

ThS. Nguyễn Thị Hương

GIÁO TRÌNH **Cấu trúc dữ liệu & giải thuật**



NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

**BỘ GIÁO DỤC VÀ ĐÀO TẠO
ĐẠI HỌC THÁI NGUYÊN
ThS. NGUYỄN THỊ HƯƠNG**

**GIÁO TRÌNH
CẤU TRÚC DỮ LIỆU
VÀ GIẢI THUẬT**



**NHÀ XUẤT BẢN KHOA HỌC KỸ THUẬT
Hà Nội - 2010**

Chịu trách nhiệm xuất bản: **TS. Phạm Văn Diễn**

Biên tập: **Minh Luận - Phương Liên**

Trình bày bìa: **Thùy Dương**

**NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT
70 - TRẦN HƯNG ĐẠO - HÀ NỘI**

In 220 bản khổ 15,5x22,5 tại Nhà in Thanh Bình.

Số đăng ký KHXB: 215 - 2010/CXB/367 - 17/KHKT ngày 5/3/2010.

Quyết định xuất bản số: 119/QĐXB - NXBKHKT ngày 5/7/2010.

In xong và nộp lưu chiểu tháng 7/2010.

LỜI NÓI ĐẦU

Môn học Cấu trúc dữ liệu và giải thuật là một trong những môn học cơ sở trong ngành khoa học máy tính. Môn học này giúp sinh viên làm quen với kiến thức cơ bản về cấu trúc dữ liệu và các giải thuật có liên quan. Từ đó tạo điều kiện cho việc nâng cao thêm kỹ thuật lập trình về phương pháp giải các bài toán, giúp sinh viên có khả năng đi sâu thêm vào các môn học như chương trình dịch, trí tuệ nhân tạo, hệ chuyên gia...

Nội dung của giáo trình gồm ba phần:

Phần 1: Giới thiệu các kiến thức cơ bản về phân tích và thiết kế giải thuật. Đưa ra những cái nhìn khái quát trong việc phân tích bài toán, lựa chọn giải thuật và đánh giá giải thuật.

Phần 2: Tập trung vào việc tìm hiểu các cấu trúc dữ liệu, từ việc tổ chức xây dựng cấu trúc, tổ chức lưu trữ, đến các thao tác cơ bản trên cấu trúc và các ứng dụng của cấu trúc dữ liệu.

Phần 3: Giải quyết hai vấn đề rất quan trọng trong lập trình, đó là sắp xếp và tìm kiếm. Các giải thuật sắp xếp và tìm kiếm được đưa ra ở đây là các giải thuật đã được áp dụng rất nhiều trong thực tế, từ các giải thuật cơ bản với độ phức tạp cao, đến các giải thuật nâng cao.

Phần 1 - GIẢI THUẬT

Chương 1 - MỞ ĐẦU

1.1. Giải thuật và cấu trúc dữ liệu

Giải thuật (statement): Là một dãy các câu lệnh chặt chẽ và rõ ràng, xác định một trình tự các thao tác trên một số đối tượng nào đó, sao cho sau một số hữu hạn các bước thực hiện, cho ta một kết quả mong muốn.

Giải thuật chỉ phản ánh các phép xử lý.

Dữ liệu (data): Biểu diễn các thông tin cần thiết cho bài toán, là đối tượng để xử lý trên máy tính.

Các phần tử của dữ liệu thường có mối quan hệ với nhau, việc tổ chức dữ liệu theo một cấu trúc thích hợp (cấu trúc dữ liệu) giúp cho việc thực hiện các phép xử lý trên dữ liệu đạt được hiệu quả cao hơn.

Mối quan hệ giữa cấu trúc dữ liệu và giải thuật: Giải thuật tác động trên cấu trúc dữ liệu để đưa ra kết quả mong muốn. Giữa giải thuật và cấu trúc dữ liệu có mối quan hệ mật thiết với nhau. Cấu trúc dữ liệu thay đổi giải thuật cũng thay đổi theo.

Ví dụ: Minh họa cấu trúc dữ liệu thay đổi, giải thuật cũng thay đổi theo.

Bài toán:

Input:

- Một danh sách gồm những cặp (tên đơn vị, số điện thoại):
 $(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n);$

- Tên đơn vị cần tìm số điện thoại.

Output:

- In ra số điện thoại ứng với tên đơn vị nhập vào.

Phép xử lý cơ bản của bài toán là “tìm kiếm”. Cụ thể:

Nếu danh sách chưa được sắp theo tên đơn vị: Duyệt lần lượt các tên trong danh sách $a_1, a_2, a_3, \dots, a_n$ cho đến lúc tìm thấy đơn vị a_i đã chỉ định, đối chiếu ra số điện thoại tương ứng.

Nếu trước đó danh mục điện thoại đã được sắp theo thứ tự từ điển đối với tên đơn vị: áp dụng một giải thuật tìm kiếm khác tốt hơn như vẫn làm khi tra từ điển.

Nếu lại tổ chức thêm một bảng danh mục chỉ dẫn theo chữ cái đầu tiên của “tên đơn vị”. Việc tìm kiếm số điện thoại của “Đại học Bách khoa” sẽ bỏ qua các tên đơn vị mà chữ cái đầu không phải là Đ.

Một cấu trúc dữ liệu sẽ có tương ứng một giải thuật, khi nghiên cứu các cấu trúc dữ liệu ta đồng thời phải xác lập các giải thuật xử lý trên cấu trúc ấy.

1.2. Cấu trúc dữ liệu và các vấn đề liên quan

1.2.1. Lựa chọn cấu trúc dữ liệu

Trong một bài toán: Dữ liệu = {Các phần tử cơ sở} // dữ liệu nguyên tử (atom): 1 ký tự, 1 chữ số...

Trên cơ sở dữ liệu nguyên tử, các cung cách khả dĩ, liên kết chúng lại với nhau ta được các cấu trúc dữ liệu khác nhau.

Lựa chọn một cấu trúc dữ liệu thích hợp để tổ chức dữ liệu vào, trên cơ sở đó xây dựng được giải thuật xử lý hữu hiệu đưa tới kết quả mong muốn là một khâu rất quan trọng.

Khi ứng dụng của máy tính điện tử chỉ mới có trong phạm vi các bài toán khoa học kỹ thuật thì ta chỉ gặp những cấu trúc dữ liệu đơn giản như biến, vector, ma trận...

Khi ứng dụng mở rộng sang các lĩnh vực khác, ta thường gọi là những bài toán phi số (non numerical problems), các cấu trúc dữ liệu này không còn đủ đặc trưng cho các mối quan hệ mới của dữ liệu nữa, đòi hỏi phải có những cấu trúc dữ liệu mới, phù hợp hơn.

Việc đi sâu vào các cấu trúc dữ liệu mới chính là sự quan tâm của chúng ta trong giáo trình này.

1.2.2. Phép toán trên cấu trúc dữ liệu

Đối với các bài toán phi số, các cấu trúc dữ liệu mới thường đi kèm với các phép toán mới tác động trên cấu trúc ấy.

Ví dụ:

- + Phép tạo lập hay huỷ bỏ một cấu trúc;
- + Phép truy cập vào từng phần tử của cấu trúc;
- + Phép bổ sung hay loại bỏ một phần tử trên cấu trúc.

Các phép toán có những tác động khác nhau đối với từng cấu trúc. Chọn một cấu trúc dữ liệu, ta phải nghĩ ngay tới các phép toán tác động trên chúng.

1.2.3. Cấu trúc lưu trữ

Biểu diễn một cấu trúc dữ liệu là cách cài đặt cấu trúc ấy trên máy tính. Trên cơ sở các cấu trúc lưu trữ để thực hiện các phép xử lý có thể có nhiều cấu trúc lưu trữ khác nhau cho cùng một cấu trúc dữ liệu và một cấu trúc dữ liệu được biểu diễn trong bộ nhớ bởi nhiều cấu trúc lưu trữ.

Cấu trúc dữ liệu tương ứng với bộ nhớ trong: lưu trữ trong.

Cấu trúc dữ liệu tương ứng với bộ nhớ ngoài: lưu trữ ngoài.

1.2.4. Cấu trúc dữ liệu tiền định

Mọi ngôn ngữ lập trình đều có cấu trúc dữ liệu tiền định. Nhưng không phải tất cả các cấu trúc tiền định đều được sử dụng, đều đáp ứng

được mọi yêu cầu cần thiết về cấu trúc, khi đó người thiết kế giải thuật phải biết linh hoạt vận dụng các ngôn ngữ để mô phỏng các cấu trúc dữ liệu đã chọn cho bài toán cần giải quyết.

Ví dụ: Nếu xử lý hồ sơ cán bộ mà dùng ngôn ngữ Pascal, ta tổ chức mỗi hồ sơ dưới dạng một bản ghi (record - bao gồm nhiều thành phần (trường), không nhất thiết phải cùng kiểu).

Nếu dùng ngôn ngữ Fortran thì gặp khó khăn (ta chỉ có thể mô phỏng các mục của hồ sơ dưới dạng vector hay ma trận, khi đó việc xử lý sẽ phức tạp hơn).

1.3. Ngôn ngữ diễn đạt giải thuật

Ngôn ngữ được sử dụng phải có đủ khả năng diễn đạt giải thuật trên các cấu trúc đề cập: có một độ linh hoạt nhất định, không quá gò bó câu nệ về cú pháp, nhưng cũng gần gũi với những chuẩn, khi cần có thể dễ dàng chuyển đổi. Trong giáo trình này chúng ta sử dụng ngôn ngữ tựa Pascal.

Ngôn ngữ lưu đồ hay sơ đồ khối là một công cụ rất trực quan để diễn đạt các thuật toán. Biểu diễn bằng lưu đồ sẽ giúp ta có được một cái nhìn tổng quan về toàn cảnh của quá trình xử lý theo thuật toán.

Lưu đồ là một hệ thống các nút có hình dạng khác nhau, thể hiện các chức năng khác nhau và được nối với nhau bởi các cung. Lưu đồ được tạo thành bởi bốn thành phần chủ yếu sau đây:

- 1) *Nút giới hạn:* được biểu diễn bởi hình ôvan có ghi chữ bên trong.

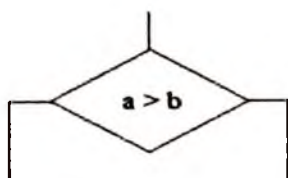


Các nút trên còn được gọi là nút đầu và nút cuối của lưu đồ.

2) *Nút thao tác*: là một hình chữ nhật có ghi các lệnh cần thực hiện. Ví dụ:



3) *Nút điều kiện*: thường là một hình thoi có ghi điều kiện cần kiểm tra. Trong các cung nối với nút này có hai cung ra chỉ hướng đi theo hai trường hợp: điều kiện đúng và điều kiện sai. Ví dụ:



4/ *Cung*: là các đường nối từ nút này đến nút khác của lưu đồ.

Hoạt động của thuật toán theo lưu đồ được bắt đầu từ nút đầu tiên. Sau khi thực hiện các thao tác hoặc kiểm tra điều kiện ở mỗi nút thì bộ xử lý sẽ theo một cung để đến nút khác. Quá trình thực hiện thuật toán dừng khi gặp nút kết thúc hay nút cuối.

Trong giáo trình này chúng ta chủ yếu sử dụng ngôn ngữ tự nhiên và mã giả để trình bày thuật toán. Trong cách sử dụng ngôn ngữ tự nhiên ta sẽ liệt kê các bước thực hiện các thao tác hay công việc nào đó của thuật toán bằng ngôn ngữ mà con người sử dụng một cách phổ thông hàng ngày. Các thuật toán được trình bày trong hai ví dụ trên chính là cách biểu diễn thuật toán dùng ngôn ngữ tự nhiên. Mặc dù cách biểu diễn này khá tự nhiên và không đòi hỏi người viết thuật toán

phải biết nhiều quy ước khác, nhưng nó không thể hiện rõ tính cấu trúc của thuật toán nên không thuận lợi cho việc thiết kế và cài đặt những thuật toán phức tạp. Hơn nữa trong nhiều trường hợp việc biểu diễn thuật toán bằng ngôn ngữ tự nhiên tỏ ra dài dòng và dễ gây ra sự nhầm lẫn đối với người đọc. Còn việc sử dụng lưu đồ sẽ rất công kềnh đối với các thuật toán phức tạp.

Mã giả:

Để biểu diễn thuật toán một cách hiệu quả, người ta thường dùng mã giả (pseudocode). Theo cách này, ta sẽ sử dụng một số quy ước của một ngôn ngữ lập trình, chẳng hạn là ngôn ngữ lập trình Pascal, nhất là các cấu trúc điều khiển của ngôn ngữ lập trình như các cấu trúc chọn, các cấu trúc lặp.

Trong mã giả ta còn sử dụng cả các ký hiệu toán học, các biến, và đôi khi cả cấu trúc kiểu thủ tục. Cấu trúc thuật toán kiểu thủ tục thường được sử dụng để trình bày các thuật toán đệ qui hay các thuật toán quá phức tạp cần phải được trình bày thành nhiều cấp độ.

Cùng với việc sử dụng các biến, trong thuật toán rất thường gặp một phát biểu hành động đặt (hay gán) một giá trị cho một biến. Ví dụ: hành động tăng biến i lên 1 có thể được viết như sau:

$i := i + 1$

hay

$i \leftarrow i + 1$

Các cấu trúc thường được sử dụng trong mã giả dựa theo ngôn ngữ lập trình Pascal gồm:

1) Cấu trúc chọn:

* **if** (điều kiện) **then** (hành động)

* **if** (điều kiện) **then** (hành động)

else (hành động)

2) Cấu trúc lặp:

* **while** (điều kiện) **do** (hành động)

* **Repeat** (hành động)

Until (điều kiện)

* **for** (biến): = (giá trị đầu) **to** (giá trị cuối) **do** (hành động)

* **for** (biến): = (giá trị đầu) **downto** (giá trị cuối) **do** (hành động)

3) Cấu trúc nhảy **goto**. Ngoài ra người ta còn sử dụng lệnh ngắt vòng lặp **break**.

Dưới đây là các thuật toán được biểu diễn bằng mã giả (sử dụng các cấu trúc điều khiển của ngôn ngữ lập trình Pascal). Trước khi viết các bước thực hiện thuật toán ta thường ghi rõ những gì được cho trước (phần *nhập*) và kết quả cần đạt được (phần *xuất*).

Thuật toán tìm phần tử lớn nhất trong một dãy hữu hạn các số nguyên:

Nhập: dãy số a_1, a_2, \dots, a_n

Xuất: max là giá trị lớn nhất trong dãy số đã cho.

Thuật toán:

1. $\text{max} := a_1$

2. **for** $i := 2$ **to** n **do**

if $\text{max} < a_i$ **then** $\text{max} := a_i$

3. max là giá trị lớn nhất trong dãy số.

Thuật toán giải phương trình bậc hai $ax^2 + bx + c = 0$ ($a \neq 0$):

Nhập: 3 hệ số a, b, c

Điều kiện: $a \neq 0$

Xuất: nghiệm của phương trình

Thuật toán:

1. $\Delta := b^2 - 4 \cdot a \cdot c$

2. if $\Delta > 0$ then

begin

$x_1 := (-b - \sqrt{\Delta}) / (2 \cdot a);$

$x_2 := (-b + \sqrt{\Delta}) / (2 \cdot a);$

Xuất kết quả: phương trình có hai nghiệm là x_1 và x_2 ;

end

3. else if $\Delta = 0$ then

Xuất kết quả: phương trình có nghiệm kép là $-b / (2 \cdot a)$

4. else { trường hợp $\Delta < 0$ }

Xuất kết quả: phương trình vô nghiệm;

(Trong thuật toán này, ký hiệu $\sqrt{\Delta}$ dùng để chỉ căn bậc hai dương của Δ).

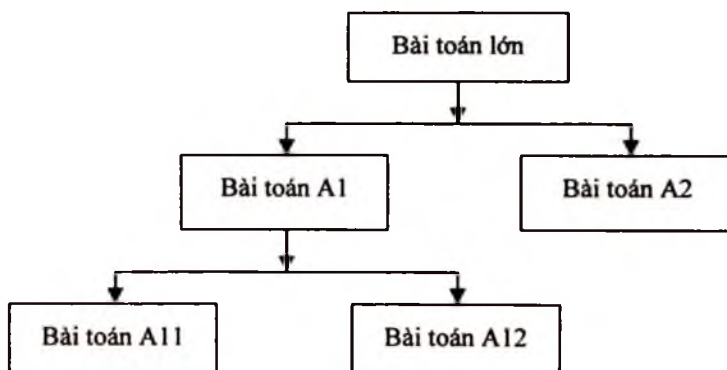
Chương 2 - PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT

2.1. Từ bài toán đến chương trình

2.1.1. Mô đun hoá và việc giải quyết bài toán

Mô đun hoá bài toán cho phép phân chia một bài toán lớn thành nhiều bài toán nhỏ độc lập, và tiếp tục phân chia bài toán nhỏ thành nhiều bài toán nhỏ hơn... Mỗi bài toán tương đương với một mô đun.

Việc tổ chức lời giải của bài toán sẽ được thể hiện theo một cấu trúc phân cấp, có dạng:



Chiến thuật này được gọi là chiến thuật “chia để trị” (divide and conquer). Thực hiện chiến thuật bằng cách thiết kế từ trên xuống (top-down design) - thiết kế từ khái quát đến chi tiết.

Cách giải quyết bài toán sử dụng phương pháp mô đun hoá:
Phân tích tổng quát toàn bộ vấn đề (căn cứ vào dữ liệu và các mục tiêu

đặt ra), đề cập đến công việc chủ yếu, rồi đi dần vào giải quyết bài toán cụ thể một cách chi tiết hơn.

Ví dụ: “Dùng máy tính để quản lý và bảo trì các hồ sơ về học bổng của các sinh viên ở diện được tài trợ, đồng thời thường kỳ phải lập các báo cáo tổng kết để đệ trình lên bộ”.

Bài toán:

Đầu vào: Các hồ sơ về học bổng của sinh viên (một tệp các hồ sơ) gồm các thông tin sau:

- + Số hiệu của học sinh;
- + Điểm trung bình theo học kỳ;
- + Điểm đạo đức.

Đầu ra: Quản lý và bảo trì các hồ sơ

1) Tìm lại và hiển thị các bản ghi của bất kỳ sinh viên nào tại đầu cuối (terminal) của người dùng.

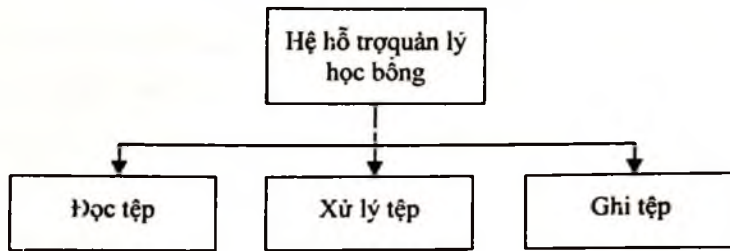
2) Cập nhật (update) được bản ghi của một sinh viên bằng cách thay đổi điểm trung bình, điểm đạo đức, khoản tài trợ nếu cần.

3) In bảng tổng kết chứa những thông tin hiện thời (đã được cập nhật mỗi khi thay đổi).

Xuất phát từ ba nhận định nêu trên, giải thuật xử lý sẽ phải giải quyết ba nhiệm vụ chính sau:

- Lưu trữ: Thông tin về sinh viên được học bổng trên đĩa phải được đọc vào bộ nhớ trong để có thể xử lý (nhiệm vụ đọc tệp).
- Xử lý các thông tin này để tạo ra kết quả mong muốn (nhiệm vụ xử lý tệp).
- Sao chép những thông tin đã được cập nhật vào tệp trên đĩa để lưu trữ cho việc xử lý sau này (nhiệm vụ ghi tệp).

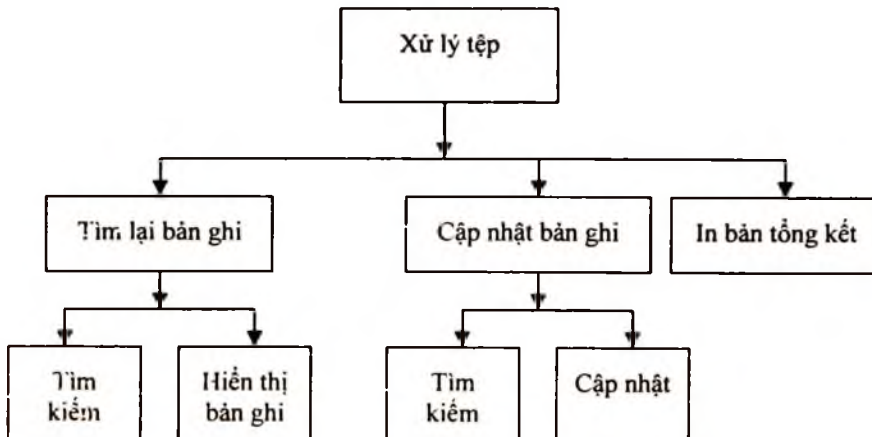
Sơ đồ:



+ Nhiệm vụ xử lý tệp được phân thành ba yêu cầu chính đã được nêu ở trên:

- Tìm lại bản ghi của một sinh viên cho trước;
- Cập nhật thông tin trong bản ghi sinh viên;
- In bảng tổng kết những thông tin về các sinh viên được học bổng.

+ Sơ đồ phân chia thành nhiệm vụ con này cũng có thể chia thành những nhiệm vụ nhỏ hơn, theo sơ đồ cấu trúc như sau:



Nhận xét: Cách thiết kế giải thuật top - down giúp cho việc giải quyết bài toán được rõ ràng hơn, tránh sa đà ngay vào các chi tiết phụ.

Chương trình được thiết kế theo cách này thì việc tìm hiểu cũng như sửa chữa, chỉnh lý sẽ dễ dàng hơn, là nền tảng cho lập trình có cấu trúc.

2.1.2. Phương pháp tinh chỉnh từng bước

Là phương pháp thiết kế giải thuật và phát triển chương trình gắn liền với lập trình: chương trình sẽ được thể hiện dần dần từ dạng ngôn ngữ tự nhiên, qua giả ngôn ngữ, rồi đến ngôn ngữ lập trình (gọi là các bước tinh chỉnh).

Chương trình đi từ mức “làm cái gì” đến mức “làm như thế nào”, ngày càng sát với các chức năng ứng với các câu lệnh của ngôn ngữ lập trình đã chọn.

Dữ liệu cũng được “tinh chế” từ dạng cấu trúc sang dạng lưu trữ, cài đặt cụ thể.

Ví dụ: Viết chương trình sắp xếp một dãy n số nguyên khác nhau theo thứ tự tăng dần.

Phác thảo giải thuật:

4 1 5 3 2 - Từ một dãy số nguyên chưa được sắp

4 5 3 2 **1** - Chọn ra một số nhỏ nhất, đặt nó vào cuối dãy đã được sắp.

+ Lặp lại quy trình này cho đến khi dãy chưa được sắp chỉ còn rỗng.

Việc phác thảo giải thuật trên còn rất thô, nó chỉ thể hiện những ý cơ bản, đòi hỏi phải chi tiết hơn về cấu trúc dữ liệu và cấu trúc lưu trữ: dãy số được coi như dãy các phần tử của một vector, được lưu trữ bởi một vector n từ máy kể tiếp từ bộ nhớ trong $(a_1, a_2, \dots, a_i, \dots, a_n)$, $1 \leq i \leq n$.

Ta định hướng chương trình của ta vào ngôn ngữ tựa Pascal.

- Bước tinh chỉnh đầu tiên sẽ là:

For $i := 1$ to n do

Begin

Xét từ a_i đến a_n để tìm số nhỏ nhất a_j

Đổi chỗ giữa a_i và a_j

End

- Tới đây ta có hai nhiệm vụ cần làm rõ thêm:

1) *Tìm số nguyên nhỏ nhất a_j trong các số từ a_i đến a_n ;*

2) *Đổi chỗ giữa a_i và a_j .*

Nhiệm vụ 1) có thể thực hiện bằng cách: “*Thoạt tiên coi a_i là số nhỏ nhất tạm thời, lần lượt so sánh a_i với a_{i+1} , a_{i+2} ... khi thấy số nào nhỏ hơn thì lại coi số đó là số nhỏ nhất mới, khi so sánh tới a_n thì số nhỏ nhất sẽ được xác định*”.

- Ta có bước tinh chỉnh sau:

$j := i;$

For $k := j+1$ to n do

if $a_k < a_j$ then $j := k;$

$B := a_i; a_i := a_j; a_j := B$

- Chương trình sắp xếp dưới dạng thủ tục như sau:

Procedure SORT(A, n)

1) For $i := 1$ to $n-1$ do

Begin

2) {Chọn số nhỏ nhất} $j := i;$

for $k := j+1$ to n do

if $A[k] < A[j]$ then $j := k;$

3) {đổi chỗ} $B := A[j]; A[i] := A[j]; A[j] := B;$

End;

Return;

Ví dụ: Cho một ma trận vuông cấp $n \times n$ các số nguyên. Nhập ma trận vào và in ra các phần tử thuộc đường chéo song song với đường chéo chính.

- Phác thảo giải thuật:

1) Nhập n;

2) Nhập các phần tử của ma trận;

3) In ra các đường chéo song song với đường chéo chính.

+ Hai nhiệm vụ 1), 2) có thể diễn đạt bằng giải thuật Pascal:

```
Readln(n);
```

```
For i:=1 to n do
```

```
    For j:=1 to n do
```

```
        Readln(a[i,j]);
```

+ Nhiệm vụ 3) cần phân tích kỹ hơn. Ta thấy về đường chéo có thể phân tích thành hai loại:

Đường chéo ứng với cột từ 1 đến n;

Đường chéo ứng với hàng từ 2 đến n.

Vậy ta tách thành hai nhiệm vụ con:

3.1 For j:=n downto 1 do

 In đường chéo ứng với cột j

3.2 For i:=2 to n do

 In đường chéo ứng với hàng i

- Thực hiện công việc chi tiết hơn: “In đường chéo ứng với cột j” (đường chéo phía trên đường chéo chính).

Với: j=n Thì in 1 phần tử: hàng 1 cột j

 j= n-1 Thì in 2 phần tử: hàng 1 cột j, hàng 2 cột j+1

 j=n-2 Thì in 3 phần tử: hàng 1 cột j, hàng 2 cột j+1, hàng 3 cột j+2.

Ta thấy: số lượng các phần tử được in ứng với mỗi cột là n-j+1; phần tử được in chính là: $A[i, j+(i-1)]$, với $1 \leq i \leq n-j+1$. Như vậy, 3.1 có thể tinh chỉnh thành:

```
For i:=1 to n-j+1 do
```

```
    Write (a[i,i+j-1]:8);
```

```
    Writeln;
```

Tương tự (đường chéo phía dưới đường chéo chính):

```
For j:=1 to n-j+1 do  
    Write(a[i+j-1,i]:8);  
    Writeln;
```

- Sau đây là chương trình hoàn chỉnh:

```
Program INCHEO  
    Const max=30;  
    Type matran=array[1..max,1..max] of Integer;  
    Var a:matran; n,i,j:Integer;  
    Begin  
        Repeat  
            Write('Nhập cỡ ma trận:');  
            Readln(n);  
        Until (n>0) and (n<=max)  
        Write('Nhập phần tử ma trận:');  
        For i:=1 to n Do  
            For j:=1 to n Do Readln  
                (a[i,j]);  
            Write('In đường chéo');  
        For j:=n down to 1 Do  
            Begin  
                For i:=1 to n-j+1 Do  
                    Write(a[i,i+j-1]:8);  
                Writeln;  
            End;  
        For i:=2 to n Do  
            Begin  
                For j:=1 to n-j+1 do  
                    Write(a[i+j-1,j]:8);  
                Writeln;  
            End;
```

2.2. Phân tích giải thuật

2.2.1. Đặt vấn đề

Khi xây dựng giải thuật và chương trình tương ứng của một bài toán có rất nhiều yêu cầu về phân tích thiết kế hệ thống.

** Yêu cầu phân tích tính đúng đắn của giải thuật:*

Một giải thuật gọi là đúng đắn nếu nó thực sự giải được yêu cầu của bài toán (thể hiện đúng được lời giải của bài toán).

** Yêu cầu về tính đơn giản của giải thuật:*

Dễ hiểu, dễ lập trình, dễ chỉnh lý;

Phân tích thời gian thực hiện giải thuật - một trong các tiêu chuẩn để đánh giá hiệu lực của giải thuật.

2.2.2. Phân tích thời gian thực hiện giải thuật

Thời gian thực hiện giải thuật phụ thuộc vào nhiều yếu tố:

1) Kích thước dữ liệu vào: Gọi n là số lượng dữ liệu đưa vào thì thời gian thực hiện T của một giải thuật được biểu diễn bởi hàm $T(n)$.

2) Các kiểu lệnh và tốc độ xử lý của máy tính.

3) Ngôn ngữ viết chương trình và chương trình dịch ngôn ngữ ấy.

Các yếu tố 2) và 3) không đồng đều với mọi máy tính. Không dựa vào chúng khi xác lập $T(n)$. Cách đánh giá $T(n)$ này cho ta khái niệm về “độ phức tạp tính toán của giải thuật” (độ lớn của thời gian thực hiện giải thuật).

a. Độ phức tạp tính toán của giải thuật:

Một chương trình máy tính thường được cài đặt dựa trên một thuật toán để giải bài toán hay vấn đề đặt ra. Một đòi hỏi đương nhiên là thuật toán phải đúng. Tuy nhiên, ngay cả khi thuật toán đúng, chương trình vẫn có thể là không sử dụng được đối với một số dữ liệu nhập nào đó bởi vì thời gian cần thiết để chạy chương trình hay vùng nhớ cần thiết để lưu trữ dữ liệu (như các biến trong chương trình, các file lưu trữ...) quá lớn.

Thuật ngữ *phân tích thuật toán* đề cập đến một quá trình tìm ra một đánh giá về thời gian và không gian cần thiết để thực hiện thuật toán. *Độ phức tạp của thuật toán* được thể hiện qua khối lượng thời gian và không gian để thực hiện thuật toán. Không gian ở đây được hiểu là các yêu cầu về bộ nhớ, thiết bị lưu trữ... của máy tính để thuật toán có thể làm việc được. Việc xem xét độ phức tạp về không gian của thuật toán phụ thuộc phần lớn vào cấu trúc dữ liệu được sử dụng trong cài đặt thuật toán. Trong phần này chúng ta chỉ đề cập đến độ phức tạp về thời gian của thuật toán.

Chúng ta cũng có thể đạt được những thông tin rất hữu ích khi phân tích độ phức tạp thời gian của thuật toán cơ sở của một chương trình máy tính. Đánh giá một cách chính xác thời gian thực hiện một chương trình phụ thuộc vào rất nhiều yếu tố và là một công việc rất khó khăn. Tuy nhiên các nhà toán học đã phân tích cho chúng ta độ phức tạp của hầu hết các thuật toán thường được sử dụng như các thuật toán sắp xếp, các thuật toán tìm kiếm, các thuật toán số học...

Độ phức tạp thời gian của thuật toán thường được đánh giá dựa vào số lượng thao tác được sử dụng trong thuật toán và số lượng thao tác này phụ thuộc vào cỡ (size) của dữ liệu nhập. Ta còn gọi độ phức tạp thời gian của thuật toán là *độ phức tạp tính toán*. Các thao tác được sử dụng để đo độ phức tạp của thuật toán có thể là phép so sánh 2 số nguyên, cộng 2 số nguyên, nhân 2 số nguyên, chia 2 số nguyên, hay bất kỳ thao tác cơ bản nào khác. Như thế ta có thể xem thời gian thực hiện thuật toán là một hàm phụ thuộc vào dữ liệu nhập (thường là cỡ của dữ liệu nhập). Nếu gọi cỡ dữ liệu nhập là n thì độ phức tạp có thể được xem là một hàm theo n .

Chúng ta có thể đặt ra câu hỏi về thời gian thực hiện thuật toán nhỏ nhất đối với các dữ liệu nhập có cỡ n . Ta có thể nêu lên một số bài toán có dữ liệu nhập có cỡ n như: sắp xếp dãy n số nguyên, tìm số nhỏ nhất trong dãy n số nguyên... Thời gian nhỏ nhất này được gọi là *thời gian thực hiện thuật toán trong trường hợp tốt nhất* đối với dữ liệu

nhập có cỡ n . Tương tự ta cũng thường đề cập đến thời gian thực hiện thuật toán lớn nhất đối với các dữ liệu nhập có cỡ n , và gọi là *thời gian thực hiện thuật toán trong trường hợp xấu nhất* đối với dữ liệu nhập có cỡ n . Ngoài ra, đối với thuật toán có dữ liệu nhập có cỡ n trong một tập hữu hạn nào đó, ta còn muốn tính ra *thời gian trung bình* để thực hiện thuật toán.

Ví dụ: Thuật toán tìm giá trị lớn nhất trong dãy gồm n số nguyên (xem ví dụ 1, mục I). Trong thuật toán này nếu ta xem thời gian thực hiện thuật toán là số lần thực hiện phép so sánh hay phép gán thì thời gian thực hiện thuật toán trong trường hợp xấu nhất là:

$$t(n) = 1 + 2*(n-1) = 2n-1$$

và thời gian thực hiện thuật toán trong trường hợp tốt nhất là:

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

Nếu thời gian thực hiện giải thuật là $T(n)=cn^2$, độ phức tạp tính toán có cấp là n^2 , ký hiệu là $T(n)=O(n^2)$. Một cách tổng quát, có thể định nghĩa:

Một hàm $f(n)$ được xác định là $O(g(n))$, $f(n)=O(g(n))$, được gọi là có cấp $g(n)$ nếu tồn tại hằng số c và n_0 sao cho: $f(n) \leq c*g(n)$ khi $n \geq n_0$.

Thường các hàm thể hiện độ phức tạp tính toán của giải thuật có dạng: $\log_2 n$, n , $n \log_2 n$, n^2 , n^3 , 2^n , $n!$, $nn \dots$

b. Xác định độ phức tạp tính toán

Với một số giải thuật, ta có thể áp dụng một số quy tắc sau:

+ Quy tắc tính tổng:

Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình p_1 , p_2 mà $T_1(n)=O(f(n))$, $T_2=O(g(n))$, thì thời gian thực hiện p_1 rồi p_2 tiếp theo sẽ là: $T_1(n)+ T_2(n)=O(\max(f(n), g(n)))$.

+ Quy tắc nhân:

$$p_1: T_1(n)=O(f(n))$$

$$p_2: T_2(n)=O(g(n))$$

Thời gian thực hiện p_1, p_2 lồng nhau sẽ là: $T_1(n) * T_2(n) = O(f(n), g(n))$

Chú ý:

+ Khi đánh giá thời gian thực hiện giải thuật ta cần chú ý tới các bước tương ứng với một phép toán mà ta gọi là các phép toán tích cực (active operation).

+ Ví dụ: Xét giải thuật tính giá trị của e^x theo công thức gần đúng:

$e^x \approx 1 + x/1! + x^2/2! + \dots + x^n/n!$, với x, n cho trước

```
Program EXP1;  
  {tính từng số hạng rồi cộng lại}  
  Read(n); S:=1; Read(x);  
  For i:=1 to n do  
    Begin  
      P:=1;  
      For j:=1 to i do p:=p*x/j;  
      S:=S+p;  
    End;  
  End;
```

Ta có thể coi phép toán tích cực ở đây là phép $p := p * x / j$, số lần thực hiện:

$$1 + 2 + 3 + \dots + n = n(n+1)/2 \text{ lần}$$

Vậy thời gian thực hiện giải thuật này được đánh giá là $T(n) = O(n^2)$.

```
Program EXP2;  
{Dựa vào số hạng trước để tính số hạng sau theo  
cách:  $x^2/2! = x/1! * x/2$ ;  $x^n/n! = x^{n-1}/(n-1)!}$   
1) Read(x); S:=1; p:=1;  
2) For i:=1 to n do  
  Begin  
    P:=p*x/i;  
    S:=S+p;  
  End;  
3) End;
```

Bây giờ (t) thực hiện là $T(n)=O(n)$, vì phép toán $p:= p*x/j$ chỉ thực hiện n lần. Thời gian thực hiện giải thuật không chỉ phụ thuộc vào kích thước của dữ liệu mà còn phụ thuộc vào chính tình trạng của dữ liệu đó.

Xét $T(n)$ trong trường hợp: xấu nhất, trung bình, tốt nhất.

+ Người ta thường đánh giá thời gian thực hiện giải thuật qua giá trị xấu nhất của $T(n)$.

+ Ví dụ:

Program SEARCH;

{cho vector v có n phần tử, tìm trong v một phần tử có giá trị bằng X cho trước}

1) Found:=false; {Found là biến logic báo hiệu ngừng tìm khi tìm thấy}

$i:=1$;

2) While ($i \leq n$) and (not Found) do

 if $v[i] = X$ then

 Begin

 Found:=true;

$K:=i$;

 Write(k);

 End

 else $i:=i+1$

3) End;

Ta coi phép toán tích cực ở đây là phép so sánh $v[i]$ và X , số lần phép toán này thực hiện phụ thuộc vào chỉ số i thoả mãn $v[i]=X$.

Chương 3 - ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY

3.1. Khái niệm về đệ quy

Một đối tượng được gọi là đệ quy nếu nó bao gồm chính nó như một bộ phận hoặc nó được định nghĩa dưới dạng của chính nó.

Ví dụ:

1) *Số tự nhiên:*

1 là số tự nhiên

x là số tự nhiên nếu $x-1$ là số tự nhiên

2) *Hàm n giai thừa: $n!$*

$0! = 1;$

Nếu $n > 0$ thì $n! = n(n-1)!$

3.2. Giải thuật đệ quy và thủ tục đệ quy

Nếu lời giải của bài toán T được thực hiện bởi lời giải của một bài toán T' , có dạng như T thì đó là một lời giải đệ quy. Giải thuật tương ứng với lời gọi như vậy được gọi là giải thuật đệ quy ($T' < T$).

Ví dụ: Xét bài toán tìm một từ trong một quyển từ điển

Giải thuật:

If từ điển là một trang

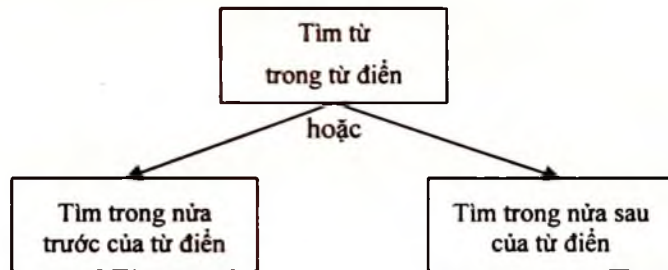
Then tìm từ trong trang này

Else Begin

Mở từ điển vào trang giữa;
Xác định xem nửa nào của từ điển chứa từ cần tìm;
If Từ đó nằm ở nửa trước của từ điển
Then Tìm từ đó trong nửa trước
Else Tìm từ đó trong nửa sau

End;

Giải thuật trên mới chỉ được nêu dưới dạng thô, còn nhiều chỗ chưa cụ thể. Có thể hình dung chiến thuật tìm kiếm này một cách tổng quát như sau:



Hai đặc điểm chính cần lưu ý:

Khi mỗi lần từ điển được tách đôi thì một nửa thích hợp sẽ lại được tìm kiếm bằng một chiến thuật như đã được dùng trước đó.

Trường hợp đặc biệt, đạt được sau nhiều lần tách đôi, đó là khi từ điển chỉ còn một trang, việc tách đôi ngừng lại và bài toán đã trở thành đủ nhỏ để có thể giải quyết trực tiếp bằng cách tìm từ trong trang đó (trường hợp suy biến).

Có thể coi đây là chiến thuật “chia để trị”. Bài toán được tách thành bài toán nhỏ hơn, bài toán nhỏ hơn lại được giải quyết với chiến thuật chia để trị như trước đó cho tới khi xuất hiện trường hợp suy biến.

Viết giải thuật tìm kiếm dưới dạng một thủ tục:

Procedure SEARCH (dict, word);

{dict được gọi là đầu mỗi để truy nhập được vào từ điển đang xét,
word chỉ từ cần tìm}

1) *If* Từ điển chỉ còn là một trang

Then Tìm từ word trong trang này

Else

Begin

Mở từ điển vào trang giữa;

Xác định xem nửa nào chứa từ word;

If word nằm ở nửa trước của từ điển

Then SEARCH(dict1, word)

Else SEARCH(dict2, word)

End;

{dict1, dict2 là hai đầu mỗi có thể truy cập được vào đầu trước và đầu
sau của từ điển}

2) Return

Đặc điểm của thủ tục đệ quy:

+ Trong thủ tục đệ quy có lời gọi đến chính nó;

+ Mỗi lần có lời gọi thì kích thước của bài toán đã thu nhỏ hơn trước;

+ Có một trường hợp đặc biệt (trường hợp suy biến): bài toán sẽ được giải quyết theo một cách khác hẳn và gọi đệ quy cũng kết thúc.

Đệ quy:

+ Đệ quy trực tiếp (directly recursive): Thủ tục chứa lời gọi đến chính nó;

+ Đệ quy gián tiếp (indirectly recursive): Thủ tục chứa lời gọi đến thủ tục khác mà thủ tục này lại chứa lời gọi đến chính nó;

Thiết kế giải thuật đệ quy:

Hàm $n!$ được định nghĩa:

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n \neq 0 \end{cases}$$

Giải thuật đệ quy được viết dưới dạng thủ tục hàm như sau:

Function FACTORIAL(n)

1) if n=0 then FACTORIAL:=1

else FACTORIAL:= n * FACTORIAL (n - 1)

2) return;

Theo thủ tục đệ quy:

Lời gọi đến chính nó nằm sau câu lệnh else

Mỗi lần gọi đệ quy FACTORIAL, thì giá trị của n giảm 1

+ Ví dụ FACTORIAL(4) gọi đến FACTORIAL(3) → FACTORIAL(2) → FACTORIAL(1) → FACTORIAL(0).

FACTORIAL(0) chính là trường hợp suy biến, nó được tính theo cách đặc biệt FACTORIAL(0) = 1

3.2.1. Dãy số Fibonacci

Dãy này bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ theo những luật sau:

1) Các con thỏ không bao giờ chết;

2) Hai tháng sau khi ra đời một cặp thỏ sẽ sinh ra một cặp thỏ con (1 đực, 1 cái);

3) Khi đã sinh con rồi, thì mỗi tháng tiếp chúng lại sinh ra được một cặp con mới.

Giả sử bắt đầu từ một cặp mới ra đời thì đến tháng thứ n sẽ có bao nhiêu cặp?

Với n = 6, ta thấy:

Tháng 1: 1 cặp (cặp ban đầu);

Tháng 2: 1 cặp (cặp ban đầu vẫn chưa đẻ);

Tháng 3: 2 cặp (đã có thêm một cặp con);

Tháng 4: 3 cặp (cặp ban đầu đẻ thêm một cặp);

Tháng 5: 5 cặp (cặp ban đầu đẻ thêm một cặp, cặp con đầu tiên đã bắt đầu đẻ);

Tháng 6: 8 cặp (cặp ban đầu, cặp con đầu tiên, cặp con thứ hai, mỗi cặp đẻ thêm một cặp).

Trong các cặp thỏ ở tháng thứ $n-1$ chỉ có những cặp ở tháng thứ $(n-2)$ mới sinh con ở tháng thứ n , do đó $F(n) = F(n-2) + F(n-1)$.

Vậy ta có thể viết $F(n)$ theo công thức:

$$F(n) = \begin{cases} 1 & n \leq 2 \\ F(n-2) + F(n-1) & n > 2 \end{cases}$$

- Thủ tục đệ quy thể hiện giải thuật tính $F(n)$:

Function $F(n)$

1) If $n \leq 2$ then $F := 1$

 else $F := F(n-2) + F(n-1)$

2) return

Trường hợp suy biến ứng với 2 giá trị $F(1) = 1$ và $F(2) = 1$

3.2.2. Bài toán “Tháp Hà Nội”

- Có n đĩa, kích thước nhỏ dần, đĩa có lỗ ở giữa. Có thể xếp chồng chúng lên nhau, xuyên qua một cọc, to dưới, nhỏ trên.

- Yêu cầu: Chuyển chồng đĩa từ cọc A sang cọc C, theo những điều kiện:

1) Mỗi lần chỉ được chuyển một đĩa;

2) Không khi nào có tình huống, đĩa to ở trên đĩa nhỏ ở dưới (dù là tạm thời);

3) Được phép sử dụng một cọc trung gian (cọc B).

Xét các trường hợp:

* Trường hợp một đĩa:

Chuyển từ cọc A sang cọc C

* Trường hợp hai đĩa:

Chuyển đĩa 1 từ cọc A sang cọc B;

Chuyển đĩa 2 từ cọc A sang cọc C;

Chuyển đĩa 1 từ cọc B sang cọc C.

Như vậy: Bài toán tháp Hà Nội tổng quát với n đĩa được dẫn tới bài toán tương tự với kích thước nhỏ hơn, chẳng hạn: chuyển n đĩa từ cọc A sang cọc C nay là chuyển $n-1$ đĩa từ cọc A sang cọc B. Ở mức này, giải thuật sẽ là:

- Chuyển $(n-2)$ đĩa từ cọc A sang cọc C;

- Chuyển 1 đĩa từ cọc A sang cọc B;

- Chuyển $(n-2)$ đĩa từ cọc C sang cọc B.

Chuyển $(n-1)$ đĩa từ cọc B sang cọc C, giải thuật sẽ là:

- Chuyển $(n-2)$ đĩa từ cọc B sang cọc A;

- Chuyển 1 đĩa từ cọc B sang cọc C;

- Chuyển $(n-2)$ đĩa từ cọc A sang cọc C.

Cứ như thế cho tới khi trường hợp suy biến xảy ra, đó là trường hợp ứng với bài toán chuyển một đĩa mà thôi.

- Giải thuật:

```
Procedure HANOI (n,B,A,C)
1) if n=1 then Chuyen(A,C)
2)   else Begin
      HANOI (n-1,A,C,B);
      HANOI (1,A,B,C);
      HANOI (n-1,B,A,C);
      End;
3) return;
```

3.2.3. Bài toán 8 quân hậu

Bài toán:

Dữ liệu vào:

Bàn cờ 8 ô;

8 quân hậu.

- Yêu cầu: Đặt 8 quân hậu vào 8 ô khác nhau trên bàn cờ, sao cho các quân hậu không ăn được lẫn nhau.

- Ý tưởng của giải thuật:

Sử dụng phương pháp vét cạn (thử) mọi khả năng có thể có

Sử dụng hai kỹ thuật đó là quay lui (lần ngược trở lại) và đệ quy

- Giải thuật (được xem là bộ xương)

{tính chất đệ quy của bài toán được thể hiện trong phép thử này}

Procedure TRY(i)

1) Khởi phát việc chọn vị trí cho quân hậu thứ j

2) Repeat Thực hiện việc chọn tiếp theo;

if an toàn then

Begin

Đặt quân hậu;

if $j < 8$ then

begin

TRY($j+1$);

if không thành công then cắt
quân hậu;

end;

End;

Until (thành công) or (Hết chỗ);

Return;

Để đi tới thủ tục chi tiết hơn, ta phải chuẩn bị dữ liệu biểu diễn các thông tin cần thiết bao gồm:

- Dữ liệu biểu diễn nghiệm (ma trận 8×8);

- Dữ liệu biểu diễn lựa chọn;
- Dữ liệu biểu diễn điều kiện.

Với quân hậu thứ j , vị trí của nó chỉ được chọn trong cột j (j : chỉ số cột). Việc lựa chọn được tiến hành trên 8 giá trị của chỉ số hàng i . Lựa chọn i được chấp nhận khi hai đường chéo qua ô (i, j) phải còn tự do.

Chú ý:

Đường chéo mọi ô có cùng tổng $(i+j)$;

Đường chéo mọi ô có cùng hiệu $(i-j)$;

Ta chọn các mảng 1 chiều Boolean để biểu diễn các tình trạng này:

$a[i]=\text{true}$: Không có quân hậu nào trên hàng i ($1 \leq i \leq 8$)

$b[i+j]=\text{true}$: không có quân hậu nào trên đường chéo $i+j$ ($2 \leq i+j \leq 16$).

$c[i-j]=\text{true}$: không có quân hậu nào trên đường chéo $i-j$ ($-7 \leq i-j \leq 7$).

Như vậy, điều kiện để quân hậu thứ i được chấp nhận là:

$a[i]$ and $b[i+j]$ and $c[i-j]$ có giá trị true

Việc đặt quân hậu được thực hiện bởi:

$X[i]:=\text{true}; a[i]:=\text{false}; b[i+j]:=\text{false}; c[i-j]:=\text{false};$

Cất quân hậu:

$a[i]:=\text{true}; b[i+j]:=\text{true}; c[i-j]:=\text{true};$

Thủ tục TRY bây giờ có thể viết như sau:

Procedure TRY(i :integer; q :boolean);

1. $i:=0$;

2. repeat $i:=i+1$; $q:=\text{false}$;

 if $a[i]$ and $b[i+j]$ and $c[i-j]$ then

 Begin

$X[j]:=i$;

$a[i]:=\text{false}$;

$b[i+j]:=\text{false}$;

```

        c[i-j]:=false;
    if j<8 then
        Begin
            call TRY(j+1,q);
        If (not q) then
            Begin
                a[i]:=true;
                b[i+j]:=true;
                c[i-j]:=true;
            End;
            End;
        Else q:=true
        End;
    Until q or (i=8);
3.return;

```

Với thủ tục TRY này, chương trình cho một lời giải của bài toán 8 quân hậu như sau:

```

1- {khởi tạo tình trạng ban đầu}
    For i:=1 to 8 do a[i]:=true;
    For i:=2 to 16 do b[i]:=true;
    For i:=-7 to 7 do c[i]:=true;
2- {tìm 1 lời giải}
    call TRY(1,q);
3- {In kết quả}
    if q then for i:=1 to 8 do Write("x[i]")
    end

```

Nếu ta muốn có tất cả các lời giải của bài toán, ta cần sửa đôi chút trong thủ tục TRY.

Điều kiện kết thúc quá trình là $j=8$ nên có thể thay câu lệnh Repeat bằng For.

Việc in kết quả thực hiện ngay khi có lời giải, ta viết dưới dạng thủ tục để gọi ngay trong thủ tục TRY.

```
Procedure PRINT(x)
  For k:=1 to 8 do Write(x[k])
Return
```

Thủ tục TRY mới sẽ là:

```
Procedure TRY(j)
1.      For i:=1 to 8 do
If a[i] and b[i+j] and c[i-j] then
      Begin
        X[j]:=i;
        a[i]:=false;
        b[i+j]:=false;
        c[i-j]:=false;
        if j<8 then TRY(j+1)
          else PRINT(x);
        a[i]:=true;
        b[i+j]:=true;
        c[i-j]:=true;
      End;
2.      Return;
```

Chương trình in toàn bộ lời giải của bài toán:

```
Program EIGHT-QUEEN
  for i=1 to 8 do a[i]:=true;
  for i:=2 to 16 do b[i]:=true;
  for i:=-7 to 7 do c[i]:=true;
  call TRY(1)
End
```

3.2.4. Chú ý

Khi thiết kế một giải thuật đệ quy cần trả lời các câu hỏi sau:

1) Có thể định nghĩa được bài toán dưới dạng một bài toán cùng loại nhưng “nhỏ” hơn, như thế nào?

2) Như thế nào là kích thước của bài toán giảm đi ở mỗi lần gọi đệ quy?

3) Trường hợp đặc biệt nào của bài toán được giảm đi ở mỗi lần được gọi là suy biến?

3.3. Hiệu lực của đệ quy

Đệ quy là một công cụ để giải các bài toán, có những bài toán bên cạnh giải thuật đệ quy còn có những giải thuật lặp khá đơn giản và hữu hiệu.

Chẳng hạn giải thuật lặp tính $n!$ có thể viết.

Đệ quy:

```
Function IFAC(n)
1- if n=0 or n=1 then begin
    IFAC:=1;
    Return
    End;
2- IFAC:=1;
   For i:=2 to n do IFAC:=IFAC*i
3- return;
```

Hay giải thuật lặp tính Fibonacci:

```
Function Fibonacci(n)
1. if n<=2 then Fibonacci:=1;
2. Fib1:=1; Fib2:=1;
3. For i=3 to n do
    Begin
    Fibn:=Fib1+Fib2;
    Fib1:=Fib2;
```

```
Fib2:=Fibn;  
End  
4. FIBONACCI:=Fibn;  
Return
```

Chú ý: Khi thay giải thuật đệ quy bằng các giải thuật không tự gọi chúng, ta gọi là khử đệ quy.

3.4. Đệ quy và quy nạp toán học

Quy nạp toán học dùng để chứng minh các tính chất có liên quan đến giải thuật đệ quy.

Ví dụ:

- 1) Chứng minh tính đúng đắn của giải thuật FACTORIAL.
- 2) Đánh giá giải thuật tháp Hà Nội.

Phần 2 - CẤU TRÚC DỮ LIỆU

Chương 4 - MẢNG VÀ DANH SÁCH

4.1. Các khái niệm

Mảng: Là một tập hợp có thứ tự gồm một số cố định các phần tử không có phép bổ sung hoặc loại bỏ một phần tử; chỉ có các phép tạo lập, tìm kiếm, lưu trữ. Mỗi phần tử của mảng ngoài giá trị (info) còn được đặc trưng bởi chỉ số để biểu hiện thứ tự của nó trong mảng. Có mảng một chiều, hai, ba chiều ...

Danh sách: Là một tập có thứ tự nhưng bao gồm một số biến động các phần tử. Phép bổ sung và loại bỏ một phần tử là phép thường xuyên tác động lên danh sách. Một danh sách mà quan hệ lân cận giữa các phần tử được hiển thị ra gọi là *danh sách tuyến tính*.

Danh sách tuyến tính (danh sách được sắp): là một dãy gồm các phần tử x_1, x_2, \dots, x_n có cùng một kiểu dữ liệu, trong đó n được gọi là độ dài của danh sách (số lượng phần tử), nếu $n=0$ thì gọi là danh sách rỗng.

Nếu $n \geq 1$ thì:

x_1 gọi là phần tử đầu tiên;

x_n gọi là phần tử cuối cùng.

Các phép toán trên danh sách:

- Bổ sung vào danh sách một phần tử mới trước hoặc sau một phần tử nào đó thuộc danh sách;
- Loại bỏ khỏi danh sách một phần tử nào đó;

- Truy cập đến một phần tử nào đó thuộc danh sách;
- Tìm số lượng phần tử trong danh sách;
- Duyệt danh sách từ trái sang phải hoặc từ phải sang trái.

Ngoài ra còn có các phép toán sau:

- Ghép hai hoặc nhiều danh sách;
- Tách một danh sách thành nhiều danh sách;
- Sao chép một danh sách;
- Cập nhật (update) danh sách;
- Sắp xếp các phần tử trong danh sách theo một thứ tự nhất định.

Tệp (file): Là một loại danh sách có kích thước lớn, được lưu trữ ở bộ nhớ ngoài.

Phần tử cơ bản của tệp là bản ghi (record) gồm n trường dữ liệu cần thiết, tương ứng với các thuộc tính khác nhau.

4.2. Cấu trúc lưu trữ mảng

Một vector A có n phần tử, nếu mỗi phần tử $a[i]$ ($1 \leq i \leq n$) chiếm c từ máy thì nó sẽ được lưu trữ trong cn từ máy kế tiếp nhau (lưu trữ kế tiếp - sequential storage allocation).



L_0 : địa chỉ gốc (là địa chỉ của từ máy đầu tiên trong miền nhớ kế tiếp dùng để lưu trữ vector);

Địa chỉ a_i sẽ được tính bởi:

$$\text{Loc}(a_i) = L_0 + c \cdot (i-1);$$

$F(i) = c \cdot (i-1)$ gọi là hàm địa chỉ (address function).

Trong ngôn ngữ như ALGOL-60, Pascal, cận dưới có thể là số nguyên b nào đó (không nhất thiết phải là 1), khi đó:

$$\text{Loc}(a_i) = L_0 + c \cdot (i-b).$$

Tương tự: với mảng nhiều chiều, việc lưu trữ vẫn bằng vector lưu trữ như trên.

Ví dụ: Trong Fortran, ma trận 3 hàng 4 cột (a_{ij}) với $1 \leq i \leq 3$, $1 \leq j \leq 4$ sẽ được lưu trữ kế tiếp như sau:

a_{11}	a_{21}	a_{31}	a_{12}	a_{22}	a_{32}	a_{13}	a_{23}	a_{33}	a_{14}	a_{24}	a_{34}
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

Lưu trữ ở đây là theo cột (gọi là thứ tự ưu tiên cột - column-major order). Giả sử mỗi phần tử chiếm một từ máy thì địa chỉ của a_{ij} sẽ được tính bởi:

$$\text{Loc}(a_{ij}) = L_0 + (j-1)*3 + (i-1)$$

Tổng quát, đối với ma trận n hàng, m cột:

$$\text{Loc}(a_{ij}) = L_0 + (j-1)*n + (i-1)$$

Trong ngôn ngữ ALGOL-60, PL-1, Pascal, cách lưu trữ của ma trận lại theo thứ tự ưu tiên hàng (Row major order) - hết hàng này đến hàng khác.

Với ma trận n hàng, m cột, ta có:

$$\text{Loc}(a_{ij}) = L_0 + (i-1)*m + (j-1)$$

Tổng quát:

$$\text{Loc}(a_{ij}) = L_0 + (i-b_2)*(u_2-b_2+1) + (j-b_2)$$

Vì mỗi hàng có (u_2-b_2+1)

$$b_1 \leq i \leq u_1$$

$$b_2 \leq j \leq u_2$$

* Với mảng ba chiều

Ví dụ: xét mảng có $1 \leq i \leq 2$, $1 \leq j \leq 3$, $1 \leq k \leq 4$, được lưu trữ theo thứ tự ưu tiên hàng.

Công thức tính địa chỉ:

$$\text{Loc}(b_{ijk}) = L_0 + (i-1)*12 + (j-1)*4 + (k-1)$$

- Với ma trận n chiều A , phần tử là $A[s_1, s_2, \dots, s_n]$ trong đó: $b_i \leq s_i \leq u_i$, ($i=1, \dots, n$), ứng với thứ tự ưu tiên hàng, ta có:

$$\text{Loc}(A[s_1, s_2, \dots, s_n]) = L_0 + \sum_{i=1}^n p_i (S_i - b_i)$$

$$\text{Với } P_i = \prod_{k=i+1}^n (u_k - b_k + 1), p_n = 1$$

(tương tự cho thứ tự ưu tiên cột).

***Chú ý:**

Khi mảng được lưu trữ kế tiếp thì việc truy cập vào phần tử của mảng được thực hiện trực tiếp dựa vào địa chỉ tính được, nên tốc độ nhanh và đồng đều đối với mọi phần tử.

Mảng thể hiện mối quan hệ về cấu trúc giữa các phần tử dữ liệu ở nhiều ứng dụng. Song có nhiều trường hợp mảng cũng lộ ra những nhược điểm của nó.

Ví dụ: Giả sử xét bài toán tính đa thức của x, y : chẳng hạn cộng hai đa thức (hay trừ, nhân chia...)

$$\begin{array}{l} \text{Cộng} \quad (3x^2 - xy + y^2 + 2y - x) \\ \quad \quad (x^2 + 4xy - y^2 + 2x) \end{array}$$

$$\text{Kết quả là: } (4x^2 + 3xy + 2y + x)$$

Nếu ta hạn chế kích thước của mảng 5×5 thì số mũ cao nhất của x, y chỉ có thể là 4, nghĩa là chỉ xử lý được với đa thức bậc 4 của x, y .

Xét đa thức: $x^2 + 4xy + 2x - y^2$ được biểu diễn dưới dạng ma trận (hệ số của số hạng x_i, y_j sẽ được lưu trữ ở phần tử hàng i cột j của ma trận – giả sử cận dưới của ma trận là 0).

$$\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 2 & 4 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array}$$

Cách biểu diễn này đưa tới phép xử lý khá đơn giản, ví dụ cộng hai đa thức tương đương cộng hai ma trận...

**Nhược điểm:*

Số mũ của đa thức bị hạn chế bởi cấp của ma trận, do đó các đa thức được xử lý bị giới hạn trong phạm vi hẹp.

Ma trận biểu diễn nhiều phần tử 0 gây ra sự lãng phí bộ nhớ (gọi là ma trận thưa).

4.3. Lưu trữ kế tiếp của danh sách tuyến tính

Có thể dùng một vector lưu trữ (V_i) với $1 \leq i \leq m$ để lưu trữ một danh sách tuyến tính (a_1, a_2, \dots, a_n), a_i chứa ở v_i .

Do số phần tử của danh sách tuyến tính thường biến động nên việc lưu trữ chỉ đảm bảo được nếu biết được $m = \max(n)$ (n - kích thước danh sách), tuy nhiên gây lãng phí bộ nhớ vì có hiện tượng “giữ chỗ để đầy” mà không dùng tới.

Việc bổ sung hay loại bỏ một phần tử khỏi danh sách, gây phí tổn thời gian và vì có sự thay đổi phần tử trong danh sách.

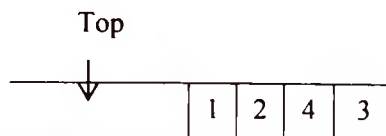
** Ưu điểm:* Cách lưu trữ này có tốc độ truy nhập rất rõ.

4.4. Stack hay danh sách kiểu ngăn xếp

4.4.1. Định nghĩa

Ngăn xếp (stack) là một danh sách tuyến tính đặc biệt mà phép bổ sung và phép loại bỏ luôn luôn thực hiện ở một đầu, gọi là đỉnh (TOP).

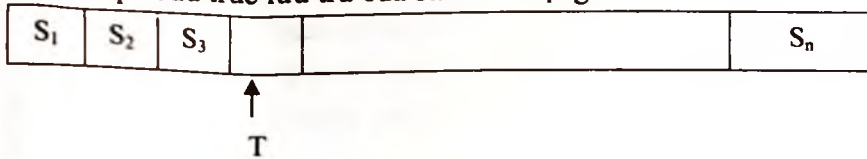
Ví dụ:



Nguyên tắc: “Vào sau ra trước” (last in first out - LIFO). Stack có thể rỗng hoặc bao gồm một số phần tử.

4.4.2. Lưu trữ stack bằng mảng

Có thể lưu trữ stack bằng một vector lưu trữ S , gồm n phần tử nhớ kế tiếp. Cấu trúc lưu trữ của stack có dạng:



T : con trỏ, trỏ tới đỉnh stack, biến đổi khi stack hoạt động.

T giảm 1: Khi loại bỏ một phần tử ra khỏi stack;

T tăng 1: Khi thêm một phần tử vào stack;

$T=0$: Khi stack rỗng.

Sau đây là giải thuật bổ sung và loại bỏ đối với stack.

Procedure Push (S, T, X)

{Giải thuật này thực hiện bổ sung phần tử X vào Stack lưu trữ bởi vector S có nhiều phần tử. T là con trỏ trỏ tới đỉnh Stack}

1- {Xét xem stack có tràn không?}

if $T \geq n$ then begin

write('Stack tran');

Return

end;

2- {Chuyển con trỏ}

$T := T + 1$;

3- {Bổ sung phần tử mới X }

$S[T] := X$;

4- Return

Function Pop (S, T)

(Hàm này thực hiện việc loại bỏ phần tử ở đỉnh stack S đang trỏ bởi T. Phần tử bị loại bỏ sẽ được thu nhận và đưa ra ngoài).

1- {Xét xem stack có cạn không? hiện tượng cạn xảy ra khi stack=rỗng, không còn phần tử nào để loại nữa, in ra thông báo cạn và kết thúc}

```
if  $T \leq 0$  then begin
    write (' Stack cạn');
    return
end;
```

2- {Chuyển con trỏ}

```
T:=T-1;
```

3- {Đưa phần tử bị loại ra}

```
Pop:=S[T+1];
```

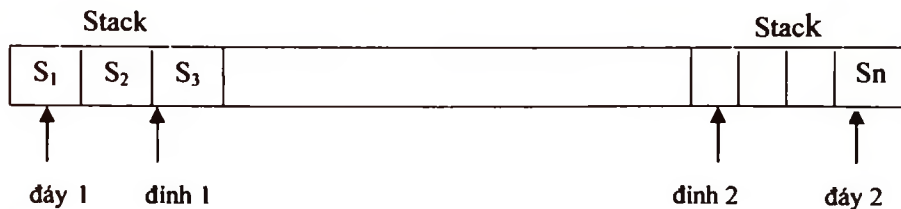
4- return

* Chú ý:

1) Đối với thủ tục hàm F nào đó, $F:=a$; return \Leftrightarrow return(a), như vậy ta có thể viết gọn bước 2), 3) và 4) như sau: return (S [T+1]).

2) Xử lý với nhiều stack: khi phải xử lý nhiều stack cùng một lúc, sẽ xảy ra tình trạng stack này bị tràn trong khi không gian lưu trữ cho stack khác vẫn còn trống (tràn cục bộ). Làm thế nào để khắc phục?

Nếu là hai stack: không quy định kích thước tối đa cho từng stack nữa mà không gian nhớ sẽ được dùng chung. Ta đặt hai stack ở hai đầu sao cho hướng phát triển của chúng ngược chiều nhau:



Do đó hiện tượng tràn xảy ra chỉ khi toàn bộ không gian nhớ dành cho nó đã được dùng hết.

Nhưng nếu số lượng stack từ ba trở lên thì phải có một biện pháp linh hoạt hơn, chẳng hạn: có ba stack, lúc đầu không gian nhớ có thể chia làm ba, nếu có một stack nào đó phát triển nhanh bị tràn trước mà stack khác vẫn còn chỗ trống thì phải dồn chỗ cho nó bằng cách:

+ Đẩy stack đứng sau sang phải;

+ Lùi stack đó sang trái.

Đáy của stack phải được phép di động và giải thuật loại, bổ sung phần tử đối với stack hoạt động theo kiểu này cũng phải thay đổi.

4.4.3. Vài ví dụ ứng dụng của stack

1) Đổi cơ số

- Ta biết:

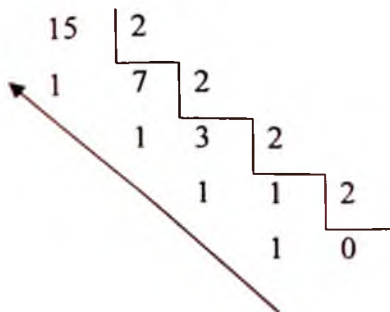
437 biểu diễn: $4 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0$ (cơ số 10)

Với số ở dạng nhị phân cũng tương tự, ví dụ:

Số 11010111 biểu diễn số:

$$1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = (215)_{10}$$

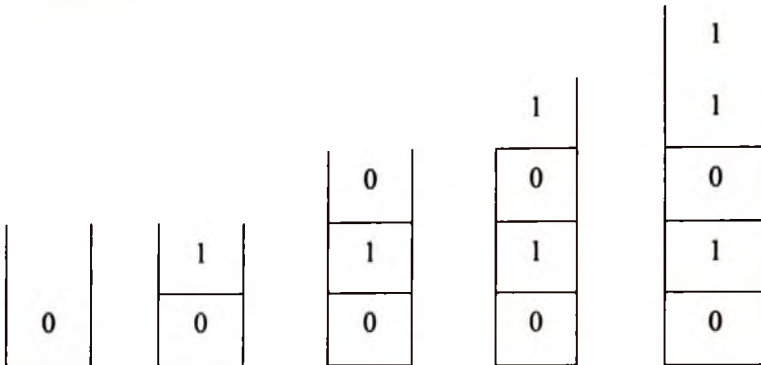
- Khi đổi một số từ thập phân sang nhị phân thì người ta dùng phép chia liên tiếp cho 2 và lấy số dư (là các chữ số nhị phân) theo chiều ngược lại.



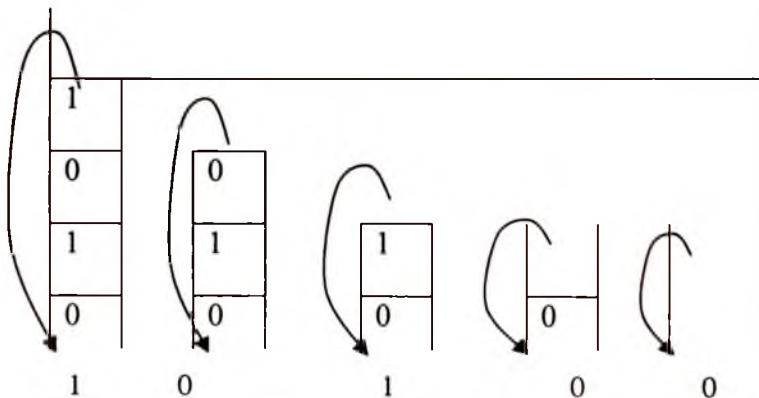
Các số dư được tạo sau lại được hiển thị trước, đây chính là cơ chế sắp xếp của stack. Để thực hiện biến đổi ta sẽ dùng một stack để lưu trữ số dư qua từng phép chia. Khi thực hiện phép chia thì nạp số dư vào stack, sau đó lấy chúng lần lượt từ stack ra.

Ví dụ: $(26)_{10} \rightarrow (01011)=(11010)_2$

PUSH:



POP:



Giải thuật chuyển đổi:

Program CHUYEN-DOI

{chuyển đổi một số từ cơ số 10 sang cơ số 2, hiển thị biểu diễn cơ số 2 này}

```

1) While N≠0 do
    Begin
        R:=N mod 2;
        PUSH(S,T,R);
        N:= N div 2;
    End;
2) While(S chưa rỗng) do
    Begin
        R:=POP(S,T);
        Write(R);
    End;
END;

```

2) Định giá biểu thức số học theo ký pháp nghịch đảo BaLan

Trong các ngôn ngữ lập trình, biểu thức số học được viết như dạng thông thường của toán học, nghĩa là theo ký pháp trung tố (infix notation).

Ví dụ: $5*(7-3)$, dấu ngoặc là không cần thiết

Dạng biểu thức số học theo ký pháp hậu tố (postfix notation), và tiền tố (prefix notation), được gọi là dạng ký pháp BaLan.

- Ở dạng hậu tố, các toán tử đi sau các toán hạng.

Ví dụ: $5*(7-3) \rightarrow 5\ 7\ 3\ -\ *$

- Ở dạng tiền tố, các toán tử đi trước toán hạng $+ 5 - 7\ 3$

Với các dạng ký pháp này các dấu ngoặc là không cần thiết

Ví dụ: $1\ 5 + 8\ 4\ 1\ -\ -\ * \Leftrightarrow (1+5)*(8-(4-1))$

Biểu thức rút gọn: $6\ 8\ 4\ 1\ -\ -\ *$

Lại lọc từ trái sang phải: Toán tử $-$ và ta xác định được hai toán hạng của nó là 4 và 1.

Thực hiện phép toán rút gọn $\Rightarrow 6\ 8\ 3\ -\ *$ tiếp tục ta có $6\ 5\ *$ và cuối cùng thực hiện phép toán nhân ta được kết quả là 30.

- Phương pháp định giá biểu thức hậu tố như trên, đòi hỏi phải lưu trữ các toán hạng cho tới khi một toán tử được đọc, đồng thời hai toán hạng cuối cùng phải được tìm và kết hợp với toán tử này. Sử dụng stack để lưu các toán hạng.

Mỗi lần đọc một toán tử: Hai giá trị được lấy ra từ stack để áp đặt toán tử đó lên chúng, kết quả lại được đẩy vào stack (*).

Giải thuật thực hiện (*)

```
Program DINH_GIA
{Sử dụng một stack S, trò bởi T, lúc đầu T=0}
1. Repeat
  Begin
    Đọc phần tử X tiếp theo trong biểu thức;
    if (X là toán hạng) then PUSH(S,T,X)
    else
      begin
        Y:=POP(S,T);
        Z:=POP(S,T);
        Tác động toán tử X vào Y, Z, gán kết
        quả cho W
        PUSH(S,T,W)
      end;
  End;
Until (gặp dấu kết thúc biểu thức);
2. R:=POP(S,T);
Write(R);
End;
```

4.4.4. Stack và việc cài đặt thủ tục đệ quy

Trong quá trình thực hiện thủ tục đệ quy, những tham số, biến cục bộ hay địa chỉ bảo lưu sau lại được khôi phục trước. Tính chất vào sau ra trước này dẫn đến việc cài đặt thủ tục đệ quy.

Mỗi khi có lời gọi đến chính nó, thì stack sẽ được nạp để bảo lưu các giá trị cần thiết. Còn khi thoát ra khỏi một mức thì phần tử ở đỉnh stack sẽ được lấy ra để khôi phục lại các giá trị cần thiết cho mức tiếp theo.

* Có thể tóm tắt các bước này như sau:

1. Mở đầu

Bảo lưu tham số, biến cục bộ và địa chỉ quay lui.

2. Thân

Nếu tiêu chuẩn cơ sở (base criterion) ứng với trường hợp suy biến đã đạt được thì thực hiện phần kết thúc và chuyển sang bước 3.

Nếu không thì thực hiện phân tích từng phần và chuyển sang bước 1.

3. Kết thúc

* Một vài chương trình thể hiện cách cài đặt thủ tục đệ quy dùng stack

Bài toán tính n!

Program FACTORIAL

{Sử dụng stack A, T trở vào đỉnh stack}

- Phần tử của A là bản ghi gồm có 2 trường:
- + Trường N: Ghi giá trị động của n ở mức hiện hành
- + RETADD: Ghi địa chỉ quay lui
- TEMREC là bản ghi trung chuyển gồm 2 trường:
- + PARA: ứng với N, ban đầu chứa giá trị N đã cho
- + ADDRESS ứng với trường RETADD: chứa địa chỉ ứng với lời gọi đến chương trình (ĐCC)
- Lúc đầu A rỗng $\rightarrow T=0$, Các giải thuật POP, PUSH.
- N(T): Cho giá trị của trường N của phần tử đang trở bởi T

1. {Bảo lưu giá trị của N và địa chỉ quay lui}

PUSH(A, T, TEMREC)

2. {tiêu chuẩn cơ sở đã đạt chưa?}

if N(T)=0 then

Begin

FACTORIAL:=1;

Go to bước 4;

```

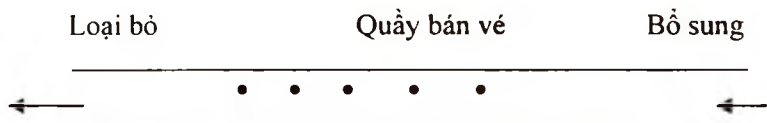
End;
Else
Begin
  PARA:=N(T)-1;
  ADDRESS:=Bước 3;
End;
3.{Tính N!}
  FACTORIAL:=N(T)* FACTORIAL(N-1);
4.{Khôi phục giá trị trước và địa chỉ quay lui}
  TEMREC:=POP(A, T);
Go to ADDRESS
5.end;

```

4.5. Queue hay danh sách kiểu hàng đợi

4.5.1. Định nghĩa

Hàng đợi là một danh sách tuyến tính trong đó phép bổ sung một phần tử mới được thực hiện ở một đầu và phép toán loại bỏ phần tử cũ được thực hiện ở đầu kia của hàng đợi.



Queue tương đương với danh sách kiểu: FIRST IN - FIRST OUT (FIFO).

4.5.2. Lưu trữ queue bằng mảng

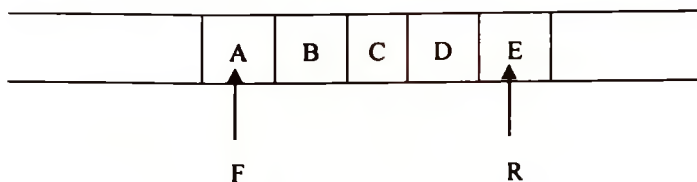
Có thể dùng một vector lưu trữ Q có n phần tử làm cầu trúc lưu trữ của queue. Để truy cập vào queue ta phải dùng hai biến con trỏ:

+ R: Trỏ tới lỗi sau;

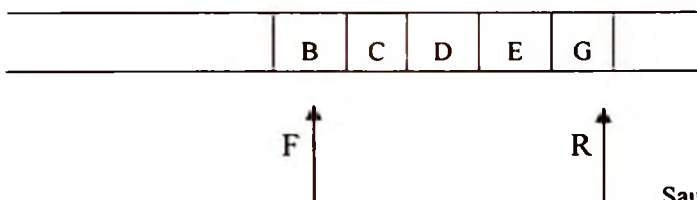
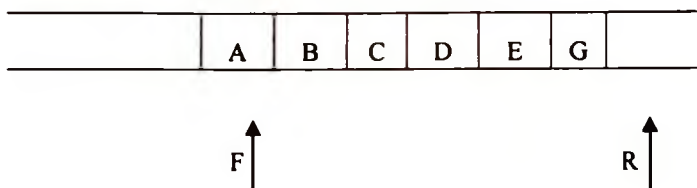
+ F Trỏ tới lỗi trước.

Queue rỗng: $R=F=0$

Nếu mỗi phần tử của queue được lưu trữ trong một từ máy thì R tăng 1, khi bổ sung một phần tử vào queue, F giảm 1 khi loại bỏ ra khỏi queue một phần tử.

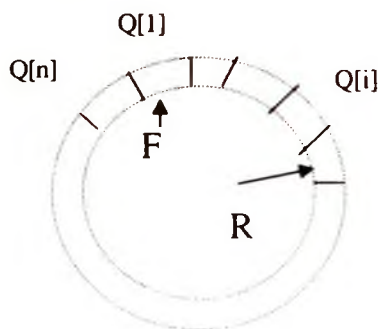


Sau khi bổ sung thêm G



Sau khi loại bỏ A

Với cách tổ chức này xuất hiện tình huống: các phần tử của queue sẽ di chuyển không gian nhớ khi thực hiện bổ sung hoặc loại bỏ. Khắc phục bằng cách coi không gian nhớ dành cho queue được tổ chức theo kiểu vòng tròn. Nghĩa là, với vector lưu trữ Q thì $Q[1]$ được coi như đứng sau $Q[n]$.



Tương ứng với cách tổ chức này, giải thuật bỏ sung và loại bỏ phần tử đối với queue sẽ như sau:

*** Giải thuật bỏ sung:**

Procedure CQINSERT(F, R, Q, n, XX)

{Con trỏ F, R lần lượt trỏ vào lổì trước và lổì sau của 1 queue được lưu trữ bởi vector n phần tử, theo kiểu vòng tròn. Bỏ sung phần tử X vào lổì sau của Queue đó. Thoạt đầu khi queue rỗng thì F=R=0}

1. {chỉnh lại con trỏ}

 if R=n then R:=1

 else R:=R+1;

2. {Kiểm tra tràn}

 if R=F then

 begin

 Write("Tràn");

 Return

 End;

3. {bỏ sung X vào}

 Q[R]:=X;

4. {Điều chỉnh con trỏ F khi bỏ sung lần đầu}

 if F=0 then F:=1;

5. return;

*** Giải thuật loại bỏ một phần tử khỏi queue:**

Function CQDELETE(F, R, Q, n)

{queue kiểu vòng tròn, F, R trỏ tới lổì trước và lổì sau của queue có n phần tử, hàm thực hiện loại một phần tử ở lổì trước của queue và hiển thị nội dung của phần tử đó ra màn hình}

1. {Kiểm tra cạn}

 if f=0 then

```

begin
    Write("Cạn");
    Return(0);
End;
2. {Loại bỏ}
   Y:=Q[F];
3. {Xử lý trường hợp queue trở thành rỗng sau phép
   loại bỏ}
   if F=R then {queue chỉ có một phần tử nếu F=R}
   Begin
       F:=R:=0;
       Return(Y);
   End;
4. {Chỉnh lại con trỏ F sau phép loại bỏ}
   if F=n then F:=1
   else F:=F+1
5. return (Y);

```



Chương 5 - DANH SÁCH MỐC NỐI

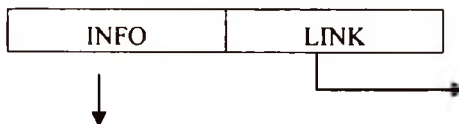
Lưu trữ kế tiếp đối với danh sách tuyến tính bộc lộ nhiều nhược điểm khi thực hiện thường xuyên các phép bổ sung hoặc loại bỏ phần tử. Giải pháp khắc phục là sử dụng con trỏ hoặc mối nối để tổ chức danh sách tuyến tính.

5.1. Danh sách nối đơn

5.1.1. Nguyên tắc

Mỗi phần tử của danh sách được lưu trữ trong một phần tử nhớ gọi là nút:

- 
- Mỗi nút bao gồm một số từ máy kế tiếp nhau;
 - Các nút có thể nằm ở vị trí bất kỳ trong bộ nhớ;
 - Mỗi nút chứa những thông tin ứng với một phần tử và địa chỉ phần tử đứng sau danh sách:



Chứa thông tin

Chứa địa chỉ
móc nối tiếp theo

Nút cuối cùng: Không có nút đứng sau, LINK chứa “địa chỉ đặc biệt” dùng để đánh dấu nút kết thúc danh sách, ta gọi là mỗi nối không, ký hiệu NULL.

Muốn truy nhập vào danh sách ta phải truy nhập được vào phần từ đầu tiên, nghĩa là sử dụng con trỏ n trở tới nút đầu tiên.

Hình ảnh danh sách nối đơn:



: Chỉ mỗi nối không

Quy ước: Danh sách rỗng: $L = \text{NULL}$

Để tổ chức danh sách móc nối, những khả năng sau đây phải có:

- + Tồn tại phương tiện để chia bộ nhớ ra thành các nút, ở mỗi nút có thể truy nhập vào từng trường (trường có kích thước ấn định);
- + Tồn tại một cơ chế xác định được một nút đang sử dụng (nút bận), không sử dụng (nút trống);
- + Tồn tại một cơ chế như một kho chứa chỗ trống, để cung cấp các nút trống (list available space).

Để tiện trình bày ta gọi cơ cấu trên là “danh sách chỗ trống”, qui ước:

$X \leftarrow \text{AVAIL}$: Cấp phát một nút có địa chỉ X, lấy từ danh sách chỗ trống;

$X \Rightarrow \text{AVAIL}$: Thu hồi một nút có địa chỉ X trả về danh sách chỗ trống.

Ban đầu bản thân danh sách chỗ trống là một stack nối đơn, AVAIL là con trỏ, trở tới nút đầu danh sách, sau khi cấp phát và thu hồi ô nhớ \Rightarrow các nút trong danh sách không còn kế tiếp nhau nữa.

5.1.2. Một số phép toán

Xét một số giải thuật thể hiện một số phép toán thường tác động vào danh sách nối đơn.

- Với một nút có địa chỉ p (được trỏ bởi P) thì:

+ INFO(P): Chỉ trường INFO - chứa thông tin của nút P;

+ LINK(P): Chỉ trường địa chỉ của nút kế tiếp nút P.

1. Bổ sung một phần tử vào danh sách nối đơn

Procedure INSERT(L, M, X)

{L: con trỏ, trỏ tới nút đầu tiên của danh sách

M: Con trỏ trỏ tới một nút đang có trong danh sách

Giải thuật: Bổ sung vào sau nút trỏ bởi M một nút có trường INFO có giá trị là X}

1) {Tạo nút mới}

New ← AVAIL;

INFO(New) := X

2) {Thực hiện bổ sung, nếu danh sách rỗng → bổ sung vào danh sách thành nút đầu tiên, else bổ sung vào sau M}

if L=null then

begin

L := New;

LINK(L) := null;

End

Else

Begin

LINK(New) = LINK(M) ;

LINK(M) := New;

End

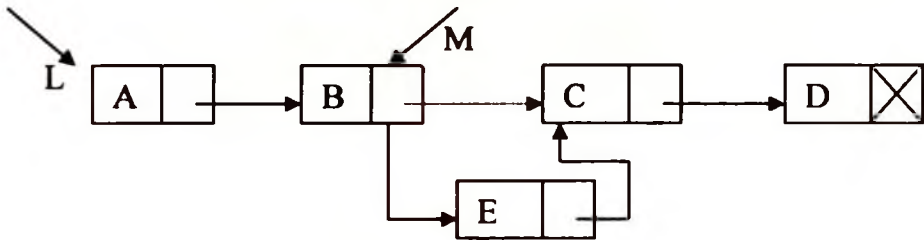
3) return

Minh họa bởi hình ảnh sau:

a) Nếu $L = \text{null}$ thì sau phép bổ sung ta có:



b) Nếu nút $L \neq \text{null}$



2. Loại bỏ một phần tử ra khỏi danh sách nối đơn

Procedure DELETE(L, M)

{cho danh sách nối đơn trỏ bởi L . Giải thuật này thực hiện loại bỏ nút trỏ bởi M ra khỏi danh sách đó}

1) {trường hợp danh sách rỗng}

if $L = \text{null}$ then

Begin

Write("danh sách rỗng");

Return;

End;

2) {Trường hợp nút trỏ bởi M là nút đầu tiên trong danh sách}

if $M = L$ then

Begin

L:=LINK(M);

M=>AVAIL;

Return;

End;

3) {Tìm đến nút đứng trước nút trỏ bởi M}

P:=L;

While LINK(P) ≠ M do

P:=LINK(P);

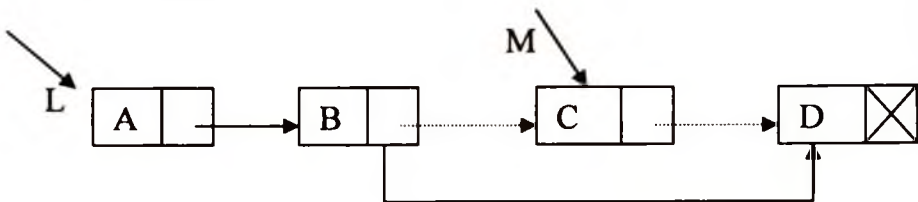
4) {Loại bỏ M}

LINK(P):=LINK(M);

5) {Đưa nút bị loại về danh sách chỗ trống}

M:=AVAIL;

6) return



3. Ghép hai danh sách nối đơn

Procedure COMBINE(P, Q)

{Ghép hai danh sách nối đơn lần lượt trỏ bởi P, Q thành một danh sách mới và cho P trỏ tới nó}

1) {Trường hợp Q rỗng}

if Q= null then return

2) {p Rỗng}

if p=null then

Begin

P:=Q;

Return;

End

3) {Tìm đến nút cuối danh sách P}

```

P1 := P;
While LINK(P1) ≠ null do P1 = LINK(P1)
4) {Ghép}
   LINK(P1) := Q;
5) return

```

***Chú ý:**

Cách tổ chức móc nối như trên rõ ràng tỏ ra thích hợp, tuy nhiên cách cài đặt cũng có những nhược điểm nhất định, như:

- + Chỉ có phần tử đầu tiên được truy cập trực tiếp, các phần tử khác chỉ được truy nhập sau khi truy nhập phần tử trước nó;
- + Tốn bộ nhớ vì có thêm trường LINK ở mỗi nút để lưu trữ địa chỉ tiếp theo.

5.2. Danh sách nối vòng

Là một cải tiến của danh sách nối đơn. Trường LINK của nút cuối cùng trong danh sách nối đơn chứa địa chỉ của nút đầu tiên của danh sách. Hình ảnh của nó như sau:



*** Ưu điểm:**

Giúp cho việc truy nhập vào các nút được linh hoạt hơn (vì nút nào cũng có thể coi là nút đầu tiên và con trỏ L trỏ tới nút nào cũng được).

Phép ghép, tách cũng có những thuận lợi nhất định.

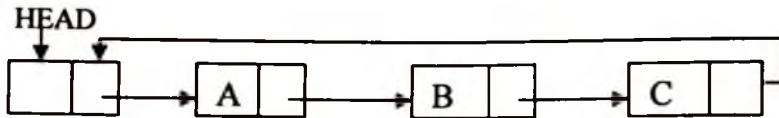
*** Nhược điểm:**

Trong xử lý, nếu không cẩn thận có thể dẫn đến một chu trình không kết thúc (vì không biết được chỗ kết thúc danh sách).

*** Khắc phục:**

Đưa thêm một nút đặc biệt trở tới phần tử đầu tiên của danh sách con trỏ HEAD bây giờ trở tới nút đầu danh sách này, nút này có trường INFO không chứa dữ liệu của phần tử nào.

Việc thêm nút đầu danh sách khiến danh sách không bao giờ rỗng, ta có hình ảnh



$LINK(HEAD)=HEAD$

* Giải thuật bổ sung một nút vào thành nút đầu tiên của danh sách nối vòng

$NEW \leq AVAIL;$

$INFO(New) := LINK(HEAD);$

$LINK(HEAD) := New;$

***Chú ý:**

Việc dùng nút đầu danh sách có những thuận lợi nhất định, vì vậy ngay đối với danh sách nối đơn người ta cũng dùng.

5.3. Danh sách nối kép

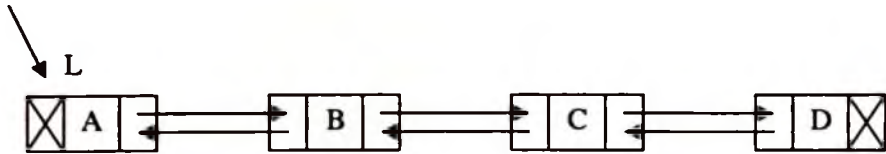
Quy cách một nút:

LPTR: là con trỏ trái, trỏ tới nút đứng trước;

RPTR: là con trỏ phải, trỏ tới nút đứng sau;

INFO: giống các danh sách trước.

Danh sách móc nối có dạng:



Ở đây: LPTR của nút cực trái, RPTR của nút cực phải phải là null.

- Để truy nhập được vào các phần tử của danh sách ta dùng hai con trỏ L, R lần lượt trỏ tới nút cực trái, phải.

- Danh sách rỗng: $L=R=null$

* Một số phép toán:

Procedure DOUBLE_INSERT(L,R,M,X)

{L, R trỏ tới nút cực trái, phải của danh sách
nối kép

M: trỏ tới một nút trong danh sách

Giải thuật: Bổ sung một nút mới mà dữ liệu chính
ở X vào trước nút trỏ bởi M}

1) {Tạo nút mới}

New ← AVAIL

INFO(New) := X;

2) {trường hợp danh sách rỗng}

if R=null then

begin

LPTR(New) := null;

RPTR(New) := null;

L := R := New;

Return

End;


```

3) {bổ sung vào giữa}
LPTR(New):=LPTR(M);
RPTR(New):=M;
LPTR(M):=New;
RPTR(LPTR(New)):=New;
Return
Procedure DOUBLE_REMOVE(L,R,M)
{L, R trỏ tới hai con trỏ trái, phải của danh
sách nối kép, loại bỏ nút trỏ bởi M ra khỏi danh
sách}
1) {trường hợp danh sách rỗng}
if R=NULL then
    Begin
        Write("danh sách rỗng");
        Return;
    End
2) {Loại bỏ}
case
L=R: {danh sách chỉ có một nút, M trỏ tới nút đó}
    L:=R:=null;
M=L: {nút cực trái bị loại}
    L:=RPTR(L);
LPTR(L):=null;
M=R: {nút cực phải bị loại}
    R:=LPTR(R);
RPTR(R):=null
Else: RPTR(LPTR(M)):=RPTR(M);
LPTR(RPTR(M)):=LPTR(M);
End case;
3) M⇒AVAIL;
Return;

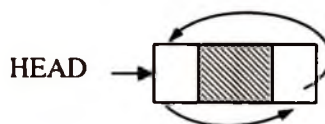
```

*** Chú ý**

Nếu ta đưa nút đầu danh sách vào thì trong hai giải thuật nêu trên sẽ không còn có tình huống ứng với nút cực trái và nút cực phải nữa.

Để đối xứng, khi đưa nút đầu danh sách vào người ta tổ chức danh sách theo kiểu lai nối vòng có dạng:

Trường hợp danh sách rỗng:



$RPTR(New) := HEAD;$

$LPRT(New) := HEAD;$

HEAD đóng vai trò vừa là nút cực trái, vừa là nút cực phải, với danh sách kiểu này giải thuật DOUBLE_REMOVE chỉ còn một bước:

$RPTR(LPTR(M)) := RPTR(M);$

$LPTR(RPTR(M)) := LPRT(M);$

$M := AVAIL$

5.4. Ví dụ áp dụng

Xét bài toán cộng hai đa thức của x, được tổ chức dưới dạng danh sách móc nối.

5.4.1. Biểu diễn đa thức

Đa thức được biểu diễn dưới dạng danh sách nối đơn, mỗi nút có quy cách

COEF	EXP	LINK
------	-----	------

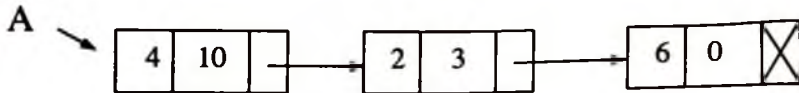
- Trường COEF chứa hệ số $\neq 0$ của 1 số hạng trong đa thức;

- Trường EXP chứa số mũ tương ứng ;
- Trường LINK chứa mỗi nối tới nút tiếp theo.

Ví dụ:

Biểu diễn đa thức:

$$A(x) = 4 \cdot x^{10} - 2 \cdot x^3 + 6:$$



$$B(x) = -6 \cdot x^{10} + 8 \cdot x^6 + 2 \cdot x^3 + 5 \cdot x \text{ có dạng:}$$



Phép cộng 2 đa thức sẽ cho

$$C(x) = A(x) + B(x) = -2 \cdot x^{10} + 8 \cdot x^6 + 5 \cdot x + 5$$

⇒ biểu diễn:



5.4.2. Giải thuật

Để cộng $A(x)$, $B(x)$ phải dùng hai biến trỏ P, Q để duyệt qua hai danh sách $A(x)$, $B(x)$ trong quá trình này sẽ có tình huống sau:

- + $EXP(P) = EXP(Q)$: cộng giá trị COEF ở hai nút đó, nếu tổng khác 0 thì tạo nút mới lưu giá trị tổng đó, gắn vào $C(x)$;
- + $EXP(P) > EXP(Q)$ (hoặc ngược lại thì cũng tương tự);
- + Nếu một danh sách kết thúc trước: phần còn lại của danh sách khi sẽ được sao chép gắn dần vào danh sách của $C(x)$.

Điều này được thể hiện bằng thủ tục ATTACH

Procedure ATTACH(H,M,d)

{COEF chứa giá trị H, EXP chứa M, gắn nút mới vào sau nút trở bởi d}

New←AVAIL;

COEF(New):=H;

EXP(New):=M;

LINK(d):=New;

d:=New;{nút mới này lại trở thành đuôi}

Return;

{Thủ tục cộng 2 đa thức}

Procedure PADD(A,B,C)

1) p:=A;

q:=B;

2) C←AVAIL;

d:=c; {lấy một nút mới làm nút "đuôi giả" để ngay từ nút đầu có thể sử dụng được thủ tục ATTACH, sau này sẽ bỏ đi}

3) While(p≠null) and (q≠null) do

Case

EXP(P)=EXP(q):

x:=COEF(p)+COEF(q);

If x≠0 then

Call ATTACH(x,EXP(p),d);

P:=LINK(p);

Q:=LINK(Q);

EXP(P)<EXP(Q):

Call ATTACH(COEF(Q),EXP(Q),d);

Q:=LINK(Q);

Else Call ATTACH(COEF(P),EXP(P),d);

End case P:=LINK(P);

4){danh sách ứng với B(x) đã hết}

```

While(P≠null) do
Begin
    Call ATTACH(COEF(P),EXP(P),d);
    P:=LINK(P);
End;
5) {danh sách ứng với A(x) đã hết}
While(Q≠Null) do
Begin
    ATTACH(COEF(Q),EXP(Q),d);
    Q:=LINK(Q);
End;
6) {Kết thúc danh sách C}
LINK(d):=null;
7) {Loại bỏ nút đóng vai trò đuôi giả lúc đầu}
t:=c; c:=LINK(c); c⇒AVAIL;
8) return;

```

* Giải thuật khác gì? Khi tổ chức đa thức dưới dạng một danh sách nối vòng có “nút đầu danh sách” lần lượt trở bởi A, B, C, khi đó PADD có gì thay đổi?

5.5. Stack hay queue móc nối

Stack: Việc truy nhập thực hiện ở một đầu (đỉnh).

Danh sách nối đơn trở bởi L thì có thể coi L như con trở, trở tới đỉnh stack.

+ Bổ sung một phần tử vào stack chính là bổ sung vào đầu danh sách;

+ Loại bỏ một phần tử khỏi stack chính là loại bỏ phần tử ở đầu danh sách.

- Với stack móc nối, khi bỏ sung ta không cần kiểm tra hiện tượng tràn như stack kế tiếp (vì stack móc nối không hề bị hạn chế về kích thước; chỉ phụ thuộc vào giới hạn của bộ nhớ toàn phần (khi danh sách chỗ trống đã cạn hết).

* Với queue: Phép loại bỏ giống stack, nhưng bỏ sung phải thực hiện vào đuôi, phải lần tìm đến nút cuối cùng. Nếu tổ chức queue như một danh sách nối kép có con trỏ L, R thì không phải tìm đuôi nữa.

Chương 6 - CÂY

6.1. Định nghĩa và các khái niệm

Định nghĩa: *Cây là một tập hữu hạn các nút trong đó có một nút đặc biệt gọi là gốc. Giữa các nút có quan hệ phân cấp.*

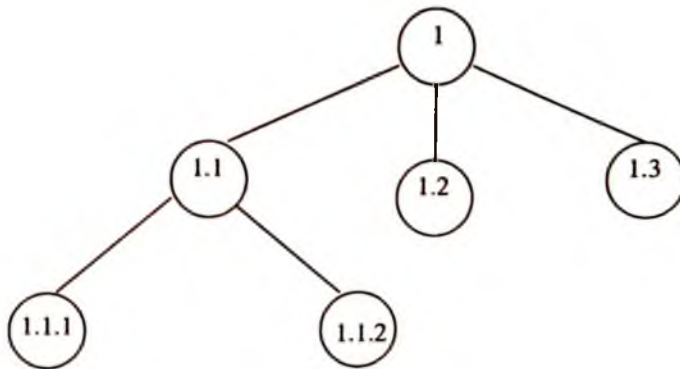
- Quan hệ cha con và tuân theo một cách đệ quy như sau:

1. Một nút là một cây, nút đó đồng thời là gốc của cây

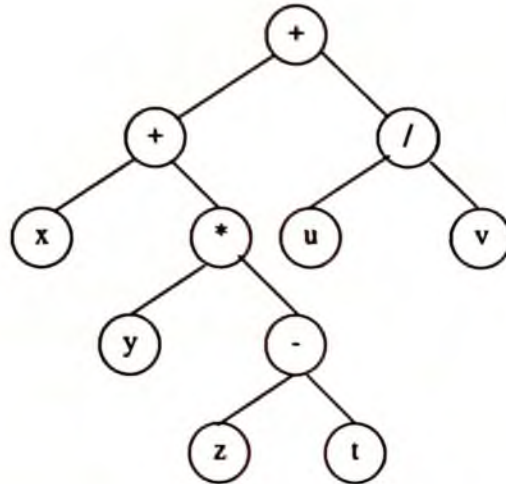
2. Nếu n là một nút và T_1, T_2, \dots, T_k là các cây con với các gốc tương ứng là n_1, n_2, \dots, n_k thì ta tạo ra các cây mới T bằng cách cho nút n là cha của các nút n_1, n_2, \dots, n_k . Ta gọi các T_i là các cây con của cây T .

Để thuận tiện người ta cho phép tồn tại cây rỗng (không có nút nào).

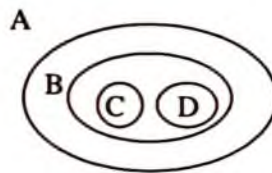
Ví dụ:



Biểu diễn biểu thức số học $x+y*(z-1)+u/v$



Biểu diễn các tập bao nhau



*** Đối với cây:**

Một số khái niệm của cây

- Số các con của một nút gọi là bậc của nút đó. Nếu nút mà có bậc bằng 0 thì gọi là lá. Nút không có lá là nút nhánh;
- Bậc cao nhất của nút có trên cây gọi là bậc của cây;
- Gốc của cây có mức là 1, con của nó có mức là $i+1$, con của nút có trên cây gọi là cấp của cây;
- Nếu cây có n nút (cây nhị phân) thì số mức sẽ là: $\lceil \log_2 n \rceil + 1$ (lấy phần nguyên của $\log_2 n$);
- Chiều cao (chiều sâu) của một cây là số mức lớn nhất của nút có trên cây;

- Nếu thứ tự các cây con của một nút được coi trọng thì cây đơn xét được gọi là cây có thứ tự.

Thông thường ta đánh thứ tự từ trái qua phải. Nhiều cây độc lập với nhau gọi là rừng.

6.2. Cây nhị phân

6.2.1. Định nghĩa và các tính chất

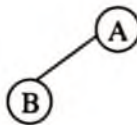
Định nghĩa cây nhị phân: Là một cây, mọi nút trên cây chỉ có tối đa hai nút con. Đối với cây con của mỗi nút người ta cũng phân biệt cây con trái và cây con phải.

Cây nhị phân là cây có thứ tự.

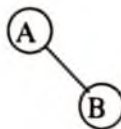
Tính chất: Cây nhị phân suy biến có dạng của một danh sách tuyến tính

Ví dụ:

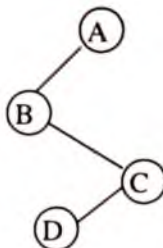
Cây lệch trái



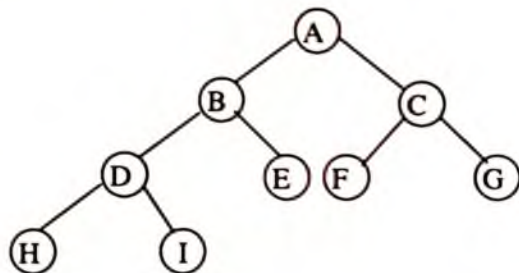
Cây lệch phải



Cây Zic-zắc



Cây nhị phân hoàn chỉnh (cây nhị phân đầy đủ).



Cây nhị phân hoàn chỉnh (complete binary tree): Các nút trừ mức cuối cùng đều đạt tối đa.

** Nhận xét:*

Trong các cây có cùng số lượng nút:

+ Cây nhị phân suy biến có chiều cao lớn nhất;

+ Cây đầy đủ có chiều cao nhỏ nhất.

Đối với cây nhị phân đầy đủ cần chú ý tới một số tính chất sau:

Bổ đề:

1) Số lượng tối đa các nút ở mức i trên cây nhị phân là 2^{i-1} ($i \geq 1$).

2) Số lượng tối đa các nút trên cây có chiều cao h là $2^h - 1$ ($h \geq 1$).

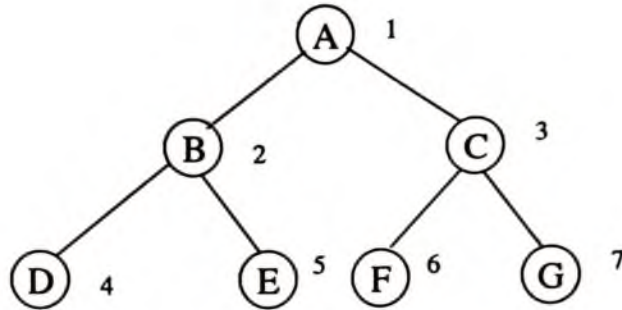
6.2.2. Biểu diễn cây nhị phân

a) Lưu trữ kế tiếp

Ta có thể đánh số cho các nút trên cây từ mức 1 đến mức cuối, mỗi mức được đánh từ trái sang phải. Cách này chỉ dùng cho cây đầy đủ nghĩa là không quá một nút chỉ có một con. Ta thấy:

Con của nút thứ i là các nút thứ $2*i$ và $2*i + 1$

Cha của nút thứ j là $\lfloor j/2 \rfloor$



Ta có thể lưu trữ cây bằng một vector V theo nguyên tắc: Nút thứ i được lưu trữ ở $V[i]$. Đó chính là các lưu trữ kế tiếp đối với cây nhị phân.

+ Cây nhị phân đầy đủ:

A	B	C	D	E	F	G
---	---	---	---	---	---	---

$V[1] \quad V[2] \quad V[3] \quad V[4] \quad V[5] \quad V[6] \quad V[7]$

+ Cây nhị phân không đầy đủ:

A	B	ϕ	C	ϕ	ϕ	ϕ	D	ϕ	ϕ	E	ϕ
---	---	--------	---	--------	--------	--------	---	--------	--------	---	--------

**Nhược điểm:*

Với cách lưu trữ này sẽ gây lãng phí (do có nhiều phần tử bỏ trống).

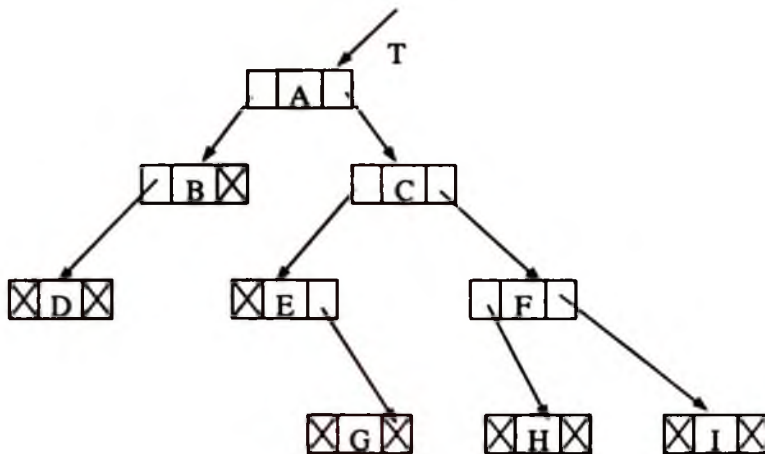
b) *Lưu trữ móc nối: Mỗi nút ứng với một bản ghi (phần tử nhớ)*

Quy cách mỗi nút:

- INFO: Chứa những thông tin (dữ liệu) của nút;
- LPTR: Trỏ tới cây con trái của nút đó;
- RPTR: Trỏ tới con phải của nút đó.

LPTR	INFO	RPTR
------	------	------

Ví dụ: Xét cây



- Để truy nhập vào cây: Sử dụng con trỏ T, trỏ tới nút gốc của cây.

Nếu cây nhị phân rỗng ta có $T = \text{null}$.

Nhận xét: Từ nút cha có thể truy nhập dễ dàng, ngược lại thì không làm được.

6.2.3. Phép duyệt cây nhị phân

- Duyệt cây: Là phép thăm các nút trên cây, sao cho mỗi nút chỉ được thăm một lần.

Có ba cách duyệt cây:

a) Duyệt theo thứ tự trước (preorder traversal)

- Thăm gốc;
- Duyệt cây con trái theo thứ tự trước;
- Duyệt cây con phải theo thứ tự trước.

b) Duyệt theo thứ tự giữa (inorder traversal)

- Duyệt cây con trái theo thứ tự giữa;
- Thăm gốc;

- Duyệt cây con phải theo thứ tự giữa.

c) Duyệt theo thứ tự sau (post order traversal)

- Duyệt cây con trái theo thứ tự sau;

- Duyệt cây con phải theo thứ tự sau;

- Duyệt gốc.

Chú ý: Nếu cây rỗng thì thăm có nghĩa là không làm gì.

Ví dụ: Cho cây như hình vẽ, hãy thực hiện phép duyệt cây.

Thăm theo thứ tự trước

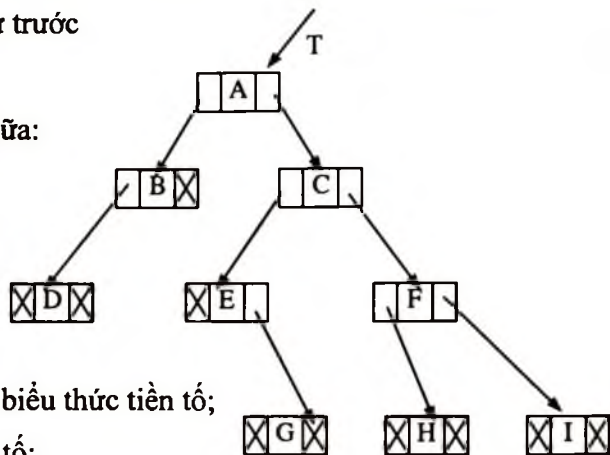
ABDECFGH

b) Theo thứ tự giữa:

DBFAEGHC

Theo thứ tự sau:

DEBGHFCA



***Nhận xét:**

a) Chính là dạng biểu thức tiền tố;

b) Là dạng trung tố;

c) Là dạng hậu tố.

6.2.4. Cây nhị phân nối vòng

Có thể thấy rằng, trên cây nhị phân lưu trữ móc nối, có n nút thì có tới $n+1$ mối nối không. Ta tận dụng các mối nối này để tạo điều kiện thuận lợi cho phép duyệt cây.

Một trong những cách tận dụng: thay “mối nối không” bằng mối nối trở tới một nút quy định, để tạo điều kiện thuận lợi cho phép duyệt cây, loại mối nối này gọi là “mối nối vòng”. Dạng biểu diễn cây nhị phân như thế gọi là “cây nhị phân nối vòng”.

Nhằm mục đích cho phép duyệt theo thứ tự giữa được thuận lợi, Perlis đưa ra quy ước:

Với P là một nút trên cây, nếu:

+ LPTR(P)=null thì thay LPTR(p) = ^+P ;

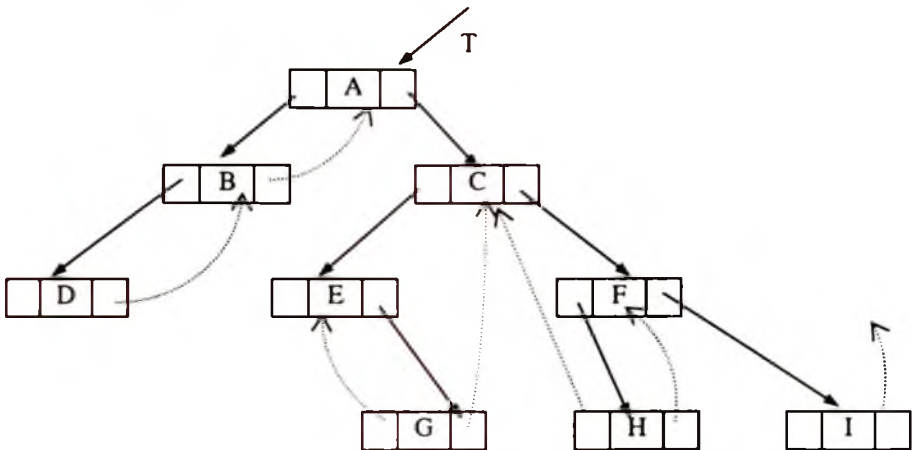
+ RPTR(P)=null thì thay RPTR (P) = P^+ .

Trong đó: ^+P : trỏ tới nút đứng trước P, trong thứ tự giữa;

P^+ : Chỉ tới nút sau P theo thứ tự giữa.

Mỗi nối vòng, được biểu diễn bởi $\cdots\cdots\cdots\rightarrow$

Quy cách của mỗi nút có dạng:



Nhận xét: Hai trường bit LBIT và RBIT dùng để đánh dấu hai loại mỗi nối đó.

nếu: LPTR(P) \neq null thì LBIT(P)=0;

RPTR(P)=null thì LBIT(P)=1 (ứng với mỗi nối vòng).

Với RBIT cũng tương tự.

Nghĩa là qui cách một nút

LBIT	LPTR	INFO	RPTR	RBIT
------	------	------	------	------

Ví dụ hình trên:

Mỗi nối vòng trái của D: gọi là nút cực trái của cây;

Mỗi nối vòng phải của nút I: gọi là nút cực phải của cây T.

⇒ Để biểu diễn nhất quán đối với mọi nút, mà cũng không gây ra điều gì phức tạp, người ta quy ước đưa thêm vào nút “đầu cây”. T được gọi là cây con trái của nút đầu cây ấy, và mỗi nối phải của nút đầu cây ấy thì luôn trở tới chính nó.

- HEAD: trở tới nút đầu cây;

- Cây rỗng thì nút đầu cây có dạng:

$RPTR(HEAD) = HEAD$

$RBIT(HEAD)=0$

$LBIT(HEAD)=HEAD$

$LBIT(HEAD)=1$

(Với cấu trúc cây nối vòng như thế này, giải thuật tìm một nút đứng trước một nút P, hoặc sau một nút P trong thứ tự giữa trở nên khá đơn giản, và việc đưa thêm nút đầu cây với quy ước như trên là hoàn toàn phù hợp).

6.3. Cây tổng quát

6.3.1. Biểu diễn cây tổng quát

- Đối với cây nhị phân cấp m nào đó, có thể sử dụng cách biểu diễn móc nối tương tự cây nhị phân.

Như vậy, mỗi nút phải dành ra m trường mỗi nối để trỏ các con của nút đó. Nhược điểm là số “mỗi nối không” sẽ rất nhiều.

Nếu cây có n nút sẽ có tới $n(m-1)+1$ “mỗi nối không” trong số $m*n$ mỗi nối.

- Còn nếu tùy theo số con của từng nút mà định ra mỗi nối, nghĩa là dùng nút có kích thước biến đổi thì sự tiết kiệm không gian nhớ được trả giá bằng những phức tạp của quá trình xử lý.

⇒ Cách khác hiện thực là biểu diễn cây tổng quát bằng cây nhị phân (cây nhị phân tương đương).

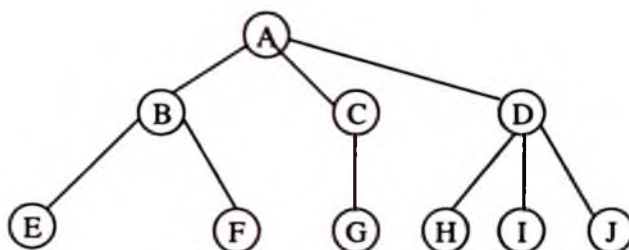
Mỗi nút có quy cách:

CHILD	INFO	SIBLING
-------	------	---------

CHILD: trỏ tới nút con cực trái;

SIBLING: trỏ tới nút em kế phải.

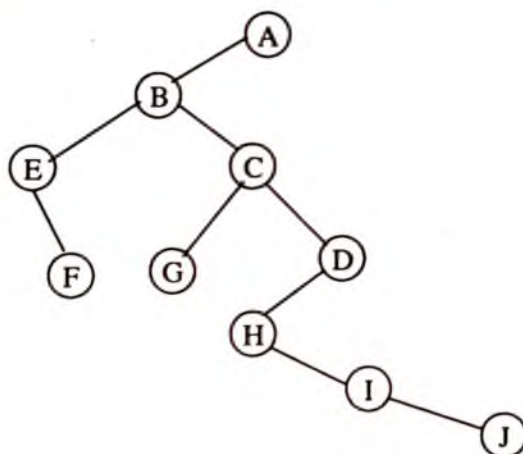
Ví dụ: Xét cây



+ Với nút B: con cực trái là E, em kế cận phải là C;

+ Với nút D: con cực trái là H, em kế cận phải không có.

⇒ Biểu diễn cây tổng quát bằng cây nhị phân tương đương.

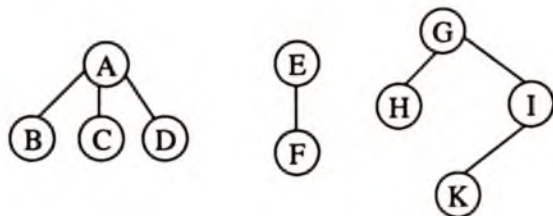


***Nhận xét:**

Nội phải của gốc cây bao giờ cũng là “môi nổi không” (vì gốc không có em kế cận phải).

Như vậy có thể biểu diễn rừng bằng cây nhị phân tương đương (trường hợp một cung thì coi như rừng đặc biệt).

Ví dụ:



Nếu T_1, T_2, \dots, T_n là một rừng thì cây nhị phân tương đương biểu diễn rừng đó, ký hiệu:

$B(T_1, T_2, \dots, T_n)$ sẽ là cây.

1) Rỗng nếu $n=0$.

2) Có gốc là gốc T_1 , cây con trái là $B(T_{11}, T_{12}, \dots, T_{1m})$ cây con phải là $B(T_2, T_3, \dots, T_n)$.

6.3.2. Phép duyệt cây tổng quát

- Duyệt theo thứ tự trước

a) Nếu T rỗng thì không làm gì.

b) Nếu T không rỗng:

+ Thăm gốc của T ;

+ Duyệt con thứ nhất T_1 là gốc của T theo thứ tự trước;

+ Duyệt cây con còn lại T_2, T_3, \dots, T_k của T theo thứ tự trước.

- Duyệt theo thứ tự giữa:

a) Nếu T rỗng thì không làm gì;

b) Nếu T không rỗng thì:

- + Duyệt cây con thứ nhất T_1 của gốc T theo thứ tự giữa;
- + Thăm gốc của T ;
- + Duyệt các cây con còn lại T_2, T_3, \dots, T_n theo thứ tự giữa.

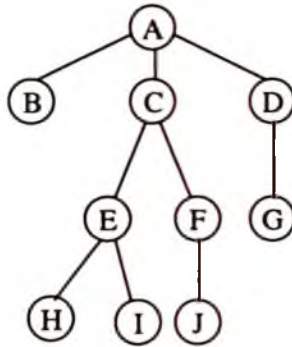
- Duyệt theo thứ tự sau

a) Nếu T rỗng thì không làm gì

b) Nếu T không rỗng thì:

- + Duyệt cây con 1 của T theo thứ tự sau;
- + Duyệt cây con còn lại T_2, T_3, \dots, T_k của gốc T theo thứ tự sau;
- + Thăm gốc của T .

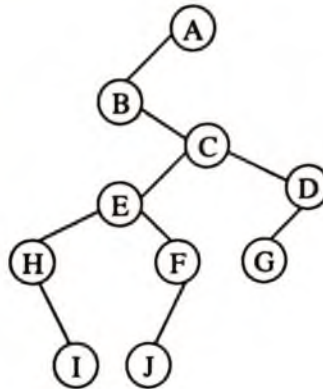
Ví dụ tổng quát:



Dãy tên các nút được thăm sẽ là:

- Thứ tự trước: ABCEHIFJDG;
- Thứ tự giữa: BAHEICJFGD;
- Thứ tự sau: BHIEJFCGD.

Ta có cây nhị phân tương đương:



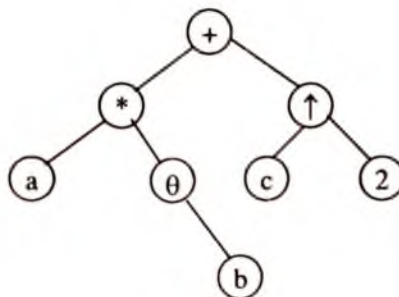
**Nhận xét:*

- Với thứ tự trước, phép duyệt cây tổng quát và cây nhị phân đều cho một dãy giống nhau;
 - Kết quả của phép duyệt sau của cây tổng quát giống phép duyệt theo thứ tự giữa của cây nhị phân tương đương với nó;
 - Phép duyệt cây tổng quát theo thứ tự giữa cho dãy tên không giống bất kỳ dãy nào đối với cây nhị phân tương đương.
- Đối với cây tổng quát người ta chỉ nêu hai phép duyệt: duyệt theo thứ tự trước và theo thứ tự sau.

6.4. Áp dụng

Cây biểu diễn biểu thức: Biểu thức số học với các phép toán hai ngôi: +, -, *, /, có thể biểu diễn dưới dạng một cây nhị phân.

Ví dụ:



- Tạo cây nhị phân biểu diễn biểu thức đó, quy cách của mỗi nút:

LPTR	TYPE	RPTR
------	------	------

+ Nếu không là lá: TYPE chỉ phép toán tương ứng với nút đó, giá trị của TYPE trong trường này sẽ là: 1, 2, 3, 4, 5, 6 tương ứng với các phép toán: +, -, *, /, ...

+ Nếu là lá: TYPE = 0: chỉ biến hoặc hằng tương ứng với nút đó

RPTR trỏ tới địa chỉ trong bảng ký hiệu (symbol table) của biến hoặc hằng đó

Bảng ký hiệu chứa tên của biến hoặc hằng (ở trường SYMBOL), và giá trị của chúng (ở trường VALUE).

Chương 7 - ĐỒ THỊ VÀ MỘT VÀI CẤU TRÚC PHI TUYẾN

7.1. Định nghĩa và các khái niệm về đồ thị

Định nghĩa:

Một đồ thị $G(V,E)$ bao gồm

V : Tập hữu hạn các nút (đỉnh- vertices);

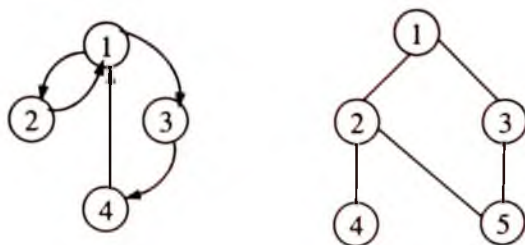
E : Tập hữu hạn các cặp đỉnh mà ta gọi là cung (edges).

Nếu (v_1, v_2) là các cặp đỉnh thuộc E thì ta nói có một cung nối v_1 và v_2 .

Nếu cung (v_1, v_2) khác cung (v_2, v_1) : đồ thị định hướng (directed graph hay digraph), (v_1, v_2) là cung định hướng từ v_1, v_2 .

Nếu thứ tự các nút trên cây không quan trọng ta có đồ thị không định hướng (undirected graph).

Ví dụ:



Cây là trường hợp đặc biệt của đồ thị

- Nếu (v_1, v_2) là một cung thuộc E thì v_1, v_2 gọi là lân cận của nhau (adjacent).

- Một đường đi (path) từ đỉnh v_p đến đỉnh v_q trong đồ thị G là một dãy các đỉnh $v_p, v_{i1}, v_{i2}, \dots, v_{in}, v_q$ là các cung trong $E(G)$. Số lượng các cung trên đường đi đó gọi là độ dài đường đi (path length).

- Một đường đi đơn (single path): mọi đỉnh trên đường đi đều khác nhau, trừ đỉnh đầu và đỉnh cuối.

- Chu trình (cycle): là đường đi đơn mà đỉnh đầu và đỉnh cuối trùng nhau.

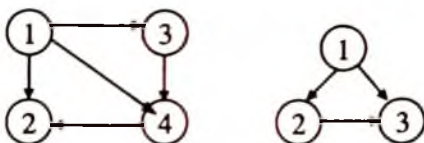
- Với đồ thị định hướng, để cho rõ người ta thường thêm vào các từ “định hướng” sau các thuật ngữ nêu ở trên.

Ví dụ: Đường đi định hướng từ v_i đến v_j

- Hai đỉnh v_i, v_j trong đồ thị được gọi là liên thông nếu có một đường đi từ v_i tới v_j (với đồ thị định hướng thì đồng thời cũng phải có một đường đi từ v_j tới v_i).

- Đồ thị $G(V,E)$ liên thông nếu: mọi cặp đỉnh (v_i, v_j) thuộc E đều liên thông.

Ví dụ:

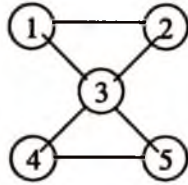


Đồ thị có trọng số: mỗi cạnh của nó mang một giá trị.

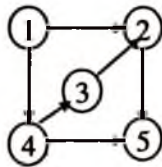
7.2. Biểu diễn đồ thị

7.2.1. Biểu diễn bằng ma trận lân cận kề

Xét đồ thị $G(V, E)$, V gồm n đỉnh $n \geq 1$



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



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

Nếu thứ tự các nút mà thay đổi thì ma trận kề cũng thay đổi, nhưng chúng được suy ra từ nhau bằng cách thay đổi hàng và cột cho nhau.

- Cùng một đồ thị, cho ta các ma trận khác nhau khi ta đánh số thứ tự khác nhau đối với các đỉnh.

7.2.2. Biểu diễn đồ thị bằng danh sách lân cận (kề)

- n hàng của ma trận được thay bằng n danh sách móc nối.
- Với mỗi đỉnh của G có một danh sách tương ứng. Các nút trong danh sách thứ i biểu diễn các đỉnh lân cận của nút i.

Mỗi nút gồm hai trường: VERTEX và LINK

- + VERTEX: Chứa chỉ số (số thứ tự) của các đỉnh lân cận của đỉnh i;
- + LINK: Chứa con trỏ, trỏ tới nút tiếp theo của danh sách.

**Chú ý:* Thứ tự các nút trong danh sách thực ra không quan trọng.

7.3. Phép duyệt đồ thị

Cần biết gốc của cây để thực hiện phép duyệt cây theo một thứ tự nào đó. Tương tự để duyệt một đồ thị không định hướng cần xác

định trước một đỉnh V trong $V(G)$, ta sẽ thăm tất cả các đỉnh còn lại thuộc G .

Có hai cách giải quyết: Phép tìm kiếm theo chiều sâu;

Phép tìm kiếm theo chiều rộng.

7.3.1. Tìm kiếm theo chiều sâu

Xuất phát từ đỉnh V (coi như được thăm), xét một đỉnh w là lân cận của V nhưng chưa được thăm.

Nếu đỉnh V đã được thăm mà mọi đỉnh lân cận của nó đều được thăm thì ta sẽ quay ngược lên đỉnh cuối cùng vừa được thăm mà nó có đỉnh w lân cận chưa được thăm và lặp lại cho tới khi không một nút nào không được thăm.

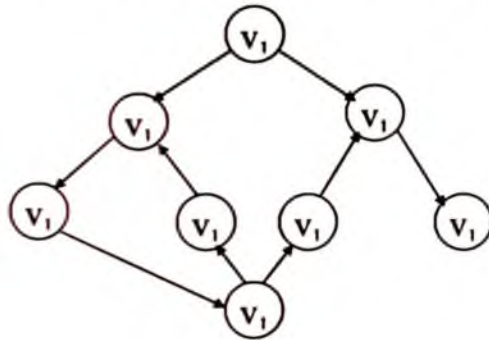


Ví dụ:

v_1 : v_1, v_3

v_2 : v_1, v_4, v_5

v_3 : v_1, v_6, v_7



Giải thuật của phép duyệt này có thể nêu:

Procedure DES(v)

- 1) VISITED(v):=1; {VISITED được dùng để đánh dấu các đỉnh đã được thăm}
- 2) for (mỗi đỉnh w lân cận của v) do
 if VISITED(w)=0 then DEF(w);
- 3) return;

***Nhận xét:**

- Trường hợp G được biểu diễn bởi danh sách lân cận: $O(t) = O(e)$.
- Trường hợp G được biểu diễn bởi ma trận lân cận: $O(t) = O(n^2)$.
- DFS (v_1):

+ Thăm mọi đỉnh liên thông với v_1 , tất cả và các cung liên quan tới các đỉnh đó được gọi là một bộ phận liên thông của G;

+ Dừng để nhận biết G có liên thông hay không, hoặc tìm các bộ phận liên thông của G nếu G không liên thông.

7.3.2. Tìm kiếm theo chiều rộng

Chọn một đỉnh xuất phát, giả sử là V, ta lần lượt thăm tất cả các lân cận đỉnh của V (nhưng chưa được thăm), sau đó lại lần lượt thăm tất cả các lân cận của các lân cận của V mà cũng chưa được thăm.

Ví dụ:



Thăm đỉnh V_1 , V_1 V_3 , ... và cuối cùng là V_8 .

Giải thuật:

Procedure BFS(v)

{ v : đỉnh xuất phát

VISITED(i):=1 => i được thăm ...}

1) VISITED(v):=1

2) Khởi tạo q với v đã được nạp vào;

3) While Q không rỗng do

Begin

call CQDELETE(v, Q); {lấy đỉnh v ra khỏi Q }

for (mỗi w lân cận của v) do

if VISITED(w):=0 then

```

begin
    call CQINSERT(w, Q);
    VISITED(w) := 1;
end;

end;

4) return

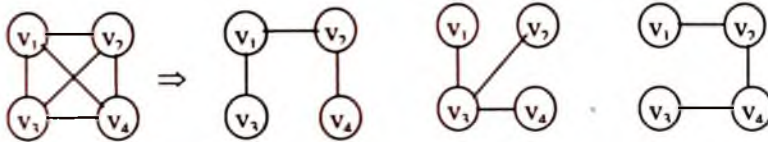
```

Thời gian chi phí là $O(n^2)$ nếu biểu diễn bởi ma trận lân cận, $O(e)$ nếu G được biểu diễn bởi danh sách lân cận.

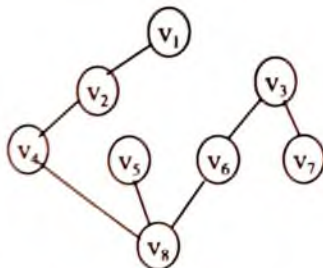
7.3.3. Cây khung và cây khung với giá trị cực tiểu (spanning trees and minimum cost spanning trees)

- Cây khung của G : một cây bao gồm tất cả các cung được dùng tới hay được duyệt qua cùng các đỉnh tương ứng.

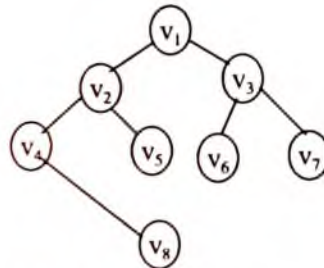
Ví dụ: xét đồ thị



- Tùy theo phép DFS hoặc BFS được sử dụng mà cây khung tương ứng sẽ được gọi là cây khung theo chiều sâu hay “cây khung theo chiều rộng”.



a) Cây khung DFS (v_1)



b) Cây khung BFS (v_1)

****Nhận xét:***

Một cung bất kỳ (v,w) nào của B được bổ sung vào cây khung T xuất hiện lập tức có chu trình xuất hiện.

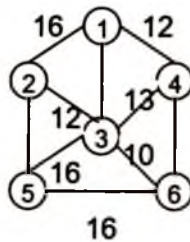
Xác định cây khung T với giá cực tiểu của đồ thị G liên thông có trọng số. Giá của cây khung bằng tổng trọng số ứng với các cung của cây ấy.

Giải thuật Kruskal:

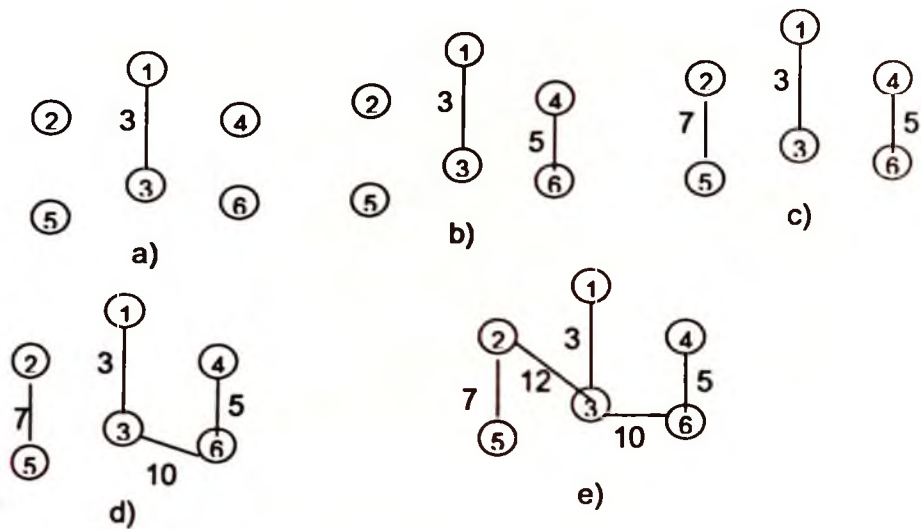
Ý tưởng: T sẽ được xác định dần dần từng cung một. Các cung được xét để đưa vào T:

- + Dựa theo thứ tự không giảm của giá trị tương ứng với chúng;
- + Không tạo nên chu trình đối với các đỉnh đã có trong T.

Ví dụ:



Giải thuật này được thực hiện từng bước như sau:



Các cung được xét để đưa vào T theo thứ tự:

(1, 3), (4,6), (2,5), (3,6), (2,3), (3,4), (1,4), (1,2), (3,5), (5,6)

Chúng tương ứng với dãy giá trị: 3,5,7,10,12,12,12,13,16,16,16

⇒ 4 cung đầu được nhận vào T, cung (3,4), (1,4) bị loại vì tạo chu trình trong T, cung (2,3) được tiếp nhận, cung (1,2), (3,5), (5,6) bị loại vì tạo chu trình trong T.

⇒ Cây có 5 cạnh và giá trị cực tiểu là 37

Giải thuật:

- 1) $T := \emptyset$; {Thoát đầu T rỗng}
- 2) While T chứa ít nhất $(n-1)$ cung do
- 3) begin Chọn 1 cung (v,w) từ E có giá trị thấp nhất
- 4) Loại (v,w) khỏi E
- 5) if (v,w) không tạo nên chu trình trong T
 then đưa (v,w) vào T
 else Loại bỏ (v,w)
- end;
- 6) return;

Thời gian thực hiện giải thuật trong trường hợp xấu nhất là $O(\text{eloge})$.

7.4. Áp dụng

7.4.1. Bài toán bao đóng truyền ứng

Bài toán: A, B là hai thành phố

- 1) Có một đường đi từ A tới B không?
- 2) Nếu có đường đi từ A đến B thì đường đi nào là đường đi ngắn nhất?

Sử dụng ma trận lân cận để giải quyết bài toán.

+ Có thể coi ma trận lân cận như một ma trận Bool, do đó có thể tác động lên ma trận các phép toán logic: \wedge, \vee

Xét ma trận lân cận A:

$a_{ij} = 1$: có một cung hoặc một đường đi từ đỉnh i đến đỉnh j ;

$$A^2 = A \wedge A \dots$$

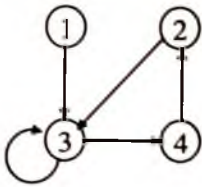
Nếu phần tử hàng i cột j của ma trận $A^2 = 1$, có ít nhất một đường đi từ i đến j có độ dài bằng 2... tương tự $A^r = A \wedge A^{(r-1)}$ với $r = 3, 4, \dots$
Nếu $a_{ij}^{(r)} = 1$ ít nhất có một đường đi độ dài r từ đỉnh i tới đỉnh j.

Vì vậy, nếu ta lập ma trận:

$$p = A \vee A^2 \vee \dots \vee A^{(n)} = \bigvee_{k=1}^n A^{(k)}$$

***Nhận xét**: Nếu phần tử i, j của ma trận $A^{(r)}$ mà bằng 1 thì sẽ tồn tại một đường đi có độ dài n từ đỉnh i đến j, p gọi là ma trận đường đi.

Ví dụ: Với đồ thị



$$A_1 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

A

$$A_2 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$A_2 = A \wedge A$

$$A_3 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$A_3 = A \wedge A_2$

$$A_4 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = P$$

$A_4 = A \wedge A_3$

Giải thuật Warshall: P được gọi là ma trận đường đi của đồ thị G. Từ ma trận P ta lại xây dựng một đồ thị khác mà các cung không biểu thị quan hệ lân cận mà biểu thị quan hệ liên thông.

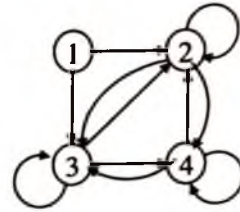
Việc tính P được thực hiện bởi giải thuật Warshall:

- 1) $P := A$; {các phần tử của A được thay lần lượt bởi các phần tử của P}
- 2) for i:=1 to n do
 - for j:=1 to n do
 - for k:=1 to n do
 $P[i,j] := P[i,j] \text{ or } a[i,k] \text{ and } a[k,j];$
- 3) return.

Các cung của P không chỉ thể hiện mối quan hệ lân cận giữa hai nút mà còn thể hiện mối quan hệ liên thông giữa chúng.

Đưa đồ thị tương ứng với P

$$P = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$



Đồ thị này còn được gọi là bao đóng truyền ứng của G.

***Chú ý:**

1) Ma trận đường đi cho ta biết có hay không ít nhất một đường đi từ giữa một cặp đỉnh i, j , có hay không một chu trình tại một điểm nào đó, nhưng không chỉ ra được mọi con đường có thể có.

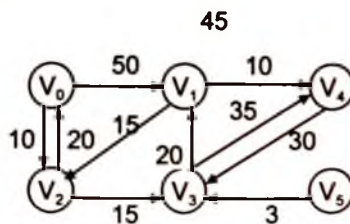
2) Ma trận P: Cho ta biết có hay không một đường đi từ nút này tới một nút khác.

7.4.2. Bài toán một nguồn mọi đích

Cho đồ thị định hướng: $G(V, E)$, $w(e)$ là hàm trọng số cho các cung e của G . V_0 : đỉnh nguồn

- Bài toán đặt ra: Xác định các đường đi ngắn nhất từ V_0 đến mọi đỉnh còn lại của G .

Ví dụ: Xét đồ thị:



Nếu V_0 là nguồn:

+ Đường đi ngắn nhất từ V_0 đến V_1 là $V_0 V_2 V_3 V_1$ với độ dài: $10+15+20=45$;

+ Không có đường đi từ V_0 đến V_5 ;

+ Các đường đi ngắn nhất từ V_0 tới V_1, V_2, V_3, V_4 được xác định bởi bảng sau:

Đường đi	Độ dài
1) V_0V_2	10
2) $V_0V_2V_3$	25
3) $V_0V_2V_3V_1$	45
4) V_0V_4	45

* Giải thuật xác định đường đi ngắn nhất:

Gọi S là tập các đỉnh (kể cả V_0), theo đó các đường đi ngắn nhất đã được xác lập;

Đối với mỗi đỉnh w không thuộc S , $DIST(w)$: là độ dài đường đi ngắn nhất từ V_0 qua các đỉnh chỉ trong S và kết thúc ở w . Khi đó:

1) Nếu đường đi ngắn nhất tiếp theo hướng tới một đỉnh w , thì đường đi đó bắt đầu từ V_0 , kết thúc ở w , chỉ qua những đỉnh đã thuộc S .

Chứng minh:

+ Mọi đỉnh trung gian từ V_0 đến w phải thuộc S ;

+ Thật vậy: Giả sử u thuộc đường đi từ V_0 đến w phải chứa đường đi từ V_0 đến u , do đó đường đi từ V_0 đến w nhỏ hơn độ dài đường đi từ V_0 đến w ;

Mà theo giả thiết: các đường đi ngắn nhất được xác định theo thứ tự không giảm, nên đường đi từ V_0 đến u phải được sinh ra trước rồi, có nghĩa là u phải thuộc S rồi.

Như vậy: không có một đỉnh trung gian nào trên đường đi mà không thuộc S .

2) Đỉnh của đường đi sinh ra tiếp theo phải là một đỉnh w nào đó hiện chưa thuộc S mà có $DIST(w)$ ngắn nhất so với giá trị của $DIST$ ứng với mọi đỉnh khác cũng đang không thuộc S .

3) Nếu đã chọn được w thoả mãn 2) và đã sinh ra một đường đi ngắn nhất từ V_0 đến w thì w sẽ trở thành một phần tử của S .

Từ đó độ dài của đường đi ngắn nhất xuất phát từ V_0 , đi qua các đỉnh chỉ thuộc S và kết thúc ở đỉnh w không thuộc S có khả năng sẽ giảm đi (nghĩa là giá trị $DIST(w)$ có thể thay đổi).

Nếu nó thay đổi thì nó phải được lập từ một đường đi ngắn hơn. Chẳng hạn: bắt đầu từ V_0 đến u rồi mới đến w . Nhưng các đỉnh trung gian trên đường đi từ V_0 đến u và từ u đến w đã thuộc S . Hơn thế nữa đường đi từ V_0 đến u phải là ngắn nhất và đường đi từ u đến w có thể chọn sao cho nó không chứa một đỉnh trung gian nào. Như vậy $DIST(w)$ sẽ thay đổi không tăng, cho nên

$$Dist(w) = Dist(u) + \text{độ dài}(u, w)$$

* Giải thuật SHORTEST-PATH($\gamma, COST, DIST, n$)

{ v : đỉnh nguồn, n : số đỉnh thuộc đồ thị, $DIST(j)$: độ dài đường đi ngắn nhất từ v đến j }

1) for $i:=1$ to n do

begin

$S[i]:=0$; $DIST[i]:=COST[v, i]$;

end;

2) $S[v]:=1$; $DIST(v):=0$; $k:=2$

3) While ($k < n$) do {xác định $n-1$ đường đi từ v }

4) Begin

Chọn u sao cho $DIST[u] = \min(DIST[u])$

$S[u]=0$

5) $S[u]:=1$; $k:=k+1$; {đưa u vào tập S }

6) For mọi w với $S[w]:=0$ do

7) $DIST[w] := \min(DIST[w], DIST[u] + COST[u, w])$

End {while}

8) return

Độ phức tạp tính toán: $O(n^2)$

7.4.3. Bài toán sắp xếp TÔPÔ

- Xét bài toán: Một sinh viên muốn đạt được một cấp nào đó trong quá trình học tập thì anh ta phải hoàn thành một số môn học quy định. Giữa các môn học có mối quan hệ “học trước, học sau” sao cho có một hệ thống, quan hệ đứng trước ký hiệu.

- Đưa bài toán về dạng đồ thị, trong đó:

+ Các môn học biểu diễn các đỉnh;

+ Quan hệ “được học trước” được biểu diễn bởi cung định hướng. Nghĩa là cung (i, j) thể hiện M_i được học trước M_j .

- Để có thể hoàn thành được môn học trong một khoảng thời gian nhất định, anh ta phải sắp xếp các môn học theo thứ tự tuyến tính để sao cho môn học trước phải được học trước. Chẳng hạn anh ta có thể học theo một trình tự: $M_1, M_{14}, M_4, M_8, M_{13}, M_5, \dots$

Thứ tự tuyến tính với đặc điểm như vậy gọi là “thứ tự TÔPÔ” (topological order) và cách sắp xếp một tập đối tượng có thứ tự bộ phận thành thứ tự TÔPÔ được gọi là sắp xếp TÔPÔ.

- Với một đồ thị G , sắp xếp TÔPÔ sẽ là một quá trình định ra một thứ tự tuyến tính cho các đỉnh của đồ thị ấy, sao cho nếu có một cung từ đỉnh i đến đỉnh j thì i cũng xuất hiện trước j theo thứ tự tuyến tính.

- Phương pháp để thực hiện sắp xếp TÔPÔ.

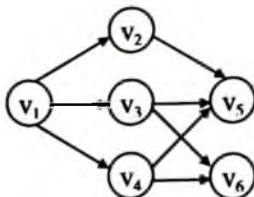
Ý tưởng:

+ Chọn ra một đỉnh mà không có cung nào đi tới nó, đỉnh này sẽ được đưa ra;

+ Loại bỏ đỉnh này cùng toàn bộ các cung xuất phát từ nó ra khỏi đồ thị;

+ Quá trình sẽ lặp lại với phần còn lại của đồ thị cho tới khi các đỉnh đó được chọn ra hết (nếu cùng một lúc có nhiều đỉnh đều không có cung đi tới nó, thì việc chọn một đỉnh nào đó trong số ấy là tùy ý).

Ví dụ: Xét đồ thị



Các bước thực hiện sắp xếp TÔPÔ:

Dãy đưa ra là: $V_1, V_2, V_4, V_3, V_5, V_6$ mà dưới dạng đồ thị có thể biểu diễn:

- Chọn cấu trúc dữ liệu: Dựa vào các phép xử lý, cụ thể ở đây là các phép:

- a) Xác định một đỉnh có đỉnh trước nó không;
- b) Loại một đỉnh cùng các cung xuất phát từ nó.

⇒ Đồ thị được biểu diễn bởi danh sách lân cận; nút đầu danh sách có hai trường COUNT và LINK.

- Giải thuật cụ thể:

+ COUNT: Chứa số đếm các cung đi đến đỉnh đó (đỉnh trước nó);

+ LINK: con trỏ, trỏ tới nút đầu tiên của danh sách lân cận tương ứng với mỗi đỉnh.

Danh sách lân cận của đồ thị này được coi như đã tạo lập xong trước khi thực hiện giải thuật.

```

Procedure TOPO-ORDER (Count, VERTEX, LINK, n)
{TOP là con trỏ, trỏ tới đỉnh stack}
1) {khởi tạo stack}
   TOP:=0;
  
```

```

2) {tạo lập stack của các đỉnh không có đỉnh trước
    nó}
for i:=1 to n do
    if COUNT(i)=0 then begin
        COUNT(i):=Top;
        Top:=1;
                                end;
3){Đưa các đỉnh theo thứ tự Tôpô}
for i:=1 to n do begin
4)    if Top:=0 then return;
5)    j:=Top; Top:=count (Top);
        write(j);
6)    ptr:=LINK (j);
7) {giảm số đếm các đỉnh sau j}while ptr ≠ 0 do
begin
                                k:=VERTEX (ptr)
                                COUNT (k):= COUNT
(k)-1;
8){Nạp vào stack phần tử mới}if COUNT (k):=0 then
begin
                                COUNT
(k):=Top;
                                Top:=k
9)                                ptr:=LINK (ptr)

end;
                                end;
10)return

```

+ Thời gian thực hiện giải thuật trên có cấp là $O(n+e)$ tuyến tính với kích thước của đồ thị.

Phần 3 - SẮP XẾP VÀ TÌM KIẾM

Chương 8 - SẮP XẾP

8.1. Đặt vấn đề

- *Sắp xếp* là quá trình bố trí lại các phần tử của một tập đối tượng nào đó, theo một thứ tự ấn định.

- *Mục đích*: giúp cho việc tìm kiếm được thuận lợi hơn

- n đối tượng được sắp ở đây ta quy ước là n bản ghi (record), mỗi bản ghi gồm một số trường (field), dữ liệu tương ứng với những thuộc tính (attribute) khác nhau.

Ví dụ:

Bài toán sắp xếp:

Input: n đối tượng tương đương n khoá sắp xếp.

Output: n đối tượng nhưng được sắp xếp theo trật tự mới (tăng hoặc giảm).

* Có hai loại sắp xếp chính là *sắp xếp trong* và *sắp xếp ngoài*.

- Sắp xếp trong (internal sorting): là loại sắp xếp mà các đối tượng cần sắp xếp đã có đầy đủ, đồng thời trong bộ nhớ RAM.

- Sắp xếp ngoài (external sort): Là trường hợp khi cần sắp xếp một tập tin mà không thể thực hiện việc sắp xếp tập tin đó trên bộ nhớ do tập tin có dung lượng quá lớn hay bộ nhớ quá nhỏ không đủ để thực hiện công việc này, chúng ta cần sử dụng đến bộ nhớ phụ để hỗ trợ việc sắp xếp đó. Việc sắp xếp trên bộ nhớ phụ hay sắp xếp được tổ

chức theo dạng tập tin được gọi là sắp xếp ngoài, chúng liên quan đến xử lý dữ liệu bên ngoài đơn vị xử lý trung tâm.

8.2. Một số phương pháp sắp xếp đơn giản

8.2.1. Sắp xếp kiểu lựa chọn

Nguyên tắc: ở lượt thứ i ($i = 1, 2, 3, \dots, n$) ta sẽ chọn trong dãy khoá k_i, k_{i+1}, \dots, k_n khoá nhỏ nhất và đổi chỗ nó với k_i .

Như vậy, sau j lượt, j khoá nhỏ hơn đã lần lượt ở vị trí nhất, nhì... thứ j theo đúng thứ tự sắp xếp

* Giải thuật:

```
Const max=20;
Type vec=array[1..max] of integer;
Procedure      SELECT-SORT(Var      A:vec;      n:byte;
Tăng:Boolean);
Var i,j,k:byte;
{dãy khoá k gồm n phần tử. Sắp xếp các phần tử của K
theo thứ tự tăng dần dựa vào phép chọn phần tử nhỏ
nhất trong mỗi lượt}
Begin
  for i:=1 to n-1 do
    begin
      j:=i;
      for k:=i+1 to n do
        if (a[k]<a[j])=tg then
          j:=k;
      if j>i then begin {đổi chỗ}
        tg:=a[i];
        a[i]:=a[j];
        a[j]:=tg;
      end;
    end;
  return;
```

8.2.2. Sắp xếp kiểu thêm dần

Nguyên tắc: dựa trên kinh nghiệm của những người chơi bài *Tìm "chỗ" thích hợp và "chèn" nó vào chỗ đó*, thoát đầu k_1 được coi như dãy chỉ gồm có một khoá đã được sắp xếp. Xét thêm k_2 , so sánh nó với k_1 để xác định chỗ chèn nó vào, sau đó ta sẽ có một dãy gồm hai khoá đã được sắp xếp. Đối với k_3 lại so sánh với k_2 , k_1 và cứ tương tự như vậy đối với k_4 , k_5 ... cuối cùng sau khi xét xong k_n thì bảng khoá đã được sắp xếp hoàn toàn.

Dấu -: Chỉ chỗ trống trong miền sắp xếp;

... : Chỉ việc phải dịch chuyển khoá cũ để lấy chỗ chèn khoá mới vào.

Giải thuật:

```
Procedure Insert-Sort(k,n)
  For i:=2 to n do
    Begin
      X:=K[i]; j:=i-1;
      While (X<K[j]) and (j > 0) do {Xác định chỗ
cho khoá mới được xét và dịch chuyển các khoá cần
thiết}
        Begin
          K[j+1]:=K[j]; j:=j-1;
        end;
      K[j+1]:=X; {đưa X vào đúng chỗ}
    End
  Return;
```

8.2.3. Sắp xếp kiểu đổi chỗ - phương pháp sủi bọt

Nguyên tắc:

Lặp lại $n-1$ lần từ trái sang phải. Mỗi lần duyệt ta xét từng cặp phần tử liên tiếp trong dãy, nếu thấy số trước mà lớn hơn số sau thì đổi chỗ.

Ví dụ: 3 2 1 4 3 2 1
 i=1 j=7 3 2 1 4 3 1 2
 j=6 3 2 1 4 1 3 2 ...

Giải thuật:

```

Procedure BUBBLE-SORT(k,n)
1) for i:=1 to n-1 do
2)     for j:=n downto i+1 do
3)         if k[j]<k[j-1] then
4)             Begin
                 X:=k[j];
                 K[j]:=k[j-1];
                 K[j-1]:=x;
             End;
5) return;

```

Chứng minh tính đúng đắn:

- Mỗi lần duyệt ta sẽ được phần tử nhỏ nhất chuyển lên trên đầu dãy.
- Sau n-1 lần duyệt, ta được dãy sắp xếp.

8.2.4. Phân tích so sánh ba phương pháp

- Hiệu lực của thuật toán được đánh giá về mặt:
- + Chi phí bộ nhớ cần thiết (chưa đề cập);
- + Chi phí về thời gian: phụ thuộc chủ yếu vào việc thực hiện các phép so sánh giá trị khoá, và các phép chuyển đổi bản ghi khi sắp. Ở đây, ta chỉ xét tới phép so sánh và coi nó như một “phép toán tích cực” của các giải thuật.

- Với phương pháp sắp xếp kiểu lựa chọn:

⇒ Độ phức tạp tính toán trong trường hợp xấu nhất là:

$$2((n-1) + (n-2) + \dots + 2 + 1) + 4(n-1) = 2n(n-1)/2 + 4n - 4 : O(n^2)$$

- Với phương pháp sắp xếp kiểu thêm dần (INSERT-SORT)

⇒ Độ phức tạp tính toán trong trường hợp xấu nhất là:

$$4(1+2+\dots+n-1) + 3(n-1) = 4n \cdot (n-1)/2 + 3n - 3 = 2n^2 + n - 3 : O(n^2)$$

- Với kiểu sắp xếp nổi bọt (sủi bọt)

⇒ Độ phức tạp trong trường hợp xấu nhất là:

$$4[(n-1) + (n-2) + \dots + 2 + 1] = 4n \cdot (n-1)/2$$

Chi phí thời gian của cả ba phương pháp đều có cấp $O(n^2)$. Đây cũng là một chi phí cao so với một số phương pháp mà ta xét thêm sau đây.

8.3. Sắp xếp kiểu phân đoạn hay sắp xếp nhanh

8.3.1. Giới thiệu phương pháp

Sắp xếp kiểu phân đoạn là một cải tiến của phương pháp sắp xếp kiểu đổi chỗ. Đây là phương pháp khá tốt (còn gọi là sắp xếp nhanh).

Phương pháp có thể tóm tắt như sau:

- Chọn một phần tử A thuộc dãy S làm khoá "chốt".

Mọi phần tử nhỏ hơn "khoá chốt" được xếp vào vị trí trước "chốt" (đầu dãy). Mọi phần tử lớn hơn "khoá chốt" được xếp vào vị trí sau "chốt" (cuối dãy).

- Chia S làm ba phân đoạn con:

+ S_1 : Gồm những phần tử có giá trị nhỏ hơn A (chốt);

+ S_2 : Gồm phần tử A;

+ S_3 : Gồm những phần tử có giá trị lớn hơn A.

- Áp dụng kỹ thuật tương tự cho các phân đoạn con của S cho đến khi dãy được sắp xếp hoàn toàn.

8.3.2. Ví dụ và giải thuật

$$i=1 \quad i=2 \quad i=3 \quad j=4 \quad i=5 \qquad j=8 \quad j=10$$

* Xét dãy: 42 23 74 11 65 58 94 36 99 87

Chốt là 42; sử dụng hai biến i, j với giá trị ban đầu $i=2, j=10$, 2 khoá 42 và k_i được so sánh:

+ Nếu $k_i < 42$ thì i được tăng lên 1 và quá trình được lặp lại cho đến khi $k_i > 42$;

+ Nếu $k_j > 42$ thì j được giảm 1 cho tới khi $k_j < 42$ lúc đó k_i và k_j được đổi chỗ cho nhau.

Dãy khoá được chia làm hai phân đoạn và 42 đã đúng vị trí của nó. Quá trình sẽ lặp lại với từng phân đoạn cho tới khi dãy khoá được sắp xếp hoàn toàn.

(11 23 36) (42) (65 58 94 74 99 87)

Giải thuật:

{Đây là giải thuật đệ quy}

Procedure QUICK-SORT(k, lb, ub)

{dãy k gồm n bản ghi, sắp xếp dãy tăng dần theo giá trị của khoá, sử dụng một bản ghi giả có khoá tương ứng là: $k[n+1]$ mà $k[i] < k[n+1]$ mọi $i = 1..n$

lb: chỉ số dưới của dãy

ub: chỉ số trên của dãy

KEY: biến chứa giá trị khoá chốt

B: biến logic, B=false dãy được phân làm hai dãy con}

1) {khởi đầu}

B:=true;

2) {Thực hiện sắp xếp}

if $lb < ub$ then

begin

$i := lb$; $j := ub + 1$; KEY := $k[lb]$;

While B do

Begin

$i := i + 1$;

while ($k[i] < KEY$) do $i := i + 1$;

```

        j:=j-1;
        while k[j]>KEY do j:=j-1;
        if (i<j) then k[i] √ k[j]
        else B:=false;
    End
    k[lb] √ k[j]; {đưa khóa chốt vào vị trí}
    QUICK-SORT(k, lb, j-1); {sắp xếp dãy con đứng trước
    chốt}
    QUICK-SORT(k, j+1, ub); {sắp xếp dãy con đứng trước
    chốt}
    End
3) return;

```

8.3.3. Nhận xét và đánh giá

a) Vấn đề chọn “chốt”

- Chọn chốt có thể tùy ý trong dãy khoá

+ Nếu chốt trùng với max (min) thì sau mỗi lượt tách chỉ được một phân đoạn con có kích thước nhỏ hơn trước là 1 (phương pháp sắp xếp kiểu nổi bọt đơn giản), khi này sẽ dẫn đến tình huống xấu nhất của phương pháp;

+ Nếu chốt trùng với “trung vị” (khoá đứng giữa dãy khoá đó). Sau mỗi lượt ta sẽ tách ta được hai phân đoạn con có độ dài gần như nhau và phân đoạn xử lý tiếp theo sẽ có kích thước chỉ bằng nửa phân đoạn đã cho.

b) Vấn đề phối hợp với cách sắp xếp khác

Đánh giá giải thuật QUICK-SORT

- Độ phức tạp trong trường hợp xấu nhất: $O(n^2)$, vì:

+ Tối đa cần $n-1$ lần tách;

+ Mỗi lần tách cần $O(n)$.

$$\Leftrightarrow (n-1) * O(n) = O(n^2)$$

- Thời gian trung bình là: $O(n \log_2 n)$: kết quả đã được chứng minh

Như vậy: rõ ràng khi n khá lớn thì QUICK-SORT tỏ ra hiệu quả hơn hẳn ba phương pháp đã nêu.

8.4. Sắp xếp kiểu vun đống

Sắp xếp kiểu phân đoạn cho ta thời gian thực hiện trung bình khá tốt, nhưng trong trường hợp xấu nhất của nó vẫn là: $O(n^2)$.

Phương pháp sắp xếp sau đây đảm bảo được trong cả hai trường hợp chi phí thời gian đều cùng là: $O(n \log_2 n)$.

8.4.1. Giới thiệu phương pháp

Sắp xếp kiểu vun đống được chia làm hai giai đoạn:

- Giai đoạn đầu (giai đoạn tạo đống): cây nhị phân được biến đổi để trở thành một đống (heap), một đống là một cây nhị phân hoàn chỉnh, mỗi nút được gán một giá trị khoá sao cho nút khoá con bao giờ cũng lớn hơn khoá ở nút con nó. Cây nhị phân này được lưu trữ kế tiếp trong máy.

Nếu j chỉ nút con thì $j/2$ chỉ nút cha, $k[j/2] > k[j]$ với $1 \leq j/2 < j \leq n$. Khoá ở gốc là khoá max (trội so với cây).

- Giai đoạn sắp xếp: nhiều lượt xử lý được thực hiện, mỗi lượt ứng với hai phép:

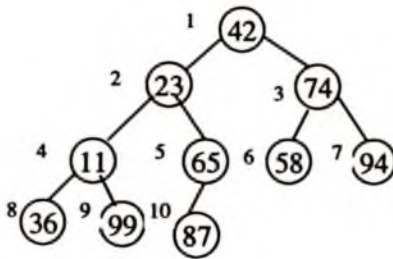
+ Đưa khoá trội về vị trí thực của nó bằng cách đổi chỗ cho khoá hiện đang ở vị trí đó;

+ “Vun lại thành đống” đối với cây gồm các khoá còn lại (sau khi đã loại khoá trội ra ngoài).

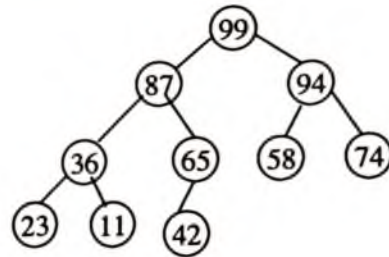
Ví dụ: Xét dãy: 42 23 74 11 65 58 94 36 99 87

Ban đầu: cấu trúc đồng sẽ có dạng đồng (hình a).

Sau giai đoạn đầu: cấu trúc có dạng hình b.



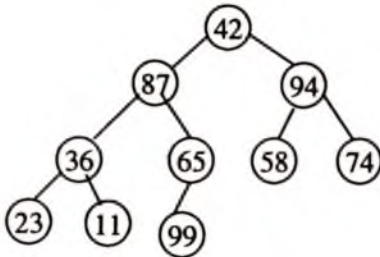
a) Cây biểu diễn cho dãy khóa



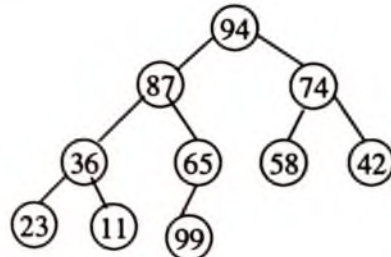
b) Đồng ban đầu

Sang giai đoạn 2:

Khoá trội 99 sẽ được đưa vào vị trí cuối cùng bằng cách đổi chỗ cho khoá 42. Sau đó nút ứng với vị trí này coi như loại khỏi cây, khi này cây mới không còn là đồng nữa, do đó ta phải vun đồng mới cho cây này. Mỗi lượt tiếp theo sẽ được thực hiện tương tự cho đến khi mọi khoá đã vào đúng chỗ sắp xếp của nó.



a) Sau khi khoá 99 đã vào đúng vị trí



b) Cây với các nút còn lại, được vun thành đồng mới

8.4.2. Giải thuật

```
1) procedure Hoan-vi-cha-con(l,r)
begin
```

```

        if (l $\neq$ lá và kl mang giá trị < giá trị nút con)
then
        begin
            Chọn con có giá trị lớn hơn j
            Đổi chỗ  $x_j$  cho  $x_l$ ;
        end;
End
2) procedure Tao_heap_dau_tien
begin
    for i:=n div 2 downto 1 do
        Hoan-vi-cha-con(i,n)
    End
3) Procedure Heap_sort(X,n)
Begin
    Tao_heap_dau_tien;
    For i:=n downto 2 do
        Begin
            Đổi chỗ  $x_1$  cho  $x_i$ 
            Hoan-vi-cha-con(1,i-1);
        End;
    End;
End;

```

8.4.3. Phân tích đánh giá

Độ phức tạp tính toán trong trường hợp xấu nhất là: $O(n \cdot \log_2 n)$.
 Thời gian thực hiện trung bình cũng là $O(n \cdot \log_2 n)$.

8.5. Sắp xếp kiểu hoà nhập

8.5.1. Phép hoà nhập hai đường

- Input: dãy X (m phần tử), Y (n phần tử) đã được sắp xếp (tăng)
- Output: dãy $Z = X$ ghép Y cũng sắp tăng.
 - + Cách 1: trộn hai đường vào nhau sắp lại;

+ Cách 2: phép hợp nhất các bản ghi của hai bảng đã được sắp xếp để ghép thành một bảng.

Nguyên tắc:

So sánh hai khoá nhỏ nhất (lớn nhất) của hai dãy X, Y chọn khoá nhỏ hơn (lớn hơn) đưa ra miền sắp xếp (có kích thước bằng tổng số các phần tử của hai dãy X, Y); khoá được chọn bị loại ra khỏi dãy chứa nó. Quá trình tiếp tục cho đến khi một dãy bị cạn. Lúc đó cần chuyển toàn bộ phần đuôi của dãy còn lại ra miền sắp xếp là xong.

* Giải thuật hoà nhập hai dãy đã được sắp là hai phân đoạn kế tiếp trong một dãy lớn X:

```
Procedure MERGE (X,b,m,n,Z)
1) i:=k:=b; j:=m+1;
2) while (i≤m and j≤n) do
    if X[i]≤X[j] then
        Begin
            Z[k]:=X[i];
            i:=i+1;
        End
    Else
        Begin
            Z[k]:=X[j];
            j:=j+1;
        End;
    k:=k+1;
3) {một trong 2 bảng con đã cạn trước}
    if i>m then (Zk, ..., Zn) := (Xj, ..., Xn)
4) Return;
```

Chú thích:

1) $(Z_k, \dots, Z_n) := (X_j, \dots, X_n)$ là phép gán quy ước thay cho một dãy các phép gán tương ứng các phần tử thuộc dãy ở vế phải cho phần tử thuộc dãy ở vế trái, nghĩa là:

```

Z[k]:=X[j];
Z[k+1]:=X[j-1];

```

.....

```

Z[n]:=X[n]

```

2) Giải thuật hoà nhập hai dãy con được sắp (X_1, X_2, \dots, X_m) và (Y_1, \dots, Y_n) cho ra một dãy mới được sắp (Z_1, \dots, Z_{n+m}) cũng tương tự.

3) Dãy con đã có thứ tự sắp xếp thường được gọi là một mạch (run).

8.5.2. Sắp xếp kiểu hoà nhập hai đường trực tiếp

Thủ tục thực hiện phép sắp xếp này

```

Procedure MPASS(X,Y,n,e)

```

```

1) i:=1

```

```

2) {hoà nhập cặp mạch có độ dài l}

```

```

while i≤n-2l-1 do begin

```

```

    call MERGE(X,i,i+l-1,i+2l-1,Y);

```

```

    i:=i+2l

```

```

end

```

```

3) {Hoà nhập cặp mạch còn lại với tổng độ dài < 2l}

```

```

if i+l-1<n then call MERGE (X,i,i+l-1,n,Y)

```

```

    else (Yi,...,Xn)

```

```

4) return

```

{Thủ tục MPASS sẽ được gọi tới thủ tục sắp xếp sau đây}

```

Procedure STRAIGHT-MSORT(X,n)

```

```

1) l:=1;

```

```

2) while l<n do begin

```

```

    Call MPASS(X,Y,n,l); :=l+1;

```

```

    Call MPASS(Y,X,n,l); :=l+1;

```

```

end

```

```

3) return;

```


8.5.3. Phân tích đánh giá

Số lượt hoà nhập sẽ là: $\log_2 n$.

Thời gian mỗi lượt $O(n)$.

Vậy sắp xếp kiểu hoà nhập hai đường trực tiếp có cấp thời gian là $O(n \log_2 n)$.

8.5.4. Nhận xét

Cùng một mục đích sắp xếp như nhau, mà có rất nhiều phương pháp và kỹ thuật giải quyết khác nhau. Việc chọn một phương pháp sắp xếp thích hợp tùy thuộc vào từng yêu cầu, từng điều kiện cụ thể.

- Các phương pháp sắp xếp đơn giản có cấp độ lớn của thời gian thực hiện chung là $O(n^2)$. Nên sử dụng chúng khi n nhỏ.

- Các giải thuật cải tiến như QUICK-SORT, HEAP-SORT đã đạt được hiệu quả cao, $O(n \log n)$. Thường sử dụng chúng khi n lớn.

- MERGE-SORT: không kém hiệu lực về thời gian thực hiện nhưng đòi hỏi nhiều bộ nhớ nên không thích hợp với việc sắp xếp trong.

- QUICK-SORT: không nên dùng nếu dãy cần sắp vốn có khuynh hướng hầu như “đã được sắp sẵn”.

- HEAP-SORT: Nên dùng nếu dãy cần sắp xếp có khuynh hướng ít nhiều có thứ tự ngược với thứ tự sắp xếp.

Chương 9 - TÌM KIẾM

9.1. Bài toán tìm kiếm

Bài toán: Cho một bảng gồm n bản ghi R_1, R_2, \dots, R_n - mỗi bản ghi R_i ($1 \leq i \leq n$) tương ứng với khoá k_i , hãy tìm bản ghi có giá trị khoá tương ứng bằng X cho trước.

- Input: + n đối tượng (bản ghi), mỗi đối tượng tương ứng với một khoá;

+ X : Là giá trị của khoá cần tìm kiếm

- Output: Tìm đối tượng đầu tiên thoả mãn có khoá bằng khoá của giá trị tìm k_i .

Ở đây, ta chỉ xét tới các phương pháp tìm kiếm cơ bản và phổ dụng, đối với dữ liệu ở bộ nhớ trong nghĩa là tìm kiếm trong, dữ liệu ở bộ nhớ ngoài là tìm kiếm ngoài.

9.2. Tìm kiếm tuần tự

9.2.1. Tìm kiếm tuần tự

Là kỹ thuật tìm kiếm rất đơn giản và cổ điển. Nội dung như sau: bắt đầu từ bản ghi thứ nhất, lần lượt so sánh khoá tìm kiếm với khoá tương ứng của các bản ghi trong bảng, cho tới khi tìm được bản ghi mong muốn hoặc đã hết bảng mà chưa thấy.

Giải thuật:

```
Funtion SEQUENCE_SEARCH(k,n,X)
1) {Khởi đầu}
```

```

        i:=1; k[n+1]:=X;
2) {Tìm khoá trong dãy}
        While k[i]≠X do i:=i+1;
3) {Tìm thấy hay không}
        if i =n+1 then return (0)
        else return (i)

```

9.2.2. Đánh giá hiệu lực của phép tìm kiếm

Dựa vào số lượng các phép so sánh

+ Thuận lợi chỉ cần một phép so sánh $C_{min}=1$;

+ Không thuận lợi $C_{max}=n+1$.

Chi phí thời gian là: $O(n)$.

Thời gian trong trường hợp xấu nhất vẫn giống thời gian của giải thuật tìm kiếm trên khi chưa được sắp, nhưng thời gian trung bình thì ít hơn.

9.3. Tìm kiếm nhị phân (phân đôi)

Thuật toán tìm kiếm tuyến tính (hay tuần tự) trên dãy n phần tử có độ phức tạp là $O(n)$. Trong phần này chúng ta nêu lên một thuật toán khác để tìm kiếm trên dãy có thứ tự: thuật toán tìm kiếm nhị phân (binary search). Nhờ tận dụng tính thứ tự của dãy trong việc tìm kiếm, thuật toán tìm kiếm nhị phân hiệu quả hơn thuật toán tìm kiếm tuần tự.

Không làm mất tính tổng quát, chúng ta có thể giả sử dãy đang xét là một dãy số tăng dần: a_1, a_2, \dots, a_n . Để tìm kiếm phần tử x trong dãy trên, ta chỉ cần so sánh x với phần tử ở giữa dãy số; nếu bằng nhau thì việc tìm kiếm kết thúc; ngược lại ta có thể bỏ qua việc dò tìm cả phần bên trái hoặc cả phần bên phải phần tử giữa dãy. Chẳng hạn, nếu x nhỏ hơn phần tử giữa dãy thì ta chỉ cần tìm kiếm tiếp phần tử x trong phần dãy bên trái phần tử giữa mà thôi vì phần bên phải chỉ gồm những phần tử lớn hơn x (do dãy số có thứ tự tăng dần). Cách làm này

sẽ rút ngắn thời gian tìm kiếm rất nhiều, nhất là khi n lớn. Thuật toán tìm kiếm nhị phân có thể được trình bày như dưới đây.

9.3.1. Phương pháp tìm kiếm nhị phân

Chọn "khoá ở giữa" dãy khoá đang xét để thực hiện so sánh với khoá tìm kiếm.

+ Nếu "Khoá" < khoá ở giữa dãy tìm kiếm, sẽ tiếp tục thực hiện với "khoá ở giữa"-1;

+ Nếu "Khoá" > khoá ở giữa dãy tìm kiếm, sẽ tiếp tục thực hiện với "khoá ở giữa"+1, ... đến hết dãy.

Quá trình tìm kiếm được tiếp tục khi tìm thấy khoá mong muốn hoặc dãy khoá xét đó trở nên rỗng.

Giải thuật

```
function BINARY_SEARCH(k,n,X)
1) {khởi đầu}
   l:=1;
   r:=n;
2) {tìm}
   while l≤r do begin
3) {tính chỉ số giữa}
   m:= ⌊(l+r)/2⌋;
4) {so sánh}
   if X<k[m] then r:=m-1
   else
   if X>k[m] then l:=m+1
   else return (m)
   end
5) {Tìm kiếm không thoả}
return (0)
```

9.3.2. Phân tích đánh giá

Chi phí thời gian:

Số lần chia là $\log_2 n$. Vì mỗi lần chia là một hằng số nhất định, độ phức tạp là $O(\log_2 n)$

9.4. Cây nhị phân tìm kiếm

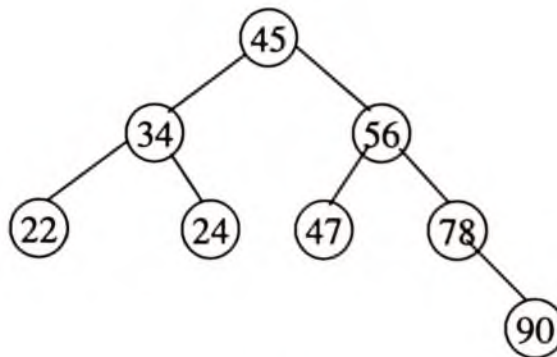
9.4.1. Định nghĩa

Cây nhị phân tìm kiếm ứng với n khoá k_1, k_2, \dots, k_n là cây: mỗi nút của nó đều được định danh bởi một khoá nào đó trong các khoá đã cho, mọi nút trên cây có tính chất:

- + Mọi khoá thuộc cây con trái nút đó đều nhỏ hơn khoá ứng với nút đó;

- + Mọi khoá thuộc cây con phải nút đó đều lớn hơn khoá ứng với nút đó.

Ví dụ:



Hình a

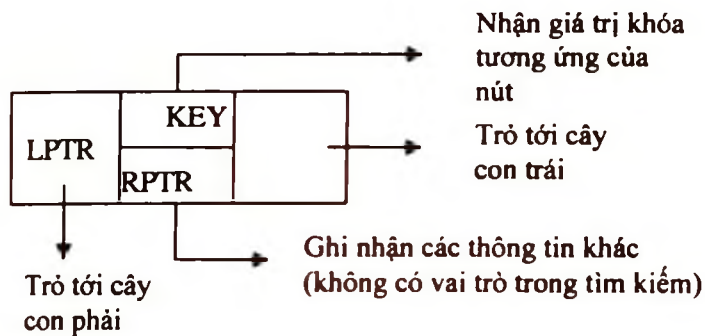
9.4.2. Giải thuật tìm kiếm

Bài toán: Xác định một khoá X có thuộc cây tìm kiếm hay không, ta thực hiện như sau:

- So sánh X với gốc, có các tình huống sau xảy ra:
 - a. Cây rỗng (không gốc): X không có trên cây, phép tìm kiếm không thoả mãn;
 - b. X trùng khoá gốc: phép tìm kiếm thoả mãn;
 - c. $X < \text{khoá gốc}$: xét tiếp đối với cây con trái;
 - d. $X > \text{khoá gốc}$: xét tiếp đối với cây con phải.

Ví dụ: Giả sử $X=9$, tìm kiếm trên cây (hình a)

- Nếu phép tìm kiếm không thoả mãn, ta bổ sung X vào cây: giả sử quy cách mỗi nút của cây nhị phân có dạng:



Giải thuật tìm kiếm có bổ sung trên cây nhị phân tìm kiếm.

Ở đây, ta chỉ xét tới các phương pháp tìm kiếm cơ bản và phổ dụng, đối với dữ liệu ở bộ nhớ trong nghĩa là tìm kiếm trong, dữ liệu ở bộ nhớ ngoài: Tìm kiếm ngoài.

TÀI LIỆU THAM KHẢO

- [1] Đỗ Xuân Lôì, *Cấu trúc dữ liệu và giải thuật*, NXB Thống kê 1996.
- [2] Nguyễn Đức Nghĩa, Nguyễn Tô Thành, *Toán rời rạc*, NXB Đại học Quốc gia Hà Nội 2003.
- [3] Gregory Heilleman, *Data Structures, Algorithms, and Object-Oriented Programming*, McGraw Hill Press 1996.
- [4] Thomas H. C., Charles E.L., Ronald L.R., *Advanced Data Structures*, Mc Graw Hill Press 1990.
- [5] Robert Sedgewick, *Cẩm nang thuật toán (tập 1, 2)*, NXB Khoa học và Kỹ thuật 1996.
- [6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithm, Second Edition*, The MIT Press 2001.

MỤC LỤC

LỜI NÓI ĐẦU	3
Phần 1 - GIẢI THUẬT	
<i>Chương 1 - Mở đầu</i>	
1.1 Giải thuật và cấu trúc dữ liệu	4
1.2 Cấu trúc dữ liệu và các vấn đề liên quan	5
1.3 Ngôn ngữ diễn đạt giải thuật	7
<i>Chương 2 - Phân tích và thiết kế giải thuật</i>	
2.1 Từ bài toán đến chương trình	12
2.2 Phân tích giải thuật	19
<i>Chương 3 - Độ qui và giải thuật đệ quy</i>	
3.1 Khái niệm về đệ quy	24
3.2 Giải thuật đệ quy và thủ tục đệ quy	24
3.3 Hiệu lực của đệ quy	34
3.5 Đệ quy và quy nạp toán học	35
Phần 2 - CẤU TRÚC DỮ LIỆU	
<i>Chương 4 - Mảng và danh sách</i>	
4.1 Các khái niệm	36
4.2 Cấu trúc lưu trữ mảng	37
4.3 Lưu trữ kế tiếp của danh sách tuyến tính	40
4.4 Stack hay danh sách kiểu ngăn xếp	40
4.5 Queue hay danh sách kiểu hàng đợi	48
<i>Chương 5 - Danh sách móc nối</i>	
5.1 Danh sách nối đơn	52
5.2 Danh sách nối vòng	57
5.3 Danh sách nối kép	58
5.4 Ví dụ áp dụng	61
	115

5.5 Stack hay queue móc nối	64
Chương 6 - Cây	
6.1 Định nghĩa và các khái niệm	66
6.2 Cây nhị phân	68
6.3 Cây tổng quát	74
6.4 Áp dụng	78
Chương 7 - Đồ thị và một vài cấu trúc phi tuyến khác	
7.1 Định nghĩa và các khái niệm về đồ thị	80
7.2 Biểu diễn đồ thị	81
7.3 Phép duyệt đồ thị	82
7.4 Áp dụng	88
Phần 3 - SẮP XẾP VÀ TÌM KIẾM	
Chương 8 - Sắp xếp	
8.1 Đặt vấn đề	96
8.2 Một số phương pháp sắp xếp đơn giản	97
8.3 Sắp xếp kiểu phân đoạn hay sắp xếp nhanh	100
8.4 Sắp xếp kiểu vun đống	103
8.5 Sắp xếp kiểu hoà nhập	105
Chương 9 - Tìm kiếm	
9.1 Bài toán tìm kiếm	109
9.2 Tìm kiếm tuần tự	109
9.3 Tìm kiếm nhị phân (phân đôi)	110
9.4 Cây nhị phân tìm kiếm	112
Tài liệu tham khảo	114
Mục lục)	115