TRƯỜNG ĐẠI HỌC SƯ PHẠM QUY NHƠN

KHOA TIN HỌC

TRẦN THIÊN THÀNH

Giáo trình

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Quy nhơn, 10/2002

LỜI NÓI ĐẦU

Cấu trúc dữ liệu và giải thuật là một môn học bắt buộc trong chương trình đào tạo cử nhân Tin học và Công nghệ thông tin. Giáo trình này được hình thành dựa trên nội dung giảng dạy nhiều năm tại khoa Tin học trường Đại học sư phạm Quy nhơn của tác giả.

Nội dung giáo trình gồm 6 chương:

Chương 1 trình bày một số kiến thức cơ bản về cấu trúc dữ liệu và giải thuật.

Chương 2 trình bày về mô hình dữ liệu danh sách. Trong chương cũng giới thiệu hai kiểu dữ liệu trừu tượng là Stack và Queue cùng với một số ứng dụng tiêu biểu.

Chương 3 trình bày về mô hình cây, chủ yếu tập trung vào cây tìm kiếm nhị phân, cây cân bằng và một số ứng dụng.

Chương 4 trình bày về mô hình đồ thị và một số thuật toán thường dùng trên đồ thị.

Chương 5 trình bày về cách tổ chức dữ liệu cho bộ nhớ ngoài.

Chương 6 trình bày các thuật toán sắp xếp trong và sắp xếp ngoài.

Giáo trình được soạn trên cơ sở chương trình đào tạo của Khoa. Một số kiến thức về thuật toán và kỹ thuật lập trình sinh viên đã được học trong các môn học trước đó nên không được đề cập trong giáo trình này. Giáo trình dùng làm tài liệu học tập cho sinh viên năm thứ ba hoặc học kỳ 2 của năm thứ hai ngành Tin học và Công nghệ thông tin với thời lượng 75 tiết. Ngoài ra, giáo trình có thể dùng cho sinh viên thuộc các ngành Toán học, Kỹ thuật và những người muốn có kiến thức sâu hơn về các cấu trúc dữ liệu thường dùng trong lập trình.

Trong mỗi chương của giáo trình, các kiến thức lý thuyết được trình bày cơ bản, rõ ràng, được minh hoạ chi tiết cùng với những ứng dụng cụ thể giúp cho người học dễ đọc, dễ hình dung những ứng dụng của các cấu trúc dữ liệu trong một số ứng dụng điển hình. Do đó giáo trình có thể dùng làm tài liệu tự học cho những người đã có những kiến thức cơ bản về thuật toán và lập trình trên một ngôn ngữ lập trình bậc cao. Nội dung trong giáo trình bám sát những nội dung cơ bản về các cấu trúc dữ liệu mà các chương trình đào tạo cử nhân Tin học và Công nghệ thông tin yêu cầu. Cuối mỗi chương đều cung cấp một hệ thống các bài tập từ cơ bản đến nâng cao nhằm giúp cho sinh viên rèn luyện tư duy, kỹ thuật lập trình và hiểu rõ hơn những nội dung lý thuyết.

Trong giáo trình sử dụng ngôn ngữ lập trình Pascal để minh hoạ các cấu trúc dữ liệu và thuật toán để giúp sinh viên dễ hình dung hơn trong cài đặt thành chương trình. Các cấu trúc dữ liệu được tổ chức dưới hình thức bao gói thông tin, mỗi cấu trúc dữ liệu được xem như một kiểu dữ liệu độc lập. Các thuật toán trình bày dưới dạng ngôn ngữ tự nhiên và được hoàn chỉnh bằng những thủ tục viết

bằng Pascal nên rất thuận tiện cho sinh viên trong thực hành bằng Pascal hay bất kỳ một ngôn ngữ lập trình bậc cao nào mà mình ưa thích.

Để hoàn thành giáo trình này tác giả đã nhận được nhiều ý kiến đóng góp và động viên của các đồng nghiệp, đặc biệt là ThS Hồ Anh Minh đã đọc bản thảo và đóng góp nhiều ý kiến quý báu.

Do thời gian và khả năng còn hạn chế nên giáo trình không thể tránh khỏi những khiếm khuyết nhất định. Chúng tôi chân thành và mong đón nhận những ý kiến đóng góp của độc giả.

Tác giả

MỤC LỤC

Chương 1

TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

1. TỔNG QUAN VỀ THUẬT TOÁN

1.1. Khái niệm thuật toán

Khái niệm thuật toán (Algorithm) xuất phát từ tên một nhà toán học Arập Abu Ja'far Mohamed ibn Musa al'Khwarizmi**,** thường gọi là al'Khwarizmi. Ông là tác giả một cuốn sách về số học, trong đó ông đã dùng phương pháp mô tả rất rõ ràng, mạch lạc cách giải những bài toán. Sau này, phương pháp mô tả cách giải của ông đã được xem là một chuẩn mực và được nhiều nhà toán học khác tuân theo. Thuật ngữ algorithm ra đời từ đó dựa theo cách phiên âm tên của ông. Cùng với thời gian khái niệm thuật toán được hoàn chỉnh dần và khái niệm hình thức chính xác của thuật toán được định nghĩa thông qua mô hình máy Turing. Giáo trình này không đi sâu vào những khía cạnh lý thuyết của thuật toán nên chỉ trình bày khái niệm không hình thức của thuật toán:

Thuật toán là một hệ thống chặt chẽ và rõ ràng các quy tắc nhằm xác định một dãy các thao tác trên những đối tượng sao cho sau một số hữu hạn bước thực hiện các thao tác thì đạt được mục tiêu định trước.

1.2. Các đặc trưng của thuật toán

Một thuật toán thông thường có 6 đặc trưng cơ bản sau:

1.2.1. Tính kết thúc (tính dừng)

Thuật toán bao giờ cũng phải dừng sau một số hữu hạn bước thực hiện.

1.2.2. Tính xác định

Thuật toán yêu cầu ở mỗi bước các thao tác phải hết sức rõ ràng, không gây nên sự nhập nhằng, lẫn lộn, tùy tiện. Khi thực hiện thuật toán, trong cùng một điều kiện thì cho cùng một kết quả.

1.2.3. Tính phổ dụng

Thuật toán phải có thể giải được một lớp các bài toán. Mỗi thuật toán có thể làm việc với những dữ liệu khác nhau trong một miền xác định.

1.2.4. Đại lượng vào

Mỗi thuật toán thường có những đại lượng vào gọi là dữ liệu vào để cung cấp dữ liệu cho thuật toán. Tuy nhiên, cũng có những thuật toán không có dữ liệu vào.

1.2.5. Đại lượng ra

Sau khi kết thúc thuật toán, tùy vào chức năng của thuật toán mà thu được một số đại lượng xác định gọi là đại lượng ra hay dữ liệu ra.

1.2.6. Tính hiệu quả

Với dữ liệu vào, sau một số hữu hạn bước thực hiện thuật toán sẽ dừng và cho đúng kết quả mong muốn với thời gian chấp nhận được.

1.3. Tiêu chuẩn đánh giá thuật toán

Một bài toán có thể có nhiều thuật toán giải, mỗi thuật toán có những ưu nhược điểm riêng. Để quyết định chọn thuật toán nào thông thường dựa vào 2 tiêu chuẩn cơ bản sau:

1. Thuật toán đơn giản, dễ hiểu, dễ cài đặt.
2. Thuật toán sử dụng tiết kiệm các tài nguyên của hệ thống máy tính như bộ nhớ, thời gian chiếm dụng CPU và đặc biệt là thời gian chạy.

Trong trường hợp chương trình ít sử dụng và giá viết chương trình vượt xa giá chạy chương trình thì tiêu chuẩn 1 được ưu tiên. Với những chương trình thường dùng như các thư viện, các chương trình hệ thống thì tiêu chuẩn 2 được ưu tiên chọn trước.

Trong tiêu chuẩn 2, tính hiệu quả của thuật toán bao gồm 2 yếu tố:

- Dung lượng không gian nhớ cần thiết để lưu các loại dữ liệu và các kết quả trung gian để giải bài toán (tổ chức dữ liệu cho bài toán).
- Thời gian cần thiết để thực hiện thuật toán (thời gian chạy).

Hai yếu tố trên thường mâu thuẫn nhau và ảnh hưởng qua lại lẫn nhau. Thường khi chọn thuật toán ta quan tâm đến thời gian thực hiện. Mặc dù hiện nay tốc độ máy tính ngày được cải thiện đáng kể, có thể thực hiện hàng trăm triệu phép tính trên giây nhưng vẫn chưa đáp ứng được cho một số thuật toán có thời gian chạy rất lớn.

1.4. Độ phức tạp của thuật toán

Việc đánh giá thời gian thực hiện của thuật toán phụ thuộc vào nhiều yếu tố:

- Dữ liệu vào.
- Tốc độ của máy tính.

- Chương trình dịch và hệ điều hành dùng cho chương trình.

Do đó việc đo, đếm chính xác thời gian thực hiện thuật toán là bao nhiêu đơn vị thời gian gần như không thể thực hiện được. Để có thể so sánh thời gian chạy của các thuật toán, trên phương diện lý thuyết thời gian thực hiện thuật toán được đánh giá là một hàm phụ thuộc vào kích thước của dữ liệu vào gọi là độ phức tạp thuật toán.

Để đánh giá độ phức tạp của thuật toán thông thường người ta tính số phép toán cơ bản thuật toán thực hiện. Các phép toán cơ bản thường dùng để đánh giá như các phép toán: +, -, *, /, các phép so sánh, phép gán, thao tác đọc, ghi file,... Tùy thuộc vào thuật toán, độ phức tạp là một hàm phụ thuộc vào kích thước của dữ liệu vào, ký hiệu T(n), với n là đại lượng đặc trưng cho kích thước của dữ liệu vào. Trong trường hợp thuật toán thực hiện nhiều phép toán cơ bản ta có thể đánh giá độ phức tạp trên từng loại phép toán hoặc tổng hợp của các phép toán. Chẳng hạn thuật toán sắp xếp thường được đánh giá trên 2 phép toán thường dùng là so sánh và phép gán.

Trong nhiều trường hợp, việc tính toán chính xác độ phức tạp thuật toán T(n) là không thể thực hiện được vì còn tùy thuộc vào sự phân bố của dữ liệu vào. Chẳng hạn thuật toán tìm một phần tử trong một danh sách cho trước không chỉ phụ thuộc vào số phần tử của danh sách mà còn phụ thuộc vào vị trí của phần tử cần tìm có trong danh sách hay không, nếu có thì phụ thuộc vào vị trí của phần tử do đó số phép so sánh phụ thuộc vào từng danh sách và phần tử cần tìm. Trong những trường hợp như thế này thông thường độ phức tạp được đánh giá trong trường hợp xấu nhất của dữ liệu vào. Trong một số tình huống cụ thể có thể tính trung bình hoặc tính theo xác suất.

Ví dụ 1: Thuật toán tìm một phần tử x trong danh sách L có n phần tử bằng cách tìm tuần tự.

```
TimThay:=False;
For i:=1 To n Do
If L[i] = x then
 begin
    TimThay:=True;
      Exit;
 end
```

Độ phức tạp được đánh giá qua số lần thực hiện phép so sánh L[i]=x trong trường hợp xấu nhất là không có phần tử cần tìm. Khi đó T(n) = n.

Ví dụ 2: Thuật toán sắp xếp dãy số a[1..n] tăng dần

```
For i:=1 to n-1 Do
  For j:=i+1 to n Do
  If a[i]>a[j] then
    Begin
       tg:=a[i];
       a[i]:=a[j];
```

```
    a[j]:=tg;
  End;
```

Độ phức tạp của thuật toán được đánh giá trên 2 phép toán cơ bản là phép so sánh trong biểu thức điều kiện của lệnh If và phép gán, ký hiệu tương ứng là C(n) và M(n). Độ phức tạp được đánh giá trong trường hợp "xấu" nhất của dữ liệu vào là dãy số ở tình trạng thứ tự giảm. Khi đó ta tính được:

Số phép so sánh C(n) = (n-1)n/2

Số phép gán M(n) = 3(n-1)n/2.

1.5. Ký hiệu O-lớn

Việc đánh giá độ phức tạp thuật toán qua hàm T(n) như trên quá chi tiết vào các phép toán thực hiện của thuật toán nên khó khăn trong việc so sánh và phân lớp các thuật toán. Để thể hiện bản chất hơn độ phức tạp của thuật toán phụ thuộc vào kích thước dữ liệu vào ta dùng khái niệm O-lớn (big oh) bằng cách bỏ qua các hằng trong độ phức tạp thuật toán.

Cho T(n), $f(n)$ là hai hàm. Ta nói T(n) là O-lớn của $f(n)$, ký hiệu T(n) = O($f(n)$), nếu và chỉ nếu tồn tại các hằng số dương c và số n_0 sao cho với mọi $n \geq n_0$ ta có T(n) $\leq c\, f(n)$.

Ví dụ: T(n) = $3n^2 + 2n$ - 10 thì T(n) = O(n^2).

Một số hàm thường dùng trong đánh giá độ phức tạp thuật toán qua ký hiệu O-lớn

Ký hiệu O-lớn	Tên gọi thường dùng
O(1)	Hằng
O(log$n$)	logarit
O(n)	tuyến tính
O($n$log$n$)	nlogn
O(n^2)	bình phương
O(2^n)	mũ

Quy 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($f_1(n)$,$f_2(n)$)).

2. KIỂU DỮ LIỆU VÀ CẤU TRÚC DỮ LIỆU

2.1. Kiểu dữ liệu

Mọi dữ liệu lưu trữ trên máy tính đều được biểu diễn dưới dạng các số nhị phân. Việc sử dụng trực tiếp các số nhị phân trên máy tính là một công việc khó

khăn cho người lập trình. Chính vì lý do này mà các ngôn ngữ lập trình cấp cao đã xây dựng nên các kiểu dữ liệu. Một kiểu dữ liệu là sự trừu tượng hóa các thuộc tính bản chất của các đối tượng trong thực tế và phù hợp với cách tổ chức thông tin trên máy tính, chẳng hạn như các kiểu số nguyên, số thực, logic,...

Một kiểu dữ liệu T là một bộ T = <*V*, *O*>, trong đó *V* là tập các giá trị hợp lệ của kiểu T và *O* là tập các phép toán trên kiểu T.

Ví dụ: Kiểu dữ liệu Byte = <V_{Byte}, O_{Byte}>,

với V_{Byte} = {0, 1, ..., 255}, O_{Byte} = {+, -, *, div, mod, >, >=, <, <=, =, <>}

2.2. Cấu trúc dữ liệu

Các kiểu dữ liệu cơ sở không đủ để mô tả các đối tượng của thế giới thực nên trong các ngôn ngữ lập trình bậc cao cho phép kết hợp các kiểu dữ liệu cơ sở để tạo nên một kiểu dữ liệu mới phức tạp hơn gọi là cấu trúc dữ liệu. Các kiểu dữ liệu cấu trúc thường dùng trên các ngôn ngữ lập trình bậc cao như: Array, String, Record, File,... là các cấu trúc dữ liệu thường dùng.

2.3. Mô hình dữ liệu

Các bài toán thực tế cần phải giải trên máy tính ngày càng phức tạp và đa dạng, do đó trước khi tổ chức các cấu trúc dữ liệu mô tả bài toán, người lập trình thường phải dùng các mô hình toán học để biểu diễn các đối tượng của bài toán và mối liên hệ giữa các đối tượng. Việc sử dụng các mô hình toán học cho phép người lập trình mô tả chính xác bản chất của các đối tượng trong bài toán và việc sử dụng toán học như một công cụ giúp cho việc giải các bài toán dễ dàng, chính xác hơn trước khi giải bài toán trên máy tính bằng chương trình. Mô hình toán học có thể biểu diễn được trên máy tính gọi là mô hình dữ liệu. Mô hình dữ liệu muốn cài đặt được trên máy tính phải có một cách tổ chức dữ liệu phù hợp. Các mô hình dữ liệu thường được sử dụng trong các bài toán tin học là: danh sách, cây, đồ thị, bảng băm, quan hệ, ...

2.4. Các tiêu chuẩn của cấu trúc dữ liệu

Khi tổ chức dữ liệu cho một bài toán thường dựa vào các tiêu chuẩn sau để lựa chọn cách tổ chức dữ liệu tối ưu.

Phản ánh đúng thực tế: đây là tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ quá trình giải bài toán. Trong khi tổ chức dữ liệu cũng dự tính các trạng thái biến đổi của dữ liệu trong tương lai để đảm bảo cho chương trình hoạt động được lâu dài.

Các thao tác phù hợp: mỗi cấu trúc dữ liệu có thể biểu diễn được một tập các đối tượng và có một tập các phép toán phù hợp. Việc chọn cách tổ chức dữ liệu không chỉ biểu diễn được các đối tượng của bài toán mà còn phải phù hợp với các thao tác trên đối tượng đó, có như vậy ta mới xây dựng được thuật toán giải bài toán đơn giản hơn.

Tiết kiệm tài nguyên hệ thống: khi tổ chức dữ liệu chỉ nên sử dụng tài nguyên hệ thống vừa đủ đáp ứng được yêu cầu công việc, tránh lãng phí. Có hai loại tài nguyên quan trọng của hệ thống là bộ nhớ và thời gian chiếm dụng CPU để thực hiện các thao tác trên dữ liệu. Thông thường hai loại tài nguyên này thường mâu thuẫn nhau trong khi giải các bài toán. Tuy nhiên nếu tổ chức khéo léo chúng ta cũng có thể tiết kiệm được cả hai loại tài nguyên.

3. MỐI LIÊN HỆ GIỮA CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Trong khi giải một bài toán, thông thường ta chỉ chú trọng đến giải thuật (hay cách giải của bài toán) mà ít khi quan tâm đến việc tổ chức dữ liệu. Tuy nhiên giữa việc tổ chức dữ liệu và thuật toán có mối liên hệ chặt chẽ nhau.

3.1. Mối liên hệ

Theo cách tiếp cận của lập trình cấu trúc, Niklaus Wirth đưa ra công thức thể hiện được mối liên hệ giữa cấu trúc dữ liệu và giải thuật:

CẤU TRÚC DỮ LIỆU + GIẢI THUẬT = CHƯƠNG TRÌNH
(Data structures + Algorithms = Programs)

Một thuật toán giải bài toán bao giờ cũng được thao tác trên một cấu trúc dữ liệu cụ thể và các thao tác phải được cấu trúc dữ liệu đó hỗ trợ.

Khi tổ chức dữ liệu cho bài toán thay đổi thì thuật toán giải cũng phải thay đổi theo cho phù hợp với cách tổ chức dữ liệu mới. Ngược lại, trong quá trình xây dựng, hoàn chỉnh thuật toán cũng gợi mở cho người lập trình cách tổ chức dữ liệu phù hợp với thuật toán và tiết kiệm tài nguyên hệ thống. Chẳng hạn dùng thêm các ô nhớ phụ để lưu các kết quả trung gian để giảm thời gian tính toán.

Quá trình giải một bài toán trên máy tính là một quá trình hoàn thiện dần cách tổ chức dữ liệu và thuật toán để tiết kiệm tài nguyên của hệ thống.

3.2. Một số ví dụ minh họa

Ví dụ 1. Xét bài toán đổi giá trị hai biến số *x*,*y*.

Với bài toán này ta có 2 phương án giải như sau:

Phương án 1. Dùng ô nhớ trung gian

```
tg := x;
x:= y;
y := tg;
```

Phương án 2. Không dùng biến trung gian.

```
x := x + y;
y := x - y;
x := x - y;
```

Qua ví dụ đơn giản trên, ta nhận thấy việc tổ chức dữ liệu khác nhau (dùng hay không dùng biến trung gian) ảnh hưởng rất lớn đến thuật toán và gần như thay

đổi toàn bộ thuật toán. Hơn nữa nó còn ảnh hưởng đến tính hiệu quả và phạm vi ứng dụng của thuật toán.

Ví dụ 2. Xét bài toán tính số tổ hợp chập *k* của *n* phần tử C_n^k.

Phương án 1. Dùng định nghĩa $C_n^k = \frac{n!}{k!(n-k)!}$. Giả sử gt(n) là hàm trả về *n*!. Ta có hàm tính hệ số C_n^k như sau:

```
Function  HeSo(n,k:word):Longint;
Begin
  HeSo := gt(n) div gt(k) div gt(n-k);
End;
```

- ***Nhận xét:*** *Với thuật toán này các chương trình chỉ tính được C_n^k với các số n, k nhỏ vì phải lưu các kết quả trung gian rất lớn là n!, k!, n-k!.*

Phương án 2. Dùng công thức $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$, với $C_n^0=1$, $C_i^i=1$. Hàm tính hệ số được viết dưới dạng đệ quy như sau.

```
Function  HeSo(n, k : word) : Longint;
Begin
  if (k=0) or (k=n) then
      HeSo:=1
  else
      HeSo := HeSo(n-1,k-1) + HeSo(n-1,k);
End;
```

- ***Nhận xét:*** *Với thuật toán này khắc phục việc phải lưu các giá trị giai thừa trung gian nhưng hạn chế của thuật toán là phải tính lại nhiều lần các giá trị đã tính ở bước trước, chẳng hạn để tính C_5^3 chương trình phải lặp lại 2 lần tính C_3^2, 3 lần tính C_2^1,....*

Phương án 3. Dùng một mảng hai chiều C[0..n,0..k] mỗi phần tử có kiểu LongInt để lưu các kết quả trung gian trong khi tính C_n^k, với C[i,j]=C_i^j (0≤j≤i,j≤k,i≤n). Từ công thức $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$, ta có C[i,j]=C[i-1,j-1]+C[i-1,j-1]. Bảng dưới minh hoạ mảng C dùng để tính C_5^3.

	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	**10**

```
Function  HeSo(n, k : word) : Longint;
Var i,j : word;
  Begin
```

```
    For i:=1 to n do C[i,0]:=1;
      For j:=1 to k do
        Begin
          C[j,j]:=1;
          For i:=j+1 to n do
            C[i,j]:=C[i-1,j-1]+C[i-1,j];
        End;
    HeSo:=C[n,k];
  End;
```

- ***Nhận xét:*** *phương án này còn nhiều ô nhớ của mảng không dùng, cụ thể là các ô nằm ở phần trên đường chéo chính. Vì mục đích của bài toán là tính giá trị* C_n^k *mà không cần các giá trị trung gian nên ta có thể tổ chức bộ nhớ tiết kiệm hơn bằng cách dùng một mảng 1 chiều.*

Phương án 4. Dùng mảng một chiều H[0..k] để lưu các giá trị của từng dòng trong mảng C của phương án trên. Mảng H được tính qua *n* bước, ở bước thứ *i*, H là dòng thứ *i* của mảng C, cụ thể tại bước thứ *i*, H[j]= C_i^j , với $j \le i$.

```
Function HeSo(n,k:Word):LongInt;
var H:array[0..1000] of LongInt;
  i,j : Word;
  Begin
    for i:=1 to k do H[i]:=0;
    H[0]:=1;
    for j:=1 to n do
      for i:=k downto 1 do
        H[i]:=H[i]+H[i-1];
    Heso:=H[k];
  End;
```

- ***Nhận xét:*** *Với phương án này vừa tiết kiệm được bộ nhớ vừa tăng khả năng tính toán với n, k lớn hơn các phương án khác.*

4. BÀI TẬP

Bài 1. Cho *n* điểm trong không gian 2 chiều. Cần tìm hình chữ nhật có các cạnh song song với các trục toạ độ chứa *n* đỉnh trên có diện tích nhỏ nhất. Hãy tổ chức dữ liệu, trình bày thuật toán và lập trình giải bài toán trên.

Bài 2. Cho một dãy số $a_1, a_2,...,a_n$. Hãy trình bày 2 thuật toán chuyển *k* phần tử đầu tiên ra cuối. Nghĩa là sau khi chuyển ta được dãy $a_{k+1}, .., a_n, a_1, ..., a_k$. Yêu cầu về tổ chức dữ liệu không được dùng mảng trung gian mà chỉ dùng một ô nhớ trung gian. Đánh giá độ phức tạp của thuật toán. Có thể cải tiến để có thuật toán tốt hơn về độ phức tạp không?.

Bài 3. Một danh sách học sinh gồm họ tên và điểm trung bình. Hãy tổ chức dữ liệu và trình bày thuật toán xếp thứ hạng cho các học sinh. Đánh giá độ phức tạp của thuật toán. Cài đặt bằng chương trình cụ thể.

Bài 4. Cho một dãy số nguyên, hãy trình bày thuật toán liệt kê các phần tử khác nhau của dãy số trên. Độ phức tạp của thuật toán? Cài đặt bằng chương trình? Có thể cải tiến thuật toán để đơn giản hơn không?.

Bài 5. Cần chia hết *m* phần thưởng cho *n* học sinh sắp theo thứ tự từ giỏi trở xuống sao cho mỗi học sinh không nhận ít phần thưởng hơn bạn xếp sau mình. Hãy đề xuất các cách tổ chức dữ liệu và trình bày thuật toán tính số cách chia thưởng, với mỗi cách phân tích những ưu, nhược điểm. Viết các thủ tục tương ứng cho từng cách tổ chức dữ liệu.

Chương 2

DANH SÁCH

1. KHÁI NIỆM VÀ CÁC THAO TÁC

1.1. Định nghĩa danh sách

Danh sách là một dãy hữu hạn các phần tử cùng loại được xếp theo một thứ tự tuyến tính.

Danh sách L gồm các phần tử $a_1, a_2, ..., a_n$ được ký hiệu: $L = (a_1, a_2, ..., a_n)$.

Trong đó n gọi là chiều dài của danh sách, a_i gọi là phần tử thứ i của danh sách. a_1 gọi là *phần tử đầu tiên* của danh sách, a_n gọi là *phần tử cuối cùng* của danh sách. Nếu $n = 0$ thì danh sách được gọi là rỗng.

Một tính chất quan trọng của danh sách là các phần tử được sắp xếp tuyến tính theo vị trí của chúng trong danh sách. Với $n>1$, $i = 1, 2,..., n-1$, phần tử a_i là *phần tử ngay trước* phần tử a_{i+1} và a_{i+1} là *phần tử ngay sau* phần tử a_i.

Trong một danh sách các phần tử có thể giống nhau.

Danh sách con

Cho $L = (a_1, a_2, ..., a_n)$ là một danh sách và i,j là các vị trí trong danh sách ($1 \le i \le j \le n$). Danh sách $L' = (b_1, b_2, ..., b_{j-i+1})$, trong đó $b_1 = a_i$, $b_2 = a_{i+1}$, ..., $b_{j-i+1}=a_j$ được gọi là danh sách con của danh sách L.

Dãy con

Danh sách L' được tạo thành từ danh sách L bằng cách bỏ đi một số phần tử của danh sách L nhưng vẫn giữ nguyên thứ tự được gọi là dãy con của danh sách L.

Ví dụ: $L = (1, 5, 2, 5, 7, 2)$, $L_1 = (5, 2, 5)$ là danh sách con của L, $L_2 = (2, 5, 7,2)$ là dãy con của danh sách L.

Trong thực tế có rất nhiều dữ liệu được tổ chức dưới dạng danh sách như danh sách nhân viên trong một cơ quan, danh sách các môn học, danh bạ điện thoại,...

1.2. Các thao tác trên danh sách

Tùy thuộc từng loại danh sách sẽ có các thao tác đặc trưng riêng. Trên danh sách thường thực hiện các thao tác cơ bản sau.

- Khởi tạo danh sách: tạo một danh sách rỗng.
- Thêm một phần tử vào danh sách.

- Loại bỏ một phần tử khỏi danh sách.
- Sắp thứ tự danh sách theo một khóa nào đó.
- Tìm kiếm một phần tử trong danh sách.
- Ghép nhiều danh sách thành một danh sách.
- Tách một danh sách thành nhiều danh sách.
- Sao chép một danh sách.
- ...

2. BIỂU DIỄN DANH SÁCH BẰNG MẢNG

Mảng là một cấu trúc dữ liệu cơ bản, thường dùng và được các ngôn ngữ lập trình cấp cao hỗ trợ. Mảng là một dãy cố định các ô nhớ chứa các phần tử cùng kiểu. Mỗi ô nhớ của mảng được xác định bởi chỉ số. Mô hình danh sách có những tính chất gần giống với cấu trúc dữ liệu kiểu mảng nên ta có thể dùng mảng để biểu diễn mô hình danh sách, trong đó các phần tử của danh sách được lưu vào các ô nhớ liên tục của mảng.

Điểm khác biệt giữa cấu trúc mảng và mô hình danh sách là số phần tử của mảng cố định trong khi số phần tử của danh sách thay đổi theo các thao tác thêm và xóa.

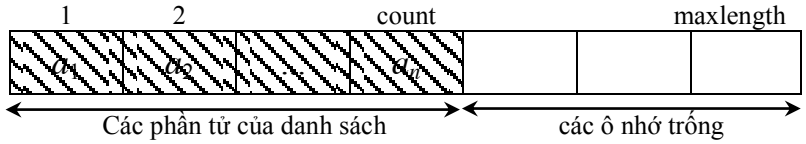
2.1. Tổ chức dữ liệu

Giả sử có danh sách $L=(a_1,a_2,...,a_n)$ trong đó mỗi phần tử có kiểu ElementType. Khi đó tổ chức dữ liệu kiểu mảng để lưu danh sách L gồm 2 thành phần:

+ Thành phần element là một mảng lưu các phần tử của danh sách.

+ Thành phần count là vị trí của ô nhớ lưu phần tử cuối cùng của danh sách và cũng là số phần tử hiện tại của danh sách.

Để đơn giản ta qui định các phần tử của mảng có chỉ số từ 1 đến maxlength, các phần tử của danh sách lưu vào mảng từ vị trí đầu tiên đến vị trí count. Khi đó các vị trí của mảng từ vị trí count+1 đến maxlength chưa sử dụng, những phần tử này sẽ được sử dụng khi thực hiện các thao tác thêm vào danh sách.



Hình 2.1 Tổ chức danh sách bằng mảng

Khai báo một danh sách trong Pascal có dạng:

Const

```
MaxLength = ... ; {Số phần tử tối đa của danh sách}

Type

   ElementType = ... ;{Định nghĩa kiểu phần tử của danh sách}

   ListArr = Record
                element : Array[1..MaxLength] Of ElementType;
                count : 0..MaxLength;
             End;

Var  L : ListArr;
```

Ví dụ: Khai báo danh bạ điện thoại gồm họ tên, địa chỉ, số điện thoại.

```
Const
   MaxLength = 100 ;

Type
   DIENTHOAI =  Record
                   Họ_tên : String[30];
                   Địa_Chỉ: String[30];
                   Số_ĐT : String[10];
                End;

   DANHBA =  Record
                element: Array[1..MaxLength] Of DIENTHOAI;
                Count    : 0..MaxLength;
             End;

Var  db : DANHBA;
```

2.2. Các thao tác trên danh sách

2.2.1. Khởi tạo danh sách

Số phần tử của danh sách được lưu vào thành phần count nên để khởi tạo danh sách rỗng ta chỉ cần thực hiện phép gán count := 0.

```
Procedure Init(var l : ListArr);
   Begin
      l.count := 0;
   End;
```

2.2.2. Kiểm tra danh sách rỗng

```
Function Empty(l : ListArr):Boolean;
   Begin
      Empty := l.count = 0;
   End;
```

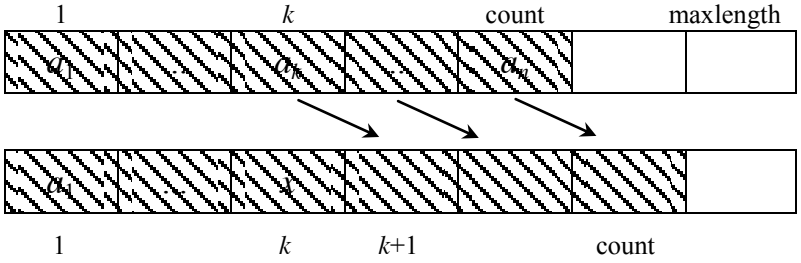
2.2.3. Kiểm tra danh sách đầy

Khi biểu diễn danh sách bằng mảng sẽ phải khống chế số lượng tối đa các phần tử của danh sách. Do đó có thể đến một lúc nào đó không đủ ô nhớ để thêm các phần tử vào danh sách. Trong trường hợp này gọi là danh sách đầy. Như vậy danh sách đầy khi số phần tử của danh sách bằng kích thước của mảng.

```
Function Full(l : ListArr):Boolean;
  Begin
    Full := l.count = maxlength;
  End;
```

2.2.4. Thêm một phần tử vào danh sách

Cho danh sách *L*, cần thêm vào trước phần tử thứ *k* trong danh sách một phần tử *x*.



Hình 2.2 Thêm một phần tử vào danh sách

Thuật toán:

+ Di chuyển các phần tử từ vị trí thứ k đến cuối danh sách ra sau một vị trí.

+ Đưa phần tử cần thêm x vào vị trí k.

+ Tăng thành phần count lên 1.

```
Procedure  Insert(var L:ListArr; x:ElementType;
                                                  k:1..maxlength);
  var i:1..maxlength;
  Begin
    If (k <= L.count+1) and (k>0) and not Full(L) then
    Begin
          For i:= L.count DownTo k Do
              L.element[i+1] := L.element[i];
          L.element[k]:=x;
          L.count := L.count + 1;
    End;
  End;
```

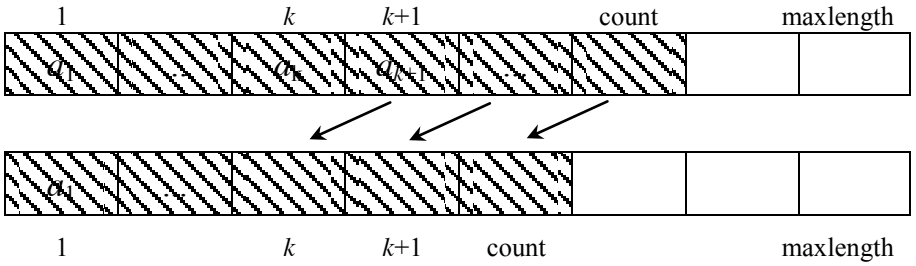
2.2.5. Loại bỏ một phần tử khỏi danh sách

Giả sử cần xóa phần tử thứ *k* trong danh sách *L*.

Thuật toán:

+ Dồn các phần tử từ vị trí k+1 đến cuối danh sách về trước một vị trí.

+ Giảm số phần tử của danh sách đi 1.



Hình 2.3 Xoá phần tử thứ k trong danh sách

```
Procedure   Delete(var L : ListArr; k : 1..maxlength);
var i : 1..maxlength;
  Begin
     If (k <= L.count) and (k>0) and not Empty(L) then
     Begin
           For i:= k To L.count-1 Do
              L.element[i] := L.element[i+1];
           L.count := L.count - 1;
     End;
  End;
```

Với danh sách có *n* phần tử, dễ thấy độ phức tạp thuật toán thêm một phần tử và thuật toán xóa một phần tử có độ phức tạp là O(*n*).

2.2.6. Ghép 2 danh sách

Cho hai danh sách L_1 và L_2. Ghép 2 danh sách này thành danh sách *L*.

```
Procedure Concat(L1, L2 : ListArr; Var L : ListArray);
var i : 1..maxlength;
   Begin
      For i:=1 To L1.count Do
            L.element[i] := L1.element[i];
      For i:=1 To L2.count Do
            L.element[i+L1.count] := L2.element[i];
      L.count := L1.count + L2.count;
   End;
```

2.2.7. Sắp xếp danh sách

Cho danh sách *L*, sắp xếp danh sách theo thứ tự tăng của trường khóa Key (Key là một trường trong kiểu dữ liệu phần tử của danh sách).

Có nhiều thuật toán sắp xếp danh sách tổ chức bằng mảng. Trong phần này trình bày thuật toán sắp xếp bằng chọn trực tiếp.

Thuật toán:

Duyệt lần lượt từng phần tử của danh sách, với phần tử thứ i thực hiện:

+ Tìm phần tử ở vị trí k có khóa nhỏ nhất trong danh sách con L[i..count]

+ Đổi giá trị phần tử thứ k với phần tử thứ i.

```
Procedure  Sort(Var L : ListArr);
var i, j, k : 1..maxlength; tg : ElementType;
   Begin
      For i:=1 To L.count-1 Do
         Begin
            k := i;
            For j := i+1 To L.count Do
               If L.element[k].Key > L.element[j].Key then
                  k := j;
            tg := L.element[i];
            L.element[i] := L.element[k];
            L.element[k] := tg;
         End;
   End;
```

2.2.8. Tìm kiếm trong danh sách

Cho danh sách *L*, tìm trong danh sách phần tử có khóa của trường Key là *x*.

Thuật toán tìm kiếm tuần tự:

Duyệt lần lượt từng phần tử của danh sách, với mỗi phần tử thực hiện:

Nếu phần tử thứ i có khóa trùng với x thì phần tử thứ i là phần tử cần tìm, dừng thuật toán và kết quả tìm thấy.

Nếu đã duyệt hết danh sách thì kết luận không tìm thấy.

Thủ tục Locate tìm trong danh sách *L* một phần tử có khóa là *x*. Kết quả nếu tìm thấy được trả về qua tham biến found kiểu boolean và vị trí của phần tử tìm được đầu tiên qua tham biến id.

```
Procedure Locate(L : ListArr; x: KeyType; var found :
                                                  Boolean; var id : 0..maxlenght);
   Begin
      id:=1; Found:=False;
      While (id<=L.count) and not found Do
         begin
            If L.element[id].Key = x then
               Found:=True
            else
               Id:=id+1;
         end;
   End;
```

Độ phức tạp thuật toán tìm kiếm tuần tự là O(n) với n là số phần tử của danh sách. Bài toán tìm kiếm là một bài toán thường dùng trong tin học. Để giảm độ phức tạp thuật toán tìm kiếm ta có thể dùng thuật toán tìm nhị phân. Yêu cầu để thực hiện được thao tác tìm nhị phân là danh sách phải được sắp xếp trên trường khóa cần tìm.

Giả sử danh sách L đã sắp xếp theo thứ tự tăng của trường Key.

Thuật toán tìm kiếm nhị phân:

Lặp lại trong khi danh sách còn ít nhất một phần tử, mỗi lần lặp thực hiện:

- So sánh khóa cần tìm với phần tử ở vị trí giữa của danh sách:

  + Nếu khóa phần tử giữa lớn hơn khóa cần tìm thì tìm trong nửa đầu của danh sách.

  + Nếu khóa phần tử giữa nhỏ hơn khóa cần tìm thì tìm trong nửa sau của danh sách.

  + Nếu khóa phần tử giữa bằng khóa cần tìm thì thuật toán kết thúc và kết quả tìm thấy.

Nếu danh sách không có phần tử nào thì kết quả không tìm thấy.

```
Procedrue Seek(L:ListArr;x:KeyType;var found:Boolean;
                                                    var id : Integer);
var left, right : 1..maxlength;
   Begin
      left:=1; right:=L.count; found:=false;
      While (left <= right) and (not found) Do
         Begin
            id := (left + right) div 2;
            if L.element[id].Key > x then right:=id-1;
            if L.element[id].Key < x then left :=id+1;
            if L.element[id].Key = x then found:=true;
         End;
   End;
```

Vì sau mỗi lần so sánh, số phần tử trong danh sách cần tìm giảm đi một nửa nên độ phức tạp thuật toán tìm nhị phân được đánh giá trong trường hợp xấu nhất là O($\log_2 n$).

2.2.9. Nhận xét về cách biểu diễn danh sách bằng mảng

Với cách tổ chức dữ liệu cho mô hình danh sách bằng mảng như trên ta có một số nhận xét về ưu, nhược điểm của cách tổ chức dữ liệu này như sau:

- Tổ chức danh sách bằng mảng thuận lợi khi truy xuất trực tiếp các phần tử của danh sách theo vị trí. Điều này thuận lợi cho một số thuật toán như tìm nhị phân.
- Khi dùng mảng phải cố định kích thước, trong khi đó các thao tác trên danh sách luôn làm cho số phần tử của danh sách thay đổi. Điều này dẫn đến hai xu hướng: nếu khai báo mảng với kích thước lớn thì gây lãng

phí vì nhiều ô nhớ không sử dụng hoặc khai báo kích thước nhỏ để ít lãng phí thì sẽ mau chóng đầy danh sách khi thêm.

- Các thao tác thêm, xóa cài đặt trên danh sách tổ chức bằng mảng có độ phức tạp là O(*n*) nên không phù hợp với các danh sách thường xuyên sử dụng các thao tác này.

3. DANH SÁCH LIÊN KẾT ĐƠN

Một trong những đặc trưng của danh sách là số phần tử không cố định mà thay đổi tùy thuộc vào thao tác trên nó. Điều này buộc cách tổ chức dữ liệu biểu diễn cho mô hình danh sách cũng phải có đặc trưng này, nghĩa là bộ nhớ phải được cấp phát động (dùng mảng không đáp ứng được yêu cầu này). Mặc khác trong danh sách các phần tử được sắp xếp tuyến tính do đó việc tổ chức dữ liệu cấp phát động phải được tổ chức sao cho thể hiện được thứ tự tuyến tính của các phần tử trong danh sách, một trong những cách thường dùng là danh sách liên kết đơn. Trong danh sách liên kết đơn mỗi phần tử phải quản lý địa chỉ ô nhớ lưu phần tử ngay sau nó. Để thuận lợi cho các thao tác với bộ nhớ cấp phát động, trong phần sau nhắc lại một số thao tác với bộ nhớ cấp phát động thông qua biến con trỏ của Pascal.

3.1. Cấp phát động, biến con trỏ và các thao tác

Ô nhớ cấp phát động là những ô nhớ được cấp phát và thu hồi bằng lệnh trong chương trình. Để quản lý các ô nhớ cập phát động cần sử dụng biến kiểu con trỏ. Biến con trỏ chứa địa chỉ của ô nhớ động được cấp phát. Mỗi biến con trỏ chỉ có thể quản lý một ô nhớ cấp phát động có kiểu xác định.

Khai báo kiểu và biến con trỏ:

```
Type  Kiểu_Con_trỏ = ^Kiểu_dữ_liệu;

Var       Biến_con_trỏ : ^Kiểu_dữ_liệu;
```

Cấp phát ô nhớ cho biến con trỏ:

```
New(biến_con_trỏ);
```

Thủ tục này cấp phát một ô nhớ đủ dùng để chứa dữ liệu của kiểu dữ liệu mà biến con trỏ trỏ đến và biến con trỏ trỏ đến ô nhớ này.

Sử dụng ô nhớ do biến con trỏ quản lý:

Mặc dù biến con trỏ chỉ quản lý địa chỉ của ô nhớ cấp phát động, tuy nhiên trong trường hợp cần thao tác với nội dung của ô nhớ ta có thể dùng cú pháp:

```
Biến_con_trỏ^
```

Giả sử có biến con trỏ p trỏ đến một ô nhớ kiểu số nguyên. (Var p:^Integer;) và đã cấp phát ô nhớ cho con trỏ p (New(p)). Muốn thao tác với nội dung của ô nhớ này ta dùng cú pháp p^ và sử dụng như một biến nguyên thuần túy. Chẳng hạn để chứa số 5 vào ô nhớ này ta có thể gán p^:=5;

Phép gán con trỏ:

Ta có thể thực hiện thao tác gán cho biến con trỏ p:=q; (p,q là 2 biến con trỏ cùng kiểu). Khi đó p và q sẽ cùng trỏ vào một ô nhớ do biến con trỏ q đang quản lý.

Hằng con trỏ Nil dùng để gán cho con trỏ có kiểu bất kỳ thường dùng khi mà chưa xác định địa chỉ mà biến con trỏ quản lý.

Thu hồi ô nhớ của một biến con trỏ:

Khi cần thu hồi ô nhớ do biến con trỏ quản lý ta dùng thủ tục Dispose.

```
Dispose(biến_con_trỏ);
```

Khi đó vùng nhớ mà biến con trỏ p quản lý đã được thu hồi và có thể dùng cho việc khác. Sau khi thu hồi ô nhớ của một biến con trỏ thì nên gán cho nó giá trị nil để tránh những sai sót khi thao tác với biến con trỏ này.

Ta cũng có thể thu hồi một số ô nhớ cấp phát bằng cách dùng cặp thủ tục Mark(p) và Release(p). Trong đó thủ tục Mark(p) (với p là một biến con trỏ) dùng để đánh dấu vị trí đầu của một vùng nhớ mà khi cần có thể thu hồi lại. Thủ tục Release(p) thu hồi vùng nhớ bắt đầu từ vị trí được đánh dấu bằng lệnh Mark(p).

3.2. Khái niệm danh sách liên kết

Danh sách liên kết là một cách tổ chức dữ liệu cho mô hình danh sách trong đó các phần tử được liên hệ với nhau nhờ vào vùng liên kết.

Danh sách liên kết sử dụng cơ chế cấp phát động nên thích hợp với các thao tác thêm vào, loại bỏ, ghép nhiều danh sách.

3.3. Tổ chức danh sách liên kết

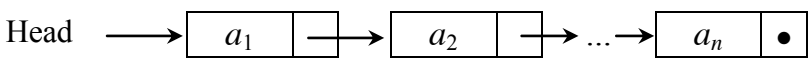
Mỗi phần tử của danh sách liên kết gồm hai thành phần:

+ Phần Data chứa dữ liệu thực sự của từng phần tử trong danh sách.

+ Phần Link dùng để liên kết một phần tử với phần tử ngay sau nó.

Từ một phần tử ta chỉ duy trì một liên kết đến phần tử ngay sau nó nên danh sách liên kết được tổ chức như vậy được gọi là danh sách liên kết đơn. Trong phần này ta chỉ xét cấu trúc dữ liệu danh sách liên kết đơn nên không gây nhầm lẫn khi ta gọi là danh sách liên kết.

Hình ảnh một danh sách liên kết biểu diễn danh sách $L = (a_1, a_2, ..., a_n)$ như sau.



Hình 2.4 Hình ảnh danh sách liên kết đơn

Để quản lý danh sách biểu diễn bởi danh sách liên kết ta chỉ cần quản lý phần tử đầu tiên của danh sách. Từ phần tử này ta có thể thao tác được với các phần tử của danh sách nhờ liên kết giữa các phần tử. Khai báo danh sách liên kết có dạng:

```
Type
    ElementType =...;{Định nghĩa kiểu phần tử của danh sách}

    ListLink = ^Cell;

    Cell = Record
              Data : ElementType;
              Link : ListLink;
        End;

Var  Head : ListLink;
```

Ví dụ : Tổ chức danh sách nhân viên kiểu danh sách liên kết như sau:

```
Type DSCB = ^HSCB;

     HSCB =Record
              Ho_Ten : String[30];
              NamSinh : 1900..3000;
              Lk : DSCB;
           End;

Var  ds1, ds2: DSCB;
```

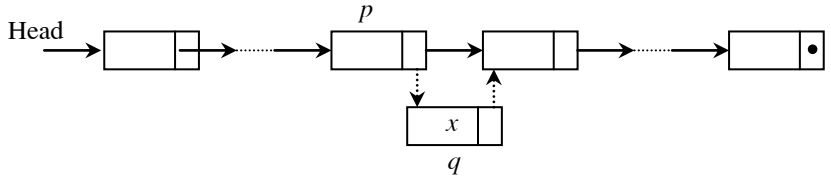
3.4. Các phép toán trên danh sách liên kết

3.4.1. Khởi tạo danh sách liên kết

```
Procedure Init(var Head : ListLink);
  Begin
    Head:=Nil;
  End;
```

3.4.2. Thêm một phần tử vào danh sách

Thêm phần tử *x* vào sau phần tử ở vị trí *p* trong danh sách có phần tử đầu là Head.



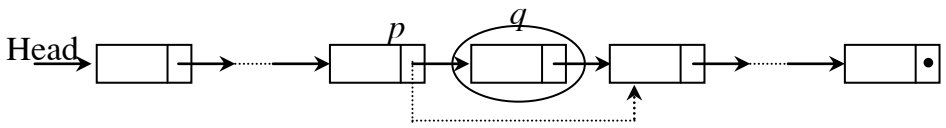
Hình 2.5. Thêm một phần tử vào danh sách liên kết

Thủ tục thêm một phần tử vào vị trí sau *p*. Trong thủ tục ta xét trường hợp khi danh sách rỗng thì phần tử thêm vào chính là phần tử duy nhất của danh sách. Chi tiết thủ tục như sau:

```
Procedure Insert(var Head:ListLink; x:ElementType;
                                                    p : ListLink);
var q : ListLink;
    Begin
        New(q);
        q^.Data := x;
        if Head = Nil then
            begin
                q^.Link:=nil;
                Head:=q;
            end
        else
            begin
                q^.Link := p^.Link;
                p^.Link := q;
            end;
    End;
```

3.4.3. Loại bỏ một phần tử khỏi danh sách

Giả sử cần loại bỏ phần tử ngay sau phần tử ở vị trí *p* trong danh sách có phần tử đầu là Head.



Hình 2.6. Xóa một phần tử trong danh sách liên kết

```
Procedure  Delete(var Head : ListLink; p : ListLink);
var q : ListLink;
    Begin
        q := p^.Link;
        if q<>nil then
            begin
                p^.Link := q^.Link;
                Dispose(q);
            end;
    End;
```

Dễ thấy độ phức tạp của thao tác thêm và xóa trong danh sách liên kết là O(1).

3.4.4. Tìm một phần tử trong danh sách liên kết

Cho danh sách liên kết có phần tử đầu Head. Cần tìm một phần tử có khóa Key bằng một giá trị *x* cho trước.

Với cách tổ chức dữ liệu của danh sách liên kết, việc truy xuất các phần tử là tuần tự nên thao tác tìm kiếm phải dùng thuật toán tìm tuần tự.

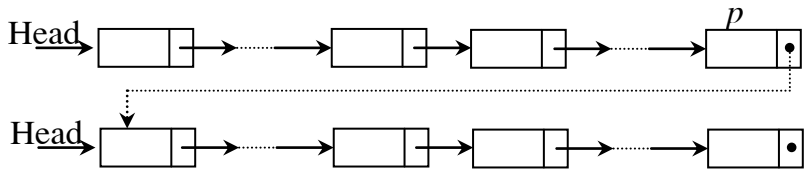
Thủ tục tìm khóa *x* trong danh sách Head, kết quả trả về qua giá trị found và vị trí của phần tử tìm được p.

```
Procedure Search(Head:ListLink; x:KeyType;
                                        var found:Boolean; var p:ListLink);
  Begin
     p:=Head; found:=false;
     While (p^.link<>nil) and not found do
        if p^.Data.Key = x then
           found:=true
        else
           p:=p^.link;
  End;
```

3.4.5. Nối 2 danh sách

Cho 2 danh sách có phần tử đầu tương ứng là Head1 và Head2. Ghép danh sách Head2 vào sau danh sách Head1.

Để ghép được danh sách 2 sau danh sách 1 phải tìm được vị trí của phần tử cuối danh sách 1 (nếu có) và liên kết với phần tử đầu của danh sách thứ 2. Trong trường hợp danh sách 1 rỗng thì kết quả ghép là danh sách 2.



Hình 2.7. Ghép 2 danh sách liên kết

```
Procedure Concat(var Head1:ListLink; Head2:ListLink);
var p :  ListLink;
  Begin
    If Head1 = nil Then Head1:=Head2
    Else
      Begin
          p:=Head1;
          While    p^.Link<>nil Do p:=p^.Link;
          p^.Link := Head2;
      End;
  End;
```

3.5. So sánh cấu trúc dữ liệu danh sách liên kết đơn và mảng

Khi biểu diễn danh sách bằng mảng phải ước lượng số phần tử tối đa của danh sách. Điều này gây ra lãng phí bộ nhớ trong trường hợp danh sách có ít phần tử nhưng sẽ không thêm phần tử vào danh sách được khi số phần tử nhiều. Trong khi đó biểu diễn danh sách bằng danh sách liên kết đơn sử dụng cơ chế cấp phát động nên bộ nhớ dùng cho danh sách được sử dụng đúng với số phần tử của danh sách tại mọi thời điểm do đó ít gây lãng phí ô nhớ và danh sách chỉ đầy khi không còn không gian nhớ để cấp phát. Tuy nhiên danh sách liên kết phải dùng một phần bộ nhớ để liên kết các phần tử trong danh sách.

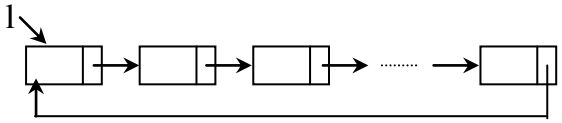
Trong danh sách tổ chức bằng mảng, thao tác truy xuất các phần tử là truy xuất trực tiếp nhưng các thao tác thêm một phần tử, xóa một phần tử có thời gian tỷ lệ với số phần tử của danh sách. Đối với danh sách liên kết đơn các thao tác thêm vào và xóa một phần tử được thực hiện với thời gian hằng trong khi thao tác với một phần tử là tuần tự. Tùy thuộc vào ứng dụng cụ thể của danh sách với các phép toán thường dùng của nó mà lựa chọn cách tổ chức danh sách bằng mảng hay danh sách liên kết.

3.6. Một số dạng danh sách liên kết khác

Một trong những hạn chế của danh sách liên kết đơn là phải quản lý được phần tử đầu tiên của danh sách và những thao tác với một phần tử phải biết phần tử ngay trước nó. Những hạn chế này có thể được khắc phục phần nào bởi việc thay đổi kiểu liên kết. Trong phần này giới thiệu 2 loại danh sách liên kết khác là liên kết vòng và liên kết kép.

3.6.1. Danh sách liên kết vòng

Danh sách liên kết vòng là danh sách liên kết đơn với một thay đổi là phần tử cuối của danh sách liên kết với phần tử đầu tiên. Hình ảnh của danh sách liên kết vòng như hình sau.



Hình 2.8. Danh sách liên kết vòng

Với cách tổ chức như trên, trong trường hợp không quan tâm đến thứ tự trước-sau của các phần tử, để quản lý danh sách ta chỉ cần quản lý phần tử bất kỳ trong danh sách mà không nhất thiết phải quản lý phần tử đầu tiên vì từ vị trí bất kỳ ta đều có thể truy xuất đến những phần tử còn lại của danh sách bằng cách duyệt tuần tự.

Về tổ chức dữ liệu, danh sách liên kết vòng có tổ chức hoàn toàn giống danh sách liên kết đơn. Để quản lý danh sách liên kết vòng ta dùng một biến con trỏ quản lý một phần tử bất kỳ, phần tử này gọi là phần tử hiện tại của danh sách. Trên danh sách liên kết vòng thường thực hiện các thao tác sau:

- Thêm một phần tử vào ngay sau phần tử hiện tại của danh sách.
- Thêm một phần tử vào ngay trước phần tử hiện tại của danh sách.
- Xóa phần tử ngay sau phần tử hiện tại.
- Duyệt danh sách.

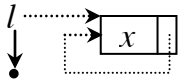
Cách tổ chức dữ liệu cho danh sách liên kết vòng giống như danh sách liên kết:

```
Type  ListLink = ^Cell;
     Cell = Record
              Data : ElementType;
              Link : ListLink;
            End;
var   l : ListLink;
```

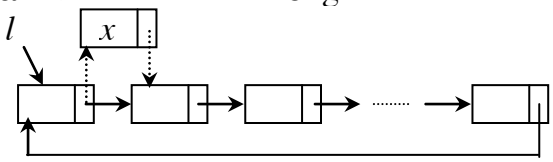
Các thủ tục trên danh sách liên kết vòng được mô tả như sau:

Thêm vào sau

a) Thêm vào danh sách rỗng



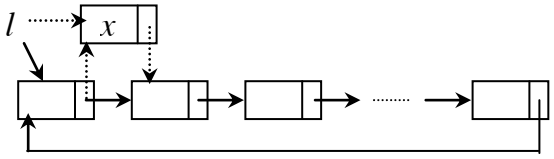
b) Thêm vào danh sách không rỗng



Hình 2.9 Thêm vào sau phần tử hiện tại trong danh sách liên kết vòng

```
Procedure InsertAfter(var l:ListLink; x:ElementType);
 var p:ListLink;
 Begin
   New(p);
   p^.Data := x;
   if l = nil then
     begin
       l:=p;
       l^.link:=l;
     end
   else
     begin
       p^.link := l^.link;
       l^.link := p;
     end;
 End;
```

Thêm vào trước



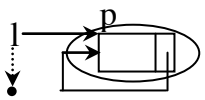
Hình 2.10 Thêm vào trước phần tử hiện tại trong danh sách liên kết vòng

Thủ tục thêm vào trước được thực hiện bằng cách thêm vào sau và chuyển phần tử hiện tại của danh sách là phần tử vừa thêm vào.

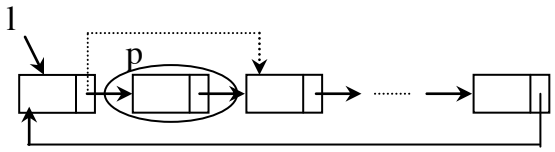
```
Procedure InsertBefore(var l: ListLink; x:ElementType);
    Begin
      InsertAfter(l, x);
      l := l^.link;
    End;
```

Xóa phần tử sau

a) Xúa danh sỏch cú 1



b) Xúa danh sỏch nhiều hơn 1 phần



Hình 2.11 xóa phần tử sau phần hiện tại trong danh sách liên kết vòng

```
Procedure DeleteAfter(var l : ListLink);
 var p : ListLink;
 Begin
   if l<>nil then
     begin
       p:=l^.link;
       if l^.link = l then l:=nil
       else l^.link := p^.link;
       dispose(p);
     end;
 End;
```

Duyệt danh sách

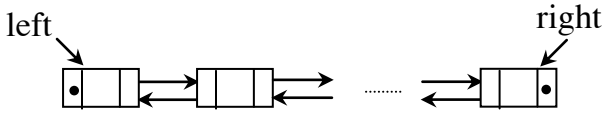
Thao tác duyệt danh sách liên kết vòng được thực hiện tương tự như thao tác duyệt danh sách liên kết đơn với chú ý vị trí của phần tử xuất phát để tránh duyệt lại danh sách.

```
Procedure Traverse(l : ListLink);
 var p : ListLink;
 Begin
  if l<>nil then
   begin
    p:=l;
    repeat
      write(p^.data:4);
      p:=p^.link;
    until p=l;
  end;
End;
```

Trong một số trường hợp ta có thể tổ chức danh sách liên kết vòng tốt hơn bằng cách dùng một phần tử đặc biệt để đánh dấu phần tử đầu tiên. Với cách tổ chức này danh sách liên kết vòng luôn khác rỗng vì luôn có phần tử đặc biệt.

3.6.2. Danh sách liên kết kép

Khi làm việc với những danh sách mà việc thao tác trên một phần tử của nó liên quan đến cả phần tử ngay trước và ngay sau nó thì việc tổ chức danh sách bằng liên kết đơn thì không đáp ứng được. Trong các trường hợp như vậy mỗi phần tử của danh sách thường được dùng hai liên kết đến phần tử ngay trước và ngay sau nó, danh sách như vậy gọi là danh sách liên kết kép.



Hình 2.12 Danh sách liên kết kép

Vì danh sách liên kết kép có thể duyệt theo cả hai chiều nên cần quản lý cả hai phần tử đầu và cuối danh sách. Khai báo dữ liệu cho danh sách liên kết kép như sau:

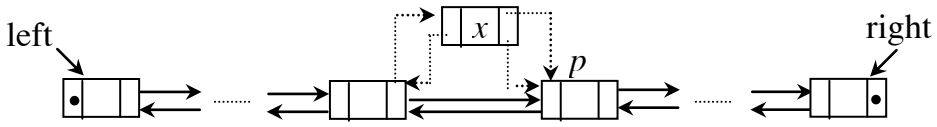
```
Type
    DLink = ^Cell2L;
    Cell2L = Record
              Data : ElementType;
              LLink, RLink : DLink;
             End;

    List2Link = Record
                  Left, Right : DLink;
                End;

Var  l : List2Link;
```

Các thao tác cơ bản trên danh sách liên kết kép:

Khởi tạo

```
Procedure Init(var l : List2Link);
   Begin
l.Left := nil;
l.Right:= nil;
   End;
```

Thêm một phần tử vào trước phần tử ở vị trí p

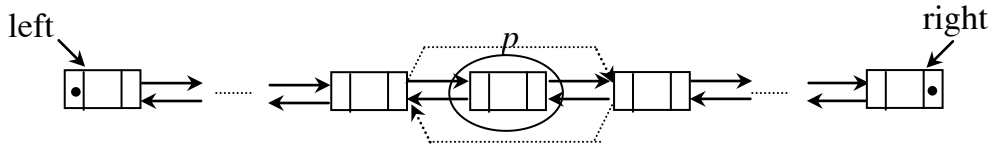


Hình 2.13 Thêm một phần tử vào vị trí p trong danh sách liên kết kép

```
Procedure InsertBefore(var l:List2Link;p:DLink;
                                                    x:ElementType);
   var  q,p1 : DLink;
 Begin
    New(q);
    q^.Data:= x;
             {danh sách rỗng}
    if (l.Left=nil) and (l.Right=nil) then
      begin
        q^.LLink:=nil;
        q^.RLink:=nil;
        l.Left:=q;
        l.Right:=q;
      end
     else
      Begin
        if l.Left = p then {thêm vào đầu}
          begin
            q^.LLink:=nil;
            q^.RLink:=p;
            p^.LLink:=q;
            l.Left:=q;
          end
          else
          begin
            p1:=p^.LLink;
            q^.LLink:=p1;
            p1^.Rlink:=q;
            q^.RLink:=p;
            p^.LLink:=q;
          end;
      end;
```

```
End;
```

Xóa phần tử tại vị trí p



Hình 2.14 Thêm một phần tử tại vị trí p trong danh sách liên kết kép

```
Procedure Delete(var l : List2Link; p : DLink);
Begin
   if l.Left=p then {xóa phần tử đầu}
     l.Left := p^.RLink
   else
     if l.Right=p then {xóa phần tử cuối}
       l.Right:=p^.LLink
     else
       begin
         p^.LLink^.RLink:=p^.RLink;
         p^.RLink^.LLink:=p^.LLink;
       end;
    dispose(p);
End;
```

Duyệt danh sách liên kết kép

Có thể duyệt từ trái sang phải hoặc ngược lại. Thủ tục duyệt từ trái sang phải được thực hiện như sau:

```
   Procedure Traverse(l : List2Link);
var p : DLink;
Begin
  p:=l.Left;
  while p<>nil do
    begin
                    Visit(p);
      p:=p^.RLink;
    end;
End;
```

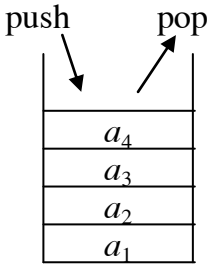
4. NGĂN XẾP (STACK)

Trong một số trường hợp chúng ta sử dụng một mô hình dữ liệu nhưng chỉ dùng một số trong các thao tác trên mô hình đó. Khi đó mô hình dữ liệu cùng với một số phép toán cụ thể trên mô hình đó gọi là một kiểu dữ liệu trừu tượng (abstract data type). Trong phần sau ta xét hai kiểu dữ liệu trừu tượng có ứng dụng nhiều trong các thuật toán tin học là *ngăn xếp* và *hàng đợi*.

4.1. Khái niệm

Ngăn xếp là một dạng danh sách đặc biệt chỉ được thực hiện hai thao tác: thêm một phần tử vào cuối danh sách (push) và lấy phần tử cuối ra khỏi danh sách (pop).

Hình ảnh một ngăn xếp



Hình 2.15 Hình ảnh ngăn xếp

Như vậy trong một ngăn xếp phần tử đưa vào sau sẽ được lấy ra trước nên còn gọi là danh sách kiểu LIFO (Last in First out). Vị trí của phần tử cuối cùng của ngăn xếp còn gọi là đỉnh (top) của ngăn xếp.

Ngăn xếp là một kiểu dữ liệu trừu tượng có nhiều ứng dụng trong tin học.

Trong các ngôn ngữ lập trình bậc cao bao giờ cũng dành riêng một vùng nhớ gọi là Stack dùng để lưu lại các giá trị của biến, hằng,... mỗi khi có lời gọi thủ tục, các giá trị này được lấy lại mỗi khi có một lời gọi thực hiện xong. Việc lưu các giá trị như trên phải theo nguyên tắc hoạt động của ngăn xếp vì lời gọi thủ tục cuối cùng sẽ kết thúc trước. Do đó ngăn xếp là một cách tổ chức dữ liệu được dùng nhiều trong các chương trình chuyển từ đệ quy sang lặp.

Trong các chương trình dịch (compiler) thường phải biến đổi các biểu thức trung tố thành các biểu thức tương đương ở dạng hậu tố. Chẳng hạn biểu thức (3 + 4) * 2 được chuyển thành 3 4 + 2 *. Việc chuyển các biểu thức từ trung tố thành hậu tố và tính giá trị các biểu thức hậu tố phải dùng cấu trúc dữ liệu kiểu ngăn xếp để lưu các kết quả trung gian. Chi tiết các thuật toán này sẽ được trình bày trong phần sau.

Như đã đề cập ở khái niệm của ngăn xếp, các thao tác trên ngăn xếp gồm hai thao tác cơ bản là :

push(x,S) : đưa phần tử x vào ngăn xếp S.

pop(x) : lấy phần tử ở đỉnh ngăn xếp S ra và lưu vào biến x.

Ngoài ra còn các thao tác bổ sung:

Init(S) : khởi tạo một ngăn xếp S rỗng.

Empty(S) : cho biết ngăn xếp S có rỗng không.

Full(S) : cho biết ngăn xếp S có đầy không.

Clear(S) : làm rỗng ngăn xếp S.

Tương tự như mô hình danh sách, trên ngăn xếp cũng có thể tổ chức bằng mảng và danh sách liên kết. Trong phần sau trình bày hai cách tổ chức dữ liệu cho ngăn xếp.

4.2. Tổ chức ngăn xếp bằng mảng

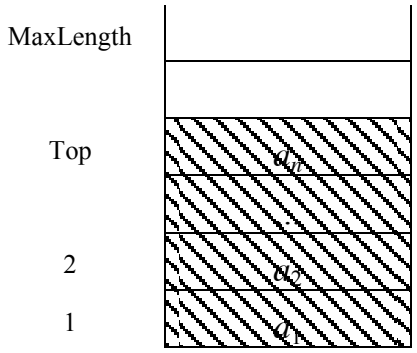
4.2.1. Tổ chức dữ liệu

Vì thao tác thêm vào và lấy ra chỉ thực hiện ở đỉnh của ngăn xếp nên dùng một biến để quản lý vị trí đỉnh của ngăn xếp. Một ngăn xếp tổ chức bằng mảng bao gồm hai thành phần:

+ Một mảng để lưu các phần tử của mảng.

+ Vị trí đỉnh của ngăn xếp.

Hình ảnh ngăn xếp $S=(a_1,a_2,...,a_n)$ tổ chức bằng mảng như sau:



Hình 2.16 Hình ảnh ngăn xếp tổ chức bằng mảng

Khai báo:

```
Const
   MaxLength = ... ; {Số phần tử tối đa của ngăn xếp}

Type

   ElementType = ... ; {Định nghĩa kiểu phần tử cho ngăn xếp}

   StackArr =Record
                  Element : Array[1..MaxLength] Of ElementType;
                  Top : 0..MaxLength;
             End;

Var   S : StackArr;
```

4.2.2. Các thao tác trên ngăn xếp

Khởi tạo: Đặt đỉnh ngăn xếp tại vị trí 0.

```
Procedure Init(var S : StackArr);
  Begin
        S.Top := 0;
  End;
```

Hàm Empty: Kiểm tra đỉnh ngăn xếp nếu top=0 thì ngăn xếp rỗng.

```
Function Empty(S : StackArr):Boolean;
  Begin
    Empty := S.Top = 0;
  End;
```

Hàm Full: Kiểm tra đỉnh ngăn xếp nếu top=MaxLength thì ngăn xếp đầy.

```
Function Full(S : StackArr):Boolean;
  Begin
    Full := S.Top = MaxLength;
  End;
```

Thêm vào: thêm phần tử x vào ngăn xếp S.

```
Procedure Push(x : ElementType; var S : StackArr);
  Begin
    if not Full(S) then
      with S do
        begin
          Inc(Top);
          element[top]:=x;
        end;
  End;
```

Lấy ra: lấy một phần tử từ ngăn xếp S ra biến x.

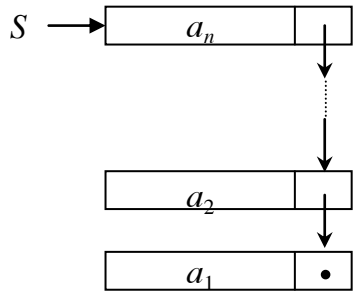
```
Procedure Pop(var x : ElementType; var S : StackArr);
  Begin
    if not Empty(S) then
      with S do
        begin
          x:=element[top];
          Dec(Top);
        end;
  End;
```

Độ phức tạp tính toán: Các thao tác thêm vào và lấy ra đều thực hiện tại đỉnh của ngăn xếp nên độ phức tạp các thao tác này là O(1).

4.3. Tổ chức ngăn xếp bằng danh sách liên kết

4.3.1. Tổ chức dữ liệu

Một ngăn xếp tổ chức bằng danh sách liên kết cũng giống như những danh sách khác, trong đó đỉnh của ngăn xếp chính là con trỏ của danh sách liên kết. Hình ảnh của ngăn xếp $S = (a_1, a_2, ..., a_n)$ tổ chức bằng danh sách liên kết như sau:



Hình 2.17 Hình ảnh ngăn xếp tổ chức bằng danh sách liên kết

4.3.2. Khai báo

```
Type
   StackLink = ^Cell;
   Cell = Record
                Data : ElementType;
                Link : StackLink;
           End;
Var
   S : StackLink;
```

4.3.3. Các thao tác

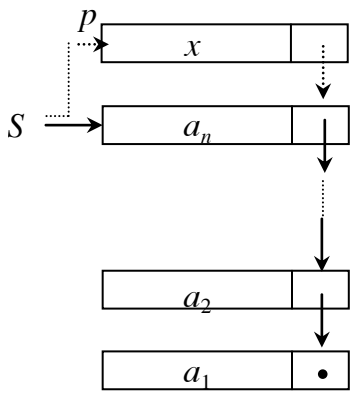
Khởi tạo:Đỉnh ngăn xếp trỏ đến nil

```
Procedure Init(var S : StackLink);
   Begin
      S := Nil;
   End;
```

Hàm Empty

```
Function Empty(S : StackLink):Boolean;
   Begin
      Empty := S = Nil;
   End;
```

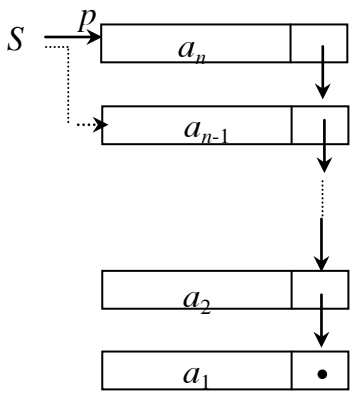
Thêm vào một phần tử: thêm phần tử x vào ngăn xếp S



Hình 2.18. Thêm một phần tử vào ngăn xếp tổ chức bằng danh sách liên kết

```
Procedure Push(x : ElementType; var S : StackLink);
   var p : StackLink;
   Begin
      New(p);
      p^.Data:=x;
      p^.Link:=S;
      S:=p;
   End;
```

Lấy ra một phần tử



Hình 2.19 Lấy một phần tử từ ngăn xếp tổ chức bằng danh sách liên kết

```
Procedure Pop(var x:ElementType; var S:StackLink);
   var p:StackLink;
   Begin
      if not Empty(S) then
      begin
         x := S^.Data;
         p := S;
         S:=p^.Link;
         Dispose(p);
      end;
   End;
```

Làm rỗng ngăn xếp: xóa và thu hồi các ô nhớ.

```
Procedure Clear(var S:StackLink);
   var p : StackLink;
   Begin
      while not Empty(S) do
      begin
         p := S;
         S:=p^.Link;
         Dispose(p);
      end;
   End;
```

4.4. Ứng dụng của ngăn xếp

4.4.1. Khử đệ qui

Giải thuật đệ qui là một trong những giải thuật thường dùng trong lập trình, tuy nhiên các thủ tục đệ qui khi cài đặt trên các ngôn ngữ lập trình phụ thuộc vào kích thước vùng nhớ dành cho Stack mà cho phép các thủ tục đệ qui được gọi bao nhiêu lần. Hơn nữa các thủ tục đệ qui dễ cài đặt nhưng khó hình dung chi tiết các bước thực hiện. Một số dạng thuật toán đệ qui có thể khử đệ qui bằng cách dùng một ngăn xếp để điều khiển quá trình gọi đệ qui. Chẳng hạn xét bài toán tháp Hà nội, ta có hai thuật toán đệ qui và không đệ qui như sau:

Thuật toán đệ qui:

Gọi số tầng cần phải chuyển là *n*, vị trí cọc xuất phát là *s*, vị trí cọc chuyển đến là *d*, cọc trung gian là *t*. Ta có thuật toán đệ qui chuyển tháp như sau:

Nếu n = 1 thì chuyển một tầng từ s đến d.

Ngược lại, nếu n > 1 thì

+ Chuyển n-1 tầng trên cùng từ s đến t.

+ Chuyển 1 tầng từ s đến d.

+ Chuyển n-1 tầng từ t sang d.

Thủ tục chuyển đĩa dạng đệ qui được thể hiện như sau:

```
Procedure Chuyen(n, s, d, t : Byte);
   Begin
      if n=1 then
         writeln(s,'-->',d);
      else
         begin
            Chuyen(n-1, s, t, d);
            Chuyen(1, s, d, t)
            Chuyen(n-1, t, d, s);
         end;
   End;
```

Thuật toán không đệ qui:

Dùng một Stack, mỗi phần tử chứa một bộ dữ liệu gồm (*n*,*s*,*d*,*t*). Việc gọi đệ qui trong thuật toán trên tương ứng với việc đưa một bộ dữ liệu vào ngăn xếp. Lúc đầu ngăn xếp được khởi tạo với bộ (*n*, 1, 2, 3). Quá trình lặp lại cho đến khi ngăn xếp rỗng.

Thủ tục khử đệ qui như sau:

```
Procedure Chuyen_Lap(n, s, d, t : Byte);
   var  st : Stack; sn, ss, sd, st : Byte;
   Begin
      Init(st);
      Push(n, s, d, t, st);
      Repeat
         pop(ns,ss,ds,ts,st);
         if ns = 1 then
            writeln(ss,'-->',ds)
         else
            begin
               push(ns-1,ts,ds,ss,st);
               push(1,ss,ds,ts,st);
               push(ns-1,ss,ts,ds,st);
            end;
      Until Empty(st);
   End;
```

4.4.2. Tính giá trị của các biểu thức

Tính giá trị của biểu thức là công việc thường làm của các ngôn ngữ lập trình. Thông thường các ngôn ngữ lập trình tính giá trị biểu thức bằng cách:

+ Chuyển biểu thức từ dạng trung tố (infix) sang dạng hậu tố (posfix).

+ Tính giá trị biểu thức hậu tố.

Biểu thức trung tố là cách con người thường sử dụng, trong biểu thức trung tố các phép toán hai ngôi được viết giữa hai toán hạng. Việc tính trực tiếp các biểu thức trung tố gặp khó khăn vì phải dùng các cặp dấu ngoặc đơn để qui định thứ tự thực hiện các biểu thức con. Để tính các biểu thức, người Balan đã đưa ra một ký pháp qui định cách viết các biểu thức (gọi là ký pháp Balan) trong đó các phép toán được đặt sau toán hạng. Việc dùng ký pháp Balan không cần dấu ngoặc nhưng vẫn thể hiện được thứ tự ưu tiên khi tính giá trị biểu thức nên dễ dàng xây dựng thuật toán tính.

Ví dụ: Biểu thức dạng trung tố 3+(5-2)*3/7-4 được viết dưới dạng biểu thức hậu tố : 3 5 2 - 3 * 7 / 4 + -

Thuật toán chuyển từ biểu thức trung tố sang hậu tố

Giả sử ta có một biểu thức E dạng trung tố trong đó ta có thể phân tích thành các thành phần của biểu thức là các toán hạng và phép toán.

Dùng một ngăn xếp S mỗi phần tử là phép toán hoặc dấu ngoặc mở. Kết quả đưa ra biểu thức hậu tố E_1.

Thuật toán:

1. Khởi tạo biểu thức E1 rỗng
2. Duyệt lần lượt các thành phần của biểu thức E, với mỗi thành phần x thực hiện
   - 2.1 Nếu x là toán hạng thì nối vào bên phải biểu thức E1
   - 2.2 Nếu x là dấu ngoặc mở thì đưa vào ngăn xếp
   - 2.3 Nếu x là phép toán thì
     - a. Đọc phần tử y ở đầu ngăn xếp
     - b. Nếu độ ưu tiên của y cao hơn x thì
       - Lấy y ra khỏi ngăn xếp
       - Nối y vào bên phải E1
       - Lặp lại bước a.
     - c. Nếu độ ưu tiên của x cao hơn y thì đưa x vào ngăn xếp
   - 2.4 Nếu x là dấu ngoặc đóng thì
     - a. Đọc phần tử y ở đầu ngăn xếp
     - b. Nếu y là phép toán thì
       - Lấy y ra khỏi ngăn xếp
       - Nối y vào bên phải biểu thức E1
       - Lặp lại bước a.
     - c. Nếu y là dấu ngoặc mở thì lấy ra khỏi ngăn xếp
3. Lặp lại bước 2 cho đến hết biểu thức E.
4. Lấy lần lượt các phần tử của ngăn xếp và nối vào bên phải biểu thức E1 cho đến khi ngăn xếp rỗng.

Ví dụ: Với biểu thức E = (2+7*3-8)*4-(3+2)*3-2, thuật toán chuyển thành biểu thức hậu tố thực hiện qua các bước thể hiện qua các kết quả ở bảng sau:

Thành phần của biểu thức E	Ngăn xếp	Biểu thức E_1
(	(	∅
2	(	2
+	(+	2
7	(+	2 7
*	(+ *	2 7
3	(+ *	2 7 3
-	(-	2 7 3 * +
8	(-	2 7 3 * + 8

)	∅	2 7 3 * + 8 -
*	*	2 7 3 * + 8 -
4	*	2 7 3 * + 8 - 4
-	-	2 7 3 * + 8 - 4 *
(	- (	2 7 3 * + 8 - 4 *
3	- (	2 7 3 * + 8 - 4 * 3
+	- (+	2 7 3 * + 8 - 4 * 3 +
2	- (+	2 7 3 * + 8 - 4 * 3 2
)	-	2 7 3 * + 8 - 4 * 3 2 +
*	- *	2 7 3 * + 8 - 4 * 3 2 +
3	- *	2 7 3 * + 8 - 4 * 3 2 + 3
-	-	2 7 3 * + 8 - 4 * 3 2 + 3 * -
2	-	2 7 3 * + 8 - 4 * 3 2 + 3 * - 2
	∅	2 7 3 * + 8 - 4 * 3 2 + 3 * - 2 -

Thuật toán tính giá trị biểu thức hậu tố

Cho biểu thức viết dưới dạng hậu tố E_1. Để tính giá trị của biểu thức ta dùng một ngăn xếp S lưu các toán hạng và các kết quả tính toán trung gian.

Thuật toán:

1. Duyệt lần lượt các phần tử của biểu thức E_1, với thành phần x thực hiện:

   Nếu x là toán hạng thì đưa vào ngăn xếp

   Nếu x là phép toán thì lấy hai phần tử đầu ngăn xếp thực hiện tính theo phép toán và đưa kết quả vào ngăn xếp

2. Lặp lại bước 1 cho đến khi hết biểu thức.

3. Giá trị duy nhất còn lại của ngăn xếp là kết quả của biểu thức E_1.

Ví dụ: Kết quả tính biểu thức hậu tố E_1=2 7 3*+8-4*3 2+3*-2- qua các bước được biểu diễn bởi bảng sau:

Thành phần biểu thức E_1	**Ngăn xếp**
2	2
7	2 7
3	2 7 3
*	2 21
+	23
8	23 8
-	15
4	15 4
*	60
3	60 3
2	60 3 2
+	60 5
3	60 5 3
*	60 15
-	45

2 -	45 2 **53**

5. HÀNG ĐỢI (QUEUE)

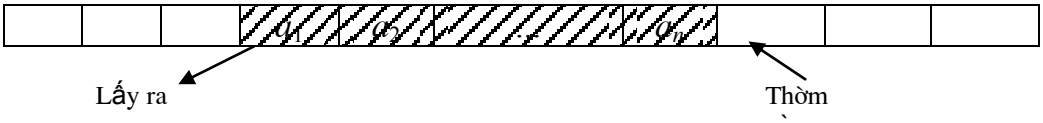
5.1. Khái niệm

Hàng đợi là một kiểu dữ liệu trừu tượng xây dựng trên mô hình danh sách với hai thao tác cơ bản:

+ Thêm một phần tử vào cuối hàng đợi.

+ Lấy phần tử đầu ra khỏi hàng đợi.

Hàng đợi được thực hiện theo nguyên tắc: các phần tử đưa vào trước lấy ra trước nên còn gọi là danh sách FIFO (First In First Out). Hàng đợi $Q = (a_1, a_2, ..., a_n)$ được thể hiện bằng hình dưới.



Hình 2.20. Hình ảnh một hàng đợi

Trong thực tế ta thường gặp những công việc thực hiện theo nguyên tắc của hàng đợi, chẳng hạn việc đăng ký mua vé tàu, việc chuyển các toa tàu trên một đường sắt,...

Trong máy tính mô hình hàng đợi được dùng khá phổ biến. Những hàng đợi được tạo ra khi có nhiều hơn một quá trình đòi hỏi được xử lý, chẳng hạn như máy in, ổ đĩa hay bộ xử lý trung tâm. Khi các quá trình đòi hỏi một tài nguyên, chúng được đặt trong một hàng đợi để chờ phục vụ. Ví dụ, nhiều máy tính có thể được phân phối để sử dụng một máy in, và một hàng đợi spool (Simultanous Peripherial Output On Line - đưa ra đồng thời với quá trình tính toán) được dùng để lập kế hoạch cho các yêu cầu đầu ra theo kiểu "đến trước được phục vụ trước". Nếu có một yêu cầu cho máy in và máy in rỗi, nó sẽ được cấp phát ngay lập tức công việc này. Trong khi in, một số công việc khác có thể cần đến máy in, chúng được đặt trong spool để chờ đến lượt. Khi công việc hiện thời của máy in kết thúc, máy in được giải phóng khỏi công việc ấy và được cấp phát cho công việc đầu tiên trong hàng spool.

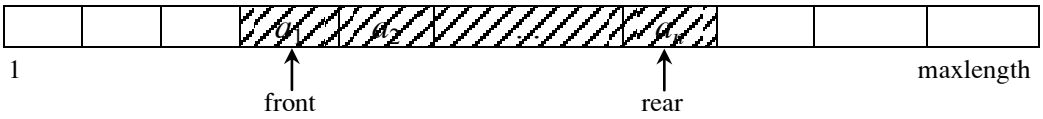
Một ứng dụng quan trọng khác của hàng đợi trong máy tính là tổ chức vùng đệm cho các đầu vào/ra (input/output buffering). Việc truyền thông tin từ một thiết bị vào hay đến một thiết bị ra là một thao tác tương đối chậm, do đó nếu phải tạm dừng chương trình trong khi truyền dữ liệu thì ảnh hưởng rất lớn đến tốc độ thực hiện chương trình. Một cách giải quyết thường dùng trên máy tính là dùng các phần của bộ nhớ trong, gọi là vùng đệm (buffer) và việc truyền dữ liệu từ một chương trình sẽ được truyền qua vùng đệm mà không thao tác trực tiếp với các thiết bị vào/ra. Chẳng hạn, một chương trình đọc dữ liệu từ một tệp trên đĩa, dữ liệu sẽ được truyền từ đĩa sang vùng đệm đầu vào trong bộ nhớ chính, trong khi bộ

xử lý trung tâm đang thực hiện một nhiệm vụ khác nào đó. Khi chương trình đòi hỏi dữ liệu, những giá trị tiếp theo trong hàng đợi này được lấy ra. Trong khi dữ liệu này đang xử lý thì các dữ liệu khác có thể được truyền từ tệp sang vùng đệm. Rõ ràng vùng đệm phải được thực hiện theo kiểu vào trước ra trước.

5.2. Tổ chức hàng đợi bằng mảng

5.2.1. Tổ chức

Hàng đợi tổ chức bằng mảng cũng giống như mô hình danh sách nhưng cần 2 thành phần để lưu vị trí phần tử sẽ được lấy ra ở đầu hàng đợi (front) và vị trí phần tử thêm vào cuối cùng của hàng đợi (rear).



Hình 2.21 Hình ảnh một hàng đợi tổ chức bằng mảng

5.2.2. Khai báo

```
Const
   MaxLength = ... ; {Số phần tử tối đa của hàng đợi}

Type
   ElementType = ... ; {Định nghĩa kiểu phần tử cho hàng đơi}

   QueueArr =    Record
                    Element: Array[1..MaxLength] Of ElementType;
                    Front, Rear : 0..MaxLength;
                 End;

Var   Q : QueueArr;
```

5.2.3. Cài đặt các thao tác trên hàng đợi

Khởi tạo hàng đợi

```
Procedure InitQueue(var Q : QueueArr);
   Begin
      Q.front:=1;
      Q.rear:=0;
   End;
```

Hàm Empty

```
Function Empty(Q : QueueArr):Boolean;
   Begin
      Empty:=Q.rear = 0;
   End;
```

Hàm Full

```
Function Full(Q : QueueArr):Boolean;
    Begin
        Full:= Q.rear = MaxLength;
    End;
```

Thêm vào một phần tử

Thêm phần tử x vào hàng đợi Q.

```
Procedure AddQueue(x : ElementType; var Q : QueueArr);
    Begin
        If not Full(Q) Then
            begin
                Q.rear := Q.rear + 1;
                Q.element[Q.rear]:=x;
            end;
    End;
```

Lấy một phần tử ra khỏi hàng đợi Q.

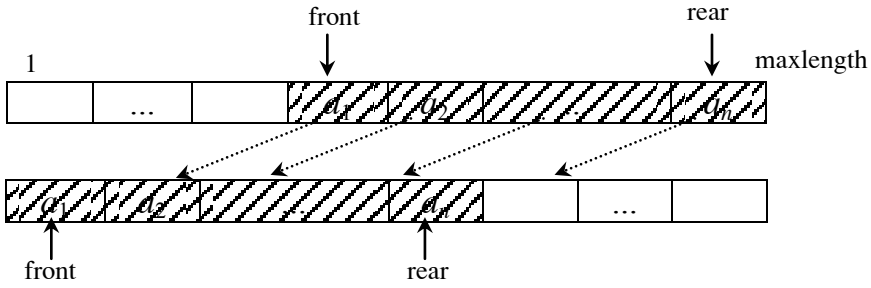
Lấy phần tử đầu ra khỏi hàng đợi *Q*, kết quả đưa vào biến *x*. Trong trường hợp phần tử lấy ra là phần tử cuối cùng (front = rear) thì khởi tạo lại các giá trị của front và rear. Trong các trường hợp còn lại tăng vị trí của front.

```
Procedure RemoveQueue(var x:ElementType;var Q: QueueArr);
        Begin
            If not Empty(Q) Then
                begin
                    x:=Q.element[Q.front];
                    if Q.front = Q.rear then
                        begin Q.front:=1; Q.rear:=0; end
                    else
                        Q.front := Q.front + 1;
                end;
        End;
```

- ***Nhận xét:*** Với cách tổ chức hàng đợi bằng mảng và các thao tác cài đặt như trên sẽ gặp hạn chế khi thao tác lấy ra không làm cho hàng rỗng vì khi đó các thao tác thêm vào và lấy ra sẽ làm cho phần sử dụng của hàng sẽ chuyển về cuối mảng, và đến một lúc nào đó ta không thể thêm vào hàng đợi vì vị trí rear đã đạt giá trị tối đa của kích thước mảng. Khi đó ta nói hàng bị tràn. Trong nhiều trường hợp khi hàng bị tràn thì vẫn chưa đầy vì các phần tử ở đầu mảng vẫn không được sử dụng do các thao tác lấy ra khỏi hàng đợi. Để khắc phục tình trạng hàng bị tràn thường dùng 2 cách khắc phục là di chuyển tịnh tiến và di chuyển vòng.

5.2.4. Khắc phục tràn hàng đợi bằng cách di chuyển tịnh tiến

Việc di chuyển tịnh tiến hàng được thực hiện khi thêm vào hàng đang ở tình trạng bị tràn, khi đó các phần tử của hàng được di chuyển về phía trước front - 1 vị trí. Sau khi di chuyển thì front=1 (xem hình).



Hình 2.22 Di chuyển tịnh tiến trên hàng đợi

Trong khi khắc phục hàng tràn bằng di chuyển tịnh tiến ta luôn có front ≤ rear và hàng đầy khi rear-front+1=maxlength.

Hàm Full

```
Function Full(Q : QueueArr):Boolean;
   Begin
      Full:= Q.rear - Q.front + 1 = MaxLength;
   End;
```

Thêm vào một phần tử

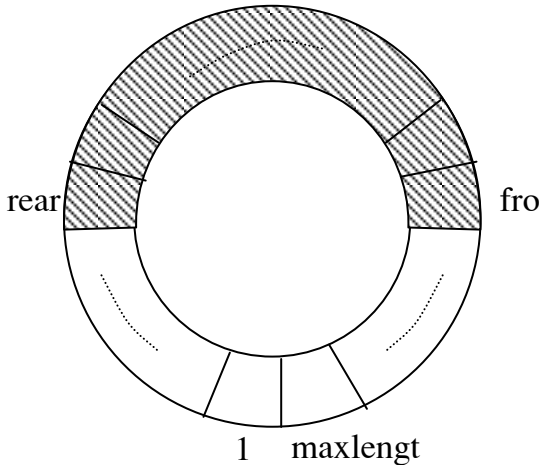
Thêm phần tử x vào hàng đợi Q có khắc phục tràn bằng dời tịnh tiến.

```
Procedure AddQueue(x : ElementType; var Q : QueueArr);
var i : Integer;
   Begin
      If not Full(Q) Then
         begin
            If Q.rear = MaxLength then {dời tịnh tiến}
               begin {dời hàng lên Q.front-1 vị trí}
                  For i:=Q.front To Q.rear Do
                     Q.element[i-Q.front+1]:=Q.element[i];
                  Q.rear:= MaxLength-Q.front+1;
                  Q.Front:=1;
               end
            else
               Q.rear := Q.rear + 1;
            Q.element[Q.rear]:=x;
         end;
   End;
```

Khắc phục tràn hàng đợi bằng di chuyển tịnh tiến thao tác thêm có độ phức tạp trong trường hợp xấu nhất là O(*n*), với *n* là số phần tử của hàng đợi. Để giảm

độ phức tạp trong các thao tác thêm nhưng vẫn khắc phục được tình trạng tràn hàng có thể tổ chức hàng đợi bởi mảng vòng tròn.

5.2.5. Khắc phục tràn hàng đợi bằng di chuyển vòng



Hình 2.23 Khắc phục tràn hàng bằng di chuyển vòng

Trong cách tổ chức hàng bằng mảng vòng tròn không bắt buộc các vị trí front và rear phải thỏa front≤rear mà các vị trí này di chuyển theo kiểu vòng tròn, nghĩa là khi tăng đến maxlength thì các vị trí này chuyển về 1. Do đó có những trường hợp front≥rear.

Với mảng vòng tròn như trên, hàng tràn trong các trường hợp rear-front+1=0 hoặc rear-front+1=maxlength.

Hàm Full

```
Function Full(Q : QueueArr):Boolean;
   Begin
      Full:= (Q.rear - Q.front + 1 = MaxLength) or
                (Q.rear - Q.front + 1 = 0);
   End;
```

Thêm vào một phần tử

Thêm phần tử x vào hàng đợi Q có khắc phục tràn bằng di chuyển vòng.

```
Procedure AddQueue(x : ElementType; var Q : QueueArr);
var i : Integer;
   Begin
      If not Full(Q) Then
         begin
            If Q.rear = MaxLength then
               Q.rear:=1
            else
               Q.rear := Q.rear + 1;
            Q.element[Q.rear]:=x;
```

```
        end;
    End;
```

Lấy một phần tử ra khỏi hàng đợi Q.

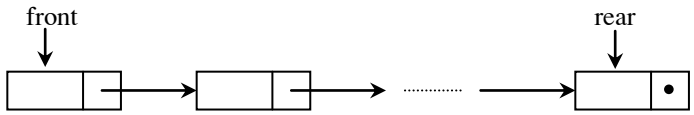
Lấy phần tử đầu ra khỏi hàng đợi Q, kết quả đưa vào biến x. Trong trường hợp phần tử lấy ra là phần tử cuối cùng (front=rear) thì khởi tạo lại các giá trị của front và rear. Trường hợp vị trí front là vị trí cuối cùng của mảng thì vị trí tiếp theo là 1. Trong các trường hợp còn lại tăng vị trí của front.

```
Procedure RemoveQueue(var x:ElementType; var Q:QueueArr);
    Begin
      If not Empty(Q) Then
        begin
          x:=Q.element[Q.front];
          if Q.front = Q.rear then
            begin Q.front:=1; Q.rear:=0; end
          else
            if Q.front = maxlength then
              Q.front:=1
            else
              Q.front := Q.front + 1;
        end;
    End;
```

5.3. Tổ chức hàng đợi bằng danh sách liên kết

5.3.1. Tổ chức

Cũng như ngăn xếp, ta có thể dùng danh sách liên kết để biểu diễn hàng đợi. Do thao tác thêm vào và lấy ra trên hàng đợi được lấy ra ở hai vị trí khác nhau nên ta dùng hai con trỏ front và rear để lưu hai vị trí này.



Hình 2.24. Biểu diễn hàng đợi bằng danh sách liên kết

Khai báo kiểu như sau:

```
Type
    ListLink = ^Cell;
    Cell =   Record
                Data : ElementType;
                Link : ListLink;
             End;
    QueueLink = Record
                    front, rear : ListLink;
                End;
```

```
Var  Q : QueueLink;
```

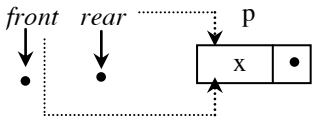
5.3.2. Cài đặt các thao tác trên hàng đợi

Khởi tạo

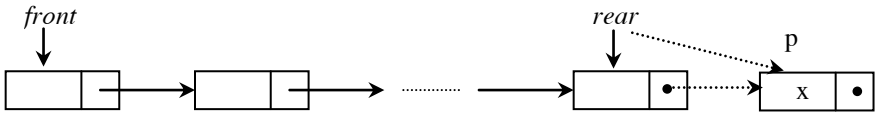
```
Procedure InitQueue(var Q : QueueLink);
   Begin
      Q.front:=Nil;
      Q.rear:=Nil;
   End;
```

Thêm vào Queue

a) Thêm vào hàng đợi rỗng



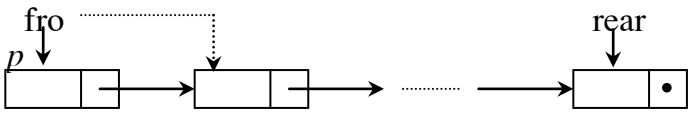
b) Thêm vào danh sách khộc rỗng



Hình 2.25 Thêm vào hàng đợi biểu diễn bằng danh sách liên kết

```
Procedure AddQueue(x : ElementType; var Q : QueueLink);
   var p : ListLink;
   Begin
      New(p);
      P^.Data := x;
      P^.Link := Nil;
      If Empty(Q) Then
         begin Q.front:=p; Q.rear:=p; end
      Else
         begin Q.rear^.Link:= p; Q.rear:=p; end;
  End;
```

Lấy một phần tử ra khỏi hàng đợi



Hình 2.26 Lấy một phần tử từ hàng đợi biểu diễn bằng danh sách liên kết

```
Procedure RemoveQueue(var Q:QueueLink;var x:ElementType);
      var p : ListLink;
         Begin
```

```
    If not Empty(Q) Then
       begin
          x:=Q.front^.Data;
          P:=Q.front;
          Q.front:=p^.Link;
          Dispose(p);
          If Q.front=Nil Then Q.rear:=Nil;
    end;
End;
```

6. BÀI TẬP

Bài 1. Cho một dãy số nguyên a_1, a_2, ..., a_n. Hãy tổ chức dữ liệu kiểu danh sách liên kết đơn để lưu dãy số và cài đặt các thủ tục thực hiện các công việc sau:

- Thủ tục thêm một số vào đầu dãy số.
- Thủ tục thêm một số vào cuối dãy số.
- Thủ tục thêm một số vào sau vị trí *p*.
- Thủ tục sắp xếp một dãy số theo thứ tự tăng.
- Thủ tục thêm một số *x* vào dãy đã sắp tăng sao cho vẫn giữ được thứ tự.
- Xóa một phần tử sau vị trí *p*.
- Thủ tục tìm-xoá một số *x* trong dãy số.

Dựa vào các thủ tục trên viết chương trình tổ chức dạng menu cho phép chọn thực hiện các công việc sau:

- Nhập một dãy số gồm n số.
- Sắp xếp dãy số
- Chèn một số
- Tìm và xóa một số
- Xóa toàn bộ dãy số
- In dãy số lên màn hình

Bài 2. Dùng ngăn xếp khử đệ quy của hàm tính C_n^m được viết như sau:

```
Function combo(n,m : integer):longint;
   Begin
      If (n=1) or (m=0) or (m=n) then
   Combo:=1
      Else combo:=combo(n-1,m)+combo(n-1.m-1);
   End;
```

Bài 3. Cho một xâu *S* chỉ gồm các dấu ngoặc tròn (,). Hãy nêu thuật toán và lập chương trình kiểm tra xem các dấu ngoặc trong xâu *S* có thể là các dấu ngoặc của một biểu thức hợp lệ không.

Ví dụ: xâu (()()) : hợp lệ; xâu ((()))(: không hợp lệ; (()(()())())) : không hợp lệ.

Bài 4. Bằng các thao tác cơ bản trên ngăn xếp và hàng đợi hãy trình bày các thuật toán:

- Đảo ngược thứ tự một hàng đợi.
- Tìm một phần tử có khóa của trường Key là x trong hàng đợi.

Cài đặt các thuật toán trên bằng hàng đợi các số nguyên với cách tổ chức hàng đợi bằng danh sách liên kết.

Bài 5. Trình bày thuật toán đảo ngược một danh sách liên kết bằng hai cách:

- Đổi giá trị của các phần tử.
- Đổi liên kết của các phần tử.

Cài đặt các thuật toán trên cho danh sách các số nguyên. Nếu dùng thủ tục đệ qui thì hãy thử dùng ngăn xếp để khử đệ qui.

Bài 6. (*Danh sách liên kết bội*) Để quản lý một danh sách các danh sách liên kết người ta tổ chức dữ liệu dạng danh sách bội. Mỗi phần tử trong danh sách bội quản lý một danh sách liên kết gồm các thành phần sau :

- Thành phần head trỏ đến một danh sách liên kết đơn
- Thành phần link liên kết tới danh sách tiếp theo (nếu có)

Hãy tổ chức dữ liệu và trình bày các thuật toán thực hiện:

- Tạo một danh sách bội trống : danh sách chưa có danh sách nào.
- Thêm một danh sách mới (trống) vào đầu danh sách bội.
- Thêm một phần tử x vào đầu danh sách trỏ bởi con trỏ plist trong danh sách bội.
- Xóa phần tử sau vị trí p trong danh sách plist trong danh sách liên kết bội.
- Xóa danh sách sau danh sách trỏ bởi con trỏ plist trong danh sách bội.
- Tìm một khóa x trong danh sách bội. Kết quả trả về con trỏ trỏ đến ô chứa phần tử có khóa x tìm được đầu tiên.
- Xóa tất cả các phần tử có khóa x trong danh sách bội.

Cài đặt các thao tác trên cho danh sách bội chứa các số nguyên.

Bài 7. Trong các danh sách tổ chức bằng liên kết đơn thường dùng thao tác chèn một phần tử vào cuối danh sách, để tiện thao tác thêm ta phải lưu vị trí phần tử cuối cùng. Hãy trình bày cách tổ chức dữ liệu và thuật toán thực hiện hai thao tác: thêm một phần tử vào cuối danh sách và xóa một phần tử sau vị trí p trong danh sách. Cài đặt các thuật toán trên cho danh sách các số nguyên.

Bài 8. Dùng ngăn xếp các số nguyên với các phép toán push, pop hãy lập chương trình đổi một số thập phân thành số nhị phân.

Bài 9. Cài đặt chương trình tính biểu thức số học bằng cách chuyển biểu thức trung tố thành biểu thức hậu tố dùng ký pháp Balan. Giới hạn chỉ xét các biểu thức số học gồm các phép toán hai ngôi : +, -, *, / trên các số có 1 chữ số.

Bài 10. Dùng danh sách liên kết đơn biểu diễn đa thức. Nêu thuật toán và cài đặt các thủ tục thực hiện :

- Nhập một đa thức
- Xuất một đa thức
- Tính giá trị của đa thức tại x_0
- Cộng hai đa thức
- Nhân hai đa thức

Bài 11. Cho một ngăn xếp với các thao tác cơ bản : Initialize, Empty, Push, Pop hãy trình bày thuật toán và viết các thủ tục thực hiện :

a. Lấy ra phần tử cuối cùng của ngăn xếp.

b. Lấy ra phần tử thứ n của ngăn xếp, với n là một số nguyên dương.

Cài đặt cụ thể các thuật toán trên bằng ngăn xếp các số nguyên trên một ngôn ngữ lập trình cụ thể với cách tổ chức ngăn xếp bằng mảng.

Bài 12. Cho một hàng đợi với các thao tác cơ bản: Initialize, Empty, Push, Pop hãy trình bày thuật toán và viết các thủ tục thực hiện:

a. Lấy ra phần tử cuối cùng của hàng đợi.

b. Lấy ra phần tử thứ k của hàng đợi, với k là một số nguyên dương.

Cài đặt cụ thể các thuật toán trên bằng hàng đợi các số nguyên trên một ngôn ngữ lập trình cụ thể với cách tổ chức hàng đợi bằng danh sách liên kết.

Bài 13. Dùng danh sách liên kết đơn trong đó các phần tử là các số nguyên được sắp theo thứ tự để cài đặt cho tập hợp các số nguyên. Hãy tổ chức dữ liệu, cài đặt các thủ tục:

- Thêm một phần tử vào tập hợp.
- Loại một phần tử khỏi tập hợp.
- Kiểm tra một phần tử có trong tập hợp hay không.
- Giao hai tập hợp
- Hợp hai tập hợp
- Hiệu hai tập hợp

Từ các thủ tục trên hãy viết chương trình thực hiện các công việc:

- Nhập vào hai tập hợp

- Tìm giao, hợp, hiệu hai tập hợp trên và in kết quả lên màn hình
- Tạo tập hợp chứa các số tự nhiên liên tiếp từ 2 đến 1000
- Cài đặt thuật toán sàng Eratosthenes để tìm các số nguyên tố nhỏ hơn 10000.

Bài 14. Trong một số ngôn ngữ lập trình bậc cao không có kiểu con trỏ. Hãy đề xuất một cách tổ chức dữ liệu biểu diễn danh sách liên kết cho những ngôn ngữ lập trình này. Cài đặt các thao tác cơ bản trên cách tổ chức dữ liệu trên. So sách với cách tổ chức bằng kiểu con trỏ.

Hướng dẫn: dùng mảng để lưu các phần tử, phần liên kết là chỉ số của phần tử tiếp theo.

Bài 15. Cho hai danh sách số nguyên tổ chức bằng mảng $L_1 = (a_1, a_2, ..., a_n)$ và $L_2 = (b_1, b_2, ..., b_m)$. Hãy viết các hàm:

a) LaDSCon(L_1, L_2) : trả về kết quả True nếu L_2 là danh sách con của L_1 và false trong trường hợp ngược lại.

b) LaDayCon(L_1, L_2) : trả về kết quả True nếu L_2 là dãy con của L_1 và false trong trường hợp ngược lại.

Hãy cài đặt các hàm trên trong trường hợp danh sách được tổ chức bằng danh sách liên kết. Từ đó so sánh độ phức tạp của hai thao tác trên đối với hai cách biểu diễn danh sách.

Tổng quát thành thuật toán cho danh sách tổng quát cho hai thao tác trên.

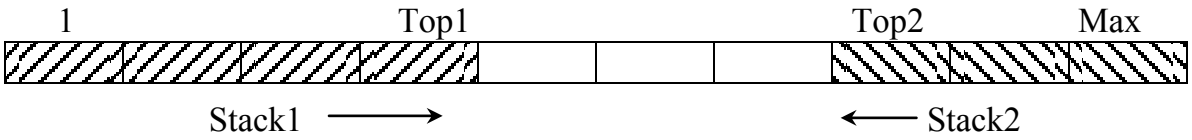
Bài 16. (Hàng đợi hai đầu - double ended queue : deque) Các phép toán thêm và lấy ra trên hàng đợi được hạn chế sao cho lấy ra ở một đầu và thêm vào ở đầu còn lại. Tuy nhiên trong thực tế trong một số trường hợp ta dùng những hàng đợi mà thao tác thêm vào và lấy ra được thực hiện ở cả hai đầu. Những hàng đợi như thế được gọi là hàng đợi hai đầu. Một trong những ứng dụng của hàng đợi hai đầu là hàng đợi cho bộ đệm bàn phím vì khi đó ta có thể dùng phím xóa để xóa một số phần tử cuối của hàng đợi.

Hãy tổ chức dữ liệu kiểu danh sách liên kết và cài đặt các thao tác trên hàng đợi hai đầu.

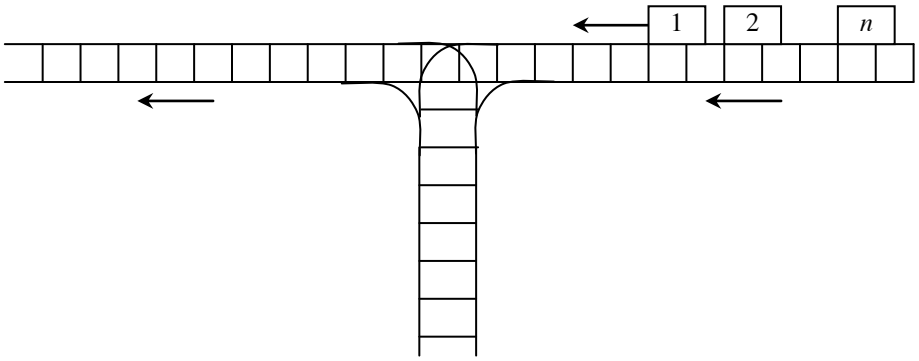
Bài 17. (Hàng đợi hai đầu bằng mảng vòng). Tương tự cách cài đặt hàng đợi bằng mảng vòng, hãy xây dựng cách cài đặt deque. Viết các khai báo và các chương trình con thích hợp cho các thao tác cơ bản (tạo một deque rỗng, kiểm tra deque có rỗng hay không, thêm vào đầu, thêm vào cuối, lấy ra phần tử đầu, lấy ra phần tử cuối).

Bài 18. (Hai ngăn xếp) Trong trường hợp cần dùng hai ngăn xếp cùng kiểu, thay vì dùng hai mảng ta có thể dùng một mảng duy nhất để lưu các phần tử của hai ngăn xếp. Hãy tổ chức dữ liệu, viết các khai báo thích hợp cho cách cài đặt này và viết các thủ tục, hàm thực hiện các thao tác trên hai ngăn xếp sao cho thuận lợi nhất.

Hướng dẫn: Tổ chức dữ liệu như hình sau:



Bài 19. Xét mạng lưới đường sắt dưới đây:

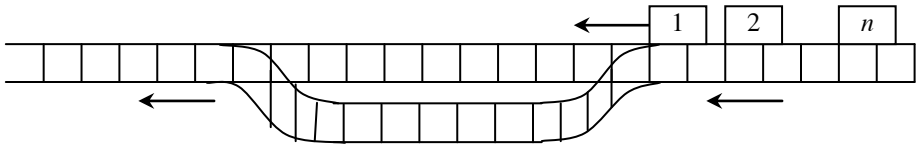


Các toa tàu ở đường ray bên phải (được đánnh số 1, 2, ..., *n*) cần phải hoán vị và chuyển sang đường ray bên trái. Như sơ đồ đã chỉ ra, một toa có thể chuyển thẳng sang đường ray bên trái, hay nó có thể rẽ qua đường ray bên cạnh để sau này đặt nó ở đường ray bên trái. Đường ray rẽ ngang bên cạnh đóng vai trò như một ngăn xếp, phép toán đẩy (push) chuyển một toa tàu từ đường ray bên phải sang bên cạnh, và phép toán lấy ra (pop) chuyển toa ở "đỉnh" của đường ray bên cạnh sang đường ray bên trái. Với giả thiết đường ray rẽ ngang đủ chỗ để chứa các toa tàu.

a) Với $n = 3$, tìm tất cả những hoán vị có thể có của các toa nhận được (ở đường ray bên trái) bằng một loạt các phép toán nói trên. Những hoán vị nào là không thể được?

b) Một cách tổng quát, những hoán vị nào của dãy 1, 2, .., *n* có thể nhận được khi ngăn xếp được dùng theo cách này?

Bài 20. Yêu cầu tương tự như bài trên nhưng mạng lưới đường sắt được tổ chức như hình sau.



Trong đó đường ray bên cạnh được tổ chức như hàng đợi.

Bài 21. (Hàng đợi ưu tiên - priority queue) Hàng đợi ưu tiên là một hàng đợi mà mỗi phần tử được gắn với một độ ưu tiên. Hàng đợi được tổ chức sao cho những phần tử có độ ưu tiên cao nhất trong hàng đợi bao giờ cũng được lấy ra trước. Với những phần tử mà độ ưu tiên bằng nhau thì thực hiện theo nguyên tắc vào trước ra trước. Hãy tổ chức dữ liệu và cài đặt các thao tác cho hàng đợi ưu tiên.

Bài 22. Nếu độ ưu tiên của các phần tử trong một hàng đợi ưu tiên là những số nguyên trong đoạn 1..*p*, ta có thể dùng *p* mảng khác nhau để cài đặt một hàng đợi ưu tiên, mỗi mảng cho một hàng chứa các phần tử cùng độ ưu tiên. Hãy viết các khai báo và các thủ tục thực hiện các phép toán cơ bản với cách xây dựng hàng đợi ưu tiên như trên.

Bài 23. Nếu độ ưu tiên của các phần tử trong hàng đợi ưu tiên có phân bố không đều, một cách khác để tổ chức dữ liệu cho hàng đợi ưu tiên là dùng một mảng duy nhất và dùng *p*+1 con trỏ, một phần tử ở đầu hàng đợi ưu tiên và một cho phần tử cuối mỗi hàng đợi cùng độ ưu tiên. Hãy tổ chức dữ liệu và viết các thủ tục cần thiết cho cách tổ chức này.

Chương 3

CÂY

Cây là một cấu trúc phân cấp trên một tập các đối tượng. Mô hình cây thường được dùng trong thực tế cũng như tin học chẳng hạn như cây gia phả, mục lục của một cuốn sách, cây thư mục, cây phân tích cú pháp,...

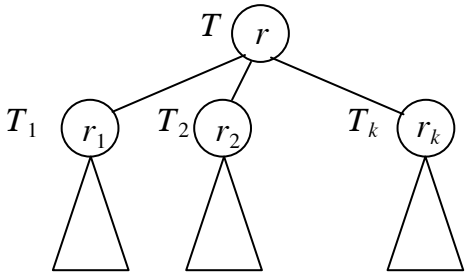
7. CÁC KHÁI NIỆM VỀ CÂY

7.1. Khái niệm cây

Cây là một tập hữu hạn các phần tử gọi là các nút (node) với một thứ tự phân cấp giữa các phần tử, còn gọi là quan hệ cha-con. Trong cây có một nút đặc biệt ở cấp cao nhất gọi là nút gốc (root), các nút còn lại, mỗi nút là con của một nút nào đó của cây.

Có nhiều cách định nghĩa cây, trong chương này định nghĩa cây bằng đệ quy như sau:

- Tập hợp gồm một phần tử là một cây và phần tử này chính là nút gốc của cây.
- Nếu r là một nút và T_1, T_2, ..., T_k là các cây với các nút gốc tương ứng là r_1, r_2, ..., r_k. Khi đó nếu có quan hệ cha-con của nút r đối với các nút r_1, r_2, ..., r_k thì tập hợp gồm nút r và tất cả các nút của những cây T_1, T_2, ..., T_k tạo thành một cây T có nút gốc là r và T_1, T_2, ..., T_k là các *cây con* của cây T, các nút r_1, r_2, ..., r_k gọi là các *nút con* của nút r.



Hình 3.1 Hình ảnh cây T với các cây con T_1, T_2, ..., T_k

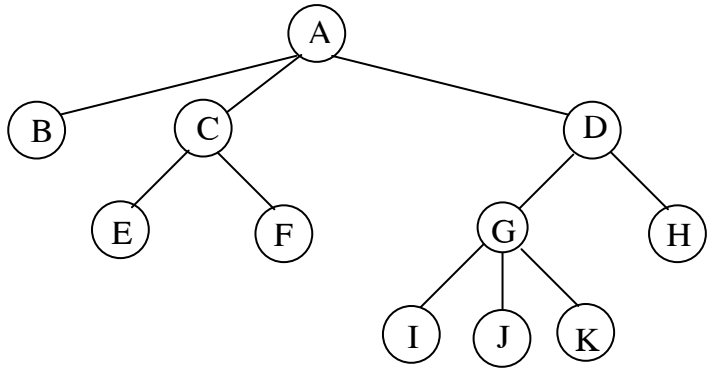
Trường hợp đặc biệt, một cây không có nút nào được gọi là cây rỗng (null tree).

Trong thực tế ta thường gặp những đối tượng có cách tổ chức theo mô hình cây như mục lục một cuốn sách, cây gia phả trong một dòng họ, phân cấp tổ chức trong một cơ quan, đoàn thể,...

Trong tin học, mô hình phân cấp của cây là một trong những mô hình có nhiều ứng dụng. Chẳng hạn cách tổ chức lưu trữ dữ liệu trên đĩa được tổ chức bằng cây thư mục, mô hình cây dùng để biểu diễn không gian lời giải của các bài

toán như trong các bài toán trò chơi. Cây còn dùng trong các bước phân tích cú pháp của các văn phạm như các biểu thức, các câu lệnh trong một chương trình. Cây có thể dùng để tổ chức dữ liệu ở bộ nhớ trong và bộ nhớ ngoài thuận tiện cho các bài toán tìm kiếm như từ điển, kiểm tra từ trong một văn bản,...

Để biểu diễn cây trên thực tế có nhiều cách, tuy nhiên cách thường dùng nhất là biểu diễn cây bằng đồ thị. Mỗi phần tử (nút) được biểu diễn bằng một đỉnh, quan hệ "cha-con" giữa hai nút được biểu diễn bằng một cạnh nối giữa chúng.



Hình 3.2 Hình ảnh một cây biểu diễn bằng đồ thị

7.2. Một số khái niệm khác

Bậc của nút: là số nút con của nút đó. Chẳng hạn với cây trên, bậc của nút A là 3, bậc của nút B là 0, bậc của nút C là 2.

Bậc của cây: là bậc của nút có bậc lớn nhất trong các nút của cây. Cây có bậc n thì còn gọi là cây n-phân. Trong thực tế ta thường dùng các cấu trúc cây nhị phân (cây bậc 2), cây tam phân. Ví dụ cây ở hình trên có cấp 3.

Nút lá (leaf node) : là nút có bậc bằng 0 (không có nút con). Trong cây ở hình trên có các nút lá là B, E, F, I, J, K, H.

Nút trung gian (interior node): là nút của cây không phải nút gốc và nút lá.

Mức (level) của nút: được định nghĩa đệ quy như sau:

+ Mức của nút gốc bằng 1.

+ Mức của nút khác nút gốc bằng mức của nút cha của nó cộng 1.

Chiều cao của cây: là mức lớn nhất trong các nút của cây.

Chiều dài đường đi (*path length*):

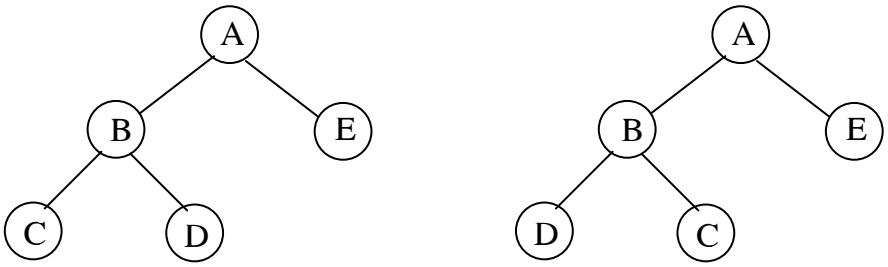
Chiều dài đường đi của một nút là số các nút cần đi qua từ nút gốc đến nút đó. Chiều dài đường đi của một nút là mức của nút đó.

Chiều dài đường đi của một cây: là tổng các độ dài đường đi của tất cả các đỉnh của nó. Chiều dài đường đi của cây còn gọi là đường đi trong của cây. Ví dụ độ dài đường đi của cây ở hình trên là 31.

Độ dài đường đi trung bình của cây: $P_I = \frac{1}{n}\sum_{i=1}^{h} n_i \times i$, trong đó n_i là số các nút ở mức thứ i, n là tổng số các nút của cây, h là chiều của cây.

Độ dài đường đi ngoài của cây: Giả sử mọi nút của cây đều có cùng bậc và là bậc của cây (nếu không ta mở rộng cây bằng cách thêm những nút đặc biệt vào vị trí của những cây con rỗng trong cây). Độ dài đường đi ngoài của cây là tổng độ dài đường đi của tất cả các nút đặc biệt. Nếu số nút đặc biệt ở mức i là m_i thì độ dài đường đi ngoài trung bình của cây là $P_E = \frac{1}{m}\sum_{i=1}^{h} m_i \times i$. Ví dụ độ dài đường đi ngoài của cây ở hình trên là 96.

Cây có thứ tự (ordered tree): là cây có xác định vị trí của các nút. Với cây có thứ tự thì hai cây sau là khác nhau:



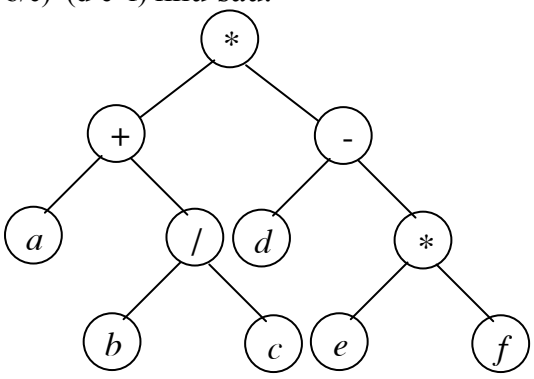
Hình 3.3 Cây có thứ tự

8. CÂY NHỊ PHÂN

8.1. Khái niệm

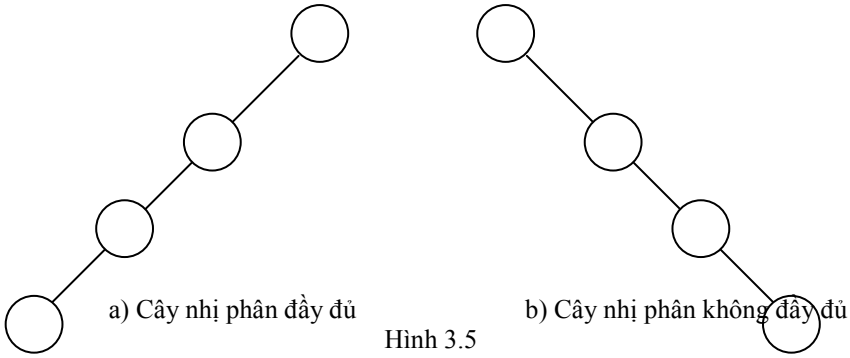
Cây nhị phân là một cây mà tại mỗi nút của cây có không quá hai cây con.

Cây nhị phân là một cấu trúc cây thường dùng trong tin học, chẳng hạn cây biểu diễn biểu thức (a + b/c)*(d-e*f) như sau:



Hình 3.4 Biểu diễn cây của biểu thức (a + b/c)*(d-e*f)

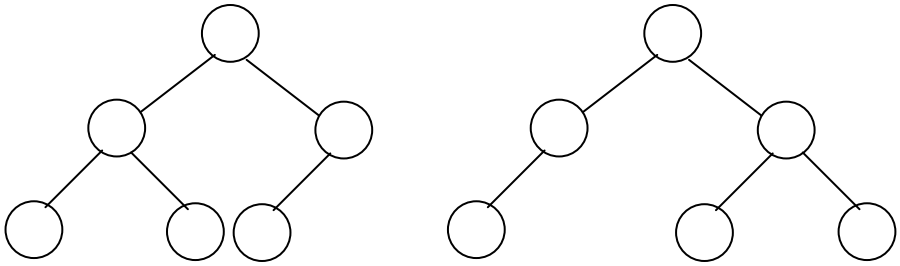
Cây nhị phân đầy đủ là cây nhị phân mà các nút lá trên cây nằm nhiều nhất ở hai mức liên tiếp nhau, và các lá ở mức dưới nằm dồn về phía bên trái.



a) Cây nhị phân đầy đủ b) Cây nhị phân không đầy đủ

Hình 3.5

Cây suy biến là cây nhị phân mà tại các nút của cây, các cây con trái hoặc cây con phải đều rỗng. Cây suy biến còn gọi là cây lệch phải (hoặc lệch trái).



a) Cây lệch trái b) Cây lệch phải

Hình 3.6. Cây nhị suy biến

Cây suy biến không thuận lợi cho các thao tác trên cây vì các thao tác tương tự như trên danh sách tuần tự.

8.2. Biểu diễn cây nhị phân

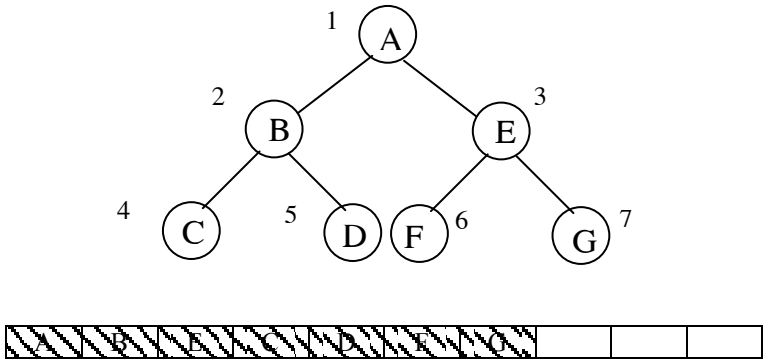
Tương tự như mô hình danh sách, với mô hình cây nhị phân ta thường dùng hai cách biểu diễn thông dụng là mảng và liên kết.

8.2.1. Biểu diễn cây nhị phân bằng mảng

Để biểu diễn cây nhị phân bằng mảng, các phần tử của cây (các nút) được lưu vào các ô nhớ của mảng. Tuy nhiên để lưu được cấu trúc phân cấp ta phải có những tổ chức khác. Ta có hai cách dùng mảng cho cây nhị phân đầy đủ và cây nhị phân bất kỳ như sau:

Cây nhị phân đầy đủ

Với cây nhị phân đầy đủ ta có thể biểu diễn cây bằng cách đánh số các nút theo thứ tự từ trên xuống dưới và từ trái sang phải. Các nút của cây được lưu trong các ô nhớ của mảng tương ứng với chỉ số đã được qui định.



Hình 3.7 Biểu diễn cây nhị phân đầy đủ bằng mảng

Tổ chức cây nhị phân đầy đủ bằng mảng như trên thuận lợi cho các thao tác tìm các nút con của một nút và nút cha của một nút. Cụ thể, với phần tử thứ *i* trong mảng thì nút con bên trái của nó ở vị trí 2**i* và nút con bên phải ở vị trí 2**i* +1, nút cha của nó ở vị trí *i* div 2 (với *i*>1). Tuy nhiên cách tổ chức này không tốt trong trường hợp cây không đầy đủ vì khi đó nhiều ô nhớ trong mảng không sử dụng.

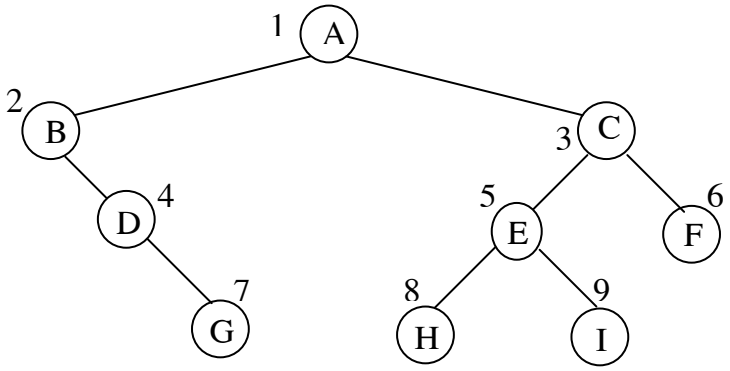
Cây nhị phân bất kỳ

Với cây nhị phân bất kỳ ta có thể lưu trong một mảng các bản ghi, mỗi bản ghi lưu một nút và thông tin về nút con của nó, gồm 3 thành phần: Data lưu dữ liệu của nút, Left, Right lưu vị trí của nút con bên trái, nút con bên phải của nó.

Khai báo:

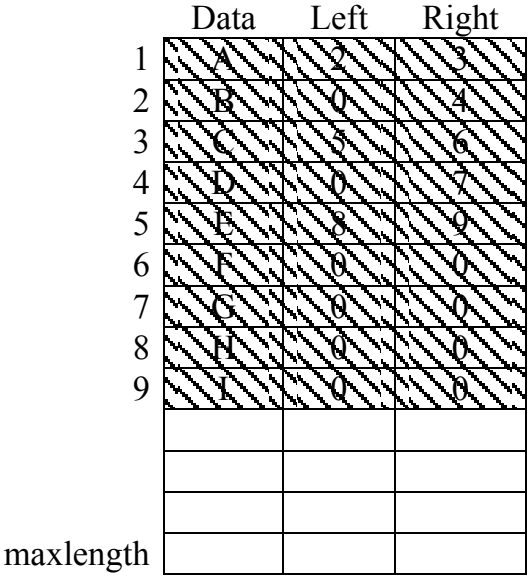
```
const maxlength = ...; {kích thước tối đa của mảng}

Type Node =  Record
                    Data : ElementType;
                    left, right : 0..maxlength;
                End;

  BtreeArr = Array[1..maxlength] of Node;
```

Ví dụ: Cho cây như hình vẽ



Hình 3.8 Biểu diễn cây nhị phân bất kỳ bằng mảng

Mảng biểu diễn cây trên có dạng



Với cách tổ chức cây nhị phân như trên đáp ứng được các yêu cầu về biểu diễn dữ liệu và phân cấp của cây nhị phân. Tuy nhiên cách biểu diễn này không thuận lợi cho các thao tác thêm, xóa các phần tử trong cây vì ta phải tìm các ô nhớ của mảng còn trống khi thêm và đánh dấu các ô nhớ trống khi xóa. Trong chương này chúng ta sử dụng cách biểu diễn cây nhị phân bằng liên kết và cấp phát động được trình bày trong phần sau.

8.2.2. Biểu diễn cây nhị phân bằng liên kết

Biểu diễn cây nhị phân bằng liên kết được dựa trên cơ chế cấp phát động, trong đó mỗi nút gồm ba thành phần:

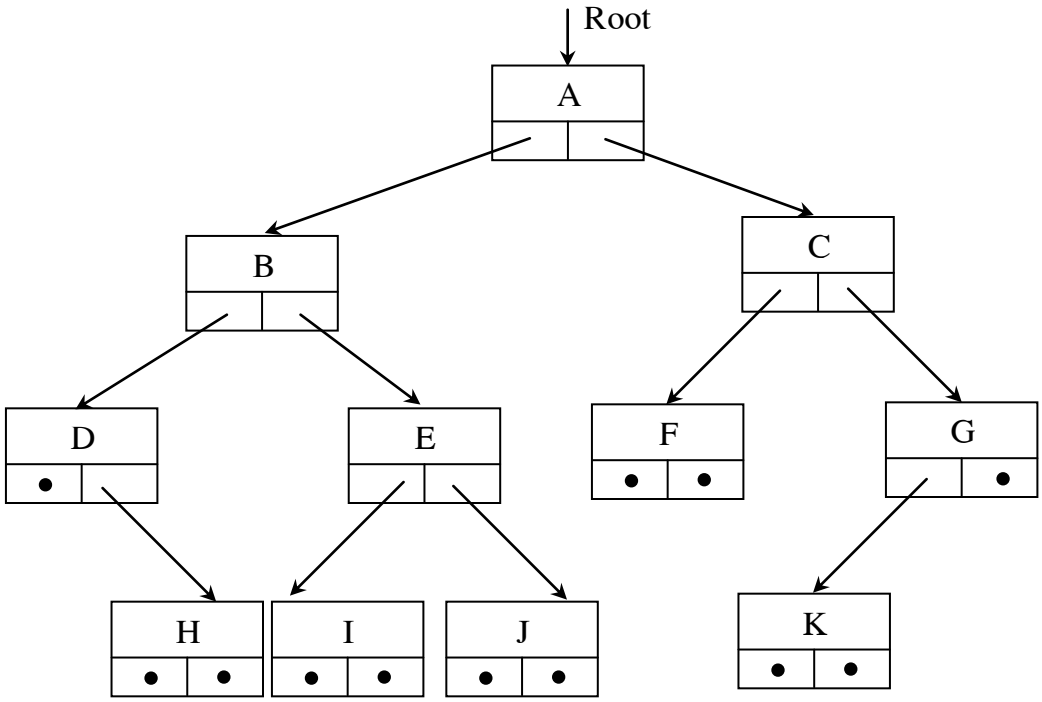
+ Thành phần Data dùng để lưu dữ liệu của phần tử của nút.

+ Thành phần Left là con trỏ quản lý nút gốc của cây con bên trái.

+ Thành phần Right là con trỏ quản lý nút gốc của cây con bên phải.

Khai báo:

```
Type BtreeLink = ^Node;

     Node =  Record
                  Data:ElementType;
                  Left,Right:BtreeLink;
             End;

Var      Root : BtreeLink;
```



Hình 3.9 Biểu diễn cây nhị phân bằng liên kết

Để quản lý một cây ta chỉ cần quản lý nút gốc của cây.

8.3. Duyệt cây nhị phân

Duyệt cây là thăm qua tất cả các đỉnh của cây, mỗi đỉnh đúng một lần. Thao tác duyệt cây là một trong những thao tác quan trọng của cây nói chung và cây nhị phân nói riêng. Phụ thuộc vào thứ tự duyệt nút gốc của cây, các thuật toán duyệt cây thường dùng gồm:

+ Duyệt theo thứ tự trước (PreOrder).

+ Duyệt theo thứ tự giữa (InOrder).

+ Duyệt theo thứ tự sau (PostOrder).

8.3.1. Duyệt cây nhị phân theo thứ tự trước

Thuật toán đệ qui:

Nếu cây không rỗng thì thực hiện:

+ Thăm nút gốc.

+ Duyệt theo thứ tự trước cây con trái.

+ Duyệt theo thứ tự trước cây con phải.

Ví dụ với cây nhị phân cho trong hình 3.9. khi duyệt theo thứ tự trước sẽ lần lượt thăm qua các đỉnh : ABDHEIJCFGK. Thủ tục đệ quy duyệt cây nhị phân theo thứ tự trước như sau:

```
Procedure TraversePreOrder(root:BtreeLink);
```

```
Begin
   If root<>nil then
   Begin
      Visit(root);
      TraversePreOrder(Root^.left);
      TraversePreOrder(Root^.right);
   End;
End;
```

Ta có thể dùng một ngăn xếp để khử đệ quy thuật toán duyệt theo thứ tự trước.

Thuật toán không đệ quy:

Dùng một ngăn xếp S mỗi phần tử của ngăn xếp là một nút. Khi thăm đỉnh gốc, trước khi duyệt theo nhánh trái của mỗi cây ta lưu nút gốc của cây con phải (nếu có) vào ngăn xếp để quay lại duyệt khi hết nhánh trái. Chi tiết thuật toán như sau:

Xuất phát từ gốc của cây.

Khởi tạo ngăn xếp rỗng.

Lặp khi cây còn khác rỗng

+ Thăm nút gốc.

+ Nếu cây con phải khác rỗng thì đưa vào ngăn xếp.

+ Chuyển sang gốc cây con trái.

+ Nếu nút gốc là nil và ngăn xếp không rỗng thì lấy cây con từ ngăn xếp.

Thủ tục duyệt cây theo thứ tự trước không đệ quy như sau:

```
Procedure TraversePreOrder(root:BtreeLink);
Var S : StackOfNodes; p : BtreeLink;
   Begin
      p:=Root;
      Init(S);
      While (p<>nil)do
      begin
         Visit(p);
         if (p^.right<>nil) then Push(S,p^.right);
         p:=p^.left;
         if (p=nil) and not empty(S) then Pop(s,p)
      end;
   End;
```

8.3.2. Duyệt cây nhị phân theo thứ tự giữa

Thuật toán (Đệ qui):

Nếu cây không rỗng thì thực hiện:

+ Duyệt theo thứ tự giữa cây con trái.

+ Thăm nút gốc.

+ Duyệt theo thứ tự giữa cây con phải.

Ví dụ với cây nhị phân cho trong hình 3.9 khi duyệt theo thứ tự giữa sẽ lần lượt thăm qua các nút: DHBIEJAFCKG. Thủ tục đệ quy duyệt cây nhị phân theo thứ tự giữa như sau:

```
Procedure TraverseInOrder(root:BtreeLink);
Begin
   If root<>nil then
   Begin
      TraverseInOrder(Root^.left);
      Visit(root);
      TraverseInOrder(Root^.right);
   End;
End;
```

Ta có thể dùng một ngăn xếp để khử đệ quy thuật toán duyệt theo thứ tự giữa.

Thuật toán không đệ quy:

Dùng một ngăn xếp S mỗi phần tử của ngăn xếp là một nút. Trước khi duyệt theo nhánh trái của mỗi cây ta lưu nút gốc vào ngăn xếp để quay lại duyệt khi hết nhánh trái. Chi tiết thuật toán như sau:

Xuất phát từ nút gốc của cây.

Khởi tạo ngăn xếp rỗng

Lặp

Lặp khi cây còn khác rỗng

+ Đưa gốc vào ngăn xếp.

+ Chuyển sang gốc cây con trái.

Nếu ngăn xếp không rỗng thì

+ Lấy cây con từ ngăn xếp.

+ Thăm nút gốc của cây.

+ Chuyển sang cây con phải.

Đến khi ngăn xếp rỗng và cây khác rỗng thì dừng

Thủ tục duyệt cây theo thứ tự giữa không đệ quy như sau:

```
Procedure TraverseInOrder(root:BtreeLink);
Var S : StackOfNodes; p : BtreeLink;
   Begin
      p:=Root;
      Init(S);
      Repeat
         While p<>nil do
            begin
```

```
            Push(S,p);
            p:=p^.left;
        end;
        if not empty(S) then
            begin
                Pop(S,p);
                Visit(p);
                p:=p^.right;
            end;
    Until Empty(S) and (p=nil);
End;
```

8.3.3. Duyệt cây nhị phân theo thứ tự sau

Thuật toán đệ qui:

Nếu cây không rỗng thì thực hiện:

+ Duyệt theo thứ tự sau cây con trái.

+ Duyệt theo thứ tự sau cây con phải.

+ Thăm nút gốc.

Ví dụ với cây nhị phân cho trong hình 3.9 khi duyệt theo thứ tự sau sẽ lần lượt thăm qua các nút: HDIJEBFKGCA. Thủ tục đệ quy duyệt cây nhị phân theo thứ tự sau như sau:

```
Procedure TraversePostOrder(root:BtreeLink);
Begin
    If root<>nil then
    Begin
        TraversePosOrder(Root^.left);
        TraversePosOrder(Root^.right);
        Visit(root);
    End;
End;
```

Thuật toán không đệ quy:

Tương tự như hai thao tác duyệt trên, với thuật toán duyệt cây nhị phân theo thứ tự sau ta cũng có thể khử đệ quy bằng cách dùng một ngăn xếp S. Do cây con phải được duyệt trước nút gốc nên mỗi bước, trước khi duyệt theo nhánh trái ta lưu nút gốc và con phải (nếu có) vào vào ngăn xếp (nút con phải lưu sau để lấy ra trước). Khi lấy từ ngăn xếp ra để duyệt tiếp ta phải phân biệt nút lấy ra là nút gốc hay nút con phải. Nếu là nút gốc thì thăm còn nếu là nút phải thì tiếp tục duyệt theo nhánh trái. Do đó mỗi phần tử của ngăn xếp gồm hai phần: thành phần dùng để chứa nút và một thành phần để phân biệt nút đưa vào là nút gốc hay nút con phải (thành phần này kiểu Boolean). Chi tiết thuật toán như sau:

Xuất phát từ nút gốc của cây.

Khởi tạo ngăn xếp rỗng.

isRoot:=false

Lặp

Lặp khi cây còn khác rỗng và không phải gốc (not isRoot)

+ Đưa gốc vào ngăn xếp.

+ Nếu nút con phải khác rỗng thì đưa vào ngăn xếp

+ Chuyển sang gốc cây con trái.

Nếu gốc khác rỗng thì thăm nút gốc.

Nếu ngăn xếp không rỗng thì

+ Lấy cây con và giá trị isRoot từ ngăn xếp.

Đến khi ngăn xếp rỗng và cây rỗng thì dừng

Thủ tục duyệt cây theo thứ tự trước không đệ quy như sau:

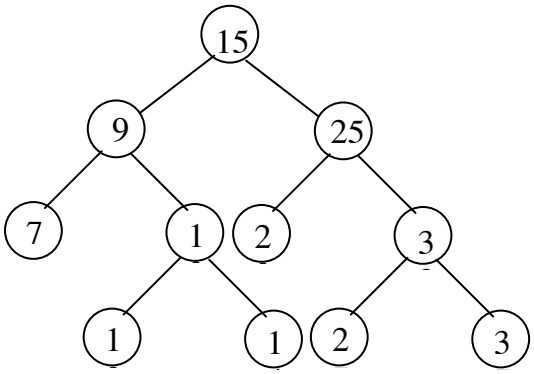
```
Procedure TraversePostOrder(root:BtreeLink);
Var S : StackOfNodes; p:BtreeLink; isRoot:Boolean;
   Begin
      p:=Root;
      Init(S);
      isRoot:=false;
      Repeat
         While (p<>nil) and not isRoot do
            begin
               Push(S,p,true);
               if p^.right<>nil then
                  Push(S,p^.right,false);
               p:=p^.left;
            end;
            if p<>nil then Visit(p);
            if not empty(S) then
               Pop(s,p,isRoot);
      Until Empty(S);
   End;
```

8.4. Cây tìm kiếm nhị phân

Cây tìm kiếm nhị phân là một dạng cây nhị phân được tổ chức theo một trật tự nào đó của các nút để thuận lợi cho việc tìm kiếm. Trong phần này ta giả sử dữ liệu tại một nút của cây có thành phần Key là khóa của các phần tử. Giả sử trên cây các nút không có hai phần tử trùng khóa. Khái niệm cây tìm kiếm nhị phân (Binary Search Tree - BST) được định nghĩa như sau:

Cây tìm kiếm nhị phân là một cây nhị phân mà tại mỗi cây con của nó thỏa điều kiện: khoá của nút gốc lớn hơn khoá của tất cả các nút của cây con trái và nhỏ hơn khoá của tất cả các nút của cây con phải.

Ví dụ: Cho cây tìm kiếm nhị phân như hình sau



Hình 3.10 Cây tìm kiếm nhị phân

8.5. Các thao tác trên cây tìm kiếm nhị phân

Ngoài thao tác như trên cây nhị phân, trên cây tìm kiếm nhị phân thường thực hiện các thao tác sau:

+ Tìm một nút trên cây theo khóa.

+ Thêm một nút vào cây.

+ Xóa một nút trên cây.

8.5.1. Tìm một nút trên cây tìm kiếm nhị phân

Cho trước một cây tìm kiếm nhị phân có nút gốc là root. Ta cần tìm một nút có khoá là x trong cây.

Thuật toán đệ quy:

Nếu khóa của nút gốc bằng khóa x thì thuật toán dừng, kết quả là tìm thấy.

Nếu khóa của nút gốc lớn hơn khóa x thì tìm khóa x ở cây con trái.

Nếu khóa của nút gốc nhỏ hơn khóa x thì tìm khóa x ở cây con phải.

Nếu cây rỗng thì thuật toán dừng, kết quả là không tìm thấy.

Thủ tục đệ quy tìm khóa x trong cây tìm kiếm nhị phân có nút gốc là root, kết quả tìm được hay không trả về qua tham biến found và p là nút tìm được (nếu tìm thấy).

```
Procedure Search(x:KeyType; root:Btree;
                                        var found:boolean;var p:Btree);
  Begin
      p:=root;
      if p<>nil then
         if p^.Data.Key=x then found:=true
         else
            if p^.Data.Key>x then
               Search(x,p^.left,found,p)
            else
               Search(x,p^.right,found,p)
```

```
        else found:=false;
    End;
```

Thủ tục không đệ quy tìm một khóa trong cây tìm kiếm nhị phân:

```
 Procedure Search(x:KeyType; root:Btree;
                                          var found:boolean; var p:Btree);
Begin
  p:=root;
  Found:=false;
  While (not found) and (p<>nil) do
   If  p^.key=x then
                found:=true
             else
     If p^.key<x then
       p:=p^.right
                   else
                       P:=p^.left;
     End;
```

8.5.2. Thêm một nút vào cây tìm kiếm nhị phân

Cho một cây tìm kiếm nhị phân có nút gốc là root, thêm vào cây một nút có dữ liệu là x sao cho sau khi thêm cũng là cây tìm kiếm nhị phân.

Thuật toán đệ quy:

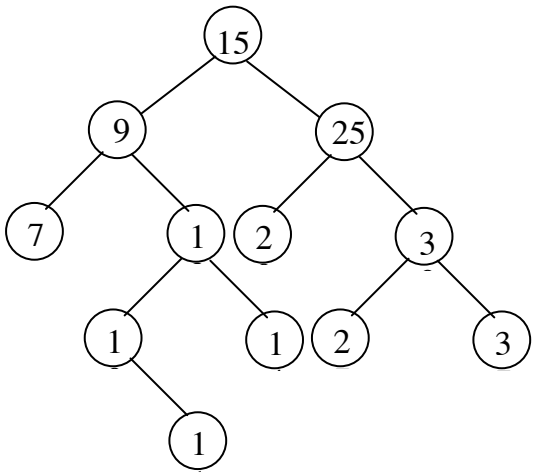
Nếu cây rỗng thì x là nút gốc của cây, thuật toán dừng.

Nếu khóa của nút gốc bằng khóa của x thì thuật toán dừng (nút đã có).

Nếu khóa của nút gốc lớn hơn khóa của x thì thêm x vào cây con trái.

Nếu khóa của nút gốc nhỏ hơn khóa của x thì thêm x vào cây con phải.

Ví dụ: Cây ở hình 3.19 sau khi thêm nút 11



Hình 3.11 Thêm vào cây tìm kiếm nhị phân

Thủ tục đệ quy thêm nút x vào cây tìm kiếm nhị phân:

```
Procedure Insert(x:ElementType; var root:BtreeLink);
Var p:BTreeLink;
   Begin
      if root=nil then
         begin
            New(root);
            root^.Data:=x;
            root^.left:=nil;
            root^.right:=nil;
         end
      else
         if root^.Data.Key>x.Key then
            Insert(x, root^.left)
         else
            if root^.Data.Key<x.Key then
               Insert(x, root^.right);
   End;
```

Có thể thay thủ tục đệ quy bằng thủ tục lặp sau:

```
Procedure Insert(x:ElementType;var root:BTreeLink);
   Var p,q:BTreeLink;
Begin
  New(p);
  p^.Data:=x;
  p^.left:=nil;
  p^.right:=nil;
          if root=nil then root:=p
          else
             begin{tìm vị trí cần chèn}
                q:=root;
                while q<>nil do
                begin
             if q^.Data.key > x then
          if q^.left <> nil then q:=q^.left
          else
                          begin
       q^.left:=p;
                          q:=nil;{để thoát khỏi vòng lặp}
      end
    else
                          if q^.key<x then
       if q^.right<>nil then q:=q^.right
       else
                               begin
        q^.right:=p;
        q:=nil;
       end
```

```
        else begin q:=nil; dispose(p); end;
    end;
End;
```

8.5.3. Xóa một nút trên cây tìm kiếm nhị phân

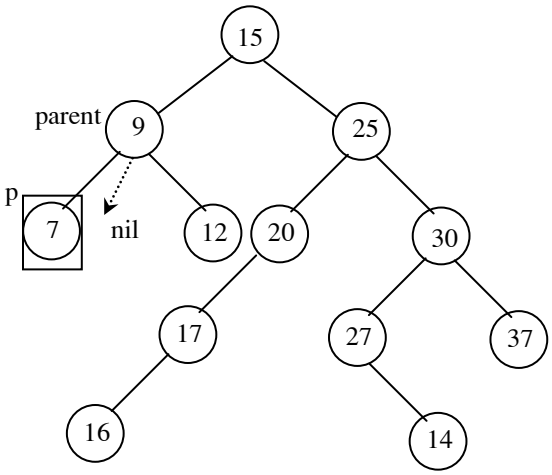
Cho một cây tìm kiếm nhị phân có nút gốc là root, xóa nút có khóa là x trên cây sao cho sau khi xóa cây vẫn còn thỏa các tính chất của cây tìm kiếm nhị phân. Để xóa, trước tiên ta phải tìm nút có khóa x cần xóa (dùng thuật toán tìm kiếm trên cây) và ta chỉ thực hiện thao tác xóa nếu tìm được. Giả sử p là nút có khóa x cần xóa. Để xóa nút p, ta xét ba trường hợp:

+ p là nút lá của cây.

+ p có một cây con trái hoặc phải.

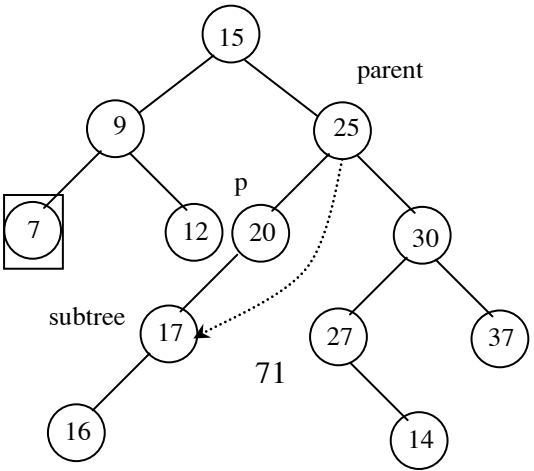
+ p có cả hai cây con trái và phải.

Trường hợp 1. Nếu p là nút lá và giả sử parent là nút cha của p. Trong trường hợp này ta chỉ cần đặt con trỏ trái hoặc phải của parent đến nil tùy thuộc vào p là nút con trái hay phải của parent. Sau đó giải phóng ô nhớ nút p. Nếu nút cần xóa là nút gốc thì đây là nút duy nhất của cây nên sau khi xóa cây sẽ rỗng. Hình sau cho ta thấy hình ảnh cây sau khi xóa nút có khóa là 7.



Hình 3.12 Xóa nút lá trong cây

Trường hợp 2. Khi p có đúngmột nút con, giả sử parent là nút cha của p và subtree là nút con khác nil của p. Trong trường hợp này ta chỉ cần đặt con trỏ trái hoặc phải của nút parent vào nút con của p là subtree tùy thuộc vào p là nút con trái



hay con phải của parent. Sau đó giải phóng ô nhớ của nút p. Trong hình sau, để xóa nút 20 ta liên kết bên trái của nút 25 đến nút 17.

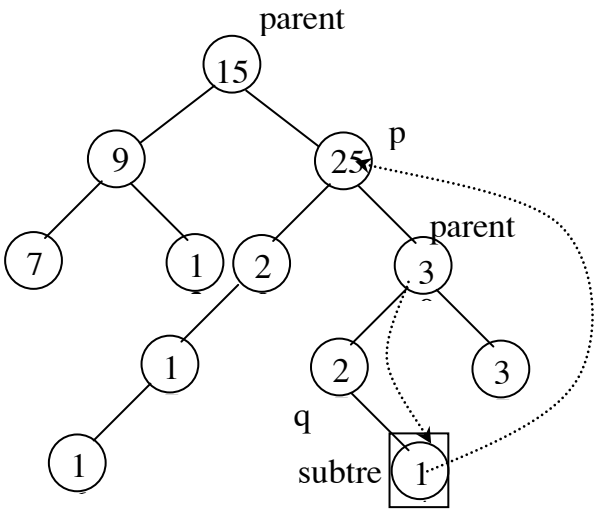
Hình 3.13 Xóa nút có một nút con

Trong trường hợp nút p là gốc của cây, khi đó chỉ cần chuyển gốc của cây về nút con khác nil của nút p sau đó giải phóng p.

Ta có thể kết hợp hai trường hợp 1 và 2 thành một trường hợp trong đó nút p có không quá một cây con khác rỗng bởi đoạn lệnh sau:

```
subtree:=p^.Left;
if subtree= nil then subtree:=p^.Right;
if parent=nil then {xóa nút gốc}
  Root:=subtree
else
  if parent^.Left=p then
      parent^.Left:=subtree
  else
      parent^.Right:=subtree;
```

Trường hợp 3. Nếu nút cần xóa p có cả hai cây con trái và phải. Để xóa nút p trong trường hợp này ta đưa về một trong hai dạng trên bằng cách tìm nút có khóa lớn nhất của các nút trong cây con trái của nút p (hoặc nút có khóa nhỏ nhất của các nút trong cây con phải) giả sử nút này là q. Dễ thấy nút q có không quá một cây con vì nó là nút tận cùng bên phải của cây con bên trái p (hoặc là nút tận cùng bên trái của cây con bên phải p). Để xóa nút p ta thay bằng chuyển dữ liệu của nút q lên nút p rồi xóa nút q như trong trường hợp 1 hoặc 2. Hình dưới minh họa thao tác xóa nút có hai cây con.



Hình 3.14 Xóa nút có hai cây con

Thủ tục dưới đây cài đặt thao tác xóa nút có khóa x trong cây tìm kiếm nhị phân có nút gốc là Root cho tất cả các trường hợp.

```
Procedure Delete(var root : BTreeLink; x : KeyType);var  p,q,parent,subtree :
BT;Begin  (* Tim nut can xoa p, parent la nut cha cua p *)   p :=r;
        parent:=nil;          while (p<>nil) and (p^.Data.Key<>x) do
    begin
        parent:=p;
        if p^.Data.Key < x then p := p^.right
        else if p^.Data.Key > x then p := p^.left;
    end;
    if p=nil then writeln('Khong co phan tu can xoa')
    else
    begin
    (* Truong hop nut can xoa co 2 nut con :
     - Tim nut q la nut trai nhat cua cay con ben phai.
     - parent la nut cha cua nut q. *)

      if (p^.left<>nil) and (p^.right<>nil) then
        begin
          q:=p^.right;
                    parent:=p;                          while q^.left <> nil do
            begin
              parent:=q;
            q:=q^.left;
              end;

                (* Dua du lieu cua nut q vao nut p *)
        p^.Data := q^.Data;                                                  p := q;
    end;    (* Xoa nut p trong truong hop khong qua mot nut con *)

        { Tim cay con cua p}

        subtree := p^.left;
        if subtree = nil then subtree := p^.right;

        if parent = nil then {nut can xoa la nut goc}
          root := subtree
        else
        begin
                {p  la  nut  trai  cua  parent  }          if parent^.Data.Key >
p^.Data.Key then
              parent^.left := subtree          else          parent^.right := subtree;
end;      dispose(p);   end;
    End;
```

Có thể dùng thủ tục đệ quy như sau:

```
    Procedure Delete(var root : BTreeLink; x : KeyType);
Begin
          if Root<>nil then
```

```
            if x < Root^.Data.Key then Delete(Root^.left,x)
            else
                if x > Root^.Data.Key then
                    Delete(Root^.right,x)
                else Del(Root);
    End;
```

Thủ tục Del xóa nút p trong cây.

```
 Procedure Del(var P : BtreeLink);
 var q, q1 : BtreeLink;
    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
                   begin
                      p^.Data:=q^.Data;
                       p^.left:=q^.left;
                   end
                else
                   begin
                      repeat
                         q1:=q;
                         q:=q^.right;
                      until q^.right=nil;
                      p^.Data:=q^.data;
                      q1^.right:=q^.left;
                   end;
             end;
       Dispose(q);
    End;
```

9. CÂY CÂN BẰNG

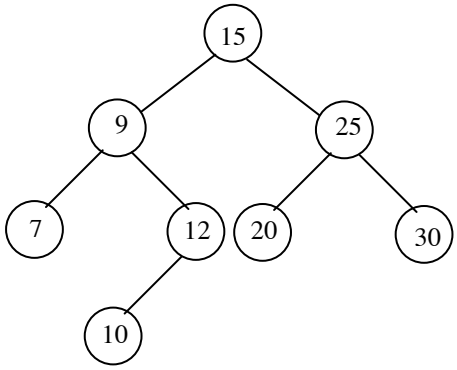
Với các thao tác trên cây tìm kiếm nhị phân đã tìm hiểu ở phần trước ta dễ thấy các thao tác thường dùng trên cây là tìm, thêm và xóa phụ thuộc nhiều vào "hình dáng" của cây. Trong trường hợp cây suy biến thì các thao tác có độ phức

tạp tương đương trên danh sách tuần tự. Trong nhiều trường hợp, các thao tác thêm vào và xóa có thể làm cây "lệch" đi, độ cao của cây tăng lên trong khi số các nút không nhiều. Điều này làm ảnh hưởng rất lớn đến các thao tác trên cây tìm kiếm nhị phân. Một nhu cầu hiển nhiên là các thao tác trên cây tìm kiếm nhị phân phải được cài đặt như thế nào để hạn chế trường hợp cây "lệch" hay phải điều chỉnh để cây "cân bằng". Lớp các cây này được hai nhà toán học người Nga là G.M. Adelsen-Velskii và E.M. Landis đưa ra vào năm 1962.

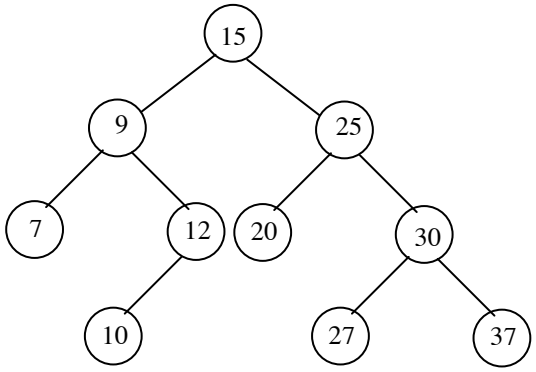
9.1. Khái niệm

Cây cân bằng hoàn toàn: Một cây tìm kiếm nhị phân được gọi là cây cân bằng hoàn toàn nếu và chỉ nếu với mọi nút của cây số nút của cây con bên trái và số nút của cây con bên phải lệch nhau không quá 1.

Hình dưới đây minh họa cây cân bằng hoàn toàn và cây không cân bằng hoàn toàn.



Hình 3.15 a) Cây cân bằng hoàn toàn



Hình 3.15 b) Cây không cân bằng hoàn toàn

Một cây cân bằng hoàn toàn thì các thao tác tìm kiếm, thêm và xóa được thực hiện rất thuận lợi. Tuy nhiên thuật toán để duy trì cây cân bằng hoàn toàn đối với thao tác thêm và xóa là rất phức tạp. Một dạng cây khác được các tác giả trên đưa ra là cây cân bằng được định nghĩa như sau.

Một cây tìm kiếm nhị phân được gọi là *cây cân bằng* nếu và chỉ nếu với mọi nút của cây chiều cao cây con trái và chiều cao cây con phải lệch nhau không quá 1.

Nếu tại một nút bất kỳ của cây, ta gọi: h_L, h_R tương ứng là chiều cao cây con trái và cây con phải thì điều kiện để cây cân bằng là $h_L=h_R$-1 hoặc $h_L=h_R$ hoặc $h_L=h_R$+1. Những cây thỏa điều kiện này còn được gọi là cây AVL.

Từ điều kiện trên ta thấy nếu cây cân bằng hoàn toàn thì nó là cây cân bằng. Tuy nhiên, điều ngược lại không đúng. Chẳng hạn cây ở hình 3.15 b) là cây cân bằng nhưng không cân bằng hoàn toàn.

Cây cân bằng cũng là một cách tổ chức tốt cho cây tìm kiếm nhị phân. Trong đa số các trường hợp tính toán trung bình, các thao tác trên cây cân bằng và cây cân bằng hoàn toàn không chênh lệch nhau nhiều tuy nhiên các thuật toán trên cây cần bằng thì đơn giản hơn nhiều. Do đó được đa số người lập trình sử dụng

như một cách tổ chức tốt cho cây tìm kiếm nhị phân. Trong phần sau sẽ đề cập đến hai thuật toán thường dùng trên cây cân bằng là thêm một nút và xóa một nút.

Để đơn giản trong các thao tác kiểm tra và cân bằng cây, tại mỗi nút của cây ta lưu hệ số cân bằng (bal) là hiệu của chiều cao cây con phải và cây con trái ($bal=h_R-h_L$). Vì cây luôn được cân bằng lại sau mỗi thao tác nên hệ số cân bằng của mỗi nút là -1, 0 hoặc 1. Khai báo kiểu dữ liệu cho cây cân bằng tương tự cây tìm kiếm nhị phân.

```
Type BalanceTree = ^Node;

     Node =  Record
                  Data:ElementType;
                  Bal : -1..1;
                  Left,Right:BalenceTree;
             End;

Var      Root : BalanceTree;
```

9.2. Thêm vào cây cân bằng

Một trong những khả năng làm cho cây tìm kiếm nhị phân mất tình trạng cân bằng là khi thêm vào một nút. Trong phần này ta sẽ xem xét các khả năng có thể xảy ra khi thêm một nút vào cây cân bằng. Trong một số trường hợp, khi thêm không làm mất tính cân bằng của cây nhưng làm thay đổi hệ số cân bằng của một số nút liên quan. Có những trường hợp khi thêm vào làm cho cây không cân bằng ở một số cây con. Khi đó ta phải cân bằng lại cây theo nguyên tắc từ cây con không cân bằng nhỏ nhất. Mỗi khi cân bằng phải tính toán lại các hệ số cân bằng của các nút liên quan. Sau đây chúng ta xem xét chi tiết các trường hợp.

Giả sử nút thêm vào ở cây có nút gốc là p. Có thể xảy ra những trường hợp sau:

Trường hợp 1. p=nil (thêm vào cây trống). Khi thêm vào một đỉnh thì cây ở tình trạng cân bằng và hệ số cân bằng tại nút này là 0.

Trường hợp 2. p≠nil và tại p hệ số cân bằng là 0 (p^.bal=0). Trong trường hợp này, khi thêm nút mới vào cây gốc p không làm mất tính cân bằng của cây tại nút p (các cây con có thể không còn cân bằng).

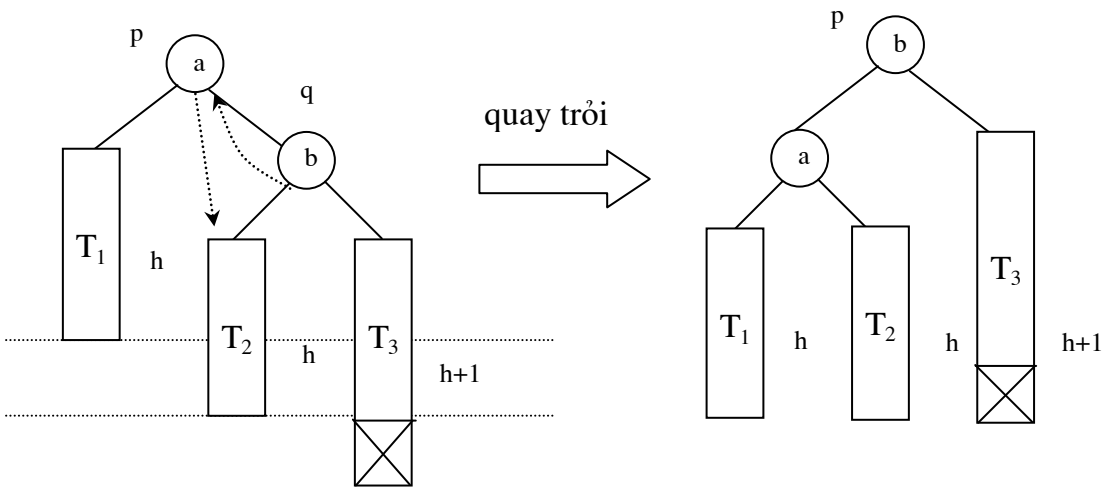
Trường hợp 3. p≠nil và tại p có p^.bal=1 (hoặc p^.bal=-1). Trong trường hợp này, nếu khi thêm nút mới vào cây gốc p không làm tăng độ cao cây con phải (hoặc cây con trái) thì tại nút p tính cân bằng vẫn không ảnh hưởng.

Trường hợp 4. p≠nil và tại p có p^.bal=1 (hoặc p^.bal=-1). Giả sử nút mới thêm vào cây con phải (hoặc trái) của p và làm tăng chiều cao cây con. Trong trường hợp này thì tại nút p tính cân bằng bị phá vỡ.

Giả sử trong trường hợp 4 ta xét cụ thể hơn: p^.bal =1 và việc thêm nút làm tăng chiều cao cây con phải của p (trường hợp p^.bal =-1 và việc thêm nút làm tăng chiều cao cây con trái của p thực hiện tương tự dành cho độc giả).

Gọi q là nút con bên phải của p (q=p^.right). Dễ thấy q khác nil vì p^.bal=1. Sau khi thêm nút và làm tăng chiều cao cây con phải, ta xét các khả năng sau.

a) Tại q có q^.bal = 1 (cây con gốc q cao bên phải). Trong trường hợp này giả sử cây con trái của p (T_1) có chiều cao là h thì cây con trái của q (T_2) có chiều cao h và cây con phải của q (T_3) có chiều cao h+1 (sau khi thêm vào), xem hình



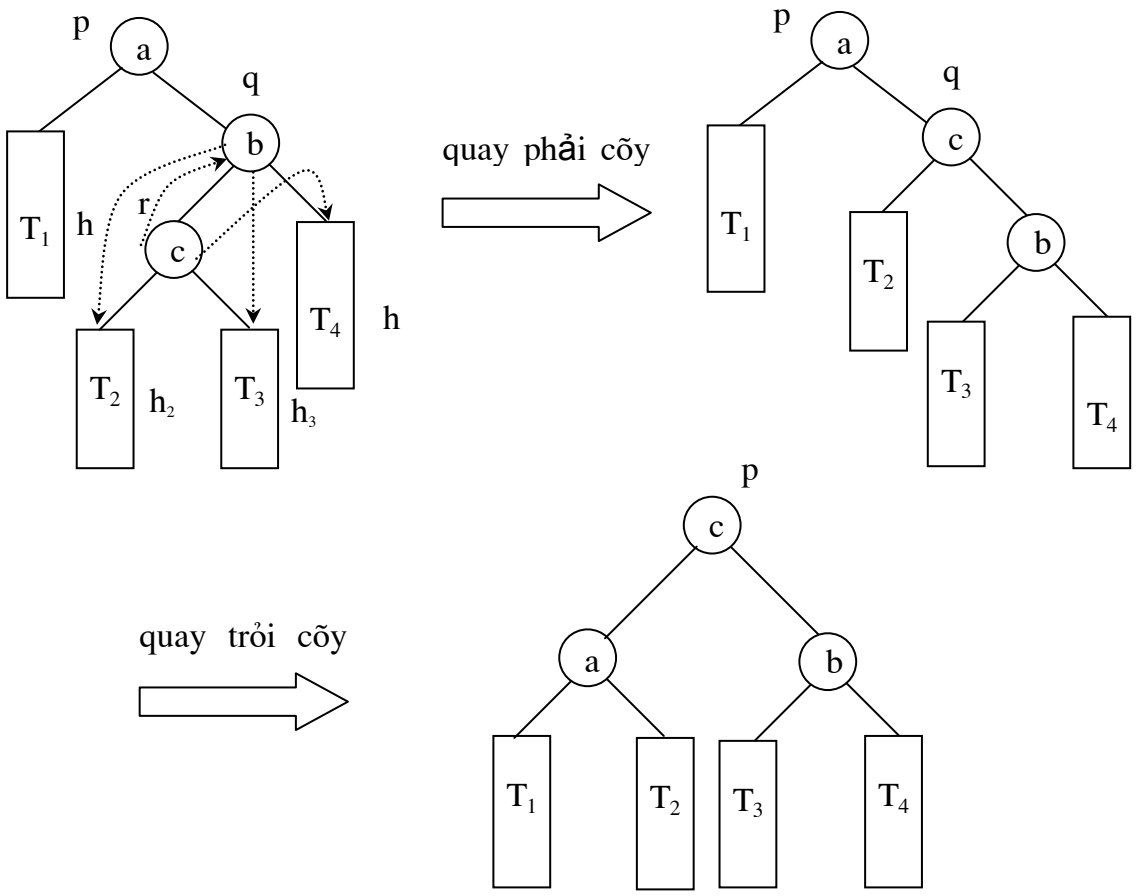
3.16.

Hình 3.16 Quay trái cây p

Trong trường hợp này bằng phép quay trái (RotateLeft) tại nút p thì nút a và b đều ở trạng thái cân bằng (bal=0). Thủ tục quay trái có thể viết như sau:

```
Procedure RotateLeft(var p : BalanceTree);
var q : BalanceTree;
 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;
```

b) Tại q có q^.bal=-1. Trong trường hợp này, gọi r là nút gốc cây con trái của q (vì q^.bal=-1 nên r khác rỗng). Ký hiệu cây con trái của p là T_1, cây này có chiều cao là h; cây con trái của r là T_2 có chiều cao h_2; cây con phải của r là T_3 có chiều cao h_3; cây con phải của q là T_4. Khi đó, theo giả thiết trên thì nút vừa thêm sẽ nằm ở cây T_2 hoặc T_3 và ít nhất một trong 2 cây này có chiều cao là h và T_4 có chiều cao là h (xem hình 3.17).



Hình 3.17

Trong trường hợp này, đầu tiên ta quay cây con gốc q sang phải, sau đó quay cây gốc p sang trái. Vì hai cây T_1 và T_4 có chiều cao h và ít nhất một trong hai cây T_2 và T_3 có chiều cao h nên sau khi cân bằng với hai phép quay trên kết quả tại p hệ số cân bằng là 0; hệ số cân bằng của nút a là 0 hoặc -1 và của nút b là 0 hoặc 1 tùy thuộc vào độ cao cây con T_2 và T_3 là h hay h-1. Thủ tục quay trái đã được trình bày ở trên, thủ tục quay phải tại một gốc cây con như sau:

```
Procedure RotateRight(var q : BalanceTree);
var r : BalanceTree;
  Begin
     if q<>nil then
        if q^.left<>nil then
           begin
              r:=q^.left;
              q^.left:=r^.right;
              r^.right:=q;
              q:=r;
           end;
  End;
```

c) Trường hợp q^.bal=0 không xảy ra vì theo giả thiết ở trên sau khi thêm nút vào cây con phải của p và làm tăng chiều cao cây con này.

Từ những trường hợp trên ta có thủ tục cân bằng tại đỉnh p khi nút thêm vào cây con phải của p.

```
Procedure RightBalance(var p : BalanceTree);
   var q, r : BalanceTree;
      Begin
         q:=p^.right;
         case q^.bal of
            1:  begin
                     p^.bal:=-1;
                     q^.bal:=-1;
                     RotateLeft(p);
                  end;
          -1:    begin
                     r:=q^.left;
                     case r^.bal of
                        0:begin
                              p^.bal:=0;
                              q^.bal:=0;
                           end;
                       -1:begin
                              p^.bal:=-1;
                              q^.bal:=1;
                           end;
                          1:begin
                              p^.bal:=-1;
                              q^.bal:=0;
                           end;
                     end; {case r^.bal}
                     r^.bal:=0;
                     RotateRight(q);
                     p^.right:=q;
                     RotateLeft(p);
                  end;
            end; {case q^.bal}
      End;
```

Hoàn toàn tương tự, khi p^.bal=-1 và nút mới thêm vào cây con trái của p và làm tăng chiều cao cây con trái thì ta lập lại tính cân bằng tại nút p bằng thủ tục LeftBalance như sau.

```
Procedure LeftBalance(var p : BalanceTree);
   var q, r : BalanceTree;
      Begin
         q:=p^.left;
         case q^.bal of
```

```
        -1:   begin
                    p^.bal:=0;
                    q^.bal:=0;
                    RotateRight(p);
                end;
          1:  begin
                    r:=q^.right;
                    case r^.bal of
                        0:begin
                              p^.bal:=0;
                              q^.bal:=0;
                          end;
                     -1:begin
                              p^.bal:=1;
                              q^.bal:=0;
                          end;
                        1:begin
                              p^.bal:=0;
                              q^.bal:=-1;
                          end;
                    end; {case r^.bal}
                    r^.bal:=0;
                    RotateLeft(q);
                    p^.left:=q;
                    RotateRight(p);
                end;
          end; {case q^.bal}
    End;
```

Trên cơ sở của hai thủ tục cân bằng trái và cân bằng phải tại nút gốc của một cây khi thêm vào một nút, tiếp theo chúng ta hoàn chỉnh thủ tục thêm một nút vào cây cân bằng sao cho sau khi thêm cây vẫn cân bằng.

Thuật toán thêm một nút vào cây cân bằng:

Đi theo đường tìm kiếm đến khi nhận biết nút cần thêm chưa có trên cây.

Thêm nút vào cây và xác định lại hệ số cân bằng.

Lần ngược theo đường tìm kiếm và kiểm tra hệ số cân bằng tại các nút.

Thủ tục đệ quy sau thực hiện thao tác thêm một nút x vào cây cân bằng có nút gốc là root. Để nhận biết sau khi thêm có làm tăng chiều cao của cây con không, trong thủ tục dùng tham biến taller kiểu boolean, taller=true nếu sau khi thêm nút x vào cây con của nút root tăng chiều cao và taller=false trong trường hợp ngược lại.

```
Procedure Insert(var root:BalanceTree; x:ElementType;
                                                    var taller:boolean);
  Begin
    if root=nil then
      begin
```

```
            new(root);
            root^.data:=x;
            root^.left:=nil;
            root^.right:=nil;
            root^.bal:=0;
            taller:=true;
        end
    else
        if x.Key<root^.Data.Key then
            begin
                Insert(root^.left, x, taller);
                if taller then {cây con trái cao lên}
                    case root^.bal of
                        -1:begin
                                LeftBalance(root);
                                taller:=false;
                           end;
                         0: begin
                                root^.bal:=-1;
                                taller:=true;
                           end;
                        1:  begin
                                root^.bal:=0;
                                taller:=false;
                           end;
                    end; {case}
            end
        else
            if x.Key>root^.Data.Key then
            begin
                Insert(root^.right,x,taller);
                if taller then {cây con phải cao lên}
                    case root^.bal of
                        -1:begin
                                root^.bal:=0;
                                taller:=false;
                           end;
                         0: begin
                                root^.bal:=1;
                                taller:=true;
                           end;
                        1:  begin
                                RightBalance(root);
                                taller:=false;
                           end;
                    end; {case}
            end
            else taller:=false; {nút đã có}
End;
```

Nhận xét: thủ tục này có một số chỗ kiểm tra thừa vì một khi sự cân bằng đã được thiết lập, thì kể từ đó không cần kiểm tra các nút cha còn lại. Vì thủ tục đệ quy nên việc lần ngược các nút trong quá trình tìm vị trí cần thêm được thực hiện dễ dàng.

9.3. Loại bỏ khỏi cây cân bằng

Từ thao tác xóa trên cây tìm kiếm nhị phân và thao tác thêm vào cây cân bằng ta có thể thực hiện tương tự cho thao tác xóa trên cây cân bằng. Tuy nhiên thao tác xóa trên cây cân bằng thực hiện phức tạp hơn. Trong thao tác xóa chúng ta vẫn sử dụng các thủ tục quay trái (RotateLeft) và quay phải (RotateRight) đã trình bày ở phần trên. Mỗi khi xóa một nút có thể làm chiều cao cây giảm và phá vỡ tính cân bằng của một số nút liên quan. Trong trường hợp này ta cũng thực hiện cân bằng lại cây tại các nút không cân bằng tương tự như thao tác thêm.

9.3.1. Cân bằng trái khi xóa

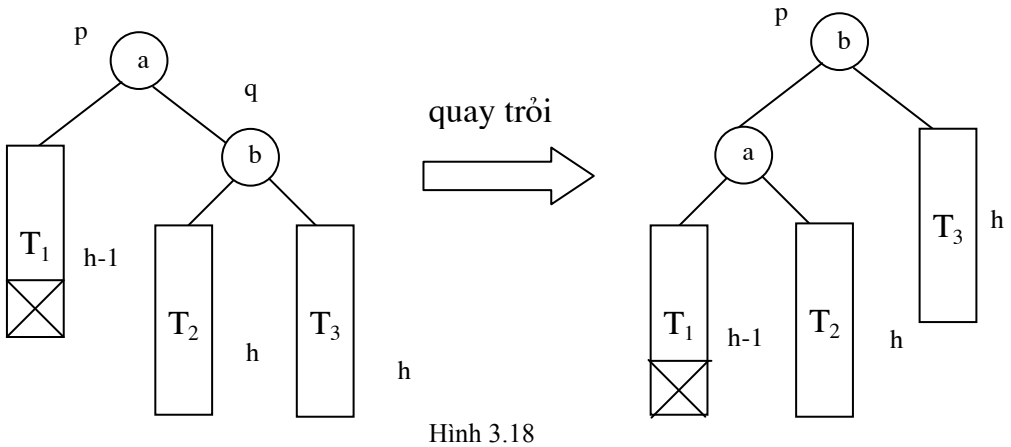
Trong phần này ta xét các trường hợp khi xóa một nút của cây có nút gốc p làm giảm chiều cao cây con trái. Ta xét các trường hợp sau:

a) Nếu tại nút p ta có p^.bal=-1 thì sau khi xóa, tại nút p cây vẫn ở trình trạng cân bằng và hệ số cân bằng tại p sau khi xóa là p^.bal=0. Trong trường hợp này sau khi xóa, chiều cao cây giảm.

b) Nếu tại nút p ta có p^.bal=0 thì sau khi xóa, tại nút p cây vẫn ở trình trạng cân bằng và hệ số cân bằng tại p sau khi xóa là p^.bal=1. Trong trường hợp này sau khi xóa, chiều cao cây gốc p không giảm.

c) Nếu tại nút p ta có p^.bal=1 thì sau khi xóa, tại nút p tính cân bằng vi phạm. Trong trường hợp này sau khi xóa, ta phải thực hiện cân bằng lại cây tại nút p. Gọi q là nút con bên phải của p. Ta có q khác rỗng vì p^.bal=1. Việc cân bằng cây trong trường hợp này phụ thuộc vào hệ số cân bằng tại q, cụ thể như sau:

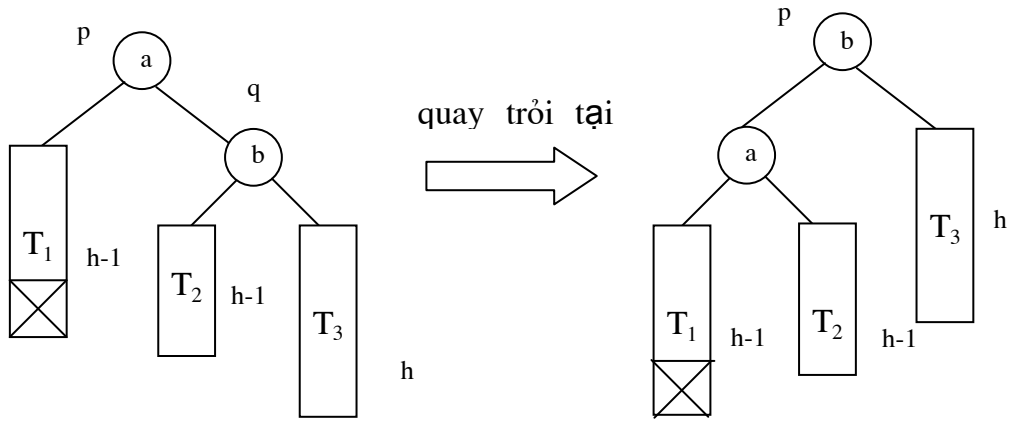
   i) Nếu q^.bal=0, khi đó sau khi xóa ta phải thực hiện thao tác quay trái tại p (xem hình 3.18).



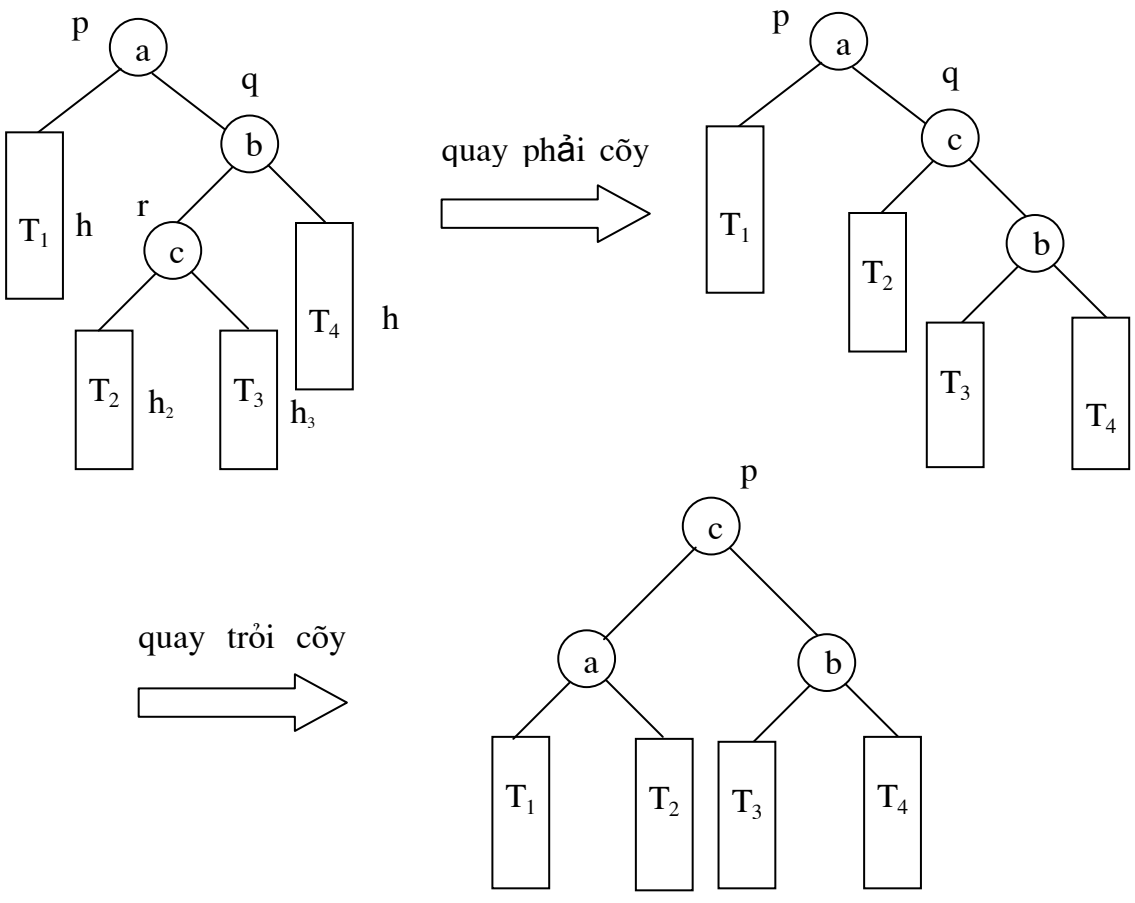
Hình 3.18

Sau khi quay trái tại p thì cây cân bằng tại p. Tại nút a hệ số cân bằng là 1 và tại b hệ số cân bằng là -1. Chiều cao cây gốc p không giảm.

ii) Nếu q^.bal =1, khi đó sau khi xóa ta phải thực hiện thao tác quay trái tại p (xem hình 3.19).



Hình 3.19



Hình 3.20

Sau khi quay trái tại p thì cây cân bằng, tại nút a và b hệ số cân bằng là 0. Sau khi quay chiều cao cây giảm.

iii) Nếu q^.bal =-1. Gọi r là nút con trái của q (r khác rỗng). Khi đó sau khi xóa ta thực hiện hai thao tác quay phải tại q và quay trái tại p (xem hình 3.20).

Sau khi cân bằng tại nút c hệ số cân bằng là 0, tại các nút a và b hệ số cân bằng tùy thuộc vào chiều cao của cây T_2 và T_3. Sau khi cân bằng chiều cao của cây giảm.

Từ các trường hợp đã phân tích trên, chúng ta có thể viết thủ tục cân bằng trái cho trường hợp xóa một nút làm giảm chiều cao cây con trái của nút p. Trong thủ tục này dùng một tham biến h kiểu boolean để chỉ độ cao cây sau khi cân bằng có giảm không. h=true nếu độ cao của cây giảm và h=false trong trường hợp ngược lại.

```
Procedure LeftBalance(var p:BalanceTree; var h:boolean);
   var q, r : BalanceTree;
      Begin
         case p^.bal of
          -1:    begin
                       p^.bal:=0;
                       h:=true;
                   end;
             0:  begin
                       p^.bal:=1;
                       h:=false;
                   end;
             1:  begin
                       q:=p^.right;
                       case q^.bal of
                          0:begin
                                 p^.bal:=1;
                                 q^.bal:=-1;
                                 RotateLeft(p);
                                 h:=false;
                             end;
                        1:begin
                                 p^.bal:=0;
                                 q^.bal:=0;
                                 RotateLeft(p);
                                 h:=true;
                             end;
                       -1:begin
                                 r:=q^.left;
                                 case r^.bal of
                                    0:begin
                                           p^.bal:=0;
                                           q^.bal:=0;
```

```
                    end;
                -1:begin
                        p^.bal:=0;
                        q^.bal:=1;
                    end;
                1:begin
                        p^.bal:=-1;
                        q^.bal:=0;
                    end;
                end; {case r^.bal}
                r^.bal:=0;
                RotateRight(q);
                p^.right:=q;
                RotateLeft(p);
                h:=true;
            end;
        end; {case q^.bal}
    end;
end; {case p^.bal}
End;
```

9.3.2. Cân bằng phải khi xóa

Trong trường hợp xóa một nút thuộc cây con phải của p và làm giảm độ cao của cây con phải, các trường hợp xảy ra tương tự như trong phần trên. Chi tiết các trường hợp các bạn tự liệt kê, sau đây là thủ tục cân bằng phải để cân bằng cây trong trường hợp này.

```
Procedure RightBalance(var p:BalanceTree;var h:boolean);
  var q, r : BalanceTree;
    Begin
      case p^.bal of
        1:   begin
                p^.bal:=0;
                h:=true;
             end;
        0:  begin
                p^.bal:=-1;
                h:=false;
             end;
       -1:   begin
                q:=p^.left;
                case q^.bal of
                  0:begin
                        p^.bal:=-1;
                        q^.bal:=1;
                        RotateRight(p);
                        h:=false;
                    end;
```

```
                -1:begin
                        p^.bal:=0;
                        q^.bal:=0;
                        RotateRight(p);
                        h:=true;
                    end;
                1:begin
                        r:=q^.right;
                        case r^.bal of
                            0:begin
                                    p^.bal:=0;
                                    q^.bal:=0;
                                end;
                             -1:begin
                                    p^.bal:=1;
                                    q^.bal:=0;
                                end;
                            1:begin
                                    p^.bal:=0;
                                    q^.bal:=-1;
                                end;
                        end; {case r^.bal}
                        r^.bal:=0;
                        RotateLeft(q);
                        p^.left:=q;
                        RotateRight(p);
                        h:=true;
                    end;
                end; {case q^.bal}
            end;
        end; {case p^.bal}
End;
```

9.3.3. Xóa một nút trên cây cân bằng

Thủ tục xóa một nút có khóa x khỏi cây cân bằng root được viết dưới dạng đệ quy và sử dụng các thủ tục LeftBalance và RightBalance đã trình bày ở phần trước.

```
Procedure Delete(var root:BalanceTree; x:KeyType;
                                                    var h:Boolean);
  Begin
    if root<>nil then
      if x<root^.Data.key then
        begin
          Delete(root^.left,x,h);
          if h then LeftBalance(root,h);
        end
      else
```

```
            if x>root^.Data.Key then
                begin
                    Delete(root^.right,x,h);
                    if h then RightBalance(root,h);
                end
            else Del(root,h);
End;
```

Trong đó thủ tục Del(p,h) dùng để xóa nút p trong cây và tham biến h cho biết sau khi xóa chiều cao cây có giảm không. Trong thủ tục Del dùng một thủ tục con Erase trong trong trường hợp nút p có hai cây con trái và phải. Thủ tục erase xóa đỉnh ngoài cùng bên phải của cây con trái của p. Nhưng trước khi xóa, dữ liệu của nút đó được đưa vào nút p.

```
Procedure Del(var p:BalanceTree; var h:Boolean);
    Procedure Erase(var q:BalanceTree; var h:Boolean);
        var q1:BalanceTree;
        Begin
            if q^.right<>nil then
                begin
                    Erase(q^.right,h);
                    if h then RightBalance(q,h);
                end
            else
                begin
                    p^.Data:=q^.Data;
                    q1:=q;
                    q:=q^.left;
                    h:=true;
                    dispose(q1);
                end;
        End;

    Begin {thủ tục Del}
        if p^.right=nil then
            begin
                q:=p;
                p:=p^.left;
                h:=true;
                dispose(q);
            end
        else
            if p^.left=nil then
                begin
                    q:=p;
                    p:=p^.right;
                    h:=true;
                    dispose(q);
                end
```

```
        else {p có 2 cây con}
            begin
                Erase(p^.left,h);
                if h then LeftBalance(p,h);
            end
End;
```

10. CÁC ỨNG DỤNG CỦA CÂY NHỊ PHÂN

Cấu trúc cây có nhiều ứng dụng trong tin học, trong phần này chúng tôi trình bày một ứng dụng của cây nhị phân trong việc xây dựng mã Huffman.

10.1. Mã Huffman

Khác với mã ASCII dùng cùng một kích thước cho mã của các ký tự, mã Huffman là một cách mã hóa các ký tự với chiều dài mã khác nhau cho mỗi ký tự tùy theo tần suất xuất hiện của ký tự đó với mục đích tiết kiệm số bit mã hóa cho văn bản. Xét bài toán cụ thể sau:

10.1.1. Bài toán

Cho trước một tập các ký tự $\{C_1, C_2, ..., C_n\}$ và các trọng số $w_1, w_2, ..., w_n$ cho những ký tự này; w_i là trọng số của ký tự C_i và là một độ đo nào đó để chỉ ra ký tự này xuất hiện thường xuyên như thế nào trong các văn bản cần được mã hóa (chẳng hạn như tần suất xuất hiện trong một văn bản). Gọi $l_1, l_2, ..., l_n$ là chiều dài của các mã cho các ký tự $C_1, C_2, ..., C_n$ theo thứ tự đó. *Chiều dài chờ đợi* (expected length) của mã cho một cách mã hóa các ký tự là tổng $\sum_{i=1}^{n} w_i l_i$. Hãy xây dựng cách mã hóa các ký tự sao cho chiều dài chờ đợi là cực tiểu.

Ví dụ: Cho bảng 5 ký tự A, B, C, D, E và trọng số của chúng như sau:

ký tự	A	B	C	D	E
trọng số	0.2	0.1	0.1	0.15	0.45

Một cách mã hóa các ký tự là A (01), B(1000), C(1010), D(100), E(0). Khi đó chiều dài chời đợi của cách mã này là 2.1.

Một trong những tính chất có ích khác của một sơ đồ mã hóa là có thể *giải mã tức thì* (immediately decodable). Điều này có nghĩa là không có dãy bít nào trong biểu diễn của một ký tự là tiền tố của một dãy bít dài hơn biểu diễn một ký tự khác. Do đó khi nhận một dãy bít là mã số cho một ký tự, nó có thể được giải mã ngay lập tức, không cần phải xem dãy này có phải là một dãy con của dãy khác dài hơn biểu diễn một ký tự khác. Cách mã các ký tự như ví dụ trên không giải mã tức thì được vì mã của E là 0 lại là tiền tố của mã hóa ký tự A là 01. Một cách khác để mã hóa các ký tự của ví dụ này đảm bảo giải mã tức thì là: A(01), B(0000), C(0001), D(001), E(1).

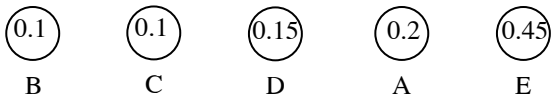
10.1.2. Thuật toán lập mã

Năm 1952 D.A. Huffman đề xuất một giải thuật xây dựng các sơ đồ mã hóa có thể giải mã tức thì và có chiều dài chờ đợi của mã cực tiểu.

Thuật toán lập mã Huffman:

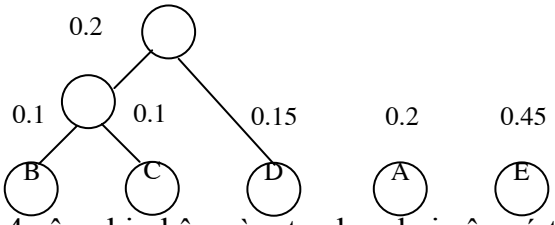
1. Khởi tạo một danh sách các cây nhị phân một nút chứa các trọng số w_1, w_2,..., w_n cho các ký tự C_1, C_2, ..., C_n.
2. Thực hiện n-1 lần các thao tác sau:
   a. Tìm 2 cây T' và T" trong danh sách với các nút gốc có trọng số cực tiểu w' và w".
   b. Thay thế hai cây T' và T" bằng một cây nhị phân với nút gốc có trọng số là w'+w", và có các cây con là T' và T".
3. Mã số của các ký tự C_i là dãy các bít được đánh dấu trên đường đi từ nút gốc của cây nhị phân cuối cùng đến nút lá C_i, mỗi khi đi sang trái thì nhận bit 0 và sang phải thì nhận bít 1.

Để minh họa thuật toán lập mã Huffman, ta xét bảng trọng số các ký tự như trong ví dụ trước. Ta bắt đầu bằng cách lập một danh sách các cây nhị phân một-

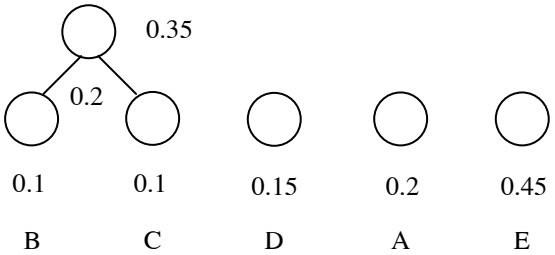


nút, cho mỗi ký tự:

Hai cây đầu tiên được chọn là các cây tương ứng với các ký hiệu B và C, vì chúng có trọng lượng nhỏ nhất, và chúng được kết hợp với nhau để tạo ra một cây mới có trọng số là 0.2, và có hai cây con là hai cây này.

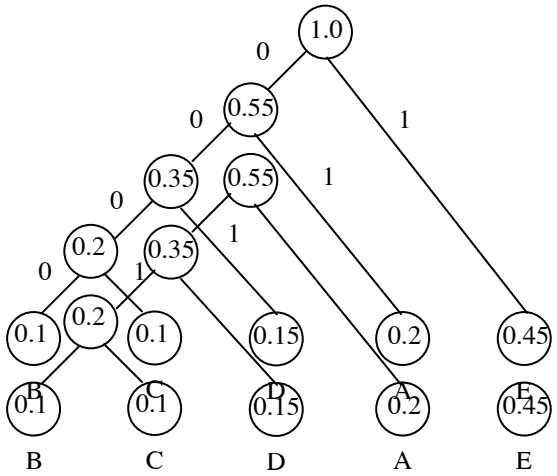


Từ danh sách 4 cây nhị phân này, ta chọn hai cây có trọng số cực tiểu là hai cây có trọng số là 0.15 và 0.2, rồi thay thế chúng bằng một cây khác có trọng số 0.35, và có hai cây này như là các cây con:

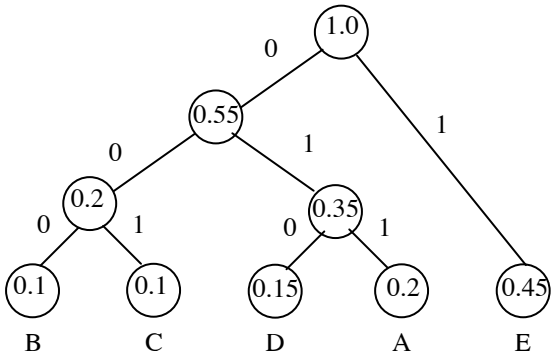


Tiếp tục, bước tiếp theo được danh sách các cây như sau:

Cuối cùng ta được cây nhị phân duy nhất và mã của các ký tự sẽ được xây dựng từ gốc đến các nút lá. Ta có được mã sau: B(0000), C(0001), D(001), A(01), E(1). Chiều dài chờ đợi của cách mã hoá cho các ký tự là 2.1.



Một cách khác ta cũng xây dựng được cây Huffman như sau:



Với cây Huffman này, mã của các ký tự sẽ là B(000), C(001), D(010), A(011), E(1). Chiều dài chời đợi của mã cho mỗi ký tự cũng đạt cực tiểu là 2.1.

Dễ thấy rằng các mã Huffman có tính chất giải mã tức thì. Mỗi ký tự tương ứng với một nút lá trong cây Huffman, và chỉ có duy nhất một đường từ nút gốc đến mỗi nút lá. Vì vậy, không có dãy bít nào vừa là mã số cho một ký tự vừa là tiền tố của một dãy bít dài hơn biểu diễn ký một tự khác.

10.1.3. Thuật toán giải mã

Thuật toán giải mã Huffman sau đây giúp cho việc giải mã tức thì các ký tự đã mã bằng mã Huffman.

Thuật toán giải mã Huffman:

1. Xuất phát con trỏ p tại gốc của cây Huffman mã hóa.
2. Lặp khi chưa hết nội dung cần giải mã, mỗi bước thực hiện:
    a. Đặt x là bít tiếp theo trong nội dung cần giải mã.
    b. Nếu x = 0 thì chuyển p sang nút con trái

Ngược lại chuyển p sang nút con phải.

c. Nếu p trỏ đến nút lá thì thực hiện:

i. Hiển thị ký tự tương ứng với nút lá.

ii. Đặt lại p ở gốc cây Huffman.

Để minh họa, giả sử rằng ta nhận được một thông báo dưới dạng dãy các bít: 0101011010 và biết rằng thông báo này được mã hóa bằng cây Huffman thứ hai ở ví dụ trên. Từ gốc của cây này đi theo các bít 010 ta đến được nút lá với ký tự nhận được là D và quay trở lại gốc.

0 1 0 1 0 1 1 0 1 0
D

Tiếp theo từ bit thứ tư ta đến được ký tự E.

0 1 0 1 0 1 1 0 1 0
D E

Tiếp tục, ta nhận được ký tự tiếp theo là A.

0 1 0 1 0 1 1 0 1 0
D E A

Cuối cùng ta nhận được ký tự D.

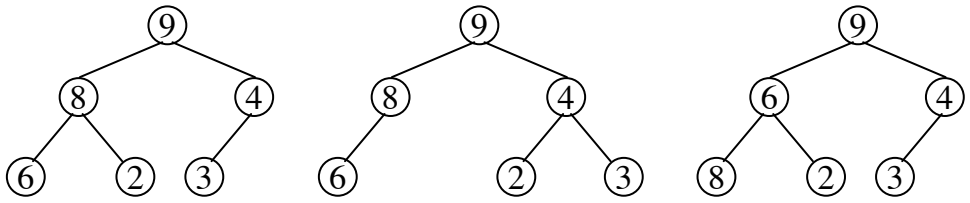
0 1 0 1 0 1 1 0 1 0
D E A D

10.2. Cấu trúc dữ liệu Heap

10.2.1. Khái niệm và tổ chức dữ liệu

Heap là một cây nhị phân đầy đủ và các giá trị khoá trong mỗi nút lớn hơn (hoặc nhỏ hơn) khoá của các nút con của nó.

Ví dụ, trong hình sau cây đầu tiên là một Heap, cây thứ hai không phải Heap vì không đầy đủ, và cây thứ ba đầy đủ nhưng không phải Heap vì không thoả điều kiện thứ tự của các nút.



Hình 3.21

Để cài đặt Heap, cũng như cây nhị phân ta có thể dùng liên kết các nút hoặc dùng mảng. Do đặc thù của Heap là cây nhị phânđầy đủ nên thuận lợi cho việc tổ

chức dữ liệu bằng mảng bằng cách đánh số các nút theo thứ tự từ trên xuống dưới và từ trái sang phải (xem phần 2.2).

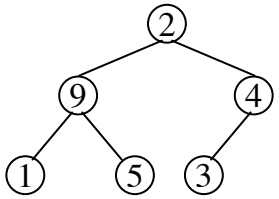
Khai báo

```
Const maxlength=...; {giới hạn số các nút của Heap}

Type
   ElementType = ...; {kiểu phần tử trong mỗi nút}

   HeapType =    Record
                        element:array[1..maxlength] of ElementType;
                        count:0..maxlength;
                 End;
Var
   Heap : HeapType;
```

Với cách tổ chức bằng mảng, việc tìm các nút con và nút cha của một nút được thực hiện dễ dàng. Cụ thể, với nút lưu ở vị trí thứ *i* của mảng sẽ có 2 nút con là 2*i* và 2*i*+1 và nút cha là *i* div 2.

10.2.2. Các thuật toán

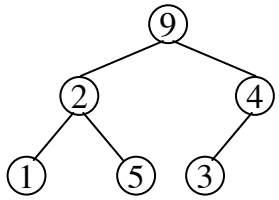
Một trong những thuật toán quan trọng trên Heap là chuyển cây nhị phân



đầy đủ thành Heap. Để hình thành thuật toán, ta xét trường hợp đơn giản là các cây con trái và phải của cây đã là những Heap nhưng bản thân cây không phải là Heap, ví dụ:

Hình 3.22

Ta thấy cây trên không phải là Heap nhưng các cây con trái và phảo đều là các Heap nên đầu tiên phải đổi giá trị của nút gốc với nút có khoá lớn nhất trong hai nút con của nó, trong hình trên là đổi với nút có khoá là 9 để được cây như hình sau:



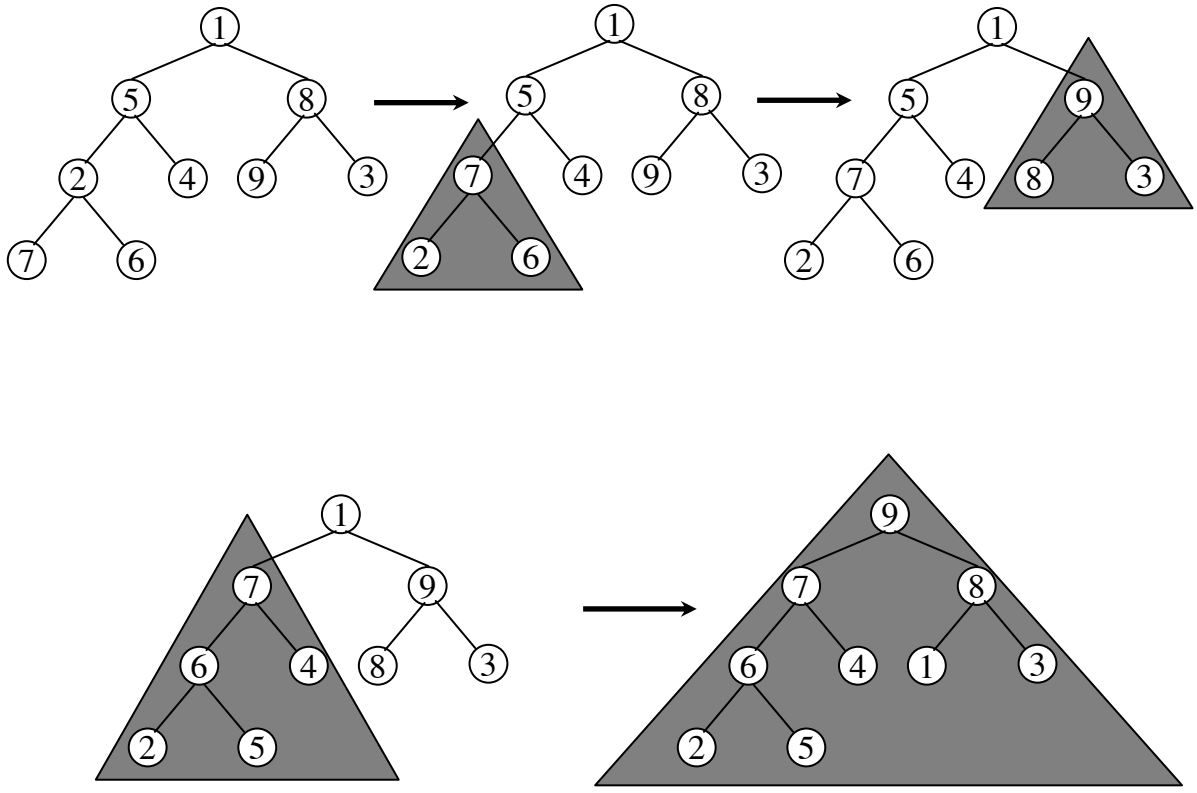
Hình 3.23

Sau khi đổi, giá trị khoá của nút gốc lớn hơn giá trị khoá tại hai nút con nhưng khi đó một cây con vẫn là Heap nhưng cây con kia có thể không còn là

Heap. Chẳng hạn trong hình trên cây con phải là Heap nhưng cây con trái không phải là Heap. Nếu trong trường hợp sau khi đổi các cây con của nút gốc vẫn là Heap thì kết thúc. Nếu có một cây con không phải là Heap thì lặp lại thao tác trên cho cây con này. Quá trình lặp lại cho đến khi cả hai cây con của nút đang xét là Heap.

Trong trường hợp tổng quát để chuyển cây nhị phân đầy đủ thành Heap, ta bắt đầu từ nút cuối cùng không phải là nút lá, áp dụng thao tác trên để chuyển cây con có gốc là nút này thành Heap. Sau đó chuyển sang nút trước và tiếp tục cho đến khi đạt đến nút gốc của cây.

Hình dưới đây minh hoạ chi tiết các bước quá trình xây dựng Heap từ một cây nhị phân đầy đủ. Các cây con chuyển thành Heap được đánh dấu.



Hình 3.24

Thuật toán SiftDown thực hiện việc chuyển một cây nhị phân đầy đủ tại các vị trí *r* .. *n* trong mảng Heap với các cây con trái và phải là các Heap thành một Heap. Thuật toán dùng biến Done kiểu logic để làm điều kiện dừng.

Thuật toán SiftDown

```
Done:=false; c:=2*r; {vị trí nút con}
While not Done and (c<=n) do
   {Tìm nút con có khoá lớn nhất}
   If (Heap.element[c].Key< Heap.element[c+1].Key) then c:=c+1;
```

{Đổi giá trị nút r với nút con lớn nhất rồi chuyển xuống cây con tiếo theo}

If (Heap.element[r].Key< Heap.element[c].Key) then

+ Đổi giá trị Heap.Element[r] với Heap.Element[c]

+ r:=c;

+ c:= c*2;

Else Done:=true;

Độ phức tạp tính toán: dễ thấy, tại mỗi bước của thuật toán số phần tử cần thao tác giảm một nửa so với bước trước. Do đó độ phức tạp của thuật toán trong trường hợp xấu nhất là $O(\log_2 n)$, với n là số phần tử của Heap.

Thủ tục SiftDown được cài đặt như sau:

```
Procedure SiftDown(var Heap:HeapType; r,n:Word);
var done:Boolean; c:Word; tmp:ElementType;
Begin
   Done:=false; c:=2*r;
   While not Done and (c<=n) do
      begin
         If(Heap.element[c].Key<Heap.element[c+1].Key)then
            c:=c+1;
         If (Heap.element[r].Key< Heap.element[c].Key) then
            begin
               tmp:=Heap.Element[r];
               Heap.Element[r]:=Heap.Element[c];
               Heap.Element[c]:=tmp;
             r:=c;
            c:= c*2;
            end
         Else
            Done:=true;
      end;
End;
```

Dùng thủ tục SiftDown ta có thể chuyển một cây nhị phân đầy đủ được lưu trong mảng Heap từ vị trí 1 đến vị trí n thành một Heap bằng thuật toán **CreateHeap** như sau:

Thuật toán CreateHeap:

For i:=n div 2 downto 1 do

SiftDown(Heap, i,n)

10.2.3. Hàng đợi ưu tiên

Tương tự như cấu trúc hàng đợi, mỗi phần tử trong hàng đợi ưu tiên được gắn với một thành phần là độ ưu tiên của phần tử trong hàng đợi. Độ ưu tiên này ảnh hưởng đến thao tác lấy một phần tử ra khỏi hàng đợi ưu tiên theo nghĩa phần

tử nào có độ ưu tiên cao nhất thì được lấy ra trước, trong các phần tử có cùng độ ưu tiên thì thực hiện theo nguyên tắc "vào trước, ra trước".

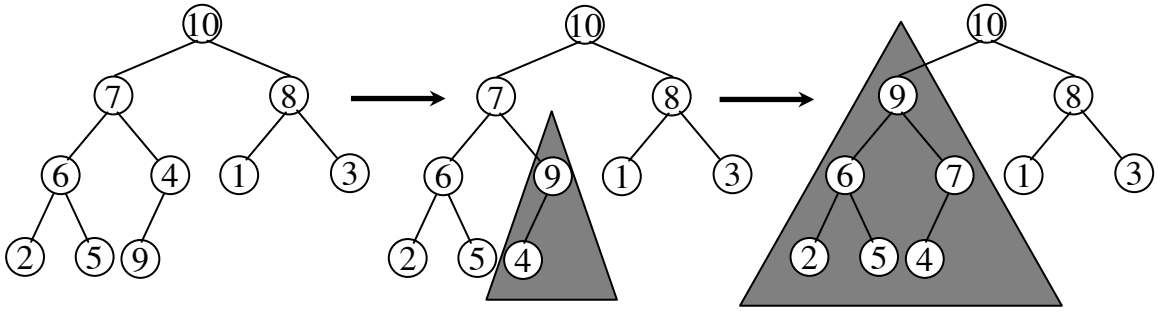
Cấu trúc Heap có hai ứng dụng quan trọng là hàng đợi ưu tiên và thuật toán sắp xếp HeapSort. Trong phần này trình bày các thao tác trên hàng đợi ưu tiên được tổ chức bằng Heap. Nếu hàng đợi ưu tiên được tổ chức bằng mảng như mô hình danh sách thì thao tác thêm vào và lấy ra sẽ có độ phức tạp là O(1) và O(n), tuỳ thuộc vào cách tổ chức, với n là số phần tử của hàng đợi. Với cách tổ chức hàng đợi bằng Heap thì thao tác thêm vào và lấy ra có độ phức tạp là O($\log_2 n$). Hàng đợi ưu tiên được tổ chức như một Heap với khoá của các phần tử là độ ưu tiên của nó trong hàng đợi. Với cách tổ chức như vậy, các thao tác thêm vào và lấy ra được cài đặt như sau:

Thêm một phần tử vào hàng đợi ưu tiên

Thao tác thêm một phần tử vào hàng đợi ưu tiên được tổ chức bằng cấu trúc Heap được thực hiện bằng cách thêm vào cuối Heap rồi dùng thủ tục SiftUp để điều chỉnh các phần tử thành Heap.

Thủ tục SiftUp được thực hiện ngược lại với thủ tục SiftDown được trình bày ở phần trên. Giả sử hàng đợi ưu tiên được lưu trên mảng từ vị trí 1 đến n-1, và là một Heap. Khi thêm phần tử vào vị trí n thì mảng các phần tử có thể không còn là một Heap. Để chuyển mảng n phần tử này thành Heap ta duyệt ngược từ cây con chứa phần tử cuối cùng vừa thêm, tiến hành điều chỉnh để cây con này trở thành một Heap. Nếu không cần điều chỉnh mà cây con đó đã là một Heap thì toàn bộ cây đã là một Heap, công việc hoàn thành. Nếu cây con chưa tạo thành Heap thì tiến hành đổi nút có khoá lớn hơn trong hai nút con cho nút gốc, sau đó tiến hành điều chỉnh cho cây con có nút gốc là nút cha của nút gốc của cây con vừa xét. Quá trình được lặp lại cho đến khi gặp một cây con đã là một Heap mà không cần điều chỉnh hoặc khi gặp nút gốc của cây.

Ví dụ hình dưới đây minh hoạ các bước điều chỉnh hàng đợi ưu tiên sau khi thêm phần tử có khoá 9.



Hình 3.25

Tương tự thủ tục SiftDown, thủ tục SiftUp được cài đặt như sau:

```
Procedure SiftUp(var Heap:HeapType; n : Word);
var done:Boolean; i,p:Word; tmp:ElementType;
```

```
Begin
   Done:=false; i:=n;
   While (i>1) and (not Done) do
      begin
         p:= i div 2;
         if Heap.element[p].key> Heap.element[i].key then
            begin
               tmp:= Heap.element[p];
               Heap.element[p]:= Heap.element[i];
               Heap.element[i]:=tmp;
            end
         else
            done:=true;
         i:=p;
      end;
End;
```

Từ thủ tục trên, thủ tục thêm một phần tử *x* vào hàng đợi ưu tiên *Q* kiểu PriQueue (là một Heap) được thực hiện như sau:

```
Procedure Insert(x:ElementType; var Q:PriQueue);
Begin
   if Q.count<maxlength then
      begin
         Q.count:=Q.count+1;
         Q.element[Q.count]:=x;
         SiftUp(Q, Q.count);
      end;
End;
```

Độ phức tạp thuật toán: dễ thấy độ phức tạp tính toán của thủ tục Insert chính là độ phức tạp cuả thủ tục SiftUp. Tương tự thủ tục SiftDown, thủ tục SiftUp có độ phức tạp là $O(\log_2 n)$, với *n* là số phần tử của hàng đợi ưu tiên.

Lấy một phần tử ra khỏi hàng đợi ưu tiên

Giả sử hàng đợi ưu tiên có *n* phần tử được lưu trên mảng theo kiểu Heap. Phần tử có độ ưu tiên cao nhất cần lấy ra chính là phần tử đầu tiên của mảng. Để xoá phần tử ra khỏi hàng đợi ưu tiên mà vẫn thoả mãn các tính chất của Heap ta thực hiện 2 thao tác:

- Chuyển phần tử cuối cùng của mảng (vị trí *n*) về đầu.

- Dùng thủ tục SiftDown để chuyển *n*-1 phần tử của mảng thành một Heap.

Thủ tục lấy một phần tử ra khỏi hàng đợi ưu tiên được cài đặt như sau:

```
Procedure ExtractMax(var x:ElementType; var Q:PriQueue);
var n:Word;
Begin
   n:=Q.count;
```

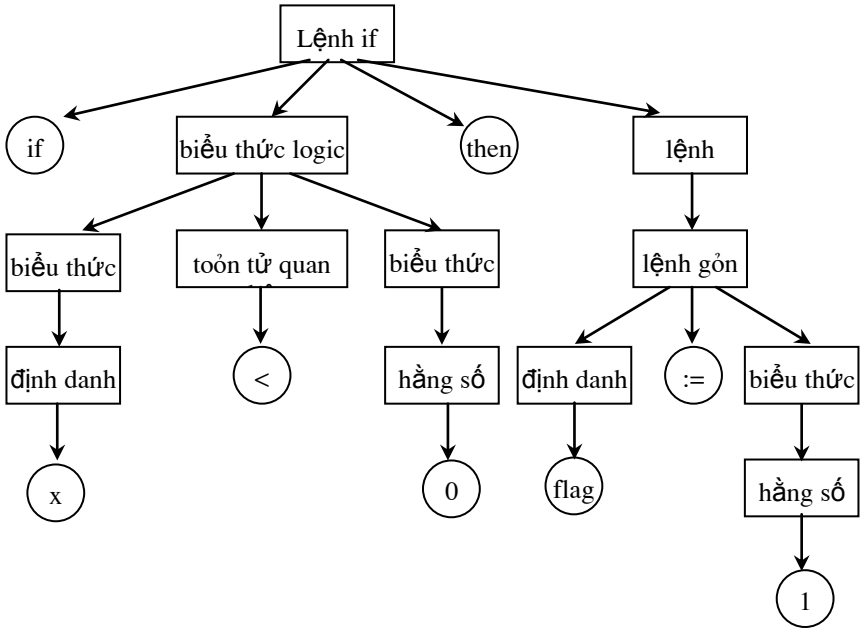
```
    x:=Q.element[1];
    Q.element[1]:= Q.element[n];
    Q.count:=Q.count-1;
    SiftDown(Q,1,n-1);
End;
```

Dễ thấy độ phức tạp tính toán của thủ tục ExtractMax là $O(\log_2 n)$.

11. CÂY TỔNG QUÁT

Trong phần trước chúng ta đã đề cập khá chi tiết các thao tác trên cây nhị phân. Trong thực tế cấu trúc cây nhị phân không đủ để mô tả các đối tượng dữ liệu. Chẳng hạn cây thư mục, cây gia phả của một dòng họ, cây quyết định, cây phân giải của chương trình dịch,...Trong phần này sẽ trình bày một số nội dung cơ bản của một cây tổng quát.

Hình dưới đây là cây tổng quát mô tả quá trình phân giải đoạn lệnh trong Pascal: if x < 0 then flag:=1;



Hình 3.26 Cây phân giải tạo bởi bộ dịch Pascal

11.1. Tổ chức dữ liệu

Một cách tự nhiên để biểu diễn cây tổng quát là mở rộng cách biểu diễn của cây nhị phân.

11.1.1. Dùng mảng các trường liên kết

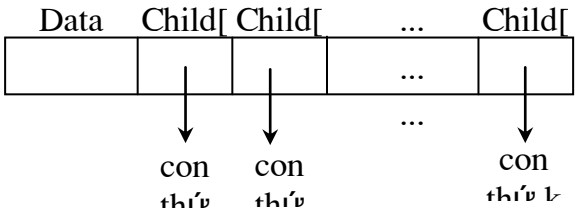
Một cách mở rộng cây nhị phân cho cây tổng quát là mỗi nút của cây tổng quát sẽ có nhiều liên kết đến các nút con của nó. Một nút gồm hai thành phần:

+ Thành phần Data: lưu dữ liệu của nút.

+ Thành phần Child: là một mảng các con trỏ đến các nút của cây tổng quát.

Khai báo:

```
const
        MaxChildren = ...; {số con tối đa của một nút}

type
        ElementType = ... ; {kiểu dữ liệu tại mỗi nút}

        TreePointer = ^TreeNode;

        TreeNode =  Record
                            Data:ElementType;
                            Child:Array[1..MaxChildren]of TreePointer;
                        End;
```

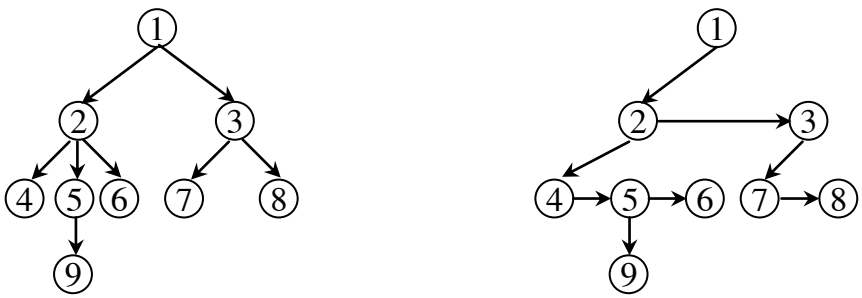


Hình 3.27 Một nút trong cây tổng quát

Một hạn chế trong cách biểu diễn dùng mảng các trường liên kết ở chỗ mỗi nút phải có một trường liên kết cho mỗi con, ngay cả khi đa số các nút không dùng tất cả các trường. Như vậy, phần bộ nhớ bị lãng phí không dùng có thể rất lớn. Để minh họa điều này, giả sử rằng một cây có n nút với nhiều nhất là 5 con cho mỗi nút. Với cách biểu diễn trên sẽ đòi hỏi *n* nút, mỗi nút chứa 5 trường liên kết, tổng cộng là 5*n* trường. Tuy nhiên, trong một cây như thế chỉ có *n*-1 liên kết được sử dụng và như vậy chỉ có *n*-1 trong 5*n* trường liên kết là được sử dụng. Như vậy tỷ lệ số các trường không dùng là (4*n*+1)/5*n*, xấp xỉ 80% các trường là rỗng.

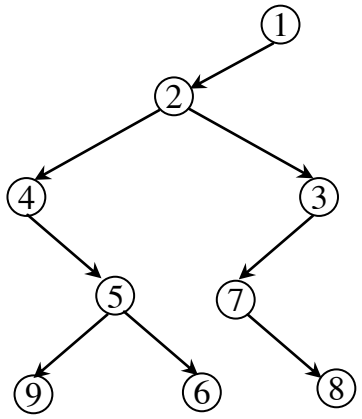
11.1.2. Dùng liên kết các nút anh em

Một cách tổ chức dữ liệu tốt hơn cho các cây tổng quát là dùng các nút chỉ chứa hai trường, trong đó một trường dùng để liên kết với một nút con đầu tiên của nó và một trường để liên kết các nút anh em (các nút có cùng nút cha) với nhau theo thứ tự xuất hiện của chúng trong cây, từ trái sang phải. Ví dụ cây tổng quát sau có thể biểu diễn bằng liên kết các nút anh em như sau.



Hình 3.28 Biểu diễn cây tổng quát bằng liên kết các nút anh em

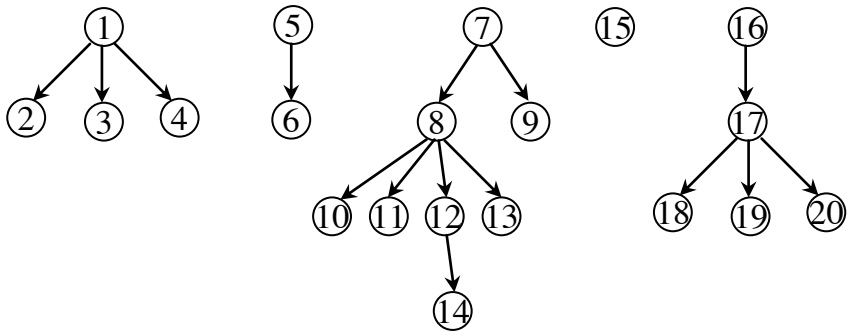
Như vậy ta có thể xem một cây tổng quát được biểu diễn như một cây tựa cây nhị phân. Ví dụ cây liên kết các nút anh em như hình trên có thể vẽ lại như sau:



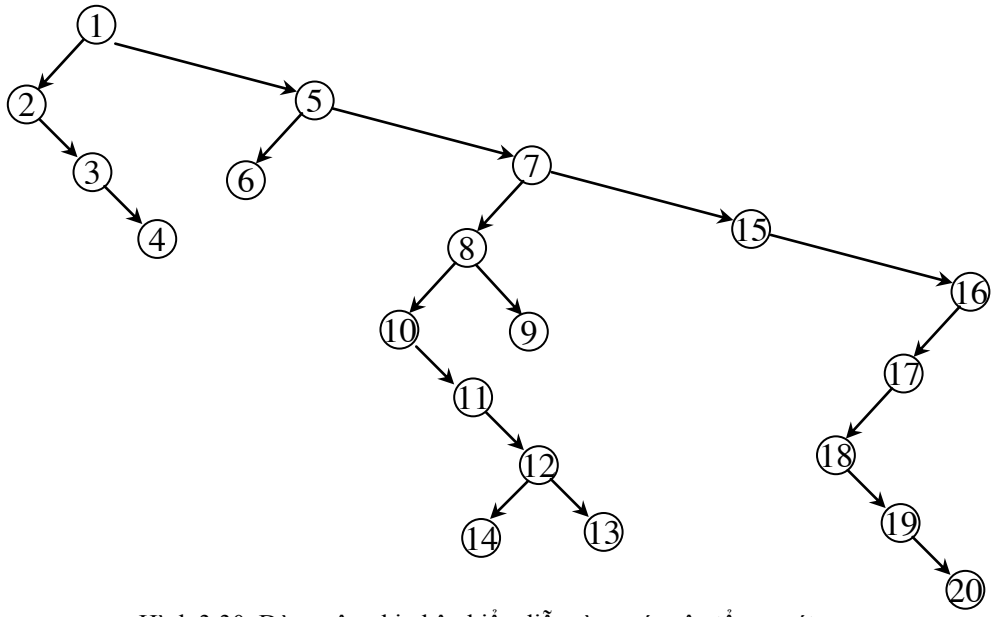
Hình 3.29 Biểu diễn cây tổng quát bằng liên kết các nút anh em

Trong cây nhị phân này, nút *x* là con bên trái của *y* nếu *x* là nút con bên trái nhất (con trưởng) của *y* trong cây, và *x* là con phải của *y* nếu *x* và *y* là anh em trong cây ban đầu.

Khi dùng cây nhị phân theo cách này để biểu diễn cây tổng quát, con trỏ bên phải ở nút gốc sẽ luôn luôn là rỗng vì nút gốc sẽ không bao giờ có con bên phải. Điều này cho phép chúng ta dùng một cây nhị phân để biểu diễn không chỉ một cây đơn giản mà cả tập hợp các cây hay rừng (forest). Ta chỉ cần đặt con trỏ bên phải chỉ đến nút gốc của cây nhị phân tiếp theo trong rừng. Ví dụ rừng dưới đây:



Được biểu diễn bằng cây nhị phân:



Hình 3.30. Dùng cây nhị phân biểu diễn rừng các cây tổng quát

Một điều tiện lợi của cách biểu diễn các cây tổng quát này là chúng ta đã nghiên cứu một cách chi tiết các cây nhị phân và đã phát triển những thuật toán để xử lý chúng. Các thuật toán này có thể được dùng để xử lý các cây tổng quát sẽ được mô tả trong các bài tập; đặc biệt là cách duyệt cây tổng quát và rừng các cây.

Khai báo cây nhị phân biểu diễn cây tổng quát như sau:

```
Type
    TreePointer = ^TreeNode;

    TreeNode = Record
                    Data : ElementType;
                    EldestChild: TreePointer;
                    NextSibling: TreePointer;
               End;
```

Một cách khác để biểu diễn các cây tổng quát là dùng đồ thị có hướng, trong đó các cây được xem như trường hợp đặc biệt của đồ thị. Mô hình đồ thị sẽ được trình bày trong chương sau.

11.2. Các thao tác trên cây tổng quát

Trong phần này trình bày một số thao tác cơ bản trên cây tổng quát được biểu diễn bằng liên kết các nút anh em.

11.2.1. Tìm con trưởng của mỗi nút

```
Function EldestChild(p : TreePointer) : TreePointer;
 Begin
    if p<>nil then
       EldestChild:=p^.EldestChild
    else
       EldestChild:=nil;
```

```
End;
```

11.2.2. Tìm em liền kề của mỗi nút

```
Function NextSibling(p : TreePointer) : TreePointer;
 Begin
    if p<>nil then
       NextSibling:=p^.NextSibling
    else
       NextSibling:=nil;
 End;
```

11.2.3. Duyệt cây tổng quát

Tương tự trên cây nhị phân, trên cây tổng quát cũng dùng các cách duyệt cơ bản: duyệt theo thứ tự trước, duyệt theo thứ tự giữa và duyệt theo thứ tự sau.

Duyệt theo thứ tự trước:

Nếu cây khác rỗng thì thực hiện

1. Thăm nút gốc.
2. Với mỗi cây con T_i của nút gốc, duyệt theo thứ tự trước cây T_i.

Ví dụ cây ở hình 5.1.2 duyệt theo thứ tự trước sẽ lần lượt đi qua các đỉnh: 1, 2, 4, 5, 9, 6, 3, 7, 8.

Thủ tục duyệt theo thứ tự trước trên cây tổng quát như sau.

```
 Procedure TraversePreOrder(root:TreePointer);
 var p:TreePointer;
    Begin
       if root<>nil then
          begin
             Visit(root);
             p:=EldestChild(root);
             while p<>nil do
                begin
                   TraversePreOrder(p);
                   p:=EldestChild(p);
                end;
          end;
    End;
```

Thủ tục duyệt theo thứ tự trước còn gọi là duyệt theo chiều sâu bởi vì khi đang ở một nút thì đi sâu xuống các nút con ở mức sâu hơn đến khi gặp nút lá.

Duyệt theo thứ tự giữa:

Nếu cây khác rỗng thì thực hiện

1. Duyệt theo thứ tự giữa cây con đầu tiên T_1 của cây.
2. Thăm nút gốc.

3. Với các cây con T_2, ..., T_k của cây, duyệt theo thứ tự giữa các cây này.

Ví dụ cây ở hình 5.1.2 duyệt theo thứ tự giữa sẽ lần lượt đi qua các đỉnh: 4, 2, 9, 5, 6, 1, 7, 3, 8.

Thủ tục duyệt theo thứ tự giữa trên cây tổng quát như sau.

```
Procedure TraverseInOrder(root:TreePointer);
var p:TreePointer;
   Begin
      if root<>nil then
         begin
            p:=EldestChild(root);
            if p<>nil then
               begin
                  TraverseInOrder(p);
                  p:=EldestChild(p);
               end;
            Visit(root);
            while p<>nil do
               begin
                  TraverseInOrder(p);
                  p:=EldestChild(p);
               end;
         end;
   End;
```

Duyệt theo thứ tự sau:

Nếu cây khác rỗng thì thực hiện

1. Với các cây con T_1, ..., T_k của cây, duyệt theo thứ tự sau các cây này.
2. Thăm nút gốc.

Ví dụ cây ở hình 5.1.2 duyệt theo thứ tự giữa sẽ lần lượt đi qua các đỉnh: 4, 9, 5, 6, 2, 7,8, 3, 1.

Thủ tục duyệt theo thứ tự sau trên cây tổng quát như sau.

```
Procedure TraversePostOrder(root:TreePointer);
var p:TreePointer;
   Begin
      if root<>nil then
         begin
            p:=EldestChild(root);
            while p<>nil do
               begin
                  TraversePostOrder(p);
                  p:=EldestChild(p);
               end;
            Visit(root);
```

```
            end;
        End;
```

Duyệt theo chiều rộng:

Duyệt cây theo chiều rộng được thực hiện bằng cách, tại mỗi nút sau khi thăm thì đi theo các nút anh em liền kề với nó, sau khi hết các nút anh em thì đi xuống mức sau thăm đỉnh ngoài cùng bên trái, rồi tiếp tục đi theo các nút anh em liền kề. Phương pháp duyệt theo chiều rộng thực hiện duyệt các nút ưu tiên theo cấp của chúng, mỗi khi duyệt hết các nút cùng cấp thì chuyển sang các nút ở cấp tiếp theo. Ví dụ cây ở hình 5.1.2 duyệt theo thứ tự giữa sẽ lần lượt đi qua các đỉnh:1, 2, 3, 4, 5, 6, 7, 8, 9.

Thủ tục duyệt theo chiều rộng dùng một hàng đợi Q, mỗi phần tử là một nút của cây tổng quát. Hàng đợi này dùng để lưu các nút con trưởng của mỗi đỉnh để sau đó lấy ra và thăm các nút liền kề với nó.

```
Procedure BreadthTraverse(root:TreePointer);
   var p:TreePointer;
       q:QueueOfTreePointer;
    Begin
       Init(Q);
       Visit(root);
       AddQueue(Q, root);
       while not empty(Q) do
          begin
             RemoveQueue(Q, p);
             p:=EldestChild(p);
             while p<>nil do
                begin
                   Visit(p);
                   AddQueue(Q, p);
                   p:=NextSibling(p);
                end;
          end;
    End;
```

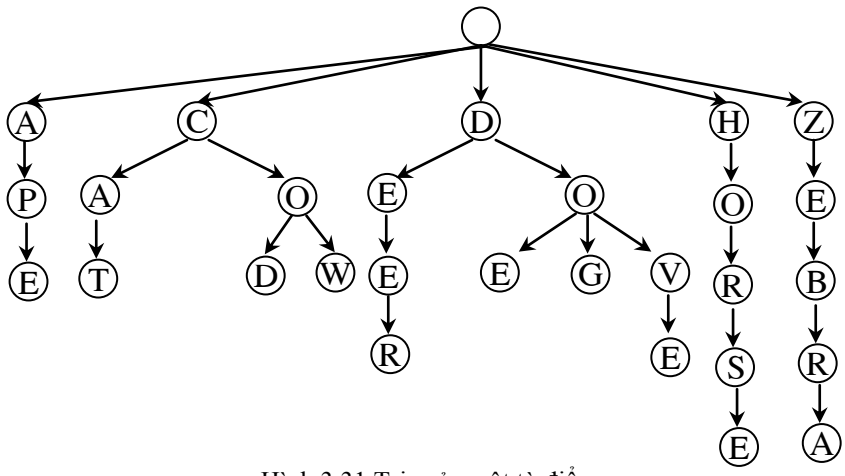
11.3. Cây tìm kiếm tổng quát

Kỹ thuật tìm kiếm dùng cho cây tìm kiếm nhị phân cũng được mở rộng cho cây tổng quát. Một phương pháp tổ chức cây tìm kiếm tổng quát được gọi là trie (lấy theo từ retrieval, nhưng phát âm như try). Nếu như trong cây tìm kiếm nhị phân khi tìm một nút có khóa cho trước nếu khóa cần tìm nhỏ hơn khóa nút gốc thì chuyển sang tìm ở cây con trái; nếu khác tìm trong cây con phải, và đó là hai cách lựa chọn duy nhất. Tuy nhiên, với trie là một cây tổng quát trong đó đường tìm từ mỗi nút có thể có nhiều hướng hơn, và nói chung nếu ta tổ chức tốt thì đường tìm ngắn hơn bởi vì trong cây sẽ có ít mức hơn.

Giả sử dữ liệu tại mỗi nút của cây tổng quát có trường Key là khóa của các nút, trên trường Key ta có thể so sánh các giá trị có thể của nó. Cây tìm kiếm tổng quát được tổ chức như cây tổng quát, trong đó các nút con của một nút được quy định một trật tự tăng hay giảm theo khóa của nút theo chiều từ trái sang phải.

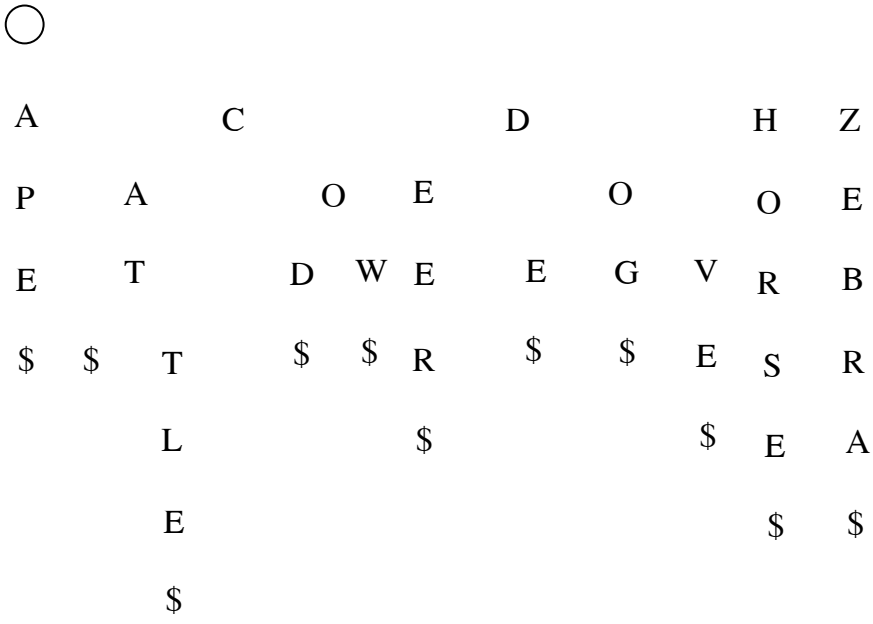
Để minh họa, giả sử chúng ta muốn lưu trữ một nhóm các từ được dùng như một từ điển trong chương trình kiểm tra từ. Khi chương trình này kiểm tra một tài liệu, nó phải đọc mỗi từ rồi tìm nó trong từ điển, nếu không tìm thấy thì báo lỗi chính tả. Vì vậy, với mỗi văn bản ta cần tìm trong từ điển nhiều lần, việc tìm kiếm có hiệu quả là quyết định. Như vậy một cách tự nhiên, chúng ta tổ chức từ điển như một trie.

Mỗi nút trong trie biểu diễn từ điển sẽ lưu trữ một ký tự. Các nút ở mức thứ nhất lưu trữ tất cả các ký tự đầu tiên của các từ trong từ điển, các nút ở mức thứ hai lưu trữ tất cả những ký tự thứ hai của các từ trong từ điển,...Một minh họa đơn giản, xét một từ điển gồm các từ sau đây: APE, CAT, COD, COW, DEER, DOE, DOG, DOVE, HORSE, và ZEBRA. Các từ này được lưu trong trie như hình sau đây:



Hình 3.31 Trie của một từ điển

Để ý rằng thời gian tìm mỗi từ tỷ lệ với độ dài của từ, trong thực tế các từ ngắn xuất hiện thường xuyên hơn các từ dài nên cách tổ chức này chương trình kiểm tra từ sẽ thực hiện tốt. Nếu để ý cẩn thận hơn thì ta sẽ thấy cách tổ chức trên gặp một trở ngại trong trường hợp các từ là tiền tố (phần đầu) của một từ khác trong từ điển. Ví dụ, cần thêm từ CATTLE vào từ điển trên. Nếu ta thêm ba nút chứa ba ký tự T, L, và E vào con đường chứa các nút cho từ CAT, chúng ta không có cách nào để kiểm tra từ CAT. Một cách khắc phục trở ngại trên là thêm vào một ký tự đặc biệt (chẳng hạn là $) để biết vị trí kết thúc một từ. Với ký tự đặc biệt này thì trên trie các từ sẽ không có từ nào là tiền tố của một từ dài hơn, và do đó việc kiểm tra từ luôn thực hiện được. Với trie trên, sau khi thêm vào từ CATTLE và các ký hiệu đặc biệt sẽ có hình ảnh như sau.



Hình 3.32 Trie của một từ điển với các ký tự đặc biệt

Các cây tìm kiếm nhị phân và trie là những cấu trúc dữ liệu được dùng trong các sơ đồ tìm kiếm nội trú, nghĩa là nhóm dữ liệu cần tìm phải đủ nhỏ để cho nó có thể lưu trữ chúng vào bộ nhớ chính. Với các dữ liệu lưu trữ ở bộ nhớ ngoài, để các thao tác nhanh chóng phải tổ chức cây theo cách khác gọi là B-tree. Cấu trúc này sẽ được trình bày trong chương sau.

12. BÀI TẬP

Bài 1. Tổ chức dữ liệu kiểu cây nhị phân để lưu cây biểu thức số học với các số có 1 chữ số và các phép toán 2 ngôi.

+ Viết chương trình tạo cây biểu thức (2+5)*3-(7-2)/3.

+ Viết thủ tục in cây biểu thức theo dạng trung tố.

+ Viết thủ tục in cây biểu thức theo dạng hậu tố.

+ Viết hàm tính giá trị của biểu thức bằng đệ quy và không đệ quy.

Bài 2. Viết chương trình tính giá trị biểu thức cho bởi một cây biểu thức bằng hai cách: đệ quy và không đệ qui.

Bài 3. a) Cho biết kết quả duyệt theo thứ tự trước của một cây nhị phân là: A D F G H K L P Q R W Z, và kết quả duyệt theo thứ tự giữa là: G F H K D L A W R Q P Z. Hãy xây dựng cây nhị phân. Tổng quát: hãy đưa ra thuật toán xây dựng cây từ kết quả duyệt theo thứ tự trước và thứ tự giữa.

b) Hãychỉ ra bằng một ví dụ rằng nếu biết kết quả duyệt theo thứ tự trước và thứ tự sau không xác định một cách duy nhất cây nhị phân. (Hãy chỉ một ví dụ của hai cây nhị phân khác nhau mà kết quả duyệt theo thứ tự trước và thứ tự sau trùng nhau).

Bài 4. Viết các hàm tính :

+ Số nút của một cây nhị phân.

+ Số nút lá của một cây nhị phân.

+ Chiều cao cây nhị phân.

+ Hàm kiểm tra một cây nhị phân có phải là cây cân bằng hoàn toàn không.

+ Hàm kiểm tra một cây nhị phân có phải là cây cân bằng không.

Bài 5. Viết hàm kiểm tra hai cây tìm kiếm nhị phân có giống nhau không.

Bài 6. Trình bày thuật toán kiểm tra một cây có phải là cây con của một cây tìm kiếm nhị phân cho trước.

Bài 7. Trình bày thuật toán xóa và thu hồi ô nhớ của tất các nút của một cây nhị phân được tổ chức bằng liên kết. Cài đặt trên một cây nhị phân mỗi nút chứa một số nguyên.

Bài 8. Để sắp xếp một danh sách ta có thể thực hiện bằng cách dùng một cấu trúc cây tìm kiếm nhị phân trung gian đưa vào cây này các phần tử của danh sách, sau đó lấy từ cây đưa vào danh sách theo thứ tự tăng (giảm) của khóa. Hãy tổ chức dữ liệu cho trường hợp tổng quát và viết các thủ tục cần thiết. Hãy cài đặt các thao tác trên cho dãy số.

Bài 9. Dùng mã Huffman của ví dụ phần 4.2., hãy giải mã các dãy bít sau:

a) 000001001

b) 001101001

c) 000101001

d) 000010011001

Bài 10. Cho bảng các ký tự và trọng số của chúng như sau:

Ký tự	Trọng số
a	0.20
b	0.10
c	0.08
d	0.08
e	0.40
f	0.05
g	0.05
h	0.04

a) Hãy lập mã Huffman cho bộ ký tự trên.

b) Dùng mã Huffman vừa lập, hãy mã hóa thông báo sau "feed a deaf aged hag".

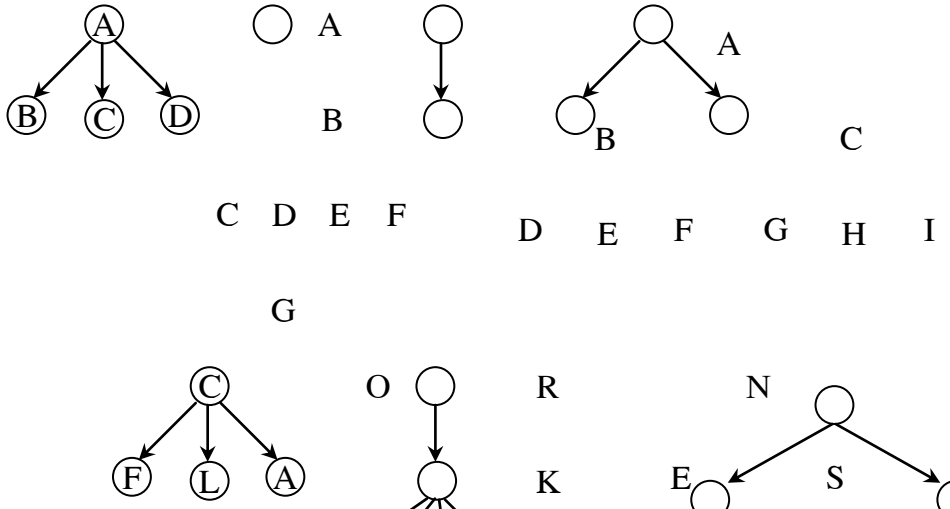
Bài 11. Viết thủ tục đọc một bảng các ký tự và trọng lượng của chúng rồi lập mã Huffman cho các ký tự này.

Bài 12. Với một cây Huffman mã hóa cho trước, hãy viết một thủ tục mã hóa một thông báo, và một thủ tục giải mã một dãy các bit đã mã hóa.

Bài 13****.** Hãy lập chương trình mã hóa và giải mã file văn bản bằng mã Huffman.

Bài 14. Vẽ một trie lưu trữ tập các từ sau đây: A, AM, AN, AND, BAT, BE, BEEN, BET, BEFORE, BEHIND, BIT, TITLE, BUT, BY, CAT, COT, CUT.

Bài 15. Hãy biểu diễn các cây tổng quát hay các rừng dưới đây bằng các cây dạng nhị phân:



Bài 16***.** Tổ chức dữ liệu cho một Trie các từ trong một từ điển. Trình bày thuật toán và viết các thủ tục thực hiện:

a) Thêm một từ vào trie.

b) Đọc các từ từ một tệp text và đưa vào trie.

c) Tìm một từ trong trie.

d) Kiểm tra các từ trong một văn bản có nằm trong từ điển không. Các từ không có trong từ điển được ghi ra tệp văn bản.

e) In các từ theo thứ tự tăng.

f) Ghi các từ trong Trie vào một tệp văn bản với thứ tự tăng.

Bài 17****.** Một từ điển Anh-Việt được tổ chức gồm 2 phần. Phần dữ liệu được lưu trong tệp có cấu trúc gồm: từ tiếng Anh, Loại từ, nghĩa tiếng Việt. Các từ này không cần lưu theo thứ tự. Phần nạp vào bộ nhớ trong là một trie với bổ sung cuối mỗi từ (trước ký hiệu kết thúc) là các ký tự xác định vị trí của từ đó trong tệp dữ liệu của từ điển. Hãy lập chương trình cho phép tạo lập tệp dữ liệu từ điển, đọc dữ liệu từ tệp từ điển vào trie, tra từ trên từ điển bằng cách tìm trong trie để biết vị trí của từ trong tập dữ liệu, rồi đọc tệp dữ liệu để có các thông tin về từ cần tra.

Chương 4

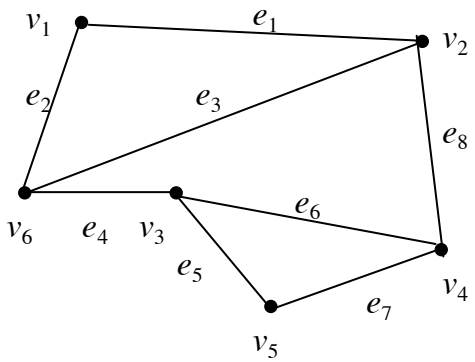
ĐỒ THỊ

13. CÁC KHÁI NIỆM

13.1. Khái niệm đồ thị (Graph)

Đồ thị là một bộ $G = (V, E)$, trong đó V là một tập hữu hạn các phần tử, còn gọi là các *đỉnh* (vertices), E là cặp các phần tử của V, được gọi là các *cạnh* hoặc *cung* (edges). Tập V gọi là tập đỉnh của đồ thị, tập E gọi là tập các cạnh (cung) của đồ thị.

Ví dụ: Đồ thị $G = (V, E)$, với $V=\{v_1, v_2,...,v_6\}$, $E=\{e_1, e_2,...,e_8\}$ được thể hiện bằng hình sau.



Hình 4.1 Hình ảnh một đồ thị

Đồ thị $G=(V, E)$ được gọi là *đồ thị vô hướng* nếu cạnh (u,v) thuộc E thì cạnh (v,u) cũng thuộc E. Khi đó ta đồng nhất hai cạnh (u,v) và (v,u) và gọi là cạnh nối hai đỉnh u, v.

Đồ thị không phải là đồ thị vô hướng gọi là *đồ thị có hướng*. Trong một đồ thị có hướng, mỗi cạnh (u, v) xác định đỉnh đầu u và đỉnh cuối v. Do đó cạnh (u,v) và cạnh (v,u) là khác nhau. Một cạnh trong đồ thị có hướng gọi là cung.

Mô hình đồ thị là một mô hình có ý nghĩa rất lớn trong thực tế, bằng mô hình này cho phép mô tả và giải quyết nhiều bài toán thực tế. Mô hình đồ thị thường gặp trong thực tế, chẳng hạn sơ đồ mạng lưới giao thông, mỗi đỉnh của đồ thị là một nút giao thông, mỗi cạnh là một đường đi nối giữa hai nút giao thông. Đồ thị dùng biểu diễn sơ đồ của một hệ thống mạng, trong đó mỗi máy tính là một đỉnh và đường cáp nối giữa hai máy.

Đồ thị có trọng số là một đồ thị mà mỗi cạnh gắn với một số, gọi là trọng số của cạnh.

Trọng số của đồ thị có ý nghĩa tùy thuộc đối tượng của nó mô tả, chẳng hạn trọng số của một cạnh trong đồ thị mạng lưới giao thông là chiều dài của đường

nối giữa hai nút giao thông, hoặc chi phí vận chuyển một đơn vị hàng hóa trên đường này,...

Cho một đồ thị $G=(V, E)$, x, y là hai đỉnh của đồ thị. Đỉnh y được gọi là *đỉnh kề* với đỉnh x nếu và chỉ nếu (x, y) là một cạnh của đồ thị. Trong đồ thị vô hướng, nếu y kề với x thì x cũng kề với y.

Đường đi giữa hai đỉnh x và y của đồ thị là một dãy các đỉnh $x_1, x_2, ..., x_k$ sao cho $x_1=x$, $x_k = y$, và với mỗi $i=1,..,k-1$, x_{i+1} kề với đỉnh x_i.

Một đường đi mà đỉnh đầu trùng với đỉnh cuối gọi là một *chu trình* của đồ thị.

Đồ thị liên thông là một đồ thị mà giữa hai đỉnh bất kỳ đều có đường đi giữa hai đỉnh.

14. TỔ CHỨC DỮ LIỆU BIỂU DIỄN ĐỒ THỊ

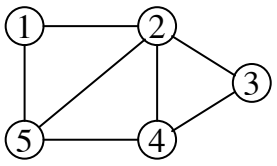
Để biểu diễn đồ thị trên máy tính thông thường có các cách tổ chức sau.

14.1. Biểu diễn đồ thị bằng ma trận kề (adjacency matrice)

Giả sử G là một đồ thị gồm n đỉnh, không mất tính tổng quát,giả sử các đỉnh này được đánh số là 1, 2, ..., n. Để biểu diễn đồ thị ta dùng một mảng hai chiều các giá trị 0,1. Nếu a là mảng hai chiều biểu diễn đồ thị thì các phần tử của mảng được xác định như sau:

$$a_{ij} = \begin{cases} 1, & \text{nếu cạnh (cung) (i, j) thuộc Đồ thị} \\ 0, & \text{trong tr- ờng hợp ng- ợc lại} \end{cases}$$

Cho đồ thị như hình vẽ, ma trận kề biểu diễn đồ thị được cho bởi bảng bên cạnh.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Hình 4.2 Biểu diễn đồ thị bằng ma trận kề

Với đồ thị vô hướng ma trận kề biểu diễn đồ thị là ma trận đối xứng. Với đồ thị có trọng số thì giá trị mỗi phần tử của ma trận kề là trọng số của cạnh (cung) tương ứng. Cụ thể, ma trận kề của đồ thị có trọng số được xác định như sau:

$$a_{ij} = \begin{cases} c_{ij}, & \text{nếu cạnh (cung) (i, j) thuộc Đồ thị} \\ 0, & \text{trong tr- ờng hợp ng- ợc lại} \end{cases}$$

Với c_{ij} là trọng số của cung (i,j).

Khai báo ma trận kề biểu diễn đồ thị như sau:

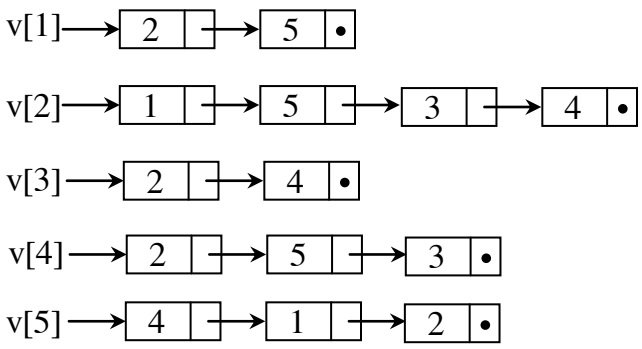
```
const MaxVertices = .... ; {số đỉnh tối đa của đồ thị}

type
   ElementType = .... ; {kiểu phần tử của ma trận kề}
   Graph = Record
                  Vnum:1..MaxVertices;
                  Adj:Array[1..MaxVertices,1..MaxVertices]of
                                                          ElementType;
               End;
Var G : Graph;
```

Với cách biểu diễn đồ thị bằng ma trận kề có ưu điểm là thao tác kiểm tra đỉnh *y* có kề với đỉnh *x* không được thực hiện đơn giản bằng cách kiểm tra Adj[*x*,*y*] nếu bằng 0 thì *y* không kề *x* và ngược lại. Tuy nhiên, cách biểu diễn đồ thị bằng ma trận kề lãng phí bộ nhớ khi số cạnh của đồ thị ít và trong trường hợp đồ thị vô hướng.

14.2. Biểu diễn đồ thị bằng danh sách kề (adjacency list)

Một cách khắc phục lãng phí của ma trận kề là dùng danh sách kề. Giả sử đồ thị có n đỉnh, để biểu diễn đồ thị ta dùng n danh sách, danh sách thứ i lưu tất cả những đỉnh kề với đỉnh i (cả trọng số nếu là đồ thị có trọng số). Để tiết kiệm số ô nhớ danh sách các đỉnh kề của một đỉnh được biểu diễn bằng danh sách liên kết.

Đồ thị như trong hình 4.2 được biểu diễn bằng danh sách kề như sau:



Hình 4.3 Biểu diễn đồ thị bằng danh sách kề

Với đồ thị có trọng số thì tại mỗi phần tử của danh sách kề, ngoài việc lưu đỉnh kề thì bổ sung thêm thành phần lưu trọng số của cạnh (cung) tương ứng.

Đồ thị biểu diễn bằng danh sách kề có thể khai báo như sau:

```
const
   MaxVertices = ....; {số đỉnh tối đa của đồ thị}

type
   AdjPointer = ^VertexNode;
   VertexNode = Record
```

```
                    Vertex : 1..MaxVerties;
                    cost : Real; {trọng số của cạnh}
                    Link : AdjPointer;
                End;
Graph =     Record
                Vnum:1..MaxVerties;
                adj: Array[1..MaxVerties] of VertexNode;
            End;

Var
   G : Graph;
```

Biểu diễn đồ thị bằng danh sách kề khắc phục được sự lãng phí của ma trận kề nhưng thao tác kiểm tra tính kề nhau của đỉnh *y* với đỉnh *x* phải thực hiện thao tác duyệt trong danh sách các đỉnh kề của đỉnh *x*, nếu *y* nằm trong danh sách này thì *y* kề với *x*.

14.3. Biểu diễn đồ thị bằng danh sách cạnh (cung)

Một cách khác để biểu diễn đồ thị có ít cạnh là dùng danh sách cạnh (cung). Trong cách biểu diễn này ta dùng một danh sách để lưu các cạnh của đồ thị, mỗi cạnh bao gồm đỉnh đầu, đỉnh cuối và trọng số của cạnh (nếu có).

Đồ thị như trong hình 4.2 được biểu diễn bằng danh sách cạnh như sau:

	Đỉnh đầu	Đỉnh cuối
1	1	2
2	1	5
3	2	3
4	2	4
5	2	5
6	3	4
7	4	5

Khai báo:

```
const
   MaxVerties= ....; {số đỉnh tối đa của đồ thị}
   MaxEdges = ....; {số cạnh tối đa của đồ thị}

type
   Edge =   Record
                FirstVertex, LastVertex : 1..MaxVerties;
                cost : Real; {trọng số của cạnh}
            End;
   Graph = Record
                Vnum:1..MaxVerties;
```

```
    Enum:1..MaxEdges;
    Edges: Array[1..MaxEdges] of Edge;
End;

Var
  G : Graph;
```

Biểu diễn đồ thị bằng danh sách cung thường được dùng trong những đồ thị mà số cạnh ít. Cũng như cách biểu diễn đồ thị bằng danh sách kề, thao tác kiểm tra tính kề nhau của đỉnh *y* với đỉnh *x* phải thực hiện thao tác duyệt trong danh sách tất cả các cạnh của đồ thị, nếu có cạnh thứ *i* nào đó mà đỉnh đầu là *x* và đỉnh cuối là *y* thì *y* kề với *x*.

15. DUYỆT ĐỒ THỊ

Duyệt đồ thị là đi theo các cạnh bằng một phương pháp nào đó để "thăm" tất cả các đỉnh của đồ thị. Duyệt đồ thị là một thao tác cơ bản trên đồ thị, nhiều thuật toán trên đồ thị trực tiếp hoặc gián tiếp dùng thao tác này như tìm đường đi, xét tính liên thông của đồ thị, tìm chu trình của đồ thị,... Có hai cách duyệt đồ thị cơ bản là duyệt theo chiều sâu (Depth-first search - DFS) và duyệt theo chiều rộng (Breadth-first search - BFS).

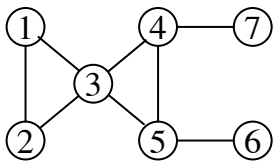
15.1. Duyệt theo chiều sâu

Phương pháp duyệt theo chiều sâu là tìm cách đi sâu theo các đỉnh của đồ thị khi còn chưa thăm và còn có thể đi được. Điều này có nghĩa, nếu ta đang thăm đỉnh *v*, ta sẽ tìm một đỉnh *u* kề với đỉnh *v* mà chưa được thăm và đi đến thăm đỉnh *u*. Trong trường hợp mọi đỉnh kề với đỉnh *v* đều đã được thăm thì ta quay lại đỉnh thăm ngay trước đỉnh *v* nếu có). Quá trình trên chỉ dừng lại khi không còn đỉnh nào có thể thăm được.

Thuật toán đệ quy duyệt theo chiều sâu xuất phát từ đỉnh *v* như sau:

1. Thăm đỉnh v.
2. Với mỗi đỉnh u kề với đỉnh v mà chưa được thăm, duyệt theo chiều sâu xuất phát từ đỉnh u.

Ví dụ: cho đồ thị như hình vẽ



Hình 4.4

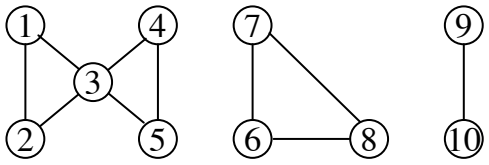
Duyệt đồ thị theo chiều sâu xuất phát từ đỉnh 1 sẽ lần lượt thăm các đỉnh như sau: xuất phát từ đỉnh 1 sẽ thăm đỉnh 1, sau đó chọn một đỉnh kề với đỉnh 1 nhưng chưa được thăm, ở đây có hai đỉnh là 2 và 3, theo thứ tự tự nhiên ta chọn đỉnh 2, thực hiện duyệt theo chiều sâu xuất phát từ đỉnh 2. Tương tự trên ta thăm đỉnh 2 và duyệt xuất phát từ đỉnh 3. Thăm đỉnh 3 rồi duyệt xuất phát từ đỉnh 4.

Thăm đỉnh 4 rồi duyệt xuất phát tại đỉnh 5. Thăm đỉnh 5 rồi tiếp tục duyệt xuất phát tại đỉnh 6. Thăm đỉnh 6, tại đỉnh 6 không còn đỉnh kề chưa được thăm nên quay lại đỉnh duyệt ngay trước đỉnh 6 là đỉnh 5. Tại đỉnh 5 các đỉnh kề đều đã được duyệt nên quay về đỉnh 4. Tại đỉnh 4 tìm được đỉnh 7 kề với đỉnh 4 chưa được duyệt. Tiến hành duyệt xuất phát tại đỉnh 7 tức là thăm đỉnh 7. Sau đó do các đỉnh kề của đỉnh 7 đều đã được duyệt nên quay lại đỉnh 4. Tại đỉnh 4 tất cả các đỉnh kề đều đã được duyệt nên quay về đỉnh được duyệt trước đỉnh 4 là đỉnh 3. Tương tự tại đỉnh 3 không có đỉnh kề chưa duyệt nên quay về đỉnh 2. Tại đỉnh 2 các đỉnh kề đã duyệt hết nên quay về đỉnh 1. Tại đỉnh 1 các đỉnh kề đã duyệt hết và đây là đỉnh xuất phát nên thuật toán dừng. Vậy thứ tự duyệt các đỉnh là: 1, 2, 3, 4, 5, 6, 7.

Trong phần sau trình bày thủ tục đệ quy duyệt theo chiều sâu xuất phát từ đỉnh *v*, trên đồ thị G được biểu diễn bằng ma trận kề. Trong thủ tục dùng mảng visitted theo số đỉnh của đồ thị, các phần tử kiểu boolean để đánh dấu các đỉnh của đồ thị đã được duyệt. Mảng này đầu tiên được khởi tạo với các giá trị false, nghĩa là các đỉnh của đồ thị chưa được duyệt.

```
Procedure DFS_Visit(v : 1..maxVerties; G : Graph);
var u : 1..maxVerties;
    Begin
        Visit(v);
        Visitted[v]:=fase;
        For u:=1 to G.Vnum do
            if (G.Adj[v,u]=1) and not visitted[u] then
                DFS_Visit(u,G);
    End;
```

Thủ tục duyệt đồ thị theo chiều sâu xuất phát từ đỉnh *v* sẽ lần lượt thăm tất cả các đỉnh liên thông với đỉnh *v* hay thăm tất cả các đỉnh trong một vùng liên thông chứa đỉnh *v*. Trong trường hợp cần duyệt toàn bộ các đỉnh của đồ thị ta phải thực hiện nhiều lần duyệt theo chiều sâu xuất phát từ các đỉnh của mỗi vùng liên thông. Chẳng hạn xét đồ thị như hình sau



Hình 4.5 Đồ thị với nhiều thành phần liên thông

Để duyệt toàn bộ đồ thị theo chiều sâu ta phải xuất phát từ 3 đỉnh của 3 vùng liên thông:

+ Xuất phát từ đỉnh 1 sẽ thăm các đỉnh: 1, 2, 3, 4, 5.

+ Xuất phát từ đỉnh 6 sẽ thăm các đỉnh: 6, 7, 8.

+ Xuất phát từ đỉnh 9 sẽ thăm các đỉnh: 9, 10.

Thủ tục duyệt toàn bộ đồ thị theo chiều sâu sử dụng thủ tục duyệt đồ thị theo chiều sâu xuất phát từ một đỉnh.

```
Procedure DFS(G : Graph);
var i : 1..maxVerties;
   Begin
      {khởi tạo mảng đánh dấu}
      For i:=1 to G.vnum do visitted[i]:=false;
      For i:=1 to G.vnum do
         if not visitted[i] then DFS_Visit(i,G);
   End;
```

15.2. Duyệt đồ thị theo chiều rộng

Thuật toán duyệt đồ thị theo chiều rộng xuất phát từ đỉnh *s* thực hiện qua các bước sau:

1. Xuất phát từ *s*, ta đi tới các đỉnh kề với đỉnh *s*.
2. Sau khi thăm mọi đỉnh kề với đỉnh *s*, ta tới các đỉnh chưa được thăm nhưng kề với những đỉnh được thăm ở bước 1.
3. Sau khi thăm các đỉnh ở bước 2, ta tới những đỉnh chưa được thăm nhưng kề với những đỉnh vừa được thăm tại bước 2.

....

Quá trình được lặp lại cho đến khi tất cả các đỉnh có thể đến được từ đỉnh s đều đã được thăm.

Ví dụ, với đồ thị như trong hình 4.4, thuật toán duyệt đồ thị theo chiều rộng xuất phát từ đỉnh 1 sẽ lần lượt thăm các đỉnh: đầu tiên thăm đỉnh 1, sau đó thăm các đỉnh kề với đỉnh 1 là đỉnh 2 và 3. Với hai đỉnh kề vừa duyệt là 2 và 3 thăm những đỉnh kề của chúng nhưng chưa được duyệt là 4, 5. Với hai đỉnh vừa duyệt 4 và 5 tiến hành duyệt các đỉnh kề chưa được duyệt của chúng là 7 và 6. Tiếp tục, các đỉnh kề của hai đỉnh vừa duyệt là 6 và 7 đều đã được duyệt nên thuật toán dừng. Vậy thứ tự các đỉnh duyệt theo chiều rộng xuất phát từ đỉnh 1 là: 1, 2, 3, 4, 5, 7, 6.

Thuật toán duyệt đồ thị theo chiều rộng xuất phát từ một đỉnh *s* sẽ tiến hành thăm các đỉnh của đồ thị theo từng mức một, và khi thăm một nút ở mức nào đó, ta phải lưu nút này để có thể quay lại thăm các nút kề với nó sau khi đi hết một mức. Thuật toán duyệt theo chiều rộng dùng một hàng đợi lưu các đỉnh kề với một đỉnh đã duyệt để xác định thứ tự duyệt của các đỉnh.

Thuật toán duyệt đồ thị theo chiều rộng xuất phát từ đỉnh s

1. Thăm nút xuất phát.
2. Khởi động một hàng đợi chỉ chứa nút xuất phát
3. Lặp khi hàng đợi khác rỗng
   a. Lấy một đỉnh v từ hàng đợi

b. Với tất cả các đỉnh w kề với đỉnh v, thực hiện các bước:

Nếu w chưa được thăm thì:

Thăm đỉnh w

Thêm đỉnh w vào hàng đợi

Thủ tục duyệt theo chiều rộng xuất phát từ đỉnh *s*, thực hiện trên đồ thị G biểu diễn bằng ma trận kề được cài đặt như sau:

```
Procedure BFS_Visit(s : 1..maxVerties; G : Graph);
var Q : QueueOfVerties; v, w : 1..maxVerties;
   Begin
      Visit(s);
      Vistted[s]:=true;
      InitQueue(Q);
      AddQueue(s, Q);
      While not empty(Q) do
         begin
            RemoveQueue(Q, v);
            For w:=1 to Q.Vnum do
               if (G.Adj[v,w]=1) and not visitted[w] then
                  begin
                     Visit(w);
                     Vistted[w]:=true;
                     AddQueue(w, Q);
                  end;
         end;
   End;
```

15.3. Tìm đường đi trên đồ thị

Trong thực tế chúng ta thường gặp những bài toán tìm đường đi giữa các điểm. Ví dụ bài toán tìm đường truyền dữ liệu từ trạm *s* đến trạm *d* trong một hệ thống truyền thông, trong một hệ thống giao thông là bài toán tìm đường đi từ nút giao thông *a* đến nút giao thông *b*,...

Bài toán: *Giả sử s, d là hai đỉnh của đồ thị G. Hãy tìm đường đi từ s đến d.*

Như đã nhận xét trong thao tác duyệt đồ thị, nếu ta dùng thuật toán duyệt theo chiều sâu hoặc theo chiều rộng xuất phát từ đỉnh *s* thì sẽ thăm tất cả các đỉnh cùng thành phần liên thông với đỉnh *s* hay nói cách khác là tập các đỉnh đến được nếu xuất phát từ đỉnh *s*. Từ đó ta thấy, có thể dùng thuật toán duyệt đồ thị xuất phát từ một đỉnh để biết có hay không đường đi từ đỉnh *s* đến đỉnh *d*, bằng cách duyệt xuất phát từ đỉnh *s*, sau khi duyệt, nếu giá trị visitted[d] là true thì có đường đi, ngược lại thì không có đường đi. Để ghi lại vết của đường đi, ta có thể dùng một mảng Previous với số phần tử bằng số đỉnh của đồ thị. Mỗi khi từ đỉnh *v* đi đến đỉnh *w* kề với nó thì lưu vết trước khi đến *w* thì ta đến *v* (Previous[w]:=v).

Thuật toán tìm đường đi từ đỉnh *s* đến đỉnh *d*, dùng thủ tục duyệt đồ thị theo chiều sâu DFS_Visit.

Khởi tạo mảng visitted với các giá trị false.

Duyệt đồ thị theo chiều sâu, xuất phát từ s (DFS_Visit(s)).

Nếu visitted[i]=false thì dừng, kết quả không có đường đi từ s đến t

Ngược lại in đường đi từ mảng previous.

Ví dụ, với đồ thị như hình 4.4, đường đi từ đỉnh 1 đến đỉnh 6 được lưu trong mảng previous như sau:

Previous[2]=1,Previous[3]=2,Previous[3]=2,Previous[4]=3,
Previous[5]=4,Previous[7]=4,Previous[6]=5.

Từ đó đường đi từ 1 đến 6 được khôi phục từ mảng previous như sau:

1=previous[2]→**2**=previous[3]→**3**=previous[4]→**4**=previous[5]→**5**=previous[6] → **6**.

Thủ tục tìm đường đi từ đỉnh *s* đến đỉnh *d* của đồ thị *G* được biểu diễn bằng ma trận kề sử dụng thủ tục DFS_Visit và mảng previous để lưu vết của đường đi.

```
Procedure Path(s, d:1..maxVerties; G : Graph);
var i:1..maxVerties;
   Begin
      For i:=1 to G.Vnum do visitted[i]:=false;
      DFS_Visit(s,G);
      if not visitted[d] then
         writeln('Không có đường đi')
      else
         WritePath(s,d);
   End;
```

Trong đó thủ tục DFS_Visit được thay đổi để lưu lại vết đường đi như sau:

```
Procedure DFS_Visit(v : 1..maxVerties; G : Graph);
var u : 1..maxVerties;
   Begin
      Visitted[v]:=fase;
      For u:=1 to G.Vnum do
         if (G.Adj[v,u]=1) and not visitted[u] then
            begin
               Previous[u]:=v;
               DFS_Visit(u,G);
            end;
   End;
```

Thủ tục WritePath dùng để in đường đi dựa vào mảng previous như sau:

```
Procedure WritePath(s, d : 1..maxVerties);
Begin
   if s<>d then
      begin
            WritePath(s,Previous[d]);
```

```
            Write('-->',d:4);
        end
    else Write(s:4);
End;
```

Hoàn toàn tương tự, ta có thể dùng thủ tục duyệt đồ thị theo chiều rộng để tìm đường đi. Hơn nữa đường đi tìm được theo cách này là đường đi qua ít cạnh nhất.

16. TÌM ĐƯỜNG ĐI NGẮN NHẤT

Trong phần trước ta đã trình bày thuật toán tìm đường đi giữa hai đỉnh trong đồ thị. Trong thực tế thì giữa hai nút *a* và *b* có thể có rất nhiều đường đi. Trong trường hợp này, bài toán đặt ra là hãy tìm ra con đường tốt nhất theo một tiêu chuẩn nào đó. Ví dụ trong bài toán tìm đường đi từ nút giao thông *a* đến nút giao thông *b* thì đường đi tốt nhất có thể là con đường đi nhanh nhất về thời gian, hoặc chi phí trên một đơn vị hàng hóa vận chuyển ít nhất,...

Trong phần này ta xét bài toán tìm đường đi ngắn nhất ở đồ thị không có trọng số và đồ thị có trọng số dương.

16.1. Đường đi ngắn nhất trên đồ thị không có trọng số

Bài toán: *Cho đồ thị không có trọng số G. Tìm đường đi từ đỉnh s đến đỉnh d sao cho qua ít cạnh (cung) nhất.*

Nếu ta dùng thuật toán tìm đường đi bằng cách dựa vào thủ tục duyệt theo chiều sâu thì dễ thấy kết quả đạt được không phải là đường đi ngắn nhất, nhưng nếu duyệt theo chiều rộng thì kết quả sẽ là đường đi ngắn nhất.

Thuật toán tìm đường đi ngắn nhất từ đỉnh *s* đến đỉnh d trong đồ thị không có trọng số. Thuật toán dùng thủ tục duyệt đồ thị theo chiều rộng với các mảng phụ: visitted dùng để đánh dấu các đỉnh đã được duyệt và mảng previous lưu vết của quá trình duyệt xuất phát từ đỉnh *s*.

Khởi tạo mảng visitted với các giá trị false.

Duyệt đồ thị theo chiều rộng, xuất phát từ đỉnh s.

Nếu visitted[i]=false thì dừng, kết quả không có đường đi từ s đến t

Ngược lại in đường đi từ mảng previous.

Ví dụ, với đồ thị như hình 4.5, nếu duyệt theo chiều rộng thì đường đi từ đỉnh 1 đến đỉnh 6 được lưu trong mảng previous như sau:
Previous[2]=1, Previous[3]=1, Previous[4]=3,
Previous[5]=3, Previous[7]=4, Previous[6]=5.

Từ đó đường đi từ 1 đến 6 được khôi phục từ mảng previous như sau:

1=previous[3]→**3**=previous[5]→**5**=previous[6] → **6**.

Thủ tục tìm đường đi ngắn nhất từ đỉnh *s* đến đỉnh *d* của đồ thị *G* được biểu diễn bằng ma trận kề sử dụng thủ tục BFS_Visit và mảng previous để lưu vết của đường đi.

```
Procedure ShortestPath(s, d:1..maxVerties; G : Graph);
var i:1..maxVerties;
   Begin
      For i:=1 to G.Vnum do visitted[i]:=false;
      BFS_Visit(s,G);
      if not visitted[d] then
         writeln('Không có đường đi')
      else
         WritePath(s,d);
   End;
```

Trong đó thủ tục BFS_Visit được thay đổi để lưu lại vết đường đi như sau:

```
Procedure BFS_Visit(s : 1..maxVerties; G : Graph);
var Q : QueueOfVerties; v, w : 1..maxVerties;
   Begin
      Vistted[s]:=true;
      InitQueue(Q);
      AddQueue(s, Q);
      While not empty(Q) do
         begin
            RemoveQueue(Q, v);
            For w:=1 to Q.Vnum do
               if (G.Adj[v,w]=1) and not visitted[w] then
                  begin
                     Vistted[w]:=true;
                     Previous[w]:=v;
                     AddQueue(w, Q);
                  end;
         end;
   End;
```

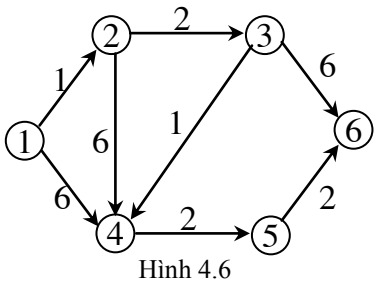
Thủ tục WritePath dùng để in đường đi dựa vào mảng previous như trong phần trước.

16.2. Đường đi ngắn nhất trên đồ thị có trọng số

Trọng số của những cạnh trong đồ thị có trọng số, tùy thuộc vào từng bài toán mà có ý nghĩa khác nhau. Ví dụ trong một hệ thống truyền thông, trọng số của mỗi đường truyền có thể là độ an toàn trên đường truyền đó. Trọng số của các cạnh trong một hệ thống giao thông có thể là độ dài của các con đường,... Bài toán tìm đường đi ngắn nhất trong đồ thị có trọng số được phát biểu như sau:

Bài toán: *Cho đồ thị có trọng số G. Tìm đường đi từ đỉnh s đến đỉnh d sao cho tổng trọng số của các cạnh trên đường đi là nhỏ nhất.*

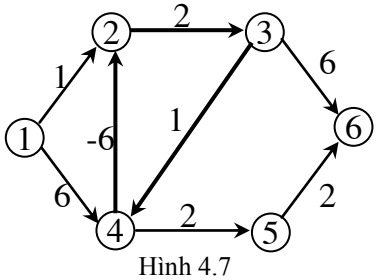
Ví dụ cho đồ thị như hình vẽ



Hình 4.6

Đường đi ngắn nhất từ đỉnh 1 đến đỉnh 6 là 1 → 2 → 3 → 4 → 5 → 6, có tổng trọng số là 1+2+1+2+2=8.

Một điều lưu ý là trọng số của các cạnh có thể là số âm. Khi đó nếu trên đường đi từ đỉnh *s* đến đỉnh *d* có một chu trình mà tổng trọng số các cạnh (cung) của chu trình là một số âm thì khái niệm đường đi ngắn nhất từ *s* đến *d* là không tồn tại. Ví dụ, với đồ thị như hình dưới đây, không tồn tại đường đi ngắn nhất từ đỉnh 1 đến đỉnh 6 do có chu trình âm (2,3,4,2) có tổng trọng số 2+1+-6=-3.



Hình 4.7

Một nhận xét quan trọng là ý tưởng chính cho các thuật toán tìm đường đi ngắn nhất đó là: *Mọi con đường trong đường đi ngắn nhất cũng là đường đi ngắn nhất*. Nghĩa là, nếu (p_1, p_2, ..., p_k) là đường đi ngắn nhất từ đỉnh p_1 đến đỉnh p_k thì mọi đường đi (p_i, p_{i+1}, ..., p_j), với $1 \le i < j \le k$ cũng là đường đi ngắn nhất từ p_i đến p_j.

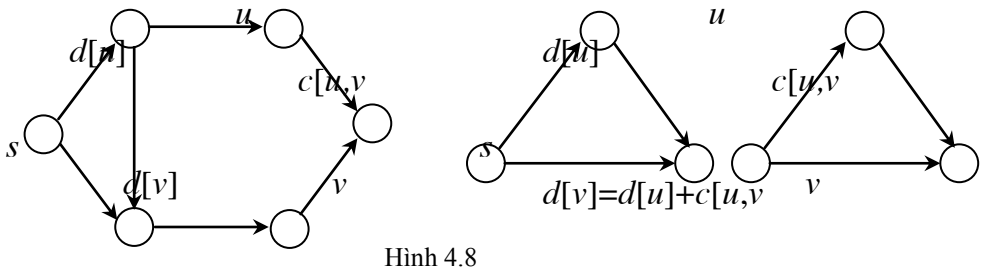
16.2.1. Thuật toán tìm đường đi ngắn nhất theo nguyên tắc giảm dần

Nguyên tắc giảm dần: xuất phát từ một đường đi từ đỉnh *s* đến đỉnh *d* (thường là đường đi trực tiếp từ *s* đến *d*), sau đó tìm những đường đi mới từ *s* đến *d* có tổng trọng số nhỏ hơn và thay đường đi cũ bởi đường đi mới. Quá trình trên lặp lại cho đến khi không tìm thêm được đường nào có trọng số nhỏ hơn đường đi đã có hoặc phát hiện ra có chu trình âm trong đường đi từ *s* đến *d*.

Để xây dựng thuật toán tìm đường đi ngắn nhất xuất phát từ đỉnh *s* dựa vào nguyên tắc giảm dần ta dùng các cấu trúc dữ liệu sau:

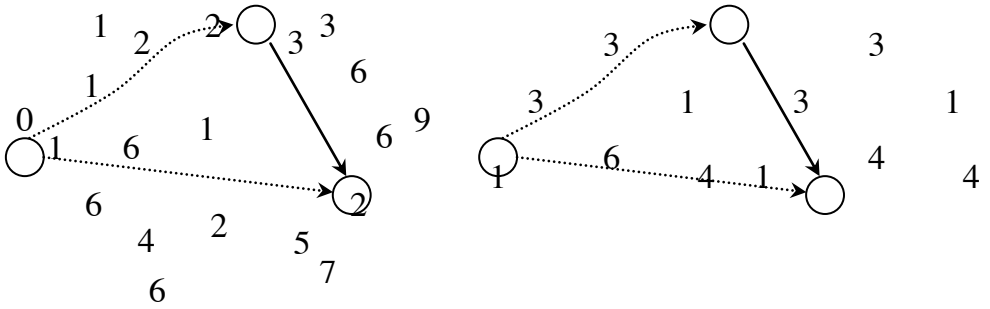
- *d* là mảng các số lưu độ dài ngắn nhất xuất phát từ s đến các đỉnh còn lại của đồ thị. Cụ thể, *d*[*v*] là độ dài đường đi ngắn nhất từ đỉnh *s* đến đỉnh *v*. Giá trị này được hoàn chỉnh qua các bước.

- Nếu có đỉnh u thỏa mãn $d[u] + c[u,v] < d[v]$ (với $c[u,v]$ là trọng số của cung (u,v) nếu có), nghĩa là có đường đi mới từ s đến u, rồi từ u đến v với trọng số mới là $d[u] + c[u,v]$ nhỏ hơn trọng số của đường đi cũ thì ta sẽ thay đường đi cũ với trọng số $d[v]$ bởi đường đi mới với trọng số $d[u] + c[u,v]$ (xem hình 4.8).
- Lặp lại quá trình trên khi không còn các đỉnh u và v thỏa mãn $d[u]+c[u,v]<d[v]$ hoặc xác định được chu trình âm.
- Để phát hiện được chu trình âm, tại mỗi bước ta kiểm tra xem có tồn tại một đỉnh v mà $d[v]$ nhỏ hơn tổng trọng số của các cạnh có trọng số nhỏ hơn 0. Nếu tồn tại đỉnh v như vậy thì kết luận là có chu trình âm.
- Để lưu chi tiết của đường đi ngắn nhất, tương tự như trong thuật toán tìm đường đi, ta dùng một mảng previous, mảng này được thay đổi mỗi khi ta chọn đường đi mới tốt hơn. Cụ thể nếu có đỉnh u thỏa mãn $d[u] + c[u,v] < d[v]$, ta sẽ thay đường đi cũ với trọng số $d[v]$ bởi đường đi mới với trọng số $d[u] + c[u,v]$ đồng thời ta cũng thay đổi đường đi previous[v]:=u.



Hình 4.8

Ví dụ: xét đồ thị như trong hình 4.7. Giả sử đã tính được đường đi từ đỉnh 1 đến các đỉnh khác (các số ghi trên các đỉnh của hình 4.9). Ta có d[4]=6 trong khi d[3]=3 và c[3,4]=1. Vậy d[4] sẽ được thay bằng d[3]+c[3,4]=4.



Hình 4.9

Thuật toán tìm đường đi ngắn nhất từ đỉnh s đến các đỉnh còn lại theo nguyên tắc giảm dần như sau:

1. Khởi tạo ban đầu d[v]=c[s,v], và previous[v]=s, nếu có cạnh (cung) từ s đến v, trong trường hợp ngược lại d[v] là một số rất lớn. t là tổng các cạnh có trọng số âm.
2. Tìm các cạnh (u,v) thỏa mãn d[u] + c[u,v] < d[v]. Nếu có thì thay đổi đường đi:

+ previous[v] := u;

+ d[v] := d[u] + c[u,v]

3. Nếu d[v] < t thì kết luận có chu trình âm và kết thúc.

4. Lặp lại bước 2 cho đến khi không tìm được cạnh (u,v).

Thủ tục tìm đường đi ngắn nhất xuất phát từ đỉnh s trên đồ thị G biểu diễn bằng ma trận kề như sau:

```
Procedure ShortesPath(s:1..maxVerties,G:Graph;
                                                    var found:Boolean);
  var u,v : 1..maxVerties; t:Real;
    Begin
      for u:=1 to G.vnum do
        if G.Adj[s,u]<>0 then
          begin
            d[u]:= G.Adj[s,u];
            previous[v]:=s;
          end
        else d[u]:=VC; {VC là một số rất lớn}
      t:=0;
      for u:=1 to G.vnum do
        for v:=1 to G.vnum do
          if G.Adj[u,v]<0 then t:=t+G.Adj[u,v];
      found:=true;
      repeat
        if find(u,v) then
          begin
            d[v]:=d[u]+G.Adj[u,v];
            previous[v]:=u;
            if d[v]<t then
              begin found:=false; break; end;
          end
        else break;
      until false;
    End;
```

Hàm tìm cặp đỉnh *u*, *v* thỏa điều kiện d[u]+c[u,v]<d[v] được viết như sau:

```
  Function Find(var u,v:1..maxVerties; G:Graph):Boolean;
    var i,j: 1..maxVerties;
    Begin
      find:=true;
      for i:=1 to G.vnum do
        for j:=1 to G.vnum do
          if (d[i]+G.Adj[i,j]<d[j]) then
            begin
              u:=i;
              v:=j;
              exit;
            end;
```

```
            find:=false;
        End;
```

Sau khi thực hiện thủ tục ShortesPath, nếu kết quả found là true thì đường đi ngắn nhất xuất phát từ đỉnh *s* được lưu trong mảng previous. Độ dài đường đi ngắn nhất từ dỉnh *s* đến đỉnh *t* là *d*[*t*].

16.2.2. Thuật toán Dijkstra

Thuật toán Dijkstra được áp dụng cho đồ thị có trọng số dương, ý tưởng chính của thuật toán như sau:

Gọi *S* là tập chứa các đỉnh đã xác định được đường đi ngắn nhất từ đỉnh *s* đến các đỉnh này. Với mỗi $u \in S$, gọi d[u] là độ dài đường đi ngắn nhất từ *s* đến *u*. Ban đầu khởi tạo $S = \{s\}$ và d[u] là trọng số của cung (*s*,*u*) nếu có; ngược lại thì d[u] được gán một số rất lớn.

Với mỗi đỉnh *x* của đồ thị không thuộc *S*, ta xác định $d[x]=\min\{d[u]+c[u,x], \forall u \in S\}$. Hay d[x] là đường đi ngắn nhất từ *s* đến *x* qua các đỉnh của tập *S*.

Xác định đỉnh *v* không thuộc vào *S* có d[v] nhỏ nhất $d[v]=\min\{d[u], u \notin S\}$. Kết nạp đỉnh *v* vào tập *S*. Do trọng số của đồ thị là các số dương nên hoàn toàn có thể chứng minh được rằng d[v] chính là đường đi ngắn nhất từ *s* đến *v*.

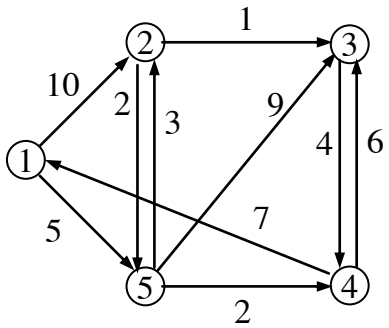
Lặp lại 2 bước trên cho đến khi tất cả các đỉnh của đồ thị đã được kết nạp vào tập *S*.

Để lưu vết của đường đi ngắn nhất, tương tự kỹ thuật tìm đường đi ta dùng một mảng previous. Mảng này được điều chỉnh mỗi khi sửa giá trị d[v]=d[u]+c[u,v] thì gán previous[v]:=u.

Thuật toán Dijkstra:

1. Khởi tạo S={s}, d[u]= c[s,u], và previous[u]=s, nếu có cạnh (cung) từ s đến u, trong trường hợp ngược lại d[u] là một số rất lớn.
2. Tìm $u \notin S$ có $d[u]=\min\{d[v], v \notin S\}$
3. Kết nạp u vào tập S.
4. Tính lại $d[v]=\min\{d[v], d[u]+c[u,v]\}$, và previous[v] cho các đỉnh $v \notin S$.
5. Lặp lại bước 2 cho đến khi S=V.

Ví dụ: cho đồ thị như hình vẽ



Hình 4.10

Chi tiết các bước của thuật toán Dijkstra tìm đường đi ngắn nhất từ đỉnh 1 đến các đỉnh còn lại thể hiện ở bảng sau:

Bước	Tập S	d[1], previous[1]	d[2], previous[2]	d[3], previous[3]	d[4], previous[4]	d[5], previous[5]
0	1	0, 0	10, 1	∞, 0	∞, 0	**5, 1**
1	1, 5	-	8, 5	14, 5	**7, 5**	-
2	1, 5, 4	-	**8, 5**	13, 4	-	-
3	1, 5, 4, 2	-	-	**9, 2**	-	-
4	1, 5, 4, 2, 3	-	-	-	-	-

Từ bảng trên ta có thể tìm đường đi ngắn nhất từ đỉnh 1 đến tất cả các đỉnh còn lại. Chẳng hạn đường đi ngắn nhất từ 1 đến 3 như sau:

3 ← previous[3]=**2** ← previous[2]=**5** ← previous[5]=**1**.

Tổng trọng số của các cạnh trên đường đi là d[3]=9.

Thủ tục sau tìm đường đi ngắn nhất xuất phát từ đỉnh *s* đến tất cả các đỉnh trên đồ thị bằng thuật toán Dijkstra có trọng số dương *G* được biểu diễn bằng ma trận kề.

```
Procedure Dijkstra(s:1..maxVerties; G:Graph);
var i,u,count:1..maxVerties;minLabel : Real;
   previous:Array[1..maxVerties] of 1..maxVerties;
       d:Array[1..maxVerties] of Real;
   Mark:Array[1..maxVerties] of Boolean;
    Begin
    {khởi tạo nhãn}
    For i:=1 to G.vnum do
     begin
      if G.Adj[s,i]>0 then
        begin
          d[i]:=G.Adj[s,i];
          Previous[i]:=s;
        end
      else d[i]:=VC; {VC là tổng của các trọng số}
      Mark[i]:=false;
     end;

     Previous[s]:=0;
     d[s]:=0;
     Mark[s]:=true;
     count:=1;
           while count<G.vnum do
       begin
          {tìm đỉnh u có nhãn nhỏ nhất}
```

```
  minLabel:=VC;
            for i:=1 to G.vnum do
    if (not mark[i]) and (minLabel>d[i]) then
      begin
         u:=i;
         minLabel:=d[i];
      end;
  Mark[u]:=true;count:=count+1;
            for i:=1 to G.vnum do  {gán lại nhãn cho các đỉnh}
   if(not mark[i]) and (d[u]+g[u,i]<d[i]) then
     begin
       d[i]:=d[u]+G.Adj[u,i];
       Previous[i]:=u;
     end;
end;
End;
```

Sau khi thực hiện thủ tục Dijkstra, với mỗi đỉnh *v*, nếu d[v]<VC, thì thực sự có đường đi ngắn nhất từ *s* đến *v*, và bằng cách duyệt ngược trong mảng previous từ vị trí *v* cho đến khi gặp đỉnh *s* thì ta tìm được chi tiết đường đi ngắn nhất từ *s* đến *v*. Nếu d[v]=VC thì thực sự không có đường đi từ *s* đến *v* vì đường đi ngắn nhất phải qua những cung không có.

16.2.3. Tìm đường đi ngắn nhất giữa các cặp đỉnh

Thuật toán Dijkstra cho phép tìm đường đi ngắn nhất từ một đỉnh đến các đỉnh còn lại của đồ thị có trọng số dương. Trong trường hợp cần tìm đường đi giữa hai đỉnh bất kỳ thì thuật toán Dijkstra thực hiện phức tạp vì phải tính lại cho đỉnh xuất phát mới. Thuật toán Floyd là một thuật toán đơn giản đáp ứng được yêu cầu trên. Ý tưởng chính của thuật toán Floyd là lần lượt dùng mỗi nút u của đồ thị làm trục quay. Khi đó chúng ta dùng *u* như một đỉnh trung gian giữa tất cả các cặp đỉnh. Đối với mỗi cặp đỉnh, chẳng hạn *v* và *w*, nếu tổng các nhãn của hai đường đi từ *v* đến *u* và từ *u* đến *w* nhỏ hơn nhãn hiện tại của đường đi từ *v* đến *w* thì chúng ta thay đường đi từ *v* đến *w* đã xây dựng ở bước trước bằng đường đi mới qua đỉnh *u*.

Thuật toán Floyd tìm đường đi ngắn nhất của mọi cặp đỉnh sử dụng các cấu trúc dữ liệu sau:

Mảng dd[1..maxVerties, 1..maxVerties] kiểu số thực để lưu độ dài đường đi ngắn nhất giữa các cặp đỉnh.

Mảng pp[1..maxVerties, 1..maxVerties] kiểu số nguyên để lưu đường đi ngắn nhất tìm được giữa các cặp đỉnh. pp[v, w]=u có nghĩa là đường đi ngắn nhất từ *v* đến *w* trước khi đến *w* phải qua *u*.

Thuật toán Floyd:

1. Khởi tạo các giá trị cho mảng dd và pp:

   Với mỗi cặp đỉnh v, w của đồ thị

nếu có cung (v, w) thì

dd[v,w] = trọng số cung (v,w)

pp[v,w] = v

ngược lại

dd[v,w] = VC

pp[v,w] = 0

2. Với mỗi đỉnh u, v, w của đồ thị thực hiện

nếu dd[v, w] > dd[v,u] + dd[u, w] thì đổi đường đi

dd[v,w]:=dd[v,u]+dd[u,w];

pp[v,w]:=u;

Thủ tục Floyd tìm đường đi ngắn nhất giữa các cặp đỉnh của đồ thị trọng số dương G biểu diễn bằng ma trận kề như sau:

```
Procedure Floyd(G:Graph);
var u, v, w:integer;
   Begin
      {Khoi tao}
      For v:=1 to G.vnum do
      For w:=1 to G.vnum do
         begin
         if G.Adj[v,w]>0 then
            begin
            dd[v,w]:=G.Adj[v,w];
         pp[v,w]:=v;
            end
         else dd[i,j]:=VC;
       end;

      For u:=1 to G.vnum do
      For v:=1 to G.vnum do
         For w:=1 to G.vnum do
         if dd[v,w]> dd[v,u]+dd[u,w] then
          begin
            dd[v,w]:=dd[v,u]+dd[u,w];
            pp[v,w]:=u;
          end;
            End;
```

Sau khi thực hiện thủ tục Floyd ta có thể tìm được đường đi ngắn nhất giữa các cặp đỉnh bất kỳ của đồ thị bằng cách duyệt trong mảng pp, và độ dài đường đi được lưu trong mảng dd.

Dễ thấy độ phức tạp của thuật toán là $O(n^3)$, với n là số đỉnh của đồ thị.

17. CÂY KHUNG CỦA ĐỒ THỊ

17.1. Khái niệm cây khung (Spanning tree)

Cho một đồ thị vô hướng, liên thông $G=<V,E>$. Một *cây khung* của đồ thị G là một đồ thị có tập các đỉnh trùng với tập đỉnh của đồ thị G và tập các cạnh là một tập con của E, ký hiệu, $H=<V, E'>$ thỏa các điều kiện sau:

+ H là một đồ thị liên thông

+ H không có chu trình

Ví dụ: Cho đồ thị như hình 4.11a, một cây khung của đồ thị như hình 4.11b.

a) Đồ thị b) Cây khung

Hình 4.11

Nhận xét: *- Với một đồ thị cho trước, có thể có nhiều cây khung.*

- Số cạnh của cây khung là n-1 với n là số đỉnh của đồ thị.

17.2. Thuật toán tìm cây khung của đồ thị

Để tìm một cây khung của đồ thị ta có thể dùng thuật toán duyệt đồ thị theo chiều sâu hoặc chiều rộng xuất phát từ một đỉnh bất kỳ của đồ thị, chi tiết như sau:

1. Xuất phát cây khung $H=<V, \varnothing>$ (không có cạnh nào)
2. Chọn đỉnh s bất kỳ của đồ thị
3. Duyệt đồ thị theo chiều sâu xuất phát từ đỉnh s, mỗi khi từ đỉnh v duyệt đỉnh kề nhưng chưa được duyệt w thì ta thêm cạnh (v,w) vào cây khung.

Thủ tục tìm một cây khung H từ đồ thị G được biểu diễn bằng ma trận kề, dùng thuật toán duyệt theo chiều sâu như sau:

```
Procedure SpanningTreeDFS(v : 1..maxVerties; G:Graph);
var i:1..maxVerties;
Begin
    Visitted[v]:=True;
    for i:=1 to G.vnum do
       if (G.Adj[v,i]<>0) and (not Visitted[i]) then
       begin
         H.Adj[v,i]:=G.Adj[v,i];
                    H.Adj[i,v]:=G.Adj[i,v];
                SpanningTreeDFS(i,G);
       end;
```

```
End;
```

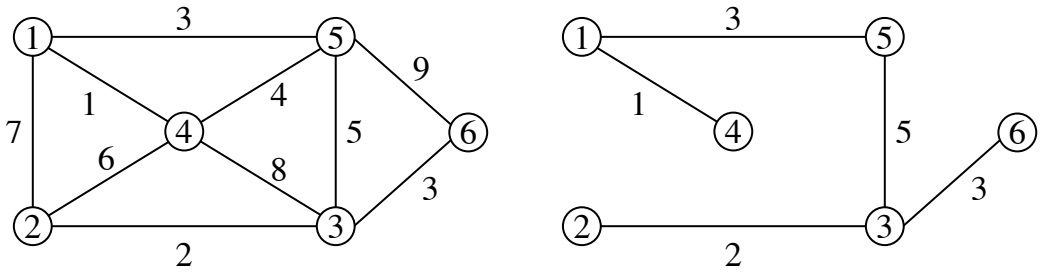
Khi thực hiện thủ tục SpanningTreeDFS xuất phát từ một đỉnh bất kỳ của đồ thị ta sẽ được một cây khung *H*.

```
Procedure SpanningTree(G:Graph; var H:Graph);
  Begin
    H.vnum:=G.vnum;
    SpanningTreeDFS(1);
  End;
```

17.3. Cây khung ngắn nhất

Cho *G* là một đồ thị có trọng số, *H* là một cây khung của *G*, cây khung *H* được gọi là cây *khung ngắn nhất* (minimal spanning tree) của đồ thị *G* nếu tổng các trọng số của cây khung *H* là nhỏ nhất trong các cây khung của *G*.

Ví dụ: Cho đồ thị có trọng số như hình 4.12a, một cây khung ngắn nhất của đồ thị ở hình 4.12b.



a) Đồ thị b) Cây khung ngắn nhất

Hình 4.12 Cây khung ngắn nhất

Bài toán tìm cây khung ngắn nhất của đồ thị là một bài toán có ý nghĩa thực tế. Chẳng hạn để xây dựng một mạng lưới giao thông sao cho giữa hai nút giao thông bất kỳ đều có đường đi và chi phí xây dựng mạng lưới giao thông là nhỏ nhất.

17.4. Thuật toán tìm cây khung ngắn nhất của đồ thị

Các thuật toán tìm cây khung ngắn nhất thường dùng kỹ thuật tham lam bằng cách xây dựng tập các cạnh *T* dần từng bước xuất phát từ tập *T* rỗng. Trong mỗi bước ta sẽ kết nạp cạnh (*u*,*v*) "tốt nhất" trong các cạnh chưa được chọn vào tập *T*. Hai thuật toán thường dùng để tìm cây khung ngắn nhất là thuật toán Prim và thuật toán Kruskal.

17.4.1. Thuật toán Prim

Thuật toán Prim dùng để tìm cây khung ngắn nhất của một đồ thị được thực hiện bằng cách chọn dần các cạnh, ở mỗi bước chọn cạnh trong các cạnh còn lại có

trọng số nhỏ nhất và tạo với các cạnh đã chọn thành một đồ thị không có chu trình. Quá trình chọn cho đến khi được đồ thị liên thông.

Gọi T là tập các cạnh được chọn (xuất phát $T=\varnothing$), U là tập các đỉnh có trong các cạnh của T. Ta xây dựng T bằng cách xuất phát từ tập U là một đỉnh bất kỳ của đồ thị, còn tập T là tập rỗng. Tại mỗi bước ta sẽ kết nạp vào T một cạnh (u,v) có trọng số nhỏ nhất trong các cạnh của đồ thị thỏa mãn điều kiện $u \in U$ và $v \in V \backslash U$, đồng thời cũng kết nạp v vào U. Điều kiện này nhằm đảm bảo cho các cạnh được chọn trong T không tạo thành chu trình. Ta phát triển T cho đến khi $U=V$ thì dừng.

Thuật toán Prim: Tìm cây khung ngắn nhất T của đồ thị G=<V,E>

Xuất phát $U=\{v_0\}$ (đỉnh bất kỳ)

$T=\varnothing$

Lặp

Chọn cạnh $(u,v) \in E$, có trọng số nhỏ nhất thỏa $u \in U$ và $v \in V \backslash U$

$U:=U \cup \{v\}$

$T:=T \cup \{(u,v)\}$

Đến khi $U=V$

Ví dụ: xét cây như hình 4.12a, thuật toán Prim tìm cây khung ngắn nhất thực hiện qua các bước:

Bước	(u,v)	U	T
khởi tạo	-	{1}	{}
1	(1,4)	{1, 4}	{(1,4)}
2	(1,5)	{1, 4, 5}	{(1,4), (1,5)}
3	(5,3)	{1, 4, 5, 3}	{(1,4), (1,5), (5,3)}
4	(3,2)	{1, 4, 5, 3, 2}	{(1,4), (1,5), (5,3), (3,2)}
5	(3,6)	{1, 4, 5, 3, 2, 6}	{(1,4), (1,5), (5,3), (3,2), (3,6)}

Cài đặt thuật toán Prim:

Giả sử đồ thị G được biểu diễn bằng ma trận kề. Dùng 2 mảng:

Mảng nearest[2..maxVerties]các số nguyên để lưu các cặp đỉnh (u,v) được chọn thỏa điều kiện trong mỗi bước của thuật toán. Cụ thể, với mỗi đỉnh $v \in V \backslash U$, $u \in U$ mà trọng số của cạnh (u,v) nhỏ nhất thì nearest[v]=u.

Mảng dist[2..maxVerties] các số thực, với $v \in V \backslash U$, dist[v]lưu độ dài của cạnh nối đỉnh v với đỉnh nearest[v].

Mảng Visitted[1..maxVerties] kiểu Boolean để đánh dấu các đỉnh đã được chọn trong U.

Mảng nearest và dist được khởi tạo ban đầu dựa vào các đỉnh kề với đỉnh được chọn để xuất phát v_0 và sửa lại sau mỗi bước chọn được một cạnh.

Thuật toán tìm cây khung ngắn nhất T của đồ thị G.

```
Procedure Prim(G:Graph;var T:Graph);
var    i,k,minL,count: Integer; Stop : Boolean;
    Visitted : Array[1..maxVerties] of Boolean;
    Nearest : Array[2..maxVerties] of 1..maxVerties;
        Dist : Array[2..maxVerties] of Real;
    Begin

        {Khởi tạo}
   For i:=1 to G.vnum do Visitted[i]:=false;
   k:=1; {chon dinh dau tien bat ky}
   Visitted[k]:=true;

   {Gán nhãn}
   For i:=1 to G.vnum do
    if (G.Adj[k,i]>0) then
      begin
        Dist[i]:=G.Adj[k,i];
        Nearest[i]:=k;
      end
    else
      begin
        Dist[i]:=VC;
        Nearest[i]:=0;
      end;

   count:=0; stop:=false;

   Repeat

     {Chọn cung có trọng số nhỏ nhất}
     minL:=VC; Stop:=true;
     For i:=1 to G.vnum do
       if not Visitted[i] and (Dist[i]<minL) then
          begin
             k:=i;
             minL:=Dist[i];
             Stop:=false;
          end;
       Visitted[k]:=true;
       H.Adj[Nearest[k],k]:=minL;
             H.Adj[k,Nearest[k]]:=minL;

       {Sửa nhãn}
       For i:=1 to G.vnum do
       if (G.Adj[k,i]>0) and not Visitted[i] and
                                                    (Dist[i]>G.Adj[k,i]) then
         begin
           Nearest[i]:=k;
```

```
        Dist[i]:=G.Adj[k,i];
      end;
    count:=count+1;
  Until (count=G.vnum-1) or stop;
  End;
```

Gọi n là số đỉnh của đồ thị. Độ phức tạp thuật toán Prim được đánh giá như sau: vòng lặp Repeat thực hiện n-1 lần, mỗi lần lặp thực hiện tối đa n lần lặp để điều chỉnh các nhãn tại các đỉnh. Do đó độ phức tạp của thuật toán Prim là $O(n^2)$.

17.4.2. Thuật toán Kruskal

Tương tự như thuật toán Prim, thuật toán Kruskal cũng xây dựng tập T các cạnh của cây khung xuất phát từ tập rỗng nhưng tại mỗi bước cạnh (u,v) được chọn là một cạnh có trọng số nhỏ nhất và khi kết hợp vào T không tạo nên chu trình.

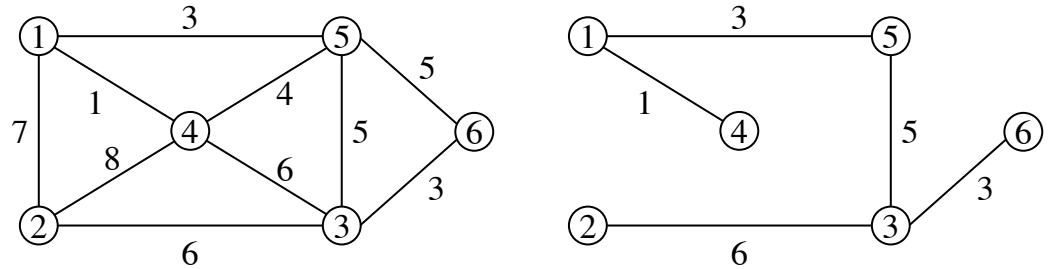
Giả sử $G=<V,E>$ là đồ thị có trọng số cho trước, cần xây dựng cây khung ngắn nhất $H=<V,T>$ của G ta tiến hành như sau:

Xuất phát $T=\varnothing$, khi đó H là một đồ thị có n thành phần liên thông, với n là số đỉnh của đồ thị, mỗi thành phần liên thông là một đỉnh.

Để lần lượt chọn các cạnh $(u,v)\in E \setminus T$ để kết nạp vào T ta thực hiện như sau: Ta xét các cạnh của đồ thị G theo thứ tự tăng của trọng số. Các cạnh (u,v) được chọn để kết nạp vào T theo thứ tự tăng dần của trọng số và chỉ kết nạp khi hai đỉnh u, v nằm ở hai vùng liên thông khác nhau của đồ thị H, bởi vì khi đó các cạnh của T sau khi kết nạp (u,v) sẽ không tạo ra chu trình. Sau khi thêm cạnh (u, v) thì hai thành phần liên thông của H chứa u và v sẽ được hợp nhất thành một thành phần liên thông. Trong trường hợp hai đỉnh u, v thuộc cùng một thành phần liên thông của H thì không chọn cạnh này (kể cả các bước tiếp theo).

Quá trình chọn các cạnh để kết nạp vào T lặp lại cho đến khi T có đủ n-1 cạnh thì dừng.

Ví dụ: Xét đồ thị như hình vẽ



a) Đồ thị b) Cây khung ngắn nhất bằng thuật toán Kruskal

Hình 4.13 Cây khung ngắn nhất

Chi tiết các bước như sau:

Bước	(u,v)	T∪{(u,v)} Chu trình?	T
khởi tạo	-	-	{}
1	(1,4)	Không	{(1,4)}
2	(1,5)	Không	{(1,4), (1,5)}
3	(3,6)	Không	{(1,4), (1,5), (3,6)}
4	(4,5)	Có	{(1,4), (1,5), (3,6)}
5	(3,5)	Không	{(1,4), (1,5), (3,6),(3,5)}
6	(5,6)	Có	{(1,4), (1,5), (3,6),(3,5)}
7	(2,3)	Không	{(1,4), (1,5), (3,6),(3,5), (2,3)}

Thuật toán Kruskal: Tìm cây khung ngắn nhất T của đồ thị G=<V,E>

Sắp xếp các cạnh của đồ thị theo thứ tự tăng của trọng số

Xuất phát T=∅

Lặp

Chọn cạnh (u,v)∈E, có trọng số nhỏ nhất

E:=E \ {(u, v)}

Nếu T∪{(u,v)} không chứa chu trình thì T:=T∪{(u,v)}

Đến khi |T|=|V|-1

Cài đặt thuật toán Kruskal:

Để đơn giản, ta biểu diễn đồ thị *G* bằng danh sách cạnh, và cây khung *T* là danh sách cạnh.

Để kiểm tra hai đỉnh *u*, *v* có cùng một thành phần liên thông của đồ thị *T* không ta sử dụng mảng comp[1..maxVerties] kiểu số nguyên, trong đó comp[i] là số hiệu của thành phần liên thông chứa đỉnh i. Để thống nhất, số hiệu của 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, 7, 5} là 2. Trong trường hợp chọn một cạnh (*u*,*v*) mà hai đỉnh *u* và *v* nằm ở hai thành phần liên thông khác nhau khi đó ta phải ghép hai thành phần liên thông bởi cạnh (*u*,*v*). Thủ tục nối hai vùng liên thông được thực hiện trên đồ thị có *n* đỉnh như sau:

```
Procedure Merge(u, v, n : 1..maxVerties);
var a, b : 1..maxVerties;
  Begin
    a:=comp[u];
    b:=comp[v];
    if a>b then begin tg:=a;a:=b;b:=a; end;
    for i:=1 to n do
        if comp[i]=b then comp[i]:=a;
  End;
```

Thủ tục Kruskal xây dựng cây khung ngắn nhất *T* từ đồ thị *G*.

```
Procedure Kruskal(G:Graph; var T:Graph);
var i, k, count : 1..maxVerties;
    Begin
        {sắp thứ tự các cạnh tăng theo trọng số}
        QuickSort(G);
        T.Vnum:=G.Vnum;
        For i:=1 to G.Vnum do comp[i]:=i;
        k:=1;count:=0;
        Repeat
            u:=G.Edges[k].firstVertex;
            v:=G.Edges[k].lastVertex;
            if comp[u]<>comp[v] then
                begin
                    Count:=Count+1;
                    T.Edges[Count]:=G.Edges[k];
                    k:=k+1;
                    Merge(u,v,G.vnum);
                end;
        Until Count=G.Vnum-1;
    End;
```

Đánh giá độ phức tạp: Gọi *m* là số cạnh của đồ thị, *n* là số đỉnh của đồ thị. Khi đó độ phức tạp trung bình của thủ tục QuickSort là $O(m\log_2 m)$. Vòng lặp repeat chọn các cạnh cho cây khung *T* lặp tối đa *m* lần, mỗi lần thực hiện thao tác trộn hai thành phần liên thông với độ phức tạp là $O(n)$. Do đó độ phức tạp của đoạn chương trình lặp repeat ... until là $O(mn)$. Do $m \le n^2$ nên $\log_2 m \le 2\log_2 n \le n$, với *n* đủ lớn. Vậy độ phức tạp của thuật toán trên là $O(mn)$.

18. BÀI TẬP

Bài 1. Viết thủ tục duyệt đồ thị theo chiều sâu xuất phát từ một đỉnh, với đồ thị tổ chức bằng danh sách kề. Dùng thủ tục này để viết thủ tục tìm đường đi giữa 2 đỉnh của đồ thị chức bằng danh sách kề.

Bài 2. Tổ chức dữ liệu lưu sơ đồ các chuyến bay của một số sân bay hàng không. Viết các thủ tục thực hiện việc tìm một cách chọn các chuyến bay từ sân bay này đến sân bay khác theo tên sân bay, có thể phải qua một số sân bay trung gian. Yêu cầu như trên nhưng qua ít sân bay trung gian nhất.

Bài 3. (*Thành phần liên thông của đồ thị*) Một thành phần liên thông của đồ thị vô hướng G = <V, E> là một tập đỉnh $V' \subseteq V$ thỏa mãn 2 điều kiện:

i) Với mọi cặp đỉnh u, $v \in V'$, tồn tại đường đi từ u đến v.

ii) Nếu thêm bất kỳ một đỉnh $x \in V \backslash V'$ vào V' thì V' không còn tính chất i).

Tổ chức dữ liệu, nêu thuật toán và viết thủ tục thực hiện liệt kê các thành phần liên thông của đồ thị.

Bài 4. (*Cầu của đồ thị*) Một cạnh e = (u,v) ∈ E của đồ thị vô hướng G=<V,E> được gọi là cầu nếu số thành phần liên thông của G sẽ tăng lên nếu xóa bỏ cạnh e trong G. Trình bày thuật toán và viết hàm kiểm tra một cạnh có phải là cầu của đồ thị hay không?.

Bài 5. (*Đỉnh khớp*) Một đỉnh u được gọi là đỉnh khớp nếu như việc bỏ u và những cạnh nối với u làm cho số thành phần liên thông của đồ thị tăng lên. Trình bày thuật toán và viết hàm kiểm tra một đỉnh có phải là đỉnh khớp của đồ thị hay không?.

Chương 5

CÁC CẤU TRÚC DỮ LIỆU Ở BỘ NHỚ NGOÀI

19. MÔ HÌNH TỔ CHỨC DỮ LIỆU Ở BỘ NHỚ NGOÀI

Các cấu trúc dữ liệu trình bày trong các chương trước được tổ chức và lưu trữ trong bộ nhớ trong. Đặc điểm của bộ nhớ trong là truy xuất nhanh nhưng dung lượng hạn chế do đó nhiều ứng dụng thực tế không thể tổ chức dữ liệu để lưu trữ các đối tượng ở bộ nhớ trong mà phải lưu trữ ở bộ nhớ ngoài, thông thường là đĩa. Các thiết bị nhớ ngoài thường có dung lượng lớn tuy nhiên thời gian truy cập đến dữ liệu chậm hơn so với bộ nhớ trong. Do đó việc tổ chức lưu trữ dữ liệu trên thiết bị nhớ ngoài như thế nào để dễ truy xuất và thực hiện các thao tác cơ bản như tìm kiếm, xóa, sửa,.. có ý nghĩa rất quan trọng.

Trước hết ta tìm hiểu một số nét chính về cách tổ chức lưu trữ dữ liệu trên thiết bị nhớ ngoài mà các hệ điều hành thường sử dụng. Các hệ điều hành hiện nay đều lưu các dữ liệu ra bộ nhớ ngoài dưới dạng các file.

Một file có thể xem là một tập hợp các bản ghi được lưu ở bộ nhớ ngoài. Các bản ghi này có thể có độ dài cố định hoặc thay đổi. Để đơn giản, trong phần này ta chỉ xét các file chứa các bản ghi với chiều dài cố định cho mỗi bản ghi. Mỗi bản ghi trong file có một khóa là dữ liệu của một hoặc nhiều trường. Khóa của bản ghi hoàn toàn xác định được bản ghi trong file.

Hệ điều hành chia bộ nhớ ngoài thành các khối vật lý (physical block) có kích thước như nhau, gọi tắt là các khối. Cỡ của các khối tùy thuộc vào 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 trên đĩa, tức là địa chỉ của byte đầu tiên trong khối. Mỗi file được lưu trọn vẹn trong một số khối, mỗi khối có thể lưu được một số bản ghi của file. Trong một khối có thể còn một số byte chưa được sử dụng đến. Mỗi bản ghi lưu trong file có địa chỉ là cặp (k, s), trong đó k là địa chỉ của khối chứa bản ghi, và s là số byte trong khối đứng trước byte bắt đầu bản ghi (còn được gọi là offset).

Khi thực hiện các thao tác trên file phải thực hiện thao tác đọc một khối dữ liệu ở bộ nhớ ngoài vào vùng đệm của bộ nhớ trong và các thao tác với dữ liệu được thực hiện ở bộ nhớ trong. Sau đó dữ liệu từ bộ nhớ trong được lưu trở lại ra bộ nhớ ngoài. Ta gọi các phép toán này là các phép toán truy cập khối. Cần lưu ý rằng thời gian đọc một khối dữ liệu từ bộ nhớ ngoài vào bộ nhớ trong đòi hỏi nhiều thời gian hơn việc tìm kiếm dữ liệu trên khối đó khi đã nạp vào bộ nhớ trong. Các thao tác với dữ liệu chỉ được thực hiện ở bộ nhớ trong nên tất cả các thao tác đều yêu cầu phải nạp dữ liệu vào bộ nhớ trong. Từ đó, để đánh giá thời gian thực hiện thuật toán đối với các dữ liệu ở bộ nhớ ngoài phải tính số lần cần thiết thực hiện các phép truy cập khối và đây được xem là tính hiệu quả của thuật toán.

Phương pháp đơn giản nhất để lưu trữ các bản ghi của file là xếp các bản ghi vào một số khối cần thiết theo một trật tự tùy ý.

Phép toán tìm kiếm các bản ghi dựa vào khóa hoặc các giá trị đã biết thường được thực hiện tuần tự bằng cách đọc lần lượt các bản ghi trong các khối.

Việc thêm các bản ghi vào file thường được thêm vào khối cuối cùng của file nếu khối đó còn đủ chỗ trống, nếu không thì thêm vào file một khối mới và đặt bản ghi cần thêm vào đầu khối đó.

Muốn xóa một bản ghi, trước hết cần xác định vị trí của bản ghi trong file. Việc xóa bỏ một bản ghi trong file có thể thực hiện bằng nhiều cách, một trong những cách thường được sử dụng là dùng bit xóa để đánh dấu bản ghi đã được xóa.

Với cách tổ chức file tuần tự như trên, các phép toán trên file sẽ thực hiện rất chậm vì gần như phải truy xuất đến tất cả các khối của file. Trong phần sau sẽ trình bày một số cách tổ chức file ưu việt hơn, cho phép mỗi lần truy xuất một bản ghi chỉ cần đọc một lượng nhỏ dữ liệu vào bộ nhớ trong.

Một điều lưu ý là các thuật toán và cách tổ chức dữ liệu trong chương này không thể thực hành bằng các ngôn ngữ bậc cao vì việc truy xuất đến dữ liệu ở mức khối vật lý và các địa chỉ khối là chức năng của hệ điều hành.

20. FILE BĂM

20.1. Cấu trúc Bảng băm (Hash Table)

Trong các cách tổ chức dữ liệu trước đây việc tìm kiếm một phần tử trong một tập hợp n phần tử có độ phức tạp là $O(n)$ với tìm kiếm tuần tự và $O(\log_2 n)$ với tìm kiếm nhị phân. Trong một số bài toán thực tế thường xuyên thực hiện các thao tác tìm kiếm theo khóa, những thuật toán này còn quá chậm.

Ví dụ, bảng các ký hiệu được lập bởi bộ dịch lưu trữ các định danh và những thông tin về chúng. Tốc độ xây dựng và tìm kiếm các định danh trong bảng sẽ quyết định tốc độ của chương trình dịch. Một cấu trúc dữ liệu hỗ trợ việc lưu trữ các phần tử phục vụ tìm kiếm nhanh thường được dùng là bảng băm. Bảng băm là một cách lưu trữ các mục dữ liệu, trong đó vị trí của một mục được xác định trực tiếp bằng một hàm dựa vào dữ liệu của mục dữ liệu. Do đó thời gian xác định vị trí của một mục trong bảng băm trong trường hợp tốt nhất là O(1), nghĩa là không phụ thuộc vào số các mục dữ liệu được lưu trữ.

Để minh họa, giả sử rằng cần lưu trữ 25 số nguyên trên đoạn 0..999 trong bảng băm. Có thể biểu diễn bảng băm này bằng một mảng Table các số nguyên được đánh chỉ số trên đoạn 0..999, trong đó mỗi phần tử của mảng được khởi tạo với một giá trị đặc biệt nào đó, chẳng hạn -1. Với mỗi phần tử có giá trị là i cần được lưu trong bảng ta có thể gán Table[i]:=i. Với cách tổ chức như vậy, để kiểm tra một số x có trong bảng không ta chỉ cần thực hiện một phép so sánh Table[x]=x?.

20.1.1. 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 phân tập hợp các phần tử cần quản lý thành *N* lớp, đánh số từ 0..*N*-1. Ta sử dụng một mảng *T* với các chỉ số từ 0 đến *N*-1, mỗi phần tử của mảng có thể lưu được các phần tử thuộc một lớp. Các phần tử thuộc một lớp được tổ chức dưới dạng một danh sách liên kết. Như vậy bảng *T* là bảng các con trỏ, trong đó T[i] là con trỏ đến danh sách các phần tử thuộc lớp thứ i.

Việc phân chia các phần tử của tập hợp *X*, với tập các khóa *K* vào các lớp được thực hiện bởi một hàm $h : K \rightarrow \{0, 1, .., N\text{-}1\}$, gọi là hàm băm (hash function). Nếu *x* là khóa của một phần tử nào đó trong tập *X*, thì *h*(*x*) là chỉ số lưu phần tử này trong bảng T. Ta gọi *h*(*x*) là giá trị băm của phần tử có khóa là *x*. Một trong các điều kiện bắt buộc của bảng băm lưu các phần tử là phải xây dựng được hàm băm hiệu quả.

Có hai tiêu chuẩn chính để lựa chọn một hàm băm: trước hết phải đơn giản trong tính toán. Thứ hai là phải phân bố đều các phần tử vào các lớp. Trên thực tế, tiêu chuẩn thứ hai rất khó đạt được. Sau đây là một số phương pháp thiết kế hàm băm.

Phương pháp cắt bỏ:

Giả sử khóa của các phần tử là những số nguyên. Ta sẽ bỏ đi một phần nào đó của khóa, và lấy phần còn lại làm giá trị băm của khóa. Chẳng hạn khóa là các số nguyên 10 chữ số và bảng băm gồm 1000 thành phần, khi đó ta có thể lấy ba chữ số ở các vị trí 1, 3, 5 làm giá trị băm.

Ví dụ h(1234567890)=135, h(24123564)=021,....

Phương pháp cắt bỏ đơn giản trong tính toán nhưng thường phân bố không đều.

Phương pháp gấp

Cũng như trên, giả sử khóa là các số nguyên. Ta phân chia khóa thành các phần, sau đó kết hợp các phần đó bằng một cách nào đó (chẳng hạn dùng phép cộng hoặc phép nhân) để nhận các giá trị băm. Chẳng hạn khóa là các số nguyên có 10 chữ số, ta phân thành 4 nhóm, hai nhóm đầu mỗi nhóm gồm 3 chữ số, và hai nhóm sau mỗi nhóm gồm 2 chữ số. Việc kết hợp các nhóm được thực hiện bằng cách cộng các nhóm với nhau, sau đó cắt bỏ các số bên trái nếu vượt quá 3 chữ số.

Ví dụ với khóa x=1234567890 được biến đổi thành 123+456+78+90=774. Với khóa 9876543210 được biến đổi thành 987+654+32+10=1683, sau khi cắt bỏ phần đầu còn lại giá trị băm là 683.

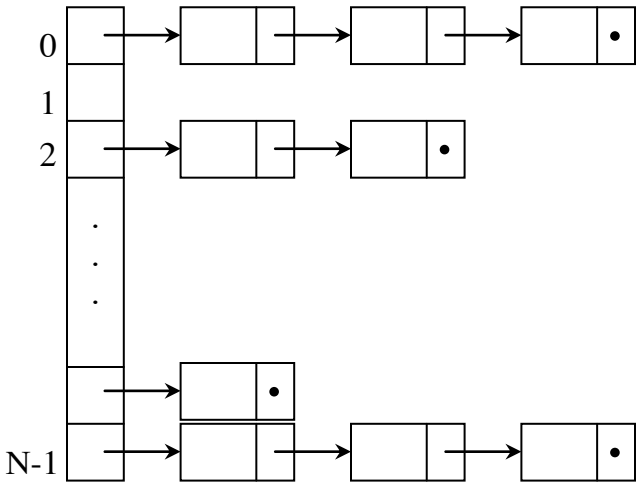
Vì mọi thông tin trong khóa đều được phản ánh vào giá trị băm, nên phương pháp gấp cho phân bố các khóa đều hơn phương pháp cắt bỏ.

Phương pháp lấy phần dư:

Với khóa là các số nguyên, giả sử cần chia tập hợp các phần tử thành *N* lớp. Ta lấy khóa chia cho *N* lấy phần dư làm giá trị băm. Hàm băm được xác định $h(x) = x \bmod N$. Tính phân bố đều của các khóa phụ thuộc nhiều vào việc chọn số *N*. Tốt nhất chọn *N* là các số nguyên tố. Chẳng hạn sau đây là một hàm băm để chia các khóa là xâu có 10 ký tự 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;
```

Cấu trúc bảng băm mở được minh họa như hình sau



Hình 5.1 Bảng băm mở

Khai báo:

```
Const N=...;

Type  Pointer = ^Element;
        Element = Record
                        Data:ElementType;
                        next:Pointer;
                  End;
        HashTable = Array[0..N-1] of Pointer;

Var     T : HashTable;
```

Các thao tác trên bảng băm mở

a/ Khởi tạo: Tạo một bảng băm rỗng.

```
Procedure Init(var T:HashTable);
var i:Integer;
   Begin
      For i:=0 to N-1 do T[i]:=nil;
   End;
```

b/ Kiểm tra một phần tử trong bảng theo khóa

Để tìm một phần tử có khóa *x* trong bảng băm mở *T* ta dựa vào kết quả giá trị băm của khóa *x*, *i=h*(*x*) để xác định vị trí của phần tử trong bảng. Sau đó duyệt trong danh sách liên kết T[i]. Chi tiết hàm tìm một khóa trong bảng như sau:

```
Function Member(x:KeyType; T:HashTable):Boolean;
var p: Pointer; found:Boolean;
   Begin
      p:=T[h(x)];
      found:=false;
      while (p<>nil) and not found do
         if p^.Data.Key=x then found:=true
         else p:=p^.next;
      Member:=found;
   End;
```

Nếu tập hợp có *n* phần tử và hàm băm phân bố đều các phần tử. Khi đó mỗi thành phần gồm *n*/*N* phần tử. Do đó thuật toán tìm một phần tử trong bảng băm mở có độ phức tạp là O(*n*/*N*).

c/ Thêm một phần tử vào bảng băm

```
Procedure Insert(x : ElelemtType; var T:HashTable);
var i:1..N-1; p:Pointer;
  Begin
    if not Member(x.Key,T) then
       begin
          i:=h(x);
          new(p);
          p^.data:=x;
          p^.next:=T[i];
          T[i]:=p;
       end;
  End;
```

d/ Xóa một phần tử trong bảng băm theo khóa

```
Procedure Delete(x:KeyType; var T:HashTable);
var i:0..N-1; p,q:Pointer; found:Boolean;
  Begin
    i:=h(x);
```

```
    if T[i]<>nil then
        if T[i]^.Data.Key=x then
            begin p:=T[i]; T[i]:=T[i]^.next; dispose(p); end
        else
            begin
                q:=T[i];
                p:=q^.next;
                found:=false;
                while (p<>nil) and not found do
                    if p^.Data.key=x then
                        begin
                            q^.next:=p^.next; found:=true;
                        end
                    else
                        begin
                            q:=p; p:=p^.next;
                        end;
                if found then dispose(p);
            end;
End;
```

20.1.2. 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..N-1). Khác với bảng băm mở, trong bảng băm đóng, mỗi phần tử của tập hợp được lưu trữ trong chính các thành phần T[i] của mảng. Do đó bảng băm đóng là một mảng các phần tử.

```
Type HashTable = Array[0..N-1] of ElementType;
```

Với cách tổ chức lưu trữ các phần tử như trên thì sẽ xảy ra trường hợp xung đột (collision) khi có hai phần tử e_1 và e_2 có khóa tương ứng là x_1 và x_2 mà $h(x_1) = h(x_2) = i$, vì khi đó hai phần tử e_1 và e_2 đều lưu ở vị trí thứ i trong mảng.

Để giải quyết sự xung đột thường dùng *phương pháp băm lại* (rehashing). Chiến lược băm lại như sau: ta sẽ lần lượt xét các vị trí $d_1(x)$, $h_2(x)$,... cho tới khi tìm được một vị trí trống để đặt phần tử x vào đó. Nếu không tìm được vị trí nào trống thì bảng đã đầy và ta không thể đưa x vào bảng. Trong đó $h_i(x)$ (i=1,2,...) là các giá trị băm lại lần thứ i, nó chỉ phụ thuộc vào khóa x. Sau đây là một số phương pháp băm lại.

Băm lại tuyến tính:

Các hàm $h_i(x)$ được xác định như sau:

$h_i(x) = (h(x) + i) \bmod N$.

Với cách băm lại như trên, mảng lưu các phần tử được xem như một mảng vòng tròn. Mỗi khi cần băm lại của khóa x sẽ xét các vị trí $h(x)$+1, $h(x)$+2, ...

Chẳng hạn, N=10 và các khóa *a, b, c, d, e* có các giá trị như sau: $h(a)$=7, $h(b)$=1, $h(c)$= 4, $h(d)$=3, $h(e)$=3. Xuất phát từ một bảng trống, các phần tử với khóa *a, b, c, d* được thêm vào bảng với các vị trí dựa vào hàm băm như hình dưới

	b		*d*	*c*	*e*		*a*		
0	1	2	3	4	5	6	7	8	9

Khi thêm phần tử khóa *e* vào sẽ xảy ra xung đột vì trùng giá trị băm với khóa *d*. Khi đó phải tính lại giá trị của hàm băm $h_1(e) = h(e)+1=4$ không thể đưa vào vị trí này vì tại vị trí này đã lưu khóa *c*. Tiếp tục với $h_2(e)$=5, vị trí này trống nên đưa *e* vào vị trí này.

Hạn chế cơ bản của phương pháp băm lại tuyến tính là các giá trị khóa sẽ được xếp liền vào sau các giá trị khóa ban đầu đã đưa vào bảng mà không xung đột. Do đó càng ngày các giá trị khóa trong bảng càng tụ lại thành các đoạn dài và giữa các đoạn lấp đầy là các khoảng trống. Vì vậy, việc tìm một vị trí trống trong bảng để đưa giá trị mới vào, càng về sau càng chậm.

Băm lại bình phương

Một cách tốt hơn phương pháp băm lại tuyến tính, tránh được sự tập trung của các giá trị khóa cùng giá trị băm là phương pháp băm bình phương, hàm băm lại được xác định như sau:

$h_i(x) = (h(x) + i^2) \bmod N$

Với cách băm lại này có một hạn chế là 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 không tìm được vị trí rỗng mặc dù trong bảng vẫn còn các vị trí trống.

Các phép toán trên bảng băm đóng

Để kiểm tra một khóa x trong bảng băm đóng T ta phải duyệt lần lượt các vị trí $h(x)$, $h_1(x)$, $h_2(x)$,... Giả sử các phần tử của bảng chưa bị xóa. Khi đó hoặc tìm được một vị trí trong bảng chứa khóa x, hoặc tìm được một vị trí trống đầu tiên $h_k(x)$. Trong trường hợp này kết luận rằng khóa x không có trong bảng. Tuy nhiên, sẽ phức tạp hơn nếu trong bảng đã thực hiện một số thao tác xóa. Trong trường hợp này, nếu tìm được một vị trí trống đầu tiên $h_k(x)$ thì chưa thể kết luận được x không có trong bảng vì có thể vị trí này lúc thêm phần tử khóa x đã có nhưng sau đó đã bị xóa. Do đó hoàn toàn có thể x nằm trong các vị trí $h_{k+1}(x)$, $h_{k+2}(x)$,...

Để đảm bảo rằng khi tìm ra vị trí trống đầu tiên $h_k(x)$ mà chắc chắn rằng bảng không chứa x, ta đưa vào bảng hai loại hằng deleted để gán cho vị trí của phần tử bị xóa và hằng empty dùng cho những vị trí trống thực sự. Mỗi khi xóa một phần tử thì vị trí này được gán hằng deleted. Khi cần thêm một phần tử mới vào bảng, có thể thêm vào vị trí đã xóa.

Khai báo:

```
const
        N=...;
        Empty = ...; {hằng xác định vị trí trống}
        Deleted = ...; {hằng xác định vị trí xóa}

Type
        HashTable = Array[0..N-1] of ElementType;

Var
        T : HashTable;
```

Với mỗi khóa *x*, các thao tác Inserst, Delete, Member trên bảng băm đóng đều thực hiện theo cách chung là tính giá trị băm $h(x)$, nếu tại vị trí này không đúng yêu cầu thì thăm dò lần lượt tại các vị trí $h_1(x)$, $h_2(x)$,... Điều đó được thực hiện bởi thủ tục Location.

Thủ tục Location trong trường hợp sử dụng hàm băm lại tuyến tính. Thủ tục này thực hiện phép thăm dò một giá trị khóa *x* trong bảng. Quá trình thăm dò được xuất phát từ vị trí $h(x)$, rồi lần lượt xem xét các vị trí $h_1(x)$, $h_2(x)$,... cho đến khi tìm được vị trí có chứa *x* hoặc tìm ra vị trí trống đầu tiên. Quá trình thăm dò cũng dừng lại khi đi qua toàn bộ bảng mà không thành công (không tìm được vị trí chứa *x* và cũng không tìm thấy vị trí trống). Vị trí mà tại đó quá trình thăm dò dừng lại được lưu vào tham biến *k*. Vị trí của phần tử được xóa hoặc vị trí trống đầu tiên được lưu vào tham biến *j* (nếu có).

```
Procedure Location(x:KeyType; var k,j:Integer);
var i:Integer;
   Begin
      i:=h(x);
      j:=i;
      if (T[i].Data.Key=x) or (T[i]=Empty) then k:=i
      else
         begin
            k:=(i+1) mod N;
            while (k<>i) and (T[k].Data.Key<>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]<>empty) then j:=k;
         end;
   End;
```

Cài đặt các thao tác trên bảng băm mở:

Hàm kiểm tra khóa *x* trong bảng

```
Function Member(x:KeyType; var T:HashTable)Boolean;
var k, j:Integer;
    Begin
        Location(x,k,j);
        if T[k].Data.key=x then Member:=True
        else Member:=false;
    End;
```

Thủ tục thêm phần tử *x* vào bảng

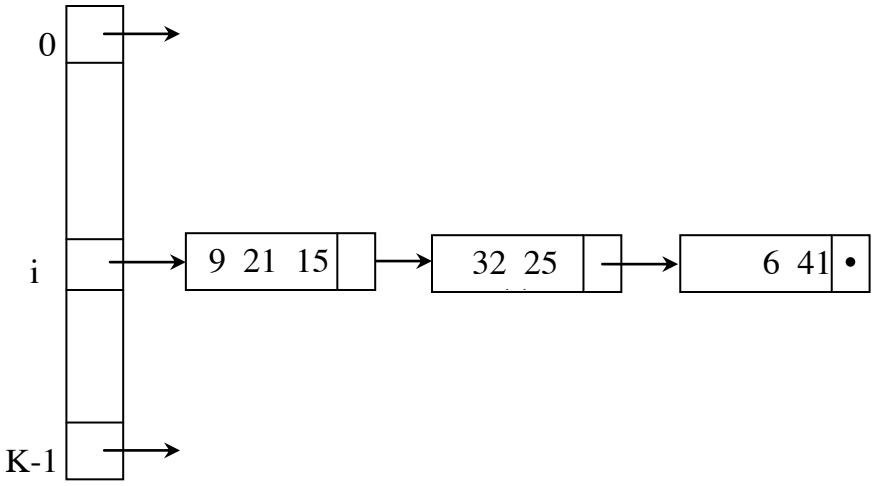
```
Procedure Insert(x:ElementType; var T:HashTable);
var k,j:Integer;
    Begin
        Location(x.Data.Key,k,j);
        if T[k].Data.Key<> x.Data.Key then
            if (T[j]=deleted)or(T[j]=empty) then T[j]:=x
            else writeln('Bảng đầy')
        else writeln('Khóa đã có);
    End;
```

Thủ tục xóa phần tử có khóa là *x*

```
Procedure Delete(x:KeyType; var T:HashTable);
var k,j:Integer;
    Begin
        Location(x,k,j);
        if T[k].Data.Key=x then T[k]:=deleted;
    End;
```

20.2. File Băm

Cấu trúc file băm hoàn toàn tương tự như cấu trúc bảng băm mở tổ chức cho bộ nhớ trong. Tư tưởng tổ chức file băm là chia tập hợp các bản ghi của file thành *K* lớp. Với mỗi lớp, tạo 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 5.2 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 khóa của các bản ghi và nhận các giá trị nguyên từ 0 đến *K*-1. Nếu *x* là một giá trị khóa và $h(x)=i$, $0 \le i \le K-1$, thì bản ghi với khóa *x* thuộc lớp thứ *i*.

Để tìm kiếm bản ghi có khóa *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 thứ *i*. Lần lượt đọc các khối, ta sẽ tìm được bản ghi với khóa *x* hoặc đọc hết các khối mà không thấy *x* nghĩa là bản ghi không có trong file.

Muốn thêm vào file bản ghi với khóa là x, ta cần kiểm tra xem khóa này đã có trong file hay chưa. Nếu chưa ta có thể thêm 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 khóa *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 xóa bỏ bản ghi này bằng cách cho bit xóa nhận giá trị 1.

Cấu trúc file băm là cấu trúc rất hiệu quả nếu các phép toán trên file chỉ đòi hỏi đến việc truy cập các bản ghi theo khóa. 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 nhanh hơn *k* lần so với file tuần tự.

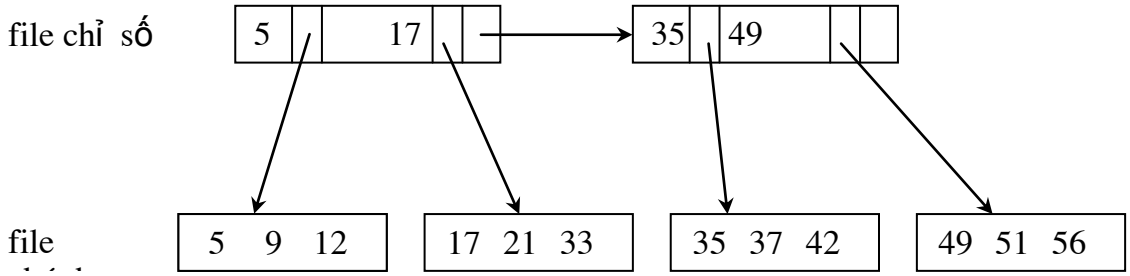
21. FILE CHỈ SỐ (INDEXED FILE)

Cấu trúc file băm được tạo ra dựa trên khóa của bản ghi. File chỉ số cũng được tổ chức dựa trên khóa của các bản ghi nhưng những khóa này được sắp xếp theo thứ tự tăng dần.

21.1. Tổ chức File chỉ số

Ta sắp xếp các bản ghi của file theo thứ tự khóa tăng dần vào một số khối cần thiết. Ta có thể sắp xếp các bản ghi vào một khối cho đến khi đầy khối. Song thông thường, trong mỗi khối để trống một vài vị trí để sau này có thể thêm vào dễ dàng hơn. Ta gọi file gồm các bản ghi chứa trong các khối là file chính. Ngoài ra ta tạo ra file chỉ số cho file chính như sau.

Chỉ số của một khối là cặp (v,b), trong đó b chỉ địa chỉ của khối, v là giá trị khóa nhỏ nhất của các bản ghi trong khối b. Từ các khối của file chính ta sẽ tạo ra file chỉ số, file này gồm các chỉ số khối của file chính. Các chỉ số khối được sắp xếp theo thứ tự tăng dần của khóa và một số khối cần thiết. Các khối này có thể được móc nối với nhau tạo thành một danh sách liên kết các khối, các khối chứa các chỉ số khối của file chính. Một cách khác có thể sử dụng một bảng để lưu trữ địa chỉ của các khối trong file chỉ số. Hình 5.3 minh họa cấu trúc của file chỉ số.



Hình 5.3 Cấu trúc file chỉ số

21.2. Các thao tác trên file chỉ số

Tìm kiếm

Giả sử cần tìm bản ghi x với khóa v cho trước. Trước hết ta cần tìm trên file chỉ số một cặp chỉ số (v_1, b_1) sao cho v_1 là giá trị khóa lớn nhất trong file chỉ số thỏa mãn điều kiện $v_1 \le v$. Khi đó ta gọi v_1 là phủ của v.

Việc tìm kiếm trên file chỉ số một giá trị v_1 phủ giá trị khóa v cho trước có thể thực hiện bằng cách tìm kiếm tuần tự hoặc tìm nhị phân.

Trong tìm 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 khóa của bản ghi đầu tiên trong file chỉ số thì điều đó có nghĩa là trong file chỉ số không chứa giá trị khóa phủ v.

Phương pháp hiệu quả hơn là tìm kiếm nhị phân. Giả sử các bản ghi của file chỉ số sắp xếp vào các khối được đánh chỉ số từ 1 đến m. Xét khối thứ m div 2. Giả sử (v_2, b_2) là bản ghi đầu tiên trong khối. So sánh giá trị khóa cho trước v với giá trị khóa v_2. Nếu $v<v_2$ ta tiến hành tìm kiếm trên các khối 1, 2, ..., m div 2-1. Nếu $v \ge v_2$, ta tiến hành tìm kiếm trên các khối m div 2, m div 2+1, ...,m. Quá trình trên được lặp lại cho đến khi ta tìm trên một khối. Lúc này ta so sánh lần lượt các giá trị khóa v với các giá trị khóa chứa trong khối này.

Để tìm bản ghi x với khóa 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 tìm bản ghi có khóa v. Nếu trên file chỉ số không chứa giá trị khóa v_1 phủ v, thì file chính không chứa bản ghi có khóa v.

Thêm vào

Giả sử cần thêm vào file bản ghi r với khóa là v. Giả sử file chính được chứa trong các khối $B_1, B_2, ..., B_k$ và các giá trị khóa của các bản ghi trong khối B_i nhỏ hơn các giá trị khóa trong khối B_{i+1}.

Trước hết cần tìm khối B_i để chứa bản ghi r vào đó. Sử dụng thủ tục tìm kiếm trên file chỉ số để tìm ra cặp chỉ số (v_1, b_1), với v_1 phủ v. Nếu tìm thấy thì B_i là khối có địa chỉ là b_1, ngược lại B_i là khối đầu tiên B_1.

Sau khi đã tìm được B_i, trong trường hợp B_i chưa đầy và bản ghi r chưa có trong khối B_i thì ta lưu r vào đúng vị trí của nó trong khối, tức là phải đảm bảo trật tự tăng của các khóa.

Nếu B_i là khối đầu tiên thì bản ghi r thêm vào vị trí đầu tiên của khối B_1, do đó cần phải sửa đổi chỉ số của khối B_1 trong file chỉ số.

Trong trường hợp B_i đã đầy, ta tiến hành thêm bản ghi r vào vị trí thích hợp trong khối B_i. Khi đó khối này sẽ thừa ra một bản ghi, ta sẽ thêm bản ghi này vào khối tiếp theo B_{i+1} (nếu được). Địa chỉ của khối B_{i+1} được xác định bằng cách tìm trong file chỉ số. Nếu khối B_{i+1} chưa đầy thì ta thêm phần tử thừa của khối B_i vào vị trí đầu tiên của khối B_{i+1} rồi cập nhật lại file chỉ số. Trong trường hợp khối B_{i+1} đầy hoặc không có khối B_{i+1} (khi $i=k$) thì thêm vào file chính một khối mới và lưu bản ghi thừa của khối B_i vào vị trí đầu tiên của khối mới, đồng thời bổ sung chỉ số của khối mới vào file chỉ số.

Loại bỏ một phần tử

Để loại bỏ bản ghi r với khóa v, đầu tiên ta dùng thủ tục tìm kiếm để xác định vị trí của bản ghi trong file. Sau đó tiến hành xóa r bằng cách đặt lại giá trị của bit nhận dạng phần tử bị xóa.

Sửa đổi

Giả sử ta cần sửa đổi bản ghi với khóa v. Nếu các giá trị không thuộc trường khóa, thì ta chỉ cần áp dụng thủ tục tìm kiếm để tìm ra bản ghi cần sửa và tiến hành các thao tác sửa đổi cần thiết. Nếu dữ liệu sửa liên quan đến khóa thì thực hiện thao tác kép: xóa bản ghi cũ và thêm bản ghi mới.

22. B-CÂY

Mục đích của chúng ta là tìm cách tổ chức dữ liệu trên file sao cho các thao tác được thực hiện hiệu quả, tức là các thao tác truy cập khối được thực hiện ít nhất có thể được khi thực hiện các thao tác tìm, thêm, xóa, cập nhật. Trong chương trước chúng ta đã xem xét cấu trúc cây nhị phân, cây tìm kiếm nhị phân và cây tìm

kiếm tổng quát. Những câu trúc này được lưu trữ ở bộ nhớ trong. Với dữ liệu lưu ở bộ nhớ ngoài một cấu trúc dạng cây tổng quát được dùng hiệu quả đó là B-Cây.

22.1. Khái niệm B-Cây

B-cây cấp *m* là cây tìm kiếm *m* nhánh thỏa mãn các tính chất sau:

1. Tất cả các nút lá đều nằm ở cùng một mức.
2. Nút gốc có ít nhất hai con và nhiều nhất *m* con.
3. Tất cả các nút bên trong của cây khác nút gốc có ít nhất *m*/2 cây con và nhiều nhất *m* cây con.

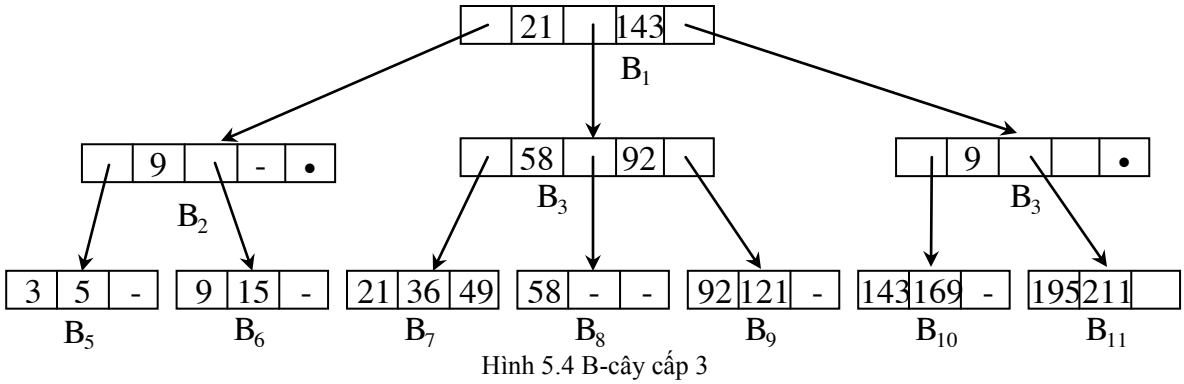
Tư tưởng của việc tổ chức file dưới dạng B-cây như sau. Ta sắp xếp các bản ghi của file vào một số khối cần thiết, mỗi khối này là lá của B-cây. Trong mỗi khối, các bản ghi được xếp theo thứ tự tăng dần của khóa. Các chỉ số của các khối này được sắp xếp vào một số khối mới. Trong mỗi khối chỉ số này, các chỉ số được sắp theo thứ tự tăng dần của khóa. Trong B-cây, các khối này sẽ là các đỉnh ở mức trên của mức các lá. Lại lấy chỉ số của các khối vừa tạo ra xếp vào một số khối mới. Các khối này lại là đỉnh ở mức trên của mức từ đó chúng được tạo ra. Quá trình trên tiếp tục cho đến khi chỉ số có thể xếp được trong một khối. Khối này là gốc của B-cây hay còn gọi là đỉnh của B-cây.

Như vậy, mỗi đỉnh của B-cây là một khối. Mỗi đỉnh trong của B-cây có dạng

$$(p_0, v_1, p_1, v_2, ..., v_n, p_n)$$

trong đó $v_1 < v_2 < ... < v_n$, và (v_i , p_i), với $0 \le i \le n$, là chỉ số của một khối, tức là v_i là giá trị khóa nhỏ nhất trong khối, p_i là con trỏ trỏ tới khối chứa khóa nhỏ nhất v_i, tức là con trỏ trỏ tới đỉnh con thứ i của đỉnh trong đang nói tới. Cần chú ý rằng, giá trị của khóa v_0 không được lưu giữ ở mỗi đỉnh trong để tiết kiệm bộ nhớ.

Ví dụ hình 5.4 biểu diễn một B-cây cấp 3. B-cây được tạo thành từ 11 khối được đánh số B_1, B_2, ..., B_{11}. Mỗi khối là đỉnh trong chứa được 3 chỉ số. Mỗi khối lá chứa 3 bản ghi (ở đây là các số nguyên). File ở đây là file các số nguyên được lưu giữ ở các khối từ B_5 đến B_{11}.



Hình 5.4 B-cây cấp 3

22.2. Các thao tác trên B-Cây

Tìm kiếm

Giả sử chúng ta cần tìm bản ghi r với khóa v cho trước. Xuất phát từ gốc của B-cây và tìm đường đến lá sao cho lá này chứa bản ghi r nếu nó có trong file.

Trong quá trình tìm kiếm, giả sử tại một thời điểm nào đó ta đang xét đỉnh B của B-cây. Nếu khối B là lá thì duyệt trong khối B để tìm bản ghi r với khóa v (có thể dùng kỹ thuật tìm kiếm tuần tự hoặc nhị phân do các khóa được sắp thứ tự).

Nếu B là một đỉnh trong chứa bộ (p_0, v_1, p_1, v_2, ..., v_n, p_n) thì ta cần phải xác định vị trí của khóa v trong dãy khóa v_1, v_2, ..., v_n. Nếu $v<v_1$ thì ta xuống đỉnh được trỏ bởi p_0. Nếu tìm được vị trí i sao cho $v_i \le v < v_{i+1}$ thì ta đi xuống đỉnh được trỏ bởi p_i (với i=1,2,..,n). Trường hợp $v_n<v$ thì đi xuống đỉnh được trỏ bởi p_n.

Thêm vào

Giả sử cần thêm vào B-cây một bản ghi r với khóa v.

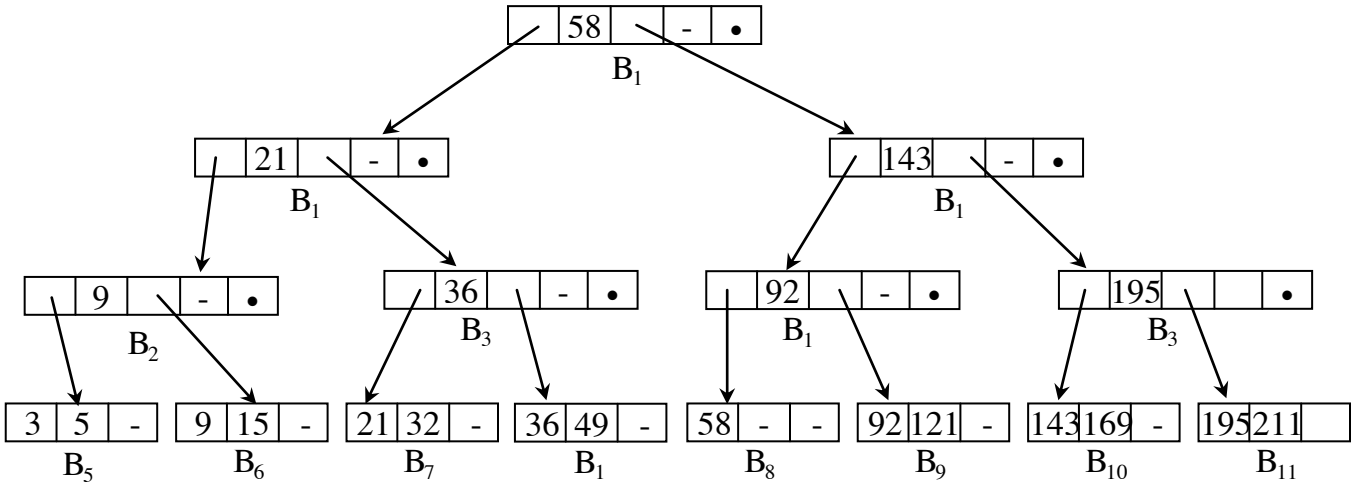
Đầu tiên áp dụng thủ tục tìm kiếm để tìm ra khối B cần phải thêm bản ghi r vào đó.

Nếu khối B còn đủ chỗ cho bản ghi r thì ta thêm bản ghi r vào khối B sao cho thứ tự tăng dần của các khóa được duy trì. Chú ý rằng r không thể là bản ghi đầu tiên của khối B, trừ khi B là lá ngoài cùng bên trái. Nếu B là lá ngoài cùng bên trái thì giá trị khóa nhỏ nhất trong khối 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 khối B là xong, không cần phải sửa đổi gì đối với các đỉnh là tiền thân của khối B.

Nếu khối B không còn đủ chỗ để lưu bản ghi r thì ta thêm vào B-cây một khối mới B' là lá. Chuyển một nửa số bản ghi ở cuối của khối B sang khối B'. Sau đó thêm bản ghi r vào khối B hay B' sao cho vẫn đảm bảo thứ tự tăng của các khóa. Giả sử Q là cha của B, ta có thể biết được Q trong quá trình tìm kiếm ta lưu lại vết của đường đi từ gốc tới B. Giả sử chỉ số của khối B' là (v',p'), trong đó v' là giá trị khóa nhỏ nhất trong B', còn p' là địa chỉ của khối B'. Tương tự trên ta thêm cặp (v',p') vào khối Q. Nếu khối Q không còn đủ chỗ thì thêm vào B-cây một đỉnh mới Q' là em liền kề của Q. Sau đó phải tìm đến cha của Q để đưa thêm vào chỉ số của khối mới Q'. Quá trình tiếp tục, nếu đến nút gốc mà không đủ chỗ để thêm một vị trí thì thêm một khối mới và chuyển một nửa các phần tử cuối của gốc vào khối này. Trong trường hợp này phải tạo một gốc mới có đúng 2 con, một con là gốc cũ, một con là đỉnh mới đưa vào.

Ví dụ cần thêm vào cây ở hình 5.4 bản ghi có khóa 32. Trước hết cần tìm khối để đưa bản ghi này vào. Bắt đầu từ gốc B_1, vì 21 < 32 < 143 nên ta đi xuống B_3. Tại B_3 ta có 32<58, nên ta đi xuống B_7. B_7 là lá do đó phần tử cần thêm sẽ bổ sung vào khối này nhưng khối B_7 đang đầy. Ta thêm vào khối mới B_{12}, và xếp các bản ghi với các khóa 21, 32 vào khối B_7, xếp các bản ghi với khóa 36, 49 vào khối

B_{12}. Chỉ số của khối B_{12} chứa giá trị khóa 36. Cần phải xếp chỉ số của khối B_{12} vào cha của B_7 là B_3. Nhưng do B_3 cũng đầy nên ta lại thêm vào khối mới B_{13}. Sau đó các chỉ số của các khối B_7, B_{12} được xếp vào B_3, còn chỉ số của các khối B_8, B_9 được xếp vào B_{13}. Tiếp tục chỉ số của khối B_{13} là 58 và địa chỉ của nó phải được lưu vào khối B_1 nhưng khối này đầy nên phải thêm khối mới B_{14}. Các chỉ số của các khối B_2 và B_3 được lưu vào B_1, các chỉ số của B_{13}, B_4 lưu vào B_{14}. Khối B_{15} được tạo mới để làm nút gốc mới cho cây, tại khối này chứa chỉ số của khối B_1 và B_{14}. Kết quả sau khi thêm bản ghi có khóa 32 được minh họa trong hình 5.5



Hình 5.5 B-cây của hình 5.4 sau khi thêm bản ghi khóa 32

Loại bỏ

Giả sử cần loại khỏi B-cây bản ghi r có khóa v. Đầu tiên sử dụng thủ tục tìm kiếm để tìm ra lá B chứa bản ghi r. Sau đó loại bỏ bản ghi r trong khối B.

Sau khi loại bản ghi, nếu khối B không rỗng và r không phải là bản ghi đầu tiên của khối thì thao tác xóa đã hoàn thành. Trong trường hợp r là bản ghi đầu tiên của B và sau khi xóa B chưa rỗng thì sau khi xóa chỉ số của khối B đã thay đổi nên phải tìm đến nút Q là cha của B. Nếu B là con trưởng của Q thì giá trị khóa v' trong chỉ số (v',p') của B không có trong Q. Trong trường hợp này cần tìm đến tiền thân A của B sao cho A không phải là con trưởng của cha mình A'. Khi đó giá trị khóa nhỏ nhất trong B được chứa trong A'. Do đó trong A' ta cần thay giá trị khóa cũ v bởi giá trị khóa mới v'.

Trường hợp sau khi loại bỏ bản ghi r, khối B trở thành rỗng. Khi đó ta loại bỏ khối B khỏi B-cây. Điều này dẫn đến cần loại bỏ chỉ số đỉnh cha Q của B.

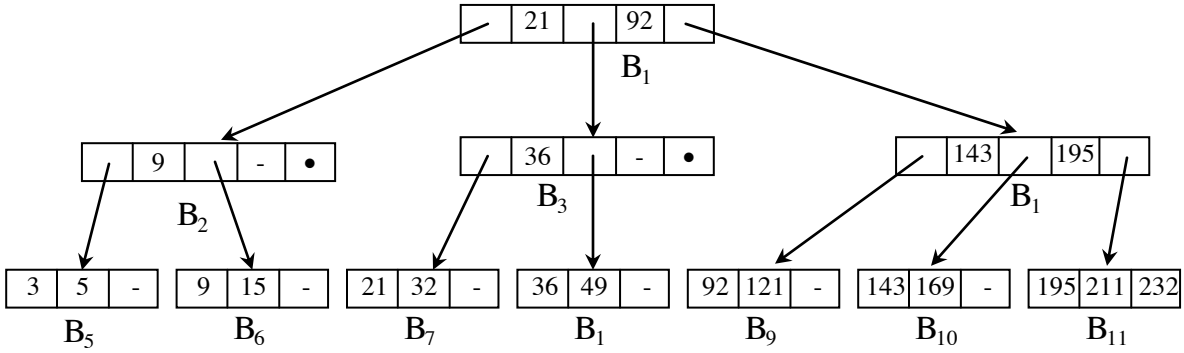
Nếu sau khi loại bỏ số các con của đỉnh Q ít hơn m/2 thì ta tìm đến đỉnh Q' là anh em liền kề của đỉnh Q. Nếu Q có nhiều hơn m/2 con thì ta phân phối lại các giá trị khóa trong Q và Q' sao cho cả hai có ít nhất m/2 con. Khi đó các chỉ số của Q hoặc Q' có thể thay đổi. Ta lại phải tìm đến tiền thân của Q để phản ánh sự thay đổi này.

Nếu Q' có đúng m/2 con, thì ta kết hợp hai đỉnh Q và Q' thành một đỉnh, một trong hai đỉnh bị loại khỏi cây, các khóa 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 của cây, khi đó ta cần kết hợp hai con gốc thành một đỉnh và đỉnh này trở thành gốc mới của B-cây.

Ví dụ cần xóa bản ghi khóa 58 khỏi B-cây trong hình 5.5. Đầu tiên tìm lá chứa khóa 58, đó là khối B_8. Xóa bản ghi 58, khối B_8 thành khối rỗng. Ta phải tìm đến cha của B_8 là B_{13} và loại bỏ chỉ số của B_8 trong B_{13}, B_{13} sau khi xóa chỉ còn 1 con, ít hơn $m/2$. Tìm đến em liền kề của B_{13} là B_4, số con của B_4 và B_{13} có thể ghép vào một khối B_{13}. Cần loại bỏ chỉ số của khối B_4 khỏi B_{14}. B_{14} lúc này chỉ có một con. Tìm đến anh liền kề của B_{14} là B_1, có thể gộp hai khối này thành một khối B_1, và B_1 cũng trở thành gốc của cây. Hình 5.5 là kết quả của cây sau khi xóa bản ghi có khóa 58.



Hình 5.6 B-Cây ở hình 5.5 sau khi xóa bản ghi khóa 58

23. BÀI TẬP

Bài 1. Dùng bảng băm với 11 vị trí và hàm băm $h(i)=i$ mod 11. hãy chỉ ra bảng băm nhận được sau khi chèn các số nguyên sau theo thứ tự: 26, 42, 5, 44, 92, 59, 40, 36, 12, 60, 80.

a) Dùng bảng băm đóng với phương pháp băm lại tuyến tính để xử lý xung đột.

b) Dùng bảng băm đóng với phương pháp băm lại bình phương để xử lý xung đột.

c) Dùng bảng băm mở.

Bài 2. Giả sử các ký tự được mã hóa như sau: 'A'=1, 'B'=2,...,'Y'=25, 'Z'=26. Dùng một bảng băm với 11 vị trí và hàm băm h(s:string) là trung bình của các mã số ký tự đầu và ký tự cuối trong s. Hãy chỉ ra bảng băm nhận được khi chèn các định danh sau theo thứ tự: BETA, RATE, FREQ, ALPHA, MEAN, SUM, NUM, BAR, WAGE, PAY, KAPPA.

a) Dùng bảng băm đóng với phương pháp băm lại tuyến tính để xử lý xung đột.

b) Dùng bảng băm mở.

Bài 3. Phương pháp băm kép để xử lý xung đột được mô tả như sau. Nếu mục i xung đột với một mục khác trong bảng tại vị trí a, dùng hàm băm thứ hai h_2 cho mục ấy để xác định $k=h_2(i)$. Sau đó thực hiện như trong băm lại tuyến tính, nhưng thay vì thử các vị trí liên tiếp, ta thử các phần tử của bảng theo thứ tự a, $a+k$, $a+2k$,... được chia theo modun n (n là kích thước của bảng). Hãy chỉ ra bảng băm nhận được khi dùng các số ở bài tập 1, nhưng dùng phương pháp băm kép để giải quyết xung đột với hàm băm thứ cấp xác định như sau:

$$h_2(i) = \begin{cases} 2i \bmod 11, & \text{nếu } i \text{ khác } 0 \\ 1, & \text{nếu } i = 0 \end{cases}$$

Bài 4. Giả sử rằng các số nguyên trên đoạn 1..100 được lưu trữ trong bảng băm bằng cách dùng hàm băm $h(i) = i \bmod n$ (n là kích thước bảng băm). Hãy viết chương trình tạo các số nguyên ngẫu nhiên trên đoạn này và chèn chúng vào bảng băm cho đến khi xảy ra xung đột.

Bài 5. Vẽ hình ảnh của B-cây bậc 3 nhận được khi chèn các ký tự sau đây theo thứ tự: C, O, R, N, F, L, A, K, E, S.

Bài 6. Vẽ hình ảnh của B-cây bậc 3 nhận được khi chèn các số nguyên sau đây theo thứ tự: 261, 381, 385, 295, 134, 400, 95, 150, 477, 291, 414, 240, 456, 80, 25, 474, 493, 467, 349, 180, 370, 257.

Bài 7. Tổ chức dữ liệu kiểu B-cây để lưu các từ. Viết chương trình đọc các từ từ một tệp văn bản và lập một B-cây để lưu trữ những từ này. Chương trình cũng cho phép tìm một từ trong B-cây đã có.

Chương 6

SẮP XẾP

Sắp xếp là là quá trình tổ chức lại một danh sách các đối tượng theo một thứ tự nhất định. Mục đích của sắp xếp là giúp cho việc tìm kiếm trên danh sách các đối tượng được dễ dàng hơn. Vì vậy sắp xếp là một bài toán cơ bản và được ứng dụng khá rộng rãi trong các bài toán tin học.

Người ta chia các phương pháp sắp xếp thành hai lớp: sắp xếp trong và sắp xếp ngoài. Sắp xếp trong được thực hiện trên danh sách các đối tượng được lưu ở bộ nhớ trong của máy tính dưới dạng mảng. Do đó phương pháp này còn gọi là sắp xếp mảng. Khi số các đối tượng trong danh sách cần sắp xếp quá lớn, không thể lưu ở bộ nhớ trong được thì phải lưu ở bộ nhớ ngoài dưới dạng các file. Khi đó phương pháp sắp xếp trong không thể sử dụng cho các dữ liệu lưu ở bộ nhớ ngoài vì có những khác biệt cơ bản về các thao tác truy cập các đối tượng. Do đó phải dùng phương pháp khác gọi là sắp xếp ngoài.

24. CÁC THUẬT TOÁN SẮP XẾP TRONG

Giả sử rằng các đối tượng cần được sắp xếp được lưu trong một mảng. Cần sắp xếp danh sách các đối tượng theo trật tự tăng (hay giảm) của một trường Key của đối tượng. Trường này gọi là khóa sắp xếp.

Bài toán sắp xếp trong được phát biểu như sau: Cho mảng A[1..n] các đối tượng. Cần sắp xếp lại các phần tử của mảng để nhận được mảng A mới với các đối tượng có các giá trị khóa tăng dần:

$$A[1].Key \le A[2].Key \le ... \le A[n].Key$$

Giả sử ta có khai báo cho mảng các đối tượng cần được sắp xếp như sau:

```
Type ElementType =   Record
                        Key : KeyType;
                        Các trường khác ;
                     End;

Var      A : Array[1..n] Of ElementType;
```

24.1. Sắp xếp bằng cách chọn trực tiếp

Tư tưởng của thuật toán chọn trực tiếp là tìm phần tử có khóa nhỏ nhất trong danh sách đổi chỗ với phần tử đầu. Sau mỗi bước như vậy đã đặt được một phần tử vào đúng vị trí của danh sách cần sắp xếp do đó ta chỉ cần sắp xếp danh sách với các phần tử còn lại với cách làm tương tự.

Thuật toán chọn trực tiếp:

1. Xuất phát i=1

2. Tìm vị trí k sao cho A[k] là phần tử có khóa nhỏ nhất trong danh sách A[i..n]

3. Đổi chỗ hai phần tử A[i] và A[k]

4. Tăng i lên 1

5. Lặp lại bước 2 khi i<n

Thủ tục sắp xếp bằng cách chọn trực tiếp như sau:

```
Procedure StraightSelection;
var  i,j,k:integer; tmp:ElementType;
Begin
  For i:=1 to n-1 do
    begin
      k:=i;
      for j:=i+1 to n do
         if a[j].Key<a[k].Key then k:=j;
      tmp:=a[i];a[i]:=a[k];a[k]:=tmp;
    end;
End;
```

24.2. Sắp xếp bằng cách đổi chỗ trực tiếp

Thuật toán sắp xếp bằng cách đổi chỗ trực tiếp được thực hiện bằng cách duyệt các phần tử trong mảng, nếu hai phần tử liền nhau sai vị trí thì đổi chỗ cho nhau. Sau mỗi lần duyệt phần tử nhỏ nhất được đưa lên đầu (hoặc phần tử lớn nhất đưa về cuối tùy thuộc vào cách duyệt từ 1 đến n hay ngược lại). Sau mỗi bước lặp danh sách các phần tử cần sắp xếp sẽ giảm đi 1.

Thuật toán sắp xếp bằng đổi chỗ trực tiếp:

1. Xuất phát i=2

2. Duyệt danh sách A[i..n] theo thứ tự ngược từ n đến i, bước thứ j thực hiện

   Nếu a[j].Key<a[j-1].Key thì đổi chỗ hai phần tử ở vị trí j và j-1.

3. Tăng i lên 1

4. Lặp lại bước 2 khi i<n.

Thủ tục sắp xếp bằng đổi chỗ trực tiếp như sau:

```
Procedure BubbleSort;
var i,j:Integer; tmp:ElementType;
Begin
   for i:=2 to n do
     for j:=n downto i do
      if a[j].Key<a[j-1].Key then
        begin
          tmp:=a[j];
          a[j]:=a[j-1];
          a[j-1]:=tmp;
        end;
End;
```

Ta có thể cải tiến thuật toán trên bằng cách ghi nhận lại sự thay đổi sau mỗi lần duyệt. Khi có lần duyệt nào không đổi chỗ các phần tử thì danh sách đã sắp xếp. Đồng thời ta cũng có thể lưu vị trí *k* của phần tử đổi chỗ cuối cùng. Bởi vì mọi phần tử cạnh nhau trước vị trí k này đã được sắp thứ tự nên lần duyệt sau ta chỉ cần duyệt tới vị trí *k* mà không cần tới *i*.

```
Procedure ShakeSort;
var i,j,l,r,k:Integer; tmp:ElementType;
Begin
    l:=2; r:=n;k:=n;
     while l<=r do
     begin
       for j:=r downto l do
       if a[j].Key<a[j-1].Key then
          begin
            tmp:=a[j];
            a[j]:=a[j-1];
            a[j-1]:=tmp;
            k:=j;
          end;
        l:=k+1;
        for j:=l to r do
          if a[j].Key<a[j-1].Key then
             begin
               tmp:=a[j];
               a[j]:=a[j-1];
               a[j-1]:=tmp;
               k:=j;
             end;
           r:=j-1;
    end;
     for j:=n downto i do
       if a[j].Key<a[j-1].Key then
          begin
            tmp:=a[j];
            a[j]:=a[j-1];
            a[j-1]:=tmp;
          end;
End;
```

24.3. Sắp xếp bằng cách chèn trực tiếp

Giả sử đoạn đầu của mảng A[1..i-1] (i>=2) đã được sắp xếp. Khi đó để sắp xếp danh sách ta chỉ cần chèn lần lượt từng phần tử A[i] vào vị trí thích hợp trong danh sách con A[1..i-1] để nhận được một danh sách con A[1..i] được sắp xếp.

Việc tìm vị trí thích hợp để chèn A[i] vào danh sách con A[1..i-1] được thực hiện như sau:

+ Đầu tiên lưu A[i] vào biến x.

+ Sau đó cho biến j chạy từ i-1, nếu x.Key < A[j].Key thì đẩy A[j] ra sau một vị trí và giảm j.

+ Quá trình tiếp tục cho đến khi ta có x.Key>A[j].Key hoặc j=0.

+ Đặt x vào vị trí j+1 của mảng.

Thuật toán chèn trực tiếp:

1. Xuất phát i=2
2. x:=A[i]
3. Tìm vị trí k ở bên phải nhất trong danh sách A[1..i] sao cho A[k].Key<x.Key
4. Chuyển các phần tử từ vị trí k+1 đến vị trí i ra sau một vị trí
5. Đưa x vào vị trí k+1.
6. Tăng i lên 1
7. Lặp lại bước 2 khi i<n

Thủ tục sắp xếp danh sách bằng cách chèn trực tiếp.

```
Procedure StraightInsertion;
var i,j: Integer; x : ElementType;
Begin
  for i:=2 to n do
    begin
      x:=a[i]; j:=i-1;
      while (a[j].Key>x.Key) and (j>0) do
        begin
          a[j+1]:=a[j];
          j:=j-1;
        end;
      a[j+1]:=x;
    end;
End;
```

Ta có thể cải tiến thuật toán trên bằng cách dùng tư tưởng của thuật toán tìm nhị phân để tìm vị trí cần chèn phần tử A[i] vào dãy con đã sắp xếp A[1..i-1].

Thủ tục sắp xếp bằng chèn trực tiếp dùng thuật toán tìm nhị phân.

```
Procedure BinaryInsertion;
var i,j, l,r,m : Integer; x:ElementType;
Begin
  for i:=2 to n do
    begin
      x:=a[i]; l:=1;r:=i-1;
      while (l<=r) do
        begin
          m:=(l+r) div 2;
```

```
            if a[m].Key<=x.Key then l:=m+1
            else r:=m-1;
          end;
        for j:=i-1 downto l do a[j+1]:=a[j];
        a[l]:=x;
      end;
End;
```

24.4. Sắp xếp với độ dài bước giảm dần

Một cải tiến của phương pháp chèn trực tiếp đã được D.L.Shell đưa ra năm 1959. Phương pháp chèn với độ dài bước giảm dần được thực hiện dựa trên ý tưởng chia danh sách thành các dãy con gồm các phần tử cách nhau *h* vị trí. Tiến hành sắp xếp các phần tử trong cùng dãy con (dùng thuật toán chèn trực tiếp) sẽ làm cho các phân tử được đưa về vị trí đúng tương đối. Sau đó giảm khoảng cách *h* để sắp xếp trên các dãy con mới và tiếp tục cho đến khi *h*=1, lúc này đảm bảo tất cả các phần tử đều được so sánh với nhau. Yếu tố quyết định tính hiệu quả của thuật toán là chọn các khoảng cách *h* trong từng bước sắp xếp và số bước cần sắp xếp. Tuy nhiên cho đến nay vẫn chưa có tiêu chuẩn rõ ràng trong việc lựa chọn dãy giá trị khoảng cách tốt nhất. Một số dãy được Knuth đề nghị:

$h_i = (h_{i-1} - 1)/3$ và $h_k = 1$, $k = \log_3 n - 1$

$h_i = (h_{i-1} - 1)/2$ và $h_k = 1$, $k = \log_2 n - 1$

Thuật toán sắp xếp với bước giảm dần:

1. Chọn k khoảng cách h[1], h[2], ..., h[k];
2. Xuất phát i = 1;
3. Phân chia dãy ban đầu thành các dãy con cách nhau h[i] khoảng cách. Sắp xếp từng dãy con bằng phương pháp chèn trực tiếp.
4. i = i+1
5. Nếu i > k thì dừng, ngược lại thì lặp lại bước 3.

Thủ tục ShellSort sắp xếp danh sách theo bước giảm dần như sau:

```
Procedure ShellSort;
var  h:array[1..3] of integer;
    m,k,i,j,len:integer;x:ElementType;
Begin
   m:=3;
   h[1]:=5;h[2]:=3;h[3]:=1;
   for k:=1 to m do
     begin
       len:=h[k];
       for i:=len+1 to n do
         begin
           x:=a[i];
           j:=i-len;
           while (x.Key<a[j].Key) and (j>0) do
```

```
            begin
               a[j+len]:=a[j];
               j:=j-len;
            end;
         a[j+len]:=x;
      end;
   end;
End;
```

24.5. Sắp xếp trộn

Để sắp xếp danh sách các phần tử A[i..j] ta đưa về việc sắp xếp hai danh sách con A[i..k] và A[k+1..j], với *k* nằm giữa *i* và *j* (k=(i+j)div2)). Sau đó trộn hai danh sách con đã được sắp xếp thành danh sách được sắp xếp.

Thủ tục có thể viết hình thức như sau:

```
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);
      end;
End;
```

trong đó Merge(i,k,j) là thủ tục trộn hai danh sách đã sắp xếp A[i..k] và A[k+1..j]. Thủ tục này sẽ được xem như bài tập.

Độ phức tạp trung bình của thuật toán MergeSort được đánh giá là $O(n\log_2 n)$.

24.6. Sắp xếp kiểu vun đống

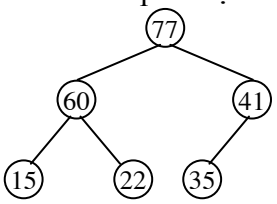
Phương pháp sắp xếp chọn trực tiếp dựa vào nguyên tắc lặp lại việc chọn khoá nhỏ nhất (hoặc lớn nhất) trong *k* phần tử nên số lần so sánh là *k*-1. Thao tác này có thể cải tiến để hạn chế số thao tác so sánh nếu có một cách tổ chức dữ liệu hợp lý. Để cải thiện phương pháp này, mỗi lần duyệt tìm phần tử có khoá nhỏ nhất cũng giữ lại các thông tin khác về thứ tự các phần tử để bước sau tìm được thuận lợi hơn. Để đáp ứng được yêu cầu này, cấu trúc Heap là một lựa chọn tốt vì mỗi khi chọn phần tử có khoá nhỏ nhất (hay lớn nhất) và điều chỉnh lại Heap được đánh giá trong trường hợp xấu nhất là $O(\log_2 n)$. Các phần tử cần sắp xếp được tổ chức như một Heap với khoá so sánh trên Heap chính là khoá sắp xếp các phần tử.

Giả sử cần sắp xếp một danh sách gồm *n* phần tử theo thứ tự tăng của trường khoá sắp xếp. Ta xem danh sách *n* phần tử là một cây nhị phân đầy đủ, dùng thủ tục CreateHeap để tạo một Heap từ n phần tử. Sau đó dùng phương pháp sắp xếp kiểu chọn trực tiếp bằng cách chọn phần tử có khoá lớn nhất trong *n* phần

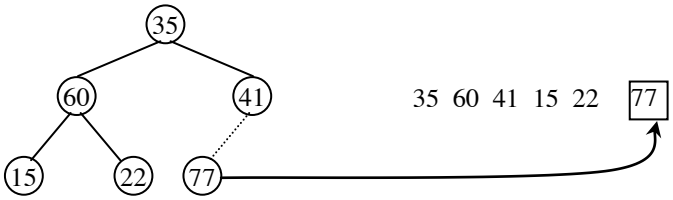
tử lúc này là phần tử ở gốc của cây nhị phân hay ở vị trí đầu tiên của mảng. Tiến hành đổi phần tử này với phần tử cuối cùng trong mảng (ở bước này là phần tử thứ *n*). Khi đó phần tử thứ *n* đã được xếp đúng vị trí nên chỉ cần sắp xếp cho *n*-1 phần tử đầu tiên của mảng. Dễ thấy dãy *n*-1 phần tử đầu dãy có thể không phải là một Heap, nhưng các cây con trái và phải của nút gốc (không xét phần tử vị trí *n*) đã là các Heap nên dùng thủ tục SiftDown sẽ đưa *n*-1 phần tử đầu dãy thành một Heap. Lặp lại các thao tác trên cho dãy *n*-1 phần tử cho đến khi dãy chỉ còn đúng 1 phần tử thì dừng. Lúc này mảng các phần tử đã được sắp theo thứ tự tăng của khoá.

Xét một ví dụ cụ thể là sắp xếp dãy số sau theo thứ tự tăng: 35, 15, 77, 60, 22, 41.

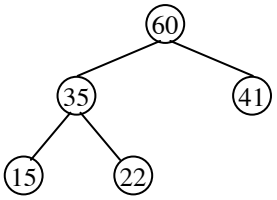
Đầu tiên dùng thủ tục CreateHeap để tạo một Heap từ dãy số trên.



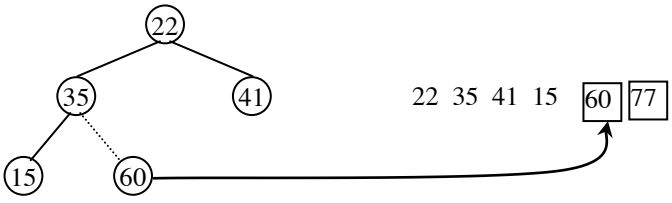
Đổi phần tử đầu với phần tử cuối ta được hình ảnh của Heap và dãy số như sau:



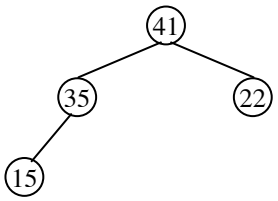
Loại bỏ phần tử cuối cùng khỏi Heap vì đã xếp đúng vị trí, thực hiện thủ tục SiftDown để tạo thành một Heap mới gồm 5 phần tử.



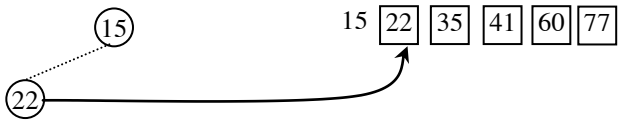
Chọn phần tử lớn nhất trong Heap đưa về cuối, ta được 2 phần tử xếp đúng vị trí:



Tiếp tục dùng thủ tục SiftDown cho 4 phần tử còn lại ta được Heap:



Tiếp tục thực hiện các bước tương tự trên, cuối cùng ta được 5 phần tử xếp đúng vị trí, lúc này dãy số được sắp xếp.



Thuật toán HeapSort: sắp xếp dãy số a[1],a[2],...,a[n].

Dùng thủ tục CreateHeap tạo Heap cho dãy số.

For i:=n Downto 2, thực hiện các công việc sau:

+ Hoán vị a[1] với a[i]

+ Dùng thủ tục SiftDown chuyển i-1 phần tử đầu của dãy thành Heap

Chi tiết thủ tục HeapSort sắp xếp dãy số a[1],a[2],...,a[n]theo thứ tự tăng.

```
Procedure HeapSort;

   {Hieu chinh day a[l..r] thanh heap}

 Procedure SiftDown(l,r:Integer);
  var i,j,x :Integer; cont:Boolean;
  Begin
    i:=l;j:=2*i;
    x:=a[i]; cont:=true;
    while (j<=r) and cont do
      begin
        if j<r then
          if a[j]>a[j+1] then j:=j+1;
          if a[j]>x then cont:=false
          else
            begin
              a[i]:=a[j];
              i:=j;
              j:=2*i;
            end;
      end;
      a[i]:=x;
  End;

   {Tạo heap từ mảng}

   Procedure CreateHeap;
```

```
   var l:Integer;
   Begin
     l:=n div 2;
     while l>1 do
       begin
         l:=l-1;
         SiftDown(l,n);
       end;
   End;

{HeapSort}

var i, tmp:Integer;
Begin
    CreateHeap;
    For i:=n downto 2 do
        begin
            tmp:=a[1];
            a[1]:=a[i];
            a[i]:=tmp;
            SiftDown(1,i-1);
        end;
End;
```

Đánh giá độ phức tạp tính toán: thuật toán thực hiện 1 lần thủ tục CreateHeap, thủ tục này thực hiện $n/2$ lần thủ tục SiftDown nên có độ phức tạp trong trường hợp xấu nhất là $O(n\log_2 n)$. Thủ tục HeapSort còn thực hiện n-1 lần thủ tục SiftDown nên độ phức tạp của đoạn này là $O(n\log_2 n)$. Do đó độ phức tạp tính toán của thuật toán HeapSort được đánh giá trong trường hợp xấu nhất là $O(n\log_2 n)$.

24.7. Sắp xếp bằng phân hoạch

Thuật toán sắp xếp bằng phân hoạch thực hiện theo các bước sau:

+ Đầu tiên chọn một trong các phần tử của dãy cần xếp làm phần tử mẫu.

+ Sau đó dãy được phân hoạch thành hai phần bằng cách chuyển tất cả các phần tử có khóa lớn hơn khóa phần tử mẫu về bên phải và các phần tử còn lại về bên trái. Kết quả của phân hoạch là một vị trí k mà mọi phần tử trong danh sách con A[1..k-1] có khóa nhỏ hơn hoặc bằng khóa của phần tử mẫu. Các phần tử trong danh sách con A[k+1..n] có khóa lớn hơn khóa của mẫu.

+ Sắp xếp bằng phân hoạch hai danh sách A[1..k-1] và A[k..n].

Thuật toán sắp xếp bằng phân hoạch được Hoare đưa ra và đặt tên là QuickSort. Thủ tục đệ quy như sau:

```
Procedure QuickSort;
  Procedure Sort(l,r:Integer);
    var i,j:integer; tmp,x:ElementType;
```

```
    Begin
      i:=l;j:=r;
      x:=a[i];
      repeat
        while (a[i].Key<x.Key) do i:=i+1;
        while (a[j].Key>x.Key) do j:=j-1;
        if i<=j then
          begin
            tmp:=a[i];a[i]:=a[j];a[j]:=tmp;
            i:=i+1;j:=j-1;
          end;
      until i>=j;
      if l<j then Sort(l,j);
      if i<r then Sort(i,r);
    End;
Begin
  Sort(1,n);
End;
```

Độ phức tạp tính toán của thuật toán QuickSort là $O(n\log_2 n)$, với n là số phần tử của danh sách.

25. SẮP XẾP NGOÀI

Các thuật toán sắp xếp trình bày trong phần trên mặc dù rất tốt nhưng không thể áp dụng khi cần sắp xếp trên danh sách các đối tượng được lưu trữ ở bộ nhớ ngoài do tính truy xuất tuần tự của các tệp. Thuật toán thường dùng để sắp xếp trên bộ nhớ ngoài là thuật toán trộn các tệp.

Trong phần này ta xét các tệp truy xuất tuần tự, các phần tử của tệp là một bản ghi có kiểu ElementType. Cần sắp xếp các phần tử của tệp theo thứ tự tăng của một trường khóa sắp xếp Key.

25.1. Trộn hai tệp được sắp

Trong phần này ta xét một thuật toán cơ sở cho các thuật toán sắp xếp theo cách trộn trên file. Cho trước hai tệp File1 và File2 đã được sắp xếp theo thứ tự tăng của khóa. Kết quả trộn được lưu vào File3.

Thuật toán trộn hai tệp có thứ tự:

Mở File1 và File2 để đọc, mở File3 để ghi.

Đọc phần tử đầu tiên X của File1 và phần tử đầu tiên Y của File2.

Lặp lại các bước sau cho đến khi kết thúc File1 hoặc File2

+ Nếu X.Key<Y.Key thì

  Ghi X vào File3

  Đọc một giá trị mới của X trong File1

+ Ngược lại

Ghi Y vào File3

Đọc một giá trị mới của Y trong File2.

Nếu quá trình trên đã ghi hết các phần tử của File1 thì chép tất cả những phần tử còn lại từ File2 sang File3.

Ngược lại, nếu đã ghi hết các phần tử của File2 thì chép tất cả những phần tử còn lại từ File1 sang File3.

25.2. Thuật toán sắp xếp trộn tự nhiên

Để minh họa cách sắp xếp trộn tự nhiên, ta xét tệp *F* gồm các số nguyên như sau:

F: 75 55 15 20 85 30 35 10 60 40 50 25 45 80 70 65

Ta thấy rằng trong *F* chứa các phần tử là các dãy con đã sắp thứ tự (còn gọi là các đường chạy):

F: 75 | 55 | 15 20 85 | 30 35 | 10 60 | 40 50 | 25 45 80 | 70 | 65

Ta lần lượt sao lần lượt từng đường chạy luân phiên vào hai tệp *F*1 và *F*2. Khi đó ta có kết quả tệp *F*1 và *F*2 như sau

*F*1: 75 | 15 20 85 | 10 60 | 25 45 80 | 65

*F*2: 55 | 30 35 40 50 70

Lưu ý rằng sau khi chép sang *F*2 thì 3 đường chạy cuối được kết hợp lại tạo thành đường chạy dài hơn.

Tiến hành trộn hai tệp *F*1 và *F*2 ra tệp *F* lần lượt theo từng đường chạy ở mỗi tệp. Nếu có một tệp nào đó kết thúc thì chép toàn bộ các phần tử của tệp còn lại vào *F*.

F: 55 75 | 15 20 30 35 40 50 70 85 | 10 60 | 25 45 80 | 65

Tiếp tục chia tệp *F* bằng cách sao luân phiên các đường chạy vào các tệp *F*1, *F*2.

*F*1: 55 75 | 10 60 65

*F*2: 15 20 30 35 40 50 70 85 | 25 45 80

Cũng như trước, ta trộn hai tệp *F*1 và *F*2 lần lượt từng đường chạy ở mỗi tệp.

F: 15 20 30 35 40 50 55 70 75 85 | 10 25 45 60 65 80

Lại chia *F* thành hai tệp *F*1 và *F*2:

*F*1: 15 20 30 35 40 50 55 70 75 85

*F*2: 10 25 45 60 65 80

Cuối cùng trộn hai tệp *F*1 và *F*2 sẽ được tệp *F* được sắp xếp.

F: 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85

Thuật toán trộn tự nhiên được thực hiện bởi các thuật toán sau:

Thuật toán 1: Chia tệp *F* thành các tệp *F*1 và *F*2 bằng cách sao luân phiên các đường chạy.

Thuật toán phân chia các đường chạy:

Mở F1 và F2 để ghi, Mở F3 để đọc.

Lặp khi chưa kết thúc tệp F thực hiện các bước:

Chép một đường chạy của F vào F1 như sau: thực hiện một cách lặp lại việc đọc một phần tử của F và và ghi vào F1 cho đến khi phần tử trong F nhỏ hơn phần tử được ghi hoặc đã hết tệp F.

Nếu tệp F chưa kết thúc thì chép một đường chạy từ F vào F2 theo cách tương tự như trên.

Thuật toán 2: Trộn luân phiên các đường chạy từ hai tệp *F*1 và *F*2 vào tệp *F*. Kết quả trả về giá trị NumRunWays cho biết số đường chạy có trong tệp *F*.

Thuật toán trộn các đường chạy:

Mở F1 và F2 để đọc, mở F để ghi.

Khởi tạo NumRunWays=0

Lặp khi chưa hết tệp F1 và chưa hết tệp F2 thực hiện các bước:

+ Lặp trong khi chưa hết tệp F1 và chưa hết tệp F2 thực hiện

Nếu phần tử tiếp theo trong F1 nhỏ hơn phần tử tiếp theo trong F2 thì ghi phần tử tiếp theo trong F1 vào F.

Ngược lại thì ghi phần tử tiếp theo trong F2 vào F.

+ Nếu kết thúc một đường chạy trong F1 thì chép phần đường chạy còn lại của F2 vào F.

+ Ngược lại thì chép phần đường chạy còn lại của F1 vào F.

+ Tăng giá trị biến NumRunWays

Chép tất cả các đường chạy còn lại trong F1 hoặc F2 vào F, với mỗi đường chạy tăng giá trị biến NumRunWays.

Thuật toán 3: sắp xếp tệp *F* theo kiểu trộn tự nhiên. Thuật toán dùng hai tệp phụ *F*1 và *F*2.

Thuật toán trộn tự nhiên:

Lặp lại các bước sau:

+ Thực hiện phân các đường chạy của tệp F vào hai tệp F1 và F2.

+ Thực hiện trộn các đường chạy ở hai tệp F1 và F2 vào F, với số đường chạy của F là NumRunWays.

Cho đến khi NumRunWays =1

Cài đặt chương trình sắp xếp ngoài bằng thuật toán trộn tự nhiên:

```
Procedure NaturalMerge(F : FileType);
var l:Integer; {số đường chạy được trộn}
```

```
        eor:Boolean; {cuối đường chạy}
        f1, f2:FileType;

{Thủ tục copy một phần tử từ tệp x vào tệp y và kiểm tra kết thúc đường chạy}

Procedure Copy(var x,y:FileType);
var buf:ElementType;
    Begin
        Read(x,buf);write(y,buf);
        if eof(x) then eor:=true else eor:=buf.Key>x^.Key;
    End;

{Thủ tục copy một đường chạy từ tệp x vào tệp y}

Procedure CopyRun(var x,y:FileType);
    Begin
        Repeat Copy(x,y) Until eor;
    End;

{Thủ tục phân bố các đường chạy từ F vào F1 và F2}

Procedure Distribute;
    Begin
        Repeat
            CopyRun(F,F1);
            If not eof(F) then CopyRun(F,F2);
        Until eof(F);
    End;

{Thủ tục trộn một đường chạy từ F1 và F2 vào F}

Procedure MergeRun;
    Begin
        Repeat
            if F1^.Key <= F2^.Key then
                begin
                    Copy(F1,F);
                    If eor then CopyRun(F2,F);
                end
            else
                begin
                    Copy(F2,F);
                    If eor then CopyRun(F1,F);
                end;
        Until eor;
    End;
            {Thủ tục trộn luân phiên các đường chạy từ F1 và F2 vào F}

Procedure Merge;
```

```
    Begin
      while not eof(f1) and not eof(F2) do
        begin
          MergeRun; l:=l+1;
        end;
      while not eof(F1) do
        begin
          CopyRun(F1,F); l:=l+1;
        end;
      while not eof(F2) do
        begin
          CopyRun(F2,F); l:=l+1;
        end;
    End;

Begin
  Repeat
    Rewrite(F1); Rewrite(F2); Reset(F);
    Distribute;
    Reset(F1); Reset(F2); Rewrite(F);
    l:=0; Merge;
  Until l=1;
End;
```

26. BÀI TẬP

Bài 1. Trình bày thuật toán và viết thủ tục thực hiện sắp xếp một dãy số nguyên sao cho các phần tử chẵn ở đầu và các phần tử lẻ ở cuối. Ràng buộc: Độ phức tạp của thuật toán không quá O(n), với n là số phần tử của dãy số và không được dùng mảng phụ.

Bài 2. Trình bày thuật toán và viết thủ tục trộn hai danh sách đã sắp thứ tự tăng thành một danh sách theo thứ tự tăng. Đánh giá độ phức tạp của thuật toán.

Bài 3. Hãy viết thủ tục trộn tự nhiên trên danh sách các đối tượng tổ chức bằng mảng.

Bài 4. Viết thủ tục trộn hai tệp đã có thứ tự thành một tệp có thứ tự.

Bài 5. Sử dụng kết quả ở bài 4, trình bày thuật toán trộn m tệp *F*1, *F*2, ..,*Fm* đã có thứ tự thành một tệp *F* có thứ tự sao cho số lần phải đọc các phần tử là ít nhất. (*Gợi ý: cho biết trước số phần tử của các tệp F*1, *F2, .., Fm lần lượt là n*1, *n*2, ..., *nm*).

Bài 6. Sử dụng kết quả bài 5. Trình bày thuật toán và viết thủ tục sắp xếp một tệp.

TÀI LIỆU THAM KHẢO

1. Niklaus Wirth, Cấu trúc dữ liệu + Giải thuật = Chương trình, bản dịch tiếng Việt của Nguyễn Quốc Cường, NXBGD, 1993.
2. Đỗ Xuân Lôi, Cấu trúc dữ liệu và Giải thuật, NXBGD, 1993.
3. Nguyễn Trung Trực, Cấu trúc dữ liệu, Khoa CNTT Trường ĐHBK Tp.HCM, 1997.
4. Đinh Mạnh Tường, Cấu trúc dữ liệu và thuật toán, NXBKHKT, 2001.
5. Trần Hạnh Nhi, Trương Anh Đức, Giáo trình Cấu trúc dữ liệu 1, Khoa CNTT, ĐHKHTN, ĐHQGHCM, 1996.
6. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Data Structures and Algorithms, Addison-Wesley Publishing, 1983.
7. Derick Wood, Data Strutures, Algorithms, and Performance, Addison-Wesley Publishing, 1993.
8. Alfred V. Aho, Jeffrey D. Ullman, Cơ sở của khoa học máy tính, Tập 1, 2, Bản dịch tiếng Việt của Trần Đức Quang, NXB Thống kê, 1999.
9. Larry Nyhoff, Sanford Leestma, Lập trình nâng cao bằng Pascal với các cấu trúc dữ liệu, Tập 1, 2, Bản dịch tiếng Việt của Lê Minh Trung, Công ty Scitec, 1991.