ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG CAO ĐẲNG CÔNG NGHỆ THÔNG TIN

BÀI GIẢNG

CẤU TRÚC DỮ LIỆU VÀ GIẢI THẬT

NGUYỄN ĐỨC HIỂN

ĐÀ NẴNG – 2007

MỤC LỤC

CHƯƠNG 1

TỔNG QUAN VỀ THUẬT TOÁN VÀ CẤU TRÚC DỮ LIỆU

I. CÁC BƯỚC CƠ BẢN KHI GIẢI QUYẾT BÀI TOÁN TIN HỌC

I.1. Xác định bài toán

Việc xác định bài toán tức là phải xác định xem ta phải giải quyết vấn đề gì?, với giả thiết nào đã cho và lời giải cần phải đạt những yêu cầu nào.

Input → Process → Output

(Dữ liệu vào → Xử lý → Kết quả ra)

Đối với những bài toán tin học ứng dụng trong thực tế, lời giải cần tìm chỉ cần tốt tới mức nào đó, thậm chí là tồi ở mức chấp nhận được. Bởi lời giải tốt nhất đòi hỏi quá nhiều thời gian và chi phí.

Ví dụ:

Khi cài đặt các hàm số phức tạp trên máy tính. Nếu tính bằng cách khai triển chuỗi vô hạn thì độ chính xác cao hơn nhưng thời gian chậm hơn hàng tỉ lần so với phương pháp xấp xỉ. Trên thực tế việc tính toán luôn luôn cho phép chấp nhận một sai số nào đó nên các hàm số trong máy tính đều được tính bằng phương pháp xấp xỉ của giải tích số

Xác định đúng yêu cầu bài toán là rất quan trọng bởi nó ảnh hưởng tới cách thức giải quyết và chất lượng của lời giải. Một bài toán thực tế thường cho bởi những thông tin khá mơ hồ và hình thức, ta phải phát biểu lại một cách chính xác và chặt chẽ để hiểu đúng bài toán.

Ví dụ:

- *Bài toán: Một dự án có n người tham gia thảo luận, họ muốn chia thành các nhóm và mỗi nhóm thảo luận riêng về một phần của dự án. Nhóm có bao nhiêu người thì được trình lên bấy nhiêu ý kiến. Nếu lấy ở mỗi nhóm một ý kiến đem ghép lại thì được một bộ ý kiến triển khai dự án. Hãy tìm cách chia để số bộ ý kiến cuối cùng thu được là lớn nhất.*
- *Phát biểu lại: Cho một số nguyên dương n, tìm các phân tích n thành tổng các số nguyên dương sao cho tích của các số đó là lớn nhất.*

Trên thực tế, ta nên xét một vài trường hợp cụ thể để thông qua đó hiểu được bài toán rõ hơn và thấy được các thao tác cần phải tiến hành. Đối với những bài toán đơn giản, đôi khi chỉ cần qua ví dụ là ta đã có thể đưa về một bài toán quen thuộc để giải.

I.2. Xác đinh cấu trúc dữ liệu

Kiểu dữ liệu (data type): kiểu dữ liệu của một biến là tập hợp các giá trị mà biến đó có thể nhận. Ví dụ một biến kiểu Boolean chỉ có thể nhận TRUE hoặc FALSE mà không nhận giá trị nào khác. Các kiểu dữ liệu cơ bản (như Integer, Char, Real, Boolean) được cung cấp khác nhau trong các ngôn ngữ lập trình khác nhau.

Một kiểu dữ liệu trừu tượng (abstract data type): là một mô hình toán học cùng với một tập hợp các phép toán trên nó. Có thể nói kiểu dữ liệu trừu tượng là một kiểu dữ liệu do chúng ta định nghĩa ở mức khái niệm (conceptual), nó chưa được cài đặt cụ thể bằng một ngôn ngữ lập trình. Như đã dẫn ra ở trên, chúng ta dùng kiểu dữ liệu trừu tượng để thiết kế giải thuật, nhưng để cài đặt giải thuật vào một ngôn ngữ lập trình chúng ta phải tìm cách biểu diễn kiểu dữ liệu trừu tượng trên các kiểu dữ liệu và toán tử do ngôn ngữ lập trình cung cấp.

Cấu trúc dữ liệu: Tập hợp các biến có thể thuộc một hoặc vài kiểu dữ liệu khác nhau được nối kết với nhau tạo thành những phần tử. Các phần tử này chính là thành phần cơ bản xây dựng nên cấu trúc dữ liệu. Cấu trúc dữ liệu là nguyên tắc kết nối các phần tử này với nhau trong bộ nhớ khi được biểu diễn bằng một ngôn ngữ lập trình cụ thể.

Khi giải một bài toán, ta cần phải định nghĩa tập hợp dữ liệu để biểu diễn tình trạng cụ thể. Việc lựa chọn này tuỳ thuộc vào vấn đề cần giải quyết và những thao tác sẽ tiến hành trên dữ liệu vào. Có những thuật toán chỉ thích ứng với một cách tổ chức dữ liệu nhất định, đối với những cách tổ chức dữ liệu khác thì sẽ kém hiệu quả hoặc không thể thực hiện được. Chính vì vậy nên bước xây dựng cấu trúc dữ liệu không thể tách rời bước tìm kiếm thuật toán giải quyết vấn đề.

Các tiêu chuẩn khi lựa chọn cấu trúc dữ liệu

- Cấu trúc dữ liệu trước hết phải biểu diễn được đầy đủ các thông tin nhập và xuất của bài toán
- Cấu trúc dữ liệu phải phù hợp với các thao tác của thuật toán mà ta lựa chọn để giải quyết bài toán.
- Cấu trúc dữ liệu phải cài đặt được trên máy tính với ngôn ngữ lập trình đang sử dụng

Đối với một số bài toán, trước khi tổ chức dữ liệu ta phải viết một đoạn chương trình nhỏ để khảo sát xem dữ liệu cần lưu trữ lớn tới mức độ nào.

I.3. Tìm thuật toán

Thuật toán và Cấu trúc dữ liệu có mối quan hệ mật thiết với nhau. Do đó, khi xây dựng một cấu trúc dữ liệu thì đi đôi với việc xác lập các thuật toán xử lý trên cấu trúc dữ liệu đó.

Data Structure + Algorithm =Program

Thuật toán là một hệ thống chặt chẽ và rõ ràng các quy tắc nhằm xác định một dãy thao tác trên cấu trúc dữ liệu sao cho: Với một bộ dữ liệu vào, sau một số hữu hạn bước thực hiện các thao tác đã chỉ ra, ta đạt được mục tiêu đã định.

Các đặc trưng của thuật toán

➢ **1. Tính đơn định**

Ở mỗi bước của thuật toán, các thao tác phải hết sức rõ ràng, không gây nên sự nhập nhằng, lộn xộn, tuỳ tiện, đa nghĩa. Thực hiện đúng các bước của thuật toán thì với một dữ liệu vào, chỉ cho duy nhất một kết quả ra.

➢ **2. Tính dừng**

Thuật toán không được rơi vào quá trình vô hạn, phải dừng lại và cho kết quả sau một số hữu hạn bước.

➢ **3. Tính đúng**

Sau khi thực hiện tất cả các bước của thuật toán theo đúng quá trình đã định, ta phải được kết quả mong muốn với mọi bộ dữ liệu đầu vào. Kết quả đó được kiểm chứng bằng yêu cầu bài toán.

➢ **4. Tính phổ dụng**

Thuật toán phải dễ sửa đổi để thích ứng được với bất kỳ bài toán nào trong một lớp các bài toán và có thể làm việc trên các dữ liệu khác nhau.

➢ **5. Tính khả thi**

a) Kích thước phải đủ nhỏ: Ví dụ: Một thuật toán sẽ có tính hiệu quả bằng 0 nếu lượng bộ nhớ mà nó yêu cầu vượt quá khả năng lưu trữ của hệ thống máy tính.

b) Thuật toán phải được máy tính thực hiện trong thời gian cho phép, điều này khác với lời giải toán (Chỉ cần chứng minh là kết thúc sau hữu hạn bước). Ví dụ như xếp thời khoá biểu cho một học kỳ thì không thể cho máy tính chạy tới học kỳ sau mới ra được.

c) Phải dễ hiểu và dễ cài đặt.

Ví dụ:

- *Input: 2 số nguyên tự nhiên a và b không đồng thời bằng 0*
- *Output: Ước số chung lớn nhất của a và b*

Thuật toán sẽ tiến hành được mô tả như sau: (Thuật toán Euclide)

- *Bước 1 (Input): Nhập a và b: Số tự nhiên*
- *Bước 2: Nếu b ≠ 0 thì chuyển sang bước 3, nếu không thì bỏ qua bước 3, đi làm bước 4*
- *Bước 3: Đặt r := a mod b; Đặt a := b; Đặt b := r; Quay trở lại bước 2.*
- *Bước 4 (Output): Kết luận ước số chung lớn nhất phải tìm là giá trị của a. Kết thúc thuật toán.*

Một số vấn đề cần lưu ý

- Khi mô tả thuật toán bằng ngôn ngữ tự nhiên, ta không cần phải quá chi tiết các bước và tiến trình thực hiện mà chỉ cần mô tả một cách hình thức đủ để chuyển thành ngôn ngữ lập trình. Viết sơ đồ các thuật toán đệ quy là một ví dụ.
- Đối với những thuật toán phức tạp và nặng về tính toán, các bước và các công thức nên mô tả một cách tường minh và chú thích rõ ràng để khi lập trình ta có thể nhanh chóng tra cứu.
- Đối với những thuật toán kinh điển thì phải thuộc. Khi giải một bài toán lớn trong một thời gian giới hạn, ta chỉ phải thiết kế tổng thể còn những chỗ đã thuộc thì cứ việc lắp ráp vào. Tính đúng đắn của những mô-đun đã thuộc ta không cần phải quan tâm nữa mà tập trung giải quyết các phần khác.

I.4. Lập trình

Sau khi đã có thuật toán, ta phải tiến hành lập trình thể hiện thuật toán đó. Muốn lập trình đạt hiệu quả cao, cần phải có kỹ thuật lập trình tốt. Kỹ thuật lập trình tốt thể hiện ở kỹ năng viết chương trình, khả năng gỡ rối và thao tác nhanh. Lập trình tốt không phải chỉ cần nắm vững ngôn ngữ lập trình là đủ, phải biết cách viết chương trình uyển chuyển, khôn khéo và phát triển dần dần để chuyển các ý tưởng ra thành chương trình hoàn chỉnh. Kinh nghiệm cho thấy một thuật toán hay nhưng do cài đặt vụng về nên khi chạy lại cho kết quả sai hoặc tốc độ chậm.

Thông thường, ta không nên cụ thể hoá ngay toàn bộ chương trình mà nên tiến hành theo phương pháp tinh chế từng bước (Stepwise refinement):

- Ban đầu, chương trình được thể hiện bằng ngôn ngữ tự nhiên, thể hiện thuật toán với các bước tổng thể, mỗi bước nêu lên một công việc phải thực hiện.
- Một công việc đơn giản hoặc là một đoạn chương trình đã được học thuộc thì ta tiến hành viết mã lệnh ngay bằng ngôn ngữ lập trình.
- Một công việc phức tạp thì ta lại chia ra thành những công việc nhỏ hơn để lại tiếp tục với những công việc nhỏ hơn đó.

Trong quá trình tinh chế từng bước, ta phải đưa ra những biểu diễn dữ liệu. Như vậy cùng với sự tinh chế các công việc, dữ liệu cũng được tinh chế dần, có cấu trúc hơn, thể hiện rõ hơn mối liên hệ giữa các dữ liệu.

Phương pháp tinh chế từng bước là một thể hiện của tư duy giải quyết vấn đề từ trên xuống, giúp cho người lập trình có được một định hướng thể hiện trong phong cách viết chương trình. Tránh việc mò mẫm, xoá đi viết lại nhiều lần, biến chương trình thành tờ giấy nháp.

I.5. Kiểm thử

➢ 1. Chạy thử và tìm lỗi

Chương trình là do con người viết ra, mà đã là con người thì ai cũng có thể nhầm lẫn. Một chương trình viết xong chưa chắc đã chạy được ngay trên máy tính để cho ra kết quả mong muốn. Kỹ năng tìm lỗi, sửa lỗi, điều chỉnh lại chương trình cũng là một kỹ năng quan trọng của người lập trình. Kỹ năng này chỉ có được bằng kinh nghiệm tìm và sửa chữa lỗi của chính mình.

Có ba loại lỗi:

- Lỗi cú pháp: Lỗi này hay gặp nhất nhưng lại dễ sửa nhất, chỉ cần nắm vững ngôn ngữ lập trình là đủ. Một người được coi là không biết lập trình nếu không biết sửa lỗi cú pháp.
- Lỗi cài đặt: Việc cài đặt thể hiện không đúng thuật toán đã định, đối với lỗi này thì phải xem lại tổng thể chương trình, kết hợp với các chức năng gỡ rối để sửa lại cho đúng.
- Lỗi thuật toán: Lỗi này ít gặp nhất nhưng nguy hiểm nhất, nếu nhẹ thì phải điều chỉnh lại thuật toán, nếu nặng thì có khi phải loại bỏ hoàn toàn thuật toán sai và làm lại từ đầu.

➢ 2. Xây dựng các bộ test

Có nhiều chương trình rất khó kiểm tra tính đúng đắn. Nhất là khi ta không biết kết quả đúng là thế nào?. Vì vậy nếu như chương trình vẫn chạy ra kết quả (không biết đúng sai thế nào) thì việc tìm lỗi rất khó khăn. Khi đó ta nên làm các bộ test để thử chương trình của mình.

Các bộ test nên đặt trong các file văn bản, bởi việc tạo một file văn bản rất nhanh và mỗi lần chạy thử chỉ cần thay tên file dữ liệu vào là xong, không cần gõ lại bộ test từ bàn phím. Kinh nghiệm làm các bộ test là:

- Bắt đầu với một bộ test nhỏ, đơn giản, làm bằng tay cũng có được đáp số để so sánh với kết quả chương trình chạy ra.
- Tiếp theo vẫn là các bộ test nhỏ, nhưng chứa các giá trị đặc biệt hoặc tầm thường. Kinh nghiệm cho thấy đây là những test dễ sai nhất.
- Các bộ test phải đa dạng, tránh sự lặp đi lặp lại các bộ test tương tự.
- Có một vài test lớn chỉ để kiểm tra tính chịu đựng của chương trình mà thôi. Kết quả có đúng hay không thì trong đa số trường hợp, ta không thể kiểm chứng được với test này.

Lưu ý rằng chương trình chạy qua được hết các test không có nghĩa là chương trình đó đã đúng. Bởi có thể ta chưa xây dựng được bộ test làm cho chương trình chạy sai. Vì vậy nếu có thể, ta nên tìm cách chứng minh tính đúng đắn của thuật toán và chương trình, điều này thường rất khó.

I.6. Tối ưu hoá chương trình

Một chương trình đã chạy đúng không có nghĩa là việc lập trình đã xong, ta phải sửa đổi lại một vài chi tiết để chương trình có thể chạy nhanh hơn, hiệu quả hơn. Thông thường, trước khi kiểm thử thì ta nên đặt mục tiêu viết chương trình sao cho đơn giản, miễn sao chạy ra kết quả đúng là được, sau đó khi tối ưu chương trình, ta xem lại những chỗ nào viết chưa tốt thì tối ưu lại mã lệnh để chương trình ngắn hơn, chạy nhanh hơn. Không nên viết tới đâu tối ưu mã đến đó, bởi chương trình có mã lệnh tối ưu thường phức tạp và khó kiểm soát.

Ta nên tối ưu chương trình theo các tiêu chuẩn sau:

- **1. Tính tin cậy**

Chương trình phải chạy đúng như dự định, mô tả đúng một giải thuật đúng. Thông thường khi viết chương trình, ta luôn có thói quen kiểm tra tính đúng đắn của các bước mỗi khi có thể.

- **2. Tính uyển chuyển**

Chương trình phải dễ sửa đổi. Bởi ít có chương trình nào viết ra đã hoàn hảo ngay được mà vẫn cần phải sửa đổi lại. Chương trình viết dễ sửa đổi sẽ làm giảm bớt công sức của lập trình viên khi phát triển chương trình.

- **3. Tính trong sáng**

Chương trình viết ra phải dễ đọc dễ hiểu, để sau một thời gian dài, khi đọc lại còn hiểu mình làm cái gì?. Để nếu có điều kiện thì còn có thể sửa sai (nếu phát hiện lỗi mới), cải tiến hay biến đổi để được chương trình giải quyết bài toán khác. Tính trong sáng của chương trình phụ thuộc rất nhiều vào công cụ lập trình và phong cách lập trình.

- **4. Tính hữu hiệu**

Chương trình phải chạy nhanh và ít tốn bộ nhớ, tức là tiết kiệm được cả về không gian và thời gian. Để có một chương trình hữu hiệu, cần phải có giải thuật tốt và những tiểu xảo khi lập trình. Tuy nhiên, việc áp dụng quá nhiều tiểu xảo có thể khiến chương trình trở nên rối rắm, khó hiểu khi sửa đổi. Tiêu chuẩn hữu hiệu nên dừng lại ở mức chấp nhận được, không quan trọng bằng ba tiêu chuẩn trên. Bởi phần cứng phát triển rất nhanh, yêu cầu hữu hiệu không cần phải đặt ra quá nặng.

Từ những phân tích ở trên, chúng ta nhận thấy rằng việc làm ra một chương trình đòi hỏi rất nhiều công đoạn và tiêu tốn khá nhiều công sức. Chỉ một công đoạn không hợp lý sẽ làm tăng chi phí viết chương trình. Nghĩ ra cách giải quyết vấn đề đã khó, biến ý tưởng đó thành hiện thực cũng không dễ chút nào.

Những cấu trúc dữ liệu và giải thuật đề cập tới trong chuyên đề này là những kiến thức rất phổ thông, một người học lập trình không sớm thì muộn cũng phải biết tới. Chỉ hy vọng rằng khi học xong chuyên đề này, qua những cấu trúc dữ liệu và giải thuật hết sức mẫu mực, chúng ta rút ra được bài học kinh nghiệm: Đừng bao giờ viết chương trình khi mà chưa suy xét kỹ về giải thuật và những dữ liệu cần thao tác, bởi như vậy ta dễ mắc phải hai sai lầm trầm trọng: hoặc là sai về giải thuật, hoặc là giải thuật không thể triển khai nổi trên một cấu trúc dữ liệu

không phù hợp. Chỉ cần mắc một trong hai lỗi đó thôi thì nguy cơ sụp đổ toàn bộ chương trình là hoàn toàn có thể, càng cố chữa càng bị rối, khả năng hầu như chắc chắn là phải làm lại từ đầu(*).

II. DIỄN TẢ THUẬT TOÁN

II.1.Dùng lưu đồ

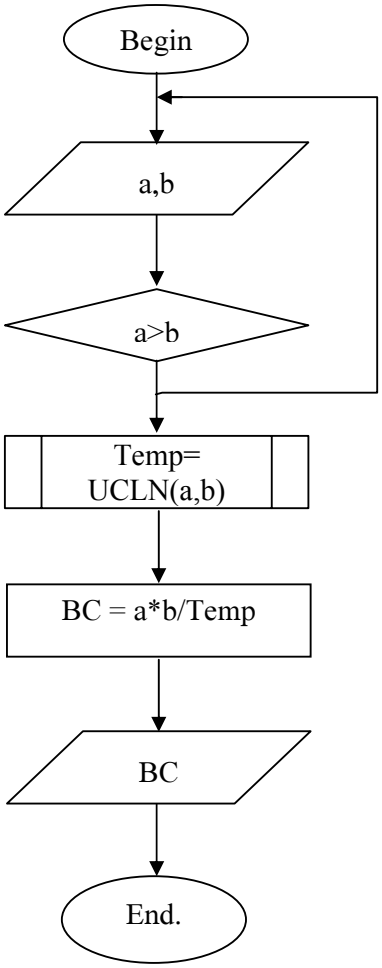
Lưu đồ thuật toán là các hình vẽ theo những qui định nào đó, được kết hợp lại nhằm mô tả lại quá trình thực hiện của thuật toán một cách trực quan nhất.

Người ta dùng các hình khối để ghép nối thành lưu đồ cơ thể hiện thuật toán.

Ví dụ:

Nhập 2 số a,b nếu a>b thì in kết quả bội số chung nhỏ nhất của a và b, ngược lại nhập lại a,b. Vẽ lưu đồ mô phỏng tiến trình làm việc của thuật toán.

Lưu đồ thuật toán



Đối với những bài toán nhỏ thì việc dùng biểu đồ thuật toán không mấy khó khăn, nhưng đối với những bài toán lớn thì việc dùng lưu đồ thuật toán để biểu diễn giải thuật thì sẽ gặp một số trở ngại nhất định về không gian biểu diễn. Chính vì hạn chế này nên người ta ít dùng lưu đồ thuật toán để biểu diễn những bài toán lớn phức tạp.

II.2. Dùng ngôn ngữ lập trình cụ thể

Việc dùng lưu đồ để biểu diễn thuật toán đã bộc lộ những nhược điểm nhất định như đã nêu trên, nên người lập trình còn thường dùng các ngôn ngữ lập trình bậc cao như PASCAL, C, C++, JAVA,… để biểu diễn thuật toán. Xét về mặt kỹ thuật, nếu dùng ngôn ngữ lập trình cụ thể để biểu diễn thuật toán, thì thông qua mã lệnh của chương trình, người đọc nếu biết ngôn ngữ lập trình đang cài đặt sẽ kiểm tra được kết quả, và có thể dò ra hướng đi của thuật toán mà không phải thông qua các bước cài đặt.

Cũng với ví dụ trên, nếu ta dùng ngôn ngữ lập trình PASCAL biểu diễn giải thuật thì nó được thể hiện như sau:

```
program TIM_BOI_CHUNG_NHO_NHAT;
function UCLN(a,b:word):word;
var r,q:word;
begin
  while(a<>b)do
   begin
    if(a>b)then a:= a - b  else b:=b - a;
   end; UCLN:=a;
end;
var a,b,BC:word;
begin
  write('a=');readln(a);write('b=');readln(b);
  if(a>b)then   begin
                          BC:=(a*b div UCLN(a,b));
 write('BCNN[',a,',',b,']=',BC);
end
   else  write('reInput'); readln;
end.
```

Và nếu dùng ngôn ngữ lập trình C biểu diễn kết quả sẽ như sau:

```
#include <conio.h>
#include <stdio.h>
//program TIM_BOI_CHUNG_NHO_NHAT;
int UCLN(int a, int b)
{
        while(a != b)
        {       if(a > b) a = a - b;
                else b = b - a; }
        return a;
}
int main()
{
        int a, b, BC;
        printf("a=");scanf("%d",&a);
        printf("b=");scanf("%d",&b);
        if(a>b)
        {       BC = ((int)a*b/UCLN(a,b));
                printf("BCNN[%d,%d]=%d",a,b,BC);        }
        else printf(" nhap lai:");
```

```
        getch();
        return 0;
}
```

Dùng ngôn ngữ lập trình cụ thể để biểu diễn giải thuật phải thừa nhận là dễ kiểm tra kết quả, nhưng mặt khác nó yêu cầu người đọc phải hiểu về ngôn ngữ đã được thể hiện, mà điều này không phải lúc nào cũng sẵn có.

II.3. Dùng ngôn ngữ giả

Cách biểu diễn bằng lưu đồ thuật toán như trên xem ra cũng có những hạn chế nhất định về không gian biểu diễn đặc biệt đối với những bài toán lớn và nhiều chức năng xử lý phức tạp, còn nếu dùng thuần túy một ngôn ngữ cấp cao nào đó như PASCAL, C, C++, JAVA,… để biểu diễn, thì ta sẽ gặp một số hạn chế sau:

- *Phải tuân thủ các qui tắc chặt chẽ về cú pháp của ngôn ngữ đó, khiến cho việc trình bày giải thuật và cấu trúc dữ liệu trở nên nặng nề, gò bó và cứng nhắc.*
- *Phải phụ thuộc vào cấu trúc dữ liệu tiền định của ngôn ngữ nên có lúc không thể hiện được đầy đủ các ý về cấu trúc mà ta muốn biểu đạt.*
- *Ngôn ngữ nào được chọn cũng chưa hẳn được mọi người ưu thích và muốn sử dụng.*

Vì vậy, người ta dùng ngôn ngữ thô hơn mềm dẻo hơn, gần gũi với ngôn ngữ tự nhiên hơn và dễ sử dụng đó là ngôn ngữ giả mã, ngôn ngữ giả mã là ngôn ngữ tự nhiên kết hợp với các từ khóa ngôn ngữ lập trình, với một mức độ linh hoạt nhất định, không quá gò bó, không câu nệ về cú pháp của ngôn ngữ lập trình, nên người ta thường hay sử dụng. Tuy nhiên, để thống nhất nhau trong cách biểu diễn người ta cũng đưa ra một số qui cách cú pháp và được xem như qui định chung.

Các kiểu dữ liệu cơ sở: integer, char, boolean, float.

Cấu trúc của một chương trình

```
Program
        S1
        S2
        ...
        Sn
Return.
```

Lưu ý:

- ✓ Phần ghi chú và thuyết minh: được đặt sau dấu '//' hoặc trong cặp dấu '/* …..*/'
- ✓ Nếu chương trình gồm nhiều bước có thể đánh số thứ tự mỗi bước kèm theo lời giải thích

Ví dụ: tính n!

```
Program TinhGiaiThua
1.Read(n)       //nhập n
2.//tính p=n!
  p := 1
For i :=1 To n
        P:= p*i
3. Write(p)     //in kết quả
Return
```

Các ký tự:

- Các phép toán số học: +, -, *, /, ↑ (luỹ thừa), Div (chia nguyên), Mod (chia lấy phần dư)
- Các ký tự quan hệ: >, <, >=, <=, =, <>
- Các phép toán lôgic: And, Or, Not

Các lệnh và cú pháp

- Phép gán: biến := biểu thức
- Lệnh ghép: lệnh ghép là lệnh có
  - ✓ số lệnh >1
  - ✓ bọc bởi cặp dấu {…}
  - ✓ Là lệnh khi có điều kiện nào đó xãy ra (thuộc vào lệnh if hoặc vòng lặp)
- Lệnh nhập: Read(biến)
- Lệnh xuất: Write(biểu thức hoặc nội dung) or Write(biểu thức hoặc nội dung)
- Lệnh rẽ nhánh
  - ✓ Cú pháp 1:
    If <biểu thức logic> Then
    Lệnh
  - ✓ Cú pháp 2:
    If <biểu thức logic> Then
    Lệnh 1
    Else
    Lệnh 2
  - ✓ Cú pháp 3 (lệnh có nhiều lựa chọn)
    Case <biểu thức nguyên> Of
    <giá trị nguyên 1>: Lệnh 1
    <giá trị nguyên 2>: Lệnh 2
    ...
    <giá trị nguyên n>: Lệnh n
    [Else Lệnh n+1]
    EndCase
- Lệnh vòng lặp
  - ✓ Vòng lặp FOR
    For <biến := giá trị đầu> To <giá trị cuối> [Step <bước nhảy>]
    Lệnh

  Chú ý: nếu [step <bước nhảy>] không có thì hiểu biến := biến + 1 đơn vị
  - ✓ Vòng lặp WHILE
    While <biểu thức> Do
    Lệnh
  - ✓ Vòng Lặp DO…WHILE
    Do{ Lệnh
    }While <biểu thức>
  - ✓ Ngoài ra để dừng vòng lặp, có thể sử dụng từ khoá break trong vòng lặp muốn dừng
- Chương trình con

Dạng hàm

Func <tên_hàm> [(Danh sách tham số hình thức)]
S1
S2
...

Sn
tên_hàm := <giá trị trả về>
Return

Ví dụ 1.2: Tính tổng $a_1 + a_2 + \ldots + a_n$

Func Tong(a,n)
S := 0
For I := 1 To n
S := S+ai
Tong := S
Return

Dạng chương trình con

Proc <tên chương trình con> [(danh sách tham số hình thức)]
S1
S2
...
Sn
Return

Ví dụ 1.3: Xây dựng chương trình con hoán vị 2 giá trị

Proc HoanVi(a,b)
tam := a
a := b
b := tam
Return

Chú ý:

- ✓ Trường hợp là dạng hàm thì phải có lệnh: ***tên_hàm := <giá trị trả về>***
- ✓ Khi gọi hàm thì tên hàm nằm bên phải phép gán
- ✓ Khi gọi chương trình con: ***Call <tên chương trình con>***
- ✓ Bên trong chương trình con có thể sử dụng lệnh ***Exit, Halt***

- Kiểu dữ liệu bản ghi

Mọi ngôn ngữ đều hỗ trợ cho việc xây dựng cấu trúc bản ghi bằng việc xây dựng kiểu dữ liệu mới từ những kiểu dữ liệu đã có

Định nghĩa kiểu bản ghi

Typedef Kiểu_Bản_Ghi=Record
kiểu_đã_có_1 Trường_1
kiểu_đã_có_2 Trường_2
...
kiểu_đã_có_n Trường_n
End Record

Truy cập vào từng trường thứ i của kiểu bản ghi

biến_kiểu_bản_ghi (trường_i)

Ví dụ: Xây dựng kiểu dữ liệu Điểm_Oxy để lưu trữ một điểm trong mặt phẳng Oxy

typedef Điểm_Oxy=Record
integer ox,oy
End Record

Tạo điểm M(1,2) trong mặt phẳng Oxy:

Điểm_Oxy M
M(ox) := 1, M(oy) := 2

III. THUẬT TOÁN ĐỆ QUI

III.1.Khái niệm đệ qui

Ta nói một đối tượng là đệ quy nếu nó được định nghĩa qua chính nó hoặc một đối tượng khác cùng dạng với chính nó bằng quy nạp.

Ví dụ: Đặt hai chiếc gương cầu đối diện nhau. Trong chiếc gương thứ nhất chứa hình chiếc gương thứ hai. Chiếc gương thứ hai lại chứa hình chiếc gương thứ nhất nên tất nhiên nó chứa lại hình ảnh của chính nó trong chiếc gương thứ nhất... Ở một góc nhìn hợp lý, ta có thể thấy một dãy ảnh vô hạn của cả hai chiếc gương.

Một ví dụ khác là nếu người ta phát hình trực tiếp phát thanh viên ngồi bên máy vô tuyến truyền hình, trên màn hình của máy này lại có chính hình ảnh của phát thanh viên đó ngồi bên máy vô tuyến truyền hình và cứ như thế...

Trong toán học, ta cũng hay gặp các định nghĩa đệ quy:

Giai thừa của n (n!): Nếu n = 0 thì n! = 1; nếu n > 0 thì n! = n.(n-1)!

Số phần tử của một tập hợp hữu hạn S (/S/): Nếu S = ∅ thì /S/= 0; Nếu S ≠ ∅ thì tất có một phần tử x ∈ S, khi đó /S/ = /S\{x}/ + 1. Đây là phương pháp định nghĩa tập các số tự nhiên.

Một định nghĩa đệ qui bao giờ cũng có một điểm dừng hoặc một trường hợp đặc biệt nào đó để xác định giá trị đơn giản nhất của định nghĩa đệ qui. Trường hợp này được gọi là trường hợp suy biến.

III.2. Thuật toán đệ qui

Nếu lời giải của một bài toán P được thực hiện bằng lời giải của bài toán P' có dạng giống như P thì đó là một lời giải đệ quy. Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ quy. Mới nghe thì có vẻ hơi lạ nhưng điểm mấu chốt cần lưu ý là: P' tuy có dạng giống như P, nhưng theo một nghĩa nào đó, nó phải "nhỏ" hơn P, dễ giải hơn P và việc giải nó không cần dùng đến P.

Định nghĩa một hàm đệ quy hay thủ tục đệ quy gồm hai phần:

- Phần neo (anchor) hay còn gọi là Suy biến: Phần này được thực hiện khi mà công việc quá đơn giản, có thể giải trực tiếp chứ không cần phải nhờ đến một bài toán con nào cả.
- Phần đệ quy: Trong trường hợp bài toán chưa thể giải được bằng phần neo, ta xác định những bài toán con và gọi đệ quy giải những bài toán con đó. Khi đã có lời giải (đáp số) của những bài toán con rồi thì phối hợp chúng lại để giải bài toán đang quan tâm.

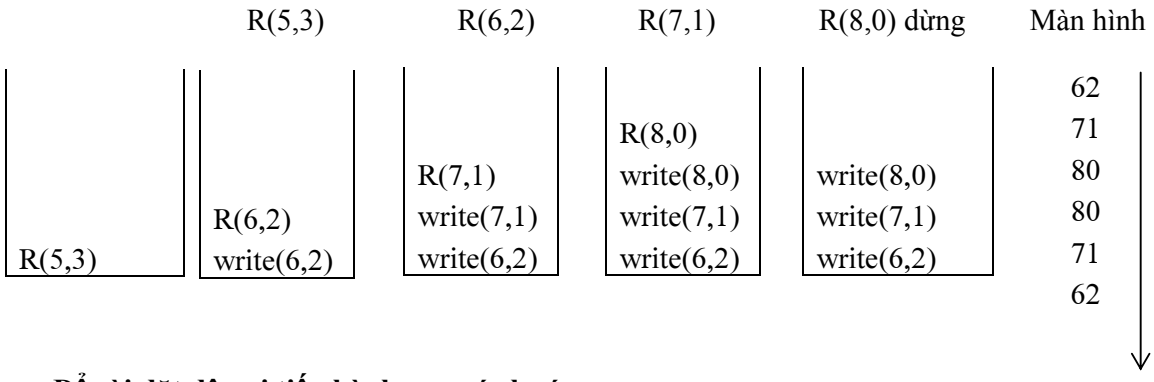
Phần đệ quy thể hiện tính "quy nạp" của lời giải. Phần neo cũng rất quan trọng bởi nó quyết định tới tính dừng của lời giải.

Ví dụ: cho chương trình con đệ qui sau:

```
Proc R(x,y)
If y>0 Then
{       x := x+1
        Y := y-1
        write(x, ‘  ‘,y)
        Call R(x,y)
        write(x, ‘  ‘,y)
}
Return
```

Khi gọi chương trình con, bộ dịch cấp phát một vùng nhớ có cơ chế hoạt động như Stack. Khi một chương trình con được gọi thì địa chỉ của lệnh ngay sau hàm đó và nội dung hiện tại của các biến sẽ được đưa vào vùng nhớ và cứ như thế cho đến khi gặp trường hợp suy biến thì sẽ lấy địa chỉ đầu tiên trong vùng nhớ và giá trị các biến ra thực hiện và quá trình lại tiếp tục cho đến khi vùng nhớ rỗng.

Với lệnh Call R(5,3) thì bộ nhớ hoạt động như sau (để cho tiện ta dùng bộ nhớ lưu trữ ngay lệnh sau hàm được gọi)



Để cài đặt đệ qui tiến hành qua các bước sau:

- Xác định đầu vào và đầu ra từ đó xác định tên chương trình con và tham số hình thức của nó
- Xác định trường hợp suy biến, trường hợp đặc biệt của bài toán
- Phân tích bài toán để xác định trường hợp chung của bài toán (đưa bài toán về dạng cùng loại nhưng nhỏ hơn)

Ví dụ: Định nghĩa đệ qui n! như sau:

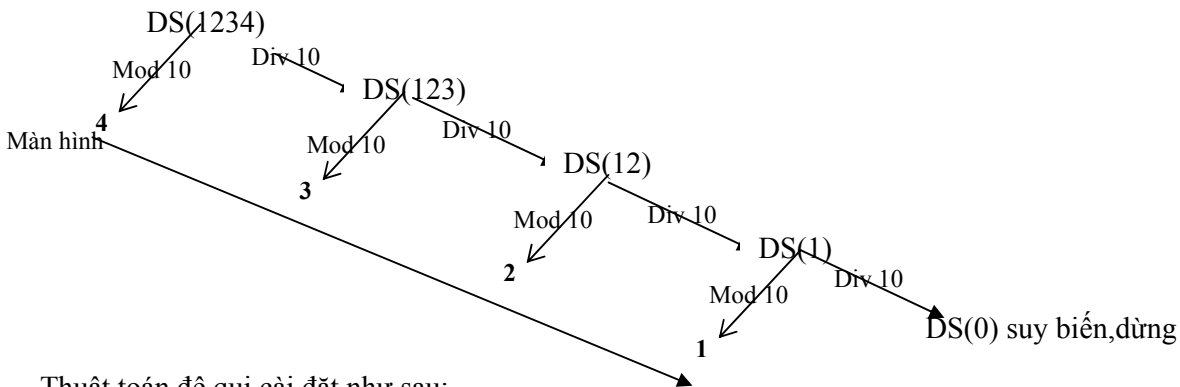
0!=1

n!=(n-1)! * n

Như vậy, tính n! = (n-1)!*n=(n-2)!*(n-1)*n=0!*1*2*….*n=1*1*2*…*n

```
Func GiaiThua(n)
        If (n=0) Then
                GiaiThua := 1
        Else
                GiaiThua :=  n*GiaiThua(n-1)
Return
```

Ví dụ: Xuất đảo ngược một số nguyên dương ra màn hình

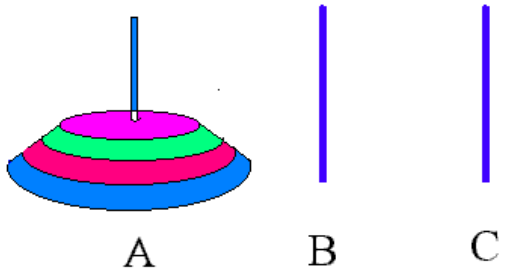


Thuật toán đệ qui cài đặt như sau:

```
Proc XuatDaoSo(n)
        If n >0 Then
        {       write( n Mod 10)
                Call XuatDaoSo ( n Div 10)
        }
Return
```

Ví dụ: Bài toán tháp Hà Nội được phát biểu như sau:

Có ba cọc A,B,C. Khởi đầu cọc A có một số đĩa xếp theo thứ tự nhỏ dần lên trên đỉnh. Bài toán đặt ra là phải chuyển toàn bộ chồng đĩa từ A sang C. Mỗi lần thực hiện chuyển một đĩa từ một cọc sang một cọc khác và không được đặt đĩa lớn nằm trên đĩa nhỏ.



Phân tích bài toán:

Trường hợp 1 đĩa: Chuyển thẳng từ A sang C. Đây là trường hợp suy biến

Trường hợp 2 đĩa: Chuyển 1 đĩa từ A sang B
Chuyển 1 đĩa từ A sang C
Chuyển 1 đĩa từ B sang C

Trường hợp chung n>1 đĩa. Ta coi n-1 đĩa trên như là 1 đĩa và ta áp dụng trong trường hợp 2 đĩa

Chuyển n-1 đĩa từ A sang B, dùng cọc C làm trung gian
Chuyển 1 đĩa từ A sang C
Chuyển n-1 đĩa từ B sang C, dùng cọc A làm trung gian

Thuật toán được lập như sau:

```
Proc HaNoi(n,A,B,C)          // Chuyển n đĩa từ cọc A sang cọc B
If n=1 Then
        chuyển (A,'→',C)
Else    {       Call HaNoi(n-1, A, C, B)
                Call HaNoi(1, A, B, C)
                Call HaNoi(n-1, B, A, C)      }
Return
```

III.3. Hiệu lực của đệ qui

Qua các ví dụ trên, ta có thể thấy đệ quy là một công cụ mạnh để giải các bài toán. Có những bài toán mà bên cạnh giải thuật đệ quy vẫn có những giải thuật lặp khá đơn giản và hữu hiệu. Chẳng hạn bài toán tính giai thừa hay xuất đảo ngược số nguyên. Tuy vậy, đệ quy vẫn có vai trò xứng đáng của nó, có nhiều bài toán mà việc thiết kế giải thuật đệ quy đơn giản hơn nhiều so với lời giải lặp và trong một số trường hợp chương trình đệ quy hoạt động nhanh hơn chương trình viết không có đệ quy.

Có một mối quan hệ khăng khít giữa đệ quy và quy nạp toán học. Cách giải đệ quy cho một bài toán dựa trên việc định rõ lời giải cho trường hợp suy biến (neo) rồi thiết kế làm sao để lời giải của bài toán được suy ra từ lời giải của bài toán nhỏ hơn cùng loại như tế. Tương tự như vậy, quy nạp toán học chứng minh một tính chất nào đó ứng với số tự nhiên cũng bằng cách chứng minh tính chất đó đúng với một số trường hợp cơ sở (thường người ta chứng minh nó đúng với 0 hay đúng với 1) và sau đó chứng minh tính chất đó sẽ đúng với n bất kỳ nếu nó đã đúng với mọi số tự nhiên nhỏ hơn n. Do đó ta không lấy làm ngạc nhiên khi thấy quy nạp toán học được dùng để chứng minh các tính chất có liên quan tới giải thuật đệ quy.

Chẳng hạn: Chứng minh số phép chuyển đĩa để giải bài toán Tháp Hà Nội với n đĩa là 2n-1:

- Rõ ràng là tính chất này đúng với n = 1, bởi ta cần 21 - 1 = 1 lần chuyển đĩa để thực hiện yêu cầu
- Với n > 1; Giả sử rằng để chuyển n - 1 đĩa giữa hai vị trí ta cần 2n-1 - 1 phép chuyển đĩa, khi đó để chuyển n đĩa từ vị trí x sang vị trí y, nhìn vào giải thuật đệ quy ta có thể thấy rằng trong trường hợp này nó cần (2n-1 - 1) + 1 + (2n-1 - 1) = 2n - 1 phép chuyển đĩa. Tính chất được chứng minh đúng với n

Vậy thì công thức này sẽ đúng với mọi n.

Thật đáng tiếc nếu như chúng ta phải lập trình với một công cụ không cho phép đệ quy, nhưng như vậy không có nghĩa là ta bó tay trước một bài toán mang tính đệ quy. Mọi giải thuật đệ quy đều có cách thay thế bằng một giải thuật không đệ quy (khử đệ quy), có thể nói được như vậy bởi tất cả các chương trình con đệ quy sẽ đều được trình dịch chuyển thành những mã lệnh không đệ quy trước khi giao cho máy tính thực hiện.

Việc tìm hiểu cách khử đệ quy một cách "máy móc" như các chương trình dịch thì chỉ cần hiểu rõ cơ chế xếp chồng của các thủ tục trong một dây chuyền gọi đệ quy là có thể làm được. Nhưng muốn khử đệ quy một cách tinh tế thì phải tuỳ thuộc vào từng bài toán mà khử đệ quy cho khéo. Không phải tìm đâu xa, những kỹ thuật giải công thức truy hồi bằng quy hoạch động là ví dụ cho thấy tính nghệ thuật trong những cách tiếp cận bài toán mang bản chất đệ quy để tìm ra một giải thuật không đệ quy đầy hiệu quả.

III.4.Thuật toán quay lui

Giải thuật quay lui có dạng : Duyệt qua tất cả các trường hợp để xác định các bộ x1, x2,…,xn thoả mãn điều kiện B nào đó.

Phương pháp: giả sử đã xác định được i-1 thành phần (x1, x2, …,xi-1), cần xác định thành phần xi. Ta duyệt tất cả các khả năng j có thể có đề cử cho xi. có 2 trường hợp xãy ra:

- Nếu tồn tại 1 khả năng j thì ta xác định xi theo j. Nếu i là trạng thái cuối thì được 1 kết quả, còn nếu i không phải trạng thái cuối thì đi xác định thành phần xi+1
- Nếu không tồn tại khả năng j nào thì ta quay lại xác định thành phần xi-1 khác

Giải thuật có dạng như sau:

```
Proc Try(i)
    For <mỗi khả năng j có thể đề cử cho xi>
        [If < chấp nhận j theo điều kiện B> Then]
            {<xác nhận xi theo j>
            [đánh dấu đã sử dụng j]
            If <i là trạng thái cuối> Then
                <Xác định được 1 kết quả >
            Else
                Call Try(i+1)
```

```
                    [Huỹ đánh dấu đã sử dụng j]
                    }
        Return
```

Ví dụ 1.9: Liệt kê tất cả các dãy nhị phân có độ dài n

Dãy nhị phân kết quả được lưu trữ trong vecto x có n phần tử, mỗi phần tử trong vecto chỉ nhận giá trị 0 hoặc 1.

```
Proc Try (i)
    For j:= 0 To 1
        {   xi := j
            If i=n Then
                Xuất  (vecto x)        //được 1 kết quả
            Else
                Call Try(i+1)
        }
Return
```

Ví dụ 1.10: Liệt kê các hoán vị của n số tự nhiên đầu tiên

Dãy các giá trị hoán vị được lưu trữ trong vécto x có n phần tử, dùng vecto y có n phần tử để xác định giá trị j đã được sử dụng chưa với yj= true là j chưa được sử dụng, yj=false là j đã được sử dụng với j=1,n

```
Proc Try(i)
    For j:=1 To n
        If yj=True Then
            {xi := j
            yj := False     //đánh dấu j đã được sử dụng
            If i=n Then
                Xuất  (vectơ x)        //được 1 kết quả
            Else
                Call Try(i+1)
            yj := True      //huỹ đánh dấu j để sử dụng cho xi+1
            }
    Return
```

IV. ĐÁNH GIÁ THUẬT TOÁN

IV.1. Phân tích thuật toán

Phân tích thuật toán nhằm dự trù chi phí thực hiện thuật toán; đó là các tài nguyên mà thuật toán yêu cầu. Tài nguyên muốn nói ở đây là: bộ nhớ, băng thông, các cổng logic và thời gian tính toán. Tuy nhiên, trên phương diện phân tích lý thuyết, ta chỉ có thể xét tới vấn đề thời gian bởi việc xác định các chi phí khác nhiều khi rất mơ hồ và phức tạp.

Thời gian tính toán của thuật toán thường phụ thuộc vào kích thước đầu vào (size of input). Nếu gọi n là kích thước dữ liệu đưa vào thì thời gian thực hiện của một giải thuật có thể biểu diễn một cách tương đối như một hàm của n: T(n).

Phần cứng máy tính, ngôn ngữ viết chương trình và chương trình dịch ngôn ngữ ấy đều ảnh hưởng tới thời gian thực hiện. Những yếu tố này không giống nhau trên các loại máy, vì vậy không thể dựa vào chúng khi xác định T(n). Tức là T(n) không thể biểu diễn bằng đơn vị thời gian giờ, phút, giây được. Tuy nhiên, không phải vì thế mà không thể so sánh được các giải

thuật về mặt tốc độ. Nếu như thời gian thực hiện một giải thuật là T1(n) = n2 và thời gian thực hiện của một giải thuật khác là T2(n) = 100n thì khi n đủ lớn, thời gian thực hiện của giải thuật T2 rõ ràng nhanh hơn giải thuật T1. Khi đó, nếu nói rằng thời gian thực hiện giải thuật tỉ lệ thuận với n hay tỉ lệ thuận với n2 cũng cho ta một cách đánh giá tương đối về tốc độ thực hiện của giải thuật đó khi n khá lớn.

Ví dụ: Hãy sắp xếp một dãy các con số theo thứ tự không giảm bằng phương pháp sắp xếp chèn (insertion sort)

Mô tả bài toán:

Input : dãy n số (a1, a2,…, an)

Output : một hoán vị (sắp xếp lại) (a1', a2',…, an') của input sao cho:

a1' ≤ a2' ≤ … ≤ an'

Thuật toán:

```
Proc Insertion_sort(A,n)                costs
      For j:=2 To n                     c1
      { key := A[j]                     c2
        i := j-1                        c3
        While i>0 And A[i]>key Do       c4
          {  A[i+1] := A[i]             c5
             i := i-1                   c6
          }
        A[i+1] := key                   c7
      }
   Return
```

Tổng thời gian T(n) để thực hiện thuật toán Insertion_sort là:

$$T(n) = c_1 \sum_{j=2}^{n} \left[c_2 + c_3 + c_4 \sum_{i=1}^{j-1} (c_5 + c_6) + c_7 \right] \qquad (1)$$

Ta xét ba trường hợp:

a) Trường hợp tốt nhất: Dãy A đã được sắp xếp sẵn, nghĩa là A[i] ≤ key. Do đó c5=c6=0.

Vậy $T(n) = c_1 \sum_{j=2}^{n} (c_2 + c_3 + c_7) = c_1 (c_2 + c_3 + c_7)(n-1)$

Do vậy thời gian thực hiện của thuật toán này có thể biểu diễn dưới dạng ***an+b (với a, b = const và phụ thuộc vào các hao phí ci)***, và đây là hàm tuyến tính bậc một theo n.

b) Trong trường hợp xấu nhất: Dãy A được sắp xếp theo thứ tự đảo ngược. Khai triển (1), ta có:

$$T(n) = c_1 \sum_{j=2}^{n} (c_2 + c_3 + c_7) + c_1 c_4 (c_5 + c_6) \sum_{j=2}^{n} (j-1)$$

$$= c_1 (c_2 + c_3 + c_7)(n-1) + c_1 c_4 (c_5 + c_6)(n-1)n/2$$

Do vậy thời gian thực hiện của thuật toán này có thể biểu diễn dưới dạng ***an2 + bn + c (với a, b, c = const và phụ thuộc vào các hao phí ci)***, và đây là hàm tuyến tính bậc hai theo n.

c) Trong trường hợp trung bình: Dãy A có một nửa đã được sắp (nghĩa là một nữa A[i]≤key), và một nửa thì được sắp theo thứ tự ngược lại (nghĩa là một nữa A[i]>key).

Do vậy thời gian hao phí để thực hiện các lệnh trong vòng lặp while sẽ là ***(c5 + c6)/2***; và ta cũng tính được T(n) cũng có dạng là một hàm tuyến tính bậc hai theo n.

IV.2. Xác đinh độ phức tạp tính toán của thuật toán

IV.2.1.Định nghĩa độ phức tạp

Nếu thời gian thực hiện một thuật toán là $T(n) = cn^2$ (với c là một hằng số), thì độ phức tạp tính toán của thuật toán đó có cấp n^2. Hay có thể ký hiệu bằng ký pháp O như sau;

$T(n) = O(n^2)$

Định nghĩa: Cho f(n) và g(n) là hai hàm xác định dương với mọi n. Hàm f(n) được xác định là O(g(n)) nếu tồn tại một hằng số c > 0 và một giá trị n0 sao cho: $f(n) \le c.g(n)$ với mọi $n \ge n0$.

Nghĩa là nếu xét những giá trị $n \ge n0$ thì hàm f(n) sẽ bị chặn trên bởi một hằng số nhân với g(n). Khi đó, nếu f(n) là thời gian thực hiện của một giải thuật thì ta nói giải thuật đó có cấp là g(n).

Ví dụ 1.12: Dùng định nghĩa hệ kí hiệu O, hãy chứng minh 3n +5 = O(n)

Để chứng minh 3n+5=O(n), ta cần phải xác định các hằng dương c, n0 sao cho:

$5n + 3 \le cn \Leftrightarrow 5n + 3 \le 6n \quad ;\forall n \ge 3.$

Vậy phải chọn c=6; n0=3.

Ví dụ 1.13: Dùng kí hiệu O chứng minh an+b=O(n) ; $\forall n>1$

Ta sẽ chứng minh được $an+b \le cn$;$\forall n \ge n0$ nếu chọn c=a+|b| và n0=1.

Ví dụ 1.14: Dùng kí hiệu O chứng minh 2n=O(n!), $\forall n>1$

Ta có $2n=2*2*...*2 \le 2*1*2*3*\ldots*n=2*n!$

Vậy $2n \le 2*n!$. chọn c=2, n0=1 thì theo định nghĩa 2n=O(n!)

Một số độ phức tạp thường sử dụng

Stt	Ký hiệu	Ghi chú
1	O(1)	Độ phức tạp hằng
2	O(lgn)	Thuật toán tìm kiếm nhị phân, cây BST
3	O(lglgn)	Tìm ước chung lớn nhất bằng EUCLID
4	O(n)	Độ phức tạp tuyến tính, duyệt dãy
5	O(nlgn)	Sắp xếp dãy tằng dần bằng QuickSort,HeapSort, dùng cây BST
6	$O(n^2)$	Các phương pháp cổ điển dùng để sắp xếp dãy hoặc duyệt ma trận
7	$O(n^3)$	Nhân 2 ma trận
8	$O(n^k)$	Độ phức tạp đa thức
9	$O(2^n)$	Bài toán tháp Hà Nội, tháp Sài Gòn
10	$O(k^n)$	
11	O(n!)	

Chú ý : lgn hiểu là log2n

IV.2.2.Một số qui tắc xác định độ phức tạp

Việc xác định độ phức tạp tính toán của một giải thuật bất kỳ có thể rất phức tạp. Tuy nhiên, trong thực tế, đối với một số giải thuật ta có thể phân tích bằng một số quy tắc đơn giản.

- Qui tắc tổng

Giả sử một thuật toán gồm hai phần T1 và T2 độc lập nhau, T1 có thời gian thực hiện O(f(n)), T2 có thời gian thực hiện O(g(n)). Lúc đó thời gian thực hiện tiệm cận T của toàn thuật toán là:

$$T = T1 + T2 = O(\max(f(n), g(n)))$$

- Qui tắc nhân

Giả sử một thuật toán gồm hai phần T1 và T2 lồng vào nhau, T1 có thời gian thực hiện O(f(n)), T2 có thời gian thực hiện O(g(n)). Lúc đó thời gian thực hiện T của toàn thuật toán là:

$$T = T1 * T2 = O(f(n) * g(n))$$

- Một số nguyên tắc chung
  - ✓ Các lệnh đọc, ghi, so sánh: thời gian thực hiện theo tiệm cận là O(1).
  - ✓ Lệnh if
    - If (ĐiềuKiện_T1) Then <côngviệc_T2>

    Thời gian thực hiện theo tiệm cận của đoạn lệnh này là: T = T1+T2 = O(max(T1, T2))
    - If(ĐiềuKiện_T0) Then
      <côngviệc_T1>
      Else
      <côngviệc_T2>

    Thời gian thực hiện tiệm cận của đoạn lệnh này là: T = T0+max(T1, T2)
    = O(max(T0, T1, T2))
  - ✓ Các lệnh tuần tự: áp dụng định lý 1.
  - ✓ Các lệnh vòng lặp: nếu vòng lặp thực hiện n lần, ti là thời gian thực hiện lệnh ở lần lặp thứ i. Khi đó, thời gian thực hiện (T) theo tiệm cận của vòng lặp sẽ là:
  - ✓ Các vòng lặp lồng nhau: áp dụng định lý 2.

Ví dụ 1.15: Tính độ phức tạp của thuật toán sắp xếp bằng phương pháp chèn ở ví dụ trước.

Ta có c2=c3=c5=c6=c7=O(1)

c4=

$$\sum_{i=1}^{j-1} \max(c_5, c_6) = \sum_{i=1}^{j-1} 1$$

c1=

$$\sum_{j=2}^{n} \max(c_2, c_3, c_4, c_7) = \sum_{j=2}^{n} \sum_{i=1}^{j-1} 1 = \sum_{j=2}^{n} (n-j) = \sum_{j=2}^{n} (j-1) = \sum_{j=2}^{n} j - \sum_{j=2}^{n} 1 = \frac{n(n+1)}{2} - 1 - (n-1) = \frac{n^2 - n}{2}$$

Theo định nghĩa ký pháp O thì độ phức tạp của thuật toán là O(n2)

- Một số tính chất

Theo định nghĩa về độ phức tạp tính toán ta có một số tính chất:

a) Với P(n) là một đa thức bậc k thì O(P(n)) = O(nk). Vì thế, một thuật toán có độ phức tạp cấp đa thức, người ta thường ký hiệu là O(nk)

b) Với a và b là hai cơ số tuỳ ý và f(n) là một hàm dương thì logaf(n) = logab.logbf(n). Tức là: O(logaf(n)) = O(logbf(n)). Vậy với một thuật toán có độ phức tạp cấp logarit của f(n), người ta ký hiệu là O(logf(n)) mà không cần ghi cơ số của logarit.

c) Nếu một thuật toán có độ phức tạp là hằng số, tức là thời gian thực hiện không phụ thuộc vào kích thước dữ liệu vào thì ta ký hiệu độ phức tạp tính toán của thuật toán đó là O(1).
d) Một giải thuật có cấp là các hàm như 2n, n!, nn được gọi là một giải thuật có độ phức tạp hàm mũ. Những giải thuật như vậy trên thực tế thường có tốc độ rất chậm. Các giải thuật có cấp là các hàm đa thức hoặc nhỏ hơn hàm đa thức thì thường chấp nhận được.

e) Không phải lúc nào một giải thuật cấp O(n2) cũng tốt hơn giải thuật cấp O(n3). Bởi nếu như giải thuật cấp O(n2) có thời gian thực hiện là 1000n2, còn giải thuật cấp O(n3) lại chỉ cần thời gian thực hiện là n3, thì với n < 1000, rõ ràng giải thuật O(n3) tốt hơn giải thuật O(n2). Trên đây là xét trên phương diện tính toán lý thuyết để định nghĩa giải thuật này "tốt" hơn giải thuật kia, khi chọn một thuật toán để giải một bài toán thực tế phải có một sự mềm dẻo nhất định.

IV.2.3.Trường hợp thuật toán đệ qui

Khi một thuật toán chứa lệnh gọi đệ qui lên chính nó, ta có thể sử dụng phép truy toán để mô tả thời gian thực hiện của nó.

Phép truy toán thường là một phương trình hoặc một bất đẳng thức mô tả hàm theo dạng giá trị của nó dựa trên các số liệu được nhập có kích thước nhỏ hơn.

Từ ví dụ thuật toán đệ qui tính n! ở trên ta có biểu thức truy toán như sau:

Gọi T(n) là thời gian thực hiện của thuật toán trên. Ta có

$$\begin{cases} T(0) = 1 & ; \text{khi } n=0 \\ T(n) = T(n-1) + 1 & ; \text{khi } n>0 \end{cases}$$

Từ ví dụ 1.7 (thuật toán tháp Hà Nội) ở trên ta có biểu thức truy toán như sau:

Gọi T(n) là thời gian thực hiện của thuật toán trên. Ta có

$$\begin{cases} T(1) = 1 & ; \text{khi } n=1 \\ T(n) = 2T(n-1) + 1 & ; \text{khi } n>1 \end{cases}$$

Để giải hệ thức truy toán thông thường sử dụng phương pháp thay thế

Ví dụ 1.16: Tìm độ phức tạp của thuật toán được biểu diễn bằng hệ thức truy toán sau:

$$\begin{cases} T(0) = 1 & ; \text{khi } n=0 \\ T(n) = T(n-1) + 1 & ; \text{khi } n>0 \end{cases}$$

Ta có T(n)=T(n-1)+1=T(n-2)+1+1=T(n-3)+3=…=T(n-n)+n=T(0)+n=n+1=O(n)

Ví dụ 1.17: Tìm độ phức tạp của thuật toán được biểu diễn bằng hệ thức truy toán sau:

$$\begin{cases} T(1) = \theta(1) & ; \text{khi } n=1 \\ T(n) = 2T(n-1) + 1 & ; \text{khi } n>1 \end{cases}$$

Ta có T(n)=2T(n-1)+1

T(n-1)=2T(n-2)+1
=> T(n)=2(2T(n-2)+1)+1=22T(n-2) + 21 + 20
T(n-2)=2T(n-3)+1
=>T(n)=23T(n-3) + 22 + 21 + 20
… T(n)=2n-1T(1) + 2n-2 + …+ 21= 2n-1 + 2n-2 + …+ 21=2n-1=O(2n)

Ví dụ 1.18: Tìm độ phức tạp của thuật toán được biểu diễn bằng hệ thức truy toán sau:

$$\begin{cases} T(1) = 0 & ; \text{khi } n=1 \\ T(n) = 2T(n/2) + n & ; \text{khi } n>1 \end{cases}$$

Ta có T(n)=2T(n/2)+n
T(n/2)=2T(n/4) + n/2
=>T(n)=22T(n/22) + 2n
… => T(n)=2kT(n/2k) + kn
đặt n=2k ta có T(n)=2kT(1) + kn .
ta có n=2k =>k=lgn
Vậy T(n)=nlgn=O(nlgn)

CHƯƠNG 2

DANH SÁCH

I. KHÁI NIỆM DANH SÁCH

Danh sách là một tập hợp hữu hạn các phần tử (Element) có cùng một kiểu. Ta biểu diễn danh sách như là một chuỗi các phần tử của nó: a_1, a_2, ... , a_n với $n \geq 0$. Nếu n=0 ta nói danh sách rỗng (empty list). Nếu $n > 0$ ta gọi a_1 là phần tử đầu tiên và a_n là phần tử cuối cùng của danh sách. Số phần tử của danh sách ta gọi là độ dài của danh sách.

Một tính chất quan trọng của danh sách là các phần tử của danh sách có thứ tự tuyến *tính theo vị trí (position) xuất hiện của các phần tử. Ta nói* a_i *đứng trước* a_{i+1}*, với i=1,* n-1; Tương tự ta nói ai là phần tử đứng sau a_{i-1},với i =2,n. Ta cũng nói a_i là phần tử tại vị trí thứ i, hay phần tử thứ i của danh sách.

Giả sử danh sách có tên là L, vị trí sau phần tử cuối cùng trong danh sách L là ENDLIST(L). Các thao tác thông thường trên danh sách là:

- INSERT_LIST(x,p,L) xen phần tử x vào danh sách L tại vị trí p. với $1 \leq p \leq$ ENDLIST(L)
- LOCATE(x,L) thực hiện việc định vị phần tử x trong danh sách L. Locate trả kết quả là vị trí của phần tử x trong danh sách.
- VALUE(p,L) cho kết quả là giá trị của phần tử ở vị trí p trong danh sách L
- DELETE_LIST(p,L) chương trình con thực hiện việc xoá phần tử thứ p của danh sách. Nếu $p \geq$ ENDLIST(L) thì phép toán không được định nghĩa và danh sách L sẽ không thay đổi
- NEXT(p,L) cho kết quả là vị trí của phần tử đi sau phần tử thứ p; nếu p là phần tử cuối cùng trong danh sách L thì NEXT(p,L) cho kết quả là ENDLIST(L). Next không xác định nếu p không phải là vị trí của một phần tử trong danh sách.
- PREVIOUS(p,L) cho kết quả là vị trí của phần tử đứng trước phần tử p trong danh sách. Nếu p là phần tử đầu tiên trong danh sách thì Previous(p,L) không xác định. Previous cũng không xác định trong trường hợp p không phải là vị trí của phần tử nào trong danh sách.
- PRINT_LIST(L) liệt kê các phần tử của L theo thứ tự xuất hiện của chúng trong danh sách.
- EMPTY_LIST(L) cho kết quả TRUE nếu danh sách có rỗng, ngược lại nó cho giá trị FALSE.
- MAKENULL_LIST(L) khởi tạo một danh sách L rỗng.
- FIRST(L) Trả về vị trí đầu tiên trong danh sách. Nếu danh sách rỗng thì FIRST(L) không xác định
- END(L) Trả về vị trí cuối cùng trong danh sách

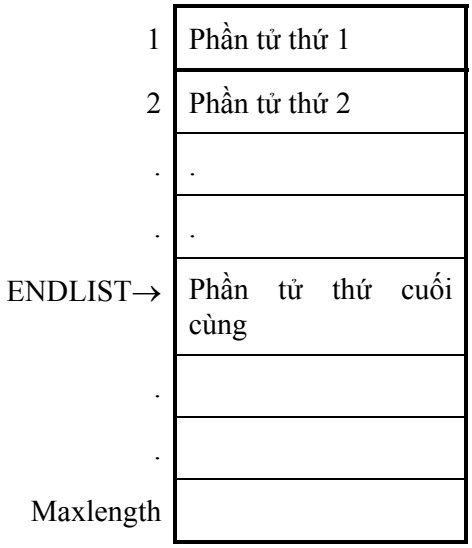
II. BIỂU DIỄN DANH SÁCH TRÊN MÁY TÍNH

Việc cài đặt một danh sách trong máy tính tức là tìm một cấu trúc dữ liệu cụ thể mà máy tính hiểu được để lưu các phần tử của danh sách đồng thời viết các đoạn chương trình con mô tả các thao tác cần thiết đối với danh sách.

III. MẢNG VÀ DANH SÁCH ĐẶC

III.1.Cài đặt mảng

Ta có thể cài đặt danh sách bằng mảng như sau: dùng một mảng để lưu giữ liên tiếp các phần tử của danh sách từ vị trí đầu tiên của mảng. Với cách cài đặt này, dĩ nhiên, ta phải ước lượng số phần tử của danh sách để khai báo số phần tử của mảng cho thích hợp. Dễ thấy rằng số phần tử của mảng phải được khai báo không ít hơn số phần tử của danh sách. Nói chung là mảng còn thừa một số chỗ trống. Mặt khác ta phải lưu giữ độ dài hiện tại của danh sách, độ dài này cho biết danh sách có bao nhiêu phần tử và cho biết phần nào của mảng còn trống như trong hình vẽ. Ta định nghĩa vị trí của một phần tử trong danh sách là chỉ số của mảng tại vị trí lưu trữ phần tử đó.



Các khai báo cần thiết

Danh sách được cài đặt bằng mảng gồm có 2 thành phần đó là mảng A và số phần tử của danh sách trong mảng là Last. Cú pháp cụ thể khai báo như sau:

```
Typedef  LIST=Record
        Kiểu_Mảng    A
        Integer      Last
End Record
```

III.2. Các thao tác trên danh sách

- **Khởi tạo danh sách rỗng MAKENULL_LIST(L)**

Danh sách rỗng có độ dài bằng 0. Theo cài đặt ở trên, biến Last chỉ vị trí của phần tử cuối cùng trong danh sách và đó cũng độ dài hiện tại của danh sách, vì vậy để khởi tạo danh sách rỗng ta chỉ việc gán Last = 0.

```
proc MAKENULL_LIST(L)
L(Last) :=0
 Return
```

- **Kiểm tra danh sách rỗng empty_LIST(L)**

Danh sách rỗng nếu độ dài của danh sách bằng 0.

```
func empty_LIST(L)
EMPTY_LIST :=L(Last)=0
Return
```

- **Xen một phần tử vào danh sách insert_list(x,p,L)**

Khi xen phần tử x vào vị trí p của danh sách ta có mấy khả năng sau:

- ✓ Trường hợp 1: Mảng đầy không thực hiện được
- ✓ Trường hợp 2: p<1 hoặc p>Last +1 cũng không thực hiện được
- ✓ Trường hợp 3: 1≤p≤Last +1 thì
  - Dời các phần tử từ vị trí p đến cuối danh sách xuống 1 vị trí.
  - Độ dài danh sách tăng 1.
  - Đưa phần tử mới vào vị trí p

```
Proc insert_list(x,p,L)
    {If L(last)>=Maxlength Then
            write('Lỗi: danh sách đầy')
    Else If (p>L(last)+1) or (p<1) Then
            write('Lỗi: vị trí không hợp lệ')
    Else
    { //dời các phần tử từ vị trí p đến cuối danh sách xuống 1 vị trí
            For q := L(Last) To p step -1
                    L(A[q+1]) := L(A[q])
            L(Last) := L(Last)+1  //độ dài danh sách tăng lên 1
            L(A[p]) := x          //đặt x vào vị trí p
               }
 Return
```

- **Xoá một phần tử của danh sách delete_list(p,L)**

Khi xoá một phần tử, ta có những khả năng sau :

Trường hợp 1 : p<1 hoặc p>Last thì không hợp lệ, không xoá

Trường hợp 2 : 1≤p≤Last thì việc xoá một phần tử của danh sách ta làm công việc ngược lại với xen nghĩa là phải dời các phần tử từ vị trí p+1 đến cuối danh sách lên một vị trí và độ dài danh sách giảm đi 1 phần tử.

```
Proc delete_list(p,L)
If (p>L(Last)) or (p<1) Then
write('Lỗi: vị trí của phần tử xoá không hợp lệ')
ELSE {
    //dời các phần tử từ vị trí p+1 đến cuối danh sách lên 1 vị trí}
```

```
For q := p+1 To L(Last)
L(A[q-1]) := L(A[q])
L(Last) := L(Last)-1        // giảm kích thước mảng đi 1 phần tử
   }
Return
```

- **Định vị một phần tử trong danh sách locate(x, L)**

Để định vị một phần tử x trong danh sách, ta tiến hành duyệt tìm từ đầu danh sách. Nếu tìm thấy x thì vị trí của phần tử tìm thấy được trả vê, nếu không tìm thấy thì hàm trả về gái trị 0. Trong trường hợp có nhiều phần tử cùng giá trị x trong danh sách thì vị trí của phần tử được tìm thấy đầu tiên được trả về.

```
func locate(x,L)
    p:= 1
    while p<=L(Last) And L(A[p])<>x) do p:= p+1
    If p<=L(Last) Then
            LOCATE:= p
    Else
            LOCATE:=0
Return
```

- **Lấy giá trị tại vị trí p trong danh sách VALUE(p,L)**

Trường hợp p<1 hoặc P>Last không thực hiện được

```
Func VALUE(p,L)
    If p<1 Or P>L(Last) Then
            Write(' Lỗi, không có vị trí này')
    Else
            VALUE := L(A[p])
Return
```

- **Xác định vị trí tiếp theo của phần tử ở vị trí p NEXT(p,L)**

Trường hợp p<1 hoặc P>Last thì không có kết quả

```
Func NEXT(p,L)
    If p<1 Or P>L(Last) Then
            Write(' Lỗi, không có vị trí này')
    Else
            NEXT := p+1
Return
```

- **Xác định vị trí trước phần tử ở vị trí p PREVIOUS(p,L)**

Trường hợp p≤1 hoặc P>Last thì không có kết quả

```
Func PREVIOUS(p,L)
    If p<=1 Or P>L(Last) Then
            Write(' Lỗi, không có vị trí này')
    Else
            NEXT := p-1
Return
```

- **Liệt kê các phần tử của L PRINT_LIST(L)**

```
Proc PRINT_LIST(L)
    For p := 1 To L(Last)
```

```
            Write(VALUE(p,L))
    Return
```

- **Xác định vị trí đầu tiên trong danh sách FIRST(L)**

Trường hợp danh sách rỗng không xác định

```
    Func FIRST(L)
        If L(Last)<1 Then
            Write(' Lỗi, danh sách rỗng')
        Else
            FIRST := 1
    Return
```

- **Xác định vị trí cuối cùng trong danh sách END(L)**

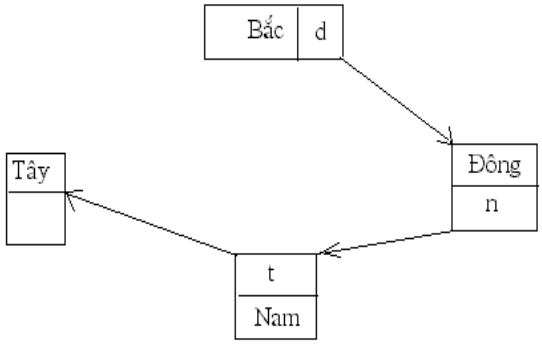
Trường hợp danh sách rỗng không xác định

```
    Func END(L)
    If L(Last)<1 Then
            Write(' Lỗi, danh sách rỗng')
        Else
            END := L(Last)
    Return
```

IV. DANH SÁCH LIÊN KẾT

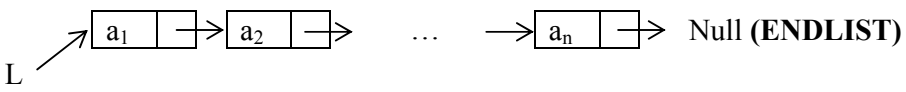
Cách khác để cài đặt danh sách là dùng con trỏ để liên kết các ô chứa các phần tử trong danh sách, các ô này có thể nằm liên tục hoặc rời nhau trong bộ nhớ. Trong danh sách này, mỗi ô gồm có 2 thành phần: Thành phần thứ nhất 1 được dùng để lưu trữ thông tin cần xử lý (như ai trong mảng), thành phần còn lại là các con trỏ dùng để lưu trữ địa chỉ các của ô khác mà nó trỏ đến

Giả sử 1 lớp có 4 bạn: Đông, Tây, Nam, Bắc có địa chỉ lần lượt là d,t,n,b. Giả sử: Đông có địa chỉ của Nam, Tây không có địa chỉ của bạn nào, Bắc giữ địa chỉ của Đông, Nam có địa chỉ của Tây (xem hình).



Như vậy, nếu ta xét thứ tự các phần tử bằng cơ chế chỉ dến này thì ta có một danh sách: Bắc, Đông, Nam, Tây. Hơn nữa để có danh sách này thì ta cần và chỉ cần giữ địa chỉ của Bắc.

Như vậy 1 danh sách liên kết có n phần tử a1, a2, …,an có thể mô phỏng qua hình vẽ sau:



Danh sách liên kết

Nút đầu tiên trong danh sách được gọi là chốt của danh sách nối đơn (Head). Để duyệt danh sách nối đơn, ta bắt đầu từ chốt, dựa vào trường liên kết để đi sang nút kế tiếp, đến khi gặp giá trị đặc biệt (duyệt qua nút cuối) thì dừng lại.

IV.1.Danh sách nối đơn

Danh sách nối đơn gồm các nút được nối với nhau theo một chiều. Mỗi nút là một bản ghi (record) gồm hai trường:

- ✓ Trường thứ nhất chứa giá trị lưu trong nút đó
- ✓ Trường thứ hai chứa liên kết (con trỏ) tới nút kế tiếp, tức là chứa một thông tin đủ để biết nút kế tiếp nút đó trong danh sách là nút nào, trong trường hợp là nút cuối cùng (không có nút kế tiếp), trường liên kết này được gán một giá trị đặc biệt.

Nút đầu tiên trong danh sách được gọi là chốt của danh sách nối đơn (Head). Để duyệt danh sách nối đơn, ta bắt đầu từ chốt, dựa vào trường liên kết để đi sang nút kế tiếp, đến khi gặp giá trị đặc biệt (duyệt qua nút cuối) thì dừng lại.

- ➢ **Các khai báo cần thiết là**

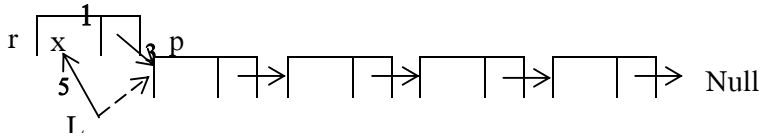
```
Typedef Node= record
    Kiểu_lưu_trữ     Info  //Trường Info dùng chứa thông tin cần
    Con_Trỏ_Kiểu_Node   Link   //Trường Link là con trỏ trỏ đến phần tử kế tiếp
                               trong danh sách
End Record
Con_trỏ_Kiểu_Node L     //Khai báo con trỏ L trỏ vào đầu danh sách
```

Chú ý:

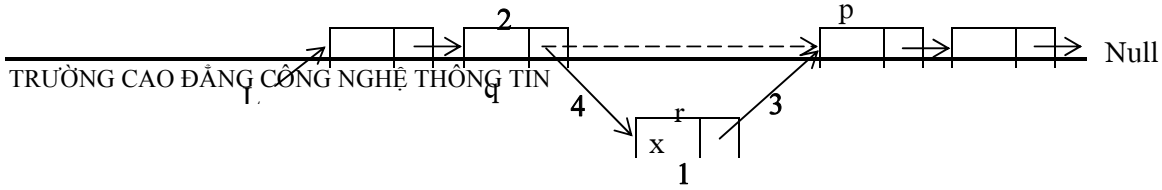
- ✓ Để quản lí danh sách khai báo một biến trỏ L (hoặc Head), dùng để giữ địa chỉ ô chứa phần tử đầu tiên của danh sách. Biến này gọi là chỉ điểm đầu danh sách.
- ✓ Con trỏ đặc biệt Null
- ✓ Danh sách rỗng L = Null
- ✓ Danh sách luôn có gía trị Null để báo kết thúc danh sách
- ✓ Trường Next của mỗi Node chỉ chứa địa chỉ Node sau nó
- ✓ Cấp phát 1 Node cho con trỏ p : New(p)
- ✓ Huỹ một Node được trỏ bởi p: Delete(p)

- ➢ **Chèn 1 giá trị x vào vị trí Node được trỏ bởi p INSERT_LIST(x,p,L)**

Trường hợp p trỏ đầu danh sách



Trường hợp p trỏ vào Node không phải đầu danh sách



```
Proc INSERT_LIST(x,p,L)
    //1. Xin cấp phát 1 Node được trỏ bởi r và gán giá trị x cho trường Info
            New(r)
            Info(r) := x
            Link(r) := p    // mũi tên số 3
If L=p Then         // trường hợp p trỏ vào Node dầu tiên trong danh sách
L :=  p     // mũi tên số 5
Else
{   q :=  PREVIOUS(p,L)   //2. Cho con trỏ q trỏ vào Node trước p
Link(q) :=  r                   // 4. chèn Node có giá trị x vào danh sách L

}
Return
```

- **Xác định địa chỉ của Node có giá trị x LOCATE(x,L)**

Nếu có giá trị x trong danh sách thì kết quả là địa chỉ Node đó, nếu không có x trong danh sách thì kết quả là Null

```
Func LOCATE(x,L)
    p :=  L
    While Info(p)<> x And p<> Null Do
            p := Link(p)
    LOCATE := p
Return
```

- **Xác định giá trị của Node được trỏ bởi p VALUE(p,L)**

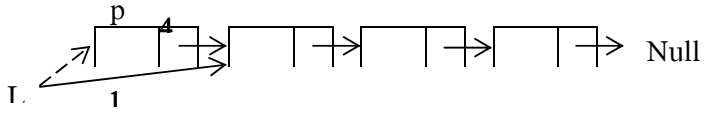
Trường hợp p=Null không xác định được giá trị

```
Func VALUE(p,L)
    If p=Null Then
            Write(' Lỗi,  Không xác định')
    Else
            VALUE := Info(p)
Return
```

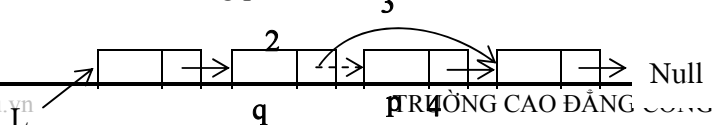
- **Xoá Node được trỏ bởi p DELETE_LIST(p,L)**

Trường hợp p=Null thì không xoá được

Trường hợp p trỏ đầu danh sách



Trường hợp p trỏ vào Node không phải đầu danh sách



```
Proc DELETE_LIST(p,L)
    If p=Null Then
            Write(' Lỗi, Không xoá ')
    Else
            {If  p=L Then
                    L := Link(p) // L trỏ vào Node thứ 2
            Else
                    {q := PREVIOUS(p,L) //Cho con trỏ q trỏ vào Node trước P
                    Link(q) :=  Link(p) //Nối với Node sau p trong danh sách L
                    }
            Delete (p)       // Giải phóng Node được trỏ bởi p
            }
Return
```

- **Xác định địa chỉ Node sau Node được trỏ bởi p NEXT(p,L)**

Trường hợp p=Null không xác định

```
Func NEXT(p,L)
    If p=Null Then
            Write(' Lỗi, Không xác định')
    Else
            NEXT := Link(p)
Return
```

- **Xác định địa chỉ Node trước Node được trỏ bởi p PREVIOUS(p,L)**

Trường hợp p trỏ đầu danh sách thì không xác định được

```
Func PREVIOUS(P,L)
    If p=L Then
            Write('Lỗi, Không xác đinh')
    Else
            {q :=  L
            While Link(q)<>p Do          q := Link(q)
            PREVIOUS := q
            }
Return
```

- **Liệt kê các phần tử trong danh sách PRINT_LIST(L)**

```
Proc PRINT_LIST(L)
    p := L
    While p<>Null Do
            {write(Info(p))
            p := Link(p)    }
Return
```

- **Kiểm tra danh sách có rỗng không EMPTY_LIST(L)**

```
Func EMPTY_LIST(L)
    EMPTY_LIST := L=Null
Return
```

- **Khởi tạo một danh sách rỗng MAKENULL_LIST(L)**

```
Proc MAKENULL_LIST(L)
    L := Null
Return
```

➢ **Xác định địa chỉ của Node đầu danh sách FIRST(L)**

Trả về vị trí đầu tiên trong danh sách. Nếu danh sách rỗng thì FIRST(L) không xác định

```
Func FIRST(L)
    If L=Null Then
            Write('Lỗi, Không xác định')
    Else
            FIRST := L
Return
```

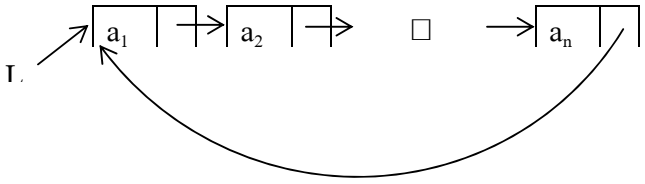
➢ **Xác định địa chỉ của Node cuối danh sách END(L)**

Trả về vị trí cuối cùng trong danh sách. Nếu danh sách rỗng thì END(L) không xác định

```
Func END(L)
    If L=Null Then
            Write('Lỗi, Không xác định')
    Else
            {       p := L
                    While Link(p)<>Null Do p := Link(p)
                    END :=  p
            }
Return
```

IV.2.Danh sách nối vòng

Danh sách liên kết đơn nối vòng là danh sách liên kết mà trường Link của Node cuối chứa địa chỉ Node đầu tiên.



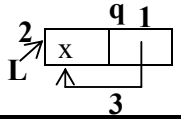
Chú ý:

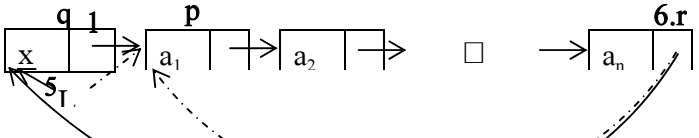
- ✓ Danh sách rỗng: L=Null
- ✓ Danh sách 1 Node: Link(L)=L

Các thao tác trên danh sách liên kết đơn nối vòng cũng như trên danh sách liên kết đơn. Để dễ hình dung ta thêm ký tự 'C' trước tên mỗi thao tác Tuy nhiên việc xây dựng các thao tác có hơi khác một ít do tính chất đặc biệt của danh sách vòng.

➢ **Chèn 1 giá trị x vào Node có địa chỉ là p C_INSERT_LIST(x,p,L)**

Trường hợp p=l=Null thì trở thành danh sách 1 Node





Trường hợp p=l ≠ Null là chèn đầu danh sách Nối vòng

Trường hợp còn lại thì giống trong trường hợp danh sách liên kết đơn

```
Proc C_INSERT_LIST(x,p,L)
    New(q)                  // cấp phát 1 Node
    Info(q) := x
    If p=Null And L=Null Then
    {       L := q          //Danh sách 1 Node
            Link(L) := L
    }
    Else
    {                       // Cho r trỏ  vào Node trước p trong danh sách
            r := C_PREVIOUS(p,L)
            Link(r) := q
            Link(q) := p
            If L=p Then
            L := q
    }
Return
```

- **Xác định địa chỉ Node có giá trị x C_LOCATE(x,L)**

```
Func C_LOCATE(x,L)
    q :=Null
    r := L
    Do{
            If Info(r)=x Then
            {       q=r
                    Break           //dừng vòng lặp
            }
            r := Link(r)
    }while (r<>L)
    C_LOCATE := q
Return
```

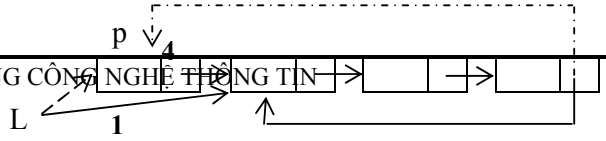
- **Xác định giá trị của Node được trỏ bởi p C_VALUE(p,L):** giống danh sách liên kết đơn

- **Xoá Node được trỏ bởi p C_DELETE_LIST(p,L)**

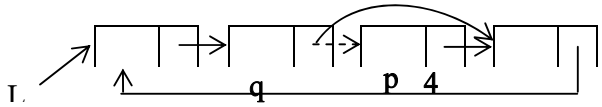
Trường hợp p=Null thì không xoá được

Trường hợp p=L có 1 Node thì thành danh sách rỗng

Trường hợp p trỏ đầu danh sách



Trường hợp p trỏ vào Node không phải đầu danh sách




```
Proc C_DELETE_LIST(p,L)
    If p=Null Then
            Write(' Lỗi, Không xoá ')
    Else
            {If  p=L Then
                    If Link(L)=L Then
                            L:= Null
                    Else
                    {q := C_PREVIOUS(p,L)       //q trỏ vào Node cuối
                    L := Link(p)                //L trỏ vào Node thứ 2
                    Link(q) := L                //Tạo danh sách vòng
                    }
            Else
                    {q := C_PREVIOUS(p,L) //Cho con trỏ q trỏ vào Node trước P
                    Link(q) := Link(p)      //Nối với Node sau p trong danh sách L
                    }
            Delete (p)                      // Giải phóng Node được trỏ bởi p

            }
Return
```

- **Xác định địa chỉ Node sau Node được trỏ bởi p C_NEXT(p,L):** giống danh sách liên kết đơn

- **Xác định địa chỉ Node trước Node được trỏ bởi p C_PREVIOUS(p,L)**

Trường hợp p =Null thì không xác định được

```
Func C_PREVIOUS(P,L)
    If p=Null Then
            Write('Lỗi, Không xác đinh')
    Else
            {       q := L
                    While Link(q)<>p Do         q := Link(q)
                    C_PREVIOUS := q
            }
Return
```

- **Liệt kê các phần tử trong danh sách vòng C_PRINT_LIST(L)**

```
Proc C_PRINT_LIST(L)
    p := L
```

```
Do{     write(Info(p))
        p := Link(p)
    }while(p<>L)
Return
```

- **Kiểm tra danh sách vòng có rỗng không C_EMPTY_LIST(L):** giống danh sách liên kết đơn
- **Xác định địa chỉ của Node đầu danh sách vòng C_FIRST(L):** giống danh sách liên kết đơn
- **Xác định địa chỉ của Node cuối danh sách vòng C_END(L)**

Trả về vị trí cuối cùng trong danh sách. Nếu danh sách rỗng thì END(L) không xác định

```
Func END(L)
    If L=Null Then
            Write('Lỗi, Không xác định')
    Else
            {       p := L
                    While Link(p)<>L Do p := Link(p)
                    C_END :=  p
            }
Return
```

IV.3. Danh sách nối kép

Một số ứng dụng đòi hỏi chúng ta phải duyệt danh sách theo cả hai chiều một cách hiệu quả. Chẳng hạn cho phần tử X cần biết ngay phần tử trước X và sau X một cách mau chóng. Trong trường hợp này ta phải dùng hai con trỏ, một con trỏ chỉ đến phần tử đứng sau (next), một con trỏ chỉ đến phần tử đứng trước (previous). Với cách tổ chức này ta có một danh sách liên kết kép. Dạng của một danh sách liên kép như sau:

Hình: danh sách liên kết kép

Cấu trúc dữ liệu cho danh sách liên kết kép như sau:

```
Typedef D_Node= record
    Kiểu_lưu_trữ    Info                    //Trường Info dùng chứa thông tin cần
    Con_Trỏ_Kiểu_D_Node   Next, Previous //Next trỏ pt sau, Previous trỏ pt trước
End Record
```

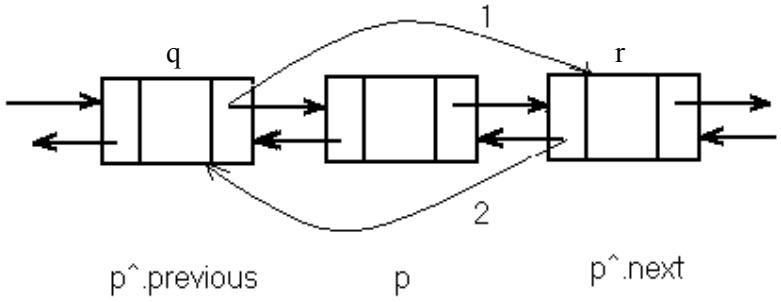
Giả sử DL (Double List) là con trỏ quản lý danh sách liên kết kép.

- **Khởi tạo danh sách rỗng:** *DL:= Null*
- **Kiểm tra danh sách liên kết kép rỗng**: *kiểm tra DL có bằng Null không ?*
- **Xoá một phần tử trong danh sách liên kết kép**

Để xoá một phần tử tại vị trí con trỏ p trong danh sách liên kết kép được trỏ bởi DL, ta phải chú ý đến mấy trường hợp sau:

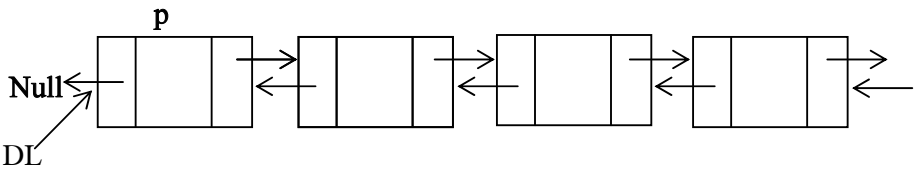
- ✓ Danh sách rỗng, tức là DL=Null không thực hiện

- ✓ Trường hợp danh sách khác rỗng, tức là DL<>Null, ta phải phân biệt hai trường hợp
  - Ô bị xoá không phải là ô được trỏ bởi DL



q := p(Previous) , r := p(Next)
q(Next) := r
If r<> Null Then r(Previous) := q
Delete(p)

  - Xoá ô đang được trỏ bởi DL, tức là p=DL: ngoài việc cập nhật lại các con trỏ để nối kết các ô trước và sau p ta còn phải cập nhật lại DL.

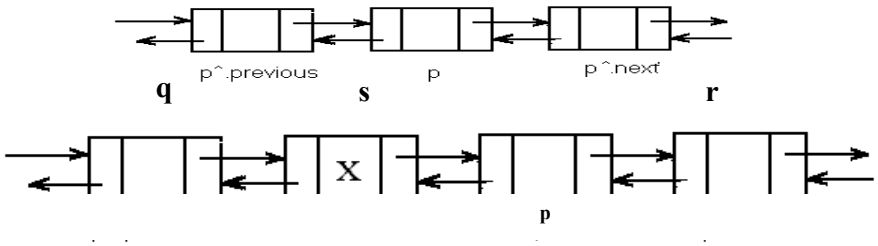


Trường hợp 1 Node: DL := Null, Delete (p)
Trường hợp còn lại: DL := DL(Next), DL(Previous) := Null, Delete(p)

➢ **Thêm một phần tử vào danh sách lên kết kép**

Để thêm một phần tử x vào vị trí p trong danh sách liên kết kép được trỏ bởi DL, ta cũng cần phân biệt mấy trường hợp sau:

- ✓ Danh sách rỗng, tức là DL=p = Null trở thành danh sách kép 1 Node
  New (DL)
  DL(Info) := x
  DL(Previous) := Null
  DL(Next) := Null
- ✓ Nếu Dl<>nil, sau khi thêm phần tử x vào vị trí p



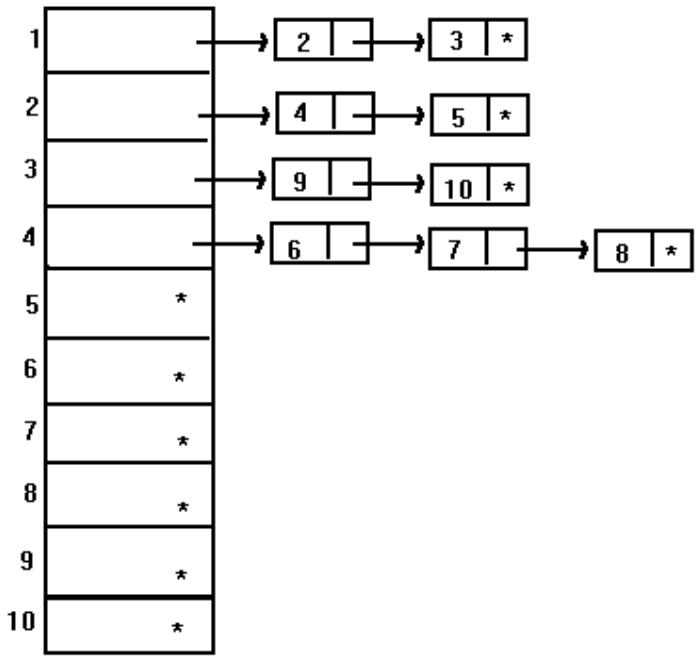
New(s), s(Info) := x,
q := p(Previous), q(Next) := s, s(Next) := q
r := p(Next), s(Next) := r

If r<>Null Then r(Previous) := s
Delete(p)

IV.4. Đa danh sách

Cấu trúc đa danh sách là hình thức kết hợp các kiểu danh sách với nhau trong cấu trúc danh sách. Chẳng hạn kết hợp một danh sách đặc với một danh sách liên kết đơn, hoặc có thể kết hợp 2 danh sách liên kết đơn với nhau để tạo thành danh sách đa chiều.

Ví dụ mô tả hình ảnh kết hợp một danh sách đặc với một danh sáh liên kết đơn như hình vẽ

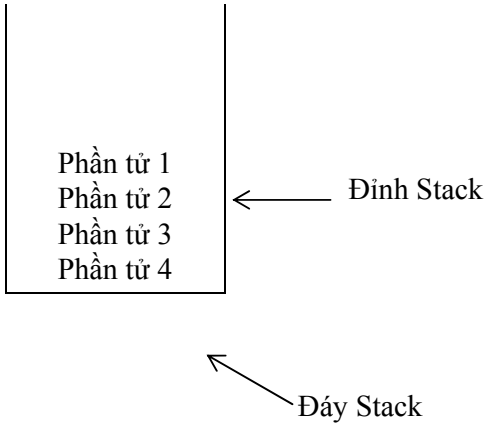


V. NGĂN XẾP

V.1. Định nghĩa ngăn xếp

Ngăn xếp (Stack) là một danh sách mà ta giới hạn việc thêm vào hoặc loại bỏ một phần tử chỉ thực hiện tại một đầu của danh sách, đầu này gọi là đỉnh (TOP) của ngăn xếp.

Một chồng đĩa đặt trên bàn cho ta hình ảnh trực quan của ngăn xếp, muốn thêm vào chồng đó 1 đĩa ta để đĩa mới trên đỉnh chồng, muốn lấy các đĩa ra khỏi chồng ta cũng phải lấy đĩa trên trước. Như vậy ngăn xếp là một cấu trúc có tính chất "vào sau - ra trước" hay LIFO (last in - first out).



- **Các thao tác cơ bản trên ngăn xếp:**
  - ✓ CREAT_STACK(S): tạo một ngăn xếp S rỗng
  - ✓ TOP(S): trả về giá trị của phần tử tại đỉnh ngăn xếp. Nếu ngăn xếp rỗng thì hàm không xác định.
  - ✓ POP(S): Lấy một phần tử tại đỉnh ngăn xếp.
  - ✓ PUSH(x,S): thêm một phần tử x vào đầu ngăn xếp.
  - ✓ EMPTY_STACK(S): kiểm tra ngăn xếp rỗng, Hàm cho kết quả TRUE nếu ngăn xếp rỗng và FALSE trong trường hợp ngược lại.
- **Ví dụ thuật toán đổi một số từ hệ 10 sang hệ 2 sử dụng ngăn xếp**

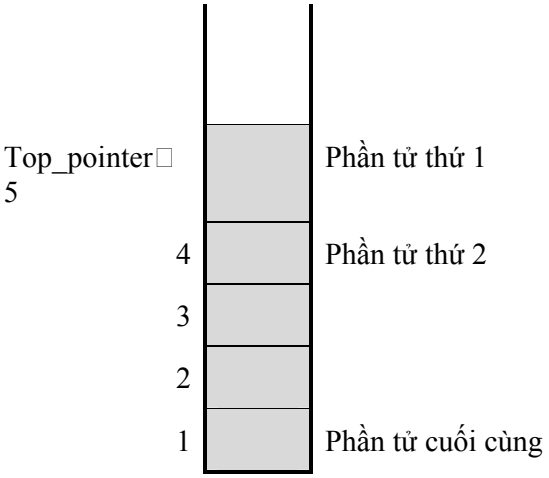
Với $n=19_{10} = 10011_2$, ta sẽ đổi như sau:

n	→	19	9	4	2	1	0
n Mod 2	→	1	1	0	0	1	

```
Proc DoiHe(n)
    CREAT_STACK(S)
    While n>0 Do
            {       PUSH(n Mod 2, S)
                    n := n Div 2
            }
    While Not EMPTY_STACK(S)
            {       write(TOP(S))
                    POP(S)
            }
Return
```

V.2. Cài đặt ngăn xếp bằng mảng

Dùng một mảng để lưu trữ liên tiếp các phần tử của ngăn xếp. Các phần tử đưa vào ngăn xếp bắt đầu từ vị trí có chỉ số thấp nhất của mảng. Ta dùng một biến số nguyên (top_pointer) giữ chỉ số của phần tử tại đỉnh ngăn xếp. Giả sử vùng nhớ tối đa được dùng cho Stack là có MaxLength phần tử.



Khai báo cài đặt

Cài đặt bằng mảng

```
CONST    maxlength=...; {độ dài của mảng}
Typedef
    STACK = record
            Kiểu_Mảng  A          //Mảng A dùng để lưu dữ liệu
            Integer  top_idx      //giữ đỉnh ngăn xếp
    end;
```

- **Tạo ngăn xếp rỗng CREAT_STACK(S)**

Ngăn xếp rỗng thì top_pointer=0

```
Proc CREAT_STACK(S)
    S(top_pointer) := 0
Return
```

- **Kiểm tra ngăn xếp rỗng EMPTY_STACK(S)**

```
 Func EMPTY_STACK(S)
    If S(top_pointer)=0 Then
            EMPTY_STACK := True
    Else EMPTY_STACK := False
 Return
```

- **Trả về phần tử đầu ngăn xếp TOP(S)**

Trường hợp Stack rỗng không có kết quả

```
Func TOP( S)
    If  EMPTY_STACK(S)
            write('lỗi: ngăn xếp rỗng');
    Else
            TOP := S(A[S(top_pointer)]
Return
```

- **Chương trình con xoá một phần tử POP(S)**

Trường hợp Stack rỗng không thực hiện

```
Proc POP(S)
    If  EMPTY_STACK(S) Then
            write('Lỗi: ngăn xếp rỗng')
```

```
        Else
            S(top_pointer) := S(top_pointer)-1
        Return
```

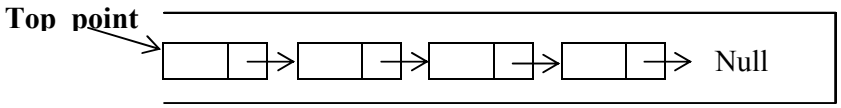
- **Thêm một phần tử vào ngăn xếp PUSH(x,S)**

trường hợp Stack đầy không thực hiện

```
        Proc PUSH(x,S)
            If S(top_pointer)=MaxLength Then
                    Write('Lỗi, Stack đầy')
            Else    {
                            S(top_pointer) := S(top_pointer)+1
                            S(A[S.top_pointer]) := x
                    }
        Return
```

V.3. Cài đặt ngăn xếp bằng danh sách liên kết đơn

Ta mô phỏng một Stack bằng danh sách liên kết như sau:




```
        Typedef Node= record
            Kiểu_lưu_trữ    Info   //Trường Info dùng chứa thông tin cần
            Con_Trỏ_Kiểu_Node   Link   //Trường Link là con trỏ trỏ đến phần tử kế tiếp
trong danh sách
        End Record
        Typedef Stack=Record
            Con_trỏ_Kiểu_Node top_pointer
        End Record
```

- **Tạo ngăn xếp rỗng CREAT_STACK(S)**

Ngăn xếp rỗng thì top_pointer=0

```
        Proc CREAT_STACK(S)
            S(top_pointer) := Null
        Return
```

- **Kiểm tra ngăn xếp rỗng EMPTY_STACK(S)**

```
         Func EMPTY_STACK(S)
            If S(top_pointer)=Null Then
                    EMPTY_STACK := True
            Else EMPTY_STACK := False
         Return
```

- **Trả về phần tử đầu ngăn xếp TOP(S)**

Trường hợp Stack không có kết quả

```
        Func TOP( S)
            If  EMPTY_STACK(S)
```

```
            write('lỗi: ngăn xếp rỗng');
        Else
            TOP := Info(S(top_pointer))
    Return
```

- **Chương trình con xoá một phần tử POP(S)**

Trường hợp Stack rỗng không thực hiện

```
    Proc POP(S)
        If  EMPTY_STACK(S) Then
            write('Lỗi: ngăn xếp rỗng')
        Else
            {p := S(top_pointer)
             S(top_pointer) := Link(S(top_pointer))
            Delete(p)
            }
    Return
```

- **Thêm một phần tử vào ngăn xếp PUSH(x,S)**

```
    Proc PUSH(x,S)
        New(p)
        Info(p) := x
        Link(S(top_pointer)) := p
        S(top_pointer) := p
    Return
```

V.4. Ứng dụng ngăn xếp để khử đệ qui

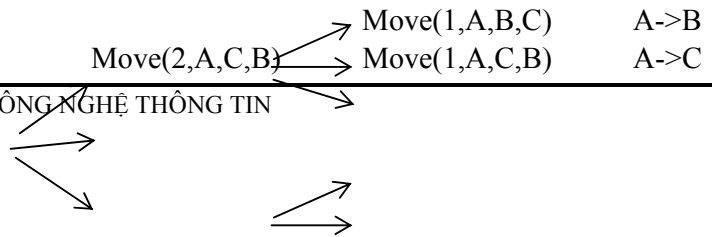
Nếu một chương trình con đệ qui P(x) được gọi từ chương trình chính ta nói chương trình con được thực hiện ở mức 1. Chương trình con này gọi chính nó, ta nói nó đi sâu vào mức 2... cho đến một mức k. Rõ ràng mức k phải thực hiện xong thì mức k-1 mới được thực hiện tiếp tục, hay ta còn nói là chương trình con quay về mức k-1.

Trong khi một chương trình con từ mức i đi vào mức i+1 thì các biến cục bộ của mức i và địa chỉ của mã lệnh còn dang dở phải được lưu trữ, địa chỉ này gọi là địa chỉ trở về. Khi từ mức i+1 quay về mức i các giá trị đó được sử dụng. Như vậy những biến cục bộ và địa chỉ lưu sau được dùng trước. Tính chất này gợi ý cho ta dùng một ngăn xếp để lưu giữ các giá trị cần thiết của mỗi lần gọi tới chương trình con. Mỗi khi lùi về một mức thì các giá trị này được lấy ra để tiếp tục thực hiện mức này. Tóm tắt quá trình:

- Bước 1: Lưu các biến cục bộ và địa chỉ trở về.
- Bước 2: Nếu thoả điều kiện ngừng đệ qui thì chuyển sang bước 3. Nếu không thì tính toán từng phần và quay lại bước 1 (đệ qui tiếp).
- Bước 3: Khôi phục lại các biến cục bộ và địa chỉ trở về.

Ví dụ sau đây minh hoạ việc dùng ngăn xếp để loại bỏ chương trình đệ qui của bài toán tháp Hà Nội như đã xây dựng ở chương 1 (phần đệ qui).

Quá trình thực hiện chương trình con được minh hoạ với ba đĩa (n=3) như sau:

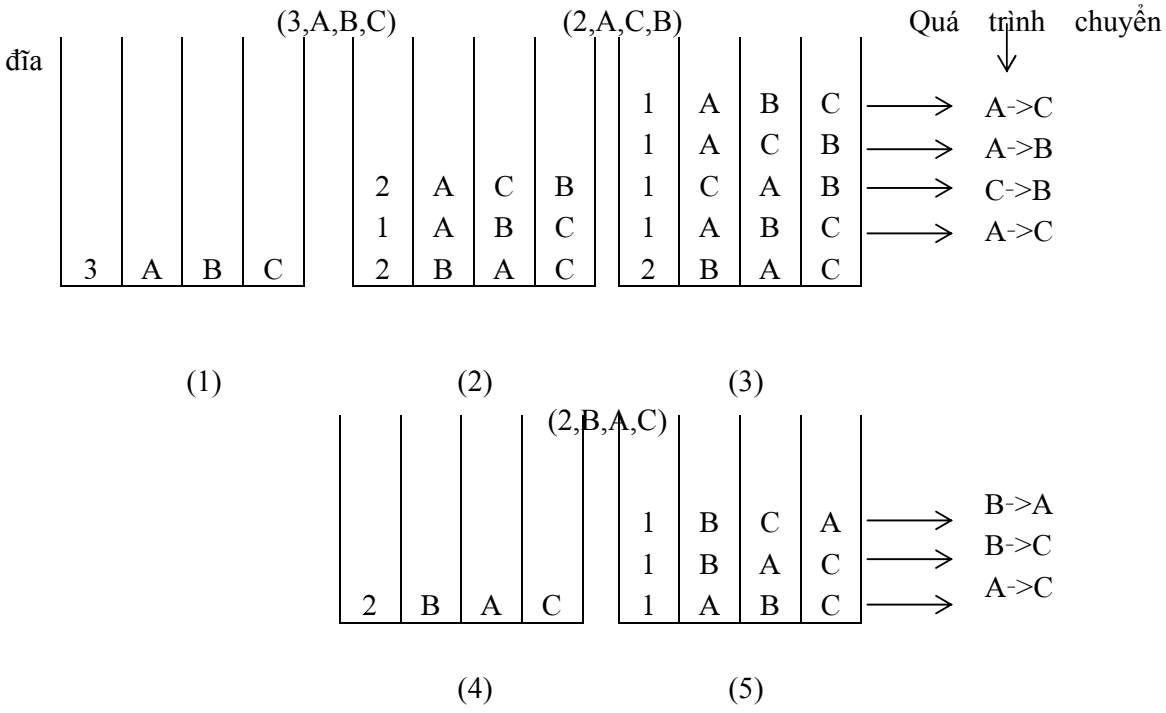


		Move(1,B,C,A)	B->C
Move(3,A,B,C)	Move(1,A,B,C)		A->B
		Move(1,C,A,B)	C->A
	Move(2,C,B,A)	Move(1,C,B,A)	C->B
		Move(1,A,B,C)	A->B
Mức 1	**mức 2**	**mức 3**	

Để khử đệ qui ta phải nắm nguyên tắc sau đây:

- Mỗi khi chương trình con đệ qui được gọi, ứng với việc đi từ mức i vào mức i+1, ta phải lưu trữ các biến cục bộ của chương trình con ở buớc i vào ngăn xếp. Ta cũng phải lưu "địa chỉ mã lệnh" chưa được thi hành của chương trình con ở mức i. Tuy nhiên khi lập trình bằng ngôn ngữ cấp cao thì đây không phải là địa chỉ ô nhớ chứa mã lệnh của máy mà ta sẽ tổ chức sao cho khi mức i+1 hoàn thành thì lệnh tiếp theo sẽ được thực hiện là lệnh đầu tiên chưa được thi hành trong mức i.
- Tập hợp các biến cục bộ của mỗi lần gọi chương trình con xem như là một mẩu tin (activation record).
- Mỗi lần thực hiện chương trình con tại mức i thì phải xoá mẩu tin lưu các biến cục bộ ở mức này trong ngăn xếp.

Như vậy nếu ta tổ chức ngăn xếp hợp lí thì các giá trị trong ngăn xếp chẳng những lưu trữ được các biến cục bộ cho mỗi lần gọi đệ qui, mà còn "điều khiển được thứ tự trở về" của các chương trình con. Ý tưởng này thể hiện trong cài đặt khử đệ qui cho bài toán tháp Hà Nội là: mẩu tin lưu trữ các biến cục bộ của chương trình con thực hiện sau thì được đưa vào ngăn xếp trước để nó được lấy ra dùng sau. Có thể mô phỏng quá trình thực hiện bài toán bằng cách dùng Stack với n=3 như sau:



Xây dựng kiểu dữ liệu để lưu trữ 1 mẫu tin (1 phần tử trong Stack)

Typedef Phần_Tử=record

```
            Integer  số_đĩa
            Char nguồn, trung_gian, đích
        End Record
```

Thuật toán Tháp HÀ NỘI khử đệ qui dùng ngăn xếp:

```
Proc  HANOI(n,A,B,C);
    CREAT_STACK(S);
    PUSH([n,A,B,C],S);
    While Not EMPTY_STACK(S) Do
    {       x := TOP(S)                     //x kiểu Phần_Tử
            POP(S)
            If x(số_đĩa)=1 Then
                    chuyển (x(nguồn) ,'→', x(đích))
            Else
            {       PUSH([x(số_đĩa)-1, x(trung_gian),x(nguồn),x(đích)],S)
                    PUSH([1,x(nguồn),x(trung_gian),x(đích)],S)
                    PUSH([x(số_đĩa)-1, x(nguồn),x(đích),x(trung_gian)],S)
            }
    }
Return
```

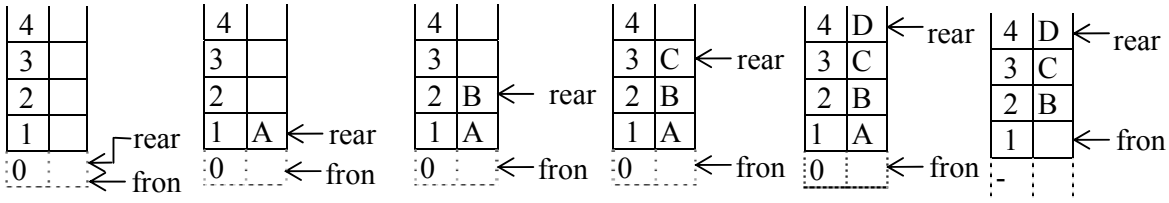
VI. HÀNG ĐỢI

VI.1. Định nghĩa hàng đợi

Hàng đợi, hay ngắn gọn là hàng (queue) cũng là một danh sách đặc biệt mà phép thêm và bớt được thực hiện ở 2 đầu khác nhau. Thêm ở cuối danh sách (REAR), còn bớt thì ở phía đầu của danh sách (FRONT).

Xếp hàng mua vé xe lửa là hình ảnh của hàng đợi, người đến trước mua vé trước, người đến sau thì sẽ mua vé sau. Vì vậy hàng đợi còn được gọi là cấu trúc FIFO (first in - first out) hay "vào trước - ra trước".

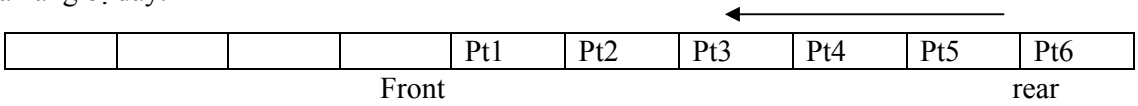
Giả sử thêm lần lượt các phần tử A, B, C, D vào queue có độ dài là 4 phần tử, sau đó loại bỏ phần tử A ra khỏi queue, ta có hình dạng queue như sau:



- **Các thao tác cơ bản trên hàng đợi:**
  - ✓ CREAT_QUEUE(Q) khởi tạo một hàng đợi rỗng
  - ✓ FRONT(Q) hàm trả về giá trị của phần tử đầu tiên của hàng Q.
  - ✓ ADD(x,Q) xen phần tử x vào hàng Q
  - ✓ REMOVE(Q) xoá phần tử tại đầu của hàng Q
  - ✓ EMPTY_QUEUE(Q) hàm kiểm tra hàng rỗng.
  - ✓ FULL_QUEUE(Q) kiểm tra hàng đầy.

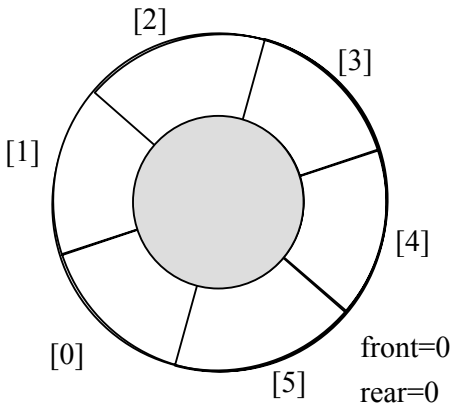
VI.2. Cài đặt hàng đợi bằng mảng

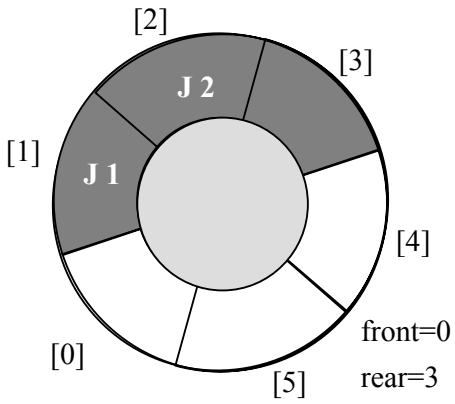
Ta dùng một mảng để chứa các phần tử của hàng, khởi đầu phần tử đầu tiên của hàng được đưa vào vị trí thứ 1 của mảng, phần tử thứ 2 vào vị trí thứ 2 của mảng... Giả sử hàng có n phần tử, ta có front=1 và rear=n. Khi xoá một phần tử front tăng lên 1, khi thêm một phần tử rear tăng lên 1. Như vậy hàng có khuynh hướng đi xuống, đến một lúc nào đó ta không thể thêm vào hàng được nữa dù mảng còn nhiều chỗ trống (các vị trí trước front) trường hợp này ta gọi là hàng bị tràn. Thực sự hàng chỉ bị tràn khi toàn bộ mảng đã chứa các phần tử của hàng ta gọi là hàng bị đầy.



Cách khắc phục hàng bị tràn là dùng mảng xoay vòng:

- Dời toàn bộ hàng lên front vị trí, Cách này gọi là di chuyển tịnh tiến. Trong trường hợp này ta luôn có front<rear.
- Xem mảng như là một vòng tròn nghĩa là khi hàng bị tràn nhưng chưa đầy ta thêm phần tử mới vào vị trí 1 của mảng, thêm một phần tử mới nữa thì thêm vào vị trí 2 (nếu có thể)...Rõ ràng cách làm này front có thể lớn hơn rear.





➢ Khai báo cài đặt

Để quản lí một hàng ta chỉ cần quản lí đầu hàng (front) và cuối hàng (rear). Giả sử cần một khoảng tối đa MAXZISE phần tử cho mảng.

```
Typedef QUEUE=record
    Kiểu_Mảng    A
    Integer      front,rear
End Record
```

➢ Tạo hàng rỗng CREAT_QUEUE(Q)

Lúc này front và rear không trỏ đến vị trí hợp lệ nào trong mảng vậy ta có thể cho front và rear đều bằng 0

```
Proc CREAT_QUEUE(Q)
    Q(front) := 0
```

```
    Q(rear) := 0
Return
```

- **Kiểm tra hàng rỗng EMPTY_QUEUE**

Trong quá trình làm việc ta có thể thêm và xoá các phần tử trong hàng. Rõ ràng, nếu ta có đưa vào hàng một phần tử nào đó thì front>0. Khi xoá một phần tử ta tăng front lên 1, nếu hàng rỗng (front=rear) cũng đặt front=0. Hơn nữa khi mới khởi tạo hàng, tức là front = 0, thì hàng cũng rỗng. Vì vậy hàng rỗng khi và chỉ khi front =0.

```
Func EMPTY_QUEUE(Q)
    If Q(front)=0 Then
    EMPTY_QUEUE := True
Else
    EMPTY_QUEUE := False
Return
```

- **Kiểm tra hàng đầy FULL_QUEUE(Q)**

Hàng đầy nếu số phần tử hiện có trong hàng bằng độ dài của mảng.

```
func FULL_QUEUE(Q)
    If (Q(rear)=Q(front))And (Q(rear)<>0) Then
            FULL_QUEUE := True
    Else
            FULL_QUEUE := False
Return
```

- **Chương trình con thêm một phần tử vào hàng ADD(x,Q)**

Trường hợp Queue đầy không thêm được

```
Proc ADD(x,Q)
    If Not FULL_QUEUE(Q) Then
    {
            Q(rear) := (Q(rear) +1 )Mod MAXSIZE
            Q(A[Q(rear)]) := x
    }
    Else write('Lỗi: hàng đầy');
Return
```

- **Xoá một phần tử của hàng REMOVE(Q)**

Xoá phần tử đầu hàng ta chỉ cần cho front tăng lên 1. Nếu front = rear thì hàng thực chất đã rỗng, nên ta khởi tạo lại hàng rỗng (tức là đặt lại giá trị front = rear =0)

```
Proc REMOVE(Q)
    If Not EMPTY_QUEUE(Q) Then
    {       Q(front) :=  (Q(front)+1) Mod MAXSIZE
            If Q(front)=Q(rear) Then
                    Q(front) := Q(rear) := 0
    }
    Else Write('Lỗi: hàng rỗng');
Return
```

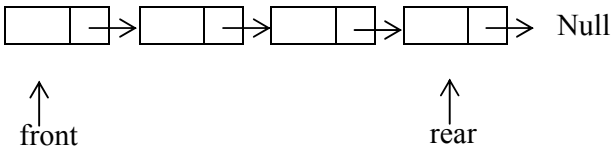
- **Xác định giá trị của phần tử đầu tiên của hàng FRONT(Q)**

Trường hợp hàng rỗng không xác định

```
Func FRONT(Q)
    If EMPTY_QUEUE(Q) Then
            Write('Lỗi, Không xác định')
    Else
            FRONT := Q(A[Q(front)+1])
Return
```

VI.3. Cài đặt hàng đợi bằng danh sách liên kết đơn

Cách tự nhiên nhất là dùng hai con trỏ front và rear để trỏ tới phần tử đầu hàng (front) và cuối hàng (rear).



- **Khai báo cấu trúc dữ liệu**

```
Typedef Node= record
    Kiểu_lưu_trữ    Info  //Trường Info dùng chứa thông tin cần lưu trữ
    Con_Trỏ_Kiểu_Node   Link   //Link là con trỏ trỏ đến phần tử kế tiếp trong
danh sách
End Record
Typedef QUEUE=Record
    Con_trỏ_Kiểu_Node front, rear
End Record
```

- **Khởi tạo hàng rỗng CREAT_QUEUE(Q)**

```
Proc CREAT_QUEUE(Q)
    Q(front) := Null
    Q(rear) := Null
Return
```

- **Kiểm tra hàng rỗng EMPTY_QUEUE(Q)**

Hàng rỗng nếu front hoặc rear bằng Null

```
Func EMPTY_QUEUE(Q)
    If Q(front)=Null Then
            EMPTY_QUEUE := True
    Else
            EMPTY_QUEUE := False
Return
```

- **Thêm một phần tử vào hàng ADD(x,Q)**

Thêm một phần tử vào hàng thực chất là chèn 1 gái trị x vào cuối danh sách liên kết

```
Proc ADD(x,Q)
    new(p)                  //cấp phát 1 Node mới
    p(Info) := x
    p(Link) := Null
    If Q(front)=Null Then
```

```
            Q(front) := p           //trường hợp Q rỗng thì thành Q có 1 phần tử
        Else
            (Q(rear))(Link) := p
            Q(rear) :=  p
    Return
```

- **Xóa một phần tử trong hàng REMOVE(Q)**

Thực chất là xoá phần tử nằm ở đầu danh sách. Trường hợp danh sách rỗng không xoá được

```
Proc REMOVE(Q)
    If Not EMPTY_QUEUE(Q) Then
            {
            P := Q(front)
            Q(front) := p(Link)
            Delete(p)
            }
    Else write('Lỗi: hàng rỗng');
Return
```

- **Xác định giá trị của phần tử đầu tiên của hàng FRONT(Q)**

Trường hợp hàng rỗng không xác định

```
Func FRONT(Q)
    If EMPTY_QUEUE(Q) Then
            Write('Lỗi, Không xác định')
    Else
            FRONT := (Q(front))(Info)
Return
```

CHƯƠNG 3

CÂY

I. MỘT SỐ KHÁI NIỆM VỀ CÂY

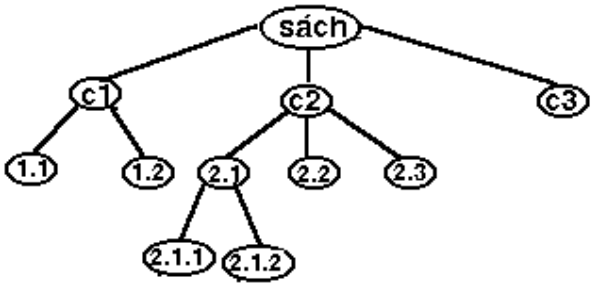
I.1. Khái niệm

Cây là một tập hợp các phần tử gọi là nút (nodes) trong đó có một nút được phân biệt gọi là nút gốc (root). Trên tập hợp các nút này có một quan hệ, gọi là mối quan hệ cha - con (parenthood), xác định hệ thống cấu trúc trên các nút. Mỗi nút, trừ nút gốc, có duy nhất một nút cha. Một nút có thể có nhiều nút con hoặc không có nút con nào. Mỗi nút biểu diễn một phần tử trong tập hợp đang xét và nó có thể có một kiểu nào đó bất kỳ, thường ta biểu diễn nút bằng một kí tự, một chuỗi hoặc một số ghi trong vòng tròn. Mối quan hệ cha con được biểu diễn theo qui ước nút cha ở dòng trên nút con ở dòng dưới và được nối bởi một đoạn thẳng.

Có thể định nghĩa cây một cách đệ qui như sau:

- Một nút đơn độc là một cây. Nút này cũng chính là nút gốc của cây.
- Giả sử ta có n là một nút đơn độc và k cây T1,.., Tk với các nút gốc tương ứng là n1,.., nk thì có thể xây dựng một cây mới bằng cách cho nút n là cha của các nút n1,.., nk. Cây mới này có nút gốc là nút n và các cây T1,.., Tk được gọi là các cây con.
- Tập rỗng cũng được coi là một cây và gọi là cây rỗng kí hiệu ^.

Ví dụ: xét mục lục của một quyển sách. Mục lục này có thể xem là một cây



Nút gốc là sách, nó có ba cây con có gốc là C1, C2, C3. Cây con thứ 3 có gốc C3 là một nút đơn độc trong khi đó hai cây con kia (gốc C1 và C2) có các nút con.

Số các con của một nút gọi là cấp (degree) của nút đó. Ví dụ cấp của nút c1 là 2 và của nút c2 là 3.

Nút có cấp bằng không gọi là nút Lá (leaf) hay nút đơn độc, nút tận cùng. Ví dụ các nút 1.1, 2.1.1, 2.1.2, 2.2, 2.3, c3 là các nút là.

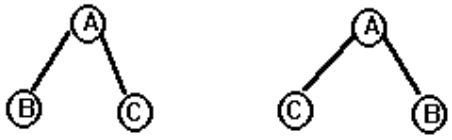
Cấp của cây là cấp cao nhất của các nút trên cây. Ví dụ cây **Sách** ở trên là cấp 3.

Gốc của cây người ta gán cho số mức (level) là 1, nếu nút cha có mức là i thì nút con sẽ có mức là i + 1. Ví dụ nút **Sách** có mức là 1, nút **c2** có mức là 2 và nút **2.1.1** có mức là 3.

Chiều cao (height) hay chiều sâu (depth) của một cây là số mức lớn nhất của nút có trên cây đó. Cây ở trên có chiều cao là 4.

Nếu n1,.., nk là một chuỗi các nút trên cây sao cho ni là nút cha của nút ni+1, với i=1..k-1, thì chuỗi này gọi là một đường đi trên cây (hay ngắn gọn là đường đi) từ n1 đến nk. Độ dài đường đi này được định nghĩa bằng **số nút trên đường đi trừ 1** hay chính là **số cung** trên đường đi. Như vậy độ dài đường đi từ một nút đến chính nó bằng không. Ví dụ đường đi từ nút **Sách** đến nút **2.1** là **Sách, c2, 2.1** và độ dài đường đi là 2.

Nếu thứ tự các cây con (hay các nút con) của một nút được coi trọng, thì cây đang xét là cây có thứ tự (ordered tree), ngược lại là cây không có thứ tự (unorderred tree). Thường là thứ tự được qui ước từ trái sang phải. Như vậy, nếu kể thứ tự thì hai cây sau là khác nhau:

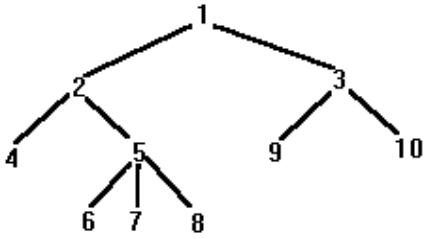


I.2. Biểu diễn cây

I.2.1. Cài đặt bằng mảng

Cho cây T, ta có thể gán tên cho các nút lần lượt là 1,2,..,n. Sau đó ta dùng một mảng A một chiều để lưu trữ cây bằng cách cho A[i]←j với j là nút cha của nút i. nếu i là nút gốc ta cho a[i]=0

Ví dụ:



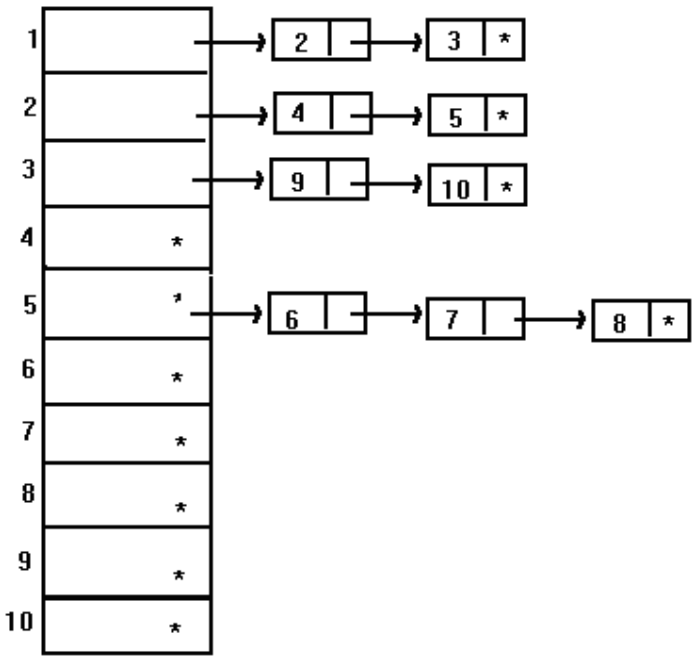
Cây trên được biểu diễn trong mảng A như sau:

Chỉ số mảng	1	2	3	4	5	6	7	8	9	10
Tên nút (chứa chỉ số nút cha)	0	1	1	2	2	5	5	5	3	3
nhãn	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

I.2.2. Cài bằng danh sách kề

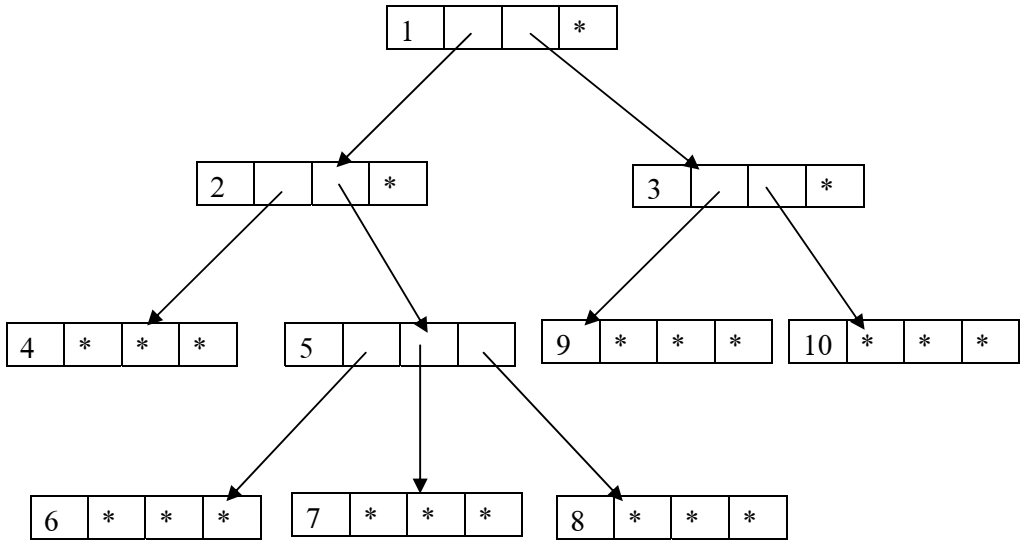
Một cách biểu diễn khác cũng thường được dùng là biểu diễn cây dưới dạng mỗi nút có một danh sách các nút con. Vì số nút con của một nút là không biết trước nên dùng danh sách liên kết sẽ thích hợp hơn.

Với cây ở trên thì có thể lưu trữ như sau (dấu '*' biểu diễn giá trị Null)



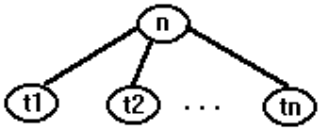
I.2.3. Cài bằng con trỏ

Mỗi Node trong cây gồm có n+ 1 trường, trong đó n là bậc của cây là số trường liên kết của cây và 1 trường Info để chứa dữ liệu. Ta có thể mô phỏng cây trên như sau:(* biểu diễn Null)



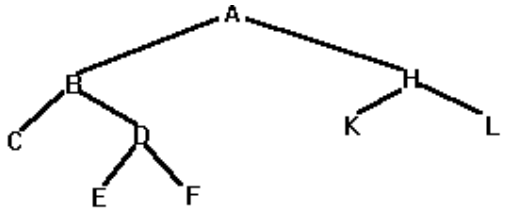
I.3. Duyệt cây

Duyệt cây là một qui tắc cho phép đi qua lần lượt tất cả các nút của cây mỗi nút đúng một lần, danh sách liệt kê các nút (tên nút hoặc giá trị chứa bên trong nút) theo thứ tự đi qua gọi là danh sách duyệt cây. Có 3 cách duyệt cây quan trọng: Duyệt tiền tự (preorder), duyệt trung tự (inorder), duyệt hậu tự (posorder). Có thể định nghĩa các phép duyệt cây tổng quát (xem hình bên dưới) một cách đệ qui như sau:



- Cây rỗng thì danh sách duyệt cây là rỗng và nó được coi là biểu thức duyệt tiền tự, trung tự, hậu tự của cây.
- Cây chỉ có một nút thì danh sách duyệt cây gồm chỉ một nút đó và nó được coi là biểu thức duyệt tiền tự, trung tự, hậu tự của cây.
- Ngược lại: giả sử cây T có nút gốc là n và có các cây con là T1,..,Tn thì:
  - ✓ Kết quả duyệt tiền tự của cây T là liệt kê nút n, kế tiếp là kết quả duyệt tiền tự của các cây T1, T2, .., Tn theo thứ tự đó.
  - ✓ Kết quả duyệt trung tự của cây T là kết quả duyệt trung tự của cây T1, kế tiếp là nút n, rồi đến kết quả duyệt trung tự của các cây T2,.., Tn theo thứ tự đó.
  - ✓ Kết quả duyệt hậu tự của cây T là kết quả duyệt hậu tự của các cây T1, T2,.., Tn theo thứ tự đó, rồi đến nút n.

Ví dụ cho cây như trong hình sau:



Biểu thứcduyệt tiền tự: A B C D E F H K L

trung tự: C B E D F A K H L

hậu tự: C E F D B K L H A

Có thể viết các giải thuật duyệt cây đệ qui như sau:

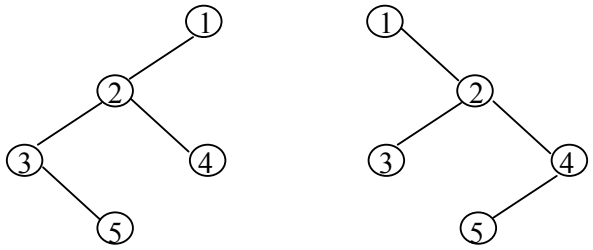
```
Proc PREORDER(n)
    Xử lý gốc n
    For (mỗi nút con c của nút n theo thứ tự từ trái sang phải)
            Call PREORDER(c)
Return
Proc INORDER(n)
    If (n là nút lá) Then
            Xử lý gốc n
    Else
    {       Call INORDER( con trái nhất của nút n);
            Xử lý gốc n
```

```
            For (mỗi nút con c của nút n, trừ nút con trái nhất, theo thứ tự từ trái
            sang phải)
                  Call INORDER(c);
    }
Return
Proc POSORDER(n)
    If (n là nút lá) Then
            Xử lý gốc n
    Else
            {for (mỗi nút con c của nút n theo thứ tự từ trái sang phải) do
            Call  POSORDER(c)
            Xử lý gốc n
            }
Return
```

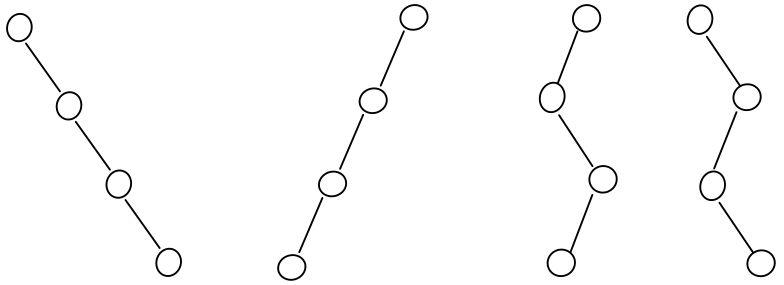
II. CÂY NHỊ PHÂN

II.1. Định nghĩa

Cây nhị phân (binary tree) là cây mà mỗi nút chỉ có tối đa hai nút con hoặc rỗng. Hơn nữa các nút con của cây được phân biệt thứ tự rõ ràng, một nút con gọi là nút con trái và một nút con gọi là nút con phải. Ta qui ước vẽ nút con trái bên trái nút cha và nút con phải bên phải nút cha, mỗi nút con được nối với nút cha của nó bởi một đoạn thẳng.



Cần chú ý tới một số dạng đặc biệt của cây nhị phân

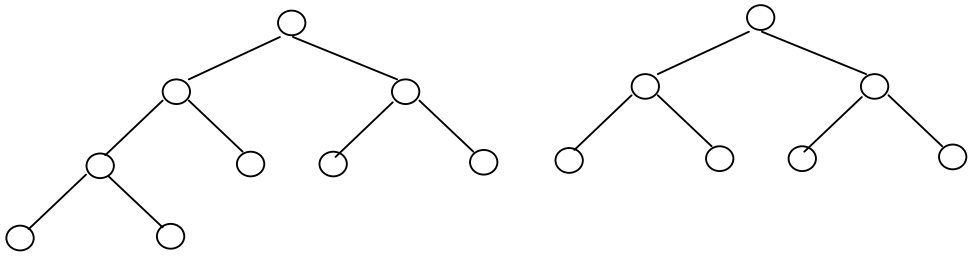


(a) (b) (c) (d)

Các cây nhị phân (a),(b),(c),(d) được gọi là cây nhị phân suy biến (degenerater binary tree), các nút không phải là lá chỉ có 1 con. Cây (a) gọi là cây lêch trái, cây (b) gọi là cây lệch phải, cây (c) và (d) gọi là cây zíc-zắc.

Nếu cây nhị phân có tất cả các mức đều đạt số nút tối đa thì gọi là cây nhị phân đày đủ (full binary tree). Nếu ở mức cuối cùng không đạt tối đa và các nút thiếu tập trung ở bên phải thì gọi là cây nhị phân hoàn chỉnh (complete binary tree).

Cây nhị phân đầy đủ là trường hợp riêng của cây nhị phân hoàn chỉnh.

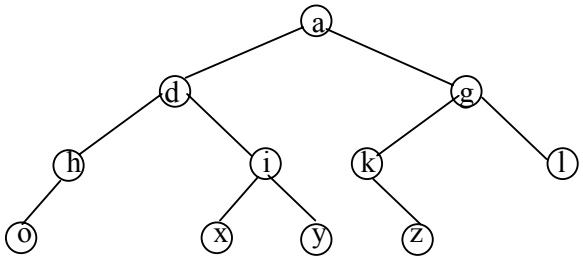


Ta có thể thấy ngay những tính chất sau bằng phép chứng minh quy nạp:

- Trong các cây nhị phân có cùng số lượng nút như nhau thì cây nhị phân suy biến có chiều caolớn nhất, còn cây nhị phân hoàn chỉnh thì có chiều cao nhỏ nhất.
- Số lượng tối đa các nút trên mức i của cây nhị phân là 2i-1, tối thiểu là 1 ($i \geq 1$).
- Số lượng tối đa các nút trên một cây nhị phân có chiều cao h là 2h-1, tối thiểu là h ($h \geq 1$).
- Cây nhị phân hoàn chỉnh, không đầy đủ, có n nút thì chiều cao của nó là h=[log2(n+1)]+1.
- Cây nhị phân đầy đủ có n nút thì chiều cao của nó là h = log2(n + 1)

II.2. Cài đặt cây nhị phân

Ví dụ: cho cây nhị phân như sau:



II.2.1.Cài đặt bằng mảng

➢ **Cách 1: dùng 3 mảng 1 chiều có n phần tử (n số phần tử trong cây)**

- Mảng Value chứa nội dung của các nút trong cây
- Mảng Left chứa nhãn của nút gốc của cây con trái có gốc ở mảng Value
- Mảng Right chứa nhãn của nút gốc của cây con phải có gốc ở mảng Value
- Với các nút lá (không con) thì Left và Right tương ứng để trống

Với cây trên thì nội dung 3 mảng như sau:

Chỉ số	1	2	3	4	5	6	7	8	9	10	11
Value	**a**	**d**	**g**	**h**	**i**	**k**	**l**	**o**	**x**	**y**	**z**
Left	**2**	**4**	**6**	**8**	**9**						
Right	**3**	**5**	**7**		**10**	**11**					

Với cách cài đặt này thì các giá trị trong cây phải khác nhau từng đôi một.

Cũng có thể dùng 1 mảng 1 chiều kiểu bản ghi gồm (Value, Left, Right) để lưu trữ.

➢ **Cách 2: Dùng 1 mảng 1 chiều để lưu trữ cây nhị phân.**

Trên cây nhị phân hoàn chỉnh ta đánh số thứ tự từ trên xuống và từ trái sang phải. Chỉ số đánh được trên cây chính là chỉ số của mảng 1 chiều.

Với cây trên ta có mảng như sau:

Chỉ số	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value	a	d	g	h	i	k	l	o		x	y		z		

Với cách lưu này, ta thấy:

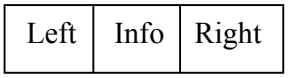
- Nếu chỉ số i là nút cha thì nút con trái có chỉ số 2i và con phải có chỉ số 2i+1
- Nếu i là chỉ số của nút con thì chỉ số nút cha là $\lfloor i/2 \rfloor$

Hai cách lưu bằng mảng trên ta nhận thấy rất phù hợp đối với cây nhị phân hoàn chỉnh hoặc đầy đủ. Nếu cây nhị phân thiếu nhiều nút ở nhiều mức khác nhau thì có khác nhiều chỗ trống.

II.2.2.Cài đặt bằng con trỏ

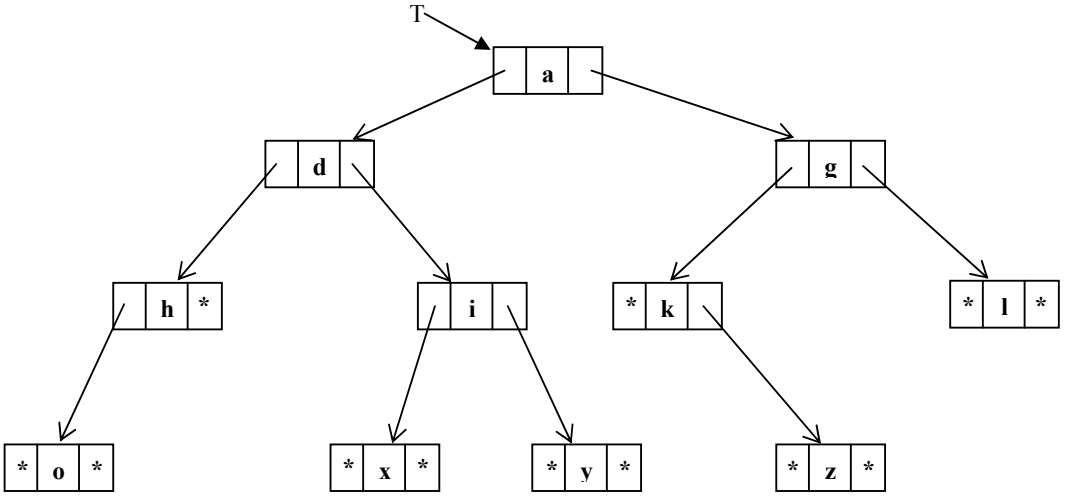
Mỗi nút của cây gồm có 2 phần. Phần Info dùng để chứa giá trị của nút. Phần liên kết gồm 2 con trỏ Left và Right được dùng để chứa địa chỉ các nút cây con trái và cây con phải

Cấu trúc của một nút như sau




```
Typedef  TreeNode=Record
            Kiểu_lưu_trữ  Info
            Con_trỏ_kiểu_TreeNode Left, Right
         End Record
```

Với cây nhị phân trên ta có thể mô phỏng như sau (dấu '*" thay cho con trỏ Null)



Hàm tạo Node có giá trị X, con trỏ Left là L, con trỏ Right là R, trả về địa chỉ của Node đó

```
Func Tao(X,L,R)
    New(p)
    Info(p) := X
    Left(p) := L
```

```
        Right(p) := R
        Tao := p
    Return
```

Sử dụng hàm để tạo cây trên như sau:

```
T := Tao('a',Tao('d',Tao('h',Tao('o',*,*),*),Tao('i',Tao('x',*,*),Tao('y',*,*))),
        Tao('g',Tao('k',*,Tao('z',*,*)),Tao('l',*,*)))
```

II.3. Các phép duyệt cây nhị phân

Phép xử lý các nút trên cây mà ta gọi chung là phép thăm (Visit) các nút một cách hệ thống sao cho mỗi nút chỉ được thăm một lần gọi là phép duyệt cây.

Giả sử rằng nếu như một nút không có nút con trái (hoặc nút con phải) thì liên kết Left (Right) của nút đó được liên kết thẳng tới một nút đặc biệt mà ta gọi là NIL (hay NULL), nếu cây rỗng thì nút gốc của cây đó cũng được gán bằng NIL. Khi đó có ba cách duyệt cây hay được sử dụng:

- **Duyệt theo thứ tự trước (preorder traversal)**

Trong phép duyệt theo thứ tự trước thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê trước giá trị lưu trong hai nút con của nó, có thể mô tả bằng thủ tục đệ quy sau:

```
proc Visit(N)       //Duyệt nhánh cây nhận N là nút gốc của nhánh đó
    if N ≠ nil then
    {
            <Output trường Info của nút N>
            Visit(Nút con trái của N);
            Visit(Nút con phải của N);
    }
Return
```

Quá trình duyệt theo thứ tự trước bắt đầu bằng lời gọi Visit(nút gốc).

Như cây ở trên, nếu ta duyệt theo thứ tự trước thì các giá trị sẽ được liệt kê theo thứ tự:
A B D H I E C F J G

- **Duyệt theo thứ tự giữa (inorder traversal)**

Trong phép duyệt theo thứ tự giữa thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê sau giá trị lưu ở nút con trái và được liệt kê trước giá trị lưu ở nút con phải của nút đó, có thể mô tả bằng thủ tục đệ quy sau:

```
proc Visit(N)       {Duyệt nhánh cây nhận N là nút gốc của nhánh đó}
    if N ≠ nil then
    {
            Visit(Nút con trái của N);
            <Output trường Info của nút N>
            Visit(Nút con phải của N);
    }
Return
```

Quá trình duyệt theo thứ tự giữa cũng bắt đầu bằng lời gọi Visit(nút gốc).

Như cây ở trên, nếu ta duyệt theo thứ tự giữa thì các giá trị sẽ được liệt kê theo thứ tự:
H D I B E A F J C G

➢ **Duyệt theo thứ tự sau (postorder traversal)**

Trong phép duyệt theo thứ tự sau thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê sau giá trị lưu ở hai nút con của nút đó, có thể mô tả bằng thủ tục đệ quy sau:

```
proc Visit(N)       {Duyệt nhánh cây nhận N là nút gốc của nhánh đó}
    if N ≠ nil then
    {
            Visit(Nút con trái của N);
            Visit(Nút con phải của N);
            <Output trường Info của nút N>
    }
Return
```

Quá trình duyệt theo thứ tự sau cũng bắt đầu bằng lời gọi Visit(nút gốc).

Cũng với cây ở trên, nếu ta duyệt theo thứ tự sau thì các giá trị sẽ được liệt kê theo thứ tự:

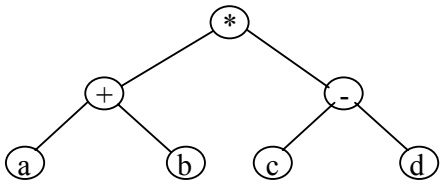
H I D E B J F G C A

III. CÂY BIỂU DIỄN BIỂU THỨC

III.1. Biểu diễn biểu thức dưới dạng cây

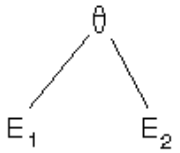
Chúng ta có thể biểu diễn các biểu thức số học gồm các phép toán cộng, trừ, nhân, chia bằng một cây nhị phân, trong đó các nút lá biểu thị các hằng hay các biến (các toán hạng), các nút không phải là lá biểu thị các toán tử (phép toán số học chẳng hạn). Mỗi phép toán trong một nút sẽ tác động lên hai biểu thức con nằm ở cây con bên trái và cây con bên phải của nút đó.

Ví dụ: Cây biểu diễn biểu thức (a+b)*(a-c)



Qui tắc biểu diễn một biểu thức toán học trên cây như sau:

- Mỗi nút có tối đa 2 con (con trái & con phải)
- Mỗi nút lá có nhãn biểu diễn cho một toán hạng.
- Mỗi nút trung gian (không phải lá) biểu diễn một toán tử.



Với một phép toán có thứ tự như '–' và '/' thì rõ ràng cây này là cây có thứ tự

Độ ưu tiên toán tử trong biểu thức trung tố ( )

^

* /

+ -

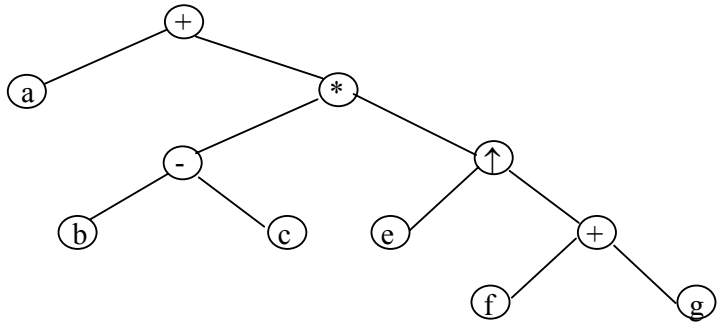
Qui tắc vẽ cây biểu thức

- Trên biểu thức trung tố, xác định thứ tự thực hiện các toán tử của biểu thức
- Chọn phép toán X thực hiện cuối cùng đặt nó vào gốc cây
- Bên trái của X là cây con trái của X, bên phải của X là cây con phải
- Với mỗi cây con, ta lại thực hiện như trên cho đến khi các toán hạng ở nút lá

Ví dụ : Vẽ cây biểu diễn biểu thức a + (b – c) * e ↑ (f + g)

a	+	(	b	-	c	)	*	e	↑	(	f	+	g	)
	5			1			4		3			2		

Cây biểu diễn như sau:



III.2. Các ký pháp dùng cho biểu thức

Với cây nhị phân biểu diễn biểu thức **a + (b – c) * e ↑ (f + g)** như trên, các phép duyệt cây theo thứ tự trước, giữa và sau, sẽ cho ta kết quả như sau:

- Nếu ta duyệt theo thứ tự trước, ta sẽ được biểu thức: **+ a * - b c ↑ e + f g**

  Đây là dạng tiền tố (prefix) của biểu thức. Trong ký pháp này, toán tử được viết trước hai toán hạng tương ứng, người ta còn gọi ký pháp này là **ký pháp Ba lan**.
- Nếu ta duyệt theo thứ tự giữa, ta sẽ đượcbiểu thức: **a + b - c * e - ↑ f + g**

  Ký pháp này hơi mập mờ vì thiếu dấu ngoặc. Nếu ta thêm vào thủ tục duyệt inorder việc bổ sung các cặp dấu ngoặc vào mỗi biểu thức con thì ta sẽ được biểu thức:

  (a + ((b – c) * (e ↑ (f + g))))

  Ký pháp này gọi là dạng trung tố (infix) của một biểu thức (Thực ra chỉ cần thêm các dấu ngoặc đủ để tránh sự mập mờ mà thôi, không nhất thiết phải thêm vào đầy đủ các cặp dấu ngoặc).
- Nếu ta duyệt theo thứ tự sau, ta sẽ được biểu thức: **a b c – e f g + ↑ * +**

  Đây là dạng hậu tố (postfix) của biểu thức. Trong ký pháp này toán tử được viết sau hai toán hạng, người ta còn gọi ký pháp này là ký pháp nghịch đảo Balan (Reverse Polish Notation - RPN) .

Chỉ có dạng trung tố mới cần có dấu ngoặc, dạng tiền tố và hậu tố không cần có dấu ngoặc.

III.3. Một số thuật toán đối với biểu thức

III.3.1.Xây dựng cây nhị phân biểu diễn biểu thức

Ngay trong phần đầu tiên, chúng ta đã biết rằng các dạng biểu thức trung tố, tiền tố và hậu tố đều có thể được hình thành bằng cách duyệt cây nhị phân biểu diễn biểu thức đó theo các trật tự khác nhau. Vậy tại sao không xây dựng ngay cây nhị phân biểu diễn biểu thức đó rồi thực hiện các công việc tính toán ngay trên cây?. Khó khăn gặp phải chính là thuật toán xây dựng cây nhị phân trực tiếp từ dạng trung tố có thể kém hiệu quả, trong khi đó từ dạng hậu tố lại có thể khôi phục lại cây nhị phân biểu diễn biểu thức một cách rất đơn giản, gần giống như quá trình tính toán biểu thức hậu tố:

Bước 1: Khởi tạo một Stack rỗng dùng để chứa các nút trên cây

Bước 2: Đọc lần lượt các phần tử của biểu thức RPN từ trái qua phải (phần tử này có thể là hằng, biến hay toán tử) với mỗi phần tử đó:

- Tạo ra một nút mới N chứa phần tử mới đọc được
- Nếu phần tử này là một toán tử, lấy từ Stack ra hai nút (theo thứ tự là y và x), sau đó đem liên kết trái của N trỏ đến x, đem liên kết phải của N trỏ đến y.
- Đẩy nút N vào Stack

Bước 3: Sau khi kết thúc bước 2 thì toàn bộ biểu thức đã được đọc xong, trong Stack chỉ còn duy nhất một phần tử, phần tử đó chính là gốc của cây nhị phân biểu diễn biểu thức.

III.3.2.Chuyển đổi biểu thức từ dạng trung tố sang hậu tố

Thuật toán sử dụng một Stack để chứa các toán tử và dấu ngoặc mở. Thủ tục CREAT_STACK(S) để tạo một Stack rỗng, PUSH(x,S) để đẩy một phần tử vào Stack, hàm POP(S) để lấy ra một phần tử từ Stack, hàm TOP(S) để đọc giá trị phần tử nằm ở đỉnh Stack mà không lấy phần tử đó ra, EMPTY_STACK(S) để xác đinh Stack rỗng hay không.

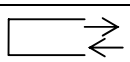
Thuật toán POLISH để chuyển biểu thức trung tố sang hậu tố

- Đầu vào: Chuỗi Sin chứa biểu thức trung tố
- Đầu ra:Chuỗi Sout chứa biểu thức hậu tố
- Sử dụng hàm Ưu_Tiên(x,y) để xác định xem toán tử x cơ độ ưu tiên lớn hơn hoặc bằng y.

```
POLISH(Sin)
    B1: CREAT_STACK(Sout)
    B2: CREAT_STACK(S)
    B3:Lặp tuần tự lấy từng phần tử X của Sin
            + Nếu X là '(' Thì  PUSH(X, S)
            + Nếu X là ')' Thì
                    - While TOP(S) <> '(' Do
                    {       PUSH(TOP(S),Sout)
                            POP(S)
                    }
                    - POP(S)
            + Nếu X là toán hạng thì PUSH(X, Sout)
            + Nếu X là toán tử
                    - Nếu Ưu_Tiên(TOP(S),X) Thì
```

```
                                While Ưu_Tiên(TOP(S),X) Do
                                    {PUSH(TOP(S),Sout)
                                     POP(S)
                                    }
                        - PUSH(X,S)
        B4: While Not EMPTY(S) Do
                {PUSH(TOP(S),Sout)
                 POP(S)
                }
    Return
```

Ví dụ:A*B+(C-D)

I	Sin[i]	Stack S	Sout
			“”
1	A		“A”
2	*	*	
3	B	*	“AB”
4	+	+	“AB*”
5	(	+(	“AB*”
6	C	+(	“AB*C”
7	-	+(-	“AB*C”
8	D	+(-	“AB*CD”
9	)	+	“AB*CD-”

Sout = “AB*CD-+”

III.3.3.Định giá biểu thức hậu tố

Có một vấn đề cần lưu ý là khi máy tính giá trị một biểu thức số học gồm các toán tử hai ngôi (toán tử gồm hai toán hạng như +, -, *, /) thì máy chỉ thực hiện được phép toán đó với hai toán hạng, nếu biểu thức phức tạp thì máy phải chia nhỏ và tính riêng từng biểu thức trung gian, sau đó mới lấy giá trị tìm được để tính tiếp. Ví dụ như biểu thức 1 + 2 + 4 máy sẽ phải tính 1 + 2 trước được kết quả là 3 sau đó mới đem 3 cộng với 4 chứ không thể thực hiện phép cộng một lúc ba số được. Khi lưu trữ biểu thức dưới dạng cây nhị phân thì ta có thể coi mỗi nhánh con của cây đó mô tả một biểu thức trung gian mà máy cần tính khi xử lý biểu thức lớn. Vậy để tính một biểu thức lưu trữ trong một nhánh cây nhị phân gốc ở nút n, máy sẽ tính gần giống như hàm đệ quy sau:

```
func Calculate(n)  //Tính biểu thức con trong nhánh cây gốc n
    if <Nút n chứa không phải là một toán tử> then
            Calculate := <Giá trị chứa trong nút n>
    else    //Nút n chứa một toán tử R
    {
```

```
            x := Calculate(nút con trái của n);
            y := Calculate(nút con phải của n);
            Calculate := x R y;
        }
    Return
```

(Trong trường hợp lập trình trên các hệ thống song song, việc tính giá trị biểu thức ở cây con trái và con phải có thể tiến hành đồng thời làm giảm đáng kể thời gian tính toán biểu thức).

Để ý rằng khi tính toán biểu thức, máy sẽ phải quan tâm tới việc tính biểu thức ở hai nhánh con trước, rồi mới xét đến toán tử ở nút gốc. Điều đó làm ta nghĩ tới phép duyệt hậu thứ tự và ký pháp hậu tố.

Trong những năm đầu 1950, nhà lô-gic học người Balan Jan Lukasiewicz đã chứng minh rằng biểu thức hậu tố không cần phải có dấu ngoặc vẫn có thể tính được một cách đúng đắn bằng cách **đọc lần lượt biểu thức từ trái qua phải** và dùng một Stack để lưu các kết quả trung gian.

Thuật toán tính giá trị biểu thức hậu tố

- Đầu vào: Biểu thức hậu tố S
- Đầu ra: Giá trị của biểu thức trung tố tương ứng

```
ĐINHGIA(S)
    B1: CREAT_STACK(S)
    B2: Lặp lấy từng phần tử X của S
            + Nếu X là toán hạng thì PUSH(X,S)
            + Nếu X là toán tử
                    - A := TOP(S), POP(S)
                    - B := TOP(S), POP(S)
                    - C :=  B X A
                    - PUSH(C,S)
    B3: Kết quả := TOP(S)
Return
```

A=4	B=2	*	C=5	D=3	-	+
		4*2=8			5-3=2	8+2=16
				3		
	2		5	5	2	
4	4	8	8	8	8	16

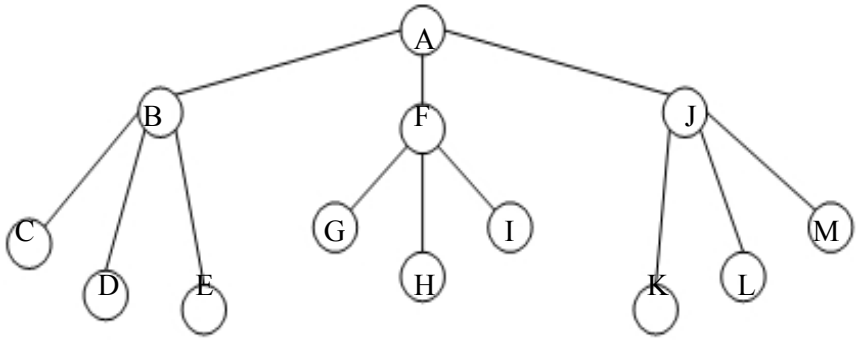
IV. CÂY TỔNG QUÁT

Trong thực tế, có một số ứng dụng đòi hỏi một cấu trúc dữ liệu dạng cây nhưng không có ràng buộc gì về số con của một nút trên cây, ví dụ như cấu trúc thư mục trên đĩa hay hệ thống đề mục của một cuốn sách. Khi đó, ta phải tìm cách mô tả một cách khoa học cấu trúc dữ liệu dạng cây tổng quát. Cũng như trường hợp cây nhị phân, người ta thường biểu diễn cây tổng quát bằng hai cách: Lưu trữ kế tiếp bằng mảng và lưu trữ bằng cấu trúc liên kết.

IV.1. Cây K – phân

Cây K-phân là một dạng cấu trúc cây mà mỗi nút trên cây có tối đa K nút con (có tính đến thứ tự của các nút con).

➢ **Biểu diễn cây K-phân bằng mảng**



Cũng tương tự như việc biểu diễn cây nhị phân, người ta có thể thêm vào cây K_phân một số nút giả để cho mỗi nút nhánh của cây K_phân đều có đúng K nút con, các nút con được xếp thứ tự từ nút con thứ nhất tới nút con thứ K, sau đó đánh số các nút trên cây K_phân bắt đầu từ 0 trở đi, bắt đầu từ mức 1, hết mức này đến mức khác và từ "trái qua phải" ở mỗi mức:

Theo cách đánh số này, nút con thứ j của nút i là: i * K + j. Nút cha của nút x là nút (x - 1) div K. Ta có thể dùng một mảng T đánh số từ 0 để lưu các giá trị trên các nút: Giá trị tại nút thứ i được lưu trữ ở phần tử T[i].

A	B	F	J	C	D	E	G	H	I	K	L	M
0	1	2	3	4	5	6	7	8	9	10		12

➢ **Biểu diễn cây K_phân bằng cấu trúc liên kết**

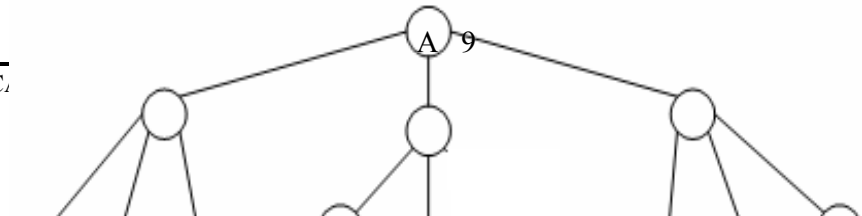
Khi biểu diễn cây K_phân bằng cấu trúc liên kết, mỗi nút của cây là một bản ghi (record) gồm hai trường:

- ✓ Trường Info: Chứa giá trị lưu trong nút đó.
- ✓ Trường Links: Là một mảng gồm K phần tử, phần tử thứ i chứa liên kết (con trỏ) tới nút con thứ i, trong trường hợp không có nút con thứ i thì Links[i] được gán một giá trị đặc biệt.

Đối với cây K_ phân, ta cũng chỉ cần giữ lại nút gốc, bởi từ nút gốc, đi theo các hướng liên kết có thể đi tới mọi nút khác.

IV.2. Cây tổng quát

Là một dạng cấu trúc cây mà số con tối đa của mỗi nút trên cây là không xác định.



B 1 F 2 J 4

C 3 G 7 M 12

D 5 E 6 H 8 10 K L 11

➢ **Biểu diễn cây tổng quát bằng mảng**

Để lưu trữ cây tổng quát bằng mảng, trước hết, ta đánh số các nút trên cây bắt đầu từ 1 theo một thứ tự tuỳ ý. Giả sử cây có n nút thì ta sử dụng:

- ✓ Một mảng Info[1..n], trong đó Info[i] là giá trị lưu trong nút thứ i.
- ✓ Một mảng Children được chia làm n đoạn, đoạn thứ i gồm một dãy liên tiếp các phần tử là chỉ số các nút con của nút i. Như vậy mảng Children sẽ chứa tất cả chỉ số của mọi nút con trên cây (ngoại trừ nút gốc) nên nó sẽ gồm n - 1 phần tử, lưu ý rằng khi chia mảng Children làm n đoạn thì sẽ có những đoạn rỗng (tương ứng với danh sách các nút con của một nút lá)
- ✓ Một mảng Head[1..n + 1], để đánh dấu vị trí cắt đoạn trong mảng Children: Head[i] là vị trí đầu đoạn thứ i, hay nói chính xác hơn: Các phần tử trong mảng Children từ vị trí Head[i] đến Head[i+1] - 1 là chỉ số các nút con của nút thứ i. Khi Head[i] = Head[i+1] có nghĩa là đoạn thứ i rỗng. Quy ước: Head[n+1] = n.
- ✓ Giữ lại chỉ số của nút gốc.

Mảng Info:

Info[i]	B	F	C	J	D	E	G	H	A	K	L	M
i	1	2	3	4	5	6	7	8	9	10	11	12

Mảng Children:

Children[i]	3	5	6	7	8	10	111	2	1	2	4	
i	1	2	3	4	5	6	7	8	9	10	11	
	Đoạn 1			Đoạn 2		Đoạn 4			Đoạn 9			

(Các đoạn 3, 5, 6, 7, 8, 10, 11, 12 là rỗng)

Mảng Head:

Head[i]	1	4	6	6	9	9	9	9	9	12	12	12	12
i	1	2	3	4	5	6	7	8	9	10	11	12	13

➢ **Lưu trữ cây tổng quát bằng cấu trúc liên kết**

Khi lưu trữ cây tổng quát bằng cấu trúc liên kết, mỗi nút là một bản ghi (record) gồm ba trường:

- ✓ Trường Info: Chứa giá trị lưu trong nút đó.
- ✓ Trường FirstChild: Chứa liên kết (con trỏ) tới nút con đầu tiên của nút đó (con cả), trong trường hợp là nút lá (không có nút con), trường này được gán một giá trị đặc biệt.

- ✓ Trường Sibling: Chứa liên kết (con trỏ) tới nút em kế cận bên phải (nút cùng cha với nút đang xét, khi sắp thứ tự các con thì nút đó đứng liền sau nút đang xét) Trong trường hợp không có nút em kế cận bên phải, trường này được gán một giá trị đặc biệt.

Dễ thấy được tính đúng đắn của phương pháp biểu diễn, bởi từ một nút N bất kỳ, ta có thể đi theo liên kết FirstChild để đến nút con cả, nút này chính là chốt của một danh sách nối đơn các nút con của nút N: từ nút con cả, đi theo liên kết Sibling, ta có thể duyệt tất cả các nút con của nút N.

CHƯƠNG 4

THUẬT TOÁN SẮP XẾP

I. BÀI TOÁN SẮP XẾP

Sắp xếp là quá trình bố trí lại các phần tử của một tập đối tượng nào đó theo một thứ tự nhất định. Chẳng hạn như thứ tự tăng dần (hay giảm dần) đối với một dãy số, thứ tự từ điển đối với các từ v.v... Yêu cầu về sắp xếp thường xuyên xuất hiện trong các ứng dụng Tin học với các mục đích khác nhau: sắp xếp dữ liệu trong máy tính để tìm kiếm cho thuận lợi, sắp xếp các kết quả xử lý để in ra trên bảng biểu v.v...

Nói chung, dữ liệu có thể xuất hiện dưới nhiều dạng khác nhau, nhưng ở đây ta quy ước: Một tập các đối tượng cần sắp xếp là tập các bản ghi (records), mỗi bản ghi bao gồm một số trường (fields) khác nhau. Nhưng không phải toàn bộ các trường dữ liệu trong bản ghi đều được xem xét đến trong quá trình sắp xếp mà chỉ là một trường nào đó (hay một vài trường nào đó) được chú ý tới thôi. Trường như vậy ta gọi là khoá (key). Sắp xếp sẽ được tiến hành dựa vào giá trị của khoá này.

Ví dụ: Hồ sơ tuyển sinh của một trường Đại học là một danh sách thí sinh, mỗi thí sinh có tên, số báo danh, điểm thi. Khi muốn liệt kê danh sách những thí sinh trúng tuyển tức là phải sắp xếp các thí sinh theo thứ tự từ điểm cao nhất tới điểm thấp nhất. Ở đây khoá sắp xếp chính là điểm thi.

Stt	Họ và tên	Ngày sinh	Giới tính	Điểm TB
1	Trần Văn Nam	12/12/1998	Nam	7.5
2	Nguyễn Thị Bé	22/10/1998	Nữ	9
3	Lê Tấn Hùng	01/01/1998	Nam	8.5
4	Đỗ Đăng Khôi	10/03/1997	Nam	6.5
5	Nguyễn Minh Trí	15/05/1998	Nam	8

Khi sắp xếp, các bản ghi trong bảng sẽ được đặt lại vào các vị trí sao cho giá trị khoá tương ứng với chúng có đúng thứ tự đã ấn định. Ta thấy rằng kích thước của khoá thường khá nhỏ so với kích thước của toàn bản ghi, nên nếu việc sắp xếp thực hiện trực tiếp trên các bản ghi sẽ đòi hỏi sự chuyển đổi vị trí của các bản ghi, kéo theo việc thường xuyên phải di chuyển, copy những vùng nhớ lớn, gây ra những tổn phí thời gian khá nhiều. Thường người ta khắc phục tình trạng này bằng cách xây dựng một bảng khoá: Mỗi bản ghi trong bảng ban đầu sẽ tương ứng với một bản ghi trong bảng khoá. Bảng khoá cũng gồm các bản ghi nhưng mỗi bản ghi chỉ gồm có hai trường:

- Trường thứ nhất chứa khoá
- Trường thứ hai chứa liên kết tới một bản ghi trong bảng ban đầu, tức là chứa một thông tin đủ để biết bản ghi tương ứng với nó trong bảng ban đầu là bản ghi nào.

Sau đó, việc sắp xếp được thực hiện trực tiếp trên bảng khoá đó. Như vậy, trong quá trình sắp xếp, bảng chính không hề bị ảnh hưởng gì, còn việc truy cập vào một bản ghi nào đó của bảng chính, khi cần thiết vẫn có thể thực hiện được bằng cách dựa vào trường liên kết của bản ghi tương ứng thuộc bảng khoá này.

Như ở ví dụ trên, ta có thể xây dựng bảng khoá gồm 2 trường, trường khoá chứa điểm và trường liên kết chứa số thứ tự của người có điểm tương ứng trong bảng ban đầu:

Stt	Điểm TB
1	7.5
2	9
3	8.5
4	6.5
5	8

Sau khi sắp xếp ta có bảng khoá sau:

Stt	Điểm TB
2	9
3	8.5
5	8
1	7.5
4	6.5

Dựa vào bảng khoá, ta có thể biết được rằng người có điểm cao nhất là người mang số thứ tự 2, tiếp theo là người mang số thứ tự 4, tiếp nữa là người mang số thứ tự 1, và cuối cùng là người mang số thứ tự 3, còn muốn liệt kê danh sách đầy đủ thì ta chỉ việc đối chiếu với bảng ban đầu và liệt kê theo thứ tự 2, 4, 1, 3.

Có thể còn cải tiến tốt hơn dựa vào nhận xét sau: Trong bảng khoá, nội dung của trường khoá hoàn toàn có thể suy ra được từ trường liên kết bằng cách: Dựa vào trường liên kết, tìm tới bản ghi tương ứng trong bảng chính rồi truy xuất trường khoá trong bảng chính. Như ví dụ trên thì người mang số thứ tự 1 chắc chắn sẽ phải có điểm thi là 20, còn người mang số thứ tự 3 thì chắc chắn phải có điểm thi là 18. Vậy thì bảng khoá có thể loại bỏ đi trường khoá mà chỉ giữ lại trường liên kết. Trong trường hợp các phần tử trong bảng ban đầu được đánh số từ 1 tới n và trường liên kết chính là số thứ tự của bản ghi trong bảng ban đầu như ở ví dụ trên, người ta gọi kỹ thuật này là **kỹ thuật sắp xếp bằng chỉ số**: Bảng ban đầu không hề bị ảnh hưởng gì cả, việc sắp xếp chỉ đơn thuần là đánh lại chỉ số cho các bản ghi theo thứ tự sắp xếp. Cụ thể hơn:

Nếu r[1], r[2], ..., r[n] là các bản ghi cần sắp xếp theo một thứ tự nhất định thì việc sắp xếp bằng chỉ số tức là xây dựng một dãy Index[1], Index[2], ..., Index[n] mà ở đây: Index[j] := Chỉ số của bản ghi sẽ đứng thứ j khi sắp thứ tự (Bản ghi r[index[j]] sẽ phải đứng sau j - 1 bản ghi khác khi sắp xếp)

Do khoá có vai trò đặc biệt như vậy nên sau này, khi trình bày các giải thuật, ta sẽ coi khoá như đại diện cho các bản ghi và để cho đơn giản, ta chỉ nói tới giá trị của khoá mà thôi. Các thao tác trong kỹ thuật sắp xếp lẽ ra là tác động lên toàn bản ghi giờ đây chỉ làm trên khoá. Còn việc cài đặt các phương pháp sắp xếp trên danh sách các bản ghi và kỹ thuật sắp xếp bằng chỉ số, ta coi như bài tập.

Bài toán sắp xếp giờ đây có thể phát biểu như sau:

Xét quan hệ thứ tự toàn phần "nhỏ hơn hoặc bằng" ký hiệu "≤" trên một tập hợp S, là quan hệ hai ngôi thoả mãn bốn tính chất:

Với ∀a, b, c ∈ S

- Tính phổ biến: Hoặc là a ≤ b, hoặc b ≤ a;
- Tính phản xạ: a ≤ a
- Tính phản đối xứng: Nếu a ≤ b và b ≤ a thì bắt buộc a = b.
- Tính bắc cầu: Nếu có a ≤ b và b ≤ c thì a ≤ c.

Trong trường hợp a ≤ b và a ≠ b, ta dùng ký hiệu "<" cho gọn

Cho một dãy gồm n khoá. Giữa hai khoá bất kỳ có quan hệ thứ tự toàn phần "≤". Xếp lại dãy các khoá đó để được dãy khoá thoả mãn a1≤ a2 ≤ ...≤ an.

Giả sử cấu trúc dữ liệu cho dãy khoá được mô tả như sau:

```
const n = ...;        //Số khoá trong dãy khoá
type
    TKey = ...;       //Kiểu dữ liệu một kho
    TArray = array[0..n + 1] of TKey;
var
    a: TArray;        //Dãy khoá có thêm 2 phần tử k0 và kn+1 để dùng cho một số
thuật toán
```

Thì những thuật toán sắp xếp dưới đây được viết dưới dạng thủ tục sắp xếp dãy khoá k, kiểu chỉ số đánh cho từng khoá trong dãy có thể coi là số nguyên Integer.

II. MỘT SỐ THUẬT TOÁN SẮP XẾP ĐƠN GIẢN

II.1. Sắp xếp kiểu chọn

➢ **Ý tưởng:**

- Ở lượt thứ nhất: chọn trong dãy khoá a1, a2, ..., an tìm ra khoá nhỏ nhất; sau đó đổi giá trị của khoá đó với a1
- Ở lượt thứ hai: chọn trong dãy khoá a2, a3, ..., an tìm ra khoá nhỏ nhất; sau đó đổi giá trị của khoá đó với a2
- ....
- Ở lượt thứ i: chọn trong dãy khoá ai, ai+1, ..., an tìm ra khoá nhỏ nhất; sau đó đổi giá trị của khoá đó với ai
- ....
- Ở lượt thứ n-1:chọn khoá nhỏ nhất trong 2 khoá an-1, an, sau đó đổi giá trị với an-1

➢ **Thuật toán**

```
Proc SelectSort(A, n)
    For i :=1 To n-1
    {       imin := i
            For j := i+1 To n
                    If aimin < aj Then
                            imin := j
```

```
                If imin <> i Then
                        ai ↔ aimin          //đổi 2 giá trị aimin và ai cho nhau
    }
Return
```

➢ **Độ phức tạp của thuật toán**

$$T(n)= \sum_{i=1}^{n-1}\sum_{j=1}^{i}1 = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1}n - \sum_{i=1}^{n-1}i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} = O(n^2)$$

II.2. Sắp xếp kiểu nổi bọt

➢ **Ý tưởng:**

Ở lượt thứ nhất: ta duyệt dãy khoá từ cuối dãy về đầu dãy (từ an đến a1), nếu gặp 2 khoá kề nhau mà không thoả mãn điều kiện sắp xếp thì đổi chỗ 2 khoá đó với nhau. Như vậy sau lượt này khoá a1 có giá trị nhỏ nhất

Tương tự như trên cho các phần tử từ a2 cho đến an-1

➢ **Thuật toán:**

```
Proc BubbleSort(A,n)
    For i :=2 To n
    {       For j := n To i Step -1
                If aj < aj-1 Then
                    aj ↔ aj-1
    }
Return
```

➢ **Độ phức tạp của thuật toán**

$$T(n)= \sum_{i=2}^{n}\sum_{j=i}^{n}1 = \sum_{i=2}^{n}(n-i) = \sum_{i=2}^{n}n - \sum_{i=2}^{n}i = n(n-1) - \frac{n(n-1)}{2} + 1 = \frac{n(n-1)}{2} + 1 = O(n^2)$$

II.3. Sắp xếp kiểu chèn

➢ **Ý tưởng:** Giả sử khoá a1 được coi là đã đúng vị trí của nó.

- Ở lượt thứ nhất: ta so sánh a1 với khoá a2; sau đó xen khoá a2 vào đúng vị trí của nó. Như vậy ta có dãy con có 2 khoá được sắp xếp
- Ở lượt thứ hai: ta đã có dãy con có 2 khoá a1 và a2 đã được sắp xếp; ta xen a3 vào 2 khoá trên vào đúng vị trí của nó để ta được dãy con có 3 khoá được sắp xếp
- ...
- Ở lượt thứ i: ta đã có dãy con có i khoá a1...ai đã được sắp xếp; ta xen ai+1 vào dãy con trên trên vào đúng vị trí của nó để ta được dãy con có i+1 khoá được sắp xếp
- ...
- Và tiếp tục cho đến hết lượt thứ n-1 thì ta được dãy khoá đã được sắp xếp

➢ **Thuật toán**

```
Proc InsertionSort(A,n)
    For i := 2 To n
    {       tam := ai               //giữ lại giá trị ai để chèn vào đúng vị trí
            j := i-1
            While j>0 And tam<aj Do     //vòng lặp xác định vị trí chèn
```

```
            {       aj+1 := aj              //dời aj>tam  ra sau một vị trí
                    j := j-1
            }
            aj+1 := tam             // đưa giá trị cần chèn vào đúng vị trí của nó
        }
    Return
```

➢ **Độ phức tạp của thuật toán**

Đối với thuật toán này thì độ phức tạp thuật toán phụ thuộc vào tình trạng của dãy ban đầu. Nếu coi tam<aj là phép toán thực hiện nhiều nhất thì

Trong trường hợp tốt nhất: ứng với dãy khoá đã được súp xếp thì mỗi lượt chỉ có 1 lần so sánh. Thì tổng số phép so sánh là n-1 lần.

Trong trường hợp xấu nhất (dãy đã được sắp xếp giảm) thì giống thuật toán SelectSort

Trong trường hợp trung bình: các giá trị khoá được coi là xuất hiện một cách ngẫu nhiên, có thể coi xác suất xuất hiện mỗi khoá là đồng khả năng, thì ở lượt thứ i thuật toán cần trung bình i/2 lần so sánh. Như vậy số phép so sánh là $\sum_{i=2}^{n} \frac{i}{2} = \frac{n(n+1)}{4} - \frac{1}{2}$

Và $T(n) = O(n^2)$

III. SẮP XẾP KIỂU PHÂN ĐOẠN (quick sort)

Quick Sort là một phương pháp sắp xếp tốt nhất, nghĩa là dù dãy khoá thuộc kiểu dữ liệu có thứ tự nào, Quick Sort cũng có thể sắp xếp được và không có một thuật toán sắp xếp nào nhanh hơn Quick Sort về mặt tốc độ trung bình (theo tôi biết). Người sáng lập ra nó là C.A.R. Hoare đã mạnh dạn đặt tên cho nó là sắp xếp "NHANH".

➢ **Ý tưởng:** Chọn một khoá một khoá ak trong dãy, dựa vào khoá ak để phân hoạch dãy thành 3 phần như sau:

- Phần 1 chứa các khoá có giá trị ≤ak. Các khoá này nằm bên trái của ak trong dãy
- Phần 2 là ak
- Phần 3 chứa các khoá có giá trị >ak. Các khoá này nằm bên phải của ak trong dãy

a_1	a_2	...	…	…	a_{k-1}	a_k	a_{k+1}	…	…	…	…	a_n
Các khoá ≤a_k							Các khoá > a_k					

Như vậy sau khi phân hoạch khoá ak sẽ nằm đúng vị trí của nó khi sắp xếp xong.

Với mỗi bên còn lại chứa các khoá chưa sắp xếp sẽ có độ dài ngắn hơn độ dài ban đầu và ta đi sắp xếp nó bằng phương pháp tương tự như trên.

➢ **Xây dựng 2 thuật toán như sau:**

Thuật toán phân hoạch để phân hoạch a1, a2, …, an thành 3 phần như trên

```
Proc Place(A, l, r, k)   // l là cận trái và r là cận phải của dãy khoá từ al, al+1,...,ar
                         //k là vị trí của ak sau khi phân hoạch
    If l<r Then          // trường hợp có số khoá >1
```

```
{       key := al               //chọn al làm giá trị để phân hoạch
        i := l+1
        j := r
        While i<j Do
        {       While ai< key Do i := i+1
                While aj ≥ key Do j := j -1
                If i<j Then
                        {ai ↔ aj
                          i := i+1
                          j := j-1
                        }
        }
        aj ↔ al
        k := j
}
Return
```

Thuật toán sắp xếp theo phương pháp QuickSort.

```
Proc QuickSort (A,l,r)
    If l<r Then
    {       Call Place(A,l,r,k)          //Phân hoạch dãy  thành 3 phần
            Call QuickSort(A,l,k-1)      //Gọi sắp xếp cho nửa bên trái
            Call QuickSort(A,k+1,r)      //Gọi sắp xếp cho nửa bên phải

    }
Return
```

- **Tính độ phức tạp**

Giả sử khoá ak sau khi phân hoạch nằm tại vị trí giữa của dãy khoá sau khi sắp xếp. gọi Tn là độ phức tạp của thuật toán QuickSort(A,l,r) (n=r-l+1). từ thuật toán ta có hệ thức truy đoán sau:

$$\begin{cases} T_n = 2T_{n/2} + O(\text{Place}) \\ T_1 = 0 \end{cases}$$

Trong thuật toán Place(A,l,r,k) chỉ số i và j chạy từ 2 đầu đến trung điểm của dãy ($\frac{l+r}{2}$). Như vậy độ phức tạp của thuật toán này tối đa là O(n). Thế vào hệ thức trên và giải ra ta có độ phức tạp của thuật toán là O(nlgn).

- **Cái tiến thuật toán Quick sort**

Việc chọn chốt cho phép phân đoạn quyết định hiệu quả của Quick Sort, nếu chọn chốt không tốt, rất có thể việc phân đoạn bị suy biến thành trường hợp xấu khiến Quick Sort hoạt động chậm và tràn ngăn xếp chương trình con khi gặp phải dây chuyền đệ qui quá dài. Một cải tiến sau có thể khắc phục được hiện tượng tràn ngăn xếp nhưng cũng hết sức chậm trong trường hợp xấu, kỹ thuật này khi đã phân được [L, H] được hai đoạn con [L, j] và [i, H] thì chỉ gọi đệ quy để tiếp tục đối với đoạn ngắn, và lặp lại quá trình phân đoạn đối với đoạn dài.

```
proc QuickSort;
    proc Partition(L, H: Integer); //Sắp xếp đoạn từ kL, kL+1, ..., kH}
    do {
```

```
                if L ≥ H then Exit;
                <Phân đoạn [L, H] được hai đoạn con [L, j] và [i, R]>
                if <đoạn [L, j] ngắn hơn đoạn [i, R]> then
                {
                        Partition(L, j); L := i;
                }
                else
                {
                        Partition(i, R); R := j;
                }
        } while True;
        Return
        Partition(1, n);
Return
```

Cải tiến thứ hai đối với Quick Sort là quá trình phân đoạn nên chỉ làm đến một mức nào đó, đến khi đoạn đang xét có độ dài ≤ M (M là một số nguyên tự chọn nằm trong khoảng từ 9 tới 25) thì không phân đoạn tiếp mà nên áp dụng thuật toán sắp xếp kiểu chèn.

Cải tiến thứ ba của Quick Sort là: Nên lấy trung vị của một dãy con trong đoạn để làm chốt, (trung vị của một dãy n phần tử là phần tử đứng thứ n / 2 khi sắp thứ tự). Cách chọn được đánh giá cao nhất là chọn trung vị của ba phần tử đầu, giữa và cuối đoạn.

➢ **Nhận xét:**

Quick Sort là một công cụ sắp xếp mạnh, chỉ có điều khó chịu gặp phải là trường hợp suy biến của Quick Sort (quá trình phân đoạn chia thành một dãy rất ngắn và một dãy rất dài). Và điều này trên phương diện lý thuyết là không thể khắc phục được: Ví dụ với n = 10000. Nếu như chọn chốt là khoá đầu đoạn (Thay dòng chọn khoá chốt bằng Key := kL) hay chọn chốt là khoá cuối đoạn (Thay bằng Key := kH) thì với dãy sau, chương trình hoạt động rất chậm: (1, 2, 3, 4, 5, ..., 9999, 10000)

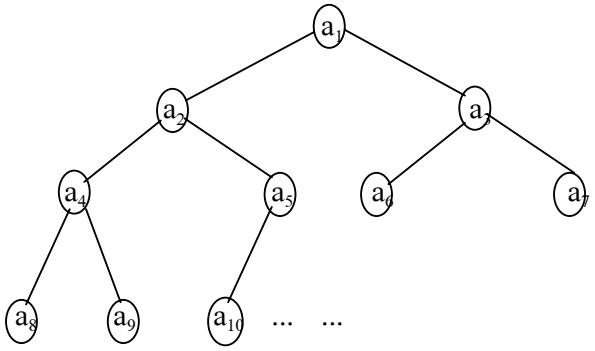
- Nếu như chọn chốt là khoá giữa đoạn (Thay dòng chọn khoá chốt bằng Key := k(L+H) div 2) thì với dãy sau, chương trình cũng rất chậm:
  (1, 2, ..., 4999, 5000, 5000, 4999, ..., 2, 1)
- Trong trường hợp chọn chốt là trung vị dãy con hay chọn chốt ngẫu nhiên, thật khó có thể tìm ra một bộ dữ liệu khiến cho Quick Sort hoạt động chậm. Nhưng ta cũng cần hiểu rằng với mọi chiến lược chọn chốt, trong 10000! dãy hoán vị của dãy (1, 2, ... 10000) thế nào cũng có một dãy làm Quick Sort bị suy biến, tuy nhiên trong trường hợp chọn chốt ngẫu nhiên, xác suất xảy ra dãy này quá nhỏ tới mức ta không cần phải tính đến, như vậy khi đã chọn chốt ngẫu nhiên thì ta không cần phải quan tâm tới ngăn xếp đệ quy, không cần quan tâm tới kỹ thuật khử đệ quy và vấn đề suy biến của Quick Sort.

IV. SẮP XẾP KIỂU VUN ĐỐNG

➢ **Khái niệm**

Đống là cây nhị phân hoàn chỉnh đặc biệt mà giá trị lưu tại mọi nút (trừ lá) đều lớn hơn hoặc bằng giá trị của 2 nút con của nó.

Trong chương cây, với một dãy khoá a1, a2,…,an có thể biểu diễn bằng một cây nhị phân hoàn chỉnh như sau:



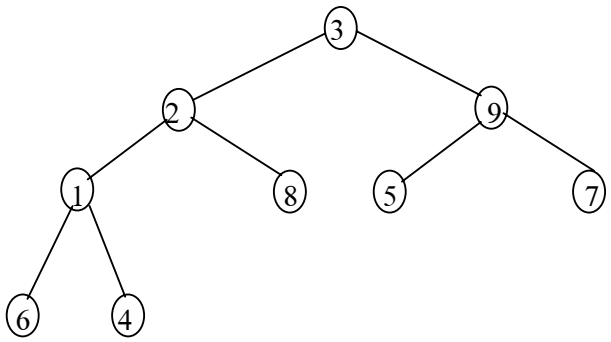
Ta xây dựng một số khái niệm sau:

- Lá là Đống
- Một nút không phải lá là đống nếu nó lớn hơn hoặc bằng cả 2 nút con

Như vậy để một cây nhị phân là đồng thì mọi nút trong cây là đống. Khi một cây là đống thì ta có nút gốc có giá trị lớn nhất.

➢ **Xây dựng thuật toán**

Giả sử dãy có 9 khoá sau: (3,2,9,1,8,5,7,6,4), thì ta có cây nhị phân hoàn chỉnh tương ứng:

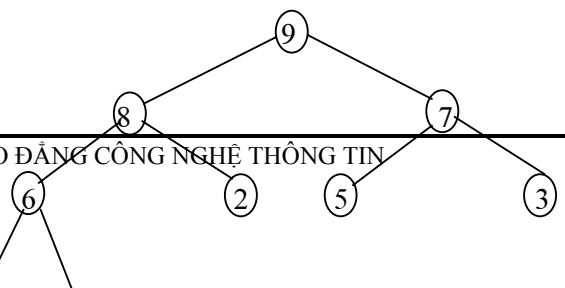


Bước 1: Tạo đống cho cây nhị phân

Vì cây nhị phân chỉ gồm có một nút hiển nhiên là đống, nên **để vun một nhánh cây gốc r thành đống, ta có thể coi hai nhánh con của nó (nhánh gốc 2r và 2r + 1) đã là đống rồi**. Và thuật toán vun đống sẽ được tiến hành từ dưới lên (bottom-up) đối với cây: Gọi h là chiều cao của cây, nút ở mức h (nút lá) đã là gốc một đống, ta vun lên để những nút ở mức h - 1 cũng là gốc của đống, ... cứ như vậy cho tới nút ở mức 1 (nút gốc) cũng là gốc của đống.

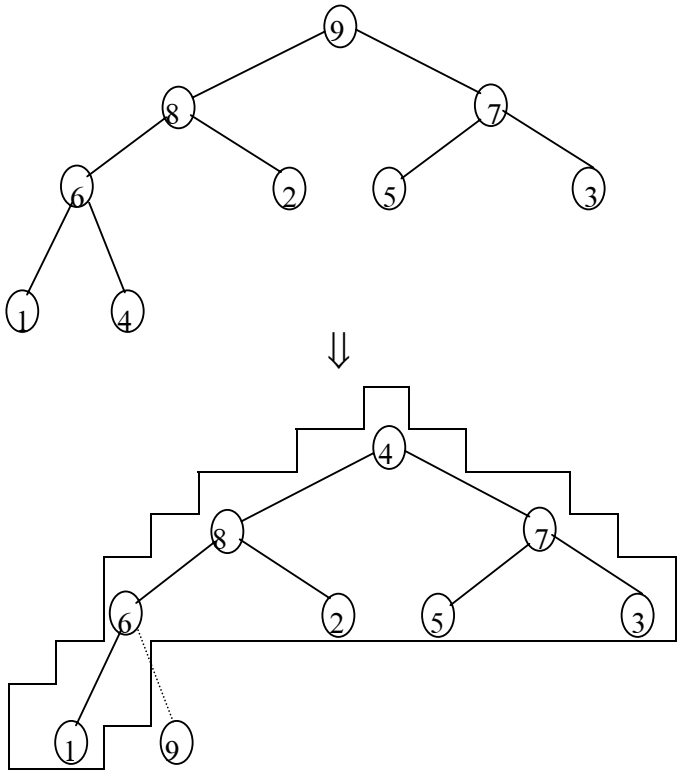
Thuật toán vun thành đống đối với cây gốc r, hai nhánh con của r đã là đống rồi

Giả sử ở nút r chứa giá trị V. Từ r, ta cứ đi tới nút con chứa giá trị lớn nhất trong 2 nút con, cho tới khi gặp phải một nút c mà mọi nút con của c đều chứa giá trị ≤ V (nút lá cũng là trường hợp riêng của điều kiện này). Dọc trên đường đi từ r tới c, ta đẩy giá trị chứa ở nút con lên nút cha và đặt giá trị V vào nút c. Sau khi tạo đống cho cây ta được cây nhị phân như sau:



Bước 2: thực hiện quá trình sắp xếp

Sau khi tạo đống cho cây, ta được nút gốc là lớn nhất ta hoán vị nút gốc với nút cuối của cây, sau đó tạo đống lại cho cây (thực chất là nút gốc) cho cây mà không có nút cuối của cây (bớt đi nút có giá trị lớn nhất vừa hoán vị). Quá trình này được thực hiện tiếp tục cho đến khi cây chỉ còn một nút. Quá trình trên được mô phỏng bằng hình ảnh sau:



Như vậy, mấu chốt ở đây là thuật toán tạo đống khoá ở nút thứ i trong cây có n nút, dĩ nhiên các nút không phải lá ở mức lớn hơn mức của nút i đã là đống.

Thuật toán Heap Sort có hai thủ tục chính:

- Thủ tục Adjust(root, endnode) vun cây gốc root thành đống trong điều kiện hai cây gốc 2.root và 2.root +1 đã là đống rồi. Các nút từ endnode + 1 tới n đã nằm ở vị trí đúng và không được tính tới nữa.
- Thủ tục Heap Sort mô tả lại quá trình vun đống và chọn phần tử theo ý tưởng trên:

```
Proc Adjust(i,A,n) //Tạo đống cho nút i trong dãy a1, a2, ...,an.
    x := ai
    c := 2*i                    //c là chỉ số nút con trái
    While c ≤ n Do
```

```
    { If c<n And ac< ac+1 Then c := c+1  //c là chỉ số nút lớn nhất trong 2 nút con
        If x<ac Then            //vi phạm tính chất đống
        {       ai := ac        // Chuyển khoá ac lên mức cao hơn
                i := c          //xác định vị trí mới cho x
                c := 2*i        //c là con trái của x
        }
        Else c := n+1           //điều kiện để dừng vòng lặp
    }
    ai := x                     //đặt x vào đúng vị trí của nó trên cây
Return
```

Thuật toán HeapSort như sau:

```
Proc HeapSort (A,n)
                n
                2
    For i ← ⌊ ⌋  To 1 Step -1   //Vòng lặp tạo đống cho cây
        Call Adjust(i,A,n)      // i =n/2..1 là những nút có quyền làm cha
    For i ← n To 2 Step -1
    {   a1 ↔ ai                 //Hoán vị nút đầu với nút cuối trên cây
        Call Adjust(1, A, i-1)  //Tạo đống lại cho cây mà không có nút cuối
    }
Return
```

➢ **Độ phức tạp của thuật toán**

Trong thuật toán tạo đống cho khoá ở nút i, ta nhận thấy trong trường hợp vòng lặp thực hiện nhiều nhất là tạo đống cho gốc với giá trị được coi là nhỏ nhất và vòng lặp thực hiện từ gốc đến hết chiều cao của cây hoàn chỉnh. Nên độ phức tạp của thuật toán là O(lgn)

Vậy độ phức tạp của thuật toán HeapSort là O(nlgn).

Có thể nhận xét thêm rằng Quick Sort đệ quy cần thêm không gian nhớ cho Stack, còn Heap Sort ngoài một nút nhớ phụ để thực hiện việc đổi chỗ, nó không cần dùng thêm gì khác. Heap Sort tốt hơn Quick Sort về phương diện lý thuyết bởi không có trường hợp tồi tệ nào Heap Sort có thể mắc phải. Cũng nhờ có Heap Sort mà giờ đây khi giải mọi bài toán có chứa mô-đun sắp xếp, ta có thể nói rằng cấp độ phức tạp của thủ tục sắp xếp đó không quá O(nlgn).

V. MỘT SỐ THUẬT TOÁN KHÁC

V.1. Phương pháp đếm

Điều kiện: các khoá sắp xếp phải là kiểu nguyên và các giá trị của các khoá nằm trong một khoảng xác định [x..y] biết trước

Ý tưởng:Do các khoá có kiểu nguyên và có giá trị nằm trong khoảng [x..y] xác định trước nên ta có thể coi khoá chính là chỉ số của mảng có kích thước y-x+1 phần tử. Từ ý tưởng trên ta sử dụng mảng Count có kích thước y-x+1 với chỉ số mảng là từ a đến b. Thuật toán thực hiện qua các bước sau:

- Bước 1: Khởi tạo mảng Count với các phần tử trong mảng bằng 0
- Bước 2: Đếm xem mỗi khoá của dãy ai xuất hiện mấy lần trong dãy nhờ vào mảng Count
- Bước 3: Dựa vào giá trị Counti để biết được i hiện diện trong dãy các khoá a1,..,an không? và nếu có thì nó xuất hiện bao nhiêu lần?

➢ **Thuật toán**

```
Proc CountSort(A,n)      // giả sử [x..y] là khoảng chứa các khoá ai
    For i ←x To y        //Khởi tạo mảng Count
        Counti ←0
    For i← 1 To n
        Count_ai ← Count_ai +1
                         //đánh dấu khoá ai là chỉ số ở mảng Count
    n ←0
    For i←x To y
        If Counti >0 Then    //xác định chỉ số i là một khoá trong dãy a1,...,an
            For j ← 1 To Counti //xác đinh có bao nhiêu khoá i trong dãy
            {   n ← n+1      //xác định lại vị trí của khoá i
                an ← i
            }
Return
```

Độ phức tạp của thuật toán là O(n). Tuy nhiên thuật toán này cần sử dụng bộ nhớ lớn cho dù kích thước dãy a1, a2, …,an có thể nhỏ

V.2. Phương pháp dùng hàng đợi

Có thể dùng 10 Queue để sắp xếp. Các Queue được đánh số từ 0..9.

- Bước 1: Đưa các khoá vào queue có chỉ số là hàng đơn vị của khoá. Sau đó đuyệt qua 10 queue lấy các giá trị đưa lại vào dãy khoá a1,…,an
- Bước 2: Đưa các khoá vào queue có chỉ số là hàng chục của khoá. Sau đó đuyệt qua 10 queue lấy các giá trị đưa lại vào dãy khoá a1,…,an
- …
- Bước k: Đưa các khoá vào queue có chỉ số là hàng thứ k (tính từ hàng đơn vị là 1)của khoá. Sau đó đuyệt qua 10 queue lấy các giá trị đưa lại vào dãy khoá a1,…,an

➢ **Thuật toán**

```
Proc RadixSort(A,n)
    For i←0 To 9
        CREATQ(Qi)
    k ← Len(Max(a1,a2,...,an)) //k là số các con số của khoá lớn nhất trong dãy
    For l ←1 To k
    {   For i ←1 To n
        {j ← (ai Div 10i-1) Mod 10i  //xác định chỉ số j của queue sử dụng
        ADD(ai, Qj )                 //đưa khoá ai vào queue j
        }
        n ←0
        For i ←0 To 9                        //duyệt qua 10 queue
            If Not EMPTYQ(Qi) Then     //trường hợp Qi có giá trị
                While Not EMPTYQ(Qi) Do  //Lấy các khoá trong queue
                    { n ← n+1
                    an ← FRONT(Qi) //đưa vào dãy a1,...,an
                    REMOVE(Qi)
                    }
```

}
Return

Độ phức tạp của thuật toán này là O(n).

V.3. Phương pháp sắp xếp trộn

➢ **Phép trộn hai đường trực tiếp**

Phép trộn 2 đường là phép hợp nhất hai dãy khoá đã sắp xếp để ghép lại thành một dãy khoá có kích thước bằng tổng kích thước của hai dãy khoá ban đầu và dãy khoá tạo thành cũng có thứ tự sắp xếp. Nguyên tắc thực hiện của nó khá đơn giản: so sánh hai khoá đứng đầu hai dãy, chọn ra khoá nhỏ nhất và đưa nó vào miền sắp xếp (một dãy khoá phụ có kích thước bằng tổng kích thước hai dãy khoá ban đầu) ở vị trí thích hợp. Sau đó, khoá này bị loại ra khỏi dãy khoá chứa nó. Quá trình tiếp tục cho tới khi một trong hai dãy khoá đã cạn, khi đó chỉ cần chuyển toàn bộ dãy khoá còn lại ra miền sắp xếp là xong.

Ví dụ: Với hai dãy khoá: (1, 3, 10, 11) và (2, 4, 9)

Dãy 1	**Dãy 2**	**Khoá nhỏ nhất trong 2 dãy**	**Miền sắp xếp**
(1, 3, 10, 11)	(2, 4, 9)	1	(1)
(3, 10, 11)	(2, 4, 9)	2	(1, 2)
(3, 10, 11)	(4, 9)	3	(1, 2, 3)
(10, 11)	(4, 9)	4	(1, 2, 3, 4)
(10, 11)	(9)	9	(1, 2, 3, 4, 9)
(10, 11)	∅	Dãy 2 là ∅, đưa nốt dãy 1 vào miền sắp xếp	(1, 2, 3, 4, 9, 10, 11)

➢ **Thuật toán sắp xếp bằng trộn 2 đường trực tiếp**

Ta có thể coi mỗi khoá trong dãy khoá k1, k2, ..., kn là một mạch với độ dài 1, các mạch trong dãy đã được sắp xếp rồi:

3	6	4	5	8	9	1	0	2	7

Trộn hai mạch liên tiếp lại thành một mạch có độ dài 2, ta lại được dãy gồm các mạch đã được sắp:

3 6	4 5	8 9	0 1	2 7

Cứ trộn hai mạch liên tiếp, ta được một mạch độ dài lớn hơn, số mạch trong dãy sẽ giảm dần xuống

3 4 5 6	0 1 8 9	2 7

0 1 3 4 5 6 8 9	2 7

0 1 2 3 4 5 6 7 8 9

Để tiến hành thuật toán sắp xếp trộn hai đường trực tiếp, ta viết các thủ tục:

- Thủ tục Merge(var x, y: TArray; a, b, c: Integer); thủ tục này trộn mạch xa, xa+1, ..., xb với mạch xb+1, xb+2 ..., xc để được mạch ya, ya+1, ..., yc.

- Thủ tục MergeByLength(var x, y: TArray; len: Integer); thủ tục này trộn lần lượt các cặp mạch theo thứ tự:
  - ♦ Trộn mạch x1...xlen và xlen+1...x2len thành mạch y1...y2len.
  - ♦ Trộn mạch x2len+1...x3len và x3len+1 ...x4len thành mạch y2len+1...y4len.

Lưu ý rằng đến cuối cùng ta có thể gặp hai trường hợp: Hoặc còn lại hai mạch mà mạch thứ hai có độ dài < len. Hoặc chỉ còn lại một mạch. Trường hợp thứ nhất ta phải quản lý chính xác các chỉ số để thực hiện phép trộn, còn trường hợp thứ hai thì không được quên thao tác đưa thẳng mạch duy nhất còn lại sang dãy y.

- Cuối cùng là thủ tục MergeSort, thủ tục này cần một dãy khoá phụ t1, t2, ..., tn. Trước hết ta gọi MergeByLength(k, t, 1) để trộn hai phần tử liên tiếp của k thành một mạch trong t, sau đó lại gọi MergeByLength(t, k, 2) để trộn hai mạch liên tiếp trong t thành một mạch trong k, rồi lại gọi MergeByLength(k, t, 4) để trộn hai mạch liên tiếp trong k thành một mạch trong t ...Như vậy k và t được sử dụng với vai trò luân phiên: một dãy chứa các mạch và một dãy dùng để trộn các cặp mạch liên tiếp để được mạch lớn hơn.

```
proc Merge(var X, Y: TArray; a, b, c: Integer);{Trộn Xa...Xb và Xb+1...Xc}
//Chỉ số p chạy trong miền sắp xếp, i chạy theo mạch thứ nhất, j chạy theo mạch thứ hai
    p := a; i := a; j := b + 1;
    while (i ≤ b) and (j ≤ c) then   /Chừng nào cả hai mạch đều chưa xét hết
    {       if Xi ≤ Xj then //So sánh hai phần tử nhỏ nhất trong hai mạch mà chưa
                            bị đưa vào miền sắp xếp
            {Yp := Xi; i := i + 1;  //Đưa x vào miền sắp xếp và cho i chạy
            }
            else
            {Yp := Xj; j := j + 1;  Đưa x vào miền sắp xếp và cho j chạy
            }
            p := p + 1;
    }
    if i ≤ b then     //Mạch 2 hết trước
    (Yp, Yp+1, ..., Yc) := (Xi, Xi+1, ..., Xb)//Đưa phần cuối của mạch 1 vào miến
                                                sắp xếp
    else     //Mạch 1 hết trước
    (Yp, Yp+1, ..., Yc) := (Xj, Xj+1, ..., Xc);       //Đưa phần cuối của mạch 2 vào
                                                miến sắp xếp
Return

proc MergeByLength(var X, Y: TArray; len: Integer)
    a := 1; b := len; c := 2 * len;
    while c ≤ n do //Trộn hai mạch xa...xb và xb+1...xc đều có độ dài len
            {Merge(X, Y, a, b, c);
            //Dịch các chỉ số a, b, c về sau 2.len vị trí
            a := a + 2 * len; b := b + 2 * len; c := c + 2 * len;
            }
    if b < n then Merge(X, Y, a, b, n) //Còn lại hai mạch mà mạch thứ hai có độ dài
                                        ngắn hơn len
    else
            if a ≤ n then //Còn lại một mạch
    (Ya, Ya+1, ..., Yn) := (Xa, Xa+1, ..., Xn); //Đưa thẳng mạch đó sang miền y}
Return
```

```
proc MergeSort;   //Thuật toán sắp xếp trộn
    Flag := True;
    len := 1;
    while len < n do
            {if Flag then MergeByLength(k, t, len) else MergeByLength(t, k, len);
            len := len * 2;
            Flag := not Flag;       //Đảo cờ để luân phiên vai trò của k và t}
            }
    if not Flag then k := t;        //Nếu kết quả cuối cùng đang nằm trong t thì sao
                                    chép kết quả vào k
Return
```

Về cấp độ phức tạp của thuật toán, ta thấy rằng trong thủ tục Merge, phép toán tích cực là thao tác đưa một khoá vào miền sắp xếp. Mỗi lần gọi thủ tục MergeByLength, tất cả các phần tử trong dãy khoá được chuyển hoàn toàn sang miền sắp xếp, nên cấp phức tạp của thủ tục MergeByLength là O(n). Thủ tục MergeSort có vòng lặp thực hiện không quá log2n + 1 lời gọi MergeByLength bởi biến len sẽ được tăng theo cấp số nhân công bội 2. Từ đó suy ra cấp độ phức tạp của MergeSort là O(nlog2n) bất chấp trạng thái dữ liệu vào.

Cùng là những thuật toán sắp xếp tổng quát với độ phức tạp trung bình như nhau, nhưng không giống như QuickSort hay HeapSort, MergeSort có tính ổn định. Nhược điểm của MergeSort là nó phải dùng thêm một vùng nhớ để chứa dãy khoá phụ có kích thước bằng dãy khoá ban đầu.

Người ta còn có thể lợi dụng được trạng thái dữ liệu vào để khiến MergeSort chạy nhanh hơn: ngay từ đầu, ta không coi mỗi phần tử của dãy khoá là một mạch mà coi những đoạn đã được sắp trong dãy khoá là một mạch. Bởi một dãy khoá bất kỳ có thể coi là gồm các mạch đã sắp xếp nằm liên tiếp nhau. Khi đó người ta gọi phương pháp này là phương pháp trộn hai đường tự nhiên.

Tổng quát hơn nữa, thay vì phép trộn hai mạch, người ta có thể sử dụng phép trộn k mạch, khi đó ta được thuật toán sắp xếp trộn k đường.

CHƯƠNG 5

CÁC THUẬT TOÁN TÌM KIẾM

I. BÀI TOÁN TÌM KIẾM

Cùng với sắp xếp, tìm kiếm thường xuyên được đề cập đến trong các ứng dụng tin học. Ta có thể hình dung bài toán tìm kiếm như sau:

Cho một dãy gồm n bản ghi r1, r2, …, rn. mỗi bản ghi ri tương ứng với khoá ki. Hãy tìm một bản ghi có giá trị khoá bằng x cho trước. Việc tìm kiếm hoàn thành có thể xãy ra một trong hai tình huống sau:

- Tìm được bản ghi có khoá tương ứng bằng x, lúc đó phép tìm kiếm thành công (true)
- Không tìm được bản ghi nào có khoá tìm kiếm bằng x cả, phép tìm kiếm thất bại (false)

Cũng như trong sắp xếp, sử dụng dãy khoá chứa các giá trị nguyên để trình bày thuật toán.

Tương tự như sắp xếp, ta coi khoá của một bản ghi là đại diện cho bản ghi đó. Và trong một số thuật toán sẽ trình bày dưới đây, ta coi kiểu dữ liệu cho mỗi khoá cũng có tên gọi là TKey.

```
const n = ...;        //Số khoá trong dãy khoá
type TKey = ...;      {Kiểu dữ liệu một khoá}
    TArray = array[0..n + 1] of TKey;
var k: TArray;        // Dãy khoá có thêm 2 phần tử k0 và kn+1 để dùng cho một số
                      thuật toán
```

II. TÌM KIẾM TUẦN TỰ

Đây là kỹ thuật tìm kiếm đơn giản. Bắt đầu từ khoá đầu tiên, lần lượt so sánh khoá x với khoá tương ứng trong dãy. Quá trình tìm kiếm kết thúc khi tìm được khoá thoả mãn hoặc đi đến hết dãy hoặc gặp điều kiện dừng vòng lặp. Có 2 thuật toán tìm tuần tự trên dãy khoá đầu vào khác nhau.

➢ **Trên dãy khoá chưa sắp xếp**

```
Func Sequential_1(x,A,n)
    i := 1
    While i <=n And ai <>x Do i := i+1
    Sequential_1 := (i <=n)
Return
```

➢ **Trên dãy khoá đã được sắp xếp**

```
Func Sequential_2(x,A,n)
    i := 1
    While i <=n And ai <x Do i := i+1
    If ai =x Then Tuan_Tu2 := True
    Else Sequential_2 := False
Return
```

Dễ thấy rằng cấp độ phức tạp của thuật toán tìm kiếm tuần tự trong trường hợp tốt nhất là O(1), trong trường hợp xấu nhất là O(n) và trong trường hợp trung bình cũng là O(n).

III.TÌM KIẾM NHỊ PHÂN

Phép tìm kiếm nhị phân được thực hiện trên dãy khoá có thứ tự a1≤a2≤…≤an.

Chia đôi dãy khoá cần tìm kiếm. So sánh khoá giữa dãy với x, có 3 trường hợp xãy ra:

- Giá trị khoá này bằng x, tìm kiếm thành công
- Giá trị khoá này lớn hơn x, thì ta tiến hành tìm x với nữa bên trái của khoá này
- Giá trị khoá này nhỏ hơn x, thì ta tiến hành tìm x với nữa bên phải của khoá này

Trường hợp tìm kiếm thất bại khi dãy khoá cần tìm không có phần tử nào.

➢ **Thuật toán**

```
Proc BinarySearch(x,A,n)// Tìm khoá x trong dãy a1,a2…,an
    left ←1              //Left trỏ về chỉ số đầu dãy
    right ←n             //right trỏ về vị trí cuối dãy
    found ← False//dùng để xác tìm thành công hay không
    While left <= right And Not found Do
    {       mid ← (left + right) Div 2     //mid ở giữa dãy
            If amid = x Then               //trường hợp tìm thấy dừng thuật toán
            found ← True
            Else
            If amid <x Then
                    left ← mid +1 //xác định lại đoạn tìm tiếp theo là bên phải
            Else
                    right ← mid -1//xác định lại đoạn tìm tiếp theo là bên trái
    }
    BinarySearch ←found
Return
```

Người ta đã chứng minh được độ phức tạp tính toán của thuật toán tìm kiếm nhị phân trong trường hợp tốt nhất là O(1), trong trường hợp xấu nhất là O(log2n) và trong trường hợp trung bình cũng là O(log2n). Tuy nhiên, ta không nên quên rằng trước khi sử dụng tìm kiếm nhị phân, dãy khoá phải được sắp xếp rồi, tức là thời gian chi phí cho việc sắp xếp cũng phải tính đến. Nếu dãy khoá luôn luôn biến động bởi phép bổ sung hay loại bớt đi thì lúc đó chi phí cho sắp xếp lại nổi lên rất rõ làm bộc lộ nhược điểm của phương pháp này.

IV. PHÉP BĂM (HASH)

Tư tưởng của phép băm là dựa vào giá trị các khoá k1, k2, ..., kn, chia các khoá đó ra thành các nhóm. Những khoá thuộc cùng một nhóm có một đặc điểm chung và đặc điểm này không có trong các nhóm khác. Khi có một khoá tìm kiếm X, trước hết ta xác định xem nếu X thuộc vào dãy khoá đã cho thì nó phải thuộc nhóm nào và tiến hành tìm kiếm trên nhóm đó.

Một ví dụ là trong cuốn từ điển, các bạn sinh viên thường dán vào 26 mảnh giấy nhỏ vào các trang để đánh dấu trang nào là trang khởi đầu của một đoạn chứa các từ có cùng chữ cái đầu. Để khi tra từ chỉ cần tìm trong các trang chứa những từ có cùng chữ cái đầu với từ cần tìm.

Một ví dụ khác là trên dãy các khoá số tự nhiên, ta có thể chia nó là làm m nhóm, mỗi nhóm gồm các khoá đồng dư theo mô-đun m.

Có nhiều cách cài đặt phép băm:

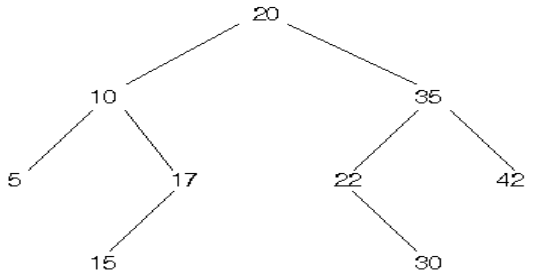
- Cách thứ nhất là chia dãy khoá làm các đoạn, mỗi đoạn chứa những khoá thuộc cùng một nhóm và ghi nhận lại vị trí các đoạn đó. Để khi có khoá tìm kiếm, có thể xác định được ngay cần phải tìm khoá đó trong đoạn nào.
- Cách thứ hai là chia dãy khoá làm m nhóm, Mỗi nhóm là một danh sách nối đơn chứa các giá trị khoá và ghi nhận lại chốt của mỗi danh sách nối đơn. Với một khoá tìm kiếm, ta xác định được phải tìm khoá đó trong danh sách nối đơn nào và tiến hành tìm kiếm tuần tự trên danh sách nối đơn đó. Với cách lưu trữ này, việc bổ sung cũng như loại bỏ một giá trị khỏi tập hợp khoá dễ dàng hơn rất nhiều phương pháp trên.
- Cách thứ ba là nếu chia dãy khoá làm m nhóm, mỗi nhóm được lưu trữ dưới dạng cây nhị phân tìm kiếm và ghi nhận lại gốc của các cây nhị phân tìm kiếm đó, phương pháp này có thể nói là tốt hơn hai phương pháp trên, tuy nhiên dãy khoá phải có quan hệ thứ tự toàn phần thì mới làm được.

V. CÂY TÌM KIẾM NHỊ PHÂN

V.1. Định nghĩa

Cây tìm kiếm nhị phân (TKNP) là cây nhị phân mà khoá tại mỗi nút cây lớn hơn khoá của tất cả các nút thuộc cây con bên trái và nhỏ hơn khoá của tất cả các nút thuộc cây con bên phải.

Minh hoạ một cây TKNP có khoá là số nguyên (với quan hệ thứ tự trong tập số nguyên).



Qui ước: Cũng như tất cả các cấu trúc khác, ta coi cây rỗng là cây TKNP

Nhận xét:

- Trên cây TKNP không có hai nút cùng khoá.
- Cây con của một cây TKNP là cây TKNP.
- Khi duyệt trung tự (InOrder) cây TKNP ta được một dãy có thứ tự tăng. Chẳng hạn duyệt trung tự cây trên ta có dãy: 5, 10, 15, 17, 20, 22, 30, 35, 42.

V.2. Cài đặt cây tìm kiếm nhị phân

Có thể áp dụng các cách cài đặt như đã trình bày trong phần cây nhị phân để cài đặt cây TKNP. Nhưng sẽ có nhiều sự khác biệt trong các giải thuật thao tác trên cây TKNP như tìm kiếm, thêm hoặc xoá một nút trên cây TKNP để luôn đảm bảo tính chất cuả cây TKNP. Thông thường sử dụng con trỏ để cài đặt cây TKNP.

Giả sử con trỏ trỏ vào cây TKNP là Root. Sau đây là các thao tác trên cây TKNP

- **Khởi tạo cây TKNP rỗng** *Root ←Null*
- **Tìm kiếm một nút có khoá cho trước trên cây TKNP**

Để tìm kiếm 1 nút có khoá x trên cây TKNP, ta tiến hành từ nút gốc bằng cách so sánh khoá cuả nút gốc với khoá x.

- Nếu nút gốc bằng NIL thì không có khoá x trên cây.
- Nếu x bằng khoá của nút gốc thì giải thuật dừng và ta đã tìm được nút chứa khoá x.
- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) việc tìm khoá x trên cây con bên phải.
- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) việc tìm khoá x trên cây con bên trái.

Ví dụ: tìm nút có khoá 30 trong cây TKNP trên

- So sánh 30 với khoá nút gốc là 20, vì 30 > 20 vậy ta tìm tiếp trên cây con bên phải, tức là cây có nút gốc có khoá là 35.
- So sánh 30 với khoá của nút gốc là 35, vì 30 < 35 vậy ta tìm tiếp trên cây con bên trái, tức là cây có nút gốc có khoá là 22.
- So sánh 30 với khóa của nút gốc là 22, vì 30 > 22 vậy ta tìm kiếm trên cây con bín phải, tức là cây có nút gốc có khoá là 30.
- So sánh 30 với khoá nút gốc là 30, 30 = 30 vậy đến đây giải thuật dừng và ta tìm được nút chứa khoá cần tìm.

Hàm dưới đây trả về kết quả là con trỏ trỏ tới nút chứa khoá x hoặc Null nếu không tìm thấy khoá x trên cây TKNP.

```
Func Search(x,Root)
    If Root = Null Then
            Search := Null                  //không tìm thấy khoá x
    Else
            If Info(Root) = x Then
                    Search := Root          //tìm thấy khoá x
            Else
                    If Info(Root) < x Then
                    Search := Search(x,Right(Root))     //tìm tiếp trên cây bên phải
                    Else
                    Search := Search(x,Left(Root))      //tìm tiếp trên cây bên trái
Return
```

Tuy nhiên có thể sử dụng giải thuật không đệ qui như sau

```
Func Search(x,Root)
    p := Root
    While p<>Null And Info(p)<>x Do
            If Info(p)<x Then
                    p := Right(p)           //Tìm kiếm bên cây con phải
            Else
                    p := Left(p)            //Tìm kiếm bên cây con trái
    Search := p
Return
```

- **Thêm một nút có khoá cho trước vào cây tìm kiếm nhị phân**

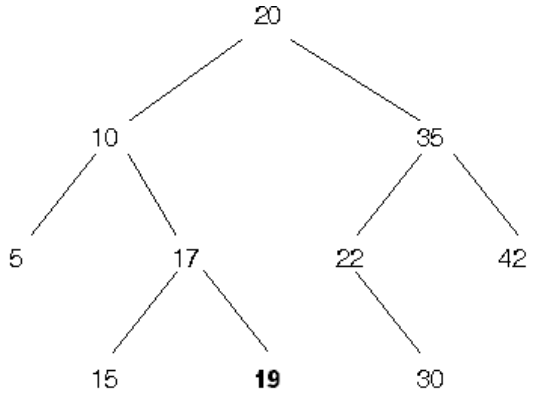
Trong quá trình chèn 1 giá trị mới vào cây TKNP, nếu đã có x trong cây thì không thực hiện chèn. trường hợp chưa có thì ta chèn x vào cây cho thoả tính chất cây TKNP. Giải thuật đệ qui cụ thể như sau:

Ta tiến hành từ nút gốc bằng cách so sánh khoá cuả nút gốc với khoá x.

- Nếu nút gốc bằng Null thì khoá x chưa có trên cây, do đó ta thêm nút mới chứa khoá x.
- Nếu x bằng khoá của nút gốc thì giải thuật dừng, trường hợp này ta không thêm nút.
- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên phải.
- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên trái.

Ví dụ: thêm khoá 19 vào cây TKNP ở trên

- So sánh 19 với khoá của nút gốc là 20, vì 19 < 20 vậy ta xét tiếp đến cây bên trái, tức là cây có nút gốc có khoá là10.
- So sánh 19 với khoá của nút gốc là 10, vì 19 > 10 vậy ta xét tiếp đến cây bên phải, tức là cây có nút gốc có khoá là 17.
- So sánh 19 với khoá của nút gốc là 17, vì 19 > 17 vậy ta xét tiếp đến cây bên phải. Nút con bên phải bằng Null, chứng tỏ rằng khoá 19 chưa có trên cây, ta thêm nút mới chứa khoá 19 và nút mới này là con bên phải của nút có khoá là 17



Chương trình con sau đây tiến hành việc thêm một khoá vào cây TKNP.

```
Proc InsertNode(x ,Root)
    If Root = Null Then
            Root←Tao(x,Null,Null)//Tạo 1 Node cho cây (hàm tạo đã có)
    Else    If x < Info(Root) Then
                    Call InsertNode(x,Left(Root)) //Chèn x vào cây con trái
            Else If x > Info(Root) Then
                    Call InsertNode(x,Right(Root))      //Chèn x vào cây con phải
Return
```

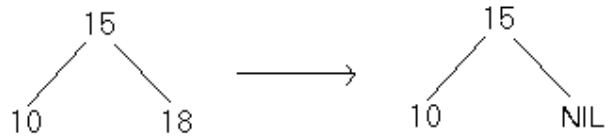
Cài đặt không đệ qui

```
Proc InserNode(x,Root)
    p := Root
    While p<>Null And Info(p)<>x Do
    {       q := p
            If Info(p)<x Then
```

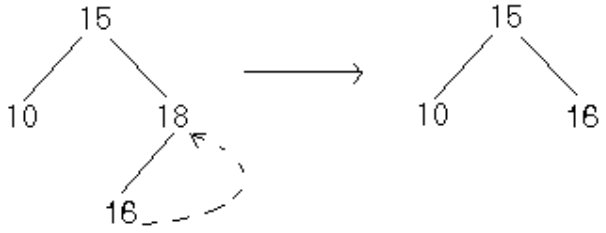
```
                p := Right(p)          //Tìm kiếm bên cây con phải
            Else
                p := Left(p)           //Tìm kiếm bên cây con trái
        }
        If p=Null Then                 //trường hợp không có x trong cây
        {       p := Tao(x, Null, Null)
                If Info(q) >x Then Left(q) := p
                Else Right(q) := p
        }
    Return
```

➢ **Xoá một nút có khoá cho trước trên cây tìm kiếm nhị phân**

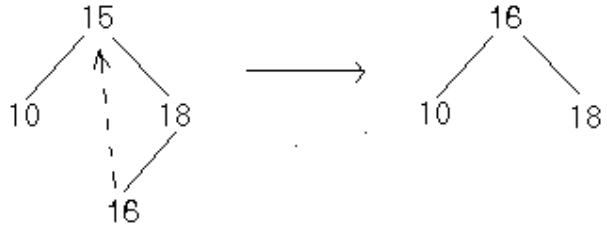
Giả sử ta muốn xoá một nút có khoá x, trước hết ta phải tìm kiếm nút chứa khoá x trên cây. Việc xoá một nút như vậy, tất nhiên, ta phải bảo đảm cấu trúc cây TKNP không bị phá vỡ. Ta có các trường hợp sau:



Xoá nút lá có khoá 18



Xoá nút có khoá 18 là nút trung gian có một nút con



Xoá nút có khoá 15 là nút trung gian có hai nút con

- Nếu không tìm thấy nút chứa khoá x thì giải thuật kết thúc.
- Nếu tìm gặp nút N có chứa khoá x, ta có ba trường hợp sau
  - ✓ Nếu N là lá ta thay nó bởi Null.
  - ✓ N chỉ có một nút con ta thay nó bởi nút con của nó.
  - ✓ N có hai nút con ta thay nó bởi nút lớn nhất trên cây con trái của nó (nút cực phải *của cây con trái) hoặc là nút bé nhất trên cây con phải của nó (nút cực trái của cây con phải). Trong giải thuật sau, ta thay x bởi khoá của nút cực trái của cây con bên* phải rồi ta xoá nút cực trái này. Việc xoá nút cực trái của cây con bên phải sẽ rơi vào một trong hai trường hợp trên.

Giải thuật xoá một nút có khoá nhỏ nhất

Hàm dưới đây trả về khoá của nút cực trái, đồng thời xoá nút này.

```
Func DeleteMin ( Root )
    If Left(Root) = Null Then
    {       DeleteMin := Info(Root)
            Root := Right(Root)
    }
    Else DeleteMin := DeleteMin(Left(Root))
 Return
```

Chương trình con xóa một nút có khoá cho trước trên cây TKNP

```
Proc DeleteNode( X,Root)
    If Root <> Null Then
            If x < Info(Root) Then
                    Call DeleteNode(x,Left(Root))
            Else    If x > Info(Root) Then
                            Call DeleteNode(x,Right(Root))
                    Else    If Left(Root) = Null And Right(Root) = Null Then
                                    Root := Null
                            Else    If Left(Root) = Null Then Root := Right(Root)
                                    Else    If Right(Root) = Null Then
                                                    Root := Left(Root)
                                            Else Info(Root) := DeleteMin(Right(Root)
Return
```

VI.CÂY TÌM KIẾM CƠ SỐ (RADIX SEARCH TREE – RST)

Mọi dữ liệu lưu trữ trong máy tính đều được số hoá, tức là đều được lưu trữ bằng các đơn vị Bit, Byte, Word v.v... Điều đó có nghĩa là một giá trị khoá bất kỳ, ta hoàn toàn có thể biết được nó được mã hoá bằng con số như thế nào. Và một điều chắc chắn là hai khoá khác nhau sẽ được lưu trữ bằng hai số khác nhau.

Đối với bài toán sắp xếp, ta không thể đưa việc sắp xếp một dãy khoá bất kỳ về việc sắp xếp trên một dãy khoá số là mã của các khoá. Bởi quan hệ thứ tự trên các con số đó có thể khác với thứ tự cần sắp của các khoá.

Nhưng đối với bài toán tìm kiếm thì khác, với một khoá tìm kiếm, Câu trả lời hoặc là "Không tìm thấy" hoặc là "Có tìm thấy và ở chỗ ..." nên ta hoàn toàn có thể thay các khoá bằng các mã số của nó mà không bị sai lầm, chỉ lưu ý một điều là: hai khoá khác nhau phải mã hoá thành hai số khác nhau mà thôi.

Nói như vậy có nghĩa là việc nghiên cứu những thuật toán tìm kiếm trên các dãy khoá số rất quan trọng, và ở đây ta sẽ tìm hiểu một trong số những phương pháp đó.

Cây tìm kiếm cơ số là một phương pháp khắc phục nhược điểm đó, nội dung của nó có thể tóm tắt như sau:

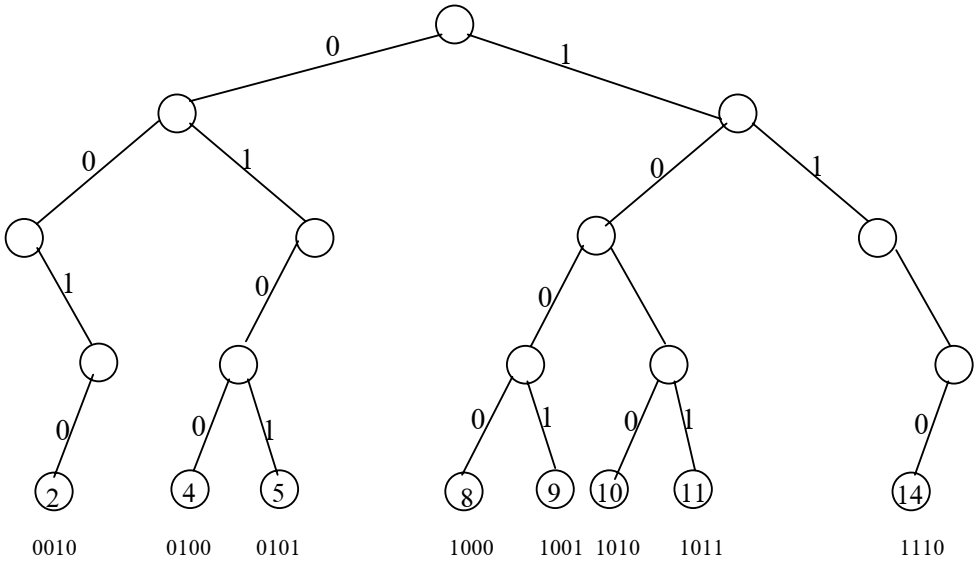
Trong cây tìm kiếm cơ số là một cây nhị phân, chỉ có nút lá chứa giá trị khoá, còn giá trị chứa trong các nút nhánh là vô nghĩa. Các nút lá của cây tìm kiếm cơ số đều nằm ở mức z + 1.

Đối với nút gốc của cây tìm kiếm cơ số, nó có tối đa hai nhánh con, mọi khoá chứa trong nút lá của nhánh con trái đều có bít cao nhất là 0, mọi khoá chứa trong nút lá của nhánh con phải đều có bít cao nhất là 1.

Đối với hai nhánh con của nút gốc, vấn đề tương tự với bít thứ z - 2, ví dụ với nhánh con trái của nút gốc, nó lại có tối đa hai nhánh con, mọi khoá chứa trong nút lá của nhánh con trái đều có bít thứ z - 2 là 0 (chúng bắt đầu bằng hai bít 00), mọi khoá chứa trong nút lá của nhánh con phải đều có bít thứ z - 2 là 1 (chúng bắt đầu bằng hai bít 01)...

Tổng quát với nút ở mức d, nó có tối đa hai nhánh con, mọi nút lá của nhánh con trái chứa khoá có bít z - d là 0, mọi nút lá của nhánh con phải chứa khoá có bít thứ z - d là 1.

Ví dụ: cho dãy khoá 9, 4, 11, 2, 8, 14, 10, 5 được biểu diễn bằng cây tìm kiếm cơ số sau:



Cây tìm kiếm cơ số được khởi tạo gồm có một nút gốc, và **nút gốc tồn tại trong suốt quá trình sử dụng**: nó không bao giờ bị xoá đi cả.

➢ **Tìm kiếm trên cây tìm kiếm cơ số**

Để tìm một giá trị x trên cây tìm kiếm cơ số, ban đầu con trỏ p ở nút gốc và đồng thời duyệt dãy bit của x từ bit z-1 đến bit 0. Trong quá trình duyệt, nếu gặp bit 0 thì p trỏ qua nút con trái, gặp bit 1 thì p trỏ qua nút con phải. Trong quá trình duyệt có thể có 2 trường hợp xãy ra:

- Bit của x còn nhưng con trỏ p trên cây ra Null thì quá trình tìm kiếm thất bại
- Duyệt hết các bit của x và p đang đứng ở nút lá, thì quá trình tìm kiếm thành công vì giá trị của nút lá bằng đúng khoá x

Hàm tìm kiếm trên cây tìm kiếm cơ số, nó trả về nút lá chứa khoá tìm kiếm X nếu tìm thấy, trả về nil nếu không tìm thấy, z là độ dài dãy bít biểu diễn một khoá

```
func RSTSearch(X: TKey): PNode;
    b := z; p := Root;      //Bắt đầu với nút gốc, đối với RST thì gốc luôn có sẵn
    do
    {       b := b - 1;     //Xét bít b của X
            if <Bít b của X là 0> then p := p^.Left         //Gặp 0 rẽ trái
                    else p := p^.Right;                     //Gặp 1 rẽ phải
```

```
        } while (p <> nil) and (b <> 0);
        RSTSearch := p;
    Return
```

➢ **Thao tác chèn một giá trị X vào RST**

Thuật toán thực hiện như sau: Đầu tiên, ta đứng ở gốc và duyệt dãy bít của X từ trái qua phải (từ bít z - 1 về bít 0), cứ gặp 0 thì rẽ trái, gặp 1 thì rẽ phải. Nếu quá trình rẽ theo một liên kết nil (đi tới nút rỗng) thì lập tức tạo ra một nút mới, và nối vào theo liên kết đó để có đường đi tiếp. Sau khi duyệt hết dãy bít của X, ta sẽ dừng lại ở một nút lá của RST, và công việc cuối cùng là đặt giá trị X vào nút lá đó.

```
proc RSTInsert(X: TKey);
    b := z; p := Root;              //Bắt đầu từ nút gốc, đối với RST thì gốc luôn ≠ nil
    do {
            b := b - 1;             //Xét bít b của X
            q := p;                 //Khi p chạy xuống nút con thì q^ luôn giữ vai trò là nút
                                    cha của p^
            if <Bít b của X là 0> then p := p^.Left         //Gặp 0 rẽ trái}
            else p := p^.Right;     //Gặp 1 rẽ phải
            if p = nil then         //Không đi được thì đặt thêm nút để đi tiếp
                    {New(p);                //Tạo ra một nút mới và đem p trỏ tới nút đó
                    p^.Left := nil; p^.Right := nil;
                    if <Bít b của X là 0> then q^.Left := p         //Nối p^ vào bên trái q^
                    else q^.Right := p;                             //Nối p^ vào bên phải q^
                    }
    } while  b <> 0;
    p^.Info := X;                           //p^ là nút lá để đặt X vào
Return
```

➢ **Thao tác xoá một nút có giá trị X khỏi cây RST**

Với cây tìm kiếm cơ số, việc xoá một giá trị khoá không phải chỉ là xoá riêng một nút lá mà còn phải xoá toàn bộ nhánh độc đạo đi tới nút đó để tránh lãng phí bộ nhớ.

Ta lặp lại quá trình tìm kiếm giá trị khoá X, quá trình này sẽ đi từ gốc xuống lá, tại mỗi bước đi, mỗi khi gặp một nút ngã ba (nút có cả con trái và con phải - nút cấp hai), ta ghi nhận lại ngã ba đó và hướng rẽ. Kết thúc quá trình tìm kiếm ta giữ lại được ngã ba đi qua cuối cùng, từ nút đó tới nút lá chứa X là con đường độc đạo (không có chỗ rẽ), ta tiến hành dỡ bỏ tất cả các nút trên đoạn đường độc đạo khỏi cây tìm kiếm cơ số. Để không bị gặp lỗi khi cây suy biến (không có nút cấp 2) ta coi gốc cũng là nút ngã ba.

```
proc RSTDelete(X: TKey);
    //Trước hết, tìm kiếm giá trị X xem nó nằm ở nút nào
    b := z; p := Root;
    do{
            b := b - 1;
            q := p;                 //Mỗi lần p chuyển sang nút con, ta luôn đảm bảo cho q^
                                    là nút cha của p^
            if <Bít b của X là 0> then p := p^.Left
            else p := p^.Right;
            if (b = z - 1) or (q^.Left ≠ nil) and (q^.Right ≠ nil) then     //q^ là nút ngã ba
            {
                    TurnNode := q; Child := p;   //Ghi nhận lại q^ và hướng rẽ
            }
    } while (p <> nil) and (b <> 0);
    if p = nil then Exit;     //X không tồn tại trong cây thì không xoá được
    //Trước hết, cắt nhánh độc đạo ra khỏi cây
```

```
    if TurnNode^.Left = Child then TurnNode^.Left := nil else TurnNode^.Right := nil
    p := Child;     //Chuyển sang đoạn đường độc đạo, bắt đầu xoá
    do {
            q := p;
                    //Lưu ý rằng p^ chỉ có tối đa một nhánh con mà thôi, cho p trỏ sang
                    nhánh con duy nhất nếu có
            if p^.Left ≠ nil then p := p^.Left
            else p := p^.Right;
            Dispose(q);     //Giải phóng bộ nhớ cho nút q^
    } while p <> nil;
 Return
```

Hình dáng của cây tìm kiếm cơ số không phụ thuộc vào thứ tự chèn các khoá vào mà chỉ phụ thuộc vào giá trị của các khoá chứa trong cây.

Đối với cây tìm kiếm cơ số, độ phức tạp tính toán cho các thao tác tìm kiếm, chèn, xoá trong trường hợp xấu nhất cũng như trung bình đều là O(z). Do không phải so sánh giá trị khoá dọc đường đi, nó nhanh hơn cây tìm kiếm số học nếu như gặp các khoá cấu trúc lớn. Tốc độ như vậy có thể nói là tốt, nhưng vấn đề bộ nhớ khiến ta phải xem xét: Giá trị chứa trong các nút nhánh của cây tìm kiếm cơ số là vô nghĩa dẫn tới sự lãng phí bộ nhớ.

Một giải pháp cho vấn đề này là: Duy trì hai dạng nút trên cây tìm kiếm cơ số: Dạng nút nhánh chỉ chứa các liên kết trái, phải và dạng nút lá chỉ chứa giá trị khoá. Cài đặt cây này trên một số ngôn ngữ định kiểu quá mạnh đôi khi rất khó.

Giải pháp thứ hai là đặc tả một cây tương tự như RST, nhưng sửa đổi một chút: nếu có nút lá chứa giá trị X được nối với cây bằng một nhánh độc đạo thì cắt bỏ nhánh độc đạo đó, và thay vào chỗ nhánh này chỉ một nút chứa giá trị X. Như vậy các giá trị khoá vẫn chỉ chứa trong các nút lá nhưng các nút lá giờ đây không chỉ nằm trên mức z + 1 mà còn nằm trên những mức khác nữa. Phương pháp này không những tiết kiệm bộ nhớ hơn mà còn làm cho quá trình tìm kiếm nhanh hơn. Giá phải trả cho phương pháp này là thao tác chèn, xoá khá phức tạp. Tên của cấu trúc dữ liệu này là Trie (Trie chứ không phải Tree) tìm kiếm cơ số.

Phép tìm kiếm bằng cơ số không nhất thiết phải chọn hệ cơ số 2. Ta có thể chọn hệ cơ số lớn hơn để có tốc độ nhanh hơn (kèm theo sự tốn kém bộ nhớ), chỉ lưu ý là cây tìm kiếm cơ số trong trường hợp này không còn là cây nhị phân mà là cây R_phân với R là hệ cơ số được chọn.

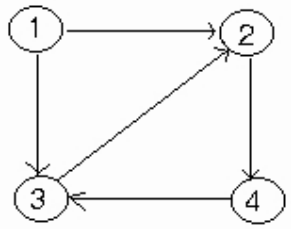
Trong các phương pháp tìm kiếm bằng cơ số, thực ra còn một phương pháp tinh tuý và thông minh nhất, nó có cấu trúc gần giống như cây nhưng không có nút dư thừa, và quá trình duyệt bít của khoá tìm kiếm không phải từ trái qua phải mà theo thứ tự của các bít kiểm soát lưu tại mỗi nút đi qua. Phương pháp đó có tên gọi là Practical Algorithm To Retrieve Information Coded In Alphanumeric (PATRICIA) do Morrison đề xuất. Tuy nhiên, việc cài đặt phương pháp này khá phức tạp (đặc biệt là thao tác xoá giá trị khoá), ta có thể tham khảo nội dung của nó trong các tài liệu khác.

CHƯƠNG 6

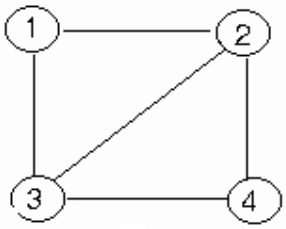
BIỂU DIỄN ĐỒ THỊ

I. MỘT SỐ KHÁI NIỆM

Một đồ thị G bao gồm một tập hợp V các đỉnh và một tập hợp E các cung, ký hiệu G=(V,E). Các đỉnh còn được gọi là nút (node) hay điểm (point). Các cung nối giữa hai đỉnh, hai đỉnh này có thể trùng nhau. Hai đỉnh có cung nối nhau gọi là hai đỉnh kề (adjacency). Một cung nối giữa hai đỉnh v, w có thể coi như là một cặp điểm (v,w). Nếu cặp này có thứ tự thì ta có cung có thứ tự, ngược lại thì cung không có thứ tự. Nếu các cung trong đồ thị G có thứ tự thì G gọi là đồ thị có hướng (directed graph). Nếu các cung trong đồ thị G không có thứ tự thì đồ thị G là đồ thị vô hướng (undirected graph). Trong các phần sau này ta dùng từ đồ thị (graph) để nói đến đồ thị nói chung, khi nào cần phân biệt rõ ta sẽ dùng đồ thị có hướng, đồ thị vô hướng. Hình V.1a cho ta một ví dụ về đồ thị có hướng, hình V.1b cho ví dụ về đồ thị vô hướng. Trong các đồ thị này thì các vòng tròn được đánh số biểu diễn các đỉnh, còn các cung được biểu diễn bằng các đoạn thẳng có hướng (Hình 1a) hoặc không có hướng (Hình 1b).



Hình 1a



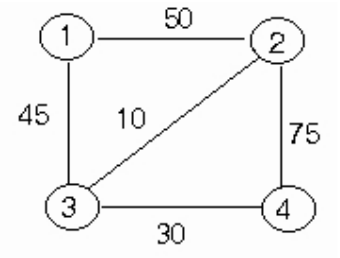
Hình 1b

Thông thường trong một đồ thị, các đỉnh biểu diễn cho các đối tượng còn các cung biểu diễn mối quan hệ (relationship) giữa các đối tượng đó. Chẳng hạn các đỉnh có thể biểu diễn cho các thành phố còn các cung biểu diễn cho đường giao thông nối giữa hai thành phố.

Một đường đi (path) trên đồ thị là một dãy tuần tự các đỉnh v1, v2,..., vn sao cho (vi,vi+1) là một cung trên đồ thị (i=1,...,n-1). Đường đi này là đường đi từ v1 đến vn và đi qua các đỉnh v2,...,vn-1. Đỉnh v1 còn gọi là đỉnh đầu, vn gọi là đỉnh cuối. Độ dài của đường đi này bằng (n-1). Trường hợp đặc biệt dãy chỉ có một đỉnh v thì ta coi đó là đường đi từ v đến chính nó có độ dài bằng không. Ví dụ dãy 1,2,4 trong đồ thị V.1a là một đường đi từ đỉnh 1 đến đỉnh 4, đường đi này có độ dài là hai.

Đường đi gọi là đơn (simple) nếu mọi đỉnh trên đường đi đều khác nhau, ngoại trừ đỉnh đầu và đỉnh cuối có thể trùng nhau. Một đường đi có đỉnh đầu và đỉnh cuối trùng nhau gọi là một chu trình (cycle). Một chu trình đơn là một đường đi đơn có đỉnh đầu và đỉnh cuối trùng nhau và có độ dài ít nhất là 1. Ví dụ trong hình V.1a thì 3, 2, 4, 3 tạo thành một chu trình có độ dài 3. Trong hình V.1b thì 1,3,4,2,1 là một chu trình có độ dài 4.

Trong nhiều ứng dụng ta thường kết hợp các giá trị (value) hay nhãn (label) với các đỉnh và/hoặc các cạnh, lúc này ta nói đồ thị có nhãn. Nhãn kết hợp với các đỉnh và/hoặc cạnh có thể biểu diễn tên, giá, khoảng cách,... Nói chung nhãn có thể có kiểu tuỳ ý. Hình 2 cho ta ví dụ về một đồ thị có nhãn. Ở đây nhãn là các giá trị số nguyên biểu diễn cho giá cước vận chuyển một tấn hàng giữa các thành phố 1, 2, 3, 4 chẳng hạn.



Hình 2

Đồ thị con của một đồ thị G=(V,E) là một đồ thị G'=(V',E') trong đó:

- V’∈V và
- E’ gồm tất cả các cạnh (v,w) ∈ E sao cho v,w ∈ V’.

Thông thường trong các giải thuật trên đồ thị, ta thường phải thực hiện một thao tác nào đó với tất cả các đỉnh kề của một đỉnh, tức là một đoạn giải thuật có dạng sau:

For (mỗi đỉnh w kề với v)
{ thao tác nào đó trên w }

Để cài đặt các giải thuật như vậy ta cần bổ sung thêm khái niệm về chỉ số của các đỉnh kề với v. Hơn nữa ta cần định nghĩa thêm các phép toán sau đây:

- FIRST(v) trả về chỉ số của đỉnh đầu tiên kề với v. Nếu không có đỉnh nào kề với v thì null được trả về. Giá trị null được chọn tuỳ theo cấu trúc dữ liệu cài đặt đồ thị.
- NEXT(v,i) trả về chỉ số của đỉnh nằm sau đỉnh có chỉ số i và kề với v. Nếu không có đỉnh nào kề với v theo sau đỉnh có chỉ số i thì null được trả về.
- VERTEX(i) trả về đỉnh có chỉ số i. Có thể xem VERTEX(v,i) như là một hàm để định vị đỉnh thứ i để thức hiện một thao tác nào đó trên đỉnh này.

II. CÁC CÁCH BIỂU DIỄN ĐỒ THỊ

Một số cấu trúc dữ liệu có thể dùng để biểu diễn đồ thị. Việc chọn cấu trúc dữ liệu nào là tuỳ thuộc vào các phép toán trên các cung và đỉnh của đồ thị. Hai cấu trúc thường gặp là biểu diễn đồ thị bằng ma trận kề (adjacency matrix) và biểu diễn đồ thị bằng danh sách các đỉnh kề (adjacency list).

II.1. Biểu diễn đồ thị bằng ma trận kề

Ta dùng một mảng hai chiều, chẳng hạn mảng A, kiểu boolean để biểu diễn các đỉnh kề. Nếu đồ thị có n đỉnh thì ta dùng mảng A có kích thước nxn. Giả sử các đỉnh được đánh số 1..n thì A[i,j] = true, nếu có đỉnh nối giữa đỉnh thứ i và đỉnh thứ j, ngược lại thì A[i,j] = false. Rõ ràng, nếu G là đồ thị vô hướng thì ma trận kề sẽ là ma trận đối xứng. Chẳng hạn đồ thị ở Hình1b có biểu diễn ma trận kề như sau:

i \ j	0	1	2	3
0	true	true	true	false
1	true	true	true	true
2	true	true	true	true
3	false	true	true	true

Ta cũng có thể biểu diễn true là 1 còn false là 0. Với cách biểu diễn này thì đồ thị Hình1a có biểu diễn ma trận kề như sau:

i \ j	0	1	2	3
0	1	1	1	0
1	0	1	0	1
2	0	1	1	0
3	0	0	0	1

Với cách biểu diễn đồ thị bằng ma trận kề như trên chúng ta có thể định nghĩa chỉ số của đỉnh là số nguyên chỉ đỉnh đó (theo cách đánh số các đỉnh) và ta cài đặt các phép toán FIRST, NEXT và VERTEX như sau:

```
const null=0;
int A[n,n];          //mảng biểu diễn ma trận kề

int FIRST(int v)   //trả ra chỉ số  [1..n] của đỉnh đầu tiên kề với v ∈ 1..n
{  int i;
   for (i=1; i<=n; i++)
   if (a[v-1,i-1] == 1)
            return (i);       //trả ra chỉ số đỉnh của đồ thị
   return (null);
}

int NEXT(int v; int i)       //trả ra đỉnh    [1..n] sau đỉnh i mà kề với v; i, v ∈ 1..n
{  int j;
   for (j=i+1; j<=n; j++)
            if (a[v-1,j-1] == 1)
                     return(j)
   return(null);
}
```

Còn VERTEX(i) chỉ đơn giản là trả ra chính i. Vòng lặp trên các đỉnh kề với v có thể cài đặt như sau

```
Int VERTEX(i)
{  i=FIRST(v);
   while (i<>null)
   {         w = VERTEX(i);
```

```
//thao tâc trín w
i =NEXT(v,i);          }}
```

Trên đồ thị có nhãn thì ma trận kề có thể dùng để lưu trữ nhãn của các cung chẳng hạn cung giữa i và j có nhãn a thì A[i,j]=a. Ví dụ ma trận kề của đồ thị hình 6.2 là:

i \ j	1	1	3	4
1		50	45	
2	50		10	75
3	45	10		30
4		75	30	

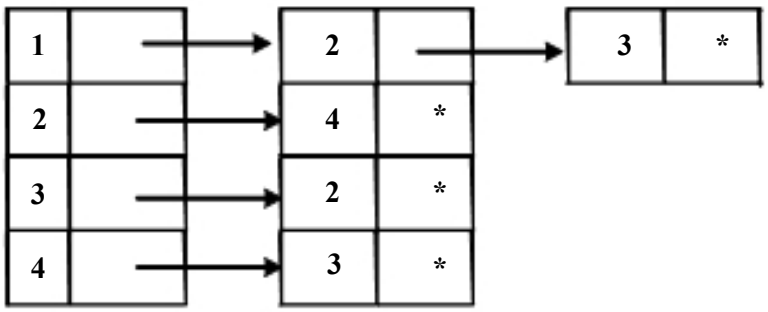
Ở đây các cặp đỉnh không có cạnh nối thì ta để trống, nhưng trong các ứng dụng ta có thể phải gán cho nó một giá trị đặc biệt nào đó để phân biệt với các giá trị có nghĩa khác. Chẳng hạn như trong bài toán tìm đường đi ngắn nhất, các giá trị số nguyên biểu diễn cho khoảng cách giữa hai thành phố thì các cặp thành phố không có cạnh nối ta gán cho nó khoảng cách bằng µ, còn khoảng cách từ một đỉnh đến chính nó là 0.

Cách biểu diễn đồ thị bằng ma trận kề cho phép kiểm tra một cách trực tiếp hai đỉnh nào đó có kề nhau không. Nhưng nó phải mất thời gian duyệt qua toàn bộ mảng để xác định tất cả các cạnh trên đồ thị. Thời gian này độc lập với số cạnh và số đỉnh của đồ thị. Ngay cả số cạnh của đồ thị rất nhỏ chúng ta cũng phải cần một mảng nxn phần tử để lưu trữ. Do vậy, nếu ta cần làm việc thường xuyên với các cạnh của đồ thị thì ta có thể phải dùng cách biểu diễn khác cho thích hợp hơn.

II.2. Biểu diễn đồ thị bằng danh sách các đỉnh kề:

Trong cách biểu diễn này, ta sẽ lưu trữ các đỉnh kề với một đỉnh i trong một danh sách liên kết theo một thứ tự nào đó. Như vậy ta cần một mảng HEAD một chiều có n phần tử để biểu diễn cho đồ thị có n đỉnh. HEAD[i] là con trỏ trỏ tới danh sách các đỉnh kề với đỉnh i. Ví dụ đồ thị Hình 1a có biểu diễn như sau:

Mảng HEAD



III. CÁC PHÉP DUYỆT ĐỒ THỊ (TRAVERSALS OF GRAPH)

Trong khi giải nhiều bài toán được mô hình hoá bằng đồ thị, ta cần đi qua các đỉnh và các cung của đồ thị một cách có hệ thống. Việc đi qua các đỉnh của đồ thị một cách có hệ thống như vậy gọi là duyệt đồ thị. Có hai phép duyệt đồ thị phổ biến đó là duyệt theo chiều sâu, tương tự như duyệt tiền tự một cây, và duyệt theo chiều rộng, tương tự như phép duyệt cây theo mức.

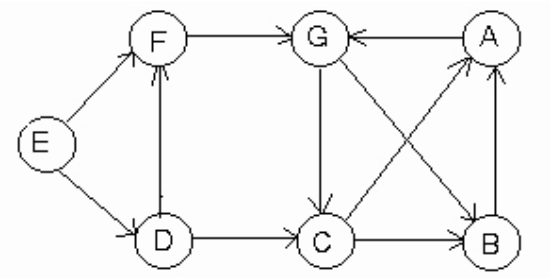
III.1. Duyệt theo chiều sâu (depth-first search)

Giả sử ta có đồ thị G=(V,E) với các đỉnh ban đầu được đánh dấu là chưa duyệt (unvisited). Từ một đỉnh v nào đó ta bắt đầu duyệt như sau: đánh dấu v đã duyệt, với mỗi đỉnh w chưa duyệt kề với v, ta thực hiện đệ qui quá trình trên cho w. Sở dĩ cách duyệt này có tên là duyệt theo chiều sâu vì nó sẽ duyệt theo một hướng nào đó sâu nhất có thể được. Giải thuật duyệt theo chiều sâu một đồ thị có thể được trình bày như sau, trong đó ta dùng một mảng mark có n phần tử để đánh dấu các đỉnh của đồ thị là đã duyệt hay chưa.

```
//đánh dấu chưa duyệt tất cả các đỉnh
for (v =1; v <=n; v++) mark[v-1]=unvisited; //duyệt theo chiều sâu từ đỉnh đánh số 1
for (v = 1; v<=n; v++)
if (mark[v-1] == unvisited)
        dfs(v);                 //duyệt theo chiều sâu đỉnh v
```

Thủ tục **dfs** ở trong giải thuật ở trên có thể được viết dạng đệ qui như sau:

```
void dfs(vertex v)          // v ∈ [1..n]
{   vertex w;
    mark[v-1]=visited;
    for (mỗi đỉnh w là đỉnh kề với v)
        if (mark[w-1] == unvisited)
            dfs(w);
}
```

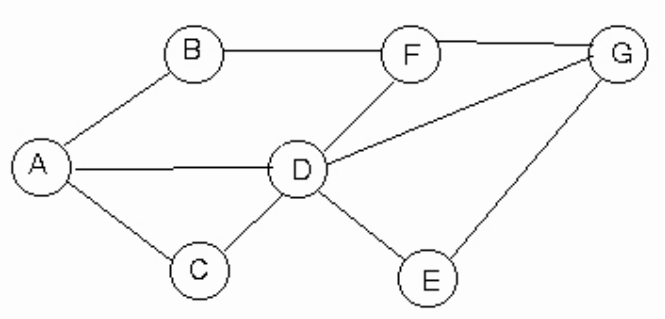


Hình 6.3

Ví dụ: Duyệt theo chiều sâu đồ thị trong hình 6.3. Giả sử ta bắt đầu duyệt từ đỉnh A, tức là dfs(A). Giải thuật sẽ đánh dấu là A đã được duyệt, rồi chọn đỉnh đầu tiên trong danh sách các đỉnh kề với A, đó là G. Tiếp tục duyệt đỉnh G, G có hai đỉnh kề với nó là B và C, theo thứ tự đó thì đỉnh kế tiếp được duyệt là đỉnh B. B có một đỉnh kề đó là A, nhưng A đã được duyệt nên phép duyệt dfs(B) đã hoàn tất. Bây giờ giải thuật sẽ tiếp tục với đỉnh kề với G mà còn chưa duyệt là C. C không có đỉnh kề nên phép duyệt dfs(C) kết thúc vậy dfs(A) cũng kết thúc. Còn lại 3 đỉnh chưa được duyệt là D,E,F và theo thứ tự đó thì D được duyệt, kế đến là F. Phép duyệt

dfs(D) kết thúc và còn một đỉnh E chưa được duyệt. Tiếp tục duyệt E và kết thúc. Nếu ta in các đỉnh của đồ thị trên theo thứ tự được duyệt ta sẽ có danh sách sau: AGBCDFE.

Ví dụ duyệt theo chiều sâu đồ thị hình 6.4 bắt đầu từ đỉnh A: Duyệt A, A có các đỉnh kề là B,C,D; theo thứ tự đó thì B được duyệt. B có 1 đỉnh kề chưa duyệt là F, nên F được duyệt. F có các đỉnh kề chưa duyệt là D,G; theo thứ tự đó thì ta duyệt D. D có các đỉnh kề chưa duyệt là C,E,G; theo thứ tự đó thì C được duyệt. Các đỉnh kề với C đều đã được duyệt nên giải thuật được tiếp tục duyệt E. E có một đỉnh kề chưa duyệt là G, vậy ta duyệt G. Lúc này tất cả các nút đều đã được duyệt nên đồ thị đã được duyệt xong. Vậy thứ tự các đỉnh được duyệt là ABFDCEG.



Hình 6.4

III.2. Duyệt theo chiều rộng (breadth-first search)

Giả sử ta có đồ thị G với các đỉnh ban đầu được đánh dấu là chưa duyệt (unvisited). Từ một đỉnh v nào đó ta bắt đầu duyệt như sau: đánh dấu v đã được duyệt, kế đến là duyệt tất cả các đỉnh kề với v. Khi ta duyệt một đỉnh v rồi đến đỉnh w thì các đỉnh kề của v được duyệt trước các đỉnh kề của w, vì vậy ta dùng một hàng để lưu trữ các nút theo thứ tự được duyệt để có thể duyệt các đỉnh kề với chúng. Ta cũng dùng mảng một chiều mark để đánh dấu một nút là đã duyệt hay chưa, tương tự như duyệt theo chiều sâu. Giải thuật duyệt theo chiều rộng được viết như sau:

```
//đánh dấu chưa duyệt tất cả các đỉnh
for (v = 1; v<= n; v++) mark[v-1] = unvisited;          //n là số đỉnh của đồ thị
//duyệt theo chiều rộng từ đỉnh đánh số 1
 for (v = 1; v<=n; v++)
    if (mark[v-1] == unvisited)
            bfs(v);
```

Thủ tục **bfs** được viết như sau:

```
void bfs(vertex v)          // v ∈ [1..n]
   {
            QUEUE of vertex Q;
            vertex x,y;
            mark[v-1] = visited;
            ENQUEUE(v,Q);
            while !(EMPTY_QUEUE(Q))
            {
                    x = FRONT(Q);
                    DEQUEUE(Q);
                    for (mỗi đỉnh y kề với x)
                            if (mark[y-1] == unvisited)
                            {
```

```
                    mark[y-1] = visited; {duyệt y} ENQUEUE(y,Q);
                }
            }
        }
```

Ví dụ duyệt theo chiều rộng đồ thị hình 6.3. Giả sử bắt đầu duyệt từ A. A chỉ có một đỉnh kề G, nên ta duyệt G. Kế đến duyệt tất cả các đỉnh kề với G; đó là B,C. Sau đó duyệt tất cả các đỉnh kề với B, C theo thứ tự đó. Các đỉnh kề với B, C đều đã được duyệt, nên ta tiếp tục duyệt các đỉnh chưa được duyệt. Các đỉnh chưa được duyệt là D, E, F. Duyệt D, kế đến là F và cuối cùng là E. Vậy thứ tự các đỉnh được duyệt là: AGBCDFE.

Ví dụ duyệt theo chiều rộng đồ thị hình 6.4. Giả sử bắt đầu duyệt từ A. Duyệt A, kế đến duyệt tất cả các đỉnh kề với A; đó là B, C, D theo thứ tự đó. Kế tiếp là duyệt các đỉnh kề của B, C, D theo thứ tự đó. Vậy các nút được duyệt tiếp theo là F, E,G. Có thể minh hoạ hoạt động của hàng trong phép duyệt trên như sau:

Duyệt A nghĩa là đánh dấu visited và đưa nó vào hàng:

A						

Kế đến duyệt tất cả các đỉnh kề với đỉnh đầu hàng mà chưa được duyệt; tức là ta loại A khỏi hàng, duyệt B, C, D và đưa chúng vào hàng, bây giờ hàng chứa các đỉnh B, C, D.

	B	**C**	**D**			

Kế đến B được lấy ra khỏi hàng và các đỉnh kề với B mà chưa được duyệt, đó là F, sẽ được duyệt, và F được đưa vào hàng đợi.

		C	**D**	**F**		

Kế đến thì C được lấy ra khỏi hàng và các đỉnh kề với C mà chưa được duyệt sẽ được duyệt. Không có đỉnh nào như vậy, nên bước này không có thêm đỉnh nào được duyệt.

			D	**F**		

Kế đến thì D được lấy ra khỏi hàng và duyệt các đỉnh kề chưa duyệt của D, tức là E, G được duyệt. E, G được đưa vào hàng đợi.

				F	**E**	**G**

Tiếp tục, F được lấy ra khỏi hàng. Không có đỉnh nào kề với F mà chưa được duyệt. Vậy không duyệt thêm đỉnh nào.

					E	**G**

Tương tự như F, E rồi đến G được lấy ra khỏi hàng. Hàng trở thành rỗng và giải thuật kết thúc.

IV. MỘT SỐ BÀI TOÁN TRÊN ĐỒ THỊ

Phần này sẽ giới thiệu với các bạn một số bài toán quan trọng trên đồ thị, như bài toán tìm đường đi ngắn nhất, bài toán tìm bao đóng chuyển tiếp, cây bao trùm tối thiểu... Các bài toán này cùng với các giải thuật của nó đã được trình bày chi tiết trong giáo trình về Qui Hoạch

Động, vì thế ở đây ta không đi vào quá chi tiết các giải thuật này. Phần này chỉ xem như là phần nêu các ứng dụng cùng với giải thuật để giải quyết các bài toán đó nhằm giúp bạn đọc có thể vận dụng được các giải thuật vào việc cài đặt để giải các bài toán nêu trên.

IV.1. Bài toán tìm đường đi ngắn nhất từ một đỉnh của đồ thị (the single source shorted path problem)

Cho đồ thị G với tập các đỉnh V và tập các cạnh E (đồ thị có hướng hoặc vô hướng). Mỗi cạnh của đồ thị có một nhãn, đó là một giá trị không âm, nhãn này còn gọi là giá (cost) của cạnh. Cho trước một đỉnh v xác định, gọi là đỉnh nguồn. Vấn đề là tìm đường đi ngắn nhất từ v đến các đỉnh còn lại của G; tức là các đường đi từ v đến các đỉnh còn lại với tổng các giá (cost) của các cạnh trên đường đi là nhỏ nhất. Chú ý rằng nếu đồ thị có hướng thì đường đi này là đường đi có hướng.

Ta có thể giải bài toán này bằng cách xác định một tập hợp S chứa các đỉnh mà khoảng cách ngắn nhất từ nó đến đỉnh nguồn v đã biết. Khởi đầu S={v}, sau đó tại mỗi bước ta sẽ thêm vào S các đỉnh mà khoảng cách từ nó đến v là ngắn nhất. Với giả thiết mỗi cung có một giá không âm thì ta luôn luôn tìm được một đường đi ngắn nhất như vậy mà chỉ đi qua các đỉnh đã tồn tại trong S. Để chi tiết hoá giải thuật, giả sử G có n đỉnh và nhãn trên mỗi cung được lưu trong mảng hai chiều C, tức là C[i,j] là giá (có thể xem như độ dài) của cung (i,j), nếu i và j không nối nhau thì C[i,j]=∞. Ta dùng mảng 1 chiều D có n phần tử để lưu độ dài của đường đi ngắn nhất từ mỗi đỉnh của đồ thị đến v. Khởi đầu khoảng cách này chính là độ dài cạnh (v,i), tức là D[i]=C[v,i]. Tại mỗi bước của giải thuật thì D[i] sẽ được cập nhật lại để lưu độ dài đường đi ngắn nhất từ đỉnh v tới đỉnh i, đường đi này chỉ đi qua các đỉnh đã có trong S.

Để cài đặt giải thuật dễ dàng, ta giả sử các đỉnh của đồ thị được đánh số từ 1 đến n, tức là V={1,..,n} và đỉnh nguồn là 1. Dưới dây là giải thuật Dijkstra để giải bài toán trên.

```
void Dijkstra()
{
   S = [1];                 //Tập hợp S chỉ chứa một đỉnh nguồn for (i =2; i<=n; i++)
   D[i-1] = C[0,i-1];       //khởi đầu các giá trị cho D
   for (i=1; i<n; i++)
   {
          Lấy đỉnh w trong V-S sao cho D[w-1] nhỏ nhất;
          Thêm w vào S;
          for (mỗi đỉnh u thuộc V-S)
                D[u-1] = min(D[u-1], D[w-1] + C[w-1,u-1]);
   }
}
```

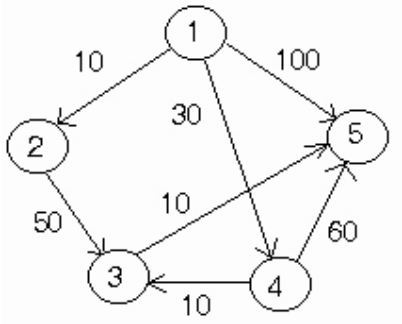
Nếu muốn lưu trữ lại các đỉnh trên đường đi ngắn nhất để có thể xây dựng lại đường đi này từ đỉnh nguồn đến các đỉnh khác, ta dùng một mảng P. Mảng này sẽ lưu P[u]=w với u là đỉnh "trước" đỉnh w trong đường đi. Lúc khởi đầu P[u]=1 với mọi u.

Giải thuật Dijkstra được viết lại như sau:

```
void Dijkstra()
{
   S =[1];                          //S chỉ chứa một đỉnh nguồn
   for(i=2; i<=n; i++)
```

```
    {
        P[i-1] =1;          //khởi tạo giá trị cho P
        D[i-1] =C[0,i-1];   //khởi đầu các giá trị cho D
    }
    for (i=1; i<n; i++)
    {
        Lấy đỉnh w trong V-S sao cho D[w-1] nhỏ nhất;
        Thêm w vào S;
        for (mỗi đỉnh u thuộc V-S)
            if  (D[w-1] + C[w-1,u-1] < D[u-1])
            {
                D[u-1] =D[w-1] + C[w-1,u-1];
                P[u-1] =w;
            }
    }
}
```

Ví dụ: áp dụng giải thuật Dijkstra cho đồ thị hình 6.5



Hình 6.5

Kết quả khi áp dụng giải thuật

Lần lặp	S	W	D[2]	D[3]	D[4]	D[5]
Khởi đầu	{1}	-	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	40	30	90
3	{1,2,3,4}	3	10	40	30	50
4	{1,2,3,4,5}	5	10	40	30	50

Mảng P có giá trị như sau:

P	**1**	**2**	**3**	**4**	**5**		
		1	4	1	3		

Từ kết quả trên ta có thể suy ra rằng đường đi ngắn nhất từ đỉnh 1 đến đỉnh 3 là 1 → 4 → 3 có độ dài là 40. đường đi ngắn nhất từ 1 đến 5 là 1 → 4 → 3→ 5 có độ dài 50.

IV.2. Tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh

Giả sử đồ thị G có n đỉnh được đánh số từ 1 đến n. Khoảng cách hay giá giữa các cặp đỉnh được cho trong mảng C[i,j]. Nếu hai đỉnh i,j không được nối thì C[i,j]= ¥. Giải thuật Floyd xác định đường đi ngắn nhất giữa hai cặp đỉnh bất kỳ bằng cách lặp k lần, ở lần lặp thứ k sẽ xác định khoảng cách ngắn nhất giữa hai đỉnh i,j theo công thức: Ak[i,j]=min(Ak-1[i,j], Ak1[i,k]+Ak-1[k,j]). Ta cũng dùng mảng P để lưu các đỉnh trên đường đi.

```
float A[n,n], C[n,n];
int P[n,n];

void Floyd()
{
    int i,j,k;
    for (i=1; i<=n; i++)
            for (j=1; j<=n; j++)
            {
                    A[i-1,j-1] = C[i-1,j-1]; P[i-1,j-1]=0;
            }
    for (i=1; i<=n; i++)
            A[i-1,i-1]=0;
    for (k=1; k<=n; k++)
            for (i=1; i<=n; i++)
                    for (j=1; j<=n; j++)
                            if (A[i-1,k-1] + A[k-1,j-1] < A[i-1,j-1)
                            {
                                    A[i-1,j-1] = A[i-1,k-1] + A[k-1,j-1]; P[i-1,j-1] = k;
                            }
}
```

TÀI LIỆU THAM KHẢO

[1] Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, NXB Khoa học và kỹ thuật, 1996
[2] Nguyễn trung Trực, *Cấu trúc dữ liệu*, ĐHBK TpHCM, 1990.
[3] N. Wirth, *Algorithms + Data structures = Programs*, Prentice Hall, 1976.
[4] Aho, A. V. , J. E. Hopcroft, J. D. Ullman, *Data Structure and Algorihtms*, 1983
[5] Mark Allen Weiss, *Data Structures and Algorithms Analysis In C*, The Benjamin / Cummings Publishing Company, Inc. 1993.
[6] R.L. Kruse, C.L. Tondo, B.P. Leung, *Data Structures and Program Design in* C, Prentice Hall, 1997.