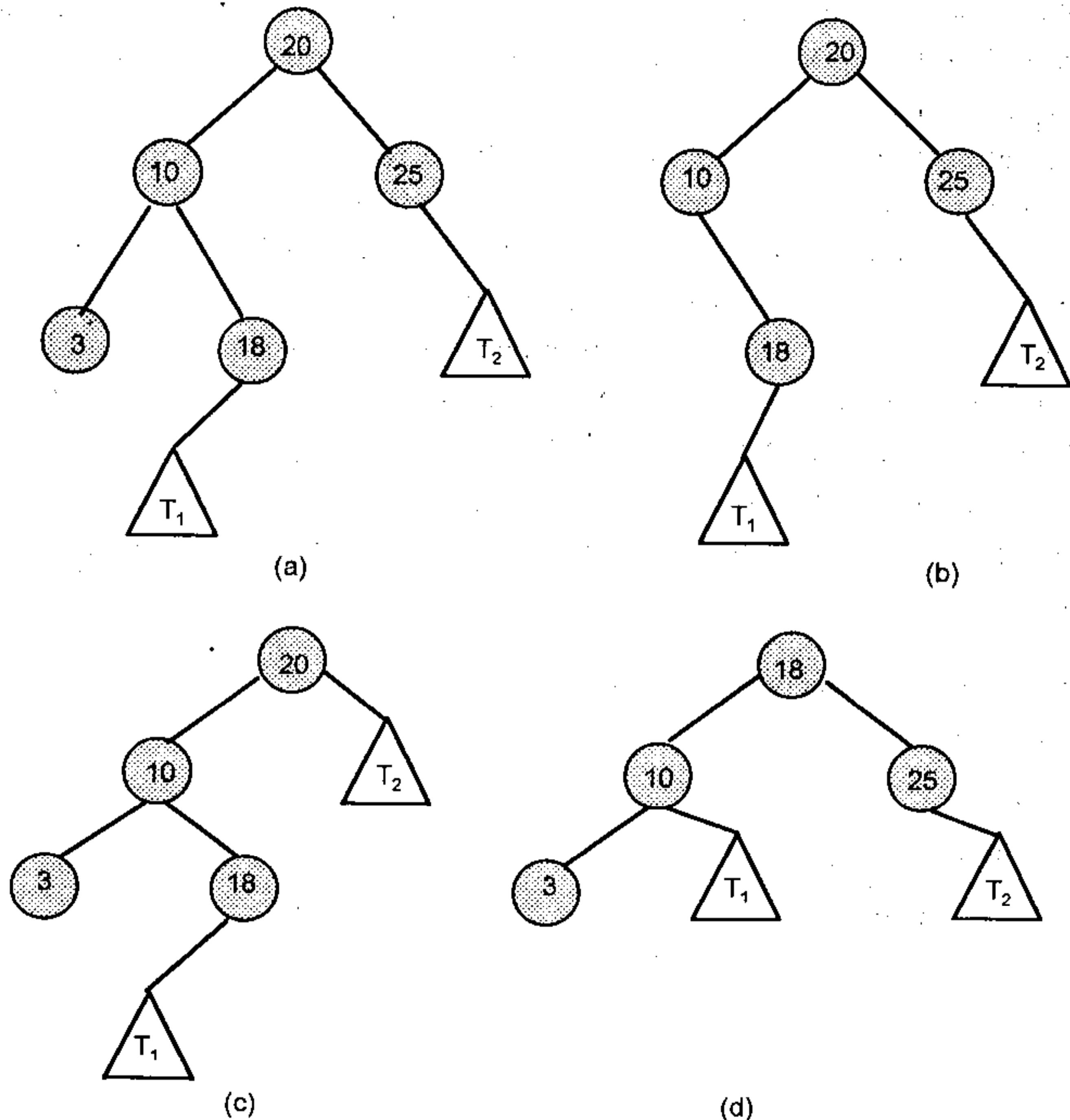
```

procedure      Insert (var Root : pointer ; x : keytype) ;
var P, Q : pointer ;
begin
    new (Q) ;
    Q^.key := x ;
    Q^.left := nil ;
    Q^.right := nil ;
    if Root = nil then Root := Q
    else begin
        P := Root ;
        while P < > nil do
            if x < P^.key then
                if P^.left < > nil then P := P^.left
                else begin
                    P^.left := Q ;
                    P := nil
                end
            else if x > P^.key then
                if P^.right < > nil then P := P^.right
                else begin
                    P^.right := Q ;
                    P := nil
                end
            else P := nil
        end
    end ;

```

### 3. Loại bỏ khóa, cây tìm kiếm

Đối lập với phép toán xen vào là phép toán loại bỏ. Chúng ta cần phải loại bỏ khỏi cây tìm kiếm một đỉnh có khóa  $x$  cho trước, sao cho sau khi loại bỏ cây vẫn còn là cây tìm kiếm nhị phân.



Hình 4.16

- a) Cây T ban đầu
- b) Cây T sau khi loại đỉnh 3 (lá)
- c) Cây T sau khi loại đỉnh 25 (chỉ có một cây con khác trống)
- d) Cây T sau khi loại đỉnh 20 (cả hai cây con khác trống).

Việc loại bỏ một đỉnh khỏi cây tìm kiếm không đơn giản như việc xén một đỉnh mới vào cây. Nếu đỉnh cần loại bỏ là lá thì rất đơn giản : ta chỉ cần "cắt" lá đó đi. Nếu đỉnh cần loại bỏ có một trong hai cây con là cây trống, ta chỉ cần "treo" cây con khác trống vào vị trí của đỉnh bị loại. Tuy nhiên vấn đề sẽ phức tạp hơn nếu cả hai cây con của đỉnh cần loại đều khác trống. Vấn đề đặt ra là, ta phải xử lý như thế nào đối với hai cây con của đỉnh bị loại. Ta có nhận xét rằng, trong một cây tìm kiếm nhị phân khác trống bất kỳ, đỉnh có khoá nhỏ nhất là đỉnh ngoài cùng bên trái, đỉnh có khoá lớn nhất là đỉnh ngoài cùng bên phải. Do đó, khi đỉnh cần loại bỏ có cả hai cây con khác trống, ta làm như sau : thay khoá của đỉnh cần loại bỏ bởi khoá của đỉnh ngoài cùng bên phải của cây con trái (hoặc đỉnh ngoài cùng bên trái của cây con phải), rồi loại bỏ đỉnh ngoài cùng bên phải của cây con trái (hoặc đỉnh ngoài cùng bên trái của cây con phải) (xem hình 4.16)

Thuật toán loại bỏ đã trình bày được mô tả bởi thủ tục Del. Thủ tục này loại khỏi cây đỉnh mà con trỏ P trỏ tới, trong đó P là con trỏ liên kết trong cây.

```

procedure Del (var P : pointer) ;
var Q, Q1 : pointer ;
begin
  if P^.right = nil then
    begin
      Q := P ;
      P := P^.left ;
    end else
    if P^.left = nil then
      begin
        Q := P ;
        P := P^.right
      end else

```

```

begin Q := P^.left ;
    if Q^.right = nil then {xem hình 4.17a}
        begin
            P^.key := Q^.key ;
            P^.left := Q^.left
        end else
        begin
            repeat
                Q1 := Q ;
                Q := Q^.right
            until Q^.right = nil ;
            P^.key := Q^.key ;
            Q1^.right := Q^.left {xem hình 4.17b}
        end ;
    end ;
    dispose (Q);
end ;

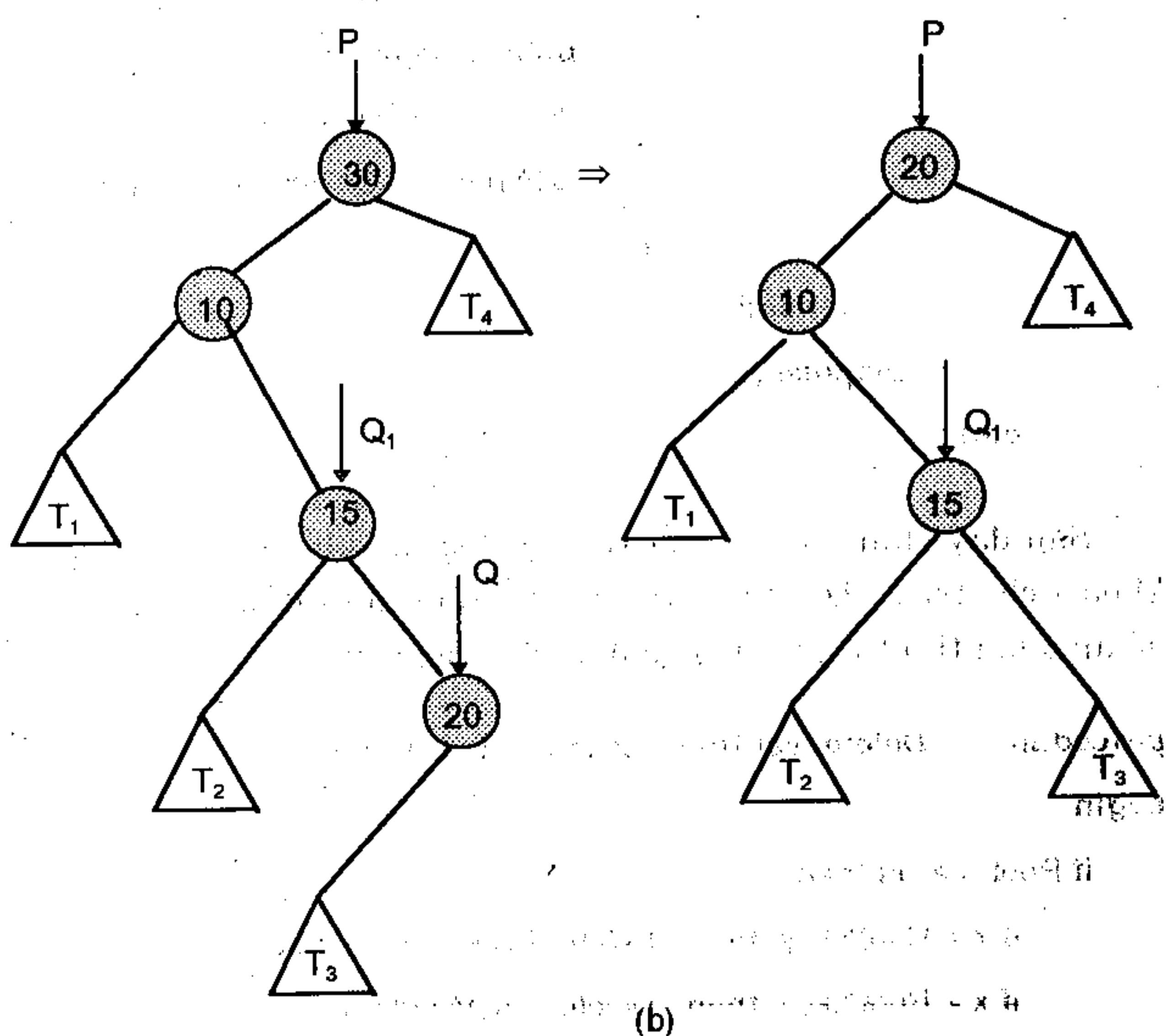
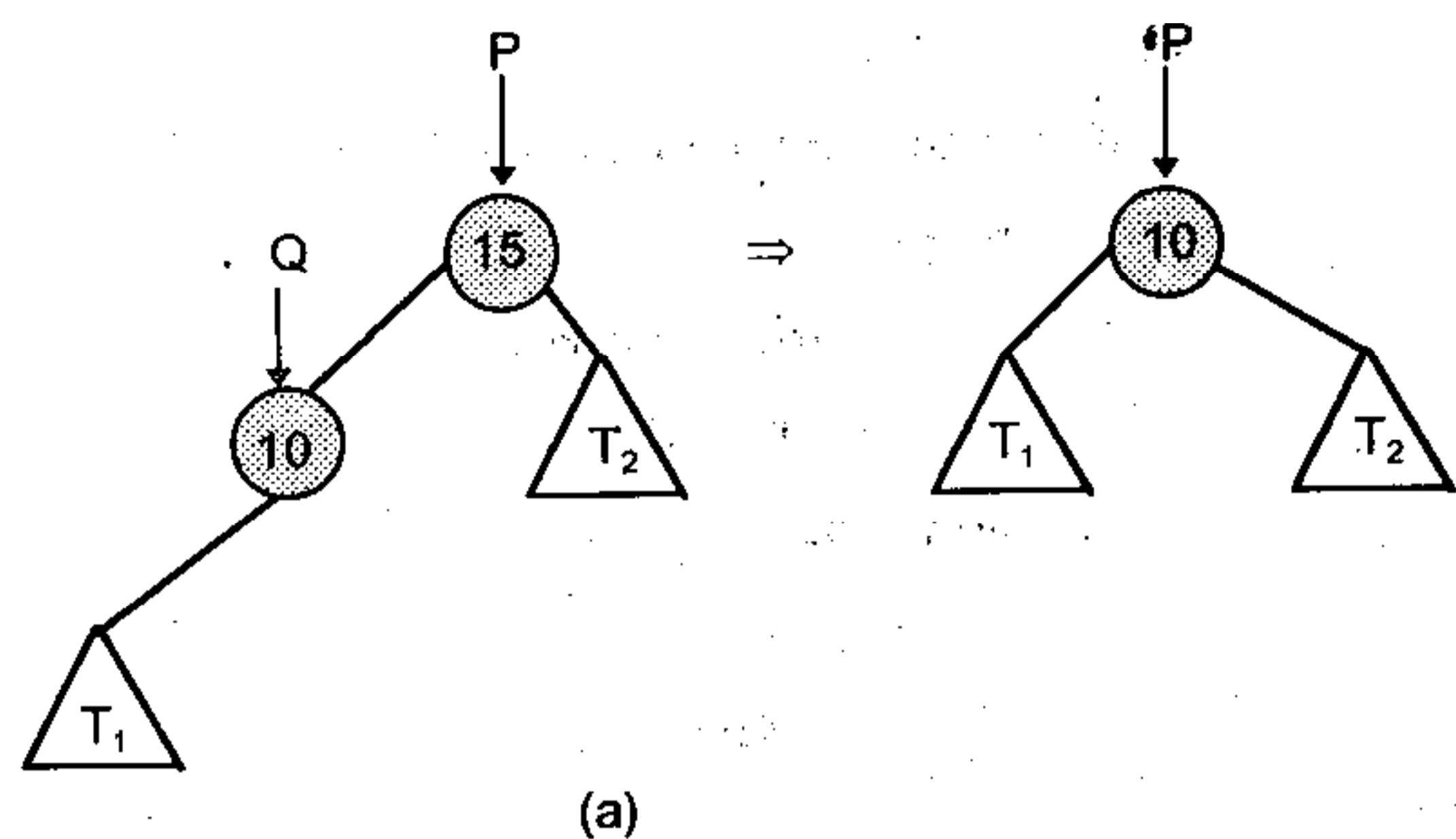
```

Sau đây chúng ta sẽ viết thủ tục loại bỏ khỏi cây gốc Root đỉnh có khoá x cho trước. Đó là thủ tục đệ qui, nó tìm ra đỉnh có khoá x, rồi sau đó áp dụng thủ tục Del để loại đỉnh đó khỏi cây.

```

procedure Delete (var Root : pointer ; x : keytype) ;
begin
    if Root < > nil then
        if x < Root^.key then Delete (Root^.left, x) else
            if x > Root^.key then Delete (Root^.right, x)
        else Del (Root) ;
    end ;

```

**Hình 4.17**

## 4.6. THỜI GIAN THỰC HIỆN CÁC PHÉP TOÁN TRÊN CÂY TÌM KIẾM NHỊ PHÂN

Trong mục này chúng ta sẽ đánh giá thời gian trung bình để thực hiện các phép toán trên cây tìm kiếm nhị phân. Ta có nhận xét rằng, thời gian thực hiện phép tìm kiếm là số phép so sánh giá trị khoá x cho trước với khoá của các đỉnh nằm trên đường đi từ gốc tới một đỉnh nào đó trong cây. Do đó thời gian thực hiện phép tìm kiếm (cũng thế, thời gian thực hiện các phép xen vào và loại bỏ) là độ dài của đường đi từ gốc tới một đỉnh nào đó trong cây.

Dễ dàng thấy rằng, trong trường hợp tốt nhất, cây tìm kiếm nhị phân với  $n$  đỉnh là cây đầy đủ (tất cả các đỉnh đều có hai con trừ các đỉnh ở mức thấp nhất), thì độ cao của cây  $xấp xỉ bằng$   $\log n$  (ta ký hiệu  $\log = \log_2 n$ ). Thực vậy, gọi mức thấp nhất là  $k$ , ta có

$$\begin{aligned} 1 + 2 + 2^2 + \dots + 2^{k-1} &< n \\ 1 + 2 + \dots + 2^k &\geq n \end{aligned}$$

hay  $2^k - 1 < n$  và  $2^{k+1} - 1 \geq n$ . Từ đó, ta có  $\log(n+1) - 1 \leq k < \log(n+1)$ . Tức là  $k \approx \log n$ .

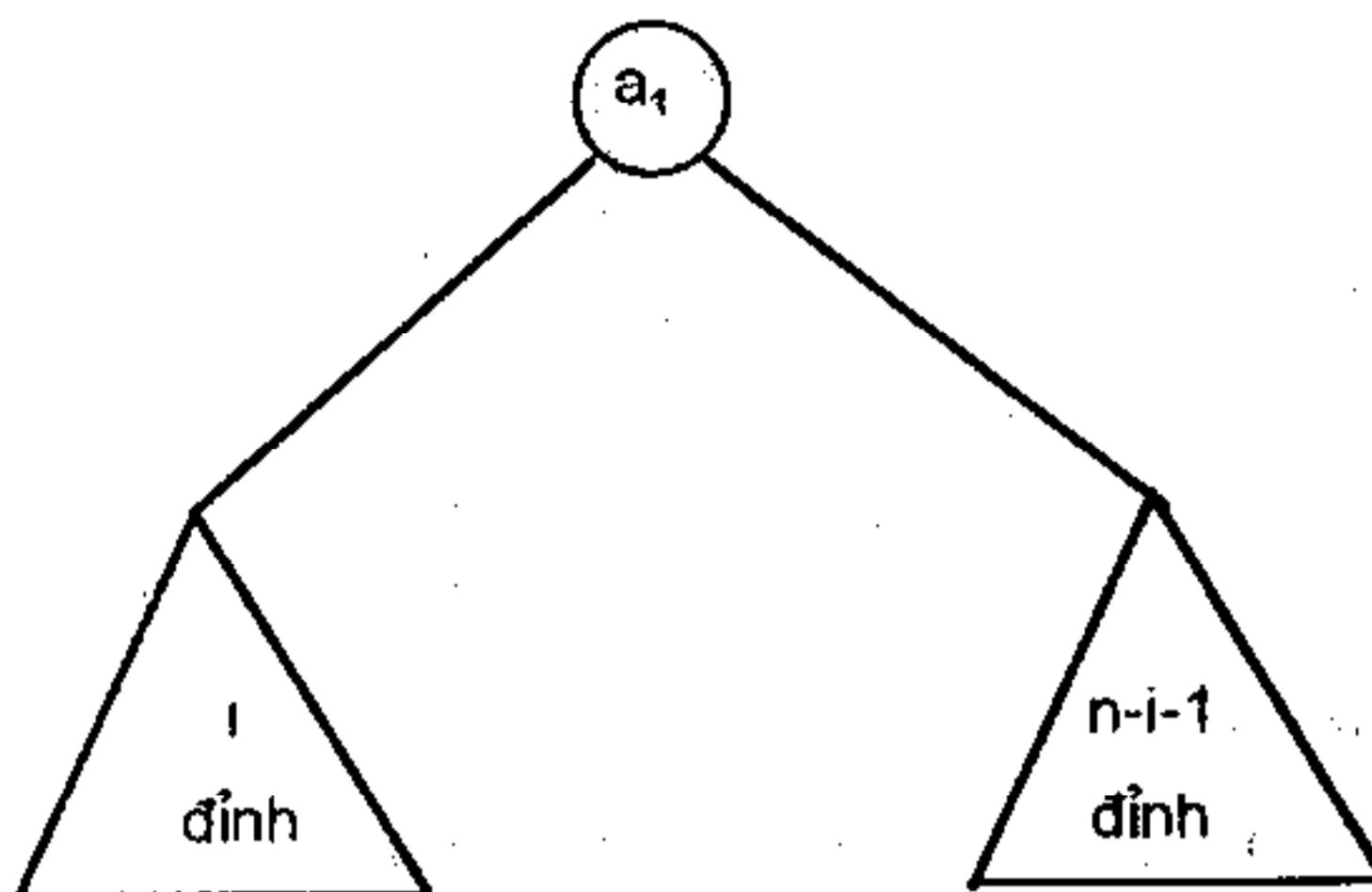
Do đó trong trường hợp này, thời gian thực hiện các phép toán là  $O(\log n)$ .

Trong trường hợp xấu nhất, cây tìm kiếm nhị phân suy biến thành danh sách liên kết. Điều này xảy ra, chẳng hạn, khi ta xây dựng cây bằng cách xen vào cây trống lần lượt  $n$  đỉnh với các giá trị khoá đã được sắp xếp theo thứ tự tăng dần. Khi đó ta có cây mà mỗi đỉnh đều có cây con trái trống, tức là cây trở thành danh sách liên kết. Trong trường hợp này thời gian thực hiện các phép toán là  $O(n)$ .

Câu hỏi được đặt ra là, thời gian trung bình để thực hiện các phép toán sẽ như thế nào, trong trường hợp tổng quát cây tìm kiếm là cây "ngẫu nhiên", tức là cây được xây dựng nên từ cây trống bằng cách xen vào  $n$  đỉnh với các giá trị khoá được sắp xếp một cách ngẫu nhiên.

Giả thiết cây tìm kiếm  $T$  được xây dựng nên từ cây rỗng bằng cách xen vào các đỉnh có khoá lần lượt là  $a_1, a_2, \dots, a_n$ , trong đó dãy các giá trị khoá trên được sắp xếp một cách ngẫu nhiên. Giả sử trong dãy giá trị khoá trên có  $i$  phần tử nhỏ hơn  $a_1$  và  $n - i - 1$  phần tử lớn hơn  $a_1$ . Khi

đó cây con trái của gốc có  $i$  đỉnh, và cây con phải có  $n - i - 1$  đỉnh (xem hình 4.18)



**Hình 4.18.** Cây tìm kiếm nhị phân "ngẫu nhiên"

Gọi  $S(n)$  là độ dài trung bình của đường đi từ gốc tới đỉnh bất kỳ trong cây  $n$  đỉnh. Ta có  $S(1) = 0$ . Giả sử  $S(i)$  là độ dài trung bình của đường đi ở cây con trái,  $S(n-i-1)$  là độ dài trung bình của đường đi ở cây con phải. Do đó độ dài trung bình của đường đi trong cây  $T$  với cây con trái của gốc có  $i$  đỉnh là :

$$\frac{i}{n} (S(i) + 1) + \frac{n - i - 1}{n} (S(n - i - 1) + 1)$$

Bằng cách lấy trung bình cộng của tổng trên với mọi  $i$  đi từ 0 tới  $n-1$ , ta nhận được độ dài trung bình của đường đi trong cây  $n$  đỉnh là

$$S(n) = \frac{1}{n} \sum_{i=0}^{n-1} \left[ \frac{i}{n} (S(i) + 1) + \frac{n - i - 1}{n} (S(n - i - 1) + 1) \right]$$

Trong tổng trên, lưu ý rằng

$$\sum_{i=0}^{n-1} iS(i) = \sum_{n=0}^{n-1} (n - i - 1)S(n - i - 1)$$

Do đó, ta có

$$S(n) = \frac{n - 1}{n} + \frac{2}{n^2} \sum_{i=0}^{n-1} iS(i) \quad (1)$$

Từ (1) ta nhận được các đẳng thức sau

$$S(n) = \frac{n-1}{n} + \frac{2(n-1)}{n^2} S(n-1) + \frac{2}{n^2} \sum_{i=0}^{n-2} iS(i) \quad (2)$$

$$S(n-1) = \frac{n-2}{n-1} + \frac{2}{(n-1)} \sum_{i=0}^{n-2} iS(i). \quad (3)$$

Từ (3), ta có

$$\frac{2}{n^2} \sum_{i=0}^{n-2} iS(i) = \frac{(n-1)^2}{n^2} S(n-1) - \frac{(n-1)(n-2)}{n^2} \quad (4)$$

Thay (4) vào (2), ta nhận được

$$S(n) = \frac{n^2 - 1}{n^2} S(n-1) + \frac{2(n-1)}{n}$$

Từ đó, ta có đánh giá

$$S(n) < S(n-1) + \frac{2}{n}$$

Bằng cách thế liên tiếp, ta nhận được

$$S(n) < 2H_n - 2 \quad (5)$$

Trong đó  $H_n$  là hàm điều hoà

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

Theo công thức Ole (với hằng số  $\gamma \approx 0,577$ ), ta có

$$H_n = \gamma + \ln(n) + \frac{1}{12n^2} + \dots \quad (6)$$

Từ (5) và (6), ta có  $S(n) = O(\log n)$ . Như vậy độ dài trung bình của đường đi từ gốc tới đỉnh bất kỳ trong cây tìm kiếm nhị phân với  $n$  đỉnh là  $O(\log n)$ . Do đó chúng ta có thể kết luận rằng, thời gian trung bình để thực hiện các phép toán trên cây tìm kiếm nhị phân là  $O(\log n)$ .

## 4.7. CÂY CÂN BẰNG

Giả sử ta có một tập hợp dữ liệu nào đó. Vấn đề đặt ra là, ta phải tổ chức các dữ liệu đó như thế nào sao cho việc cập nhật thông tin (tìm

kiếm, thêm vào và loại bỏ) được nhanh chóng. Trong mục trước ta đã thấy rằng, nếu tổ chức dữ liệu dưới dạng cây tìm kiếm nhị phân thì thời gian trung bình thực hiện các phép toán là  $O(\log n)$ . Trong nhiều áp dụng chúng ta cần thường xuyên thực hiện các phép toán xen vào và loại bỏ khỏi cây tìm kiếm. Điều đó làm cho cây có thể trở nên rất "lệch", trường hợp xấu nhất nó có thể suy biến thành danh sách liên kết. Đối với những cây tìm kiếm lệch, việc thực hiện các phép toán sẽ kém hiệu quả.

Trong mục này chúng ta sẽ nghiên cứu một lớp cây tìm kiếm đặc biệt, trong đó các phép toán tìm kiếm, xen vào và loại bỏ đối với cây nimbly luôn luôn được thực hiện trong thời gian  $O(\log n)$ , ngay cả trong trường hợp xấu nhất.

Lớp cây này được các nhà toán học Nga G.M. Adelson Velskii và E.M. Lendis đưa ra vào năm 1962.

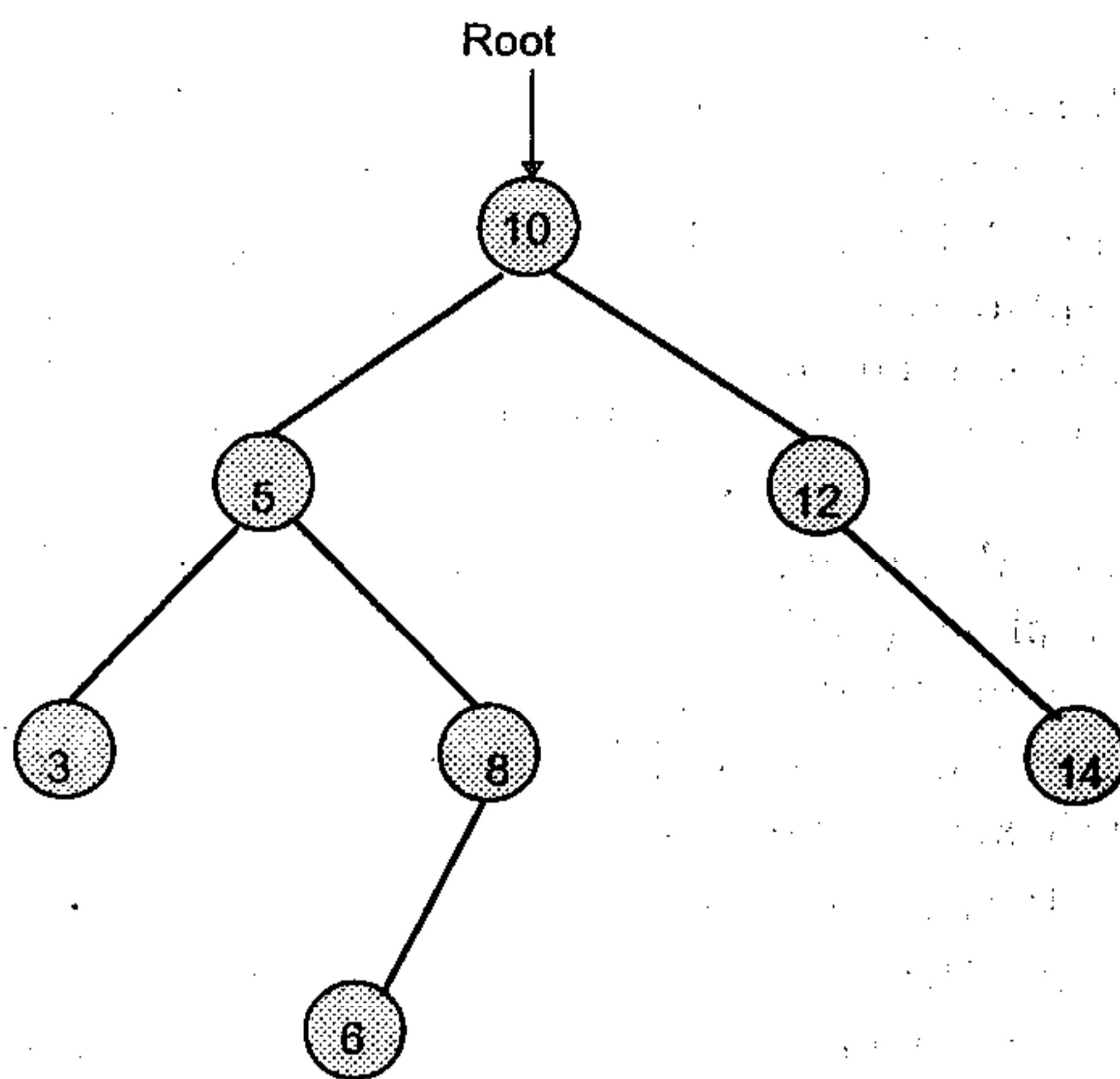
Cây cân bằng (hay còn gọi là AVL-cây) là cây tìm kiếm nhị phân sao cho tại mỗi đỉnh của cây, độ cao của cây con trái và cây con phải khác nhau không quá một.

Để biểu diễn cây cân bằng, ta thêm vào mỗi bản ghi mô tả đỉnh của cây một trường mới bal (balance : cân bằng). Trường này nhận một trong ba giá trị LH (Left Height : cao bên trái), EH (Equal Height : hai cây con cao bằng nhau), RH (Right Height : cao bên phải). Ta sẽ nói đỉnh của cây ở trạng thái cân bằng LH, EH, hoặc RH.

Ta có thể khai báo cấu trúc dữ liệu biểu diễn cây cân bằng như sau.

```

type pointer = ^ Node;
Node = record
    key : keytype ;
    left, right : pointer ;
    bal : (LH, EH, RH)
end ;
var Root : pointer ;
;
```

**Hình 4.19.** Một cây cân bằng.

Sau đây chúng ta sẽ xét các phép toán xen vào và loại bỏ trên cây cân bằng

### **1. Xen vào cây cân bằng**

Việc xen vào cây cân bằng một đỉnh mới với khoá  $x$  cho trước được thực hiện bằng cách sau. Đầu tiên ta áp dụng thuật toán xen vào cây tìm kiếm, sau đó "cân bằng" lại các đỉnh mà tại đó tính cân bằng bị phá vỡ (độ cao của hai cây con khác nhau 2).

Giả sử ta cần xen một đỉnh mới vào cây gốc  $P$  ( $P$  là con trỏ trả tới gốc cây). Có thể xảy ra những trường hợp sau.

1.  $P = \text{nil}$  (cây trống). Khi xen vào đỉnh mới, ta sẽ được cây cân bằng và  $P^{\wedge}.\text{bal} = \text{EH}$ .
2.  $P \neq \text{nil}$  và  $P^{\wedge}.\text{bal} = \text{EH}$ . Trong trường hợp này, khi **xen đỉnh mới** vào cây con trái hoặc cây con phải của  $P$ , dù có làm tăng độ cao của cây con, thì tính cân bằng của đỉnh  $P$  vẫn không bị phá vỡ.

3.  $P \neq \text{nil}$  và  $P^{\wedge}.\text{bal} = \text{RH}$  ( $P^{\wedge}.\text{bal} = \text{LH}$ ). Trong trường hợp này, nếu ta xen đỉnh mới vào cây con trái (cây con phải), thì dù có làm tăng độ cao của cây con, đỉnh P vẫn ở trạng thái cân bằng.

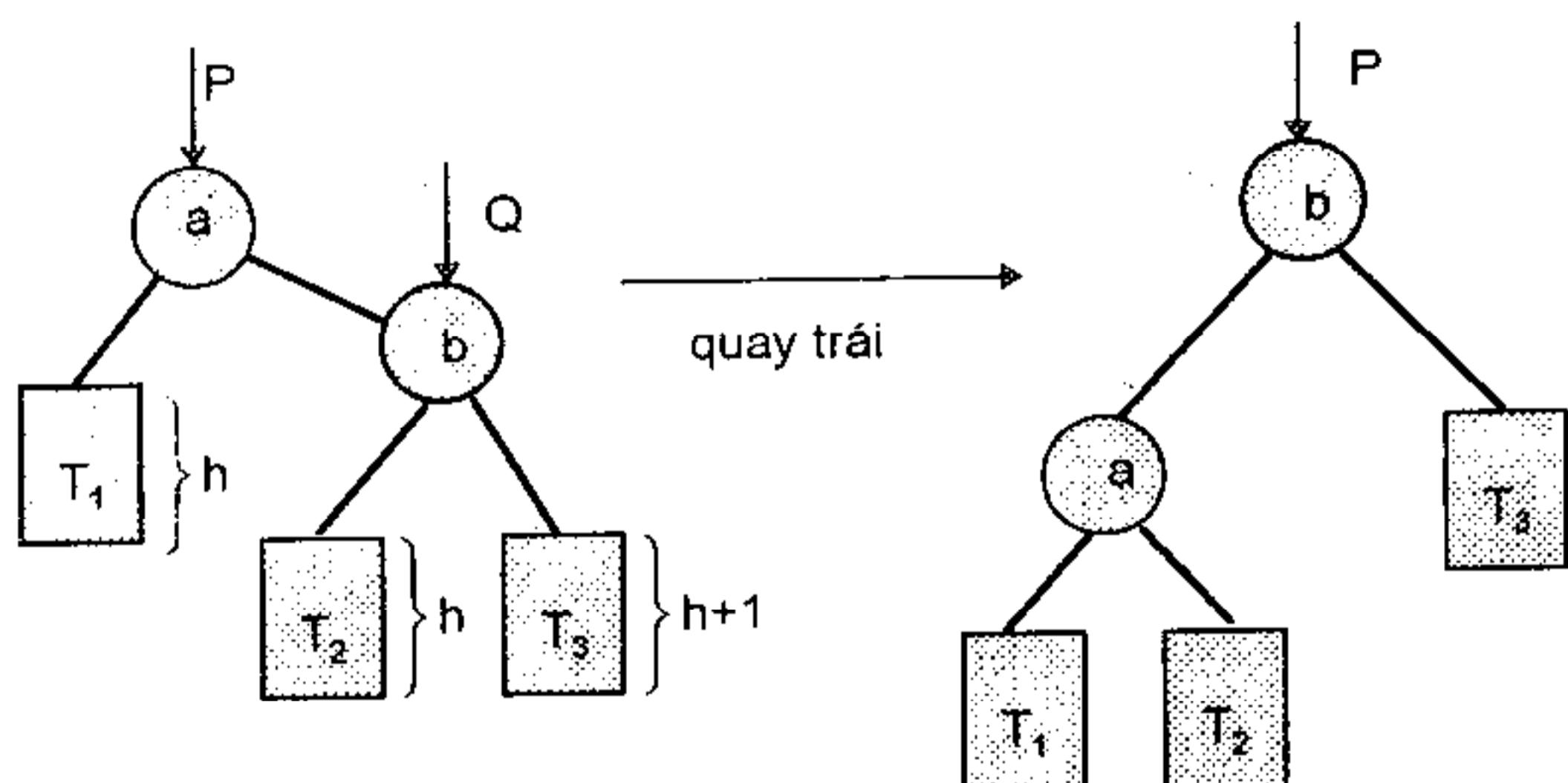
4.  $P \neq \text{nil}$  và  $P^{\wedge}.\text{bal} = \text{RH}$  ( $P^{\wedge}.\text{bal} = \text{LH}$ ). Giả sử đỉnh mới được xen vào cây con phải (cây con trái) và việc xen vào làm tăng độ cao của cây con. Trong trường hợp này, tính cân bằng tại đỉnh bị phá vỡ : cây con phải của P cao hơn cây con trái 2 (cây con trái của P cao hơn cây con phải 2).

Như vậy chỉ có trường hợp 4) là phá vỡ tính cân bằng tại P. Sau đây ta sẽ đưa ra phương pháp biến đổi cây P để nó trở nên cân bằng tại P, khi tính cân bằng tại P bị vi phạm.

Giả sử  $P^{\wedge}.\text{bal} = \text{RH}$  và đỉnh mới được xen vào cây con phải của P, đồng thời việc xen vào làm tăng độ cao của cây con phải đó.

Gọi  $Q = P^{\wedge}.\text{right}$ . Xét các khả năng sau.

a)  $Q^{\wedge}.\text{bal} = \text{RH}$  (cây con gốc Q cao bên phải). Trong trường hợp này, ta biến đổi cây P bằng phép quay trái (Rotateleft) (xem hình 4.20). Sau phép quay này cả hai đỉnh a và b đều ở trạng thái cân bằng EH.



Hình 4. 20. Quay trái cây P

```

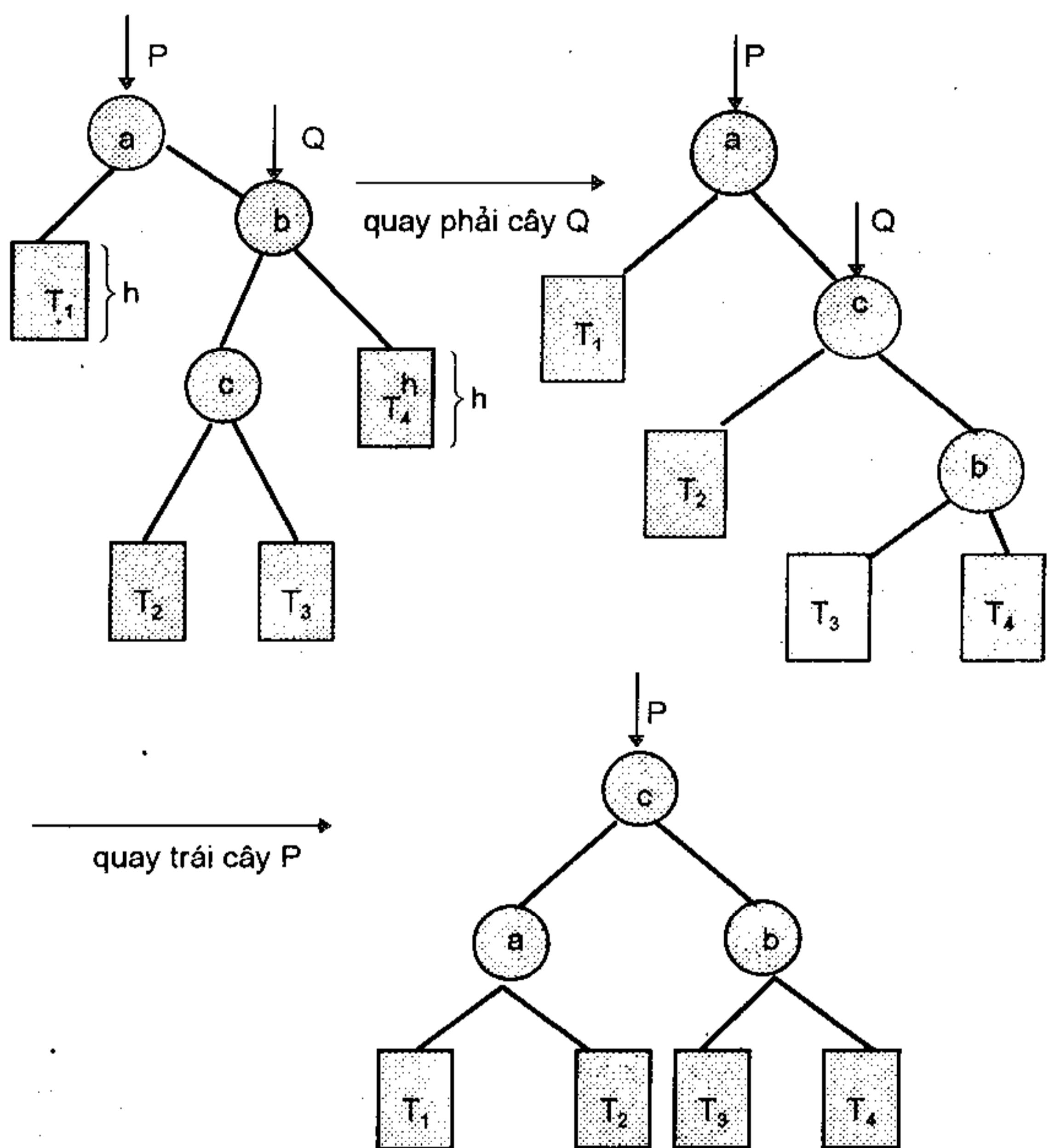
procedure      RotateLeft (var P : pointer);
    {quay trái cây gốc P }
var  Q : pointer ;
begin
    if P < > nil then
        if P^.\right < > nil then

```

```

begin
    Q := P^.right ;
    P^.right := Q^.left ;
    Q^.left := P ;
    P := Q;
end ;
end ;

```



Hình 4.21

b)  $Q^{\text{bal}} = \text{LH}$  (cây con gốc Q cao bên trái). Trong trường hợp này, đầu tiên ta quay cây con gốc Q sang phải (RotateRight), sau đó quay gốc P sang trái (xem hình 4.21). Cần chú ý rằng, hai cây con  $T_1$  và  $T_4$  có độ

cao h, còn ít nhất một trong hai cây con  $T_2$  và  $T_3$  phải có độ cao h. Do đó, sau hai phép quay đỉnh c ở trạng thái cân bằng EH, còn đỉnh a sẽ ở trạng thái cân bằng EH hay LH tùy thuộc vào  $T_2$  có độ cao h hay h-1. Tương tự, đỉnh b sẽ ở trạng thái cân bằng EH hay RH tùy theo  $T_3$  có độ cao h hay h-1.

c)  $Q^.bal = EH$ . Ta đã giả thiết đỉnh mới được xen vào cây con phải của P và sau khi xen vào độ cao của cây con phải tăng lên. Nếu sau khi xen mà  $Q^.bal = EH$  thì độ cao của cây Q không thể tăng lên được. Do đó, trường hợp này không xảy ra.

Việc thiết lập lại tính cân bằng của đỉnh P bằng phương pháp biến đổi cây mà ta đã trình bày ở trên được mô tả trong thủ tục RightBalance (cân bằng bên phải). Thủ tục này sử dụng các thủ tục RotateLeft và RotateRight (bạn đọc tự viết lấy thủ tục này) để thực hiện các phép biến đổi cây được chỉ ra trong hình 4.20 và hình 4.21.

```

procedure      RightBalance (var P : pointer) ;
var Q,R : pointer
begin
  Q := P^.right ;
  case Q^.bal of
    RH : begin
      P^.bal := EH ;
      Q^.bal := EH ;
      RotateLeft (P)
    end ;
    LH : begin
      R := Q^.left ;
      case R^.bal of
        EH : begin
          P^.bal := EH ;
          Q^.bal := EH ;
        end ;
    end ;
  end ;
end

```

```

LH : begin
    P^.bal := EH ;
    Q^.bal := RH
end ;

RH : begin
    P^.bal := LH ;
    Q^.bal := EH
end

end ; {hết case R^.bal }

R^.bal := EH ;
RotateRight (Q) ;
P^.right := Q ;
RotateLeft (P)

end ; {hết lệnh case Q^.bal}

end ;

```

Hoàn toàn tương tự, khi  $P^.bal = LH$  và đỉnh mới được xen vào cây con trái của P, đồng thời việc xen vào làm tăng độ cao của cây con trái, thì ta lập lại tính cân bằng ở đỉnh P bằng thủ tục LeftBalance (cân bằng bên trái). Bạn đọc tự viết lấy thủ tục này.

Sau đây chúng ta sẽ viết thủ tục xen vào cây cân bằng Root, một đỉnh mới với khoá x cho trước. Đây là thủ tục đệ quy. Tính cân bằng tại một đỉnh có bị phá vỡ hay không phụ thuộc vào việc khi ta xen vào cây con trái (phải) của đỉnh đó có làm tăng độ cao của cây con đó không. Do đó, ta đưa vào thủ tục tham biến taller kiểu boolean, taller = true nếu việc xen vào làm tăng độ cao của cây và taller = false nếu không.

```

procedure      Insert (var Root : pointer ; x : KeyType ;
                      var taller : boolean) ;

begin
    if Root = nil then
        begin

```

```

new (Root) ;
with Root^ do

    begin
        key := x ;
        left := nil ;
        right := nil ;
        bal := EH ;
    end ;

    taller := true ;
end else

    if x < Root^.key then
        begin
            Insert (Root^.left, x, taller) ;
            if taller then {cây con trái cao lên}
                case Root^.bal of
                    LH : begin
                        LeftBalance (Root) ;
                        taller := false ;
                    end ;
                    EH : begin
                        Root^.bal := LH ;
                        taller := true ;
                    end ;
                    RH : begin
                        Root^.bal := EH ;
                        taller := false ;
                    end ;
            end ;
        end else
        if x > Root^.key then

```

```

begin
    Insert (Root^.right, x, taller) ;
    if taller then {cây con phải cao lên}
        case Root^.bal of
            LH : begin
                Root^.bal := EH ;
                taller := false ;
            end ;
            EH : begin
                Root^.bal := RH ;
                taller := true ;
            end ;
            RH : begin
                RightBalance (Root) ;
                taller := false
            end ;
        end
    end else taller := false
end ;

```

## **2. Loại bỏ khỏi cây cân bằng**

Trong mục này ta sẽ xét phép toán loại một đỉnh có khoá x cho trước khỏi cây cân bằng, sao cho sau khi loại cây vẫn còn là cây cân bằng.

Chúng ta sẽ sử dụng thuật toán loại một đỉnh khỏi cây tìm kiếm nhị phân. Cần lưu ý rằng, thuật toán này làm cho độ cao của cây giảm đi 1 hoặc không thay đổi. Chúng ta sẽ đưa vào các thủ tục tham biến h kiểu boolean để chỉ độ cao của cây sau khi loại bỏ có ngắn đi hay không, h = true nếu độ cao của cây giảm đi 1 và h = false nếu độ cao của cây không thay đổi.

Nếu đỉnh bị loại thuộc cây con trái của cây P và việc loại bỏ làm giảm độ cao của cây con trái thì ta phải biến đổi cây P và xác định lại trạng thái cân bằng của các đỉnh chịu sự biến đổi. Phép biến đổi cây trong trường hợp này được mô tả bởi thủ tục LeftBalance.

Tương tự, nếu đỉnh bị loại thuộc cây con phải của cây P và việc loại bỏ làm giảm độ cao của cây con phải, thì ta biến đổi cây P bởi thủ tục RightBalance.

Các thủ tục LefBalance và RightBalance sẽ sử dụng các phép quay RotateLeft, RotateRight và các kỹ thuật tương tự như trong các thủ tục RightBalance và LeftBalance trong phép toán xen vào.

Sau đây sẽ viết thủ tục LeftBalance, còn thủ tục RightBalance giành lại cho bạn đọc.

```

procedure LeftBalance (var P : pointer ; var h : boolean) ;
  {áp dụng thủ tục này khi độ cao cây con trái của P giảm đi}
  var Q, R : pointer ;
  begin
    case P^.bal of
      LH : begin
        P^.bal := EH ;
        h := true ;
      end ;
      EH : begin
        P^.bal := RH ;
        h := false
      end ;
      RH : begin
        Q := P^.right ;
        case Q^.bal of
          EH : begin
            P^.bal := RH ;
            Q^.bal := LH ;
            RotateLeft (P) ;
            h := false ;
          end ;

```

```

RH : begin
    P^.bal := EH ;
    Q^.bal := EH ;
    RotateLeft (P) ;
    h := true
end ;

LH : begin
    R := Q^.left ;
    case R^.bal of
        EH : begin
            P^.bal := EH ;
            Q^.bal := EH ;
        end ;
        LF : begin
            P^.bal := EH ;
            Q^.bal := RH ;
        end ;
        RH : begin
            P^.bal := LH ;
            Q^.bal := EH ;
        end ;
    end {hết case R^.bal}
    R^.bal := EH ;
    RotateRight (Q) ;
    P^.right := Q ;
    RotateLeft (P) ;
    h := true
end ;

end {hết case Q^.bal }

end
end {hết case P^.bal}
end ;

```

Sau đây là thủ tục Del. Thủ tục này loại khỏi cây đỉnh P. Trong trường hợp cả hai con của P đều khác trống, thủ tục Del sử dụng thủ tục Erase để xoá đi đỉnh ngoài cùng bên phải của cây con trái của P. Nhưng trước khi xoá, khoá của đỉnh P được thay bằng khoá của đỉnh ngoài cùng bên phải đó.

```

procedure    Del (var P : pointer ; var h : boolean) ;
procedure    Erase (var Q : pointer ; var h : boolean) ;
begin
  if Q^.right < > nil then
    begin
      Erase (Q^.right, h) ;
      if h then RightBalance (Q,h)
    end
  else
    begin
      P^.key := Q^.key ;
      Q := Q^.left ;
      h := true;
    end
  end ; {hết thủ tục Erase}
begin {bắt đầu thủ tục Del }
  if P^.right = nil then
    begin
      P := P^.left ;
      h := true
    end else
      if P^.left = nil then
        begin
          P := P^.right ;
          h := true
        end else
          begin
            Erase(P^.left, h) ;
            if h then LeftBalance (P, h)
          end ;
    end ;

```

Đến đây chúng ta có thể viết được thủ tục loại bỏ khỏi cây gốc Root, định có khoá x cho trước. Đó là thủ tục đệ quy Delete, nó sử dụng các thủ tục LeftBalance, RightBalance và Del.

```

procedure Delete (var Root : pointer ; x : keytype ; var h : boolean);
begin
  if Root < > nil then
    if x < Root^.key then
      begin
        Delete (Root^.left, x, h) ;
        if h then LeftBalance (Root,h)
      end else
        if x > Root^.key then
          begin
            Delete (Root^.right , x, h) ;
            if h then RightBalance (Root,h)
          end else
            Del (Root, x, h);
    end ;

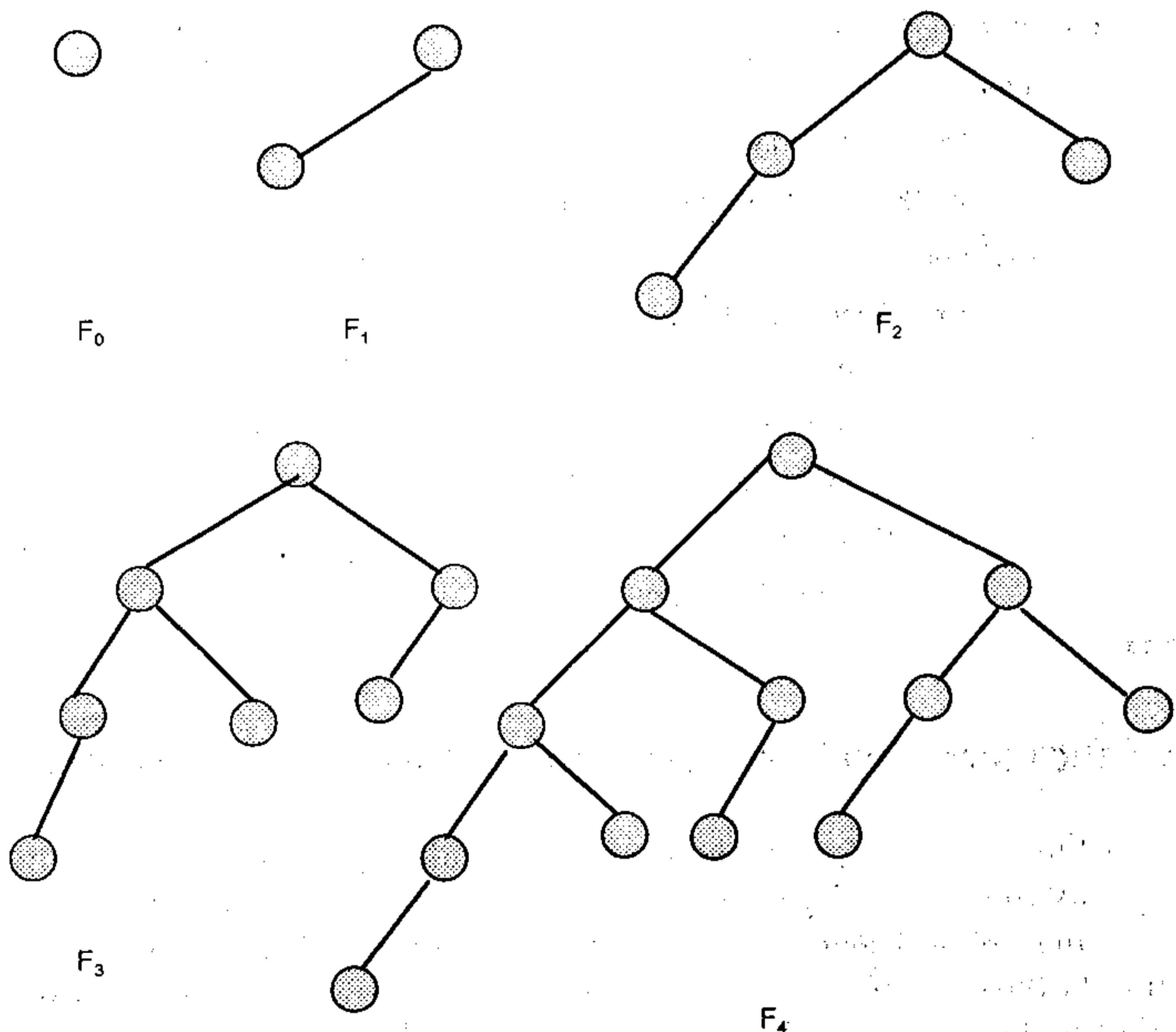
```

#### 4.8. THỜI GIAN THỰC HIỆN CÁC PHÉP TOÁN TRÊN CÂY CÂN BẰNG

Chúng ta đã biết rằng, thời gian trung bình thực hiện các phép toán trên cây tìm kiếm nhị phân là  $O(\log n)$ . Chúng ta sẽ chỉ ra rằng, trên cây cân bằng, trong trường hợp xấu nhất, thời gian thực hiện các phép toán cũng là  $O(\log n)$ . Muốn vậy ta cần phải xác định, trong trường hợp xấu nhất thì độ cao của cây cân bằng  $n$  đỉnh sẽ như thế nào.

Thay cho việc xác định độ cao lớn nhất (độ cao trong trường hợp xấu nhất) mà cây cân bằng  $n$  đỉnh có thể có, ta xác định số nhỏ nhất các đỉnh mà cây cân bằng có độ cao  $h$  phải có. Gọi  $F_h$  là cây có độ cao  $h$  với số đỉnh nhỏ nhất. Khi đó một trong hai cây con của nó phải có độ cao  $h - 1$ , còn cây con kia phải có độ cao  $h - 1$  hoặc  $h - 2$ . Nhưng vì  $F_h$  là cây có

độ cao  $h$  với số đỉnh nhỏ nhất, nên ta suy ra rằng, một cây con của nó (cây con trái) có độ cao  $h-1$ , còn cây con phải có độ cao  $h-2$ , đồng thời các cây con này phải có số đỉnh nhỏ nhất. Vậy cây con trái của  $F_h$  là  $F_{h-1}$ , cây con phải là  $F_{h-2}$ . Chúng ta sẽ gọi các cây cân bằng  $F_h$  là cây Fibonacci (vì qui luật xây dựng chúng tương tự như qui luật xây dựng các thành phần của dãy số Fibonacci).



Hình 4. 22. Minh họa một số cây Fibonacci

Nếu ký hiệu  $|T|$  là số đỉnh của cây  $T$ , chúng ta có quan hệ đệ quy sau đây

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1$$

trong đó  $|F_0|=1$ ,  $|F_1|=2$ . Thêm 1 vào hai vế ta có

$$(|F_h|+1) = (|F_{h-1}|+1) + (|F_{h-2}|+1)$$

Như vậy  $|F_h|+1$  là các thành phần của dãy số Fibonacci, cụ thể đó là thành phần  $f_{h+3}$  của dãy số Fibonacci. Theo công thức De Moivre, ta có đánh giá sau

$$|F_h|+1 \approx \frac{1}{\sqrt{5}} \left[ \frac{1+\sqrt{5}}{2} \right]^{h+3}$$

Gọi số đỉnh của cây Fibonacci có độ cao  $h$  là  $n$ ,  $|F_h|=n$ . Giải phương trình trên theo  $h$ , bằng cách lấy logarit cả hai vế, ta có

$$h \approx 1,44 \log n$$

Như vậy, độ cao của cây cân bằng  $n$  đỉnh trong trường xấu nhất là  $O(\log n)$ . Từ đó ta suy ra rằng, thời gian thực hiện các phép toán trên cây cân bằng trong trường hợp xấu nhất là  $O(\log n)$ .

## *Chương 5*

# **TẬP HỢP**

Tập hợp là một cấu trúc cơ bản của toán học. Trong thiết kế thuật toán, chúng ta thường xuyên phải sử dụng đến mô hình dữ liệu tập hợp. Trong chương này chúng ta sẽ nghiên cứu mô hình dữ liệu tập hợp, các phương pháp cài đặt tập hợp. Sau đó chúng ta sẽ nghiên cứu một số kiểu dữ liệu trùu tượng, đó là từ điển và hàng ưu tiên, được xây dựng dựa trên khái niệm tập hợp, nhưng chỉ quan tâm đến một số phép toán nào đó.

### **5.1. TẬP HỢP VÀ CÁC PHÉP TOÁN TRÊN TẬP HỢP**

Chúng ta xem rằng, độc giả đã làm quen với tập hợp. Do đó chúng ta chỉ trình bày ngắn gọn một số khái niệm được sử dụng đến sau này.

Trong toán học, có hai phương pháp để xác định một tập hợp A. Đơn giản nhất là liệt kê tất cả các phần tử của tập A (nếu tập A hữu hạn). Chẳng hạn,  $A = \{1, 2, 3\}$  có nghĩa là A là tập hợp chỉ gồm 3 phần tử 1, 2, 3. Cách khác, ta cũng có thể xác định một tập A bằng cách nêu lên các đặc trưng cho ta biết chính xác một đối tượng bất kỳ có là một phần tử của tập A hay không. Ví dụ,  $A = \{x | x \text{ là số nguyên chẵn}\}$ . Ta cần quan tâm đến một tập đặc biệt : tập trống  $\emptyset$ , đó là tập hợp không chứa phần tử nào cả.

Với hai tập bất kỳ A, B và một đối tượng x bất kỳ, ta có các quan hệ sau đây:

$x \in A$  ( $x$  thuộc A), quan hệ này đúng nếu và chỉ nếu  $x$  là phần tử của tập A.

$A \subseteq B$  ( $A$  là tập con của  $B$ ), quan hệ này đúng nếu và chỉ nếu mọi phần tử của tập  $A$  là phần tử của tập  $B$ .

$A = B$  nếu và chỉ nếu  $A \subseteq B$  và  $B \subseteq A$ .

## Các phép toán cơ bản trên tập hợp

Các phép toán cơ bản trên tập hợp là hợp, giao, hiệu. Cho hai tập  $A$  và  $B$ , khi đó hợp của  $A$  và  $B$ ,  $A \cup B$ , là tập hợp gồm tất cả các phần tử thuộc  $A$  hoặc thuộc  $B$ . Còn giao của  $A$  và  $B$  là tập  $A \cap B$  gồm tất cả các phần tử vừa thuộc  $A$ , vừa thuộc  $B$ . Hiệu  $A - B$  là tập hợp gồm tất cả các phần tử thuộc  $A$  nhưng không thuộc  $B$ . Chẳng hạn, nếu  $A = \{1, 2, 3, 4\}$  và  $B = \{3, 4, 5\}$  thì  $A \cup B = \{1, 2, 3, 4, 5\}$ , còn  $A \cap B = \{3, 4\}$  và  $A - B = \{1, 2\}$ .

Tích đê-cac của hai tập hợp  $A$  và  $B$  là tập hợp  $A \times B$  gồm tất cả các cặp phần tử  $(a, b)$ , trong đó  $a \in A$  và  $b \in B$ . Chẳng hạn, nếu  $A = \{1, 2\}$ ,  $B = \{a, b, c\}$  thì  $A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)\}$ .

## Quan hệ nhị nguyên trên tập hợp

Khi xét một tập hợp, trong nhiều trường hợp ta cần quan tâm đến quan hệ giữa các phần tử của tập hợp. Một quan hệ nhị nguyên (gọi tắt là quan hệ)  $R$  trên tập  $A$  là một tập con nào đó của tích đê-cac  $A \times A$ , tức là  $R \subseteq A \times A$ .

Nếu  $a, b$  là các phần tử của tập  $A$  và  $(a, b) \in R$  thì ta nói  $a$  có quan hệ  $R$  với  $b$  và ký hiệu là  $aRb$ . Ví dụ :  $A = \{a, b, c\}$  và  $R = \{(a, a), (a, c), (b, a), (c, b)\}$ , khi đó  $a$  có quan hệ  $R$  với  $c$  vì  $(a, c) \in R$  còn  $b$  không có quan hệ  $R$  với  $c$  vì cặp  $(b, c) \notin R$ .

Một quan hệ  $R$  có thể có các tính chất sau :

- Quan hệ  $R$  trên tập  $A$  có tính *phản xạ*, nếu  $aRa$ , với mọi  $a \in A$ .
- Quan hệ  $R$  có tính *đối xứng*, nếu mỗi khi có  $aRb$  thì cũng có  $bRa$ .
- Quan hệ  $R$  có tính *bắc cầu*, nếu mỗi khi có  $aRb$  và  $bRc$  thì cũng có  $aRc$ .
- Quan hệ  $R$  có tính *phản đối xứng*, nếu mỗi khi có  $aRb$  và  $a \neq b$  thì không có  $bRa$ .

Ví dụ nếu A là tập các số nguyên Z và R là quan hệ nhỏ hơn ( $<$ ) trên các số, tức là với các số nguyên n và m bất kỳ,  $nRm$  nếu và chỉ nếu  $n < m$ , thì dễ dàng thấy rằng,  $<$  là quan hệ có tính bắc cầu và phản đối xứng, nhưng không có tính phản xạ và đối xứng.

Hai dạng đặc biệt quan trọng là quan hệ *tương đương* và quan hệ *thứ tự bộ phận*. Một quan hệ R trên tập A được gọi là quan hệ tương đương, nếu nó thoả mãn các tính chất phản xạ, đối xứng và bắc cầu. Khi trên tập A được xác định một quan hệ tương đương R, ta có thể phân hoạch tập A thành các lớp tương đương sao cho hai phần tử bất kỳ thuộc cùng một lớp nếu và chỉ nếu chúng tương đương với nhau.

Chẳng hạn, trên tập các số nguyên Z ta xác định quan hệ R như sau:  $nRm$  nếu và chỉ nếu  $n-m$  chia hết cho 3. Dễ dàng thấy rằng, quan hệ đó thoả mãn cả ba tính chất phản xạ, đối xứng và bắc cầu. Tập Z được phân thành 3 lớp tương đương, đó là các tập số nguyên có dạng  $3k$ ,  $3k+1$  và  $3k+2$ .

Một quan hệ R trên tập A được gọi là quan hệ thứ tự bộ phận, nếu nó thoả mãn các tính chất phản xạ, phản đối xứng và bắc cầu. Khi trên tập A được xác định quan hệ thứ tự bộ phận, ta nói A là tập được sắp thứ tự bộ phận. Chẳng hạn, A là tập các số nguyên dương, quan hệ R được xác định như sau :  $nRm$  nếu và chỉ nếu  $n$  là ước của  $m$ . Khi đó R có cả ba tính chất phản xạ, phản đối xứng và bắc cầu, do đó là quan hệ thứ tự bộ phận. Quan hệ thứ tự bộ phận R sẽ được ký hiệu là  $\leq$ , do đó  $aRb$  sẽ được viết là  $a \leq b$ . Tập được sắp thứ tự bộ phận A được gọi là tập được sắp thứ tự hoàn toàn, hay tập được sắp thứ tự tuyến tính, nếu với mọi cặp phần tử a, b thuộc A ta luôn luôn có  $a \leq b$  hoặc  $b \leq a$ . Chẳng hạn, tập các số nguyên, tập các số thực đều là các tập được sắp thứ tự tuyến tính với quan hệ  $\leq$  thông thường.

## Mô hình dữ liệu tập hợp

Trong thiết kế thuật toán, khi sử dụng tập hợp như một mô hình dữ liệu, ngoài các phép toán hợp, giao, hiệu, chúng ta phải cần đến nhiều phép toán khác. Sau đây chúng ta sẽ đưa ra một số phép toán quan trọng nhất, các phép toán này sẽ được mô tả bởi các thủ tục hoặc hàm.

**1. Phép hợp :**

**Procedure** Union (A, B : **set**; **var** C : **set**);

Thủ tục tìm hợp của tập A và tập B, kết quả là tập C.

**2. Phép giao :**

**Procedure** Intersection (A, B : **set**; **var** C : **set**);

Thủ tục tìm giao của tập A và tập B, kết quả là tập C.

**3. Phép trừ :**

**Procedure** Difference ( A,B: **set** ; **var** C: **set**);

Thủ tục tìm hiệu của tập A và tập B, kết quả là C.

**4. Xác định một phần tử có thuộc tập hợp hay không :**

**Function** Member ( x: element ; A: **set** ) : **boolean** ;

Hàm Member nhận giá trị true nếu  $x \in A$  và false nếu không.

**5. Phép xen vào :**

**Procedure** Insert ( x: element ; **var** A: **set** );

Thủ tục này thêm phần tử x vào tập A, do đó sau khi thực hiện thủ tục, giá trị mới của A là  $A \cup \{x\}$ .

**6. Phép loại bỏ :**

**Procedure** Delete ( x : element ; **var** A: **set** );

Thủ tục này loại bỏ x khỏi tập A. Sau thủ tục này, tham biến A nhận giá trị mới là  $A - \{x\}$ .

**7. Tìm phần tử nhỏ nhất ( phần tử lớn nhất ).**

**Procedure** Min ( A: **set** ; **var** x: element );

Phép toán này chỉ áp dụng trên các tập hợp sắp thứ tự tuyến tính. Sau khi thực hiện thủ tục, x là phần tử nhỏ nhất của tập A.

Vấn đề được đặt ra bây giờ là, ta cần biểu diễn tập hợp như thế nào để các phép toán được thực hiện với hiệu quả cao.

## 5.2. CÀI ĐẶT TẬP HỢP

Có nhiều phương pháp biểu diễn tập hợp. Trong từng áp dụng, tùy thuộc vào các phép toán cần thực hiện và cỡ (số phần tử) của tập hợp mà ta lựa chọn cách cài đặt sao cho các phép toán thực hiện có hiệu quả.

Trước hết, chúng ta cần biết rằng, các phần tử của tập hợp có thể là đối tượng phức tạp (không phải là các số nguyên, số thực hoặc các kí tự). Các đối tượng này có thể được biểu diễn bởi bản ghi mà các trường là các thuộc tính của đối tượng. Mỗi phần tử được hoàn toàn xác định bởi các giá trị của một số trường nào đó (khoá). Trong trường hợp này, ta có thể mô tả kiểu dữ liệu của các phần tử của tập hợp như sau.

```
type elementtype = record
    key : keytype;
    [Các trường khác]
end;
```

### 5.2.1. Cài đặt tập hợp bởi vectơ bit

Giả sử các tập hợp mà ta quan tâm đều là tập con của một tập "vũ trụ" nào đó. Giả sử cỡ của tập vũ trụ tương đối nhỏ và các phần tử của nó là các số nguyên từ 1 đến n (hoặc được mã hoá bởi các số nguyên 1..n). Khi đó ta có thể dùng vectơ bit (mảng boolean) để biểu diễn tập hợp. Một tập A được biểu diễn bởi vectơ bit ( $A[1], A[2], \dots, A[i], \dots, A[n]$ ), trong đó thành phần thứ i,  $A[i]$  là true nếu và chỉ nếu i là phần tử của tập A.

```
const n = ... ;
type Set = array[1..n] of boolean;
var A,B,C : set;
    x : 1..n;
```

Dễ dàng thấy rằng, với cách cài đặt này, tất cả các phép toán cơ bản trên tập hợp đều được thực hiện rất dễ dàng, và với thời gian thực hiện cũng lăm là tỷ lệ với cỡ của tập vũ trụ, tức là  $O(n)$ . Chẳng hạn, để thêm x vào tập A, ta chỉ cần thực hiện lệnh

```
A[x] := true
```

Còn để xác định x có là phần tử của tập A hay không ta chỉ cần biết A[x] là true hay false.

Các phép hợp, giao, hiệu của hai tập hợp cũng được thực hiện rất đơn giản. Sau đây là thủ tục Union thực hiện phép lấy hợp của hai tập A và B.

```
procedure Union (A, B : Set; var C: Set) ;
var i: integer;
begin
    for i := 1 to n do C[i] := A[i] or B[i];
end;
```

### 5.2.2. Cài đặt tập hợp bởi danh sách

Chúng ta cũng có thể biểu diễn tập hợp bởi danh sách  $L = (a_1, a_2, \dots, a_n)$ , trong đó các thành phần  $a_i$  của danh sách là các phần tử của tập hợp. Nhớ lại rằng, một danh sách có thể được cài đặt bởi mảng, hoặc bởi danh sách liên kết. Do đó chúng ta có thể cài đặt tập hợp bởi mảng hoặc bởi danh sách liên kết.

#### 1. Cài đặt tập hợp bởi mảng:

Giả sử số phần tử của tập hợp không vượt quá một hằng nào đó maxsize. Khi đó ta có thể biểu diễn tập hợp bởi một mảng. Các thành phần của mảng bắt đầu từ thành phần đầu tiên sẽ lưu giữ các phần tử của tập hợp. ta sẽ đưa vào một biến last ghi lại chỉ số của thành phần cuối cùng của mảng có chứa phần tử của tập hợp.

```
const maxsize =...;
type Set = record
    last : integer;
    element : array [1..maxsize] of elementtype;
end;
```

Trong cách cài đặt này, một không gian nhớ cố định (do cỡ của mảng qui định) được dùng để lưu giữ các phần tử của tập hợp. Việc thực hiện các phép hợp, xen vào có thể dẫn đến các tập hợp có số phần tử

vượt quá cỡ của mảng. Do đó khi sử dụng cách cài đặt này chúng ta phải chọn maxsize thích hợp để tiết kiệm bộ nhớ và tránh trường hợp bị tràn.

Chúng tôi để lại cho độc giả tự viết các thủ tục và hàm thực hiện các phép toán tập hợp trong cách cài đặt này.

## **2. Cài đặt tập hợp bởi danh sách liên kết**

Việc biểu diễn tập hợp bởi danh sách liên kết sẽ khắc phục được hạn chế về không gian khi dùng mảng. ta có thể sử dụng phương pháp này để biểu diễn tập hợp có số phần tử nhiều ít tùy ý, miễn là bộ nhớ của máy cho phép. Tuy nhiên trong cách cài đặt này, việc thực hiện các phép toán tập hợp sẽ phức tạp hơn. Mỗi thành phần trong danh sách liên kết biểu diễn tập hợp là một tế bào có khai báo như sau :

```
type pointer = ^ Cell;
Cell = record
    element : elementtype;
    next : pointer;
end;
```

Các tập hợp A, B, C sẽ được biểu diễn bởi các danh sách liên kết, trong đó các con trỏ A, B, C sẽ trỏ tới đầu của các danh sách đó.

```
var A, B, C : pointer;
```

Sau đây chúng ta sẽ trình bày sự thực hiện các phép toán khi tập hợp được cài đặt bởi danh sách liên kết. Phép toán Member (x,A) chính là phép tìm kiếm phần tử x trong danh sách liên kết A.

Cho hai tập hợp A và B được biểu diễn bởi các danh sách liên kết. Việc tìm danh sách C biểu diễn hợp, giao hoặc hiệu của A và B được tiến hành bởi cùng một phương pháp. Chẳng hạn, muốn tìm giao của A và B, ta phải so sánh mỗi phần tử e của danh sách A với lần lượt từng phần tử của danh sách B. Nếu trong danh sách B có một phần tử cùng là e thì phần tử e được đưa vào danh sách C.

Sau đây là thủ tục thực hiện phép giao

```

procedure      Intersection (A, B : pointer; var C : pointer);
var
    Ap, Bp, Cp : pointer;
    found : boolean;
begin
    C := nil;
    Ap := A;
    while Ap <> nil do
        begin
            Bp := B;
            found := false;
            while (Bp <> nil) and (not found) do
                if Bp ^ . element = Ap ^ . element then
                    found := true else Bp := Bp ^ . next;
            if found then
                begin
                    new (Cp);
                    Cp ^ . element := Ap ^ . element;
                    Cp ^ . next := C;
                    C := Cp;
                end;
            Ap := Ap ^ . next;
        end;
    end;

```

Để tìm hợp của A và B, đầu tiên ta sao chép danh sách B để có danh sách C là bản sao của B. Sau đó ta so sánh mỗi phần tử e của danh sách A với từng phần tử của danh sách B. Nếu không có phần tử nào của B là e thì ta thêm e vào danh sách C. Một cách tương tự đối với phép toán A - B.

Trong cách cài đặt tập hợp bởi danh sách (không được sắp) như trên, khi thực hiện các phép toán hợp, giao, trừ, ta phải so sánh mỗi phần tử của danh sách A với từng phần tử của danh sách B. Do đó thời gian thực hiện các phép toán đó là  $O(n^2)$ , trong đó  $n = \max(|A|, |B|)$ , ở đây  $|A|$  ký hiệu số phần tử của tập A.

### 3. Cài đặt tập hợp bởi danh sách được sắp :

Trong trường hợp các tập hợp là các tập con của tập vū trụ được sắp tuyến tính bởi quan hệ thứ tự nào đó, thì các phép toán tập hợp sẽ được thực hiện nhanh hơn nếu ta cài đặt các tập bởi các danh sách được sắp. Một tập được biểu diễn bởi danh sách được sắp, nếu các thành phần của danh sách được sắp xếp theo thứ tự tăng dần (hoặc giảm dần) :  $a_1 < a_2 < \dots < a_n$ .

*Chú ý:* Thay cho việc xét chính các phần tử của tập hợp, ta có thể xét các khoá của chúng. Nếu tập các khoá là tập được sắp tuyến tính thì ta cũng có thể cài đặt tập hợp bởi danh sách được sắp theo khoá.

Với các danh sách được sắp A và B, để tìm danh sách được sắp C biểu diễn hợp, giao, hiệu của chúng, ta chỉ cần so sánh mỗi phần tử a của danh sách A với các phần tử của danh sách B cho tới khi hoặc tìm được trong danh sách B một phần tử bằng a, hoặc tìm được một phần tử  $b > a$ . Hơn nữa, nếu đối với một phần tử  $a_i$  trong danh sách A, ta đã tìm được một phần tử  $b_k$  trong danh sách B sao cho  $a_i \leq b_k$ , thì đối với phần tử tiếp theo  $a_{i+1}$  trong danh sách A ta chỉ cần bắt đầu sự tìm kiếm trong danh sách B kể từ thành phần  $b_k$ . Do đó thời gian thực hiện các phép toán hợp, giao, trừ sẽ tỷ lệ với số phần tử của tập hợp,  $O(n)$ , trong đó  $n = \max(|A|, |B|)$ .

Sau đây chúng ta sẽ viết các thủ tục thực hiện các phép hợp và giao của các tập hợp được biểu diễn bởi các danh sách được sắp A và B. Danh sách được sắp C biểu diễn hợp (hoặc giao) là danh sách vòng tròn, con trỏ C trỏ tới cuối danh sách, còn  $C^.next$  trỏ tới đầu danh sách.

```

procedure Union (A,B : pointer ; var C: pointer );
var Ap, Bp, Cp : pointer ;
procedure ADD ( Cp : pointer ; var C: pointer);
    {Thêm Cp vào cuối danh sách C }
begin
    if C=nil then
        begin
            C:=Cp;
            C^.next :=C;
        end

```

```
else
begin
    Cp^.next := C^.next;
    C^.next := Cp;
    C:=Cp;
end;
end;

begin
C:= nil;
Ap:=A;
Bp:=B;
while ( Ap<>nil) and (Bp<> nil) do
if Ap^.element < = Bp^.element then
begin
    new(C);
    Cp^.element:=Ap^.element;
    Add(Cp,C);
    if Ap^.element=Bp^.element then
begin
        Ap := Ap^.next ;
        Bp := Bp^.next ;
    end else Ap:=Ap^.next;
end else
begin
    new(Cp);
    Cp^.element:=Bp^.element;
    Add(Cp,C);
    Bp:=Bp^.next ;
end;
while Ap < > nil do
begin
    new(Cp);
    Cp^.element:=Ap^.element;
    Add (Cp,C);
end;
```

```

while Bp < > nil do
    begin
        new (Cp);
        C^.element := Bp ^.element ;
        Add (Cp, C);
    end;
end;

procedure Intersection (A,B : pointer; var C: pointer);
var
    Ap, Bp, Cp : pointer;
begin
    C:=nil;
    Ap:=A;
    Bp:=B;
    while ( Ap <> nil ) and (Bp<> nil) do
        if Ap^.element= Bp^.element then
            begin
                new(Cp);
                Cp^.element := Ap^.element;
                Add(Cp,C);
                Ap:= Ap^.next;
                Bp := Bp^.next;
            end else
                if Ap^.element < Bp^.element then Ap := Ap^.next
                else Bp := Bp^.next;
    end;

```

### 5.3. TỪ ĐIỂN

#### 5.3.1. Từ điển

Trong nhiều áp dụng, khi sử dụng mô hình dữ liệu tập hợp để thiết kế thuật toán, ta không cần đến các phép toán lấy hợp, giao, hiệu của

các tập. Thông thường khi đã lưu giữ một tập hợp thông tin nào đó, ta chỉ cần đến phép toán thêm một phần tử mới vào tập hợp, loại khỏi tập hợp một phần tử nào đó, và tìm xem trong tập hợp có chứa một phần tử nào đó hay không.

Mô hình giữ liệu tập hợp, nhưng chỉ xét đến những phép toán Insert, Delete và Member được gọi là kiểu giữ liệu trừu tượng từ điển (Dictionary).

Sau đây chúng ta sẽ nêu ra các phương pháp đơn giản mà chúng ta đã biết trong các chương trước để cài đặt từ điển. Trong mục 5.4 chúng ta sẽ trình bày một kỹ thuật mới để cài đặt từ điển.

### 5.3.2. Các phương pháp đơn giản cài đặt từ điển

Từ điển là một tập hợp, do đó đương nhiên ta có thể sử dụng các phương pháp cài đặt tập hợp để cài đặt từ điển.

Chúng ta có thể biểu diễn từ điển bởi vectơ bit. Khi đó các phép toán trong từ điển được thực hiện rất đơn giản với thời gian hằng. Tuy nhiên, ta chỉ có thể áp dụng được phương pháp này nếu từ điển là tập hợp có thể dùng làm tập chỉ số cho mảng.

Chúng ta có thể biểu diễn từ điển bởi danh sách. Đến lượt mình, danh sách có thể được cài đặt bởi mảng hoặc bởi danh sách liên kết. Khi cài đặt từ điển bởi mảng hoặc bởi danh sách liên kết, mỗi phương pháp đều có ưu điểm và nhược điểm mà chúng ta đã phân tích ở chương 3. Thời gian để thực hiện các phép toán Insert, Delete, Member nói chung là  $O(n)$  với từ điển có  $n$  phần tử.

Giả sử từ điển là một tập được sắp thứ tự tuyến tính. Trong trường hợp này, ta có thể biểu diễn từ điển bởi cây tìm kiếm nhị phân. Với cách cài đặt này các phép toán Member, Insert và Delete là các phép toán tìm kiếm, xen vào và loại bỏ trên cây tìm kiếm nhị phân được xét trong chương 4. Thời gian trung bình để thực hiện các phép toán trên cây tìm kiếm nhị phân là  $O(\log n)$ , trong trường hợp xấu nhất khi cây suy biến thành danh sách là  $O(n)$ . Nếu ta biểu diễn từ điển bởi cây cân bằng, thì thời gian thực hiện các phép toán, ngay cả trong trường hợp xấu nhất cũng là  $O(\log n)$ . Tuy nhiên như chúng ta đã biết, việc thực hiện các phép toán xen vào và loại bỏ trên cây cân bằng khá phức tạp.

## 5.4. CẤU TRÚC DỮ LIỆU BẢNG BĂM. CÀI ĐẶT TỪ ĐIỂN BỞI BẢNG BĂM

Trong mục này chúng ta sẽ trình bày một kỹ thuật quan trọng, được gọi là phương pháp băm (hashing). Chúng ta sẽ áp dụng phương pháp băm để cài đặt từ điển. Băm là phương pháp rất thích hợp để cài đặt tập hợp có số phần tử lớn và thời gian cần thiết để thực hiện các phép toán từ điển, ngay cả trong trường hợp xấu nhất, là tỷ lệ với cỡ của tập hợp.

Chúng ta sẽ đề cập đến hai phương pháp băm khác nhau. Một gọi là băm mở (open hashing) cho phép sử dụng một không gian không hạn chế để lưu giữ các phần tử của tập hợp. Phương pháp băm khác được gọi là băm đóng (closed hashing) sử dụng một không gian cố định và do đó tập hợp được cài đặt phải có cỡ không vượt quá không gian cho phép.

### 5.4.1. Bảng băm mở

Tư tưởng cơ bản của băm mở là phân chia tập hợp đã cho thành một số cố định các lớp. Chẳng hạn, ta muốn phân thành N lớp được đánh số 0, 1, ..., N-1. Ta sử dụng mảng T với chỉ số chạy từ 0 đến N-1. Mỗi thành phần T [i] của mảng được nói đến như một "rổ" đựng các phần tử của tập hợp thuộc lớp thứ i. Các phần tử của tập hợp thuộc mỗi lớp được tổ chức dưới dạng một danh sách liên kết. Do đó T [i] sẽ chứa con trỏ trỏ đến danh sách của lớp i. Ta sẽ gọi mảng T là bảng băm (hash table).

Việc phân chia các phần tử của tập hợp vào các lớp được thực hiện bởi *hàm băm* (hash function) h. Nếu x là một giá trị khoá của phần tử nào đó của tập hợp thì h(x) là chỉ số nào đó của mảng T và ta gọi h(x) là giá trị băm (hash value) của x. Như vậy h là ánh xạ từ tập hợp các khoá K vào tập hợp {0, 1, ..., N-1}.

#### *Hàm băm*

Có hai tiêu chuẩn chính để lựa chọn một hàm băm. Trước hết nó phải cho phép tính được dễ dàng và nhanh chóng giá trị băm của mỗi khoá. Thứ hai nó phải phân bổ đều các khoá vào các rổ. Trên thực tế tiêu chuẩn thứ hai rất khó được thực hiện. Sau đây chúng ta đưa ra một số phương pháp thiết kế hàm băm.

**1. Phương pháp cắt bỏ.** Giả sử khoá là số nguyên (nếu khoá không phải là số nguyên, ta xét đến các mã số của chúng). Ta sẽ bỏ đi một phần nào đó của khoá, và lấy phần còn lại làm giá trị băm của khoá. Chẳng hạn, nếu khoá là các số nguyên 10 chữ số và bảng băm gồm 1000 thành phần, khi đó ta có thể lấy chữ số thứ nhất, thứ ba và thứ bảy từ bên trái làm giá trị băm. Ví dụ :  $h(7103592810) = 702$ . Phương pháp cắt bỏ rất đơn giản, nhưng nó thường không phân bố đều các khoá.

**2. Phương pháp gấp.** Giả sử khoá là số nguyên. Ta phân chia khoá thành một số phần, sau đó kết hợp các phần lại bằng một cách nào đó (chẳng hạn, dùng phép cộng hoặc phép nhân) để nhận giá trị băm. Chẳng hạn, nếu khoá là số nguyên 10 chữ số, ta phân thành các nhóm ba, ba, hai và hai chữ số từ bên trái, cộng các nhóm với nhau, sau đó cắt cụt nếu cần thiết, ta sẽ nhận được giá trị của hàm băm. Ví dụ 7103592810 được biến đổi thành  $710+359+28+10 = 1107$ , do đó ta có giá trị băm là 107. Vì mọi thông tin trong khoá đều được phản ánh vào giá trị băm, nên phương pháp gấp cho phân bố đều các khoá tốt hơn phương pháp cắt bỏ.

**3. Phương pháp sử dụng phép toán lấy phần dư.** Giả sử khoá là số nguyên, và giả sử ta muốn chia tập hợp các khoá thành N lớp. Chia số nguyên cho N rồi lấy phần dư làm giá trị băm. Điều này trong Pascal được thực hiện bằng phép toán MOD. Tính phân bố đều các khoá của hàm băm được xác định bằng phương pháp này phụ thuộc nhiều vào việc chọn N. Tốt nhất chọn N là số nguyên tố. Chẳng hạn thay cho chọn  $N = 1000$ , ta lấy  $N = 997$  hoặc  $N = 1009$ .

Sau đây ta sẽ viết một hàm băm trong Pascal để băm các khoá là các xâu kí tự có độ dài 10 thành các giá trị từ 0 đến N-1

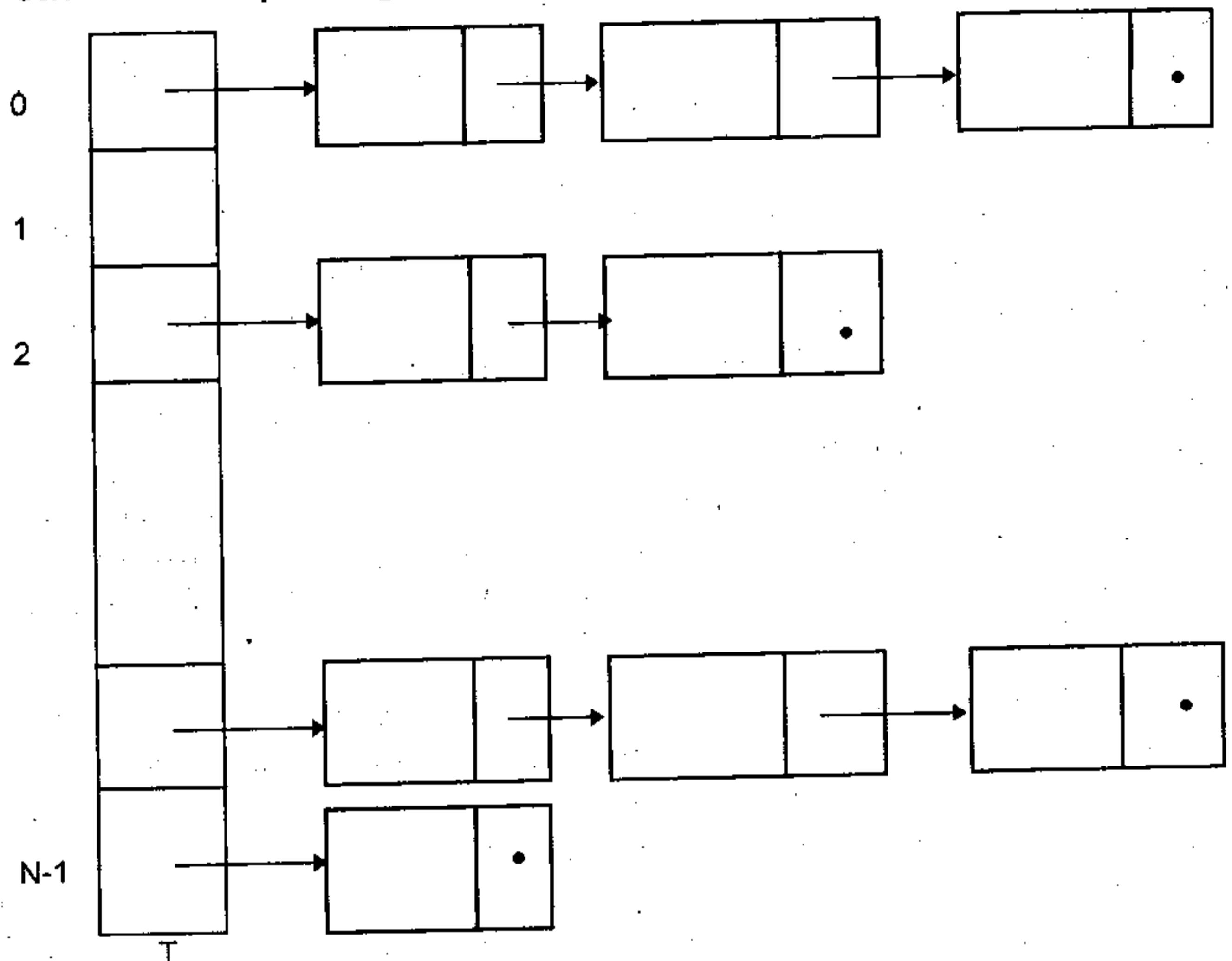
```

type keytype = string [10];
function h (x : keytype) : 0..N-1;
var i, Sum : integer;
begin
    Sum := 0;
    for i = 1 to 10 do
        Sum := Sum + ord(x[i]);
    h := Sum mod N;
end;

```

Trong hàm băm trên, ta đã chuyển đổi các xâu kí tự thành các số nguyên bằng cách lấy tổng số của các mã số của từng kí tự trong xâu (ord (c) là mã số của kí tự c).

Cấu trúc dữ liệu bảng băm mở được minh họa trong hình 5.1



Hình 5.1. Bảng băm mở

Chúng ta có thể khai báo cấu trúc dữ liệu bảng băm mở biểu diễn từ điển như sau :

```

const      N = ...;
type  pointer = ^ element;
        element = record
                    key : keytype;
                    next : pointer;
                end;
        Dictionary = array [0..N-1] of pointer;
var    T : Dictionary;
    
```

Việc khởi tạo một từ điển rỗng được thực hiện bằng lệnh sau

```
for i := 0 to N-1 do T [i] := nil;
```

### *Các phép toán từ điển trên bảng băm mở*

Sau đây chúng ta sẽ đưa ra các thủ tục thực hiện các phép toán từ điển.

```
function Member' (x : keytype; var T : Dictionary) : boolean;
var P : pointer; found : boolean;
begin
    P := T [h(x)];
    found := false;
    while (P < > nil) and (not found) do
        if P ^ . key = x then found := true
        else P := P ^ . next;
    Member := found;
end;

procedure Insert (x : keytype; var T : Dictionary);
var i : 1.. N-1;
    P : pointer;
begin
    if not Member (x, T) then
        begin
            i := h (x);
            new (P);
            P ^ . key := x;
            P ^ . next := T [i];
            T [i] := P;
        end;
    end;
```

```

procedure Delete (x : keytype; var T : dictionary);
var i : 0.. N-1;
    P, Q : pointer;
    found : boolean;
begin
    i := h (x);
    if T[i] < > nil then
        if T [i] ^ . key = x then{ loại x khỏi danh sách}
            T [i] := T [i] ^ . next;
        else begin
            {xem xét các thành phần tiếp theo trong danh sách}
            Q := T [i];
            P := Q ^ . next;
            found := false;
            while (P < > nil) and (not found) do
                if P ^ . key = x then
                    begin { loại x khỏi danh sách }
                        Q ^ . next := P ^ . next;
                        found := true;
                    end else
                        begin
                            Q := P;
                            P := Q ^ . next;
                        end;
                end;
            end;
        end;
end;

```

#### 5.4.2. Bảng băm đóng :

Trong bảng băm mở, mỗi thành phần  $T[i]$  của bảng lưu giữ con trỏ trỏ tới danh sách các phần tử của tập hợp được đưa vào lớp thứ i

( $i = 0, \dots, N-1$ ). Khác với bảng băm mở, trong bảng băm đóng, mỗi phần tử của tập được lưu giữ trong chính các thành phần  $T[i]$  của mảng. Do đó ta có thể khai báo kiểu dữ liệu từ điển được cài đặt bởi bảng băm đóng như sau :

**type Dictionary = array [0..N-1] of keytype;**

Ở đây keytype là kiểu dữ liệu của khoá của các phần tử trong từ điển. Nhớ lại rằng, hàm băm

$$h : K \rightarrow \{0, 1, \dots, N-1\}$$

là ánh xạ từ tập hợp các khoá  $K$  vào tập hợp các chỉ số  $0, 1, \dots, N-1$  của mảng. Đây là ánh xạ nhiều-vào-một, nên có thể xảy ra một số khoá khác nhau được ánh xạ vào cùng một chỉ số. Do đó có thể có trường hợp, ta muốn đặt khoá  $x$  vào thành phần  $i = h(x)$  của mảng, nhưng ở đó đã lưu giữ một khoá khác. Hoàn cảnh này được gọi là *sự va chạm* (collision). Vấn đề đặt ra là *giải quyết sự va chạm* như thế nào.

Sự va chạm được giải quyết bằng cách băm lại (rehashing). Chiến lược băm lại là như sau. ta sẽ lần lượt xét các vị trí  $h_1(x), h_2(x), \dots$  cho tới khi tìm được một vị trí nào trống để đặt  $x$  vào đó. Nếu không tìm được vị trí nào trống thì bảng đã đầy và ta không thể đưa  $x$  vào bảng được nữa. Ở đây  $h_i(x)$  ( $i = 1, 2, \dots$ ) là các giá trị băm lại lần thứ  $i$ , nó chỉ phụ thuộc vào khoá  $x$ . Sau đây chúng ta sẽ xét một số phương pháp băm lại.

## Các phương pháp băm lại

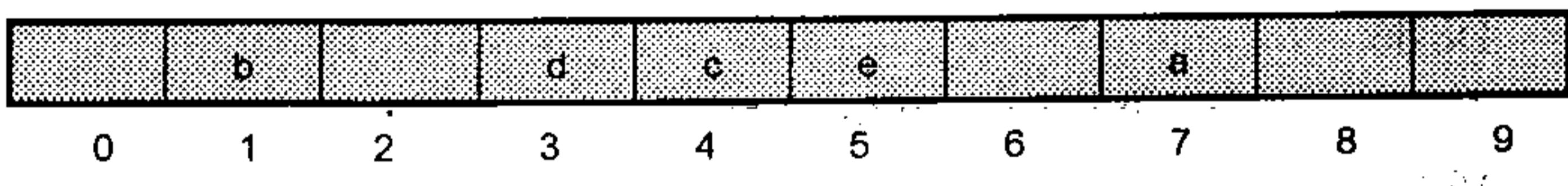
### 1. Băm lại tuyến tính

Đây là phương pháp băm lại đơn giản nhất. Các hàm  $h_i(x)$  được xác định như sau :

$$h_i(x) = (h(x) + i) \bmod N.$$

Tức là, ta xem mảng là mảng vòng tròn và lần lượt xem xét các vị trí  $h(x) + 1, h(x) + 2, \dots$

Chẳng hạn,  $N = 10$  và các khoá  $a, b, c, d, e$  có các giá trị băm như sau  $h(a) = 7, h(b) = 1, h(c) = 4, h(d) = 3, h(e) = 3$ .



Hình 5.2

Giả sử ban đầu bảng rỗng, tức là tất cả các thành phần của bảng đều chứa một giá trị empty nào đó khác với tất cả các giá trị khoá. Giả sử ta muốn đưa vào bảng rỗng lần lượt các giá trị khoá a, b, c, d, e. Khi đó a, b, c, d lần lượt được đặt vào các vị trí 7, 1, 4, 3 vào bảng. Vì  $h(e) = 3$ , ta tìm đến thành phần thứ 3 của mảng và thấy nó đã chứa d. Tìm đến thành phần  $h_1(e) = h(e) + 1 = 4$ , lại thấy nó đã chứa c. Tìm đến thành phần  $h_2(e) = 5$ , vị trí này rỗng, ta đưa e vào đó. Kết quả là ta có bảng băm đóng được minh họa trong hình 5.2.

Hạn chế cơ bản của phương pháp băm lại tuyến tính là các giá trị khoá sẽ được xếp liền vào sau các giá trị khoá ban đầu đã đưa vào bảng mà không gặp va chạm. Do đó càng ngày các giá trị khoá trong bảng càng tụ lại thành các đoạn dài bị lấp đầy và giữa các đoạn bị lấp đầy là các khoảng trống. Và vì vậy, việc tìm ra một vị trí trống trong bảng để đưa giá trị mới vào, càng về sau càng chậm.

## 2. Băm lại bình thường

Phương pháp băm lại tốt hơn, cho phép ta tránh được sự tích tụ trong bảng các giá trị xung quanh các giá trị đưa vào bảng ban đầu, là sử dụng các hàm băm lại được xác định như sau :

$$h_i(x) = (h(x) + i^2) \bmod N;$$

Hạn chế của phương pháp này là ở chỗ, các giá trị băm lại không lấp đầy tất cả các chỉ số của mảng. Do đó khi cần đưa vào bảng một giá trị mới, có thể ta không tìm được vị trí rỗng, mặc dù trong bảng hãy còn các vị trí rỗng.

Xét trường hợp chiều của mảng N là số nguyên tố. Giả sử với  $i \neq j$  ta có

$$h_i(x) = h_j(x)$$

hay

$$h(x) + i^2 \equiv h(x) + j^2 \pmod{N}$$

Do đó

$$(i - j)(i + j) \equiv 0 \pmod{N}$$

Vì N là số nguyên tố, ta suy ra, một trong hai nhân thức  $i - j$  và  $i + j$  phải chia hết cho N. Do đó hoặc  $i \geq N/2$  hoặc  $j \geq N/2$ . Từ đó ta suy ra, với i đi từ 1 đến N div 2 tất cả các giá trị băm lại đều khác nhau. Như vậy

có tất cả  $N$  div 2 giá trị băm lại khác nhau. Tức là, khi gặp va chạm, phương pháp băm lại bình thường sẽ cho phép tìm đến một nửa số vị trí trong bảng. Việc tìm đến một nửa số vị trí của bảng để tìm ra một vị trí trống, trên thực tế, là ít khi cần đến, trừ trường hợp bảng đã gần đầy.

Trong các phương pháp băm lại trên, thực chất ta đã thêm vào giá trị băm ban đầu  $h(x)$  một giá số  $\Delta(i)$  để nhận được giá trị băm lại ở lần thứ  $i$ .

$$h_i(x) = (h(x) + \Delta(i)) \bmod N$$

Trong trường hợp băm lại tuyến tính  $\Delta(i) = i$ , còn trong trường hợp băm lại bình phương  $\Delta(i) = i^2$ .

Còn có thể sử dụng các hàm gia số khác để nhận được các giá trị băm lại. Chẳng hạn,  $\Delta(i) = c i$ , trong đó  $c$  hằng số  $> 1$ .

$$h_i(x) = (h(x) + c i) \bmod N.$$

Ví dụ, với  $N = 8$ ,  $c = 3$  và  $h(x) = 4$ , các vị trí trong bảng được tìm đến là 4, 7, 2, 5, 0, 3, 6 và 1. Tất nhiên, nếu  $N$  và  $c$  có ước chung lớn hơn 1, thì phương pháp băm lại này không cho ta tìm đến tất cả các vị trí trong bảng; chẳng hạn với  $N = 8$  và  $c = 2$ .

Một cách tiếp cận khác là sử dụng các gia số là các số ngẫu nhiên:

$$h_i(x) = (h(x) + d_i) \bmod N$$

trong đó,  $d_1, d_2, \dots, d_{N-1}$  là một hoán vị ngẫu nhiên của các số 1, 2, ...,  $N-1$ . Cần lưu ý rằng, khi đã chọn một dãy ngẫu nhiên  $d_1, d_2, \dots, d_{N-1}$ , thì trong mọi phép toán tìm kiếm, xen vào và loại bỏ, nếu gặp va chạm, ta phải sử dụng cùng một dãy ngẫu nhiên đã chọn để tính các giá trị băm lại.

### Các phép toán từ điển trên bảng băm đóng

Sau đây chúng ta sẽ xét các phép toán từ điển (Insert, Delete, Member) khi từ điển được cài đặt bởi bảng băm đóng.

Để biết trong bảng có chứa khoá  $x$  hay không, ta phải "thăm dò" lần lượt các vị trí  $h(x), h_1(x), h_2(x), \dots$ . Giả sử ta chưa thực hiện phép loại bỏ nào đối với bảng. Khi đó có hai khả năng. Hoặc là tìm được một vị trí của bảng chứa  $x$ , hoặc là tìm được một vị trí trống đầu tiên  $h_k(x)$ . Trong trường hợp thứ hai, ta có thể kết luận rằng, bảng không chứa  $x$ .

vì  $x$  không thể được đặt vào một trong các vị trí  $h_{k+1}(x)$ ,  $h_{k+2}(x)$ . Tuy nhiên tình hình sẽ khác, nếu trong bảng đã thực hiện một số lần loại bỏ. Trong trường hợp đã có sự loại bỏ trong bảng, nếu tìm ra vị trí trống đầu tiên  $h_k(x)$  ta không thể đảm bảo rằng  $x$  không ở đâu đó trong các vị trí  $h_{k+1}(x)$ ,  $h_{k+2}(x)$ , ... Vì rằng có thể lúc đưa  $x$  và bảng, vị trí  $h_k(x)$  đã đầy, nhưng sau đó nó trở thành trống bởi một phép loại bỏ nào đó.

Để đảm bảo rằng, khi tìm ra vị trí trống đầu tiên  $h_k(x)$ , ta có thể tin chắc rằng bảng không chứa  $x$ , ta đưa vào một hằng mới "bị loại bỏ" (deleted) khác với hằng "trống" (empty). Với việc đưa vào hằng deleted, mỗi khi cần loại bỏ một giá trị khỏi một vị trí nào đó trong bảng, ta chỉ cần thay giá trị của bảng tại vị trí đó bởi hằng deleted. Khi cần đưa một giá trị mới vào bảng, ta có thể đặt nó vào vị trí đã loại bỏ.

Ta có thể khai báo cấu trúc dữ liệu bảng băm đóng biểu diễn từ điển như sau :

```
const      N = ... ;
empty = ... ;
deleted = ... ;
```

{empty và deleted là hai hằng khác với tất cả các giá trị khoá của các phần tử của từ điển}.

```
type Dictionary = array [ 0..N-1] of keytype;
var       T : Dictionary;
```

Với mỗi giá trị khoá  $x$ , để thực hiện các phép toán Insert, Delete, Member, ta đều phải xác định vị trí trong bảng có chứa  $x$ , hoặc vị trí trong bảng cần đặt  $x$  vào. Tư tưởng để tìm ra các vị trí đó là thăm dò lần lượt các vị trí  $h(x)$ ,  $h_1(x)$ ,  $h_2(x)$ , ... Điều đó được thực hiện bởi thủ tục Location.

Sau đây ta sẽ mô tả thủ tục Location trong trường hợp sử dụng phương pháp băm lại tuyến tính.

Với mỗi giá trị khoá  $x$ , thủ tục này cho phép thăm dò các vị trí trong bảng, xuất phát từ vị trí được xác định bởi giá trị băm  $h(x)$ , rồi lần lượt qua các vị trí  $h_1(x)$ ,  $h_2(x)$ , ... cho tới khi hoặc tìm được vị trí có chứa  $x$ , hoặc tìm ra vị trí trống đầu tiên.

Quá trình thăm dò cũng sẽ dừng lại nếu đi qua toàn bộ bảng mà không thành công (không tìm thấy vị trí chứa x cũng không tìm thấy vị trí trống). Vị trí mà tại đó quá trình thăm dò dừng lại được ghi vào tham biến k. Ta đưa vào thủ tục tham biến j để ghi lại vị trí loại bỏ (deleted) đầu tiên hoặc vị trí trống đầu tiên mà quá trình thăm dò phát hiện ra, nếu trong bảng còn có các vị trí như thế.

```

procedure Location (x : keytype; var k, j : integer);
var i : integer;
    {biến i ghi lại giá trị băm đầu tiên h(x)}
begin
    i := h (x);
    j := i;
    if (T [i] = x) or (T [i] = empty) then k := i
    else begin
        k := (i + 1) mod N;
        while (k < > i) and ( T [k] < > x) and (T [k] < > empty) do
            begin
                if (T [k] = deleted) and (T [j] < > deleted) then
                    j := k;
                k := (k + 1) mod N;
            end;
        if (T [k] = empty) and (T [j] < > deleted) then j := k;
    end;
end;

```

Sau đây ta sẽ mô tả các thủ tục và hàm thực hiện các phép toán từ điển

```

function Member (x : keytype; var T : Dictionary) : boolean;
var k, j : integer;
begin
    Location (x, k, j);
    if T [k] = x then Member := true
    else Member := false;
end;

```

```

procedure      Insert (x : keytype; var T : Dictionary);
var k, j : integer;
begin
    Location (x, k, j);
    if T [k] < > x then
        if (T [j] = deleted) or (T[j] = empty) then T [j] := x;
        else writeln (' bảng đầy')
    else writeln (' bảng đã có x');
end;

```

```

procedure      Delete (x : keytype; var T : Dictionary);
var k, j : integer;
begin
    Location (x, k, j);
    if T [k] = x then T [k] := deleted;
end;

```

## 5.5. PHÂN TÍCH VÀ ĐÁNH GIÁ CÁC PHƯƠNG PHÁP BĂM

Bảng băm là một cấu trúc dữ liệu rất thích hợp để biểu diễn từ điển và các kiểu dữ liệu trừu tượng khác được xây dựng trên khái niệm tập hợp. Trong mục này chúng ta sẽ so sánh những ưu điểm và hạn chế của hai phương pháp băm mở và băm đóng. Chúng ta cũng sẽ phân tích và đánh giá hiệu quả của từng phương pháp.

Trong bảng băm mở, mỗi thành phần T[i] của bảng chứa con trỏ trỏ tới danh sách liên kết các phần tử của tập hợp thuộc lớp thứ i. Do đó không cần thiết để biểu diễn tập hợp bởi bảng băm mở sẽ là không gian cần để lưu các bản ghi biểu diễn các phần tử của tập hợp cộng thêm không gian giành cho các con trỏ (mỗi con trỏ chỉ đòi hỏi một từ máy). Trong khi đó các bản ghi biểu diễn các phần tử của tập hợp sẽ được lưu giữ trong chính bảng băm đóng. Do đó, với bảng băm đóng một không gian nhớ cố định được giành để biểu diễn tập hợp. Bảng sẽ chứa một số

vị trí rỗng (càng nhiều vị trí rỗng thì càng hạn chế sự va chạm và tránh được hiện tượng đầy tràn). Như vậy, nếu các bản ghi có cỡ lớn (không gian nhớ cần cho mỗi bản ghi lớn), và ta sử dụng bảng băm đóng thì sẽ lãng phí một không gian đáng kể.

Một ưu điểm khác của bảng băm mở là không cần phải đặt ra vấn đề giải quyết sự va chạm, vì các phần tử thuộc cùng một lớp được tổ chức dưới dạng danh sách liên kết.

Sau đây chúng ta sẽ đánh giá thời gian trung bình cần để thực hiện mỗi phép toán trên từ điển trong các bảng băm.

### **Bảng băm mở**

Giả sử có N rổ  $T[0], T[1], \dots, T[N-1]$  và có M phần tử được lưu giữ trong bảng. Giả sử rằng, hàm băm phân phối đều các phần tử vào mỗi rổ. Do đó trung bình mỗi rổ chứa  $M/N$  phần tử. Vì vậy thời gian trung bình để thực hiện mỗi phép toán từ điển Insert, Delete và Member là 0 ( $M/N$ ). Nếu ta chọn  $N = M$  thì thời gian trung bình cho mỗi phép toán Insert, Delete và Member sẽ trở thành hằng số.

Cần lưu ý rằng, ta đã đánh giá thời gian trung bình để thực hiện mỗi phép toán từ điển với giả thiết hàm băm phân phối đều các phần tử cho mỗi rổ. Trên thực tế, giả thiết này khó được thực hiện. trong trường hợp xấu nhất, tất cả các phần tử đều được đưa vào cùng một rổ, thì thời gian trung bình cho mỗi phép toán sẽ tỉ lệ với cỡ của tập hợp như trong trường hợp danh sách.

### **Bảng băm đóng**

Sau đây chúng ta sẽ tiến hành đánh giá thời gian trung bình để thực hiện mỗi phép toán từ điển trong bảng băm đóng. ta sẽ sử dụng phương pháp xác suất để đánh giá.

Giả sử rằng, hàm băm h phân phối đều các phần tử của tập hợp trên các chỉ số của bảng. Giả sử ta cần phải đưa một phần tử vào bảng  $T$  có chiều  $N$  và bảng đã chứa  $k$  phần tử. Khi đó xác suất để trong lần đầu ta tìm ra được một vị trí trống là  $\frac{N-k}{N}$  ta gọi xác suất này là  $p_1$ , nó

chính là xác suất của sự kiện cần một lần thăm dò để đưa phần tử mới vào bảng. Xác suất  $p_2$  của sự kiện cần hai lần thăm dò để đưa phần tử

mới vào bảng sẽ bằng xác suất lần thăm dò thứ nhất gặp va chạm nhân với xác suất lần thăm dò thứ hai tìm được vị trí trống, tức là

$$p_2 = \frac{k}{N} \cdot \frac{N-k}{N-1}$$

Một cách tuần tự, ta tính được xác suất  $p_i$  của sự kiện cần i lần thăm dò để đưa phần tử mới vào bảng. Như vậy ta có

$$p_1 = \frac{N-k}{N}$$

$$p_2 = \frac{k}{N} \cdot \frac{N-k}{N-1}$$

$$p_3 = \frac{k}{N} \cdot \frac{k-1}{N-1} \cdot \frac{N-k}{N-2},$$

.....

$$p_i = \frac{k}{N} \cdot \frac{k-1}{N-1} \cdots \frac{k-i+2}{N-i+2} \cdot \frac{N-k}{N-i+1}$$

Cần lưu ý rằng, để đưa phần tử mới vào bảng đã chứa k phần tử đòi hỏi nhiều nhất là  $k + 1$  lần thăm dò. Từ công thức tính giá trị trung bình (phương sai) của một đại lượng ngẫu nhiên, ta tính được số trung bình các lần thăm dò để đưa một phần tử mới vào bảng đã chứa k phần tử

$$E_k = \sum_{i=1}^{k+1} i \cdot p_i = \frac{N+1}{N-k+1}$$

Ta có nhận xét rằng, số lần thăm dò cần để tìm kiếm một phần tử trong bảng cũng chính là số lần thăm dò để đưa nó vào bảng.

Giả sử bảng có chiều là N và nó chứa M phần tử. Khi đó số trung bình các lần thăm dò cần để tìm kiếm một phần tử trong bảng là :

$$\begin{aligned} E &= \frac{1}{M} \sum_{k=0}^{M-1} E_k = \frac{N+1}{M} \sum_{k=0}^{M-1} \frac{1}{N-k+1} \\ &= \frac{N+1}{M} \left( \frac{1}{N+1} + \frac{1}{N} + \dots + \frac{1}{N-M+2} \right) \\ &= \frac{N+1}{M} (H_{N+1} - H_{N-M+1}) \end{aligned}$$

trong đó,  $H_N = 1 + \frac{1}{2} + \dots + \frac{1}{N}$  là hàm điều hoà.

Giá trị gần đúng của hàm điều hoà được cho bởi công thức :

$$H_N = \ln N + \gamma + \frac{1}{2N} - \frac{1}{12N^2} + \frac{1}{120N^4} - \epsilon$$

trong đó  $0 < \epsilon < \frac{1}{252N^6}$ , còn  $\gamma = 0, 5772156649$  là hằng số Ole. Do đó ta có thể xem  $H_N \approx \ln N + \gamma$ .

Vậy

$$E = \frac{N+1}{M} [\ln(N+1) - \ln(N-M+1)]$$

Đặt  $\frac{M}{N+1} = \alpha$ , ta có

$$E = \frac{1}{\alpha} \ln \frac{N+1}{N-M+1} = -\frac{1}{\alpha} \ln \frac{N-M+1}{N+1} = -\frac{1}{\alpha} \ln(1-\alpha)$$

Số  $\alpha$  được gọi là *hệ số đầy*. vì nó gần bằng tỉ số giữa số phần tử có trong bảng và chiều của bảng. Với  $\alpha = 0$  có nghĩa là bảng trống, còn  $\alpha = \frac{N}{N+1}$  có nghĩa là bảng đã đầy. Công thức :

$$E = -\frac{1}{\alpha} \ln(1-\alpha)$$

cho phép ta tính được số trung bình E các lần thăm dò cần thiết để tìm kiếm, xen vào bảng một phần tử, theo hệ số đầy của bảng  $\alpha$ . Giá trị của  $\alpha$  và E tương ứng được cho trong bảng sau :

$\alpha$	E
0,1	1,05
0,25	1,15
0,5	1,39
0,75	1,55
0,9	2,56
0,95	3,15

Nhìn vào bảng này ta thấy, bảng băm đóng là một phương pháp cực kỳ có hiệu quả để cài đặt từ điển (tập hợp với các phép toán tìm kiếm, xen vào và loại bỏ), cũng như nhiều kiểu dữ liệu trừu tượng khác. Ngay cả khi bảng đã đầy tới 95%, thì cũng chỉ cần gần 3 lần thăm dò là tìm ra được phần tử cần tìm trong bảng, hoặc tìm ra được vị trí trống để đưa phần tử mới vào bảng.

Hạn chế căn bản của bảng băm đóng là không gian nhớ giành để lưu giữ các phần tử của tập hợp bị cố định. Vì vậy muốn vừa để tiết kiệm không gian nhớ vừa để tránh đầy tràn, ta cần phải đánh giá để lựa chọn chiều của bảng cho thích hợp.

## 5.6. HÀNG ƯU TIÊN

Trong mục này chúng ta sẽ xét kiểu dữ liệu trừu tượng hàng ưu tiên. Hàng ưu tiên là tập hợp cùng với hai phép toán Insert và DeleteMin. Phép toán Insert có ý nghĩa thông thường : xen phần tử mới vào tập hợp. Ta cần phải xác định phép toán DeleteMin. Giả sử Pri là hàm ưu tiên trên tập hợp A nào đó tức Pri là ánh từ từ tập A vào một tập P nào đó

$$\text{Pri} : A \rightarrow P$$

trong đó P là tập được sắp thứ tự tuyến tính (thông thường P là tập số nguyên hay tập số thực nào đó). Với mỗi  $a \in A$ , ta gọi Pri(a) là giá trị ưu tiên của a.

Phép toán DeleteMin trên tập A là tìm trên tập a phần tử a có Pri(a) nhỏ nhất và loại nó khỏi tập A.

Thuật ngữ : "hàng ưu tiên" có ý nghĩa như sau. Từ : "hàng" nói lên rằng, các phần tử thuộc tập hợp (người hoặc đối tượng nào đó) chờ đợi được "phục vụ". Từ "ưu tiên" có nghĩa rằng, sự phục vụ ở đây không được tiến hành theo chế độ "ai vào hàng trước được phục vụ trước" như đối với hàng đã xét trong chương 3 mà phụ thuộc vào mức ưu tiên được xác định trên các phần tử của hàng.

Chúng ta có thể kể ra rất nhiều ví dụ về hàng ưu tiên. Chẳng hạn, trong một hệ phân chia thời gian thường có nhiều nhiệm vụ chờ đợi được xử lý, trong đó có những nhiệm vụ cần được xử lý trước các nhiệm vụ khác. Khi đó tập hợp các nhiệm vụ chờ đợi được xử lý lập thành một hàng ưu tiên. Trong thực tế, việc mô phỏng các quá trình gồm các sự kiện độc lập với thời gian cũng dẫn đến việc xét các hàng ưu tiên.

### Cài đặt hàng ưu tiên

Chúng ta có thể biểu diễn hàng ưu tiên bởi danh sách được sắp hoặc không được sắp. Danh sách này có thể cài đặt bởi mảng hoặc dưới dạng danh sách liên kết. Tốt nhất ta biểu diễn hàng ưu tiên dưới dạng danh sách liên kết. Nếu danh sách được sắp xếp theo thứ tự tăng dần của giá trị ưu tiên của các phần tử, thì phần tử cần loại bỏ trong phép toán DeleteMin là phần tử đầu tiên trong danh sách, do đó chỉ cần thời gian không đổi để thực hiện phép toán này. Song để xen vào danh sách một phần tử mới, ta phải tìm vị trí thích hợp để đặt nó vào, do đó phép toán Insert đòi hỏi phải có thời gian  $O(n)$ , với danh sách có  $n$  phần tử. Nếu ta cài đặt hàng ưu tiên bởi danh sách liên kết không được sắp, thì khi xen phần tử mới vào hàng, ta chỉ cần đưa nó vào đầu danh sách. Nhưng việc thực hiện phép toán DeleteMin lại chậm.

Bạn đọc hãy viết (xem như bài tập) các thủ tục thực hiện các phép toán Insert và DeleteMin trong cách cài đặt hàng ưu tiên bởi danh sách liên kết được sắp và không được sắp.

Ta có nhận xét rằng, bảng băm không thích hợp để biểu diễn hàng ưu tiên. Lý do là, bảng băm không cho ta cách nào thuận tiện để tìm ra phần tử có giá trị ưu tiên nhỏ nhất.

Trong phần sau ta sẽ đưa ra một phương pháp mới để biểu diễn hàng ưu tiên.

## 5.7. CÂY THÚ TỰ BỘ PHẬN VÀ CÀI ĐẶT HÀNG UY TIÊN BỞI CÂY THÚ TỰ BỘ PHẬN

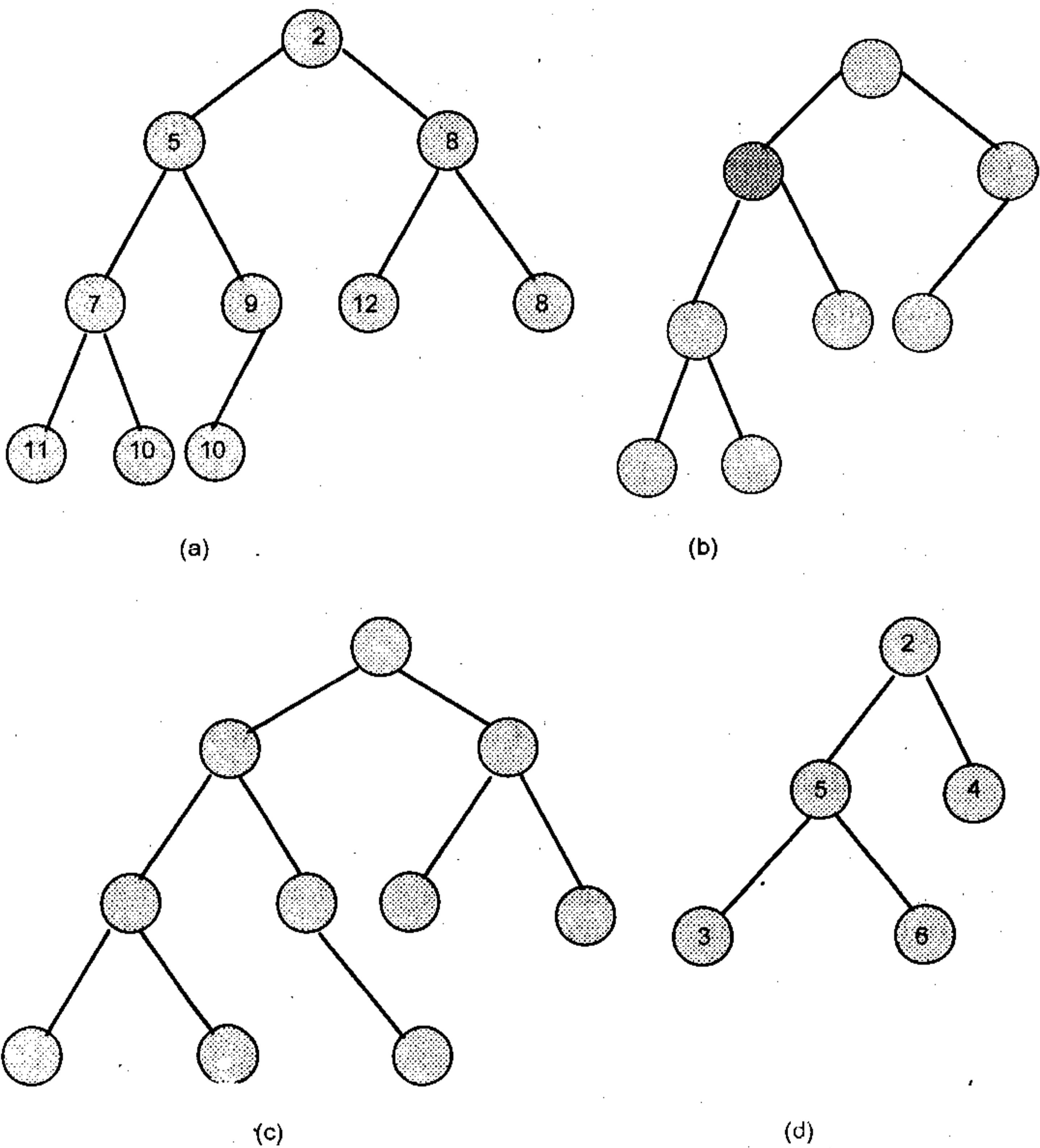
*Cây thứ tự bộ phận* (partially ordered tree hoặc heap) là một cây nhị phân gắn nhãn với các nhãn là các giá trị thuộc tập hợp được sắp thứ tự tuyến tính, sao cho những điều kiện sau đây được thực hiện.

1. Tất cả các mức của cây đều đầy, trừ mức thấp nhất có thể thiếu một số đỉnh.
2. Ở mức thấp nhất, tất cả các lá đều xuất hiện liên tiếp từ bên trái.
3. Giá trị của mỗi đỉnh không lớn hơn giá trị của các đỉnh con của nó.

Cần chú ý rằng, điều kiện 3 không đảm bảo cây thứ tự bộ phận là cây tìm kiếm nhị phân.

Hình 5.3 minh họa một số cây nhị phân với các giá trị của các đỉnh là các số nguyên được ghi trong mỗi đỉnh. Hình a là một cây thứ tự bộ

phận. Còn các hình b, c, d không phải là cây thứ tự bộ phận. Cây trong b vi phạm điều kiện 1, hình c vi phạm điều kiện 2, còn hình d vi phạm điều kiện 3.

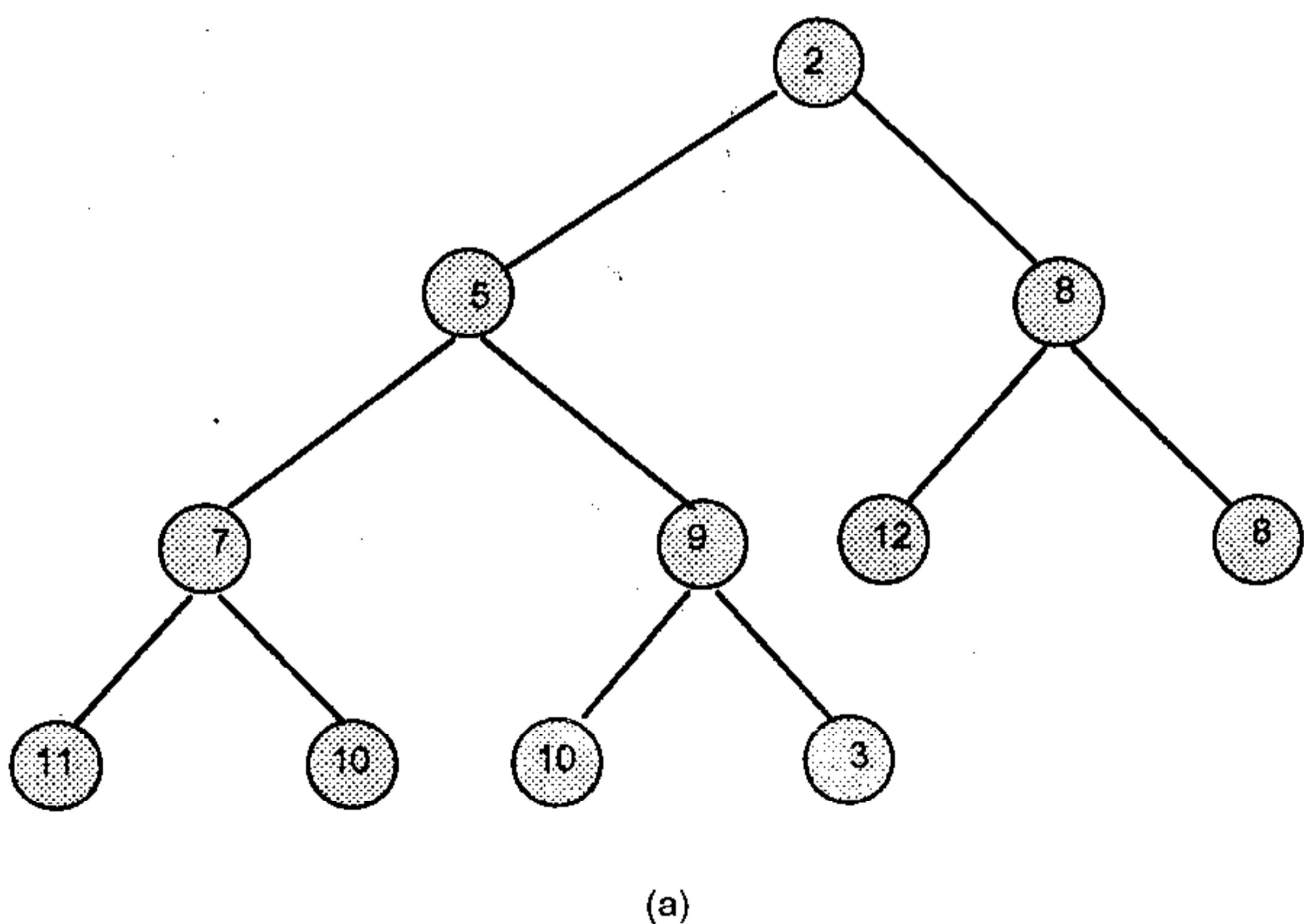


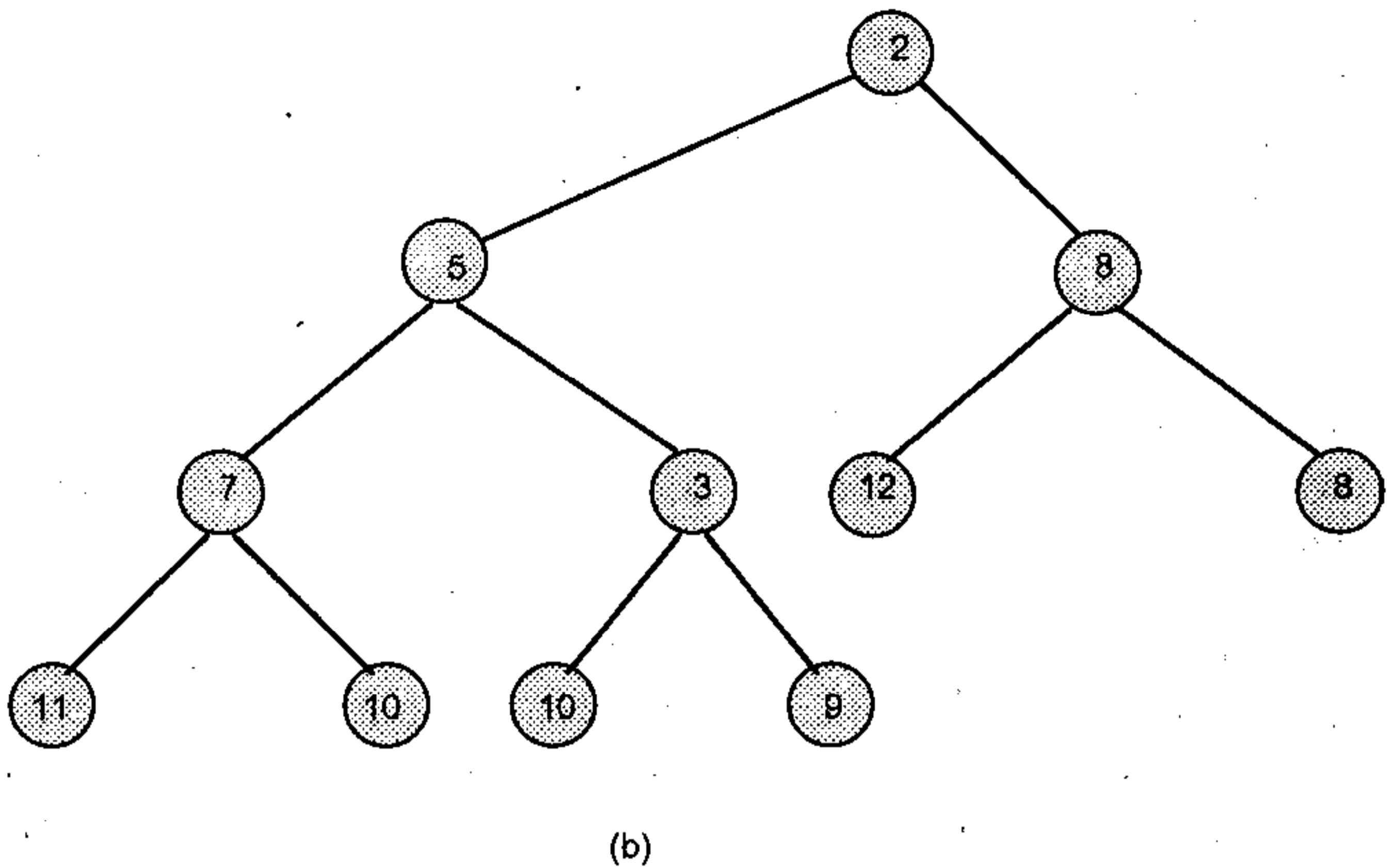
**Hình 5.3**

Khi lấy giá trị ưu tiên làm giá trị của mỗi đỉnh, ta có thể sử dụng cây thứ tự bộ phận để biểu diễn hàng ưu tiên. Sau đây ta sẽ xét xem các phép toán đổi với hàng ưu tiên được thực hiện trên cây thứ tự bộ phận như thế nào.

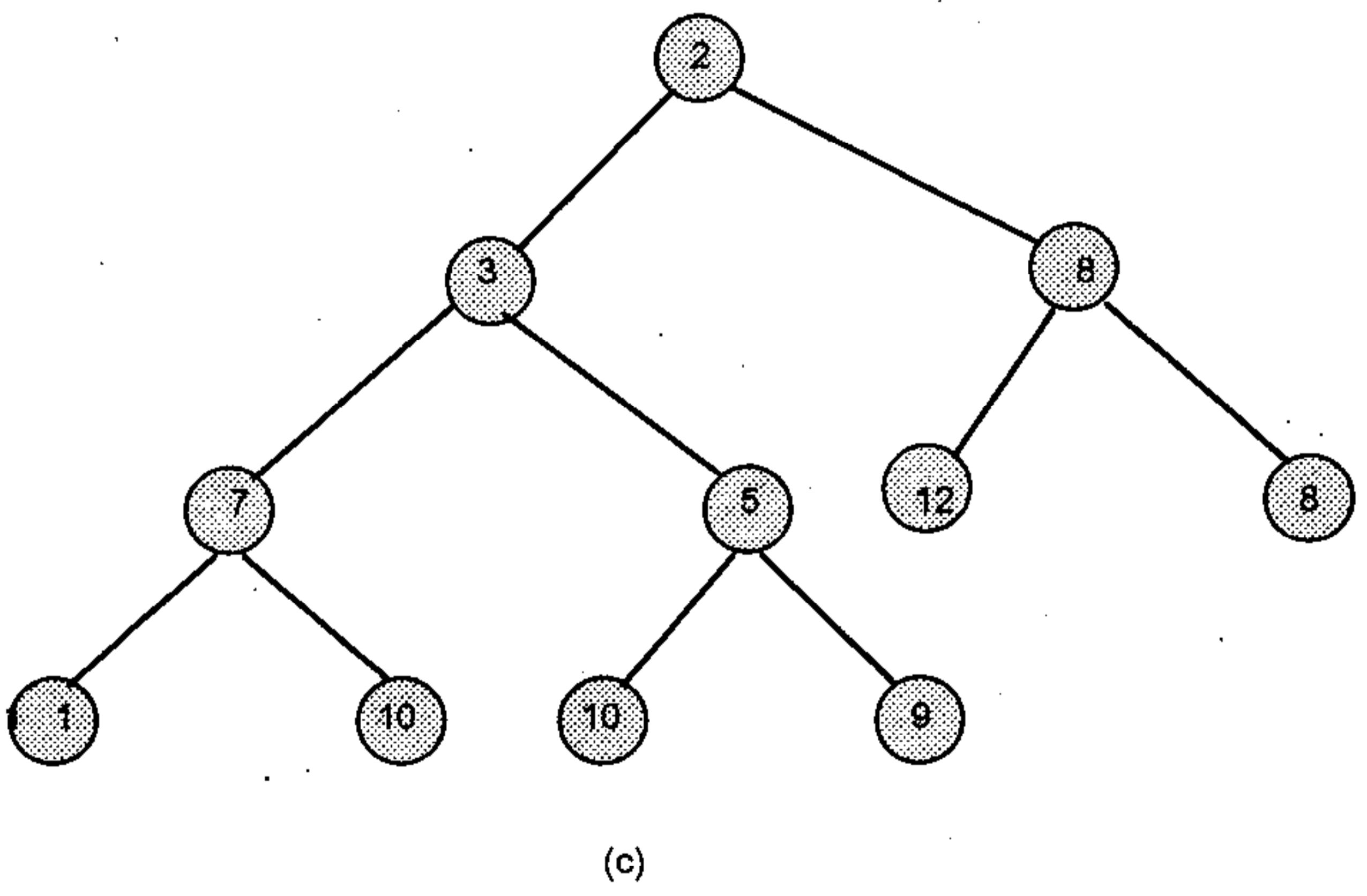
### **Phép toán Insert**

Để xen một phần tử mới vào cây thứ tự bộ phận, đầu tiên ta thêm một lá mới liền kề với các lá ở mức thấp nhất, nếu mức thấp nhất chưa đầy; còn nếu mức thấp nhất đầy, thì ta thêm vào một lá ở mức mới sao cho các điều kiện của 1 và 2 của cây thứ tự bộ phận được bảo tồn. Hình 5.4a minh họa cây sau khi thêm một lá mới với giá trị ưu tiên là 3 vào cây thứ tự bộ phận trong hình 5.3 a. Tất nhiên, cây nhị phân trong hình 5.4 a nói chung không còn là cây thứ tự bộ phận, vì điều kiện 3 có thể bị vi phạm. Nếu sau khi thêm vào lá mới cây không còn là cây thứ tự bộ phận, thì ta theo đường từ lá mới tới gốc cây. Nếu một đỉnh có giá trị ưu tiên nhỏ hơn đỉnh cha của nó, thì ta trao đổi đỉnh đó với cha của nó. Quá trình này được minh họa trong hình 5.4 b và c. Dễ dàng chứng minh được rằng, sau quá trình biến đổi trên ta có một cây thứ tự bộ phận (bài tập).





(b)

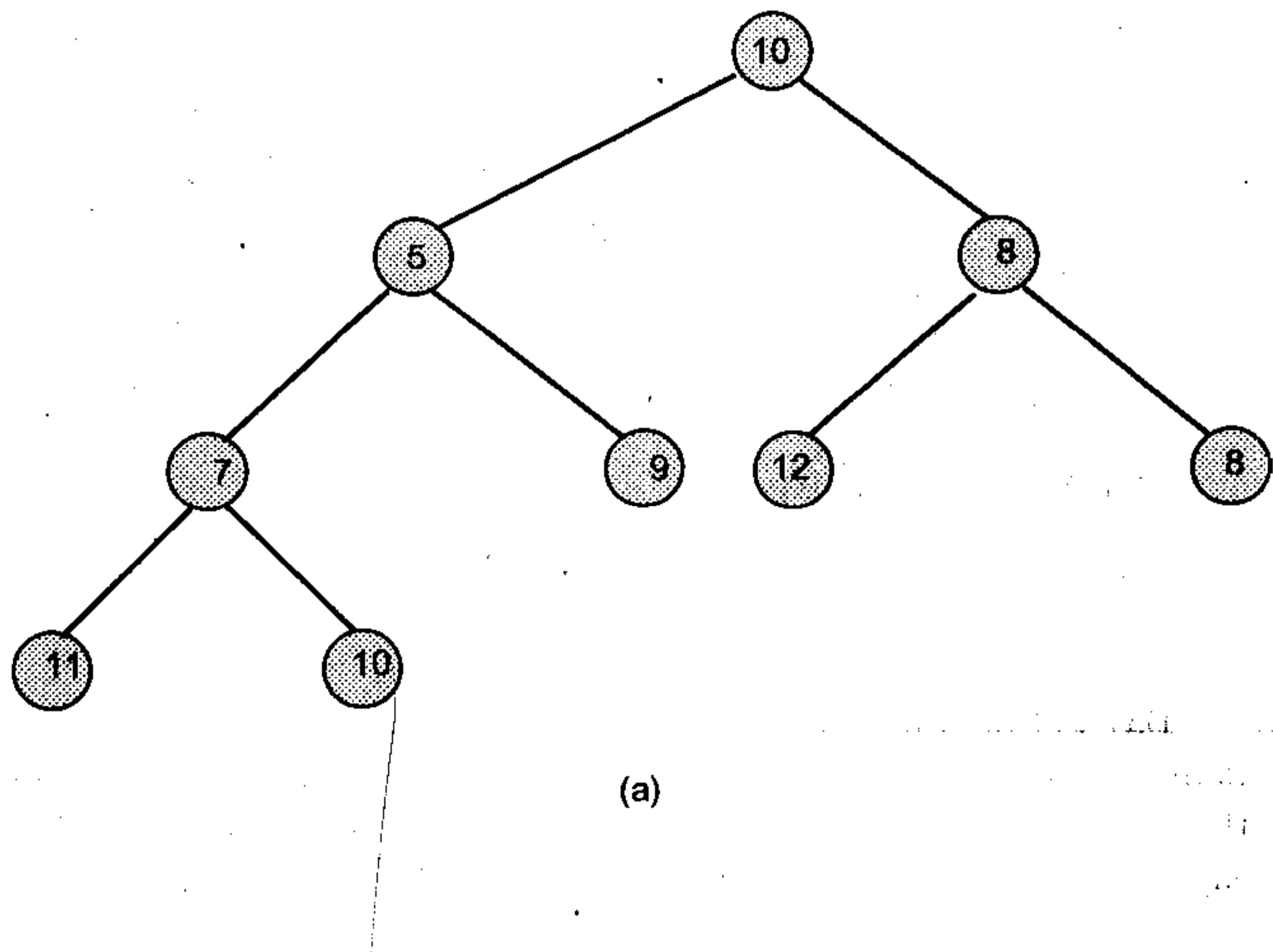


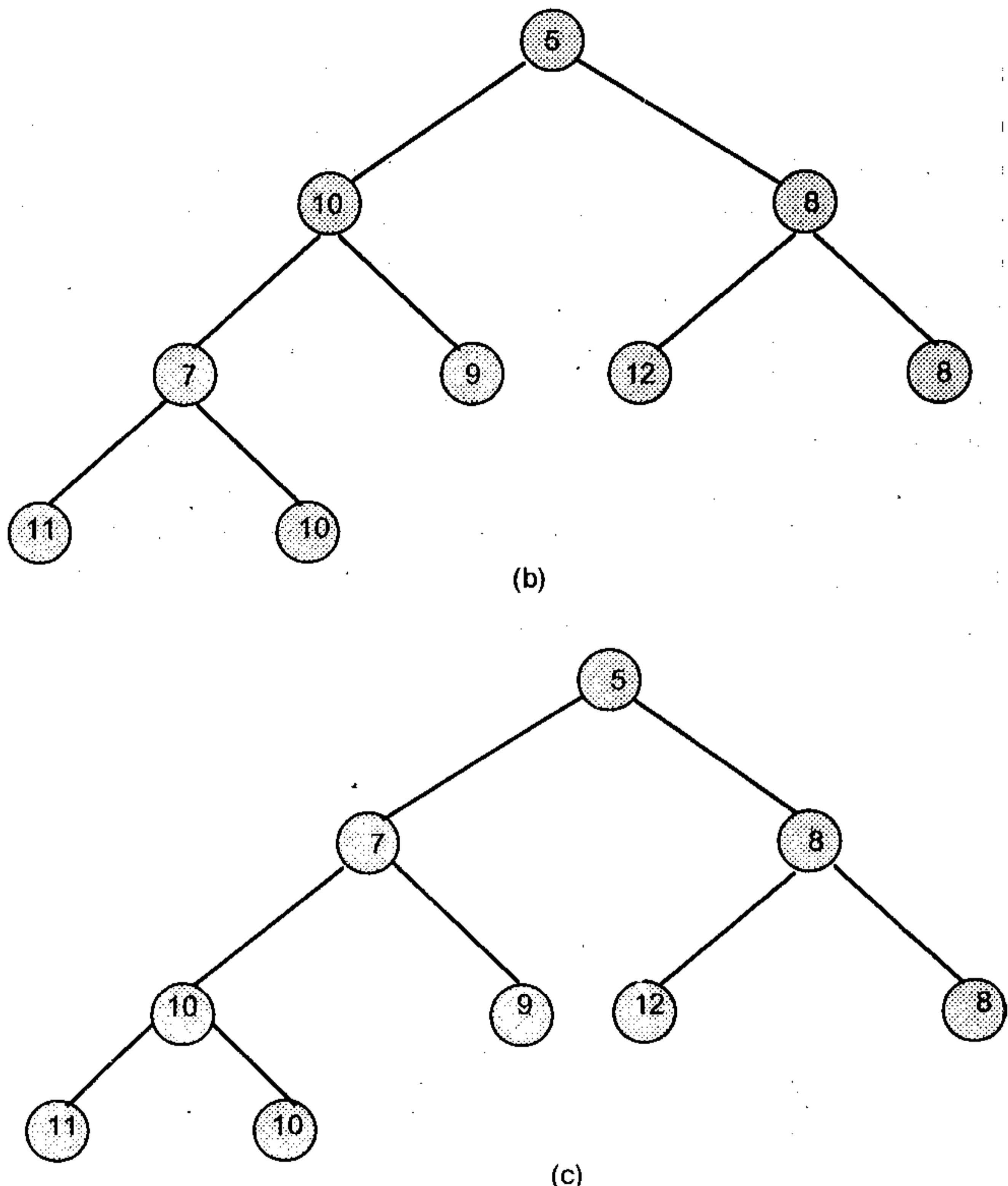
(c)

**Hình 5.4**

### Phép toán DeleteMin

Hiển nhiên là gốc của cây có giá trị ưu tiên nhỏ nhất. Tuy nhiên nếu loại bỏ gốc thì cây không còn là cây nữa. Do đó ta tiến hành như sau : đặt vào gốc phần tử của hàng ưu tiên chứa trong lá ngoài cùng bên phải ở mức thấp nhất, sau đó loại bỏ lá này khỏi cây. Hình 5.5 a minh họa cây nhận được từ cây trong hình 5.3 a sau phép biến đổi trên. Tới đây cây có thể không còn là cây thứ tự bộ phận, do điều kiện 3 của định nghĩa cây thứ tự bộ phận bị vi phạm ở gốc cây. Nay giờ ta đi từ gốc xuống. Giả sử tại một bước nào đó ta đang ở đỉnh a và hai đỉnh con của nó là b và c. Giả sử đỉnh a có giá trị ưu tiên lớn hơn giá trị ưu tiên của ít nhất một trong hai đỉnh b và c. Để xác định, ta giả sử rằng giá trị ưu tiên của đỉnh b không lớn hơn giá trị ưu tiên của đỉnh c,  $\text{pri}(b) \leq \text{pri}(c)$ . Khi đó ta sẽ trao đổi đỉnh a với đỉnh b và đi xuống đỉnh b. Quá trình đi xuống sẽ dừng lại cùng lầm là khi ta đạt tới một lá của cây. Ta có thể thấy quá trình diễn ra như thế nào trong hình 5.5 b và c.





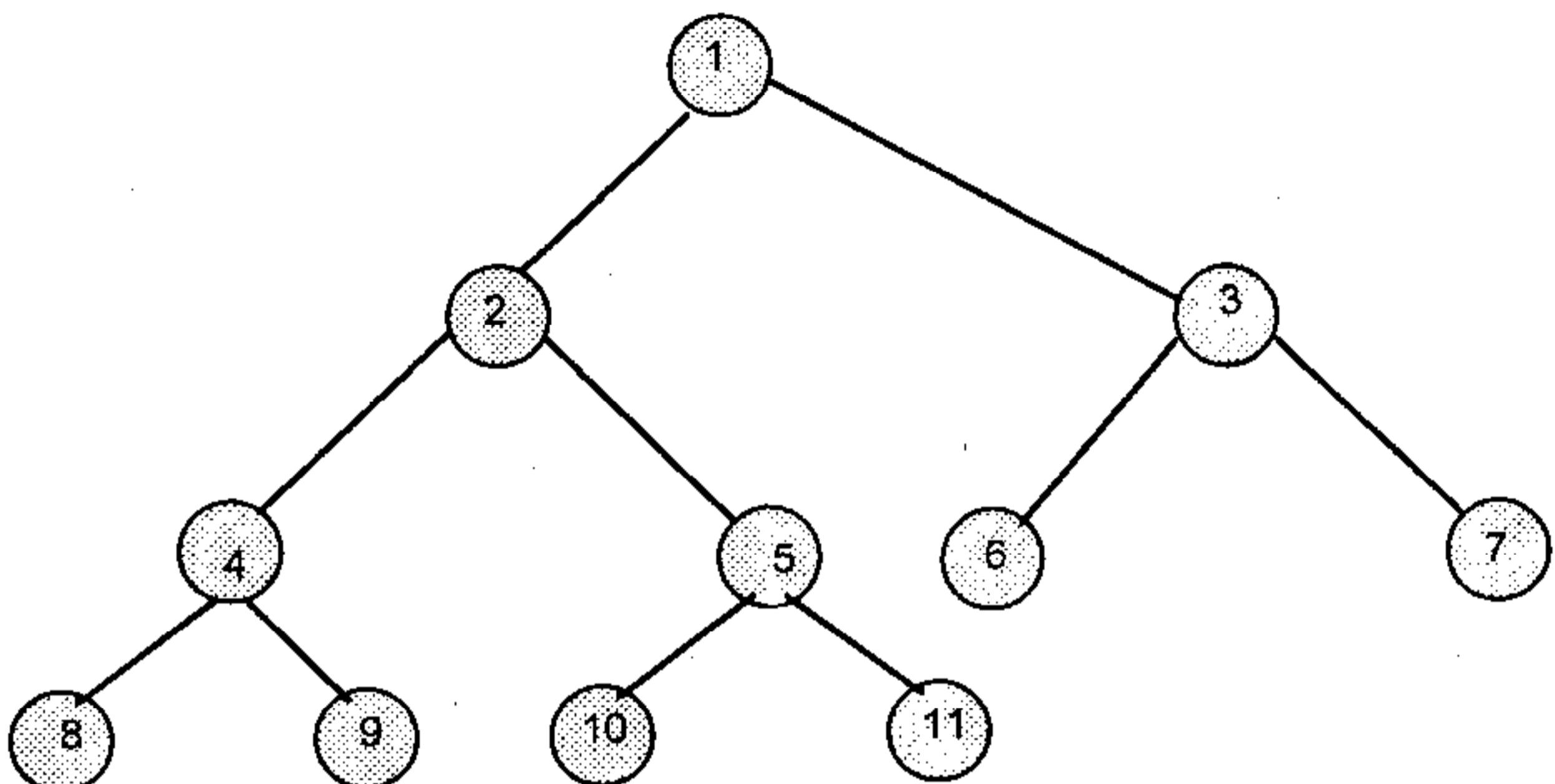
Hình 5.5

Bây giờ ta thử đánh giá thời gian cần thiết để thực hiện các phép toán Insert và DeleteMin đối với hàng ưu tiên được biểu diễn bởi cây thứ tự bộ phận. Giả sử hàng chứa  $n$  phần tử. Khi đó mọi đường đi trong cây không chứa nhiều hơn  $1 + \log n$  đỉnh. Thủ tục thực hiện các phép

toán đã trình bày ở trên đều chứa quá trình đi từ lá lên gốc hoặc ngược lại. Trong quá trình mỗi đỉnh đòi hỏi một thời gian không đổi c nào đó. Do đó thời gian để thực hiện mỗi phép toán cùng lăm là  $c(1 + \log n) \leq 2c \log n$ , với  $n \geq 2$ , tức là  $O(\log n)$ .

### Cài đặt cây thứ tự bộ phận bởi mảng

Giả sử ta đánh số các đỉnh của cây thứ tự bộ phận từ trên xuống dưới và từ trái sang phải (trong cùng một mức), bắt đầu từ gốc có số hiệu là 1 (xem hình 5.6)



Hình 5.6

Từ các điều kiện 1 và 2 trong định nghĩa của cây thứ tự bộ phận, ta nhận thấy rằng, nếu một đỉnh có số hiệu là  $i$ , thì đỉnh con bên trái (nếu có) của nó là  $2i$ , và đỉnh con bên phải (nếu có) của nó là  $2i + 1$ . Hay nói cách khác cha của đỉnh  $i$  là đỉnh  $i \text{ div } 2$  với  $i > 1$ . Nếu ta sử dụng mảng  $H$  với chỉ số chạy từ 1 đến  $N$  ( $N$  là số lớn nhất các phần tử có trong hàng ưu tiên), thì các đỉnh của cây kể từ gốc lần lượt theo các mức và trong cùng một mức được kể từ trái sang phải, sẽ chứa trong các thành phần của mảng  $H[1], H[2], \dots, H[N]$ . Do đó ta sẽ cài đặt hàng ưu tiên bởi mảng  $H$  cùng với một biến  $last$  để ghi chỉ số cuối cùng của thành phần mảng có chứa phần tử của hàng. Chúng ta có khai báo sau

```

const N = ...;

type PriQueue = record
    element : array [1.. N] of item;
    last : integer;
end;

```

{item là kiểu bản ghi nào đó mô tả phần tử của hàng ưu tiên}.

```
var H : PriQueue;
```

Việc khởi tạo một hàng rỗng được thực hiện bởi lệnh

```
H.last := 0;
```

Từ các thuật toán thực hiện các phép toán Insert và DeleteMin đã trình bày, ta dễ dàng viết được các thủ tục thực hiện các phép toán trên hàng ưu tiên.

```

procedure Insert (x : item; var H : PriQueue);
var i : integer;
    temp : item;
begin
    if H.last >= N then writeln ('hàng đầy')
    else begin
        H.last := H.last + 1;
        H.element [H.last] := x;
        i := H.last;
        {i ghi lại số hiệu lá mới thêm vào}
        while (i > 1) and (Pri(H.element [i]) < Pri(H.element[i div 2])) do
            begin
                temp := H.element [i];
                H.element [i] := H.element [i div 2];
                H.element [i div 2] := temp;
                i := i div 2;
            end;
        {vòng lặp while thực hiện quá trình di từ lá mới lên}
    end;
end;

```

```

procedure DeleteMin (var H : PriQueue; var x : item);
    {loại bỏ phần tử có giá trị ưu tiên nhỏ nhất khỏi hàng
    và phần tử này được lưu vào biến x}
var i, j : integer;
    {i ghi lại số hiệu của các đỉnh trong quá trình đi từ gốc xuống;
    j ghi lại một trong hai đỉnh con của i có giá trị ưu tiên nhỏ hơn đỉnh kia}.
    temp : item;
    Condition3 : boolean;
    {biến condition3 chỉ ra điều kiện 3 của cây thứ tự bộ phận có thỏa mãn hay
    không}
begin
    if H.last = 0 then writeln ('hàng rỗng')
    else begin
        x := H.element [1];
        H.element [1] := H.element [H.last];
        H.last := H.last + 1;
        i = 1;
        condition3 := false;
        while (i <= H.last div 2) and (not condition3) do
            begin
                if 2 * i = H.last then j := 2 * i
                else if Pri(H.element[2*i]) <= Pri(H.element[2*i+1]) then
                    j := 2 * i
                else j := 2 * i + 1;
                if Pri (H.element [i]) > Pri (H. element[j]) then
                    begin
                        temp := H.element [i];
                        H.element [i] := H. element [j];
                        H. element [j] := temp;
                        i := j;
                    end else
                    condition3 := true;
            end;
        end;
    end;

```

Trong thủ tục trên, vòng lặp while thực hiện quá trình đi từ gốc xuống và dừng lại khi điều kiện 3 được thỏa mãn.

## *Chương 6*

# BẢNG

Trong chương trước chúng ta đã nghiên cứu mô hình dữ liệu tập hợp và một số kiểu dữ liệu trừu tượng (từ điển, hàng ưu tiên) được xây dựng trên cơ sở khái niệm tập hợp. Trong chương này chúng ta sẽ nghiên cứu kiểu dữ liệu trừu tượng bảng được xây dựng trên cơ sở khái niệm hàm (ánh xạ). Chúng ta cũng sẽ xét việc cài đặt một trường hợp đặc biệt của bảng, đó là các bảng chữ nhật.

### **6.1. KIỂU DỮ LIỆU TRỪU TƯỢNG BẢNG**

Trước hết chúng ta nhắc lại khái niệm hàm trong toán học. Nhớ lại rằng, một quan hệ  $R$  từ tập  $A$  đến tập  $B$  là một tập con nào đó của tích đềcac  $A \times B$ , tức là  $R$  là một tập hợp nào đó các cặp  $(a, b)$  với  $a \in A$ ,  $b \in B$ . Một hàm  $f : A \rightarrow B$  ( $f$  là hàm từ  $A$  đến  $B$ ) là một quan hệ  $f$  từ  $A$  đến  $B$  sao cho nếu  $(a, b) \in f$  và  $(a, c) \in f$  thì  $b = c$ . Tức là, quan hệ  $f$  là một hàm, nếu nó không chứa các cặp  $(a, b), (a, c)$  với  $b \neq c$ . Nếu  $(a, b) \in f$ , thì ta nói  $b$  là giá trị của hàm  $f$  tại  $a$  và ký hiệu là  $f(a)$ ,  $b = f(a)$ . Tập hợp tất cả các  $a \in A$ , sao cho tồn tại cặp  $(a, b) \in f$ , được gọi là miền xác định của hàm  $f$  và ký hiệu là Dom ( $f$ ).

Có những hàm, chẳng hạn hàm  $f(x) = e^x$ , ta có thuật toán để xác định giá trị của hàm  $f(x)$  với mỗi  $x$ . Với những hàm như thế ta có thể cài đặt bởi các hàm trong Pascal hoặc C. Tuy nhiên có rất nhiều hàm. Chẳng hạn hàm cho tương ứng mỗi nhân viên làm việc trong một công ty quan với lương hiện tại của người đó, ta chỉ có thể mô tả bởi bảng lương. Trong các trường hợp như thế, để mô tả một hàm  $f : A \rightarrow B$ , ta phải lưu

giữ một bảng mô tả các thông tin về các đối tượng  $a \in A$  và các thông tin về các đối tượng  $b \in B$  tương ứng với mỗi  $a$ .

Một bảng với tập chỉ số  $A$  và tập giá trị  $B$  là một hàm  $f$  nào đó từ  $A$  đến  $B$  cùng với các phép toán sau đây :

**1. Truy xuất** : với chỉ số cho trước  $a$  thuộc miền xác định của hàm, tìm ra giá trị của hàm tại  $a$ .

**2. Sửa đổi** : với chỉ số cho trước  $a$  thuộc miền xác định của hàm, thay giá trị của hàm tại  $a$  bởi một giá trị khác cho trước.

**3. Xen vào** : thêm vào miền xác định của hàm một chỉ số mới và xác định giá trị của hàm tại đó.

**4. Loại bỏ** : loại một chỉ số nào đó khỏi miền xác định của hàm cùng với giá trị của hàm tại đó.

Đối với bảng, các phép toán truy xuất và sửa đổi là quan trọng nhất. Thông thường trong các áp dụng, khi đã lưu giữ một bảng, ta chỉ cần đến việc truy xuất thông tin từ bảng và sửa đổi thông tin trong bảng. Tuy nhiên trong một số áp dụng ta phải cần đến các phép toán xen vào và loại bỏ.

Sau đây chúng ta đưa ra một ví dụ về bảng. Một ma trận  $m$  hàng,  $n$  cột  $B = [b_{ij}]$  có thể xem như một bảng. Tập chỉ số  $A$  ở đây là tập các cặp  $(i, j)$  với  $i = 1, 2, \dots, M$  và  $j = 1, 2, \dots, N$ . Nếu ma trận là ma trận các số nguyên, ta có thể xét ma trận như một hàm  $f$  từ tập chỉ số đến tập các số nguyên, trong đó  $F(i, j) = b_{ij}$ . Sau này chúng ta sẽ gọi các bảng mà tập chỉ số là tập các cặp  $(i, j)$  với  $i = 1, 2, \dots, M$  và  $j = 1, 2, \dots, N$  là các bảng chữ nhật có  $M$  hàng và  $N$  cột.

## 6.2. CÀI ĐẶT BẢNG

### 6.2.1. Cài đặt bảng bởi mảng

Giả sử tập chỉ số của bảng là một kiểu đơn có thể dùng làm kiểu chỉ số của một mảng. Trong Pascal kiểu chỉ số của mảng có thể là miền con của các số nguyên, chẳng hạn  $1 \dots 1000$ ; có thể là kiểu ký tự hoặc miền con của kiểu ký tự, chẳng hạn ' $'A'$  .. ' $'Z'$ ', có thể là một kiểu liệt kê hoặc miền con của kiểu liệt kê nào đó. Trong trường hợp này, ta có thể biểu

diễn bảng bởi mảng. Để chỉ rằng, tại một chỉ số nào đó hàm không xác định, ta đưa thêm vào một giá trị mới undefined (không xác định) khác với tất cả các giá trị thuộc tập giá trị của bảng. Tại các chỉ số mà hàm không xác định, ta sẽ gán cho các thành phần của mảng tại các chỉ số đó giá trị undefined.

Ta có thể khai báo kiểu bảng như sau

**typetable = array [indextype] of valuetype;**

{indextype là kiểu chỉ số, valuetype là kiểu giá trị của bảng bao gồm giá trị undefined}.

**var T : table;**

**i : indextype;**

Giả sử value1, value2 là chỉ số đầu tiên và cuối cùng, khi đó việc khởi tạo một bảng rỗng (ánh xạ không xác định khắp nơi) được thực hiện bởi lệnh

**For i := value1 to value2 do T [i] := undefined;**

Việc cài đặt bảng bởi mảng cho phép ta truy cập trực tiếp đến mỗi thành phần của bảng. Các phép toán đối với bảng được thực hiện rất dễ dàng (Bạn đọc có thể thấy ngay cần phải làm gì) và chỉ đòi hỏi một thời gian O (1). Cần chú ý rằng, nếu tập chỉ số của bảng không thể dùng làm kiểu chỉ số của mảng, nhưng có thể mã hóa bởi một kiểu chỉ số nào đó của mảng, thì ta cũng có thể cài đặt bảng bởi mảng. Quá trình cài đặt gồm hai giai đoạn, đầu tiên là mã hóa tập chỉ số để có một kiểu chỉ số của mảng, sau đó mới dùng mảng.

### 6.2.2. Cài đặt bảng bởi danh sách

Vì một bảng với tập chỉ số A và tập giá trị B có thể xem như một tập nào đó các cặp (a, b), trong đó  $a \in A$  và  $b \in B$  là giá trị tương ứng với a. Do đó, ta có thể biểu diễn bảng bởi danh sách các cặp (a, b).

Nói cụ thể hơn, ta có thể cài đặt bảng bởi danh sách các phần tử, mỗi phần tử là một bản ghi có dạng :

```

type    element = record
            index : indextype;
            value : valuetype;
        end;

```

Ở đây danh sách có thể được cài đặt bởi một trong các cách mà ta đã xét trong chương 3. Tức là ta có thể cài đặt bởi danh sách kế cận (dùng mảng) hoặc danh sách liên kết. Các phép toán đối với bảng được qui về các phép toán tìm kiếm, xen vào và loại bỏ trên danh sách. Rõ ràng là, với cách cài đặt này, các phép toán đối với bảng được thực hiện kém hiệu quả, vì chúng đòi hỏi thời gian trung bình tỉ lệ với số thành phần của bảng.

Nếu có một thứ tự tuyến tính xác định trên tập chỉ số của bảng, ta nên cài đặt bảng bởi danh sách được sắp theo thứ tự đã xác định trên tập chỉ số.

### 6.2.3. Cài đặt bảng bởi bảng băm

Trong nhiều cảnh huống, bảng băm là cấu trúc dữ liệu thích hợp nhất để cài đặt một bảng.

Việc xây dựng các bảng băm (mở hoặc đóng) để biểu diễn một bảng hoàn toàn giống như việc xây dựng bảng băm cho từ điển. Chúng ta chỉ cần lưu ý một số điểm khác sau đây.

Các hàm băm sẽ "băm" các phần tử của tập chỉ số A của bảng vào các 'rổ'. Tức là nếu bảng băm gồm N rổ thì hàm băm là hàm h từ tập chỉ số A vào tập {0, 1 . . . N-1}.

Trong bảng băm mở, đối với từ điển ta có mỗi rổ là một danh sách các phần tử của từ điển; Còn đối với bảng, với tập chỉ số A và tập giá trị B thì mỗi rổ là một danh sách nào đó, các cặp (a, b) trong đó a ∈ A, b ∈ B. Chính xác hơn, cấu trúc dữ liệu bảng băm mở biểu diễn bảng được khai báo như sau :

```

type  pointer = ^cell;
        cell   = record
                    index : indextype;

```

```

    value : valuetype;
    next : pointer;
  end;
  table = array [0..N-1] of pointer;

```

Hiển nhiên bảng băm đóng biểu diễn bảng sẽ có cấu trúc được mô tả như sau

**type**      **table** = **array** [0..N-1] **of** **element**;  
 trong đó, element là bản ghi đã khai báo trong mục 6.2.2.

Trong cách cài đặt bảng bởi bảng băm (mở hoặc đóng), phép toán truy xuất và sửa đổi bảng chính là phép tìm kiếm trên bảng băm theo chỉ số  $a \in A$  và đọc hoặc thay đổi giá trị của trường value của bản ghi có trường index là  $a$ . Còn phép toán xen vào và loại bỏ trên bảng là phép toán xen vào và loại bỏ trên bảng băm theo chỉ số đã cho.

### 6.3. BẢNG CHỮ NHẬT

Trong mục này chúng ta sẽ xét việc cài đặt các bảng chữ nhật, tức là các bảng mà các thành phần của bảng được xếp thành hình chữ nhật gồm  $M$  hàng và  $N$  cột. Vì tầm quan trọng đặc biệt của các bảng chữ nhật, nên trong hầu hết các ngôn ngữ lập trình bậc cao đều có phương tiện thuận tiện và hiệu quả để biểu diễn bảng chữ nhật, đó là mảng hai chiều. Chẳng hạn, trong Pascal một bảng chữ nhật  $M$  hàng và  $N$  cột có khai báo

**type**      **table** = **array** [1..M,1..N] **of** **elementtype**;  
 trong đó **elementtype** là kiểu của các phần tử của bảng.

Cách tự nhiên nhất để đọc thông tin trong một bảng chữ nhật là lần lượt theo hàng và trong một hàng từ trái sang phải. Đó cũng chính là cách mà các chương trình dịch xếp các thành phần vào các vùng nhớ liên tiếp của bộ nhớ trong. Do đó, nếu  $T$  là một *table* và biết được địa chỉ của vùng nhớ lưu giữ thành phần  $T[1,1]$ , ta sẽ xác định được ngay địa chỉ của  $T[i, j]$ , với  $i, j$  bất kỳ. Mảng  $T$  sẽ được giành riêng một không

gian nhớ cố định gồm  $M \times N$  đơn vị nhớ liên tiếp (ở đây, đơn vị nhớ được xem là vùng nhớ để lưu giữ một thành phần của mảng).

Trong nhiều bài toán, ta không cần thiết phải biểu diễn thông tin ở tại mọi vị trí trong bảng. Có thể xảy ra, trong một bảng chỉ có một số giá trị tại một số vị trí là có nghĩa, còn các giá trị tại các vị trí còn lại là bằng nhau hoặc ta không cần quan tâm đến. Chẳng hạn như bảng các số nguyên trong hình 6.1.

6	0	0	7	0
0	0	0	0	0
0	8	0	1	9
3	0	0	0	0

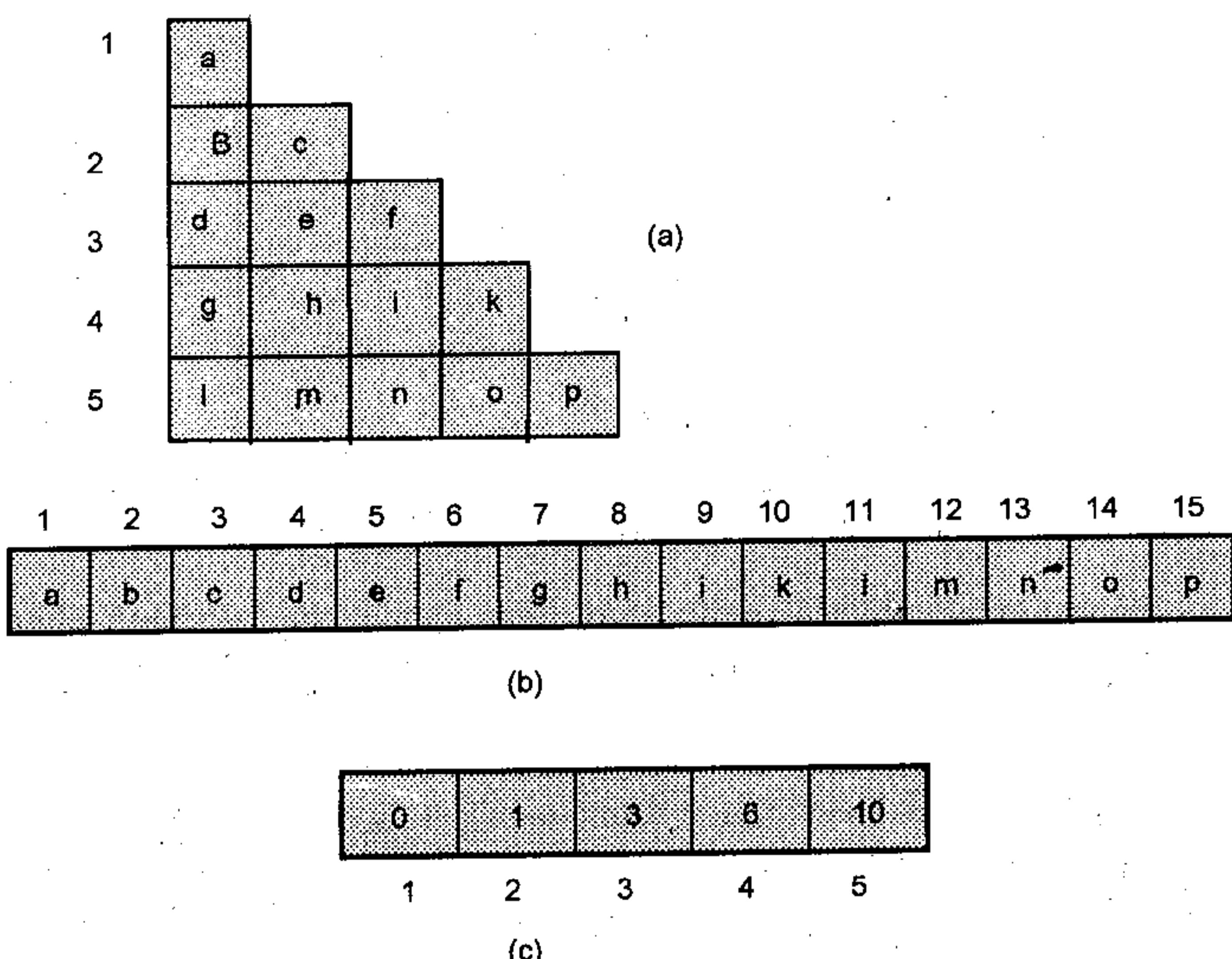
Hình 6.1. Một ví dụ về bảng thưa thớt

Bảng này chỉ chứa 6 thành phần khác không, còn 14 thành phần còn lại bằng 0. Các bảng như thế gọi là các *bảng thưa thớt*. Hiển nhiên nếu cài đặt các bảng thưa thớt bởi mảng hai chiều sẽ lãng phí nhiều bộ nhớ. Chẳng hạn một ma trận nguyên  $200 \times 200$ , mỗi hàng không có quá 4 thành phần khác 0, nếu dùng mảng sẽ dùng đến 80.000 byte. Trong khi đó, nếu dùng phương pháp thích hợp, có thể chỉ cần đến  $1/10$  không gian nhớ đó. Sau đây ta sẽ nghiên cứu việc cài đặt những bảng có dạng đặc biệt.

### 6.3.1. Bảng tam giác và bảng răng lược

Bảng tam giác là bảng vuông (số dòng bằng số cột) mà tất cả các thành phần có nghĩa trong bảng đều nằm ở các vị trí  $(i, j)$  với  $j \leq i$ ). Ví dụ bảng trong hình 6.2a là bảng tam giác, phần chứa thông tin có nghĩa đều nằm ở các vị trí  $(i, j)$  với  $j \leq i$ . Với bảng tam giác  $n$  dòng, ta chỉ cần lưu giữ  $1 + 2 + \dots + n = n(n + 1)/2$  thành phần. Ta sẽ dùng một mảng một chiều để lưu giữ các thành phần của bảng (hình 6.2b). Các thành

phần của bảng lần lượt theo dòng, trong một dòng kể từ trái sang phải, được lưu vào các thành phần của mảng. Để biết được thành phần của mảng T chứa thành phần của bảng tại vị trí  $(i, j)$  bất kỳ, ta đưa vào một mảng phụ P. Mảng này có số chiều bằng số dòng của bảng. Với mỗi dòng  $i$ ,  $1 \leq i \leq n$ ,  $P[i]$  chứa vị trí trong mảng T kể từ đó ta sẽ lưu giữ các thành phần của bảng ở dòng  $i$  (hình 6.2c)



**Hình 6.2. Bảng tam giác**

Ta dễ dàng tính được các giá trị của mảng P

$$P[1] = 0$$

$$P[2] = 1$$

$$P[3] = 2 + 1 = 3$$

$$\dots \dots \dots$$

$$P[ ] = i - 1 + P[i - 1]$$

Biết được các  $P[i]$ , ta xác định được thành phần của mảng  $T$  lưu giữ thành phần của bảng tại vị trí  $(i, j)$  bất kỳ. Cụ thể, thành phần của bảng tại vị trí  $(i, j)$  được lưu giữ tại vị trí  $P[i] + j$  của mảng  $T$ . Chẳng hạn, ký tự  $h$  ở vị trí  $(4, 2)$  trong bảng được lưu giữ ở vị trí  $P[4] + 2 = 6 + 2 = 8$  trong mảng  $T$ . Như vậy, ta đã cài đặt một bảng tam giác bởi hai mảng một chiều  $T$  và  $P$ . Như trên đã chứng tỏ với cách cài đặt này ta có thể truy cập trực tiếp đến từng thành phần của bảng.

Một bảng răng lược là bảng chữ nhật mà trong mỗi dòng các thông tin trong bảng được xếp liên tục kể từ cột thứ nhất (số phần tử trong mỗi dòng nhiều ít tuỳ ý). Hình 6.3 minh họa một bảng răng lược.

1	a	b	c				
2	d	e	f	g	h	i	k
3							
4	l	m					
5	n	o	p	q	r		
6	s	t	u	v			

Hình 6.3. Minh họa một bảng răng lược

Bằng cách hoàn toàn tương tự như đối với bảng tam giác, ta có thể cài đặt bảng răng lược bởi hai mảng một chiều  $T$  và  $P$ . Các thành phần của bảng răng lược cũng được xếp vào mảng  $T$  lần lượt theo hàng và theo cột. Điều khác nhau duy nhất ở đây là, giá trị chứa trong mỗi thành phần khác nhau của mảng  $P$  được xác định như sau :

$$P[1] = 0$$

$P[i] = P[i - 1] + \text{số thành phần của bảng ở dòng } i - 1 \text{ với mọi } i > 1.$   
 Chẳng hạn, với bảng trong hình 6.2, ta có  $P[1] = 0$ ,  $P[2] = 3$ ,  $P[3] = 10$ ,  $P[4] = 10$ ,  $P[5] = 12$ ,  $P[6] = 17$ .

### 6.3.2. Bảng thưa thớt

Bảng thưa thớt là bảng chữ nhật mà các thông tin có nghĩa trong bảng được phân bố một cách thưa thớt, rải rác không theo một qui luật nào cả. Hình 6.1 cho ta một ví dụ về bảng thưa thớt, thông tin chứa trong bảng là các số nguyên. Natürlich ở đây là không thể dùng mảng hai chiều để biểu diễn một bảng thưa thớt, vì lãng phí nhiều bộ nhớ, ta cũng không thể dùng mảng một chiều để lưu giữ các thành phần của bảng như ta đã làm đối với bảng tam giác và bảng răng lược. Nguyên nhân là vì, các thành phần có nghĩa của bảng phân bố không theo một qui luật nào, nên ta không thể định vị được thành phần của bảng ở trong mảng một chiều.

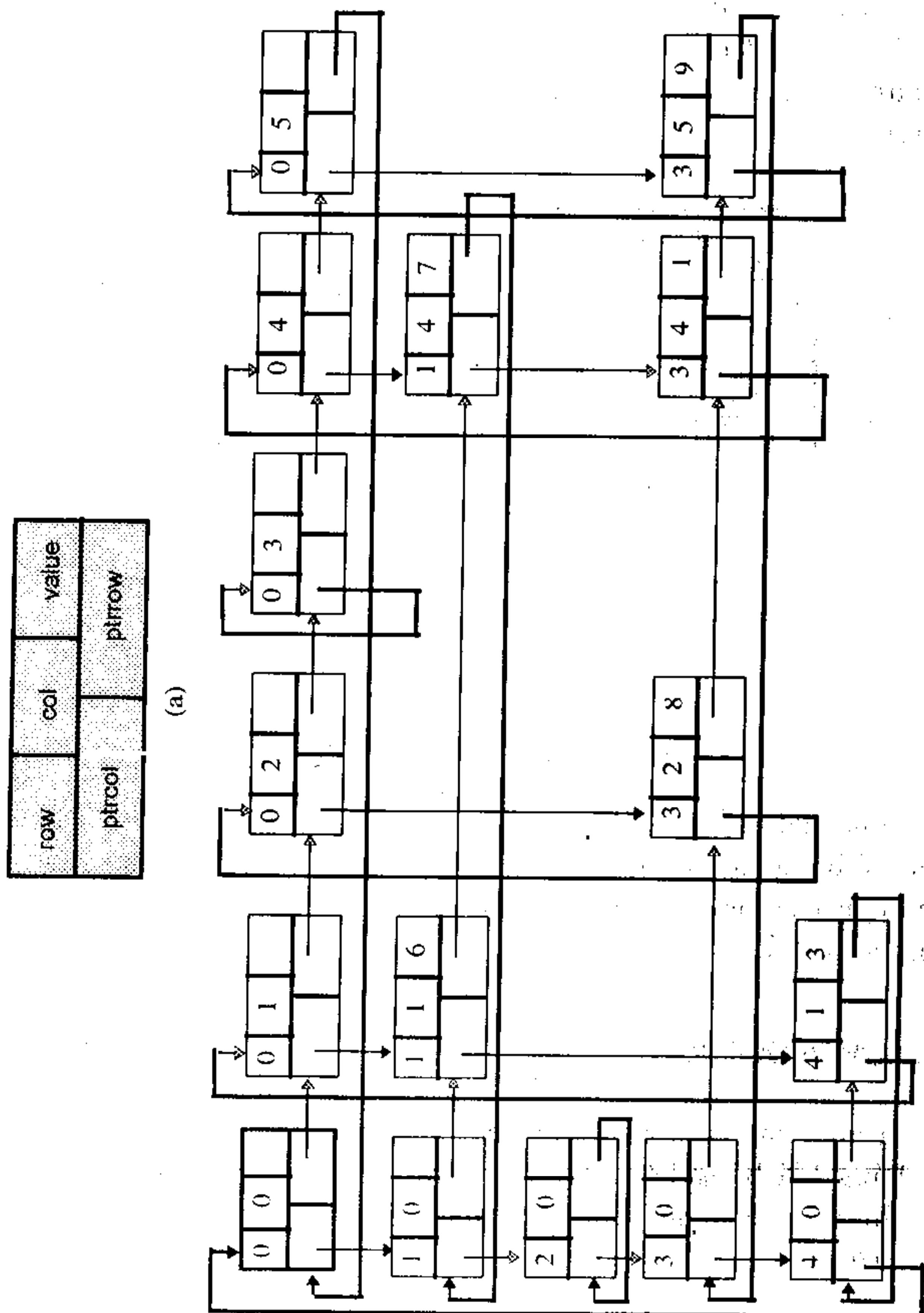
Một phương pháp tốt để cài đặt các bảng thưa thớt là dùng các danh sách liên kết để biểu diễn các hàng và các cột của bảng. Mỗi thành phần của bảng ở vị trí  $(i, j)$  được đưa vào hai danh sách: danh sách các thành phần của bảng ở dòng thứ  $i$  và danh sách các thành phần của bảng ở cột thứ  $j$ . Tức là mỗi thành phần của bảng được biểu diễn bởi một bản ghi có kiểu element được khai báo như sau:

```

type pointer = ^ element;
element = record
    row : integer;
    col : integer ;
    value : valuetype;
    ptrrow : pointer;
    ptrcol : pointer;
end;

```

trong đó **row** và **col** là chỉ số hàng và cột; **ptrrow** và **ptrcol** là con trỏ liên kết trong danh sách hàng và danh sách cột; còn **value** là giá trị của mỗi thành phần. Ta sẽ biểu diễn mỗi bản ghi dưới dạng hình 6.4a. Mỗi hàng và mỗi cột của bảng được biểu diễn bởi danh sách liên kết, vòng tròn và có đầu. Đầu của mỗi hàng có trường  $col = 0$ , còn đầu của mỗi cột có trường  $row = 0$ . Khi đó, cấu trúc dữ liệu biểu diễn bảng trong hình 6.1 được minh họa trong hình 6.4b.



Hình 6.4

Một phương pháp khác để cài đặt một bảng thưa thớt là sử dụng hai mảng. Một mảng hai chiều có cỡ như cỡ của bảng, mảng sẽ có giá trị là 1 tại các vị trí mà giá trị của bảng có ý nghĩa và có giá trị là 0 tại các vị trí khác. Ví dụ để biểu diễn bảng trong hình 6.1, ta sử dụng mảng được minh họa bởi hình 6.5 a. Bên cạnh mảng hai chiều chỉ chứa 1 hoặc 0, ta sẽ sử dụng một mảng một chiều để lưu lại các giá trị có nghĩa của bảng. Chẳng hạn, hình 6.5b minh họa mảng một chiều ứng với bảng trong hình 6.1

1	0	0	1	0
0	0	0	0	0
0	1	0	1	1
1	0	0	0	0

(a)

6	7	8	1	9	8
---	---	---	---	---	---

(b)

**Hình 6.5.** Mảng một chiều ứng với bảng trong hình 6.1

Hiệu quả tiết kiệm bộ nhớ của phương pháp này là rõ ràng. Chẳng hạn, đối với bảng các số nguyên (như trong hình 6.1), thay cho việc sử dụng mảng các số nguyên (mỗi số nguyên cần 2 byte = 16 bit) ta đã sử dụng mảng các bit.

## 6.4. TRÒ CHƠI ĐỜI SỐNG

Trong mục này chúng ta trình bày một áp dụng của phương pháp cài đặt bảng bởi bảng băm để giải quyết bài toán 'trò chơi đời sống' (game of life). Trò chơi đời sống được nhà toán học Anh J.H Conway đưa ra năm 1970.

Đời sống của một cộng đồng các cơ thể sống diễn ra trên một lưới ô vuông không giới hạn. Mỗi ô vuông có thể có một cơ thể sống hoặc không. Ô vuông có một cơ thể sống gọi là tế bào sống, ngược lại là tế bào chết. Các tế bào thay đổi từ thế hệ này sang thế hệ sau tùy thuộc vào các tế bào sống ở lân cận. Mỗi tế bào có tám tế bào lân cận tiếp giáp với tế bào đã cho theo các cạnh và các góc.

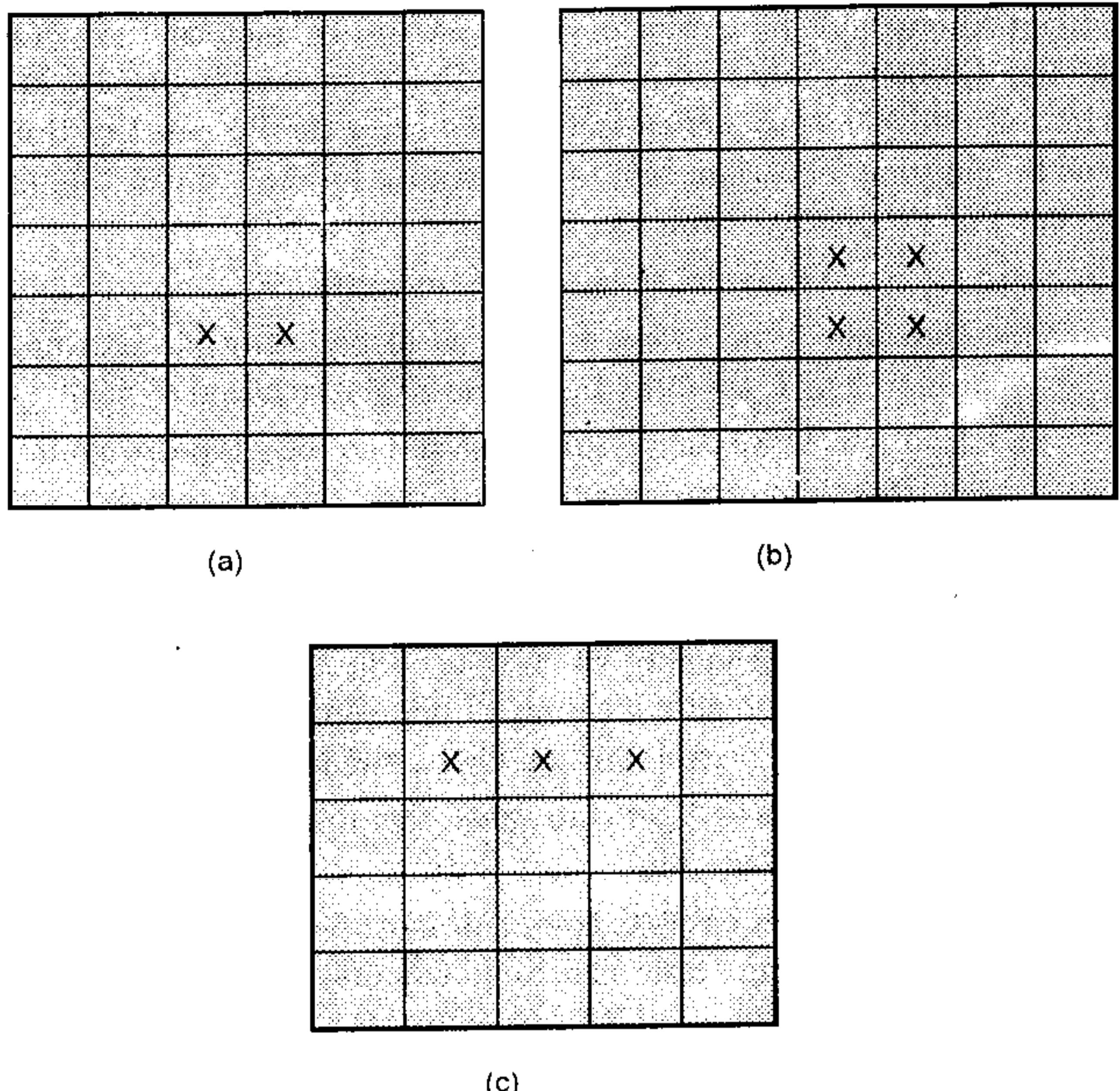
Các tế bào sẽ thay đổi theo các qui luật sau

1. Nếu một tế bào sống, nhưng số tế bào sống lân cận nó không nhiều hơn 1, thì ở thế hệ sau nó sẽ chết (chết vì cô độc)
2. Nếu một tế bào sống, nhưng số tế bào sống lân cận nó là 2 hoặc 3, thì ở thế hệ sau nó vẫn sống.
3. Nếu một tế bào sống, nhưng số tế bào sống lân cận nó không ít hơn 4, thì ở thế hệ sau nó sẽ chết (chết vì quá đông!).
4. Nếu một tế bào chết và có đúng 3 tế bào sống ở lân cận, thì ở thế hệ sau nó sẽ trở thành tế bào sống. Trong trường hợp còn lại, một tế bào chết vẫn còn chết ở thế hệ sau.
5. Trong mỗi thế hệ, sự sinh ra và chết đi diễn ra đồng thời, không ảnh hưởng đến nhau.

Sau đây ta sẽ xét sự phát triển của một số cộng đồng tế bào mà thế hệ đầu tiên được cho trong các hình 6.6 (dấu chéo đánh dấu tế bào sống). Cộng đồng trong hình 6.6a sẽ biến mất sau một thế hệ, vì cả hai tế bào cùng chết do cô độc. Cộng đồng trong hình 6.6 b sẽ ổn định từ thế hệ này sang thế hệ khác, vì không có tế bào nào sinh ra cũng không có tế bào nào chết đi. Sự phát triển của cộng đồng trong hình 6.6c giàn cho bạn đọc.

Chúng ta có thể thấy rằng, từ những cảnh huống ban đầu nhỏ bé, một số cộng đồng có thể phát triển thành những cộng đồng rộng lớn, một số cộng đồng có thể thay đổi và ổn định sau một số thế hệ, hoặc có thể lặp đi lặp lại một số cảnh huống nào đó; một số khác có thể mất đi.

Sau khi ra đời không lâu, trò chơi đời sống đã được Martin Gardner bàn tới trong báo 'Scientific American'. Từ đó nó đã thu hút sự chú ý của nhiều người.



**Hình 6.6.** Thế hệ đầu tiên của một số cộng đồng tế bào

Trò chơi đời sống là một bài tập lập trình rất hay cho những người mới học lập trình và cho cả những ai đã nắm được những cấu trúc dữ liệu phức tạp.

Đầu tiên ta có thể nghĩ ngay đến dùng một mảng chữ nhật khá lớn để biểu diễn các thế hệ của một cộng đồng tế bào. Mảng sẽ có giá trị là 1 tại các vị trí của các tế bào sống và 0 ứng với các tế bào chết. Để xác định được sự thay đổi của các tế bào từ thế hệ này sang thế hệ khác, ta chỉ cần đếm số tế bào sống lân cận mỗi tế bào và áp dụng các luật từ 1) đến 4). Cảnh huống của thế hệ tiếp theo được ghi vào một mảng mới. Ta dễ dàng viết được lược đồ chương trình thực hiện phương án trên.

```

program LifeGame;
const
    N = ... ;
    M = ... ;
type
    row = 1..N; {hàng của mảng}
    col = 1..M; {cột của mảng}
    status = (alive, dead) ;
{mỗi tế bào ở một trong hai trạng thái (status) : sống (alive) và chết (dead)}
    Table = array [row, col] of status;
var
    T, newT : Table;
    i : row;
    j : col;
    again : boolean;
{Sau đây mô tả các thủ tục và hàm}

procedure Initiation (var T : Table);
{đưa vào bảng ban đầu}
function Count (i, j : integer) : 0..8;
{đếm số tế bào sống lân cận tế bào ở vị trí (i, j)}
procedure COPY (newT : Table; var T : Table);
{sao chép mảng newT sang mảng T}
begin
    Initiation (T);
    WriteTable (T);
    repeat
        for i := 1 to N do
            for j := 1 to M do
                case count (i, j) of

```

```

    0,1 : newT [i, j] := dead;
    2 : newT [i, j] := T [i, j];
    3 : newT [i, j] := alive;
    4, 5, 6, 7, 8 : newT[i, j] := dead;
end;
    Copy (newT, T);
    WriteTable (T);
    ReadIn (again);
until not again;
end.

```

Bạn đọc dễ dàng viết được các thủ tục và hàm trong chương trình trên. Riêng đối với hàm count, ta cần lưu ý rằng, để tính được số tế bào sống lân cận một tế bào ta chỉ cần xét 8 tế bào lân cận với tế bào đã cho, nếu nó ở giữa bảng. Còn nếu nó ở rìa bảng thì số tế bào lân cận nó sẽ không phải thế.

Bây giờ ta hãy xét những ưu tiên và hạn chế của chương trình trên. Ưu điểm của chương trình trên là đơn giản và dễ hiểu. Song nó có nhiều nhược điểm. Trước hết ta đã sử dụng mảng để mô tả cảnh huống của các thế hệ tế bào tức là ta đã đóng khung chỉ xét sự phát triển của cộng đồng tế bào trong phạm vi mảng. Trên thực tế từ cảnh huống ban đầu, một cộng đồng có thể sẽ phát triển vượt quá giới hạn của mảng, chỉ sau một số thế hệ. Ta cũng đã xét lưới tế bào như một bảng chữ nhật đầy (mỗi thành phần của bảng đều chứa thông tin), song thực ra, ta có một bảng thừa thót, vì không cần xét tới các tế bào chết mà các tế bào lân cận nó đều chết. Một nhược điểm khác nữa là để xác định các thế hệ tế bào, ta đã tính lại số tế bào sống lân cận của mọi tế bào. Điều này làm lãng phí nhiều thời gian, vì không phải tất cả, mà chỉ có một số tế bào có số tế bào sống lân cận thay đổi. Để khắc phục nhược điểm này, bên cạnh mảng T ta đưa vào một mảng khác LiveNeighbors để ghi lại số các tế bào sống lân cận mỗi tế bào. Tức là

**var** LiveNeighbors : **array** [row, col] **of** 0..8;

Khi đó, từ thế hệ này sang thế hệ khác, ta chỉ cần xác định lại các giá trị của mảng LiveNeighbors tại các tế bào kề với các tế bào từ sống thành chết hoặc từ chết thành sống. Như vậy thay cho hàm Count, ta sử dụng LiveNeighbors. Bạn đọc hãy viết chương trình thực hiện đề án này. Chương trình này vẫn chưa phải là chương trình tốt, vì ta còn phải đi qua toàn bộ mảng LiveNeighbors để phát hiện ra các tế bào sẽ sinh hoặc sẽ chết đi ở thế hệ sau. Còn có nhiều cách để cải tiến chương trình (bài tập)

Như trên đã nói, việc sử dụng mảng để mô tả các thế hệ tế bào là không thích hợp, nó không phản ánh đầy đủ sự phát triển của các cộng đồng tế bào. Cần phải xét bảng các tế bào là một bảng vô hạn, trong đó tương ứng với mỗi vị trí  $(i, j)$  là một tế bào được hoàn toàn xác định bởi vị trí của nó và số các tế bào sống lân cận nó. Do đó phương pháp tốt nhất là dùng các cấu trúc dữ liệu bảng băm mở để biểu diễn bảng các tế bào. Ở đây cần chú ý rằng, hàm băm h sẽ 'băm' các vị trí  $(i; j)$  của các tế bào vào các vị trí  $0, 1, \dots, N-1$  của bảng băm.

$$h : \{(i; j) \mid (i; j) \text{ vị trí của tế bào}\} \rightarrow \{0, 1, \dots, N-1\}$$

Mỗi tế bào sẽ được đưa vào danh sách liên kết các tế bào thuộc một 'rổ' nào đó của bảng băm. Do đó mỗi tế bào được mô tả bởi cấu trúc bản ghi sau

```

type cell = record
    row, col : integer;
    state : (alive, dead);
    count : 0..8;
    next : ^ cell;
end;
Table = array [0..N-1] of ^cell;

```

Khi muốn xác định các tế bào sẽ sống hoặc sẽ chết ở thế hệ sau, để tránh phải xem xét toàn bộ bảng băm, ta đưa vào hai danh sách. Danh sách Change ghi lại các tế bào mà số tế bào sống lân cận ở thế hệ hiện tại có thay đổi so với ở thế hệ trước; do đó, danh sách Change sẽ chứa các tế bào đang sống có thể sẽ trở thành chết hoặc đang chết có thể sẽ

trở thành sống ở thế hệ sau. Danh sách NextChange sẽ ghi lại các tế bào mà số tế bào sống lân cận ở thế hệ sau có thay đổi so với thế hệ hiện tại:

Chúng ta có thể đưa thêm vào mỗi bản ghi biểu diễn tế bào hai con trỏ. Một con trỏ để liên kết các tế bào thuộc danh sách Change và một con trỏ để liên kết các tế bào thuộc danh sách NextChange. Như vậy mỗi bản ghi tế bào sẽ có 3 trường con trỏ. Điều này sẽ tiêu tốn nhiều không gian nhớ, vì có thể chỉ có một số ít tế bào được đưa vào hai danh sách này. Do đó cách tốt hơn là ta sẽ đưa các con trỏ trỏ đến các tế bào thuộc danh sách Change (NextChange) vào danh sách Change (NextChange) thay cho việc đưa chính các tế bào thuộc danh sách Change(NextChange) vào danh sách Change (NextChange). Nói một cách khác, danh sách Change và NextChange sẽ là các danh sách liên kết, gồm các bản ghi có cấu trúc sau

```
type node = record
    entry : ^cell;
    next : ^node;
end;
```

Mỗi khi cần xác định cảnh huống của một cộng đồng tế bào ở một thế hệ nào đó, ta đi qua danh sách Change. Khi gặp tế bào từ trạng thái chết trở thành sống (hoặc từ trạng thái sống trở thành chết), ta chỉ cần thay đổi trường state cho nó nhận giá trị alive (hoặc dead) đồng thời ta sẽ tìm trong bảng băm những tế bào lân cận của các tế bào đó và đưa chúng vào danh sách NextChange. Nếu một tế bào là lân cận của tế bào từ chết trở thành sống (hoặc từ sống trở thành chết) thì trường count được tăng lên 1 (hoặc được giảm đi 1). Quá trình trên được thực hiện bởi thủ tục Traverse (qua đi).

Chúng ta có phác thảo chương trình sau :

```
program LifeGame;
const
    N = ... ; {N là số thành phần của bảng băm}
```

```

type
    ptrcell = ^cell;
        {con trỏ liên kết các tế bào trong các danh sách của bảng băm}
    cell = record
        row, col : integer; {vị trí của tế bào}
        state : (alive, dead); {trạng thái của tế bào}
        count : 0..8; {số lần cận sống của tế bào}
        next : ptrcell;
    end;
    ptrnode = ^node;
        {con trỏ liên kết trong danh sách Change và NextChange}
    node = record
        entry : ptrcell;
        next : ptrnode;
    end;
    Table = array [0..N-1] of ptrcell;
var
    T : Table;
    Change, NextChange : ptrnode;
    again : boolean;

procedure Initiation (var T : Table; var Change : ptrnode);
    {thủ tục này xây dựng bảng băm T và danh sách Change (ban đầu Change là danh sách các tế bào đang chết sẽ thành sống và đang sống sẽ thành chết) ứng với cảnh huống ban đầu}.

procedure Traverse (var Change, NextChange : ptrnode);
    {đi qua danh sách Change, cập nhật bảng T và xây dựng bảng NextChange}.

procedure WriteTable (var T : Table);
    {viết ra bảng T}

begin {chương trình chính}

```

```

Initiation (T, Change);
repeat
    Traverse (Change, NextChange);
    WriteTable;
    Change := NextChange;
    readIn (again);
until    not again;
end.

```

Sau đây chúng ta sẽ mô tả chi tiết thủ tục Traverse, còn các thủ tục Initiation và WriteTable được để lại xem như bài tập.

Trong thủ tục Traverse, với mỗi lân cận ( $i, j$ ) của một tế bào chết trở thành sống, (hoặc sống trở thành chết), ta phải tìm xem trong bảng băm đã có tế bào ở vị trí ( $i, j$ ) chưa, nếu chưa thì đưa nó vào bảng băm (với trường state là dead và trường count bằng 1), ta sẽ dùng biến con trỏ  $q$  ghi lại địa chỉ của tế bào ở vị trí ( $i, j$ ). Quá trình trên được thực hiện bằng thủ tục

**GetTable (i, j : integer; var q : ptrcell)**

Chúng ta sẽ sử dụng thủ tục

**Add (q : ptrcell; var NextChange : ptrnode)**

để đưa  $q$  vào danh sách NextChange.

Chúng ta có thể mô tả thủ tục Traverse như sau

```

procedure    Traverse;
var      p, p1 : ptrnode;
        i, j : integer;
        q : ptrcell;
procedure    GetTable (i, j : integer; var q : ptrcell);
procedure    Add (q : ptrcell; var NextChange : ptrnode);
begin {bắt đầu thủ tục Traverse}
    p := Change;

```

```

NextChange := nil;
while p < > nil do
    begin
        with p^ do
            with entry^ do
                begin
                    if (state = dead) and (count = 3) then
                        begin
                            state := alive;
                            for i := row - 1 to row + 1 do
                                for j := col - 1 to col + 1 do
                                    if (i < > row) and (j < > col) then
                                        begin
                                            GetTable (i, j, q);
                                            q^.count := q^.count + 1;
                                            Add (q, NextChange)
                                        end;
                                    end
                                end
                            else if (state = alive) and ((count <= 1) or (count >= 4)) then
                                begin
                                    state := dead;
                                    for i := row - 1 to row + 1 do
                                        for j := col - 1 to col + 1 do
                                            if (i < > row) and (j < > col) then
                                                begin
                                                    Get Table (i, j, q);
                                                    q^.count := q^.count - 1;
                                                    Add (q, NextChange);
                                                end;
                                    end
                                end; {hết lệnh with}

```

```
p1 := p;  
p := p^.next;  
dispose(p1);  
end; {hết vòng lặp while}  
end; {hết thủ tục Traverse}
```

Trong chương trình LifeGame, ta đã dùng danh sách Change ghi lại các tế bào mà số tế bào sống lân cận chúng ở thế hệ hiện tại có thay đổi so với ở thế hệ trước. Chúng ta cần phải đi qua danh sách Change để tìm các tế bào đang chết sẽ trở thành sống và đang sống sẽ trở thành chết ở thế hệ sau. Do đó vẫn còn lãng phí nhiều thời gian. Nay thay cho dùng danh sách Change, nếu chúng ta sử dụng hai danh sách BecomeLive ghi lại các tế bào chết có khả năng thành sống và BecomeDead ghi lại các tế bào sống có khả năng thành chết thì ta sẽ thu hẹp được phạm vi các tế bào cần xem xét. Chúng tôi để lại cho bạn đọc tiếp tục phát triển và viết chương trình thực hiện đề án này.

## *Chương 7*

# **CÁC CẤU TRÚC DỮ LIỆU Ở BỘ NHỚ NGOÀI**

Chương này giành để trình bày mô hình tổ chức dữ liệu ở bộ nhớ ngoài, các cấu trúc dữ liệu để lưu giữ và tìm kiếm thông tin ở bộ nhớ ngoài : file băm, file có chỉ số, B- cây. Với mỗi phương pháp tổ chức file, chúng ta sẽ trình bày các thuật toán để thực hiện các phép toán tìm kiếm, xen vào, loại bỏ và sửa đổi trên file.

### **7.1. MÔ HÌNH TỔ CHỨC DỮ LIỆU Ở BỘ NHỚ NGOÀI**

Các cấu trúc dữ liệu (CTDL) mà chúng ta xét từ đầu tới nay đều là các CTDL được lưu giữ trong bộ nhớ chính. Nhưng trong nhiều áp dụng, số các dữ liệu cần được lưu giữ vượt quá khả năng của bộ nhớ chính. Các máy tính hiện nay đều được trang bị các thiết bị bộ nhớ ngoài, thông thường là đĩa. Nó có khả năng lưu giữ một khối lượng rất lớn các dữ liệu. Tuy nhiên các thiết bị nhớ ngoài có những đặc trưng truy cập hoàn toàn khác bộ nhớ chính. Sau đây chúng ta sẽ trình bày mô hình tổng quát mà các hệ điều hành hiện đại sử dụng để quản lý dữ liệu ở bộ nhớ ngoài. Trong các mục sau chúng ta sẽ xét các CTDL để lưu giữ file sao cho các phép toán trên file được thực hiện một cách hiệu quả. Đó là file băm, file có chỉ số, B - cây.

Các hệ điều hành hiện đại đều cho chúng ta khả năng tổ chức dữ liệu ở bộ nhớ ngoài dưới dạng các file.

Chúng ta có thể quan niệm file như là một tập hợp nào đó các dữ liệu (các bản ghi) được lưu giữ ở bộ nhớ ngoài. Các bản ghi trong file có thể có độ dài cố định (số các trường của bản ghi là cố định) hoặc có thể có độ dài thay đổi. Các file với các bản ghi có độ dài cố định được sử dụng

nhiều trong các hệ quản trị cơ sở dữ liệu. Các file với các bản ghi có độ dài thay đổi hay được sử dụng để lưu giữ các thông tin văn bản. Chúng ta sẽ chỉ xét các file với các bản ghi có độ dài cố định. Các kỹ thuật mà chúng ta sẽ trình bày để lưu giữ và thao tác với các file này có thể sửa đổi để áp dụng cho các file với các bản ghi có độ dài thay đổi.

Trong chương này chúng ta sẽ hiểu khoá của bản ghi là một tập hợp nào đó các trường của bản ghi hoàn toàn xác định bản ghi, tức là hai bản ghi khác nhau phải có các giá trị khác nhau trên ít nhất một trường thuộc khoá.

Trên file chúng ta cần thực hiện các phép toán sau đây

1. *Tìm kiếm* : tìm trong file các bản ghi với các giá trị cho trước trên một nhóm nào đó các trường của bản ghi.
2. *Xen vào* : xen vào file một bản ghi nào đó
3. *Loại bỏ* : loại bỏ khỏi file tất cả các bản ghi với các giá trị cho trước trên một nhóm nào đó các trường của bản ghi.
4. *Sửa đổi* : sửa tất cả các bản ghi với các giá trị cho trước trên một nhóm nào đó các trường bằng cách đặt lại giá trị trên các trường được chỉ định bởi các giá trị mới đã cho.

*Ví dụ.* Giả sử chúng ta có file với các bản ghi chứa các trường (tên sản phẩm, nơi sản xuất, giá). Ta có thể cần tìm tất cả các bản ghi với tên sản phẩm = bóng đèn 60W; thêm vào file bản ghi (quạt bàn, nhà máy điện cơ, 69.000); loại bỏ tất cả các bản ghi với nơi sản xuất = nhà máy X; sửa tất cả các bản ghi với nơi sản xuất = nhà máy Z bằng cách thay giá cũ bởi giá mới.

Hệ điều hành chia bộ nhớ ngoài thành các khối vật lý (physical block) có cỡ như nhau, ta gọi tắt là các khối. Cỡ của các khối thay đổi tùy theo hệ điều hành, thông thường là từ  $2^9$  byte đến  $2^{12}$  byte. Mỗi khối có địa chỉ, đó là địa chỉ tuyệt đối của khối ở trên đĩa, tức là byte đầu tiên của khối.

File được lưu giữ trong một số khối, mỗi khối có thể lưu giữ một số bản ghi của file. Trong một khối có thể còn một số byte chưa được sử dụng đến. Mỗi bản ghi có địa chỉ, địa chỉ của bản ghi là cặp (k, s), trong đó k là địa chỉ của khối chứa bản ghi, còn s là số byte trong khối đứng

trước byte bắt đầu bản ghi (s được gọi là offset). sau này khi nói đến con trỏ tới khối (tới bản ghi) thì ta hiểu đó là địa chỉ khối (bản ghi).

File có thể được lưu giữ trong một danh sách liên kết các khối. Điều hình hơn, file có thể được lưu giữ trong các khối tổ chức dưới dạng cây. Các khối không là lá của cây chứa các con trỏ trỏ tới một số khối trong cây.

Trong mỗi khối có thể giành ra một số byte (phần này được gọi là đầu khối) để chứa các thông tin cần thiết về khối, chẳng hạn để ghi số bản ghi trong khối.

Trong một khối, không gian để lưu trữ một bản ghi được gọi là khối con. Cần phân biệt khối con đầy và rỗng. Khối con đầy là khối con có chứa bản ghi, ngược lại là khối con rỗng. Để chỉ một khối con là đầy hoặc rỗng, trong đầu khối ta giành cho mỗi khối con một bit (gọi là bit đầy), bit nhận giá trị 1 (0) nếu khối con tương ứng là đầy (rỗng). Một cách khác, trong mỗi khối con ta giành ra một bit (bit xoá), bit nhận giá trị 1 có nghĩa là bản ghi đã bị xoá.

### ***Đánh giá thời gian thực hiện các phép toán trên file***

Các phép toán trên file (tìm kiếm, xen vào, loại bỏ, sửa đổi) được thực hiện thông qua phép toán cơ bản, đọc một khối dữ liệu ở bộ nhớ ngoài vào vùng đệm trong bộ nhớ chính hoặc viết các dữ liệu ở vùng đệm trong bộ nhớ chính vào một khối ở bộ nhớ ngoài. Ta gọi phép toán này là phép toán truy cập khối (block access). Cần chú ý rằng, việc chuyển một khối dữ liệu ở bộ nhớ ngoài vào bộ nhớ chính đòi hỏi nhiều thời gian hơn việc tìm kiếm dữ liệu trong khối khi nó đã ở trong bộ nhớ chính. Cũng cần biết rằng, các dữ liệu cần phải có ở bộ nhớ chính trước khi nó được sử dụng bằng cách nào đó. Vì vậy khi đánh giá thời gian thực hiện một thuật toán thao tác với các dữ liệu được lưu giữ trong file, chúng ta phải tính số lần cần thiết phải thực hiện phép toán truy cập khối. Số lần thực hiện phép toán truy cập khối được dùng để biểu diễn tính hiệu quả của các thuật toán trên các file.

### ***Tổ chức file đơn giản***

Phương pháp đơn giản nhất, đồng thời cũng kém hiệu quả nhất để lưu giữ các bản ghi của file là, xếp các bản ghi của file vào một số khối

cần thiết theo một trật tự tuỳ ý. Các khối có thể liên kết với nhau bởi các con trỏ tạo thành một danh sách liên kết các khối. Một cách khác, ta cũng có thể sử dụng một bảng để lưu giữ địa chỉ của các khối.

Phép toán tìm kiếm các bản ghi theo các giá trị đã biết trên một số trường được thực hiện bằng cách đọc lần lượt các bản ghi trong các khối. Việc xen vào file một bản ghi mới được thực hiện bằng cách xen nó vào khối cuối cùng của file nếu trong đó còn chỗ, nếu không thì thêm vào file một khối mới và đặt bản ghi cần xen vào đó. Muốn loại bỏ các bản ghi, trước hết ta cần định vị được các bản ghi cần loại bỏ, sau đó ta sẽ tiến hành xoá bỏ. Việc xoá bỏ một bản ghi có thể thực hiện bằng nhiều cách. Chẳng hạn có thể đặt lại giá trị của bit xoá trong bản ghi. Trong trường hợp này việc sử dụng lại không gian của bản ghi này để lưu giữ bản ghi mới cần phải thận trọng. Nếu trong hệ cơ sở dữ liệu có sử dụng con trỏ trỏ tới bản ghi (trường hợp này, bản ghi được xem là bị đóng chặt), thì ta không được sử dụng không gian của nó để lưu giữ bản ghi mới này.

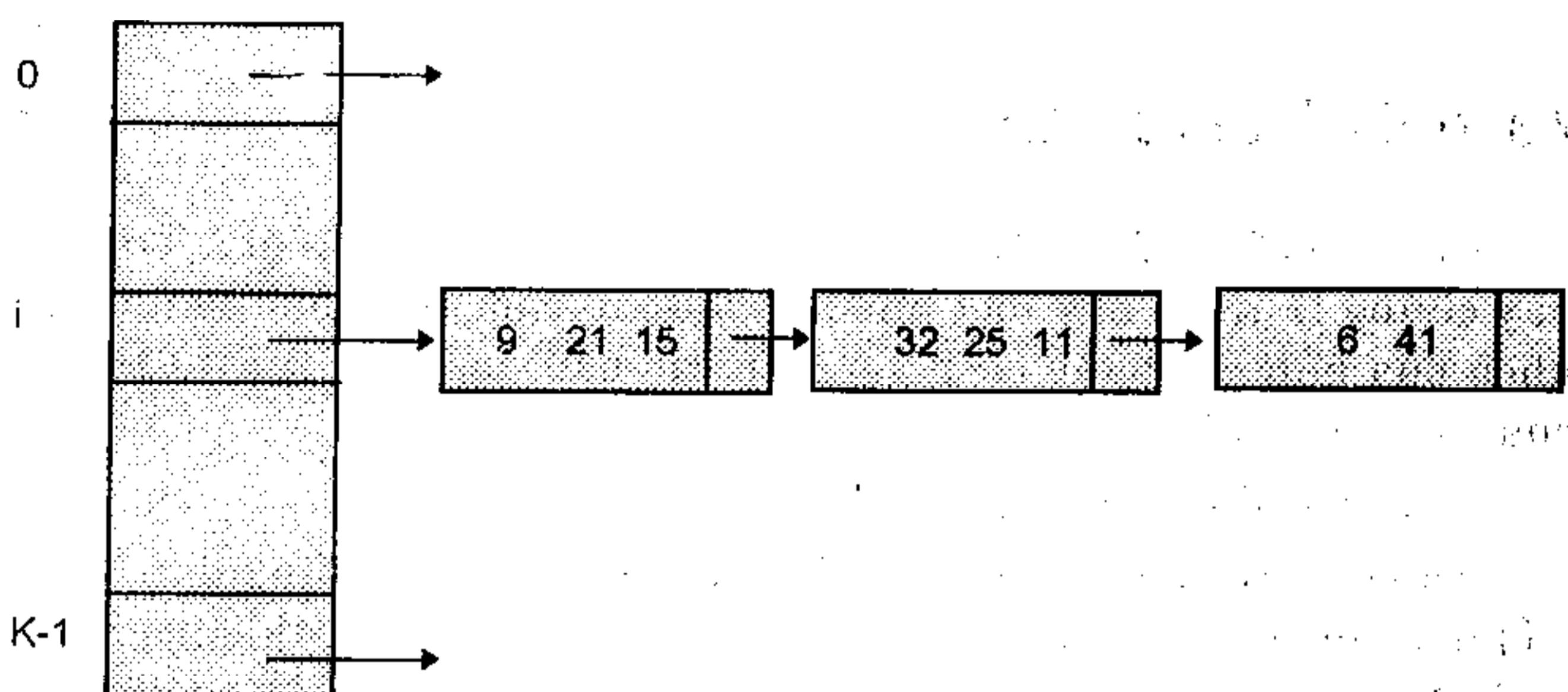
Với cách tổ chức file tuân tự như trên, các phép toán trên file sẽ chậm, vì chúng đòi hỏi phải xem xét toàn bộ các bản ghi trong file. Trong các mục sau này chúng ta sẽ trình bày các phương pháp tổ chức file ưu việt hơn, cho phép ta mỗi lần cần truy cập đến một bản ghi, chỉ cần đọc vào bộ nhớ chính một phần nhỏ của file.

Chúng ta không thể viết bằng Pascal hoặc bằng một ngôn ngữ bậc cao khác các thủ tục có đề cập đến các dữ liệu ở mức khối vật lý và các địa chỉ khối. Do đó trong các phương pháp tổ chức file được trình bày sau đây, ta sẽ mô tả một cách không hình thức các thuật toán thực hiện các phép toán trên các file.

## 7.2. FILE BĂM

Cấu trúc của file băm hoàn toàn tương tự như cấu trúc bảng băm mở ở bộ nhớ trong đã được chúng ta đề cập đến trong chương 5. Tư tưởng của tổ chức file băm là như sau : ta chia tập hợp các bản ghi của file thành K lớp. Với mỗi lớp, tạo ra một danh sách liên kết các khối, các khối này chứa các bản ghi của lớp. Ta sử dụng một bảng gồm K con trỏ (bảng chỉ dẫn), mỗi con trỏ trỏ tới khối đầu tiên trong danh sách liên kết các khối của một lớp.

Hình 7.1 biểu diễn cấu trúc của một file băm.



Hình 7.1. Cấu trúc file băm

Việc phân phối các bản ghi của file vào các lớp được thực hiện bởi hàm băm  $h$ . Đó là hàm xác định trên tập các giá trị khoá của các bản ghi và nhận các giá trị nguyên từ 0 đến  $K-1$ . Nếu  $x$  là một giá trị khoá và  $h(x) = i$ ,  $0 \leq i \leq K-1$ , thì bản ghi với khoá  $x$  thuộc lớp thứ  $i$ .

Để tìm kiếm bản ghi với khoá  $x$  cho trước, đầu tiên ta tính  $h(x)$ , con trỏ chứa ở thành phần thứ  $i = h(x)$  trong bảng chỉ dẫn ta tìm đến các khối của lớp  $i$ . lần lượt đọc các khối, ta sẽ tìm ra bản ghi với khoá  $x$ , hoặc đọc hết các khối mà không thấy có nghĩa là bản ghi không có ở trong file.

Muốn xen vào file bản ghi với khoá  $x$ , ta cần kiểm tra xem nó có ở trong file hay chưa. Nếu chưa ta có thể xen nó vào khối đầu tiên trong danh sách các khối của  $h(x)$ , nếu tại đó còn đủ chỗ cho bản ghi. Nếu tất cả các khối của lớp  $h(x)$  đều đầy, ta thêm vào danh sách các khối của lớp  $h(x)$  một khối mới và đặt bản ghi vào đó.

Để loại bỏ bản ghi với khoá  $x$ , trước hết ta cần xác định vị trí của bản ghi trong file bằng cách áp dụng thủ tục tìm kiếm. Sau đó có thể xoá bỏ bản ghi này bằng cách, chẳng hạn cho bit xoá nhận giá trị 1.

Cấu trúc file băm là cấu trúc rất có hiệu quả nếu các phép toán trên file chỉ đòi hỏi đến việc truy cập các bản ghi theo khoá. Giả sử file có  $n$  bản ghi, nếu hàm băm được thiết kế tốt, thì trung bình mỗi lớp chứa  $n/k$  bản ghi. Giả sử mỗi khối chứa được  $m$  bản ghi. Như vậy mỗi lớp gồm

khoảng  $n/mk$  khối. Tức là các phép toán trên file băm sẽ k lần nhanh hơn so với tổ chức file tuần tự.

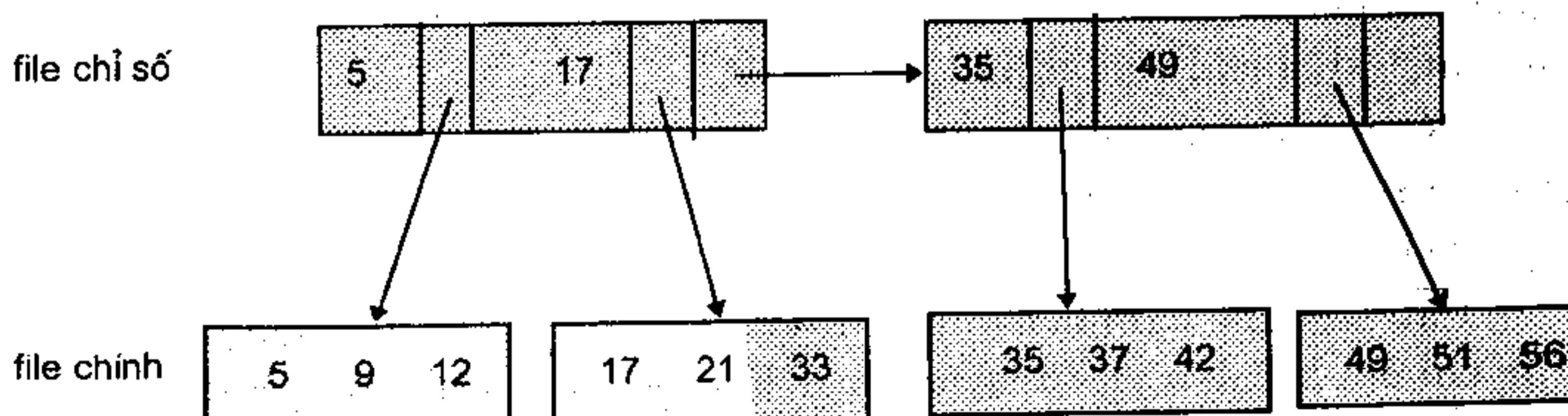
### 7.3. FILE CÓ CHỈ SỐ (INDEXED FILE)

Cấu trúc file băm được tạo ra dựa trên khoá của bản ghi. Trong mục này chúng ta trình bày một phương pháp tổ chức file khác cũng dựa vào khoá của bản ghi bằng cách sắp xếp các bản ghi theo thứ tự tăng dần của các giá trị khoá.

Cấu trúc file có chỉ số được hình thành như sau

Ta sắp xếp các bản ghi của file theo thứ tự khoá tăng dần vào một số khối cần thiết. Ta có thể sắp xếp các bản ghi vào một khối cho tới khi khối đầy. Song thông thường, trong mỗi khối người ta để giành lại một không gian cho các bản ghi được thêm vào file sau này. Lý do là để phép toán xen vào file được thực hiện dễ dàng hơn. Ta sẽ gọi file gồm các bản ghi chứa trong các khối này là file chính, để phân biệt với file chỉ số được tạo ra sau đây.

Chỉ số của một khối là cặp ( $v, b$ ), trong đó  $b$  là địa chỉ của khối, còn  $v$  là giá trị khoá nhỏ nhất của các bản ghi trong khối  $b$ . Từ các khối của file chính, ta sẽ tạo ra file chỉ số (index file), file này gồm các chỉ số khối của file chính. Các chỉ số khối được sắp xếp theo thứ tự tăng dần của khoá vào một số khối cần thiết. Các khối này có thể được mọc nối với nhau tạo thành một danh sách liên kết. Trong trường hợp này file chỉ số gồm một danh sách liên kết các khối, các khối chứa các chỉ số khối của file chính. Một cách khác ta cũng có thể sử dụng một bảng để lưu giữ địa chỉ của các khối trong file chỉ số. Hình 7.2 minh họa cấu trúc của file có chỉ số.



Hình 7.2. Cấu trúc file có chỉ số

Sau đây chúng ta sẽ xét sự thực hiện các phép toán trên file được tổ chức dưới dạng file có chỉ số

### **Tìm kiếm**

Giả sử ta cần tìm bản ghi  $x$  với khoá  $v$  cho trước. Trước hết ta cần tìm trên file chỉ số một chỉ số  $(v_1, b_1)$  sao cho  $v_1$  là giá trị khoá lớn nhất trong file chỉ số thoả mãn điều kiện  $v_1 \leq v$ . Ta sẽ nói  $v_1$  phủ  $v$ .

Việc tìm kiếm trên file chỉ số một giá trị khoá  $v_1$  phủ giá trị khoá  $v$  cho trước có thể thực hiện bằng cách tìm kiếm tuần tự hoặc tìm kiếm nhị phân.

Trong tìm kiếm tuần tự, ta cần xem xét tất cả các bản ghi của file chỉ số cho tới khi tìm thấy một chỉ số  $(v_1, b_1)$  với  $v_1$  phủ  $v$ . Nếu  $v$  nhỏ hơn giá trị khoá của bản ghi đầu tiên trong file chỉ số thì điều đó có nghĩa là trong file chỉ số không chứa giá trị khoá phủ  $v$ .

Phương pháp có hiệu quả hơn là tìm kiếm nhị phân. Giả sử các bản ghi của file chỉ số được sắp xếp vào các khối được đánh số từ 1 đến  $m$ . Xét khối thứ  $\lceil m/2 \rceil$ . Giả sử  $(v_2, b_2)$  là bản ghi đầu tiên trong khối. So sánh giá trị khoá cho trước  $v$  với giá trị khoá  $v_2$ . Nếu  $v < v_2$  ta tiến hành tìm kiếm trên các khối 1, 2, ...,  $\lceil m/2 \rceil - 1$ . Còn nếu  $v \geq v_2$ , ta tiến hành tìm kiếm trên các khối  $\lceil m/2 \rceil, \lceil m/2 \rceil + 1, \dots, m$ . Quá trình trên được lặp lại cho tới khi ta chỉ cần tìm kiếm trên một khối. Lúc này ta chỉ việc lần lượt so sánh giá trị khoá  $v$  cho trước với giá trị khoá chứa trong khối này.

Để tìm ra bản ghi  $x$  với khoá  $v$  cho trước, trước hết ta tìm trên file chỉ số một chỉ số  $(v_1, b_1)$  với  $v_1$  phủ  $v$ . Sau đó lần lượt xét các bản ghi trong khối có địa chỉ  $b_1$  để phát hiện ra bản ghi có khoá  $v$ , hoặc không nếu trong khối không có bản ghi nào với khoá là  $v$ . Nếu trên file chỉ số không chứa giá trị khoá  $v_1$  phủ  $v$ , thì file chính không chứa bản ghi có khoá  $v$ .

### **Xem vào**

Giả sử ta cần thêm vào file bản ghi  $r$  với giá trị khoá  $v$ .

Giả sử file chính được chứa trong các khối  $B_1, B_2, \dots, B_k$  và các giá trị khoá của các bản ghi trong khối  $B_i$  nhỏ hơn các giá trị khoá trong khối  $B_{i+1}$ .

Trước hết ta cần tìm ra khối  $B_i$  cần phải xếp bản ghi  $r$  vào đó. Muốn vậy ta áp dụng thủ tục tìm kiếm trên file chỉ số để tìm ra chỉ số  $(v_1, b_1)$  với  $v_1$  phủ  $v$ . Nếu tìm thấy thì  $B_i$  là khối có địa chỉ  $b_1$ , ngược lại  $B_i$  là khối  $B_{i+1}$ .

Trong trường hợp  $B_i$  chưa đầy và  $r$  còn chưa có ở trong khối  $B_i$ , thì ta xếp bản ghi  $r$  vào đúng vị trí của nó, tức là phải đảm bảo trật tự tăng dần theo khoá.

Nếu  $B_i$  là  $B_1$  thì sau khi thêm vào bản ghi  $r$ , nó trở thành bản ghi đầu tiên trong khối  $B_1$ , do đó ta cần phải tiến hành sửa đổi chỉ số của khối  $B_1$  trong file chỉ số.

Trong trường hợp  $B_i$  đã đầy, ta tiến hành xếp bản ghi  $r$  vào đúng vị trí của nó trong  $B_i$ , khi đó còn thừa ra một bản ghi. Tìm đến khối  $B_{i+1}$  (ta biết được địa chỉ của khối  $B_{i+1}$  bằng cách tìm trong chỉ số). Nếu  $B_{i+1}$  chưa đầy thì ta xếp bản ghi thừa ra của  $B_i$  vào vị trí đầu tiên trong khối  $B_{i+1}$ , đồng thời sửa lại chỉ số của  $B_{i+1}$  trong file chỉ số. Nếu khối cũng đầy hoặc không tồn tại khi  $B_i$  là  $B_k$  thì ta thêm vào file chính một khối mới và xếp bản ghi  $r$  vào khối mới này. Chỉ số của khối mới thêm vào cần phải được xen vào file chỉ số.

### *Loại bỏ*

Để loại bỏ bản ghi  $r$  với khoá  $v$ , ta cần áp dụng thủ tục tìm kiếm để định vị bản ghi trong file. Sau đó sẽ tiến hành xoá bỏ  $r$  bằng nhiều cách khác nhau, chẳng hạn có thể đặt lại giá trị của bit đầy/rỗng tương ứng với bản ghi cần xoá ở đầu khối.

### *Sửa đổi*

Giả sử ta cần sửa đổi bản ghi với khoá  $v$ . Nếu các giá trị cần sửa không là giá trị của các trường thuộc khoá, thì ta chỉ cần áp dụng thủ tục tìm kiếm để tìm ra bản ghi và tiến hành các sửa đổi cần thiết. Nếu các giá trị cần sửa thuộc khoá thì việc sửa đổi được thực hiện bằng cách loại bỏ bản ghi cũ, xen vào bản ghi mới.

## 7.4. B-CÂY

Mục đích của chúng ta là nghiên cứu các cấu trúc dữ liệu biểu diễn file sao cho các phép toán trên file được thực hiện hiệu quả, tức là với số lần thực hiện phép toán truy cập khôi ít nhất có thể được, khi cần phải tìm kiếm, xen vào, loại bỏ hoặc sửa đổi các bản ghi trên file. B - cây là một cấu trúc dữ liệu đặc biệt thích hợp để biểu diễn file. Trong mục này chúng ta sẽ trình bày B - cây và các kỹ thuật để thực hiện các phép toán tìm kiếm, xen vào và loại bỏ trên B - cây.

### **Cây tìm kiếm đa nhánh ( Multiway Search Trees)**

Cây tìm kiếm m nhánh là sự tổng quát hóa của cây tìm kiếm nhị phân, trong đó mỗi đỉnh của cây có nhiều nhất m con. Các đỉnh của cây được gắn với các giá trị khoá của các bản ghi. Nếu đỉnh a có r con ( $r \leq m$ ) thì nó chứa đúng  $r - 1$  khoá ( $k_1, k_2, \dots, k_{r-1}$ ), trong đó  $k_1 < k_2 < \dots < k_{r-1}$  (Chúng ta giả thiết rằng các giá trị khoá được sắp xếp thứ tự tuyến tính). Tổng quát hóa tính chất về khoá gắn với các đỉnh của cây tìm kiếm nhị phân, cây tìm kiếm m nhánh phải thoả mãn tính chất sau đây. Nếu đỉnh a có r con và chứa các khoá ( $k_1, k_2, \dots, k_{r-1}$ ) thì các khoá chứa trong các đỉnh của cây con thứ nhất của đỉnh a nhỏ hơn  $k_1$ , còn các khoá chứa trong các đỉnh của cây con thứ i ( $i = 2, \dots, r-1$ ) phải lớn hơn hoặc bằng  $k_{i-1}$  và nhỏ hơn  $k_i$ , các khoá chứa trong các đỉnh của cây con thứ r phải lớn hơn hoặc bằng  $k_{r-1}$ . Mỗi lá của cây chứa một số khoá, tối đa là s.

Các phép toán tìm kiếm, xen vào và loại bỏ trên cây tìm kiếm m nhánh được thực hiện bằng các kỹ thuật tương tự như đối với cây tìm kiếm nhị phân.

### **B - cây (B - Trees)**

B - cây là một loại đặc biệt của cây tìm kiếm m nhánh cân bằng (xem lại khái niệm cây cân bằng trong mục 7, chương 4). Cụ thể, B - cây được định nghĩa như sau

B - cây cấp m là cây tìm kiếm m nhánh thoả mãn các tính chất sau đây

1. Nếu cây không phải là cây chỉ gồm có gốc thì gốc có ít nhất hai con và nhiều nhất m con.
2. Mỗi đỉnh trong của cây, trừ gốc, có ít nhất  $\lceil m/2 \rceil$  con và nhiều nhất m con.
3. Tất cả các lá của cây trên cùng một mức. (Nói cách khác, tất cả các đường đi từ gốc tới lá cây có cùng độ dài).

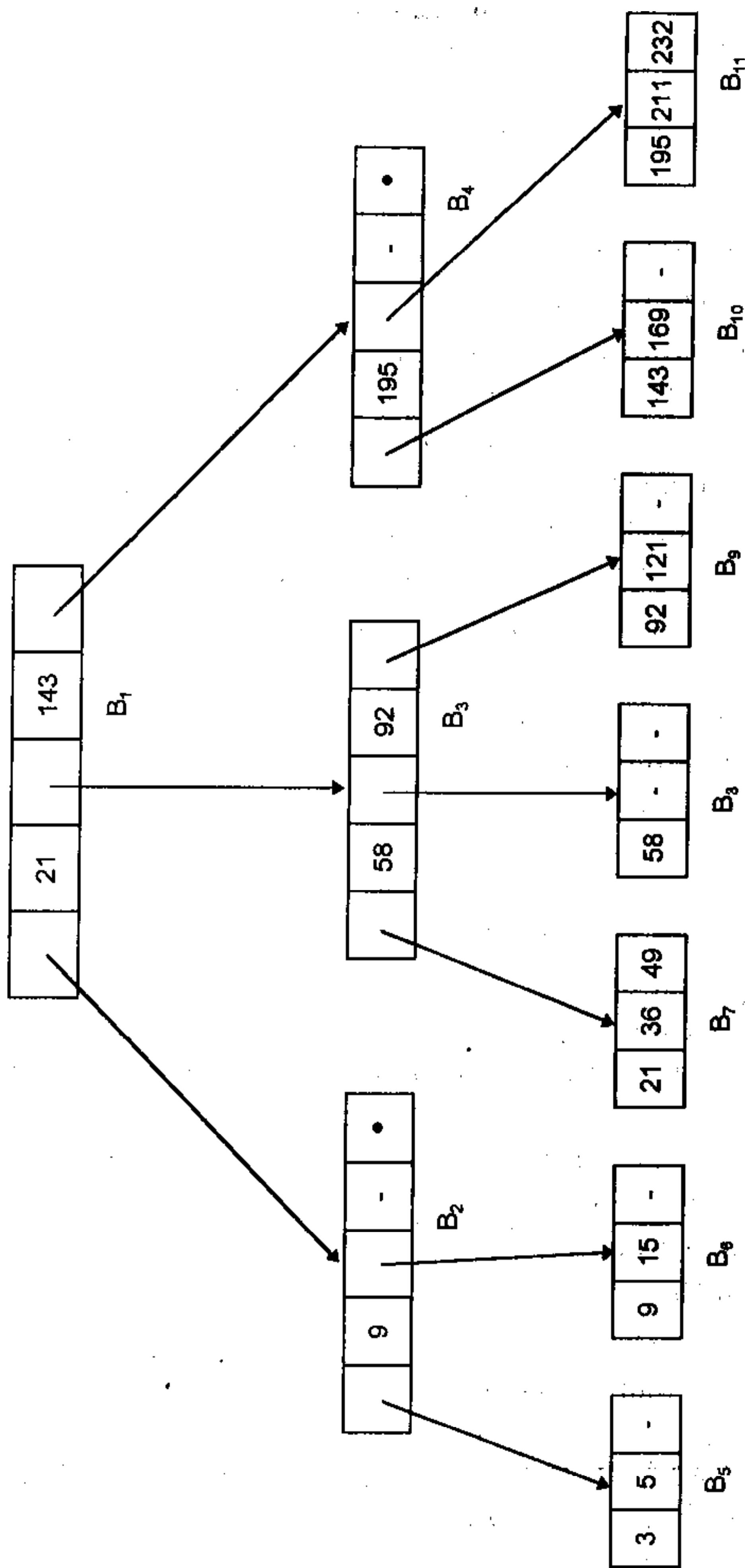
Tư tưởng của việc tổ chức file dưới dạng B - cây là như sau. Ta sắp xếp các bản ghi của file (file chính) vào một số khối cần thiết. Mỗi khối này sẽ là lá của B - cây. Trong mỗi khối các bản ghi được sắp xếp theo thứ tự tăng dần của khoá. Các chỉ số của các khối này (các lá) lại được sắp xếp vào một số khối mới. Trong mỗi khối này, các chỉ số được sắp xếp theo thứ tự tăng dần của khoá. Trong B - cây, các khối này sẽ là các đỉnh ở mức trên của mức các lá. Ta lại lấy chỉ số của các khối vừa tạo ra sắp xếp vào một số khối mới. Các khối này lại là các đỉnh ở mức trên của mức từ đó chúng được tạo ra. Quá trình trên sẽ tiếp tục cho tới khi các chỉ số có thể xếp gọn vào một khối. Khối này là đỉnh của B - cây.

Như vậy, mỗi đỉnh của B - cây là một khối. Mỗi đỉnh trong của B - cây có dạng

$$(p_0, v_1, p_1, v_2, p_2, \dots, v_n, p_n)$$

trong đó  $v_1 < v_2 < \dots < v_n$  và  $(v_i, p_i), 0 \leq i \leq n$ , là chỉ số của một khối, tức là  $v_i$  là giá trị khoá nhỏ nhất trong một khối, còn  $p_i$  là con trỏ trỏ tới khối chứa khoá nhỏ nhất  $v_i$ , tức là con trỏ trỏ tới đỉnh con thứ  $i$  của đỉnh trong đang nói tới. Cần lưu ý rằng, giá trị của khoá  $v_0$  không được lưu giữ ở mỗi đỉnh trong, lý do là để tiết kiệm bộ nhớ.

**Ví dụ.** Hình 7.3 biểu diễn một B - cây cấp 3. B - cây được tạo thành từ 11 khối được đánh số  $B_1, B_2, \dots, B_{11}$ . Mỗi khối là đỉnh trong chứa được 3 chỉ số. Mỗi khối là lá chứa được 3 bảng ghi (3 số nguyên). File ở đây là file các số nguyên được lưu giữ ở các khối từ  $B_5$  đến  $B_{11}$ .



Hình 7.3. B - cây

Sau đây chúng ta sẽ nghiên cứu sự thực hiện các phép toán tìm kiếm, xen vào và loại bỏ trên B - cây.

### **Tìm kiếm**

Giả sử chúng ta cần tìm bản ghi r có khoá v cho trước. Chúng ta cần phải tìm đường đi từ gốc của B - cây tới lá, sao cho lá này cần phải chứa bản ghi r nếu nó có ở trong file.

Trong quá trình tìm kiếm, giả sử tại một thời điểm nào đó ta đạt tới đỉnh B. Nếu khôi B là lá, ta tìm trong khôi B xem nó có chứa bản ghi r hay không. Nhớ lại rằng các bản ghi của file được xếp vào các khôi theo thứ tự tăng dần của khoá, do đó sự tìm kiếm trong khôi B có thể tiến hành bằng kỹ thuật tìm kiếm tuần tự hoặc tìm kiếm nhị phân.

Nếu B là một đỉnh trong chứa ( $p_0, v_1, p_1, \dots, v_n, p_n$ ) thì ta cần xác định vị trí của giá trị khoá v trong dãy giá trị khoá  $v_1, v_2, \dots, v_n$ . Nếu  $v < v_1$  thì ta đi xuống đỉnh được trỏ bởi  $p_0$ . Nếu  $v_i \leq v < v_{i+1}$  thì ta đi xuống đỉnh được trỏ bởi  $p_i$  ( $i = 1, 2, \dots, n - 1$ ). Còn nếu  $v_n < v$  thì đi xuống đỉnh được trỏ bởi  $p_n$ .

### **Xen vào**

Giả sử ta cần phải xen vào B - cây một bản ghi r với khoá là v. Đầu tiên ta áp dụng thủ tục tìm kiếm để tìm ra khôi B cần phải xen bản ghi r vào đó.

Nếu khôi B còn đủ chỗ cho bản ghi r thì ta xếp bản ghi r vào khôi B sao cho thứ tự tăng dần của khoá được bảo tồn. Chú ý rằng, r không thể là bản ghi đầu tiên của khôi B, trừ khi B là lá ngoài cùng bên trái. Nếu B là lá ngoài cùng bên trái thì giá trị khoá nhỏ nhất trong khôi B không có mặt trong các đỉnh là tiền thân của đỉnh B. Vì vậy trong trường hợp này chỉ cần thêm bản ghi r vào khôi B là xong, không cần sửa đổi gì với các đỉnh là tiền thân của khôi B.

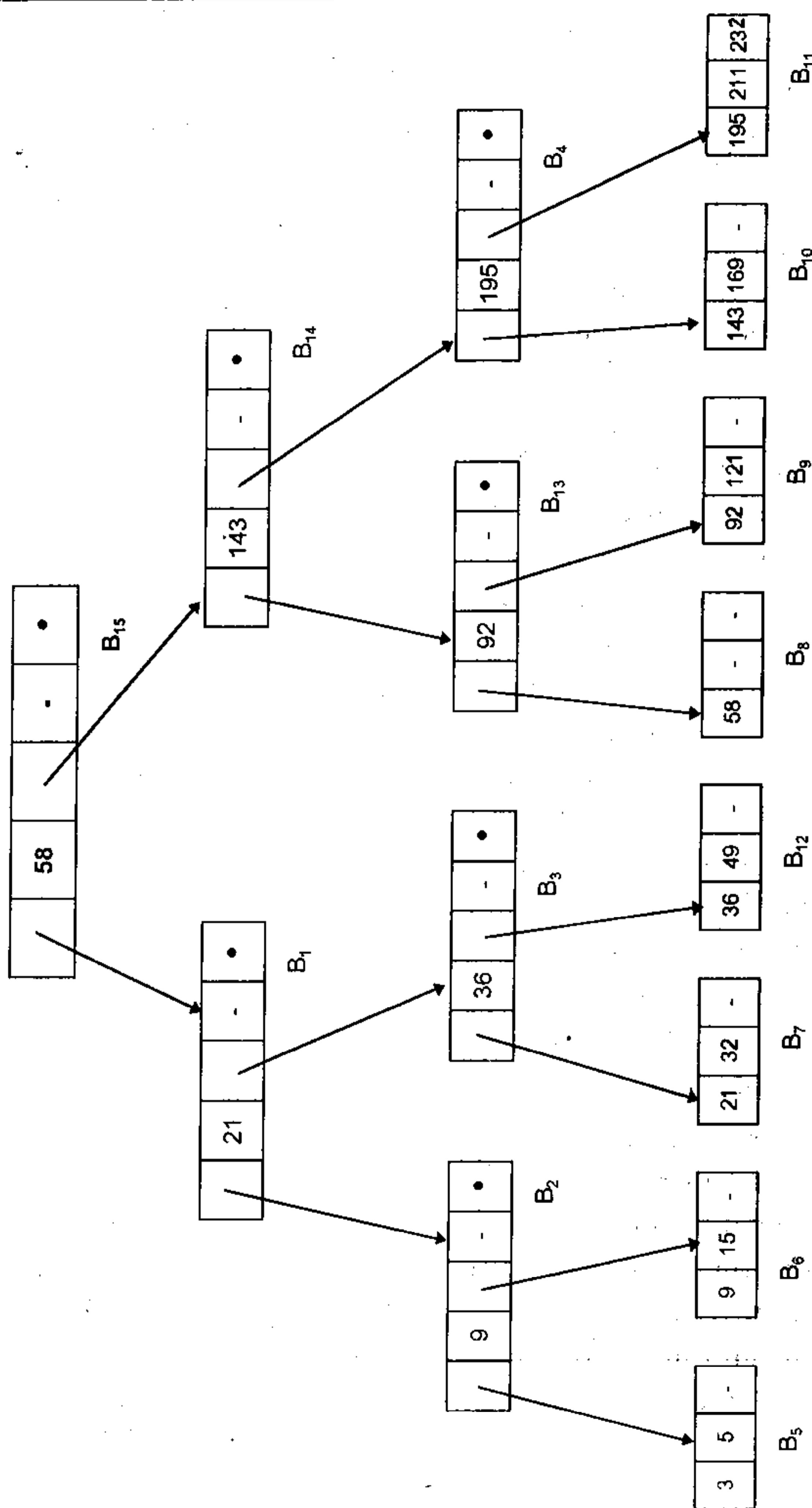
Nếu khôi B không còn đủ không gian để lưu giữ bản ghi r thì ta thêm vào B - cây một lá mới, khôi B'. Chuyển một nửa số bản ghi ở cuối của khôi B sang khôi B'. Sau đó xếp bản ghi r vào khôi B hoặc khôi B' sao cho vẫn đảm bảo được tính tăng dần của các giá trị khoá. Giả sử Q là cha của B, ta có thể biết được Q nếu trong quá trình tìm kiếm, ta lưu lại vết của đường đi từ gốc tới B. Giả sử chỉ số của khôi B' là ( $v', p'$ ),

trong đó  $v'$  là giá trị khoá nhỏ nhất trong  $B'$ , còn  $p'$  là địa chỉ của khối  $B'$ . Áp dụng thủ tục trên để xen  $(v', p')$  vào khối  $Q$ . Nếu khối  $Q$  không còn đủ chỗ cho  $(v', p')$  thì ta lại phải thêm vào  $B$  - cây một đỉnh mới  $Q'$ , nó là em liền kề của  $Q$ . Sau đó lại phải tìm đến cha của đỉnh  $Q$  để đưa vào chỉ số của khối mới  $Q'$ . Quá trình có thể tiếp diễn và dẫn đến việc phải phân đôi số giá trị khoá ở gốc, nửa sau được chuyển vào khối mới. Trong trường hợp này, ta phải tạo ra một gốc mới có đúng hai con, một con là gốc cũ, một con là đỉnh mới đưa vào.

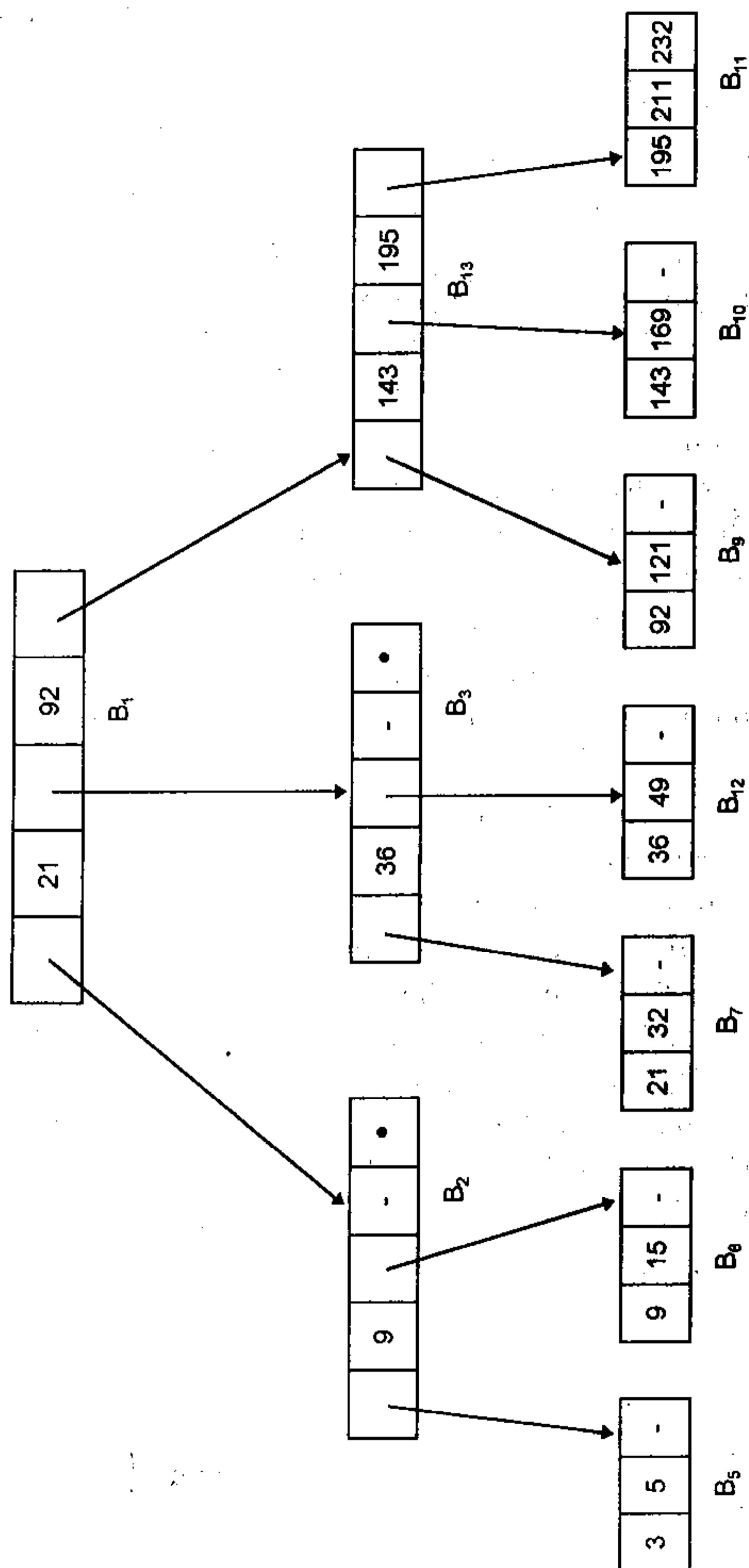
**Ví dụ.** Giả sử ta cần xen vào  $B$  - cây trong hình 7.3 bản ghi có khoá 32. Trước hết ta phải tìm khối cần phải đưa bản ghi này vào. Bắt đầu từ gốc  $B_1$ , vì  $21 < 32 < 143$ , ta đi xuống  $B_3$ . Tại  $B_3$ ,  $32 < 58$ , ta đi xuống  $B_7$ .  $B_7$  là lá, vậy cần phải đưa bản ghi với khoá 32 vào  $B_7$ . Nhưng khối  $B_7$  đã đầy. Ta thêm vào khối mới  $B_{12}$  và xếp các bản ghi với khoá 21, 32 vào khối  $B_7$ , xếp các bản ghi với khoá 36, 49 vào khối  $B_{12}$ . Chỉ số của khối  $B_{12}$  chưa giá trị khoá 36. Cần phải xếp chỉ số  $B_{12}$  vào cha của  $B_7$  là  $B_3$ . Nhưng  $B_3$  cũng đầy. Thêm vào khối mới  $B_{13}$ . Sau đó các chỉ số của các khối  $B_7$ ,  $B_{12}$  được xếp vào  $B_3$  còn các chỉ số của các khối  $B_8$ ,  $B_9$  được xếp vào  $B_{13}$ . Nay chỉ số của khối  $B_{13}$  là 58 và địa chỉ của khối  $B_{13}$  cần được xếp vào khối  $B_1$ . Nhưng  $B_1$  cũng đầy thêm vào  $B$  - cây khối mới  $B_{14}$ . Xếp các chỉ số của  $B_2$ ,  $B_3$  vào  $B_1$ , các chỉ số của  $B_{13}$ ,  $B_4$  vào  $B_{14}$ . Vì  $B_1$  là gốc, ta phải thêm vào gốc mới, khối  $B_{15}$  và xếp các chỉ số của  $B_1$  và  $B_{14}$  vào  $B_{15}$ . Kết quả là ta có  $B$  - cây trong hình 7.4.

Giả sử ta cần loại khỏi  $B$  - cây bản ghi  $r$  với khoá  $v$ . Đầu tiên áp dụng thủ tục tìm kiếm để tìm ra lá  $B$  chứa bản ghi  $r$ . Sau đó loại bỏ bản ghi  $r$  trong khối  $B$ .

Giả sử sau khi loại bỏ  $B$  không rỗng. Trong trường hợp này, nếu  $r$  không phải là bản ghi đầu tiên trong  $B$ , ta không phải làm gì thêm. Nếu  $r$  là bản ghi đầu tiên trong  $B$ , thì sau khi xoá  $r$ , chỉ số của  $B$  đã thay đổi. Do đó ta cần tìm đến đỉnh  $Q$  là cha của  $B$ . Nếu  $B$  là con trưởng của  $Q$  thì giá trị khoá  $v'$  trong chỉ số  $(v', p')$  của  $B$  không có trong  $Q$ . Trong trường hợp này ta cần tìm đến tiền thân  $A$  của  $B$  sao cho  $A$  không phải là con trưởng của cha mình  $A'$ . Khi đó giá trị khoá nhỏ nhất trong  $B$  được chứa trong  $A'$ . Do đó trong  $A'$ , ta cần thay giá trị khoá cũ  $v$  bởi giá trị mới  $v'$ .



Hình 7.4. B - cây sau khi thêm vào B- cây trong hình 7.3 bัน ghi với giá trị khóa 32



Hình 7.5. B-tree nhận được từ B-tree trong hình 7.4 sau khi xóa bản ghi có khóa 58

### *Loại bỏ*

Giả sử sau khi loại bỏ bản ghi  $r$ ,  $B$  trở thành rỗng. Loại bỏ lá  $B$  khỏi  $B$ -cây. Điều đó dẫn đến cần loại bỏ chỉ số của  $B$  trong đỉnh cha  $Q$  của  $B$ .

Nếu sau khi loại bỏ, số các con của đỉnh  $Q$  ít hơn  $\lceil m/2 \rceil$  thì ta tìm đến đỉnh  $Q'$  là anh em liền kề của đỉnh  $Q$ . Nếu  $Q'$  có nhiều hơn  $\lceil m/2 \rceil$  con thì ta phân phối lại các giá trị khoá trong  $Q$  và  $Q'$  sao cho cả hai có ít nhất  $\lceil m/2 \rceil$  con. Khi đó các chỉ số của  $Q$  hoặc  $Q'$  có thể thay đổi. Ta lại phải tìm đến các tiền thân của  $Q$  để phản ánh sự thay đổi này.

Nếu  $Q'$  có đúng  $\lceil m/2 \rceil$  con, thì ta kết hợp hai đỉnh  $Q$  và  $Q'$  thành một đỉnh, một trong hai đỉnh bị loại khỏi cây, các khoá chứa trong đỉnh này được chuyển sang đỉnh còn lại. Điều này dẫn đến cần loại bỏ chỉ số của đỉnh bị loại ra khỏi cha của  $Q$ . Sự loại bỏ này được thực hiện bằng cách áp dụng thủ tục loại bỏ đã trình bày.

Quá trình loại bỏ có thể dẫn đến việc loại bỏ gốc cây, khi chúng ta cần kết hợp hai con của gốc thành một đỉnh, đỉnh này trở thành gốc mới của  $B$ -cây.

**Ví dụ.** Giả sử chúng ta cần loại bỏ ghi với khoá 58 khỏi  $B$ -cây trong hình 7.4. Đầu tiên tìm lá chứa khoá 58, đó là khôi  $B_8$ . Xoá bản ghi 58, khôi  $B_8$  thành rỗng. Tìm đến cha của  $B_8$  là  $B_{13}$ . Loại bỏ chỉ số của  $B_8$  khỏi  $B_{13}$ ,  $B_{13}$  chỉ còn một con. Số con của  $B_{13}$  ít hơn  $\lceil m/2 \rceil$  (ở đây  $\lceil m/2 \rceil = \lceil 3/2 \rceil = 2$ ). Tìm đến em liền kề của  $B_{13}$  là  $B_4$ , số con của  $B_4$  là hai. Kết hợp hai đỉnh này thành một đỉnh  $B_{13}$ . Cần phải loại bỏ chỉ số của khôi  $B_4$  khỏi  $B_{14}$ .  $B_{14}$  trở thành chỉ có một con. Tìm đến anh liền kề của  $B_{14}$  là  $B_1$ . Số con của  $B_1$  là hai. Kết hợp  $B_1$  và  $B_{14}$  thành một đỉnh  $B_1$ .  $B_1$  trở thành gốc mới của  $B$ -cây. Hình 7.5 minh họa  $B$ -cây nhận được từ  $B$ -cây trong hình 7.4 sau khi loại đỉnh có khoá 58.

## *Chương 8*

# CÁC CHIẾN LƯỢC THIẾT KẾ THUẬT TOÁN

Mặc dù không tồn tại một phương pháp vạn năng có thể giúp ta thiết kế được thuật toán giải quyết mọi vấn đề, nhưng các nhà nghiên cứu đã tìm ra một số phương pháp thiết kế thuật toán cơ bản, các phương pháp này còn được gọi là các chiến lược thiết kế thuật toán. Mỗi phương pháp này có thể áp dụng để giải quyết một phạm vi khá rộng các bài toán. Trong chương này chúng ta sẽ trình bày các phương pháp thiết kế thuật toán: Chia - để - trị (divide - and - conquer), phương pháp tham ăn (greedy method), quay lui (backtracking), quy hoạch động (dynamic programming), nhánh và cận (branch - and - bound). Trong mỗi chiến lược, chúng ta sẽ trình bày phương pháp chung, sau đó đưa ra một số ví dụ minh họa.

Cần lưu ý rằng, các phương pháp thiết kế thuật toán mà chúng ta sẽ trình bày chỉ là các chiến lược có tính định hướng sự tìm tòi thuật toán. Việc áp dụng một số chiến lược nào đó để tìm ra thuật toán cho một bài toán còn đòi hỏi nhiều sáng tạo.

### **8.1. CHIA - ĐỂ - TRỊ**

#### **8.1.1. Phương pháp**

Có lẽ phương pháp thiết kế thuật toán được áp dụng rộng rãi nhất là chiến lược chia - để - trị. Tư tưởng của chiến lược này là như sau: Người ta phân bài toán cần giải thành các bài toán con. Các bài toán con lại được tiếp tục phân thành các bài toán con nhỏ hơn, cứ thế tiếp tục cho tới khi ta nhận được các bài toán con hoặc là đã có thuật giải, hoặc là có thể dễ dàng đưa ra thuật giải. Sau đó ta tìm cách kết hợp các

nghiệm của các bài toán con để nhận được nghiệm của bài toán con lớn hơn, để cuối cùng nhận được nghiệm của bài toán cần giải. Thông thường các bài toán con nhận được trong quá trình phân chia là cùng dạng với bài toán ban đầu, chỉ có cỡ của chúng là nhỏ hơn. Trong các trường hợp như thế, thuật toán tìm được có thể biểu diễn một cách tự nhiên bởi thủ tục đệ quy.

Sau đây là lược đồ phương pháp chia - để - trị.

```

procedure DivideConquer (A,x);
    {Tim nghiệm x của bài toán A}

    begin
        if A đủ nhỏ then solve (A)
        else begin
            Phân A thành các bài toán con  $A_1, A_2, \dots, A_m$ ;
            for i := 1 to m do DivideConquer ( $A_i, x_i$ );
            Kết hợp các nghiệm  $x_i$  ( $i = 1, 2, \dots, m$ ) của các bài toán con
             $A_i$  để nhận được nghiệm x của bài toán A
        end
    end;

```

Trong thủ tục trên, *Solve* là thuật giải bài toán A trong trường hợp A có cỡ đủ nhỏ.

Thuật toán tìm kiếm nhị phân mà chúng ta đã biết trong mục 3.3 là thuật toán được thiết kế dựa trên chiến lược chia - để - trị. Cho mảng  $A[1 .. n]$  được sắp xếp theo thứ tự tăng dần,  $A[1] \leq A[2] \leq \dots \leq A[n]$ . Với x cho trước, ta cần xác định xem x có chứa trong mảng  $A[1 .. n]$  hay không, tức là có hay không chỉ số  $1 \leq i \leq n$ , sao cho  $A[i] = x$ . Phương pháp chia - để - trị gợi ý ta chia mảng  $A[1 .. n]$  thành 3 mảng con  $A[1 .. k - 1]$ , mảng con gồm một phần duy nhất  $A[k]$  và mảng con  $A[k + 1 .. n]$ , ( $k$  là chỉ số nằm giữa 1 và  $n$ ). Với mảng con chỉ gồm một phần ta biết ngay được nó có chứa x hay không. Nếu  $x = A[k]$  thì mảng A chứa x và  $i = k$ . Nếu  $x < A[k]$  thì ta chỉ cần tìm kiếm trên mảng con  $A[1 .. k - 1]$ , còn nếu  $x > A[k]$  ta chỉ cần tìm kiếm trên mảng con

$A[k + 1 .. n]$ . Để tìm kiếm x trên mảng con  $A[1 .. k - 1]$  hoặc  $A[k + 1 .. n]$  ta lại áp dụng cách phân chia như đã làm với mảng  $A[1 .. n]$ .

Thuật toán sắp xếp nhanh (QuickSort) sẽ được trình bày ở chương 9 cũng được thiết kế bởi kỹ thuật chia - để - trị. Sau đây chúng ta sẽ đưa ra một số ví dụ minh họa cho kỹ thuật chia - để - trị.

### 8.1.2. Tìm max và min

Cho mảng  $A[1 .. n]$ , chúng ta cần tìm thành phần nhỏ nhất và lớn nhất của mảng này. Đây là bài toán rất đơn giản có thể giải bằng các thuật toán khác nhau, trong đó có thuật giải bằng kỹ thuật chia - để - trị.

Một thuật toán rất tự nhiên là, đầu tiên ta lấy max, min là thành phần đầu tiên  $A[1]$  của mảng. Sau đó so sánh max, min với các thành phần  $A[i]$ ,  $2 \leq i \leq n$ , và cập nhật max, min một cách thích ứng. Thuật toán được mô tả bởi thủ tục sau

```

procedure Simaxmin ( $A[i .. n]$ , max, min);
begin
    max :=  $A[1]$ ;
    min :=  $A[1]$ ;
    for  $i := 2$  to  $n$  do
        if max <  $A[i]$  then max :=  $A[i]$ 
        else if min >  $A[i]$  then min :=  $A[i]$ 
    end;

```

Thời gian thực hiện thuật toán này được xác định bởi số phép toán so sánh. Từ vòng lặp **for**, ta thấy số phép toán so sánh cần thực hiện trong trường hợp xấu nhất (trường hợp mảng  $A[1 .. n]$  được sắp theo thứ tự giảm dần) là  $2(n - 1)$ .

Áp dụng kỹ thuật chia - để - trị, ta chia mảng  $A[1 .. n]$  thành 2 mảng con  $A[1 .. k]$  và  $A[k + 1 .. n]$  ( $k = \lfloor n/2 \rfloor$ ). Nếu tìm được max và min của các mảng con  $A[1 .. k]$  và  $A[k + 1 .. n]$ , ta sẽ dễ dàng xác định được max và min trên toàn mảng  $A[1 .. n]$ . Để tìm được max và min trên các mảng con  $A[1 .. k]$  và  $A[k + 1 .. n]$  ta lại tiếp tục chia đôi chúng. Quá

trình sẽ dừng lại khi ta nhận được các mảng con chỉ có một hoặc hai phần tử. Trong các trường hợp đơn giản này max và min được xác định dễ dàng. Từ phương pháp đã trình bày trên, ta có thể đưa ra thủ tục đệ quy **MaxMin(i, j, fmax, fmin)**. Thủ tục này cần tìm max và min trên mảng A[i .. j],  $1 \leq i \leq j \leq n$ . Để tìm max và min trên mảng A[i .. n], ta chỉ cần gọi thủ tục này với  $i = 1, j = n$ .

```

procedure MaxMin (i, j, fmax , fmin);
    {fmax, fmin ghi lại phần tử lớn nhất, nhỏ nhất của mảng A[i .. j] }

begin
    if i = j then begin fmax := A[i];
        fmin := A[i];
        end
    else if j = i + 1 then
        if A[i] < A[j] then begin fmax := A[j];
            fmin := A[i];
        end
        else begin fmax := A[i];
            fmin := A[j];
        end
    else begin mid := i + j div 2;
        MaxMin (i, mid,gmax, gmin );
        MaxMin (mid + 1, j, hmax, hmin );
        if gmax < hmax then fmax := hmax
        else fmax := gmax;
        if gmin < hmin then fmin := gmin
        else fmin := hmin;
    end
end;

```

Bây giờ ta đánh giá thời gian thực hiện thuật toán này trên mảng n phần tử A[1 .. n]. Gọi T(n) là số phép toán so sánh cần thực hiện. Từ thủ tục trên, ta xác định được quan hệ đệ quy sau đây:

$$T(n) = \begin{cases} 0 & \text{nếu } n = 1 \\ 1 & \text{nếu } n = 2 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 & \text{nếu } n > 2 \end{cases}$$

Giả sử  $n = 2^k$ , với k là số nguyên dương nào đó. Bằng phương pháp thế, ta tính được T(n) như sau:

$$\begin{aligned} T(n) &= T(2^k) = 2T(2^{k-1}) + 2 \\ &= 2^2 T(2^{k-2}) + 2^2 + 2 \\ &= 2^3 T(2^{k-3}) + 2^3 + 2^2 + 2 \\ &\quad \dots \dots \\ &= 2^{k-1} T(2) + \sum_{i=1}^{k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 \\ &= n/2 + n - 2 \\ &= 3n/2 - 2 \end{aligned}$$

Như vậy với  $n = 2^k$ , thuật toán **MaxMin** cần  $3n/2 - 2$  phép so sánh, so với thuật toán trước, nó tiết kiệm được khoảng 25% phép so sánh. Tuy nhiên **MaxMin** là thuật toán đệ quy, nó tiêu tốn nhiều bộ nhớ hơn thuật toán trước.

### 8.1.3. Trao đổi hai phần của một mảng

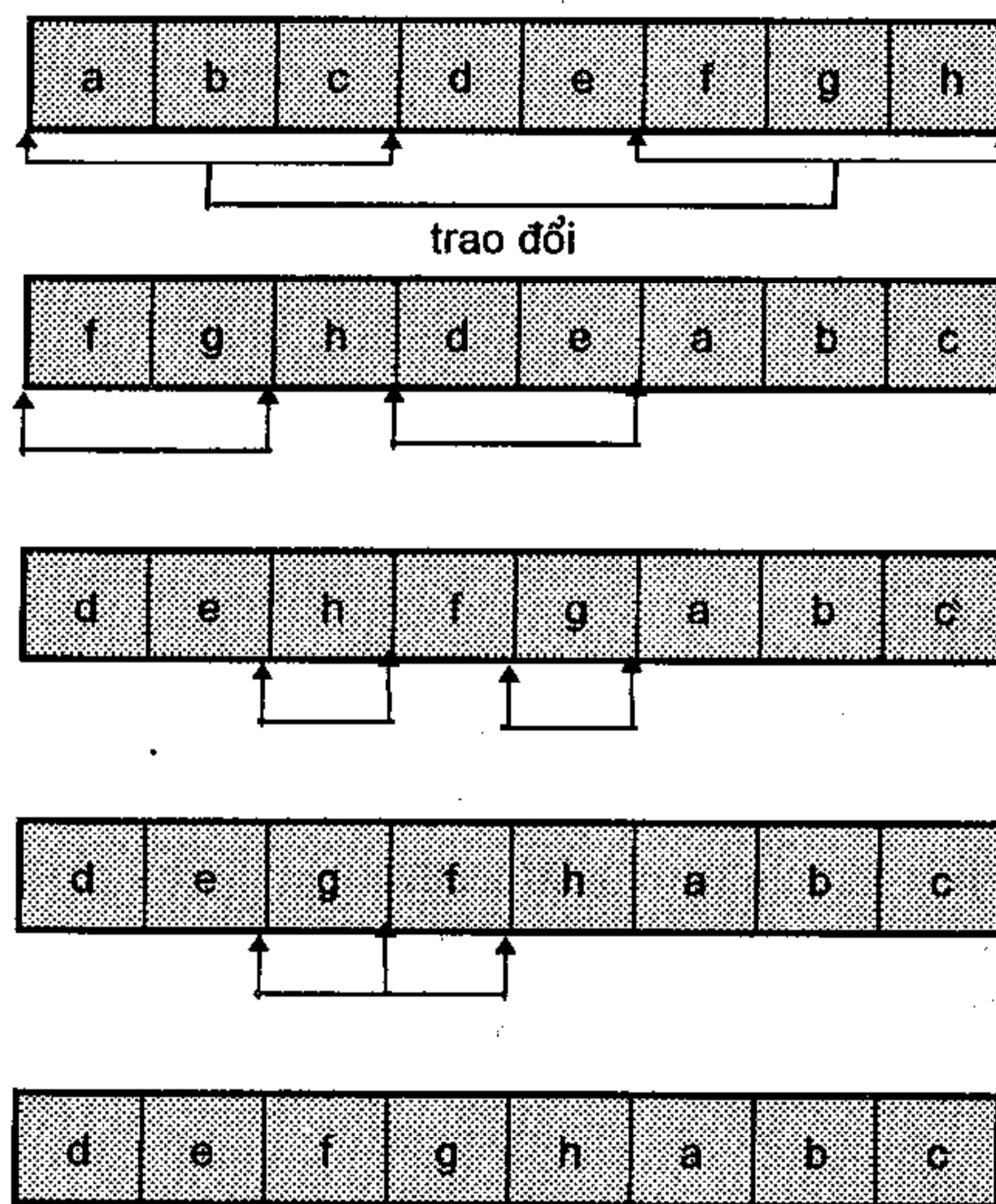
Giả sử T[1 .. n] là một mảng. Chúng ta muốn trao đổi phần đầu (k phần tử đầu tiên) của mảng với phần còn lại ( $n - k$  phần tử còn lại), nhưng không sử dụng mảng phụ. Chẳng hạn, T là mảng

a	b	c	d	e	f	g	h
---	---	---	---	---	---	---	---

và  $k = 3$ . Sau khi trao đổi ta cần nhận được mảng



Sử dụng kỹ thuật chia - để - trị, ta phân bài toán trên thành hai bài toán con như sau: Giả sử  $k \leq n - k$ , đầu tiên ta trao đổi  $k$  phần tử của phần đầu với  $k$  phần tử cuối của phần còn lại. Sau đó trong mảng  $T[1 .. n - k]$ , ta chỉ cần trao đổi  $k$  phần tử đầu với phần còn lại. Chẳng hạn với mảng  $T$  ở trên và  $k = 3$ , quá trình sẽ diễn ra như sau:



Còn nếu  $k > n - k$ , thì ta trao đổi  $n - k$  phần tử đầu tiên với  $n - k$  phần tử của phần sau. Sau đó trong mảng  $T[n - k + 1 .. n]$ , ta trao đổi  $n - k$  phần tử cuối mảng với các phần tử của phần đầu.

Như vậy, ta đã phân bài toán trao đổi hai phần của mảng thành hai bài toán con. Bài toán con thứ nhất là trao đổi hai mảng con có độ dài bằng nhau. Bài toán con thứ hai cùng dạng với bài toán đã cho, nhưng

cõ của mảng đã nhỏ đi. Bài toán con thứ nhất có thể giải được dễ dàng bằng cách trao đổi từng cặp phần tử tương ứng. Thủ tục sau đây sẽ thực hiện trao đổi hai mảng con có độ dài m bắt đầu từ i và j tương ứng.

```
procedure Interchange (i,j,m) ;
begin
    for p := 0 to m - 1 do Swap ( T[i + p], T[j + p] )
end;
```

Trong thủ tục trên, *Swap* (*x,y*) là thủ tục trao đổi giá trị của hai biến *x* và *y*.

Bài toán con thứ hai cũng dạng với bài toán ban đầu với cõ của mảng nhỏ đi. Do đó, ta dễ dàng đưa ra thuật toán đệ quy để trao đổi hai phần của một mảng. Tuy nhiên, quá trình gọi đệ quy sẽ dừng lại khi ta đạt tới việc trao đổi hai phần có độ dài bằng nhau của một mảng. Do đó, ta có thể đưa ra thuật toán không đệ quy sau đây.

```
Procedure Transpose (k);
{Thủ tục trao đổi k phần tử đầu của mảng A[1 .. n] với n - k phần tử còn lại}
begin
    i := 1;
    j := n;
    while k >= i do
        if k < (i + j) div 2 then
            begin
                Interchange (i, i + j - k, k - i + 1);
                j := i + j - k - 1;
            end
        else begin
            Interchange (i, k + 1, j - k);
            i := i + j - k;
        end;
    end;
```

## 8.2. PHƯƠNG PHÁP THAM ĂN

### 8.2.1. Phương pháp

Phương pháp tham ăn (greedy method) là một chiến lược thiết kế thuật toán thường được sử dụng để giải quyết các bài toán tối ưu.

Nhiều vấn đề cần giải quyết có thể quy về vấn đề sau đây: Cho trước một tập A các đối tượng nào đó, đòi hỏi phải chọn ra một tập con S các đối tượng thoả mãn một số điều kiện nào đó. Bất kỳ một tập con S nào của A thoả mãn các yêu cầu đã đặt ra được gọi là *nghiệm chấp nhận được* của bài toán. Một hàm mục tiêu gắn mỗi nghiệm chấp nhận được với một giá trị nào đó. Một nghiệm chấp nhận được mà tại đó hàm mục tiêu có giá trị lớn nhất (hoặc nhỏ nhất) được gọi là *nghiệm tối ưu*.

Tư tưởng của phương pháp ăn tham là như sau: Ta xây dựng tập S dần dần từng bước, bắt đầu từ tập rỗng. Tại mỗi bước, ta sẽ chọn một phần tử “tốt nhất” trong các phần tử còn lại của A để đưa vào S. Việc lựa chọn một phần tử như thế ở mỗi bước được hướng dẫn bởi hàm chọn. Phần tử được chọn sẽ bị loại khỏi tập A. Nếu khi thêm phần tử được chọn vào tập S mà S vẫn còn thoả mãn các điều kiện của bài toán thì ta mở rộng S bằng cách thêm vào phần tử được chọn.

```

procedure Greedy (A,S);
    {A là tập các ứng cử viên, S là nghiệm}
begin
    S ← Ø;
    while A < > Ø do
        begin
            x ← select (A);
            A ← A - {x};
            if S ∪ {x} chấp nhận được then S ← S ∪ {x}
        end;
    end;

```

Trong lược đồ tổng quát trên, **Select** là hàm chọn, nó cho phép ta chọn ra từ tập A một phần tử được xem là tốt nhất, nhiều hứa hẹn nhất là thành viên của nghiệm.

Ta có thể dễ dàng thấy tại sao các thuật toán như thế được gọi là “tham ăn”. Tại mỗi bước, nó chọn “miếng ngon nhất” (được xác định bởi hàm chọn), nếu thấy có thể nuốt được (có thể đưa vào nghiệm) nó sẽ xơi ngay, nếu không nó sẽ bỏ đi, sau này không bao giờ xem xét lại.

Cần nhấn mạnh rằng, thuật toán tham ăn trong một số bài toán, nếu xây dựng được hàm chọn thích hợp có thể cho nghiệm tối ưu. Trong nhiều bài toán, thuật toán tham ăn chỉ tìm được nghiệm gần đúng với nghiệm tối ưu.

### 8.2.2. Xử lý các công việc trong thời hạn

Cho  $n$  công việc được đánh số  $1, 2, \dots, n$ . Mỗi công việc  $i$  đòi hỏi phải hoàn tất trong thời hạn  $d_i$  ( $\text{số nguyên} > 0$ ) kể từ một thời điểm nào đó. Công việc  $i$  cho ta lợi nhuận  $p_i > 0$  nếu nó được hoàn thành trong thời hạn. Một nhà máy thực hiện các công việc này, tại mỗi thời điểm nhà máy chỉ thực hiện được một công việc, mỗi công việc được làm xong trong một đơn vị thời gian. Đường nhiên nhà máy phải chọn các công việc để làm, sao cho thu được nhiều lợi nhuận nhất.

Trong bài toán trên, một tập con  $J$  các công việc sao cho nhà máy có thể hoàn thành tất cả các công việc trong thời hạn của chúng là nghiệm chấp nhận được. Giá trị của nghiệm  $J$  là tổng lợi nhuận  $\sum_{i \in J} p_i$ .

Nghiệm tối ưu là nghiệm chấp nhận được và có lợi nhuận cao nhất.

**Ví dụ.** Với  $n = 4$  và thời hạn, lợi nhuận như sau

i	1	2	3	4
$p_i$	100	10	15	27
$d_i$	2	1	2	1

Ta có các nghiệm chấp nhận được và giá trị của chúng như sau

Nghiệm	Dãy xử lý	Giá trị
{1}	1	100
{2}	2	10
{3}	3	15
{4}	4	27
{1,2}	2,1	110
{1,3}	1,3 hoặc 3,1	115
{1,4}	4,1	127
{2,3}	2,3	25
{3,4}	4,3	42

Nghiệm tối ưu là tập các công việc {1,4}, được hoàn thành đúng hạn theo thứ tự 4 trước rồi đến 1, và cho lợi nhuận 127.

Áp dụng chiến lược tham ăn, ta sẽ xây dựng J theo từng bước, xuất phát từ  $J = \emptyset$ . Tại mỗi bước ta sẽ chọn công việc cho lợi nhuận lớn nhất trong số các công việc còn lại. Không mất tính tổng quát, ta giả sử rằng các công việc đã được đánh số theo thứ tự lợi nhuận giảm dần  $p_1 \geq p_2 \geq \dots \geq p_n$ . Do đó, ta có thể đưa ra thuật toán tham ăn tìm nghiệm của bài toán xử lý công việc trong thời hạn như sau.

```

procedure Jobs;
begin   J  $\leftarrow \emptyset$ ;
          for i := 1 to n do
              if tất cả các công việc trong J  $\cup \{i\}$  có thể hoàn thành trong thời hạn
              then J  $\leftarrow J \cup \{i\}$ 
          end;

```

Trong thuật toán trên còn một vấn đề cần giải quyết là: làm thế nào để kiểm tra tập J các công việc có thể hoàn thành trong thời hạn. Nếu J có k công việc, sẽ có  $k!$  thứ tự xử lý ( $k!$  hoán vị của k công việc). Việc kiểm tra  $k!$  thứ tự xử lý đòi hỏi rất nhiều thời gian. Bổ đề sau đây chỉ ra rằng, ta chỉ cần kiểm tra một thứ tự xử lý, đó là thứ tự các công việc được sắp xếp theo thời hạn tăng dần.

**Bổ đề.** Giả sử  $J$  là tập gồm  $k$  công việc và  $\sigma = i_1 i_2 \dots$  là một hoán vị của các công việc trong  $J$  sao cho  $d_{i_1} \leq d_{i_2} \leq \dots \leq d_{i_k}$ . Khi đó  $J$  là nghiệm chấp nhận được nếu và chỉ nếu các công việc trong  $J$  được thực hiện theo thứ tự  $\sigma$  đều hoàn thành trong thời hạn.

Thật vậy, nếu các công việc trong  $J$  được thực hiện theo thứ tự  $\sigma$  đều đúng hạn thì  $J$  là chấp nhận được. Do đó, ta chỉ cần chỉ ra rằng, nếu  $J$  chấp nhận được thì khi thực hiện các công việc trong  $J$  theo thứ tự  $\sigma$ , tất cả các công việc đều đúng hạn.  $J$  là chấp nhận được có nghĩa là tồn tại một cách sắp xếp  $\sigma' = (r_1, r_2, \dots, r_k)$  sao cho  $d_{r_j} \geq j$ ,  $1 \leq j \leq k$ . Giả sử  $\sigma' \neq \sigma$ , gọi  $a$  là chỉ số nhỏ nhất mà  $r_a \neq i_a$ . Giả sử  $i_a = r_b$ , rõ ràng là  $b > a$ . Trong hoán vị  $\sigma'$  ta trao đổi  $r_a$  và  $r_b$  để nhận được hoán vị mới  $\sigma''$ . Vì  $d_{r_a} \geq d_{r_b}$  (do  $r_a = i_c$  với  $c > a$ , nên  $d_{r_a} = d_{i_c} \geq d_{i_a} = d_{r_b}$ ), nên khi thực hiện các công việc theo thứ tự  $\sigma''$ , các công việc đều hoàn thành đúng hạn. Tiếp tục cách này, ta sẽ biến đổi  $\sigma'$  thành  $\sigma$ . Bổ đề được chứng minh.

Sau đây ta sẽ chứng minh rằng thuật toán được mô tả bởi thủ tục **Jobs** luôn luôn cho ta nghiệm tối ưu của bài toán xử lý các công việc trong thời hạn.

Thật vậy, giả sử tập  $I$  các công việc là nghiệm tối ưu và  $J$  là tập các công việc được xác định bởi thủ tục **Jobs**. Giả sử  $S_I = (i_1, i_2, \dots, i_k)$  là thứ tự xử lý trong thời hạn của các công việc trong  $I$ , tức là  $i_t$  là công việc được làm từ thời điểm  $t - 1$  tới  $t$ . Tương tự ta giả sử  $S_J = (j_1, j_2, \dots, j_k)$  là thứ tự xử lý trong thời hạn của các công việc trong  $J$ . Giả sử  $i$  là một công việc thuộc cả  $I$  và  $J$ ,  $i = i_t$  và  $i = j_t$ . Nếu  $t < t'$  thì trong dãy  $S_I$  ta hoán vị  $i_{t'}$  với  $i_t$  (nếu trong dãy  $S_I$  có  $i_{t'}$ ), nếu không có  $i_{t'}$  thì ta chuyển  $i_t$  tới vị trí  $t'$ . Nếu  $t' < t$  thì trong dãy  $S_J$  ta hoán vị  $j_{t'}$  với  $j_t$  (nếu trong dãy  $S_J$  có  $j_{t'}$ ), nếu không có  $j_{t'}$  thì ta chuyển  $j_t$  tới vị trí  $t$ . Làm như thế với mọi công việc  $i$  vừa thuộc  $I$ , vừa thuộc  $J$ , ta biến đổi hai dãy  $S_I$  và  $S_J$  thành hai dãy  $S'_I$  và  $S'_J$  sao cho các công việc  $i \in I \cap J$  đứng ở cùng một vị trí trong cả hai dãy. Hai dãy  $S'_I$  và  $S'_J$  có thể có vị trí trống (không có công việc nào được thực hiện có thời điểm đó). Cả hai dãy  $S'_I$  và  $S'_J$  đều là lịch thực hiện trong thời hạn của các công việc trong  $I$  và  $J$  tương ứng (tại

sao?). Ngoài các vị trí tại đó cả hai dãy  $S'_1$  và  $S'_J$  cùng chứa một công việc, còn các khả năng sau:

- Ở vị trí nào đó, dãy  $S'_1$  chứa công việc a, còn dãy  $S'_J$  trống. Khi đó  $JU\{a\}$  là chấp nhận được. Thuật toán Jobs phải chọn a đưa vào tập J. Vậy khả năng này không xảy ra.
- Ở vị trí nào đó, dãy  $S'_J$  chứa b, còn dãy  $S'_1$  trống. Khi đó  $IU\{b\}$  là chấp nhận được và nó cho lợi nhuận cao hơn I. Điều này mâu thuẫn với I là nghiệm tối ưu.
- Chỉ còn khả năng dãy  $S'_1$  chứa a, dãy  $S'_J$  chứa b và  $a \neq b$ . Nếu  $p_a > p_b$  thì thuật toán tham ăn phải chọn a vì  $(J \setminus \{b\}) \cup \{a\}$  là chấp nhận được. Nếu  $p_a < p_b$  thì khi thay a trong I bởi b ta nhận được nghiệm chấp nhận được với lợi nhuận cao hơn. Điều này mâu thuẫn với I là nghiệm tối ưu. Như vậy  $p_a = p_b$ .

Tóm lại, trong hai dãy  $S'_1$  và  $S'_J$ , tại mỗi vị trí cả hai chứa cùng một công việc, hoặc chứa hai công việc khác nhau nhưng cho cùng lợi nhuận. Do đó nghiệm tìm được J bởi thuật toán tham ăn là nghiệm tối ưu.

Sau đây ta đưa ra một cách cài đặt thuật toán *Jobs*.

Giả sử các công việc được đánh số theo thứ tự lợi nhuận giảm dần. Mảng  $P[1 .. n]$  lưu lợi nhuận:  $P[1] \geq P[2] \geq \dots \geq P[n]$ . Mảng  $D[0 .. n]$  lưu thời hạn, trong đó  $D[0] = 0$ . Mảng  $J[0 .. n]$ , trong đó  $J[0] = 0$ , và  $J[r]$ ,  $1 \leq r \leq k$ , để lưu các công việc đã được chọn bởi thuật toán, các công việc được sắp xếp theo thời hạn tăng dần, tức là  $D[J[1]] \leq D[J[2]] \leq \dots \leq D[J[k]]$ .

Theo bổ đề, công việc i sẽ được chọn và được đưa vào thành phần thứ  $r + 1$  của mảng J nếu

$$D[J[r]] \leq D[i] \leq D[J[r + 1]]$$

đồng thời  $D[i] > r$  và  $D[J[t]] > t$  với  $t = r + 1, \dots, k$

Chúng ta có thủ tục sau đây.

**procedure** *Jobs* :

**begin**

$D[0] := 0;$

$J[0] := 0;$

$k := 1;$

```

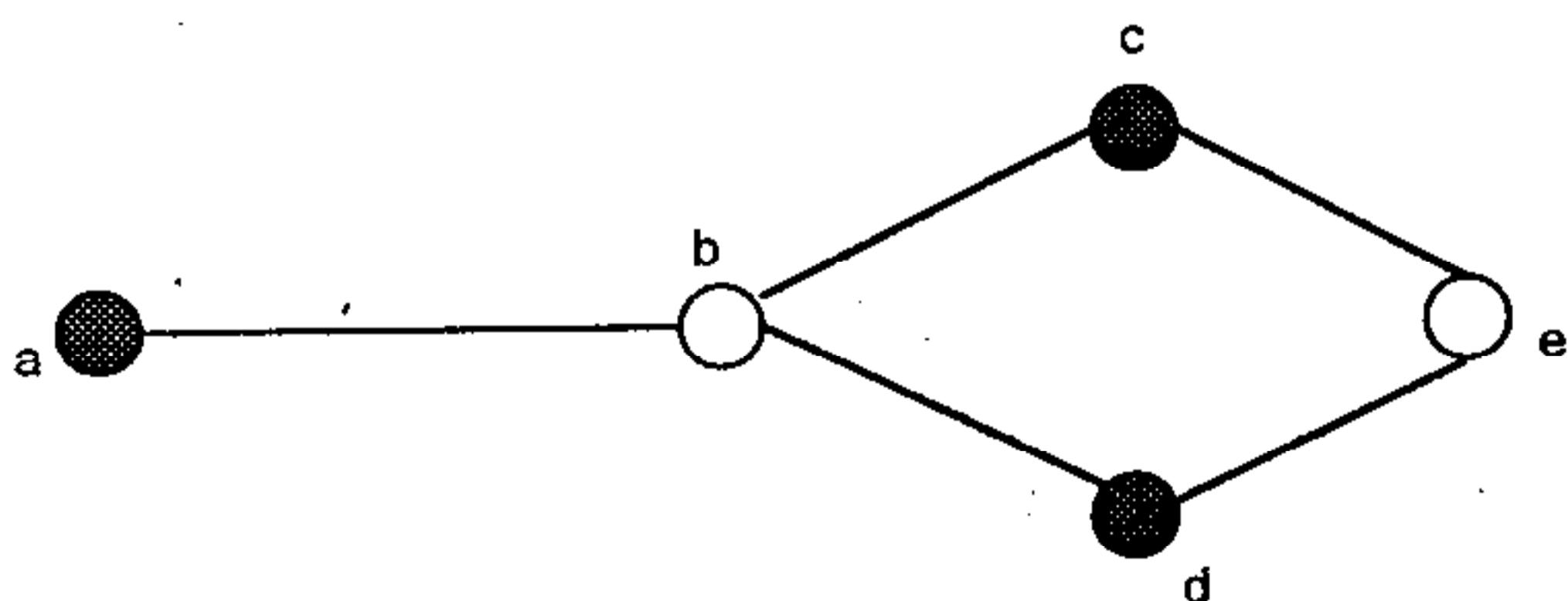
J[1] := 1;
for i := 2 to n do
    begin r := k;
        while (D[J[r]] > D[i]) and (D[J[r]] > r) do r := r - 1;
        if (D[J[r]] <= D[i]) and (D[i] > r) then
            begin
                For l := k downto r + 1 do J[l + 1] := J[l];
                J[r + 1] := i;
                k := k + 1;
            end;
        end;
    end;

```

### 8.2.3. Sơn đồ thị

Giả sử  $G = (V, E)$  là một đồ thị không định hướng. Chúng ta cần sơn các đỉnh của đồ thị sao cho hai đỉnh kề nhau được sơn bởi hai màu khác nhau và sử dụng số màu ít nhất có thể được.

Sử dụng chiến lược tham ăn, ta có thể đưa ra thuật toán như sau: Dùng một màu để sơn (ví dụ màu đỏ). Chọn một đỉnh chưa sơn và sơn đỉnh đó. Sau đó với mỗi đỉnh chưa sơn, nếu nó không kề với các đỉnh được sơn bởi màu đang sử dụng thì ta sơn nó bởi màu đó. Khi không còn đỉnh nào có thể sơn được bởi màu đó thì ta dùng màu mới (chẳng hạn màu xanh) để sơn và áp dụng cách sơn như trên. Cứ thế tiếp tục cho tới khi ta sơn hết các đỉnh của đồ thị.



Hình 8.1. Một đồ thị được sơn

**Ví dụ.** Sơn đồ thị trong hình 8.1. Ta sơn đỉnh a bởi màu đỏ, khi đó đỉnh b không được sơn bởi màu đỏ, vì nó kề a đã sơn màu đỏ. Xét tiếp đỉnh c, nó không kề a, ta sơn c màu đỏ. Xét đến đỉnh d, nó không kề a và c, do đó d sơn màu đỏ. Xét tiếp đỉnh e, nó kề đỉnh c đã sơn màu đỏ, do đó không thể sơn e màu đỏ. Còn lại hai đỉnh chưa sơn là b và e. Sử dụng màu mới (chẳng hạn màu xanh) để sơn b. Đỉnh e không kề b nên sơn nó màu xanh. Như vậy, ta chỉ cần hai màu, đây là giải pháp tối ưu.

Tuy nhiên, thuật toán tham ăn đã trình bày chỉ cho ta nghiệm “tốt”. Chẳng hạn, sau khi sơn a màu đỏ, nếu xét đến đỉnh e, ta sơn được e màu đỏ. Các đỉnh còn lại đều kề với a hoặc e, nên không thể sơn màu đỏ. Sơn b màu xanh. Không thể sơn c và d màu xanh. Phải chọn màu mới (vàng) và có thể sơn c và d màu vàng. Trong giải pháp này ta phải sử dụng 3 màu.

Giả sử  $V$  là tập hợp các đỉnh của đồ thị. Gọi  $V_0$  là tập các đỉnh chưa được sơn và  $V_1$  là tập các đỉnh được sơn bởi màu mới. Ta mã hoá các màu bởi số nguyên  $k = 1, 2, 3, \dots$ . Khi đó thuật toán đã đưa ra được mô tả bởi thủ tục sau.

```

procedure Coloring;
begin
    k := 0;
    V0 := V;
    while V0 < > Ø do
        begin
            k := k + 1;
            Chọn x ∈ V0 và sơn x bởi màu k;
            V1 := {x};
            for mỗi v ∈ V0 do
                if v không kề mọi w ∈ V1 then
                    begin
                        Sơn v bởi màu k;
                        V1 := V1 ∪ {v};
                    end;
            V0 := V0 - V1;
        end;
    end;

```

Chúng ta đã chứng tỏ thuật toán **Coloring** chỉ cho ta nghiệm tốt, gần đúng với nghiệm tối ưu. Thế thì tại sao ta lại phải quan tâm đến các thuật toán như thế? Đối với bài toán sơn đồ thị và rất nhiều bài toán khác, các thuật toán tìm nghiệm chính xác đòi hỏi thời gian mű, trong thực tế không sử dụng được khi cõ bài toán khá lớn. Bài toán người bán hàng (salesperson) mà chúng ta đã xét trong mục 2.4 cũng là bài toán mà các thuật toán cho nghiệm tối ưu đều có độ phức tạp mű. Đối với các bài toán như thế, chúng ta đành phải sử dụng các thuật toán cho nghiệm gần đúng, nhưng thời gian tính toán nhanh.

### 8.3. QUAY LUI

#### 8.3.1. Phương pháp

Trong các kỹ thuật cơ bản thiết kế thuật toán, quay lui là một trong những kỹ thuật quan trọng nhất. Nó có thể được áp dụng để thiết kế thuật toán tìm ra một nghiệm hoặc tất cả các nghiệm của bài toán.

Trong nhiều vấn đề, việc tìm nghiệm có thể quy về việc tìm một vectơ hữu hạn ( $x_1, x_2, \dots, x_n, \dots$ ) nhưng độ dài vectơ có thể không được xác định trước. Vectơ này cần phải thoả mãn một số điều kiện nào đó tùy thuộc vào vấn đề cần giải. Các thành phần  $x_i$  của vectơ được chọn ra từ một tập hữu hạn  $A_i$  ( $i = 1, 2, \dots, n, \dots$ ).

**Ví dụ.** Bài toán 8 con hậu.

Chúng ta cần đặt 8 con hậu vào bàn cờ  $8 \times 8$  sao cho chúng không tấn công nhau, tức là không có cặp con hậu nào nằm cùng hàng, cùng cột, cùng đường chéo.

Do các con hậu phải nằm trên các hàng khác nhau, ta sẽ đánh số các con hậu từ 1 đến 8, con hậu  $i$  là con hậu nằm ở dòng thứ  $i$  ( $i = 1, 2, \dots, 8$ ). Gọi  $x_i$  là cột mà con hậu thứ  $i$  đứng. Như vậy nghiệm của bài toán 8 con hậu là vectơ  $(x_1, x_2, \dots, x_8)$ , trong đó  $1 \leq x_i \leq 8$ , tức là  $x_i$  được chọn từ tập  $A_i = \{1, 2, \dots, 8\}$ . Vectơ  $(x_1, x_2, \dots, x_8)$  là nghiệm nếu  $x_i \neq x_j$  và hai ô  $(i, x_i), (j, x_j)$  không nằm trên cùng một đường chéo.

Tư tưởng của phương pháp quay lui là như sau: Ta xây dựng vectơ nghiệm dần từng bước, bắt đầu từ vectơ không  $( )$ . Thành phần đầu tiên  $x_1$  được chọn ra từ tập  $S_1 = A_1$ . Khi đã chọn được các thành phần  $x_1, \dots, x_{i-1}$

thì từ các điều kiện của bài toán ta sẽ xác định được tập  $S_i$ , các ứng cử viên có thể chọn làm thành phần  $x_i$ . Tập  $S_i$  là tập con của  $A_i$  và phụ thuộc vào các thành phần  $x_1, x_2, \dots, x_{i-1}$  đã chọn. Chọn một phần tử  $x_i$  từ  $S_i$  ta mở rộng nghiệm một phần  $(x_1, x_2, \dots, x_{i-1})$  đến nghiệm một phần  $(x_1, x_2, \dots, x_i)$ . Lặp lại quá trình trên để tiếp tục mở rộng nghiệm một phần  $(x_1, x_2, \dots, x_{i-1}, x_i)$ . Nếu không thể chọn được thành phần  $x_{i+1}$  (khi  $S_{i+1} = \emptyset$ ) thì ta quay lại chọn một phần tử khác của  $S_i$  làm  $x_i$ . Nếu không còn một phần tử nào khác của  $S_i$  ta quay lại chọn một phần tử khác của  $S_{i-1}$  làm  $x_{i-1}$  và cứ thế tiếp tục. Trong quá trình mở rộng nghiệm, ta phải kiểm tra nghiệm một phần có là nghiệm của bài toán hay không. Nếu chỉ cần tìm một nghiệm thì khi gặp nghiệm ta dừng lại. Còn nếu cần tìm tất cả các nghiệm thì quá trình chỉ dừng lại khi tất cả các khả năng chọn các thành phần của vectơ nghiệm đã bị vét cạn.

Lược đồ tổng quát của thuật toán quay lui có thể biểu diễn bởi thủ tục ***Backtrack*** sau:

```

procedure Backtrack;
begin
     $S_1 := A_1;$ 
     $k := 1;$ 
    while  $k > 0$  do
        begin
            while  $S_k \neq \emptyset$  do
                begin
                    Chọn  $x_k \in S_k$ ;
                     $S_k := S_k - \{x_k\}$ ;
                    if  $(x_1, \dots, x_k)$  là nghiệm then viết ra nghiệm;
                     $k := k + 1$ ;
                    Xác định  $S_k$ ;
                end;
                 $k := k - 1$ ; {quay lui}
            end;
        end;
    end;

```

Thuật toán quay lui có thể được biểu diễn bởi thủ tục đệ quy **RBacktrack**. Đó là thủ tục chọn thành phần thứ i của vectơ nghiệm. Trong thủ tục này ta sử dụng phép toán thêm thành phần mới vào vectơ nghiệm (ký hiệu là +) và phép toán loại thành phần cuối cùng khỏi vectơ (ký hiệu là -)

$$(a_1, a_2, \dots, a_n) + (b) = (a_1, a_2, \dots, a_n, b)$$

$$(a_1, a_2, \dots, a_n) - (a_n) = (a_1, a_2, \dots, a_{n-1})$$

```

procedure RBacktrack (vector, i);
begin
    Xác định  $S_i$ ;
    for  $x_i \in S_i$  do
        begin
            vector := vector + ( $x_i$ );
            if vector là nghiệm then viết ra vector;
            RBacktrack (vector, i + 1);
            vector := vector - ( $x_i$ );
        end;
    end;

```

Khi áp dụng lược đồ tổng quát của thuật toán quay lui cho các bài **bản cù** thể, có ba điểm quan trọng cần lưu ý là:

- Tìm cách biểu diễn nghiệm của bài toán dưới dạng một dãy các đối tượng được chọn dần từng bước ( $x_1, x_2, \dots, x_i, \dots$ ).
- Xác định được tập  $S_i$  các ứng cử viên được chọn làm thành phần thứ i của vectơ nghiệm. Chọn cách thích hợp để biểu diễn  $S_i$ .
- Tìm các điều kiện để một vectơ đã chọn là nghiệm của bài toán.

### *Cây không gian trạng thái*

Việc tìm kiếm vectơ nghiệm ( $x_1, \dots, x_i, x_{i+1}, \dots$ ) bằng phương pháp quay lui có thể quy về việc tìm kiếm trên cây không gian trạng thái. Cây được xây dựng theo từng mức như sau: Các đỉnh con thuộc gốc là các phần tử thuộc  $S_1$ . Giả sử  $x_i$  là đỉnh ở mức thứ i. Khi đó các đỉnh con

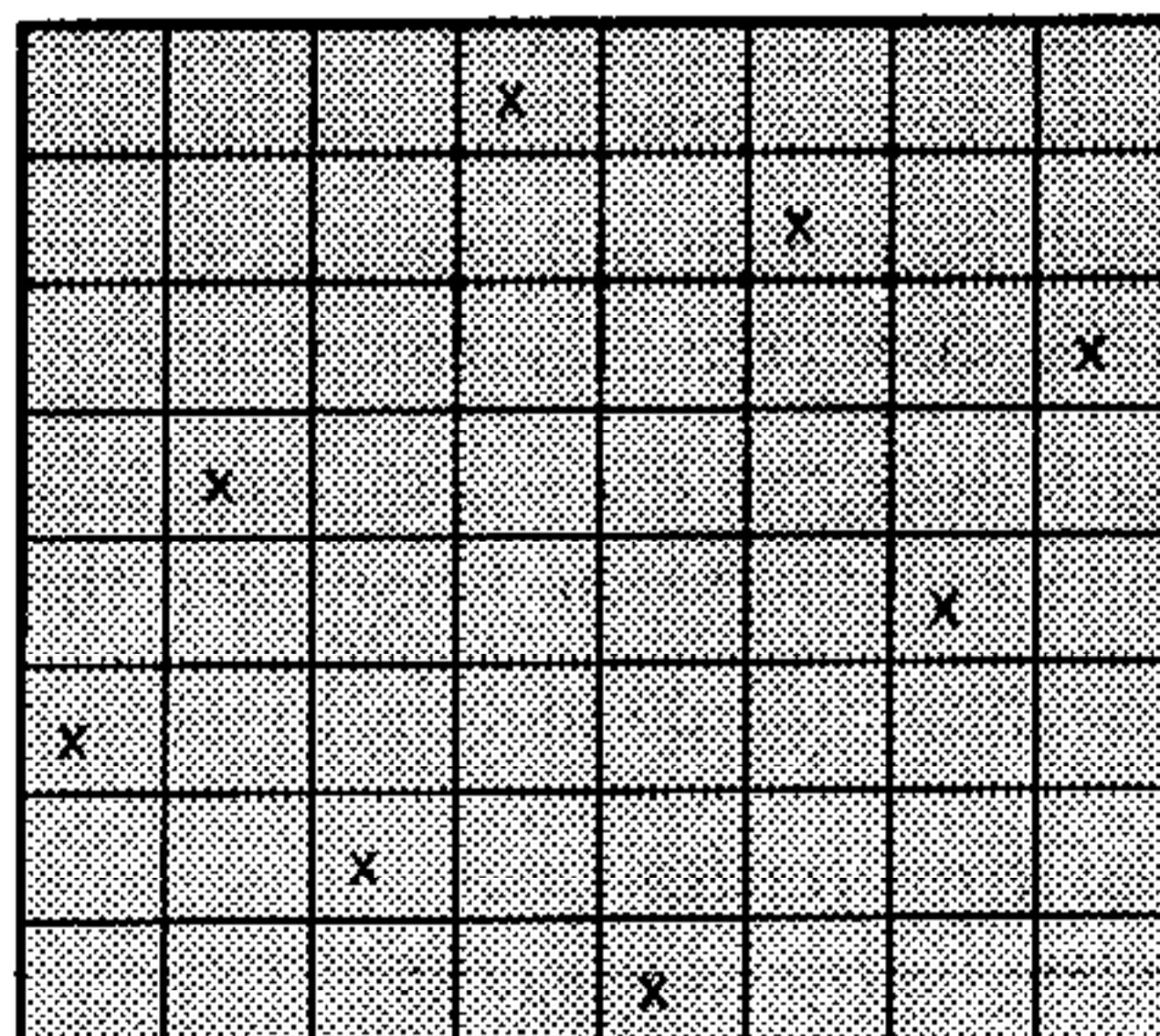
của  $x_i$  là các phần tử thuộc  $S_{i+1}$ , trong đó  $S_{i+1}$  là tập các ứng cử viên của thành phần thứ  $i+1$  của vectơ nghiệm khi ta đã chọn các thành phần  $x_1, \dots, x_i$ . Ở đây  $x_1, \dots, x_i$  là các đỉnh nằm trên đường đi từ gốc tới  $x_i$ . Như vậy, mỗi đỉnh của cây không gian trạng thái biểu diễn một nghiệm một phần, đó là vectơ mà các thành phần của nó theo thứ tự là các đỉnh nằm trên đường đi từ gốc tới đỉnh đó.

Việc tìm kiếm nghiệm theo chiến lược quay lui chẳng qua là tìm kiếm theo độ sâu trên cây không gian trạng thái (hay đi qua cây theo thứ tự preorder).

### 8.3.2. Bài toán 8 con hậu

Trong bài toán 8 con hậu, nghiệm của bài toán có thể biểu diễn dưới dạng vectơ  $(x_1, x_2, \dots, x_8)$ , trong đó  $x_i$  là tọa độ cột của con hậu đứng ở dòng thứ  $i$ ,  $x_i \in \{1, 2, \dots, 8\}$ . Các con hậu không đứng cùng cột, tức là  $x_i \neq x_j$  với  $i \neq j$ . Điều kiện để ô  $(i, x_i)$  không cùng đường chéo với ô  $(j, x_j)$  là  $|i - j| \neq |x_i - x_j|$ . Do đó khi ta đã chọn được  $(x_1, \dots, x_{k-1})$  thì  $x_k$  được chọn là cột thoả mãn các điều kiện:

$$\begin{aligned} x_k &\neq x_i \\ |x_k - x_i| &\neq |k - i| \\ \text{với mọi } 1 \leq i &< k \end{aligned}$$



**Hình 8.2.** Một nghiệm của bài toán 8 con hậu

Trong thủ tục *Queen* dưới đây, vecto nghiệm được biểu diễn bởi mảng  $x[1 .. 8]$ . Với mỗi  $k$ , ta lần lượt cho  $x[k]$  nhận giá trị từ 1 tới 8 và kiểm tra các điều kiện mà  $x[k]$  cần thoả mãn, nếu nó không thoả mãn (biến OK = false) thì tăng  $x[k]$  lên 1 đơn vị.

```

procedure Queen;
var x: Array [1 .. 8] of integer;
    i, k : integer;
    OK : boolean;
begin
    k := 1 ;
    x[k] := 0;
    while k > 0 do
        begin
            x[k] := x[k] + 1;
            OK := false;
            while (x[k] <= 8) and (not OK) do
                begin
                    i := 1;
                    OK := true;
                    while (i < k) and OK do
                        if (x[i] <> x[k]) and (abs(x[i]-x[k]) <> abs(i-k)) then
                            i := i + 1 else OK := false;
                    if not OK then x[k] := x[k] + 1;
                end;
                if OK then
                    if k = 8 then viết ra mảng x[1 .. 8]
                    else begin
                        k := k + 1;
                        x[k] := 0;
                    end
                else k := k - 1; {quay lùi}
            end;
        end;

```

### 8.3.3. Các tập con có tổng cho trước

Giả sử A là một tập n số nguyên dương và M là một số nguyên dương cho trước. Chúng ta muốn tìm tập con các số trong A sao cho tổng của chúng là M.

Để giải quyết bài toán này, ta biểu diễn tập A dưới dạng dãy  $(a_1, a_2, \dots, a_n)$ . Ta cần tìm dãy con  $(a_{i_1}, a_{i_2}, \dots, a_{i_k})$ ,  $1 \leq i_1 < i_2 < \dots < i_k \leq n$  sao cho  $a_{i_1} + a_{i_2} + \dots + a_{i_k} = M$ . Như vậy, nghiệm của bài toán là dãy  $(i_1, i_2, \dots, i_k)$  sao cho  $1 \leq i_1 < i_2 < \dots < i_k \leq n$  và  $a_{i_1} + a_{i_2} + \dots + a_{i_k} = M$ .

Đương nhiên có thể chọn  $i_1$  là một trong các chỉ số 1, 2, ..., n mà  $a_{i_1} \leq M$ . Khi đã chọn được  $i_1, \dots, i_k$  và  $S = a_{i_1} + a_{i_2} + \dots + a_{i_k} < M$  thì  $i_{k+1}$  có thể chọn là một trong các chỉ số từ  $i_{k+1}$  tới n mà  $S + a_{i_{k+1}} \leq M$ .

Trong thủ tục sau đây, ta sử dụng mảng A[1 .. n] để lưu các số nguyên thuộc tập đã cho. Mảng I[1 .. n] lưu chỉ số các thành phần thuộc tập con cần tìm. Biến S lưu tổng các số của tập con trong quá trình hình thành.

```

procedure Subsets;
begin
  k := 1;
  I[1] := 0;
  S := 0;
  while k > 0 do
    begin
      I[k] := I[k] + 1;
      if I[k] <= n then
        begin
          if S + A[I[k]] <= M then
            if S + A[I[k]] = M then viết ra mảng I[1 .. k]
          else begin
            S := S + A[I[k]];
            I[k + 1] := I[k];
          end
        end
    end
  end
end

```

```

k := k + 1;
end
end
else begin
    k := k - 1;
    S := S - A[l[k]];
end;
end;

```

## 8.4. NHÁNH VÀ CẬN

### 8.4.1. Phương pháp

Phương pháp nhánh và cận là một dạng cải tiến của phương pháp quay lui, được áp dụng để tìm nghiệm của bài toán tối ưu.

Giả sử nghiệm của bài toán có thể biểu diễn dưới dạng một vectơ  $(a_1, a_2 \dots, a_n)$ , mỗi thành phần  $a_i$  ( $i = 1, \dots, n$ ) được chọn ra từ tập  $S$  các đối tượng nào đó. Mỗi nghiệm  $(a_1, \dots, a_k)$  của bài toán được gắn với một giá trị  $\text{cost}(a_1, \dots, a_k)$  và ta cần tìm nghiệm có giá thấp nhất (nghiệm tối ưu).

Giả sử giá của các nghiệm một phần cũng được xác định và là các số thực không âm, đồng thời với nghiệm một phần bất kỳ  $(a_1, \dots, a_{k-1})$  và nghiệm mở rộng của nó  $(a_1, \dots, a_{k-1}, a_k)$  ta luôn có

$$\text{cost}(a_1, \dots, a_{k-1}) \leq \text{cost}(a_1, \dots, a_{k-1}, a_k)$$

Tư tưởng của phương pháp nhánh và cận là như sau. Trong quá trình mở rộng từng bước nghiệm một phần, khi ta đạt được nghiệm một phần  $(a_1, \dots, a_k)$ , nếu biết rằng tất cả các nghiệm mở rộng của nó  $(a_1, \dots, a_k, a_{k+1}, \dots)$  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  $(a_1, \dots, a_k)$  nữa.

Giả sử  $\text{cost}^*(a_1, \dots, a_k)$  là cận dưới của giá của các nghiệm  $(a_1, \dots, a_k, a_{k+1}, \dots)$ , với  $(a_1, \dots, a_k, a_{k+1}, \dots)$  là mở rộng của nghiệm một phần  $(a_1, \dots, a_k)$ . Gọi giá của nghiệm tốt nhất là  $\text{lowcost}$ .

Thực chất của phương pháp nhánh và cận là tìm kiếm theo độ sâu trên cây không gian trạng thái như kỹ thuật quay lui. Chỉ có điều khác là khi đạt tới đỉnh  $a_k$  mà  $\text{cost}^*(a_1, \dots, a_{k-1}, a_k) \geq \text{lowcost}$  thì ta cắt đi tất cả các nhánh từ  $a_k$  đi xuống các đỉnh con của nó. Tức là, khi đó ta không đi xuống các đỉnh con của  $a_k$  nữa mà quay lên cha của nó là  $a_{k-1}$ .

Dùng làm giá trị ban đầu của lowcost ta có thể lấy  $\text{lowcost} = +\infty$  hoặc  $\text{lowcost}$  là giá trị của một nghiệm được tìm thấy bằng phương pháp thực nghiệm nào đó. Thuật toán nhánh và cận có thể được mô tả bởi thủ tục **BanchBound** sau.

```

procedure BanchBound;
begin
    lowcost :=  $\infty$ ;
    cost* := 0;
    Tính  $S_1$ ;
    k := 1;
    while k > 0 do
        begin
            while ( $S_k \neq \emptyset$ ) and (cost* < lowcost) do
                begin
                    Chọn  $a_k \in S_k$ ;
                     $S_k := S_k - \{a_k\}$ ;
                    cost* := cost*( $a_1, \dots, a_k$ );
                    if ( $a_1, \dots, a_k$ ) là nghiệm then
                        if cost( $a_1, \dots, a_k$ ) < lowcost then
                            lowcost := cost( $a_1, \dots, a_k$ );
                    k := k + 1;
                    Tính  $S_k$ ;
                end;
                k := k - 1;
                cost* := cost*( $a_1, \dots, a_k$ );
            end;
        end;

```

Như vậy, với phương pháp nhánh và cận, ta không phải duyệt toàn bộ cây không gian trạng thái để tìm ra nghiệm tốt nhất mà bằng cách đánh giá cận dưới cost\* của các nghiệm là mở rộng của nghiệm một phần, ta có thể cắt bỏ đi những nhánh không cần thiết, do đó việc tìm ra nghiệm tối ưu sẽ nhanh hơn. Cái khó nhất trong việc áp dụng phương pháp nhánh và cận là xây dựng được hàm đánh giá cận dưới cost\*. Hàm này có được xây dựng tốt thì mới giúp ta cắt bỏ được nhiều nhánh không cần thiết và thuật toán nhận được mới có cải tiến đáng kể so với thuật toán vét cạn.

#### 8.4.2. Bài toán người bán hàng

Chúng ta trở lại với bài toán người bán hàng đã được xét ở mục 2.4. Bài toán được quy về bài toán trên đồ thị: Cho  $G = (V, E)$  là đồ thị định hướng, tìm chu trình xuất phát từ một đỉnh qua tất cả các đỉnh còn lại với độ dài ngắn nhất. Trong mục 2.4, chúng ta đã đưa ra một thuật toán tham ăn cho ta nghiệm gần đúng. Sau đây chúng ta sẽ áp dụng kỹ thuật nhánh và cận để giải bài toán này.

Giả sử đồ thị có  $n$  đỉnh,  $V = \{1, 2, \dots, n\}$  và đỉnh xuất phát là 1. Nghiệm một phần là đường đi  $(a_1, a_2, \dots, a_k)$ ,  $a_1 = 1$ ,  $a_i \neq a_j$  ( $i \neq j$ ). Các nghiệm là mở rộng của nghiệm một phần  $(a_1, a_2, \dots, a_k)$  sẽ là các chu trình có đoạn đầu trùng với  $(a_1, a_2, \dots, a_k)$ . Sau đây ta sẽ đưa ra một cách đánh giá cận dưới của các chu trình là mở rộng của đường đi  $(a_1, a_2, \dots, a_k)$ .

Chúng ta sẽ trình bày phương pháp đánh giá cận dưới qua một đồ thị cụ thể. Giả sử  $G$  là một đồ thị gồm 5 đỉnh với độ dài các cung được cho bởi ma trận sau:

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

Một đường đi xuất phát từ 1, chặng hạn  $(1, 4, 3)$  sẽ có hai nghiệm mở rộng, đó là chu trình  $(1, 4, 3, 5, 2, 1)$  có độ dài là  $10 + 9 + 16 + 7 + 14 = 56$

và chu trình  $(1, 4, 3, 2, 5, 1)$  có độ dài  $10 + 9 + 5 + 7 + 8 = 49$ . Giả sử ta cần đánh giá độ dài đường đi từ đỉnh  $i$  tới đỉnh  $j$  và phải đi qua các đỉnh  $k$  thuộc một tập  $K$  các đỉnh nào đó, tức là đường đi có dạng  $(i, \dots, k, \dots, j)$ ,  $k \in K$ . Giá của đường đi này được phân thành: giá rời khỏi đỉnh  $i$ , giá thăm các đỉnh  $k$ , giá đến đỉnh  $j$ . Giá thăm đỉnh  $k$  lại được phân thành giá đến  $k$  và giá rời  $k$ . Chẳng hạn, khi rời đỉnh 1 ta phải đi một đoạn đường ít nhất là 2, vì  $2 = \min(14/2, 4/2, 10/2, 20/2)$ . Khi đến đỉnh 2 ta phải đi một đoạn đường ít nhất là  $5/2 = \min(14/2, 5/2, 7/2, 7/2)$ , khi rời đỉnh 2 đoạn đường ít nhất phải đi là  $7/2 = \min(14/2, 7/2, 8/2, 7/2)$ . Vậy giá thăm đỉnh 2 ít nhất là  $5/2 + 7/2 = 6$ . Bằng cách này ta đánh giá được cận dưới độ dài các đường đi từ  $i$  đến  $j$  qua các đỉnh  $k \in K$ . Chẳng hạn, trong đồ thị trên, các chu trình xuất phát từ 1 qua các đỉnh 2, 3, 4, 5, rồi trở về 1 có độ dài ít nhất là

$$2 + 6 + 4 + 3 + 3 + 2 = 20$$

Bây giờ, ta đánh giá cận dưới độ dài các chu trình là mở rộng của đường đi  $(1, 2)$ :

- Độ dài cung  $(1, 2)$  là 14
- Cận dưới độ dài các đường đi từ 2 tới 1 và qua các đỉnh 3, 4, 5 được tính như sau:

Giá rời đỉnh 2:  $\min(7/2, 8/2, 7/2) = 7/2$

Giá đến đỉnh 3:  $\min(7/2, 9/2, 17/2) = 7/2$

Giá rời đỉnh 3:  $\min(4/2, 7/2, 16/2) = 4/2$

Như vậy giá thăm đỉnh 3 là  $11/2$

Tương tự giá thăm đỉnh 4 là 3

Giá thăm đỉnh 5 là 3

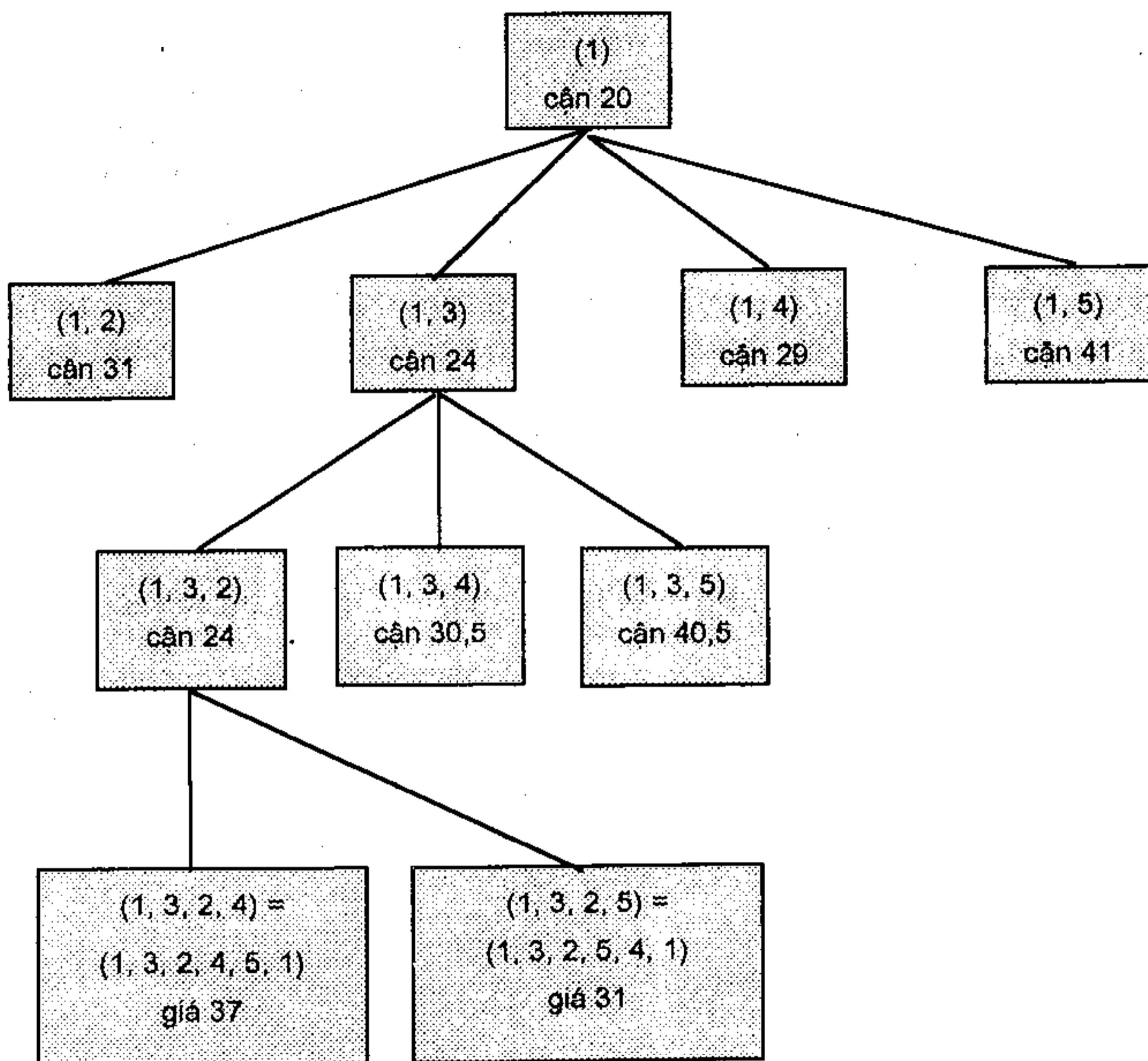
Giá trở về đỉnh 1 (từ 3 hoặc 4 hoặc 5) là 2

Như vậy cận dưới độ dài các đường đi từ 2 đến 1 qua 3, 4, 5 là

$$7/2 + 11/2 + 3 + 3 + 2 = 17$$

Do đó cận dưới độ dài các chu trình là mở rộng của đường đi  $(1, 2)$  là:  $14 + 17 = 31$  (tức là  $\text{cost}^*(1, 2) = 31$ ). Một cách tương tự ta tính được  $\text{cost}^*(1, 3) = 24$ ,  $\text{cost}^*(1, 4) = 29$ ,  $\text{cost}^*(1, 5) = 41$ .

Hình 8.3 biểu diễn một phần cây không gian trạng thái. Mỗi đỉnh biểu diễn một nghiệm một phần và cận dưới của nó.



**Hình 8.3.** Một phần cây không gian trạng thái

Khi mở rộng các nghiệm một phần tới thời điểm nhận được các nghiệm một phần được biểu diễn bởi hình 8.3, ta đã tìm được nghiệm tốt nhất ở thời điểm đó là chu trình  $(1, 3, 2, 5, 4, 1)$  có độ dài là 31. Tức là tới thời điểm đó  $\text{lowcost} = 31$  và do đó không cần mở rộng các nghiệm một phần  $(1, 2)$ ,  $(1, 5)$  và  $(1, 3, 5)$  vì cận dưới của chúng  $\geq 31$ .

## 8.5. QUY HOẠCH ĐỘNG

### 8.5.1. Phương pháp

Trong nhiều trường hợp, để giải một bài toán đã cho ta đưa về giải một số bài toán con rồi kết hợp nghiệm của các bài toán con ta nhận được nghiệm của bài toán cần giải. Để giải các bài toán con này ta lại đưa về việc giải các bài toán con nhỏ hơn. Quá trình trên sẽ tiếp tục cho tới khi ta nhận được các bài toán con có thể giải được dễ dàng. Đó là kỹ thuật chia - để - trị mà ta đã xét trong mục 8.1. Chia - để - trị là kỹ thuật từ trên xuống (top - down), nó giải các bài toán con nhận được trong quá trình chia nhỏ một cách độc lập. Tuy nhiên, trong quá trình phân chia như thế, thông thường ta gấp rất nhiều lần cùng một bài toán con. Do đó nếu giải quyết bài toán bằng kỹ thuật top - down chúng ta sẽ phải tính lại nhiều lần cùng một bài toán. Thuật toán nhận được sẽ kém hiệu quả.

**Ví dụ.** Tính các hệ số nhị thức

Số tổ hợp chập  $k$  của  $n$ , ký hiệu là  $C_n^k$  hoặc  $\binom{n}{k}$ , là số các cách chọn  $k$  phần tử khác nhau từ một tập  $n$  phần tử. Các số tổ hợp còn được gọi là hệ số nhị thức. Các hệ số nhị thức có rất nhiều ứng dụng trong toán học và thường được sử dụng để phân tích và đánh giá các thuật toán.

Công thức sau đây cho phép ta tính được  $C_n^k$  thông qua  $C_{n-1}^{k-1}$  và  $C_{n-1}^k$

$$C_j^i = 1 \text{ nếu } i = 0 \text{ hoặc } i = j$$

$$C_j^i = C_{j-1}^i + C_{j-1}^{i-1} \text{ nếu } 0 < i < j$$

Chúng ta có thể tính các hệ số nhị thức một cách trực tiếp bởi hàm đệ quy sau:

```

function      Coef(k, n: integer) : integer;
begin
    if (k = 0) or (k = n) then coef := 0
    else coef := coef(k, n - 1) + coef(k - 1, n - 1);
end;

```

Khi đó, để tính  $\text{coef}(k, n)$ , rất nhiều giá trị  $\text{coef}(i, j)$  với  $i < k, j < n$  được tính lặp lại nhiều lần. Chẳng hạn, để tính  $\text{coef}(3, 5)$  ta phải tính lặp lại 2 lần  $\text{coef}(2, 3)$ , 3 lần  $\text{coef}(1, 2)$  ... Thời gian tính sẽ rất lớn, thực tế là không chấp nhận được khi các giá trị đầu vào lớn.

Một cách tiếp cận khác để tính  $C_n^k$  là ta tính các  $C_j^i$  với  $i < k, j < n$  xuất phát từ trường hợp đơn giản nhất  $C_j^0 = 1$  với  $j \leq n$ . Trong quá trình tính ta sẽ sử dụng một bảng để lưu các giá trị đã tính. Bằng cách đó ta sẽ tránh được việc tính lại nhiều lần cùng một giá trị nào đó. Cụ thể ở đây ta sử dụng mảng  $C[0 .. k, 0 .. n]$ , trong đó  $C[i,j]$  ( $0 \leq i \leq k, 0 \leq j \leq n$ ) lưu  $C_j^i$  và tính  $C[i,j]$  lần lượt theo hàng. Thực tế, vì  $i \leq j$ , ta chỉ cần tính các giá trị của tam giác nằm trên đường chéo chính. Vì  $C_j^i = C_{j-1}^i + C_{j-1}^{i-1}$ , mỗi phần tử  $C[i,j]$  ở hàng  $i$  cột  $j$ , là tổng của hai phần tử, một phần tử ở bên trái nó, phần tử  $C[i,j - 1]$ , và phần tử nằm trên phần tử này, phần tử  $C[i - 1,j - 1]$ .

Sau đây là bảng lưu các giá trị  $C_j^i$ ,  $0 \leq i \leq k = 3$  và  $0 \leq j \leq n = 5$

$j =$	0	1	2	3	4	5
$i =$	0	1 1 1 1 1 1				
	1		1 2 3 4 5			
	2			1 3 6 10		
	3				1 4 10	

Chúng ta dễ dàng viết được thủ tục tính  $C_j^i$ , với  $0 \leq i \leq k, 0 \leq j \leq n$  và  $i \leq j$ , bằng phương pháp này:

```

begin
  for j := 0 to n do C[0,j] := 1;
  for i := 1 to k do
    begin
      C[i,i] := 1;

```

```

for j := i + 1 to n do
    C[i,j] := C[i,j - 1] + C[i - 1,j - 1];
end;
end;

```

Quy hoạch động là kỹ thuật từ dưới lên (bottom - up). Chúng ta xuất phát từ những trường hợp riêng đơn giản nhất của bài toán, thường là thấy ngay nghiệm của chúng. Sau đó kết hợp nghiệm của chúng, ta được nghiệm của bài toán con có cỡ lớn hơn. Rồi lại kết hợp nghiệm của các bài toán con này để nhận được nghiệm của bài toán lớn hơn nữa, và cứ thế tiếp tục cho đến khi nhận được nghiệm của bài toán đã cho. Tư tưởng cơ bản của phương pháp quy hoạch động là trong quá trình “đi từ dưới lên”, ta sử dụng một bảng để lưu giữ lời giải của các bài toán con đã giải. Khi giải một bài toán con cần đến nghiệm của bài toán con cỡ nhỏ hơn, ta chỉ cần tìm ở trong bảng, không cần phải giải lại. Chính vì thế mà các thuật toán được thiết kế bằng quy hoạch động sẽ rất có hiệu quả.

Để giải một bài toán bằng quy hoạch động, chúng ta cần phải tiến hành những công việc sau.

- Tìm nghiệm của các bài toán con (các trường hợp riêng) đơn giản nhất.
- Tìm ra các công thức (hoặc quy tắc) xây dựng nghiệm của bài toán con thông qua nghiệm của các bài toán con cỡ nhỏ hơn.
- Tạo ra một bảng để lưu giữ các nghiệm của các bài toán con. Sau đó tính nghiệm của các bài toán con theo các công thức đã tìm ra và lưu vào bảng.
- Từ bảng đã làm đầy tìm cách xây dựng nghiệm của bài toán. Sau đây chúng ta sẽ đưa ra một số ví dụ minh họa.

### 8.5.2. Bài toán sắp xếp các đồ vật vào ba lô

Một chiếc ba lô có thể chứa được một khối lượng w. Có n loại đồ vật được đánh số 1, 2, ..., n. Mỗi đồ vật loại i có khối lượng  $a_i$  và có giá trị

$c_i$  ( $i = 1, 2, \dots, n$ ). Cần sắp xếp các đồ vật vào ba lô để ba lô có giá trị lớn nhất có thể được. Giả sử rằng mỗi loại đồ vật có đủ nhiều để xếp.

Phát biểu lại bài toán. Cho trước  $w, a_i, c_i$  ( $i = 1, 2, \dots, n$ ) là các số nguyên dương. Cần tìm  $x = (x_1, x_2, \dots, x_n)$ , trong đó  $x_i$  là số nguyên không âm sao cho:

$$\sum_{i=1}^n x_i a_i \leq w \quad (1)$$

$$\text{và } \sum_{i=1}^n x_i c_i \text{ đạt max} \quad (2)$$

Xét trường hợp riêng đơn giản nhất: chỉ có một loại đồ vật ( $n = 1$ ). 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  $x_1 = w \text{ div } a_1$ .

Gọi  $f(k, v)$  là giá trị lớn nhất của ba lô chứa được khối lượng  $v$  và chỉ xếp  $k$  loại đồ vật  $1, 2, \dots, k$ , với  $1 \leq k \leq n, 1 \leq v \leq w$ . Ta tìm công thức để tính  $f(k, v)$ .

Với  $k = 1, 1 \leq v \leq w$ , ta có :

$$x_1 = v \text{ div } a_1 \text{ và } f(1, v) = x_1 c_1$$

Giả sử ta đã tính được  $f(s, u)$ , với  $1 \leq s < k$  và  $1 \leq u \leq v$ , ta cần tính  $f(k, v)$  với  $1 \leq v \leq w$ .

Gọi  $y_k = v \text{ div } a_k$ , ta có:

$$f(k, v) = \max\{f(k - 1, u) + x_k c_k\} \quad (3)$$

trong đó, max được lấy với tất cả  $x_k = 0, 1, \dots, y_k$  và  $u = v - x_k a_k$

Giá trị lớn nhất của ba lô sẽ là  $f(n, w)$ .

Chúng ta sẽ sử dụng mảng  $A[1..n, 1..w]$  để lưu các kết quả trung gian.  $A[k, v]$  ( $1 \leq k \leq n, 1 \leq v \leq w$ ) là bản ghi gồm hai trường, một trường chứa  $f(k, v)$  và một trường chứa  $x_k$ , trong đó  $x_k$  là giá trị mà trong (3) biểu thức  $f(k - 1, u) + x_k c_k$  đạt max.

Từ công thức (3), ta suy ra rằng các giá trị của bảng A có thể tính được dễ dàng lần lượt theo dòng 1, 2, ..., n.

Vấn đề đặt ra bây giờ là từ bảng A đã làm đầy, làm thế nào để xác định được  $x = (x_1, x_2, \dots, x_n)$  và  $f(n, w)$ .

Ô A[n, w] chứa  $f(n, w)$  và  $x_n$ . Tính  $v = w - x_n a_n$ . Tìm đến ô A[n - 1, v] ta biết được  $x_{n-1}$ . Tiếp tục quá trình đó, ta tìm được  $x_{n-2}, \dots, x_2, x_1$ .

### 8.5.3. Dãy con chung của hai dãy số

Cho hai dãy số nguyên  $a = (a_1, \dots, a_m)$  và  $b = (b_1, \dots, b_n)$ . Chúng ta cần tìm dãy số nguyên  $c = (c_1, \dots, c_k)$  sao cho  $c$  là dãy con của  $a$  và  $b$  và  $c$  có độ dài lớn 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, khi một trong hai dãy  $a$  và  $b$  rỗng ( $m = 0$  hoặc  $n = 0$ ), ta thấy ngay dãy con dài nhất là dãy rỗng.

Ta xét các đoạn đầu của hai dãy  $a$  và  $b$  có độ dài  $i$  và  $j$  tương ứng  $(a_1, a_2, \dots, a_i)$  và  $(b_1, b_2, \dots, b_j)$ , với  $0 \leq i \leq m$ ,  $0 \leq j \leq n$ . Gọi  $L(i, j)$  là độ dài lớn nhất của dãy con chung của hai dãy  $(a_1, a_2, \dots, a_i)$  và  $(b_1, b_2, \dots, b_j)$ . Như vậy,  $L(m, n)$  sẽ là độ dài lớn nhất của dãy con chung của  $a$  và  $b$ .

Bây giờ ta tìm cách tính  $L(i, j)$  thông qua  $L(s, t)$  với  $s \leq i$ ,  $t \leq j$ . Để dàng thấy sự đúng đắn của công thức sau:

- Nếu  $i = 0$  hoặc  $j = 0$  thì  $L(i, j) = 0$

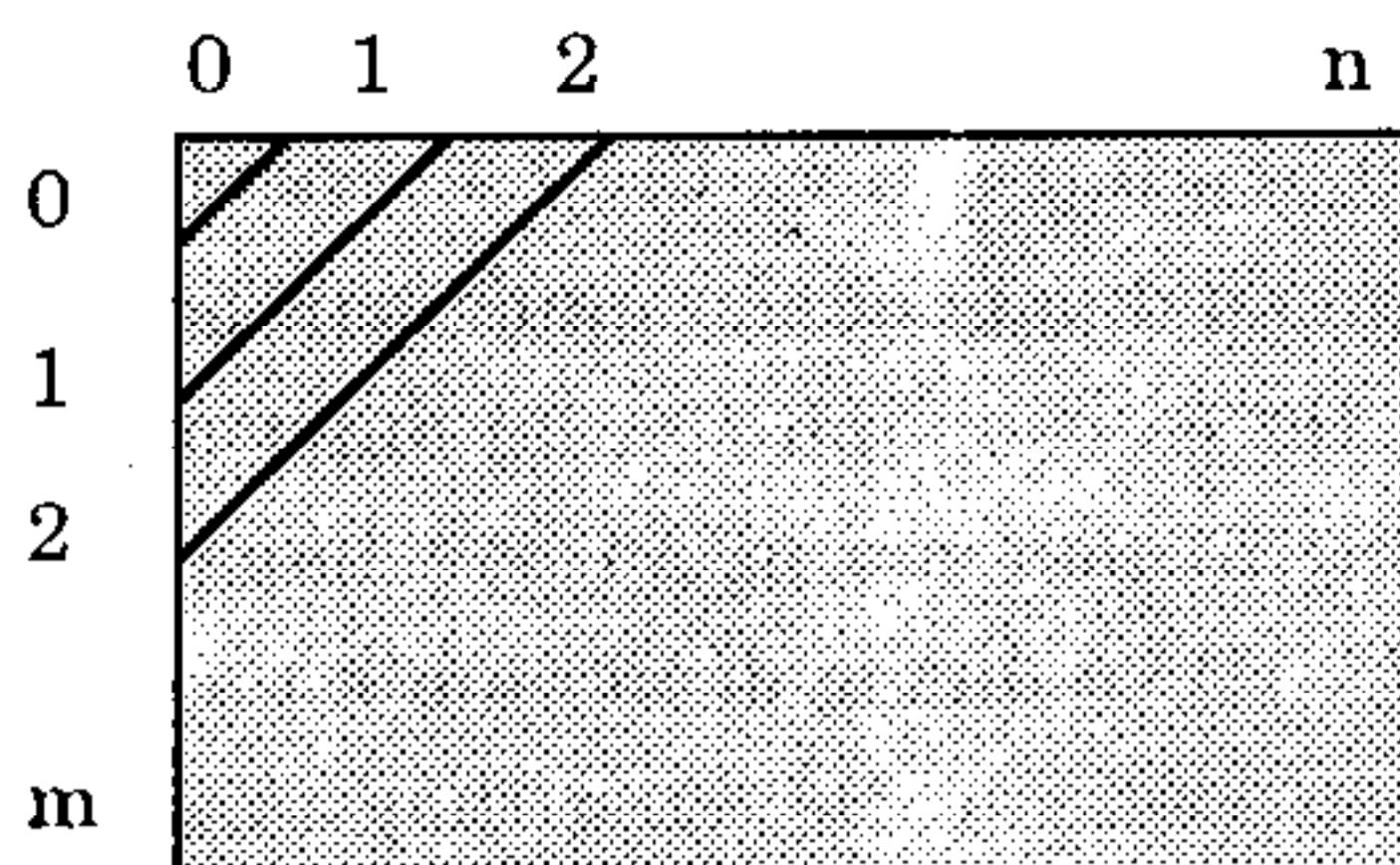
- Nếu  $i > 0$  và  $j > 0$  và  $a_i \neq b_j$  thì

$$L(i, j) = \max \{L(i, j - 1), L(i - 1, j)\}$$

- Nếu  $i > 0$  và  $j > 0$  và  $a_i = b_j$  thì

$$L(i, j) = 1 + L(i - 1, j - 1)$$

Chúng ta sẽ lưu các giá trị  $L(i, j)$  vào mảng  $L[0 .. m, 0 .. n]$ . Từ công thức (2) và (3) ta thấy rằng, nếu biết  $L[i, j - 1]$ ,  $L[i - 1, j]$  và  $L[i - 1, j - 1]$  ta tính ngay được  $L[i, j]$ , do đó ta có thể tính được các phần tử của mảng  $L[0 .. m, 0 .. n]$  từ góc trên bên trái lần lượt theo các đường chéo song song



Bây giờ từ mảng  $L$  đã được làm đầy, ta xây dựng dãy con chung dài nhất là  $k = L[m,n]$ . Ta xác định các thành phần của  $c = (c_1, \dots, c_{k-1}, c_k)$  lần lượt từ bên phải. Trong bảng  $L$  ta đi từ ô  $L[m,n]$ . Giả sử ta đang ở ô  $L[i,j]$  và ta đang cần xác định  $c_l$  ( $1 \leq l \leq k$ ). Nếu  $a_i = b_j$  thì ta lấy  $c_l = a_i$ , giảm 1 đi một và đi lên ô  $L[i-1,j-1]$ . Còn nếu  $a_i \neq b_j$  thì 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  $L[i,j] = L[i-1,j]$ , thì ta đi lên ô  $L[i-1,j]$ . Tiếp tục quá trình trên ta xác định được tất cả các thành phần của dãy con dài nhất  $c$ .

Bài toán này và bài toán chiếc ba lô thuộc lớp các bài toán tối ưu. Quy hoạch động là phương pháp hay được sử dụng để giải các bài toán tối ưu.當然 không phải bài toán tối ưu nào cũng có thể giải bằng quy hoạch động mà chỉ có thể sử dụng kỹ thuật quy hoạch động cho các bài toán thỏa mãn **nguyên lý tối ưu**. Có thể phát biểu **nguyên lý tối ưu** một cách đơn giản như sau: nghiệm tối ưu của một bài toán con bất kỳ (của bài toán) là một sự kết hợp các nghiệm tối ưu của các bài toán con của nó.

#### 8.5.4. Bài toán người bán hàng

Chúng ta trở lại bài toán người bán hàng đã xét ở mục 2.4 và 8.4.2. Trong mục 2.4 ta đã đưa ra một thuật toán được thiết kế dựa trên kỹ thuật tham ăn. Thuật toán này chỉ cho nghiệm gần đúng. Trong mục 8.4.2 ta đã giải bài toán này bằng phương pháp nhánh - và - cận. Trong mục này ta áp dụng phương pháp quy hoạch động để giải bài toán này.

Giả sử  $G = (V, E)$  là đồ thị định hướng với  $V = \{1, 2, \dots, n\}$  và độ dài các cung là  $C[i,j] > 0$  nếu  $(i,j) \in E$  và  $C[i,j] = \infty$  nếu  $(i,j) \notin E$ . Không mất tính tổng quát, ta giả sử đường đi của người bán hàng là đường đi xuất phát từ đỉnh 1 qua tất cả các đỉnh còn lại đúng một lần rồi lại trở về đỉnh 1. Bất kỳ đường đi nào của người bán hàng có thể phân thành: cung  $(1,k)$  với  $k \in V - \{1\}$  và đường đi từ đỉnh  $k$  tới đỉnh 1 qua mỗi đỉnh thuộc  $V - \{1, k\}$  đúng một lần. Để dễ dàng thấy rằng, nếu đường đi của người bán hàng là ngắn nhất thì đường đi từ  $k$  tới 1 qua các đỉnh thuộc  $V - \{1, k\}$  phải ngắn nhất. Do đó nguyên lý tối ưu được thoả mãn. Gọi  $d(i,S)$  là độ dài đường đi ngắn nhất từ đỉnh  $i$  tới đỉnh 1 qua mọi đỉnh  $k \in S$  đúng một lần. Ta có công thức sau

$$d(i,S) = \min_{k \in S} (C[i,k] + d(k, S - \{k\})) \quad (1)$$

Nghiệm tối ưu cần tìm là  $d(1, V - \{1\})$  và

$$d(1, V - \{1\}) = \min_{2 \leq k \leq n} (C[1,k] + d(k, V - \{1,k\})) \quad (2)$$

Như vậy chúng ta sẽ tính được  $d(1, V - \{1\})$  nếu ta biết được  $d(k, V - \{1,k\})$  với mọi  $k = 2, \dots, n$ .

Rõ ràng là  $d(i, \emptyset) = C[i,1]$ ,  $2 \leq i \leq n$ . Sử dụng (1) ta tính được  $d(i,S)$  với tất cả  $S$  chỉ chứa 1 đỉnh. Từ đó ta tính được  $d(i,S)$  với  $S$  chỉ chứa 2 đỉnh. Tiếp tục ta tính được  $d(i,S)$  với  $S$  chứa 3 đỉnh, 4 đỉnh, ... , cho tới khi ta tính được các  $d(k, V - \{1,k\})$  với  $k = 2, \dots, n$ . Từ đó, theo (2) ta tính được độ dài đường đi ngắn nhất của người bán hàng.

**Ví dụ.** Xét đồ thị định hướng có 4 đỉnh và độ dài các cung được cho bởi ma trận  $C$  như sau

$$\begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix}$$

Ta có

$$d(2, \emptyset) = C[2,1] = 5$$

$$d(3, \emptyset) = C[3,1] = 6$$

$$d(4, \emptyset) = C[4,1] = 8$$

Sử dụng (2) ta tính được

$$d(2,\{3\}) = C[2,3] + d(3, \emptyset) = 15$$

$$d(2,\{4\}) = 18$$

$$d(3,\{2\}) = 18$$

$$d(3,\{4\}) = 20$$

$$d(4,\{2\}) = 13$$

$$d(4,\{3\}) = 15$$

Tiếp theo ta tính được  $d(i,S)$  với  $i \neq 1$ , và  $S$  gồm hai đỉnh khác 1 và  $i$ .

$$d(2,\{3,4\}) = \min \{ C[2,3] + d(3,\{4\}), C[2,4] + d(4,\{3\}) \} = 25$$

$$d(3,\{2,4\}) = \min \{ C[3,2] + d(2,\{4\}), C[3,4] + d(4,\{2\}) \} = 25$$

$$d(4,\{2,3\}) = \min \{ C[4,2] + d(2,\{3\}), C[4,3] + d(3,\{2\}) \} = 25$$

Cuối cùng ta có

$$\begin{aligned} d(1,\{2,3,4\}) &= \min \{ C[1,2] + d(2,\{3,4\}), C[1,3] + d(3,\{2,4\}), \\ &\quad C[1,4] + d(4,\{2,3\}) \} \\ &= \min \{ 35, 40, 43 \} = 35 \end{aligned}$$

Như vậy, đường đi ngắn nhất của người bán hàng có độ dài 35. Đường đi ngắn nhất này có thể xây dựng được bằng cách với mỗi  $d(i,S)$  ta lưu lại đỉnh  $j$  mà tại đó vế phải của (1) đạt min. Gọi  $J(i,S)$  là đỉnh  $j$  này. Chẳng hạn, từ các kết quả tính toán trên ta có

$$J(1,\{2,3,4\}) = 2$$

$$J(2,\{3,4\}) = 4$$

$$J(4,\{3\}) = 3$$

Do đó đường đi ngắn nhất là (1, 2, 4, 3, 1).

Chúng ta thử đánh giá thời gian thực hiện thuật toán này. Gọi N là số các  $d(i, S)$  cần phải tính. Trước hết có  $n - 1$  khả năng chọn i (i chạy từ 2 tới n). Với mỗi i số các tập S có k phần tử, S không chứa 1 và i, là  $C_{n-2}^k$ . Do đó

$$N = \sum_{k=0}^{n-2} (n - 1) C_{n-2}^k = (n - 1) 2^{n-2}$$

Mặt khác để tính  $d(i, S)$  với S gồm k phần tử, theo công thức (1) ta phải thực hiện  $k - 1$  phép so sánh để tìm min. Do đó thuật toán đòi hỏi thời gian  $n^2 2^n$ . So sánh với phương pháp vét cạn  $n!$  đường đi để tìm ra đường đi ngắn nhất, thuật toán này tốt hơn. Bạn đọc hãy tự cài đặt thuật toán này (hãy cho chạy chương trình với n nhỏ).

## *Chương 9*

# SẮP XẾP

### 9.1. VẤN ĐỀ 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 một vai trò rất quan trọng trong việc tìm kiếm dữ liệu. Chẳng hạn, chúng ta thử hình dung xem một cuốn từ điển, nếu các từ không được sắp xếp theo thứ tự mà người ta vẫn thường làm, sẽ khó khăn như thế nào trong việc tra cứu các từ. Trong việc thiết kế các thuật toán, chúng 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 trật tự nào đó.

Người ta chia các phương pháp sắp xếp thành hai lớp: *sắp xếp trong* và *sắp xếp ngoài*. Sắp xếp trong được thực hiện ở bộ nhớ trong khi 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 mà các đối tượng cần sắp xếp quá lớn cần phải lưu ở bộ nhớ ngoài, dưới dạng các file, ta cần sử dụng phương pháp sắp xếp ngoài, hay *sắp xếp file*. Trong chương này ta chỉ đề cập đến sắp xếp trong (sắp xếp mảng).

Từ đây về sau, chúng ta giả sử rằng các đối tượng cần sắp xếp được biểu diễn bởi bản ghi gồm một số trường. Một trong các trường đó được gọi là *khoá sắp xếp*. Kiểu của khoá là kiểu có thứ tự tuyến tính (chẳng hạn, kiểu nguyên, kiểu thực ...).

```
type Object = record
    key : keytype;
    [các trường khác]
end;
var A: array[1 .. n] of object;
```

Vấn đề sắp xếp trong được phát biểu như sau: Cho mảng  $A[1 .. n]$  các đối tượng, chúng ta cần sắp xếp lại các thành phần của mảng  $A$  để nhận được mảng  $A$  mới với các thành phần có các giá trị khoá tăng dần:

$$A[1].key \leq A[2].key \leq \dots \leq A[n].key$$

Có nhiều thuật toán sắp xếp mảng. Đầu tiên chúng ta trình bày các phương pháp sắp xếp đơn giản: sắp xếp bằng lựa chọn (selectionsort), sắp xếp bằng xen vào (insertionsort), sắp xếp nổi bọt (bubblesort). Các thuật toán này đòi hỏi thời gian  $O(n^2)$  để sắp xếp một mảng chứa  $n$  đối tượng. Sau đó chúng ta sẽ nghiên cứu các thuật toán sắp xếp hiệu quả hơn. Thuật toán được xem là hiệu quả nhất là thuật toán sắp xếp nhanh (quicksort), nó đòi hỏi thời gian trung bình là  $O(n\log n)$ . Ngoài ra còn có một số thuật toán khác, sắp xếp bằng cây thứ tự bộ phận (heapsort), sắp xếp hoà nhập (mergesort), cũng chỉ đòi hỏi thời gian trung bình  $O(n\log n)$ .

## 9.2. CÁC PHƯƠNG PHÁP SẮP XẾP ĐƠN GIẢN

### 9.2.1. Sắp xếp bằng lựa chọn (Selection Sort)

Tư tưởng của phương pháp sắp xếp bằng lựa chọn là như sau: Ta tìm thành phần có khoá nhỏ nhất trong mảng  $A[1 .. n]$ . Giả sử đó là  $A[k]$ . Trao đổi  $A[k]$  với  $A[1]$ . Như vậy  $A[1]$  trở thành phần tử có khoá nhỏ nhất trong mảng. Giả sử ta đã có  $A[1].key \leq \dots \leq A[i - 1].key$ . Bây giờ ta tìm thành phần có khoá nhỏ nhất trong mảng con  $A[i .. n]$ . Giả sử thành phần tìm được là  $A[k]$ ,  $i \leq k \leq n$ . Lại trao đổi  $A[k]$  với  $A[i]$ , ta nhận được  $A[1].key \leq \dots \leq A[i].key$ . Lặp lại quá trình trên cho tới khi  $i = n - 1$ , ta sẽ nhận được mảng  $A$  được sắp xếp.

**Ví dụ.** Xét mảng  $A[1 .. 6]$  các số nguyên. Kết quả các bước sắp xếp được cho bởi bảng sau:

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$		
5	9	1	8	3	7	$i = 1$	$k = 3$
1	9	5	8	3	7	$i = 2$	$k = 5$
1	3	5	8	9	7	$i = 3$	$k = 3$
1	3	5	8	9	7	$i = 4$	$k = 6$
1	3	5	7	9	8	$i = 5$	$k = 6$
1	3	5	7	8	9		

Sau đây là thủ tục sắp xếp bằng lựa chọn.

```

procedure SelectionSort;
var
    i, j, k: integer;
begin
    (1) for i := 1 to n - 1 do
        begin
            (2)     k := i;
            (3)     for j := i + 1 to n do
                (4)         if A[j].key < A[k].key then k := j;
            (5)         swap(A[i], A[k]);
        end;
    end;

```

Trong thủ tục trên (và trong nhiều thuật toán được trình bày sau này) ta sử dụng thủ tục Swap để trao đổi giá trị của hai biến.

```

procedure Swap(var x, y:item);
var a: item;
begin
    a := x;
    x := y;
    y := a;
end;

```

### *Phân tích sắp xếp bằng lựa chọn*

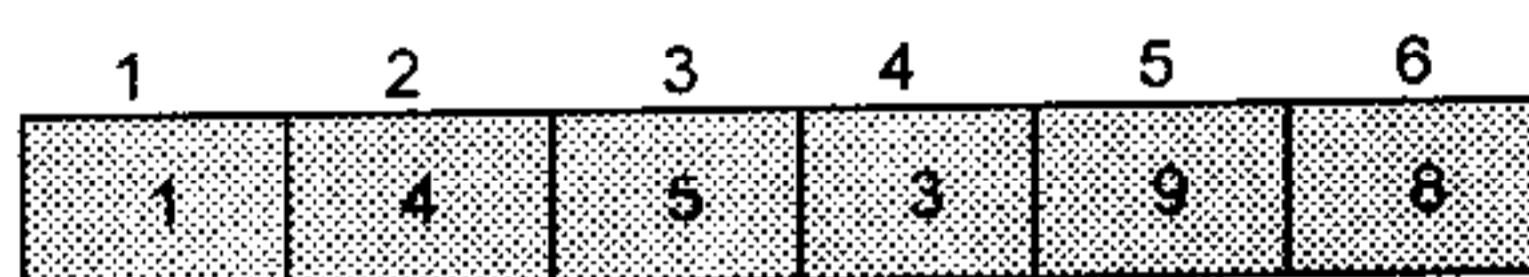
Thời gian thực hiện các lệnh (2), (4) và (5) là  $O(1)$ . Thời gian thực hiện vòng lặp (3) sẽ là  $(n - i)O(1)$ . Do đó thời gian thực hiện thân của vòng lặp (1) (các lệnh (2) - (5)) cũng là  $(n - i)O(1)$ . Như vậy, thời gian thực hiện vòng lặp (1) sẽ là  $\sum_{i=1}^{n-1} (n - i)O(1) = O(1) \cdot \sum_{i=1}^{n-1} (n - i) = O(1)n(n - 1)/2 \leq O(n^2)$

### 9.2.2. Sắp xếp bằng xen vào (InsertionSort)

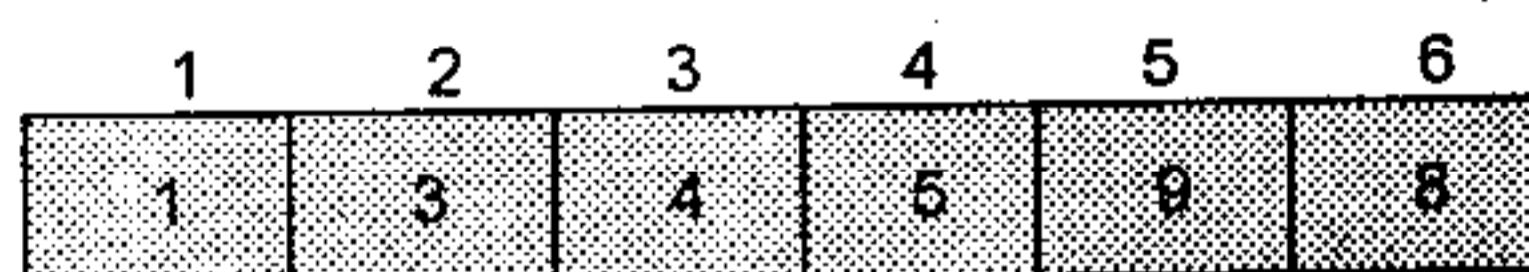
Phương pháp này có thể mô tả như sau: Giả sử đoạn đầu của mảng  $A[1..i - 1]$  ( $i \geq 2$ ) đã được sắp xếp, tức là ta có  $A[1].key \leq \dots \leq A[i - 1].key$ . Xen  $A[i]$  vào vị trí thích hợp trong đoạn đầu  $A[1 .. i - 1]$  để nhận được đoạn đầu  $A[1 .. i]$  được sắp xếp. Lặp lại quá trình xen  $A[i]$  như thế với  $i$  chạy từ 2 đến  $n$ , ta sẽ nhận được toàn bộ mảng  $A[1 .. n]$  được sắp.

Việc tìm vị trí thích hợp để xen  $A[i]$  vào được thực hiện như sau: Đầu tiên là lưu  $A[i]$  vào biến  $x$ ,  $x := A[i]$ . Sau đó ta cho biến  $j$  chạy từ  $i - 1$ , nếu  $x.key < A[j].key$  thì ta sẽ đẩy  $A[j]$  ra sau một vị trí và giảm  $j$ ,  $j := j - 1$ . Quá trình sẽ tiếp tục cho đến khi ta có  $x.key \geq A[j].key$  hoặc  $j = 0$ . Sau đó ta đặt  $x$  vào vị trí  $j + 1$  của mảng.

**Ví dụ.** Giả sử ta có mảng các số nguyên  $A[1 .. 6]$ , trong đó  $A[1 .. 3]$  đã được sắp xếp



Vì  $A[4] = 3 < A[3] = 5$  và  $A[4] < A[2] = 4$ , ta cần đẩy 5 rồi 4 ra sau một vị trí, rồi xen 3 vào vị trí thứ 2 của mảng. Kết quả ta nhận được đoạn đầu  $A[1 .. 4]$  được sắp



```

procedure InsertionSort;
var i, j: integer;
    x: object;
    ok: boolean;
begin
    (1) for i := 2 to n do
        begin
            (2)     j := i - 1; x := A[i]; ok := false;
            (3)     while j >= 1 and not ok do
                    begin
                        if A[j].key > x.key then
                            A[j+1] := A[j];
                        else
                            ok := true;
                        j := j - 1;
                    end;
            A[j+1] := x;
        end;
end;

```

```

(4)      if x.key < A[j].key then
            begin
                A[j + 1] := A[j];
                j := j - 1;
            end
        else ok := true;
(5)      A[j + 1] := x;
            end;
        end;
    
```

### *Phân tích sắp xếp bằng xen vào*

Thân lệnh lặp (3) là lệnh (4). Thời gian thực hiện lệnh (4) là  $O(1)$ . Số lần lặp nhiều nhất là  $i - 1$ , do đó lệnh (3) đòi hỏi thời gian  $O(i)$ . Thời gian thực hiện các lệnh (2) và (5) là  $O(1)$ . Do đó thân của lệnh lặp (1), các lệnh (2) - (5), cần thời gian  $O(i)$ . Lệnh lặp (1) đòi hỏi thời gian nhiều nhất là  $c \sum_{i=2}^n i$  ( $c$  là hằng nào đó), tức là  $O(n^2)$ .

### **9.2.3. Sắp xếp nổi bọt (BubbleSort)**

Ý tưởng của sắp xếp nổi bọt là như sau: Ta đi từ trái sang phải mảng, khi gặp hai thành phần kề nhau mà không đúng trật tự (khoá của thành phần đi trước lớn hơn khoá của thành phần đi sau), thì ta trao đổi hai thành phần này. Sau quá trình này ta sẽ có  $A[n]$  là thành phần có khoá lớn nhất trong mảng  $A[1 .. n]$ .

**Ví dụ.** Xét mảng số nguyên  $A[1 .. 5] = (6, 1, 7, 3, 5)$ . Kết quả của quá trình trên được cho bởi bảng sau:

1	2	3	4	5	
6	1	7	3	5	trao đổi $A[1]$ và $A[2]$
1	6	7	3	5	trao đổi $A[3]$ và $A[4]$
1	6	3	7	5	trao đổi $A[4]$ và $A[5]$
1	6	3	5	7	

Tiếp theo ta lại đi từ  $A[1]$  tới  $A[n - 1]$  và trao đổi hai thành phần kề nhau không đúng trật tự, ta sẽ nhận được  $A[n - 1]$  là thành phần có khoá lớn nhất trong mảng  $A[1 .. n - 1]$  và  $A[n - 1].key \leq A[n].key$ . Lặp lại quá trình trên với các đoạn đầu của mảng  $A[1 .. i]$ ,  $i$  chạy từ  $n$  giảm tới 2, ta sẽ nhận được mảng  $A[1 .. n]$  được sắp xếp theo thứ tự khoá tăng dần. Nói một cách hình ảnh thì quá trình trên là quá trình làm “nổi lên” phần tử có khoá lớn nhất trong  $A[1 .. i]$ , với  $i$  chạy từ  $n$  tới 2.

```
procedure BubbleSort;
  var i, j: integer;
  begin
    (1) for i := n downto 2 do
      (2) for j := 1 to i - 1 do
        if A[j].key > A[j + 1].key then Swap(A[j], A[j + 1]);
  end;
```

Rõ ràng là, nếu tới bước thứ  $i$  nào đó hai thành phần kề nhau trong mảng  $A[1 .. i]$  đã đúng trật tự, thì ta có thể dừng lại. Do đó ta có thể cải tiến thủ tục trên bằng cách đưa vào biến boolean  $ok$ , biến này nhận giá trị true nếu hai thành phần kề nhau trong mảng  $A[1 .. i]$  đã đúng trật tự và nhận giá trị false nếu ngược lại. Biến  $ok$  được dùng để điều khiển vòng lặp **while** thay cho vòng lặp **for** (1) ở thủ tục trên. Ta có thủ tục cải tiến sau:

```
procedure BubbleSort;
  var i, j: integer;
  ok: boolean;
  begin
    i := n;
    ok := false;
    while (i >= 2) and (not ok) do
      begin
        ok := true;
        . . .
      end;
    . . .
  end;
```

```

for j := 1 to i - 1 do
    if A[j].key > A[j + 1].key then
        begin
            Swap(A[j],A[j + 1]);
            ok := false;
        end;
    i := i - 1;
end;
end;

```

### ***Phân tích sắp xếp nổi bọt***

Trong thủ tục trên lệnh **if** đòi hỏi thời gian  $O(1)$ . Số lần lặp của lệnh **for** là  $i - 1$ , do đó thời gian thực hiện lệnh **for** là  $O(i)$ . Vì  $i$  chạy từ  $n$ , cùng lăm là giảm tới 2, do đó lệnh **while** đòi hỏi thời gian cùng lăm là  $c \sum_{i=2}^n i$  ( $c$  là hằng nào đó), tức là  $O(n^2)$ . Vậy thời gian thực hiện sắp xếp nổi bọt cũng là  $O(n^2)$ .

Chúng ta thấy rằng cả ba phương pháp sắp xếp trên đều đòi hỏi thời gian  $O(n^2)$  để sắp xếp một mảng  $n$  thành phần.

## **9.3. SẮP XẾP NHANH**

Thuật toán sắp xếp được đưa ra bởi Hoare, nổi tiếng với tên gọi sắp xếp nhanh (quicksort), được thiết kế dựa trên chiến lược chia - để - trị. Bước đầu tiên, thuật toán chọn một trong các thành phần của mảng cần sắp xếp để làm mốc. Sau đó mảng được phân hoạch thành hai phần bằng cách chuyển tất cả các thành phần có khoá lớn hơn khoá của mốc sang bên phải mốc, còn tất cả các thành phần có khoá nhỏ hơn hoặc bằng khoá của mốc được chuyển sang bên trái mốc. Kết quả của phân hoạch, mốc đứng ở vị trí thứ  $k$  và mọi thành phần của mảng con bên trái mốc  $A[1 .. k - 1]$  có khoá nhỏ hơn hoặc bằng khoá của mốc, còn mọi thành phần của mảng con bên phải mốc  $A[k + 1 .. n]$  có khoá lớn hơn

khoá của mốc. Vậy giờ hai mảng con  $A[1 .. k - 1]$  và  $A[k + 1 .. n]$  được sắp xếp độc lập bằng cách gọi đệ quy thuật toán trên.

Sau đây là thuật toán sắp xếp nhanh. Đó là thuật toán đệ quy sắp xếp mảng  $A[i .. j]$ .

```

procedure QuickSort(i, j: integer);
var k: integer;
begin
  if i < j then
    begin
      Partition(i, j, k);
      {Phân hoạch A[i .. j] thành hai mảng con A[i .. k - 1] và A[k + 1 .. j]}
      Quicksort(i, k - 1);
      Quicksort(k + 1, j);
    end;
end;

```

Vấn đề còn lại bây giờ là xây dựng thủ tục phân hoạch (Partition). Trước hết, mốc được chọn như thế nào? Dường nhiên ta mong muốn chọn được phần tử làm mốc sao cho kết quả phân hoạch ta nhận được hai mảng con cân bằng nhau. Điều này là có thể làm được, tuy nhiên, nó tiêu tốn nhiều thời gian hơn sự cần thiết. Vì vậy, người ta thường chọn phần tử đầu tiên của mảng làm mốc, tức là lấy  $p = A[i]$  làm mốc. Sau đó ta sử dụng hai biến, biến  $l$  chạy từ trái sang phải bắt đầu từ  $i$ , còn biến  $k$  chạy từ phải sang trái bắt đầu từ  $j + 1$ . Biến  $l$  được tăng cho tới khi  $A[l].key > p.key$ , còn biến  $k$  được giảm cho tới khi  $A[k] \leq p.key$ . Vậy giờ nếu  $l < k$  thì ta trao đổi  $A[l]$  và  $A[k]$ . Quá trình trên được lặp lại cho tới khi nào  $l > k$ . Cuối cùng ta cần trao đổi  $A[i]$  và  $A[k]$  để đặt mốc vào đúng vị trí của nó.

Chúng ta có thủ tục phân hoạch sau.

```

procedure Partition (i, j: integer; var k: integer);
var l: integer;
    p: object;
begin
    p := A[i];
    l := i;
    k := j + 1;
    repeat l := l + 1 until (A[l].key > p.key) or (l > j);
    repeat k := k - 1 until A[k].key <= p.key;
    while l < k do
        begin
            Swap (A[l],A[k]);
            repeat l := l + 1 until A[l].key > p.key;
            repeat k := k - 1 until A[k].key <= p.key;
        end;
    Swap (A[i],A[k]);
end;

```

*Ví dụ.* Giả sử ta cần phân hoạch mảng các số nguyên A[1 .. 8] như sau

1	2	3	4	5	6	7	8
8	13	2	9	4	1	3	12

Lấy mốc  $p = [A1] = 8$ ,  $l = 1$ ,  $k = 9$ . Tăng  $l$  ( $l := l+1$ ) và giảm  $k$  ( $k := k-1$ ) cho tới khi  $A[l] > 8$  và  $A[k] \leq 8$ , ta có  $l = 2$  và  $k = 7$ . Vì  $l < k$ , trao đổi  $A[2]$  và  $A[7]$  ta được bảng sau

A horizontal array of 8 boxes, each containing a number. Above the array are indices 1 through 8. Two arrows point upwards from below to the 1st and 7th elements.

1	2	3	4	5	6	7	8
8	3	2	9	4	1	13	12

Tiếp tục tăng  $l$  và giảm  $k$  cho tới khi  $A[l] > 8$  và  $A[k] \leq 8$ , ta nhận được  $l = 4$ ,  $k = 6$ . Vì  $l < k$ , trao đổi  $A[4]$  và  $A[6]$  ta được mảng sau

1	2	3	4	5	6	7	8
8	3	2	1	4	9	13	12
↑ l				↑ k			

Lại tiếp tục tăng l và giảm k cho tới khi  $A[l] > 8$  và  $A[k] \leq 8$ , ta có  $l = 6$ ,  $k = 5$ . Đến đây, vì  $l > k$ , trao đổi  $A[1]$  và  $A[5]$  để đặt mốc  $p = A[1]$  vào vị trí  $k = 5$  của mảng, ta được mảng sau

1	2	3	4	5	6	7	8
4	3	2	1	8	9	13	12
↑ k				↑ l			

Như vậy, ta nhận được  $k = 5$ , và mảng  $A[1 .. 8]$  được phân hoạch thành 2 mảng con  $A[1 .. 4]$  và  $A[6 .. 8]$ .

### Phân tích sắp xếp nhanh

Thời gian thực hiện thủ tục phân hoạch **Partition** là thời gian đi qua mảng (hai biến k và l chạy từ hai đầu cho tới khi chúng gặp nhau), và so sánh khoá của mỗi thành phần mảng với khoá của mốc. Do đó để phân hoạch mảng n phần tử ta cần thời gian  $O(n)$ .

Trong trường hợp xấu nhất, sau mỗi lần phân hoạch mảng n phần tử, ta nhận được mảng con  $n - 1$  phần tử ở một phía của mốc, còn bên kia không có phần tử nào (đó là trường hợp mảng đã được sắp xếp theo thứ tự không giảm hoặc không tăng). Trong trường hợp này, nếu gọi  $T(n)$  là thời gian thực hiện **Quicksort**, chúng ta có quan hệ đệ quy sau:

$$T(n) = \begin{cases} O(1) & \text{nếu } n=1 \\ O(n) + T(n-1) & \text{nếu } n>1 \end{cases}$$

Bằng cách thế liên tiếp, ta nhận được:

$$\begin{aligned} T(n) &= O(n) + T(n-1) \\ &= O(n) + O(n-1) + T(n-2) \end{aligned}$$

.....

$$= \sum_{i=1}^n O(i) = O(n^2).$$

Như vậy, trong trường hợp xấu nhất **Quicksort** đòi hỏi thời gian  $O(n^2)$ .

Bây giờ ta xác định thời gian trung bình mà **Quicksort** đòi hỏi để sắp xếp một mảng n phần tử. Gọi  $T_{tb}(n)$  là thời gian đó. Giả sử mảng A ban đầu chứa n đối tượng, được đưa vào một cách ngẫu nhiên. Khi ta gọi thủ tục **Partition**(1,n,k) để phân hoạch mảng A[1 .. n] thành hai mảng con A[1 .. k - 1] và A[k + 1 .. n] thì k có thể nhận một trong các giá trị từ 1 tới n với xác suất bằng nhau và bằng  $1/n$ . Vì thời gian thực hiện phân hoạch là  $O(n)$ , từ thủ tục **Quicksort**(1,n), ta nhận được quan hệ đệ quy sau:

$$T_{tb}(n) = O(n) + \frac{1}{n} \sum_{k=1}^n [T_{tb}(k - 1) + T_{tb}(n - k)]$$

$$\text{hay } T_{tb}(n) = cn + \frac{1}{n} \sum_{k=1}^n [T_{tb}(k - 1) + T_{tb}(n - k)] \quad (1)$$

trong đó c là hằng số dương nào đó.

Chú ý rằng  $\sum_{k=1}^n T_{tb}(n - k) = \sum_{k=1}^n T_{tb}(k - 1)$ , do đó ta có thể viết lại (1) như sau.

$$T_{tb}(n) = cn + \frac{2}{n} \sum_{k=1}^n T_{tb}(k - 1) \quad (2)$$

Trong (2) thay n bởi n - 1, ta có

$$T_{tb}(n - 1) = c(n - 1) + \frac{2}{n - 1} \sum_{k=1}^{n-1} T_{tb}(k - 1) \quad (3)$$

Nhân (2) với n, nhân (3) với n - 1 và trừ cho nhau ta nhận được

$$nT_{tb}(n) = c(2n - 1) + (n + 1)T_{tb}(n - 1)$$

Chia cho  $n(n + 1)$ , ta nhận được quan hệ đệ quy sau

$$\frac{T_{tb}(n)}{n+1} = \frac{c(2n-1)}{n(n+1)} + \frac{T_{tb}(n-1)}{n} \quad (4)$$

Để giải (4), áp dụng phép thế liên tiếp ta có

$$\begin{aligned}
 \frac{T_{tb}(n)}{n+1} &= \frac{c(2n-1)}{n(n+1)} + \frac{T_{tb}(n-1)}{n} \\
 &= \frac{c(2n-1)}{n(n+1)} + \frac{c(2(n-1)-1)}{(n-1)n} + \frac{T_{tb}(n-2)}{n-1} \\
 &\quad \dots \\
 \frac{T_{tb}(n)}{n+1} &= \sum_{k=1}^n \frac{2i-1}{i(i+1)} + \frac{T_{tb}(1)}{2}
 \end{aligned} \tag{5}$$

Ta có đánh giá

$$\sum_{i=2}^n \frac{2i-1}{i(i+1)} \leq \sum_{i=2}^n \frac{2}{i} \leq 2 \int_2^n \frac{dx}{x} \leq 2\log n$$

Như vậy

$$\frac{T_{tb}(n)}{n+1} = O(\log n)$$

hay  $T_{tb}(n) = O(n \log n)$

*Quicksort* là thuật toán được thiết kế dựa trên kỹ thuật chia - để - trị. Còn một thuật toán sắp xếp khác cũng được thiết kế dựa trên kỹ thuật chia - để - trị, đó là thuật toán sắp xếp hoà nhập (*mergesort*). Để sắp xếp mảng  $A[i .. j]$  ta quy về việc sắp xếp hai mảng con  $A[i .. k]$  và  $A[k+1 .. j]$ , với  $k$  nằm giữa  $i$  và  $j$ ,  $k = (i+j) \text{ div } 2$ . Sau đó ta hoà nhập hai mảng con đã được sắp xếp thành mảng được sắp.

```

procedure MergeSort(i, j: integer);
begin
  if i < j then
    begin
      k := (i + j) div 2;
      MergeSort(i, k);
      MergeSort(k + 1, j);
      Merge(i, k, j);
      {thủ tục hoà nhập hai mảng A[i .. k], A[k + 1 .. j]
       đã được sắp xếp thành mảng A[i .. j] được sắp}
    end;
  end;

```

Việc thực hiện hoà nhập hai mảng đã được sắp  $A[i..k]$  và  $A[k+1..j]$  thành mảng  $A[i..j]$  được sắp (thủ tục  $\text{Merge}(i,k,j)$ ) là một bài tập dễ giành cho độc giả.

Thuật toán *MergeSort* cũng chỉ đòi hỏi thời gian trung bình  $O(n \log n)$

#### 9.4. 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 (heap), phương pháp này được gọi là *HeapSort*. Nhắc lại rằng, một cây thứ tự bộ phận có thể biểu diễn bởi mảng  $A[1..n]$  có tính chất  $A[i].key \leq A[2*i].key$  và  $A[i].key \leq A[2*i + 1].key$ , với  $1 \leq i \leq n \text{ div } 2$  (1). Như vậy  $A[1]$  là thành phần có khoá nhỏ nhất trong mảng. Sử dụng tính chất này ta có thể đưa ra thuật toán sắp xếp mảng sau.

Đầu tiên ta biến đổi mảng cần sắp  $A[1..n]$  thành mảng biểu diễn cây thứ tự bộ phận. Sau đó ta trao đổi  $A[1]$  và  $A[n]$ . Mảng  $A[1..n-1]$  bây giờ thoả mãn tính chất (1) với mọi  $i \geq 2$ , trừ  $i = 1$ . Biến đổi mảng  $A[1..n-1]$  để nó thoả mãn tính chất (1). Lại trao đổi  $A[1]$  và  $A[n-1]$ . Lặp lại quá trình trên, cuối cùng ta sẽ nhận được mảng  $A[i..n]$  được sắp xếp theo thứ tự giảm dần,  $A[1].key \geq \dots \geq A[n-1].key \geq A[n].key$ .

Trong quá trình trên, sau mỗi lần trao đổi  $A[1]$  với  $A[r]$  ( $r$  chạy từ  $n$  giảm tới 2), ta nhận được mảng  $A[1..r-1]$  thoả mãn (1) với mọi  $i \geq 2$ , trừ  $i = 1$ . Để điều kiện (1) thoả mãn với mọi  $i$ , ta sử dụng phép toán đẩy  $A[1]$  xuống vị trí thích hợp trong mảng (phép toán Pushdown). Phép toán này đã được chúng ta sử dụng trong thủ tục *DeleteMin* (xem mục 5.7).

Còn một vấn đề cần giải quyết, đó là biến đổi mảng ban đầu  $A[1..n]$  thành mảng biểu diễn cây thứ tự bộ phận như thế nào? Ta có nhận xét rằng, với  $i > n \text{ div } 2$  thì điều kiện (1) được xem là thoả mãn, vì  $2i$  và  $2i+1 > n$ . Như vậy, nếu ta đẩy  $A[i]$  xuống vị trí thích hợp trong mảng  $A[i..n]$ , với  $i$  chạy từ  $n \text{ div } 2$  giảm xuống 1, thì cuối cùng ta sẽ nhận được mảng  $A[1..n]$  thoả mãn (1).

Giả sử rằng mảng  $A[a..b]$  đã thoả mãn điều kiện (1) với mọi  $i \geq a+1$ . Thủ tục *Pushdown*( $a,b$ ) dưới đây thực hiện việc đẩy  $A[a]$  xuống vị trí thích hợp trong mảng  $A[a..b]$ , để điều kiện (1) đúng với mọi  $i \geq a$ .

```

procedure Pushdown (a,b: integer);
var
    i, j: integer;
    ok: boolean;
begin
    i := a;
    ok := false;
    (1) while (i <= b div 2) and not ok do
        begin
            (2) if i = b div 2 then j := 2*i else
                if A[2*i].key < A[2*i + 1].key then j := 2*i else j := 2*i + 1;
            (3) if A[i].key > A[j].key then
                begin
                    Swap(A[i],A[j]);
                    i := j;
                end
            else ok := true;
        end;
    end;

```

Sử dụng thủ tục ***Pushdown***, ta đưa ra thủ tục sắp xếp ***HeapSort*** sau đây. Cần nhớ rằng, kết quả của thủ tục này ta nhận được mảng A[1 .. n] được sắp xếp theo thứ tự khoá giảm dần.

```

procedure HeapSort;
var i: integer;
begin
    for i := n div 2 downto 1 do Pushdown(i,n);
        {Biến đổi mảng A[1 .. n] thành mảng biểu diễn cây thứ tự bộ phận}
    for i := n downto 2 do
        begin
            Swap(A[1],A[i]);
            Pushdown(1,i - 1);
        end;
    end;

```

### Phân tích HeapSort

Trước hết ta đánh giá thời gian thực hiện thủ tục ***Pushdown***. Thân của vòng lặp (1) là các lệnh (2) và (3), mỗi lệnh này cần thời gian  $O(1)$ . Do đó thân của vòng lặp (1) đòi hỏi thời gian  $O(1)$ . Sau mỗi lần lặp, biến  $i$  được tăng lên ít nhất 2 lần, và giá trị ban đầu của  $i$  là  $a$ . Do đó nếu gọi  $k$  là số lần lặp, ta có  $a2^k \leq b$ , hay  $k \leq \log(b/a)$ . Vì vậy thời gian thực hiện thủ tục ***Pushdown***( $a, b$ ) là  $O(\log(b/a))$ .

Xét thủ tục ***HeapSort***. Vòng lặp for đầu tiên có số lần lặp là  $n/2$ . Mỗi lần gọi thủ tục ***Pushdown*** đòi hỏi thời gian nhiều nhất là  $O(n\log n)$ . Do đó thời gian thực hiện vòng lặp for đầu tiên là  $O(n\log n)$ . Tương tự, vòng lặp for thứ hai có số lần lặp là  $n-1$ . Mỗi lần lặp ta cần thực hiện lệnh trao đổi ***Swap*** và gọi thủ tục ***Pushdown***, tất cả cần thời gian nhiều nhất là  $O(n\log n)$ . Do đó vòng lặp for thứ hai đòi hỏi thời gian  $O(n\log n)$ . Vì vậy thời gian thực hiện ***HeapSort*** là tổng thời gian thực hiện hai vòng lặp for, và là  $O(n\log n)$ .

## *Chương 10*

# CÁC THUẬT TOÁN TRÊN ĐỒ THỊ

### 10.1. MỘT SỐ KHÁI NIỆM CƠ BẢN

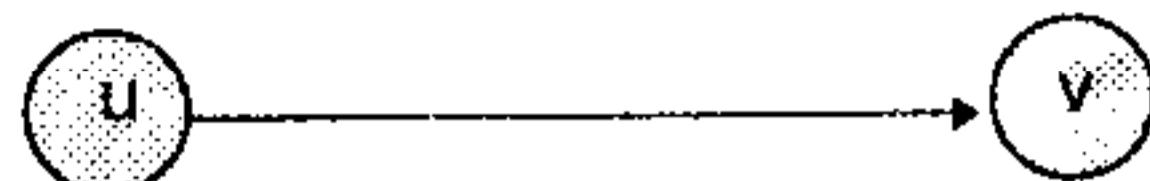
Đồ thị được sử dụng để mô hình hóa các vấn đề bao gồm một tập hữu hạn các đối tượng có quan hệ với nhau theo một cách nào đó. Các vấn đề như thế xuất hiện trong rất nhiều lĩnh vực khác nhau: khoa học máy tính, công nghệ điện, hoá học, chính trị, kinh tế... Chẳng hạn, một mạng truyền thông được mô hình hóa bởi đồ thị như sau: chúng ta biểu diễn các thành viên trong mạng như là các đỉnh của đồ thị, nếu từ thành phần được biểu diễn bởi đỉnh a có thể truyền tin trực tiếp tới thành phần được biểu diễn bởi đỉnh b thì sẽ có cung (a,b) đi từ a tới b. Các mạng giao thông, mạng máy tính... cũng được mô hình hóa một cách tương tự.

Sau khi cấu trúc của vấn đề được biểu diễn bởi đồ thị, việc giải quyết vấn đề đặt ra sẽ được quy về việc giải quyết một vấn đề nào đó trên đồ thị, chẳng hạn như tìm đường đi ngắn nhất, hay là tìm các thành phần liên thông, hay là tìm cây bao trùm ngắn nhất, ... Vì vậy, việc nghiên cứu các thuật toán hiệu quả giải quyết các vấn đề thuộc lý thuyết đồ thị có tầm quan trọng đặc biệt.

Bây giờ chúng ta nhắc lại khái niệm đồ thị. Đồ thị được sử dụng trong các thuật toán sau này là **đồ thị đơn** (đồ thị không có khuyên và không có các cung hoặc cạnh song song). Cụ thể ta có các định nghĩa sau.

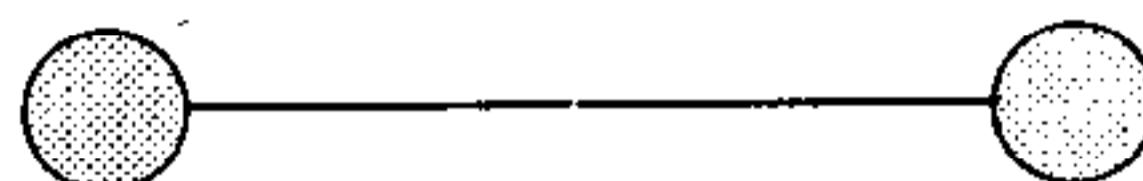
Một **đồ thị định hướng** G gồm một tập hữu hạn V các đỉnh và một tập E các cung,  $G = (V, E)$ . Mỗi cung là một cặp có thứ tự các đỉnh khác nhau  $(u,v)$ , tức là  $(u,v) \neq (v,u)$ . Cung  $(u,v)$  được gọi là cung đi từ đỉnh u

tới đỉnh  $v$ , ký hiệu là  $u \rightarrow v$ , và trong biểu diễn hình học nó được biểu diễn như sau.



Nếu có cung  $u \rightarrow v$ , thì ta nói  $v$  là đỉnh kề với đỉnh  $u$ .

Một **đồ thị vô hướng**  $G = (V, E)$  gồm một tập hữu hạn  $V$  các đỉnh và một tập các cạnh thuộc  $E$ . Khác với đồ thị định hướng, trong đồ thị vô hướng, mỗi cạnh  $E$  là một cặp không có thứ tự các đỉnh khác nhau  $(u, v)$ , tức là  $(u, v) = (v, u)$ . Nếu có cạnh  $(u, v)$  thì ta nói hai đỉnh  $u, v$  kề nhau ( $u$  kề  $v$  và  $v$  kề  $u$ ), và cạnh  $u, v$  được biểu diễn hình học như sau.



Cần lưu ý rằng, một đồ thị vô hướng có thể xem như một đồ thị định hướng bằng cách xem mỗi cạnh  $(u, v)$  như là hai cung  $u \rightarrow v$  và  $v \rightarrow u$ .

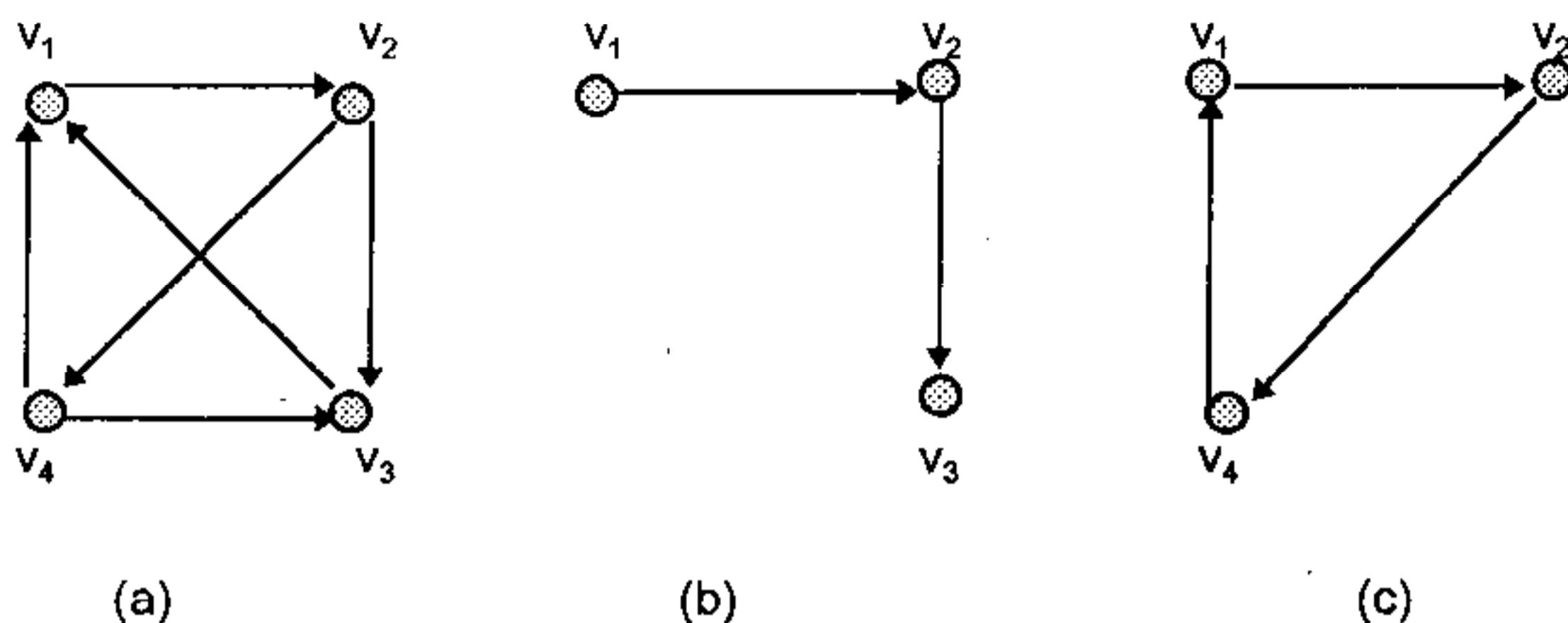
Trong nhiều ứng dụng chúng ta cần phải xét các **đồ thị có trọng số**. Đó là các đồ thị mà mỗi cung (cạnh)  $(u, v)$  được gắn với một số  $c(u, v)$ . Số  $c(u, v)$  được gọi là **trọng số** của cung (cạnh)  $(u, v)$  (nó còn được gọi là **giá** hoặc **độ dài** của cung (cạnh)  $(u, v)$ ).

Từ đây về sau, nếu chỉ nói đồ thị thì ta hiểu nó có thể là đồ thị định hướng hoặc đồ thị vô hướng.

Sau đây ta trình bày một số khái niệm cơ bản trong đồ thị, các khái niệm khác sẽ được đưa ra khi cần thiết. Một **đường đi đơn** (gọi tắt là **đường đi**) trên đồ thị  $G = (V, E)$  là một dãy hữu hạn các đỉnh  $(v_0, v_1, \dots, v_k)$ , trong đó các  $v_0, v_1, \dots, v_k$  là khác nhau, trừ ra có thể  $v_0 = v_k$ , đồng thời  $v_{i+1}$  là đỉnh kề đỉnh  $v_i$ ,  $i = 0, 1, \dots, k-1$ . Trong đồ thị định hướng ta nói nó là đường đi từ  $v_0$  đến  $v_k$ , còn trong đồ thị vô hướng nó được gọi là đường đi giữa  $v_0$  và  $v_k$ . Trong đồ thị không có trọng số, độ dài của đường đi  $(v_0, v_1, \dots, v_k)$  là  $k$ . Còn trong đồ thị có trọng số, độ dài của nó được xem là bằng  $\sum_{i=0}^{k-1} c(v_i, v_{i+1})$ .

Một chu trình là một đường đi khép kín, tức là một đường đi  $(v_0, v_1, \dots, v_k)$  mà  $v_0 = v_k$ . Chẳng hạn, trong đồ thị định hướng được biểu

diễn bởi hình 10.1a,  $(v_4, v_1, v_2, v_3)$  là đường đi từ  $v_4$  đến  $v_3$ , và  $(v_4, v_1, v_2, v_4)$  là một chu trình.



**Hình 10.1.** Đồ thị, đồ thị con và đồ thị con cảm sinh

Đồ thị  $G' = (V', E')$  là đồ thị con của đồ thị  $G = (V, E)$  nếu các đỉnh của  $G'$  là các đỉnh của  $G$  và các cung (cạnh) của  $G'$  là các cung (cạnh) của  $G$ , tức là  $V' \subseteq V$ ,  $E' \subseteq E$ . Chẳng hạn, đồ thị trong hình 10.1b là đồ thị con của đồ thị 10.1a. Nếu  $G' = (V', E')$  là đồ thị con của  $G$  mà  $E'$  gồm tất cả các  $(u, v) \in E$  sao cho cả  $u$  và  $v$  đều nằm trong  $V'$  thì  $G'$  được gọi là đồ thị con cảm sinh bởi tập đỉnh  $V'$ . Ví dụ, đồ thị trong hình 10.1c là đồ thị con của đồ thị 10.1a cảm sinh bởi tập đỉnh  $\{v_1, v_2, v_4\}$ .

## 10.2. BIỂU DIỄN ĐỒ THỊ

Để giải quyết các vấn đề của đồ thị bằng máy tính, chúng ta cần phải lưu giữ đồ thị trong bộ nhớ. Do đó chúng ta cần phải đề cập đến các phương pháp biểu diễn đồ thị bởi các cấu trúc dữ liệu. Có hai cách biểu diễn đồ thị: dùng ma trận kề và dùng danh sách kề.

### 10.2.1. Biểu diễn đồ thị bởi ma trận kề

Đồ thị  $G = (V, E)$  với  $V = \{1, 2, \dots, n\}$  được lưu giữ trong mảng hai chiều  $A[1..n, 1..n]$ , trong đó

$$A[i,j] = \begin{cases} 1 & \text{nếu } (i,j) \in E \\ 0 & \text{nếu } (i,j) \notin E \end{cases}$$

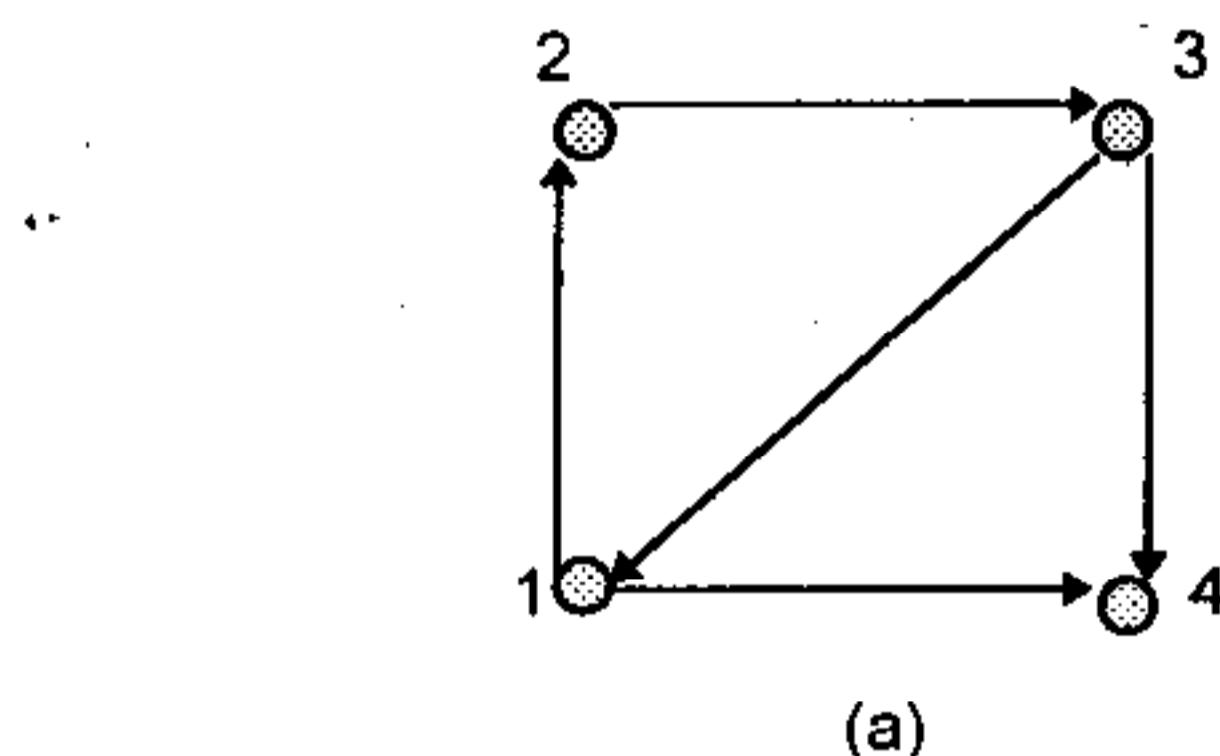
**Ví dụ.** Đồ thị trong hình 10.2a được biểu diễn bởi ma trận kề trong hình 10.2b.

Bằng cách này, đồ thị được mô tả như sau.

```
type graph = array[1..n,1..n] of boolean;
```

Ta cũng có thể dùng ma trận kề  $A[1..n, 1..n]$  để biểu diễn đồ thị có trọng số, trong đó  $A[i,j]$  lưu trọng số của cung  $(i,j)$  nếu  $(i,j) \in E$ , còn nếu  $(i,j) \notin E$  thì  $A[i,j] = \infty$  hoặc  $A[i,j] = 0$  tuỳ thuộc vào ứng dụng cụ thể.

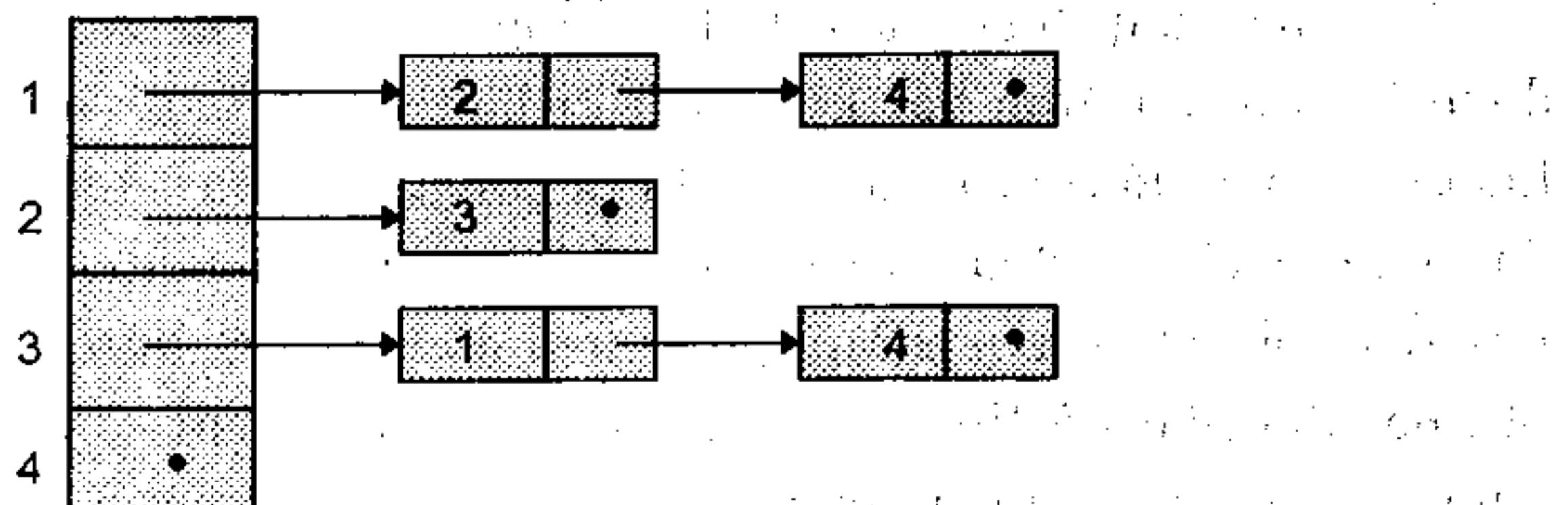
*Nhận xét.* Vì rằng trong đồ thị vô hướng,  $(i,j)$  và  $(j,i)$  biểu diễn cùng một cạnh, do đó ma trận kề của đồ thị vô hướng là ma trận đối xứng qua đường chéo chính.



(a)

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	1	0	0	1
4	0	0	0	0

(b)



(c)

**Hình 10.2.** Biểu diễn đồ thị bởi ma trận kề và danh sách kề

### 10.2.2. Biểu diễn đồ thị bởi danh sách kề

Trong phương pháp này, với mỗi đỉnh ta lập một danh sách các đỉnh kề đỉnh đó. Danh sách các đỉnh kề đỉnh  $i \in V = \{1, 2, \dots, n\}$  được tổ chức dưới dạng danh sách liên kết, mỗi thành phần trong danh sách này chứa số hiệu đỉnh kề đỉnh  $i$ . Chúng ta sử dụng mảng  $A[1..n]$ , trong đó  $A[i]$  lưu giữ con trỏ trỏ tới đầu danh sách các đỉnh kề đỉnh  $i$ .

**Ví dụ.** Hình 10.2c là cấu trúc dữ liệu biểu diễn đồ thị trong hình 10.2a bằng danh sách kề.

Chúng ta cũng có thể sử dụng phương pháp này để biểu diễn đồ thị có trọng số. Muốn vậy, ta chỉ cần đưa thêm vào mỗi thành phần trong danh sách kề một trường lưu trọng số. Cụ thể, ta có thể mô tả cấu trúc dữ liệu biểu diễn đồ thị bằng danh sách kề như sau

```

type pointer = ^member;
member = record
    number :1..n;{số hiệu đỉnh kề}
    cost : real; {giá của cung}
    next : pointer;
end;
graph = array[1..n] of pointer;

```

#### *Phân tích hai phương pháp biểu diễn đồ thị*

Phương pháp biểu diễn đồ thị bằng ma trận kề  $A[1..n, 1..n]$  có ưu điểm là, với hai đỉnh bất kỳ  $i$  và  $j$  ta biết ngay được có cung  $(i, j)$  hay không và nếu đồ thị có trọng số thì ta cũng biết ngay được giá của cung đó, bằng cách truy cập tới thành phần  $A[i, j]$  của mảng. Phương pháp này đòi hỏi mảng  $n \times n$  thành phần nếu đồ thị có  $n$  đỉnh. Do đó sẽ lãng phí bộ nhớ nếu đồ thị có số đỉnh  $n$  lớn, nhưng 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, phương pháp này có nhược điểm ở chỗ, muốn biết có cung  $(i, j)$  hay không và giá của nó bao nhiêu, ta lại phải tốn thời gian để duyệt danh sách các đỉnh kề đỉnh  $i$ .

### 10.3. ĐI QUA ĐỒ THỊ

Việc giải quyết rất nhiều vấn đề hay của lý thuyết đồ thị đòi hỏi ta phải xem xét tất cả các đỉnh và tất cả các cung (cạnh) của đồ thị theo một hệ thống nào đó. Vì vậy các kỹ thuật đi qua đồ thị để thăm tất cả các đỉnh và các cung của đồ thị đóng vai trò quan trọng trong việc thiết kế các thuật toán trên đồ thị.

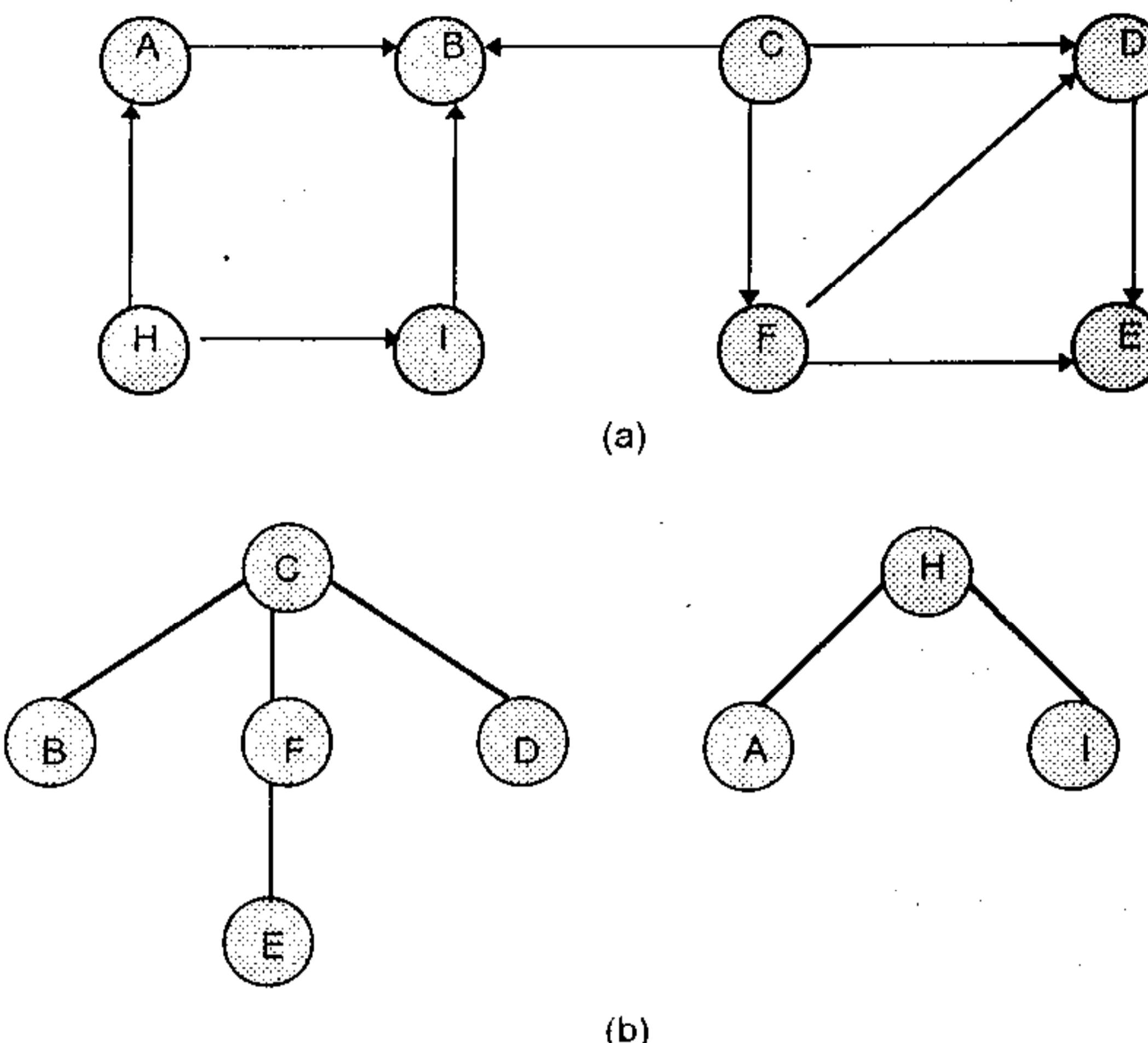
Có hai kỹ thuật đi qua đồ thị. Đó là đi qua đồ thị theo bề rộng và theo độ sâu. Cả hai đều là sự tổng quát hoá của các kỹ thuật đi qua cây theo bề rộng và theo độ sâu.

#### 10.3.1. Đi qua đồ thị theo bề rộng

Tư tưởng của phương pháp đi qua đồ thị theo bề rộng là như sau: Chọn một đỉnh  $u$  và thăm đỉnh đó. Từ đỉnh  $u$  đã được thăm, ta sẽ lần lượt đến thăm tất cả các đỉnh  $v$  chưa được thăm kề đỉnh  $u$ . Sau đó đỉnh  $v$  nào được thăm trước thì các đỉnh kề nó cũng sẽ được thăm trước. Quá trình trên sẽ tiếp tục cho tới khi không thể đến thăm đỉnh nào nữa. Bay giờ nếu còn đỉnh chưa được thăm thì ta chọn ra một đỉnh để thăm đỉnh đó và lặp lại quá trình trên. Tiếp tục cho tới khi nào tất cả các đỉnh của đồ thị đã được thăm.

Quá trình đi thăm các đỉnh của đồ thị xuất phát từ đỉnh  $u$  sẽ tạo ra một cây gốc  $u$ , trong đó nếu  $w$  đã được thăm từ đỉnh  $v$  đi tới thì  $w$  là đỉnh con của đỉnh  $v$  trong cây. Khi hoàn thành việc đi qua đồ thị, một rừng cây được tạo thành.

**Ví dụ.** Nếu ta đi qua đồ thị trong hình 10.3a thì một rừng cây trong hình 10.3b có thể được tạo thành. Dường nhiên là rừng các cây được tạo thành khi đi qua đồ thị theo bề rộng không phải là duy nhất, vì nó phụ thuộc vào việc lựa chọn các đỉnh  $u$  xuất phát và thứ tự ta thăm các đỉnh kề của mỗi đỉnh.

**Hình 10.3.** Rừng cây khi đi qua đồ thị theo bề rộng

Trong thuật toán dưới đây, ta giả sử  $G = (V, E)$  là đồ thị có tập các đỉnh được đánh số từ 1 tới  $n$ ,  $V = \{1, 2, \dots, n\}$ . Ta sử dụng một hàng đợi  $Q$  để lưu các đỉnh đã được thăm nhưng chưa xem xét đến các đỉnh kề nó. Để lưu lại vết của đường đi xuất phát từ đỉnh  $u$  tới các đỉnh trong quá trình tìm kiếm ta sử dụng mảng  $\text{father}[1..n]$ , trong đó  $\text{father}[w] = v$  nếu  $w$  được thăm từ  $v$  tới ( $v$  là cha  $w$  trong một cây nào đó). Ban đầu ta đánh dấu các đỉnh chưa được thăm bằng cách đặt  $\text{father}[u] = \infty$  với mọi  $u = 1, 2, \dots, n$ , trong đó  $\infty$  là giá trị đặc biệt nào đó lớn hơn  $n$ . Danh sách các đỉnh kề đỉnh  $v$  được ký hiệu là  $\text{adj}(v)$ . Sau đây là thủ tục đi thăm theo bề rộng xuất phát từ đỉnh  $u$ , thủ tục có tên là **BFS( $u$ )** (viết tắt từ Breath First Search).

```
procedure BFS( $u$ );
```

```
var
```

```
     $v, w$ : đỉnh;
```

```
     $Q$ : hàng đợi;
```

```

begin
    father[u] := 0;
    (1) Khởi tạo hàng Q chỉ chứa đỉnh u;
    (2) while not empty(Q) do
        begin
            (3)     delete(Q,v); {loại đỉnh v ở đầu hàng Q}
            (4)     for mỗi w ∈ adj(v) do
            (5)         if father[w] = ∞ then
                begin
                    (6)             father[w] := v;
                    (7)             add(w,Q);{đưa w vào cuối hàng Q}
                end;
            end;
        end;

```

Sử dụng thủ tục **BFS(u)**, ta có thủ tục đi qua đồ thị theo bề rộng như sau.

```

Procedure BreathTraversal(G);
    {đi qua đồ thị G theo bề rộng}

begin
    (8) for u := 1 to n do father[u] := ∞;
    (9) for u := 1 to n do
    (10)     if father[u] = ∞ then BFS(u);
end;

```

**Chú ý.** Nếu đồ thị G không có trọng số thì khi gọi thủ tục **BFS(u)** ta sẽ tìm được đường đi ngắn nhất từ đỉnh u tới các đỉnh có thể đạt tới từ đỉnh u, sử dụng mảng father ta sẽ xây dựng được đường đi ngắn nhất đã tìm ra.

### *Phân tích thuật toán đi qua đồ thị theo bề rộng*

Thời gian thực hiện (8) là O(n). Thời gian thực hiện lệnh (9) là tổng thời gian thực hiện các lần gọi **BFS(u)**. Thời gian thực hiện thủ tục

BFS( $u$ ) là thời gian thực hiện vòng lặp (2). Chú ý rằng mỗi đỉnh được đưa vào hàng và bị loại ra khỏi hàng đúng một lần. Khi loại một đỉnh  $v$  ra khỏi hàng ta phải xem xét tất cả các cung  $(v,w)$  (vòng lặp (4)). Tổng kết lại ta phải tiến hành một số thao tác với tất cả các đỉnh và các cung của đồ thị. Các thao tác với các đỉnh là các lệnh (3), (6) và (7). Các thao tác với các cung  $(v,w)$  là truy cập tới  $w$  trong danh sách  $\text{adj}(v)$  và kiểm tra xem nó đã được thăm chưa. Thời gian thực hiện các lệnh (3), (6) và (7) là  $O(1)$ . Vậy thuật toán đi qua đồ thị  $G = (V, E)$  theo bề rộng có thời gian thực hiện là  $O(|V| + |E|)$ , trong đó  $|V|$  và  $|E|$  là số đỉnh và số cạnh, tức là thuật toán này đòi hỏi thời gian tuyến tính với cỡ của đồ thị.

### 10.3.2. Đi qua đồ thị theo độ sâu

Đi qua đồ thị theo độ sâu là một kỹ thuật quan trọng được sử dụng để xây dựng nhiều thuật toán khác trên đồ thị. Chúng ta sẽ thấy được điều này trong các mục tiếp theo. Tư tưởng của kỹ thuật đi qua đồ thị theo độ sâu là như sau: Khi ta đang ở thăm đỉnh  $v$ , ta sẽ đi tới đỉnh  $w$  nào đó kề đỉnh  $v$ , nếu đỉnh  $w$  chưa được thăm, ta đi xuống thăm  $w$ . Nếu tất cả các đỉnh  $w$  kề  $v$  đã được thăm hoặc đỉnh  $v$  không có đỉnh kề thì ta quay lại đỉnh  $u$  mà từ  $u$  ta đã đi tới thăm  $v$ . Từ  $u$  ta lại đi tới đỉnh  $v'$  khác kề  $u$ . Nếu  $v'$  chưa được thăm, ta đi tới thăm  $v'$  và áp dụng đệ quy quá trình cho  $v'$ .

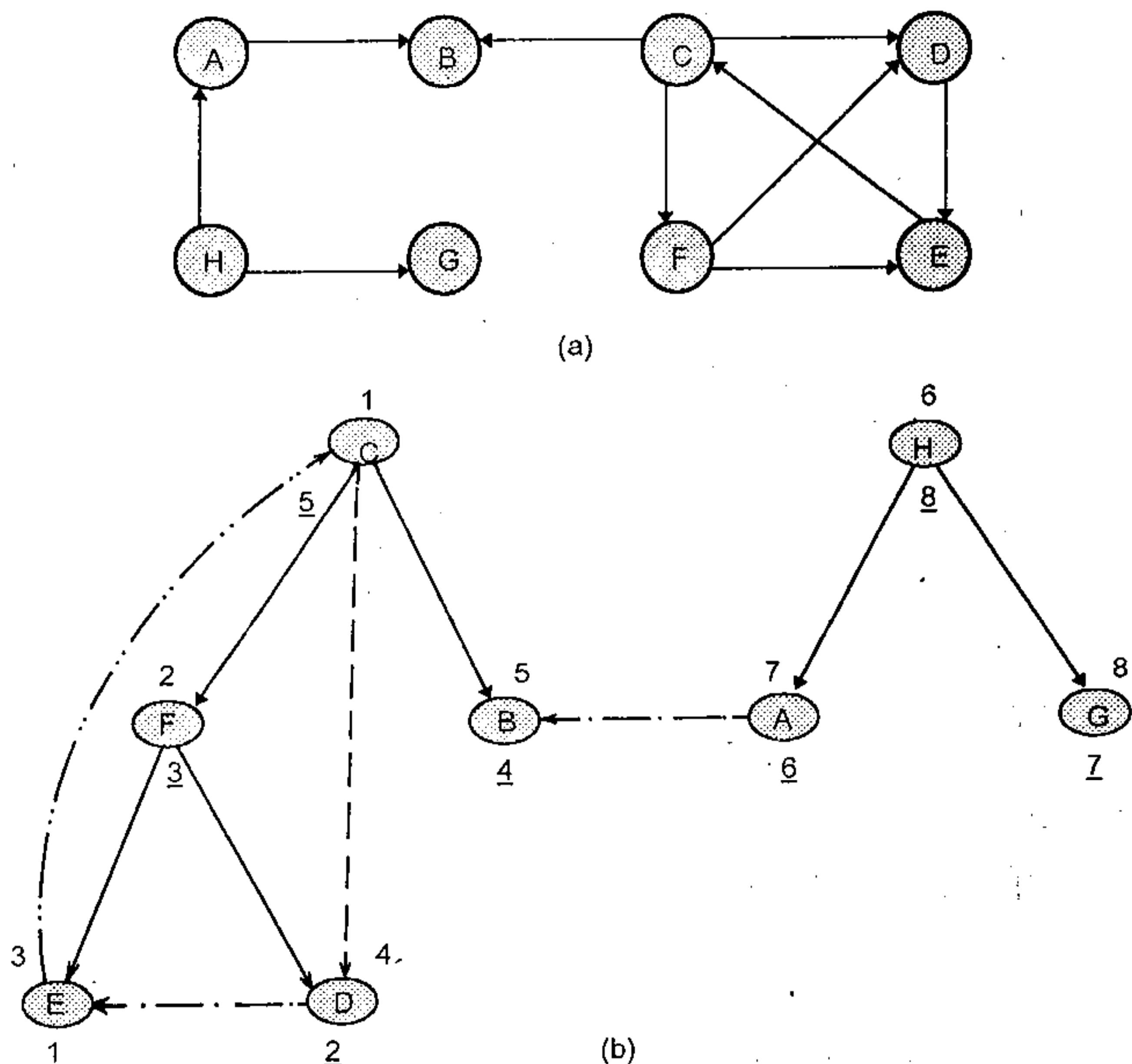
Kỹ thuật này được gọi là kỹ thuật tìm kiếm theo độ sâu bởi vì ta đã theo các cung (hoặc cạnh) để đi sâu xuống thăm các đỉnh chừng nào còn có thể đi được.

Quá trình đi qua đồ thị theo độ sâu cũng tạo ra một rừng các cây. Mỗi lần ta chọn đỉnh  $u$  làm đỉnh xuất phát để đi thăm thì một cây gốc  $u$  được tạo thành, trong cây này đỉnh  $w$  là đỉnh con của đỉnh  $v$  nếu từ  $v$  ta đã đi xuống thăm  $w$ .

Trong quá trình đi qua đồ thị theo độ sâu, ta sẽ gán cho các đỉnh của đồ thị hai loại số hiệu: số hiệu trước và số hiệu sau. Số hiệu trước là thứ tự được thăm của các đỉnh. Còn số hiệu sau được gán cho các đỉnh vào thời điểm ta đã hoàn thành thăm tất cả các đỉnh kề nó. Trong mỗi cây của rừng các cây được tạo thành thì số hiệu trước và số hiệu sau

chính là cách đánh số các đỉnh của cây theo thứ tự trước (preorder) và thứ tự sau (postorder).

**Ví dụ.** Xét đồ thị trong hình 10.4a, khi đi qua đồ thị này ta có thể nhận được hai cây trong hình 10.4b. Số hiệu trước và sau được ghi cạnh mỗi đỉnh, số hiệu sau được gạch dưới. Chú ý là rùng các cây tạo thành không phải duy nhất và các số hiệu trước, sau của các đỉnh cũng không phải duy nhất.



Hình 10.4. Rừng các cây khi đi qua đồ thị theo độ sâu

Trong thuật toán đi qua đồ thị theo độ sâu (Depth Travelsal), ta giả sử  $G = (V, E)$  là đồ thị với tập các đỉnh  $V = \{1, 2, \dots, n\}$ . Ta sử dụng mảng prenum[1..n] để lưu số hiệu trước và mảng postnum[1..n] để lưu số hiệu sau của các đỉnh. Ban đầu các đỉnh được xem là chưa được thăm bằng

sau của các đỉnh. Ban đầu các đỉnh được xem là chưa được thăm bằng cách đặt  $\text{prenum}[v] := 0$  với  $v = 1, 2, \dots, n$ . Danh sách các đỉnh kề đỉnh  $v$  là  $\text{adj}(v)$ .

Thủ tục ***DepthTravelsal*** sau đây sử dụng thủ tục ***DFS(v)*** (viết tắt từ Depth First Search). ***DFS(u)*** là thủ tục đệ quy tìm kiếm theo độ sâu xuất phát từ đỉnh  $v$ .

```

Procedure DepthTravelsal(G);
var
    v : đỉnh;
    i, k : integer;
    {i, k là các biến đếm số hiệu trước, sau}
begin
    for v := 1 to n do prenum[v] := 0;
    {đánh dấu các đỉnh chưa được thăm}
    i := 0;
    k := 0;
    for v := 1 to n do
        if prenum[v] = 0 then DFS(v);
end;

procedure DFS(v);
begin
    i := i+1;
    prenum[v] := i;
    for each w ∈ adj(v) do
        if prenum[w] = 0 then DFS(w);
    k := k+1;
    postnum[v] := k;
end;

```

Dễ dàng thấy rằng thời gian chạy của thuật toán đi qua đồ thị  $G = (V, E)$  theo độ sâu cũng là  $O(|V| + |E|)$ . Bởi vì ở mỗi đỉnh  $v \in V$ , thủ tục DFS được gọi đúng một lần. Trong mỗi lần gọi  $DFS(v)$  ta cần xem xét tất cả các đỉnh  $w$  kề  $v$  (vòng lặp for each  $w \in adj(v)$ ), tức là tất cả các cung  $(v, w)$ .

### *Phân lớp các cung của đồ thị định hướng*

Khi đi qua đồ thị theo độ sâu, một rừng các cây được tạo thành. Trong rừng các cây này, các cung của đồ thị định hướng được phân thành bốn lớp sau đây:

- **Các cung cây.** Đó là các cung liên kết các đỉnh tạo thành cây.
- **Các cung tiến.** Đó là các cung  $u \rightarrow v$ , trong đó  $u$  và  $v$  nằm cùng một cây và đỉnh  $u$  là đỉnh tiền thân của đỉnh  $v$ .
- **Các cung ngược.** Đó là các cung  $u \rightarrow v$ , trong đó  $u$  và  $v$  nằm trong cùng một cây, nhưng đỉnh  $u$  là hậu thế của đỉnh  $v$ .
- **Các cung xiên.** Đó là các cung  $u \rightarrow v$ , trong đó  $u$  và  $v$  nằm trên hai cây khác nhau, hoặc  $u$  và  $v$  nằm trong cùng một cây nhưng  $u$  không phải là tiền thân cũng không phải là hậu thế của  $v$ .

**Ví dụ.** Xét rừng các cây trong hình 10.4b được tạo thành khi đi qua đồ thị trong hình 10.4a. Các cung cây là các cung  $C \rightarrow F$ ,  $C \rightarrow B$ , ... Còn  $C \rightarrow D$  là cung tiến,  $E \rightarrow C$  là cung ngược, và  $D \rightarrow E$ ,  $A \rightarrow B$  là các cung xiên.

Ta có nhận xét sau

- Nếu  $(u, v)$  là cung tiến thì  $\text{pnum}[u] < \text{pnum}[v]$  và  $\text{postnum}[u] > \text{postnum}[v]$ .
- Nếu  $(u, v)$  là cung ngược thì  $\text{pnum}[u] > \text{pnum}[v]$  và  $\text{postnum}[u] < \text{postnum}[v]$ .
- Nếu  $(u, v)$  là cung xiên thì  $\text{pnum}[u] > \text{pnum}[v]$  và  $\text{postnum}[u] > \text{postnum}[v]$ .

Trong quá trình đi qua đồ thị theo độ sâu, ta có thể sử dụng thủ tục sau đây để gán nhãn cho các cung.

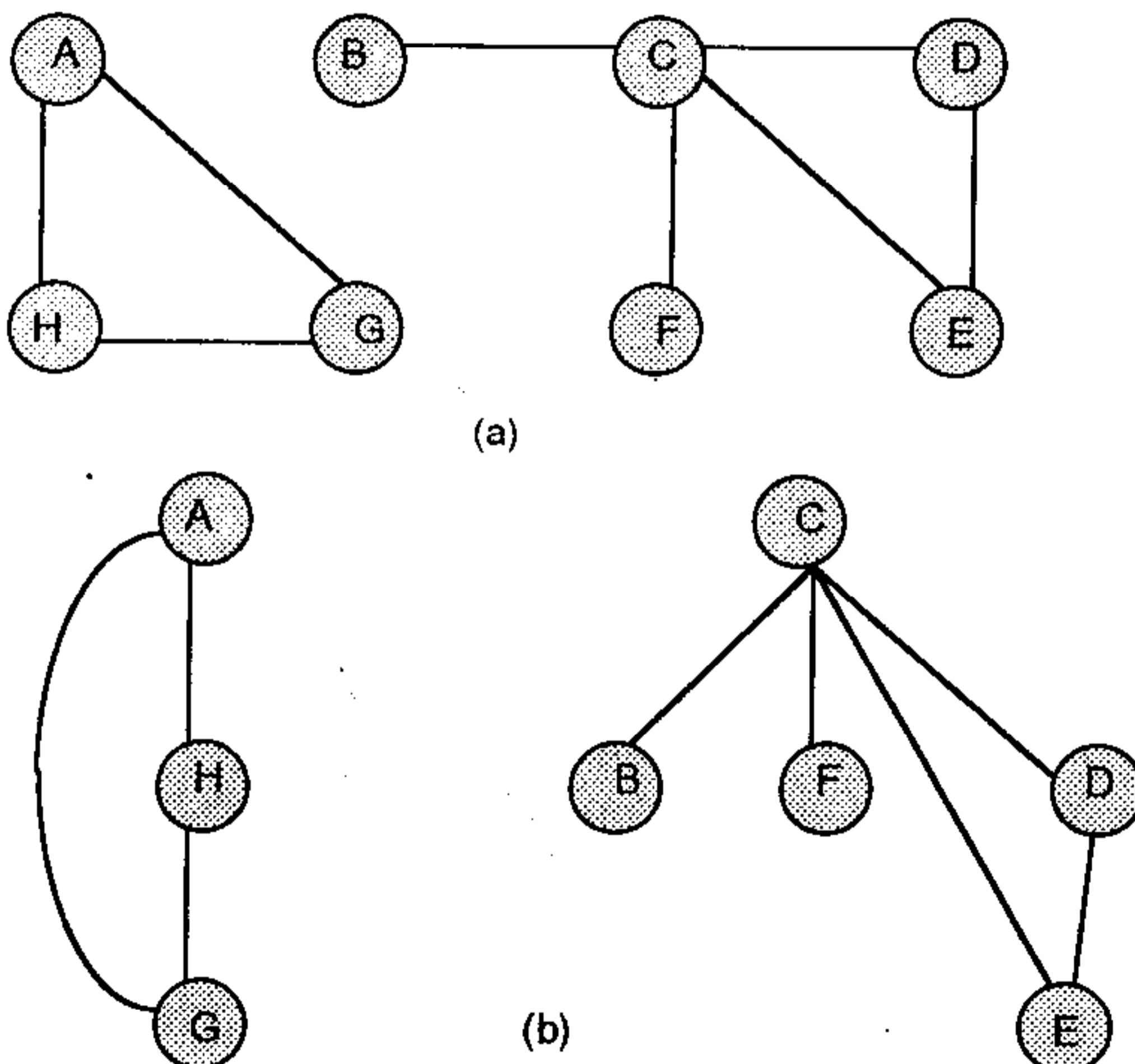
Giả sử ta đang ở đỉnh  $u$  và đi theo cung  $(u, v)$  để tới  $v$ . Khi đó :

- Nếu  $\text{pnum}[u] \neq 0$  ( $u$  đã được thăm) và  $\text{pnum}[v] = 0$  ( $v$  chưa được thăm) thì  $(u, v)$  là cung cây.

- Nếu  $\text{prenum}[u] \neq 0$  và  $\text{prenum}[v] \neq 0$ , nhưng  $\text{postnum}[v] = 0$  (chưa hoàn thành thăm các đỉnh kề đỉnh  $v$ ) thì  $(u,v)$  là cung ngược.
- Nếu  $\text{prenum}[u] \neq 0$ ,  $\text{prenum}[v] \neq 0$  và  $\text{postnum}[v] \neq 0$ , nhưng  $\text{prenum}[u] < \text{prenum}[v]$  thì  $(u,v)$  là cung tiến.
- Nếu  $\text{prenum}[u] \neq 0$ ,  $\text{prenum}[v] \neq 0$  và  $\text{postnum}[v] \neq 0$  nhưng  $\text{prenum}[u] > \text{prenum}[v]$  thì  $(u,v)$  là cung xiên.

Đối với đồ thị vô hướng, ta chỉ có hai loại cạnh: cạnh cây và cạnh ngược. Để thấy rõ tại sao như thế, ta xét một cạnh  $(u,v)$  bất kỳ. Giả sử trong quá trình tìm kiếm ta đạt tới đỉnh  $u$  trước  $v$ . Khi đó việc gọi thủ tục  $\text{DFS}(u)$  chỉ được hoàn thành sau khi  $v$  đã đạt tới. Khi thực hiện thủ tục  $\text{DFS}(u)$ , nếu ta đi theo cạnh  $(u,v)$  để tới thăm  $v$  thì  $(u,v)$  là cạnh cây, còn nếu ta đã đi theo một đường khác để tới thăm  $v$  thì  $(u,v)$  là cạnh ngược.

**Ví dụ.** Xét đồ thị vô hướng trong hình 10.5a. Khi đi qua đồ thị này theo độ sâu, ta có thể nhận được rừng gồm hai cây trong hình 10.5b. Trong đó ngoài các cạnh cây thì  $(G,A)$  và  $(E,C)$  là các cạnh ngược.



Hình 10.5. Rừng các cây khi đi qua đồ thị vô hướng theo độ sâu

Một điều quan trọng cần biết là, sử dụng kỹ thuật đi qua đồ thị theo độ sâu và phân lớp các cung (cạnh) ta có thể xác định được đồ thị có chu trình hay không. Ta có thể chứng minh được khẳng định sau đây: Một đồ thị không có chu trình nếu và chỉ nếu nó không có các cung (cạnh) ngược. Thật vậy, để cho xác định ta giả sử đồ thị là vô hướng. Điều kiện cần là hiển nhiên, vì cạnh ngược nối đỉnh hậu thế với đỉnh tiền thân sẽ tạo thành một chu trình, chẳng hạn chu trình (A, H, G, A) hoặc (C, D, E, C) trong hình 10.5a. Bây giờ ta sẽ chỉ ra rằng, nếu đồ thị không có cạnh ngược thì nó không có chu trình. Thật vậy, nếu đồ thị không có cạnh ngược thì đồ thị là rừng các cây, các cây đương nhiên là không có chu trình.

## 10.4. THÀNH PHẦN LIÊN THÔNG VÀ THÀNH PHẦN LIÊN THÔNG MẠNH

Trong mục này chúng ta sẽ áp dụng các kỹ thuật đi qua đồ thị để tìm các thành phần liên thông của đồ thị vô hướng và các thành phần liên thông mạnh của đồ thị định hướng.

### 10.4.1. Thành phần liên thông

Đồ thị vô hướng  $G = (V, E)$  được gọi là *liên thông* nếu tồn tại đường đi nối mọi cặp đỉnh của nó.

Nếu  $G$  không liên thông thì một đồ thị con liên thông cực đại của  $G$  được gọi là một *thành phần liên thông* của  $G$  (đồ thị con cực đại là đồ thị con không thực sự chứa trong đồ thị con khác). Như vậy, nếu  $G$  chỉ có một thành phần liên thông thì nó liên thông.

Áp dụng kỹ thuật đi qua đồ thị theo độ sâu, ta dễ dàng tìm được tất cả các thành phần liên thông. Mỗi cây được tạo thành trong quá trình tìm kiếm theo độ sâu là một thành phần liên thông. Ví dụ, đồ thị trong hình 10.5a có hai thành phần liên thông là hai cây trong hình 10.5b.

Trong thủ tục tìm các thành phần liên thông của đồ thị  $G$ , ***CompSearch(G)***, ta sử dụng biến  $c$  để đếm số thành phần liên thông. Giả sử các đỉnh của đồ thị được đánh số từ 1 đến  $n$ . Ta sử dụng mảng  $num[1..n]$  để lưu số hiệu thành phần liên thông chứa mỗi đỉnh, tức là

$\text{num}[v]$  là số hiệu của thành phần liên thông chứa đỉnh  $v$ . Thủ tục  $\text{Comp}(v)$  là thủ tục đi theo độ sâu từ  $v$  và gán cho tất cả các đỉnh đạt tới từ  $v$  một số hiệu thành phần liên thông  $c$  nào đó.

```

Procedure CompSearch(G);
var
    v, w : đỉnh;
    c : integer;
begin
    for v := 1 to n do num[v] := 0;
    c := 0;
    for v := 1 to n do
        if num[v] = 0 then
            begin
                c := c+1;
                comp(v);
            end;
    end;

procedure Comp(v);
begin
    num[v] := c;
    for each w ∈ adj(v) do
        if num[w] = 0 then comp(w);
    end;

```

#### 10.4.2. Thành phần liên thông mạnh

Đồ thị định hướng  $G = (V, E)$  được gọi là *liên thông mạnh*, nếu với mọi cặp đỉnh  $(u,v)$  của nó đều tồn tại đường đi từ  $u$  đến  $v$  và đường đi từ  $v$  đến  $u$ .

Trong một đồ thị định hướng không liên thông mạnh, ta gọi một đồ thị con liên thông mạnh cực đại của nó là một *thành phần liên thông*

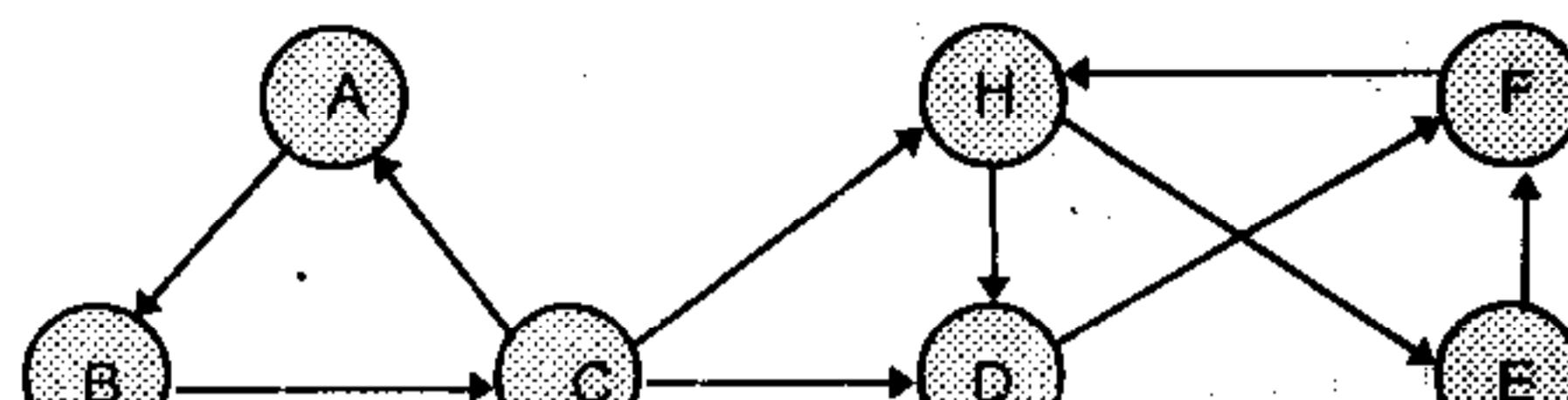
mạnh. Chẳng hạn, đồ thị trong hình 10.6a có hai thành phần liên thông mạnh, đó là các đồ thị con cảm sinh bởi các tập đỉnh  $\{A, B, C\}$  và  $\{D, E, F, H\}$ .

Sử dụng kỹ thuật đi qua đồ thị theo độ sâu, ta có thể xây dựng được thuật toán xác định thành phần liên thông mạnh. Thuật toán gồm những bước sau:

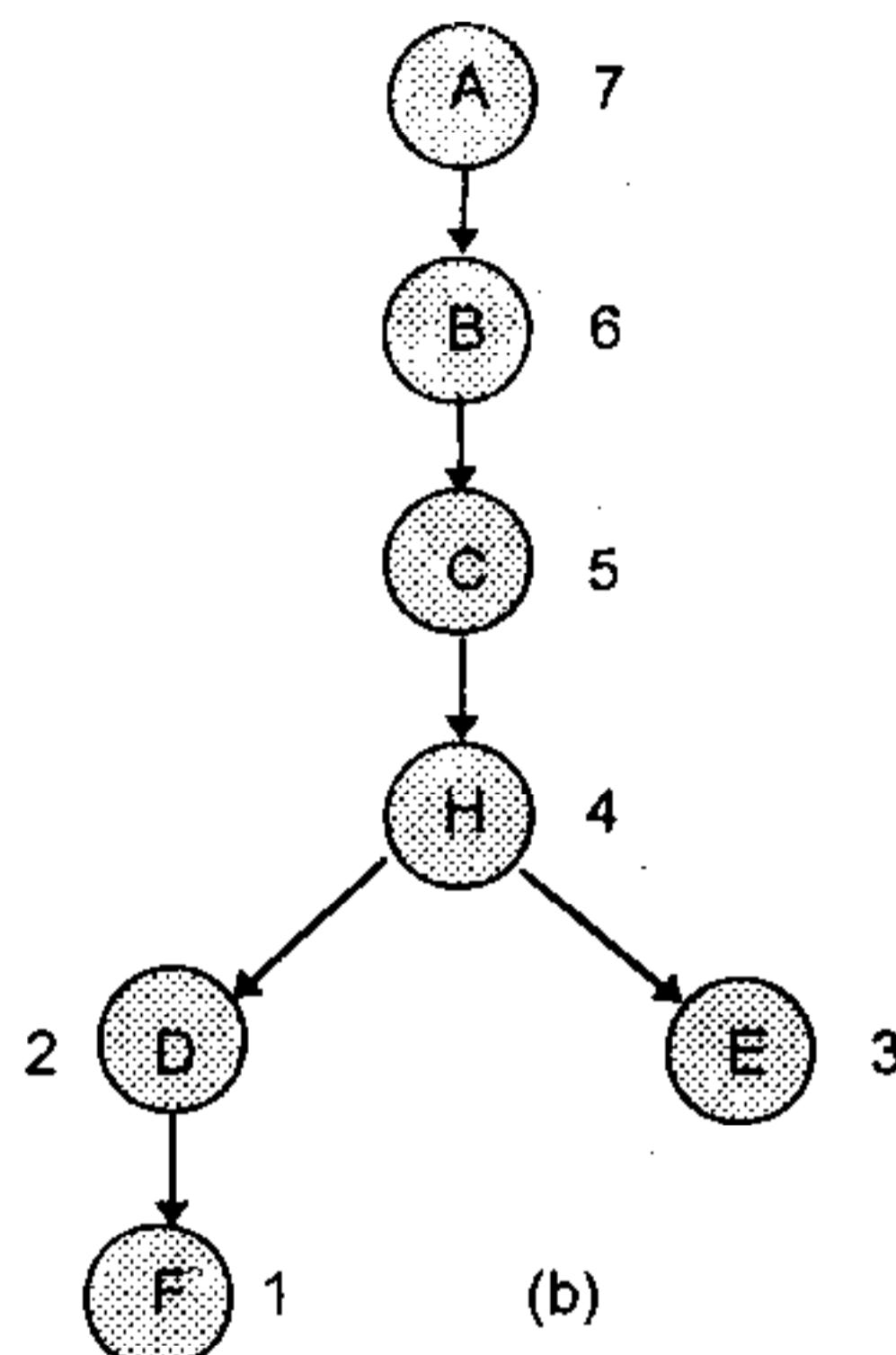
- Thực hiện đi qua đồ thị  $G$  theo độ sâu và đánh số các đỉnh của nó theo thứ tự sau (postnum).
- Xây dựng đồ thị  $G'$  từ đồ thị  $G$  bằng cách đổi hướng tất cả các cung.
- Áp dụng thủ tục DFS( $v$ ) đi qua đồ thị  $G'$  theo độ sâu bắt đầu từ đỉnh  $v$  có postnum[ $v$ ] lớn nhất (nhận được từ bước 1). Khi đi từ  $v$  mà chưa thăm hết các đỉnh của đồ thị  $G'$  thì ta lại đi thăm các đỉnh còn lại của  $G'$  theo độ sâu bắt đầu từ đỉnh  $w$  có postnum[ $w$ ] lớn nhất trong các đỉnh còn lại. Lặp lại quá trình trên cho đến khi ta thăm hết các đỉnh của  $G'$ .
- Mỗi cây nhận được ở bước 3 xác định một thành phần liên thông mạnh của đồ thị  $G$ .

Một câu hỏi đặt ra là, tại sao thực hiện các bước trên ta lại tìm được thành phần liên thông mạnh của đồ thị  $G$ . Chứng minh tính đúng đắn của thuật toán trên đòi hỏi ta phải sử dụng đến sự phân loại các cung và cách đánh số các đỉnh theo thứ tự trước và sau. Chúng ta bỏ qua chứng minh này. Sau đây ta sẽ đưa ra một ví dụ minh họa cho thuật toán trên.

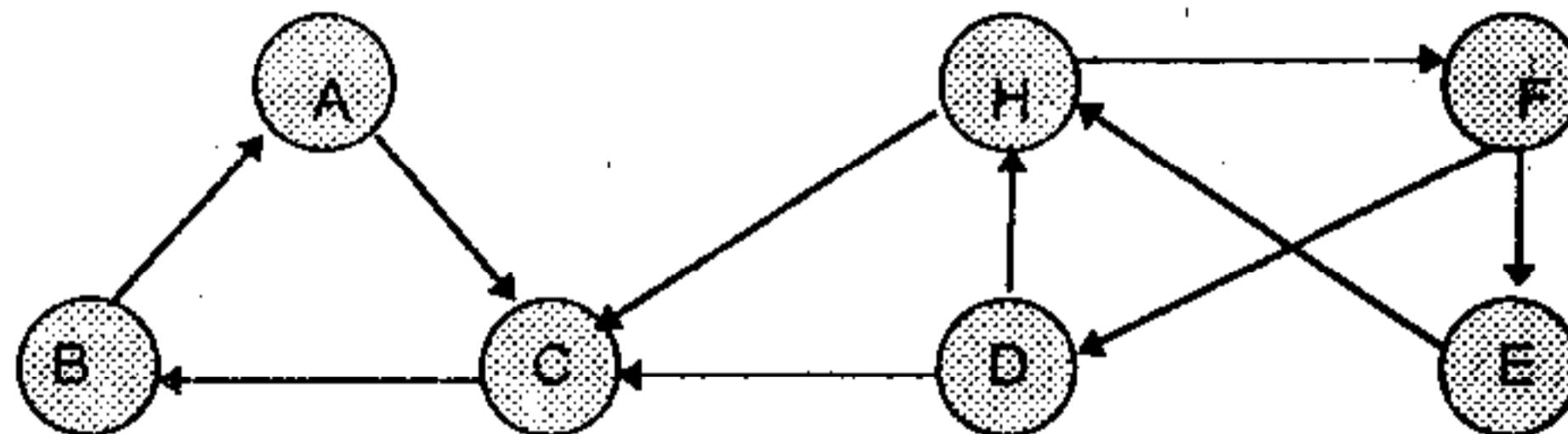
**Ví dụ.** Chúng ta áp dụng thuật toán trên cho đồ thị trong hình 10.6a. Sau bước 1 ta nhận được số hiệu sau của các đỉnh được cho trong hình 10.6b. Đổi hướng các cung của đồ thị 10.6a ta nhận được đồ thị trong hình 10.6c. Thực hiện bước 3, đi qua đồ thị 10.6c theo độ sâu bắt đầu từ đỉnh A (vì postnum[A] = 7 là lớn nhất) ta nhận được cây bên trái trong hình 10.6d. Trong các đỉnh còn lại, đỉnh H có số hiệu sau lớn nhất. Tiếp tục đi trên đồ thị theo độ sâu bắt đầu từ đỉnh H ta nhận được cây bên phải trong hình 10.6d. Toàn bộ các đỉnh của đồ thị 10.6c đã được thăm. Do đó đồ thị 10.6a có hai thành phần liên thông mạnh. Đó là các đồ thị con của nó cảm sinh bởi các đỉnh  $\{A, C, B\}$  và  $\{H, F, D, E\}$ .



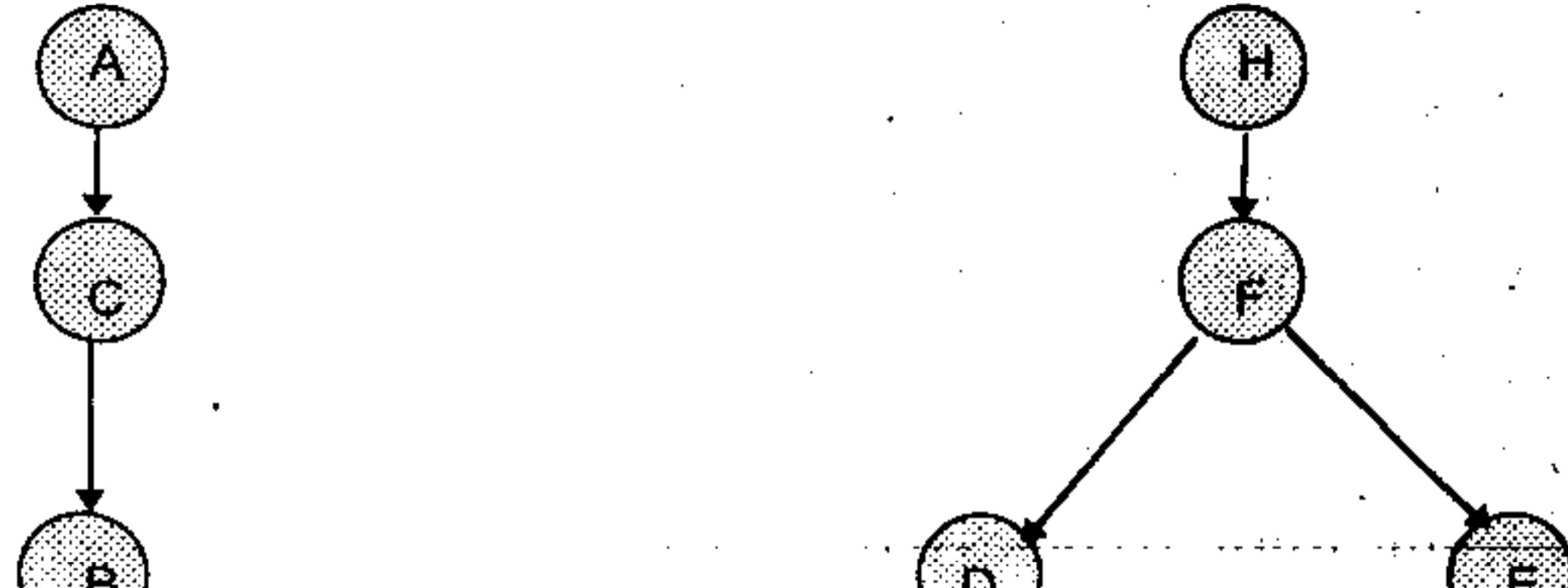
(a)



(b)



(c)

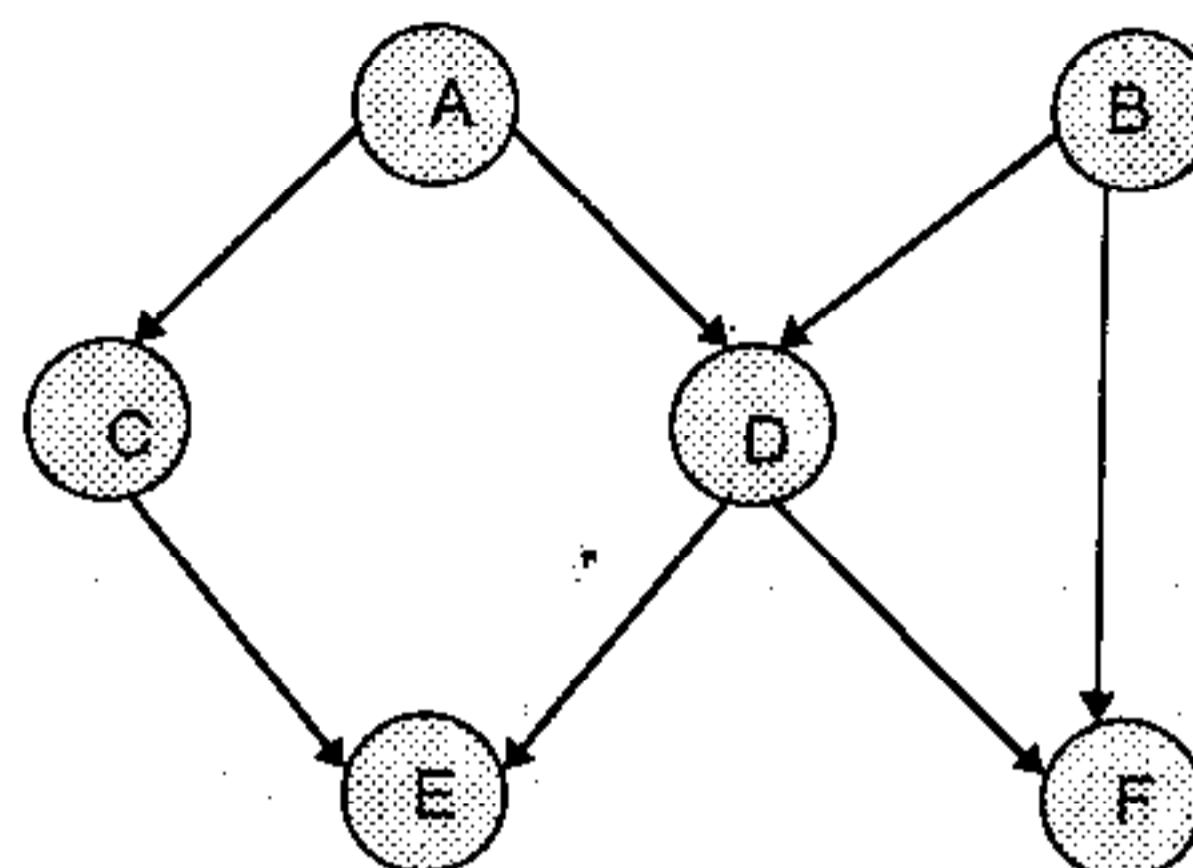


(d)

**Hình 10.6.** Tìm các thành phần liên thông mạnh

## 10.5. ĐỒ THỊ ĐỊNH HƯỚNG KHÔNG CÓ CHU TRÌNH. SẮP XẾP TOPO

Hình 10.7 minh họa một đồ thị định hướng không có chu trình. Đồ thị định hướng không có chu trình gọi tắt là DAG (viết tắt của Directed Acyclic Graph). DAG là một trường hợp riêng của đồ thị định hướng, nhưng tổng quát hơn khái niệm cây.



**Hình 10.7.** Một đồ thị định hướng không có chu trình

Nhiều quan hệ có thể biểu diễn bởi DAG. Chẳng hạn, quan hệ thứ tự bộ phận trên một tập  $A$  có thể biểu diễn bởi DAG như sau: Mỗi phần tử của  $A$  là một đỉnh của đồ thị và nếu  $a, b \in A$  và  $a < b$  thì  $(a, b)$  là một cung của đồ thị. Do tính chất của quan hệ thứ tự bộ phận, đồ thị này không có chu trình, tức là nó là một DAG.

Giả sử chúng ta có một đề án phức tạp bao gồm nhiều nhiệm vụ. Trong quá trình thực hiện, một nhiệm vụ có thể cần phải được hoàn thành trước khi một số nhiệm vụ khác được bắt đầu thực hiện. Ở đây, ta có thể sử dụng DAG để biểu diễn đề án này. Mỗi nhiệm vụ là một đỉnh, nếu nhiệm vụ  $A$  cần phải được hoàn thành trước khi bắt đầu thực hiện nhiệm vụ  $B$  thì  $(A, B)$  là một cung của đồ thị. Giả sử một công ty thực hiện đề án này và tại mỗi thời điểm công ty chỉ làm được một nhiệm vụ, làm xong một nhiệm vụ nó mới có thể bắt đầu làm nhiệm vụ khác. Vấn đề của công ty là phải sắp xếp các nhiệm vụ theo một thứ tự nào đó để khi thực hiện đảm bảo quan hệ thời gian giữa các nhiệm vụ.

Vấn đề sắp xếp topo là như sau: Cho  $G = (V, E)$  là một DAG, ta cần sắp xếp các đỉnh của  $G$  thành một dãy (ta gọi dãy này là dãy topo) sao cho nếu  $(u, v) \in E$  thì  $u$  đứng trước  $v$  trong dãy topo.

**Ví dụ.** Một dãy topo của DAG trong hình 10.7 là (A, C, B, D, E, F). Cần lưu ý rằng, dãy topo không phải là duy nhất. Chẳng hạn, một dãy topo khác của đồ thị 10.7 là (A, B, D, C, E, F).

Trong thuật toán sau đây,  $G = (V, E)$  là đồ thị định hướng với tập các đỉnh  $V = \{1, 2, \dots, n\}$ . Thuật toán sẽ cho ta biết  $G$  có chu trình hay không, và nếu không thuật toán sẽ cho ta dãy topo của  $G$ .

Sử dụng thuật toán đi qua đồ thị theo độ sâu và đánh số các đỉnh theo thứ tự prenum và postnum ta có thể đưa ra được thuật toán sắp xếp topo. Trong quá trình tìm kiếm theo độ sâu, nếu gặp cung ngược thì  $G$  có chu trình và ta dừng lại. Nếu đồ thị không có chu trình thì trong rừng các cây được tạo thành khi đi qua đồ thị theo độ sâu chỉ có các cung cây, cung tiến và cung xiên. Nhớ lại rằng, nếu  $(u, v)$  là một trong ba loại cung trên thì  $\text{postnum}[u] > \text{postnum}[v]$ . Như vậy,  $\text{postnum}[u]$  là cách đánh số các đỉnh trong dãy topo theo thứ tự ngược lại.

Trong thuật toán sắp xếp topo (**TopoSort**) sau đây, ta sử dụng thủ tục **TPS(v)**, đó là thủ tục tìm kiếm theo độ sâu **DFS(v)**, chỉ khác là thay cho mảng  $\text{postnum}[1..n]$  ta dùng mảng  $\text{toponum}[1..n]$  để ghi lại số thứ tự của các đỉnh trong dãy topo.

```

Procedure TopoSort;
var
    v, w : đỉnh;
    i, k : integer;
begin
    for v := 1 to n do
        begin
            prenum[v] := 0;
            toponum[v] := 0;
        end;
    i := 0;
    k := n+1;
    for v := 1 to n do
        if prenum[v] = 0 then TPS(v);
    end;

```

```

procedure TPS(v);
begin
    i := i+1;
    prenum[v] := i;
    for each w ∈ adj(v) do
        if prenum[w] = 0 then TPS(w)
        else if topnum[w] = 0 then write('đồ thị có chu trình');
    k := k-1;
    topnum[v] := k;
end;

```

## 10.6. ĐƯỜNG ĐI NGẮN NHẤT

Chúng ta đã chỉ ra trong mục 10.3.1 rằng có thể sử dụng tìm kiếm theo bề rộng để giải quyết vấn đề tìm đường đi ngắn nhất từ một đỉnh tới các đỉnh khác trong đồ thị không có trọng số. Đối với đồ thị có trọng số vấn đề sẽ khó khăn hơn.

Trong mục này ta giả thiết rằng,  $G = (V, E)$  là đồ thị định hướng có trọng số, độ dài cung  $(u,v)$  là  $c(u,v)$  và  $c(u,v) \geq 0$ . Nếu đồ thị vô hướng thì ta có thể xét nó như đồ thị định hướng, trong đó mỗi cạnh  $(u,v)$  được xem như hai cung  $u \rightarrow v$  và  $v \rightarrow u$  có độ dài bằng nhau và bằng độ dài của cạnh  $(u,v)$ . Nhắc lại rằng, nếu  $(v_0, v_1, \dots, v_k)$  là một đường đi từ  $v_0$  đến  $v_k$  thì độ dài của đường đi này, ký hiệu là  $\delta(v_0, v_k) = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$ .

Chúng ta sẽ xét hai vấn đề sau:

- Tìm đường đi ngắn nhất từ một đỉnh nguồn tới các đỉnh còn lại của đồ thị.
- Tìm đường đi ngắn nhất giữa mọi cặp đỉnh của đồ thị.

### 10.6.1. Đường đi ngắn nhất từ một đỉnh nguồn: Thuật toán Dijkstra

Giả sử đồ thị  $G = (V, E)$  có các đỉnh được đánh số từ 1 đến  $n$  và đỉnh nguồn được chọn là 1. Đồ thị được biểu diễn bởi ma trận kề  $C[1..n, 1..n]$ ,

trong đó  $C[i,j]$  là độ dài cung  $(i,j)$ , nếu không có cung  $(i,j)$  thì  $C[i,j] = \infty$  ( $\infty$  là giá trị đủ lớn, chẳng hạn lớn hơn tổng độ dài của tất cả các cung trong đồ thị).

Sau đây chúng ta sẽ trình bày thuật toán Dijkstra.

Thuật toán này được thiết kế dựa trên kỹ thuật tham ăn. Ta xác định đường đi ngắn nhất từ nguồn 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 mà đường đi ngắn nhất tới chúng đã được xác định vào tập  $S$ . Ban đầu  $S$  chỉ chứa đỉnh nguồn 1. Chúng ta sẽ gọi các đường đi từ nguồn tới các đỉnh  $u$  khác nhưng chỉ đi qua các đỉnh nằm trong  $S$  là đường đi đặc biệt. Ta sử dụng mảng  $D[2..n]$ , trong đó  $D[u]$  lưu độ dài đường đi đặc biệt ngắn nhất từ 1 tới  $u$ . Ban đầu vì  $S$  chỉ chứa 1, ta lấy  $D[u] = C[1,u]$  ( $u = 1, 2, \dots, n$ ). Tại mỗi bước ta sẽ chọn một đỉnh  $v \in V - S$  mà  $D[v]$  là nhỏ nhất và thêm  $v$  vào  $S$ . Sau khi thêm  $v$  vào  $S$ , ta xác định lại các  $D[w]$  với  $w \notin S$ . Nếu độ dài đường đi đặc biệt qua đỉnh  $v$  (vừa được chọn) tới  $w$  nhỏ hơn  $D[w]$  thì ta lấy  $D[w]$  là độ dài đường đi đó. Khi  $S = V$  thì  $D[u]$  sẽ lưu độ dài đường đi ngắn nhất từ 1 tới  $u$ , với  $u = 2, \dots, n$ .

Để lưu lại vết của đường đi, ta sử dụng mảng  $P[2..n]$ , trong đó  $P[w] = v$  nếu ta đi tới  $w$  theo cung  $(v,w)$ , tức là  $v$  đứng kề trước  $w$  trên đường đi.

```

procedure Dijkstra;
begin
    S := {1};
    (1) for i := 2 to n do
        begin
            D[i] := C[1,i];
            P[i] := 1;
            {khởi tạo mảng D và P}
        end;
    (2) while V - S ≠ ∅ do
        begin
            (3) Chọn v ∈ V - S mà D[v] nhỏ nhất;
```

S := S ∪ {v}; {thêm đỉnh v vào tập S}

(4)      **for each** w ∈ V - S **do**

(5)            **if** D[v] + C[v,w] < D[w] **then**

**begin**

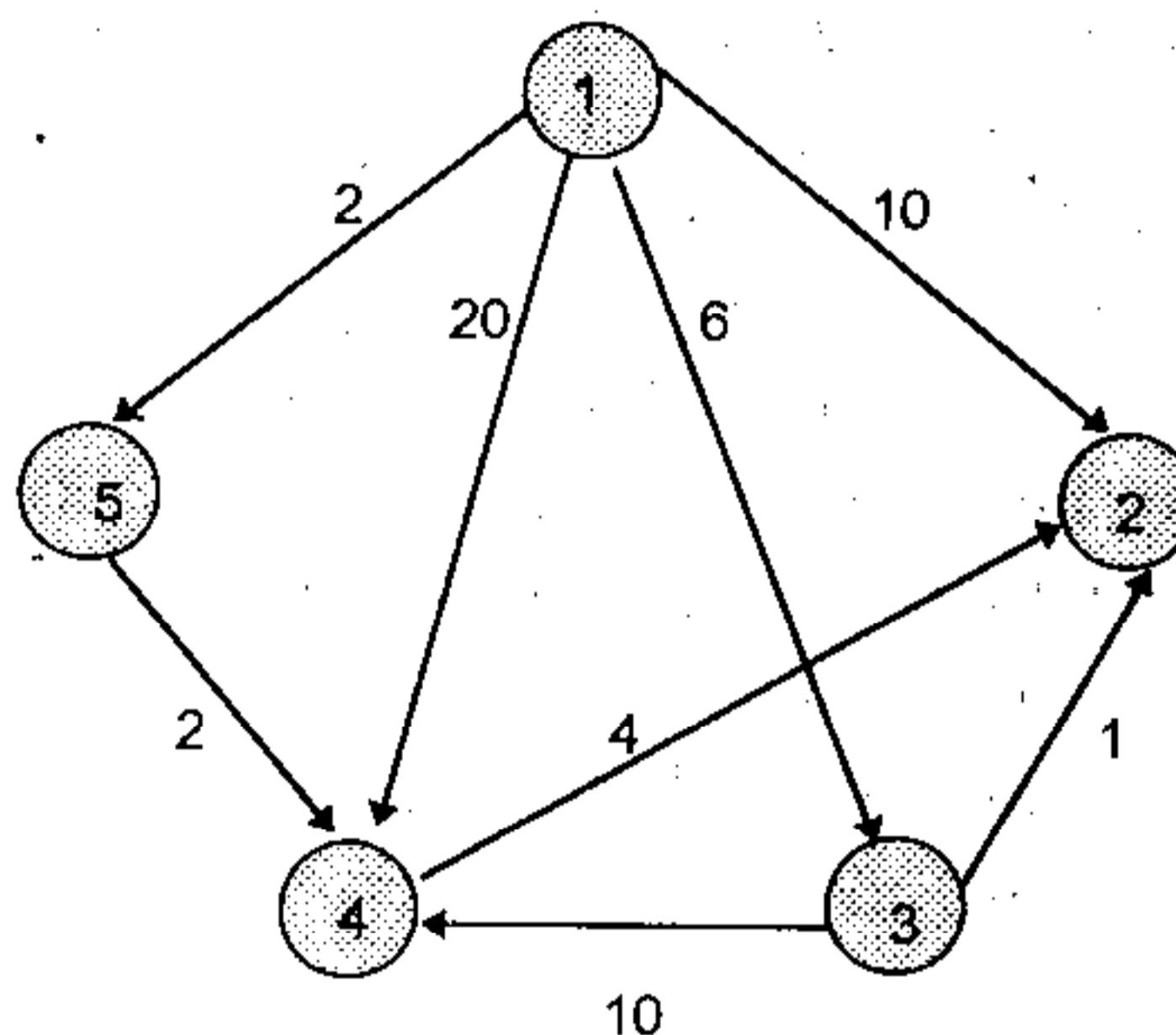
                  D[w] := D[v] + C[v,w];

                  P[w] := v;

**end;**

**end;**

**Ví dụ.** Xét đồ thị định hướng có trọng số sau:



Kết quả thực hiện các bước của thuật toán Dijkstra được cho trong bảng sau

Bước	v	V - S	D	P
khởi tạo	-	{2, 3, 4, 5}	[10, 6, 20, 2]	[1, 1, 1, 1]
1	5	{2, 3, 4}	[10, 6, 4, 2]	[1, 1, 5, 1]
2	4	{2, 3}	[8, 6, 4, 2]	[4, 1, 5, 1]
3	3	{2}	[7, 6, 4, 2]	[3, 1, 5, 1]

### **Chứng minh tính đúng đắn của thuật toán Dijkstra**

Chúng ta chứng minh bằng quy nạp theo sự mở rộng của tập  $S$  hai khẳng định sau đây

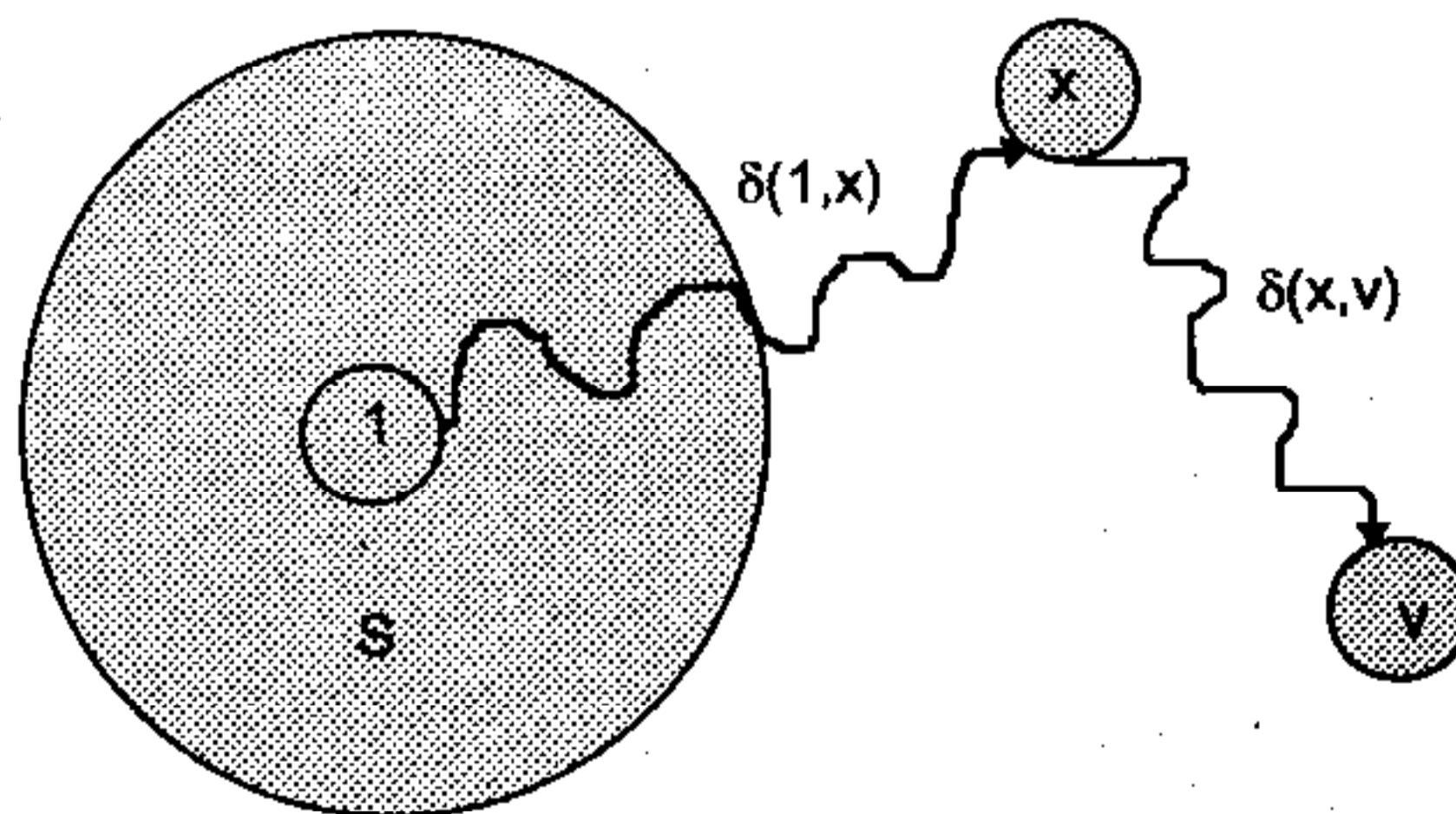
- Nếu  $v \in S$  thì  $D[v]$  là độ dài đường đi ngắn nhất từ 1 tới  $v$ .
- Nếu  $v \notin S$  thì  $D[v]$  là độ dài đường đi đặc biệt ngắn nhất từ 1 tới  $v$ .

Rõ ràng là, ban đầu  $S = \{1\}$  thì cả hai khẳng định trên đều đúng.

Giả sử rằng hai khẳng định trên đã đúng đối với tập  $S$ , ta cần chứng minh nó đúng với tập  $S'$  là mở rộng của tập  $S$  bằng cách thêm vào đỉnh  $v$  được chọn ở lệnh (3), và đối với các  $D[w]$  ( $w \notin S'$ ) được xác định lại ở lệnh (5).

Trước hết ta chứng minh rằng, với  $v$  được chọn bởi lệnh (3) thì  $D[v]$  là độ dài đường đi ngắn nhất từ 1 tới  $v$ . Thật vậy, xét một đường đi khác bất kỳ từ 1 tới  $v$ , có độ dài là  $\delta(1,v)$ . Giả sử đỉnh đầu tiên trên đường đi này không thuộc  $S$  là  $x$ . Nếu  $x$  trùng với  $v$  thì đường đi này là đường đi đặc biệt và theo giả thiết quy nạp  $D[v]$  là độ dài đường đi đặc biệt ngắn nhất, do đó  $\delta(1,v) \geq D[v]$ . Giả sử  $x$  khác với  $v$  (xem hình 10.8). Khi đó đường đi từ 1 tới  $v$  được phân thành hai đoạn: đường đi từ 1 tới  $x$ , độ dài là  $\delta(1,x)$  và đường đi từ  $x$  tới  $v$ , độ dài là  $\delta(x,v)$ . Theo giả thiết quy nạp ta có  $D[x] \leq \delta(1,x)$ , vì  $\delta(1,x)$  là độ dài đường đi đặc biệt từ 1 tới  $x$  còn  $D[x]$  là độ dài đường đi đặc biệt ngắn nhất. Do cách chọn  $v$  ở lệnh (3) ta có

$$D[v] \leq D[x] \leq \delta(1,x) \leq \delta(1,x) + \delta(x,v) = \delta(1,v)$$



Hình 10.8. Một đường đi bất kỳ từ 1 tới  $v$

Bây giờ ta chứng minh rằng,  $D[w]$  (với  $w \notin S' = S \cup \{x\}$ ) là độ dài đường đi đặc biệt ngắn nhất từ 1 tới  $w$ . Thật vậy, xét một đường đi đặc

biệt (chỉ đi qua các đỉnh thuộc  $S'$ ) bất kỳ từ 1 tới w, có độ dài  $\delta(1,w)$ . Nếu đường đi này không qua v thì nó là đường đi đặc biệt (đối với  $S$ ) từ 1 tới w. Theo giả thiết quy nạp, ta có  $\delta(1,w) \geq D[w]$  (đối với  $S$ )  $\geq D[w]$  (đối với  $S'$ ). Nếu đường này đi qua v, khi đó có hai khả năng sau

- Đường đi này tới v và sau đó đi theo cung  $(v,w)$  để tới w (xem hình 10.9a). Khi đó đoạn đường từ 1 tới v là đường đi đặc biệt (đối với  $S$ ). Do đó

$$\delta(1,w) = \delta(1,v) + C[v,w] \geq D[v] + C[v,w] \geq D[w]$$

vì  $D[w]$  được xác định bởi lệnh (5)

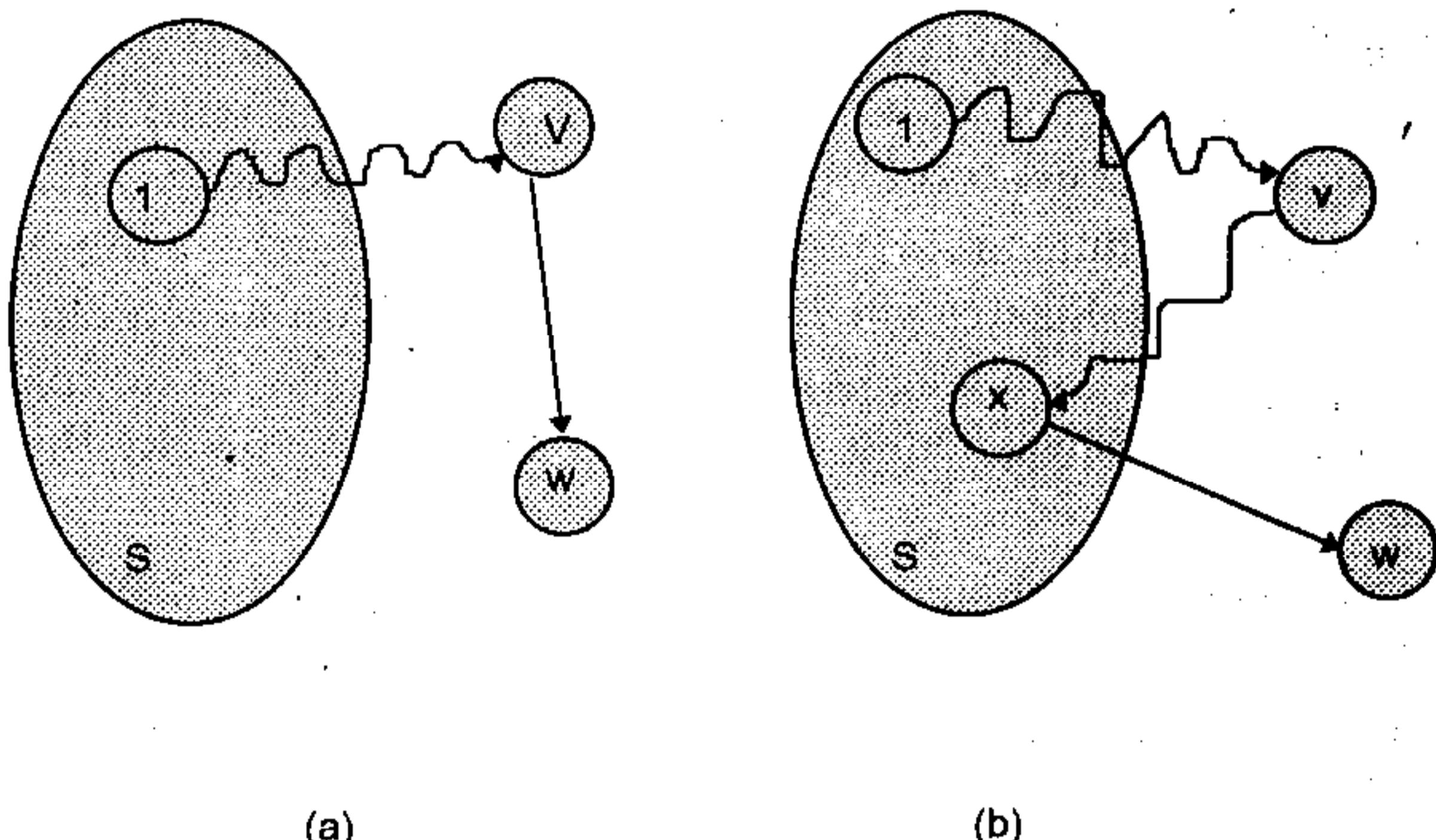
- Đường đi này qua v, từ v nó qua một số đỉnh thuộc  $S$  để tới w và đỉnh sau cùng nằm trong  $S$  là x (xem hình 10.9b). Gọi  $\delta(1,x)$  là độ dài đoạn đường từ 1 qua v tới x, ta có  $\delta(1,w) = \delta(1,x) + C[x,w]$ . Vì  $x \in S$ , nên theo giả thiết quy nạp  $D[x]$  là độ dài đường đi ngắn nhất từ 1 tới x. Do đó  $\delta(1,x) \geq D[x]$ .

$$\text{Ta có } \delta(1,w) = \delta(1,x) + C[x,w] \geq D[x] + C[x,w]$$

$$\geq D[w] \text{ (khi thêm } x \text{ vào tập } S\text{)}$$

$$\geq D[w] \text{ (khi thêm } v \text{ vào tập } S\text{)}$$

vì x được thêm vào  $S$  trước v và khi ta mở rộng tập  $S$  thì  $D[w]$  không tăng.



Hình 10.9. Các đường đi đặc biệt qua v

### *Phân tích thuật toán Dijkstra*

Vòng lặp for (1) đòi hỏi thời gian  $O(n)$ . Với  $S$  có  $i$  phần tử, thì  $V - S$  có  $n-i$  phần tử, do đó dòng lệnh (3) cần thời gian  $(n-i)O(1)$ . Dòng lệnh for (4) cũng cần thời gian  $(n-i)O(1)$ . Như vậy, vòng lặp while (2) đòi hỏi thời gian

$$(n-1)O(1) + (n-2)O(1) + \dots + O(1) = O(n^2)$$

Do đó thời gian thực hiện thuật toán Dijkstra là  $O(n^2)$ .

Trên đây ta đã đánh giá thời gian thực hiện lệnh (3) khi mà tập  $V - S$  được cài đặt bởi danh sách. Để cho lệnh (3) được thực hiện có hiệu quả, ta có thể cài đặt  $V - S$  bởi hàng ưu tiên theo  $D[w]$ . Khi đó dòng lệnh (3) được thực hiện bởi phép toán **DeleteMin** trên hàng ưu tiên. Khi thực hiện lệnh (5), đối với các  $D[w]$  được xác định lại và sẽ nhỏ đi, ta cần thực hiện phép toán đẩy lên (pushup) (tương tự như phép toán pushdown trong mục 9.4). Nếu cài đặt như thế, thời gian thực hiện thuật toán Dijkstra sẽ tốt hơn đáng kể.

### 10.6.2. Đường đi ngắn nhất giữa mọi cặp đỉnh: Thuật toán Floyd

Bây giờ chúng ta xét bài toán đường đi ngắn nhất giữa mọi cặp đỉnh trong đồ thị định hướng có trọng số. Cũng như trước đây, chúng ta giả sử rằng  $G = (V, E)$  là đồ thị định hướng với tập các đỉnh  $V = \{1, 2, \dots, n\}$  và ma trận giá của nó là  $C$ , trong đó  $C[i,j]$  là độ dài cung  $i \rightarrow j$ , nếu không có cung từ  $i$  đến  $j$  thì  $C[i,j] = \infty$  và  $C[i,i] = 0$  với mọi  $i = 1, 2, \dots, n$ . Đương nhiên chúng ta có thể giải quyết vấn đề này bằng cách sử dụng thuật toán Dijkstra với các đỉnh được chọn làm nguồn lần lượt là  $1, 2, \dots, n$ . Có một cách trực tiếp giải quyết vấn đề trên, đó là thuật toán được đưa ra bởi Floyd. Thuật toán này được thiết kế theo kỹ thuật quy hoạch động.

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 từ  $k$  tới  $j$  phải là đường đi ngắn nhất từ  $i$  tới  $k$  và từ  $k$  tới  $j$  tương ứng (nguyên lý tối ưu được thỏa mãn).

Chúng ta sử dụng ma trận  $A$  để lưu độ dài đường đi ngắn nhất giữa mọi cặp đỉnh. Ban đầu ta đặt  $A[i,j] = C[i,j]$ , tức là ban đầu  $A$  chứa độ dài

đường đi trực tiếp (không đi qua đỉnh nào cả). Sau đó ta thực hiện n lần lặp. Sau lần lặp thứ k, ma trận A sẽ chứa độ dài các đường đi ngắn nhất chỉ đi qua các đỉnh thuộc  $\{1, 2, \dots, k\}$ . Do đó, sau n lần lặp ta nhận được A chứa độ dài các đường đi ngắn nhất.

Ta ký hiệu  $A_k$  là ma trận A sau lần lặp thứ k, tức là  $A_k[i,j]$  là độ dài đường đi ngắn nhất từ i đến j chỉ đi qua các đỉnh thuộc  $\{1, 2, \dots, k\}$ .  $A_k[i,j]$  được tính bởi công thức sau

$$A_k[i,j] = \min (A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])$$

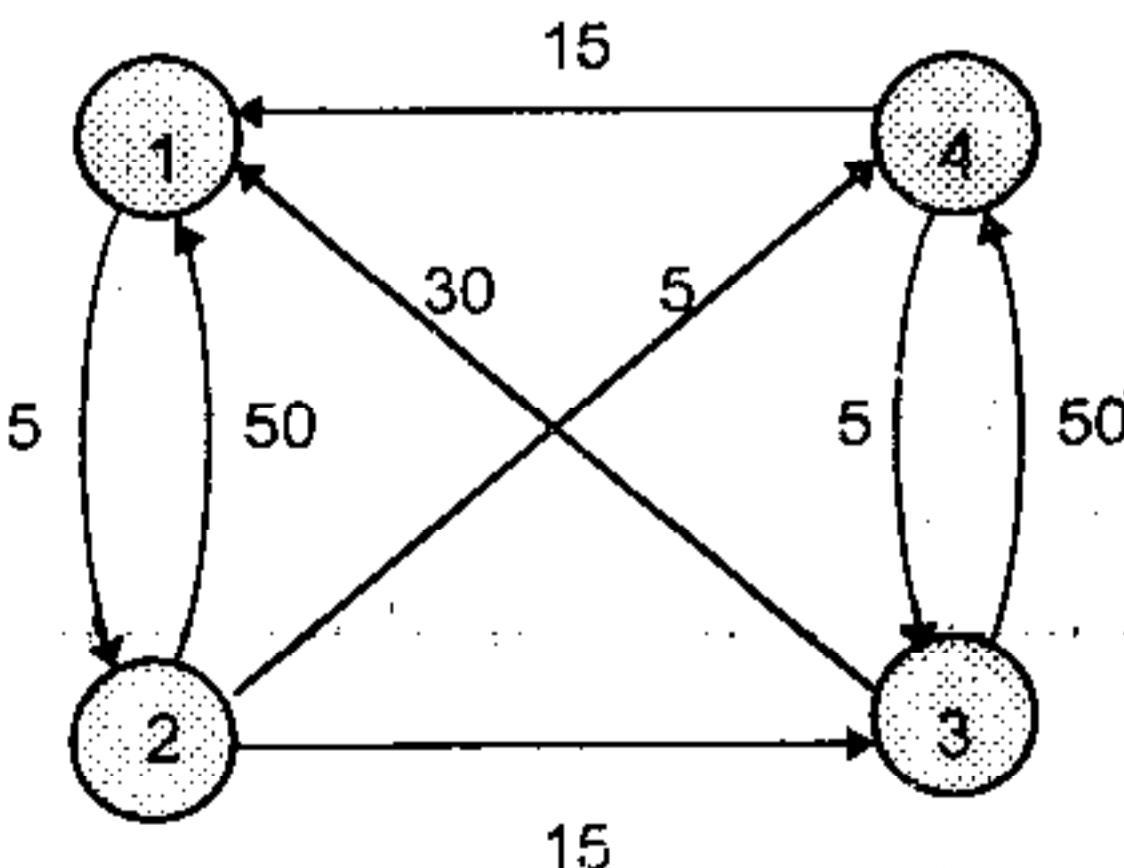
Ở lần lặp thứ k, các giá trị của ma trận A ở dòng thứ k và ở cột thứ k không thay đổi vì  $A[k,k]$  luôn luôn bằng 0. Do đó ta không cần quan tâm tới các giá trị đó khi cập nhật ma trận A ở lần lặp thứ k.

Thuật toán Floyd được mô tả bởi thủ tục sau

```

procedure Floyd;
var
    i, j, k: integer;
begin
    for i := 1 to n do
        for j := 1 to n do A[i,j] := C[i,j];
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                if A[i,k] + A[k,j] < A[i,j] then A[i,j] := A[i,k] + A[k,j];
end;
```

*Ví dụ.* Xét đồ thị sau



Ma trận biểu diễn bởi đồ thị trên là:

$$C = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

Ban đầu ma trận  $A = C$ . Sau mỗi lần lặp ( $k = 1, 2, 3, 4$ ), ma trận  $A$  trở thành các ma trận  $A_1, A_2, A_3, A_4$  sau

$$A_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad A_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad A_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Trong nhiều trường hợp, chúng ta muốn biết không chỉ độ dài của đường đi ngắn nhất mà còn cần biết đường đi ngắn nhất đi qua các đỉnh nào. Khi đó ta sử dụng mảng phụ  $P$ , trong đó  $P[i,j]$  lưu đỉnh  $k$  nếu đường đi ngắn nhất từ  $i$  đến  $j$  được tìm ra bởi thuật toán Floyd đi qua đỉnh  $k$ . Ban đầu,  $P[i,j] = 0$  với mọi  $i,j$ , vì lúc đó đường đi ngắn nhất là đường đi trực tiếp, không đi qua đỉnh nào cả. Như vậy, nếu dùng mảng  $A[1..n, 1..n]$  để lưu độ dài các đường đi ngắn nhất và mảng  $P[1..n, 1..n]$  để lưu vết của các đường đi ngắn nhất thì thuật toán Floyd được biểu diễn bởi thủ tục sau

```

procedure Floyd;
var
    i, j, k: integer;
begin
    for i := 1 to n do
        for j := 1 to n do
            begin
                A[i,j] := C[i,j];
                P[i,j] := 0;
            end;
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                if A[i,k] + A[k,j] < A[i,j] then
                    begin
                        A[i,j] := A[i,k] + A[k,j];
                        P[i,j] := k;
                    end;
    end;

```

**Ví dụ.** Áp dụng thủ tục trên vào đồ thị trong ví dụ trước, ta nhận được ma trận P như sau

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Từ ma trận này, ta dễ dàng xây dựng được đường đi từ đỉnh i đến đỉnh j ( $i, j = 1, 2, 3, 4$ ). Chẳng hạn, đường đi ngắn nhất từ đỉnh 1 đến đỉnh 3 được xây dựng như sau: Vì  $P[1,3] = 4$  nên đường đi ngắn nhất từ

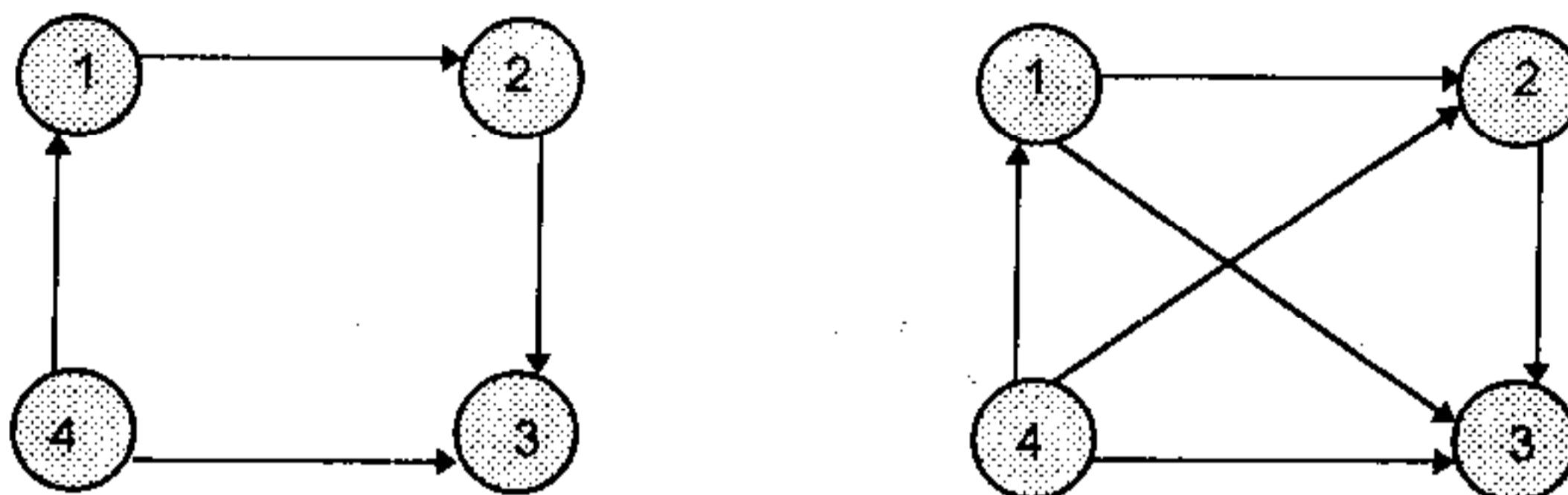
đỉnh 1 tới đỉnh 3 qua đỉnh 4. Tìm trong ma trận P ta có  $P[1,4] = 2$ ,  $P[3,4] = 0$  nên đường đi ngắn nhất từ 1 tới 4 qua 2, còn đường đi ngắn nhất từ 4 tới 3 là đường đi trực tiếp. Vậy đường đi ngắn nhất từ 1 tới 3 là (1, 2, 4, 3). Bạn đọc hãy tự viết thủ tục xây dựng đường đi ngắn nhất giữa mọi cặp đỉnh từ ma trận P được tìm ra bởi thuật toán Floyd.

Chúng ta dễ dàng thấy rằng, thuật toán Floyd đòi hỏi thời gian thực hiện là  $O(n^3)$  đối với đồ thị có n đỉnh.

### Bao đóng bắc cầu : Thuật toán Warshall

Trong nhiều vấn đề ta không quan tâm đến độ dài của đường đi, mà chỉ cần biết có hay không một đường đi từ đỉnh i tới đỉnh j.

Cho đồ thị định hướng  $G = (V, E)$  với tập đỉnh  $V = \{1, 2, \dots, n\}$  và ma trận kề C, trong đó  $C[i,j] = 1$  nếu có cung từ đỉnh i tới đỉnh j và  $C[i,j] = 0$  nếu không có cung từ đỉnh i tới đỉnh j. Chúng ta cần xác định ma trận A sao cho  $A[i,j] = 1$  nếu có đường đi từ i tới j và  $A[i,j] = 0$  nếu không. Ma trận A xác định một đồ thị định hướng  $G' = (V, E)$ . Ngoài các cung của đồ thị G, đồ thị  $G'$  còn được thêm vào các cung mới  $(u,v)$  nếu tồn tại đường đi trong G từ đỉnh u tới đỉnh v. Đồ thị  $G'$  được gọi là bao đóng bắc cầu của đồ thị G. Chẳng hạn, bao đóng bắc cầu của đồ thị trong hình 10.10a là đồ thị trong hình 10.10b.



$$C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

(a)

(b)

**Hình 10.10.** Bao đóng bắc cầu

Từ thuật toán Floyd, ta có thể đưa ra thuật toán tìm bao đóng bắc cầu sau đây. Thuật toán này mang tên là thuật toán Warshall. Trong thuật toán này, C và A là các mảng boolean.

$$C[i,j] = \begin{cases} \text{true} & \text{nếu có cung } i \rightarrow j \\ \text{false} & \text{nếu không} \end{cases}$$

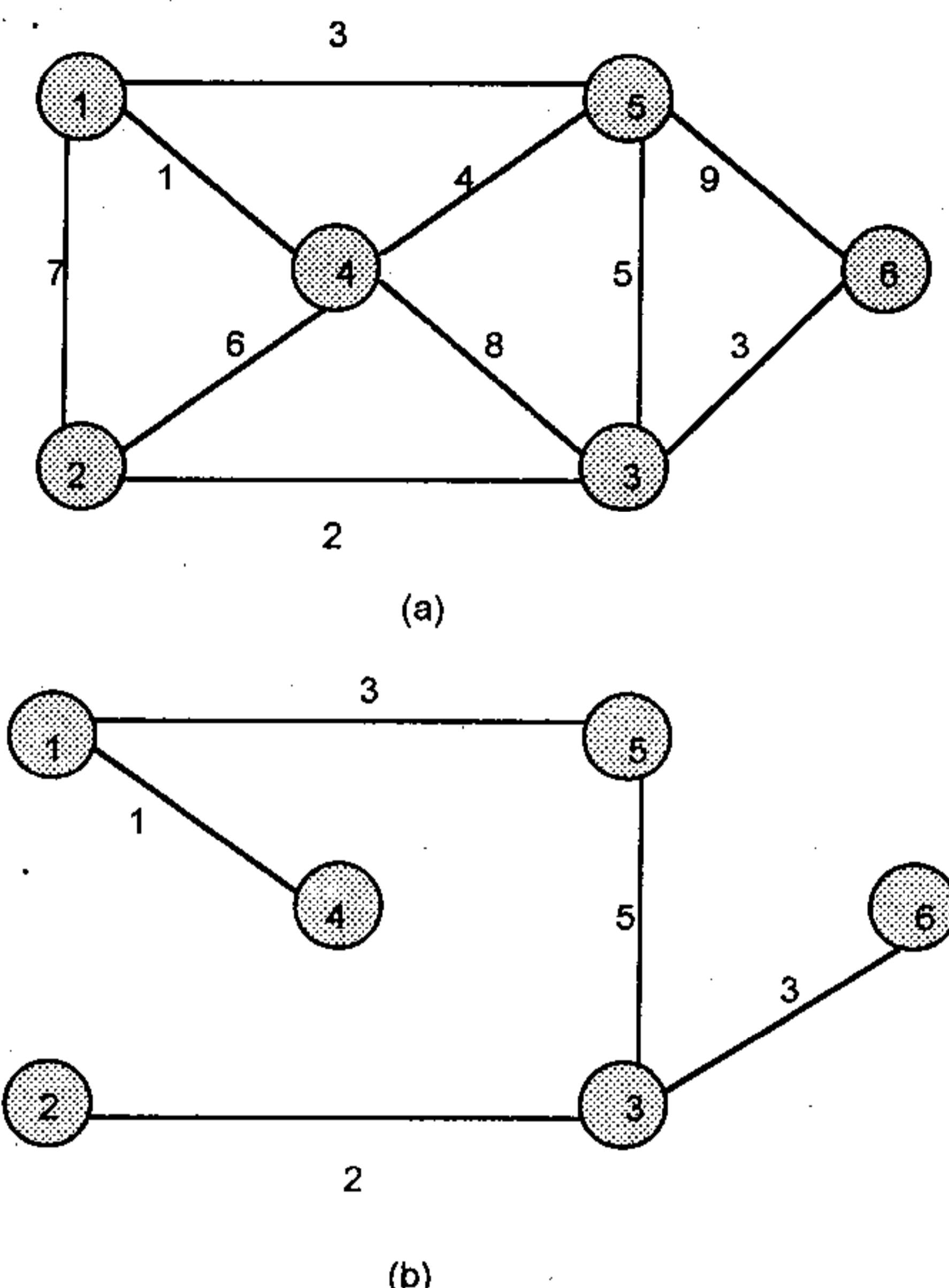
```

procedure Warshall;
var
    i, j, k: integer;
begin
    for i := 1 to n do
        for j := 1 to n do A[i,j] := C[i,j];
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                if A[i,j] = false then A[i,j] := A[i,k] and A[k,j];
end;
```

## 10.7. CÂY BAO TRÙM NGẮN NHẤT

Giả sử  $G = (V, E)$  là đồ thị vô hướng và mỗi cạnh  $(u,v) \in E$  có độ dài  $c(u,v) \geq 0$ . Giả sử  $G$  liên thông. Ta gọi  $T$  là cây bao trùm (hoặc cây khung) của đồ thị  $G$  nếu  $T$  là đồ thị con liên thông, không có chu trình và chứa tất cả các đỉnh của  $G$ . Độ dài của cây  $T$  được xem là tổng độ dài của tất cả các cạnh tạo thành cây  $T$ . Vấn đề của chúng ta là tìm cây bao trùm  $T$  của đồ thị  $G$  sao cho  $T$  có độ dài ngắn nhất.

**Ví dụ.** Đồ thị trong hình 10.11b là một cây bao trùm ngắn nhất của đồ thị trong hình 10.11a.

**Hình 10.11.** Một cây bao trùm ngắn nhất

Cây bao trùm ngắn nhất có nhiều ứng dụng, chẳng hạn trong việc thiết kế các mạng truyền thông. Trong trường hợp này, các đỉnh của đồ thị là các địa điểm, các cạnh biểu diễn các đường truyền thông nối các địa điểm. Giá của các cạnh là giá thi công các đường nối. Mạng truyền thông nối tất cả các địa điểm với giá thấp nhất chính là cây bao trùm ngắn nhất.

Chúng ta sẽ xét cây bao trùm  $T$  như là một tập các cạnh, chẳng hạn cây bao trùm trong hình 10.11b là tập các cạnh  $T = \{(1,4), (1,5), (2,3), (3,5), (3,6)\}$ . Do tính chất của cây, nên nếu đồ thị  $G$  có  $n$  đỉnh thì cây bao trùm có đúng  $n-1$  cạnh.

Các thuật toán tìm cây bao trùm ngắn nhất mà chúng ta sẽ trình bày đều được thiết kế theo kỹ thuật tham ăn. Chúng ta xây dựng tập

các cạnh T dần từng bước xuất phát từ T rỗng. Trong mỗi bước lặp, ta sẽ chọn một cạnh  $(u,v)$  “tốt nhất” trong các cạnh còn lại để đưa vào T. Có hai phương pháp chọn cạnh “tốt nhất” 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 cây. Còn trong phương pháp thứ hai (thuật toán Kruskal), cạnh được chọn ở mỗi bước là cạnh ngắn nhất không tạo thành chu trình với các cạnh đã chọn.

### 10.7.2. Thuật toán Prim

Trong thuật toán Prim, ta gọi U là tập đỉnh kề các cạnh trong T. Ban đầu U chứa một đỉnh tùy ý của G, còn tập các cạnh T rỗng. Ở mỗi bước, ta sẽ chọn cạnh  $(u,v)$  ngắn nhất sao cho  $u \in U$  và  $v \in V - U$ , rồi thêm v vào U và thêm  $(u,v)$  vào T. Điều này đảm bảo rằng sau mỗi bước T luôn luôn là một cây. Chúng ta tiếp tục phát triển cây T cho tới khi  $U = V$ , lúc đó T trở thành cây bao trùm của đồ thị G. Chúng ta biểu diễn thuật toán Prim bởi thủ tục sau

```

procedure Prim;
var
    U: tập các đỉnh;
    u, v : đỉnh;
begin
    U := {một đỉnh tùy ý};
    T := Ø;
repeat
    Chọn cạnh  $(u,v)$  ngắn nhất với  $u \in U, v \in V-U$ 
    U := U ∪ {v};
    T := T ∪ {(u,v)};
until U = V;
end;
```

Để minh họa thuật toán Prim làm việc như thế nào, ta xét đồ thị trong hình 10.11a. Ta chọn đỉnh xuất phát là đỉnh 1,  $U = \{1\}$ . Kết quả của các bước chọn cạnh  $(u,v)$  được cho trong bảng sau:

Bước	(u,v)	U
khởi tạo	-	{1}
1	(1,4)	{1, 4}
2	(1,5)	{1, 4, 5}
3	(5,3)	{1, 4, 5, 3}
4	(3,2)	{1, 4, 5, 3, 2}
5	(3,6)	{1, 4, 5, 3, 2, 6}

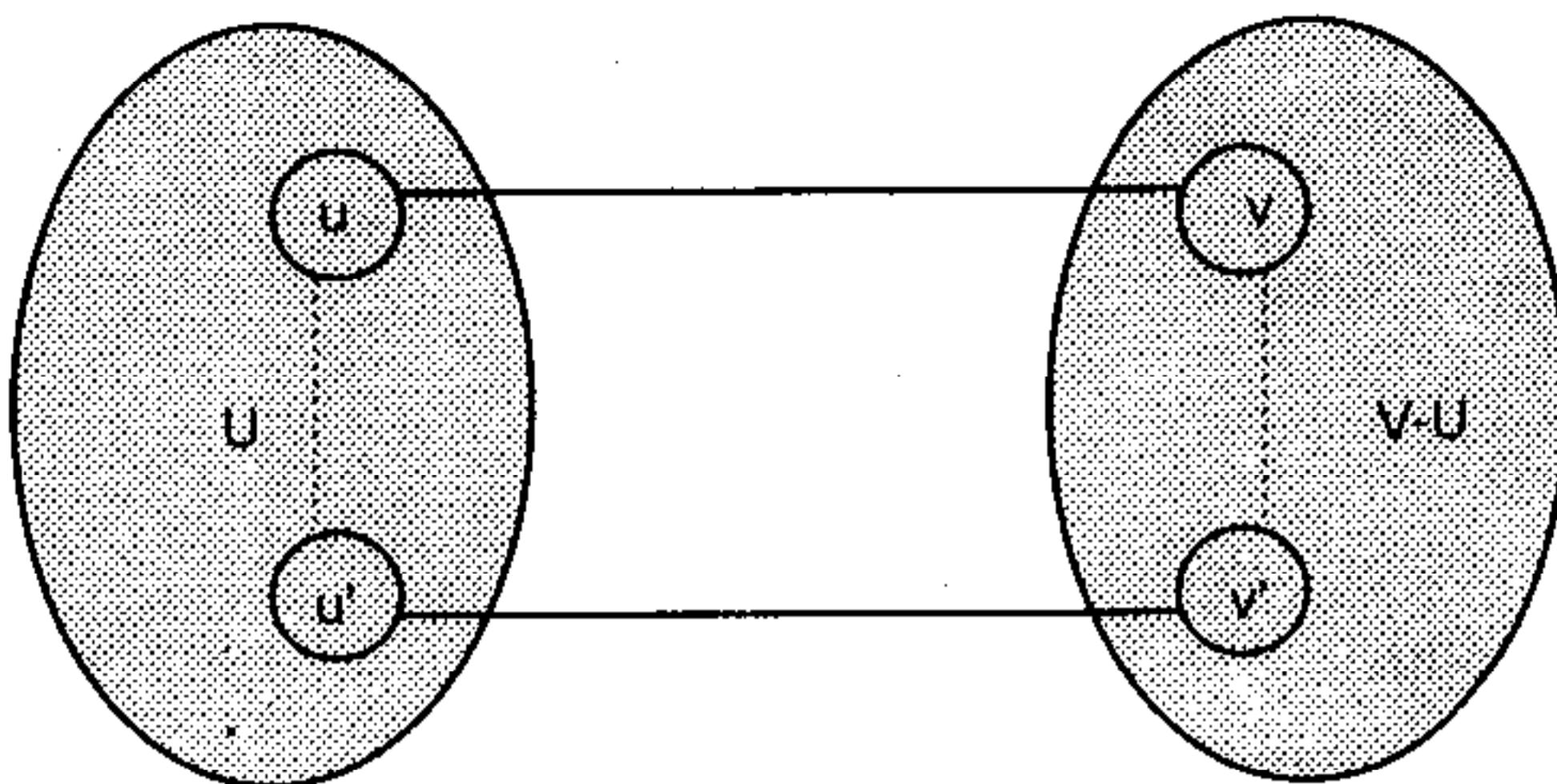
Do đó, cây T được xây dựng là cây trong hình 10.11b.

### ***Chứng minh tính đúng đắn của thuật toán Prim***

Chúng ta cần chứng minh rằng, cây T được xây dựng bởi thuật toán Prim là cây bao trùm ngắn nhất của đồ thị G. Trước hết ta chứng minh khẳng định sau

**Bổ đề.** Giả sử U là một tập con các đỉnh,  $U \neq V$ , và  $(u,v)$  là cạnh ngắn nhất của đồ thị G với  $u \in U$ ,  $v \in V-U$ . Khi đó, nếu  $T'$  là cây bao trùm ngắn nhất không chứa cạnh  $(u,v)$  thì ta có thể biến đổi  $T'$  thành cây bao trùm ngắn nhất  $T''$  chứa cạnh  $(u,v)$ .

Thật vậy, vì  $T'$  là cây bao trùm nên  $u$  và  $v$  là các đỉnh của cây  $T'$ . Nếu ta thêm cạnh  $(u,v)$  vào  $T'$  thì ta sẽ tạo ra một chu trình (đó là một tính chất của cây). Trên chu trình này, vì  $(u,v)$  là một cạnh nối tập đỉnh  $U$  với tập đỉnh  $V-U$ , do đó cần phải tồn tại một cạnh  $(u',v')$  nằm trên chu trình với  $u' \in U$  và  $v' \in V-U$  (xem hình 10.12). Nếu ta bỏ cạnh  $(u',v')$  đi thì ta nhận được cây bao trùm mới  $T''$ . Tức là  $T''$  là cây nhận được từ  $T'$  bằng cách bỏ đi cạnh  $(u',v')$  và thêm vào cạnh  $(u,v)$ . Vì  $c(u,v) \leq c(u',v')$ , nên độ dài của cây  $T''$  nhỏ hơn hoặc bằng độ dài cây  $T'$ . Do đó  $T''$  là cây bao trùm ngắn nhất và nó chứa cạnh  $(u,v)$ .



**Hình 10.12.** Một chu trình được tạo ra khi thêm cạnh ( $u, v$ ).

Bây giờ ta giả sử  $T'$  là cây bao trùm ngắn nhất, còn  $T$  là cây bao trùm được xây dựng bởi thuật toán Prim. Áp dụng bổ đề ta sẽ biến đổi cây  $T'$  thành cây  $T$  qua một dãy các phép biến đổi sau

$$T' = T_0 \rightarrow T_1 \rightarrow \dots \rightarrow T_i \rightarrow \dots T_m = T$$

Các phép biến đổi trong dãy lần lượt tương ứng với các bước chọn cạnh ( $u, v$ ) trong thuật toán Prim. Giả sử ở bước thứ nhất, thuật toán Prim chọn cạnh  $(u_1, v_1)$ , nếu  $T'$  không chứa  $(u_1, v_1)$  thì ta áp dụng bổ đề biến đổi  $T'$  thành  $T_1$  chứa  $(u_1, v_1)$ . Ở bước thứ hai, giả sử thuật toán chọn  $(u_2, v_2)$ , nếu  $T_1$  không chứa  $(u_2, v_2)$  thì ta lại áp dụng bổ đề biến đổi  $T_1$  thành  $T_2$  chứa  $(u_2, v_2)$ . Tiếp tục quá trình trên, cùng lầm là qua  $m$  phép biến đổi ( $m \leq n-1$ ,  $n$  là số đỉnh của  $G$ ) ta sẽ nhận được  $T_m$  chứa tất cả các cạnh như  $T$ , tức là  $T_m = T$ . Vì tất cả các cây  $T_i$  đều là cây bao trùm ngắn nhất, do đó  $T$  là cây bao trùm ngắn nhất.

### Cài đặt thuật toán Prim

Sau đây chúng ta đưa ra một cách cài đặt đơn giản thuật toán Prim. Giả sử  $G$  là đồ thị có  $n$  đỉnh,  $V = \{1, 2, \dots, n\}$  và ma trận giá  $C$  là ma trận đối xứng không âm,  $C[i,j] = \infty$  nếu không có cạnh nối  $i$  với  $j$ . Vấn đề đặt ra là ta phải sử dụng các CTDL như thế nào để thao tác mô tả trong thuật toán ở dòng (1) dễ dàng được thực hiện. Ta sử dụng hai mảng, mảng  $\text{nearest}[2..n]$  và mảng  $\text{dist}[2..n]$ . Với mỗi đỉnh  $v \in V - U$ , thì  $\text{nearest}[v]$  là đỉnh  $u \in U$  gần đỉnh  $v$  nhất và  $\text{dist}[v]$  là độ dài của cạnh nối đỉnh  $v \in V - U$  với đỉnh  $\text{nearest}[v]$ . Khi đó để thực hiện dòng lệnh (1)

trong thuật toán Prim, ta chỉ cần tìm min của các  $\text{dist}[v]$  với  $v \in V - U$ . Các đỉnh  $u \in U$  được đánh dấu bằng cách đặt  $\text{dist}[u] = -1$ . Bỏ qua các khai báo cần thiết ta có thủ tục sau

```

procedure Prim;
begin
    T :=  $\emptyset$ 
    for v := 2 to n do
        begin
            nearest[v] := 1;
            dist[v] := C[1,v];
        end;
        {khởi tạo hai mảng nearest, dist với U chỉ chứa đỉnh 1}
    (1) for i := 2 to n do {lặp lại n-1 lần các bước sau}
        begin
            min :=  $\infty$ ;
            (2) for k := 2 to n do
                if dist[k]  $\geq 0$  and dist[k] < min then
                    begin
                        min := dist[k];
                        v := k;
                        u := nearest[v];
                    end;
                    T := T U {(u,v)};
                    dist[v] := -1; {thêm v vào U}
                (3) for k := 2 to n do
                    if C[v,k] < dist[k] then
                        begin
                            dist[k] := C[v,k];
                            nearest[k] := v;
                        end;
                    {xác định lại dist[k] và nearest[k] sau khi đã thêm v vào U}
            end;
        end;

```

Chúng ta dễ dàng đánh giá thời gian thực hiện thủ tục trên. Vòng lặp (1) được thực hiện  $n-1$  lần. Trong mỗi lần lặp, cần thực hiện các vòng lặp (2) và (3). Các vòng lặp này đòi hỏi thời gian  $O(n)$ . Do đó thời gian thực hiện thuật toán Prim là  $O(n^2)$  với đồ thị n đỉnh.

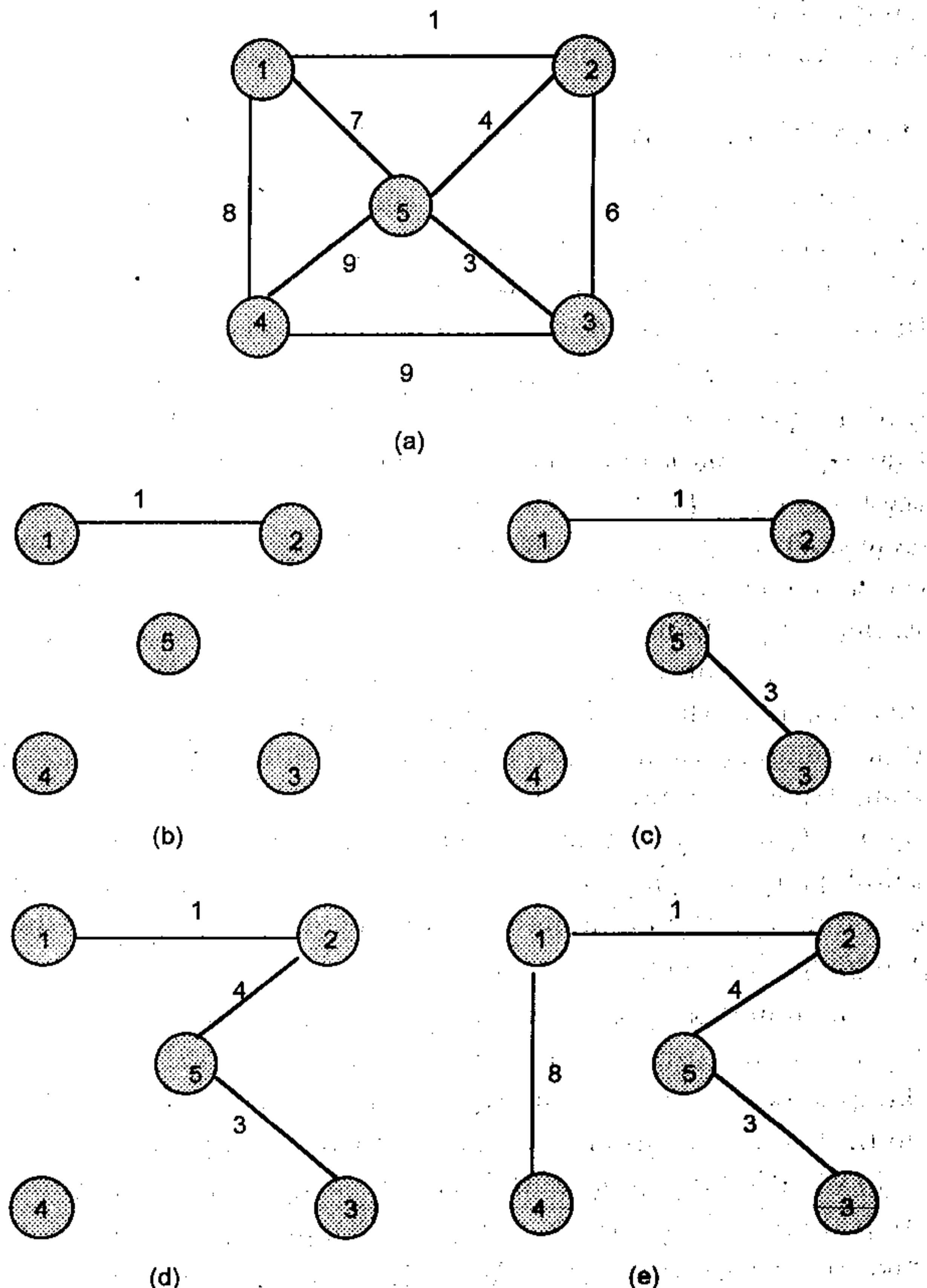
### 10.7.3. Thuật toán Kruskal

Trong thuật toán Kruskal, ta cũng xây dựng tập  $T$  các cạnh dồn từng bước, xuất phát từ  $T$  rỗng. Song khác với thuật toán Prim, tại mỗi bước ta sẽ chọn cạnh  $(u,v)$  ngắn nhất trong các cạnh còn lại sao cho khi thêm nó vào  $T$ , không có chu trình nào được tạo ra.

Xét đồ thị  $G' = (V, T)$ , tức là  $G'$  chứa các đỉnh như đồ thị  $G$ , nhưng tập các cạnh của  $G'$  là  $T \subset E$ . Ban đầu, khi  $T = \emptyset$  thì  $G' = (V, \emptyset)$  có  $n$  thành phần liên thông, mỗi thành phần chỉ chứa một đỉnh. Trong quá trình xây dựng  $T$  thì  $G' = (V, T)$  gồm một số thành phần liên thông. Việc chọn  $(u,v)$  trong mỗi bước để thêm vào  $T$  được thực hiện như sau: Ta xét các cạnh của đồ thị  $G$  theo thứ tự độ dài tăng dần. Với mỗi cạnh  $(u,v) \in E$ , nếu  $u$  và  $v$  nằm trong hai thành phần liên thông khác nhau của  $G'$  thì ta thêm cạnh  $(u,v)$  vào  $T$ , và khi đó hai thành phần liên thông này được hợp nhất thành một thành phần liên thông của  $G'$ . Còn nếu  $u$  và  $v$  thuộc cùng một thành phần liên thông thì ta loại bỏ cạnh  $(u,v)$  đó (bởi vì trong trường hợp này nếu ta thêm  $(u,v)$  vào  $T$  thì một chu trình sẽ được tạo ra). Quá trình trên được lặp lại cho tới khi nào  $G' = (V, T)$  chỉ có một thành phần liên thông và khi đó  $T$  là cây bao trùm. Cần lưu ý rằng, vì  $T$  có  $n$  đỉnh nên  $T$  chứa đúng  $n-1$  cạnh, do đó quá trình trên được lặp lại cho tới khi  $T$  chứa đủ  $n-1$  cạnh.

Để minh họa thuật toán Kruskal làm việc như thế nào, ta xét đồ thị  $G$  trong hình 10.12a. Danh sách các cạnh của đồ thị này theo thứ tự độ dài tăng dần là :  $(1,2), (3,5), (2,5), (2,3), (1,5), (1,4), (4,5), (3,4)$ . Ban đầu đồ thị  $G'$  chứa 5 đỉnh như  $G$ , nhưng không có cạnh nào. Các cạnh  $(1,2)$ ,  $(3,5)$  và  $(2,5)$  lần lượt được thêm vào  $G'$  để nhận được các đồ thị  $G'$  trong hình (b), (c) và (d), vì các đỉnh kề các cạnh trên đều thuộc các thành phần liên thông khác nhau của  $G'$ . Xét cạnh tiếp theo  $(2,3)$ , các đỉnh 2 và 3 cùng nằm trong một thành phần liên thông của đồ thị (d), loại  $(2,3)$ . Cạnh tiếp theo  $(1,5)$  cũng bị loại. Xét tiếp cạnh  $(1,4)$ , các đỉnh 1 và 4 nằm trong hai thành phần liên thông khác nhau của đồ thị (d), do đó

cạnh  $(1, 4)$  được thêm vào để nhận được đồ thị (e). Đồ thị (e) là đồ thị liên thông và nó là cây bao trùm ngắn nhất của đồ thị (a).



**Hình 10.12.** Quá trình hình thành cây bao trùm ngắn nhất  
theo thuật toán Kruskal

Chứng minh tính đúng đắn của thuật toán Kruskal được tiến hành theo cùng một phương pháp như chứng minh tính đúng đắn của thuật toán Prim. Chúng tôi để lại chứng minh này cho độc giả, xem như bài tập.

### **Cài đặt thuật toán Kruskal**

Trong thuật toán Kruskal, chúng ta cần xem xét các cạnh của đồ thị theo thứ tự độ dài tăng dần. Để thực hiện điều này, ta có thể sắp xếp các cạnh của đồ thị thành danh sách D theo thứ tự độ dài tăng dần. Sau đó ở mỗi lần lặp ta thực hiện phép toán Delete(D, (u,v)) để loại cạnh ngắn nhất (u,v) ở đầu hàng D và xét cạnh (u,v) này.

Khi xem xét một cạnh (u,v), chúng ta cần tìm thành phần liên thông chứa u và thành phần liên thông chứa v. Nếu hai thành phần này khác nhau, ta cần hợp nhất chúng thành một. Để các thao tác này dễ dàng thực hiện ta có thể cài đặt một cách đơn giản các thành phần liên thông của đồ thị  $G' = (V, T)$  như sau: Ta sử dụng mảng  $\text{comp}[1..n]$ , trong đó  $\text{comp}[i]$  là số hiệu của thành phần liên thông chứa đỉnh i. Ở đây số hiệu của một thành phần liên thông được lấy là số hiệu đỉnh nhỏ nhất trong thành phần liên thông đó. Chẳng hạn, số hiệu của thành phần liên thông gồm các đỉnh {2, 5, 7} là 2. Chúng ta dễ dàng viết được thủ tục hợp nhất thành phần liên thông chứa u và thành phần liên thông chứa v.

```

Procedure Merge(u, v);
begin
    a := comp[u];
    b := comp[v];
    if a > b then swap(a,b);
    for i := 1 to n do
        if comp[i] = b then comp[i] := a;
end;

```

Sử dụng thủ tục hợp nhất hai thành phần liên thông trên đây, ta có thể cài đặt thuật toán Kruskal bởi thủ tục sau

```

procedure Kruskal;
begin
    T := Ø;
    Sắp xếp các cạnh thành danh sách D theo thứ tự độ dài tăng dần;
    for i := 1 to n do comp[i] := i;
    {ban đầu mỗi đỉnh là một thành phần liên thông}
    k := 0; {k đếm số cạnh được thêm vào T}
    repeat
        delete(D, (u,v));
        if comp[u] <> comp[v] then
            begin
                merge(u,v);
                T := T U {(u,v)};
                k := k+1;
            end;
    until k = n-1;
end;

```

Đánh giá thời gian thực hiện thuật toán Kruskal trong cách cài đặt trên đây là công việc đơn giản, chúng tôi để lại việc này cho độc giả xem như bài tập.

## TÀI LIỆU THAM KHẢO

Tài liệu viết về CTDL và thuật toán rất phong phú, dưới đây chúng tôi chỉ liệt kê một số tài liệu chính mà chúng tôi đã tham khảo để viết cuốn sách này.

1. **Aho, J. E. Hopcroft, J. D. Ullman**  
Data Structures and Algorithms.  
Addison-Wesley, 1983
2. **G. Barassard and P. Bratley**  
Algorithmics: Theory and Practice.  
Prentice Hall, 1988.
3. **G. H. Gonnet**  
Handbook of Algorithms and Data Structures.  
Addison-Wesley, 1984
4. **L. Heileman**  
Data Structures, Algorithms, and Object-oriented Programming.  
McGraw-Hill ,1996
5. **E. Horowitz, S. Sahni**  
Fundamentals of Computer Algorithms.  
Computer Science Press, 1978
6. **D. E. Knuth**  
The Art of Computer Programming, volume 1, Fundamental Algorithms.  
Addison-Wesley, 1973.
7. **R. L. Kruse**  
Data Structures and Program Design.  
Prentice-Hall, 1984
8. **E. M. Reingold, J. Nievergelt, N. Deo**  
Combinatorial Algorithms: Theory and Practice.  
Prentice-Hall, 1977
9. **N. Wirth**  
Data Structures + Algorithms = Programs.  
Prentice-Hall, 1976.

## MỤC LỤC

Lời nói đầu	3
<b>Chương 1. Thuật toán và phân tích thuật toán</b>	5
1.1. Thuật toán	5
1.2. Phân tích thuật toán	10
<b>Chương 2. Kiểu dữ liệu, cấu trúc dữ liệu và mô hình dữ liệu</b>	23
2.1. Biểu diễn dữ liệu	23
2.2. Kiểu dữ liệu và cấu trúc dữ liệu	24
2.3. Hệ kiểu của ngôn ngữ Pascal	27
2.4. Mô hình dữ liệu và kiểu dữ liệu trừu tượng	31
<b>Chương 3. Danh sách</b>	36
3.1. Danh sách	36
3.2. Cài đặt danh sách bởi mảng	38
3.3. Tìm kiếm trên danh sách	42
3.4. Cấu trúc dữ liệu danh sách liên kết	46
3.5. Các dạng danh sách liên kết khác	53
3.6. Ứng dụng danh sách: Các phép tính số học trên đa thức	58
3.7. Ngăn xếp	63
3.8. Giá trị của một biểu thức	69
3.9. Hàng	74
<b>Chương 4. Cây</b>	83
4.1. Cây và các khái niệm về cây	83
4.2. Các phép toán trên cây	87
4.3. Cài đặt cây	93
4.4. Cây nhị phân	103
4.5. Cây tìm kiếm nhị phân	108
4.6. Thời gian thực hiện các phép toán trên cây tìm kiếm nhị phân	117
4.7. Cây cân bằng	119
4.8. Thời gian thực hiện các phép toán trên cây cân bằng	131
<b>Chương 5. Tập hợp</b>	134
5.1. Tập hợp và các phép toán trên tập hợp	134
5.2. Cài đặt tập hợp	138
5.3. Từ điển	144
5.4. Cấu trúc dữ liệu bảng băm. Cài đặt từ điển bởi bảng băm	146

5.5. Phân tích và đánh giá các phương pháp băm	156
5.6. Hàng ưu tiên	160
5.7. Cây thứ tự bộ phận và cài đặt ưu tiên bởi cây thứ tự bộ phận	161
<b>Chương 6. Bảng</b>	170
6.1. Kiểu dữ liệu trùu tượng bảng	170
6.2. Cài đặt bảng	171
6.3. Bảng chữ nhặt	174
6.4. Trò chơi đời sống	180
<b>Chương 7. Các cấu trúc dữ liệu ở bộ nhớ ngoài</b>	191
7.1. Mô hình tổ chức dữ liệu ở bộ nhớ ngoài	191
7.2. File băm	194
7.3. File có chỉ số (Indexed File)	196
7.4. B-Cây	199
<b>Chương 8. Các chiến lược thiết kế thuật toán</b>	207
8.1. Chia - để - trị	207
8.2. Phương pháp tham ăn	214
8.3. Quay lui	221
8.4. Nhánh và cận	227
8.5. Quy hoạch động	232
<b>Chương 9. Sắp xếp</b>	241
9.1. Vấn đề sắp xếp	241
9.2. Các phương pháp sắp xếp đơn giản	242
9.3. Sắp xếp nhanh	247
9.4. Sắp xếp sử dụng cây thứ tự bộ phận	253
<b>Chương 10. Các thuật toán trên đồ thị</b>	256
10.1. Một số khái niệm cơ bản	256
10.2. Biểu diễn đồ thị	258
10.3. Đi qua đồ thị	261
10.4. Thành phần liên thông và thành phần liên thông mạnh	269
10.5. Đồ thị định hướng không có chu trình. Sắp xếp topo	273
10.6. Đường đi ngắn nhất	275
10.7. Cây bao trùm ngắn nhất	285
<b>Tài liệu tham khảo</b>	295



**TS. ĐÌNH MẠNH TƯỜNG**

**CẤU TRÚC DỮ LIỆU  
và  
THUẬT TOÁN**

*Chịu trách nhiệm xuất bản* : PGS, TS **TÔ ĐĂNG HẢI**  
*Biên tập* : **ĐỖ THỊ CẨM**  
*Sửa bản in* : **LÊ MINH**  
*Trình bày bìa* : **HƯƠNG LAN**

**NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT  
70 TRẦN HƯNG ĐẠO - HÀ NỘI**

---

In 1.000 cuốn, khổ 16 x 24 cm, tại Xí nghiệp in 19 - 8 số 3 đường  
Nguyễn Phong Sắc - Nghĩa Tân - Cầu Giấy - Hà Nội.  
Giấy phép xuất bản số: 84 - 290, ngày 7 - 2 - 2001  
In xong và nộp lưu chiểu tháng 5 năm 2001.

# CẤU TRÚC DỮ LIỆU & THUẬT TOÁN

D

Thu vien - DHDL Hai Phong



2001DVL972

Gia: 35.000đ