

ĐỖ XUÂN LÔI

# Cấu trúc dữ liệu và giải thuật

*(In lần thứ chín, có sửa chữa)*

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

**NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI**

16 Hàng Chuối - Hai Bà Trưng - Hà Nội

Điện thoại: (04) 9718312; (04) 7547936. Fax: (04) 9714899

E-mail: nxb@vnu.edu.vn

★ ★ ★

***Chịu trách nhiệm xuất bản:***

*Giám đốc:* PHÙNG QUỐC BẢO

*Tổng biên tập:* PHẠM THÀNH HƯNG

***Biên tập:*** HỒ ĐỒNG

NGUYỄN TRỌNG HẢI

***Trình bày bìa:*** SÁNG ĐỒNG

---

**CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

Mã số: 1L-17 ĐH2006

In 1000 cuốn, khổ 16 x 24 cm tại Trung tâm In tranh Tuyên truyền cổ động - Mai Dịch, Cầu Giấy, Hà Nội

Số xuất bản: 85 - 2006/CXB/75 - 01/ĐHQGHN, ngày 24/01/2006.

Quyết định xuất bản số: 74 LK/XB

In xong và nộp lưu chiểu quý I năm 2006.

## **LỜI GIỚI THIỆU**

*(Cho lần xuất bản thứ bảy)*

Kể từ năm 1993 đến nay, cuốn "Cấu trúc dữ liệu và giải thuật" của PGS. Đỗ Xuân Lôi đã được đông đảo bạn đọc đón nhận và hoan nghênh.

Cuốn sách này đã trở thành tài liệu học tập và tham khảo của sinh viên ngành Công nghệ Thông tin ở nhiều cơ sở đào tạo Cao đẳng, Đại học và sau Đại học.

Khác với 6 lần in trước, trong lần xuất bản này tác giả đã bổ sung và chỉnh lý lại nhiều phần. Tác giả cũng đã chú ý đúc rút kinh nghiệm qua nhiều năm giảng dạy để việc giới thiệu các nội dung kiến thức cũng như bài tập phù hợp hơn và dễ tiếp cận hơn đối với các đối tượng bạn đọc khác nhau.

Hy vọng rằng cuốn sách sẽ đáp ứng tốt hơn yêu cầu của bạn đọc trong việc nâng cao trình độ về công nghệ thông tin.

**NHÀ XUẤT BẢN**  
**ĐẠI HỌC QUỐC GIA HÀ NỘI**

## LỜI NÓI ĐẦU

(Cho lần xuất bản đầu tiên)

Cuốn sách này phản ánh nội dung của một môn học cơ sở trong chương trình đào tạo kỹ sư tin học. Ở đây sinh viên sẽ được làm quen với một số 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 về kỹ thuật lập trình, về phương pháp giải các bài toán, giúp sinh viên có khả năng đi sâu thêm vào các môn học chuyên ngành như cơ sở dữ liệu, trí tuệ nhân tạo, hệ chuyên gia, ngôn ngữ hình thức, chương trình dịch v.v...

Nội dung cuốn sách được chia làm 3 phần

**Phần I:** Bổ sung thêm nhận thức về mối quan hệ giữa cấu trúc dữ liệu và giải thuật, về vấn đề thiết kế, phân tích giải thuật và về giải thuật đệ qui.

**Phần II:** Giới thiệu một số cấu trúc dữ liệu, giải thuật xử lý chúng và vài ứng dụng điển hình. Ở đây sinh viên sẽ tiếp cận với các cấu trúc như mảng, danh sách, cây, đồ thị và một vài cấu trúc phi tuyến khác. Sinh viên cũng có điều kiện để hiểu biết thêm về một số bài toán thuộc loại "phi số", cũng như thu lượm thêm kinh nghiệm về thiết kế, cài đặt và xử lý chúng.

**Phần III:** Tập trung vào "sắp xếp và tìm kiếm", một yêu cầu xử lý rất phổ biến trong các ứng dụng tin học. Có thể coi đây như một phần minh họa thêm cho việc ứng dụng các cấu trúc dữ liệu khác nhau trong cùng một loại bài toán.

Cuốn sách bao gồm 11 chương, chủ yếu giới thiệu các kiến thức cần thiết cho 90 tiết học, cả lý thuyết và bài tập (sau khi sinh viên đã học tin học đại cương). Tuy nhiên, với mục đích vừa làm tài liệu học tập, vừa làm tài liệu tham khảo, nên nội dung của nó có bao hàm thêm một số phần nâng cao.

Bài tập sau mỗi chương đã được chọn lọc ở mức trung bình, để sinh viên qua đó hiểu thêm bài giảng và thu hoạch thêm một số nội dung mới không được trực tiếp giới thiệu.

Cuốn sách có thể được dùng làm tài liệu học tập cho sinh viên hệ kỹ sư tin học, cử nhân tin học, cao đẳng tin học; làm tài liệu tham khảo cho sinh

viên cao học, nghiên cứu sinh, giảng viên tin học và các cán bộ tin học muốn nâng cao thêm trình độ.

Trong quá trình chuẩn bị, tác giả đã nhận được những ý kiến đóng góp về nội dung, cũng như các hoạt động hỗ trợ cho việc cuốn sách được sớm ra mắt bạn đọc. Tác giả xin chân thành cảm ơn GS. Nguyễn Đình Trí chủ nhiệm đề tài cấp nhà nước KC01-13 về Tin học - Điện tử - Viễn thông; PGS Nguyễn Xuân Huy, Viện tin học VN; PGS. Nguyễn Văn Ba, PTS Nguyễn Thanh Thủy và các đồng nghiệp trong Khoa Tin học trường ĐH Bách khoa HN.

Mặc dầu cuốn sách đã thể hiện được phần nào sự cân nhắc lựa chọn của tác giả trong việc kết hợp giữa yêu cầu khoa học với tính thực tiễn, tính sư phạm của các bài giảng, nhưng chắc chắn vẫn không tránh khỏi các thiếu sót. Tác giả mong muốn nhận được các ý kiến đóng góp thêm để có thể hoàn thiện hơn nữa nội dung cuốn sách, trong những lần tái bản sau.

*Hà Nội ngày 15/8/1993*

**ĐỖ XUÂN LÔI**

# **PHẦN I**

# **GIẢI THUẬT**

## Chương 1

# MỞ ĐẦU

### 1.1 Giải thuật và cấu trúc dữ liệu

Có thể, có lúc, khi nói tới việc giải quyết bài toán trên máy tính điện tử, người ta chỉ chú ý đến *giải thuật* (algorithms). Đó là một dãy các *câu lệnh* (statements) 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 bước thực hiện ta đạt được kết quả mong muốn.

Nhưng, xét cho cùng, giải thuật chỉ phản ánh các phép xử lý, còn đối tượng để xử lý trên máy tính điện tử, chính là *dữ liệu* (data) chúng biểu diễn các thông tin cần thiết cho bài toán: các dữ kiện đưa vào, các kết quả trung gian... Không thể nói tới giải thuật mà không nghĩ tới: giải thuật đó được tác động trên dữ liệu nào, còn khi xét tới dữ liệu thì cũng phải hiểu: dữ liệu ấy cần được tác động giải thuật gì để đưa tới kết quả mong muốn.

Bản thân các phần tử của dữ liệu thường có mối quan hệ với nhau, ngoài ra nếu lại biết "tổ chức" theo các cấu trúc thích hợp thì việc thực hiện các phép xử lý trên các dữ liệu sẽ càng thuận lợi hơn, đạt hiệu quả cao hơn. Với một cấu trúc dữ liệu đã chọn ta sẽ có giải thuật xử lý tương ứng. Cấu trúc dữ liệu thay đổi, giải thuật cũng thay đổi theo. Ta sẽ thấy rõ điều đó qua ví dụ sau: Giả sử ta có một danh sách gồm những cặp "Tên đơn vị, số điện thoại":  $(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)$ .

Ta muốn viết một chương trình cho máy tính điện tử để khi cho biết "tên đơn vị" máy sẽ in ra cho ta: "số điện thoại". Đó là một loại bài toán mà phép xử lý cơ bản là "tìm kiếm".

- Một cách đơn giản là cứ điểm lần lượt các tên trong danh sách  $a_1, a_2, a_3$  v.v... cho tới lúc tìm thấy tên đơn vị  $a_i$  nào đó, đã chỉ định, thì đối chiếu ra số điện thoại tương ứng  $b_i$  của nó. Nhưng việc đó chỉ làm được khi danh mục điện thoại ngắn, nghĩa là với  $n$  nhỏ, còn với  $n$  lớn thì rất mất thời gian.
- Nếu trước đó danh mục điện thoại đã được sắp xếp theo thứ tự tự diễn

(dictionary order) đối với tên đơn vị, tất nhiên sẽ áp dụng một giải thuật tìm kiếm khác tốt hơn, như ta vẫn thường làm khi tra từ điển.

- Nếu tổ chức thêm một bảng mục lục chỉ dẫn theo chữ cái đầu tiên của "tên đơn vị", chắc rằng khi tìm số điện thoại của Đại học Bách khoa ta sẽ bỏ qua được các tên đơn vị mà chữ đầu không phải là chữ Đ.

Như vậy: giữa cấu trúc dữ liệu và giải thuật có mối quan hệ mật thiết. Có thể coi chúng như hình với bóng. Không thể nói tới cái này mà không nhắc tới cái kia.

Chính điều đó đã dẫn tới việc, cần nghiên cứu các *cấu trúc dữ liệu* (data structures) đi đôi với việc xác lập các giải thuật xử lý trên các cấu trúc ấy.

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

**1.2.1.** Trong một bài toán, dữ liệu bao gồm một tập các phần tử cơ sở, mà ta gọi là *dữ liệu nguyên tử* (atoms). Nó có thể là một chữ số, một ký tự... nhưng cũng có thể là một con số, hay một từ..., điều đó tùy thuộc vào từng bài toán.

Trên cơ sở của các dữ liệu nguyên tử, các cung cách (manners) khá đi theo đó liên kết chúng lại với nhau, sẽ dẫn tới các cấu trúc dữ liệu khác nhau.

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

Cần chú ý rằng, trong những năm gần đây, lớp các khái niệm về cấu trúc dữ liệu đã tăng lên đáng kể. Thoạt đầu, 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 các cấu trúc dữ liệu đơn giản như biến, vectơ, ma trận v.v... nhưng khi các ứng dụng đó đã mở rộng sang các lĩnh vực khác mà ta thường gọi là các *bài toán phi số* (non-numerical problems), với đặc điểm thể hiện ở chỗ: khối lượng dữ liệu lớn, đa dạng, biến động; phép xử lý thường không phải chỉ là các phép số học... thì các cấu trúc này không đủ đặc trưng cho các mối quan hệ mới của dữ liệu nữa. Việc đi sâu thêm vào các cấu trúc dữ liệu phức tạp hơn, chính là sự quan tâm của ta trong giáo trình này.

**1.2.2.** Đối với các bài toán phi số đi đôi với các cấu trúc dữ liệu mới cũng xuất hiện các phép toán mới tác động trên các cấu trúc ấy: Phép tạo lập hay huỷ bỏ một cấu trúc, phép truy cập (access) vào từng phần tử của cấu trúc, phép bổ sung (insertion) hoặc loại bỏ (deletion) một phần tử trên cấu trúc v.v...

Các phép đó sẽ có những tác dụng khác nhau đối với từng cấu trúc. Có phép hữu hiệu đối với cấu trúc này nhưng lại tỏ ra không hữu hiệu trên cấu trúc khác.



Vì vậy chọn một cấu trúc dữ liệu phải nghĩ ngay tới các phép toán tác động trên cấu trúc ấy. Và ngược lại, nói tới phép toán thì lại phải chú ý tới phép đó được tác động trên cấu trúc nào. Cho nên cũng không có gì lạ khi người ta quan niệm: nói tới cấu trúc dữ liệu là bao hàm luôn cả phép toán tác động trên các cấu trúc ấy. Ở giáo trình này tuy ta tách riêng hai khái niệm đó nhưng cấu trúc dữ liệu và các phép toán tương ứng vẫn luôn được trình bày cùng với nhau.

**1.2.3.** Cách biểu diễn một cấu trúc dữ liệu trong bộ nhớ được gọi là *cấu trúc lưu trữ* (storage structures). Đó chính là cách cài đặt cấu trúc ấy trên máy tính điện tử và trên cơ sở các cấu trúc lưu trữ này mà thực hiện các phép xử lý. Sự phân biệt giữa cấu trúc dữ liệu và cấu trúc lưu trữ tương ứng, cần phải được đặt ra. Có thể có nhiều cấu trúc lưu trữ khác nhau cho cùng một cấu trúc dữ liệu, cũng như có thể có những cấu trúc dữ liệu khác nhau mà được thể hiện trong bộ nhớ bởi cùng một kiểu cấu trúc lưu trữ. Thường khi xử lý, mọi chú ý đều hướng tới cấu trúc lưu trữ, nên ta dễ quên mất cấu trúc dữ liệu tương ứng.

Khi đề cập tới cấu trúc lưu trữ, ta cũng cần phân biệt: cấu trúc lưu trữ tương ứng với bộ nhớ trong - *lưu trữ trong*, hay ứng với bộ nhớ ngoài - *lưu trữ ngoài*. Chúng đều có những đặc điểm riêng và kéo theo các cách xử lý khác nhau.

**1.2.4.** Thường trong một ngôn ngữ lập trình bao giờ cũng có các cấu trúc dữ liệu tiền định (predefined data structures). Chẳng hạn: cấu trúc *mảng* (array) là cấu trúc rất phổ biến trong các ngôn ngữ. Nó thường được sử dụng để tổ chức các tập dữ liệu, có số lượng ấn định và có cùng kiểu. Nếu như sử dụng một ngôn ngữ mà cấu trúc dữ liệu tiền định của nó phù hợp với cấu trúc dữ liệu xác định bởi người dùng thì tất nhiên rất thuận tiện. Nhưng không phải các cấu trúc dữ liệu tiền định của ngôn ngữ lập trình được sử dụng đều đáp ứng được mọi yêu cầu cần thiết về cấu trúc, chẳng hạn nếu xử lý hồ sơ cán bộ mà dùng ngôn ngữ PASCAL, thì ta có thể tổ chức mỗi hồ sơ dưới dạng một *bản ghi* (record) bao gồm nhiều thành phần, mỗi thành phần của bản ghi đó ta gọi là *trường* (field) sẽ không nhất thiết phải cùng kiểu. Ví dụ, trường: "Họ và tên" có *kiểu ký tự* (char), trường: "ngày sinh" có *kiểu số nguyên* (integer).

Nhưng nếu dùng ngôn ngữ FORTRAN thì lại gặp khó khăn. Ta chỉ có thể mô phỏng các mục của hồ sơ dưới dạng các vectơ hay ma trận và do đó việc xử lý sẽ phức tạp hơn.

Cho nên chấp nhận một ngôn ngữ tức là chấp nhận các cấu trúc tiền định của ngôn ngữ ấy và phải biết linh hoạt vận dụng chúng để mô phỏng các cấu trúc dữ liệu đã chọn cho bài toán cần giải quyết.

Tuy nhiên, trong thực tế việc lựa chọn một ngôn ngữ không phải chỉ xuất phát từ yêu cầu của bài toán mà còn phụ thuộc vào rất nhiều yếu tố khách quan cũng như chủ quan của người lập trình nữa.

Tóm lại, trừ vấn đề thứ tư vừa nêu, có thể tách ra và xét riêng, tới đây ta cũng thấy được: ba vấn đề trước đều liên quan tới cấu trúc dữ liệu. Chúng ảnh hưởng trực tiếp đến giải thuật để giải bài toán.

Vì vậy ba vấn đề này chính là đối tượng bàn luận đến trong giáo trình của chúng ta.

## 1.3 Ngôn ngữ diễn đạt giải thuật

Mặc dầu vấn đề ngôn ngữ lập trình không được đặt ra ở giáo trình này, nhưng để diễn đạt các giải thuật mà ta sẽ trình bày trong giáo trình, ta cũng không thể không lựa chọn một ngôn ngữ. Có thể nghĩ ngay tới việc sử dụng một ngôn ngữ cấp cao hiện có, chẳng hạn PASCAL, C, C', ... nhưng như vậy ta sẽ gặp một số hạn chế sau:

- Phải luôn luôn tuân thủ các quy tắc chặt chẽ về cú pháp của ngôn ngữ đó khiến cho việc trình bày về giải thuật và cấu trúc dữ liệu có thiên hướng nặng nề, gò bó.

- Phải phụ thuộc vào cấu trúc dữ liệu tiền định của ngôn ngữ nên có lúc không thể hiện được đầy đủ các ý về cấu trúc mà ta muốn biểu đạt.

- Ngôn ngữ nào được chọn cũng không hẳn đã được mọi người yêu thích và muốn sử dụng.

Vì vậy, ở đây ta sẽ dùng một ngôn ngữ "thô hơn", có đủ khả năng diễn đạt được giải thuật trên các cấu trúc đề cập đến (mà ta giới thiệu bằng tiếng Việt), với một mức độ linh hoạt nhất định, không quá gò bó, không cầu nệ nhiều về cú pháp nhưng cũng gần gũi với các ngôn ngữ chuẩn để khi cần thiết dễ dàng chuyển đổi. Ta tạm gọi nó bằng cái tên: "ngôn ngữ tựa PASCAL". Sau đây là một số quy tắc bước đầu, ở các chương sau sẽ có thể bổ sung thêm.

### 1.3.1 Quy cách về cấu trúc chương trình

Mỗi chương trình đều được gán một tên để phân biệt, tên này được viết bằng chữ in hoa, có thể có thêm dấu gạch nối và bắt đầu bằng từ khoá **Program**.

**Ví dụ:**

**Program NHAN-MA-TRAN**

Độ dài tên không hạn chế.

Sau tên có thể kèm theo lời thuyết minh (ở đây ta quy ước dùng tiếng Việt) để giới thiệu tóm tắt nhiệm vụ của giải thuật hoặc một số chi tiết cần thiết. Phần thuyết minh được đặt giữa hai dấu { ..... }.

Chương trình bao gồm nhiều đoạn (bước) mỗi đoạn được phân biệt bởi số thứ tự, có thể kèm theo những lời thuyết minh.

### 1.3.2 Ký tự và biểu thức

\* Ký tự dùng ở đây cũng giống như trong các ngôn ngữ chuẩn, nghĩa là gồm:

- 26 chữ cái Latin in hoa hoặc in thường
- 10 chữ số thập phân
- Các dấu phép toán số học  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\uparrow$  (lũy thừa)
- Các dấu phép toán quan hệ  $<$ ,  $=$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$
- Giá trị logic: **true**, **false**
- Dấu phép toán logic: **and**, **or**, **not**
- Tên biến: dãy chữ cái và chữ số, bắt đầu bằng chữ cái.
- Biến chỉ số có dạng:  $A[i]$ ,  $B[i,j]$ , v.v...

\* Còn biểu thức cũng như thứ tự ưu tiên của các phép toán trong biểu thức cũng theo quy tắc như trong PASCAL hay các ngôn ngữ chuẩn khác.

### 1.3.3 Các câu lệnh (hay các chỉ thị)

Các câu lệnh trong chương trình được viết cách nhau bởi dấu ; chúng bao gồm:

#### 1.3.3.1 Câu lệnh gán

Có dạng  $V := E$

với  $V$  chỉ tên biến, tên hàm

$E$  chỉ biểu thức

Ở đây cho phép dùng phép gán chung.

**Ví dụ:**

$A := B := 0.1$

#### 1.3.3.2 Câu lệnh ghép

Có dạng:

**Begin  $S_1, S_2, \dots, S_n$ ; end**

với  $S_i, i = 1, \dots, n$  là các câu lệnh.

Nó cho phép ghép nhiều câu lệnh lại để được coi như một câu lệnh.

### 1.3.3.3 Câu lệnh điều kiện

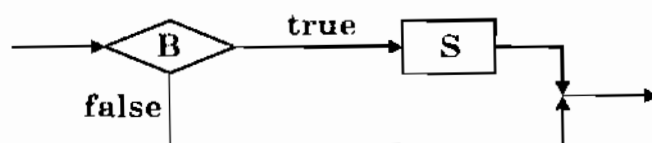
Có dạng:

**if B then S**

với B là biểu thức logic

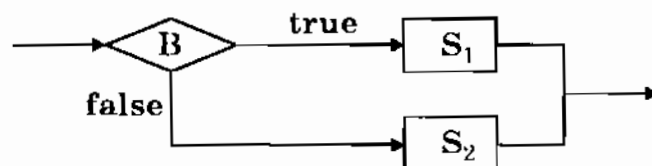
S là một câu lệnh khác

có thể diễn tả bởi sơ đồ



hoặc:

**if B then  $S_1$  else  $S_2$**



### 1.3.3.4 Câu lệnh tuyến

**Case**

$B_1: S_1;$

$B_2: S_2;$

...

...

$B_n: S_n;$

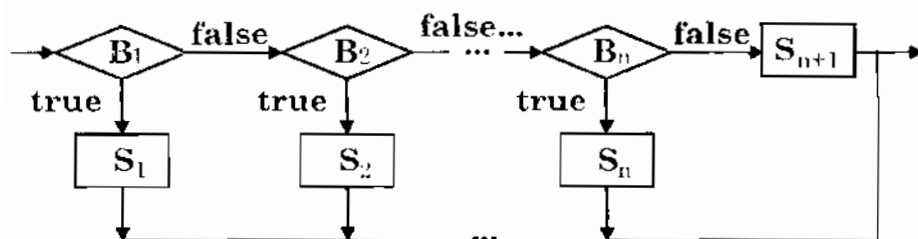
**else:  $S_{n+1}$**

**end case**

với:  $B_i (i = 1, 2, \dots, n)$  là các điều kiện

$S_i (i = 1, 2, \dots, n)$  là các câu lệnh

\* Câu lệnh này cho phép phân biệt các tình huống xử lý khác nhau trong các điều kiện khác nhau mà không phải dùng tới các câu lệnh **if - then - else** lồng nhau. Có thể diễn tả bởi sơ đồ:



\* Vài điểm linh động

**else** có thể không có mặt.

$S_i$  ( $i = 1, 2, \dots, n$ ) có thể được thay bằng một dãy các câu lệnh thể hiện một dãy xử lý khi có điều kiện  $B_i$  mà không cần phải đặt giữa **begin** và **end**

### 1.3.3.5 Câu lệnh lặp

\* Với số lần lặp biết trước

**for**  $i := m$  **to**  $n$  **do**  $S$

nhằm thực hiện câu lệnh  $S$  với  $i$  lấy giá trị nguyên từ  $m$  tới  $n$  ( $n \geq m$ ), với bước nhảy tăng bằng 1;

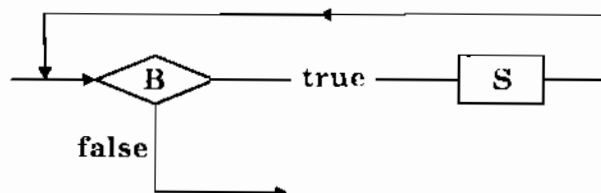
Hoặc:

**for**  $i := n$  **down to**  $m$  **do**  $S$

tương tự như câu lệnh trên với bước nhảy giảm bằng 1.

\* Với số lần lặp không biết trước

**while**  $B$  **do**  $S$

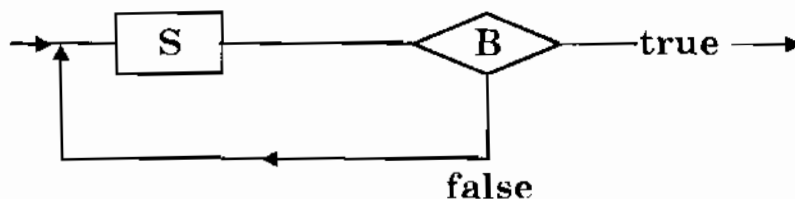


chừng nào mà  $B$  có giá trị bằng **true** thì thực hiện  $S$ ,

hoặc:

### **repeat S until B**

Lặp lại S cho tới khi B có giá trị true (S có thể là một dãy lệnh)



#### **1.3.3.6 Câu lệnh chuyển**

**go to n** (n là số hiệu của một bước trong chương trình)

Thường người ta hạn chế việc dùng **go to**. Tuy nhiên với mục đích diễn đạt cho tự nhiên, trong một chừng mực nào đó ta vẫn sử dụng.

#### **1.3.3.7 Câu lệnh vào, ra**

Có dạng:

**read** (<danh sách biến>)

**write** (<danh sách biến hoặc dòng ký tự>)

Các biến trong danh sách cách nhau bởi dấu phẩy.

Dòng ký tự là một dãy các ký tự đặt giữa hai dấu nháy ''.

#### **1.3.3.8 Câu lệnh kết thúc chương trình**

**end**

### **1.3.4 Chương trình con**

\* Chương trình con hàm

Có dạng:

**function** <tên hàm> (<danh sách tham số>)

$S_1, S_2, \dots, S_n$

**return**

Câu lệnh kết thúc chương trình con ở đây là **return** thay cho **end**

\* Chương trình con thủ tục

Tương tự như trên, chỉ khác ở chỗ:

Từ khoá **procedure** thay cho **function**.

Trong cấu tạo của chương trình con hàm bao giờ cũng có câu lệnh gán mà tên hàm nằm ở vế trái. Còn đối với chương trình con thủ tục thì không có.

Lời gọi chương trình con hàm thể hiện bằng tên hàm cùng danh sách tham số thực sự, nằm trong biểu thức. Còn đối với chương trình con thủ tục lời gọi được thể hiện bằng câu lệnh **call** có dạng:

**Call** <tên thủ tục> (<danh sách tham số thực sự>)

**Chú ý:** Trong các chương trình diễn đạt một giải thuật ở đây phần khai báo dữ liệu được bỏ qua. Nó được thay bởi phần mô tả cấu trúc dữ liệu bằng ngôn ngữ tự nhiên, mà ta sẽ nêu ra trước khi bước vào giải thuật.

Như vậy nghĩa là các chương trình được nêu ra chỉ là đoạn thể hiện các phép xử lý theo giải thuật đã định, trên các cấu trúc dữ liệu được mô tả trước đó, bằng ngôn ngữ tự nhiên.

## BÀI TẬP CHƯƠNG 1

- 1.1. Tìm thêm các ví dụ minh họa mối quan hệ giữa cấu trúc dữ liệu và giải thuật.
- 1.2. Các bài toán phi số khác với các bài toán khoa học kỹ thuật ở những đặc điểm gì?
- 1.3. Cấu trúc dữ liệu và cấu trúc lưu trữ khác nhau ở chỗ nào?
- 1.4. Hãy nêu một vài cấu trúc dữ liệu tiền định của các ngôn ngữ mà anh (chị) biết.
- 1.5. Các cấu trúc dữ liệu tiền định trong một ngôn ngữ có đủ đáp ứng mọi yêu cầu về tổ chức dữ liệu không?  
Có thể có cấu trúc dữ liệu do người dùng định ra không?
- 1.6. Một chương trình PASCAL có phải là một tập dữ liệu có cấu trúc không?
- 1.7. Hãy nêu các tính chất của một giải thuật và cho ví dụ minh họa.



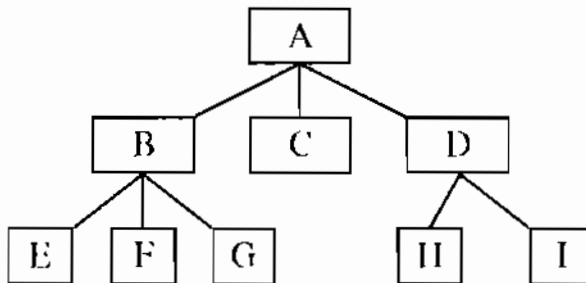
# THIẾT KẾ VÀ PHÂN TÍCH GIẢI THUẬT

## 2.1 Từ bài toán đến chương trình

### 2.1.1 Mô-đun hoá và việc giải quyết bài toán

Các bài toán giải được trên máy tính điện tử ngày càng đa dạng và phức tạp. Các giải thuật và chương trình để giải chúng cũng ngày càng có quy mô lớn và càng khó khi thiết lập cũng như khi muốn tìm hiểu.

Tuy nhiên, ta cũng thấy rằng mọi việc sẽ đơn giản hơn nếu như có thể phân chia bài toán lớn của ta thành các bài toán nhỏ. Điều đó cũng có nghĩa là nếu coi bài toán của ta như một mô-đun chính thì cần chia nó thành các mô-đun con, và dĩ nhiên, với tinh thần như thế, đến lượt nó, mỗi mô-đun này lại được phân chia tiếp cho tới những mô-đun ứng với các phần việc cơ bản mà ta đã biết cách giải quyết. Như vậy việc tổ chức lời giải của bài toán sẽ được thể hiện theo một cấu trúc phân cấp, có dạng như hình sau:



Hình 2.1

Chiến thuật giải quyết bài toán theo tinh thần như vậy chính là chiến thuật "chia để trị" (divide and conquer). Để thể hiện chiến thuật đó, người ta dùng cách thiết kế "từ đỉnh xuống" (top-down design). Đó là cách phân

tích tổng quát toàn bộ vấn đề, xuất phát từ dữ kiện và các mục tiêu đặt ra, để đề cập đến những công việc chủ yếu, rồi sau đó mới đi dần vào giải quyết các phần cụ thể một cách chi tiết hơn (cũng vì vậy mà người ta gọi là cách *thiết kế từ khái quát đến chi tiết*). Ví dụ ta nhận được từ Chủ tịch Hội đồng xét cấp học bổng của trường một yêu cầu là:

"Dùng máy tính điện tử 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".

Như vậy trước hết ta phải hình dung được cụ thể hơn đầu vào và đầu ra của bài toán.

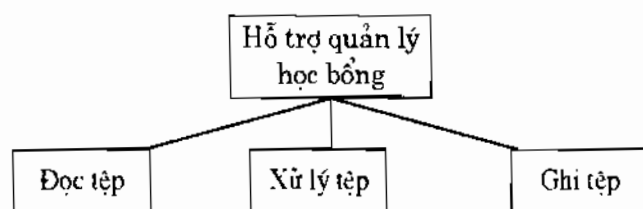
Có thể coi như ta đã có một tập các hồ sơ (mà ta gọi là *tệp* - file) bao gồm các bản ghi (records) về các thông tin liên quan tới học bổng của sinh viên, chẳng hạn: số hiệu sinh viên, điểm trung bình (theo học kỳ), điểm đạo đức, khoản tiền tài trợ. Và chương trình lập ra phải tạo điều kiện cho người sử dụng giải quyết được các yêu cầu sau:

- 1) Tìm lại và hiển thị được bản ghi của bất kỳ sinh viên nào tại thiết bị cuối (terminal) của người dùng.
- 2) Cập nhật (update) được bản ghi của một sinh viên cho trước bằng cách thay đổi điểm trung bình, điểm đạo đức, khoản tiền tài trợ, nếu cần.
- 3) In bản tổng kết chứa những thông tin hiện thời (đã được cập nhật mỗi khi có thay đổi) gồm số hiệu, điểm trung bình, điểm đạo đức, khoản tiền tài trợ.

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

- 1) Những thông tin về sinh viên được học bổng, lưu trữ trên đĩa phải được đọc vào bộ nhớ trong để có thể xử lý (ta gọi là nhiệm vụ "đọc tệp").
- 2) Xử lý các thông tin này để tạo ra kết quả mong muốn (nhiệm vụ: "xử lý tệp").
- 3) 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").

Có thể hình dung, cách thiết kế này theo sơ đồ cấu trúc ở hình 2.2

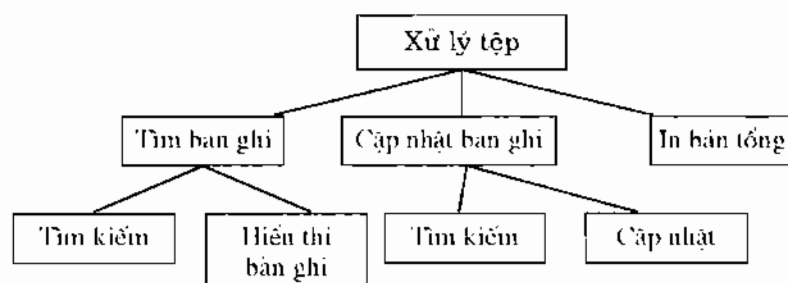


Hình 2.2

Các nhiệm vụ ở mức đầu này thường tương đối phức tạp, cần phải chia thành các nhiệm vụ con. Chẳng hạn, nhiệm vụ "xử lý tệp" sẽ được phân thành ba, tương ứng với việc giải quyết ba yêu cầu chính đã được nêu ở trên:

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

Những nhiệm vụ con này cũng có thể chia thành nhiệm vụ nhỏ hơn. Có thể hình dung theo sơ đồ cấu trúc như sau:



Hình 2.3

Cách thiết kế giải thuật theo kiểu top-down như trên giúp cho việc giải quyết bài toán được định hướng rõ ràng, tránh sa đà ngay vào các chi tiết phụ. Nó cũng là nền tảng cho việc lập trình có cấu trúc.

Thông thường, đối với các bài toán lớn, việc giải quyết nó phải do nhiều người cùng làm. Chính phương pháp mô-đun hoá sẽ cho phép tách bài toán ra thành các phần độc lập tạo điều kiện cho các nhóm giải quyết phần việc của mình mà không làm ảnh hưởng gì đến nhóm khác. Với chương trình được xây dựng trên cơ sở của các giải thuật được thiết kế theo cách này thì việc tìm hiểu cũng như sửa chữa chính lý sẽ dễ dàng hơn.

Việc phân bài toán thành các bài toán con như thế không phải là một việc làm dễ dàng. Chính vì vậy mà có những bài toán nhiệm vụ phân tích và thiết kế giải thuật giải bài toán đó còn mất nhiều thời gian và công sức hơn cả nhiệm vụ lập trình.

### 2.1.2 Phương pháp tinh chỉnh từng bước (Stepwise refinement)

Tinh chỉnh từng bước là phương pháp thiết kế giải thuật gắn liền với lập trình. Nó phản ánh tinh thần của quá trình mô-đun hoá bài toán và thiết kế kiểu top-down.

Thoạt đầu chương trình thể hiện giải thuật được trình bày bằng ngôn ngữ tự nhiên phản ánh ý chính của công việc cần làm. Từ các bước sau, những lời, những ý đó sẽ được chi tiết hoá dần dần tương ứng với những công việc nhỏ hơn. Ta gọi đó là các bước tinh chỉnh, sự tinh chỉnh này sẽ được hướng về phía ngôn ngữ lập trình mà ta đã chọn. Càng ở các bước sau các lời lẽ đặc tả công việc xử lý sẽ được thay thế dần bởi các câu lệnh hướng tới các lệnh của ngôn ngữ lập trình. Muốn vậy ở các giai đoạn trung gian người ta thường dùng pha tạp cả ngôn ngữ tự nhiên lẫn ngôn ngữ lập trình, mà người ta gọi là *giả ngôn ngữ* (pseudo - language) hay *giả mã* (pseudo code). Như vậy nghĩa là quá trình thiết kế giải thuật và phát triển chương trình sẽ được thể hiện dần dần từ dạng ngôn ngữ tự nhiên, qua giả ngôn ngữ rồi đến ngôn ngữ lập trình và đi từ mức "làm cái gì" đến mức "làm thế nào", ngày càng sát với các chức năng ứng với các câu lệnh của ngôn ngữ lập trình đã chọn.

Trong quá trình này dữ liệu cũng được "tinh chế" dần dần từ dạng cấu trúc đến dạng lưu trữ cài đặt cụ thể.

Sau đây ta xét một vài ví dụ:

**Ví dụ 1.** Giả sử ta muốn lập một chương trình sắp xếp một dãy  $n$  số nguyên khác nhau theo thứ tự tăng dần.

\* Có thể phác thảo giải thuật như sau:

Từ dãy các số nguyên chưa được sắp xếp chọn ra số nhỏ nhất, đặt nó vào cuối dãy đã được sắp xếp.

Cứ lặp lại quy trình đó cho tới khi dãy chưa được sắp xếp trở thành rỗng.

Ta thấy phác hoạ trên còn đang rất thô, nó chỉ thể hiện những ý cơ bản.

Hình dung cụ thể hơn một chút ta thấy, thoạt đầu dãy số chưa được sắp xếp chính là dãy số đã cho. Dãy số đã được sắp xếp còn rỗng, chưa có phần tử nào. Vậy thì nếu chọn được số nhỏ nhất đầu tiên và đặt vào cuối dãy đã được sắp thì cũng chính là đặt vào vị trí đầu tiên của dãy này. Nhưng dãy này đặt ở đâu?

Thế thì phải hiểu dãy số mà ta sẽ sắp xếp được đặt tại chỗ cũ hay đặt ở chỗ khác? Điều đó đòi hỏi phải chi tiết hơn về cấu trúc dữ liệu và cấu trúc lưu trữ của dãy số cho.

Trước hết ta ấn định: dãy số cho ở đây được coi như dãy các phần tử của một vectơ (sau này ta nói: nó có cấu trúc của mảng một chiều) và dãy này được lưu trữ bởi một vectơ lưu trữ gồm  $n$  từ máy kế tiếp ở bộ nhớ trong ( $a_1, a_2, \dots, a_n$ ) mỗi từ  $a_i$  lưu trữ một phần tử thứ  $i$  ( $1 \leq i \leq n$ ) của dãy số.

Ta cũng quy ước: dãy số được sắp xếp rồi vẫn để tại chỗ cũ như đã cho.

Vậy thì việc đặt "số nhỏ nhất" vừa được chọn, ở một lượt nào đó, vào cuối dãy đã được sắp xếp phải thực hiện bằng cách đổi chỗ với số hiện đang ở vị trí đó (nếu như nó khác số này).

Giả sử ta định hướng chương trình của ta vào ngôn ngữ tựa PASCAL, nêu ở chương 1, thì bước tinh chỉnh đầu tiên sẽ như sau:

**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 thấy có hai nhiệm vụ con, 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  $a_j$  với  $a_i$

Nhiệm vụ đầu có thể thực hiện bằng cách

*"Thoạt tiên coi  $a_i$  là "số nhỏ nhất" tạm thời; lần lượt so sánh  $a_i$  với  $a_{i+1}$ ,  $a_{i+2}$ , v.v... Khi thấy số nào nhỏ hơn thì lại coi đó là "số nhỏ nhất" mới. Khi đã so sánh với  $a_n$  rồi thì số nhỏ nhất sẽ được xác định".*

Nhưng xác định bằng cách nào?

- Có thể bằng cách chỉ ra chỗ của nó, nghĩa là nắm được chỉ số của phần tử ấy.

Ta có bước tinh chỉnh 2.1.

$j := i$

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

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

Với nhiệm vụ thứ hai thì có thể giải quyết theo cách tương tự như khi ta muốn chuyển hai thứ rượu trong hai ly, từ ly nọ sang ly kia: ta sẽ phải dùng một ly thứ ba (không đựng gì) để làm ly trung chuyển.

Ta có bước tinh chỉnh 2.2.

$B := a_i; \quad a_i := a_j; \quad a_j := B$

Sau khi đã chỉnh lại cách viết biến chỉ số cho đúng với quy ước, ta có chương trình sắp xếp dưới dạng thủ tục như sau:

**Procedure** SORT ( $A, n$ )

1- **For**  $i := 1$  **to**  $n$  **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}  $B := A[i]; A[i] := A[j]; A[j] := B$

**end**

4- **Return**

**Ví dụ 2.** Cho một ma trận vuông  $n \times n$  các số nguyên. Hãy in ra các phần tử thuộc các đường chéo song song với đường chéo chính.

Ví dụ: Cho ma trận vuông với  $n = 3$

$$\begin{bmatrix} 3 & 5 & 11 \\ 4 & 7 & 0 \\ 9 & 2 & 8 \end{bmatrix}$$

Giả sử ta chọn cách in từ phải sang trái, thì có kết quả:

$$\begin{array}{rcccc} & & & & 11 \\ & & & 5 & 0 \\ & & 3 & 7 & 8 \\ & 4 & 2 & & \\ 9 & & & & \end{array}$$

Ta sẽ hướng việc thể hiện giải thuật về một chương trình PASCAL.

Giải thuật có thể phác họa như sau:

- 1- Nhập  $n$
- 2- Nhập các phần tử của ma trận
- 3- In các đường chéo song song với đường chéo chính

Hai nhiệm vụ 1 và 2 có thể diễn đạt dễ dàng bằng PASCAL:

1. **Readln** ( $n$ );
2. **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 làm hai loại:

- Đường chéo ứng với cột từ  $n$  đến 1
- Đường chéo ứng với hàng từ 2 đến  $n$

Vì vậy ta tách thành 2 nhiệm vụ con:

- 3.1. **for**  $j:=n$  **down to** 1 **do**  
    in đường chéo ứng với cột  $j$ ;
- 3.2. **for**  $i:=2$  **to**  $n$  **do**  
    in đường chéo ứng với hàng  $i$ ;

Tới đây lại phải chi tiết hơn công việc

"in đường chéo ứng với cột  $j$ "

Với:  $j = n$  thì in một 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 chính là  $(n - j + 1)$ , còn phần tử được in chính là  $A[i, j + (i-1)]$  với  $i$  lấy giá trị từ 1 tới  $(n - j + 1)$

Vậy 3.1. có thể tinh chỉnh tiếp, tác vụ "in đường chéo ứng với cột  $j$ " thành:

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

Ở đây ta tận dụng khả năng của PASCAL để in mỗi phần tử trong một quãng 8 và mỗi đường chéo sẽ được in trên một dòng, sau đó để cách một dòng trống.

Với 3.2. cũng tương tự

```
for j := 1 to n - i + 1 do
    write (a[i + j - 1, j] : 8);
writeln;
```

Để có một chương trình PASCAL hoàn chỉnh tất nhiên ta phải tuân thủ mọi quy định của PASCAL: chẳng hạn trước khi bước vào phần câu lệnh thể hiện phần xử lý, phải có phần khai báo dữ liệu.

Ngoài ra ta có thể thêm vào những lời thuyết minh cho các bước (với một ngoại lệ là ta viết bằng tiếng Việt).

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 kích thước n của ma trận');
        readln (n);
    until (0 < n) and (n <= max);
```

```

writeln ('nhập phần tử ma trận');
    for i:= 1 to n do
        for j:= 1 to n do readln(a[i,j]);
writeln ('In các đường chéo');
for j:= n down to 1 do begin
    for i:= 1 to n - j + 1 do
        write (a[i, j+i-1] : 8);
    writeln;
        end;
for i:= 2 to n do begin
    for j:= 1 to n - i + 1 do
        write (a[i + j - 1, j] : 8);
    writeln;
        end;
end.

```

\* Trong các ví dụ nêu trên ta còn gặp các cấu trúc dữ liệu quen thuộc, đó là các vectơ và ma trận.

Đối với một số bài toán khác việc hình dung các dữ liệu và cấu trúc của chúng nhiều khi không còn đơn giản như thế nữa. Lựa chọn được mô hình thích hợp cho bài toán, trong đó có cả vấn đề ấn định một cấu trúc cho dữ liệu của nó, đòi hỏi một sự phân tích kỹ lưỡng dữ liệu cũng như yêu cầu đặt ra cho bài toán đó. Ta sẽ thấy điều này qua ví dụ sau:

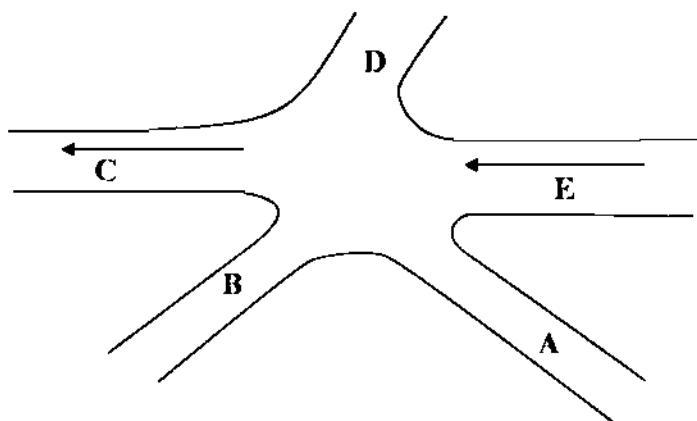
**Ví dụ 3.** Giả sử ta cần thiết lập một quy trình điều khiển đèn giao thông ở một chốt giao thông phức tạp, có nhiều giao lộ.

Như vậy nghĩa là điều khiển đèn báo sao cho trong một khoảng thời gian ấn định một số tuyến đường được thông, trong khi một số tuyến khác bị cấm, tránh xảy ra ùn tắc.

Đối với bài toán này ta thấy: Ta nhận ở đầu vào một số tuyến đường cho phép (tại chốt giao thông đó) và phải phân hoạch tập này thành một số ít nhất các nhóm, sao cho mọi tuyến trong một nhóm đều có thể cho thông đồng thời mà không xảy ra ùn tắc. Ta sẽ gán mỗi pha của việc điều khiển đèn giao thông với một nhóm trong phân hoạch này và việc tìm một phân hoạch với số nhóm ít nhất sẽ dẫn tới một quy trình điều khiển đèn với số pha ít nhất. Điều đó có nghĩa là thời gian chờ đợi tối đa để được thông cũng ít nhất.



Ví dụ như ở đầu mối sau:



Hình 2.4.

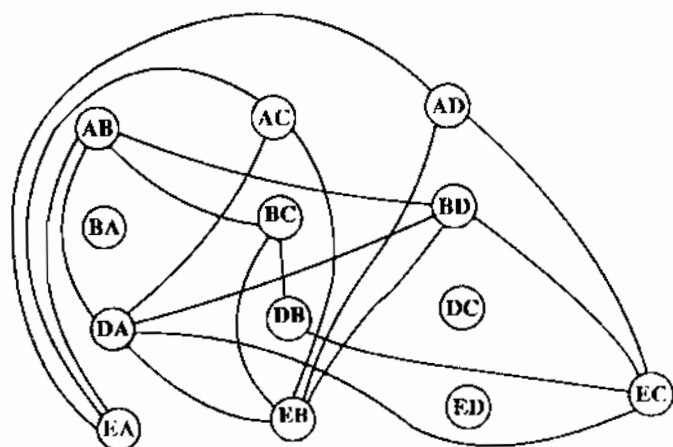
Ở đây C và E là đường một chiều (1 tuyến) còn các đường khác đều 2 chiều (2 tuyến).

Như vậy sẽ có 13 tuyến có thể thực hiện qua đầu mối này. Những tuyến như AB (từ A tới B), EC có thể thông đồng thời. Những tuyến như AD và EB thì không thể thông đồng thời được vì chúng giao nhau, có thể gây ra ùn ứ (ta sẽ gọi là các tuyến xung khắc). Như vậy đèn tín hiệu phải báo sao cho AD và EB không thể được thông cùng một lúc, trong khi đó lại cho phép AB và EC chẳng hạn, được thông đồng thời.

Ta có thể mô hình hoá bài toán của ta dựa vào một khái niệm, vốn đã được đề cập tới trong toán học, đó là *đồ thị* (graph).

Đồ thị bao gồm một tập các điểm gọi là đỉnh (vertices) và các đường nối các đỉnh gọi là cung (edges). Như vậy đối với bài toán "điều khiển đèn hướng dẫn giao thông" của ta thì có thể hình dung một đồ thị mà các đỉnh biểu thị cho các tuyến đường, còn cung là nối một cặp đỉnh ứng với 2 tuyến đường xung khắc.

Với một đầu mối giao thông như hình 2.4, đồ thị biểu diễn nó sẽ như sau:



Hình 2.5

Bây giờ ta sẽ tìm lời giải cho bài toán của ta dựa trên mô hình đồ thị đã nêu.

Ta sẽ đưa thêm vào khái niệm "tô màu cho đồ thị". Đó là việc gán màu cho mỗi đỉnh của đồ thị sao cho không có hai đỉnh nào nối với nhau bởi một cung lại cùng một màu.

Với khái niệm này, nếu ta hình dung mỗi màu đại diện cho một pha điều khiển đèn báo (cho thông một số tuyến và cấm một số tuyến khác) thì bài toán đang đặt ra chính là bài toán: tô màu cho đồ thị ứng với các tuyến đường ở một đầu mối, như đã quy ước ở trên (xem hình 2.5.) sao cho phải dùng ít màu nhất.

Bài toán *tô màu cho đồ thị* được nghiên cứu từ nhiều thập kỷ nay. Tuy nhiên bài toán tô màu cho một đồ thị bất kỳ, với số màu ít nhất lại thuộc vào một lớp khá rộng các bài toán, được gọi là "bài toán N - P đầy đủ", mà đối với chúng thì những lời giải hiện có chủ yếu thuộc loại "cố hết mọi khả năng". Trong trường hợp bài toán tô màu của ta thì "cố hết mọi khả năng" nghĩa là cố gán màu cho các đỉnh, trước hết bằng một màu đã, không thể được nữa thì mới dùng đến màu thứ hai, thứ ba... Cho tới khi đạt được mục đích, nghe ra thì có vẻ tầm thường, nhưng đối với bài toán loại này, đây lại chính là một giải thuật có hiệu lực thực tế. Vấn đề gay gắt nhất ở đây vẫn là khả năng tìm được lời giải tối ưu cho bài toán.

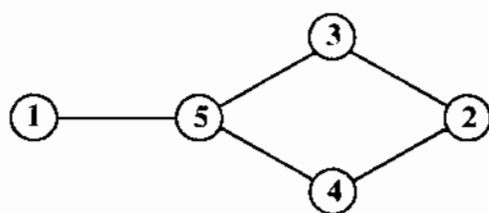
Nếu đồ thị nhỏ ta có thể cố thử theo mọi phương án, để có thể đi tới một lời giải tối ưu. Nhưng với đồ thị lớn thì cách làm này sẽ tốn phí rất nhiều thời gian, trên thực tế khó chấp nhận được. Cũng có thể có trường hợp do dựa vào một số thông tin phụ của bài toán mà việc tìm được lời giải tối ưu không cần phải thử tới mọi khả năng. Nhưng đó chỉ là những "đội may hiếm có". Còn một cách khác nữa là thay đổi cách nhìn nhận về lời

giải của bài toán đi đôi chút: Thay vì đi tìm lời giải tối ưu, ta đi tìm một lời giải "tốt" theo nghĩa là: nó đáp ứng được yêu cầu, trong một thời gian ngắn mà thực tế chấp nhận được. Một giải thuật "tốt" như vậy (tuy lời giải không phải là tối ưu) được gọi là *giải thuật heuristic*.

Một giải thuật heuristic hợp lý đối với bài toán tô màu cho đồ thị đã nêu là giải thuật sau đây mà nó được gán cho một cái tên khá ngộ nghĩnh là giải thuật "tham ăn" (greedy algorithm). Đầu tiên, ta cố tô màu cho các đỉnh nhiều hết mức có thể, bằng một màu. Với các đỉnh còn lại (chưa được tô) lại làm hết mức có thể với màu thứ hai, và cứ như thế.

Để tô màu cho các đỉnh với màu mới, ta sẽ thực hiện các bước sau:

- 1- Chọn một đỉnh chưa được tô màu nào đó và tô cho nó bằng màu mới.
- 2- Tìm trong danh sách các đỉnh chưa được tô màu, với mỗi đỉnh đó xác định xem có một cung nào nối nó với một đỉnh đã được tô bằng màu mới chưa. Nếu chưa có thì tô đỉnh đó bằng màu mới.



Hình 2.6

Cách tiếp cận này có cái tên là "tham ăn" vì nó thực hiện tô màu một đỉnh nào đó mà nó có thể tô được, không hề chú ý gì đến tình hình bất lợi có thể xuất hiện khi làm điều đó. Chẳng hạn, với đồ thị (hình 2.6) thì như sau:

Nếu nó tô màu xanh cho đỉnh ① thì theo thứ tự nó sẽ tìm đến ② và cũng tô luôn màu xanh. Như vậy thì ③, ④ sẽ phải tô đỏ, rồi ⑤ sẽ phải tô tím chẳng hạn, như vậy là phải dùng tới 3 màu. Còn nếu biết cân nhắc hơn, ta tô ①, ③, ④ màu xanh thì chỉ cần tô đỏ cho ② và ⑤ nghĩa là giảm bớt được một màu.

Bây giờ ta xét đến việc áp dụng giải thuật "tham ăn" cho đồ thị hình 2.5.:

Ta có thể tô xanh cho đỉnh ①, như vậy sẽ có thể tô xanh cho ②, ③, ④ nhưng không thể cho ⑤, ⑥, ⑦, ⑧; với ⑨ lại được, nhưng với ⑩, ⑪, ⑫ thì không. Cuối cùng cũng có thể tô xanh cho ⑬. Với màu thứ hai, màu đỏ, ta có thể tô cho BC, EC tất phải dùng màu vàng. Tóm lại ta dùng 4 màu, như trong bảng tóm tắt sau:

Tô màu	Cho tuyến	Số tuyến
Xanh	AB, AC, AD, BA, DC, EB	6
Đỏ	BC, BD, EA	3
Tím	DA, DB	2
Vàng	EB, EC	<u>2</u>
		13

Nếu liên hệ với quy trình điều khiển đèn ở chốt giao thông như hình 2.4, thì những kết quả trên là tương đương với 4 pha. Ở mỗi pha, chỉ các tuyến ứng với một màu trong bảng trên là được thông, còn các tuyến ứng với màu khác sẽ bị cấm. Điều đó cũng có nghĩa là cùng lắm mới phải chờ đến pha thứ tư. Giả sử mỗi pha mất 2 phút thì sau 6 phút tuyến EB và EC mới được thông, sau 8 phút tuyến AB, AC... mới lại được phục hồi và cứ như thế.

Như vậy ta đã chọn được mô hình và xác định được giải thuật xử lý dựa trên mô hình ấy.

Còn bây giờ ta bắt đầu đi vào các bước tinh chỉnh giải thuật "tham ăn", bằng giả ngôn ngữ, để rồi hướng tới một chương trình bằng PASCAL.

Giả sử ta gọi đồ thị được xét là G. Thủ tục "THAM\_AN" sau đây sẽ xác định chi tiết hơn các đỉnh sẽ cùng được tô màu mới, dưới dạng các phần tử của một tập newclr.

**Procedure THAM\_AN (var G: GRAPH; var newclr: SET);**

**Begin**

1. newclr :=  $\emptyset$ ; {Ký hiệu  $\emptyset$  chỉ tập rỗng}
  2. Với mỗi đỉnh V chưa được tô màu của G **do**
  3. **if** V không phải là lân cận của đỉnh nào trong newclr  
**then begin**
    4. Tô màu cho V;
    5. Gán thêm V vào tập newclr**end**
- end;**

Dĩ nhiên, so với thủ tục viết bằng ngôn ngữ tự nhiên ở trên thì thủ tục viết bằng giả ngôn ngữ này đã tiếp cận gần hơn với chương trình PASCAL.

Bây giờ ta cụ thể thêm một bước nữa vào những xử lý chi tiết. Chẳng hạn, trong bước 3 để nhận biết được một đỉnh V có là lân cận của đỉnh nào đó trong Newclr hay không ta sẽ phải giải quyết thế nào?

Ta có thể xét mỗi phần tử W của newclr và thử xem trong đồ thị G có

một cung nào nối giữa V và W không. Để kiểm tra như vậy ta sẽ sử dụng một biến logic Bool để đánh dấu: trị của Bool bằng true tức là có, bằng false tức là không.

Như vậy thì bước 3 có thể chi tiết hơn thành các bước 3.1. đến 3.5. như trong giải thuật sau:

**Procedure THAM\_AN (var G: GRAPH; var newclr: SET)**

**Begin**

1. Newclr :=  $\emptyset$ ;
2. Với mỗi đỉnh V của G chưa được tô màu **do begin**
  - 3.1. Bool := false;
  - 3.2. Với mỗi đỉnh W trong newclr **do**
  - 3.3. **if** có một cung giữa V và W
  - 3.4. **then** Bool := true;
  - 3.5. **if** Bool = false **then begin**
    4. Tô màu cho V;
    5. Gán thêm V vào tập newclr

**end**

**end**

**end;**

Thế là trong giải thuật của ta đã xuất hiện các phép toán tác động trên hai tập đỉnh. Chu trình ngoài 2-5 lặp trên tập các đỉnh chưa được tô màu của G. Còn chu trình trong 3.3 -3.4 lặp trên các đỉnh hiện có trong newclr.

Có nhiều cách để biểu diễn tập hợp trong một ngôn ngữ lập trình như PASCAL, ta có thể biểu diễn tập các đỉnh newclr một cách đơn giản bởi một cấu trúc gọi là *danh sách* (list) được cài đặt cụ thể bởi một vectơ các số nguyên, kết thúc bởi một giá trị null đặc biệt (có thể dùng giá trị 0). Mỗi phần tử của vectơ này là một số nguyên đại diện cho một đỉnh (chẳng hạn số thứ tự gán cho đỉnh đó). Với cách hình dung như vậy ta có thể thay câu lệnh 3.2. bởi một câu lệnh chu trình, mà W lúc đầu là phần tử đầu tiên của newclr, sau đó lại chuyển sang phần tử bên cạnh mỗi khi được lặp lại. Với câu lệnh 2 cũng tương tự.

Chương trình chi tiết hơn có thể viết như sau:

**Procedure THAM\_AN (var G:GRAPH; var newclr : SET)**

**var**

Bool : **Boolean**;

V, W: **integer**;

**begin**

newclr := 0;

V := đỉnh đầu tiên chưa được tô màu trong G;

**While** V < > null **do begin**

    Bool := false

    W := đỉnh đầu tiên trong newclr;

**While** W < > null **do begin**

        if có một cung giữa V và W trong G

**then** Bool := true;

        W := đỉnh lân cận trong newclr

**end;**

**if** Bool = false **the begin**

        Tô màu cho V;

        gán thêm V vào newclr

**end**

V := đỉnh chưa được tô màu lân cận trong G

**end**

**end;**

Cứ như vậy, qua nhiều bước tinh chỉnh ta sẽ cụ thể thêm chi tiết đối với các phép xử lý cũng như đối với cấu trúc dữ liệu và sự cài đặt của nó. Vì chương trình ứng với giải thuật này dài nên ta dừng lại ở đây, mà không tiếp tục triển khai nữa.

## 2.2 Phân tích giải thuật

### 2.2.1 Đặt vấn đề

Khi ta đã xây dựng được giải thuật và chương trình tương ứng, để giải một bài toán thì có thể có hàng loạt yêu cầu về phân tích được đặt ra. Chẳng hạn: yêu cầu phân tích *tính đúng đắn* của giải thuật, liệu nó có thể hiện được đúng lời giải của bài toán không? Thông thường, người ta có thể cài đặt chương trình thể hiện giải thuật đó trên máy và thử nghiệm nó nhờ một số bộ dữ liệu nào đấy rồi so sánh kết quả thử nghiệm với kết quả mà ta đã biết. Nhưng cách thử này chỉ phát hiện được tính sai chứ chưa thể đảm bảo được tính đúng của giải thuật. Với các công cụ toán học người ta cũng có thể chứng minh được tính đúng đắn của giải thuật nhưng công việc này không phải là dễ dàng, ta cũng không đặt vấn đề đi sâu thêm ở đây.

Loại yêu cầu thứ hai là về *tính đơn giản* của giải thuật. Thông thường ta vẫn mong muốn có được một giải thuật đơn giản, nghĩa là dễ hiểu, dễ lập trình, dễ chỉnh lý. Nhưng cách đơn giản để giải một bài toán chưa hẳn lúc nào cũng là cách tốt. Thường thường nó hay gây ra tốn phí thời gian hoặc bộ nhớ khi thực hiện. Đối với chương trình chỉ để dùng một vài lần thì tính đơn giản này cần được coi trọng vì như ta đã biết công sức và thời gian để xây dựng được chương trình giải một bài toán thường rất lớn so với thời gian thực hiện chương trình đó. Nhưng nếu chương trình sẽ được sử dụng nhiều lần, nhất là đối với loại bài toán mà khối lượng dữ liệu đưa vào khá lớn, thì thời gian thực hiện rõ ràng phải được chú ý. Lúc đó yêu cầu đặt ra lại là tốc độ, hơn nữa khối lượng dữ liệu quá lớn mà dung lượng bộ nhớ lại có giới hạn thì không thể bỏ qua yêu cầu về tiết kiệm bộ nhớ được. Tuy nhiên cân đối giữa yêu cầu về thời gian và không gian không mấy khi có được một giải pháp trọn vẹn.

Sau đây ta sẽ chú ý đến việc phân tích thời gian thực hiện giải thuật, một trong các tiêu chuẩn để đánh giá hiệu lực của giải thuật vốn hay được đề cập tới.

## 2.2.2 Phân tích thời gian thực hiện giải thuật

Với một bài toán, không phải chỉ có một giải thuật. Chọn một giải thuật đưa tới kết quả nhanh là một đòi hỏi thực tế. Nhưng, căn cứ vào đâu để có thể nói được: giải thuật này nhanh hơn giải thuật kia?

Có thể thấy ngay: thời gian thực hiện một giải thuật (hay chương trình thể hiện giải thuật đó) phụ thuộc vào rất nhiều yếu tố. Một yếu tố cần chú ý trước tiên đó là kích thước của dữ liệu đưa vào. Chẳng hạn thời gian sắp xếp một dãy số phải chịu ảnh hưởng của số lượng các số thuộc dãy số đó. Nếu gọi  $n$  là số lượng này (kích thước của dữ liệu vào) thì thời gian thực hiện  $T$  của một giải thuật phải được biểu diễn như một hàm của  $n$ :  $T(n)$ .

Các kiểu lệnh và tốc độ xử lý của máy tính, ngôn ngữ viết chương trình và chương trình dịch ngôn ngữ ấy đều ảnh hưởng tới thời gian thực hiện; nhưng những yếu tố này không đồng đều với mọi loại máy trên đó cài đặt giải thuật, vì vậy không thể dựa vào chúng khi xác lập  $T(n)$ . Điều đó cũng có nghĩa là  $T(n)$  không thể được biểu diễn thành đơn vị thời gian bằng giây, bằng phút... được. Tuy nhiên, không phải vì thế mà không thể so sánh được các giải thuật về mặt tốc độ. Nếu như thời gian thực hiện của một giải thuật là  $T_1(n) = cn^2$  và thời gian thực hiện một giải thuật khác  $T_2(n) = kn$  (với  $c$  và  $k$  là một hằng số nào đó), thì khi  $n$  khá lớn, thời gian thực hiện giải thuật  $T_2$  rõ ràng ít hơn so với giải thuật  $T_1$ . Và như vậy thì nếu nói thời gian thực hiện giải thuật  $T(n)$  tỉ lệ với  $n^2$  hay tỷ lệ với  $n$  cũng cho ta ý niệm về tốc độ thực hiện giải thuật đó khi  $n$  khá lớn (với  $n$  nhỏ thì việc xét  $T(n)$  không có ý nghĩa). Cách đánh giá thời gian thực hiện giải thuật độc lập với máy tính và

các yếu tố liên quan tới máy như vậy sẽ dẫn tới khái niệm về "cấp độ lớn của thời gian thực hiện giải thuật" hay còn gọi là "độ phức tạp về thời gian của giải thuật".

### 2.2.2.1 Độ phức tạp về thời gian của giải thuật

Nếu thời gian thực hiện một giải thuật là  $T(n) = cn^2$  (với  $c$  là hằng số) thì ta nói: Độ phức tạp về thời gian của giải thuật này có cấp là  $n^2$  (hay cấp độ lớn của thời gian thực hiện giải thuật là  $n^2$ ) và ta ký hiệu

$$T(n) = O(n^2) \quad (\text{ký hiệu chữ O lớn})$$

Một cách tổng quát có thể định nghĩa:

Một hàm  $f(n)$  được xác định là  $O(g(n))$

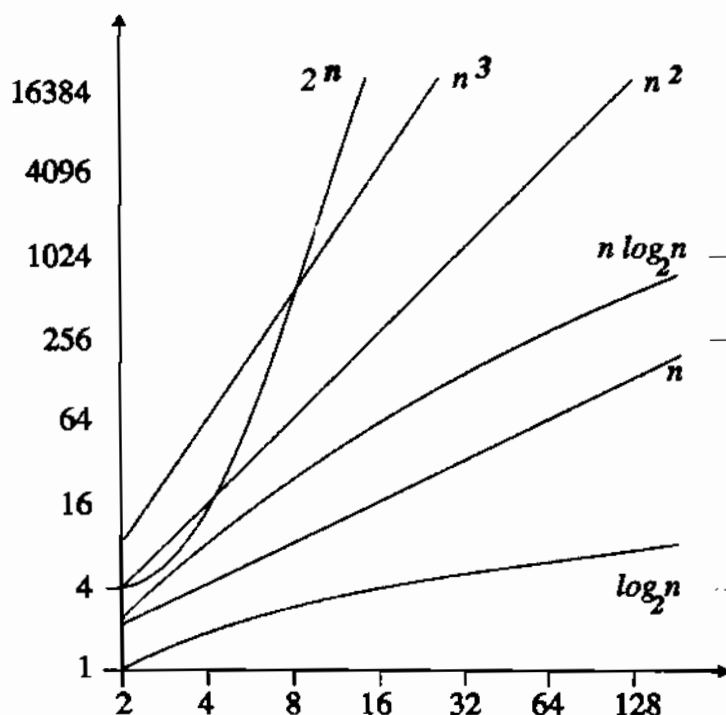
$$f(n) = O(g(n))$$

và được gọi là có cấp  $g(n)$  nếu tồn tại các hằng số  $c$  và  $n_0$  sao cho

$$f(n) \leq cg(n) \text{ khi } n \geq n_0$$

nghĩa là  $f(n)$  bị chặn trên bởi một hằng số nhân với  $g(n)$ , với mọi giá trị của  $n$  từ một điểm nào đó. Thông thường các hàm thể hiện độ phức tạp về thời gian của giải thuật có dạng:  $\log_2 n$ ,  $n$ ,  $n \log_2 n$ ,  $n^2$ ,  $n^3$ ,  $2^n$ ,  $n!$ ,  $n^n$

Sau đây là đồ thị và bảng giá trị của một số hàm đó





$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2.147.483.648

Các hàm như  $2^n$ ,  $n!$ ,  $n^n$  được gọi là hàm loại mũ. Một giải thuật mà thời gian thực hiện của nó có cấp là các hàm loại mũ thì tốc độ rất chậm. Các hàm như  $n^3$ ,  $n^2$ ,  $n \log_2 n$ ,  $n$ ,  $\log_2 n$  được gọi là các hàm loại đa thức. Giải thuật với thời gian thực hiện có cấp hàm đa thức thì thường chấp nhận được.

### 2.2.2.2 Xác định độ phức tạp về thời gian

Xác định độ phức tạp về thời gian của một giải thuật bất kỳ có thể dẫn tới những bài toán phức tạp. Tuy nhiên, trong thực tế, đối với một số giải thuật ta cũng có thể phân tích được bằng một số quy tắc đơn giản.

\* **Quy tắc tổng:** Giả sử  $T_1(n)$  và  $T_2(n)$  là thời gian thực hiện của hai đoạn chương trình  $P_1$  và  $P_2$  mà  $T_1(n) = O(f(n))$ ;  $T_2(n) = O(g(n))$  thì thời gian thực hiện  $P_1$  và  $P_2$  kế tiếp nhau sẽ là:

$$T_1(n) + T_2(n) = O(\max(f(n), g(n)))$$

**Ví dụ:** Trong một chương trình có 3 bước thực hiện mà thời gian thực hiện từng bước lần lượt là  $O(n^2)$ ,  $O(n^3)$  và  $O(n \log_2 n)$  thì thời gian thực hiện 2 bước đầu là  $O(\max(n^2, n^3)) = O(n^3)$ . Thời gian thực hiện chương trình sẽ là  $O(\max(n^3, n \log_2 n)) = O(n^3)$ .

- Một ứng dụng khác của quy tắc này là nếu  $g(n) \leq f(n)$  với mọi  $n \geq n_0$  thì  $O(f(n) + g(n))$  cũng là  $O(f(n))$ . Chẳng hạn:  $O(n^4 + n^2) = O(n^4)$  và  $O(n + \log_2 n) = O(n)$ .

\* **Quy tắc nhân:** Nếu tương ứng với  $P_1$  và  $P_2$  là  $T_1(n) = O(f(n))$ ,  $T_2(n) = O(g(n))$  thì thời gian thực hiện  $P_1$  và  $P_2$  lồng nhau sẽ là:

$$T_1(n) T_2(n) = O(f(n)g(n))$$

**Ví dụ:** Câu lệnh gán:  $x := x + 1$  có thời gian thực hiện bằng  $c$  (hằng số) nên được đánh giá là  $O(1)$ .

Câu lệnh: **for**  $i := 1$  **to**  $n$  **do**  $x := x + 1$ ;

có thời gian thực hiện  $O(n.1) = O(n)$

Câu lệnh: **for**  $i := 1$  **to**  $n$  **do**

**for**  $j := 1$  **to**  $n$  **do**  $x := x + 1$ ;

có thời gian thực hiện được đánh giá là

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

- Cũng có thể thấy  $O(cf(n)) = O(f(n))$

$$\text{chẳng hạn } O(n^2/2) = O(n^2)$$

(Phần chứng minh hai quy tắc trên xin dành cho độc giả).

**Chú ý:** Dựa vào những nhận xét đã nêu ở trên về các quy tắc khi đánh giá thời gian thực hiện giải thuật ta chỉ cần chú ý tới các bước tương ứng với một phép toán mà ta gọi là *phép toán tích cực* (active operation) đó là phép toán thuộc giải thuật mà thời gian thực hiện nó không ít hơn thời gian thực hiện các phép khác (tất nhiên phép toán tích cực không phải là duy nhất), hay nói một cách khác: số lần thực hiện nó không kém gì các phép khác.

Bây giờ ta xét tới một vài giải thuật cụ thể:

Giải thuật tính giá trị của  $e^x$  theo công thức gần đúng:

$$e^x \approx 1 + x/1 + x^2/2 + \dots + x^n/n! \quad \text{với } x \text{ và } n \text{ cho trước.}$$

### Program EXP1

{ Tính từng số hạng rồi cộng lại }

1.        **Read**(x); S:= 1;
2.        **for** i:= 1 **to** n **do begin**  
          p := 1  
          **for** j:= 1 **to** i **do** p := p\*x/j;  
          S := S + p  
          **end**
3.    **end.**

Ta có thể coi phép toán tích cực ở đây là phép

$$p := p * x/j$$

Ta thấy nó được thực hiện:

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

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

Ta có thể viết giải thuật theo một cách khác

### Program EXP2

{ Dựa vào số hạng trước để tính số hạng sau }

$$\text{theo cách } \frac{x^2}{2!} = \frac{x}{1!} \cdot \frac{x}{2}; \dots; \frac{x^n}{n!} = \frac{x^{n-1}}{(n-1)!} \cdot \frac{x}{n} \}$$

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 thời gian thực hiện lại là:

$$T(n) = O(n)$$

vì phép  $p := p*x/i$  chỉ thực hiện n lần.

### 2.2.2.3 Độ phức tạp về thời gian trung bình

Có những trường hợp thời gian thực hiện giải thuật không phải chỉ phụ thuộc vào kích thước của dữ liệu vào mà còn phụ thuộc vào chính tình trạng của dữ liệu đó nữa.

Chẳng hạn: sắp xếp một dãy số theo thứ tự tăng dần, nếu gặp dãy số đưa vào đã có đúng thứ tự sắp xếp rồi thì sẽ khác với trường hợp dãy số đưa vào chưa có thứ tự hoặc có thứ tự ngược lại. Lúc đó khi phân tích thời gian thực hiện giải thuật ta sẽ phải xét tới: đối với mọi dữ liệu vào có kích thước n thì  $T(n)$  trong trường hợp thuận lợi nhất là thế nào? rồi  $T(n)$  trong trường hợp xấu nhất? và  $T(n)$  trung bình? Việc xác định  $T(n)$  trung bình thường khó vì sẽ phải dùng tới những công cụ toán đặc biệt, hơn nữa tính trung bình có thể có nhiều cách quan niệm. Trong các trường hợp mà  $T(n)$  trung bình khó xác định người ta thường đánh giá giải thuật qua giá trị xấu nhất của  $T(n)$ .

Qua giải thuật sau đây, ta có thể thấy rõ hơn.

#### Program SEARCH

{Cho vector V có n phần tử, giải thuật này thực hiện tìm trong V một phần tử có giá trị bằng X cho trước}

1. Found := **false**; {Found là biến logic để báo hiệu việc  
ngừng tìm khi đã thấy}

i := 1;

2. **while** i ≤ 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  $V[i]$  với  $X$ . Có thể thấy số lần phép toán tích cực này thực hiện phụ thuộc vào chỉ số  $i$  mà  $V[i] = X$ . Trường hợp thuận lợi nhất xảy ra khi  $X$  bằng  $V[1]$ : một lần thực hiện.

Trường hợp xấu nhất: khi  $X$  bằng  $V[n]$  hoặc không tìm thấy:  $n$  lần thực hiện.

Vậy:  $T_{\text{tốt}} = O(1)$

$T_{\text{xấu}} = O(n)$

Thời gian trung bình được đánh giá thế nào?

Muốn trả lời ta phải biết được xác suất mà  $X$  rơi vào một phần tử nào đó của  $V$ . Nếu ta giả thiết khả năng này là đồng đều với mọi phần tử của  $V$  (đồng khả năng) thì có thể xét như sau:

Gọi  $q$  là xác suất để  $X$  rơi vào một phần tử nào đó của  $V$  thì xác suất để  $X$  rơi vào phần tử  $V[i]$  là:  $p_i = \frac{q}{n}$ , còn xác suất để  $X$  không rơi vào phần tử nào (nghĩa là không thấy) sẽ là  $1 - q$ .

Thời gian thực hiện trung bình sẽ là:

$$\begin{aligned} T_{\text{tb}}(n) &= \sum_{i=1}^n p_i \cdot i + (1 - q)n \\ &= \sum_{i=1}^n \frac{q}{n} i + (1 - q)n \\ &= \frac{q}{n} \frac{n(n+1)}{2} + (1 - q)n \\ &= q \frac{(n+1)}{2} + (1 - q)n \end{aligned}$$

Nếu  $q = 1$  (nghĩa là luôn tìm thấy) thì  $T_{\text{tb}} = \frac{n+1}{2}$

Nếu  $q = \frac{1}{2}$  (khả năng tìm thấy và không tìm thấy bằng nhau)

$$\text{thì } T_{\text{tb}}(n) = \frac{n+1}{4} + \frac{n}{2} = \frac{3n+1}{4}$$

Nói chung  $T_{\text{tb}}(n) = O(n)$

Nếu ta lấy trường hợp xấu nhất để đánh giá thì thấy cũng là  $O(n)$ .

## BÀI TẬP CHƯƠNG 2

- 2.1. Việc chia bài toán ra thành các bài toán nhỏ có những thuận lợi gì?
- 2.2. Nêu nguyên tắc của phương pháp thiết kế từ đỉnh xuống (thiết kế kiểu top-down). Cho ví dụ minh họa.
- 2.3. Người ta có thể thực hiện giải bài toán theo kiểu từ đáy lên (thiết kế kiểu bottom - up) nghĩa là đi từ các vấn đề cụ thể trước, sau đó mới ghép chúng lại thành một vấn đề lớn hơn. Cho ví dụ thực tế thể hiện cách thiết kế này và thử nhận xét so sánh nó với cách thiết kế kiểu top-down.
- 2.4. Tóm tắt ý chủ đạo của phương pháp tinh chỉnh từng bước.
- 2.5. Có 6 đội bóng A, B, C, D, E, F thi đấu để tranh giải vô địch (vòng đấu)
- Đội A đã đấu với B và C  
Đội B đã đấu với D và F  
Đội E đã đấu với C và F
- Mỗi đội chỉ đấu với đội khác 1 trận trong 1 tuần. Hãy lập lịch thi đấu sao cho các trận còn lại sẽ được thực hiện trong một số ít tuần nhất.
- 2.6. Hãy nêu một giải thuật mà độ phức tạp về thời gian của nó là  $O(1)$ .
- 2.7. Giải thích tại sao  $T(n) = O(n)$  thì cũng sẽ đúng khi viết  $T(n) = O(n^2)$ .
- 2.8. Với mỗi hàm  $f(n)$  sau đây hãy tìm hàm  $g(n)$  (trong dãy các hàm đã nêu ở mục 2.2.21)) nhỏ nhất để sao cho

$$f(n) = O(g(n))$$

a)  $f(n) = (2 + n) * (3 + \log_2 n)$

b)  $f(n) = 11 * \log_2 n + n/2 - 3452$

c)  $f(n) = n * (3 + n) - 7 * n$

d)  $f(n) = \log_2(n^2) + n$

e)  $f(n) = \frac{(n+1) * \log_2(n+1) - (n+1) + 1}{n}$

- 2.9. Với các đoạn chương trình dưới đây hãy xác định độ phức tạp về thời gian của giải thuật bằng ký pháp chữ O lớn:

a) Sum := 0;

for i:= 1 to n do begin

readln (x);

Sum := Sum + x

```

        end;
b) for i := 1 to n do
    for j:= 1 to n do begin
        C[i,j] := 0;
        for k:= 1 to n do
            C[i,j] := C[i,j] + A[i,k] * B[k,j]
        end;
c) for i:= 1 to n -1 do begin
    for j:= n-1 down to i do
        if X[j] > X[j+1] then begin
            temp := X[j]
            X[j] := X[j+1];
            X[j +1] := Temp
        end;
    end;
end;

```

# GIẢI THUẬT ĐỆ QUI

## 3.1 Khái niệm về đệ qui

Ta nói một đối tượng là đệ qui (recursive algorithm) nếu nó bao gồm chính nó như một bộ phận hoặc nó được định nghĩa dưới dạng của chính nó.

**Ví dụ:** Trên vô tuyến truyền hình có lúc ta thấy có những hình ảnh đệ qui: phát thanh viên ngồi bên máy vô tuyến truyền hình, trên màn hình của máy này lại có chính hình ảnh của phát thanh viên ấy ngồi bên máy vô tuyến truyền hình và cứ như thế...

Trong toán học ta cũng hay gặp các định nghĩa đệ qui.

### 1. Số tự nhiên

- a) 1 là một số tự nhiên
- b)  $x$  là số tự nhiên nếu  $x - 1$  là số tự nhiên.

### 2. Hàm $n$ giai thừa: $n!$

- a)  $0! = 1$
- b) Nếu  $n > 0$  thì  $n! = n(n-1)!$

## 3.2 Giải thuật đệ qui và thủ tục đệ qui

Nếu lời giải của bài toán  $T$  được thực hiện bằng lời giải của một bài toán  $T'$ , có dạng giống như  $T$ , thì đó là một lời giải đệ qui. Giải thuật tương ứng với lời giải như vậy gọi là *giải thuật đệ qui*.

Thoạt nghe thì có vẻ hơi lạ, nhưng điểm mấu chốt cần lưu ý là:  $T'$  tuy có dạng giống như  $T$ , nhưng theo một nghĩa nào đó, nó phải "nhỏ" hơn  $T$ .

Hãy xét bài toán tìm một từ trong một quyển từ điển. Có thể nêu giải thuật như sau:

**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.

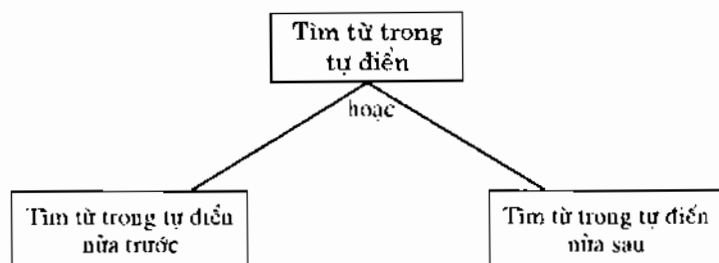
**else** tìm từ đó trong nửa sau.

**end;**

Tất nhiên giải thuật trên mới chỉ được nêu dưới dạng thô, còn nhiều chỗ chưa cụ thể, chẳng hạn:

- Tìm từ trong một trang thì làm thế nào?
- Thế nào là mở từ điển vào trang giữa?
- Làm thế nào để biết từ đó nằm ở nửa nào của từ điển?...

Trả lời rõ những câu hỏi trên không phải là khó, nhưng ta sẽ không sa vào các chi tiết này mà muốn tập trung vào việc xét chiến thuật của lời giải. Có thể hình dung chiến thuật tìm kiếm này một cách khái quát như hình 3.1.



Hình 3.1

Ta thấy có hai điểm chính cần lưu ý:

- Sau mỗi lần từ điển được tách đôi thì một nửa thích hợp sẽ lại được tìm kiếm bằng một chiến thuật như đã dùng trước đó.
- Có một trường hợp đặc biệt, khác với mọi trường hợp trước, sẽ đạt được sau nhiều lần tách đôi, đó là trường hợp từ điển chỉ còn duy nhất một trang. Lúc đó việc tách đôi ngừng lại và bài toán đã trở thành đủ nhỏ để ta có thể giải quyết trực tiếp bằng cách tìm từ mong muốn trên trang đó chẳng hạn, bằng cách tìm tuần tự. Trường hợp đặc biệt này được gọi là *trường hợp suy biến* (degenerate case).



Có thể coi đây là chiến thuật kiểu "chia để trị" (divide and conquer). Bài toán được tách thành bài toán nhỏ hơn và bài toán nhỏ hơn lại được giải quyết với chiến thuật chia để trị như trước, cho tới khi xuất hiện trường hợp suy biến.

Bây giờ ta hãy thể hiện giải thuật tìm kiếm này dưới dạng một thủ tục.

#### **Procedure SEARCH (dict, word)**

{dict được coi là đầu mối để truy nhập được vào tự điển đang xét, word chỉ từ cần tìm}

1. **if** tự điển chỉ còn là một trang.  
    **then** Tìm từ word trong trang này.  
    **else begin**
2. Mở tự điển vào trang "giữa"  
    Xác định xem nửa nào của tự điển chứa từ word;  
    **if** word nằm ở nửa trước của tự điển.  
        **then call** SEARCH (dict 1, word)  
        **else call** SEARCH (dict 2, word)  
    **end;**

{dict 1 và dict 2 là đầu mối để truy nhập được vào nửa trước và nửa sau của tự điển}

#### **3. Return**

Thủ tục như trên được gọi là thủ tục đệ qui. Có thể nêu ra mấy đặc điểm sau:

- a. Trong thủ tục đệ qui có lời gọi đến chính thủ tục đó. Ở đây: trong thủ tục SEARCH có **call** SEARCH.
- b. Mỗi lần có lời gọi lại thủ tục thì kích thước của bài toán đã thu nhỏ hơn trước. Ở đây khi có **call** SEARCH thì kích thước tự điển chỉ còn bằng một "nửa" trước đó.
- c. Có một trường hợp đặc biệt: trường hợp suy biến. Đó chính là trường hợp mà tự điển chỉ còn là một trang. Khi trường hợp này xảy ra thì bài toán còn lại sẽ được giải quyết theo một cách khác hẳn và gọi đệ qui cũng kết thúc. Chính tình trạng kích thước của bài toán cứ giảm dần sẽ đảm bảo dẫn tới trường hợp suy biến.

Một số ngôn ngữ cấp cao chẳng hạn C, PASCAL... cho phép viết các thủ tục đệ qui. Nếu thủ tục chứa lời gọi đến chính nó, như thủ tục SEARCH ở trên, thì nó được gọi là *đệ qui trực tiếp* (directly recursive). Cũng có dạng thủ tục chứa lời gọi đến thủ tục khác mà ở thủ tục này lại chứa lời gọi đến nó. Trường hợp này gọi là *đệ qui gián tiếp* (indirectly recursive).

### 3.3 Thiết kế giải thuật đệ qui

Khi bài toán đang xét hoặc dữ liệu đang xử lý được định nghĩa dưới dạng đệ qui thì việc thiết kế các giải thuật đệ qui tỏ ra rất thuận lợi. Hầu như nó phản ánh rất sát nội dung của định nghĩa đó.

Có thể thấy điều này qua một số bài toán sau:

#### 3.3.1 Hàm $n!$

Định nghĩa đệ qui của  $n!$  có thể nhắc lại

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{Factorial}(n - 1) & \text{if } n > 0 \end{cases}$$

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

**Function** FACTORIAL (n)

1- **if**  $n = 0$  **then** FACTORIAL := 1.

**else** FACTORIAL :=  $n * \text{FACTORIAL}(n-1)$

2. **return**

Đối chiếu với 3 đặc điểm của thủ tục đệ qui nêu ở trên ta thấy:

- Lời gọi tới chính nó ở đây nằm trong câu lệnh gán đứng sau **else**.

- Ở mỗi lần gọi đệ qui đến FACTORIAL, thì giá trị của  $n$  giảm đi 1.

Ví dụ, FACTORIAL (4) gọi đến FACTORIAL (3), FACTORIAL (3) gọi đến FACTORIAL (2), FACTORIAL (2) gọi đến FACTORIAL (1), FACTORIAL (1) gọi đến FACTORIAL (0) - FACTORIAL (0) chính là trường hợp suy biến, nó được tính theo cách đặc biệt FACTORIAL (0) = 1.

#### 3.3.2 Dãy số Fibonacci

Dãy số Fibonacci bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ. Bài toán được đặt ra như sau:

- 1) Các con thỏ không bao giờ chết.
- 2) Hai tháng sau khi ra đời một cặp thỏ mới sẽ sinh ra một cặp thỏ con (một đực, một cái).
- 3) Khi đã sinh con rồi thì cứ mỗi tháng tiếp theo chúng lại sinh 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?

Ví dụ  $n = 6$ , ta thấy.

- Tháng thứ 1: 1 cặp (cặp ban đầu).
- Tháng thứ 2: 1 cặp (cặp ban đầu vẫn chưa đẻ)
- Tháng thứ 3: 2 cặp (đã có thêm một cặp con)
- Tháng thứ 4: 3 cặp (cặp đầu vẫn đẻ thêm)
- Tháng thứ 5: 5 cặp (cặp con bắt đầu đẻ)
- Tháng thứ 6: 8 cặp (cặp con vẫn đẻ tiếp).

Bây giờ ta xét tới việc tính số cặp thỏ ở tháng thứ  $n$ :  $F(n)$ .

Ta thấy nếu mỗi cặp thỏ ở tháng thứ  $(n-1)$  đều sinh con thì :

$$F(n) = 2 * (n-1).$$

Nhưng không phải như vậy. Trong các cặp thỏ ở tháng thứ  $(n-1)$  chỉ có những cặp đã có ở tháng thứ  $(n-2)$  mới sinh con ở tháng thứ  $n$  được thỏ.

Do đó:  $F(n) = F(n-2) + F(n-1)$

Vì vậy có thể tính  $F(n)$  theo:

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

Dãy số thể hiện  $F(n)$  ứng với các giá trị của  $n = 1, 2, 3, \dots$  có dạng:

1      1      2      3      5      8      13      21      34      55...

Nó được gọi là *dãy số Fibonacci*. Nó là mô hình của rất nhiều hiện tượng tự nhiên và cũng được sử dụng nhiều trong tin học. Ta sẽ thấy một số ứng dụng của nó ở chương sau trong giáo trình này.

Sau đây là thủ tục đệ qui 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**

Ở đây chỉ có một chi tiết hơi khác là trường hợp suy biến ứng với hai giá trị  $F(1) = 1$  và  $F(2) = 1$ .

### 3.3.3 Chú ý

Đối với hai bài toán nêu trên việc thiết kế các giải thuật đệ qui tương ứng khá thuận lợi vì cả hai đều thuộc dạng tính giá trị hàm mà định nghĩa đệ qui của hàm đó xác định được dễ dàng.

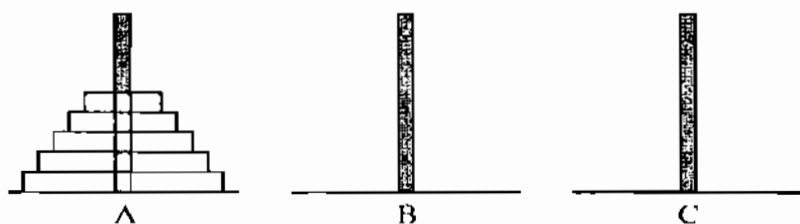
Nhưng không phải lúc nào tính đệ qui trong cách giải bài toán cũng thể hiện rõ nét và đơn giản như vậy. Thế thì vấn đề gì cần lưu tâm khi thiết kế một giải thuật đệ qui? Có thể thấy câu trả lời qua việc giải đáp các câu hỏi sau:

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 được giảm đi ở mỗi lần gọi đệ qui?
3. Trường hợp đặc biệt nào của bài toán sẽ được coi là trường hợp suy biến?  
Sau đây ta xét thêm một vài bài toán phức tạp hơn.

### 3.3.4 Bài toán "Tháp Hà Nội" (Tower of Hanoi)

Đây là một bài toán mang tính chất một trò chơi, nội dung như sau:

Có  $n$  đĩa, kích thước nhỏ dần, đĩa có lỗ ở giữa (như đĩa hát). Có thể xếp chồng chúng lên nhau xuyên qua một cọc, to dưới nhỏ trên để cuối cùng có một chồng đĩa dạng như hình tháp (hình 3.2).



Hình 3.2

Yêu cầu đặt ra là:

Chuyển chồng đĩa từ cọc A sang cọc khác, chẳng hạn 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 (dù là tạm thời).
- 3- Được phép sử dụng một cọc trung gian, chẳng hạn cọc B để đặt tạm đĩa (gọi là cọc trung gian) khi chuyển từ cọc A sang cọc C.

Để đi tới cách giải tổng quát, trước hết xét vài trường hợp đơn giản.

\* Trường hợp một đĩa:

- Chuyển đĩa từ cọc A sang cọc C.

\* Trường hợp hai đĩa:

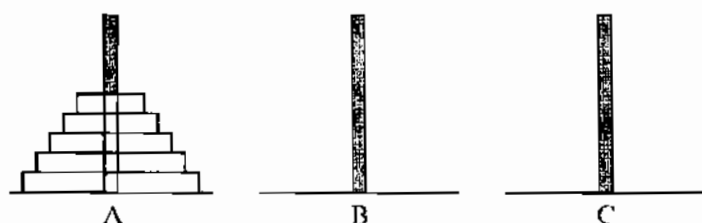
- Chuyển đĩa thứ nhất từ cọc A sang cọc B.

- Chuyển đĩa thứ hai từ cọc A sang cọc C.
- Chuyển đĩa thứ nhất từ cọc B sang cọc C.

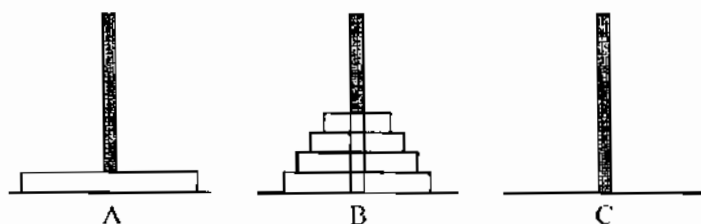
Ta thấy với trường hợp  $n$  đĩa ( $n > 2$ ) nếu coi  $(n-1)$  đĩa ở trên, đóng vai trò như đĩa thứ nhất thì có thể xử lý giống như trường hợp 2 đĩa được, nghĩa là:

- Chuyển  $(n-1)$  đĩa trên từ A sang B.
- Chuyển đĩa thứ  $n$  từ A sang C.
- Chuyển  $(n-1)$  đĩa từ B sang C.

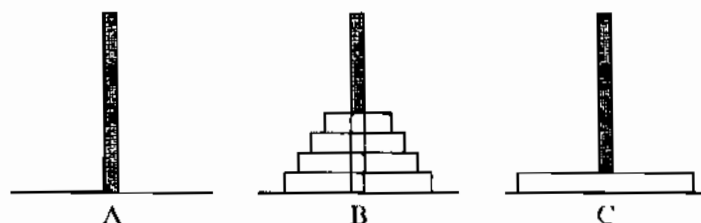
Có thể hình dung việc thể hiện 3 bước này theo mô hình như sau:



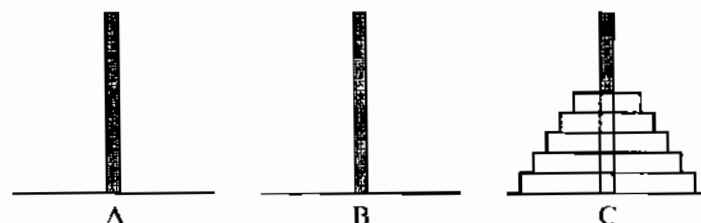
Bước 1



Bước 2



Bước 3



Hình 3.3

Như vậy, bài toán "Tháp Hà Nội" tổng quát với  $n$  đĩa được dẫn đến bài toán tương tự với kích thước nhỏ hơn, chẳng hạn từ chỗ chuyển  $n$  đĩa từ cọc A sang cọc C nay là chuyển  $(n-1)$  đĩa từ cọc A sang cọc B. Và ở mức này thì giải thuật lại 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 B sang cọc C

và cứ như thế cho tới khi trường hợp suy biến xảy ra, đó là trường hợp ứng với bài toán chuyển 1 đĩa thôi.

Vậy thì các đặc điểm của đệ qui trong giải thuật đã được xác định và ta có thể viết giải thuật đệ qui của bài toán "Tháp Hà Nội" như sau:

**Procedure HANOI** ( $n, A, B, C$ )

1- if  $n = 1$  then chuyển đĩa từ A sang C

2- else begin

    callHANOI ( $n - 1, A, C, B$ );

    callHANOI(1, A, B, C);

    callHANOI ( $n - 1, B, A, C$ )

end

3- return

### 3.3.5 Bài toán 8 quân hậu và giải thuật quay lui (back tracking)

Một bàn cờ quốc tế là một bảng hình vuông gồm 8 hàng, 8 cột. Quân hậu là một quân cờ có thể ăn được bất kỳ quân nào nằm trên cùng một hàng, cùng một cột hay cùng một đường chéo. Bài toán đặt ra là: hãy xếp 8 quân hậu trên bàn cờ sao cho không có quân hậu nào có thể ăn được quân hậu nào. Điều đó cũng có nghĩa là trên mỗi hàng, mỗi cột, mỗi đường chéo chỉ có thể có một quân hậu thôi.

Dĩ nhiên ta không nên tìm lời giải bằng cách xét mọi trường hợp ứng với mọi vị trí của 8 quân hậu trên bàn cờ rồi lọc ra các trường hợp chấp nhận được. Khuynh hướng "thử từng bước", thoát nghe có vẻ hơi lạ, nhưng lại thể hiện một giải pháp hiện thực: nó cho phép tìm ra tất cả các cách sắp xếp để không có quân hậu nào ăn được nhau (số lượng cách sắp xếp này là 92).

Nét đặc trưng của phương pháp là ở chỗ các bước đi tới lời giải hoàn toàn được làm thử. Nếu có một lựa chọn được chấp nhận thì ghi nhớ các thông tin cần thiết và tiến hành bước thử tiếp theo. Nếu trái lại không có một lựa chọn nào thích hợp cả thì làm lại bước trước, xoá bớt các ghi nhớ và quay về chu trình thử với các lựa chọn còn lại. Hành động này được gọi là *quay lui*, và các giải thuật thể hiện phương pháp này gọi là các *giải thuật quay lui*.

Đối với bài toán 8 quân hậu: do mỗi cột chỉ có thể có một quân hậu nên lựa chọn đối với quân hậu thứ  $j$ , ứng với cột  $j$ , là đặt nó vào hàng nào để đảm bảo "an toàn" nghĩa là không cùng hàng, cùng đường chéo với  $(j-1)$  quân hậu đã được xếp trước đó. Rõ ràng để đi tới các lời giải ta phải thử tất cả các trường hợp sắp xếp quân hậu đầu tiên tại cột 1. Với mỗi vị trí như vậy ta lại phải giải quyết bài toán 7 quân hậu với phần còn lại của bàn cờ, nghĩa là ta đã "quay lại bài toán cũ"! Tính chất đệ quy của bài toán đó thể hiện trong phép thử này. Ta có thể phác thảo thủ tục thử như sau:

### Procedure TRY ( $j$ )

1. Khởi phát việc chọn vị trí cho quân hậu thứ  $j$ .
2. **repeat** thực hiện phép chọn tiếp theo;

**if** an toàn **then begin**

đặt quân hậu;

**if**  $j < 8$  **then begin**

**call** TRY ( $j+1$ )

**if** không thành công.

**then** cất quân hậu.

**end**

**end**

**until** thành công **or** hết chỗ.

3. **return**

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

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

Như trên đã nêu, đối với quân hậu thứ  $j$ , vị trí của nó chỉ chọn trong cột thứ  $j$ . Vậy tham biến  $j$  trở thành chỉ số cột và việc chọn lựa được tiến hành trên 8 giá trị của chỉ số hàng  $i$ .

Để lựa chọn  $i$  được chấp nhận, thì hàng  $i$  và hai đường chéo qua ô  $(i;j)$  phải còn tự do (không có quân hậu nào khác ở trên đó). Chú ý rằng trong hai đường chéo thì đường chéo theo chiều  $\uparrow$  có các ô  $(i; j)$  mà tổng  $i + j$  không đổi, còn đường chéo theo chiều  $\downarrow$  có các ô  $(i; j)$  mà  $i - j$  không đổi (hình 3.4).

Do đó ta sẽ chọn các mảng một chiều Boolean để biểu diễn các tình trạng này:

$a[i] = \text{true}$  có nghĩa là không có quân hậu nào chiếm hàng  $i$ .

$b[i + j] = \text{true}$  có nghĩa là không có quân hậu nào chiếm được đường chéo  $i + j$

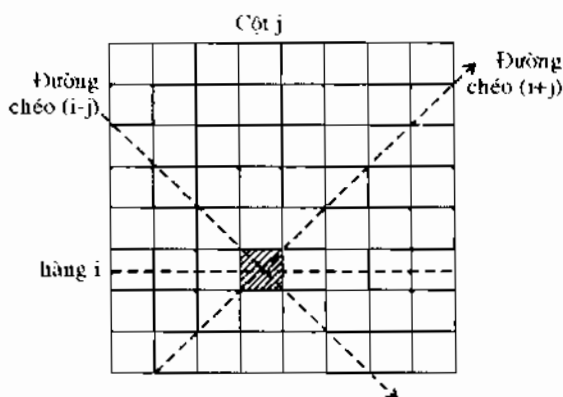
$c[i - j] = \text{true}$  có nghĩa là không có quân hậu nào chiếm được đường chéo  $i - j$ .

Ta biết:  $1 \leq i, j \leq 8$

nên suy ra  $1 \leq j \leq 8$

$$2 \leq i + j \leq 16$$

$$-7 \leq i - j \leq 7$$



Hình 3.4

Như vậy điều kiện để lựa chọn  $i$  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

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

Còn cất quân hậu thì bởi:

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

Thủ tục TRY bây giờ có thể viết

**Procedure** TRY( $j, q$ )

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 v(i = 8)
3.  return

```

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

#### **Program EIGHT-QUEEN 1**

```

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 một lời giải}
    call TRY (1,q);
3- {in kết quả}
    if q then for i := 1 to 8 do write (x[i]);
    end

```

Còn nếu ta muốn có tất cả các lời giải của bài toán thì cần sửa đổi đôi chút trong thủ tục TRY:

- Điều kiện kết thúc của quá trình chọn rút lại còn  $j = 8$  thôi, nên câu lệnh **repeat** được thay bởi câu lệnh **for**

- Việc in kết quả được thực hiện ngay sau khi có một lời giải vì vậy ta viết dưới dạng một thủ tục để gọi nó ngay trong thủ tục TRY.

#### **Procedure PRINT(x)**

```

    for k := 1 to 8 do write (x[k])
return

```

Như vậy, dạng mới của TRY 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 call** TRY(j + 1)  
        **else call** PRINT(x);  
        a[i] := true;  
        b[i + j] := true;  
        c[i - j] := true;  
    **end**
2. **return**

Chương trình cho toàn bộ các lời giải của bài toán sẽ là:

**Program EIGHT-QUEEN**

1. **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. **call** TRY (1)
- end**

Sau đây là lời giải của 12 trong số 92 lời giải của bài toán:

$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	$x_6$	$x_7$	$x_8$
1	5	8	6	3	7	2	4
1	6	8	3	7	4	2	5
1	7	4	6	8	2	5	3
1	7	5	8	2	4	6	3
2	4	6	8	3	1	7	5
2	5	7	1	3	8	6	4
2	5	7	4	1	8	6	3
2	6	1	7	4	8	3	5
2	6	8	3	1	4	7	5
2	7	3	6	8	5	1	4
2	7	5	8	1	4	6	3
2	8	6	1	3	5	7	4

### 3.4 Hiệu lực của đệ qui

Qua các ví dụ trên ta có thể thấy: đệ qui là một công cụ để giải các bài toán. Có những bài toán, bên cạnh giải thuật đệ qui vẫn có những giải thuật lặp khá đơn giản và hữu hiệu. Chẳng hạn giải thuật lặp tính  $n!$  có thể viết:

```
Function IFAC (n)  
1.      if  $n = 0$  or  $n = 1$  then begin  
         $IFACT := 1$  ;  
        return  
                                end;  
2.       $IFACT := 1$ ;  
        for  $i := 2$  to  $n$  do  $IFACT := IFACT * i$   
3.      return
```

hay giải thuật lặp tính số Fibonacci:

```
Function FIBONACCI (n)  
1.      if  $\leq 2$  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
```

Tuy vậy, đệ qui vẫn có vai trò xứng đáng của nó. Có những bài toán việc nghĩ ra lời giải đệ qui thuận lợi hơn nhiều so với lời giải lặp và có những giải thuật đệ qui thực sự cũng có hiệu lực cao nữa, chẳng hạn giải thuật sắp xếp kiểu phân đoạn (Quick sort) mà ta sẽ xét tới trong chương 9.

Một điều nữa cần nói thêm là: về mặt định nghĩa, công cụ đệ qui đã cho phép xác định một tập vô hạn các đối tượng bằng một phát biểu hữu hạn. Ta sẽ thấy vai trò của công cụ này trong định nghĩa văn phạm, định nghĩa cú pháp ngôn ngữ, định nghĩa một số cấu trúc dữ liệu.v.v...

**Chú thích:** Khi thay các giải thuật đệ qui bằng các giải thuật không tự gọi chúng, như giải thuật lặp nêu trên, ta gọi là *khử đệ qui*.

### 3.5 Đệ qui và qui nạp toán học

Có một mối quan hệ giữa đệ qui và qui nạp toán học. Cách giải đệ qui một bài toán dựa trên việc định rõ lời giải cho trường hợp suy biến rồi thiết kế làm sao để lời giải của bài toán được suy từ lời giải của bài toán nhỏ hơn, cùng loại như thế.

Tương tự như vậy, qui nạp toán học chứng minh một tính chất nào đó ứng với số tự nhiên cũng bằng cách chứng minh tính chất ấy đúng đối với một số trường hợp cơ sở (chẳng hạn với  $n = 1$ ) và rồi chứng minh tính chất ấy đúng với  $n$  bất kỳ, nếu nó đã đúng với các số tự nhiên nhỏ hơn  $n$ .

Do đó, ta sẽ không lầy lăm lã khi thấy qui nạp toán học được dùng để chứng minh các tính chất có liên quan đến giải thuật đệ qui. Chẳng hạn, sau đây ta sẽ chứng minh tính đúng đắn của giải thuật đệ qui FACTORIAL và biểu thức đánh giá số lần di chuyển đĩa trong giải thuật tháp HANOI.

#### 3.5.1 Tính đúng đắn của giải thuật FACTORIAL

Ta muốn chứng minh rằng thủ tục hàm FACTORIAL( $n$ ) như đã nêu ở trên sẽ cho giá trị:

$$\text{FACTORIAL}(0) = 1$$

$$\text{FACTORIAL}(n) = n * (n-1) * \dots * 2 * 1 \text{ (nếu } n > 0)$$

Ta thấy: trong thủ tục có chỉ thị

$$\text{if } n = 0 \text{ then FACTORIAL} := 1$$

chúng tỏ thủ tục đã đúng với  $n = 0$  (cho ra giá trị bằng 1).

Giả sử nó đã đúng với  $n = k$ , nghĩa là với FACTORIAL( $k$ ) thủ tục cho ra giá trị bằng

$$k * (k-1) * \dots * 2 * 1$$

Bây giờ ta sẽ chứng minh nó đúng với  $n = k + 1$

Với  $n = k + 1 > 0$  chỉ thị đứng sau else

$$\text{FACTORIAL} := n * \text{FACTORIAL}(n-1)$$

sẽ được thực hiện. Vậy với  $n = k + 1$  thì giá trị cho ra là  $(k + 1) * \text{FACTORIAL}(k)$  nghĩa là  $(k + 1) * k * (k-1) * \dots * 2 * 1$  và việc chứng minh đã hoàn tất.

#### 3.5.2 Đánh giá giải thuật Tháp Hà Nội

Ta xét xem với  $n$  đĩa thì số lần di chuyển đĩa sẽ là bao nhiêu? (Ta coi phép di chuyển đĩa trong giải thuật này là phép toán tích cực).

Giả sử gọi  $Moves(n)$  là số đó, ta có  $Moves(1) = 1$ .

Khi  $n > 1$  thì  $Moves(n)$  không thể tính trực tiếp như vậy, nhưng ta biết rằng: nếu biết  $Moves(n-1)$  thì ta sẽ tính được  $Moves(n)$ :

$$Moves(n) = Moves(n-1) + Moves(1) + Moves(n-1)$$

(dựa vào giải thuật).

Vậy ta đã xác định được mối quan hệ truy hồi của cách tính:

$$Moves(1) = 1$$

$$Moves(n) = 2 * Moves(n-1) + 1, \text{ nếu } n > 1$$

**Ví dụ:**

$$\begin{aligned} Moves(3) &= 2 * Moves(2) + 1 \\ &= 2 * [2 * Moves(1) + 1] + 1 \\ &= 2 [2 * 1 + 1] + 1 \\ &= 7 \end{aligned}$$

Tuy nhiên công thức truy hồi này không cho ta tính trực tiếp theo  $n$  được. Nếu để ý ta thấy  $Moves(1) = 1 = 2^1 - 1$

$$Moves(2) = 3 = 2^2 - 1$$

$$Moves(3) = 7 = 2^3 - 1$$

Vậy phải chăng

$$Moves(n) = 2^n - 1 \text{ với } n \text{ là số tự nhiên bất kỳ?}$$

Ta sẽ chứng minh công thức tính này là đúng, bằng qui nạp toán học.

Với  $n = 1$ ,  $Moves(1) = 2^1 - 1 = 1$  công thức đã đúng.

Giả sử công thức đã đúng với  $n = k$ , nghĩa là:

$$Moves(k) = 2^k - 1$$

Với  $n = k + 1$  theo công thức tính truy hồi

$$\begin{aligned} Moves(k+1) &= 2 * Moves(k) + 1 \\ &= 2 * (2^k - 1) + 1 \\ &= 2^{k+1} - 2 + 1 = 2^{k+1} - 1 \end{aligned}$$

Như vậy công thức đã đúng với  $k + 1$ . Do đó có thể kết luận công thức vẫn đúng với mọi  $n$ .

## BÀI TẬP CHƯƠNG 3

### 3.1. Xét định nghĩa đệ qui:

$$\text{Acker}(m, n) = \begin{cases} n + 1 & \text{nếu } m = 0 \\ \text{Acker}(m - 1, 1) & \text{nếu } n = 0 \\ \text{Acker}(m - 1, \text{Acker}(m, n - 1)) & \text{với các trường hợp khác} \end{cases}$$

Hàm này được gọi là hàm Ackermann. Nó có đặc điểm là giá trị của nó tăng rất nhanh, ứng với giá trị nguyên của  $m$  và  $n$ .

- Hãy xác định  $\text{Acker}(1, 2)$
- Viết một thủ tục đệ qui thực hiện tính giá trị của hàm này.

### 3.2. Giải thuật tính ước số chung lớn nhất của hai số nguyên dương $p$ và $q$ ( $p > q$ ) được mô tả như sau:

Gọi  $r$  là số dư trong phép chia  $p$  cho  $q$ .

- Nếu  $r = 0$  thì  $q$  là ước số chung lớn nhất.
- Nếu  $r \neq 0$  thì gán cho  $p$  giá trị của  $q$ , gán cho  $q$  giá trị của  $r$  rồi lặp lại quá trình.

- a) Hãy xây dựng một định nghĩa đệ qui cho hàm  $\text{USCLN}(p, q)$ .
- b) Viết một giải thuật đệ qui và một giải thuật lặp thể hiện hàm đó.
- c) Hãy nêu rõ các đặc điểm của một giải thuật đệ qui được thể hiện trong trường hợp này.
- d) Trường hợp người ta nhầm cho giá trị  $q$  lớn hơn  $p$  thì giải thuật có xử lý được không?

### 3.3. Hàm $C(n, k)$ với $n, k$ là các giá trị nguyên không âm và $k \leq n$ , được định nghĩa:

$$C(n, n) = 1$$

$$C(n, 0) = 1$$

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \text{ nếu } 0 < k < n$$

Viết một thủ tục đệ qui thực hiện tính giá trị  $C(n, k)$  khi biết  $n, k$ .

### 3.4. Hãy nêu rõ các bước thực hiện khi có lời gọi $\text{call HANOI}(3, A, B, C)$ .

### 3.5. Viết một thủ tục đệ qui thực hiện in ngược một dòng ký tự cho trước.

Ví dụ cho dòng "PASCAL" thì in ra "LACSAP".

### 3.6. Viết một thủ tục đệ qui nhằm in ra tất cả các hoán vị của $n$ phần tử của một dãy số $a = \{a_1, a_2, \dots, a_n\}$ .

Ví dụ:  $n = 3, a_1 = 1, a_2 = 2, a_3 = 3$ ; thì in ra:

1 2 3; 1 3 2; 2 1 3; 2 3 1; 3 1 2; 3 2 1;

(Gợi ý: Hãy để ý nhận xét:

1 2 3	1 3 2	2 3 1;
2 1 3	3 1 2	3 2 1

## **PHẦN II**

# **CẤU TRÚC DỮ LIỆU**

## MẢNG VÀ DANH SÁCH

### 4.1 Các khái niệm

Không có gì lạ khi cấu trúc dữ liệu đầu tiên mà ta nói tới là cấu trúc mảng vì đó là cấu trúc rất quen thuộc ở mọi ngôn ngữ lập trình.

*Mảng* (array) là một tập có thứ tự gồm một số cố định các phần tử. Không có phép bổ sung phần tử hoặc loại bỏ phần tử được thực hiện đối với mảng. Thường chỉ có các phép tạo lập (create) mảng, tìm kiếm (retrieve) một phần tử của mảng, cập nhật (update) một phần tử của mảng... Ngoài giá trị, một phần tử của mảng còn được đặc trưng bởi chỉ số (index) thể hiện thứ tự của phần tử đó trong mảng. Vector là mảng một chiều, mỗi phần tử  $a_i$  của nó ứng với một chỉ số  $i$ . Ma trận là mảng hai chiều, mỗi phần tử  $a_{ij}$  ứng với hai chỉ số  $i$  và  $j$ . Tương tự người ta cũng mở rộng ra: mảng ba chiều, mảng bốn chiều..., mảng  $n$  chiều.

*Danh sách* (list) có hơi khác với mảng ở chỗ: nó là một tập có thứ tự nhưng bao gồm một số biến động các phần tử. Phép bổ sung và phép loại bỏ một phần tử là phép thường xuyên tác động lên danh sách. Tập hợp các người đến khám bệnh cho ta hình ảnh một danh sách. Họ sẽ được khám theo một thứ tự. Số người có lúc tăng lên (do có người mới đến), có lúc giảm đi (do bỏ về vì không chờ lâu được). Một danh sách mà quan hệ lân cận giữa các phần tử được hiển thị ra thì được gọi là *danh sách tuyến tính* (linear list). Vector chính là trường hợp đặc biệt của danh sách tuyến tính, đó là hình ảnh của danh sách tuyến tính xét tại một thời điểm nào đấy. Như vậy danh sách tuyến tính là một danh sách hoặc rỗng (không có phần tử nào) hoặc có dạng  $(a_1, a_2, \dots, a_n)$  với  $a_i$  ( $1 \leq i \leq n$ ) là các dữ liệu *nguyên tử*. Trong danh sách tuyến tính tồn tại một phần tử đầu  $a_1$ , phần tử cuối  $a_n$ . Đối với mỗi phần tử  $a_i$  bất kỳ với  $1 \leq i \leq n - 1$  thì có một phần tử  $a_{i+1}$  gọi là *phần tử sau*  $a_i$ , và với  $2 \leq i \leq n$  thì có một phần tử  $a_{i-1}$  gọi là *phần tử trước*  $a_i$ .  $a_i$  được gọi là phần tử thứ  $i$  của danh sách tuyến tính,  $n$  được gọi là độ dài hoặc kích thước của danh sách, nó có giá trị thay đổi.



Mỗi phần tử trong một danh sách thường là một bản ghi (gồm một hoặc nhiều *trường* (fiels)) đó là phần thông tin nhỏ nhất có thể "tham khảo" được trong một ngôn ngữ lập trình. Ví dụ: danh mục điện thoại là một danh sách tuyến tính, mỗi phần tử của nó ứng với một đơn vị thuê bao, nó gồm ba trường:

- Tên đơn vị hoặc tên chủ hộ thuê bao;
- Địa chỉ;
- Số điện thoại;

Đối với một danh sách, ngoài phép bổ sung và loại bỏ, còn một số phép sau đây cũng hay được tác động:

- *Glép* 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 ấn định;
- *Tìm kiếm* trong danh sách một phần tử mà một trường nào đó có một giá trị ấn định.
- v.v...

Nhân đây cũng cần nói đến một khái niệm, đó là *tệp* (file). *Tệp* là một loại danh sách có kích thước lớn được lưu trữ ở bộ nhớ ngoài (chẳng hạn đĩa từ). Phần tử của tệp là *bản ghi* (records) nó bao gồm nhiều trường dữ liệu, tương ứng với các thuộc tính khác nhau. Ví dụ: tệp hồ sơ nhân sự của cán bộ trong một cơ quan. Phần lý lịch của mỗi cán bộ là một bản ghi, nó bao gồm các trường dữ liệu tương ứng với các thuộc tính như: Họ và tên, ngày sinh, nơi sinh, quê quán, trình độ văn hoá...

Khác với các bộ nhớ trong, bộ nhớ ngoài có những đặc điểm riêng, do đó xử lý tệp cũng cần có những kỹ thuật riêng, ta không đề cập tới ngay ở đây mà sẽ xét ở những chương cuối cùng.

Còn bây giờ, chủ yếu các vấn đề mà ta sắp bàn luận tới là điều liên quan đến bộ nhớ trong. Ta có thể hình dung bộ nhớ này như một dãy có thứ tự các từ máy (words) mà ứng với nó là một địa chỉ. Mỗi từ máy thường chứa từ 8 đến 64 bit, việc tham khảo đến nội dung của nó thông qua địa chỉ.

Thường có hai cách để xác định được địa chỉ của một phần tử trong danh sách. Cách thứ nhất là dựa vào những đặc tả của dữ liệu cần tìm. Địa chỉ thuộc loại này thường được gọi là *địa chỉ được tính* (computed address). Cách này thường hay được sử dụng trong các ngôn ngữ lập trình để tính địa chỉ các phần tử của vector, của ma trận; để tính địa chỉ lệnh thực hiện tiếp theo trong quá trình thực hiện chương trình đích. Cách thứ hai là lưu trữ các địa chỉ cần thiết ấy ở một chỗ nào đó trong bộ nhớ, khi cần xác định sẽ lấy

ở đó ra. Loại địa chỉ này được gọi là *con trỏ* (pointer) hoặc *mối nối* (link). Địa chỉ quay lui để quay trở về chỗ gọi ở chương trình chính, khi kết thúc chương trình con, chính là loại địa chỉ này, cũng có một số cấu trúc đòi hỏi sự phối hợp của cả hai loại địa chỉ nói trên.

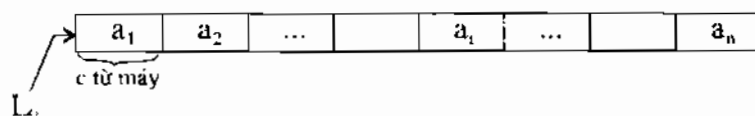
Sau đây, ta sẽ xét tới cách tổ chức lưu trữ mảng.

## 4.2 Cấu trúc lưu trữ của mảng

Cấu trúc dữ liệu đơn giản nhất dùng địa chỉ được tính để thực hiện lưu trữ và tìm kiếm phần tử, là mảng một chiều hay vector.

Thông thường một số từ máy kế tiếp sẽ được giành ra để lưu trữ các phần tử của mảng (vì vậy người ta gọi là cách *lưu trữ kế tiếp* - sequential storage allocation). Xét trường hợp mảng một chiều hay vector có  $n$  phần tử và mỗi phần tử của nó có thể lưu trữ được trong một từ máy thì cần phải dành cho nó  $n$  từ máy kế tiếp nhau. Do kích thước của vector đã được xác định nên không gian nhớ dành ra cũng đã được ấn định trước.

Một cách tổng quát, 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 như sau:



Hình 4.1

Địa chỉ của  $a_i$  sẽ được tính bởi:

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

$L_0$  được gọi là địa chỉ gốc - đó là địa chỉ của từ máy đầu tiên trong miền nhớ kế tiếp dành để lưu trữ vector (mà ta gọi là vector lưu trữ).

$f(i) = c*(i-1)$  gọi là *hàm địa chỉ* (address function)

Trong ngôn ngữ như PASCAL, cận dưới của chỉ số không nhất thiết phải là 1, mà có thể là một số nguyên  $b$  nào đó. Trường hợp đó địa chỉ của  $a_i$  được tính bởi:

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

Đối với mảng nhiều chiều, việc tổ chức lưu trữ cũng được thực hiện tương tự nghĩa là vẫn bằng vector lưu trữ kế tiếp như trên.

Ví dụ trong FORTRAN một 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:

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

Hình 4.2

Ta thấy cách lưu trữ ở trên đây là theo cột, "hết cột này đến cột khác" vì vậy người ta gọi lưu trữ theo *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 có n hàng, m cột thì

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

Trong ngôn ngữ như PASCAL, cách lưu trữ các phần tử của ma trận lại theo *thứ tự ưu tiên hàng* (row major order) nghĩa là "hết hàng này đến hàng khác".

Với ma trận có n hàng, m cột thì công thức tính địa chỉ sẽ là

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

Trường hợp cận dưới của chỉ số không phải là 1, nghĩa là ứng với  $a_{ij}$  thì  $b_1 \leq i \leq u_1$ ,  $b_2 \leq j \leq u_2$ , ta sẽ có:

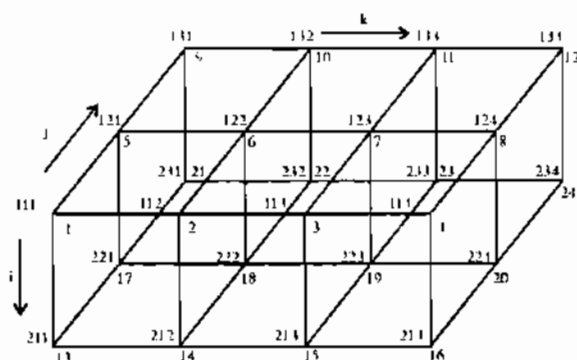
$$\text{Loc}(a_{ij}) = L_0 + (i - b_1) * (u_2 - b_2 + 1) + (j - b_2)$$

vì mỗi hàng có  $(u_2 - b_2 + 1)$  phần tử.

\* Bây giờ ta thử xét tới mảng ba chiều. Giả sử mảng B có các phần tử  $b_{ijk}$  với  $1 \leq i \leq 2$ ;  $1 \leq j \leq 3$ ;  $1 \leq k \leq 4$ ; được lưu trữ theo thứ tự ưu tiên hàng thì các phần tử của nó sẽ được sắp xếp kế tiếp như sau:

$b_{111}, b_{112}, b_{113}, b_{114}, b_{121}, b_{122}, b_{123}, b_{124}, b_{211}, b_{212}, b_{213}, b_{214}, b_{221}, b_{222}, b_{223}, b_{224}, b_{231}, b_{232}, b_{233}, b_{234}$

Có thể hình dung trong không gian bởi hình 4.3 sau:



Hình 4.3.

Công thức tính địa chỉ sẽ là:

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

**Ví dụ:**

$$\text{Loc}(b_{233}) = L_0 + 22$$

Xét trường hợp tổng quát với mảng  $n$  chiều  $A$  mà phần tử là  $A[s_1, s_2, \dots, s_n]$  trong đó  $b_i \leq s_i \leq u_i$  ( $i = 1, 2, \dots, n$ ), ứng với thứ tự ưu tiên hàng, ta sẽ có:

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

với

$$p_i = \prod_{k=i+1}^n (u_k - b_k + 1)$$

mà đặc biệt  $p_n = 1$

Đối với thứ tự ưu tiên cột công thức cũng tương tự.

**\* Chú ý:**

- 1) Khi mảng được lưu trữ kế tiếp thì việc truy nhập vào phần tử của mảng được thực hiện trực tiếp dựa vào địa chỉ được tính nên tốc độ nhanh và đồng đều đối với mọi phần tử.
- 2) Mặc dầu có rất nhiều ứng dụng ở đó mảng có thể được sử dụng để thể hiện mối quan hệ về cấu trúc giữa các phần tử dữ liệu, nhưng không phải không có những trường hợp mà mảng cũng lộ rõ những nhược điểm của nó. Ta thử xét tới một trường hợp như vậy.

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

**Ví dụ:** Cộng  $(3x^2 - xy + y^2 + 2y - x)$

với  $(x^2 + 4xy - y^2 + 2x)$

để có kết quả là  $(4x^2 + 3xy + 2y + x)$

Ta biết rằng khi thực hiện phép cộng hai đa thức ta phải tìm kiếm từng số hạng, phải phân biệt được các biến, hệ số và mũ trong từng số hạng đó. Ta phải biểu diễn như thế nào để phép toán trên được thuận tiện và có hiệu quả?

Với đa thức của 2 biến  $x, y$  như trên ta có thể dùng ma trận để biểu diễn: hệ số của số hạng  $x^i y^j$  sẽ được lưu trữ ở phần tử thuộc hàng  $i$  cột  $j$  của ma trận (giả sử cận dưới của chỉ số là 0).

Như vậy nếu ta hạn chế kích thước của ma trận  $5 \times 5$  thì số mũ cao nhất của  $x$  hoặc  $y$  chỉ có thể là 4 nghĩa là chỉ xử lý được với đa thức bậc 4 của  $x$  và  $y$  thôi.

**Ví dụ:** Với  $x^2 + 4xy - y^2 + 2x$  thì ma trận biểu diễn nó sẽ có dạng

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

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

Với cách biểu diễn này việc thực hiện phép cộng hai đa thức chỉ còn là phép cộng hai ma trận thôi. Như vậy cách biểu diễn này đã đưa tới phép xử lý khá đơn giản. Tuy nhiên ta có thể thấy rõ một số nhược điểm.

Như trên đã nói: số mũ trong đa thức bị hạn chế bởi kích thước của ma trận, do đó lớp các đa thức được xử lý bị giới hạn trong một phạm vi hẹp. Ngoài ra còn thấy thêm: ma trận biểu diễn có rất nhiều phần tử bằng không, khuynh hướng này tạo ra sự lãng phí bộ nhớ rất rõ (dạng ma trận này được gọi là *ma trận thưa* - sparse matrices). Để khắc phục những nhược điểm này cần có một cách tổ chức lưu trữ khác, ta sẽ xét ở sau.

### 4.3 Lưu trữ kế tiếp đối với danh sách tuyến tính

Qua phần trên ta cũng thấy: có thể dùng mảng một chiều làm cấu trúc lưu trữ của danh sách tuyến tính nghĩa là có thể dùng một vector lưu trữ ( $V_i$ ) với  $1 \leq i \leq n$  để lưu trữ một danh sách tuyến tính ( $a_1, a_2, \dots, a_n$ ): phần tử  $a_i$  được chứa ở  $V_i$ .

Nhưng do số phần tử của danh sách tuyến tính thường biến động, nghĩa là kích thước  $n$  thay đổi, nên việc lưu trữ chỉ có thể đảm bảo được nếu biết được  $m = \max(n)$ . Nhưng điều này thường không dễ xác định chính xác mà chỉ là dự đoán. Vì vậy nếu  $\max(n)$  lớn thì khả năng lãng phí bộ nhớ càng nhiều vì có thể có hiện tượng "giữ chỗ để dấy" mà không dùng tới. Hơn nữa, ngay khi đã dự trữ đủ rồi thì việc bổ sung hay loại bỏ phần tử trong danh sách mà không phải là phần tử cuối sẽ đòi hỏi phải dón hoặc dãn danh sách, nghĩa là dịch chuyển một số phần tử lùi xuống (để lấy chỗ bổ sung) hoặc tiến lên (để lấp chỗ các phần tử đã loại bỏ) và điều này sẽ gây tổn phí thời gian không ít nếu các phép toán này được thực hiện thường xuyên.

Tuy nhiên với cách lưu trữ này, như đã nêu ở trên, ưu điểm về tốc độ truy nhập lại rất rõ.

## 4.4 Lưu trữ móc nối đối với danh sách tuyến tính

Lưu trữ kế tiếp đối với danh sách tuyến tính đã bộc lộ rõ nhược điểm trong trường hợp thực hiện thường xuyên các phép bổ sung hoặc loại bỏ phần tử, trường hợp xử lý đồng thời nhiều danh sách hoặc trường hợp các ma trận thưa.v.v...

Việc sử dụng con trỏ hoặc mối nối để tổ chức danh sách tuyến tính, mà ta gọi là danh sách móc nối, chính là một giải pháp nhằm khắc phục các nhược điểm đó. Sau đây ta sẽ xét tới cấu trúc của danh sách móc nối.

### 4.4.1 Nguyên tắc

Ở đây mỗi phần tử của danh sách được lưu trữ trong một phần tử nhớ mà ta gọi là nút (node). Mỗi nút bao gồm một số từ máy kế tiếp. Các nút này có thể nằm bất kỳ ở chỗ nào trong bộ nhớ. Trong mỗi nút, ngoài phần thông tin ứng với một phần tử, còn chứa địa chỉ của phần tử đứng sau nó trong danh sách. Quy cách của mỗi nút có thể hình dung như sau:

INFO	LINK
------	------

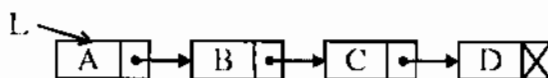
Trường INFO chứa thông tin của phần tử.


Trường LINK chứa địa chỉ (mối nối) nút tiếp theo.

Riêng nút cuối cùng thì không có nút đứng sau nó nên mối nối ở nút này phải là một "địa chỉ đặc biệt" chỉ dùng để đánh dấu nút kết thúc danh sách chứ không như các địa chỉ ở các nút khác, ta gọi là "mối nối không" và ký hiệu là null.

Để có thể truy nhập được vào mọi nút trong danh sách, tất nhiên phải truy nhập được vào nút đầu tiên, nghĩa là cần có một con trỏ L trỏ vào nút đầu tiên này.

Nếu dùng mũi tên để chỉ mối nối, ta sẽ có một hình ảnh một danh sách móc nối như sau (mà ta sẽ gọi là "danh sách nối đơn - singly linked list")



Dấu  chỉ mối nối không. Người ta quy ước: Nếu danh sách rỗng thì L = null. Các chữ A, B tượng trưng cho phần biểu diễn thông tin.

Rõ ràng là để tổ chức danh sách móc nối thì những khả năng sau đây cần phải có:

- 1) Tồn tại những phương tiện để chia bộ nhớ ra thành các nút và ở mỗi nút có thể truy nhập được vào từng trường (ta chỉ xét tới trường hợp nút và trường có kích thước ấn định).
- 2) Tồn tại một cơ chế để xác định được một nút đang sử dụng (mà ta gọi là *nút bận*) hoặc không sử dụng (mà ta gọi là *nút trống*).
- 3) Tồn tại một cơ cấu như một "kho chứa chỗ trống" để cung cấp các nút trống khi có yêu cầu sử dụng và thu hồi lại các nút đó khi không cần dùng nữa.

Ta sẽ xét đến các vấn đề này sau. Trước mặt để tiện trình bày ta cứ gọi cơ cấu này là "danh sách chỗ trống" (list of available space).

Việc "danh sách chỗ trống" cấp phát một nút có địa chỉ là  $P$ , cho người sử dụng (còn đối với người sử dụng thì đây là phép "xin một nút trống") được thực hiện bởi một thủ tục New mà lời gọi là một câu lệnh `call new(p)`:

Còn phép thu hồi một nút không dùng nữa hay là phép trả một nút, có địa chỉ  $p$  về danh sách chỗ trống thì được thực hiện bởi `call dispose(p)`;

## 4.4.2 Một số phép toán

Các giải thuật sau đây thể hiện một số phép toán thường được tác động vào danh sách nối đơn.

Với một nút có địa chỉ là  $p$  (được trả bởi  $p$ ) thì `INFO(p)` chỉ trường `INFO` của nút ấy, `LINK(p)` chỉ trường `LINK` của nó.

### 4.4.2.1 Bổ sung nút mới vào danh sách nối đơn

#### Procedure INSERT(L,M,X)

{Cho  $L$  là con trỏ, trỏ tới nút đầu tiên của một danh sách nối đơn,  $M$  là con trỏ trỏ tới một nút đang có trong danh sách.

Giải thuật này thực hiện bổ sung vào sau nút trỏ bởi  $M$  một nút mới mà trường `INFO` của nó sẽ có giá trị lấy từ ô nhớ có địa chỉ  $X$ }

1. {Tạo nút mới}

`call new(p)`;

`INFO(p) := X`;

2. {Thực hiện bổ sung, nếu danh sách rỗng thì bổ sung nút mới vào thành nút đầu tiên, nếu danh sách không rỗng thì nắm lấy  $M$  và bổ sung nút mới vào sau nút đó}

```

if L = null then begin
    L := p
    LINK(p) := null;
end

```

```

else begin
    LINK(p) := LINK(M);
    LINK(M) := p
end;

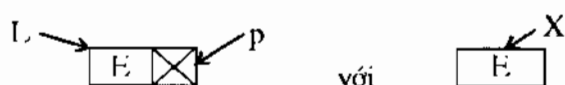
```

3. **return**

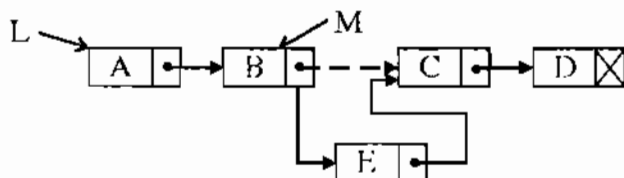
\* Phép xử lý được minh hoạ bởi hình sau:

a) Nếu L = null

thì sau phép bổ sung ta có:



b) Nếu nút L  $\neq$  null



#### 4.4.2.2 Loại bỏ một nút 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}

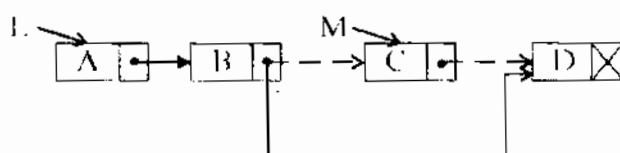
```

if L = null then begin
    write (' Danh sách rỗng');
    return
end;

```



2. { Trường hợp nút trỏ bởi  $M$  là nút đầu tiên của danh sách }  
     **if**  $M = L$  **then begin**  
          $L := \text{LINK}(M)$ ;  
         **call**  $\text{dispose}(M)$   
         **return**  
     **end;**
3. { Tìm đến nút đứng trước nút trỏ bởi  $M$  }  
      $P := L$ ;  
     **while**  $\text{LINK}(p) \neq M$  **do**  $P := \text{LINK}(p)$ ;
4. { Loại bỏ nút trỏ bởi  $M$  }  
      $\text{LINK}(p) := \text{LINK}(M)$ ;
5. { Đưa nút bị loại về danh sách chỗ trống }  
     **call**  $\text{dispose}(M)$
- 6 **return**



#### 4.4.2.3 Ghép hai danh sách nối đơn

##### Procedure COMBINE (P,Q)

{ Cho hai danh sách nối đơn lần lượt trỏ bởi P và Q. Giải thuật này thực hiện ghép hai danh sách đó thành một danh sách mới và cho P trỏ tới nó }

1. { Trường hợp danh sách trỏ bởi Q rỗng }  
     **if**  $Q = \text{null}$  **then return;**
2. { Trường hợp danh sách trỏ bởi P rỗng }  
     **if**  $P = \text{null}$  **then begin**  
          $P := Q$ ;  
         **return**  
     **end;**
3. { Tìm đến nút cuối danh sách }  
      $P1 := P$ ;  
     **while**  $\text{LINK}(P1) \neq \text{null}$  **do**  $P1 := \text{LINK}(P1)$ ;

#### 4. {Ghép}

$LINK(P1) := Q$

#### 5. return

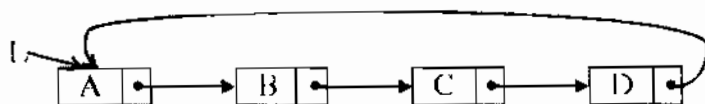
**Chú ý:** Rõ ràng với các danh sách tuyến tính mà kích thước luôn biến động trong quá trình xử lý hay thường xuyên có các phép bổ sung và loại bỏ tác động, thì cách tổ chức móc nối như trên tỏ ra thích hợp. Tuy nhiên cách cài đặt này cũng có những nhược điểm nhất định:

- Chỉ có phần tử đầu tiên trong danh sách được truy nhập trực tiếp còn các phần tử khác chỉ được truy nhập sau khi đã qua các phần tử đứng trước nó.
- Tốn bộ nhớ hơn do ở chỗ phải có thêm trường LINK ở mỗi nút để lưu trữ địa chỉ nút tiếp theo.

### 4.4.3 Các dạng khác của danh sách móc nối

#### 4.4.3.1 Danh sách nối vòng (Circularly linked list)

Một cải tiến của danh sách nối đơn là kiểu danh sách nối vòng. Nó khác với danh sách nối đơn ở chỗ: mỗi nối ở nút cuối cùng không phải là "mối nối không" mà lại là địa chỉ của nút đầu tiên của danh sách. Hình ảnh của nó như sau:



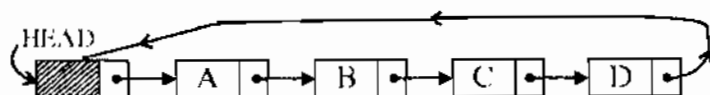
Cải tiến này làm cho việc truy nhập vào các nút trong danh sách được linh hoạt hơn. Ta có thể truy nhập vào mọi nút trong danh sách bắt đầu từ một nút nào cũng được, không nhất thiết phải từ nút đầu tiên. Điều đó cũng có nghĩa là nút nào cũng có thể coi là nút đầu tiên và con trỏ L trỏ tới nút nào cũng được.

Như vậy đối với danh sách nối vòng chỉ cần cho biết con trỏ trỏ tới nút muốn loại bỏ ta vẫn thực hiện được vì vẫn tìm được đến nút đứng trước nó. Với phép ghép, phép tách cũng có những thuận lợi nhất định.

Tuy nhiên, danh sách nối vòng có một nhược điểm rất rõ là trong xử lý, nếu không cẩn thận sẽ dẫn tới một chu trình không kết thúc. Sở dĩ như vậy là vì không biết được chỗ kết thúc của danh sách.

Điều này có thể khắc phục được bằng cách đưa thêm vào một nút đặc

biệt gọi là "nút đầu danh sách" (list head node). Trường INFO của nút này không chứa dữ liệu của phần tử nào và con trỏ HEAD bây giờ trỏ tới nút đầu danh sách này, cho phép ta truy nhập vào danh sách.



Việc dùng thêm nút đầu danh sách đã khiến cho danh sách về mặt hình thức, không bao giờ rỗng. Lúc đó ta có hình ảnh



với qui ước  $\text{LINK}(\text{HEAD}) = \text{HEAD}$

Sau đây là đoạn giải thuật bổ sung một nút vào thành nút đầu tiên của một danh sách nối vòng có "nút đầu danh sách" trỏ bởi HEAD.

**call** new(p);

INFO(p) := X {đưa dữ liệu mới đặt ở ô X vào trường INFO của nút trỏ bởi p}

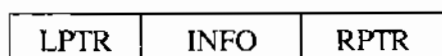
LINK(p) := LINK(HEAD);

LINK(HEAD) := p;

**Chú ý:** Việc dùng "nút đầu danh sách" cũng có phần tiện lợi nhất định vì vậy ngay đối với danh sách nối đơn người ta cũng dùng.

#### 4.4.3.2 Danh sách nối kép (Doubly linked list)

Với các kiểu danh sách như đã nêu ta chỉ có thể duyệt qua danh sách theo một chiều. Trong một số ứng dụng đôi khi vẫn xuất hiện yêu cầu đi ngược lại. Để có được cả hai khả năng, cần phải đặt ở mỗi nút hai con trỏ, một trỏ tới nút đứng trước nó và một trỏ tới nút đứng sau nó. Như vậy nghĩa là qui cách của một nút sẽ như sau:

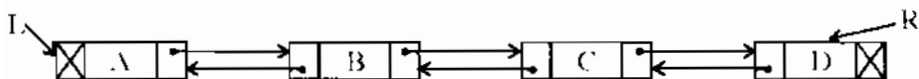


Với LPTR là con trỏ trái, trỏ tới nút đứng trước

còn RPTR là con trỏ phải, trỏ tới nút đứng sau.

Còn INFO vẫn giống như trên.

Như vậy danh sách móc nối sẽ có dạng:



Ta gọi đó là một danh sách nối kép. Ở đây LPTR của nút cực trái và RPTR của nút cực phải, là null. Tất nhiên để có thể truy nhập vào danh sách cả hai chiều thì phải dùng hai con trỏ: con trỏ L trỏ tới nút cực trái và con trỏ R trỏ tới nút cực phải. Khi danh sách rỗng ta quy ước  $L = R = \text{null}$ .

Bây giờ ta xét một vài giải thuật tác động trên danh sách nối kép được tổ chức như đã nêu.

#### **Procedure DOUBIN (L, R, M, X)**

{ Cho con trỏ L và R lần lượt trỏ tới nút cực trái và nút cực phải của một danh sách nối kép, M là con trỏ trỏ tới một nút trong danh sách này. Giải thuật này thực hiện bổ sung một nút mới, mà dữ liệu chứa ở X, vào trước nút trỏ bởi M }

1. { Tạo nút mới }  
**call** new(p);  
 INFO(p) := X;
2. { Trường hợp danh sách rỗng }  
**if** R = null **then begin**  
     LPTR(p) := RPTR(p) := null;  
     L := R := p;  
   **return**  
**end;**
3. { M trỏ tới nút cực trái }  
**if** M = L **then begin**  
     LPTR(p) := null;  
     RPTR(p) := M;  
     LPTR(M) := p;  
     L := p;  
   **return**  
**end;**
4. { Bổ sung vào giữa }  
     LPTR(p) := LPTR(M);  
     RPTR(p) := M;

```

LPTR(M) := p;
RPTR(LPTR(p)) := p;
return

```

### Procedure DOUBDEL(L,R,M)

{ Cho L và R là hai con trỏ trái và phải của danh sách nối kép, M trỏ tới một nút trong danh sách. Giải thuật này thực hiện việc loại 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 và 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;

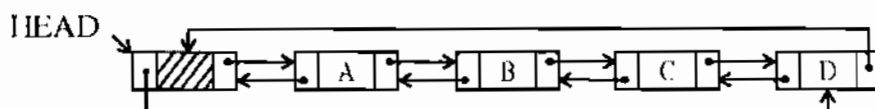
```
3. call dispose(M);
 

```

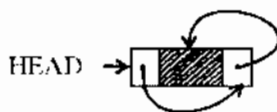
      return

```

**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. Vì thường để cho đối xứng, khi đưa nút đầu danh sách vào người ta tổ chức danh sách nối kép theo kiểu lai nối vòng, với dạng như sau:



Như vậy nghĩa là nút đầu danh sách đóng cả hai vai trò vừa là nút cực trái, vừa là nút cực phải. Trường hợp danh sách rỗng thì chỉ còn nút đầu danh sách.



Lúc đó:  $RPTR(HEAD) = HEAD$

$LPTR(HEAD) = HEAD$

Với danh sách kiểu này giải thuật DOUBIN ở trên chỉ có bước 1 và bước 4. Còn giải thuật DOUBDEL chỉ còn một bước:

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

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

call dispose(M);

#### 4.4.4 Ví dụ áp dụng

Bây giờ ta xét tới bài toán cộng hai đa thức của x được tổ chức dưới dạng danh sách móc nối.

##### 4.4.4.1 Biểu diễn đa thức

Đa thức sẽ được biểu diễn dưới dạng danh sách nối đơn, với mỗi nút có qui cách như sau:

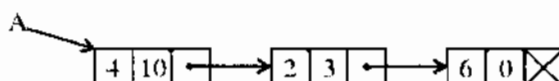
COEF	EXP	LINK
------	-----	------

Trường COEF chứa hệ số khác không của một số hạng trong đa thức.

Trường EXP chứa số mũ tương ứng.

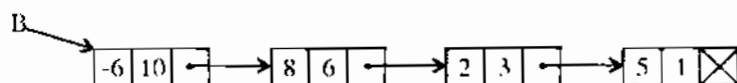
Trường LINK chứa mối nối tới nút tiếp theo. Như vậy đa thức

$A(x) = 4x^{10} - 2x^3 + 6$  sẽ được biểu diễn bởi:



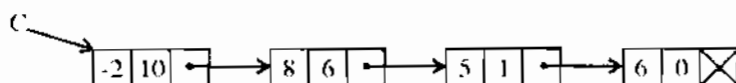
Còn đa thức:

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



Phép cộng hai đa thức này sẽ cho:

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



#### 4.4.4.2 Giải thuật

Trước hết cần phải thấy rằng để thực hiện cộng  $A(x)$  với  $B(x)$  ta phải tìm đến từng số hạng của đa thức đó, nghĩa là phải dùng hai biến trỏ  $p$  và  $q$  để duyệt qua hai danh sách tương ứng với  $A(x)$  và  $B(x)$  trong quá trình tìm này.

Ta sẽ thấy có những tình huống như sau:

- $EXP(p) = EXP(q)$ , ta sẽ phải thực hiện cộng giá trị COEF ở hai nút đó, nếu giá trị tổng khác không thì phải tạo ra nút mới thể hiện số hạng tổng đó và gắn vào danh sách ứng với  $C(x)$ .
- Nếu  $EXP(p) > EXP(q)$  (hoặc ngược lại thì cũng tương tự): phải sao chép nút  $p$  và gắn vào danh sách của  $C(x)$ .
- Nếu một danh sách kết thúc trước: phần còn lại của danh sách kia sẽ được sao chép và gắn dần vào danh sách của  $C(x)$ .

Mỗi lần một nút mới tạo ra đều phải gắn vào "đuôi" của  $C$ . Như vậy phải thường xuyên nắm được nút đuôi này bằng một con trỏ  $d$ .

Công việc này được lặp lại nhiều lần vì vậy cần được thể hiện bằng một chương trình con thủ tục: gọi là  $ATTACH(H, M, d)$ . Nó thực hiện: lấy một nút mới, đưa vào trường COEF của nút này giá trị  $H$ , đưa vào trường EXP giá trị  $M$  và gắn nút mới đó vào sau nút trỏ bởi  $d$ .

#### Procedure ATTACH( $H, M, d$ )

```
call new(p);
COEF(p) := H;
EXP(p) := M;
LINK(d) := p;
```

```

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

```

Sau đây là thủ tục cộng hai đa thức:

**Procedure** PADD(A, B, C)

1.  $p := A; q := B;$
2. **call** new (C);
 

$d := C$  {lấy một nút mới để làm "nút đuôi giả" để ngay từ lúc đầu có thể sử dụng được thủ tục ATTACH, sau này sẽ bỏ đi }
3. **while**  $p \neq \text{null}$  **and**  $q \neq \text{null}$  **do**

**case**

$\text{EXP}(p) = \text{EXP}(q): x := \text{COEF}(p) + \text{COEF}(q);$

**if**  $x \neq 0$  **then**

**call** ATTACH( $x, \text{EXP}(p), d$ );

$p := \text{LINK}(p); q := \text{LINK}(q);$

$\text{EXP}(p) < \text{EXP}(q):$  **call** ATTACH( $\text{COEF}(q), \text{EXP}(q), d$ );

$q := \text{LINK}(q);$

**else :** **call** ATTACH( $\text{COEF}(p), \text{EXP}(p), d$ );

$p := \text{LINK}(p);$

**end case;**
4. {Danh sách ứng với B(x) đã hết}
 

**while**  $p \neq \text{null}$  **do begin**

**call** ATTACH( $\text{COEF}(p), \text{EXP}(p), d$ );

$p := \text{LINK}(p)$

**end;**
5. {Danh sách ứng với A(x) đã hết}
 

**while**  $q \neq \text{null}$  **do begin**

**call** ATTACH( $\text{COEF}(q), \text{EXP}(q), d$ );

$q := \text{LINK}(q)$

**end;**
6. {Kết thúc danh sách C}
 

$\text{LINK}(d) := \text{null};$
7. {Loại bỏ nút đóng vai trò đuôi giả lúc đầu}
 

$t := C; C := \text{LINK}(C);$  **call** dispose( $t$ );
8. **return**



\* Hãy xét thêm trường hợp: nếu ta tổ chức đa thức dưới dạng một danh sách nối vòng có "nút đầu danh sách" lần lượt trở bởi A, B, C thì giải thuật PADD sẽ có gì thay đổi.

Rõ ràng phải sửa lại một số bước:

Bước 1 sẽ là:  $p := \text{LINK}(A); q := \text{LINK}(B);$

Bước 3 sẽ là: **while**  $p \neq A$  **and**  $q \neq B$  **do ...**

Bước 4 sẽ là: **while**  $p \neq A$  **do...**

Bước 5 sẽ là: **while**  $q \neq B$  **do...**

Bước 6 sẽ là:  $\text{LINK}(d) := C$

Bước 7: **ho**

\* Nếu bây giờ ta sử dụng thêm trường EXP của nút đầu danh sách và gán cho  $\text{EXP}(A) = -1$  đối với B và C cũng vậy, thì giải thuật sẽ gọn hơn. Sở dĩ như vậy là vì khi một đa thức đã kết thúc, chẳng hạn đa thức  $A(x)$ , thì  $\text{EXP}(p) = -1$  mà  $-1 < \text{EXP}(q)$  lúc đó.

Vì vậy việc sao chép phần đuôi của  $B(x)$  để gắn vào đuôi của  $C(x)$  sẽ được xử lý ở bước 3. Do đó các bước 4, 5 có thể bỏ được. Giải thuật bây giờ sẽ là:

**Procedure CPADD (A, B, C)**

1.  $p := \text{LINK}(A); q := \text{LINK}(B);$

2. **call**  $\text{new}(C); d := C;$

3. **while true do**

**case**

$\text{EXP}(p) = \text{EXP}(q)$ : **if**  $\text{EXP}(p) = -1$  **then exit**;

$x := \text{COEF}(p) + \text{COEF}(q);$

**if**  $x \neq 0$  **then**

**call**  $\text{ATTACH}(x, \text{EXP}(p), d);$

$p := \text{LINK}(p); q := \text{LINK}(q);$

$\text{EXP}(p) < \text{EXP}(q)$ : **call**  $\text{ATTACH}(\text{COEF}(q), \text{EXP}(q), d);$

$q := \text{LINK}(q);$

**else :** **call**  $\text{ATTACH}(\text{COEF}(p), \text{EXP}(p), d);$

$p := \text{LINK}(p)$

**end case;** { ở đây ta sẽ dùng câu lệnh **exit** để thoát khỏi vòng lặp vô tận khi  $\text{EXP}(p) = -1$  }

4. **return**

## BÀI TẬP CHƯƠNG 4

- 4.1. Dựa trên đặc điểm gì để phân biệt vector và danh sách tuyến tính?
- 4.2. Hãy nêu một số ví dụ về bản ghi, về tệp.
- 4.3. Cho ma trận  $A = (A_{ij})$  với  $1 \leq i \leq 7, 1 \leq j \leq 6$ . Mỗi phần tử của ma trận chiếm 2 từ máy. Hãy viết công thức tính địa chỉ của phần tử  $(A_{ij})$  ứng với cách lưu trữ:
- Theo thứ tự ưu tiên cột.
  - Theo thứ tự ưu tiên hàng.
- 4.4. Giả sử 4 từ máy mới đủ lưu trữ một phần tử của ma trận trong PASCAL. Biết rằng một ma trận A được khai báo kiểu

**array [1..8, 1..3] of integer**

và được lưu trữ trong một miền nhớ kế tiếp bắt đầu từ từ máy có địa chỉ là 2000.

Hãy tính Loc ( $A[4,2]$ )

- 4.5. Cho mảng  $B = B_{ijk}$  với
- $2 \leq i \leq 6$
  - $5 \leq j \leq -1$
  - $3 \leq k \leq 10$

Hãy tính địa chỉ của phần tử  $B[5, -2, 4]$  biết rằng mảng này được lưu trữ theo thứ tự ưu tiên hàng, mỗi phần tử chiếm 3 từ máy và địa chỉ của từ máy đầu tiên là 1500.

- 4.6. Một ma trận vuông A chỉ có các phần tử thuộc đường chéo chính và hai đường chéo sát đường chéo chính là khác không (nghĩa là  $A[i,j] = 0$  nếu  $|i-j| > 1$ ). Nó có dạng:

$$\begin{bmatrix} x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ 0 & x & x & x & 0 \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{bmatrix}$$

Để tiết kiệm người ta chỉ lưu trữ các phần tử khác không này dưới dạng một vector B theo thứ tự ưu tiên hàng. Chẳng hạn:

$$\left\{ \begin{array}{lll} A[1;1] & \text{lưu trữ} & \text{tại } B[1] \\ A[1;2] & \text{lưu trữ} & \text{tại } B[2] \\ A[2;1] & \text{lưu trữ} & \text{tại } B[3] \\ A[2;2] & \text{lưu trữ} & \text{tại } B[4] \\ A[2;3] & \text{lưu trữ} & \text{tại } B[5] \\ \dots & \dots & \dots \end{array} \right.$$

Hãy lập giải thuật cho phép xác định được giá trị của  $A[i;j]$  từ vector  $B$ , khi biết  $i$  và  $j$  ( $1 \leq i; j \leq n$ ). Ví dụ cho  $i = 2, j = 3$  thì giá trị của  $A[2;3]$  được xác định qua giá trị của  $B[5]$ .

**4.7.** Cho hệ phương trình đại số tuyến tính có dạng:

$$a_{11}x_1 = b_1$$

$$a_{21}x_1 + a_{22}x_2 = b_2$$

...

$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n$$

Để tiết kiệm, người ta lưu trữ ma trận hệ số của hệ phương trình dưới dạng một vector. Như vậy chỉ cần  $\frac{n(n+1)}{2}$  từ máy (giả sử một hệ số chứa trong một từ) chứ không cần tới  $n^2$  từ.

Hãy lập giải thuật giải hệ phương trình đó ứng với một cấu trúc lưu trữ như đã định.

**4.8. a)** Nếu dùng một vector lưu trữ có độ dài  $n+2$  để biểu diễn một đa thức:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

theo dạng:

$$(n, a_n, a_{n-1}, \dots, a_1, a_0)$$

với phần tử đầu biểu diễn cấp của  $p(x)$ ,  $n+1$  phần tử sau lần lượt biểu diễn hệ số các số hạng của  $p(x)$ ; thì cách lưu trữ này có ưu, nhược điểm gì?

b) Nếu người ta chỉ lưu trữ mũ và hệ số của các số hạng khác không thôi theo kiểu, chẳng hạn như  $x^{1000} + 1$  thì vector biểu diễn có dạng: (2, 1000, 1, 0, 1)

Hay  $x^7 + 3x^4 - 5x + 3$  thì

$$(4, 7, 1, 4, 3, 1, -5, 0, 3)$$

Cách lưu trữ mới này có ưu, nhược điểm gì?

**4.9.** Hãy lập giải thuật cộng hai đa thức được lưu trữ dưới dạng như đã nêu ở bài 4.8.

**4.10.** Lập các giải thuật thực hiện các phép sau đây đối với danh sách nối đơn mà nút đầu tiên của nó được trỏ bởi  $L$ :

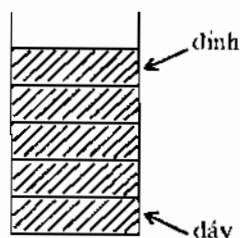
- a) Tính số lượng các nút của danh sách.
  - b) Tìm tới nút thứ  $k$  trong danh sách, nếu có nút thứ  $k$  thì cho ra địa chỉ nút đó, nếu không thì cho ra địa chỉ null.
  - c) Bổ sung một nút vào sau nút thứ  $k$ .
  - d) Loại bỏ nút đứng trước nút thứ  $k$ .
  - e) Cho thêm con trỏ  $M$  trỏ tới một nút có trong danh sách nói trên và một danh sách nối đơn khác có nút đầu tiên trỏ bởi  $P$ . Hãy chèn danh sách  $P$  này vào sau nút trỏ bởi  $M$ .
  - f) Tách thành hai danh sách mà danh sách sau trỏ bởi  $M$  (cho như ở câu e).
  - g) Đảo ngược danh sách đã cho (tạo một danh sách  $L'$  mà các nút móc nối theo thứ tự ngược lại so với  $L$ ).
- 4.11.** Cho một danh sách nối đơn có nút đầu danh sách trỏ bởi  $p$ . Giá trị của trường INFO trong các nút giả sử là các số khác nhau và các nút đã được sắp xếp theo thứ tự tăng dần của giá trị này. Hãy lập giải thuật:
- a) Bổ sung một nút mới mà trường INFO có giá trị là  $X$ , vào danh sách.
  - b) Loại bỏ nút mà trường INFO có giá trị bằng  $K$  cho trước.
- 4.12.** Lập giải thuật thực hiện các phép sau đây đối với danh sách nối vòng:
- a) Ghép hai danh sách nối vòng có nút "đầu danh sách" lần lượt trỏ bởi  $p$  và  $q$ , thành một danh sách mà nút đầu danh sách trỏ bởi  $p$ .
  - b) Lập "bản sao" của một danh sách nối vòng có nút đầu danh sách trỏ bởi  $L$ .
- 4.13.** Cho danh sách nối kép có nút đầu danh sách và cách cấu trúc như trong phần chú ý 4.3. trong bài giảng.  $p$  là con trỏ, trỏ tới nút đầu danh sách đó. Hãy lập giải thuật loại bỏ tất cả các nút mà trường INFO của nó có giá trị bằng  $K$  cho trước.
- 4.14.** Cho một đa thức  $P(x)$  được tổ chức dưới dạng một danh sách móc nối như đã nêu trong bài giảng. Gọi  $p$  là con trỏ trỏ tới danh sách đó. Hãy lập giải thuật tính giá trị của  $P(x)$  ứng với giá trị  $x$  cho biết.

## NGĂN XẾP VÀ HÀNG ĐỢI

### 5.1 Định nghĩa ngăn xếp (stack)

Stack là một kiểu 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).

Có thể hình dung nó như cơ cấu của một hộp chứa đạn súng trường hoặc súng tiểu liên. Lắp đạn vào hay lấy đạn ra cũng chỉ ở một đầu hộp. Viên đạn mới nạp vào sẽ nằm ở đỉnh còn viên nạp vào đầu tiên sẽ nằm ở *đáy* (bottom). Viên nạp vào sau cùng lại chính là viên lên nòng súng trước tiên.

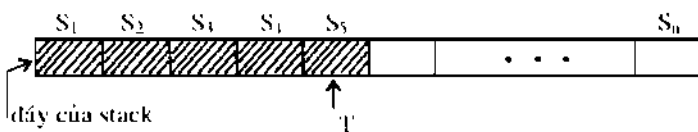


Hình 5.1

Nguyên tắc "vào sau ra trước" như vậy của stack đã đưa tới một tên gọi khác: danh sách kiểu LIFO (last - in - first - out). Stack có thể rỗng hoặc bao gồm một số phần tử.

### 5.2. Lưu trữ stack bằng mảng (lưu trữ kế tiếp)

Có thể lưu trữ stack bởi một vectơ lưu trữ  $S$  gồm  $n$  phần tử nhớ kế tiếp. Nếu  $T$  là địa chỉ của phần tử đỉnh của stack thì  $T$  sẽ có giá trị biến đổi khi stack hoạt động (vì vậy người ta gọi  $T$  là một *biến trỏ* - variable pointer). Nếu ta quy ước dùng địa chỉ tương đối (như chỉ số) thì khi stack rỗng  $T = 0$ . Khi một phần tử mới được bổ sung vào stack,  $T$  sẽ tăng lên 1. Khi một phần tử bị loại ra khỏi stack,  $T$  sẽ giảm đi 1. Có thể thấy cấu trúc dữ liệu của stack như hình sau:



Hình 5.2

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

**Procedure PUST(S, T, X)**

{Giải thuật này thực hiện việc bổ sung phần tử X vào stack lưu trữ bởi vectơ S có n phần tử. Ở đây T là con trỏ, trỏ tới đỉnh stack}

1. {Xét xem stack có **TRÀN** (overflow) không? Hiện tượng **TRÀN** xảy ra khi S không còn chỗ để tiếp tục lưu trữ các phần tử của stack nữa. Lúc đó sẽ in ra thông báo **TRÀN** và kết thúc}

**if**  $T \geq n$  **then begin**

**write** ('STACK **TRÀN**');

**return**

**end;**

2. {Chuyển con trỏ}

$T := T + 1;$

3. {Bổ sung phần tử mới X}

$S[T] := X;$

4. **return**

**Procedure POP(S, T, Y)**

{Giải thuật này thực hiện việc loại bỏ phần tử ở đỉnh stack S đang trỏ bởi T. Phần tử bị loại sẽ được thu nhận và đưa ra bởi Y}

1. {Xét xem stack có **CẠN** (underflow) không? Hiện tượng **CẠN** xảy ra khi stack đã rỗng, không còn phần tử nào để loại nữa, lúc đó sẽ 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}

$Y := S[T + 1];$

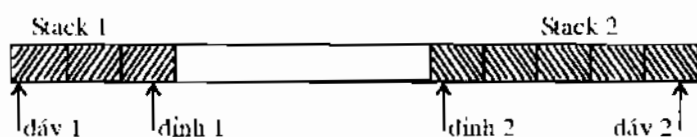
4. **return**

**Chú ý:** Xử lý với nhiều stack:

Có những trường hợp cùng một lúc ta phải xử lý nhiều stack. Như vậy có thể xảy ra tình trạng một stack này đã bị tràn trong khi không gian dự trữ cho stack khác vẫn còn chỗ trống (Tràn cục bộ).

Làm thế nào để khắc phục được?

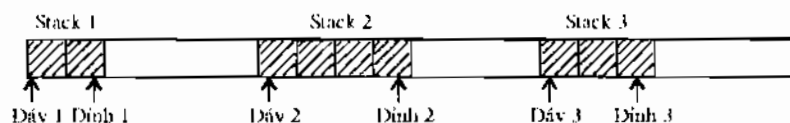
Nếu là hai stack thì có thể giải quyết dễ dàng. Ta không quy định kích thước tối đa cho từng stack nữa mà không gian nhớ dành ra sẽ được dùng chung. Ta sẽ đặt hai stack ở hai đầu sao cho hướng phát triển của chúng ngược nhau, như ở hình 5.3



Hình 5.3

Như vậy có thể một stack này dùng gần hết không gian dự trữ nếu như stack kia chưa dùng đến. Do đó hiện tượng Tràn chỉ xảy ra khi toàn bộ không gian nhớ dành cho chúng đã được dùng hết.

Nhưng nếu số lượng stack từ 3 trở lên thì không thể làm theo kiểu như vậy được, mà phải có một giải pháp linh hoạt hơn. Chẳng hạn có 3 stack, lúc đầu không gian nhớ có thể chia đều cho cả 3 như ở hình 5.4.



Hình 5.4

Nhưng nếu có một stack nào đó phát triển nhanh bị Tràn trước mà stack khác vẫn còn chỗ thì phải dọn chỗ cho nó bằng cách hoặc đẩy stack đứng sau sang phải hoặc lùi chính stack đó sang trái trong trường hợp có thể. Như vậy thì đáy của các stack phải được phép di động và dĩ nhiên các giải thuật bổ sung hoặc loại bỏ phần tử đối với các stack hoạt động theo kiểu này cũng phải thay đổi theo.

## 5.3 Ví dụ về ứng dụng của stack

### 5.3.1 Đổi cơ số

Ta biết rằng dữ liệu lưu trữ trong bộ nhớ của MTĐT đều được biểu diễn nhị phân. Như vậy nghĩa là các số xuất hiện trong chương trình đều phải chuyển đổi từ thập phân sang nhị phân trước khi thực hiện các phép xử lý.

Đối với hệ thập phân: khi ta viết 437 thì nó biểu diễn con số mà giá trị là  $4 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0$

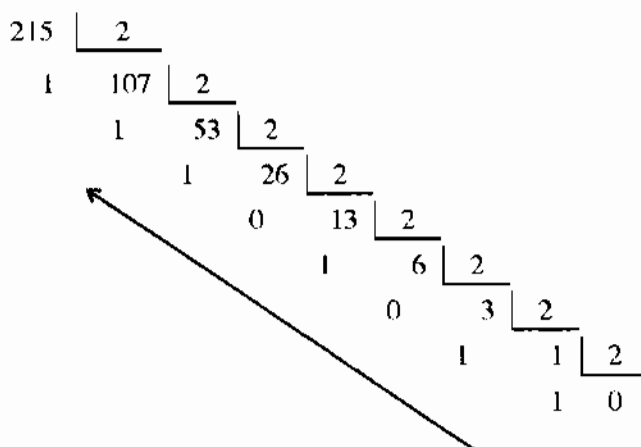
Với số ở hệ nhị phân cũng tương tự.

Chẳng hạn:

11010111 thì biểu diễn số

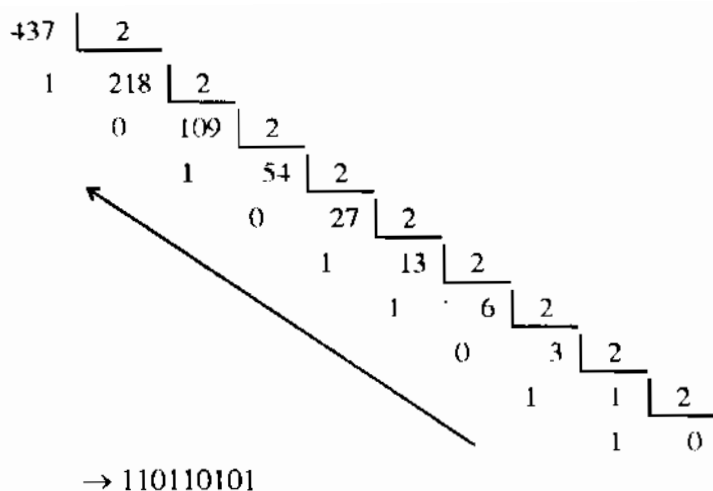
$$1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = (215)_{10}$$

Khi đổi một số nguyên 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



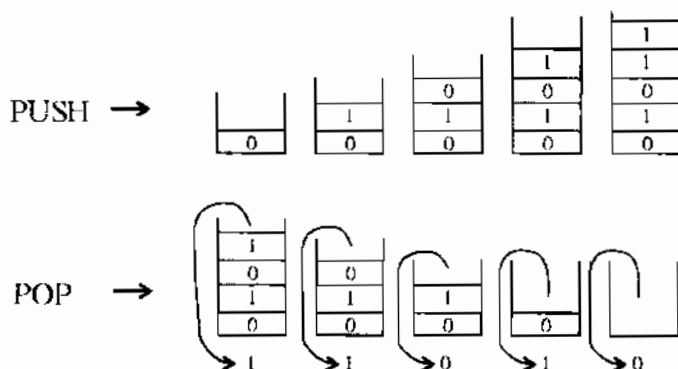
→ 11010111





Như vậy rõ ràng trong các biến đổi này các số dư được tạo ra sau lại được hiển thị trước. Cơ chế sắp xếp này chính là cơ chế 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 để hiển thị.

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



Hình 5.5

Có thể viết giải thuật chuyển đổi như sau:

#### Program CHUYEN\_DOI

{Giải thuật này thực hiện chuyển đổi biểu diễn cơ số mười của một số nguyên dương N sang cơ số 2 và hiển thị biểu diễn cơ số 2 này}

1. **Read(N);**

2. **While**  $N \neq 0$  **do begin**

$R := N \bmod 2$ ; {Tính số dư R trong phép chia N cho 2}

**call** PUSH(S, T, R); { nạp R vào đỉnh stack S}

$N := N \div 2$ ; {Thay N bằng thương của phép chia N cho 2}

**end**

3. **While** S chưa rỗng **do begin**

**call** POP(S, T, R);

**write** (R)

**end**

**end**

### 5.3.2 Biểu thức số học và ký pháp Ba Lan

#### 5.3.2.1 Đặt vấn đề

Thông thường, trong các biểu thức số học, với phép toán hai ngôi như phép cộng, phép trừ, phép nhân, phép chia, phép lũy thừa (dấu phép toán được ký hiệu bởi  $\uparrow$ ), v.v... dấu phép toán (toán tử) bao giờ cũng được đặt ở giữa hai toán hạng (ký pháp *trung tố* (infix notation)).

Với cách biểu diễn này, việc sử dụng các cặp dấu ngoặc để phân biệt các toán hạng (cũng là để phân biệt thứ tự thực hiện các phép toán) là cần thiết.

**Ví dụ:** Hai biểu thức sau:

a)  $(A + B) * C$

b)  $A + (B * C)$

là hoàn toàn khác nhau. Với a) thì phép cộng được thực hiện trước rồi mới tới phép nhân, còn với b) thì ngược lại.

Trong trường hợp nếu không muốn dùng dấu ngoặc thì lại phải theo qui ước về thứ tự ưu tiên thực hiện các phép toán, như đã được quy định trong các ngôn ngữ lập trình, nghĩa là theo thứ tự:

1) Phép lũy thừa

2) Phép nhân, phép chia

3) Phép cộng, phép trừ.

Đối với các phép toán cùng một thứ tự thì sẽ được thực hiện theo trình tự: trái trước, phải sau trong biểu thức.

**Ví dụ:** Với biểu thức

$$A + B * C - D/E \uparrow F$$

thì thứ tự thực hiện sẽ như sau:

$$E \uparrow F$$

$$B * C$$

$$D / (E \uparrow F)$$

$$A + (B * C)$$

$$(A + B * C) - (D / E \uparrow F)$$

Rõ ràng cách viết biểu thức theo ký pháp trung tố với việc sử dụng dấu ngoặc hoặc thứ tự ưu tiên giữa các phép toán đã khiến cho việc tính toán giá trị biểu thức trở nên "cồng kềnh".

Nhà toán học Ba Lan J. Lukasiewicz là người đầu tiên phát hiện rằng: dấu ngoặc là không cần thiết, thông qua cách biểu diễn biểu thức số học theo ký pháp *hậu tố* (poitfix notation) hoặc *tiền tố* (postfix notation) mà gọi chung là *ký pháp Ba Lan* (Polish notation)

### 5.3.2.2 Biểu thức dạng hậu tố và tiền tố

Với ký pháp hậu tố thì toán tử được đặt sau toán hạng một và toán hạng hai.

**Ví dụ:**

$A + B$	thì dưới dạng hậu tố sẽ được viết là	$A B +$
$E / F$	-	$E F /$
$(A + B) * C$	-	$A B + C *$
$A + B * C$	-	$A B C * +$

Với ký pháp tiền tố thì việc đặt toán tử sẽ ngược lại, chẳng hạn

$E / F$	sẽ được viết là	$/ E F$
$A + B * C$	-	$+ A * B C$
$(A + B) / (C - D) + E$	-	$+ / + A B - C D E$

Dựa theo quy tắc của các ký pháp nêu trên, ta có thể viết được biểu thức dưới dạng hậu tố, hoặc tiền tố nếu như từ dạng trung tố, với mỗi toán tử ta xác định được toán hạng 1 và toán hạng 2 ứng với nó.

### 5.3.2.3 Tính giá trị của biểu thức dạng hậu tố

Xét một biểu thức dạng hậu tố, và để cho đơn giản, ta giả sử rằng: mỗi ký tự trong biểu thức (xâu ký tự) biểu diễn cho một biến, hằng hoặc toán tử.

Để ý ta sẽ thấy: khi duyệt biểu thức từ trái sang phải, hễ gặp một toán tử thì hai toán hạng đứng sát ngay trước nó sẽ được tác động bởi toán tử này để tạo thành một toán hạng mới cho toán tử gặp tiếp theo.

Do đó việc tính giá trị của một biểu thức hậu tố, tương ứng với giá trị đã biết của các biến, có thể thực hiện theo cách như sau:

Đọc biểu thức hậu tố từ trái qua phải:

- Nếu ký tự được đọc  $X$  là toán hạng (biến hoặc hằng) thì bảo lưu giá trị của nó.

- Nếu ký tự được đọc  $X$  là toán tử thì 2 giá trị vừa được bảo lưu sẽ lần lượt được lấy ra và tác động toán tử  $X$  vào giữa giá trị lấy ra sau với giá trị lấy ra trước và bảo lưu kết quả lại.

Quá trình trên được tiếp tục cho tới khi kết thúc biểu thức. Giá trị cuối cùng được bảo lưu, chính là giá trị của biểu thức.

Rõ ràng rằng trước khi đọc tới một toán tử thì giá trị của toán hạng phải được bảo lưu để chờ thực hiện phép tính.

Hai toán hạng được đọc sau thì lại được kết hợp với toán tử đọc trước đó cũng có nghĩa là: hai giá trị được bảo lưu sau lại được lấy ra trước để tính toán. Vì vậy người ta phải dùng tới stack để bảo lưu các giá trị toán hạng trong quá trình tính toán.

Giải thuật sau đây sẽ phản ánh cách tính giá trị của một biểu thức hậu tố, ứng với giá trị của các biến trong biểu thức đó.

**Procedure EVAL (P, VAL):**

{ Thủ tục này thực hiện tính giá trị của biểu thức hậu tố  $P$ , tương ứng với giá trị của các biến; kết quả sẽ được gán cho  $VAL$ .

Ở đây có sử dụng một stack  $S$  với  $T$  trỏ tới đỉnh; thoát đầu  $T = 0$  (stack rỗng) }.

1. Ghi thêm dấu ")" vào cuối biểu thức  $P$  để làm dấu kết thúc;

2. **repeat**

Đọc ký tự  $X$  trong  $P$ , khi duyệt từ trái sang phải;

3. **if**  $X$  là một toán hạng **then**

**call** PUSH ( $S, T, X$ )

4. **else begin**

**call** POP ( $S, T, Y$ );

**call** POP ( $S, T, Z$ );

$W := Z (X) Y$ ;

{ ( $X$ ) chỉ toán tử  $X$  }

**call** PUSH ( $S, T, W$ )

**end;**

**until** gặp dấu kết thúc ")";

5. **call** POP (S, T, VAL);

6. **return**

Với biểu thức

$$A B + C D A + - *$$

mà dạng trung tố của nó là

$$(A + B) * (C - (D + A))$$

Nếu  $A = 1$ ;  $B = 5$ ;  $C = 8$ ;  $D = 4$

thì hình ảnh của stack S, khi tính giá trị của biểu thức theo giải thuật EVAL, sẽ như sau:

Ký tự được đọc	A	B	+	C	D	A	+	-	*
Tình trạng của S			1+5				4+1	8-5	6*3
						1			
					4	4	5		
		5		8	8	8	8	3	
	1	1	6	6	6	6	6	6	
									Kết thúc với VAL = 18

Hình 5.6

Tới đây ta cũng thấy rằng: việc tính giá trị của biểu thức ở dạng hậu tố rõ ràng là "đơn giản" hơn, "máy móc" hơn.

Vì vậy trong đa số các ngôn ngữ lập trình, các biểu thức số học vẫn được viết theo ký pháp trung tố. Nhưng chương trình dịch sẽ tự động đổi sang dạng hậu tố (hoặc tiền tố) và sau đó việc tính giá trị biểu thức sẽ được thực hiện trong máy theo dạng hậu tố (hoặc tiền tố) này.

Tất nhiên việc chuyển đổi sẽ được thực hiện bởi một giải thuật, ta sẽ xét sau đây.

#### 5.3.2.4. Chuyển đổi biểu thức từ dạng trung tố sang hậu tố

Ta thấy: trong biểu thức dạng trung tố nếu xuất hiện dấu ngoặc mở "(" thì phải có dấu ngoặc đóng ")" tương ứng với nó, với các cặp dấu ngoặc lồng nhau thì dấu ngoặc mở gặp trước lại tương ứng với dấu ngoặc đóng gặp sau. Ngoài ra khi gặp toán tử thì phải chờ xác định được toán hạng 2 rồi mới được đặt toán tử này vào sau toán hạng 2. Vì vậy trong giải thuật chuyển đổi người ta phải sử dụng một stack để bảo lưu dấu ngoặc mở (để chờ dấu ngoặc đóng tương ứng) và toán tử (để chờ toán hạng 2 tương ứng).

Sau đây là giải thuật:

**Procedure** POLISH (Q, P);

{Q là biểu thức viết dưới dạng trung tố, giải thuật này biến đổi Q sang dạng hậu tố P, thoát đầu P rỗng}

1. Thêm dấu ")" vào cuối Q; {để làm dấu kết thúc}

**call** PUSH (S,T, "("); {Dấu ngoặc mở này tương ứng với dấu ngoặc đóng mới thêm vào cuối Q}

2. **Repeat**

Đọc ký tự X trong Q khi duyệt từ trái qua phải;

3. **Case**

X là toán hạng: Bổ sung thêm X vào P;

X là dấu ngoặc mở: **call** PUSH (S,T, "(");

X là toán tử (X): **While** thứ tự ưu tiên của S[T]  $\geq$  thứ tự ưu tiên của (X) **do begin**

**call** POP (S, T, W);

Bổ sung thêm W vào P

**end;**

**call** PUSH (S, T, '(X)');

X là dấu ngoặc đóng: **repeat**

**call** POP(S,T, W);

Bổ sung thêm W vào P;

**until** gặp dấu '('

loại dấu '(' ra khỏi stack S

**end case**

**until** stack rỗng;

4. **return**

**Ví dụ:** Với biểu thức

$$Q := A + (B * C - (D/E \uparrow F) * G) * H$$

nếu áp dụng giải thuật POLISH thì tình trạng của stack S và dạng hiển thị của P sau mỗi lần đọc ký tự sẽ được minh hoạ qua bảng sau:

STT	Ký tự đọc	Tình trạng của S	Dạng hiển thị của P
		(	Dấu ")" được ghi vào cuối Q, dấu "(" được nạp vào S, P thbat đầu rỗng
1	A	(	A
2	+	( +	A
3	(	( + (	A
4	B	( + (	A B
5	*	( + ( *	A B
6	C	( + ( *	A B C
7	-	( + (-	A B C *
8	(	( + (- (	A B C *
9	D	( + (- (	A B C * D
10	/	( + (- (/	A B C * D
11	E	( + (- (/	A B C * D E
12	↑	( + (- (/ ↑	A B C * D E
13	F	( + (- (/ ↑	A B C * D E F
14	)	( + (-	A B C * D E F ↑ /
15	*	( + (- *	A B C * D E F ↑ /
16	G	( + (- *	A B C * D E F ↑ / G
17	)	( +	A B C * D E F ↑ / G * -
18	*	( + *	A B C * D E F ↑ / G * -
19	H	( + *	A B C * D E F ↑ / G * - H
20	)		A B C * D E F ↑ / G * - H * +

### Chú ý:

- Ở lần đọc thứ 7, ký tự được đọc X là toán tử -, khi đó ở đỉnh stack S đang có toán tử \* mà thứ tự ưu tiên lớn hơn - nên \* được lấy ra khỏi stack để bổ sung vào P trước khi - được nạp vào stack.
- Ở lần đọc thứ 14, ký tự được đọc X là dấu ngoặc đóng (nó ứng với dấu ngoặc mở đọc ở lần thứ 8) nên các toán tử đang nằm ở đỉnh stack S sẽ được lấy ra để bổ sung vào P cho tới khi gặp dấu ngoặc mở và sau đó dấu ngoặc mở này bị loại ra khỏi stack.
- Cuối cùng khi stack S rỗng thì ta sẽ có P dưới dạng:

$$P: A B C * D E F \uparrow / G * - H * +$$

## 5.4 Stack và việc cài đặt thủ tục đệ qui

Khi một thủ tục đệ qui được gọi tới từ chương trình chính, ta nói: Thủ tục được thực hiện ở mức 1 (hay độ sâu 1 của tính đệ qui). Nhưng khi thực hiện ở mức 1 lại gặp lời gọi tới chính nó, nghĩa là phải đi sâu vào mức 2 và cứ như thế cho tới một mức  $k$  nào đấy. Rõ ràng mức  $k$  phải được hoàn thành xong thì mức  $(k-1)$  mới được thực hiện. Lúc đó ta nói: Việc thực hiện được quay về mức  $(k-1)$ .

Khi từ một mức  $i$ , đi sâu vào mức  $(i+1)$  thì có thể có một số tham số, biến cục bộ hay địa chỉ (gọi là địa chỉ quay lui) ứng với mức  $i$  cần phải được bảo lưu để khi quay về tiếp tục sử dụng.

Còn khi quay từ mức  $i$  về mức  $(i-1)$  các tham số, biến cục bộ và địa chỉ ứng với mức  $(i-1)$  lại phải được khôi phục để sử dụng.

Như vậy, trong quá trình thực hiện, 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 tới việc sử dụng stack trong cài đặt thủ tục đệ qui. Mỗi khi có lời gọi tới chính nó thì stack sẽ được nạp để bảo lưu các giá trị cần thiết. Còn mỗi khi thoát ra khỏi một mức thì phần tử ở đỉnh stack sẽ được "móc" 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 được phần tính kết thúc (final computation) và chuyển sang bước 3.

Nếu không thì thực hiện phần tính từng phần (partial computation) và chuyển sang bước 1 (khởi tạo một lời gọi đệ qui).

### 3- Kết thúc

Khôi phục lại tham số, biến cục bộ và địa chỉ quay lui và chuyển tới địa chỉ quay lui này.

Sau đây là chương trình thể hiện cách cài đặt thủ tục đệ qui, có thể dùng stack cho bài toán tính  $n!$  và "tháp Hà Nội".

### Program FACTORIAL

{Cho số nguyên  $n$ , giải thuật này thực hiện tính  $n!$  Ở đây sử dụng một stack  $A$  mà đỉnh được trỏ bởi  $T$ . Mỗi phần tử của  $A$  là một bản ghi gồm có hai trường:

Trường  $N$  ghi giá trị động của  $n$  ở mức hiện hành.



Trường RETADD ghi địa chỉ quay lui.

Lúc đầu stack A rỗng:  $T = 0$ .

Một bản ghi TEMREC được dùng làm bản ghi trung chuyển, nó cũng có 2 trường:

**PARA** ứng với N

**ADDRESS** ứng với RETADD

Ở đây đặt giả thiết: Lúc đầu TEMREC đã chứa các giá trị cần thiết, nghĩa là PARA chứa giá trị n đã cho, ADDRESS chứa địa chỉ ứng với lời gọi trong chương trình chính mà ta gọi là ĐCC (viết tắt của địa chỉ chính).

Các giải thuật PUSH và POP nêu ở 4.4.2. sẽ được sử dụng ở đây. Trong giải thuật này ta viết  $N(T)$  thì điều đó có nghĩa là giá trị ở trường N của phần tử đang trở bởi T (phần tử ở đỉnh stack A) }

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

**call** 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;**

**go to** bước 1

3. { Tính  $N!$  }

**FACTORIAL** :=  $N(T) * \text{FACTORIAL}$ ;

4. { Khôi phục giá trị trước của N và địa chỉ quay lui }

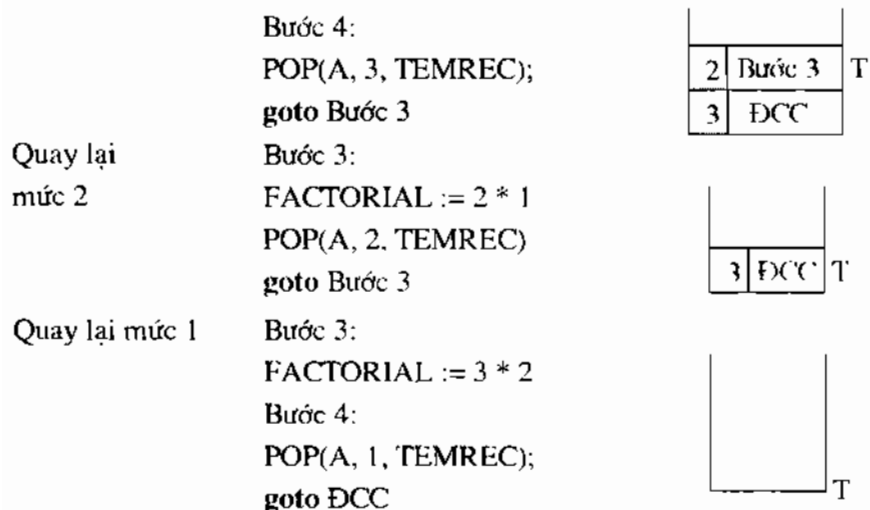
**call** POP(A,T,TEMREC);

**go to** ADDRESS

5. **end**

\* Sau đây là hình ảnh minh hoạ tình trạng của stack A, trong quá trình thực hiện giải thuật (mà ta gọi là vết (trace) của việc thực hiện giải thuật), ứng với  $n = 3$ .

Số mức	Các bước thực hiện	Nội dung của A															
Vào mức 1 (lời gọi chính)	Bước 1 PUSH (A, 0, (3,DCC)) Bước 2: N ≠ 0 PARA := 2; ADDRESS:= Bước 3	<table border="1"> <tr><td></td><td></td><td></td></tr> <tr><td>3</td><td>DCC</td><td>T</td></tr> </table>				3	DCC	T									
3	DCC	T															
Vào mức 2 (gọi đệ qui lần 1)	Bước 1: PUSH (A, 1, (2, bước 3)) Bước 2: N ≠ 0; PARA := 1; ADDRESS:= Bước 3	<table border="1"> <tr><td></td><td></td><td></td></tr> <tr><td>2</td><td>Bước 3</td><td>T</td></tr> <tr><td>3</td><td>DCC</td><td></td></tr> </table>				2	Bước 3	T	3	DCC							
2	Bước 3	T															
3	DCC																
Vào mức 3 (gọi đệ qui lần 2)	Bước 1: PUSH (A, 2, (1, bước 3)) Bước 2: N ≠ 0; PARA := 0; ADDRESS:= Bước 3	<table border="1"> <tr><td></td><td></td><td></td></tr> <tr><td>1</td><td>Bước 3</td><td>T</td></tr> <tr><td>2</td><td>Bước 3</td><td></td></tr> <tr><td>3</td><td>DCC</td><td></td></tr> </table>				1	Bước 3	T	2	Bước 3		3	DCC				
1	Bước 3	T															
2	Bước 3																
3	DCC																
Vào mức 4 (gọi đệ qui lần 3)	Bước 1: PUSH (A, 3, (0, bước 3)) Bước 2: N = 0; FACTORIAL := 1  Bước 4: POP(A, 4, TEMREC); goto Bước 3	<table border="1"> <tr><td></td><td></td><td></td></tr> <tr><td>0</td><td>Bước 3</td><td>T</td></tr> <tr><td>1</td><td>Bước 3</td><td></td></tr> <tr><td>2</td><td>Bước 3</td><td></td></tr> <tr><td>3</td><td>DCC</td><td></td></tr> </table>				0	Bước 3	T	1	Bước 3		2	Bước 3		3	DCC	
0	Bước 3	T															
1	Bước 3																
2	Bước 3																
3	DCC																
Quay lại mức 3	Bước 3: FACTORIAL := 1 * 1	<table border="1"> <tr><td></td><td></td><td></td></tr> <tr><td>1</td><td>Bước 3</td><td>T</td></tr> <tr><td>2</td><td>Bước 3</td><td></td></tr> <tr><td>3</td><td>DCC</td><td></td></tr> </table>				1	Bước 3	T	2	Bước 3		3	DCC				
1	Bước 3	T															
2	Bước 3																
3	DCC																



### Program HANOI\_TOWER

{Gọi N là số lượng đĩa, SN chỉ cọc xuất phát, IN chỉ cọc trung chuyển, DN chỉ cọc đích.

Giải thuật này thực hiện việc chuyển chồng N đĩa từ cọc SN sang cọc DN. Ở đây sử dụng một stack ST, mỗi phần tử của nó là một bản ghi gồm có 5 trường tương ứng với N, SN, IN, DN và RETADD để chứa các giá trị của N, SN, IN, DN và địa chỉ quay lui. Một bản ghi TEMREC cũng có các trường tương ứng với các trường nêu trên, lần lượt gọi là NVAL, SNVAL, DNVAL và ADDRESS. TEMREC được dùng làm bản ghi trung chuyển. Thoạt đầu nó ghi nhận các giá trị ban đầu của N, SN, IN, DN và RETADD.

Stack ST có đỉnh được trỏ bởi T, lúc đầu stack rỗng thì T = 0}

1. {Bảo lưu tham số và địa chỉ quay lui}

**call** PUSH (ST, T, TEMREC);

2. {Kiểm tra giá trị dừng của N, nếu chưa đạt thì chuyển N - 1 đĩa từ cọc xuất phát sang cọc trung chuyển}

**if** N(T) = 0 **then go to** RETADD(T)

**else begin**

NVAL := N(T) - 1;

SNVAL := SN(T);

INVAL := DN(T);

DNVAL := IN(T);

ADDRESS := Bước 3;

**go to** Bước 1;

3. { Chuyển đĩa thứ N từ cọc xuất phát sang cọc đích và (N-1) đĩa từ cọc trung chuyển sang cọc đích }

call POP(ST, T, TEMREC);

write('Đĩa', N, 'từ cọc', SN, 'đến cọc', DN)

NVAL := N(T) - 1;

SNVAL := IN(T);

INVAL := SN(T);

DNVAL := DN(T);

ADDRESS := Bước 4;

go to Bước 1;

4. { Quay lại mức trước }

Call POP(ST, T, TEMREC);

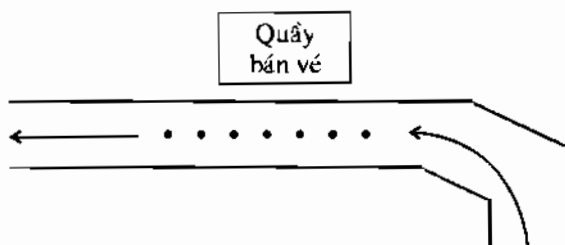
go to RETADD (T)

5. end

## 5.5 Định nghĩa hàng đợi (queue)

Khác với stack, queue là kiểu danh sách tuyến tính mà phép bổ sung được thực hiện ở một đầu, gọi là *lối sau* (rear) và phép loại bỏ thực hiện ở một đầu khác, gọi là *lối trước* (front).

Như vậy cơ cấu của queue giống như một hàng đợi (chẳng hạn để mua vé xe lửa) vào ở một đầu, ra ở đầu khác, nghĩa là vào trước thì ra trước. Vì vậy queue còn được gọi là danh sách kiểu FIFO (first - in - first - out).

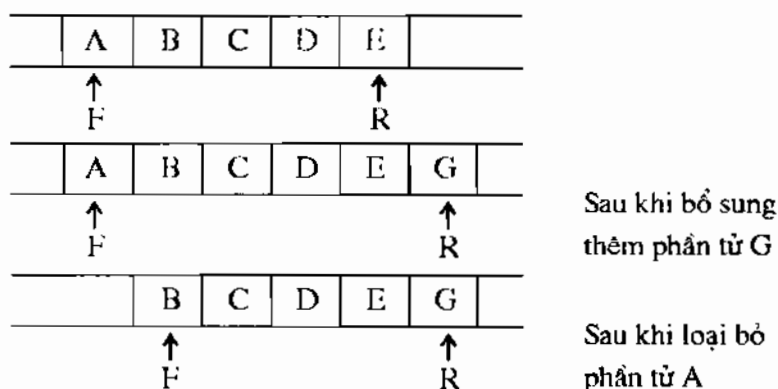


Hình 5.7

## 5.6 Lưu trữ queue bằng mảng (Lưu trữ kế tiếp)

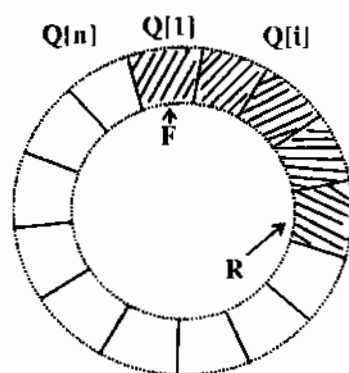
Có thể dùng một vectơ lưu trữ Q có n phần tử làm cấu trúc lưu trữ của queue. Để có thể truy cập vào queue ta phải dùng hai biến trỏ: R trỏ tới lối

sau và F trở về vị trí trước. Nếu ta qui ước dùng địa chỉ tương đối thì khi queue rỗng  $R = F = 0$ , khi bổ sung một phần tử vào queue, R sẽ tăng lên 1, còn khi loại bỏ phần tử ra khỏi queue F sẽ tăng lên 1.



Hình 5.8

Tuy nhiên với cách tổ chức này có thể xuất hiện tình huống là các phần tử của queue sẽ di chuyển khắp không gian nhớ khi thực hiện bổ sung và loại bỏ, chẳng hạn cứ liên tiếp thực hiện một phép bổ sung rồi lại một phép loại bỏ đối với queue có dạng như ở hình 5.8. Do đó người ta phải khắc phục bằng cách coi không gian nhớ dành cho queue như được tổ chức theo kiểu vòng tròn, nghĩa là với vector lưu trữ Q thì Q[1] được coi như đứng sau Q[n] như ở hình 5.9.



Hình 5.9

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

**Procedure CQINSERT(Q, F, R, X);**

{Ổ đầy queue được lưu trữ bởi vectơ Q có n phần tử, có cấu trúc kiểu vòng tròn. F và R là 2 con trỏ trỏ tới lối trước và lối sau của queue. Giải thuật này thực hiện bổ sung một phần tử mới X vào queue }

1. { Trường hợp queue đã đầy }
  - if**  $F = 1$  **and**  $R = n$  **or**  $F = R + 1$  **then begin**
  - write** ('TRÀN');
  - return**
  - end;**
2. { Chính lý con trỏ }
  - if**  $F = 0$  **then**  $F := R + 1$
  - else**
  - if**  $R = n$  **then**  $R := 1$
  - else**  $R := R + 1$
3. { Bổ sung X }
  - $Q[R] := X;$
4. **return**

**Procedure CQDELETE(Q, F, R, Y)**

{Giải thuật này thực hiện việc loại phần tử đang ở lối trước ra khỏi queue, thông tin ứng với phần tử bị loại sẽ được bảo lưu ở Y }

1. { Trường hợp queue rỗng }
  - if**  $F = 0$  **then begin**
  - write** ('CAN');
  - return**
  - end;**
2. { Bảo lưu thông tin của phần tử bị loại }
  - $Y := Q[F];$
3. { Chính lý con trỏ }
  - if**  $F = R$  **then**  $F := R := 0$
  - else**
  - if**  $F = n$  **then**  $F := 1$
  - else**
  - $F := F + 1;$
4. **return**

**Chú ý:** Queue thường dùng để thực hiện các *tuyến chờ* (waiting line) trong các xử lý động, đặc biệt trong các *hệ mô phỏng* (Simulation).

Chẳng hạn: hệ xử lý việc đặt chỗ trước của khách, cho một chuyến bay, là hệ mô phỏng việc xếp hàng mua vé. Thông tin về khách hàng sẽ được lưu trữ trong queue để xử lý: người đăng ký trước sẽ được giải quyết trước (nếu như số chỗ bị hạn chế). Trong các phần sau ta sẽ có dịp làm quen với ứng dụng cụ thể của queue.

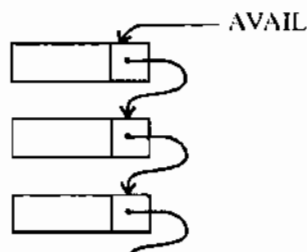
## 5.7 Stack và queue móc nối

Như ta đã biết, đối với stack việc truy nhập đều thực hiện ở một đầu (đỉnh). Vì vậy việc cài đặt stack bằng cách dùng danh sách móc nối là khá tự nhiên. Chẳng hạn với danh sách nối đơn trở bởi L thì có thể coi L như con trỏ trở tới đỉnh stack. Bổ sung một nút vào stack chính là bổ sung một nút vào thành nút đầu tiên của danh sách; loại bỏ một nút ra khỏi stack chính là loại bỏ nút đầu tiên của danh sách đang trở bởi L. Trong thủ tục bổ sung với stack móc nối ta không phải kiểm tra hiện tượng **TRẦN** như đối với stack kế tiếp, vì stack móc nối không hề bị giới hạn về kích thước, nó chỉ phụ thuộc vào giới hạn của bộ nhớ toàn phần (chỉ khi danh sách chỗ trống đã cạn hết).

Đối với queue thì loại bỏ ở một đầu, bổ sung lại ở đầu khác. Nếu coi danh sách nối đơn như một queue và coi L trở tới lối trước thì việc loại bỏ một nút sẽ không khác gì với stack, nhưng bổ sung một nút thì phải thực hiện vào "đuôi" nghĩa là phải lần tìm đến nút cuối cùng. Nếu tổ chức queue móc nối có dạng như danh sách nối kép có hai con trỏ L và R như đã nói ở trên, thì sẽ không phải tìm "đuôi" nữa. Lúc đó coi lối trước được trở bởi L, còn lối sau được trở bởi R (hoặc ngược lại cũng được).

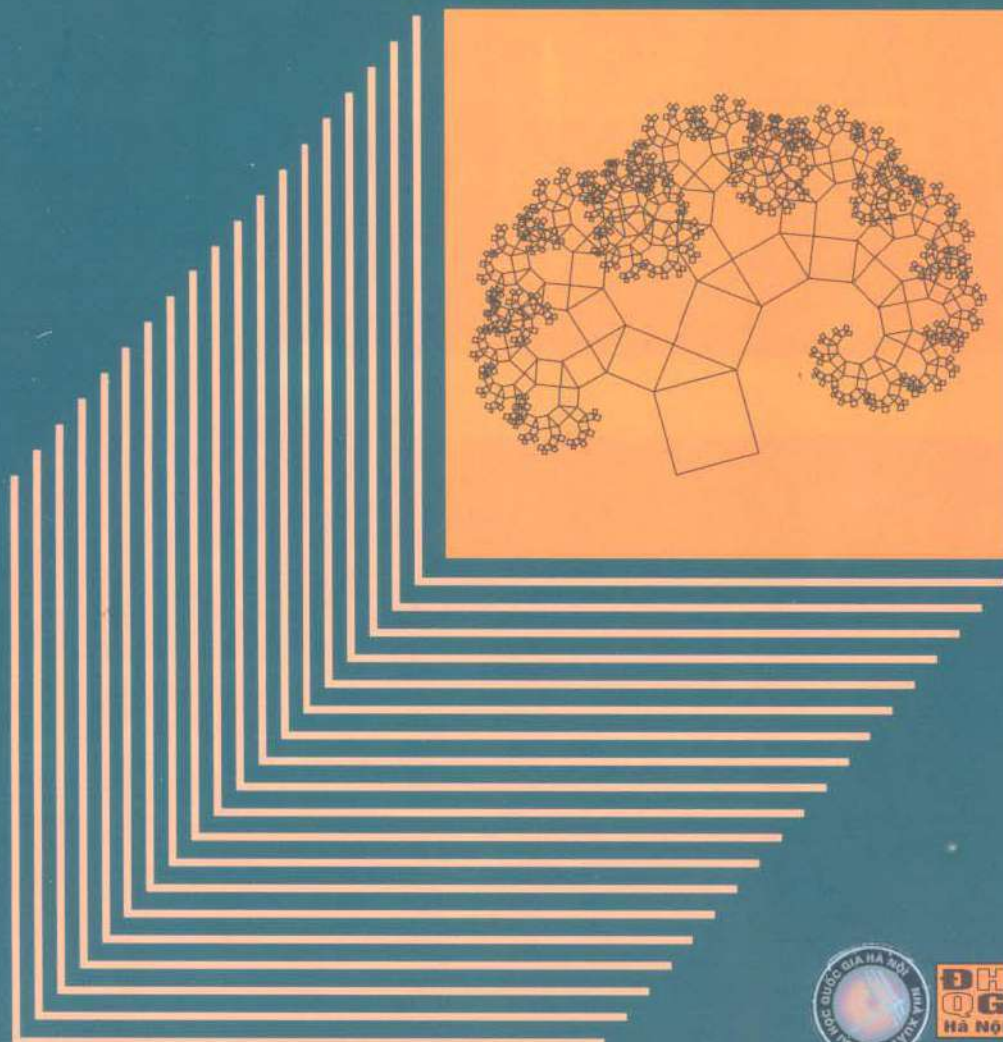
### Chú thích:

Trước đây ta đã nói tới một cơ cấu quản lý "chỗ trống" và ta gọi là "danh sách chỗ trống". Sở dĩ như vậy vì nó được tổ chức dưới dạng stack nối đơn mà một con trỏ AVAIL trở tới đỉnh stack đó. (Hình 5.10)



ĐỖ XUÂN LÔI

# Cấu trúc dữ liệu và giải thuật



NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI