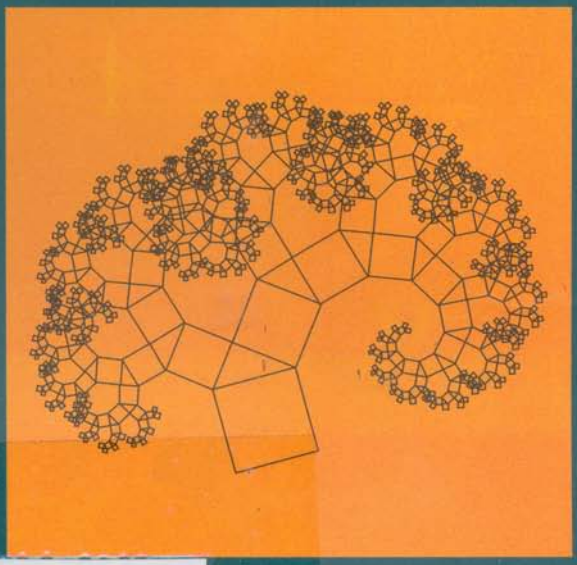
ĐỖ XUÂN LÔI

Cấu trúc dữ liệu và giải thuật

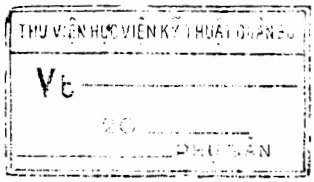


NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

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 hoạ 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), ..., (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 điển

(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 tự 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 tuỳ thuộc vào từng bài toán.

Trên cơ sở của các dữ liệu nguyên tử, các cung cách (manners) khả dĩ 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 câu 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 +, -, *, / , ↑ (luỹ thừa)
- Các dấu phép toán quan hệ <, =, >, ≤, ≥, ≠
- 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,...; S_n; **end**

với S_i, i = 1, ..., 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

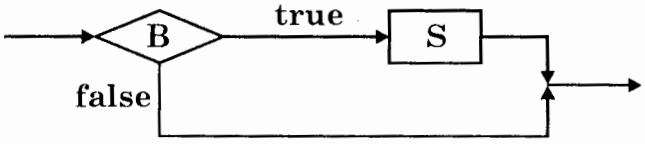
Có dạng:

if B **then** S

với B là biểu thức logic

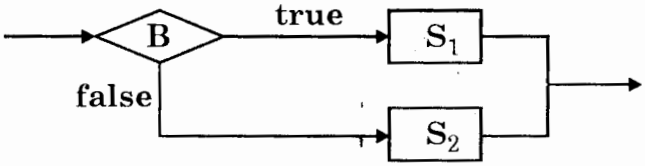
S là một câu lệnh khác

có thể diễn tả bởi sơ đồ



hoặc:

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



1.3.3.4 Câu lệnh tuyển

Case

B_1: S_1;

B_2: S_2;

...

...

B_n:S_n;

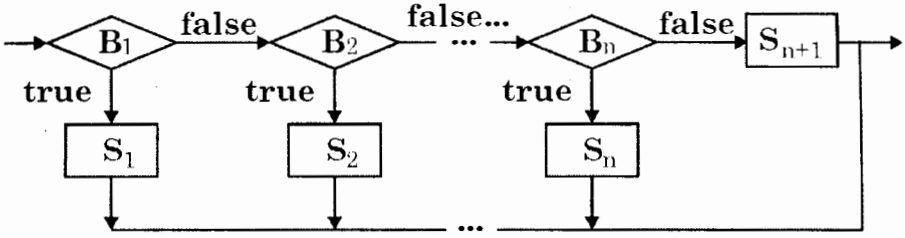
else: S_{n+1}

end case

với: B_i (i = 1, 2, ..., n) là các điều kiện

S_i (i = 1, 2, ..., 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, ..., 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 ≥m), với bước nhảy tăng bằng 1;

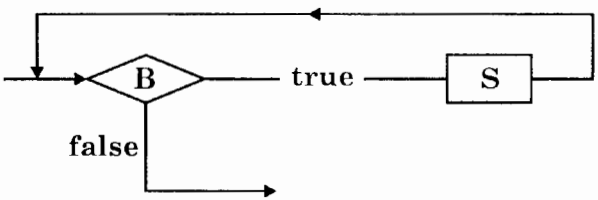
Hoặc:

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

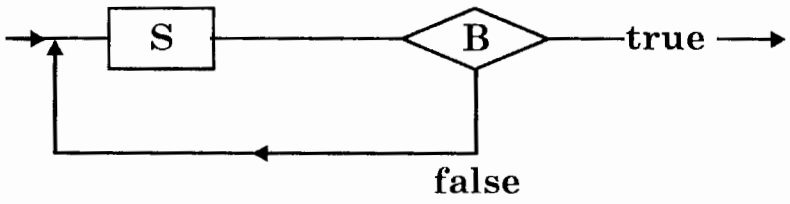


chừng nào mà B có giá trị bằng true thì thực hiện S,

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>)

$S_1, S_2, ...; S_n$

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.**

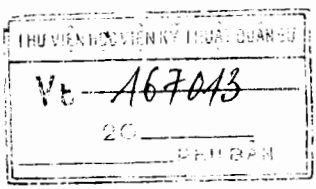
Trong cấu tạo của chương trình con hàm bao giờ cũng có câu lệnh gán mà tên hàm nằm ở vế trái. Còn đối với chương trình con thủ tục thì không có.

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 hoạ.

Chương 2

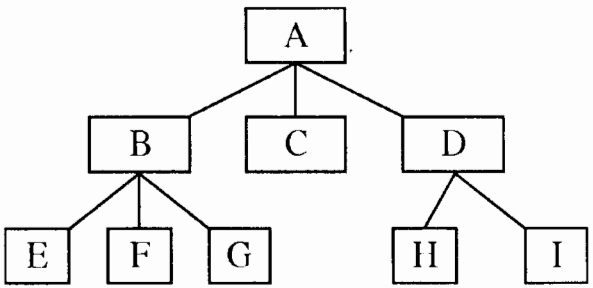
THIẾT KẾ VÀ PHÂN TÍCH GIẢI THUẬT

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.

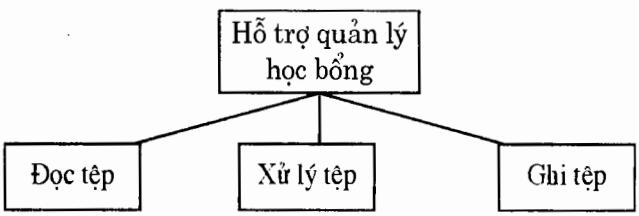
Có thể coi như ta đã có một tập các hồ sơ (mà ta gọi là *tệp* - file) bao gồm các bản ghi (records) về các thông tin liên quan tới học bổng của sinh viên, chẳng hạn: số hiệu sinh viên, điểm trung bình (theo học kỳ), điểm đạo đức, khoản tiền tài trợ. Và chương trình lập ra phải tạo điều kiện cho người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

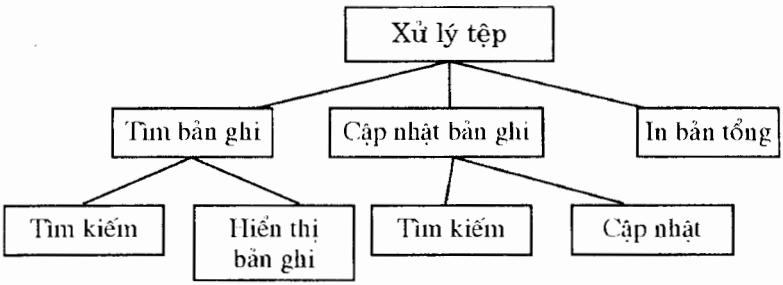


Hình 2.2

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 chỉnh 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 $(a_1, a_2, ..., a_n)$ mỗi từ a_i lưu trữ một phần tử thứ i $(1 \le i \le n)$ 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];        a[i] := a[j];        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×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:

```
11
5       0
3       7       8
4       2
9
```

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 phân 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.

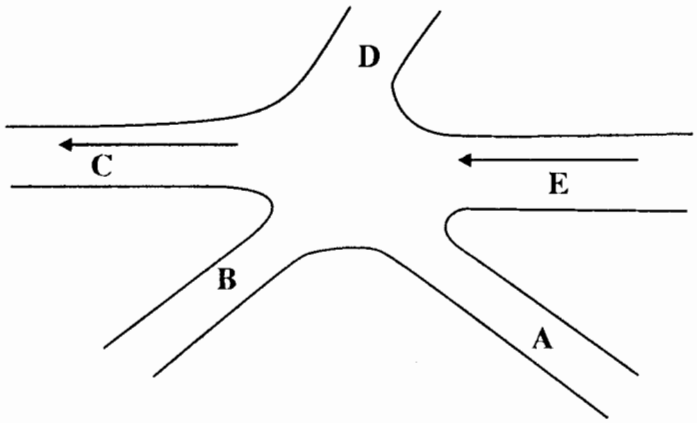
Đối với một số bài toán khác việc hình dung các dữ liệu và cấu trúc của chúng nhiều khi không còn đơn giản như thế nữa. Lựa chọn được mô hình thích hợp cho bài toán, trong đó có cả vấn đề ấn định một cấu trúc cho dữ liệu của nó, đòi hỏi một sự phân tích kỹ lưỡng dữ liệu cũng như yêu cầu đặt ra cho bài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

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 đụng độ. Ta sẽ gắn 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.

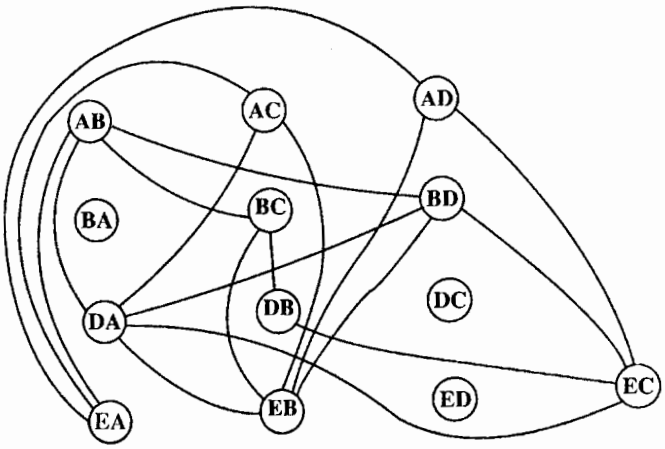
Ở đây C và E là đường một chiều (1 tuyến) còn các đường khác đều 2 chiều (2 tuyến).

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 đụng độ (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 trện (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 cấn 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 tổn 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 "dịp may hiếm có". Còn một cách khác nữa là thay đổi cách nhìn nhận về lời

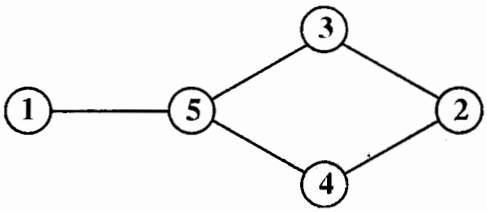
giải của bài toán đi đôi chút: Thay vì đi tìm lời giải tối ưu, ta đi tìm một lời giải "tốt" theo nghĩa là: nó đáp ứng được yêu cầu, trong một thời gian ngắn mà thực tế chấp nhận được. Một giải thuật "tốt" như vậy (tuy lời giải không phải là tối ưu) được gọi là *giải thuật heuristic*.

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 Ⓐ, như vậy sẽ có thể tô xanh cho ⒶⒸ, ⒶⒹ, ⒷⒶ nhưng không thể cho ⒷⒸ, ⒷⒹ, ⒹⒶ, ⒹⒷ; với ⒸⒹ lại được, nhưng với ⒺⒶ, ⒺⒷ, ⒺⒸ thì không. Cuối cùng cũng có thể tô xanh cho ⒺⒹ. 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	2
		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 := ∅; {Ký hiệu ∅ 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 lôgic 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 : = ∅;
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
    newcler : = 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

$T(n) = O(n^2)$ (ký hiệu chữ O lớn)

Một cách tổng quát có thể định nghĩa:

Một hàm f(n) được xác định là O(g(n))

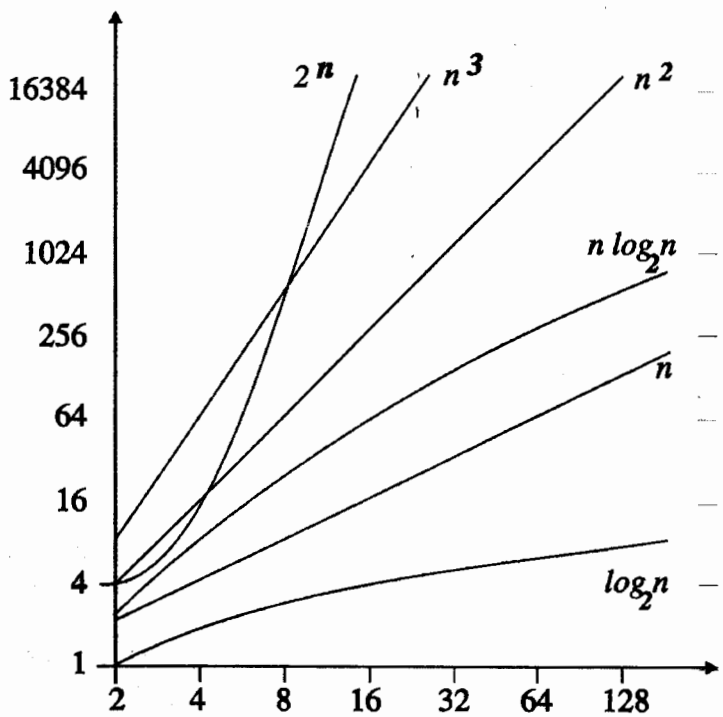
$f(n) = O(g(n))$

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

$f(n) \leq cg(n)$ 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 $g(n) \le f(n)$ với mọi $n \ge n_0$ 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 $O(cf(n)) = O(f(n))$

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:

$e^x \approx 1 + x/1 + x^2/2 + ... + x^n/n!$ với x và n 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:

$1 + 2 + ... + n = n(n + 1)/2$ 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
theo cách x^2/2! = x/1!.x/2; ... ; x^n/n! = x^(n-1)/(n-1)!.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 vectơ 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 lôgic 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: $T_{tốt} = O(1)$

$T_{xấu} = O(n)$

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:

$$T_{tb}(n) = \sum_{i=1}^{n} p_i * i + (1-q)n$$

$$= \sum_{i=1}^{n} \frac{q}{n} i + (1-q)n$$

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

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

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

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

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

Nói chung $T_{tb}(n) = O(n)$

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 hoạ.

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 đầu)

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

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;
```

Chương 3

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 tự điển chứa từ cần tìm;

if từ đó nằm ở nửa trước của tự điển.

then tìm từ đó trong nửa trước.

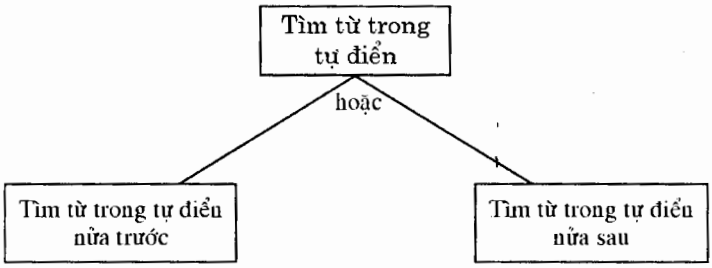
else tìm từ đó trong nửa sau.

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 tự điển vào trang giữa?
- Làm thế nào để biết từ đó nằm ở nửa nào của tự 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:

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

b) Có một trường hợp đặc biệt, khác với mọi trường hợp trước, sẽ đạt được sau nhiều lần tách đôi, đó là trường hợp tự 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

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

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* 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 thôi.

Do đó: $F(n) = F(n-2) + F(n-1)$

Vì vậy có thể tính F(n) theo:

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

Dãy số thể hiện F(n) ứng với các giá trị của n = 1, 2, 3,... 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** $n \le 2$ **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:

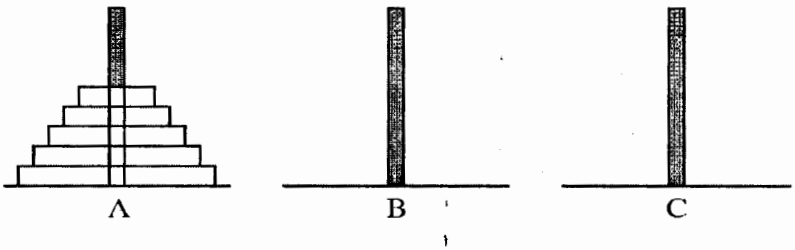
1. Có thể định nghĩa được bài toán dưới dạng một bài toán cùng loại, nhưng "nhỏ" hơn, như thế nào?
2. Như thế nào là kích thước của bài toán được giảm đi ở mỗi lần gọi đệ qui?
3. Trường hợp đặc biệt nào của bài toán sẽ được coi là trường hợp suy biến?

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:

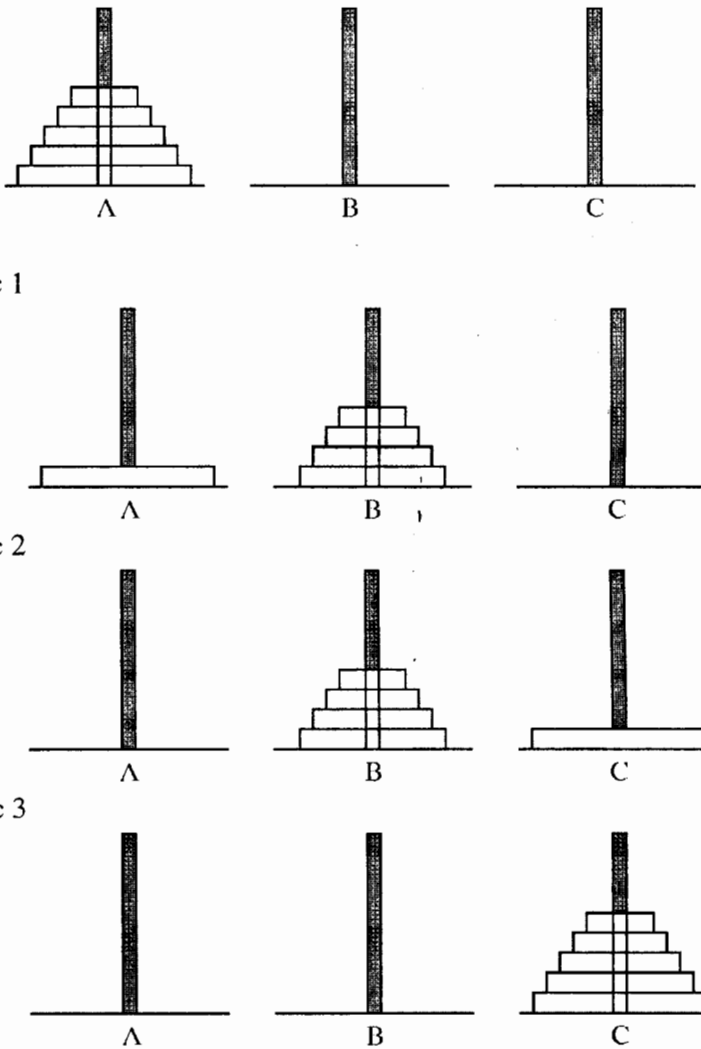
    - Chuyển đĩa thứ nhất từ cọc A sang cọc B.

- 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:



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", thoạt 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 qui 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 ↑ có các ô (i; j) mà tổng i + j không đổi, còn đường chéo theo chiều ↓ 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] = **true** có nghĩa là không có quân hậu nào chiếm hàng i.

b[i + j] = **true** có nghĩa là không có quân hậu nào chiếm được đường chéo i + j

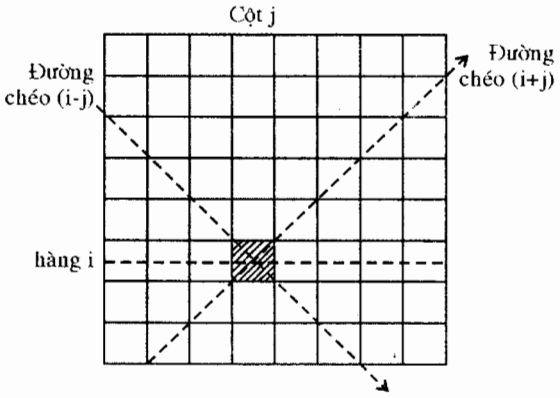
c[i - j] = **true** có nghĩa là không có quân hậu nào chiếm được đường chéo i - j.

Ta biết: $1 \le i, j \le 8$

nên suy ra $1 \le j \le 8$

$2 \le i + j \le 16$

$-7 \le i - j \le 7$



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] := **false**; b[i + j] := **false**; c[i-j] := **false;**

Còn cất quân hậu thì bởi:

a[i] = **true**; b[i + j] := **true**; c[i - j] := **true**

Thủ tục TRY bây giờ có thể viết

Procedure TRY(j, q)

```
1. i := 0;
2. repeat i := i + 1; q := false;
        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 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 ≤ 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 lấy làm lạ 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:

FACTORIAL(0) = 1

FACTORIAL(n) = n*(n-1) ...* 2 * 1 (nếu n >0)

Ta thấy: trong thủ tục có chỉ thị

if n = 0 **then** FACTORIAL := 1

chứng 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) ...* 2 * 1

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

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

FACTORIAL := n * FACTORIAL (n-1)

sẽ được thực hiện. Vậy với n = k + 1 thì giá trị cho ra là (k + 1) * FACTORIAL (k) nghĩa là (k + 1) *k *(k-1) *... * 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 với n 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:

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

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 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 ≠ 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 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 ≤ n, được định nghĩa:

$C(n, n) = 1$

$C(n, 0) = 1$

$C(n, k) = C(n-1, k-1) + C(n-1, k)$ 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 **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 $a = \{a_1, a_2, ..., a_n\}$.

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

Chương 4

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. Vectơ 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). Vectơ 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 $(a_1, a_2, ..., a_n)$ với a_i $(1 \le i \le n)$ 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 $1 \le i \le n - 1$ thì có một phần tử a_{i+1} gọi là *phần tử sau* a_i, và với $2 \le i \le n$ 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:

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

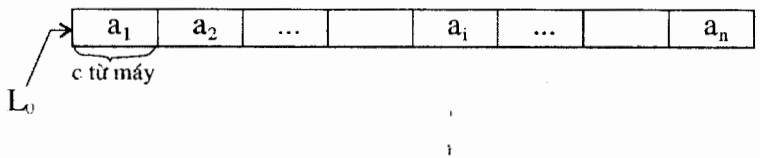
Sau đây, ta sẽ xét tới cách tổ chức lưu trữ mảng.

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 vectơ.

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 vectơ 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 vectơ đã được xác định nên không gian nhớ dành ra cũng đã được ấn định trước.

Một cách tổng quát, một véc tơ A có n phần tử nếu mỗi phần tử a_i ($1 \le i \le n$) 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:

$$Loc(a_i) = L_0 + c * (i-1)$$

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 vectơ (mà ta gọi là vectơ lưu trữ).

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

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:

$$Loc(a_i) = L_0 + c * (i - b)$$

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

Ví dụ trong FORTRAN một ma trận 3 hàng 4 cột (a_{ij}) với $1 \le i \le 3$, $1 \le j \le 4$ 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

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

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

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

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

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

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

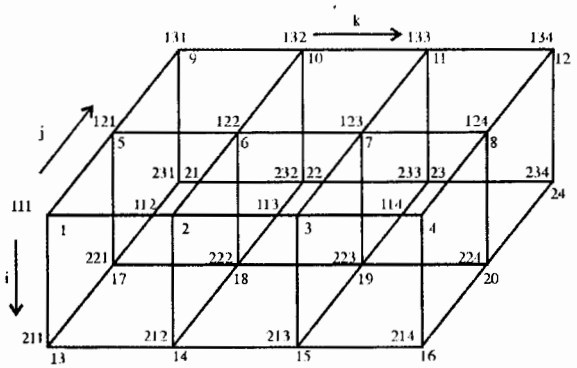
$$Loc(a_{ij}) = L_o + (i - b_1) * (u_2 - b_2 + 1) + (j - b_2)$$

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 $1 \le i \le 2$; $1 \le j \le 3$: $1 \le k \le 4$; 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_{131}, b_{132}, b_{133}, b_{134}, 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:

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

Ví dụ:

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

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

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

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^iy^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 5x5 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

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

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 vectơ lưu trữ (V_i) với $1 \leq i \leq n$ để lưu trữ một danh sách tuyến tính (a_1, a_2, ..., 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 đấy" 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 dồn 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.

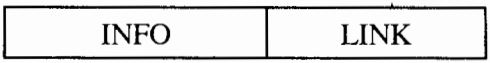
4.4 Lưu trữ móc nối đối với danh sách tuyến tính

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:



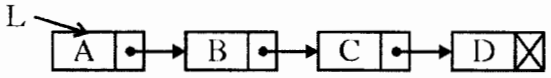
Trường INFO chứa thông tin của phần tử.

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* - signly 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 mắt 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 trỏ 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}

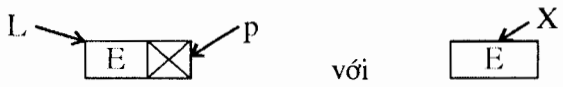
```
if L = null then begin
                    L := p
                    LINK(p) := null;
                    end
else begin
        LINK(p) := LINK(M);
        LINK(M) := p
      end;
```

3. **return**

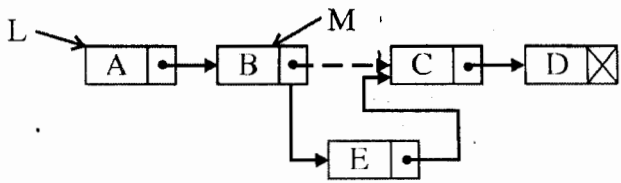
* Phép xử lý được minh hoạ bởi hình sau:

a) Nếu L = null

thì sau phép bổ sung ta có:



b) Nếu nút L ≠ null



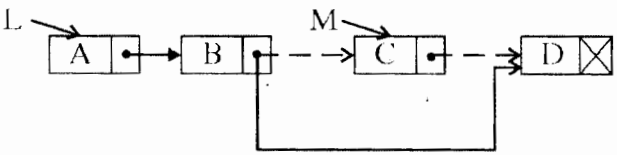
4.4.2.2 Loại bỏ một nút ra khỏi danh sách nối đơn

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:= LINK(M);
   call dispose(M)
   return
   end;
3. {Tìm đến nút đứng trước nút trỏ bởi M}
   P := L;
   while LINK(p) ≠ M do P := LINK(p);
4. {Loại bỏ nút trỏ bởi M}
   LINK(p) := LINK(M);
5. {Đưa nút bị loại về danh sách chỗ trống}
   call 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 = null **then return;**
2. {Trường hợp danh sách trỏ bởi P rỗng}
   if P = null **then begin**
   P:= Q;
   return
   end;
3. {Tìm đến nút cuối danh sách}
   P1 := P;
   while LINK(P1) ≠ null **do** P1 := 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:

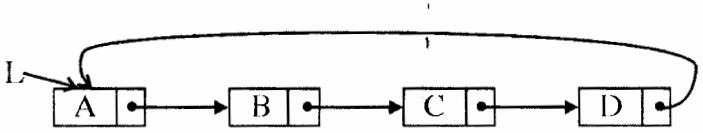
- Chỉ có phần tử đầu tiên trong danh sách được truy nhập trực tiếp còn các phần tử khác chỉ được truy nhập sau khi đã qua các phần tử đứng trước nó.

- 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: mối 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:



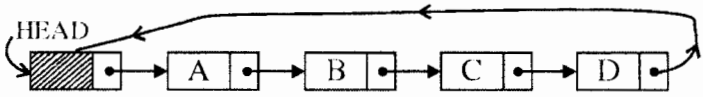
Cải tiến này làm cho việc truy nhập vào các nút trong danh sách được linh hoạt hơn. Ta có thể truy nhập vào mọi nút trong danh sách bắt đầu từ một nút nào cũng được, không nhất thiết phải từ nút đầu tiên. Điều đó cũng có nghĩa là nút nào cũng có thể coi là nút đầu tiên và con trỏ L trỏ tới nút nào cũng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

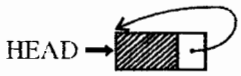
Tuy nhiên, danh sách nối vòng có một nhược điểm rất rõ là trong xử lý, nếu không cẩn thận sẽ dẫn tới một chu trình không kết thúc. Sở dĩ như vậy là vì không biết được chỗ kết thúc của danh sách.

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 LINK(HEAD) = 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)

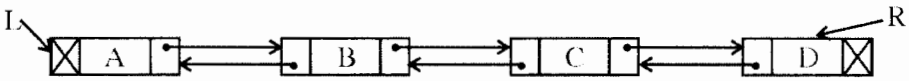
Với các kiểu danh sách như đã nêu ta chỉ có thể duyệt qua danh sách theo một chiều. Trong một số ứng dụng đôi khi vẫn xuất hiện yêu cầu đi ngược lại. Để có được cả hai khả năng, cần phải đặt ở mỗi nút hai con trỏ, một trỏ tới nút đứng trước nó và một trỏ tới nút đứng sau nó. Như vậy nghĩa là qui cách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

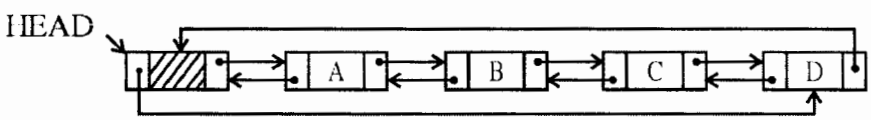
```
        LPTR(M) := p;
        RPTR(LPTR(p)) := p;
        return
```

Procedure DOUBDEL(L,R,M)

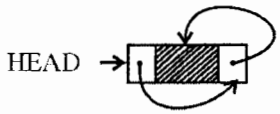
{Cho L và R là hai con trỏ trái và phải của danh sách nối kép, M trỏ tới một nút trong danh sách. Giải thuật này thực hiện việc loại nút trỏ bởi M ra khỏi danh sách}

```
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:

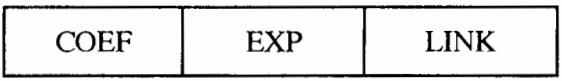
```
RPTR(LPTR(M)) := RPTR(M);
LPTR(RPTR(M)) := LPTR(M);
call dispose(M);
```

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:

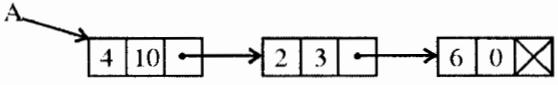


Trường COEF chứa hệ số khác không của một số hạng trong đa thức.

Trường EXP chứa số mũ tương ứng.

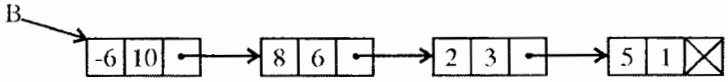
Trường LINK chứa mối nối tới nút tiếp theo. Như vậy đa thức

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



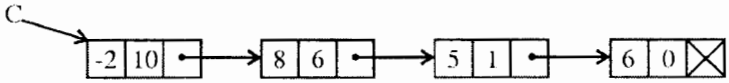
Còn đa thức:

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

a) 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).

b) Nếu EXP(p) > EXP(q) (hoặc ngược lại thì cũng tương tự): phải sao chép nút p và gán vào danh sách của C(x).

c) 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 ≠ null and q ≠ null do
    case
    EXP(p) = EXP(q): x:= COEF(p) + COEF(q);
                     if x ≠ 0 then
                     call ATTACH(x,EXP(p),d);
                     p:= LINK(p); q := LINK(q);
    EXP(p)<EXP(q): call ATTACH(COEF(q), EXP(q), d);
                     q:= LINK(q);
          else :     call ATTACH(COEF(p), EXP(p),d);
                     p := LINK(p);
end case;
4. {Danh sách ứng với B(x) đã hết}
    while p ≠ null do begin
            call ATTACH(COEF(p), EXP(p),d);
            p:= LINK(p)
                 end;
5. {Danh sách ứng với A(x) đã hết}
    while q ≠ null do begin
                 call ATTACH(COEF(q), EXP(q), d);
                 q := LINK(q)
                   end;
6. {Kết thúc danh sách C}
    LINK(d) := null;
7. {Loại bỏ nút đóng vai trò đuôi giả lúc đầu}
    t := C; C := LINK(C); 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 trỏ 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 := LINK(A); q:= LINK(B);

Bước 3 sẽ là: **while** p ≠ A **and** q ≠ B **do ...**

Bước 4 sẽ là: **while** p ≠ A **do...**

Bước 5 sẽ là: **while** q ≠ B **do...**

Bước 6 sẽ là: LINK(d) := C

Bước 7: bỏ

* Nếu bây giờ ta sử dụng thêm trường EXP của nút đầu danh sách và gán cho 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 EXP(p) = - 1 mà - 1 < 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 := LINK(A); q := LINK(B);
2. **call** new(C); d := C;
3. **while true do**
   case
   EXP(p) = EXP(q): **if** EXP(p) = -1 **then exit;**
   x := COEF(p) + COEF(q);
   if x ≠ 0 **then**
   call ATTACH(x, EXP(p),d);
   p := LINK(p); q := LINK(q);
   EXP(p) < EXP(q): **call** ATTACH(COEF(q), EXP(q), d);
   q:= LINK(q);
   else : **call** ATTACH (COEF(p), EXP(p),d);
   p := LINK(p)

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

4. **return**

BÀI TẬP CHƯƠNG 4

4.1. Dựa trên đặc điểm gì để phân biệt vectơ 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 $1 \le i \le 7$, $1 \le j \le 6$. 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:

a) Theo thứ tự ưu tiên cột.

b) 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

$-2 \le i \le 6$

$-5 \le j \le -1$

$3 \le k \le 10$

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 vectơ B theo thứ tự ưu tiên hàng. Chẳng hạn:

$$\begin{cases} A[1;1] & \text{lưu trữ} & \text{tại B [1]} \\ A[1;2] & \text{lưu trữ} & \text{tại B [2]} \\ A[2;1] & \text{lưu trữ} & \text{tại B [3]} \\ A[2;2] & \text{lưu trữ} & \text{tại B [4]} \\ A[2;3] & \text{lưu trữ} & \text{tại B [5]} \end{cases}$$

. . . . . .

Hãy lập giải thuật cho phép xác định được giá trị của A[i;j] từ vectơ B, khi biết i và j ($1 \le i; j \le n$). 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:

$$a_{11}x_1 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$
$$...$$
$$a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n$$

Để tiết kiệm, người ta lưu trữ ma trận hệ số của hệ phương trình dưới dạng một vectơ. 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 vectơ lưu trữ có độ dài n +2 để biểu diễn một đa thức:

$$p(x) = a_nx^n + a_{n-1}x^{n-1} + ... + a_1x + a_0$$

theo dạng:

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

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 vectơ 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.

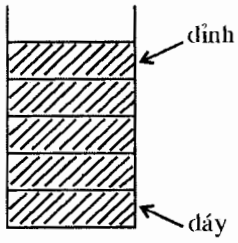
Chương 5

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.

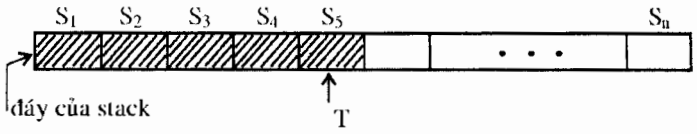


Hình 5.1

Nguyên tắc "vào sau ra trước" như vậy của stack đã đưa tới một tên gọi khác: danh sách kiểu LIFO (last - in - firts - 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 ≥ 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 ≤ 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];

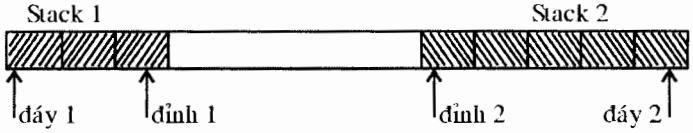
4. **return**

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?

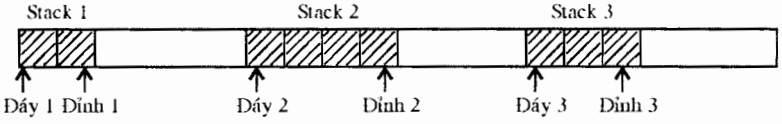
Nếu là hai stack thì có thể giải quyết dễ dàng. Ta không quy định kích thước tối đa cho từng stack nữa mà không gian nhớ dành ra sẽ được dùng chung. Ta sẽ đặt hai stack ở hai đầu sao cho hướng phát triển của chúng ngược nhau, như ở hình 5.3



Hình 5.3

Như vậy có thể một stack này dùng lấn sang quá nửa 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 dồn 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 $4.10^2 + 3.10^1 + 7.10^0$

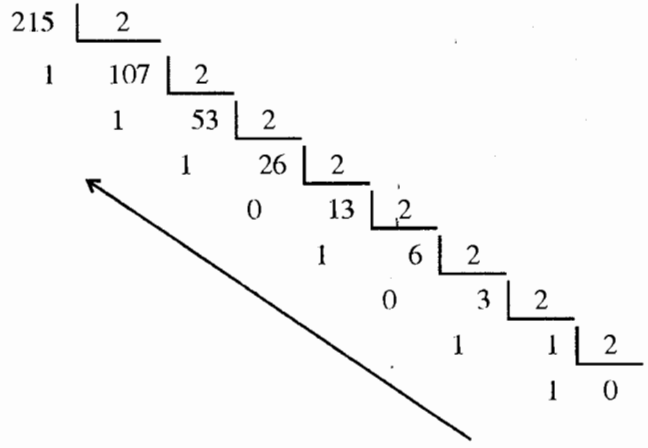
Với số ở hệ nhị phân cũng tương tự.

Chẳng hạn:

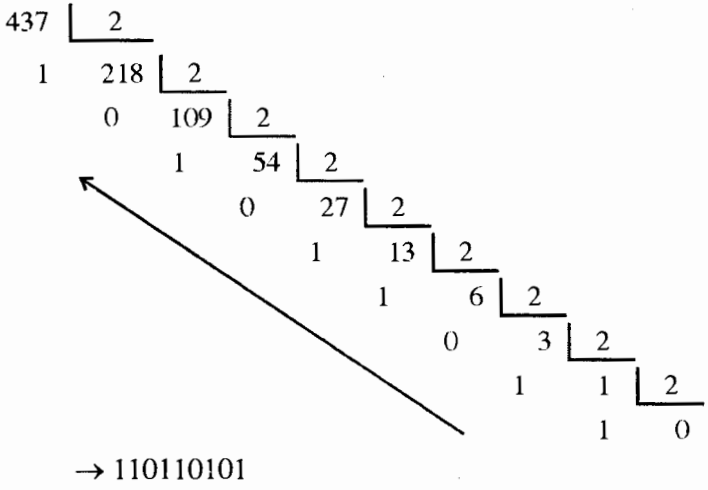
11010111 thì biểu diễn số

$1.2^7 + 1.2^6 + 0.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = (215)_{10}$

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

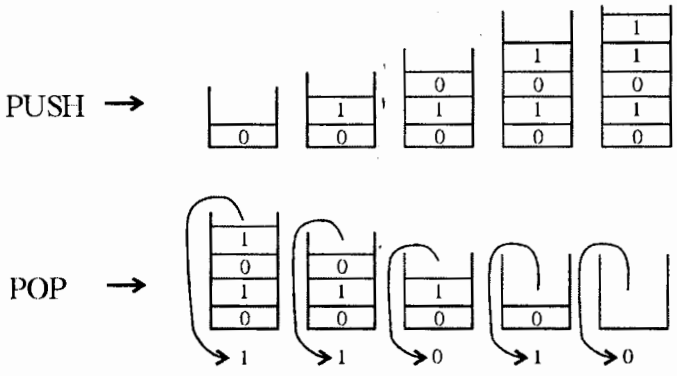


→ 11010111



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 ≠ 0 **do begin**
   R := N mod 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 luỹ thừa (dấu phép toán được ký hiệu bởi ↑).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 luỹ 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 ↑ F

thì thứ tự thực hiện sẽ như sau:

E ↑ F

B * C

D/ (E ↑ F)

A + (B * C)

(A + B * C) - (D/E ↑ 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; thoạt 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

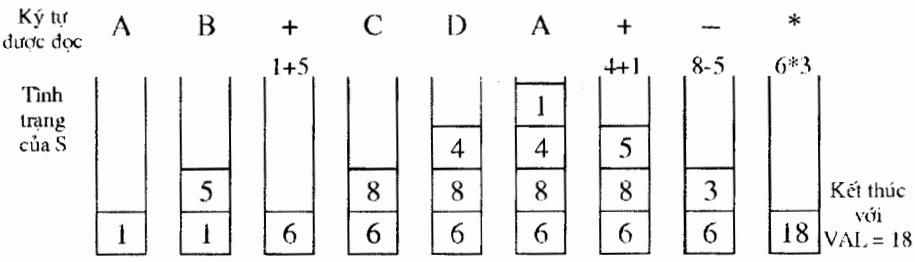
A B + C D A + - *

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:



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, thoạt 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 Ⓧ: **While** thứ tự ưu tiên của S[T] ≥ thứ tự ưu tiên của Ⓧ **do begin**

   call POP (S, T, W);

   Bổ sung thêm W vào P

   end;

   call PUSH (S, T, 'Ⓧ ');

   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 thoạt 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:

1) 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.

2) 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.

3) Cuối cùng khi stack S rỗng thì ta sẽ có P dưới dạng:

P : ABC * DEF ↑ /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

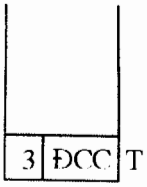
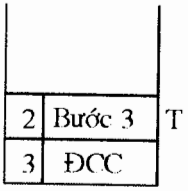
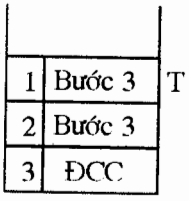
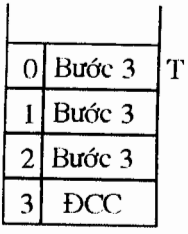
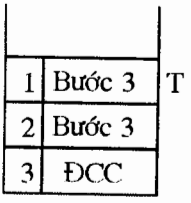
ADDRESS ứng với RETADD

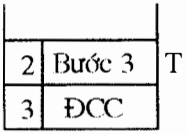
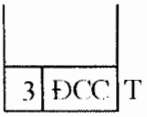
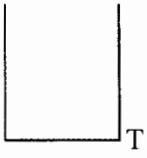
Ở đây đặt giả thiết: Lúc đầu TEMREC đã chứa các giá trị cần thiết, nghĩa là PARA chứa giá trị n đã cho, ADDRESS chứa địa chỉ ứng với lời gọi trong chương trình chính mà ta gọi là ĐCC (viết tắt của địa chỉ chính).

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 trỏ 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) * 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,ĐCC)) Bước 2: N≠ 0 PARA := 2; ADDRESS:= Bước 3	
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	
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	
Vào mức 4 (gọi đệ qui lần 3)	Bước 1: PUSH (A, 3, (0, bước 3)) Bước 2: N = 0; FACTORIAL :=1	
Quay lại mức 3	Bước 4: POP(A, 4, TEMREC); **goto** Bước 3 Bước 3: FACTORIAL := 1 * 1	

	Bước 4: POP(A, 3, TEMREC); **goto** Bước 3	
Quay lại mức 2	Bước 3: FACTORIAL := 2 * 1 POP(A, 2, TEMREC) **goto** Bước 3	
Quay lại mức 1	Bước 3: FACTORIAL := 3 * 2 Bước 4: POP(A, 1, TEMREC); **goto** ĐCC	

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);

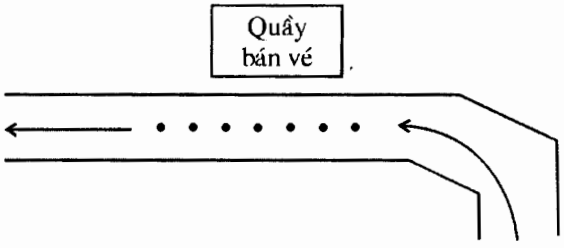
go to RETADD (T)

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).

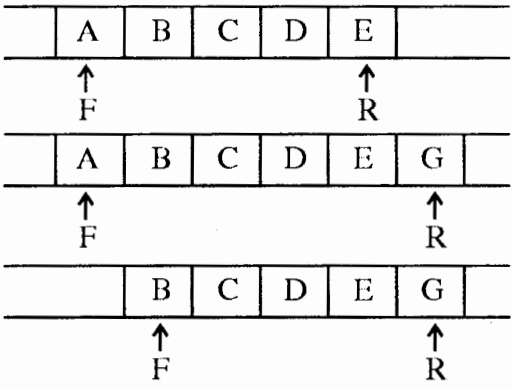


Hình 5.7

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

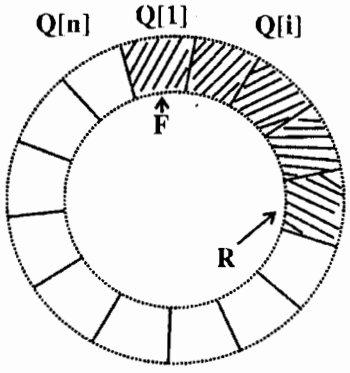
Có thể dùng một vectơ lưu trữ Q có n phần tử làm cấu trúc lưu trữ của queue. Để có thể truy cập vào queue ta phải dùng hai biến trỏ: R trỏ tới lối

sau và F trỏ tới lối 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 vectơ 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);

{Ở đây 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. {Chỉnh 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 ('CẠN');
        return
            end;
```

2. {Bảo lưu thông tin của phần tử bị loại}

```
Y:= Q[F];
```

3. {Chỉnh 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.

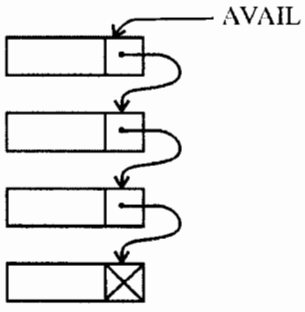
5.7 Stack và queue móc nối

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 trỏ bởi L thì có thể coi L như con trỏ trỏ 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 trỏ bởi L. Trong thủ tục bổ sung với stack móc nối ta không phải kiểm tra hiện tượng TRÀN 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 trỏ 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 trỏ bởi L, còn lối sau được trỏ 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 trỏ tới đỉnh stack đó. (Hình 5.10).



Hình 5.10

Như vậy phép: **call** new(p), tương ứng với việc gọi thủ tục:

Procedure NEW(p);

1. **if** AVAIL = null **then begin**
   write ('CẠN');
   return
   end;
2. p := AVAIL;
   AVAIL := LINK(AVAIL);
3. **return**

Còn phép **call** dispose(p), tương ứng với việc gọi thủ tục

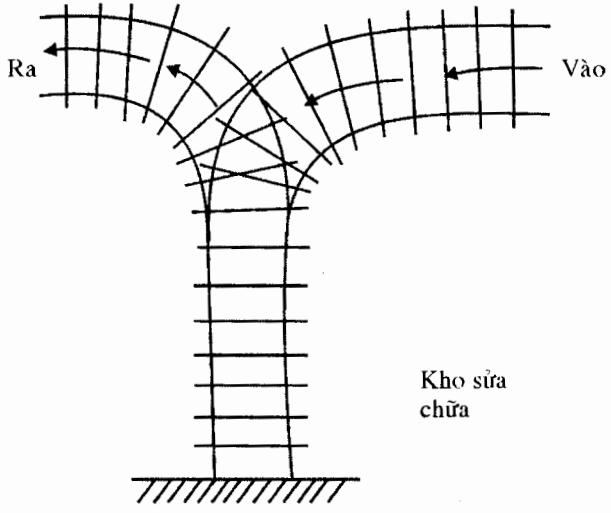
Procedure DIPOSE(p);

1. LINK(p):= AVAIL;
   AVAIL := p;
2. **return**

Cần chú ý rằng: lúc đầu có thể không gian nhớ dành cho "danh sách chỗ trống" là một miền kế tiếp. Nó sẽ được phận thành nút có cùng kích thước nào đấy, các nút đều có các trường dữ liệu qui định và sẽ được móc nối với nhau, chẳng hạn theo danh sách kiểu nối đơn, có con trỏ AVAIL trỏ tới nút đầu tiên (đỉnh). Nhưng sau, do yêu cầu về cấp phát cũng như thu hồi luôn biến động, không theo một qui luật nào, nên xuất hiện tình trạng các nút trong "danh sách chỗ trống" không còn kế tiếp với nhau nữa.

BÀI TẬP CHƯƠNG 5

5.1. Xét một cơ cấu đường tàu vào kho sửa chữa như hình sau:



Giả sử ở đường vào có 4 đầu tàu được đánh số 1, 2, 3, 4. Gọi V là phép đưa một đầu tàu vào kho sửa chữa, R là phép đưa một đầu tàu từ kho ra. Nếu ta thực hiện dãy VVRVVRRR thì thứ tự các đầu tàu lúc ra sẽ là 2 4 3 1 (kho sửa chữa có cơ cấu như một stack). Như vậy có thể coi như ta đã làm một phép hoán vị thứ tự đầu tàu.

Xét trường hợp có 6 đầu tàu: 1, 2, 3, 4, 5, 6 có thể thực hiện một dãy các phép V và R thế nào để đổi thứ tự đầu tàu ở đường ra là 3 2 5 6 4 1? 1 5 4 6 2 3?

5.2. Hãy biến đổi số thập phân 1774 sang dạng nhị phân.

Minh hoạ tình trạng của stack S được sử dụng để lưu trữ các số dư trong quá trình này.

5.3. Dựa vào qui tắc của ký pháp hậu tố, tiền tố hãy viết biểu thức dạng hậu tố, dạng tiền tố tương ứng với biểu thức dạng trung tố dưới đây:

a) (A + B ↑ C) * D/E - (F + G)

b) ((A + B) * D) ↑ (E - F)

5.4. Cho biểu thức hậu tố P

P : A B C - / D E F + * +

a) Hãy viết biểu thức dạng trung tố, dạng tiền tố tương ứng với P.

b) Minh hoạ tình trạng của stack qua các bước thực hiện giải thuật EVAL để tính giá trị của P, ứng với A = 12, B = 7, C = 3, D = 2, E = 1, F = 5.

5.5. a) Cho biểu thức Q dưới dạng trung tố:

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

Hãy áp dụng giải thuật POLISH để biến đổi Q sang dạng hậu tố P. Nêu rõ tình trạng của stack và dạng hình thành của P qua từng bước thực hiện, kèm theo những thuyết minh cần thiết.

b) Thực hiện yêu cầu tương tự như câu a), với biểu thức

$$Q: (d \uparrow e - f) * (g/h + i) - a/b + c.$$

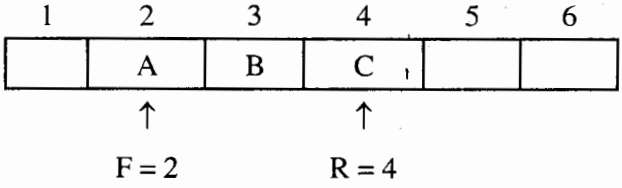
c) Tính giá trị của biểu thức hậu tố P tương ứng với Q ở câu b) theo các giá trị của biến:

$$d = 6; e = 2; f = 9; g = 12; h = 4; i = 1; a = 27; b = 3; c = 11.$$

5.6. Hãy lập giải thuật bổ sung và loại bỏ đối với hai stack hoạt động theo hướng ngược chiều nhau trên một miền nhớ cho phép là một vectơ V có n phần tử (đáy của stack 1 sẽ là V[1], còn đáy của stack 2 là V[n]). Giả sử mỗi phần tử của stack ứng với một phần tử của V.

5.7. Hãy dựng hình ảnh minh hoạ tình trạng của stack trong quá trình thực hiện giải thuật HANOI-TOWER ứng với 3 đĩa.

5.8. Cho một queue được lưu trữ trong bộ nhớ bởi vectơ Q có 6 ô nhớ, được hoạt động theo cấu trúc vòng tròn. Thoạt đầu queue có dạng



Các chữ A, B, C, ... ở đây biểu thị cho thông tin ứng với các phần tử của queue).

Hãy vẽ tình trạng của Q và nêu rõ các giá trị tương ứng của F và R sau mỗi lần thực hiện các phép toán dưới đây:

a) Phần tử E được bổ sung vào queue.

b) Loại hai phần tử ra khỏi queue.

c) Các phần tử I, J, K được bổ sung vào queue.

d) Loại hai phần tử ra khỏi queue.

e) Phần tử O được bổ sung vào queue.

f) Loại hai phần tử ra khỏi queue.

g) Phần tử W được bổ sung vào queue.

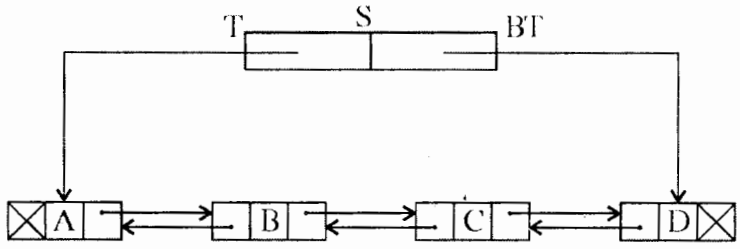
h) Loại một phần tử ra khỏi queue.

5.9. Cho một danh sách nối kép có 2 con trỏ L và R lần lượt trỏ tới nút cực trái và cực phải của danh sách

a) Hãy viết giải thuật bổ sung, loại bỏ để danh sách này hoạt động như một queue.

b) Có thể cho nó hoạt động như một stack được không?

5.10. Nếu tổ chức stack theo kiểu danh sách nối đơn như đã nêu trong phần 4.5. của bài giảng thì ta có thể duyệt qua stack đó (nghĩa là thăm lần lượt các phần tử của stack) từ đỉnh tới đáy một cách dễ dàng. Nhưng nếu muốn duyệt theo chiều ngược lại thì rất khó. Để giải quyết yêu cầu này người ta tổ chức stack theo kiểu nối kép. Có hai con trỏ trỏ tới đỉnh và đáy của stack. Con trỏ T trỏ tới đỉnh stack đặt ở trường TOP của một nút S, con trỏ BT trỏ tới đáy stack đặt ở trường BOTTOM của S. Có thể minh hoạ như sau:



Hãy lập giải thuật:

a) BTRAVERSE: thực hiện phép duyệt qua stack từ đáy lên.

b) PUSH: thực hiện phép bổ sung một phần tử vào stack.

c) POP: thực hiện phép loại bỏ một phần tử ra khỏi stack.

Chương 6

CÂY

6.1. Định nghĩa và các khái niệm

Cấu trúc phi tuyến đầu tiên mà ta xét tới ở đây là cấu trúc cây (tree).

Một cây (tree) là một tập hợp hữu hạn các nút trong đó có một nút đặc biệt được gọi là *gốc* (root) giữa các nút có một quan hệ phân cấp gọi là quan hệ "cha - con".

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

1. Một nút là một cây. Nút đó cũng là gốc của cây ấy.

2. Nếu $T_1, T_2, ..., T_k$ là các cây, với $n_1, n_2, ..., n_k$ lần lượt là các gốc. n là một nút và n có quan hệ cha - con với $n_1, n_2, ..., n_k$ thì lúc đó một cây mới T sẽ được tạo lập, với n là gốc của nó. n được gọi là cha của $n_1, n_2, ..., n_k$; ngược lại $n_1, n_2, ..., n_k$ được gọi là con của n. Các cây $T_1, T_2, ..., T_k$ được gọi là các cây con (substrees) của n.

Người ta quy ước: Một cây không có nút nào được gọi là *cây rỗng* (null tree).

Có nhiều đối tượng có cấu trúc cây.

- **Ví dụ:** Mục lục của một cuốn sách, hoặc một chương trong sách, có cấu trúc cây.

Chẳng hạn mục lục của chương 6 này:

Chương 6: Cây

6.1. Định nghĩa và các khái niệm

6.2. Cây nhị phân

6.2.1. Định nghĩa và tính chất của cây nhị phân

6.2.2. Biểu diễn cây nhị phân

6.2.3. Phép duyệt cây nhị phân

6.2.4. Cây nhị phân nối vòng

6.3. Cây tổng quát

6.3.1 Biểu diễn cây tổng quát

6.3.2. Phép duyệt cây tổng quát

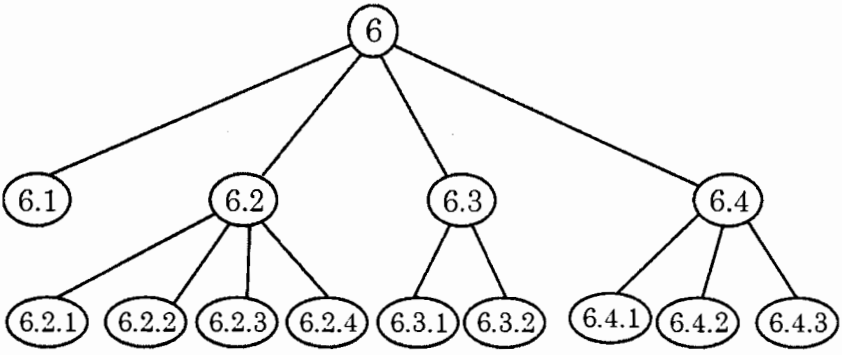
6.4. Áp dụng

6.4.1. Cây biểu diễn biểu thức

6.4.2. Cây biểu diễn các tập

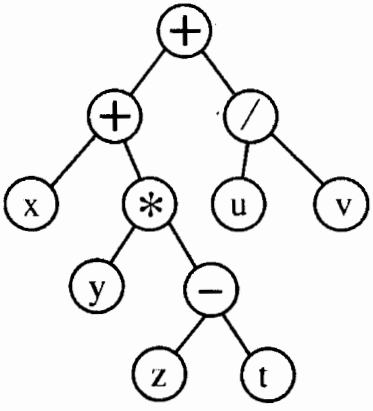
6.4.3. Cây quyết định

Ta có thể biểu diễn bằng một cây có dạng như sau:



Hình 6.1

* Biểu thức số học x + y * (z - t) + u/v, ta có thể biểu diễn dưới dạng cây như hình 6.2.



Hình 6.2

* Các tập bao nhau như hình 6.3 có thể biểu diễn bởi cây như hình 6.4.

* Đối với cây, chẳng hạn xét cây ở hình 6.4.

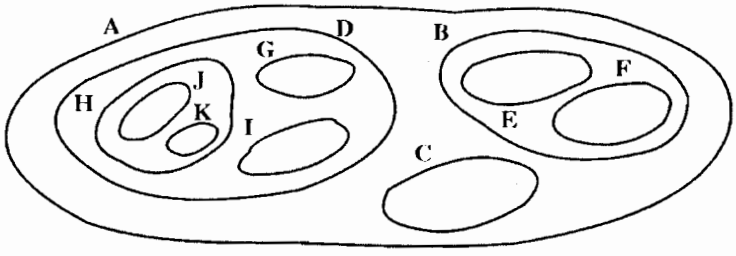
Nút A được gọi là gốc của cây.

B, C, D là gốc của các cây con của A

A là cha của B, C, D; còn B, C, D là con của A.

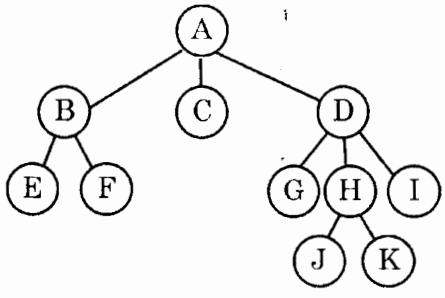
* Số các con của một nút gọi là *cấp* (degree) của nút đó. Ví dụ cấp của A là 3, cấp của H là 2.

Nút có cấp bằng không gọi là *lá* (leaf) hay *nút tận cùng* (termimal node). Ví dụ các nút E, C, K, I.v.v... Nút không là lá được gọi là *nút nhánh* (branch node).



Hình 6.3

Cấp cao nhất của nút trên cây gọi là *cấp của cây* đó. Cây ở hình 6.4 là cây cấp 3.



Hình 6.4

* Gốc của cây có số *mức* (level) là 1. Nếu nút cha có số mức là i thì nút con có số mức là i+1. Ví dụ nút A có số mức là 1

D có số mức là 2

G có số mức là 3

J có số mức là 4

* *Chiều cao* (heigh) hay *chiều sâu* (depth) của một cây là số mức lớn nhất của nút có trên cây đó.

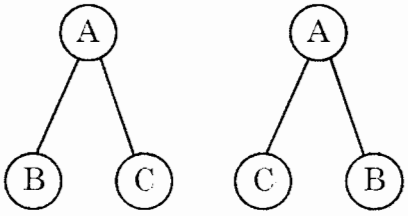
Cây ở hình 6.2 có chiều cao là 5.

Cây ở hình 6.4 có chiều cao là 4.

Nếu n_1, n_2, ..., n_k là dãy các nút mà n_i là cha của n_{i+1} với $1 \le i < k$, thì dãy đó gọi là *đường đi* (path) từ n_1 tới n_k. *Độ dài của đường đi* (path length) bằng số nút trên đường đi trừ đi 1. Ví dụ trên cây hình 6.4 độ dài đường đi từ A đến G là 2, từ A tới K là 3.

* Nếu thứ tự các cây con của một nút được coi trọng thì cây đang xét là *cây thứ tự* (ordered tree), ngược lại là cây *không có thứ tự* (unordered tree). Thường thứ tự các cây con của một nút được đặt từ trái sang phải.

Hình 6.5 cho ta hai *"cây có thứ tự"* khác nhau:



Hình 6.5

Đối với cây, từ quan hệ cha con người ta có thể mở rộng thêm các quan hệ khác phỏng theo các quan hệ như trong gia tộc.

* Nếu một tập hữu hạn các cây phân biệt thì ta gọi đó là *rừng* (forest).

Khái niệm về rừng ở đây phải hiểu theo cách riêng vì: có một cây, nếu ta bỏ nút gốc đi ta sẽ có một rừng! Như ở hình 6.4 nếu bỏ nút gốc A đi, ta sẽ có một rừng gồm 3 cây.

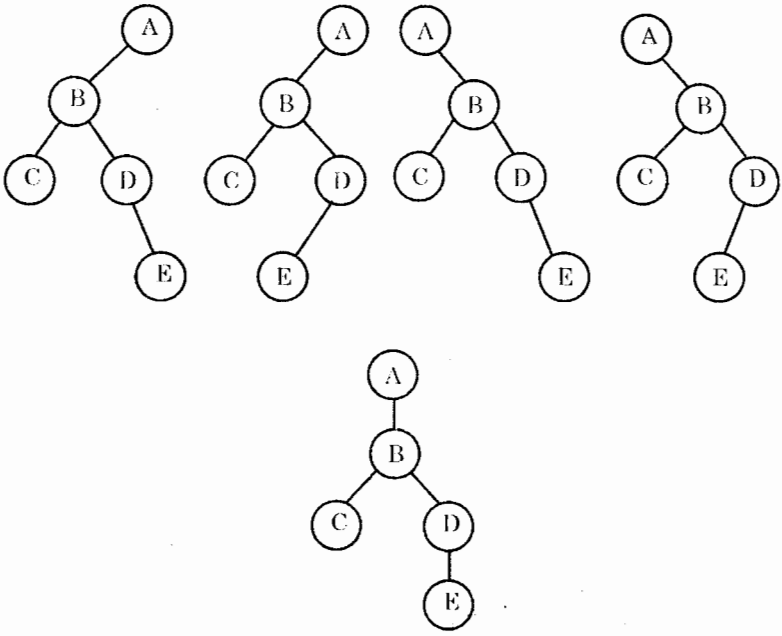
6.2 Cây nhị phân (Binary tree)

6.2.1 Định nghĩa và tính chất

Cây nhị phân là một dạng quan trọng của cấu trúc cây. Nó có đặc điểm là: Mọi nút trên cây chỉ có tối đa là hai con. Đối với cây con của một nút người ta cũng phân biệt *cây con trái* (left subtree) và *cây con phải* (right subtree). Như vậy cây nhị phân là cây có thứ tự.

Ví dụ: Cây ở hình 6.2 là cây nhị phân với toán tử ứng với gốc, toán hạng 1 ứng với cây con trái, toán hạng 2 ứng với cây con phải. Các cây nhị phân

sau đây là khác nhau, nhưng nếu coi là cây không có thứ tự thì chúng chỉ là 1 (hình 6.6).



Hình 6.6

Cũng cần chú ý tới một số dạng đặc biệt của cây nhị phân (hình 6.7)

Các cây a) b) c) d) được gọi là *cây nhị phân suy biến* (degenerate binary tree) vì thực chất nó có dạng của một danh sách tuyến tính.

Riêng cây a) được gọi là *cây lệch trái*

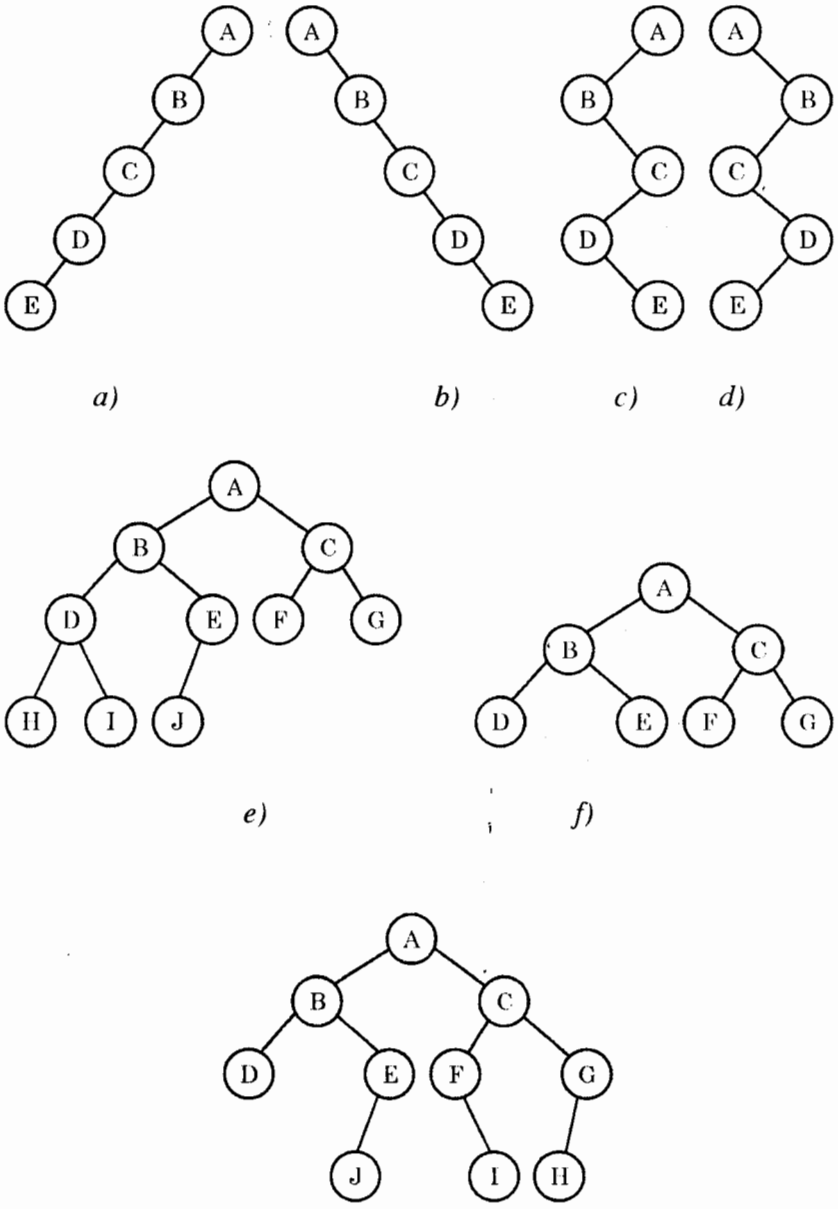
Riêng cây b) được gọi là *cây lệch phải*

Riêng cây c) và d) được gọi là *cây zic - zắc*

Cây e) được gọi là *cây nhị phân hoàn chỉnh* (complete binary tree). Ở đây ta thấy: Các nút ứng với các mức trừ mức cuối cùng đều đạt tối đa và ở mức cuối các nút đều đạt về phía trái. Riêng f) có các nút tối đa ở cả mọi mức nên còn gọi là *cây nhị phân đầy đủ* (full binary tree). Cây nhị phân đầy đủ là một trường hợp đặc biệt của cây cây nhị phân hoàn chỉnh. Cây g) khác với cây e) ở chỗ các nút ở mức cuối không dạt về phía trái, ta sẽ gọi là *cây gần đầy*.

Ta thấy ngay: Trong các cây nhị phân cùng có số lượng nút như nhau thì cây nhị phân suy biến có chiều cao lớn nhất, còn cây nhị phân hoàn chỉnh hoặc cây nhị phân gần đầy thì có chiều cao nhỏ nhất, loại cây này cũng là cây có dạng "cân đối" nhất.

Tất cả các khái niệm đã nêu ở 6.1 đều có thể áp dụng vào cây nhị phân.



Hình 6.7

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

Bổ đề

1) Số lượng tối đa các nút ở mức i trên một cây nhị phân là

$$2^{i-1} \ (i \geq 1)$$

2) Số lượng tối đa các nút trên một cây nhị nhân có chiều cao h là

$$2^h - 1 \ (h \geq 1)$$

*** Chứng minh:**

1) Sẽ được chứng minh bằng qui nạp: Ta biết:

Ở mức 1: i = 1, cây nhị phân có tối đa $1 = 2^0$ nút.

Ở mức 2: i = 2, cây nhị phân có tối đa $2 = 2^1$ nút.

Giả sử kết quả đúng với mức i - 1, nghĩa là ở mức này cây nhị phân có tối đa là 2^{i-2} nút. Mỗi nút ở mức i - 1 sẽ có tối đa hai con vậy 2^{i-2} nút ở mức i - 1 sẽ cho:

$2^{i-2} \times 2 = 2^{i-1}$ nút tối đa ở mức i.

Bổ đề 1) đã được chứng minh.

2) Ta biết rằng chiều cao của cây là số mức lớn nhất có trên cây.

Theo 1) ta suy ra số nút tối đa có trên cây nhị phân với chiều cao h là:

$$2^0 + 2^1 + 2^2 + ... + 2^{h-1} = 2^h - 1$$

* Từ kết quả này có thể suy ra:

Nếu cây nhị phân hoàn chỉnh có n nút thì chiều cao của nó là:

$$h = \lceil \log_2(n + 1) \rceil$$

(Ta quy ước $\lceil x \rceil$ là số nguyên trên của x.

$\lfloor x \rfloor$ là số nguyên dưới của x.

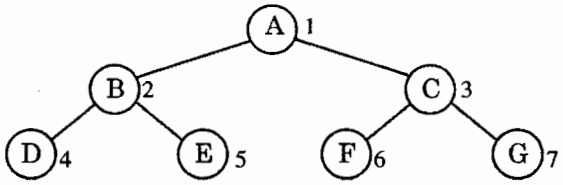
Nghĩa là: $\lfloor x \rfloor \leq x \leq \lceil x \rceil$

6.2.2 Biểu diễn cây nhị phân

6.2.2.1 Lưu trữ kế tiếp

Nếu có một cây nhị phân đầy đủ, ta có thể dễ dàng đánh số cho các nút trên cây đó theo thứ tự lần lượt từ mức 1 trở lên, hết mức này đến mức khác và từ trái sang phải đối với các nút ở mỗi mức.

Ví dụ: Với hình 6.7, cây f) có thể đánh số như sau:



Hình 6.8

Ta thấy ngay một qui luật quan hệ như sau:

Con của nút thứ i là các nút thứ 2i và 2i + 1 hoặc

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

Nếu như vậy thì ta có thể lưu trữ cây nhị phân đầy đủ bằng một vectơ V, theo nguyên tắc: nút thứ i của cây được lưu trữ ở V[i]. Đó chính là cách lưu trữ kế tiếp đối với cây nhị phân. Với cách lưu trữ này nếu biết được địa chỉ nút cha sẽ tính được địa chỉ nút con và ngược lại.

Như với cây đầy đủ nêu trên thì hình ảnh lưu trữ sẽ như sau:

A	B	C	D	E	F	G
V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]

Tất nhiên với cây nhị phân hoàn chỉnh, mà các nút ở mức cuối đều dạt về phía trái (để việc đánh số các nút này được liên tục) thì cách lưu trữ này vẫn thích hợp. Còn với cây nhị phân dạng khác thì cách lưu trữ này có thể sẽ gây ra lãng phí do có nhiều phần tử nhớ bị bỏ trống (ứng với cây con rỗng). Chẳng hạn đối với cây lệch trái ở hình 6.7a thì phải lưu trữ bằng một vectơ gồm 31 phần tử mà chỉ có 5 phần tử khác rỗng; hình ảnh miền nhớ lưu trữ nó như sau:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...	31
A	B	∅	C	∅	∅	∅	D	∅	∅	∅	∅	∅	∅	∅	E	∅	∅	...	∅

(∅: chỉ chỗ trống).

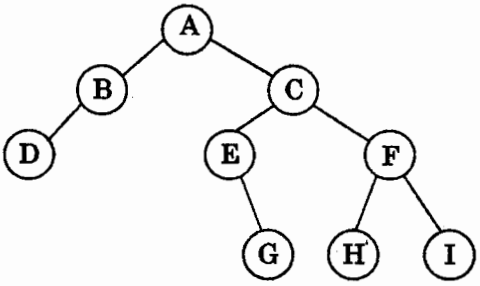
Ngoài ra nếu cây luôn biến động nghĩa là có phép bổ sung, loại bỏ các nút thường xuyên tác động, thì cách lưu trữ này tất không tránh được các nhược điểm như đã nêu.

Cách lưu trữ móc nối sau đây vừa khắc phục được nhược điểm này, vừa phản ánh được dạng tự nhiên của cây.

6.2.2.2 Lưu trữ móc nối

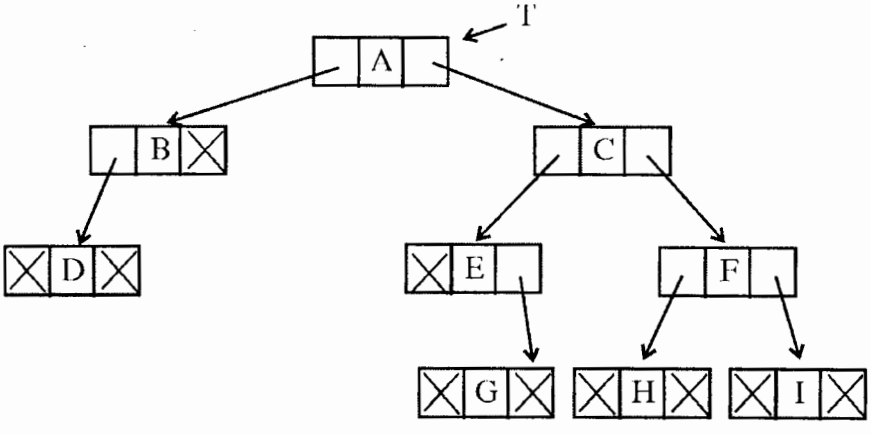
Trong cách lưu trữ này, mỗi nút ứng với một phần tử nhớ có qui cách như sau:

- Trường INFO ứng với thông tin (dữ liệu) của nút.
- Trường LPTR ứng với con trỏ, trỏ tới cây con trái của nút đó.
- Trường RPTR ứng với con trỏ, trỏ tới cây con phải của nút đó.



Hình 6.9

Ví dụ: Cây nhị phân hình 6.9 có dạng lưu trữ móc nối như ở hình 6.10.



Hình 6.10

Để có thể truy nhập vào các nút trên cây cần có một con trỏ T, trỏ tới nút gốc của cây đó.

Người ta quy ước: nếu cây nhị phân rỗng thì T = null.

Với cách biểu diễn này từ nút cha có thể truy nhập trực tiếp vào nút con, nhưng ngược lại thì không làm được.

6.2.3 Phép duyệt cây nhị phân (Traversing binary tree)

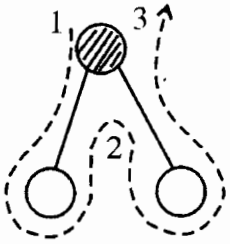
Thông thường ta hay phải thực hiện các phép xử lý trên mỗi nút của cây theo một thứ tự nào đó.

Ví dụ: Đối với cây biểu diễn biểu thức số học nêu ở hình 6.2., để xác định giá trị của biểu thức ta sẽ phải xử lý ở mỗi nút trên cây như sau: nếu là nút biểu diễn toán hạng ta sẽ xác định giá trị của toán hạng đó, nếu là nút biểu diễn dấu phép toán ta sẽ áp đặt phép toán đó lên giá trị của các cây con của nút ấy. Nhưng ta không thể thực hiện phép toán này, nếu như các cây con chưa được xử lý. Từ đó ta thấy ngay: thứ tự xử lý nút trên cây là rất quan trọng.

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

Nếu hình dung một nút với hai con, ta thấy khi đi qua nút ấy và các con nó theo đường mũi tên như hình 6.11, ta sẽ gặp nút ấy tới ba lần.

Từ đấy cũng hình thành ba phép duyệt khác nhau đối với cây nhị phân.



Hình 6.11

Các phép đó được định nghĩa một cách đệ qui như sau:

a) Duyệt theo thứ tự trước (preorder traversal)

1) Nếu cây rỗng thì không làm gì.

2) Nếu cây không rỗng thì:

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

b) Duyệt theo thứ tự giữa (inorder traversal)

1) Nếu cây rỗng thì không làm gì.

2) Nếu cây không rỗng thì:

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

c) Duyệt theo thứ tự sau (postorder traversal)

1) Nếu cây rỗng thì không làm gì.

2) Nếu cây không rỗng thì:

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

Với cây nhị phân ở hình 6.9, dãy các tên ứng với nút được thăm trong các phép duyệt sẽ là:

a) Theo thứ tự trước:

A B D C E G F H I

b) Theo thứ tự giữa:

D B A E G C H F I

c) Theo thứ tự sau:

D B G E H I F C A

Với cây nhị phân ở hình 6.2, ta lại có:

a) + + x * y - z t / u v

b) x + y * z - t + u / v

c) x y z t - * + u v / +

Để ý ta thấy:

a) chính là dạng biểu thức tiền tố (prefix)

c) là dạng hậu tố (postfix)

còn b) là dạng trung tố (infix), nhưng thiếu dấu ngoặc.

Như vậy hai phép duyệt a) và c) nêu trên đối với cây nhị phân biểu diễn biểu thức số học sẽ cho ta các dạng ký pháp Ba Lan của biểu thức số học.

* Nếu viết dưới dạng thủ tục đệ qui thì các giải thuật của các phép duyệt cây nhị phân sẽ như sau:

Procedure RPREORDER (T)

{Ở đây T là con trỏ, trỏ tới gốc cây đã cho có dạng lưu trữ móc nối. Mỗi nút trên cây T có qui cách như đã nêu ở 6.2.2b). Phép thăm ở đây được đặc trưng bởi việc in trường INFO của nút}

1. **if** T ≠ null **then begin**
   write(INFO(T));
   call RPREORDER(LPTR(T));
   call RPREORDER(RPTR(T));
   end;
2. **return**

Procedure RPOSTORDER (T)

1. **if** T ≠ null **then begin**
   call RPOSTORDER (LPTR(T));
   call RPOSTORDER (RPTR(T));
   write (INFO(T));
   end;
2. **return**

(Giải thuật RINORDER(T) bạn hãy tự làm xem như bài tập)

Ta thấy viết các thủ tục đệ qui này khá dễ dàng vì nó phản ánh đúng dạng các định nghĩa đã nêu của các phép duyệt bằng các câu lệnh tương ứng.

Còn nếu muốn thể hiện phép duyệt dưới dạng giải thuật không đệ qui thì có thể sử dụng stack, để nạp vào đó địa chỉ các nút cần thiết khi "đi xuống" theo một nhánh nào đó và lấy các địa chỉ ra khỏi stack, để xác định đường "đi lên", trong khi thực hiện phép duyệt. Sau đây là giải thuật cụ thể.

Procedure PREORDER(T)

{T là con trỏ trỏ tới gốc cây đã cho, S là một stack (tổ chức theo kiểu kế tiếp) với TOP trỏ tới đỉnh. Biến trỏ P được dùng để trỏ tới nút hiện đang được xem xét. Trong phép duyệt này khi thăm "gốc" xong thì *địa chỉ gốc cây con phải* của nút vừa được thăm sẽ được lưu trữ vào stack trước khi "đi xuống" cây con trái}

1. {Khởi đầu}
   if T = null **then begin**
   write ('cây rỗng');
   return

```
            end;
        else begin
        TOP := 0
        call PUSH(S, TOP, T)
            end;
```

{PUSH là thủ tục nạp một phần tử vào stack như đã nêu ở chương 5).

2. {Thực hiện duyệt}

```
    while TOP > 0 do begin
        call POP(S, TOP, P); {POP là thủ tục lấy một phần tử
                              từ stack ra}
```

3. {Thăm nút rồi xuống con trái}

```
    while P ≠ null do begin
            write (INFO(P)); {Thăm P}
            if RPTR(P) ≠ null then
            call PUSH(S, TOP, RPTR(P));
            {lưu trữ địa chỉ gốc cây con phải}
            P := LPTR(P); {xuống con trái}
                end
                  end
```

4. **return**

Procedure POSTORDER (T)

{Trong phép duyệt này nút "gốc" chỉ được thăm sau khi đã duyệt xong hai con của nó. Như vậy chỉ khi từ con phải đi lên gặp gốc mới được thăm, chứ không phải ở lần gặp khi từ con trái đi lên. Do đó để phân biệt người ta đưa thêm dấu âm vào địa chỉ (địa chỉ âm) để đánh dấu cho lần đi lên từ con phải}

1. {Khởi đầu}

```
    if T = null then begin
            write ('cây rỗng);
            return
                end;
            else begin
            TOP := 0;
            P := T;
                  end;
```

```
2. {Thực hiện duyệt}
   while true do begin
3. {lưu trữ địa chỉ "gốc" và xuống con trái}
   while P ≠ null do begin
         call PUSH(S,TOP,P);
         P:= LPTR(P)
               end;
4. {thăm nút mà con trái và con phải đã duyệt}
   while S[TOP] < 0 do begin
         call POP(S, TOP, P);
         write(INFO(P));
               if TOP = 0 then return
                     end;
5. {xuống con phải và đánh dấu}
   P := RPTR(S[TOP]); S[TOP] := - S[TOP]
         end;
6. return
```

6.2.4 Cây nhị phân nối vòng (Threaded binary tree)

Nếu để ý đến dạng biểu diễn móc nối của cây nhị phân, ta thấy có khá nhiều mối nối không. Có thể chứng minh được rằng trên cây nhị phân biểu diễn móc nối, có n nút thì có tới n+1 mối nối không (người đọc tự chứng minh).

Vì vậy, một vấn đề đặt ra là làm thế nào để tận dụng hơn nữa các trường mối nối này cho đỡ "lãng phí". Một trong những cách tận dụng như vậy đã được A.J. Perlis và C. Thomtor nêu ra:

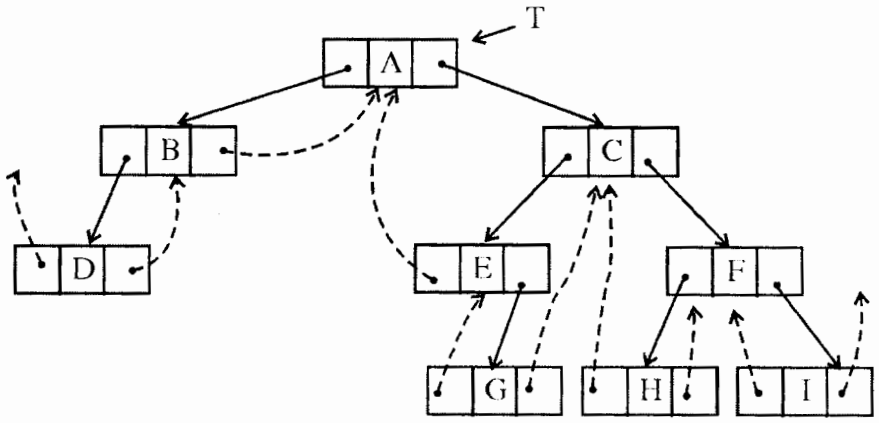
"Thay mối nối không bằng mối nối trỏ đến một nút quy định để tạo điều kiện thuận lợi cho phép duyệt cây".

Loại mối nối này ta gọi là *mối nối vòng* và dạng biểu diễn cây nhị phân như thế gọi là *cây nhị phân nối vòng.*

Tất nhiên phải đặt ngay câu hỏi: "mối nối vòng này trỏ tới đâu?".

Để nhằm mục đích giúp cho phép duyệt theo thứ tự giữa được thuận lợi, Perlis đưa ra quy ước: Đối với một nút P nào đó trên cây, nếu LPTR(P) = null thì thay LPTR(P) bằng ^{+}P, nếu RPTR(P) = null thì thay RPTR(P) bằng P^{+}, với ^{+}P chỉ con trỏ tới nút đứng trước P trong thứ tự giữa, còn P^{+} chỉ con trỏ trỏ tới nút sau P trong thứ tự giữa.

Với cây như hình 6.10 thì dạng biểu diễn nối vòng sẽ như sau:



Hình 6.12

Ở đây mối nối vòng được biểu diễn bởi — — —>

* Rõ ràng là trong máy thì không thể phân biệt được mối nối thẳng và mối nối vòng giống như trên hình vẽ. Vì vậy trong qui cách của một nút phải thêm vào hai trường bit: LBIT và RBIT để đánh dấu hai loại mối nối đó. Nghĩa là qui cách một nút sẽ có dạng

LBIT	LPTR	INFO	RPTR	RBIT

Nếu LPTR(P) ≠ null thì LBIT(P) = 0

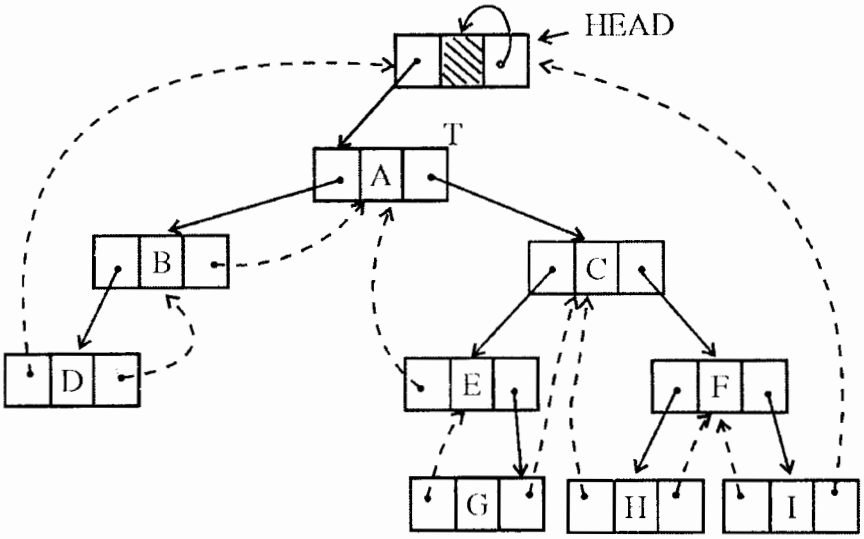
LPTR(P) = null thì LBIT(P) = 1

(ứng với mối nối vòng)

Đối với RBIT cũng tương tự.

Ta cũng thấy: với cây như hình 6.12 mối nối vòng trái của D (ta gọi là nút cực trái của cây T) và mối nối vòng phải của nút I (nút cực phải của cây T) còn chưa được xác định vì trong thứ tự giữa không có nút trước nút D và nút sau nút I. Tuy nhiên để cách biểu diễn được nhất quán đối với mọi nút, mà cũng không gây ra điều gì phức tạp, người ta qui ước đưa thêm vào nút "đầu cây". Trong cách biểu diễn này cây T được coi như con trái của nút "đầu cây" ấy và mối nối phải của nút đầu cây thì luôn trỏ tới chính nó.

Như vậy cây ở hình 6.12 sẽ có dạng như ở hình 6.13.



Hình 6.13

HEAD là con trỏ, trỏ tới nút đầu cây.

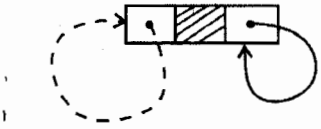
Trường hợp cây rỗng thì nút đầu cây có dạng như hình bên, nghĩa là:

RPTR (HEAD) = HEAD

RBIT (HEAD) = 0

LPTR(HEAD) = HEAD

LBIT (HEAD) = 1



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

Function INS (P)

{Cho P là con trỏ trỏ tới một nút trong cây nối vòng. Giải thuật này cho ta địa chỉ Q là con trỏ trỏ tới nút sau P trong thứ tự giữa}

1. {Nối phải là mối nối vòng}

   Q := RPTR(P);

   if RBIT(P) = 1 **then return** (Q);

2. {Tìm đến nút cực trái của con phải}

   while LBIT(Q) ≠ 1 **do** Q:= LPTR(Q);

3- **return** (Q)

Function IN(P)

{Giải thuật này chỉ khác giải thuật trước ở chỗ nó cho Q là con trỏ trỏ tới nút trước P trong thứ tự giữa}

1. Q:= LPTR(P);
   if LBIT(P) = 1 **then return** (Q)
2. **while** RBIT(Q) ≠ 1 **do** Q := RPTR(Q);
3. **return** (Q)

Phép duyệt cây theo thứ tự giữa bây giờ chỉ là phép gọi liên tục thủ tục INS(P) bắt đầu từ nút đầu cây, trỏ bởi HEAD

Procedure TINORDER(HEAD)

1. P := HEAD;
2. **while true do begin**
   P := INS(P);
   if P = HEAD **then return**
   else write (INFO(P))
   end

Từ các giải thuật trên ta có thể thấy các ưu điểm của cây nhị phân nối vòng. Tuy nhiên ta cũng thấy ta phải trả giá ở chỗ:

- Có hơi tốn không gian nhớ hơn (vì thêm các trường LBIT, RBIT).

- Các phép toán bổ sung hay loại bỏ một nút vào cây nối vòng có phức tạp hơn.

Sau đây là giải thuật thực hiện phép bổ sung một nút vào thành con trái của nút P cho trên cây nối vòng.

Procedure LEFT(P,X);

{Cho P trỏ tới một nút trên cây nối vòng, giải thuật này thực hiện bổ sung một nút mới vào bên trái nút P, thông tin liên quan tới nút này hiện đặt ở X}

1. {Tạo nút mới}
   call NEW (Q);
   INFO(Q) := X;
2. {Chỉnh lý lại các con trỏ ở P và Q}
   LPTR(Q) := LPTR(P); LBIT(Q) := LBIT(P);
   LPTR(P) := Q; LBIT(P) := 0;
   RPTR(Q) := P; RBIT(Q) := 1;

3. {Dựng lại mối nối vòng ở nút đứng trước Q nếu cần}

if LBIT(Q) = 0 **then begin**

W:= INP(Q) ;

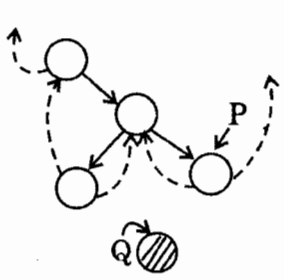
RPTR(W) := Q;

RBIT(W) := 1;

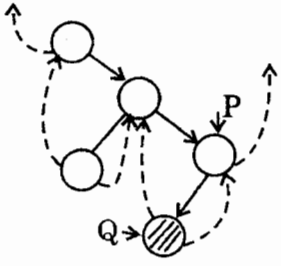
end

4. **return**

Hình 6.14 minh hoạ phép bổ sung thực hiện bởi LEFT

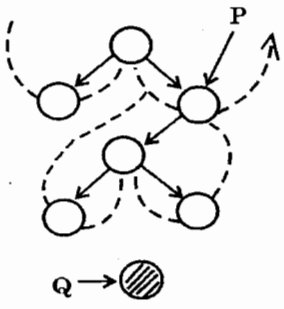


trước

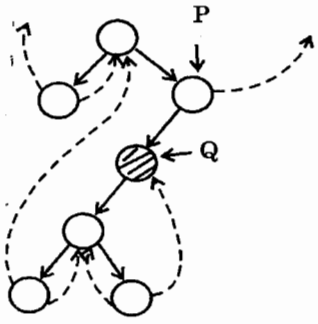


sau

a) Trường hợp P không có con trái



trước



sau

b) Trường hợp P đã có con trái.

Hình 6.14

Giải thuật thực hiện bổ sung một nút mới vào thành con phải một nút đã cho trên cây nhị phân nối vòng cũng tương tự (người đọc hãy tự làm).

Các giải thuật này cũng cho phép ta khởi tạo và phát triển cây nhị phân nối vòng.

6.3 Cây tổng quát

6.3.1 Biểu diễn cây tổng quát

Một ý nghĩ đầu tiên có thể đến với chúng ta là đối với cây tổng quát cấp m nào đó có thể sử dụng cách biểu diễn móc nối tương tự như đối với cây nhị phân. Như vậy ứng với mỗi nút tất phải dành ra m trường mối nối để trỏ tới các con của nút đó và như vậy số "mối nối không" sẽ rất nhiều: Nếu cây có n nút sẽ có tới n(m-1) + 1 "mối nối không" trong số m.n mối nối (hãy tự chứng minh).

Còn nếu tuỳ theo số con của từng nút mà định ra mối nối nghĩa là dùng nút có kích thước biến đổi thì sự tiết kiệm không gian nhớ này sẽ phải trả giá bằng những phức tạp của quá trình xử lý trên cây.

Một trong các phương pháp khác khá hiện thực là biểu diễn cây tổng quát bằng cây nhị phân. Như vậy quan hệ giữa các nút trên cây tổng quát chỉ được thể hiện qua hai đặc điểm thôi.

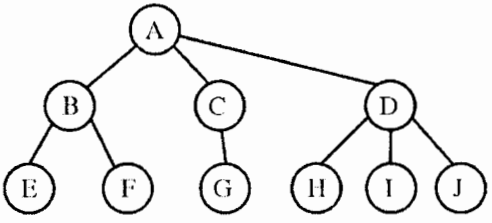
Điều này rõ ràng có thể đáp ứng được nếu như ta để ý rằng: Với bất kỳ một nút nào trên cây tổng quát, nếu có thì chỉ có:

- Một nút con cực trái (con cả)
- Một nút em kế cận phải

(Giả sử ta đánh số thứ tự cho các con từ trái qua phải). Lúc đó cây nhị phân biểu diễn cây tổng quát theo hai quan hệ này được gọi là cây *nhị phân tương đương* (equivalent binary tree).

Ví dụ: Xét cây ở hình 6.15. Với nút B: Con cực trái là E, em kế cận phải là C.

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



Hình 6.15

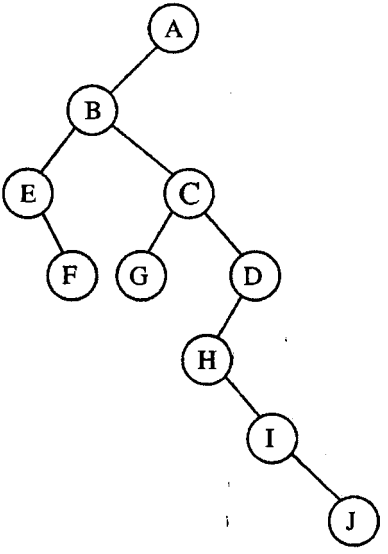
Như vậy nếu mỗi nút có qui cách

CHILD	INFO	SIBLING

CHILD: Con trỏ trỏ tới nút con cực trái

SIBLING: Con trỏ, trỏ tới nút em kế cận phải

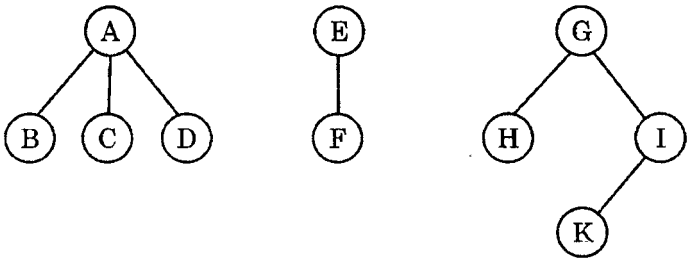
thì có thể biểu diễn cây tổng quát nêu trên bằng cây nhị phân tương đương như sau:



Hình 6.16

Ta thấy ngay: Nối phải của gốc cây bao giờ cũng là "mối nối không" vì gốc không có em kế cận phải. Nhưng nếu xét cả một rừng thì tình trạng trên lại không xuất hiện. Vì vậy: có thể biểu diễn rừng bằng một cây nhị phân tương đương (trường hợp một cây thì coi như rừng đặc biệt).

Ví dụ: Với rừng ở hình 6.17 thì cây nhị phân tương đương biểu diễn nó như ở hình 6.18.



Hình 6.17

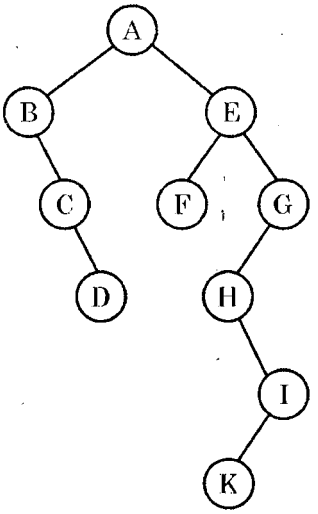
Có thể định nghĩa phép biến đổi tổng quát đối với rừng như sau:

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

$B(T_1, T_2, ..., T_n)$ sẽ là cây:

1) Rỗng, nếu n = 0

2) Có gốc là gốc của T_1, có cây con trái là $B(T_{11}, T_{12}, ..., T_{1m})$ với $T_{11}, T_{12}, ..., T_m$ là cây con gốc T_1, có cây con phải là $B(T_2, ..., T_n)$.



Hình 6.18

6.3.2 Phép duyệt cây tổng quát

Phép duyệt cây tổng quát cũng được đặt ra tương tự như đối với cây nhị phân. Tuy nhiên, có một điều cần phải xem xét thêm, khi định nghĩa phép duyệt, đó là:

1) Sự nhất quán về thứ tự các nút được thăm giữa phép duyệt cây ấy (theo định nghĩa của phép duyệt cây tổng quát) và phép duyệt cây nhị phân tương đương của nó (theo định nghĩa của phép duyệt cây nhị phân).

2) Sự nhất quán giữa định nghĩa phép duyệt cây tổng quát với định nghĩa phép duyệt cây nhị phân. Vì cây nhị phân có thể được coi là cây, để duyệt được theo phép duyệt cây tổng quát.

Đối với cây nhị phân ta đã xây dựng 3 phép duyệt dựa trên cách thăm nút khi gặp nút đó lần đầu, lần thứ hai hay lần cuối ở hình 6.11.

Nếu cũng phỏng theo cách như vậy, ta sẽ xây dựng được một định nghĩa của phép duyệt cây tổng quát T như sau:

* Duyệt theo thứ tự trước

a) Nếu T rỗng thì không làm gì

b) Nếu T không rỗng thì:

1) Thăm gốc của T

2) Duyệt các cây con thứ nhất T_1 của gốc T theo thứ tự trước.

3) Duyệt các cây con còn lại T_2, T_3, ..., T_k của gốc T theo thứ tự trước.

* Duyệt theo thứ tự giữa

a) Nếu T rỗng thì không làm gì.

b) Nếu T không rỗng thì:

1) Duyệt cây con thứ nhất T_1 của gốc của T theo thứ tự giữa.

2) Thăm gốc của T

3) Duyệt các cây con còn lại T_2, T_3, ..., T_k của gốc T theo thứ tự giữa.

* Duyệt theo thứ tự sau

a) Nếu T rỗng thì không làm gì

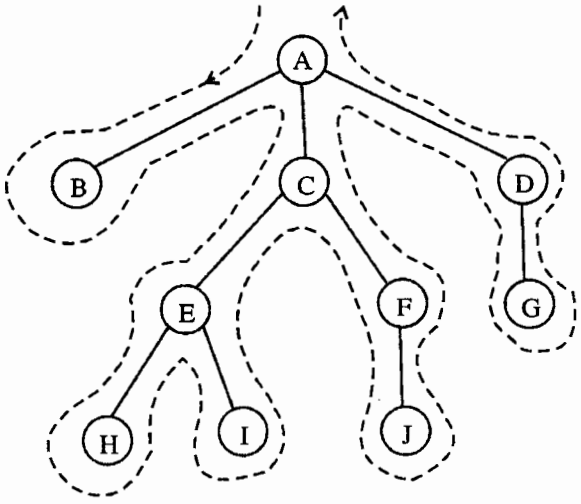
b) Nếu T không rỗng thì:

1) Duyệt cây con thứ nhất T_1 của gốc T theo thứ tự sau.

2) Duyệt cây con còn lại T_2, T_3, ..., T_k của gốc của T theo thứ tự sau.

3) Thăm gốc của T

Ví dụ với cây:



Hình 6.19

thì dãy tên các nút được thăm sẽ là:

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa: B A H E I C J F G D

Thứ tự sau: B H I E J F C G D A

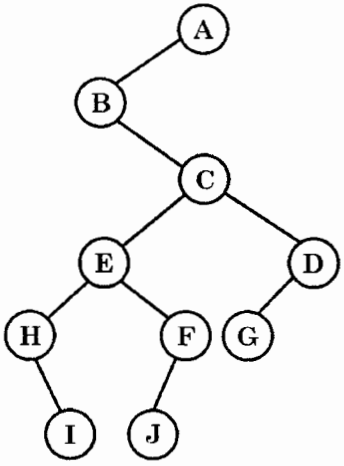
Ta cũng thấy các phép duyệt trên đã dựa trên việc thăm một nút trên cây ở lần gặp đầu, lần gặp thứ hai hay lần gặp cuối trên đường đi vẽ bằng nét chấm ở hình 6.19.

Cũng có thể thấy ngay với định nghĩa nêu trên, điểm xem xét 2) đã được đảm bảo; nghĩa là với cây nhị phân dù áp đặt vào nó phép duyệt của cây nhị phân hay phép duyệt của cây tổng quát thì dãy tên các nút được thăm cũng hoàn toàn như nhau.

Còn với điểm 1) thì sao?

Ta sẽ thấy câu trả lời qua nội dung xem xét đối với cây ở hình 6.19.

Ta có cây nhị phân tương đương với nó trên hình 6.20



Hình 6.20

Dãy tên các nút được thăm khi duyệt nó theo phép duyệt cây nhị phân:

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa: B H I E J F C G D A

Thứ tự sau: I H J F E G D C B A

Với thứ tự trước phép duyệt cây tổng quát và phép duyệt cây nhị phân tương đương của nó đều cho một dãy tên như nhau. Phép duyệt cây tổng quát theo thứ tự sau đã cho dãy tên giống hệt dãy cho bởi phép duyệt theo thứ tự giữa của cây nhị phân tương đương với nó. Còn phép duyệt cây tổng quát theo thứ tự giữa thì cho dãy tên không giống bất kỳ dãy nào đối với cây nhị phân tương đương!

Chính vì vậy đối với cây tổng quát, nếu định nghĩa phép duyệt như trên, người ta thường chỉ nêu hai phép: duyệt theo thứ tự trước và thứ tự sau.

Chú ý: Với rừng ta có thể duyệt theo cách như đã định nghĩa nêu trên, bằng cách coi như có một nút gốc giả mà các con của nó chính là các cây của rừng ấy.

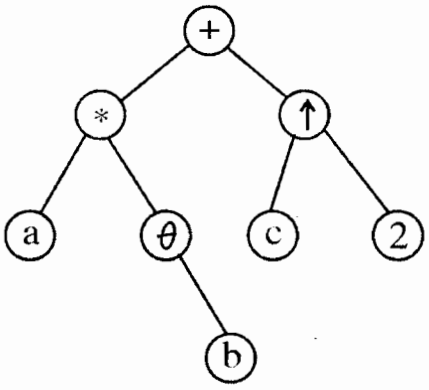
6.4 Áp dụng

6.4.1 Cây biểu diễn biểu thức

Như ta đã biết: Một biểu thức số học với phép toán hai ngôi như +, -, *, /, ↑ (luỹ thừa), có thể được biểu diễn rất tự.nhiên bởi một cây nhị phân.

Nếu ta đưa phép toán một ngôi vào, chẳng hạn phép thêm dấu âm (mà ta ký hiệu là θ để dễ phân biệt) thì vẫn có thể biểu diễn biểu thức có chứa phép đó bằng một cây nhị phân được, nếu như ta ấn định thêm một qui ước, ví dụ toán hạng của θ luôn là con phải của nó.

Như vậy: Biểu thức a* θ b + c ↑2 có thể được biểu diễn bởi cây như hình 6.21



Hình 6.21

Để cho đơn giản dưới đây ta chỉ xét tới biểu thức mà dạng biểu diễn của nó là cây nhị phân, như ở hình 6.21.

* Trước hết ta xét tới phép định giá trị của một biểu thức số.

Ta sẽ tạo cây nhị phân biểu diễn biểu thức đó với qui cách một nút như hình 6.22.

LPTR	TYPE	RPTR

Hình 6.22

Ngoài hai trường hợp LPTR và RPTR giống như qui cách đã nêu trước đây, thì trường thứ ba INFO được thay bởi trường TYPE.

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

Nếu là nút lá thì TYPE có giá trị 0 để chỉ biến hoặc hằng tương ứng với nút đó. Trong trường hợp này RPTR trỏ tới địa chỉ trong bảng ký hiệu (Symbol table) của biến hoặc hằng đó.

Nhớ rằng với qui cách như trên, nút trên cây sẽ lưu trữ loại của phép toán chứ không phải dấu phép toán đó. Điều này sẽ làm cho quá trình xử lý cây đơn giản đi. Còn bảng ký hiệu thì được tổ chức để chứa tên của biến (ở trường SYMBOL) hoặc hằng, và giá trị của chúng (ở trường VALUE).

Như với cây ở hình 6.21 thì biểu diễn cụ thể sẽ như hình 6.23.

Sau đây là giải thuật định giá của một biểu thức biểu diễn bởi một cây nhị phân. Giải thuật này được viết dưới dạng một hàm đệ qui.

Function EVAL(E)

{Cho E là con trỏ, trỏ tới gốc cây biểu diễn một biểu thức theo cách như đã nêu. Hàm này cho ta giá trị của biểu thức đó. F ở đây là một biến trỏ phụ}

1. **Case**

TYPE(E)= 0: F: = RPTR(E); **return** (VALUE(F));

TYPE(E)=1:**return** (EVAL(LPTR(E)) + EVAL(RPTR(E)));

TYPE(E)= 2 : **return** (EVAL(LPTR(E)) - EVAL(RPTR(E)));

TYPE(E)= 3 : **return** (EVAL(LPTR(E)) * EVAL(RPTR(E)));

TYPE(E)= 4 : **return** (EVAL(LPTR(E)) / EVAL(RPTR(E)));

TYPE(E)= 5 : **return** (EVAL(LPTR(E)) ↑ EVAL(RPTR(E)));

TYPE(E)= 6 : **return** (θ EVAL(RPTR(E)));

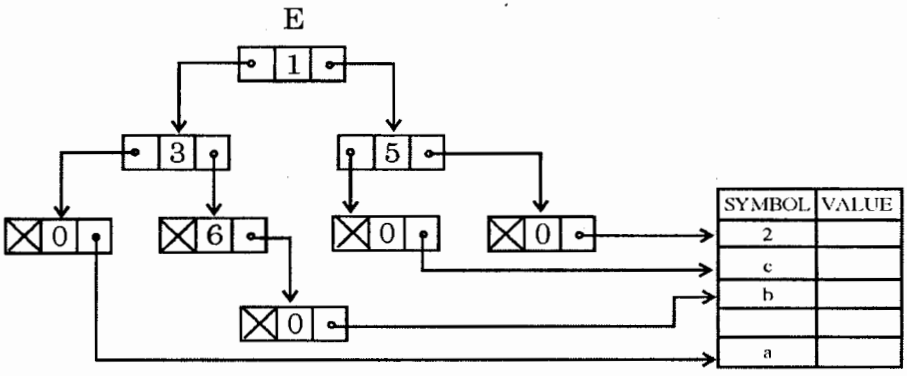
else: return (00) {Giá trị 00 chỉ trường hợp biểu thức sai}

end case

Chú ý:

Đối với một thủ tục hàm H nào đó, để cho tiện người ta thay câu lệnh: H := a; **return** bằng câu lệnh **return**(a);

Như với hàm trên, câu lệnh **return**(VALUE(F)); là tương đương với: EVAL := VALUE(F); **return**



Hình 6.23

* Với cách tổ chức cây biểu thức theo qui cách như trên, bài toán xác định xem hai biểu thức có tương tự (similar) hay không (nghĩa là giống như nhau hoặc chỉ có thể khác nhau ở chỗ là thứ tự toán hạng của phép cộng hoặc phép nhân đổi chỗ cho nhau) có thể giải quyết bởi giải thuật sau:

Function SIMILAR(A, B)

{Cho hai cây biểu thức mà cây nhị phân biểu diễn nó được trỏ bởi A và B. Hàm lôgic này cho giá trị **true** nếu A và B tương tự nhau, ngược lại nó sẽ cho giá trị **false**}

1. {Kiểm tra loại của gốc cây}

   if TYPE(A) ≠ TYPE(B) **then return** (false)

2. {Kiểm tra tính tương tự}

   case

   TYPE (A) = 0 : if VALUE (RPTR(A)) ≠ VALUE (RPTR(B))

   then return (false)

   else return (true);

   TYPE(A) = 1 or TYPE(A) = 3: {Trường hợp phép + và *}

   return (SIMILAR (LPTR(A), RPTR(B)) **and** SIMILAR

   (RPTR(A),LPTR(B)))

   or SIMILAR (LPTR(A), LPTR(B)) **and** SIMILAR (RPTR(A,RPTR(B));

   TYPE(A) = 2 or TYPE(A) = 4 or TYPE(A) = 5:

   return (SIMILAR(LPTR(A),LPTR(B)) **and** SIMILAR(RPTR(A),RPTR(B));

   TYPE(A) = 6: **return** (SIMILAR (RPTR(A), RPTR(B)))

end case

* Cuối cùng ta xét tới phép tính đạo hàm của một biểu thức đại số. Để cho đơn giản, ta gói gọn bài toán trong những qui định như sau:

Về qui tắc đạo hàm, giả sử ta cho các qui tắc sau:

$D(x) = 1$ $\quad$ $D(a) = 0$ (với a là hằng hoặc biến khác x)

$DLN(u) = D(u)/u$

$D(-u) = - D(u)$

$D(u + v) = D(u) + D(v)$

$D(u - v) = D(u) - D(v)$

$D(u * v) = u* D(v) + v * D(u)$

$D(u/v) = D(u)/v - (u * D(v))/v^2$

Biểu thức của chúng ta, giả sử cũng chỉ chứa các phép: LN (logarit tự nhiên), θ (trừ một ngôi), +, -, *, /

Ở đây ta thấy: khi cho một biểu thức đại số biểu diễn dưới dạng cây thì bài toán tính đạo hàm của biểu thức đó là bài toán tạo nên một cây biểu diễn biểu thức đạo hàm ấy, chứ không phải là đưa ra giá trị số học hoặc lôgic như hai bài toán nêu trên. Vì vậy cần thể hiện được dấu phép toán cũng như ký hiệu của toán hạng ở ngay các nút trên cây ấy. Cho nên qui cách của mỗi nút bây giờ có khác trước, nó gồm 4 trường, như sau:

LPTR	SYM	TYPE	RPTR

nghĩa là thêm vào trường SYM để ghi ký hiệu biểu diễn phần tử tương ứng với nút đó. Còn trường TYPE ghi mã chỉ loại của phần tử đó, theo bảng qui định sau đây:

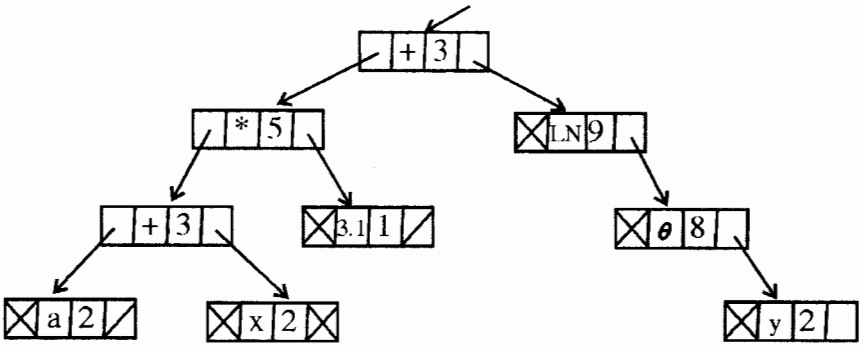
Loại	Ý nghĩa	Loại	Ý nghĩa
1	hằng	6	/ (chia)
2	biến	7	↑ (luỹ thừa)
3	+ (cộng)	8	θ (trừ một ngôi)
4	- (trừ)	9	LN (logarit tự nhiên
5	* (nhân)		

Tuy ta qui ước trong biểu thức đại số đưa vào không có phép luỹ thừa, nhưng trong biểu thức đạo hàm vẫn có thể có, vì vậy ta vẫn nêu ở đây.

Với qui cách nêu trên, biểu thức.

$$(a + x) * 3.1 + (LN\ \theta\ y)$$

sẽ được biểu diễn như sau:



Hình 6.24

Để tạo nên cây biểu diễn biểu thức của đạo hàm (ta gọi tắt là cây đạo hàm) ta thấy, trước hết phải giải quyết:

- Tạo một nút thể hiện một phần tử của biểu thức (nút biểu diễn phép toán, hay hằng, hay biến).

- Gắn một cây con vào thành con phải của một nút (ứng với phép toán trừ một ngôi θ).

- Gắn hai cây con vào thành cây con trái và con phải của một nút (ứng với các phép toán hai ngôi).

Bây giờ ta xét tới các giải thuật tương ứng với các phép này.

Function MAKE-NODE (VAL)

{Cho VAL là một biến dòng (xâu) có giá trị biểu diễn một phần tử của biểu thức (một biến, một hằng hoặc một dấu phép toán), hàm này tạo nên một nút có LPTR và RPTR bằng null, trường TYPE của nút đó được xác định từ VAL. Địa chỉ của nút này được cho bởi biến trỏ X. Ở đây OPS là một xâu ký tự ứng với các phép toán, DIGIT là xâu ký tự ứng với ký tự của hằng}

1. {khởi tạo}
   OPS := ' + - * / ↑ LN';
   DIGIT := ' - , 0 1 2 3 4 5 6 7 8 9'
2. {Tạo nút}
   call New(X);
   LPTR(X) := RPTR(X) := null; SYM(X) := VAL;
   TYPE (X) := 0;
3. {Xác định loại của nút}
   if VAL ≠ " " **then**

```
begin TYPE(X) := INDEX(OPS, VAL);
    if TYPE(X) ≠ 0 then TYPE(X) := TYPE(X) + 2
    else if INDEX(DIGITS, SUB(VAL,1,1)) ≠ 0
            then TYPE(X) := 1 {VAL là hằng}
            else TYPE(X) := 2 {VAL là biến}
end
```

4. **return** (X)

Ở đây ta đã dùng tới hai hàm mẫu INDEX và SUB:

INDEX(OPS,VAL) thực hiện so sánh giá trị của VAL (cũng là một xâu ký tự) với xâu OPS và cho chỉ số (số thứ tự) của ký tự thuộc OPS trùng với ký tự cực trái của VAL nếu có xâu con ký tự của OPS; trùng hoàn toàn với xâu ký tự của VAL. Trường hợp không trùng thì cho số 0. Ví dụ nếu VAL có giá trị là * thì INDEX(OPS,VAL) sẽ cho giá trị là 3, nếu VAL có giá trị là x thì cho giá trị 0.

SUB(VAL,1,1) cho ta xâu con của VAL có độ dài là 1, bắt đầu bằng ký tự thứ 1 của VAL. Như vậy INDEX(DIGITS, SUB(VAL 1,1)) thực hiện so sánh ký tự đầu tiên của VAL với xâu ký tự DIGITS để phát hiện hằng số.

Function CREATE1(N1, N2)

{Cho N1 là con trỏ, trỏ tới một nút đơn; N2 là con trỏ trỏ tới một cây con nhị phân. Hàm này nối cây trỏ bởi N2 vào thành cây con phải của N1 và cho ra N1}

1 - RPTR(N1) := N2;

2 - **return** (N1)

Function CREAT2 (N1, N2, N3)

{Cho N1 trỏ tới một nút đơn, N2 và N3 trỏ tới hai cây con nhị phân; Hàm này nối N2 và N3 vào thành con trái và con phải của N1 và cho ra N1}

1 - LPTR(N1) := N2;
    RPTR(N1) := N3;

2 - **return** (N1)

Trên cơ sở của các thủ tục trên, ta đi tới giải thuật đạo hàm biểu thức đại số như sau:

Function DIFFER (ROOT,VAR)

{Cho ROOT là con trỏ, trỏ tới gốc "cây biểu thức" và VAR là biến mà theo đó ta tính đạo hàm. Hàm này tạo nên "cây đạo hàm" của biểu thức và cho ra con trỏ trỏ tới gốc cây đó.

OPER1 và OPER2 là hai con trỏ, trỏ tới đạo hàm của toán hạng thứ nhất và toán hạng thứ hai (nếu có).

OP2 là con trỏ, trỏ tới bản sao của toán hạng thứ hai. Nếu phép toán chỉ có một toán hạng thì qui ước toán hạng đó ứng với cây con phải của nó.

Ở đây COPY là hàm thực hiện một bản sao của một cây nhị phân (tạo một cây giống hệt cây này).

```
    TEMP1, TEMP2, TEMP3, TEMP4 là các biến con trỏ}
1 - {Kiểm tra xem biểu thức có rỗng không}
        if ROOT = NULL then return (NULL);
2 - {Nút đang xét có là hằng không?}
        if TYPE (ROOT) = 1 then return (MAKE-NODE('0'));
3- {Nút đang xét có là biến không?}
        if TYPE (ROOT) = 2 then
                if SYM(ROOT) = VAR
                then return (MAKE-NODE('1'))
                else return (MAKE-NODE('0'));
4- {Tạo toán hạng cho phép toán}
        OPER2 := DIFFER (RPTR(ROOT), VAR);
        if LPTR(ROOT) ≠ NULL then
                OPER1 := DIFFER (LPTR(ROOT), VAR);
    5- {Xử lý nút phép toán}
        case
    TYPE(ROOT) = 3 : {phép cộng}
            return(CREATE2(MAKE-NODE('+'),OPER1,OPER2));
    TYPE(ROOT) = 4 : {phép trừ}
            return(CREATE2(MAKE-NODE('-'),OPER1, OPER2));
    TYPE(ROOT) = 5 : {phép nhân}
TEMP1 := CREATE2 (MAKE-NODE ('*'), OPER1, COPY(RPTR(ROOT)));
TEMP2 := CREATE2 (MAKE-NODE ('*'), OPER1, COPY(RPTR(ROOT)));
        return (CREATE 2 (MAKE-NODE ('+'), TEMP1, TEMP2))
    TYPE(ROOT) = 6: {phép chia}
                OP2 := COPY (RPTR(ROOT));
TEMP1 := CREATE 2 (MAKE-NODE ('/'), OPER1, OP2));
TEMP2 := CREATE2 (MAKE-NODE ('*'), COPY(LPTR (ROOT), OPER2);
    TEMP3 := CREATE2 (MAKE-NODE ('↑'), OP2, MAKE-NODE('2'));
```

TEMP4 := CREATE2 (MAKE-NODE ('/'), TEMP2, TEMP3));

return (CREATE2(MAKE-NODE('-'),TEMP1, TEMP4));

TYPE(ROOT) = 8: {phép trừ một ngôi}

return (CREATE1 (MAKE - NODE('θ'), OPER2));

TYPE(ROOT) = 9: { Phép lôga tự nhiên}

return (CREATE 2 (MAKE-NODE ('/') OPER2, COPY(RPTR(ROOT)));

else: write ('ERROR'); **return** (NULL);

{Trường hợp không đúng các phép toán đã qui ước thì báo "nhầm" và cho ra NULL}

end case

Trong giải thuật trên bước 5 thực hiện tạo cây đạo hàm theo đúng như qui tắc đạo hàm đã nêu ở trên. Nhớ rằng: đây vẫn là một giải thuật đệ qui.

6.4.2 Cây biểu diễn các tập

Ta xét việc sử dụng cây để biểu diễn tập hợp. Để đơn giản, ta giả thiết rằng các phần tử của tập là các số tự nhiên 1, 2, 3,..., n. Trong thực tế các số này chính là các chỉ số tương ứng với một bảng ký hiệu, ở đó lưu trữ các tên thường dùng của các phần tử ấy. Ta cũng giả sử các tập xét ở đây đều là các tập không giao nhau, nghĩa là tập S_i và S_j với $i \neq j$ không có chứa một phần tử chung nào. Ví dụ có 10 phần tử, phân làm ba tập thì có thể

$$S_1 = \{1, 7, 8, 9\}, S_2 = \{3, 5\}, S_3 = \{2, 4, 6, 10\}$$

Các phép toán mà ta muốn thực hiện trên các tập này là:

1) Phép hợp (Union)

Nếu S_i, S_j là hai tập không giao nhau thì hợp $S_i \cup S_j$ = {mọi phần tử x sao cho $x \in S_i$ hoặc $x \in S_j$}

Như ví dụ trên:

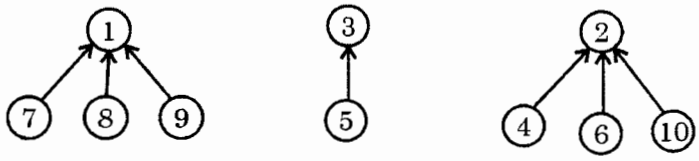
$$S_2 \cup S_3 = \{3, 5, 2, 4, 6, 10\}$$

Vì ta giả sử mọi tập đều không giao nhau nên sau phép hợp $S_i \cup S_j$ ta phải coi như S_i và S_j không còn tồn tại độc lập nữa nghĩa là chúng đã được thay thế bởi $S_i \cup S_j$ trong tập tất cả các tập.

2) Phép tìm (Find)

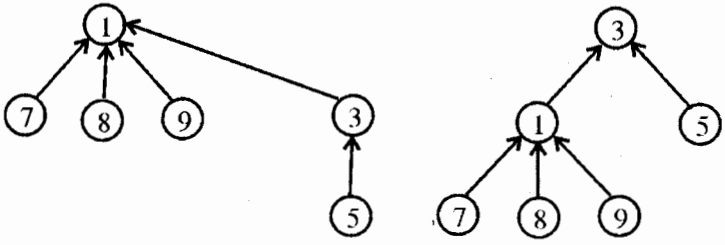
Tìm tập chứa phần tử i, ví dụ: Tập S_3 chứa phần tử 4, tập S_1 chứa phần tử 7.

Ta sẽ biểu diễn các tập ở đây dưới dạng cây. Nhưng có một điều khác với trước là: Trừ nút gốc ra còn ở mọi nút đều có con trỏ, trỏ tới nút cha của nó. Ví dụ: với các tập S_1, S_2, S_3 nếu chọn phần tử đầu tiên của tập là gốc thì cây biểu diễn chúng sẽ có dạng như ở hình 6.25



Hình 6.25

Thuận lợi của cách biểu diễn này sẽ thấy rõ khi thể hiện giải thuật UNION và FIND. Trước hết để tạo nên cây hợp của S_i và S_j ta chỉ cần cho một trong hai cây trở thành con của cây kia. Chẳng hạn hợp của S_1 và S_2 có thể biểu diễn như ở hình 6.26.



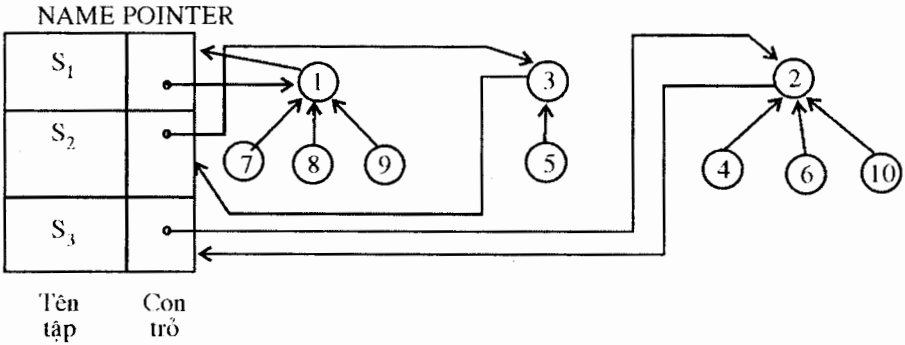
Hình 6.26

Như vậy cần phải tạo con trỏ trỏ từ gốc cây này đến gốc cây kia, nghĩa là phải nắm được gốc cây. Điều này có thể thực hiện được dễ dàng nếu cùng với tên tập hợp, ta có con trỏ trỏ tới gốc cây biểu diễn tập hợp ấy. Nếu, thêm vào đó, ở gốc cây lại có con trỏ trỏ về nơi giữ tên của tập ấy thì khi cần tìm xem một phần tử thuộc tập nào ta chỉ cần tìm đến gốc cây của phần tử ấy và dùng con trỏ này để xác định tên tập chứa nó.

Cụ thể hình dung cách tổ chức cây biểu diễn ba tập S_1, S_2, S_3 cụ thể hơn, như hình 6.27.

Để cho việc bàn luận về giải thuật UNION và FIND đơn giản hơn, ta sẽ bỏ qua tên tập và thay nó bằng tên gốc của cây biểu diễn tập đó; vì thực ra với cách tổ chức như ở hình 6.27 việc chuyển đổi từ tên gốc sang tên tập không có gì là khó khăn cả. Vì vậy ta chỉ nói tới cây có gốc i, cây có gốc j (với i, j là các số từ 1 tới n, như ta đã giả thiết). Do đó FIND (i) sẽ xác định

gốc của cây chứa phần tử i. UNION(i,j) sẽ nối hai gốc i và gốc j với nhau. Ngoài trường chứa các thông tin cần thiết thì mỗi nút trên cây chỉ có một trường nối PARENT chứa con trỏ trỏ tới nút cha nó. Nút gốc có trường PARENT bằng 0. Với qui ước như vậy thì giải thuật ứng với phép UNION và FIND khá đơn giản.



Hình 6.27

Procedure U(i;j) {Ta qui ước: gốc cây U(i;j) là j}

1) **PARENT** [i] := j;
2) **return**

Function F(i)

1) j := i;
2) **while** PARENT [j] >0 **do** j := PARENT [j];
3) **return** (j)

Tuy nhiên, việc thực hiện các giải thuật này không hẳn là đã thật tốt. Chẳng hạn, nếu ta bắt đầu với p phần tử mà mỗi phần tử đó nằm trong tập chỉ có chính nó, nghĩa là $S_i = \{i\}$, $1 \le i \le p$, thì tình huống ban đầu sẽ là một rừng với p nút và PARENT [i] = 0, $1 \le i \le p$. Bây giờ ta sẽ thực hiện dãy các phép xử lý sau:

Hình 6.28

U(1, 2), F(1), U(2,3), F(1), U(3,4), F(1)... U((n-1), n)

Dãy này sẽ dẫn tới một cây suy biến ở hình 6.28.

Vì thời gian thực hiện phép hợp là hằng số nên (n-1) phép hợp sẽ có chi phí thời gian O(n). Nhưng mỗi phép FIND thì lại phụ thuộc vào dãy các mối nối PARENT từ nút 1 tới gốc cây, mà theo dãy phép toán thực hiện thì có (n-2) lần thực hiện FIND, do đó chi phí thời gian tương ứng sẽ là $O(\sum_{i=1}^{n-2} i) = O(n^2)$ và đây chính là một trường hợp xấu đối với các giải thuật UNION - FIND.

Ta có thể làm tốt hơn nếu chú ý tránh việc gây ra trường hợp cây suy biến. Để đạt được yêu cầu này ta sẽ áp dụng ở đây qui tắc sau: "Nếu số các nút trên cây i nhỏ hơn số các nút trên cây j thì cho i là con của j, nếu không thì i làm cha của j". Như vậy tất phải biết được số nút của bất kỳ cây nào. Điều đó sẽ được giải quyết bằng cách: ở nút gốc i của một cây ta đưa vào trường COUNT(i) ghi nhận số nút của cây đó. Có thể tận dụng ngay trường PARENT của nút gốc đó (vì theo qui ước: lấy chỉ số của gốc cây, biểu diễn thay cho tên tập, thì trường này của nút gốc không dùng đến) nhưng phải gán cho dấu âm (dùng thêm một trường 1 bit) để phân biệt với giá trị PARENT dương, chỉ con trỏ, ở các nút khác gốc. Như vậy giải thuật của phép UNION có thể viết:

Procedure UNION (i;j)

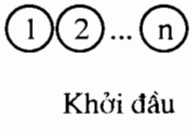
{Thực hiện phép hợp hai tập với gốc tương ứng là i và j, $j \neq i$, mà PARENT (i) = - COUNT(i) và PARENT(j) = -COUNT(j)}

```
1)  x := PARENT(i) + PARENT(j)
2)  if PARENT (i) > PARENT(j)
        then begin
                PARENT (i) := j;
                PARENT (j) := x
        end
    else begin
        PARENT (j) := i;
        PARENT (j) := x
        end
3)  return
```

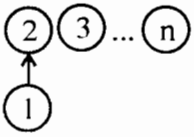
Với giải thuật này cây tương ứng với dãy phép UNION-FIND nêu ở trên sẽ có dạng như hình 6.29.

Ta thấy ngay: Thời gian thực hiện tất cả các phép FIND ở đây chỉ là O(n) vì cây biểu diễn các tập UNION đều chỉ có mức tối đa là 2. Rõ ràng

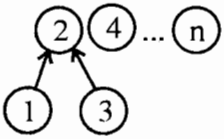
nó không còn "tồi" như trước nữa. Trong trường hợp tổng quát, bổ đề sau đây sẽ chứng tỏ mức tối đa với cây UNION cũng chỉ là $\lfloor \log_2 n \rfloor + 1$ và thời gian thực hiện của một phép FIND chỉ là $O(\log_2 n)$.



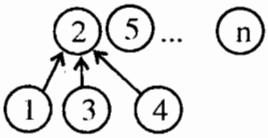
Khởi đầu



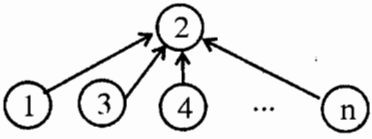
UNION (1,2), 2 = F(1)



UNION (2,3), 2 = F(1)



UNION (2,4), 2 = F(1)



... 2 = F(1), UNION (2,n)

Hình 6.29

Bổ đề: Nếu T là một cây với n nút được tạo lập bởi giải thuật UNION thì không nút nào trên T có mức lớn hơn $\lfloor \log_2 n \rfloor + 1$.

Chứng minh

Với n = 1 bổ đề hoàn toàn đúng.

Giả sử nó đã đúng với mọi cây có i nút mà $i \leq n - 1$ ta sẽ chứng minh nó cũng đúng với i = n. Gọi T là nút được tạo bởi giải thuật UNION nêu trên. Hãy xét tới phép hợp được thực hiện cuối cùng: UNION(k,j). Gọi m là số nút của cây j và n - m là số nút của cây k. Không hề mất tính chất tổng quát nếu ta giả sử $1 \leq m \leq n/2$. Vậy thì mức lớn nhất của nút trên cây T sẽ hoặc bằng số mức lớn nhất ở cây k, hoặc bằng số mức lớn nhất ở cây j cộng thêm 1 (vì j là con của k). Nếu trường hợp thứ nhất xảy ra thì số mức lớn nhất ở T sẽ $\lfloor \log_2(n-m) + 1 \rfloor \leq \lfloor \log_2 n \rfloor + 1$. Nếu trường hợp sau xảy ra thì số mức lớn nhất trong T sẽ $\leq \lfloor \log_2 m \rfloor + 1 + 1 \leq \lfloor \log_2 n/2 \rfloor + 2 \leq \lfloor \log_2 n \rfloor + 1$.

Sau đây ta xét tới một ví dụ về tác động của giải thuật UNION trên một dãy các phép hợp đối với các tập mà tình trạng khởi đầu giống như ở các ví dụ nêu trên, cụ thể có 8 tập {1}, {2}, ..., {8}.

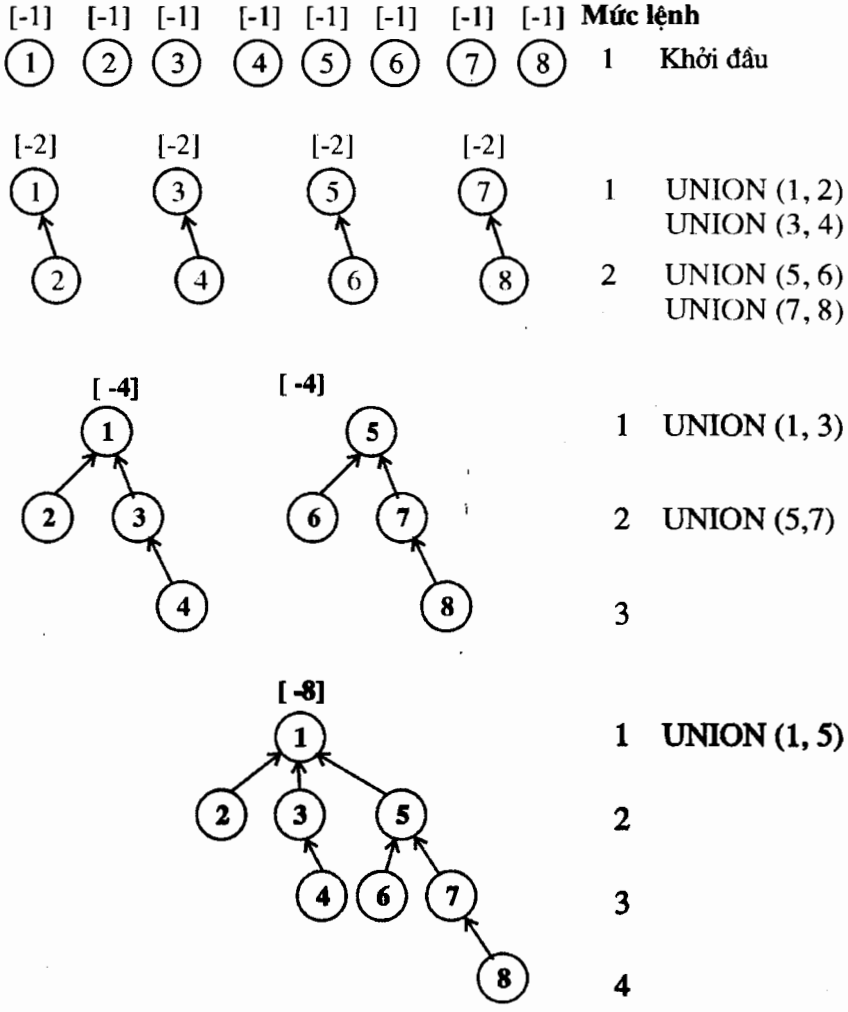
Cây biểu diễn tương ứng có PARENT (i) = -COUNT(i) $1 \le i \le 8$.

Còn dãy các phép toán là:

UNION (1,2), UNION(3,4), UNION(5,6), UNION(7,8)

UNION(1,3), UNION(5,7), UNION(1,5)

Các cây biểu diễn kết quả, qua quá trình tác động các phép hợp trên, cho bởi hình 6.30.



Hình 6.30

Ta thấy ở đây mức lớn nhất là $4 = \lfloor \log_2 8 \rfloor + 1$

Vẫn còn có thể có những cải tiến khác, nhưng ta sẽ không đi sâu thêm, mà dừng lại ở đây.

6.4.3 Cây quyết định (decision tree)

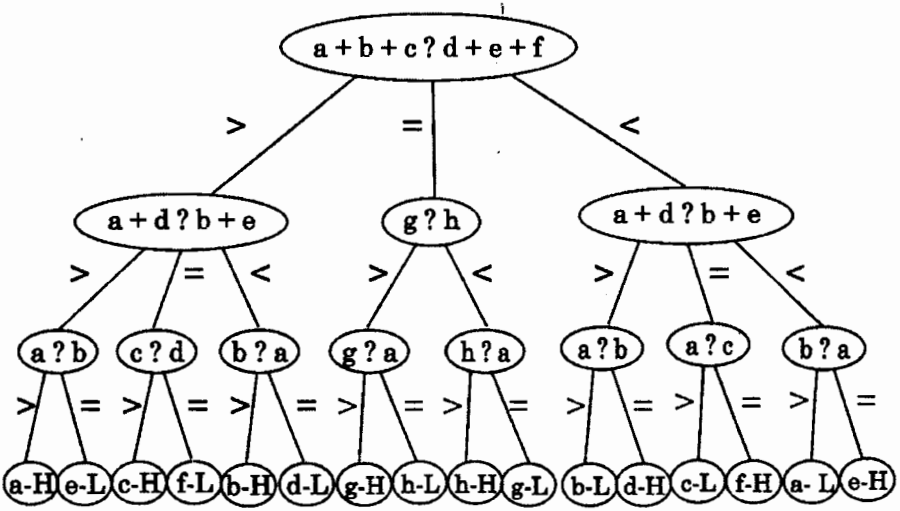
Cây còn được dùng khá phổ biến để biểu diễn lời giải của những bài toán mà đặc điểm thể hiện ở chỗ xuất hiện nhiều tình huống, nhiều khả năng đòi hỏi phải có một quyết định lựa chọn. Cây biểu diễn như vậy gọi là *cây quyết định.*

Xét bài toán khá quen thuộc: bài toán 8 đồng tiền vàng. Giả sử có 8 đồng tiền vàng a, b, c, d, e, f, g, h mà trong đó biết chắc rằng có một đồng tiền có trọng lượng không chuẩn. Vấn đề đặt ra là: xác định đồng tiền không chuẩn đó bằng cách sử dụng một cân Roberval (cân 2 đĩa như ở các cửa hàng vàng bạc). Ta cũng muốn số phép so sánh (cân thử) là ít nhất và đồng thời chỉ ra được đồng tiền đó nhẹ hơn hay nặng hơn đồng tiền chuẩn. Cây ở hình 6.31 sẽ biểu diễn một tập các quyết định (ứng với kết quả của các phép cân thử) mà theo đó ta sẽ đi tới lời giải của bài toán.

Ta sẽ ký hiệu:

H để chỉ đồng tiền không chuẩn là nặng hơn đồng tiền chuẩn.

L để chỉ đồng tiền nhẹ hơn.



Hình 6.31

Ta hãy theo dõi một dãy các khả năng (theo một đường đi trên cây).

Nếu a + b + c < d + e + f thì nghĩa là đồng tiền không chuẩn phải nằm trong 6 đồng tiền này chứ không phải là g và h. Nếu a + d < b + e thì sự đổi chỗ giữa d và b từ đĩa cân này sang đĩa cân kia và bớt c, f đi đã không hề làm thay đổi tình huống, vậy thì c và f không phải là đồng tiền cần tìm, b và d cũng vậy, thế thì chỉ còn a và e. Nếu a = b nghĩa là a là đồng tiền chuẩn, vậy e là không chuẩn mà e trước đó nằm ở đĩa cân bên phải (bên nặng) nên có thể kết luận e sẽ nặng hơn đồng tiền chuẩn.

Như vậy dọc trên cây này tuỳ theo "quyết định" mà sẽ đi tới khả năng này hay khả năng khác của lời giải bài toán. Dù ở tình huống nào số lượng phép so sánh cũng chỉ là 3, đây chính là số lượng tối thiểu. Ở đây ta cũng thấy được mọi khả năng có thể xảy ra và từ đó hình dung được rõ nét hơn giải thuật giải bài toán.

Riêng với bài toán 8 đồng tiền vàng, thì giải thuật là sự phản ánh rất trung thực cây quyết định ở hình 6.31.

Sau đây là giải thuật được viết dưới dạng đệ qui.

```
Program EIGHTCOINS
1) {Trọng lượng của 8 đồng tiền vàng được đưa vào}
   read (a, b, c, d, e, f, g, h);
1) case
   a+b+c = d+e+f : if g >h then call COMP(g,h,a)
                                else call COMP(h,g,a);
   a+b+c > d+e+f : case
           a + d = b + e : call COMP (c, f, a);
           a + d > b + e : call COMP (a, e, b);
           a + d < b + e : call COMP (b, d, a);
                   end case;
   a + b + c < d + e + f : case
           a + d = b + e: call COMP (f, c, a);
           a + d > b + e: call COMP (d, b, a);
           a + d < b + e: call COMP (e, a, b);
                   end case;
   end case
3) end
```

Trong đó, thủ tục COMP được viết như sau:

Procedure COMP(x,y,z) {x được so sánh với đồng tiền chuẩn z}

1) **if** x > z **then write** ('x nặng')

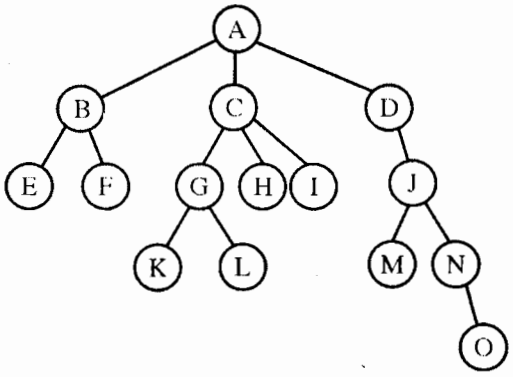
   else write ('y nhẹ')

2) **return**

Chú thích. Cây còn có nhiều ứng dụng khác. Ngay trong giáo trình này ở phần III: "Sắp xếp và tìm kiếm", ta sẽ còn gặp lại những ứng dụng mới của cây.

BÀI TẬP CHƯƠNG 6

6.1. Cho cây:



Hãy trả lời các câu hỏi sau:

a) Các nút nào là nút lá?

b) Các nút nào là nút nhánh?

c) Cha của nút G là nút nào?

d) Con của nút C là nút nào?

e) Các nút nào là anh em của B?

f) Mức của D, của L là bao nhiêu?

g) Cấp của B, của D là bao nhiêu?

Cấp của cây này là bao nhiêu?

h) Chiều cao của cây này là bao nhiêu?

i) Độ dài đường đi từ A tới F, từ A tới O là bao nhiêu?

j) Có bao nhiêu đường đi có độ dài 3 trên cây này?

6.2. Vẽ cây nhị phân biểu diễn các biểu thức sau đây và viết chúng dưới dạng tiền tố, hậu tố.

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

b) A/(B + C) + D * E - A * C

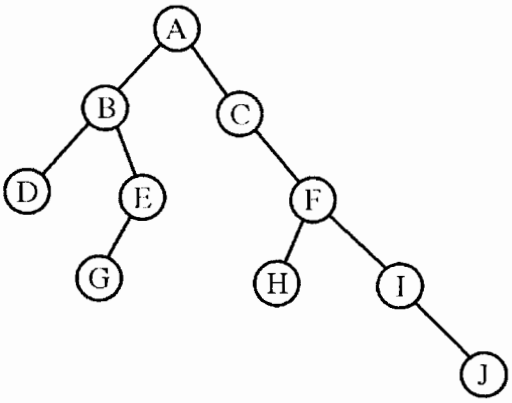
6.3. Cho cây nhị phân

Hãy viết dãy các nút được thăm khi duyệt cây này

a) Theo thứ tự trước.

b) Theo thứ tự giữa

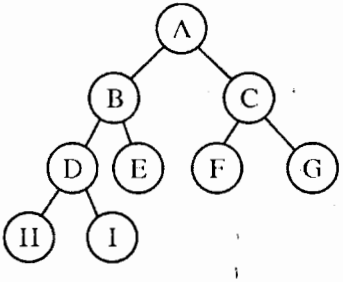
c) Theo thứ tự sau.



6.4. Cho cây nhị phân

a) Hãy minh hoạ phần bộ nhớ khi thực hiện lưu trữ kế tiếp đối với cây đó.

b) Vẽ cây nhị phân nối vòng biểu diễn cây đó.



6.5. Chứng minh rằng: đối với cây nhị phân nếu n_0 là số lượng nút tận cùng, n_2 là số lượng nút cấp 2, thì $n_0 = n_2 + 1$.

6.6. Chứng tỏ rằng nếu cho biết dãy các nút được thăm của một cây nhị phân khi duyệt theo thứ tự trước và thứ tự giữa, thì có thể dựng được cây nhị phân đó.

Điều này còn đúng nữa không đối với thứ tự trước và thứ tự sau? đối với thứ tự giữa và thứ tự sau?

6.7. Tìm tất cả các cây nhị phân mà các nút sẽ xuất hiện theo một dãy giống nhau khi duyệt.

a) Theo thứ tự trước và thứ tự giữa.

b) Theo thứ tự trước và thứ tự sau.

c) Theo thứ tự giữa và thứ tự sau.

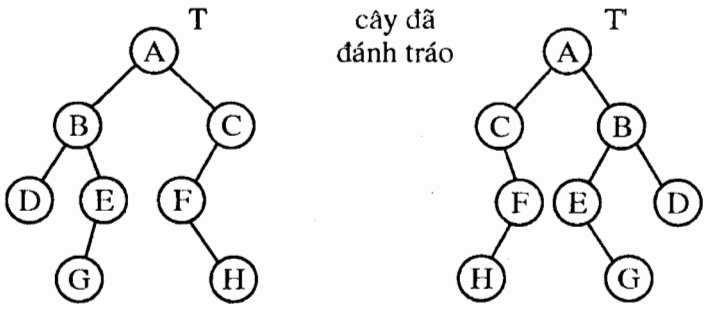
6.8. Khi thực hiện duyệt cây nhị phân theo thứ tự trước theo thủ tục PREORDER nêu ở 6.2.3, đối với cây nhị phân có n nút thì số lượng từ

máy dự trữ cho stack S phải là bao nhiêu (giả sử mỗi từ máy lưu trữ 1 địa chỉ - một con trỏ)?

6.9. Lập thủ tục không đệ qui thực hiện phép duyệt cấp nhị phân trỏ bởi T theo thứ tự giữa.

6.10. Lập giải thuật đệ qui thực hiện việc lập bản sao của một cây nhị phân trỏ bởi T (tạo lập một cây y hệt cây T).

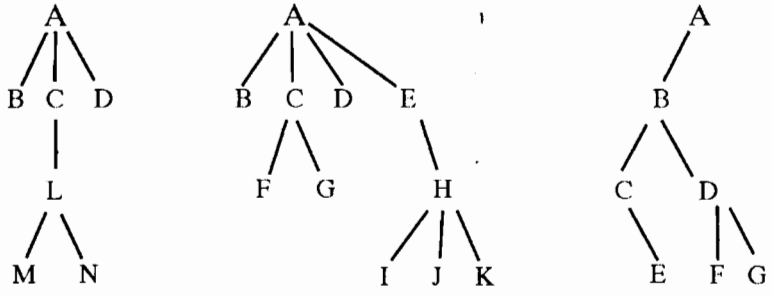
6.11. Lập giả thiết đệ qui thực hiện việc đánh tráo thứ tự trái, phải của các con của mọi nút trên cây nhị phân trỏ bởi T.



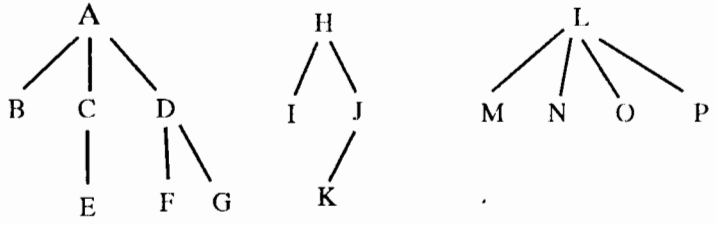
6.12. Hãy nêu qui ước, tương tự như trong bài, đối với cây nhị phân nối vòng hướng theo thứ tự trước và thứ tự sau (qui ước trong bài là hướng theo thứ tự giữa).

6.13. Dựng cây nhị phân tương đương với

a) Các cây sau:



b) Với rừng



6.14. Nêu dãy tên các nút được thăm trong các phép duyệt đối với rừng cho ở bài 6.13b.

6.15. Theo qui ước của Perlis ở cây nối vòng nếu tại một nút nào đó có mối nối không, dù nối trái hay nối phải thì nó cũng được thay đổi mối nối vòng.

Có trường hợp người ta chỉ thực hiện phép thay đó với mối nối phải thôi, nghĩa là:

Nếu LPTR(P) = null thì vẫn để nguyên.

Nếu RPTR(P) = null thì thay nó bằng P^+ và RBIT(P)=1

Như vậy trường LBIT không cần có trong một nút nữa. Cây nối vòng như vậy được gọi là cây nối vòng phải.

Xét bài toán: cho một cây nối vòng phải và con trỏ P trỏ tới một nút trên cây đó.

Hãy lập giải thuật cắt P và cây con của nó ra khỏi cây cũ, rồi gắn nó vào thành con trái của một nút trỏ bởi HEAD (để HEAD trở thành nút đầu cây của cây mới này).

Chú ý rằng sau phép cắt này cả cây mới lẫn cây cũ đều phải có dạng cây nối vòng phải.

6.16. Hãy sửa lại giải thuật DIFFER nêu trong bài nếu ta muốn đưa thêm vào phép luỹ thừa mà đạo hàm của nó theo qui tắc:

D(u ↑ v) = D(u) *(v * (u ↑ (v-1)) + (LN (u) * D(v)) * (u ↑ v)

Chương 7

ĐỒ THỊ VÀ CÁC CẤU TRÚC PHI TUYẾN KHÁC

7.1 Định nghĩa và các khái niệm

Một đồ thị (graph) G (V, E) bao gồm một tập hợp hữu hạn V các *nút*, hay *đỉnh* (vertices) và một tập hợp hữu hạn E các cặp đỉnh mà ta gọi là *cung* (edges).

Nếu (v_1, v_2) là cặp đỉnh thuộc E thì ta nói: có một cung nối v_1 và v_2. Nếu cung (v_1, v_2) khác với cung (v_2, v_1) thì ta có một *đồ thị định hướng* (directed graph hay digraph). Lúc đó (v_1, v_2) được gọi là cung định hướng v_1, v_2. Nếu thứ tự các nút trên cung không được coi trọng thì ta gọi *đồ thị không định hướng* (undirected graph)

Bằng hình vẽ có thể biểu diễn bằng đồ thị như hình sau:

a) Đồ thị định hướng b) Đồ thị không định hướng

Hình 7.1

Mạch điện, mạng lưới giao thông, mạng lưới máy tính... là các ví dụ thực tế của đồ thị. Cây là một trường hợp đặc biệt của đồ thị.

Nếu (v_1, v_2) là một cung trong tập E(G) thì v_1 và v_2 gọi là *lân cận* của nhau (adjacent).

Một đường đi (path) từ đỉnh v_p đến đỉnh v_q trong đồ thị G là một dãy đỉnh v_p, v_{i1}, v_{i2}..., v_{in}, v_q mà (v_p, v_{i1}), (v_{i1}, v_{i2})..., (v_{in}, v_q) là các cung trong E(G). Số lượng các cung trên đường đi ấy gọi là *độ dài của đường đi* (path length). Ví dụ: Trên hình 7.1a) 1, 3, 4 là đường đi từ đỉnh 1 tới 4, nó có độ dài 2. Còn trên hình 7.1b) đường đi từ đỉnh 1 tới đỉnh 4 thì có thể là 1, 3, 5, 2, 4; nó có độ dài bằng 4.

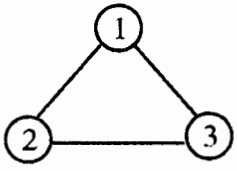
Một *đường đi đơn* (simple path) là đường đi mà mọi đỉnh trên đó, trừ đỉnh đầu và đỉnh cuối, đều khác nhau.

Một *chu trình* (cycle) là một đường đi đơn mà đỉnh đầu và đỉnh cuối trùng nhau. Ví dụ trong hình 7.1a) đường đi 1, 3, 4, 1 là một chu trình..

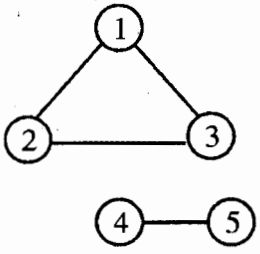
Đối với đồ thị định hướng, để cho rõ , thường người ta thêm vào các từ "định hướng" sau các thuật ngữ nêu ở trên. Ví dụ: đường đi định hướng từ v_i đến v_j.

Trong đồ thị G hai đỉnh v_i, v_j gọi là *liên thông* (connected) nếu có một đường đi từ v_i tới v_j (dĩ nhiên với đồ thị không định hướng thì đồng thời cũng có đường đi từ v_j đến v_i).

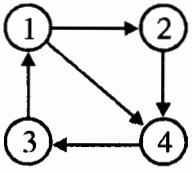
Một đồ thị G gọi là *liên thông* nếu đối với *mọi* cặp đỉnh phân biệt v_i, v_j trong V(G) đều có một đường đi từ v_i tới v_j. Ví dụ:



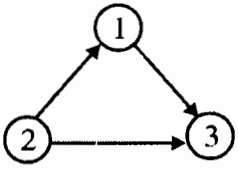
a) Đồ thị không định hướng, liên thông



b) Đồ thị không định hướng, không liên thông



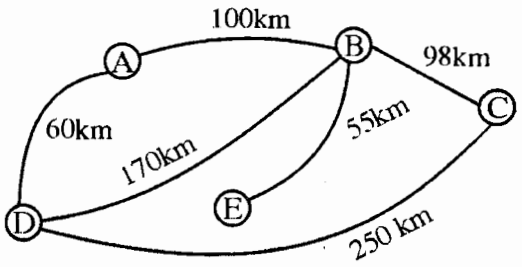
c) Đồ thị định hướng liên thông



d) Đồ thị định hướng không liên thông

Hình 7.2

Có khi, ở mỗi cung của đồ thị người ta gắn một giá trị thể hiện một thông tin nào đó liên quan đến cung (mà nó còn được gọi là *trọng số*), trong trường hợp này đồ thị được gọi là: *đồ thị có trọng số* (weighted graph). Ví dụ mạng lưới giao thông đường bộ giữa các tỉnh với trọng số ứng với mỗi tuyến đường giữa hai thành phố là độ dài của tuyến đường đó, hoặc giá cước vận chuyển theo tấn/km, hoặc thông lượng xe trong ngày...



Hình 7.3

7.2 Biểu diễn đồ thị

Có nhiều cấu trúc được sử dụng để biểu diễn đồ thị. Việc lựa chọn cấu trúc nào là tuỳ thuộc vào các ứng dụng và các phép xử lý cần tác động lên đồ thị trong ứng dụng ấy.

Trước hết ta xét tới hai cách biểu diễn sau đây:

7.2.1 Biểu diễn bằng ma trận lân cận (kề) (adjacency matrice)

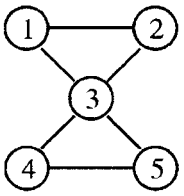
Xét một đồ thị G(V, E) định hướng với V gồm có n đỉnh ($n \geq 1$) mà giả sử các đỉnh đã được đánh số theo một quy định nào đó. Ma trận lân cận A biểu diễn G là một ma trận vuông kích thước nxn. Các phần tử của ma trận có giá trị 0 hoặc 1. Nếu phần tử $a_{ij} = 1$ thì điều đó có nghĩa là tồn tại một cung định hướng (v_i, v_j) trong E; còn $a_{ij} = 0$ thì không tồn tại cung như vậy.

Rõ ràng với đồ thị không định hướng thì ma trận lân cận biểu diễn nó có các phần tử đối xứng qua đường chéo chính, nghĩa là có phần tử bằng 1 ở hàng i cột j thì cũng có phần tử 1 ở hàng j cột i. Còn đồ thị định hướng thì không phải như vậy.

Ví dụ:

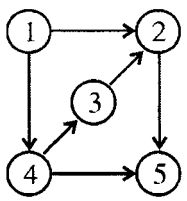
Đồ thị Ma trận lân cận

a)



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

b)



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

Hình 7.4

Dĩ nhiên đối với một đồ thị đã cho G = (V, E) thì ma trận lân cận phụ thuộc vào thứ tự ấn định đối với các nút. Với cách đánh số thứ tự khác nhau đối với các đỉnh của V, ta sẽ có các ma trận lân cận khác nhau của cùng một đồ thị G. Tuy nhiên từ một ma trận lân cận nào đó của G cũng có thể lập được một ma trận lân cận khác của G, bằng cách đổi chỗ một số hàng và cột tương ứng của ma trận. Vì vậy vấn đề số thứ tự ấn định cho các nút có thể để tuỳ ý được.

Đối với đồ thị có trọng số thì ma trận lân cận có thể lập bằng cách thay giá trị 1 bởi trọng số tương ứng của các cung.

Ta cũng thấy ngay không gian nhớ cần cho cách biểu diễn này là các bit (với đồ thị không định hướng thì có thể giảm bớt một nửa bằng cách chỉ lưu trữ phần trên (hoặc dưới), đường chéo chính bằng một ma trận tam giác). Trong trường hợp mà đồ thị có ít cung (ma trận lân cận chứa nhiều phần tử 0), thì ta thấy ngay nhược điểm của cách biểu diễn này. Điều đó sẽ khắc phục trong cách biểu diễn tiếp theo.

7.2.2 Biểu diễn bằng danh sách lân cận (kề) (adjacency list)

Trong cách biểu diễn này, n hàng của ma trận lân cận được thay bởi n danh sách móc nối.

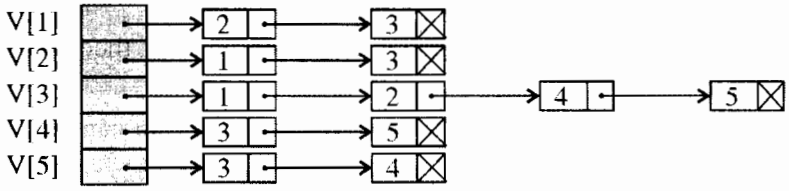
Với mỗi đỉnh của G có một danh sách tương ứng. Các nút trong danh sách i biểu diễn các đỉnh lân cận của nút i. Mỗi nút có hai trường VERTEX và LINK.

Trường VERTEX chứa chỉ số (số thứ tự) của các đỉnh lân cận của đỉnh i.

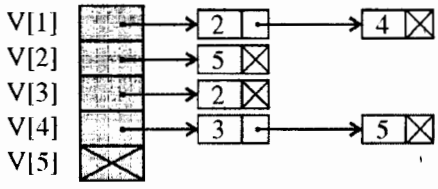
Trường LINK chứa con trỏ, trỏ tới nút tiếp theo trong danh sách.

Với các đồ thị ở hình 7.4 cách biểu diễn sẽ như sau:

a)



b)



Hình 7.5

Mỗi danh sách có một nút "đầu danh sách". Các nút đầu này được tổ chức kế tiếp (dưới dạng vectơ) để truy nhập được nhanh.

Trường hợp đồ thị không định hướng có n đỉnh, e cung thì cần n nút đầu và 2e "nút danh sách".

Với đồ thị định hướng thì số nút danh sách chỉ cần e.

Cần chú ý là thứ tự của các nút trong từng danh sách thực ra không quan trọng.

7.3 Phép duyệt một đồ thị

Khi biết gốc của một cây ta có thể thực hiện phép duyệt cây đó để thăm các nút của cây theo thứ tự nào đấy.

Với đồ thị vấn đề đặt ra cũng tương tự. Xét một đồ thị không định hướng G(V, E) và một đỉnh v trong V(G), ta cần thăm tất cả các đỉnh thuộc G mà có thể "với tới" được từ đỉnh v (nghĩa là thăm mọi nút liên thông với v).

Ta chú ý tới hai cách giải quyết trên đây:

- Phép *tìm kiếm theo chiều sâu* (Depth first search)
- Phép *tìm kiếm theo chiều rộng* (Breadth first search)

7.3.1 Tìm kiếm theo chiều sâu

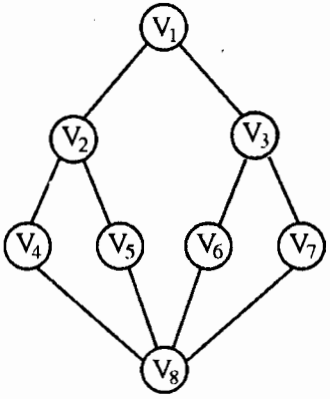
Tìm kiếm theo chiều sâu đối với một đồ thị không định hướng được thực hiện như sau:

Đỉnh xuất phát v được thăm. Tiếp theo đó một đỉnh ω chưa được thăm, mà là lân cận của v, sẽ được chọn và một phép tìm kếm theo chiều sâu xuất phát từ ω lại được thực hiện.

Khi một đỉnh u đã được "với tới" mà mọi đỉnh lân cận của nó đều đã được thăm rồi, thì ta sẽ quay ngược lên đỉnh cuối cùng vừa được thăm (mà còn có đỉnh ω lân cận với nó chưa được thăm), và một phép tìm kiếm theo chiều sâu xuất phát từ ω lại được thực hiện. Phép tìm kiếm sẽ kết thúc khi không còn một nút nào chưa được thăm mà vẫn có thể với tới được từ nút đã được thăm.

Nếu đồ thị G có dạng như ở hình 7.6 và đỉnh xuất phát là V_1 thì dãy các đỉnh được thăm sẽ như sau:

Thoạt đầu là V_1, rồi tới V_2 (tất nhiên có thể là V_3) rồi V_4, V_8, V_5; do V_5 chưa có lân cận nào chưa được thăm nên phải quay lại V_8 để thăm tiếp V_6 rồi V_3, V_7.



Hình 7.6 .

Giải thuật của phép duyệt này:

Procedure DFS (v)

{Cho một đồ thị không định hướng G(V, E) với n đỉnh và một vectơ VISITED (n) gồm n phần tử thoạt đầu có giá trị bằng 0. Giải thuật này được thực hiện việc thăm mọi đỉnh "với tới" được từ đỉnh v. Đối với thủ tục này G và VISITED không phải là cục bộ}

1) VISITED (v) := 1; {ở đây VISITED dùng để đánh dấu các đỉnh đã được thăm}
2) **for** mỗi đỉnh ω lân cận của v **do**

   if VISITED (ω) = 0 **then call** DFS (W);
3) **return**

* Ta thấy:

Trong trường hợp G được biểu diễn bởi một danh sách lân cận thì đỉnh ω lân cận của v sẽ được xác định bằng cách dựa vào danh sách móc nối ứng với v. Vì giải thuật DFS chỉ xem xét mỗi nút trong một danh sách lân cận nhiều nhất một lần thôi mà lại có 2e nút danh sách (ứng với e cung), nên thời gian để hoàn thành phép tìm kiếm chỉ là O(e).

Còn nếu G được biểu diễn bởi ma trận lân cận thì thời gian để xác định mọi đỉnh lân cận của v là O(n). Vì tối đa có n đỉnh được thăm, nên thời gian tìm kiếm tổng quát sẽ là $O(n^2)$.

Giải thuật DFS (V_1) sẽ đảm bảo thăm mọi đỉnh liên thông với V_1. Tất cả các đỉnh được thăm cùng với các cung liên quan tới các đỉnh đó gọi là một *bộ phận liên thông* của G (connected component of G). Với phép duyệt như DFS ta có thể xác định được G có liên thông hay không, hoặc tìm được các bộ phận liên thông của G nếu G không liên thông.

7.3.2 Tìm kiếm theo chiều rộng

Đỉnh xuất phát v ở đây cũng được thăm đầu tiên, nhưng có khác với DFS ở chỗ là: sau đó các đỉnh chưa được thăm mà là lân cận của v sẽ được thăm kế tiếp nhau, rồi mới đến các đỉnh chưa được thăm là lân cận lần lượt của các đỉnh này và cứ tương tự như vậy.

Ví dụ với đồ thị đã nêu ở hình 7.6 thì V_1 được thăm rồi đến V_2, V_3...tiếp theo là V_4, V_5 và V_6, V_7 cuối cùng là V_8.

Sau đây là giải thuật:

Procedure BFS (v)

{Phép tìm kiếm theo chiều rộng đối với G được thực hiện bắt đầu từ đỉnh v. Mọi đỉnh i được thăm sẽ được đánh dấu với VISITED (i) := 1. Thoạt đầu VISITED(i) có giá trị đều bằng 0. Đối với thủ tục này G và VISITED không phải là cục bộ. Ở giải thuật này người ta còn dùng một queue kế tiếp có kích thước n, với F và R trỏ tới lối trước và lối sau. Thủ tục CQINSERT, CQDELETE (xem 3.5) sẽ được sử dụng để bổ sung hoặc loại bỏ phần tử}.

```
1) VISITED (v) := 1
2) Khởi tạo queue với v đã được nạp vào;
3) while Q không rỗng do begin
        call CQDELETE (v, Q); {lấy đỉnh v ra khỏi Q}
        for mỗi đỉnh w lân cận với v do
            if VISITED (w) := 0 then begin
                call CQINSERRT (w, Q);
                VISITED (w) := 1
                    end;
        end;
4) return
```

Mỗi đỉnh được thăm sẽ được nạp vào queue chỉ một lần vì vậy câu lệnh **while** lặp lại nhiều nhất n lần.

Nếu G được biểu diễn bởi ma trận lân cận thì câu lệnh **for** sẽ chi phí O(n) thời gian đối với mỗi đỉnh, do đó thời gian chi phí toàn bộ sẽ là $O(n^2)$.

Trường hợp G được biểu diễn với danh sách lân cận thì chi phí tổng quát chung là O(e).

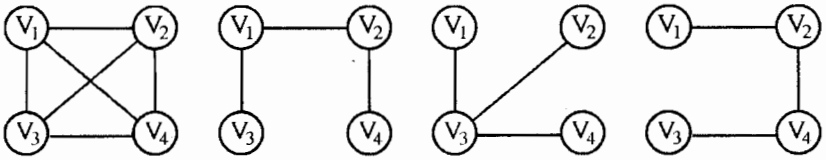
7.3.3 Cây khung và cây khung với giá trị cực tiểu (spanning trees and minimum cost spanning trees)

Khi một đồ thị G liên thông thì một phép tìm kiếm theo chiều sâu hoặc theo chiều rộng, sẽ xuất phát từ một đỉnh bất kỳ nào, cũng cho phép thăm được mọi đỉnh của G. Trong trường hợp này các cung của G sẽ được phân làm hai tập:

Tập T bao gồm tất cả các cung được dùng tới hoặc được duyệt qua trong phép tìm kiếm và tập B bao gồm các cung còn lại.

Tất cả các cung trong T cùng với các đỉnh tương ứng sẽ tạo thành một cây bao gồm mọi đỉnh (nút) của G. Một cây như vậy gọi là *cây khung* của G.

Hình 7.7 cho thấy một đồ thị và ba cây khung tương ứng với nó.



a) b) c)

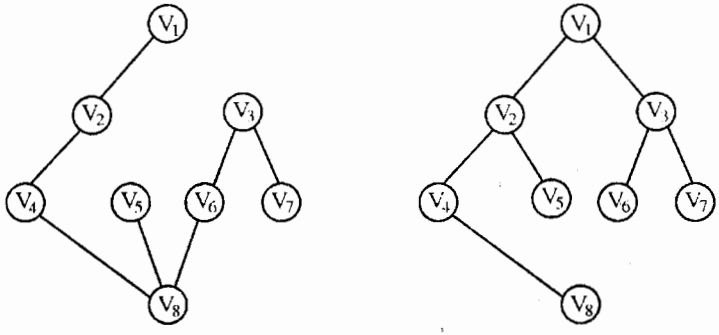
Hình 7.7

Tuỳ theo phép DFS hoặc BFS được sử dụng mà cây khung tương ứng sẽ được gọi là "cây khung theo chiều sâu" hay "cây khung theo chiều rộng".

Với đồ thị ở hình 7.6 ta có

a) Cây khung theo chiều sâu DFS (V_1)

b) Cây khung theo chiều rộng BFS (V_1)



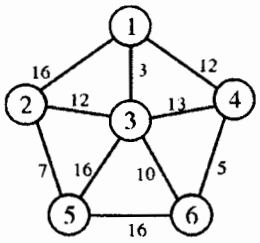
a) b)

Hình 7.8

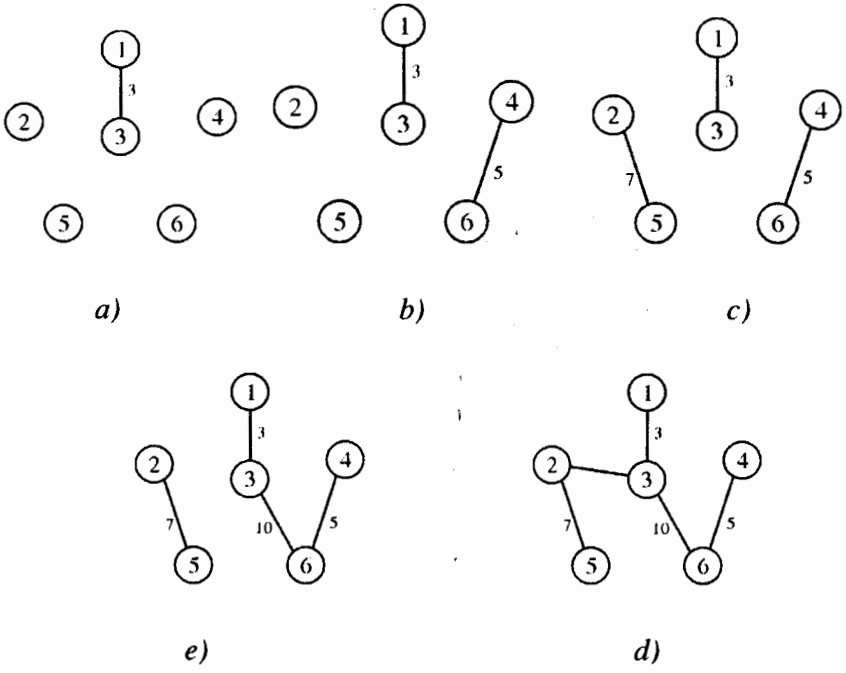
Ta thấy ngay một cung bất kỳ (v, ω) nào của B được bổ sung vào cây khung T thì một chu trình sẽ lập tức xuất hiện. Chu trình này bao gồm cung (v, ω) và mọi cung trên đường đi từ ω tới v trong T.

* Sau đây ta xét tới một ứng dụng của cây khung, đó là việc xác định cây khung với giá cực tiểu của một đồ thị liên thông có trọng số. Giá của một cây khung là tổng các trọng số ứng với các cành (các cung) của cây ấy. Nếu các đỉnh của G biểu diễn các thành phố, cung biểu diễn đường nối giữa hai thành phố và trọng số biểu diễn giá tiền xây dựng hoặc độ dài của đường nối... thì việc chọn một cây khung với giá cực tiểu là tương ứng với việc xây dựng một mạng đường giao thông có khả năng liên thông mọi thành phố với giá xây dựng tổng cộng là ít nhất hoặc với tổng độ dài là ngắn nhất...

Có nhiều giải thuật khác nhau để xây dựng cây khung với giá trị cực tiểu, dưới đây ta sẽ nêu một giải thuật của Kruskal. Với cách tiếp cận này, cây khung với giá trị cực tiểu T sẽ được xây dựng dần dần từng cung một. Các cung được xét để đưa vào T dựa theo thứ tự không giảm của giá tương ứng với chúng. Một cung được đưa vào T nếu nó không tạo nên một chu trình với các cung đã ở trong T. Vì G liên thông với số đỉnh n > 0, nên sẽ có n - 1 cung được chọn để đưa vào T. Ví dụ: với đồ thị ở hình 7.9, cây khung với giá trị cực tiểu được dựng bởi giải thuật của Kruskal được thể hiện từng bước ở hình 7.10



Hình 7.9



Hình 7.10

Các cung của đồ thị này được xét để đưa vào cây khung với giá cực tiểu theo thứ tự:

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

Chúng tương ứng với dãy các giá

3, 5, 7, 10, 12, 12, 13, 16, 16, 16

Bốn cung đầu tiên đều được nhận đưa vào T, cung tiếp theo (1, 4) bị loại bỏ vì nó nối 2 đỉnh ở trong T và tạo nên một chu trình, cung (2, 3) được tiếp nhận, nhưng (3, 4), (1, 2), (3, 5), (5, 6) đều bị loại bỏ vì cũng tạo nên chu trình trong T.

Rõ ràng ở đây n = 5 và cây khung có 5 cành cây khung này có giá bằng 37 (hình 7.10e). Đây chính là cây khung với giá cực tiểu:

Giải thuật Kruskal có thể viết một cách "thô" như sau:

1) T := Φ; {thoạt đầu T rỗng}
2) **while** T chứa ít hơn (n - 1) cung **do**
3) **begin** Chọn một cung (v,ω) từ E có giá trị thấp nhất;
4) loại (v, ω khỏi E;
5) **if** (v, ω) không tạo nên chu trình trong T
6) **then** đưa (v, ω) vào T
7) **else** loại bỏ (v, w)
end
8) **return**

Ứng dụng biểu diễn tập như 6.4.2 và các giải thuật FIND, UNION ta có thể thực hiện thuận lợi ở các bước 5) và 6).

Thời gian thực hiện giải thuật này được xác định qua đánh giá thời gian thực hiện ở bước 3) và 4) mà trong trường hợp xấu nhất sẽ là $O(elog_2e)$ với e là các số cung thuộc E(G).

7.4 Áp dụng

Như ta đã biết đồ thị thường được dùng để biểu diễn mạng đường giao thông, nếu được gán thêm trọng số vào cung thì các trọng số đó sẽ mang ý nghĩa thực tiễn riêng trong từng trường hợp như đã nêu ở phần trên. Đối với một người đi mô tô chẳng hạn từ thành phố A tới thành phố B, thì thường hai câu hỏi sau đây được đặt ra.

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

Các bài toán này chỉ là trường hợp riêng của bài toán đặt ra chung dưới đây đối với một đồ thị mà độ dài đường đi được coi là tổng các trọng số của các cung trên đường đi đó. Đỉnh xuất phát của đường đi được gọi là *nguồn* (source), đỉnh tận cùng được gọi là *đích* (destination). Đồ thị ta xét ở đây là đồ thị định hướng (trường hợp không định hướng có thể xem xét một cung ứng với hai cung định hướng) và trọng số ta giả sử là dương.

7.4.1 Bài toán bao đóng truyền ứng (trasitive closure)

Bài toán 1) nêu trên có thể giải quyết được dễ dàng bằng cách sử dụng ma trận lân cận của đồ thị.

Trước hết ta thấy: có thể coi ma trận lân cận như một ma trận Bool. Như vậy có thể tác động lên ma trận đó các phép toán lôgic đối với ma trận Bool được.

Ta nhắc lại hai phép toán lôgic:

- Phép cộng (hay phép tuyển, phép hoặc): $\vee$
- Phép nhân (hay phép hội, phép và): $\wedge$

mà định nghĩa được cho bởi bảng sau:

a	b	$a \vee b$	$a \wedge b$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Đối với ma trận Bool A và B, kích thước n x n, phép cộng A với B để cho ma trận tổng C

$$C = A \vee B$$

được thực hiện bằng cách tính

$$c_{ij} = a_{ij} \vee b_{ij} \text{ với } \quad 1 \le i \le n$$
$$1 \le j \le n$$

Phép nhân A với B để cho ma trận tích D được thực hiện bằng phép tính:

$$d_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge b_{kj}) \quad \text{với} \quad 1 \le i \le n$$
$$1 \le j \le n$$

Ta đã biết với ma trận lân cận A, nếu $a_{ij} = 1$ thì nghĩa là có một cung, hay một đường đi độ dài 1, từ đỉnh i tới đỉnh j.

Bây giờ thử xét

$$A^{(2)} = A \wedge A$$

rõ ràng nếu phần tử ở hàng i cột j của $A^{(2)}$ bằng 1 thì ít nhất cũng có một đường đi có độ dài 2 từ đỉnh i tới đỉnh j, vì

$$a_{ij}^{(2)} = \bigvee_{k=1}^{n} (a_{ik} \wedge a_{kj}) = 1$$

thì ít nhất cũng có một giá trị k mà $(a_{ik} \wedge a_{kj}) = 1$, mà $a_{ik} \wedge a_{kj}$ chỉ bằng 1 khi

và chỉ khi cả $a_{ik} = 1$ và $a_{kj} = 1$, nghĩa là khi có đường đi độ dài 1 từ đỉnh i tới đỉnh k và có đường đi có độ dài 1 từ đỉnh k tới đỉnh j.

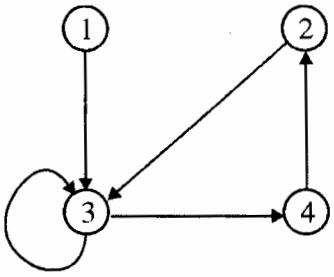
Từ đó suy ra điều tương tự cho $A^{(r)} = A \wedge A^{(r-1)}$ với r = 3, 4... nghĩa là nếu $a_{ij}^{(r)} = 1$ thì ít nhất có một đường đi độ dài r từ đỉnh i tới đỉnh j.

Như vậy, nếu ta lập ma trận

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

thì p sẽ cho biết: có hay không đường đi, có độ dài lớn nhất là n, từ đỉnh i tới đỉnh j. P được gọi là ma trận đường đi (path matrice) của đồ thị G.

Ví dụ: Với đồ thị



Hình 7.11

ta sẽ có

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

A $\qquad$ $A^{(2)}$ $\qquad$ $A^{(3)}$

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

$A^{(4)}$ $\qquad$ P

Việc tính P có thể dễ dàng thực hiện bởi giải thuật Warshall sau đây:

Procedure WARSHALL (A, P, n)

1) P := A; {Đây là phép gán quy ước: thay vào cho việc gán lần lượt các phần tử của A cho các phần tử tương ứng của P}

2) **for** k := 1 **to** n **do**

for i := 1 **to** n **do**

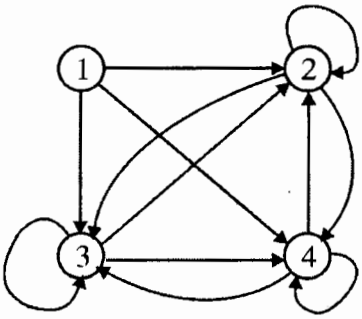
for j := 1 **to** n **do**

p[i; j]:= p[i;j] **or** p[i; k] **and** p[k; j];

3) **return**

Ma trận đường đi P có thể coi là ma trận lân cận của một đồ thị G'. G' khác với G ở chỗ: các cung không phải chỉ thể hiện mối quan hệ lân cận giữa 2 nút mà thể hiện mối quan hệ liên thông giữa chúng.

Với ma trận P nêu trên, đồ thị G' tương ứng sẽ là:



Hình 7.12

Một đồ thị như vậy được gọi là *bao đóng truyền ứng* của G.

Chú ý

1) Ma trận đường đi chỉ cho ta biết có hay không ít nhất một đường đi giữa một cặp đỉnh và có hay không một chu trình tại một đỉnh nào đó chứ nó không chỉ ra được mọi con đường có thể có đó. Về một mặt nào đấy ma trận đường đi không cho ta đầy đủ thông tin về đồ thị như là ma trận lân cận. Tuy nhiên ma trận đường đi vẫn có ý nghĩa quan trọng riêng của nó

2) Nếu ta chỉ quan tâm tới việc có hay không đường đi từ nút này tới một nút khác thì chỉ cần tính ma trận.

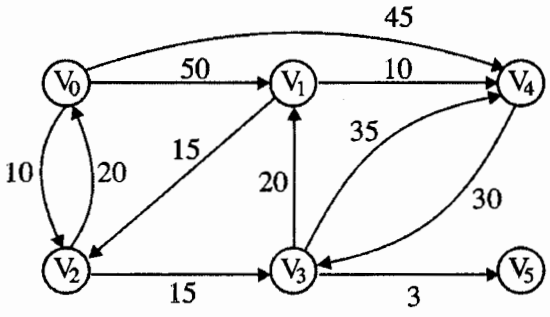
$$P_{n-1} = A \vee A^{(2)} \vee A^{(3)} \vee \ldots \vee A^{(n-1)}$$

là đủ, vì với n đỉnh thì một đường đi từ một đỉnh i đến đỉnh j có độ dài n không phải là một đường đi đơn.

7.4.2 Bài toán một nguồn, mọi đích (single source, all destination)

Cho đồ thị định hướng G(V, E), một hàm trọng số w(e) cho các cung e của G và một đỉnh nguồn V_o.

Bài toán đặt ra là: Xác định các đường đi ngắn nhất từ V_o đến mọi đỉnh còn lại của G (vẫn với cách hiểu: độ dài đường đi là tổng các trọng số tương ứng với các cung trên đường đi đó và các trọng số đều là số dương). Có thể xét ví dụ với đồ thị ở hình 7.13.



Hình 7.13

Nếu V_o là nguồn thì đường ngắn nhất từ V_o đến V_1 là V_o V_2 V_3 V_1 với độ dài bằng 10 + 15 + 20 = 45. Không có đường đi từ V_o đến V_5, còn các đường đi ngắn nhất từ V_o tới V_1, V_2, V_3, V_4 được cho bởi bảng sau, sắp xếp theo thứ tự không giảm

Đường đi	Độ dài
1) $V_0 V_2$	10
2) $V_0 V_2 V_3$	25
3) $V_0 V_2 V_3 V_1$	45
4) $V_0 V_4$	45

Bây giờ ta hãy đi sâu vào giải thuật xác định các đường đi ngắn nhất theo thứ tự như trong bảng trên.

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

Đối với một đỉnh ω không thuộc S, gọi DIST (ω) là độ dài của đường đi ngắn nhất từ V_o, qua các đỉnh, chỉ trong S, và kết thúc ở ω.

Ta có một số nhận xét như sau:

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

Để chứng minh điều đó ta sẽ chỉ ra rằng mọi đỉnh trung gian trên đường đi ngắn nhất từ V_0 tới ω đều phải thuộc S.

Giả sử một đỉnh u nào đó trên đường đi mà không thuộc S thì đường đi từ V_0 tới ω tất phải chứa đường đi từ V_0 tới u. Vậy độ dài đường đi từ V_0 tới u phải nhỏ hơn độ dài đường đi từ V_0 tới ω. Với giả thiết là các đường đi ngắn nhất được xác định theo thứ tự không giảm của độ dài đường đi thì độ dài đường đi từ V_0 tới u phải được sinh ra rồi, nghĩa là u đã phải thuộc S rồi!

Như vậy thì không có một đỉnh trung gian nào trên đường đi từ V_0 tới ω lại không thuộc S cả.

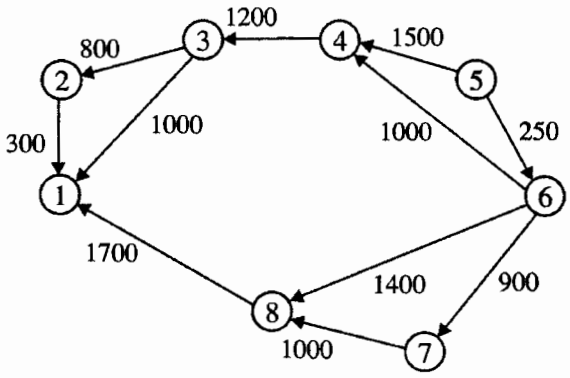
2) *Đích của đường đi sinh ra tiếp theo phải là một đỉnh ϖ nào đó hiện chưa thuộc S, mà có DIST(ω) ngắn nhất so với giá trị của DIST ứng với mọi đỉnh khác cũng đang không thuộc S.*

Điều này hiển nhiên vì do định nghĩa của DIST và do 1). Trong trường hợp có nhiều đỉnh đang không thuộc S mà cùng có một giá trị DIST như nhau thì chọn một đỉnh nào trong số đó để làm đích của đường đi ngắn nhất tiếp theo cũng được.

3) *Nếu đã chọn được một đỉnh ω như trong 2) và đã sinh ra được một đường đi ngắn nhất từ V_0 tới ω thì ω trở thành một phần tử của S.*

Với quan điểm này thì độ dài đường đi ngắn nhất xuất phát từ V_0, đi qua các đỉnh chỉ thuộc S và kết thúc ở đỉnh ω không thuộc S, có khả năng sẽ giảm đi; nghĩa là giá trị của DIST (ω) có thể thay đổi. Nếu nó thay đổi thì nó phải được lập từ một đường đi ngắn hơn, bắt đầu từ V_0 tới u rồi đến ω. Các đỉnh trung gian trên đường V_0 đến u và từ u tới ω tất đã phải thuộc S. Hơn nữa đường đi từ V_0 đến u phải là đường ngắn nhất vì nếu không thì DIST (ω) sẽ không được xác định đúng nghĩa. Ngoài ra, đường đi từ u tới ω thì có thể chọn sao cho nó không chứa một đỉnh trung gian nào. Như vậy có thể kết luận là DIST(ω) sẽ thay đổi (không tăng) vì đường đi từ V_0 tới u rồi tới ω đã có đoạn đường từ V_0 tới u là đường đi ngắn nhất và từ u tới ω chỉ là cung (u,ω). Độ dài đường đi đó sẽ là: DIST(u) + độ dài (u, ω).

Giải thuật SHORTEST-PATH dựa theo quan điểm này được đưa ra đầu tiên bởi Dijkstra. Ở đây giả thiết n đỉnh của G được đánh số từ 1 tới n. Tập S được thể hiện dưới dạng một vectơ bit với S[i] = 0 nếu đỉnh i không thuộc S, S[i] = 1 nếu nó thuộc S. Còn đồ thị có trọng số thì được biểu diễn bởi ma trận lân cận COST với phần tử COST[i;j] là trọng số của cung (i; j), COST[i;j] = + ∞ nếu cung (i; j) không có, COST[i, j] = 0 nếu i = j



Hình 7.14

Với đồ thị ở hình 7.14 thì ma trận COST có dạng

	1	2	3	4	5	6	7	8
1	0							
2	300	0				+∞		
3	1000	800	0					
4			1200	0				
5				1500	0	250		
6		+∞		1000		0	900	1400
7							0	1000
8	1700							0

Sau đây là giải thuật.

Procedure SHORTEST-PATH (v, COST, DIST, n)

{ DIST(j), $1 \le j \le n$: thể hiện độ dài đường đi ngắn nhất từ đỉnh v đến đỉnh j trong đồ thị định hướng G có n đỉnh, DIST(v)=0. G được biểu diễn bởi ma trận COST kích thước nxn}

1) **for** i: = 1 **to** n **do begin**

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

end;

2) S[v] : = 1; DIST[v] := 0; k :=2; { đưa v vào S }

3) **while** k < n **do** {xác định n - 1 đường đi từ đỉnh v}

4) **begin**

chọn u sao cho DIST [u] = $\min_{S[u]=0}$ (DIST[u])

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

6) **for** mọi ω với S[ω] = 0 **do**

7) DIST [ω] := min (DIST [ω], DIST[u] + COST[u, ω]

end

8) **return**

Với đồ thị ở hình 7.14 và giả sử v là nút 5 thì diễn biến của tác động giải thuật SHORTEST-PATH được thể hiện qua bảng sau:

Bước lặp	S	Đỉnh được chọn	DIST	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Khởi đầu	-			∞	∞	∞	1500	0	250	∞	∞
1	5	6		∞	∞	∞	1250	0	250	1150	1650
2	5, 6	7		∞	∞	∞	1250	0	250	1150	1650
3	5, 6, 7	4		∞	∞	2450	1250	0	250	1150	1650
4	5, 6, 7, 4	8		3350	∞	2450	1250	0	250	1150	1650
5	5, 6, 7, 4, 8	3		3350	3250	2450	1250	0	250	1150	1650
6	5, 6, 7, 4, 8. 3	2		3350	3250	2450	1250	0	250	1150	1650
	5, 6, 7, 4, 8, 3, 2										

Ta thấy: để đánh giá thời gian thực hiện giải thuật trên ta dựa vào phép toán 7/ vì đây là phép toán tích cực. Với 6/7/ phép toán này có thời gian chi phí là Cn với C là hằng số. Nhưng 6/7/ này thực hiện (n - 1) lần, vì vậy độ phức tạp về thời gian của giải thuật là $O(n^2)$.

7.4.3 Bài toán sắp xếp tôpô (topological sort)

Thông thường đối với một công việc lớn bao giờ ta cũng phải chia nó thành từng việc nhỏ để thực hiện. Có những phần việc có thể thực hiện được độc lập nhưng cũng có phần việc chỉ thực hiện được khi một số phần việc khác đã được làm xong.

Một sinh viên hàm thụ ngành tin học muốn đạt được một cấp nào đó trong quá trình học tập anh ta sẽ phải hoàn thành một số môn học quy định, chẳng hạn:

Số hiệu môn học	Tên môn học	Môn cần học trước
M_1	Nhập môn tin học	Không
M_2	Giải tích số	M_1, M_{14}
M_3	Cấu trúc dữ liệu và giải thuật	M_1, M_{14}
M_4	Ngôn ngữ Assembly	M_1, M_{14}
M_5	Lý thuyết Otomat	M_{13}
M_6	Trí tuệ nhân tạo	M_3
M_7	Đồ hoạ trên máy tính	M_3, M_1, M_9
M_8	Số học trên máy tính	M_4
M_9	Ngôn ngữ cấp cao	M_3, M_1
M_{10}	Chương trình dịch	M_9
M_{11}	Hệ điều hành	M_{10}
M_{12}	Phân tích và Thiết kế hệ thống	M_3
M_{13}	Đại số tuyến tính	M_{14}
M_{14}	Giải tích	không

Rõ ràng có những môn học như môn "nhập môn tin học" hay "giải tích" có thể hoàn thành độc lập. Nhưng môn "cấu trúc dữ liệu và giải thuật" thì không thể bắt đầu nếu như trước đó sinh viên chưa học "nhập môn tin học" và "giải tích số". Như vậy giữa M_1, M_{14} và M_3 có một quan hệ "được học trước" M_1 "được học trước" M_3, M_{14} "được học trước" M_3 mà một cách tổng quát quan hệ đó được gọi là một "thứ tự bộ phận" (partial order). Đó là một quan hệ giữa các phần tử của một tập S, ký hiệu là $\prec$ đọc là "đứng trước" và quan hệ đó thoả mãn tính chất sau đối với các phần tử phân biệt x, y, z của S:

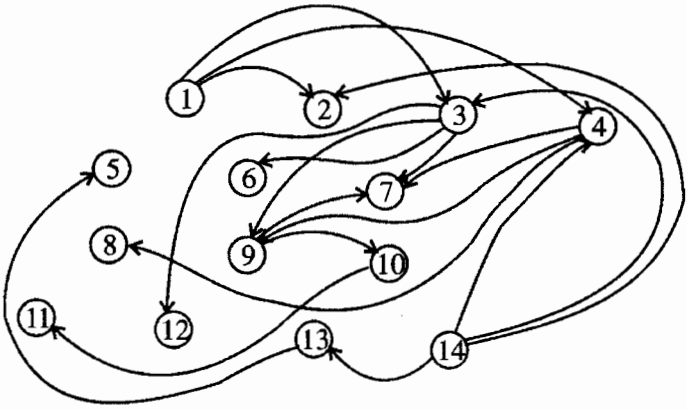
1) Nếu $x \prec y$ và $y \prec z$ thì $x \prec z$ (tính bắc cầu)

2) Nếu $x \prec y$ thì không có $y \prec x$ (tính phản xứng)

3) Không có $x \prec x$ (tính không phản xạ)

Do ý nghĩa thực tế ta luôn giả thiết S là hữu hạn. Một thứ tự bộ phận có thể minh hoạ bằng đồ thị định hướng trong đó các đỉnh ứng với các phần tử, các cung ứng với quan hệ thứ tự. Như ở ví dụ trên: các môn được biểu

diễn bởi các đỉnh và quan hệ "được học trước" được biểu diễn bởi cung định hướng, nghĩa là cung (i, j) thể hiện M_i được học trước M_j. Ta có đồ thị ở hình 7.15.



Hình 7.15

Nếu như một sinh viên, trong một khoảng thời gian nhất định chỉ có thể hoàn thành được một môn học (để kiểm tra và lấy chứng chỉ) thì anh ta sẽ phải sắp xếp các môn học theo một thứ tự tuyến tính để sao cho khi học một môn nào đó, thì các môn cần học trước nó đã hoàn thành rồi. Chẳng hạn anh có thể học theo trình tự:

$$M_1, M_{14}, M_4, M_8, M_{13}\ M_5, M_2, M_3, M_9, M_7\ M_{12}, M_6, M_{10}, M_{11},$$

hoặc

$$M_{14}, M_{13}, M_5, M_1, M_4\ M_8, M_2, M_3, M_9, M_{10}\ M_7, M_6, M_{11}, M_{17}$$

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

Như vậy đối với một đồ thị định hướng, không có chu trình (do tính chất của thứ tự bộ phận, đồ thị biểu diễn nó không có chu trình) thì sắp xếp tôpô là một quá trình định ra một thứ tự tuyến tính cho các đỉnh của đồ thị ấy, sao cho nếu có một cung từ đỉnh i đến đỉnh j, thì i cũng xuất hiện trước j trong thứ tự tuyến tính. Trong thực tế có những yêu cầu liên quan đến sắp xếp tôpô.

Ví dụ đối với một công việc cần làm trên công trường thì bài toán sắp xếp tôpô sẽ là: sắp xếp công việc có liên quan với nhau sao cho mọi công

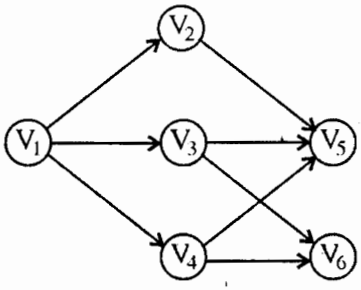
việc được thực hiện thuận lợi theo tiến trình của thời gian cụ thể là việc nào cần làm trước để chuẩn bị cho việc khác thì phải thực hiện trước, chẳng hạn phải xẻ gỗ rồi mới đóng cốt pha, rồi mới trộn bê tông...

Vì vậy sắp xếp tôpô có liên quan tới kỹ thuật PERT dùng trong lý thuyết quy hoạch.

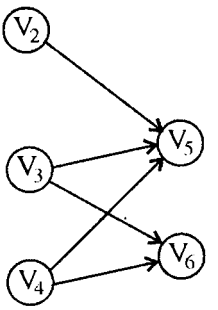
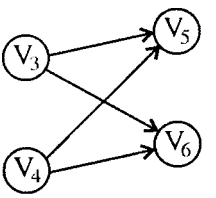
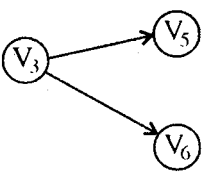
* Có một phương pháp rất đơn giản để thực hiện sắp xếp tôpô.

Ta bắt đầu chọn một đỉnh mà không có cung nào đi tới nó (không có đỉnh nào "trước" nó), đỉnh này sẽ được đưa ra. Sau đó nó cùng các cung xuất phát từ nó sẽ bị loại ra khỏi đồ thị, quá trình được lặp lại với phần còn lại của đồ thị cho tới khi các đỉnh đó được chọn ra hết. Nếu cùng một lúc có nhiều đỉnh đều không có cung tới nó, thì việc chọn một đỉnh nào đó trong số ấy là tuỳ ý.

Ví dụ với đồ thị hình 7.16 thì các bước thực hiện sắp xếp tôpô được thể hiện theo hình 7.17

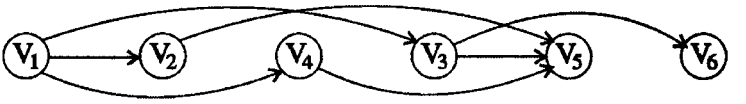


Hình 7.16

Đỉnh được đưa ra	Phần đồ thị còn lại
V_1	
V_2	
V_4	
V_3	
	
V_5	
V_6	

Hình 7.17

Như vậy dãy đưa ra là V_1, V_2, V_4, V_3, V_5, V_6 mà dưới dạng đồ thị có thể biểu diễn



Hình 7.18

Việc chọn cấu trúc dữ liệu cần dựa vào phép xử lý, cụ thể ở đây là các phép:

a) Xác định xem một đỉnh có đỉnh trước nó không

b) Loại một đỉnh cùng với các cung xuất phát từ nó.

Ta thấy phép a) sẽ thuận lợi nếu như ứng với mỗi đỉnh ta giữ được số đếm các đỉnh "trước" nó. Còn phép b) sẽ được thực hiện nếu đồ thị được biểu diễn bởi danh sách lân cận. Và việc loại bỏ các cung xuất phát từ một đỉnh v nào đó có thể thực hiện bằng giảm số đếm đỉnh trước, của các đỉnh thuộc danh sách lân cận của nó. Khi số đếm của đỉnh nào giảm xuống bằng 0, thì đỉnh đó sẽ được xếp vào danh sách các đỉnh có số đếm bằng 0 để sẽ được chọn đưa ra.

Sau đây ta xét cụ thể vào giải thuật.

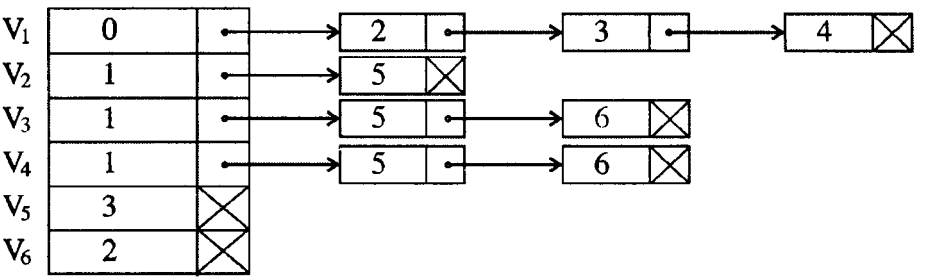
Với giải thuật này, đồ thị coi như được biểu diễn dưới dạng danh sách lân cận. Tuy nhiên, nút đầu danh sách không phải chỉ có một trường mà có hai trường: COUNT và LINK.

Trường COUNT chứa số đếm các cung đi các đỉnh đó (đỉnh trước nó).

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

Còn các nút danh sách thì vẫn có hai trường VERTEX và LINK như đã nêu ở 7.2.2.

Như đồ thị ở hình 7.16 thì danh sách lân cận biểu diễn nó có dạng



Hình 7.19

Danh sách lân cận của đồ thị này được coi như đã tạo lập xong trước khi thực hiện giải thuật. Tất nhiên giá trị của COUNT(j) sẽ được tăng lên 1 mỗi khi một cung (i, j) được đưa vào và nút danh sách tương ứng với đỉnh j cũng được nạp vào danh sách lân cận của i, khi thực hiện việc tạo lập này.

Ở đây danh sách các đỉnh với số đếm COUNT đã bằng 0, sẽ được tổ chức dưới dạng stack (cũng có thể tổ chức dưới dạng queue). Để tiết kiệm người ta dùng luôn trường COUNT ở nút đầu danh sách (ứng với đỉnh mà COUNT bằng 0) để làm trường móc nối stack, vì khi đó trường này không còn cần dùng nữa.

```
Procedure TOPO-ORDER (COUNT, VERTEX, LINK, n)
{ Trong giải thuật này, Top là con trỏ, trỏ tới đỉnh stack}
1) {Khởi tạo stack}
     Top := 0;
2) { tạo lập stack của các đỉnh không có đỉnh trước nó}
     for i := 1 to n do
3)         if COUNT(i) := 0 then begin
                  COUNT(i) := Top
                  Top := i
                                    end;
4) {Đưa các đỉnh ra theo thứ tự Tôpô}
     for i := 1 to n do begin
5)         if Top:=0 then return; {có chu trình trong đồ thị!}
6)         j := Top; Top := COUNT (Top);
           Write (j);
7)         ptr := LINK (j)
8) {giảm số đếm các đỉnh sau j} while ptr ≠ 0 do begin
                                    k := VERTEX (ptr)
                                    COUNT (k) := COUNT(k) — 1;
9) { Nạp vào stack phần tử mới} if COUNT(k) := 0 then begin
                                    COUNT (k) := Top;
                                    Top :=k
                                                      end;
10) ptr := LINK (ptr)
                                                   end
                  end
```

11) **return**

Ta thấy:

Nếu đồ thị có n đỉnh e cung thì chu trình **for** 2) 3) sẽ thực hiện với cấp thời gian là O(n), còn chu trình **while** thì sẽ thực hiện với cây thời gian là $O(d_i)$ với mỗi đỉnh i, với d_i là số các cung xuất phát từ i, mà nó lại nằm trong **for** 4) nên thời gian sẽ là $O(\sum_{i=1}^{n} d_i) = O(e)$. Thời gian thực hiện giải thuật trên có cấp O(n + e) tuyến tính với kích thước của đồ thị.

7.5 Cấu trúc đa danh sách (multilist)

Một cách tổng quát thì "đa danh sách" là một cấu trúc bao gồm nhiều nút mà mỗi nút có nhiều mối nối và có thể là phần tử đồng thời của nhiều danh sách. Tất nhiên đối với nút như vậy thì vấn đề khá quan trọng là phải phân biệt được các mối nối để lần vào từng danh sách cần thiết mà không nhầm sang danh sách khác.

Với cấu trúc đa danh sách, ở đây ta chỉ xét cụ thể trong một phạm vi hẹp: đa danh sách để biểu diễn ma trận thưa.

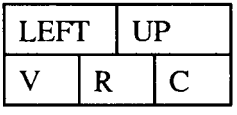
Như ta đã biết, với ma trận thưa nếu ta lưu trữ kế tiếp như ma trận thông thường thì rất lãng phí (xem chương 3). Tất nhiên cũng có cách lưu trữ kế tiếp ma trận thưa một cách tiết kiệm, chẳng hạn chỉ lưu trữ các phần tử khác không, kèm theo chỉ số hàng, chỉ số cột của nó dưới dạng 3 vectơ.

Tuy nhiên, lưu trữ kế tiếp vẫn mắc phải nhược điểm cố cựu của nó là phép bổ sung hay loại bỏ phần tử sẽ không thực hiện được thuận lợi, mà điều này thì không tránh khỏi vì khi tác động các phép toán trên các ma trận thưa đó vẫn có thể có phần tử mới khác không, xuất hiện ở hàng cột mới cũng như sự triệt tiêu của phần tử ứng với hàng cột cũ.

Với cách tổ chức theo cấu trúc đa danh sách, ta không hề gặp khó khăn gì khi xuất hiện các tình huống này.

Nguyên tắc tổ chức như sau:

Mỗi nút của cấu trúc, ứng với một phần tử khác không của ma trận và có quy cách



Trường LEFT: chứa con trỏ, trỏ tới nút tiếp theo trong danh sách nối vòng ứng với hàng (mà ta gọi là danh sách hàng). Ở đây nút tiếp theo là nút có chỉ số cột nhỏ hơn.

Trường UP: Chứa con trỏ, trỏ tới nút tiếp theo trong danh sách nối vòng ứng với cột (mà ta gọi là danh sách cột). Ở đây nút tiếp theo là nút có chỉ số hàng nhỏ hơn.

Trường V: ghi giá trị khác không của phần tử

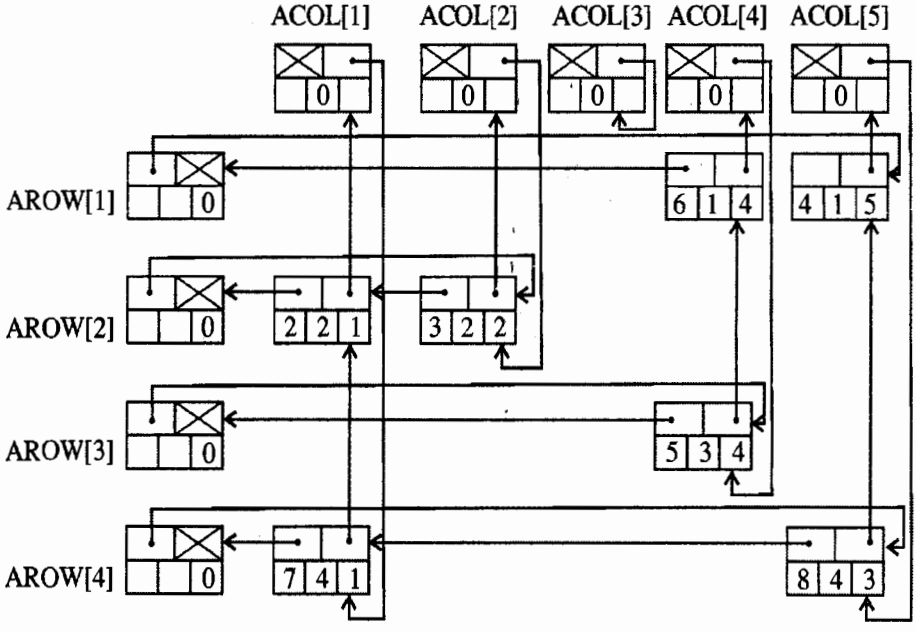
R: ghi chỉ số hàng của phần tử

C: ghi chỉ số cột của phần tử.

Với ma trận

$$\begin{bmatrix} 0 & 0 & 0 & 6 & 4 \\ 2 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 7 & 0 & 0 & 0 & 8 \end{bmatrix}$$

thì đa danh sách biểu diễn nó có dạng ở hình 7.20:



Hình 7.20

Mỗi phần tử ở đây là chung một danh sách hàng và một danh sách cột. Mỗi danh sách đều có nút đầu danh sách. Nút đầu danh sách hàng i có giá trị 0 ở trường C và được trỏ bởi AROW[i], nút đầu danh sách cột j có giá trị

0 ở trường R và được trỏ bởi ACOL[j], AROW và ACOL đều là các vectơ mà phần tử là các con trỏ. Các danh sách hàng và danh sách cột được tổ chức theo kiểu danh sách nối vòng. Với quy định đối với con trỏ LEFT và UP đã nêu ở trên ta thấy trong danh sách hàng nút được móc nối theo chiều giảm của chỉ số cột, còn trong danh sách cột thì lại theo chiều giảm của chỉ số hàng. Thuận lợi của quy cách này sẽ thấy trong các giải thuật tạo lập ma trận và nhân hai ma trận nêu dưới đây.

Procedure CONSTRUCT MATRIX (A)

{Thủ tục này thực hiện việc tạo lập một đa danh sách biểu diễn ma trận thưa A với các nút có quy cách đã nêu ở trên. Ở đây hai vectơ AROW và ACOL sẽ chứa con trỏ trỏ tới các nút đầu tiên của danh sách. Các biến ROW, COLUMN và VALUE để chứa các giá trị tương ứng R, C, V của các nút khi đưa vào. Giả sử kích thước m x n của ma trận và số lượng k các phần tử khác không đã biết}

```
1) {khởi tạo danh sách hàng, danh sách cột}
   for i: = 1 to n do begin
            Call New (AROW[i]);
            C(AROW[i]) : = 0;
            LEFT(AROW[i]) := AROW[i]
                  end;
   for i:= 1 to n do begin
            Call New (ACOL[i]);
            R(ACOL[i]) : = 0;
            UP(ACOL[i]) := ACOL[i]
                  end;
2) {Tạo các danh sách}
   for i:= 1 to k do begin
3) {Nhập phần tử khác không}
   read (VALUE, ROW, COLUMN)
4) {Khởi tạo nút}
   call New (P);
   V(P) := VALUE;
   R(P) := ROW;
   C(P) := COLUMN;
5) {Định vị trong danh sách hàng cho nút mới}
   Q := AROW[R(P)];
   while C(P) < C(LEFT(Q)) do Q := LEFT(Q);
```

```
            LEFT(P) := LEFT(Q);
            LEFT(Q) := P;
6) {Định vị trong danh sách cột cho nút mới}
            Q := ACOL[C(P)];
    while R(P) < R(UP(Q)) do Q := UP(Q);
            UP(P) :=UP(Q);
            UP(Q) := P
                    end
7)  return
```

Procedure MATRICES MULTI (A, B, C)

{Cho các vectơ AROW, ACOL, BROW, BCOL trỏ tới đa danh sách biểu diễn ma trận thưa A và B với kích thước lần lượt là mxn và nxt. Thủ tục này thực hiện tính tích C của A và B: C = A x B. Các vectơ con trỏ CROW, CCOL được dùng tương ứng cới ma trận C mà kích thước là m x t. Các biến i và j được dùng để đếm hàng của ma trận A và cột của ma trận B. Các biến A và B là biến trỏ dùng để lần vào hàng của ma trận A và cột của ma trận B}

```
1) {khởi tạo danh sách hàng, danh sách cột của C}
    for i:= 1 to m do begin
            call new (CROW[i]);
            C(CROW[i]) := 0;
            LEFT(CROW[i]) := CROW[i];
                    end;
    for j:= 1 to t do begin
            call new (CCOL[j]);;
            R(CCOL[j]) := 0;
            UP(CCOL[j]) := CCOL[j];
                    end;
2) {Thực hiện với m hàng của ma trận A}
    for i: = 1 to m do
3) {Thực hiện với t cột của ma trận B}
    for j: = 1 to t do begin
4) {Bắt đầu vào hàng i của ma trận A, cột j của ma trận B}
    A := LEFT(AROW[i]);
    B := UP(BCOL[j]);
    PRODUCT := 0;
```

```
5) {Xác định các phần tử tương ứng và thực hiện phép nhân}
   while R(B) ≠ 0 and C(A) ≠ 0 do
         if C(A) > R(B) then A:= LEFT(A)
                        else if R(B) > C(A)
                  then B := UP(B)
                  else begin
                        PRODUCT:=PRODUCT+V(A)*V(B);
                        A := LEFT(A);
                        B := UP(B)
                     end;
6) {Nếu kết quả khác không thì bổ sung vào C}
                  if PRODUCT ≠ 0 then begin
                        call new(P);
                        V(P) := PRODUCT
                        R(P) := i;
                        C(P) := j;
                        LERT(P) := LEFT(CROW[i]);
                        UP(P) := UP(CCOL[j]);
                        LEFT(CROW[i]) := P;
                        UP(CCOL[j] := P
                                          end;
                  end;
         end;
7) return
```

7.6 Danh sách tổng quát (generalized list) hay cấu trúc danh sách (list structures)

Ta xét tới trường hợp mở rộng của một danh sách tuyến tính, đó là *danh sách tổng quát* hay còn gọi là *cấu trúc danh sách* (tuy nhiên để cho gọn có khi ta chỉ gọi là danh sách).

7.6.1 Định nghĩa

Danh sách tổng quát A là một dãy hữu hạn của $n \geq 0$ phần tử $a_1, a_2, ..., a_n$, trong đó a_i là một nút (mà ta gọi là nguyên tử (atom)) hoặc là một danh sách (mà ta gọi là *danh sách con* (sublist)).

Danh sách tổng quát A được ký hiệu bởi

$$A = (a_1, a_2, \ldots, a_n)$$

n được gọi là độ dài của danh sách, A được coi là tên của danh sách, ta quy ước tên này được ký hiệu bởi chữ cái hoa. Nếu $n \geq 1$ thì a_1 gọi là *đầu* (head) của A, $(a_2, a_3, ..., a_n)$ gọi là *đuôi* (tail) của A.

Ví dụ:

1) D = (0) D là danh sách rỗng có độ dài 0
2) A = (a, (b, c), d, (e, f, g)) A là danh sách có độ dài 4 với a, d là nguyên tử (b, c), (e, f, g) là các danh sách con.
3) B = (A, A, (())) B là danh sách có độ dài 3 với A là danh sách con, (()) là danh sách rỗng
4) C = (a, C) C là danh sách đệ quy có độ dài 2

C = (a, (a, (a,...)). C tương ứng với một danh sách vô hạn

Đối với danh sách A "đầu" của nó là a, "đuôi" của nó là (b, c), d, (e, f, g) còn với B thì "đầu" là A, "đuôi" là (A, ())

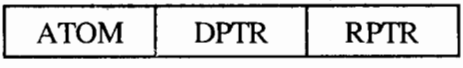
Chú ý: Với cách mở rộng thế này ngoài tính chất đệ qui, còn có cả tính chất "dùng chung" ở danh sách tổng quát, nghĩa là một danh sách có thể đồng thời là danh sách con của nhiều danh sách tổng quát.

7.6.2 Biểu diễn danh sách tổng quát

Rõ ràng cách biểu diễn móc nối là thuận tiện cho danh sách tổng quát. Nó tạo ra khả năng phân bố động của các nút khi cần thiết, dễ dàng vận hành và có thể dùng chung các danh sách con.

Có nhiều qui cách được định ra cho một nút khi tạo dựng danh sách tổng quát. Sau đây ta sẽ xét tới một trong các qui cách ấy.

Mỗi nút gồm có 3 trường



Trường ATOM: trường một bit để đánh dấu phân biệt nút ứng với nguyên tử hay nút ứng với danh sách con.

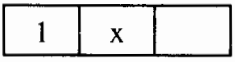
Nếu ATOM = 1 là ứng với nút nguyên tử, lúc đó DPTR là con trỏ, trỏ tới nơi lưu trữ thông tin của nguyên tử đó.

ATOM = 0 là ứng với nút danh sách con, lúc này DPTR trỏ tới phần tử đầu tiên của danh sách con này.

RPTR là con trỏ trỏ tới phần tử tiếp theo của danh sách tổng quát.

Quy cách này thuận lợi cho trường hợp mà thông tin của nguyên tử không có kích thước như nhau, và khá lớn.

Với quy cách này nếu một nút có dạng

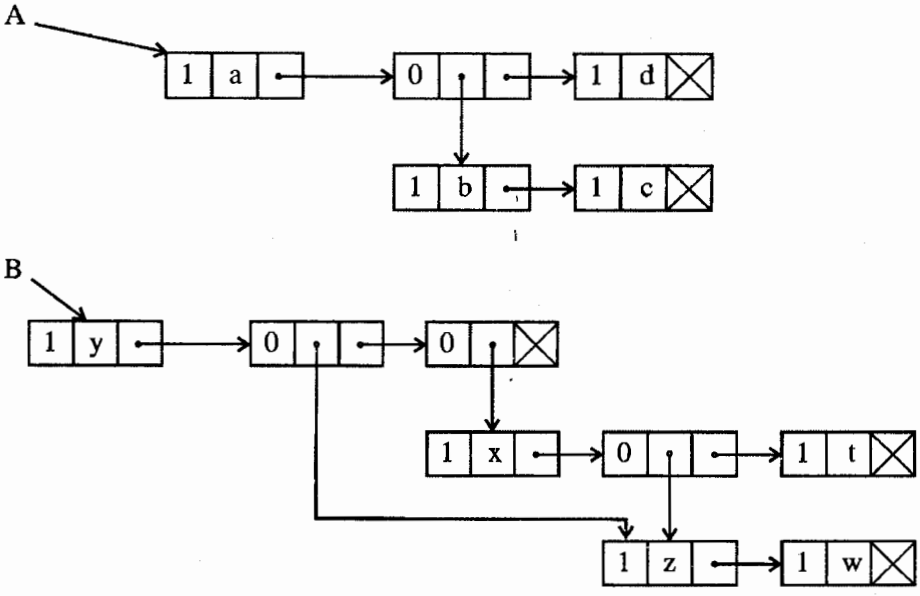


thì ta sẽ hiểu đây là nút nguyên tử mà x là con trỏ trỏ tới chỗ lưu trữ thông tin của nó.

Hình 7.21 cho ta cách biểu diễn danh sách tổng quát

1) A = (a, (b, c), d)

2) B = (y, (z, w), (x, (z, w), t)

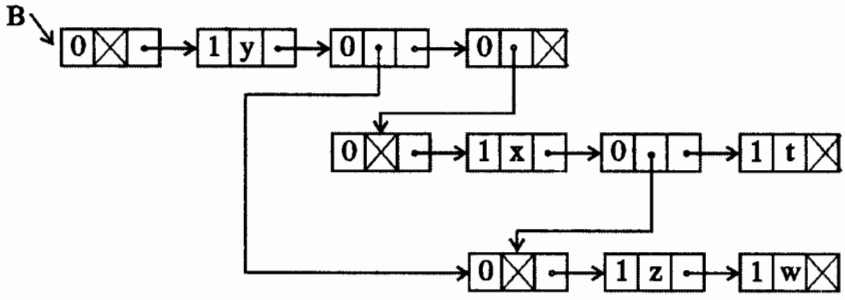


Hình 7.21

Bây giờ ta thử xem xét điều gì đã xảy ra nếu ta muốn loại bỏ nút z ra khỏi danh sách con (z, w) của danh sách B, nghĩa là ta muốn tạo nên một danh sách mới (y, (w), (x, w, t)).

Tất nhiên ta phải thay đổi hai con trỏ hiện đang trỏ tới z. Công việc này không đơn giản vì ta không thể lần ngược lên để biết được nút đang chứa hai con trỏ ấy mà phải dò "từ trên xuống".

Nếu ta đưa thêm vào "nút đầu danh sách" với quy cách tương tự như các nút khác, nhưng ATOM bằng O và DPTR của nó thì bằng null. Nếu danh sách rỗng thì cả DPTR và RPTR đều null. Như vậy danh sách B nêu trên sẽ được biểu diễn như hình 7.22.

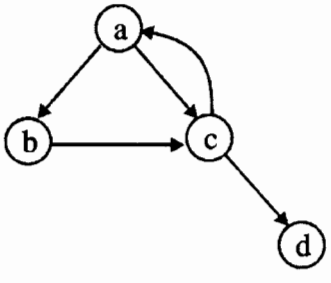


Hình 7.22

Rõ ràng, với cách tổ chức này khi loại bỏ nút z hai con trỏ đang trỏ tới danh sách con (z, w) không hề bị ảnh hưởng gì. Do đó với các danh sách, việc đưa thêm vào nút đầu danh sách thường được sử dụng.

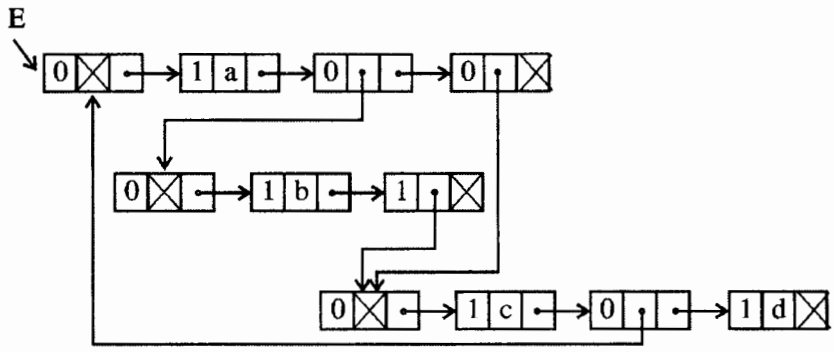
Danh sách tổng quát cũng là cấu trúc được dùng để biểu diễn đồ thị. Chẳng hạn với đồ thị ở hình 7.23 ta có thể coi như danh sách tổng quát có dạng

$$E = ((a, (b, (c, (a), d)), (c, (a), d)$$



Hình 7.23

và có thể biểu diễn như ở hình 7.24



Hình 7.24.

7.6.3. Một số giải thuật xử lý danh sách tổng quát

Trước hết ta xét tới giải thuật tạo lập một danh sách tổng quát khi cho biết dạng biểu diễn bằng dấu ngoặc (như đã nêu ở 7.6.1.) của nó.

Ở đây ta giả sử dãy ký tự đưa vào không có khoảng trống (blank), tên của nút là một chữ in hoa và danh sách không có chu trình cũng như phần tử dùng chung.

Funtion CREATE (INPUT)

{ Cho INPUT là một xâu ký tự ứng với dạng biểu diễn bằng dấu ngoặc của danh sách. Thủ tục này thực hiện việc tạo nên danh sách tổng quát, có các nút với quy cách như đã nêu, và cho biết con trỏ trỏ tới nút đầu danh sách đó. Ở đây có sử dụng một stack S để thực hiện việc tạo lập cấu trúc của danh sách, đối với S sẽ sử dụng hai thủ tục PUSH và POP như đã nêu trước đây, CURSOR là một biến nguyên dùng để ghi nhận vị trí của ký tự đang xét trong sâu INPUT. Hàm SUB(A, m, n) cho một xâu con trong xâu A có độ dài n kể từ thứ tự m. P và Q là các biến trỏ}

```
1. {Tạo nút đầu danh sách}
   if SUB(INPUT, 1, 1) = '('
         then begin call new(P);
                    RPTR(P) := DPTR(P) := null;
                    ATOM(P) := 0;
                    call PUSH(S, TOP, P)
                end
```

```
            else begin write ("không đúng quy cách")
            return (null)
            end
2. {lặp lại cho tới khi toàn bộ xâu INPUT đã được xử lý}
   CURSOR := 1;
   while CURSOR < LENGTH (INPUT) do
                    {hàm LENGTH cho độ dài xâu}
3. begin
            CURSOR := CURSOR + 1;
   case
   SUB(INPUT,CURSOR,1) = '(':  call new(Q); {danh sách con}
                               RPTR(Q) := null;
                               ATOM(Q) := 0;
                               RPT (P) : = Q;
                               call PUSH(S, TOP, Q);
                               call new (P);
                               RPTR(P) := DPTR(P) := null;
                               ATOM(P) := 0;
                               DPTR(Q) := P;
   SUB(INPUT, CURSOR, 1) = '(': call POP(S, TOP, P);
   SUB(INPUT, CURSOR, 1) = ",": {không làm gì}
   SUB(INPUT, CURSOR, 1) = 'A' to 'Z':
                                    call new (Q)
                                    RPTR(Q) := null;
                                    ATOM(Q) := 1;
                                    DPTR(Q) :=SUB(INPUT,
                                     CURSORR, 1);
                                    RPTR(P) := Q;
                                    P := Q;
            else write ('ký tự không đúng quy cách");
            return (null);
   end case;
   end;
4. return (P)
```

Tiếp theo đây là các thủ tục xác định "đầu" và "đuôi" của một danh sách tổng quát

Function HEAD(ROOT)

{Cho ROOT là con trỏ trỏ tới nút đầu danh sách của một danh sách tổng quát, hàm này sẽ cho con trỏ tới "đầu" của danh sách tổng quát này. Nếu "đầu" là một nút nguyên tử thì con trỏ này trỏ tới chỗ chứa thông tin của nguyên tử đó, P là một biến trỏ}

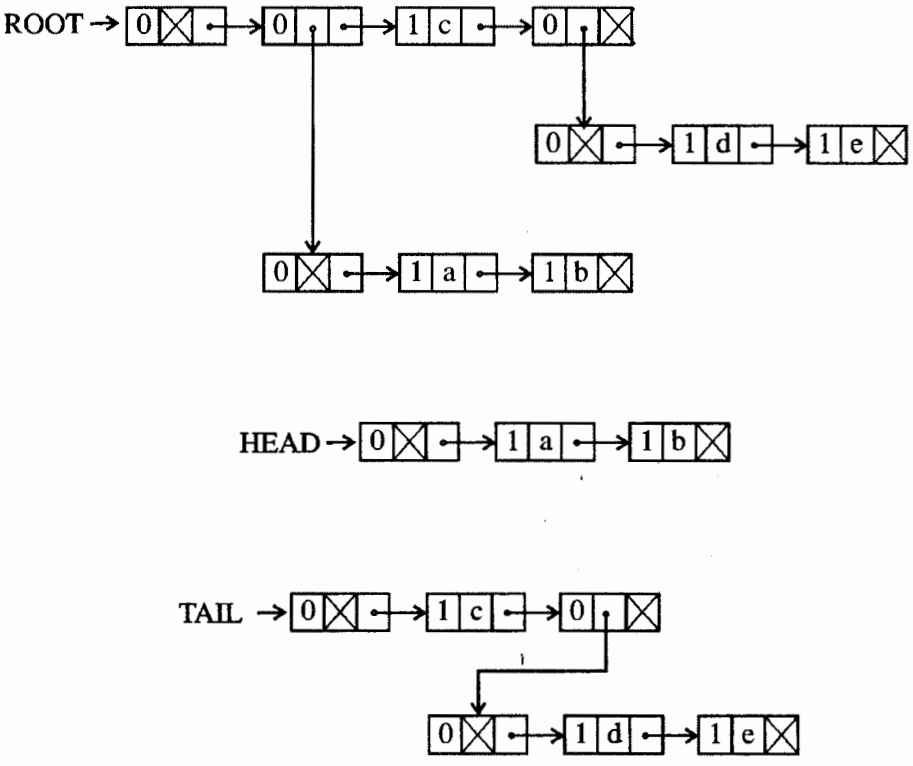
```
1. {Kiểm tra quy cách của danh sách}
   if ROOT = null or DPT(ROOT) ≠ null then
            begin
            {write ("không đúng quy cách"};
            return (null)
            end;
2. { Danh sách rỗng}
   P := RPTR(ROOT);
   if P = null then begin {write ("danh sách rỗng'')};
            return (null)
                  end;
3. {Cho con trỏ, trỏ tới "đầu"}
   return (DPTR(P))
```

Function TAIL(ROOT)

{Hàm này thực hiện việc tạo nên một nút đầu danh sách và cho địa chỉ nút đó. Trường RPTR của nút đầu danh sách này sẽ trỏ tới đuôi của danh sách tổng quát trỏ bởi ROOT}

```
1. {Kiểm tra qui cách của danh sách}
   if ROOT = null or DPTR(ROOT) ≠ null then
            begin
                  {write ("không đúng qui cách");}
                  return (null)
            end;
2. {Danh sách rỗng}
   P := RPTR(ROOT);
   if P = null then begin
            {write ("Danh sách rỗng");}
            return (null)
                  end;
```

3. {Tạo nút đầu danh sách và "đuôi"}
   call new (Q);
   ATOM (Q) := 0; DPTR(Q) := null;
   RPTR (Q) := RPTR(P);
   return (Q)



Hình 7.25

* Phép xử lý tạo ra kết quả ngược với phép HEAD và TAIL đó là phép dựng danh sách. Cho hai đối tượng: đối tượng 1 là nguyên tử hoặc một danh sách, đối tượng 2 là một danh sách. Hãy dựng lên một danh sách tổng quát nhận đối tượng 1 làm "đầu" và đối tượng 2 làm "đuôi". Ví dụ nếu A là một danh sách (q) và B là một danh sách (d, e) thì phép dựng A và B sẽ cho một danh sách ((q), d, e). Nếu A là một nguyên tử s và B là một danh sách (t, u) thì dựng A và B sẽ cho (s, t, u). Chú ý rằng qua phép dựng này mức của các phần tử thuộc đối tượng 1 sẽ tăng lên 1, mức của các phần tử thuộc đối tượng 2 vẫn giữ nguyên.

Procedure CONSTRUCT(A, B)

{Cho A là con trỏ trỏ tới nút B đầu danh sách của một danh sách tổng quát hoặc một nguyên tử và B là con trỏ trỏ tới một danh sách tổng quát khác. Thủ tục này thực hiện dựng lên một danh sách mới L mà HEAD(L) = A, TAIL(L) = B. Nút đầu danh sách của B trở thành nút đầu danh sách mới}

```
1. {Kiểm tra qui cách}
   if A = null or B = null then
            begin
                     write ("không đúng qui cách");
                     return
            end;
2. {B có là danh sách không?}
   if DPTR(B) ≠ null then
            begin
                     write ("B không hợp qui ước");
                     return
            end;
3. {Ghép A vào B}
   if ATOM(A) = 1 then
            begin
                     P := RPTR(B); { A là nguyên tử}
                     RPTR(B) :=A;
                     RPTR(A) := P
            end;
                     else
            begin { A là một danh sách }
                     call new (Q);
                     P := RPTR(B);
                     RPTR(B) := Q;
                     RPTR(Q) :=P;
                     DPTR(Q) := A;
            end
4. return
```

*Giải thuật cuối cùng đối với danh sách móc nối tổng quát là ứng dụng với phép đếm số lượng các nguyên tử trong một danh sách. Ví dụ danh sách (a, b, (c, d), e) có 5 nguyên tử còn danh sách ( ) thì không có nguyên tử nào.

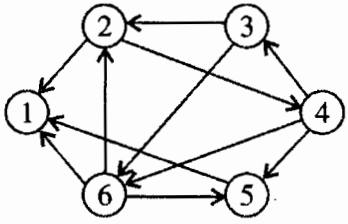
Procedure COUNTER (A, COUNT)

{Cho A là con trỏ, trỏ tới nút đầu danh sách, thủ tục đệ qui này sẽ đếm số lượng nguyên tử có trong danh sách đó và cho kết quả đưa ra bởi COUNT}

1. **while** A ≠ null **do begin**
2. {Xử lý một nút và lần vào danh sách}
   if ATOM(A) = 1 **then** COUNT := COUNT + 1
   else
   if DPT(A) ≠ null **then**
   call COUNTER (DPTR(A); COUNT);
   A := RPTR(A)
   end;
3. **return**

BÀI TẬP CHƯƠNG 7

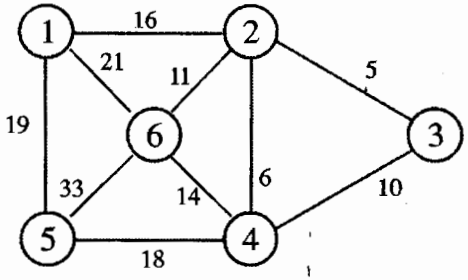
7.1. Cho đồ thị định hướng G_1



Hãy lập a) Ma trận lân cận của G_1

b) Danh sách lân cận của G_1

7.2. Cho đồ thị không định hướng G_2 có trọng số



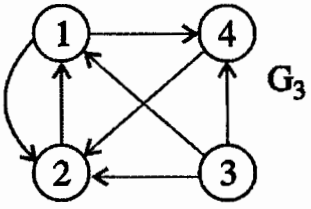
a) Hãy dựng cây khung theo chiều sâu của G_2

b) Hãy dựng cây khung theo chiều rộng của G_2

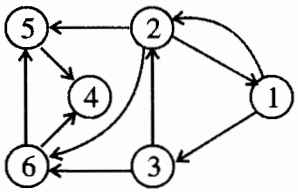
c) Dựa vào giải thuật Kruskal, nêu các bước dựng cây khung với giá trị cực tiểu của G_2.

7.3. Cho đồ thị G_3

Hãy lập ma trận lân cận A và tính $A^{(2)}$, $A^{(3)}$, $A^{(4)}$, P. Vẽ đồ thị ứng với P



7.4. Cho đồ thị có trọng số: G_4



G_4

Hãy lập ma trận lân cận COST tương ứng với đồ thị và dựa vào giải thuật SHORTEST-PATH lập bảng biểu diễn các tác động của giải thuật qua các bước xác định đường đi ngắn nhất từ đỉnh 1 tới các đỉnh khác của đồ thị.

7.5. Quan hệ "đứng trước" ($\prec$) dưới đây có thể coi là một thứ tự bộ phận đối với các phần tử được đánh số từ 1 đến 5 hay không? tại sao?

$$① \prec ②, ② \prec ④, ② \prec ③, ③ \prec ④, ③ \prec ⑤, ⑤ \prec ①$$

7.6. Một đội thợ phải thực hiện 7 công việc kế tiếp nhau. Các công việc này được đánh số từ 1 đến 7. Giữa chúng có quan hệ "làm trước", mà ký hiệu bằng $\prec$, cụ thể là

$2 \prec 1$ (công việc 2 làm trước công việc 1)

$2 \prec 6$

$3 \prec 6$

$4 \prec 2$

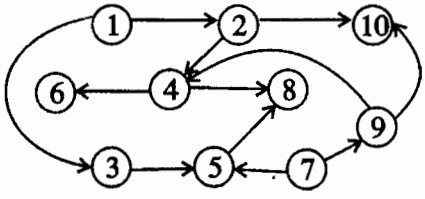
$5 \prec 3, 5 \prec 6$

$7 \prec 5$

a) Hãy biểu diễn tập các tập công việc này bằng đồ thị.

b) Hãy áp dụng quy tắc sắp xếp Tôpô để đưa ra một vài thứ tự Tôpô khác nhau.

7.7. Các công việc và quan hệ "làm trước" giữa chúng được biểu diễn bởi đồ thị sau:



Người ta áp dụng giải thuật TOPO ORDER để thực hiện sắp xếp Tôpô.

a) Hãy minh hoạ cấu trúc lưu trữ của đồ thị trên ứng với giải thuật đó.

b) Minh hoạ tình trạng của stack được sử dụng, qua từng bước thực hiện giải thuật, kèm theo những thuyết minh cần thiết.

c) Cuối cùng các công việc này đã được sắp xếp theo thứ tự nào?

7.8. Cho ma trận thưa

$$\begin{bmatrix} 0 & 0 & 11 & 0 & 0 & 13 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 14 \\ 0 & -4 & 0 & 0 & 0 & -8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & -9 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Dùng hình vẽ minh hoạ cấu trúc đa danh sách biểu diễn ma trận này, theo quy ước như đã nêu trong bài.

7.9. Lập giải thuật thực hiện phép cộng hai ma trận thưa kích thước m x n, được biểu diễn bởi cấu trúc đa danh sách.

7.10. Minh hoạ cách biểu diễn các danh sách tổng quát sau đây, theo qui cách như đã nêu trong bài:

(a, (b, (c, d)), e, f)

((x), y, A, z) với A = (a, b, (c, d))

7.11. Hãy cho một ví dụ và dùng hình vẽ minh hoạ việc tạo lập một danh sách tổng quát qua các bước thực hiện thủ tục CREATE.

7.12. Cho hai con trỏ A và B lần lượt trỏ tới hai danh sách tổng quát. Hãy lập thủ tục thực hiện ghép hai danh sách đó sao cho:

- Mức của các nút trên danh sách được giữ nguyên như cũ.
- Các phần tử của danh sách A đứng trước các phần tử của B.
- Nút đầu danh sách của A trở thành nút đầu danh sách chung.

7.13. Lập một giải thuật đệ qui thực hiện việc sao chép một danh sách tổng quát trỏ bởi con trỏ L (giả sử danh sách này không đệ qui mà không có danh sách con dùng chung) và cho ra con trỏ trỏ tới danh sách sao này. Giả sử quy cách và cách tổ chức danh sách giống như 7.21. (không có nút đầu danh sách).

Chương 8

QUẢN LÝ BỘ NHỚ

8.1 Các vấn đề phát sinh trong quản lý bộ nhớ

Khi đặt vấn đề quản lý bộ nhớ (memory management) "có giới hạn" để đảm bảo thoả mãn cho nhiều nhu cầu sử dụng, thậm chí "cạnh tranh" nhau, sẽ có rất nhiều tình huống đòi hỏi phải xử lý kịp thời. Đối với người lập trình, không hề tham gia vào việc cài đặt các chương trình hệ thống (như chương trình dịch, hệ điều hành.v.v...) thì không nhận thức được tình huống này vì họ là những người "đứng ngoài cuộc". Chẳng hạn khi viết chương trình PASCAL, lập trình viên chỉ nhận thức được là thủ tục new(p) sẽ tạo một con trỏ p, trỏ tới một nút mới để đem ra sử dụng chứ không hề biết rằng nút mới đó từ đâu mà ra? Hoặc làm sao mà biết được trong một miền nhớ cho phép còn có nút trống chưa sử dụng,như vậy để đưa ra.

Trong chương 4 và 5 ta cũng có đề cập sơ bộ tới vấn đề này. Bây giờ ta sẽ đi sâu hơn một chút.

Trước đây ta chỉ nói tới trường hợp các nút có kích thước cố định, nghĩa là ứng với các trường hợp các nhu cầu về nút nhớ đồng đều nhau. Nhưng trong thực tế còn có những nhu cầu về không gian nhớ với kích thước bất kỳ. Ta sẽ thấy yêu cầu khác nhau này qua một số trường hợp cụ thể sau đây:

1) Trong ngôn ngữ LISP (ngôn ngữ xử lý danh sách móc nối) không gian nhớ được chia thành từng "ô" (cell) mỗi ô có hai trường cùng kích thước (như vậy nghĩa là các ô cũng cùng kích thước). Mỗi trường ở đây có thể lưu trữ một dữ liệu nguyên tử (atom) hoặc một con trỏ. Tất cả các cấu trúc dữ liệu có thể được tạo lập từ các ô như thế. Ví dụ: với danh sách móc nối thì trường thứ nhất chứa dữ liệu nguyên tử, trường thứ hai chứa con trỏ tới nút tiếp theo. Với cây nhị phân thì trường thứ nhất chứa mối nối trái, trường thứ hai chứa mối nối phải. Khi mà một chương trình LISP được thực hiện thì các ô nhớ như vậy ở từng thời điểm có thể nằm ở những cấu trúc danh sách khác nhau, vì hoặc bản

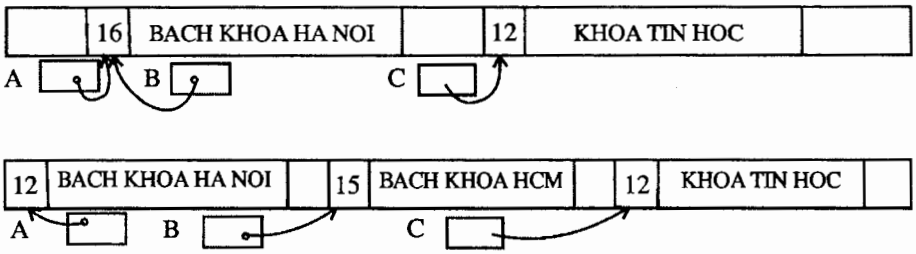
thân một ô cũng có thể di động giữa các cấu trúc (ví dụ nó được tách ra, ghép vào với cấu trúc khác) hoặc ô nhớ đó bị loại khỏi cấu trúc này và được dùng lại cho nhu cầu của cấu trúc khác.

2) Một hệ quản lý tệp thường chia bộ nhớ ngoài, chẳng hạn đĩa từ, thành các khối (block) có kích thước cố định. Ví dụ UNIX dùng các khối chuẩn với kích thước 512 byte. Tệp được lưu trữ trong một dãy các khối (không nhất thiết phải kế tiếp) khi tệp bị huỷ các khối sẽ được dùng lại.

3) Một hệ điều hành đa chương trình (multipropgramming) thường cho phép các chương trình được chia sẻ bộ nhớ cùng một lúc. Mỗi chương trình đều có đòi hỏi riêng về không gian nhớ. Hệ điều hành sẽ phải nắm bắt được yêu cầu này để thực hiện phân bố bộ nhớ khi chương trình được thực hiện. Chẳng hạn, khi một chương trình 100 kbyte kết thúc thì có thể thay thế bởi hai chương trình 50 kbyte hoặc một chương trình 20 kbyte và một chương trình 30 kbyte (còn thừa 50 kbyte không dùng đến). Cũng có thể 100 kbyte vừa được trả lại bởi một chương trình đã kết thúc phải ghép với 50 kbyte kế cận, còn chưa sử dụng, để đáp ứng cho một chương trình cần 150 kbyte. Tất nhiên cũng có thể không có điều kiện để làm được như vậy và lúc đó 100 kbyte đành để không mà chương trình cần chạy khi chưa thực hiện được vì chưa đủ không gian nhớ cho nó.

4) Cũng có một số ngôn ngữ như SNOBOL, APL... lại phân bố cho các đối tượng có kích thước bất kỳ. Các đối tượng này có thể là giá trị gán cho các biến, được lưu trữ trong các khối thuộc một khối lớn hơn của không gian nhớ mà thường gọi là *"đống"* (heap) khi một biến thay đổi giá trị, thì một giá trị mới sẽ chiếm một chỗ trong đống và một con trỏ ứng với biến đó sẽ được thiết lập để trỏ tới chỗ ấy. Cũng có thể giá trị cũ của biến bây giờ không dùng đến nữa và không gian của nó có thể được tái sử dụng. Tuy nhiên, ngôn ngữ như SNOBOL chẳng hạn thường cài đặt phép gán, ví dụ A = B, bằng cách tạo con trỏ ứng với A trỏ tới cùng một đối tượng mà con trỏ B đang trỏ tới. Nếu chỉ A hoặc B được gán giá trị mới thôi thì đối tượng đó vẫn chưa được giải phóng và trong trường hợp đó chỗ trống ứng với nó chưa thể lấy lại để dùng cho yêu cầu khác được.

Ví dụ: Trong hình 8.1 ta có hình ảnh một "đống" được sử dụng bởi một chương trình SNOBOL với ba biến A, B và C. Giá trị của biến trong SNOBOL là xâu ký tự và ở đây cả hai biến A và B đều có giá trị là "BACH KHOA HA NOI". Còn C thì có giá trị là "KHOA TIN HOC".



Hình 8.1

Các xâu ký tự được xác định qua con trỏ, trỏ tới các khối nhớ trong "đống". Các khối này có hai byte đầu tiên (đây chỉ là một quy định đặt ra cho tiện, tất nhiên có thể không nhất thiết phải là 2 byte) dành ra để ghi nhận một số nguyên thể hiện độ dài của xâu. Chẳng hạn "BACH KHOA HA NOI" có độ dài 16 (kể cả khoảng trống giữa các từ) như vậy giá trị của A và B chiếm một khối 18 byte.

Nếu giá trị của B đổi thành "BACH KHOA HCM" thì phải tìm một khối nhớ khác trong "đống" có độ dài 15 byte để lưu trữ giá trị mới này. Lúc đó con trỏ ứng với B phải trỏ tới khối mới ấy còn khối cũ chứa "BACH KHOA HA NOI" vẫn chưa "bỏ" được vì phải dùng cho giá trị của A. Nếu giá trị của A cũng biến đổi thì khối đó mới trở thành "trống" và mới được tái sử dụng cho yêu cầu khác.

Qua 4 trường hợp nêu trên ta thấy nổi lên hai vấn đề:

1) Độ dài của các phần tử nhớ là như nhau hay khác nhau (kích thước cố định hay kích thước biến đổi). Hai ví dụ đầu là rơi vào trường hợp kích thước cố định. Điều này cũng tạo ra một số khả năng đơn giản hơn trong vấn đề quản lý bộ nhớ. Chẳng hạn: trong cài đặt LISP, miền nhớ sẽ được chia thành từng phần ứng với từng "ô" và việc cấp phát hay thu hồi là ứng với từng ô ấy cũng như đối với trường hợp của ví dụ 2, đĩa từ sẽ được chia thành từng khối có kích thước như nhau, mỗi khối như vậy sẽ được dùng để lưu trữ các phần của tệp, không bao giờ một khối lại chứa hai phần của hai tệp khác nhau, ngay cả khi một tệp đã kết thúc ở giữa khối (nghĩa là vẫn còn một phần khối đó không dùng hết).

Trái với trên, hai ví dụ sau lại ứng với miền nhớ có kích thước biến đổi. Điều này sẽ làm xuất hiện những vấn đề mà trong trường hợp kích thước cố định không hề có. Chẳng hạn: tình huống chia cắt nhỏ vùng nhớ (hay còn gọi là "cắt đoạn" - fragmentation) nghĩa là sau một số yêu cầu cấp phát những chỗ trống "còn thừa" còn lại khá nhiều nhưng kích thước hoặc nhỏ, hoặc nằm phân tán (không ghép lại thành khối lớn được) nên không sử dụng được cho những yêu cầu mới, lớn hơn. Như vậy là vẫn còn chỗ trống mà đành chịu không cấp phát được.

2) Thu hồi các "chỗ không dùng đến nữa" được thực hiện theo cách tường minh hay ẩn dụ nghĩa là bằng lệnh của chương trình hay bằng những giải pháp đặc biệt khác của hệ thống. Trong trường hợp hệ quản lý tệp, khi một tệp nào đó bị loại bỏ đi thì hệ quản lý sẽ biết được và nó sẽ chủ động đưa các khối ứng với tệp đó về "đống" để dành cho yêu cầu tái sử dụng sau này. Nhưng đối với các "ô" của LISP thì không phải như vậy. Do cấu trúc dữ liệu của LISP là cấu trúc danh sách tổng quát nên có tình trạng một "ô" được dùng trong nhiều danh sách (nó nằm trong danh sách con của các danh sách khác nhau) cho nên người lập trình không thể biết lúc nào nó hoàn toàn "bị thải", có thể nó bị thải ở danh sách này nhưng vẫn còn được dùng ở danh sách khác. Vì vậy công việc "thu hồi chỗ bị thải" (garbage collection) này, phải được thực hiện bởi một chương trình đặc biệt. Việc quản lý bộ nhớ trong bởi một hệ đa chương trình cũng được thực hiện theo giải pháp tường minh nhưng với các khối có kích thước biến đổi. Cụ thể là khi một chương trình nào đó đã kết thúc thì hệ điều hành biết được miền nhớ nào đã dành cho chương trình đó và biết rằng không có chương trình nào khác có thể cần dùng tới miền đó nữa, nó sẽ đưa miền nhớ đó về "chỗ trống" để sẽ dùng cho một chương trình khác.

Việc quản lý "đống" của SNOBOL hay một số ngôn ngữ khác cũng theo nguyên tắc "thu hồi chỗ bị thải", tương tự như LISP. Chỉ có một điều là ở đây các khối có độ dài thay đổi nên nó phải giải quyết vấn đề "cắt đoạn" vùng nhớ bằng cách chú ý tới việc ghép nối các khối trống kề cận nhau để luôn tạo nên các khối lớn hơn trong trường hợp cụ thể.

8.2 Trường hợp kích thước cố định

Để cho tiện, ta có thể hình dung như đang xét tới việc xử lý các danh sách móc nối tổng quát mà phần tử của nó có thể là một nguyên tử (atom) hoặc một danh sách con (sublist), xem 7.6, và các nút khi nó không còn được sử dụng nữa thường phải do một chương trình đặc biệt phụ trách, dựa trên phương pháp sau đây:

8.2.1 Phương pháp "đếm tham trỏ" (reference count)

Phương pháp này đòi hỏi phải giữ ở mỗi đầu danh sách một "số đếm các tham trỏ": đó là số lượng các con trỏ đang trỏ tới danh sách đó. Khi chương trình được thực hiện, số đếm này sẽ thay đổi. Nếu một danh sách trở thành danh sách con của một danh sách khác thì "số đếm tham trỏ" của nó sẽ tăng lên 1, còn nếu nó bị loại ra khỏi một danh sách khác, nghĩa là nó

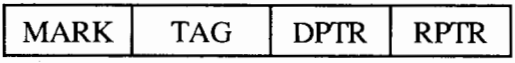
không còn là danh sách con của danh sách này nữa, thì "số đếm tham trỏ" của nó sẽ giảm đi 1. Khi số "đếm tham trỏ" tụt xuống không, nghĩa là danh sách đó không cần dùng nữa, nó đã "bị thải", thì chỗ của nó sẽ được đưa về "danh sách chỗ trống" để tái sử dụng. Với phương pháp này nếu muốn đưa về danh sách chỗ trống từng nút một tất phải gắn với mỗi nút đó một số đếm tham trỏ. Dĩ nhiên điều đó kéo theo sự tốn phí thêm không gian nhớ. Ngoài ra cũng còn thấy, một nhược điểm khác nữa của phương pháp là đối với danh sách có tính đệ qui, nghĩa là danh sách đó lại là danh sách con của nó thì không thể nào giải phóng chỗ của nó được vì bao giờ cũng có con trỏ tới chính nó: số đếm tham trỏ không bao giờ giảm xuống bằng không được.

8.2.2 Phương pháp "sưu tầm chỗ bị thải" (garbage collection)

Đối với phương pháp này tại mỗi nút có một trường 1 bit gọi là "bit đánh dấu" (mark bit). Trong khi chương trình chính làm việc thì không có một nút nào được đưa về "danh sách CHỖ TRỐNG" cả. Chừng nào mà "danh sách CHỖ TRỐNG" đã cạn (hoặc gần cạn) thì lúc đó chương trình "sưu tầm chỗ bị thải" (garbage collector) mới được gọi để làm việc. Lúc đó nó sẽ duyệt qua các danh sách và đánh dấu vào các nút có liên quan đến các danh sách hiện còn đang dùng. Còn đối với các nút lúc đó không gắn với một danh sách nào còn đang dùng nữa, nghĩa là đã bị thải, thì sẽ không được đánh dấu sau khi hoàn thành giai đoạn này nó sẽ chuyển sang giai đoạn 2: đưa tất cả các nút không được đánh dấu về "danh sách CHỖ TRỐNG" và xoá dấu ở các chỗ khác đi.

Việc thực hiện phương pháp này sẽ gặp một khó khăn đặc biệt, khi tiến hành "đánh dấu". Trong quá trình này ta phải duyệt qua danh sách để đi sâu vào từng nút của danh sách đó. Như vậy ta sẽ phải thực hiện chẳng hạn, phép tìm kiếm theo chiều sâu (xem 7.1.3) và nếu thế thì phải sử dụng stack nghĩa là phải có chỗ dự trữ cho stack này. Nhưng lúc đó thì "chỗ trống" lại đang rất hiếm. Nếu như danh sách CHỖ TRỐNG có gần cạn thì chỗ dành cho stack này cũng rất hạn chế và như vậy rất có thể không đủ cho cả quá trình duyệt toàn bộ danh sách. Vì vậy giải thuật thực hiện đánh dấu có dùng stack chưa phải là giải thuật thích hợp. Việc đi tới một giải thuật không đòi hỏi thêm chỗ trống là một yêu cầu thực tế. Giải thuật của Schorr và Waite sau đây sẽ đáp ứng yêu cầu này. Ý tưởng cơ bản của nó nằm ở chỗ: sau khi từ một nút đi tới một nút khác thì đảo ngược mối nối lại để tạo đường quay lui khi đã xong nhiệm vụ đánh dấu. Như vậy nghĩa là trong quá trình thực hiện đánh dấu các mối nối sẽ tạm thời bị thay đổi. Tất nhiên khi đã xong việc thì chúng lại được khôi phục lại như cũ.

Trước khi đi sâu vào giải thuật đánh dấu, ta hãy xét tới qui cách của một nút trong cấu trúc danh sách được sử dụng. Dĩ nhiên như ta đã biết, qui cách của nút trong danh sách tổng quát có nhiều dạng khác nhau, tuỳ thuộc vào những ứng dụng cụ thể. Ở đây ta qui ước: mỗi nút có 4 trường như hình 8.2.



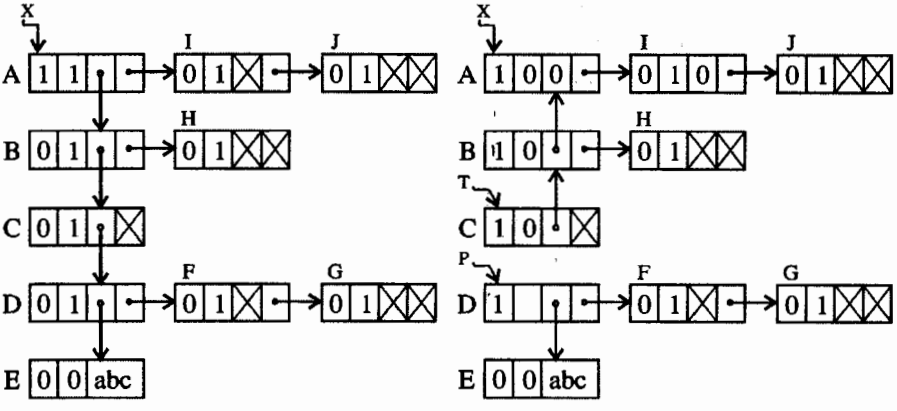
Hình 8.2

MARK : là trường 1 bit dùng để đánh dấu;

TAG: là trường 1 bit, TAG có giá trị bằng 0 ứng với nút nguyên tử, có giá trị bằng 1 ứng với nút danh sách (list node).

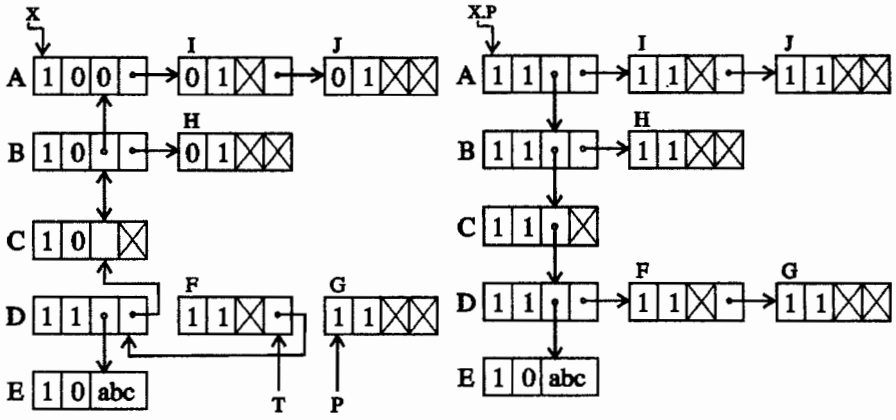
Một nút với TAG bằng 1 có hai trường con trỏ DPTR và RPTR lần lượt trỏ tới danh sách con của nó và nút bên phải của nó, còn với nút mà TAG bằng 0 thì không gian nhớ của hai trường này chỉ để chứa dữ liệu nguyên tử (nếu so với qui cách mà ta đã biết. Trong 7.6.2. thì qui cách này hơi khác).

Bây giờ ta minh hoạ nội dung của giải thuật qua một ví dụ đơn giản như ở hình 8.3.



a) Danh sách lúc khởi đầu

b) Tình trạng khi P ở D



c) Tình trạng khi P ở G d) Kết thúc

Hình 8.3

Thoạt đầu, mọi nút, trừ nút A, đều chưa được đánh dấu và chỉ có một nút E là nút nguyên tử thôi. Từ A ta có thể đi xuống B hoặc đi sang I. Giải thuật này ứng với phép "tìm kiếm theo chiều sâu" nghĩa là luôn đi xuống dưới trước. Ở đây con trỏ P được dùng để trỏ tới nút hiện đang được xem xét (ta gọi tắt là nút hiện hành) con trỏ T trỏ tới nút trước P trên con đường từ nút khởi đầu X đến nút P. Con đường quay lại từ T tới X sẽ được tạo dựng dần dần bằng các mối nối ngược qua các nút trên con đường đó. Nếu ta tiến từ nút P đến nút Q thì hoặc Q = RPTR(P) hoặc Q = DPTR(P) và Q sẽ trở thành nút hiện hành. Nút trước Q trên con đường X - Q là nút P và P sẽ được bổ sung vào làm nút mới trên con đường T-X. Các nút như vậy sẽ được móc nối lại trên con đường đó qua DPTR hoặc RPTR và chỉ có nút danh sách mới được làm như thế. Khi một nút P được bổ sung vào dãy các nút nằm trên đường, thì P sẽ được móc nối với T qua DPTR nếu Q = DPTR(P) và qua RPTR nếu Q = RPTR(P). Để có thể phân biệt được một nút trên đường T-X được móc nối qua DPTR hay RPTR ta sẽ dùng trường TAG. Chú ý rằng chỉ có các nút danh sách là được móc nối trên đường này nên giá trị trường TAG của các nút đó sẽ là 1. Khi DPTR được dùng để móc nối thì TAG sẽ đổi giá trị thành 0. Như vậy nghĩa là trên đường đi T - X thì

$$TAG = \begin{cases} 0 \text{ nếu nút được móc nối qua DPTR} \\ 1 \text{ nếu nút được móc nối qua RPTR} \end{cases}$$

Giá trị của TAG sẽ được khôi phục lại bằng 1 khi nút chứa nó đã bứt ra khỏi danh sách các nút trên đường T - X.

Hình 8.3b cho biết các nút trên đường T - X khi P là nút hiện hành. Các nút A, B, C có TAG bằng 0 cho thấy các mối nối được thực hiện qua DPTR, các nút này cũng đã được đánh dấu. Để tiếp tục "tìm đường" từ P trước hết phải thử "đi xuống" Q=DPTR(P) tức là E, nhưng E là nút nguyên tử nên từ P phải "rẽ ngang" sau khi đã đánh dấu vào E. Như vậy là bây giờ Q=DPTR(P) = F. Đây là nút danh sách chưa được đánh dấu nên được tiếp tục xem xét và P được bổ sung vào làm nút mới trên đường T-X, qua RPTR của nó.

Hình 8.3c thể hiện tình trạng của danh sách lúc nút G được xem xét. Vì G là nút "đường cùng" từ đó không thể "đi xuống" cũng không thể "rẽ ngang", nên phải quay lui theo đường T-X cho tới khi tìm thấy nút nào có nút trỏ bởi RPTR của nó chưa được xem xét. Trong quá trình quay lui này các mối nối và giá trị trường TAG của nút đi qua sẽ được khôi phục lại. Công việc đánh dấu lại được tiếp tục kể từ nút "hiện hành" mới này và cứ tương tự cho tới khi kết thúc.

Sau đây là giải thuật:

Procedure MARK(X)

{X là một nút danh sách, giải thuật này thực hiện đánh dấu tất cả các nút truy nhập được từ X. Trước khi thực hiện giải thuật này tất cả các nút đều chưa được đánh dấu (nghĩa là MARK(i) = 0 với mọi nút i). Để tạo điều kiện kết thúc giải thuật một cách dễ dàng, trong giải thuật này ta chấp nhận một qui ước: MARK(null) = 1, TAG(null) = 0}

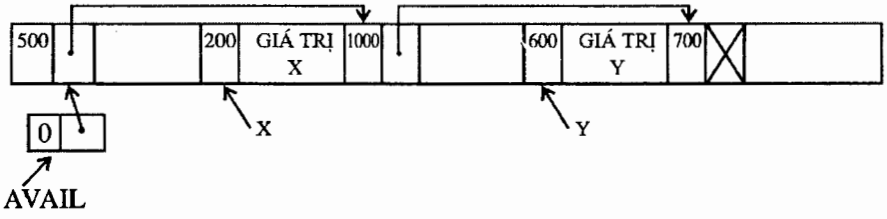
1. {Khởi tạo danh sách các nút trên đường đi T-X}
   P := X; T := null;
2. **repeat**
   Q := DPTR(P); {"đi xuống"}
   case
3. MARK(Q) = 0 **and** TAG(Q) = 1:
   {Q là một nút danh sách còn chưa được xem xét}
   MARK(Q) := 1; TAG(P) := 0;
   DPTR(P) := T; T := P;
   {Bổ sung P vào danh sách các nút trên đường T-X}
   P := Q; {"đi xuống" để thăm dò Q}
4. **else:** {Q là một nút nguyên tử hoặc đã được xem xét rồi}
   MARK(Q) := 1;
5. Q := RPTR(P); {"rẽ ngang" sang phải P}
   case
   MARK(Q) = 0 **and** TAG(Q) = 1:

```
            MARK(Q) := 1;
            RPTR(P) := T; T := P; P := Q;
      else: MARK(Q) := 1; {đánh dấu và chuẩn bị quay lui}
      while T≠null do begin {danh sách các nút trên đường đi chưa rỗng}
            Q := T;
            if TAG(Q) = 0 then begin
            {móc nối tới P qua DPTR} T := DPTR(P);
                  DPTR(Q) := P;
                  TAG(Q) := 1;
                  P := Q; go to 5
                        end;
      {P là nút bên phải Q}
                  T := RPTR (Q); RPTR(Q) := P;
                  P := Q
                  end
            end case
      end case
      until T = null;
6.    return
```

8.3 Trường hợp kích thước thay đổi

Như trên ta đã nói: trường hợp quản lý bộ nhớ theo các nút có kích thước thay đổi làm xuất hiện các tình huống khác hẳn trường hợp nút với kích thước cố định.

Danh sách chỗ trống bây giờ không phải bao gồm các nút như nhau được móc nối lại mà gồm các khối có kích thước khác hẳn nhau. Như vậy ta phải hình dung là ít ra ở mỗi khối ngoài phần không gian nhớ để lưu trữ dữ liệu còn phải có một trường SIZE để ghi nhận kích thước của khối đó (số các byte hoặc các từ tương ứng với một hệ máy), một trường LINK để móc nối với khối tiếp theo trong danh sách chỗ trống. Việc móc nối các khối với nhau cũng có thể có nhiều cách: Theo thứ tự tăng dần hoặc giảm dần của kích thước hoặc theo thứ tự tăng dần của địa chỉ khối... Ở đây ta giả sử chúng được móc nối theo thứ tự tăng dần của địa chỉ và hình 8.4 sau đây sẽ cho ta hình ảnh một miền nhớ gồm 3000 từ, trong đó có hai khối có kích thước 200 từ và 600 từ đang được sử dụng để lưu trữ giá trị của X và Y còn ba khối 500 từ, 1000 từ, 700 từ đang trống, được móc nối lại danh sách chỗ trống trỏ bởi AVAIL.



Hình 8.4

Có một điều cần phải thấy thêm là với cách hình dung như trên thì trong khối có phần lúc này ghi nhận dữ liệu nhưng lúc khác lại ghi nhận con trỏ (chẳng hạn trường LINK khi khối được sử dụng và khi khối được đưa về danh sách chỗ trống - vấn đề này ta coi như không bàn luận ở đây).

Bây giờ ta hãy xét một số tình huống và cách xử lý tương ứng.

8.3.1 Chọn khối "trống"

Nếu bây giờ có yêu cầu: "cần lấy một khối trống có n từ để lưu trữ dữ liệu mới" thì có hai vấn đề sẽ phải xét tới:

Thứ nhất là: chọn khối "trống" nào?

Thứ hai là: Nếu ta chỉ cần sử dụng một phần của khối "trống" thôi, thì phần để lại là phần nào?

Vấn đề thứ hai này có thể giải quyết dễ dàng nếu ta chọn một khối m từ mà chỉ cần dùng n từ thôi (n <m) thì phần để lại sẽ là (m - n) từ đầu. Lúc đó ta chỉ phải sửa giá trị trường kích thước của khối thành (m-n) thôi, phần này vẫn nằm lại trong danh sách chỗ trống sau khi ta đã đem khối n từ cuối ra sử dụng.

Còn vấn đề chọn khối nào, tất nhiên không đơn giản. Một mặt ta muốn chọn nhanh được khối thích hợp, nhưng mặt khác lại muốn không cắt nhỏ các khối lớn ra, kích thước của khối được chọn tốt nhất là bằng n hoặc gần sát n. Hai thái cực này dẫn tới hai cách chọn:

a) Chọn khối "thích hợp đầu tiên" (first fit) hay ta gọi tắt là "chọn khối đầu": ta sẽ tìm trong danh sách chỗ trống khối đầu tiên bắt gặp có kích thước m ≥ n và sẽ đưa ra sử dụng n từ cuối của khối này, như đã nói trên.

b) Chọn khối "thích hợp tốt nhất" (best fit) hay ta gọi tắt là "chọn khối nhỏ": ta sẽ phải tìm trong toàn bộ danh sách chỗ trống một khối có kích

thước m nhỏ nhất mà m ≥ n (sát với kích thước yêu cầu nhất) và đưa ra sử dụng n từ cuối của khối này.

Rõ ràng phương pháp "chọn khối đầu" sẽ đạt được kết quả nhanh chóng hơn vì thông thường ta sẽ không phải tìm suốt cả danh sách chỗ trống, còn phương pháp "chọn khối nhỏ" lại có khả năng làm giảm bớt hiện tượng "cắt đoạn" do những khối có kích thước lớn hơn sẽ được bảo tồn để dành cho các yêu cầu về không gian nhớ lớn hơn sau này.

Tuy nhiên ta cũng không thể khẳng định được là phương pháp nào thích hợp hơn. Điều này tuỳ thuộc vào yêu cầu và từng tình huống cụ thể, có thể thấy qua ví dụ sau:

Giả sử danh sách chỗ trống gồm có bốn khối lần lượt có kích thước là 600, 500, 1000 và 700 từ. Nếu dùng phương pháp "chọn khối đầu" với yêu cầu đầu tiên về không gian nhớ là 400 từ, thì ta sẽ lấy nó từ khối 600 và sau đó danh sách chỗ trống sẽ có bốn khối kích thước lần lượt là 200, 500, 1000, 700. Tiếp theo đó, nếu có ba yêu cầu về khối 600 từ thì danh sách chỗ trống sẽ không đáp ứng được. Mặc dù tổng số chỗ trống của nó là 2400 từ. Còn nếu "chọn khối nhỏ" thì khối 500 từ sẽ được trích ra 400 từ để đáp ứng yêu cầu đầu tiên và để lại danh sách chỗ trống với bốn khối 600, 100, 1000, 700. Ba yêu cầu tiếp theo đó cũng sẽ được thoả mãn dễ dàng. Nhưng nếu sau yêu cầu thứ nhất 400 từ là hai yêu cầu 1000 từ và 700 từ, rồi đến yêu cầu 200 và 500 từ thì "chọn khối nhỏ" sẽ không giải quyết được, trong khi "chọn khối đầu" lại không vấp phải trở ngại nào.

Thứ tự móc nối các khối trong danh sách chỗ trống cũng có ảnh hưởng nhất định. Những điều nhận xét qua ví dụ nêu ở trên là dựa trên giả thuyết các khối trống được móc nối theo thứ tự tăng dần theo địa chỉ. Tuy nó không cải thiện gì về thời gian tìm kiếm cho hai phương pháp chọn khối đã nêu nhưng ta sẽ thấy dưới đây, (xem 8.3.2) nó tạo điều kiện cho việc ghép các khối trống thực hiện được thuận lợi. Còn nếu các khối trống được móc nối theo các danh sách ứng với các khoảng kích thước qui định thì thời gian tìm kiếm khối thích hợp, đối với phương pháp "chọn khối nhỏ", sẽ có khả năng giảm đi. Nếu là thứ tự tăng dần của kích thước thì phương pháp "chọn khối đầu" cũng chính là phương pháp "chọn khối nhỏ". Nếu thứ tự là giảm dần theo kích thước thì "chọn khối đầu" lại chính là chọn khối lớn nhất thích hợp theo yêu cầu. Đây cũng là một cách chọn khác với hai phương pháp nêu trên; được gọi là phương pháp chọn khối "thích hợp tồi nhất" (worst fit) nhưng để cho tiện, ta gọi tắt là phương pháp "chọn khối lớn". Nó cũng có ưu điểm là ít tạo ra những khối có kích thước nhỏ.

Sau đây là giải thuật "chọn khối đầu" (các giải thuật tương ứng với cách chọn khác, coi như bài tập).

Function FIRST-FIT (AVAIL, N, MIN)

{Cho AVAIL là con trỏ, trỏ tới nút đầu danh sách của danh sách chỗ trống, N là kích thước của khối cần dùng, thủ tục này sẽ cho con trỏ P trỏ tới khối có kích thước ≥ N với SIZE(P) ghi cụ thể kích thước khối đó. Nếu kích thước của phần còn thừa lại (sau khi đã lấy ra N từ) ở khối được chọn nhỏ hơn hoặc bằng MIN thì phần đó được coi như bỏ qua, không tách ra để lưu lại trong danh sách chỗ trống nữa mà cho luôn khối trống đưa ra sử dụng. Như vậy nghĩa là không có khối trống nào với kích thước bằng hoặc nhỏ hơn MIN được tạo ra trong danh sách chỗ trống cả. Ở đây biến trỏ Q luôn trỏ tới nút trước P}

1. {Khởi đầu}

```
Q := AVAIL;
P := LINK (Q)
```

2. {Tìm khối thích hợp}

```
while P ≠ null do begin
    if SIZE (P) ≥ N then begin
        K := SIZE (P) - N;
        if K ≤ MIN then LINK (Q) := LINK(P)
        else begin
            SIZE (P) := K;
            P := P + K;
            SIZE(P) := N;
        end;
    return(P)
    end;
    else begin
        Q := P;
        P := LINK(P)
```

3. {Không tìm được khối thích hợp}

```
        end
    return (null)
```

8.3.2 Giải phóng chỗ trống và vấn đề ghép khối

Để thấy rõ hơn các vấn đề đặt ra khi đưa một khối "không sử dụng nữa" về danh sách chỗ trống, ta xét cụ thể vào tình trạng miền nhớ như ta nêu ở hình 8.4. Giả sử bây giờ khối biểu diễn giá trị của Y "được giải phóng" nghĩa là ta sẽ đưa nó về danh sách chỗ trống. Ta thấy ngay cả một vùng nhớ gồm 2300 từ mà bị cắt làm ba đoạn (hiện tượng "cắt đoạn bộ nhớ" - (fragmentation) 1000 từ, 600 từ, 700 từ. Nếu như lúc đó có yêu cầu lấy một khối trống 2000 từ thì vẫn không thể đáp ứng được. Nhưng nếu kèm theo với việc đưa một khối trống về danh sách chỗ trống ta luôn thực hiện ghép nối chúng với các khối kế cận, trong trường hợp có thể (như trường hợp nêu trong ví dụ trên) thì việc đáp ứng các yêu cầu xin chỗ trống sau đó rõ ràng sẽ thuận lợi hơn.

Giải thuật sau đây thực hiện việc đưa một khối về danh sách chỗ trống và ghép nối nó với các khối kế cận đứng trước nó và đứng sau nó trong trường hợp có thể.

Procedure FREE-BLOCK (AVAIL, RB).

{Cho con trỏ AVAIL, trỏ tới nút đầu danh sách của danh sách trống, RB là địa chỉ khối trống được đưa về AVAIL. Ở đây giả thiết trường SIZE của nút đầu danh sách chỗ trống có giá trị bằng 0, nghĩa là SIZE (AVAIL)= 0}

1. {Khởi đầu}
   Q := AVAIL;
   P := LINK(Q);
2. {Tìm khối đứng trước và khối đứng sau}
   while P ≠ null **and** RB > P **do begin**
   Q := P;
   P := LINK(Q);
   end;
3. {Ghép với khối đứng sau}
   if P = null **or** RB + SIZE(RB) ≠ P
   then LINK(RB) := P
   else begin

```
        LINK(RB) := LINK(P);
        SIZE(RB) := SIZE (RB) + SIZE(P)
              end;
```

4. {Ghép với khối đứng trước}

```
    if Q = AVAIL or Q + SIZE(Q) ≠ RB
        then LINK(Q) := RB
        else begin
        LINK(Q) := LINK(RB);
        SIZE(Q) := SIZE(Q) + SIZE(RB)
              end;
```

5. **return**

8.4 Chú thích

Từ trước tới giờ ta luôn luôn quan niệm danh sách chỗ trống bao gồm các nút hoặc cùng có kích thước ấn định hoặc có kích thước tự nhiên bất kỳ, được móc nối lại với nhau theo kiểu như một stack. Điều đó chỉ nhằm mục đích làm cho việc xem xét các vấn đề xuất hiện trong quản lý bộ nhớ được tập trung và nhất quán, chứ không phải vì chỉ có duy nhất một cách tổ chức danh sách chỗ trống như vậy. Một cách khác cũng thường được sử dụng đó là tổ chức danh sách chỗ trống với các khối có kích thước qui định theo các loại khác nhau, chẳng hạn kích thước các khối đều là số luỹ thừa của 2 nghĩa là gồm hoặc 1 từ, 2 từ, 8 từ, 16 từ... kế tiếp nhau.

Như vậy không gian trống được tổ chức như một tập các danh sách chỗ trống, mỗi danh sách ứng với một loại nút có kích thước ấn định. Tất nhiên ở mỗi khối vẫn có hai trường SIZE và LINK như đã nêu trên và chúng được móc nối với nhau theo trường LINK, khi chúng cùng một loại kích thước.

Nếu bây giờ có yêu cầu lấy một khối n từ để sử dụng thì có thể xảy ra các tình huống sau:

1) Có khối lượng đúng với kích thước yêu cầu.

Ví dụ: $n = 32 = 2^5$ thì một khối thuộc danh sách chỗ trống ứng với kích thước 2^5 được lấy ra.

2) Không có khối đúng với kích thước yêu cầu.

- Nếu có khối $2^{k-1} < n < 2^k$ thì ta sử dụng khối 2^k.

Ví dụ: n = 100 ta luôn thấy $64 = 2^6 < 100 < 2^7 = 128$. Như vậy ta sẽ dùng khối có kích thước 2^7, tất nhiên thừa ra 28 từ.

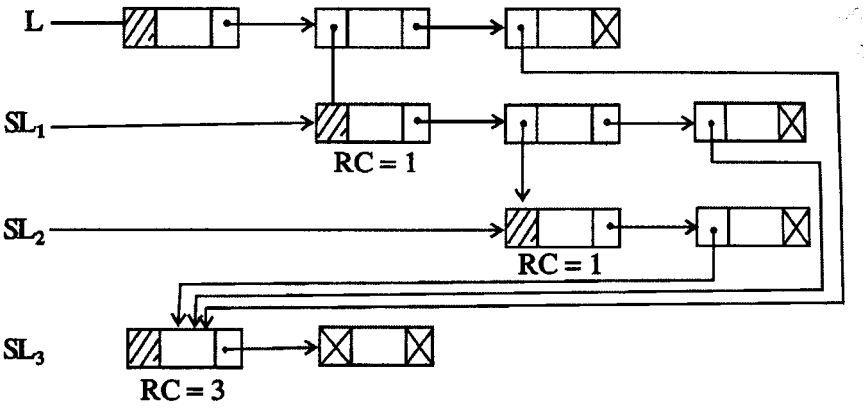
- Nếu không có khối có kích thước 2^k như trên mà chỉ có khối lớn hơn, chẳng hạn có khối kích thước 2^9. Lúc đó khối này sẽ được tách đôi dần và cuối cùng sẽ được khối có kích thước sát hợp với yêu cầu. Như với khối 2^9 có thể tách thành 2 khối có kích thước 2^8 và một trong 2 khối này lại tách thành hai khối có kích thước 2^7 để một trong hai khối này đáp ứng yêu cầu (như vậy một khối 2^8 và 2^7 còn lại sẽ được đưa về danh sách chỗ trống ứng với các kích thước đó).

Còn nếu ngược lại có một khối nào đó không dùng nữa thì nó sẽ được đưa về danh sách chỗ trống tương ứng với kích thước của nó và trong trường hợp có thể nếu các khối kế cận có thể ghép lại được thì chúng sẽ được ghép lại thành khối có kích thước lớn hơn và được chuyển sang danh sách chỗ trống tương ứng với kích thước lớn hơn đó.

Việc chọn kích thước qui định trên cũng làm cho việc tách và ghép khối được thuận tiện. Ta sẽ không đi sâu vào việc xem xét thêm cách tổ chức này.

BÀI TẬP CHƯƠNG 8

8.1 Một danh sách L có cấu trúc như hình vẽ



L có 2 phần tử là danh sách con SL_1 và SL_3, SL_1 lại có hai phần tử là SL_2 và SL_3. Mỗi danh sách đều có nút đầu danh sách, ở đó có trường RC để ghi số đếm tham trỏ.

Hãy vẽ lại hình ảnh của danh sách khi SL_1 bị loại ra khỏi L và giải thích.

8.2. Một danh sách L có hai phần tử là hai danh sách con SL_1 và SL_3. SL_1 lại có ba phần tử là SL_2, SL_3, SL_4. SL_2 có một phần tử duy nhất là SL_3, SL_3 là một danh sách con rỗng, SL_4 có một phần tử duy nhất là SL_2.

a) Số đếm tham trỏ của SL_1, SL_2, SL_3, SL_4 là bao nhiêu?

b) SL_1 bị loại khỏi L, cấu trúc của L sẽ thay đổi như thế nào?

8.3. Cho danh sách L có dạng như trong bài 8.1. Giả sử bộ sưu tầm chỗ bị thải hoạt động trước khi SL_1 bị loại khỏi L.

a) Hãy nêu thứ tự các nút được đánh dấu.

b) Vẽ hình phản ánh sự thay đổi các mối nối trong quá trình đánh dấu.

8.4. Tương tự như bài 8.3 với giả thiết, bộ sưu tầm chỗ bị thải hoạt động sau khi SL_1 bị loại.

8.5. Cho một miền nhớ kế tiếp mà giới hạn được xác định bởi hai con trỏ HEAD và TAIL.

HEAD là địa chỉ đầu tiên của miền nhớ.

TAIL là địa chỉ cuối cùng của miền nhớ.

Trong miền nhớ này có các khối, ứng với mỗi khối đều có một trường MARK để đánh dấu, trường SIZE ghi kích thước và trường LINK để móc nối, khi đưa về danh sách chỗ trống.

Hãy viết giải thuật thực hiện giai đoạn II của phương pháp sưu tầm chỗ bị thải: Thu hồi các khối không được đánh dấu về danh sách chỗ trống trỏ bởi AVAIL, và sau đó xoá các dấu đó ở các khối còn đang sử dụng.

8.6. Thử xét xem trong các trường hợp sau dùng phương pháp "chọn khối đầu" hoặc "chọn khối nhỏ" thì phương pháp nào lợi hơn?

	Yêu cầu bộ nhớ	Không gian trống
a)	1000 từ	1300, 1200
	1100 từ	
	250 từ	
b)	800 từ	2000, 1000
	1300 từ	

8.7. Lập giải thuật thực hiện lấy từ danh sách chỗ trống một khối N từ bằng phương pháp "chọn khối nhỏ".

PHẦN III
SẮP XẾP VÀ TÌM KIẾM

Chương 9

SẮP XẾP

9.1 Đặt vấn đề

Sắp xếp (Sorting) là quá trình bố trí lại các phần tử của một tập đối tượng nào đó, theo một thứ tự ấn định. Chẳng hạn thứ tự tăng dần (hay giảm dần) đối với một dãy số, thứ tự từ điển đối với một dãy chữ.v.v...

Yêu cầu về sắp xếp thường xuyên xuất hiện trong các ứng dụng tin học, với những mục đích khác nhau: sắp xếp dữ liệu lưu trữ trong máy tính để tìm kiếm cho thuận lợi, sắp xếp các kết quả xử lý để in ra trên bảng biểu...

Nói chung, dữ liệu có thể xuất hiện dưới nhiều dạng khác nhau, nhưng ở đây ta quy ước: tập đối tượng được sắp xếp là tập các bản ghi (records), mỗi bản ghi bao gồm một số trường (fields) dữ liệu, tương ứng với những thuộc tính (attributs) khác nhau.

Trong chương này ta chỉ xét tới các phương pháp *sắp xếp trong* (internal sorting), nghĩa là các phương pháp tác động trên một tập các bản ghi lưu trữ đồng thời ở bộ nhớ trong, mà ta gọi là *bảng* (table) để phân biệt với tệp (file) hiện nay thường được dùng để chỉ một tập lớn các bản ghi lưu trữ ở bộ nhớ ngoài, và sắp xếp tương ứng với tệp sẽ được gọi là *sắp xếp ngoài* (external sorting).

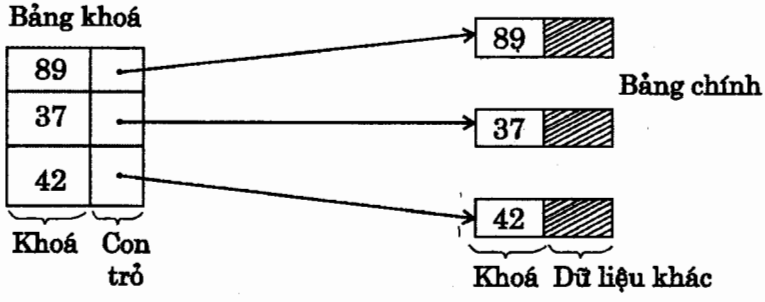
Việc xét các phương pháp sắp xếp, cũng như tìm kiếm, đối với tệp sẽ được đề cập ở chương 11.

Như vậy bài toán được đặt ra ở đây là sắp xếp đối với một bảng gồm n bản ghi $R_1, R_2,..., R_n$.

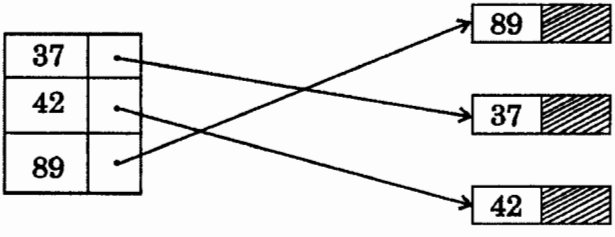
Tuy nhiên, ta thấy: Không phải toàn bộ các trường dữ liệu trong bản ghi đều được xem xét đến trong quá trình sắp xếp mà chỉ một trường nào đó (hoặc một vài trường nào đó - nhưng trường hợp này ta sẽ không đề cập đến) dược chú ý tới thôi. Trường như vậy gọi là *khoá* (key). Sắp xếp sẽ được tiến hành dựa vào giá trị của khoá nầy.

Ví dụ: Bảng danh mục điện thoại các cơ quan nhà nước ở Hà Nội, bao gồm các bản ghi ứng với từng cơ quan. Mỗi bản ghi gồm có ba trường: Tên cơ quan, địa chỉ, số điện thoại. Ở đây khoá sắp xếp chính là tên cơ quan.

Khi sắp xếp, các bản ghi trong bảng sẽ được bố trí lại các vị trí sao cho giá trị khoá sắp xếp tương ứng với chúng có đúng thứ tự ấn định. Ta thấy kích thước của khoá thường khá nhỏ so với kích thước bản ghi. Sắp xếp nếu thực hiện trực tiếp trên các bản ghi của bảng sẽ kéo theo sự chuyển đổi vị trí của các bản ghi và việc đó có thể đòi hỏi phải sao chép lại toàn bộ thông tin của bản ghi vào chỗ mới, gây ra tốn phí thời gian khá nhiều. Thường người ta khắc phục tình trạng này bằng cách xây dựng một bảng phụ, cũng gồm n bản ghi như bảng chính nhưng mỗi bản ghi ở đây chỉ có hai trường: (khoá, con trỏ). Trường "khoá" chứa giá trị của khoá ứng với từng bản ghi trong bảng chính, trường "con trỏ" chứa con trỏ, trỏ tới bản ghi tương ứng. Bảng phụ này được gọi là *bảng khoá* (key table), sắp xếp sẽ thực hiện trên bảng khoá đó. Như vậy, trong quá trình sắp xếp, bảng chính không hề bị ảnh hưởng gì còn việc truy nhập vào bản ghi nào đó của bảng chính, khi cần thiết, vẫn thực hiện được bằng cách dựa vào con trỏ của bản ghi tương ứng thuộc bảng khoá này.



a) Trước khi sắp xếp



b) Sau khi sắp xếp

Hình 9.1

Do khoá có vai trò đặc biệt như vậy nên sau này, khi trình bày các phương pháp cũng như giải thuật hay trong các ví dụ minh hoạ, ta sẽ coi khoá như đại diện cho các bản ghi và để cho đơn giản ta chỉ nói tới giá trị khoá thôi. Thực ra phép đổi chỗ được tác động vào các bản ghi nhưng ở đây ta cũng chỉ nói tới phép đổi chỗ đối với khoá, và bài toán sắp xếp bây giờ coi như được đặt ra một cách đơn giản với một bảng các khoá (dãy khoá) $K_1, K_2,...,K_n$ (tương ứng với các bản ghi $R_1, R_2, ..., R_n$) và $K_i \neq K_j$ nếu $i \neq j$.

Tất nhiên giá trị của khoá có thể là số, là chữ và thứ tự sắp xếp cũng được quy định tương ứng với khoá. Nhưng ở đây, để minh hoạ cho các phương pháp ta sẽ coi giá trị khoá là số và thứ tự sắp xếp là thứ tự tăng dần. Để nhất quán, ta sẽ dùng một dãy khoá sau đây:

42 23 74 11 65 58 94 36 99 87

để làm ví dụ.

9.2 Một số phương pháp sắp xếp cơ bản

9.2.1 Sắp xếp kiểu lựa chọn (selection sort)

Một trong những phương pháp đơn giản nhất để thực hiện sắp xếp một bảng khoá là dựa trên phép lựa chọn.

Nguyên tắc cơ bản của phương pháp sắp xếp này là "ở lượt thứ i (i = 1, 2, ..., n) ta sẽ chọn trong dãy khoá $K_i, K_{i+1}, ..., K_n$ khoá nhỏ nhất và đổi chỗ nó với K_i".

Như vậy thì rõ ràng là sau j lượt, j khoá nhỏ hơn đã lần lượt ở các vị trí thứ nhất, thứ hai,..., thứ j theo đúng thứ tự sắp xếp. Ví dụ

i	K_i	lượt	1	2	3	4	...	9
1	42		11	11	11	11		11
2	23		23	23	23	23		23
3	74		74	74	36	36		36
4	11		42	42	42	42		42
5	65		65	65	65	65		58
6	58		58	58	58	58		65
7	94		94	94	94	94		74
8	36		36	36	74	74		87
9	99		99	99	99	99		94
10	87		87	87	87	87		99

Sau đây là giải thuật:

Proceduce SELECT-SORT (K,n)

{Cho dãy khoá K gồm n phần tử. Giải thuật này thực hiện sắp xếp các phần tử của K theo thứ tự tăng dần, dựa vào phép chọn phần tử nhỏ nhất trong mỗi lượt}

```
1.  for i: = 1 to n-1 do
2.          begin
                    m := i;
3.                  for j: = i+1 to n do
                    if K[j] < K[m] then m: = j;
4.                          if m ≠ j then
                            begin { đổi chỗ }
                                    X:= K[i];
                                    K[i]: = K[m]
                                    K[m]: = X
                            end
            end
5.  return
```

9.2.2 Sắp xếp kiểu thêm dần (insertion sort)

Nguyên tắc sắp xếp ở đây dựa theo kinh nghiệm của những người chơi bài. Khi có i-1 lá bài đã được sắp xếp đang ở trên tay, nay rút thêm lá bài thứ i nữa thì sắp xếp lại như thế nào? - Có thể so sánh lá bài mới lần lượt với lá bài thứ (i-1), thứ (i-2)... để tìm ra "chỗ" thích hợp và "chèn" nó vào chỗ đó.

Dựa trên nguyên tắc này có thể triển khai một cách sắp xếp như sau:

Thoạt đầu K_1 được coi như bảng chỉ gồm có một khoá đã sắp xếp. Xét thêm K_2, so sánh nó với K_1 để xác định chỗ "chèn" nó vào, sau đó ta sẽ có một bảng gồm hai khoá đã được sắp xếp. Đối với K_3 lại so sánh với K_2, K_1 và cứ tương tự như vậy đối với K_4, K_5, K_6.v.v... cuối cùng sau khi xét xong K_n thì bảng khoá đã được sắp xếp hoàn toàn.

Ta thấy ngay phương pháp này rất thuận lợi khi các khoá của dãy được đưa dần vào miền lưu trữ. Đó cũng chính là không gian nhớ dùng để sắp xếp. Có thể minh hoạ qua bảng sau:

Lượt	1	2	3	4		8	9	10
Khoá đưa vào	42	23	74	11		36	39	87
1	42	23	23	11		11	11	11
2	-	42↓	42	23↓		23↓	23	23
3	-	-↓	74	42↓		36↓	36	36
4	-	-	-	74↓		42↓	42	42
5	-	-	-	-		58↓	58	58
6	-	-	-	-		65↓	65	65
7	-	-	-	-		74↓	74	74↓
8	-	-	-	-		94↓	94	87↓
9	-	-	-	-		-	99	95↓
10	-	-	-	-		-	-	99↓

(Dấu - chỉ chỗ trống trong miền sắp xếp

Dấu ↓ chỉ việc phải dịch chuyển khoá cũ để lấy chỗ chèn khoá mới vào)

* Nhưng nếu các khoá đã có mặt ở bộ nhớ trong trước lúc sắp xếp rồi thì sao?

Sắp xếp vẫn có thể thực hiện được ngay tại chỗ chứ không phải chuyển sang một miền sắp xếp khác. Lúc đó các khoá cũng sẽ lần lượt được xét tới và việc xác định chỗ cho khoá mới vẫn làm tương tự, chỉ có khác là: để dành chỗ cho khoá mới nghĩa là phải dịch chuyển một số khoá lùi lại sau, ta không có sẵn chỗ trống như trường hợp nói trên (vì khoá đang xét và các khoá sẽ được xét đã chiếm các vị trí đằng sau này rồi), do đó phải đưa khoá mới này ra một chỗ nhớ phụ và sẽ đưa vào vị trí thực của nó sau khi đã đẩy các khoá cần thiết lùi lại.

Sau đây là giải thuật ứng với trường hợp này.

*** Procedure** INSERT-SORT (K,n)

{Trong thủ tục này người ta dùng X làm ô nhớ phụ để chứa khoá mới đang được xét. Để đảm bảo cho khóa mới trong mọi trường hợp, ngay cả khi vị trí thực của nó là vị trí đầu tiên, đều được "chèn" vào giữa khoá nhỏ hơn nó và khoá lớn hơn nó, ở đây đưa thêm vào một khoá giả K_0, có giá trị nhỏ hơn mọi khoá của bảng, và đứng trước mọi khoá đó. Ta qui ước $K_0 = -\infty$}.

1. K[0] = - ∞
2. **for** i:= 2 to n **do begin**
3. X := K[i]; j:= i-1;

4. {Xác định chỗ cho khoá mới được xét và dịch chuyển các khoá cần thiết}

```
while x < K[j] do begin
        K[j + 1] := K[j];
        j := j - 1;
              end;
```

5. {đưa X vào đúng chỗ} K[j + 1] := X

```
end;
```

6. **return**

Bảng ví dụ minh hoạ tương ứng với các lượt sắp xếp theo giải thuật này, tương tự như bảng đã nêu ở trên, chỉ có khác là không có chỗ nào trống trong miền sắp xếp cả, vì những chỗ đó đang chứa các khoá chưa được xét tới trong mỗi lượt (người đọc có thể tự lập ra bảng ví dụ minh hoạ này).

9.2.3 Sắp xếp kiểu đổi chỗ (exchange sort)

Trong các phương pháp sắp xếp nêu trên, tuy kỹ thuật đổi chỗ đã được sử dụng nhưng nó chưa trở thành một đặc điểm nổi bật. Bây giờ ta mới xét tới phương pháp mà việc đổi chỗ một cặp khoá kế cận, khi chúng ngược thứ tự, sẽ được thực hiện thường xuyên cho tới khi toàn bộ bảng các khoá đã được sắp xếp. Ý cơ bản có thể nêu như sau:

Bảng các khoá sẽ được duyệt từ đáy lên đỉnh. Dọc đường, nếu gặp hai khoá kế cận ngược thứ tự thì đổi chỗ chúng cho nhau. Như vậy trong lượt đầu khoá có giá trị nhỏ nhất sẽ chuyển dần lên đỉnh. Đến lượt thứ hai khoá có giá trị nhỏ thứ hai sẽ được chuyển lên vị trí thứ hai.v.v... Nếu hình dung dãy khoá được đặt thẳng đứng thì sau từng lượt sắp xếp các giá trị khoá nhỏ sẽ "nổi" dần lên giống như các bọt nước nổi lên trong nồi nước đang sôi. Vì vậy phương pháp này thường được gọi bằng cái tên khá đặc trưng là: sắp xếp kiểu nổi bọt (buble sort).

Ví dụ:

i	K_i	lượt 1	2	3	4	5	. . .	9
1	42	11	11	11	11	11		11
2	23	42	23	23	23	23		23
3	74	23	42	36	36	36		36
4	11	74	36	42	42	42		42
5	65	36	74	58	58	58		58
6	58	65	58	74	65	65		65
7	94	58	65	65	74	74		74
8	36	94	87	87	87	87		87
9	99	87	94	94	94	94		94
10	87	99	99	99	99	99		99

Sau đây là giải thuật:

Procedure BUBBLE- SORT(K,n)

1. **For** i:=1 **to** n - 1 **do**
2. **for** j:= n **down to** i + 1 **do**
3. **if** K[j] < K[j-1] **then**
   begin
4. x:= K [j];
   K[j] := K[j-1];
   K[j-1] := x
   end;
5. **return**

Giải thuật này rõ ràng còn có thể cải tiến được nhiều. Chẳng hạn xét qua ví dụ ở trên ta thấy: sau lượt thứ ba không phải chỉ có ba khoá 11, 23, 36 vào đúng vị trí sắp xếp của nó mà là 5 khoá. Còn sau lượt thứ tư thì tất cả các khoá đã nằm đúng vào vị trí của nó rồi. Như vậy nghĩa là năm lượt cuối không có tác dụng gì thêm cả. Từ đó có thể thấy: nếu "nhớ" được vị trí của khoá được đổi chỗ cuối cùng ở mỗi lượt thì có thể coi đó là "giới hạn" cho việc xem xét ở lượt sau. Chừng nào mà giới hạn này chính là vị trí thứ n, nghĩa là trong lượt ấy không có một phép đổi chỗ nào nữa thì sắp xếp có thể kết thúc được. Nhận xét này sẽ dẫn tới một giải thuật cải tiến hơn, chắc chắn có thể làm cho số lượt giảm đi và số lượng các phép so sánh trong mỗi lượt cũng giảm đi nữa. Người đọc hãy tự xây dựng giải thuật theo ý cải tiến này (coi như một bài tập).

9.2.4 Phân tích so sánh ba phương pháp

Đối với một phương pháp sắp xếp, khi xét tới hiệu lực của nó ngoài những đánh giá về mặt không gian nhớ cần thiết, người ta thường lưu ý đặc biệt tới chi phí về thời gian. Mà thời gian thì chủ yếu phụ thuộc vào việc thực hiện các phép so sánh giá trị khoá và các phép chuyển chỗ bản ghi, khi sắp xếp. Vì vậy thông thường người ta lấy số lượng trung bình các phép so sánh hoặc các phép chuyển chỗ làm đại lượng đặc trưng cho chi phí về thời gian thực hiện của từng phương pháp. Ở đây phép so sánh sẽ được coi như một "phép toán tích cực" để đánh giá thời gian thực hiện của các giải thuật.

Đối với phương pháp sắp xếp kiểu lựa chọn, ta thấy: ở lượt thứ i (i = 1, 2, ..., n-1) để tìm khoá nhỏ nhất bao giờ cũng cần $C_i = (n-i)$ phép so sánh. Số lượng phép so sánh này không hề phụ thuộc gì vào tình trạng ban đầu của dãy khoá cả. Từ đó suy ra:

$$C_{min} = C_{max} = C_{tb} = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

Còn đối với phương pháp sắp xếp kiểu thêm dần (giải thuật INSERT-SORT) thì có hơi khác. Rõ ràng số lượng phép so sánh phụ thuộc và dãy khoá ban đầu. Trường hợp thuận lợi nhất ứng với dãy khoá đã được sắp xếp rồi. Như vậy ở mỗi lượt chỉ cần 1 phép so sánh. Do đó:

$$C_{min} = \sum_{i=2}^{n} 1 = n-1$$

Nhưng nếu dãy khoá ban đầu có thứ tự ngược với thứ tự sắp xếp thì ở lượt thứ i phải cần có: $C_i = (i-1)$ phép so sánh. Vì vậy:

$$C_{max} = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$

Nếu giả sử mọi giá trị khoá đều xuất hiện đồng khả năng thì trung bình ở lượt thứ i có thể coi như $C_i = \frac{i}{2}$ phép so sánh. Suy ra:

$$C_{tb} = \sum_{i=2}^{n}\frac{i}{2} = \frac{n^2+n-2}{4}$$

Với sắp xếp kiểu nổi bọt theo giải thuật BUBBLE-SORT như trên, nghĩa là chưa hề có cải tiến gì, thì tương tự như phương pháp đầu, ta cũng có:

$$C_{min} = C_{max} = C_{tb} = \frac{n(n-1)}{2}$$

Nhìn vào các kết quả đánh giá ở trên ta thấy INSERT-SORT tỏ ra "tốt hơn" so với hai phương pháp kia. Tuy nhiên, với n khá lớn, chi phí về thời

gian thực hiện được đánh giá qua cấp độ lớn, thì cả ba phương pháp đều có cấp $O(n^2)$ và đây vẫn là một chi phí cao so với một số phương pháp mà ta sẽ xét thêm sau đây:

9.3 Sắp xếp kiểu phân đoạn (Partition - Sort) hay sắp xếp "nhanh" (Quick - Sort)

9.3.1. Giới thiệu phương pháp

Sắp xếp kiểu phân đoạn là một cải tiến của phương pháp sắp xếp kiểu đổi chỗ. Đây là một phương pháp khá tốt, do đó người sáng lập ra nó C.A.R. Hoare, đã mạnh dạn đặt cho nó cái tên hấp dẫn là sắp xếp NHANH.

Ý chủ đạo của phương pháp có thể tóm tắt như sau:

Chọn một khoá ngẫu nhiên nào đó của dãy làm "chốt" (pivot). Mọi phần tử nhỏ hơn khoá "chốt" phải được xếp vào vị trí ở trước "chốt" (đầu dãy), mọi phần tử lớn hơn khoá "chốt" phải được xếp vào vị trí sau "chốt" (cuối dãy). Muốn vậy, các phần tử trong dãy sẽ được so sánh với khoá chốt và sẽ đổi vị trí cho nhau, hoặc cho chốt, nếu nó lớn hơn chốt mà lại nằm trước chốt hoặc nhỏ hơn chốt mà lại nằm sau chốt. Khi việc đổi chỗ đã thực hiện xong thì dãy khoá lúc đó được phân làm hai đoạn: một đoạn gồm các khoá nhỏ hơn chốt, một đoạn gồm các khoá lớn hơn chốt còn khoá chốt thì ở giữa hai đoạn nói trên, đó cũng là vị trí thực của nó trong dãy khi đã được sắp xếp, tới đây coi như kết thúc một lượt sắp xếp.

Ở các lượt tiếp theo cũng áp dụng một kỹ thuật tương tự đối với các phân đoạn còn lại. Lẽ tất nhiên chỉ có một phân đoạn được xử lý ngay sau đó, còn một phân đoạn phải để lúc khác, nghĩa là phải được "ghi nhớ" lại.

Quá trình xử lý một phân đoạn, ghi nhớ phân đoạn còn lại được thực hiện tiếp tục cho tới khi gặp một phân đoạn chỉ gồm có một phần tử thì việc ghi nhớ không cần nữa. Lúc đó một phân đoạn mới sẽ được xác định và đối với phân đoạn này quá trình lặp lại tương tự. Sắp xếp sẽ kết thúc khi phân đoạn cuối cùng đã được xử lý xong.

9.3.2 Ví dụ và giải thuật

Giả sử, ta qui ước chọn khoá "chốt" là khoá đầu tiên của dãy. Như vậy với dãy

42 23 74 11 65 58 94 36 99 87

thì chốt là 42.

Ta sẽ phải làm sao để các số nhỏ hơn 42 như 23, 11, 36 phải chuyển về phía trước còn các số lớn hơn 42 như 74, 65, 58, 94, 99, 87 phải nằm ở phía sau 42. Nếu được như vậy thì ta đã tách dãy khoá cho ra thành 2 dãy con (hay 2 phân đoạn), một dãy chiếm 3 vị trí đầu, một dãy chiếm 6 vị trí sau và khoá "chốt" 42 sẽ được đưa vào vị trí thứ 4, nghĩa là vị trí đúng của nó trong sắp xếp. Lúc đó ta đã thực hiện xong một lượt phân đoạn.

Giả sử ta thể hiện việc phân đoạn như trên bằng một thủ tục, mà ta sẽ gọi là thủ tục PART(K, LB,UB,j) với qui ước: LB là chỉ số của phần tử đầu của dãy khoá đang xét (ta gọi tắt là "biên dưới") và UB là chỉ số của phần tử cuối của dãy khoá đó (ta gọi là "biên trên"). Còn K là vectơ biểu diễn dãy khoá cho, j là chỉ số ứng với khoá chốt sau khi đã tách dãy khoá đang xét thành 2 phân đoạn.

Với thủ tục PART này thì thủ tục đệ qui thể hiện phương pháp sắp xếp NHANH có thể viết như sau:

```
Procedure QUICK-SORT (K, LB, UB);
1.  if LB < UB then begin
2.          call PART (K, LB, UB,j);
3.          call QUICK_SORT(K, LB, j-1);
4.          call QUICK - SORT(K, j+1, UB);
                    end;
5.  return
```

Dĩ nhiên để sắp xếp dãy khoá K với n phần tử thì phải thực hiện lời gọi

call QUICK-SORT (k,1,n);[1]

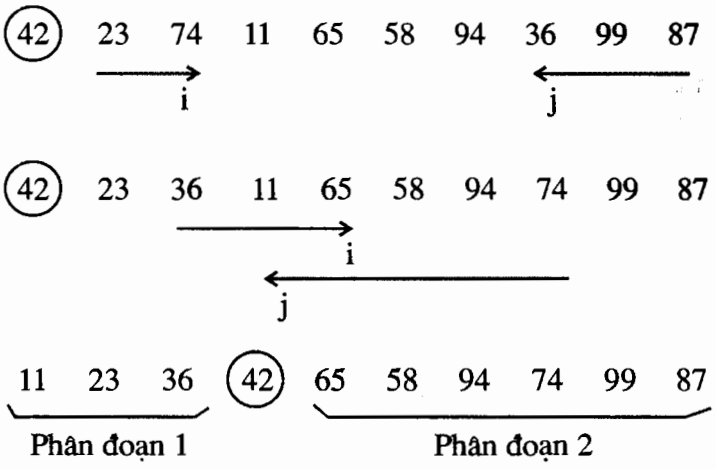
Việc xây dựng thủ tục PART sẽ dựa trên kỹ thuật sau:

Người ta đưa thêm vào một giá trị khoá giả K[n+1], lớn hơn mọi giá trị khoá trong dãy khoá cho. Nó sẽ đóng vai trò như một lính gác (ta sẽ gọi là "khoá gác biên") để khống chế biên trên, giúp cho việc xử lý được thuận lợi. Quá trình tìm số nhỏ hơn chốt để chuyển về phía trước chốt và khoá lớn hơn chốt để chuyển về phía sau chốt sẽ dựa vào hai biến chỉ số i và j để duyệt qua dãy khoá theo chiều ngược nhau.

Thoạt đầu i lấy giá trị của LB+1 (vì K[LB] đã được chọn làm chốt rồi) còn j lấy giá trị của UB.

i sẽ được tăng giá trị lên 1 chừng nào mà K[i] còn bé hơn "chốt", cho tới khi k[i] lớn hơn chốt thì thôi. Tiếp đó j bắt đầu được giảm giá trị, chừng nào mà k[j] còn lớn hơn chốt, cho tới khi K[j] bé hơn chốt. Nếu lúc đó i < j thì K[i] và K[j] được đổi chỗ cho nhau. Công việc lại được tiếp tục theo cách tương tự. Cho tới khi i > j thì lúc đó K[j] và khoá chốt sẽ đổi chỗ cho nhau và quá trình phân đoạn cũng kết thúc.

Lượt phân đoạn đầu tiên của dãy khoá cho ở trên, có thể minh hoạ như sau:



Sau đây là thủ tục thể hiện cách làm trên.

Procedure PART(K,LB, UB, j);

1. i := LB + 1; j := UB;
2. **while** i ≤ j **do begin**
3. **while** K[i] < K[LB] **do** i:= i + 1;
4. **while** K[j] > K[LB] **do** j := j - 1;
5. **if** i <j **then begin**

   K[i] ↔ K[j];

   i := i + 1;

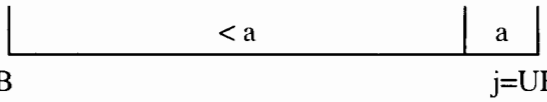
   j := j - 1;

   end

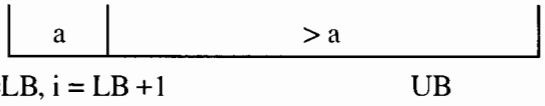
   end;
6. K[LB] ↔ K[j]
7. **return**

Cần chú ý tới một vài trường hợp đặc biệt khi thực hiện thủ tục này. Giả sử ta gọi giá trị của khoá chốt là a.

Trong trường hợp mà tất cả các phần tử của dãy đang xét đều nhỏ hơn a, khi thực hiện câu lệnh **while** ở bước 3. i cứ tăng lên, nhưng nó sẽ dừng lại khi gặp "khoá gác biên", nghĩa là lúc i = UB + 1 còn j thì vẫn giữ giá trị là UB vì K[j] lúc đó đã bé hơn khoá chốt rồi. Lúc này khoá chốt sẽ được đổi chỗ với K[j]: a sẽ được đưa về cuối dãy, đúng với vị trí của nó trong sắp xếp.



Trường hợp tất cả các phần tử của dãy đang xét đều lớn hơn a thì i vẫn giữ nguyên giá trị LB + 1, còn j thì giảm và dừng lại ở giá trị LB vì lúc đó K[j] = K[LB], điều kiện để thoát ra khỏi vòng **while** ở bước 4, đồng thời i >j nên cũng thoát ra khỏi vòng **while** ngoài cùng ở bước 2 và bước 6 được thực hiện, nghĩa là a vẫn ở vị trí cũ, đúng với vị trí của nó trong sắp xếp.



Qua đây ta cũng thấy do việc chọn "chốt" là khoá đầu tiên nên K[LB] đã đóng vai trò "gác biên dưới" còn K[UB + 1] được đưa thêm vào để "gác biên trên", giúp cho việc kết thúc thực hiện thủ tục PART.

Với hai trường hợp nói trên, sau khi thực hiện thủ tục PART ta đã không tách được dãy khoá thành 2 phân đoạn con, mà chỉ được một thôi. Nếu đối với các phân đoạn sau lại gặp tình huống tương tự như vậy (chẳng hạn ta thực hiện QUICK - SORT với 1 dãy khoá K vốn đã có thứ tự tăng dần, hoặc giảm dần rồi), thì thời gian thực hiện giải thuật sẽ chẳng khác gì thời gian như đối với BUBBLE-SORT cả. Đây chính là trường hợp bất lợi nhất của QUICK-SORT.

Bảng sau đây cho kết quả của sắp xếp theo QUICK-SORT sau từng lượt, đối với dãy khoá làm ví dụ nêu trên.

lượt	K_i 42	23	74	11	65	58	94	36	99	87
1	(11	23	36)	42	(65	58	94	74	99	87)
2	11	(23	36)	42	(65	58	94	74	99	87)
3	11	23	(36)	42	(65	58	94	74	99	87)
4	11	23	36	42	(58)	65	(94	74	99	87)
5	11	23	36	42	58	65	(94	74	99	87)
6	11	23	36	42	58	65	(87	74)	94	(99)
7	11	23	36	42	58	65	(74)	87	94	(99)
8	11	23	36	42	58	65	74	87	94	(99)
9	11	23	36	42	58	65	74	87	94	99

9.3.3 Nhận xét và đánh giá

Trước hết ta hãy để ý đến một vài chi tiết có ảnh hưởng tới hiệu lực của phương pháp đồng thời cũng thể hiện rõ đặc điểm của phương pháp này.

a) Vấn đề chọn "chốt"

Theo giới thiệu chung thì chốt có thể chọn tuỳ ý trong dãy khoá cho, nhưng rõ ràng khi thể hiện giải thuật ta phải định ra một cách chọn khoá chốt cụ thể. Nếu chốt ta chọn rơi vào đúng khoá nhỏ nhất (hoặc lớn nhất) của phân đoạn cần xử lý thì sau mỗi lượt ta chỉ tách ra được một phân đoạn con có kích thước nhỏ hơn trước là 1 (vì đã bớt đi một khoá chính là "chốt") và chính phân đoạn này sẽ được xử lý tiếp theo luôn. Như vậy là ta đã quay trở lại phương pháp sắp xếp kiểu nổi bọt đơn giản. Việc chọn chốt như thế này đã dẫn đến tình huống xấu nhất của phương pháp.

Nếu gọi *"trung vị"* (median) của một dãy khoá là khoá sẽ đứng ở giữa dãy đó sau khi dãy đã được sắp xếp, nghĩa là nó lớn hơn một nửa số khoá của dãy và nhỏ hơn số còn lại, thì tốt nhất vẫn là chọn được đúng trung vị làm "chốt". Lúc đó sau mỗi lượt ta sẽ tách ra được hai phân đoạn con có độ dài gần như nhau và phân đoạn xử lý tiếp theo có kích thước chỉ bằng "nửa" phân đoạn đã chứa nó.

Nhưng làm sao có thể chọn được đúng trung vị? Nếu giả thiết: sự xuất hiện của các khoá trong dãy là đồng khả năng thì trung vị có thể là bất kỳ một khoá nào trong dãy. Trong giải thuật trên, ta chọn khoá đứng đầu làm chốt là dựa trên cơ sở này. Nhưng với cách chọn này, nếu dãy khoá có khuynh hướng đã theo thứ tự sắp xếp thì khả năng xấu nhất lại xuất hiện. Tuy nhiên nếu khuynh hướng này hay xuất hiện thì việc chọn khoá đang được đứng ở giữa dãy lại gặp thuận lợi.

Để dung hoà với cách chọn như trên, đồng thời cũng để kết hợp với một đề nghị sau này của Hoare là: *"chọn trung vị của một dãy khoá nhỏ hơn, thuộc dãy khoá cho, làm chốt"* R.C. Singleton đã đưa ra một cách chọn là:

Chọn K_q là "chốt" với K_q là trung vị của ba khoá K_l, $K_{\lfloor (l+r)/2 \rfloor}$ và K_r trong đó l và r là chỉ số của khoá đầu và khoá cuối của dãy cho. Các kiểu chọn khoá chốt còn được nhiều tác giả khác nữa đưa ra và cũng có nhiều kết quả đáng chú ý.

b) Vấn đề phối hợp với cách sắp xếp khác

Khi kích thước của các phân đoạn đã khá nhỏ, việc tiếp tục phân đoạn nữa theo QUICK-SORT thực ra sẽ không có lợi. Lúc đó sử dụng một phương pháp sắp xếp đơn giản lại tiện lợi hơn. Vì vậy, sắp xếp NHANH thường không tiến hành triệt để mà dừng lại ở lúc cần thiết để gọi tới một

phương pháp sắp xếp đơn giản, giao cho nó tiếp tục thực hiện sắp xếp với các phân đoạn nhỏ còn lại. Kunth (1974) có nêu: 9 có thể coi là kích thước giới hạn của phân đoạn để sau đó QUICK-SORT gọi tới phương pháp sắp xếp đơn.

Ngoài ra vấn đề chọn phân đoạn nào để xử lý tiếp theo cũng là vấn đề cần được xem xét tới. Tuy nhiên ta sẽ không đi sâu vào ở đây.

* Bây giờ ta xét sang việc đánh giá giải thuật QUICK-SORT. Do giải thuật là đệ qui nên ta áp dụng một cách tiếp cận khác một chút.

Gọi T(n) là thời gian thực hiện giải thuật ứng với một bảng n khoá, P(n) là thời gian để phân đoạn một bảng n khoá thành hai bảng con. Ta có thể viết:

$$T(n) = P(n) + T(j - LB) + T(UB-j)$$

Chú ý rằng P(n) = Cn với C là một hằng số.

Trường hợp xấu nhất xảy ra khi bảng khoá vốn đã có thứ tự sắp xếp: sau khi phân đoạn một trong hai bảng con là rỗng (j = LB hoặc j = UB).

Giả sử j = LB, ta có:

$$\begin{aligned} T_x(n) &= P(n) + T_x(0) + T_x(n-1) \qquad (T_x(0) = 0) \\ &= Cn + T_x(n-1) \\ &= Cn + C(n - 1) + T_x(n-2) \\ &\quad \ldots \\ &= \sum_{k=1}^{n} Ck + T_x(0) \\ &= C.\frac{n(n+1)}{2} = O(n^2) \end{aligned}$$

Trường hợp này QUICK-SORT không hơn gì các phương pháp đã nêu trước đây.

* Trường hợp tốt nhất xảy ra khi bảng luôn luôn được chia đôi, nghĩa là $j = \frac{LB + UB}{2}$. Lúc đó:

$$\begin{aligned} T_t &= P(n) + 2T_t(n/2) \\ &= Cn + 2T_t(n/2) \\ &= Cn + 2C(n/2) + 4T_t(n/4) = 2Cn_n + 2^2T_t(n/2) \\ &= Cn + 2C(n/2) + 4C(n/4) + 8T_t(n/8) = 3C_n + 2^3T_t(n/8) \\ &\quad \ldots \\ &= (\log_2 n)\, Cn + 2^{\log_2 n} T_t(1) \\ &= O(n\log_2 n). \end{aligned}$$

Việc xác định giá trị trung bình $T_{tb}(n)$ không còn đơn giản như hai trường hợp trên, nên ta sẽ không xét đến chi tiết. Kết quả mà ta cần ghi nhận là: người đã chứng minh được:

$$T_{tb}(n) = O(n\log_2 n)$$

Như vậy rõ ràng là, khi n khá lớn QUICK-SORT đã tỏ ra có hiệu lực hơn hẳn 3 phương pháp đã nêu.

9.4 Sắp xếp kiểu vun đống (Heap - sort)

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

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

Với phương pháp sắp xếp này, bảng khoá sẽ có cấu trúc cây nhị phân hoàn chỉnh và được lưu trữ kế tiếp trong máy (xem 6.2.2.).

9.4.1 Giới thiệu phương pháp

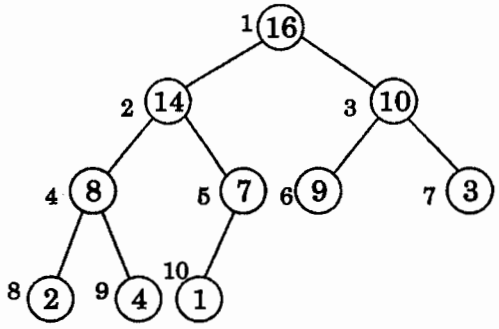
Cấu trúc cây đặc biệt được sử dụng trong phương pháp này là cấu trúc đống.

a) Định nghĩa "đống"

Đống là một cây nhị phân hoàn chỉnh mà mỗi nút được gán một giá trị khoá sao cho khoá ở nút cha bao giờ cũng lớn hơn khoá ở nút con nó.

Đống được lưu trữ trong máy bởi một vectơ K mà K[i] thì lưu trữ giá trị khoá ở nút thứ i trên cây nhị phân hoàn chỉnh (theo cách đánh số thứ tự khi lưu trữ kế tiếp, đã nói ở 6.2.2.)

Ví dụ: Cây ở hình 9.2. dưới đây là một đống với vectơ tương ứng K biểu diễn nó trong máy (lưu trữ trong máy).



Hình 9.3

K	16	14	10	8	7	9	3	2	4	1
	1	2	3	4	5	6	7	8	9	10

Ta thấy ngay rằng: Nếu ta có một đống thì giá trị khoá ở nút gốc (mà ta sẽ gọi là "đỉnh đống" - nút được đánh số 1) chính là giá trị khoá lớn nhất trong dãy khoá ứng với đống đó.

Từ đó có thể hiểu: việc chọn ra khoá lớn nhất trong dãy khoá cho sẽ thuận lợi hơn nếu ta tạo được ra một đống ứng với dãy khoá này.

Ta đã biết: một cây nhị phân hoàn chỉnh thường được biểu diễn trong máy dưới dạng một vectơ.

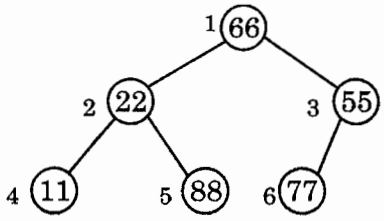
Vậy thì ngược lại: một vectơ biểu diễn một dãy giá trị khoá trong máy cũng có thể coi nó như dạng biểu diễn của một cây nhị phân hoàn chỉnh mà mỗi nút có gán một giá trị khoá tương ứng của dãy.

Dĩ nhiên cây này có thể chưa phải là đống.

Ví dụ: Nếu có dãy khoá biểu diễn trong máy dưới dạng vectơ V như sau:

V	66	22	55	11	88	77
	1	2	3	4	5	6

thì có thể coi V như là vectơ biểu diễn trong máy của cây nhị phân hoàn chỉnh có dạng



Hình 9.4

Rõ ràng cây này chưa phải là đống.

b) Phép tạo đống

Xét một cây nhị phân hoàn chỉnh có n nút, mà mỗi nút đã được gán một giá trị khoá. Cây này chưa phải là đống.

Muốn tạo nó thành đống thì làm thế nào?

Trước hết ta có một số nhận xét như sau:

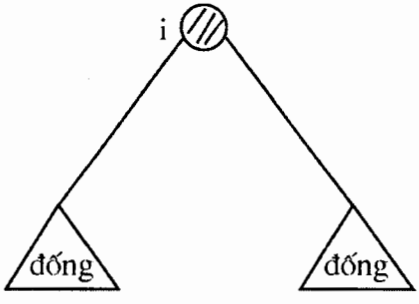
- Nếu một cây nhị phân hoàn chỉnh đã là đống thì các cây con của các nút (nếu có) cũng là cây nhị phân hoàn chỉnh và cũng là đống.

- Trên cây nhị phân hoàn chỉnh có n nút thì chỉ có $\lfloor n/2 \rfloor$ nút được là "cha" thôi.

- Một nút lá bao giờ cũng có thể coi là đống.

Từ đó ta thấy: có thể thực hiện phép tạo đống theo kiểu "từ đáy lên" (bottom up), nghĩa là từ các cây con mà gốc có số thứ tự là: $\lfloor n/2 \rfloor$; $\lfloor n/2 \rfloor$ -1; $\lfloor n/2 \rfloor$-2; ...; 1. Ta sẽ bắt đầu từ những cây mà con của nó là lá, nghĩa là đã là đống. Do đó việc tạo đống này sẽ chỉ cần một phép xử lý chung, mà ta có thể phát biểu như sau:

"Hãy tạo thành đống cho một cây nhị phân hoàn chỉnh có gốc được đánh số thứ tự là i, và gốc có 2 cây con đã là đống rồi". Có thể minh hoạ dạng cây này như hình 9.5



Hình 9.5

Ta có thể thể hiện phép xử lý này bằng giải thuật sau:

Procedure ADJUST (i,n);

{Ở đây vectơ K với n phần tử được coi như vectơ lưu trữ của một cây nhị phân hoàn chỉnh có n nút}

1. KEY := K[i]; {KEY nhận giá trị khoá ở nút gốc i}
   j := 2 * i; {j ghi nhận số thứ tự nút con trái của nút i}
2. **while** j ≤ n **do begin**
3. **if** j < n **and** K[j] < K[j +1] **then** j:= j +1;

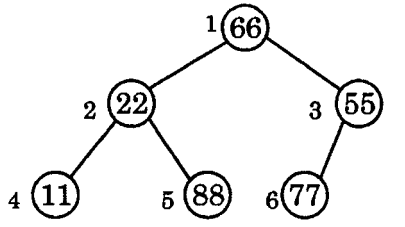
{Nếu khoá con phải lớn hơn thì j ghi nhận số thứ tự của nó}

4. **if** KEY > K[j] **then begin**
   K[⌊j/2⌋] := KEY;
   return
   end; {đây là trường hợp khoá cha lớn hơn khoá con}
5. K[⌊j/2⌋] := K[j]; {Đưa khoá con lớn lên}
   j := 2 *j
   end;
6. K[⌊j/2⌋] := KEY;
7. **return**

Với giải thuật ADJUST này, việc tạo thành đống cho một cây nhị phân hoàn chỉnh có n nút sẽ được thực hiện bởi:

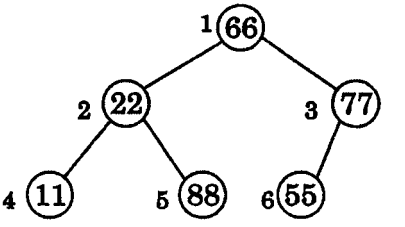
for i :=⌊n/2⌋ **down to** 1 **do call** ADJUST(i,n);

Hình 9.6 sau đây minh hoạ diễn biến của dạng cây ở hình 9.4, trong quá trình tạo nó trở thành đống.



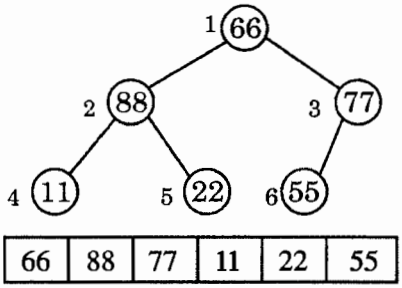
66	22	55	11	88	77
1	2	3	4	5	6

a) cấu trúc dữ liệu và cấu trúc lưu trữ lúc ban đầu

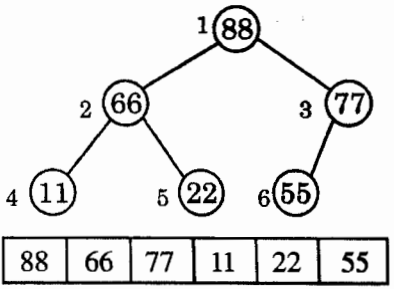


66	22	77	11	88	55

b) Sau khi thực hiện ADJUSST (3,6)



c) Sau khi thực hiện ADJUST (2,6)



d) Sau khi thực hiện ADJUST(1,6): Cây đã là đống, khoá lớn nhất đang ở đỉnh đống

Hình 9.6

c) Sắp xếp kiểu vun đống

Sắp xếp kiểu vun đống là một cải tiến của phương pháp sắp xếp kiểu lựa chọn. Tuy nhiên để chọn ra số lớn nhất, ở đây người ta đã dựa vào cấu trúc đống và để sắp xếp theo thứ tự tăng dần của các giá trị khoá là số, như ta đã qui ước, thì khoá lớn nhất sẽ được xếp vào cuối dãy, nghĩa là nó được đổi chỗ với khoá đang ở "đáy đống", và sau phép đổi chỗ này một khoá trong dãy đã vào đúng vị trí của nó trong sắp xếp. Nếu không kể tới khoá này thì phần còn lại của dãy khoá ứng với một cây nhị phân hoàn chỉnh, với số lượng khoá nhỏ hơn 1, sẽ không còn là đống nữa, ta lại gặp bài toán tạo đống mới cho cây này (ta sẽ gọi là "vun đống") và lại thực hiện tiếp phép đổi chỗ giữa khoá ở đỉnh đống và khoá ở đáy đống tương tự như đã làm.v.v... Cho tới khi cây chỉ còn là 1 nút thì các khoá đã được xếp vào đúng vị trí của nó trong sắp xếp.

Như vậy sắp xếp kiểu vun đống bao gồm 2 giai đoạn:

1- Giai đoạn tạo đống ban đầu (như đã nêu ở mục b)

2- Giai đoạn sắp xếp, bao gồm 2 bước:

- Vun đống
- Đổi chỗ

được thực hiện (n-1) lần.

Thủ tục sắp xếp kiểu vun đống được thể hiện bởi giải thuật sau:

Procedure HEAP-SORT (K, n);

1. {Tạo đống ban đầu}

for i:=⌊n/2⌋ **down to** 1 **do**

call ADJUST (i,n);

2. {Sắp xếp}

for i := n - 1 **down to** 1 **do begin**

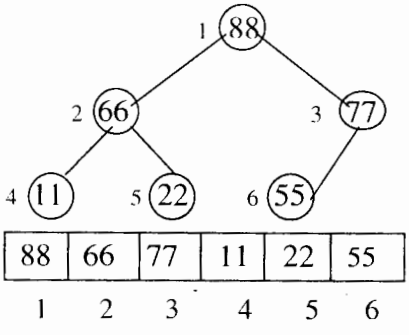
K[1] ↔ K[i + 1];

call ADJUST (1,i)

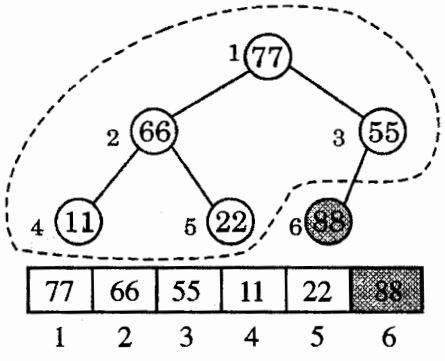
end

3. **return**

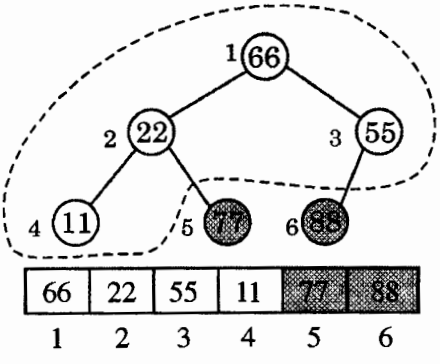
Hình 9.5 sau đây minh hoạ giai đoạn sắp xếp đối với đống đã được tạo trong ví dụ ở mục b.



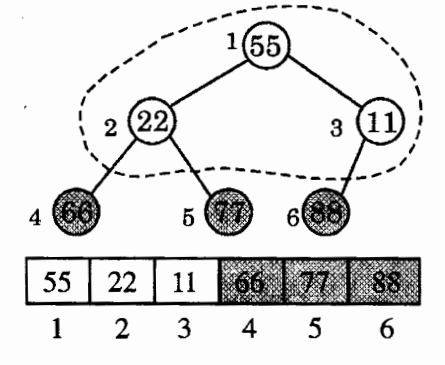
a) Đống đã được tạo ra sau khi thực hiện giai đoạn 1.



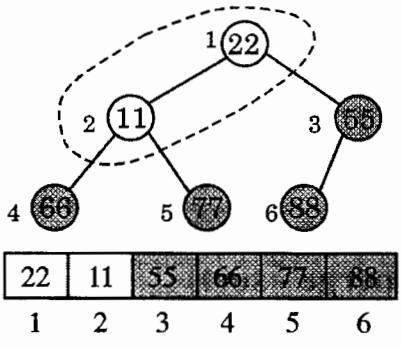
b) Sau khi đổi chỗ và thực hiện ADJUST (1,5)



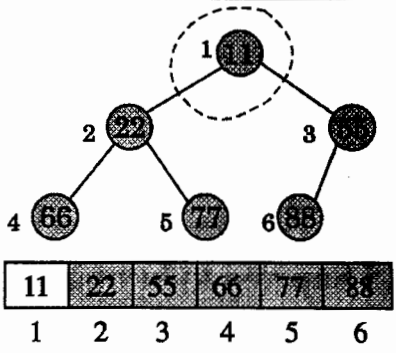
c) Sau khi đổi chỗ và thực hiện ADJUST (1,4)



d) Sau khi đổi chỗ và thực hiện ADJUST (1,3)



e) Sau khi đổi chỗ và thực hiện ADJUST (1,2)



f) Sau khi đổi chỗ và thực hiện ADJUST (1,1) lúc này các khoá đã được sắp xếp.

Chú ý:

* Hình  là ứng với các khoá đã vào đúng vị trí của nó trong sắp xếp.

* Dấu ----- bao quanh các nút ứng với cây đã được vun đống.

Hình 9.7

9.4.3. Nhận xét và đánh giá

Đối với HEAP-SORT, người ta cũng chứng minh được thời gian thực hiện trung bình $T_{tb}(n) = O(n\log_2 n)$

Ở đây ta sẽ chú ý đặc biệt tới việc đánh giá thời gian thực hiện giải thuật trong trường hợp xấu nhất.

Có thể thấy rằng ở giai đoạn 1 (tạo đống ban đầu) có $\lfloor n/2 \rfloor$ lần gọi thực hiện ADJUST(i,n). Còn ở giai đoạn 2 (sắp xếp) thì phải gọi thực hiện ADJUST(1,i) (n-1) lần. Như vậy có thể coi như phải gọi khoảng $\frac{3}{2}n$ lần thực hiện giải thuật ADJUST mà cây được xét ứng với ADJUST thì nhiều nhất cũng chỉ có n nút, nghĩa là chiều cao của cây lớn nhất cũng chỉ xấp xỉ $\log_2 n$.

Số lượng phép so sánh giá trị khoá, khi thực hiện giải thuật ADJUST, cùng lắm cũng chỉ bằng chiều cao của cây tương ứng.

Cho nên, có thể nói rằng: cùng lắm thì số lượng phép so sánh cũng chỉ xấp xỉ $\frac{3}{2}n\log_2 n$ (coi $\log_2 n$ là ứng với chiều cao của cây nhị phân hoàn chỉnh có n nút).

Từ đó suy ra:

$$T_x(n) = 0\ (n\log_2 n)$$

đây chính là ưu điểm của HEAP-SORT nếu so với QUICK-SORT.

Còn về thời gian thực hiện trung bình thì, như ta đã nêu, cả hai phương pháp đều như nhau.

9.5 Sắp xếp kiểu hoà nhập (merge - sort)

Bây giờ ta xét tới một phương pháp có vai trò khá đặc biệt do ở chỗ nó dựa trên một phép xử lý đơn giản hơn sắp xếp để thực hiện sắp xếp, đó là phép hoà nhập.

9.5.1 Phép hoà nhập hai đường (two - way merge)

Giả sử A là một dãy khoá, bao gồm r phần tử, đã được sắp xếp (mà ta sẽ gọi là một *"mạch"* - run) và B là một dãy khác cũng đã được sắp xếp, gồm s phần tử.

Phép hợp nhất các khoá của A và B thành một "mạch" mới bao gồm n = r + s phần tử được gọi là phép hoà nhập hai đường.

Một cách đơn giản có thể nghĩ ngay rằng: lưu trữ các phần tử của A và B trong một vectơ gồm n phần tử sau đó dùng một phương pháp sắp xếp nào đó để sắp xếp lại các phần tử của dãy mới này.

Tuy nhiên, cách làm này đã không tận dụng được tính "đã được sắp" của A và B!

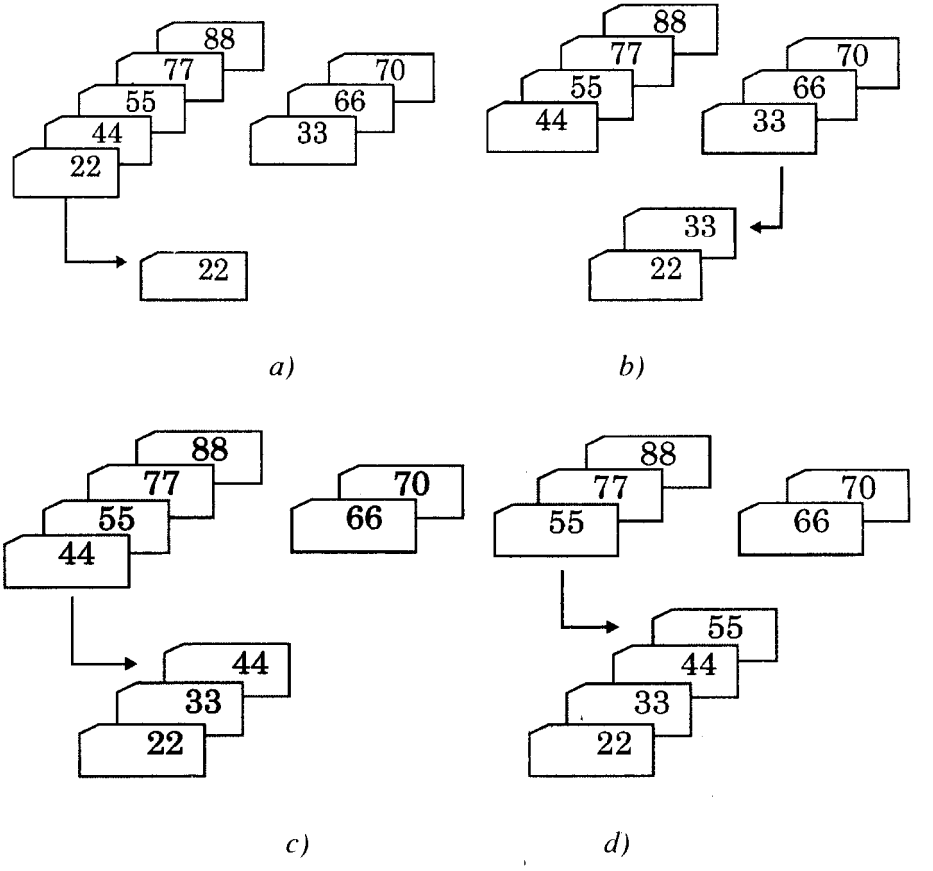
Giải thuật MERGING sau đây sẽ thể hiện một cách xử lý hiệu quả hơn.

Ý cơ bản của nó như sau:

So sánh hai khoá nhỏ nhất của A và B chọn khoá nhỏ hơn để đưa vào miền sắp xếp (một vectơ lưu trữ, có kích thước bằng n) và đặt vào vị trí thích hợp. Khoá được chọn từ đấy bị loại ra khỏi "mạch" chứa nó.

Quá trình như vậy cứ tiếp tục cho tới khi một trong hai mạch đã cạn. Lúc đó chỉ cần chuyển toàn bộ "phần đuôi" của mạch còn lại (nếu có) ra miền sắp xếp là xong.

Có thể minh hoạ cách làm này bởi hình ảnh dưới đây (Hình 9.8).



Hình 9.8

Bây giờ ta xét tới giải thuật MERGING. Các mạch A, B và mạch tổng hợp C được lưu trữ dưới dạng vectơ, với các chỉ số lần lượt là i, j, k.

Việc so sánh giá trị giữa A[i] và B[j] sẽ xác định phần tử của mạch nào được gán vào C[k].

Procedure MERGING(A, r, B, s, C);

1. {Khởi tạo các chỉ số}
   i := 1; j := 1; k := 1;
2. {So sánh để chọn phần tử nhỏ hơn}
   while i ≤ r **and** j ≤ s **do**
3. **if** A[i] < A[j] **then begin**
   C[k] := A[i];

```
            i := i+1;
            k := k +1;
                        end;
    else begin
            C[k] := B[j];
            j := j + 1;
            k := k +1
        end;
```

5. {Một trong hai mạch đã cạn}

```
    if i > r then
            for t := 0 to s - j do
                    C[k + t] := B[j + t];
            else
            for t := 0 to r - i do
                    C[k + t] := A[i + t]
```

6. **return**

Trường hợp biên dưới (chỉ số của phần tử đầu) của A, B, C không phải là 1 thì giải thuật hợp nhất hai đường vẫn tương tự như trên với một vài thay đổi nhỏ. Giải thuật tương ứng sẽ như sau:

Procedure MERGE(A, r, LBA, B, s, LBB, C, LBC);

{LBA, LBB, LBC là các biên dưới của A, B, C. Lúc đó biên trên của A sẽ là UBA = LBA + r - 1 và của B sẽ là UBB = LBB + s - 1}

1. i := LBA; j := LBB; k := LBC;
   UBA := LBA + r - 1, UBB := LBB + s - 1;

2. {Tương tự như giải thuật MERGING chỉ khác ở chỗ: r được thay bằng UBA và s được thay bằng UBB}

3. {Như ở MERGING}

4. {Như ở MERGING}

5. {Tương tự như giải thuật MERGING chỉ khác ở chỗ r thay bằng UBA và s thay bằng UBB}

4. **return**

9.5.2 Sắp xếp kiểu hoà nhập hai đường trực tiếp (straight two-way merge)

Từ phép hoà nhập hai đường người ta đã triển khai thành một số phương pháp sắp xếp mới.

Ta hãy xuất phát từ một nhận xét:

Mỗi khoá trong dãy khoá (bảng khoá) cho, có thể được coi là một mạch có độ dài bằng 1.

Nếu hoà nhập hai mạch như vậy ta sẽ được một mạch mới, có độ dài 2. Lại hoà nhập hai mạch có độ dài 2 ta sẽ được một mạch có độ dài 4; và cứ tương tự như thế, cuối cùng ta sẽ được một mạch có độ dài n, đó chính là dãy khoá đã được sắp xếp hoàn toàn.

Sắp xếp kiểu hoà nhập hai đường trực tiếp đã phản ánh ý tưởng trên với các cặp mạch liền kề nhau.

Với dãy khoá đã nêu làm ví dụ ở trên, thì quá trình thực hiện sẽ như sau:

	[42]	[23]	[74]	[11]	[65]	[58]	[94]	[36]	[99]	[87]
lượt 1	[23	42]	[11	74]	[58	65]	[36	94]	[87	99]
- 2	[11	23	42	74]	[36	58	65	94]	[87	99]
- 3	[11	23	36	42	58	65	74	94]	[87	99]
- 4	[11	23	36	42	58	65	74	87	96	99]

(Các khoá nằm trong dấu [ ] là ứng với một mạch).

Giải thuật dưới đây thực hiện một lượt hoà nhập 2 mạch với độ dài l (có thể một trong 2 mạch có độ dài nhỏ hơn l) để cho một mạch lớn hơn.

Ở đây K là dãy khoá cho, có n phần tử K[1], K[2], ..., K[n]. Các dãy con trong K đã là các mạch có độ dài bằng l, trừ dãy con cuối cùng có thể có độ dài bé lớn hơn l.

Nếu Q là số cặp mạch có đọ dài l thì $Q = \lfloor n/2 * l \rfloor$

Gọi $S = 2 * l * Q$ thì S là số các phần tử của Q cặp mạch. Từ đó suy ra $R = n - S$ là số các phần tử còn lại của dãy K.

Procedure MPASS(K, n, l, X);

{X là một vectơ có n phần tử được dùng để lần lượt chứa các phần tử của K sau khi hoà nhập}.

1. Q := n div (2* l);

```
    S := 2 * l * Q;
    R := n - S;
2. {Hoà nhập từng cặp mạch }
    for j := 1 to Q do begin
            LB := 1 + (2 * j - 2) * l;
{Xác định biên dưới của mạch thứ nhất}
            call MERGE (K, l, LB, K, l, LB + l, X, LB);
                    end
3. {Chỉ còn một mạch}
    if R ≤ l then
            for j:= 1 to R do
                    X[S + j] := A[S + j]
4. {Còn 2 mạch nhưng một mạch có độ dài nhỏ hơn l}
    else
            call MERGE(K, l, S+1, K, R-l, l +S+1, X, S+l);
5.  return
```

Thủ tục MPASS này sẽ được gọi tới trong thủ tục thực hiện sắp xếp kiểu hoà nhập hai đường trực tiếp.

```
Procedure STRAIGHT-MSORT(K,n);
1. {Khởi tạo số phần tử trong mạch}
    l := 1;
2.  while l <n do begin
            call MPASS (K, n, l, X);
{Thực hiện hoà nhập và chuyển các phần tử vào X}
3.          Call MPASS (X, n, 2*l, K);
{Thực hiện hoà nhập và chuyển các phần tử vào K}
4.          l := 4 * l;
            end;
5.  return
```

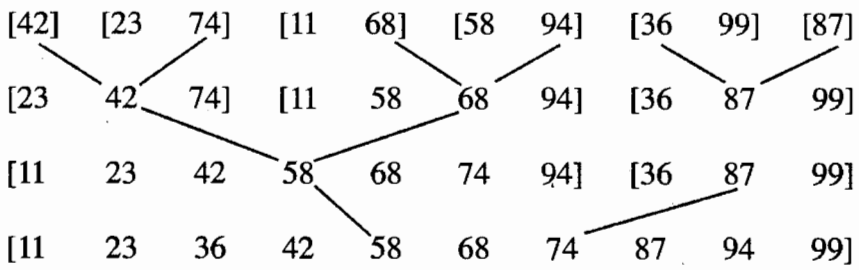
Sau khi sắp xếp xong các phần tử của dãy khoá vẫn ở chỗ cũ.

Chú ý:

1) Phương pháp trên dựa vào phép hoà nhập hai mạch kế cận nhau, nhưng cách chọn này không phải là duy nhất, chẳng hạn có thể chọn hai mạch con theo hướng tiến từ hai đầu vào. Từ những cách chọn khác nhau

có thể đưa tới những giải thuật sắp xếp kiểu hoà nhập hai đường trực tiếp khác nhau.

2) Hoà nhập kiểu hai đường trực tiếp đã không thể tận dụng được khả năng "vốn đã được sắp xếp tự nhiên rồi" của các khoá trong bảng lúc ban đầu. Ở mỗi lượt nó thực hiện hoà nhập hai mạch với độ dài đã ấn định: ở lượt thứ k thì độ dài hai mạch được hoà nhập là 2^{k-1}, cho dù lúc đó mạch được hoà nhập có thể có độ dài lớn hơn cũng thế. Như ta đã biết khi hoà nhập không nhất thiết hai mạch phải có kích thước bằng nhau cho nên nếu triệt để lợi dụng các mạch vốn đã hình thành sẵn rồi (mà ta có thể gọi là một *mạch tự nhiên*), với kích thước càng lớn thì sắp xếp có khả năng kết thúc với số lượt ít hơn. Đây cũng là ý cơ bản của một phương pháp sắp xếp khác gọi là *sắp xếp kiểu hoà nhập hai đường tự nhiên* (natural two - way merge). Với dãy khoá làm ví dụ ở trên, ngay từ đầu ta đã có 6 mạch, phần lớn đã có độ dài hai.



Rõ ràng số lượt ở đây đã giảm đi 1.

9.5.3 Phân tích đánh giá

Có thể thấy ngay trong phương pháp sắp xếp kiểu hoà nhập, số lượng phép chuyển chỗ thường vượt số lượng các phép so sánh. Ví dụ ở thủ tục MERGE. Trong câu lệnh **while** ứng với một phép so sánh thì có một phép chuyển chỗ, nhưng nếu một mạch nào đó kết thúc sớm thì cả phần đuôi của mạch còn lại được chuyển sang miền sắp xếp mà không tương ứng với một phép so sánh nào nữa. Vì vậy đối với phương pháp này, ta lại chọn phép tích cực là phép chuyển chỗ để làm căn cứ đánh giá thời gian thực hiện giải thuật.

Ta thấy ở lượt sắp xếp nào (thủ tục MPASS) thì toàn bộ các khoá (bản ghi) cũng được chuyển sang miền mới (từ X sang Y hoặc từ Y sang X) như vậy chi phí thời gian cho một lượt có cấp O(n). Ngoài ra nếu để ý sẽ thấy: số lượt gọi MPASS trong giải thuật STRAIGHT -MSORT chính là $\lceil \log_2 n \rceil$. Sở dĩ như vậy là vì: ở lượt 1 kích thước của bảng con là $1 = 2^0$. Ở lượt i kích

thước bảng con sẽ là 2^{i-1} mà sau lượt gọi cuối cùng thì bảng con đã có kích thước bằng n.

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

Một đặc điểm khác và cũng là nhược điểm của phương pháp sắp xếp kiểu hoà nhập là chi phí về không gian khá lớn. Nó đòi hỏi tới 2n phần tử nhớ, gấp đôi so với phương pháp thông thường. Do đó người ta thường sử dụng phương pháp này khi sắp xếp ngoài, đối với các tệp. Trong chương 11 ta sẽ xét thêm và cũng thấy rõ hơn hiệu lực của nó.

9.6 Những nhận xét cuối cùng

Qua những giải thuật nêu trên ta đã thấy rõ: cùng một mục đích sắp xếp như nhau mà có rất nhiều phương pháp và kỹ thuật giải quyết khác nhau. Cấu trúc dữ liệu được lựa chọn để hình dung đối tượng của sắp xếp đã ảnh hưởng rất sát tới giải thuật xử lý. Các phương pháp sắp xếp đơn giản đã thể hiện ba kỹ thuật cơ sở của sắp xếp (dựa vào phép so sánh giá trị khoá) cấp độ lớn của thời gian thực hiện chung là $O(n^2)$, vì vậy chỉ nên sử dụng chúng khi n nhỏ. Các giải thuật cải tiến như QUICK_SORT, HEAP_SORT đã đạt được hiệu quả cao: $O(n\log_2 n)$ thường được sử dụng khi n lớn. MERGE_SORT cũng không kém hiệu lực về thời gian thực hiện nhưng về không gian thì đòi hỏi của nó không thích nghi với sắp xếp trong. Nếu bảng cần sắp xếp vốn có khuynh hướng hầu như "đã được sắp sẵn" thì QUICK_SORT lại không nên dùng. Nhưng nếu ban đầu bảng có khuynh hướng ít nhiều có thứ tự ngược với thứ tự sắp xếp thì HEAP_SORT lại tỏ ra thuận lợi. Việc khẳng định một kỹ thuật sắp xếp nào đó nói trên luôn luôn tốt hơn mọi kỹ thuật khác là một điều không nên. Việc chọn một phương pháp sắp xếp thích hợp thường tuỳ thuộc vào từng yêu cầu, từng điều kiện cụ thể.

BÀI TẬP CHƯƠNG 9

9.1. Khi quyết toán tiền điện của một quí, sở phân phối điện có trong tay hai tệp dữ liệu:

Tệp một bao gồm các bản ghi tương ứng với phiếu thanh toán tiền điện mà mỗi chủ hộ trong thành phố phải trả.

Tệp hai bao gồm các bản ghi tương ứng với phiếu ghi nhận tiền điện đã trả của từng chủ hộ.

Sở muốn biết chủ hộ nào chưa trả hết tiền điện. Liệu "sắp xếp" có giúp gì cho việc giải quyết yêu cầu này không?

9.2. Cho dãy từ khoá

50 08 34 06 98 17 83 25 66 42 21 59 62 71 85 76

Hãy minh hoạ ba phương pháp sắp xếp cơ bản đã nêu, qua dãy khoá này theo thứ tự tăng dần, theo thứ tự giảm dần.

9.3. Trong các giải thuật đã nêu, ta đã coi khoá đại diện cho bản ghi, nên việc đổi chỗ các bản ghi đã được thay bằng việc đổi chỗ khoá tương ứng.

Hãy viết lại ba giải thuật

SELECT_SORT

INSERT_SORT

BUBBLE_SORT

trong đó việc đổi chỗ thực sự tiến hành với các bản ghi.

9.4. Hãy tính số lượng phép chuyển chỗ các bản ghi trong hai trường hợp: thuận lợi nhất và xấu nhất đối với ba giải thuật sắp xếp cơ bản (kích thước của bảng cần sắp xếp là n).

9.5. Một giải thuật sắp xếp được gọi là *ổn định* (stable) nếu thứ tự tương đối giữa các bản ghi có khoá bằng nhau vẫn không thay đổi sau khi thực hiện sắp xếp. Hãy xem các giải thuật SELECT_SORT và BUBBLE_SORT có thể sửa đổi để áp dụng cho dãy khoá có chứa khoá bằng nhau không? Lúc đó chúng có ổn định không?

9.6. Hãy sửa lại ba giải thuật sắp xếp cơ bản để thực hiện sắp xếp theo thứ tự giảm dần.

9.7. Cho dãy số sau được lưu trữ trong bộ nhớ dưới dạng một vectơ:

44, 30, 50, 22, 60, 55, 77, 65.

a) Nếu coi như vectơ này biểu diễn một cây nhị phân hoàn chỉnh thì cây đó có dạng thế nào?

b) Cây này không phải là "đống" nhưng có thể tạo thành đống được.

Muốn vậy phải dựa vào giải thuật nào? Minh hoạ diễn biến của cây trong quá trình này.

c) Nếu dựa vào đống đó để thực hiện sắp xếp dãy số đã cho, thì còn phải làm gì nữa? Minh hoạ các bước thực hiện và thuyết minh cách làm.

9.8. Với dãy khoá cho ở bài 9.2. hãy minh hoạ hai phương pháp sắp xếp kiểu phân đoạn và kiểu vun đống.

9.9. Nhận xét và so sánh về cấu trúc dữ liệu và cấu trúc lưu trữ của dữ liệu ứng với hai phương pháp sắp xếp kiểu phân đoạn và kiểu vun đống.

9.10. Nếu áp dụng cách chọn "chốt" theo kiểu Singleton thì đối với dãy khoá làm ví dụ nêu trong bài, "chốt" sẽ là khoá nào? Hãy thực hiện sắp xếp với cách chọn khoá "chốt" này với dãy đó.

9.11. Nếu thực hiện sắp xếp theo thứ tự giảm dần thì đống sẽ được định nghĩa thế nào cho thích hợp?

9.12. Hãy giải thích tại sao khi hoà nhập hai đường với S mạch ban đầu thì số lượt sẽ là $\lceil \log_2 S \rceil$.

9.13. Có thể phát triển phép hoà nhập hai đường để hoà nhập k đường được không (k >2).

9.14. Hãy thực hiện sắp xếp kiểu hoà nhập hai đường tự nhiên với dãy khoá cho ở bài tập 9.2.

Chương 10

TÌM KIẾM

10.1 Bài toán tìm kiếm

Tìm kiếm (Searching) là một đòi hỏi rất thường xuyên trong đời sống hàng ngày cũng như trong xử lý tin học. Ngay trong chương trước ta cũng thấy xuất hiện các yêu cầu về tìm kiếm. Tuy nhiên, lúc đó vấn đề này đã được xét và giải quyết trong mối quan hệ mật thiết với phép xử lý chính, đó là phép sắp xếp. Còn bây giờ, trong chương này, bài toán tìm kiếm sẽ được đặt ra một cách độc lập và tổng quát, không liên quan đến mục đích xử lý cụ thể nào khác. Ta có thể phát biểu như sau:

"Cho một bảng gồm n bản ghi R_1, R_2, ..., R_n. Mỗi bản ghi R_i ($1 \le i \le n$) tương ứng với một khoá k_i. Hãy tìm bản ghi có giá trị khoá tương ứng bằng X cho trước".

X được gọi là *khoá tìm kiếm hay đối trị tìm kiếm* (argument).

Công việc tìm kiếm sẽ hoàn thành khi có một trong hai tình huống sau đây xảy ra:

1) Tìm được bản ghi có giá trị khoá tương ứng bằng X, lúc đó ta nói: phép *tìm kiếm được thoả* (successfull).

2) Không tìm thấy được bản ghi nào có giá trị khoá bằng X cả: Phép *tìm kiếm không thoả* (unsuccessfull). Sau một phép tìm kiếm không thoả có khi xuất hiện yêu cầu bổ sung thêm bản ghi mới có khoá bằng X vào bảng. Giải thuật thể hiện cả yêu cầu này được gọi là giải thuật "tìm kiếm có bổ sung".

Tương tự như sắp xếp, khoá của mỗi bản ghi chính là đặc điểm nhận biết của bản ghi đó trong tìm kiếm, ta sẽ coi nó như đại diện cho bản ghi ấy và trong các giải thuật, trong ví dụ ta cũng chỉ nói tới khoá. Để cho tiện, ta cũng coi các khoá $k_i (1 \le i \le n)$ là các số khác nhau. Ở đây ta cũng chỉ xét tới các phương pháp tìm kiếm cơ bản và phổ dụng, đối với dữ liệu ở bộ nhớ trong nghĩa là *tìm kiếm trong*, còn *tìm kiếm ngoài* sẽ được xét ở chương sau.

10.2 Tìm kiếm tuần tự (sequential searching)

10.2.1 Tìm kiếm tuần tự là kỹ thuật tìm kiếm rất đơn giản và cổ điển

Nội dung có thể tóm tắt như sau:

"Bắt đầu từ bản ghi thứ nhất, lần lượt so sánh khoá tìm kiếm với khoá tương ứng của các bản ghi trong bảng, cho tới khi tìm được bản ghi mong muốn hoặc đã hết bảng mà chưa thấy".

Sau đây là giải thuật:

Function: SEQUEN-SEARCH (k, n, X)

{Cho dãy khoá k gồm n phần tử. Thủ tục này sẽ được tìm kiếm trong dãy xem có khoá nào bằng X không. Nếu có nó sẽ đưa ra chỉ số của khoá ấy, nếu không nó sẽ đưa ra giá trị 0. Trong thủ tục này có sử dụng một khoá phụ k_{n+1} mà giá trị của nó chính là X}

1) {Khởi đầu}

i := 1; k[n + 1] := X;

2) {Tìm khoá trong dãy}

while k[i] ≠ X **do** i := i+1;

3) {Tìm thấy hay không?}

if i = n + 1 **then return** (o)

else return (i)

10.2.2. Ở đây, để đánh giá hiệu lực của phép tìm kiếm ta cũng dựa vào số lượng các phép so sánh. Ta thấy với giải thuật trên thuận lợi thì chỉ cần 1 phép so sánh: $C_{min} = 1$; còn xấu nhất thì $C_{max} = n + 1$. Nếu giả sử hiện tượng khoá tìm kiếm trùng với một khoá nào đó của bảng là đồng khả năng thì $C_{tb} = \frac{n+1}{2}$. Tóm lại cả trường hợp xấu nhất cũng như trung bình, cấp độ lớn của thời gian thực hiện giải thuật trên là O(n).

Nhưng nếu xác suất để xuất hiện $k_i = X$ mà $p_i \neq \frac{1}{n}$ thì sao? Lúc đó ta sẽ có $C_{tb} = 1 * p_1 + 2 * p_2 + ...+ n * p_n$

với: $\sum_{i=1}^{n} p_i = 1$

Rõ ràng là nếu $p_1 \geq p_2 \geq ... \geq p_n$ thì thời gian trung bình sẽ nhỏ hơn. Nhưng muốn như vậy thì phải sắp xếp trước!

10.3 Tìm kiếm nhị phân (Binary searching)

10.3.1. Tìm kiếm nhị phân là một phương pháp tìm kiếm khá thông dụng. Nó tương tự như cách thức ta đã làm khi tra tìm số điện thoại của một cơ quan, trong bảng danh mục điện thoại hay khi tìm một từ trong từ điển. Chỉ có một điều hơi khác là trong các công việc trên để so sánh với khoá tìm kiếm ta chọn hú hoạ một phần tử, còn với phép tìm kiếm nhị phân thì nó luôn luôn chọn khoá "ở giữa" dãy khoá đang xét để thực hiện so sánh với khoá tìm kiếm. Giả sử dãy khoá đang xét là k_l, ..., k_r thì khoá ở giữa dãy sẽ là k_i với $i = \left\lfloor \frac{l+r}{2} \right\rfloor$. Tìm kiếm sẽ kết thúc nếu $X = k_i$. Nếu $X < k_i$ tìm kiếm sẽ được thực hiện tiếp với k_l,..., k_{i-1}; còn nếu $X > k_i$ tìm kiếm lại được làm với k_{i+1},..., k_r. Với dãy khoá sau, một kỹ thuật tương tự lại được sử dụng. Quá trình tìm kiếm được tiếp tục khi tìm thấy khoá mong muốn hoặc dãy khoá xét đó trở nên rỗng (không thấy).

Giải thuật sau đây thể hiện phép tìm kiếm này.

Function BINARY_SEARCH (k, n, X)

{Cho dãy k gồm n khoá đã được sắp xếp theo thứ tự tăng dần. Giải thuật này tìm xem trong dãy có khoá nào bằng giá trị X hay không. Ở đây dùng các biến *l*, r, m để ghi nhận chỉ số của phần tử đầu, phần tử cuối và phần tử giữa của dãy khoá k. Nếu phép tìm kiếm được thoả, giá trị cho ra là chỉ số của khoá đã tìm thấy, nếu không thì cho ra giá trị 0}.

1. {Khởi đầu}
   l := 1;
   r := n;
2. {Tìm}
   while *l* ≤ r **do begin**
3. {Tính chỉ số giữa}
   m := ⌊(*l*+r)/2⌋;
4. {So sánh}
   if X < k[m] **then** r := m-1
   else
   if X > k[m] **then** *l* := m+1
   else return (m)
   end
5. {Tìm kiếm không thoả mãn}
   return(0)

*Ví dụ: Với dãy khoá

11 23 36 42 58 65 74 87 94 99

a) Nếu X = 23: phép tìm kiếm được thoả mãn và các bước sẽ như sau:

[11 23 36 42 **58** 65 74 87 94 99]

[11 **23** 36 42]

b) Nếu X = 71: phép tìm kiếm không thoả

[11 23 36 42 **58** 65 74 87 94 99]

[65 74 **87** 94 99]

[**65** 74]

[**74**]

(Ở đấy dấu [ứng với *l*, dấu] ứng với r, dấu – ứng với m)

Giải thuật tìm kiếm nhị phân có thể viết dưới dạng đệ quy như sau:

Function RBINARY_SEARCH (*l*, r, k, X)

{*l*, r là chỉ số dưới và chỉ số trên của dãy k, m vẫn là chỉ số giữa. Ở đây dùng thêm biến nguyên LOC để đưa ra chỉ số ứng với khoá cần tìm, nếu tìm kiếm không thoả LOC có giá trị 0}

```
1- if l> r then LOC := 0
          else m := ⌊(l+ r)/2⌋;
   if x < k[m]
          then LOC := RBINARY_SEARCH (l, m-1, k, X)
          else if X> k[m]
                 then
                  LOC := RBINARY_SEARCH (m+1,r,k, X)
                 else LOC := m;
2- return (LOC)
```

10.3.2 Phân tích đánh giá

Ta thấy số lượng phép toán so sánh phụ thuộc vào X. Với giải thuật đệ quy nêu trên, trường hợp thuận lợi nhất đối với dãy khoá $k_1,..., k_n$ (mà lời gọi sẽ là: POS := RBINARY-SEARCH (1, n, k, X)) là X = k[⌊(n+1)/2⌋] nghĩa là chỉ cần một phép so sánh, lúc đó $T_t(n) = O(1)$. Trường hợp xấu nhất xét có hơi phức tạp hơn. Giả sử ta gọi w(r-*l*+1) là hàm biểu thị số lượng phép so sánh trong trường hợp xấu nhất ứng với một phép gọi RBINARY_SEARCH (*l*, r, k, X) và đặt n = r - *l* + 1 (ứng với dãy khoá mà

$l = 1$, r = n) thì trong trường hợp xấu nhất ta phải gọi đệ quy cho tới khi dãy khoá xét chỉ còn là 1 phần tử và vì vậy ta có:

$$w(n) = 1 + w(\lfloor n / 2 \rfloor)$$

Với phương pháp truy hồi, ta có thể viết

$$w(n) = 1 + 1 + w(\lfloor n/2^2 \rfloor)$$
$$= 1 + 1 + 1 + w(\lfloor n/2^3 \rfloor)$$

Như vậy w(n) có thể viết dưới dạng $w(n) = k + w(\lfloor n/2^k \rfloor)$

Khi $\lfloor n/2^k \rfloor = 1$ ta có $w(\lfloor n/2^k \rfloor) = w(1)$ mà $w(1) = 1$ và khi đó tìm kiếm phải kết thúc. Song $\lfloor n/2^k \rfloor = 1$ thì suy ra $2^k \le n \le 2^{k+1}$, do đó $k \le \log_2 n < k+1$, nghĩa là có thể viết $k = \lfloor \log_2 n \rfloor$. Vì vậy cuối cùng ta có:

$$w(n) = \lfloor \log_2 n \rfloor + 1$$

hay

$$T_x(n) = O(\log_2 n)$$

Người ta cũng chứng minh được

$$T_{tb}(n) = O(\log_2 n)$$

Rõ ràng là so với tìm kiếm tuần tự, chi phí tìm kiếm nhị phân ít hơn khá nhiều. Sau này ta sẽ thấy rằng không có một phương pháp tìm kiếm nào dựa trên so sánh giá trị khoá lại có thể đạt được kết quả tốt hơn.

Tuy nhiên ta cũng không nên quên rằng trước khi sử dụng tìm kiếm nhị phân dãy khoá đã phải được sắp xếp rồi, nghĩa là thời gian chi phí cho sắp xếp cũng phải kể đến. Nếu dãy khoá luôn biến động (được bổ sung thêm hoặc loại bớt đi) thì lúc đó chi phí cho sắp xếp lại nổi lên rất rõ và chính điều ấy đã bộc lộ nhược điểm của phương pháp tìm kiếm này.

10.4 Cây nhị phân tìm kiếm (binary search tree)

Để khắc phục nhược điểm vừa nêu trên đối với tìm kiếm nhị phân và đáp ứng yêu cầu tìm kiếm đối với bảng biến động, một phương pháp mới đã được hình thành dựa trên cơ sở bảng được tổ chức dưới dạng cây nhị phân (mỗi bản ghi ứng với một nút trên cây đó) mà ta gọi *cây nhị phân tìm kiếm*.

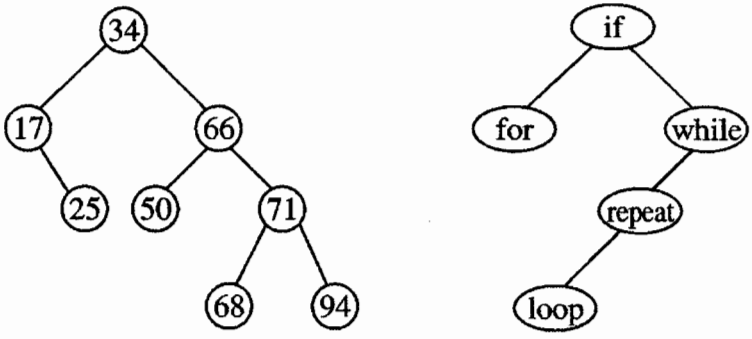
10.4.1 Định nghĩa cây nhị phân tìm kiếm

Cây nhị phân tìm kiếm ứng với n khoá $k_1, ..., k_n$ là một cây nhị phân mà mỗi nút của nó đều được gán một giá trị khoá nào đó trong các giá trị khoá đã cho và đối với mọi nút trên cây tính chất sau đây luôn được thoả mãn:

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

Ở đây thứ tự chọn, ta quy ước là thứ tự tăng dần đối với số và thứ tự tự điển đối với chữ.

Sau đây là ví dụ về cây nhị phân tìm kiếm đối với khoá là số và chữ



a) *b)*

Hình 10.1

10.4.2 Giải thuật tìm kiếm

Đối với một cây nhị phân tìm kiếm để tìm xem một khoá X nào đó có trên cây đó không ta có thể thực hiện như sau:

So sánh X với khoá ở gốc và 1 trong 4 tình huống sau đây sẽ xuất hiện:

1) Không có gốc (cây rỗng): X không có trên cây; phép tìm kiếm không thoả.

2) X trùng với khoá gốc: phép tìm kiếm được thoả

3) X nhỏ hơn khoá ở gốc: tìm kiếm thực hiện tiếp tục bằng cách xét cây con trái của gốc với cách làm tương tự.

4) X lớn hơn khoá ở gốc: tìm kiếm được thực hiện tiếp tục bằng cách xét cây con phải của gốc với cách làm tương tự.

Như với cây ở hình 10.1a, nếu X = 68 ta sẽ thực hiện;

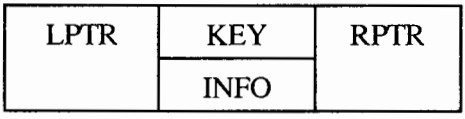
- So sánh X với 34: X > 34, ta chuyển sang cây con phải.
- So sánh X với 66: X > 66, ta chuyển sang cây con phải.
- So sánh X với 71: X < 71, ta chuyển sang cây con trái.
- So sánh X với 68: X = 68, vậy tìm kiếm đã được thoả.

Nhưng nếu X = 30 thì phải

- So sánh X với 34: X < 34, ta chuyển sang cây con trái.
- So sánh X với 17: X > 17, ta chuyển sang cây con phải.
- So sánh X với 25: X < 25, ta chuyển sang cây con phải, nhưng cây con phải rỗng vậy phép tìm kiếm không thoả.

Nếu sau phép tìm kiếm không thoả ta bổ sung luôn X vào cây nhị phân tìm kiếm (như ví dụ vừa xét ta bổ sung khoá 30 vào thành con phải của nút 25) ta thấy phép bổ sung này thực hiện rất đơn giản và không làm ảnh hưởng gì tới vị trí của các khoá hiện có trên cây cả, tính chất của cây nhị phân tìm kiếm vẫn được đảm bảo.

Nếu giả sử quy cách mỗi nút của cây nhị phân tìm kiếm có dạng:



Ở đây trường LPTR và RPTR chứa các con trỏ trỏ tới gốc cây con trái và cây con phải của nút.

Trường KEY ghi nhận giá trị khoá tương ứng của nút, trường INFO ghi nhận các thông tin khác, không có vai trò trong tìm kiếm.

Giải thuật tìm kiếm có bổ sung trên cây tìm kiếm nhị phân sẽ như sau:

Procedure BST (T, X, q);

{Thủ tục này thực hiện tìm kiếm trên cây nhị phân tìm kiếm, có gốc được trỏ bởi T, nút có khoá bằng X. Nếu tìm kiếm được thoả thì đưa ra con trỏ q trỏ tới nút đó, nếu tìm kiếm không thoả thì bổ sung nút mới có khoá là X vào T và đưa ra con trỏ q trỏ tới nút mới đó kèm theo thông báo}.

1. {Khởi tạo con trỏ}

    p := null; q := T;

2. {Tìm kiếm}

while q ≠ null **do**

case

X < KEY (q): p : = q; q := LPTR (q);

X = KEY (q) : **return**

X > KEY(q) : p := q; q := RPTR (q);

end case;

3. {Bổ sung}

call new (q);

KEY(q) := X;

LPTR (q) := RPTR (q) := null;

case

T = null : T :=q; {cây rỗng, đã bổ sung}

X < KEY(p) : LPTR (p) := q;

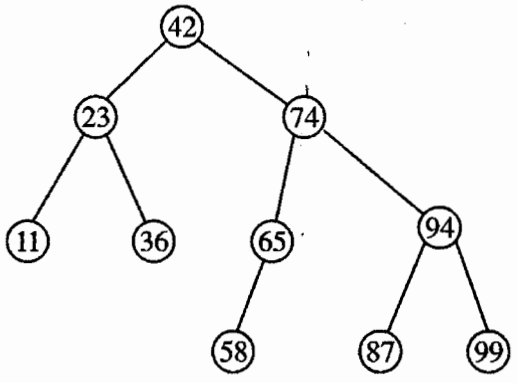
else : RPTR (p) :=q

end case

write ('không thấy, đã bổ sung');

return

Với giải thuật trên có thể suy ra: ta có thể dựng được cây nhị phân tìm kiếm ứng với một dãy khoá đưa vào bằng cách liên tục bổ sung các nút ứng với từng khoá, bắt đầu từ một cây rỗng. Tất nhiên thoạt đầu phải dựng lên nút gốc cây ứng với khoá đầu tiên (trường hợp tìm kiếm trên cây rỗng) sau đó đối với các khoá tiếp theo, tìm trên cây không thấy thì bổ sung vào. Ví dụ, với dãy khoá như đã nêu trong chương 9 trước đây thì cây nhị phân tìm kiếm dựng được sẽ có dạng ở hình 10.2.



Hình 10.2

10.4.3 Phân tích đánh giá

Rõ ràng, với giải thuật BST nêu trên, ta thấy dạng cây nhị phân tìm kiếm dựng được hoàn toàn phụ thuộc vào dãy khoá đưa vào. Như vậy có nghĩa là, trong quá trình xử lý động ta không thể biết trước được cây sẽ

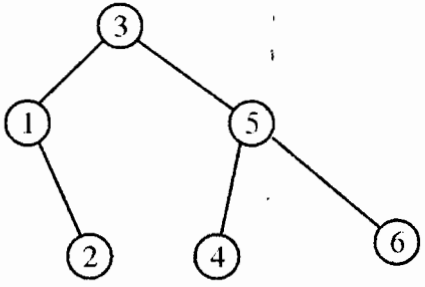
phát triển ra sao, hình dạng của nó sẽ thế nào. Rất có thể đó là một cây nhị phân hoàn chỉnh hoặc cây nhị phân gần đấy (mà ta sẽ gọi là cây cân đối) chiều cao của nó là $\lceil \log_2(n+1) \rceil$, nên chi phí tìm kiếm có cấp độ lớn chỉ là $O(\log_2 n)$: một trường hợp rất thuận lợi mà ta luôn mong đợi. Nhưng cũng có thể đó là một cây nhị phân suy biến (chẳng hạn dãy khoá đưa vào vốn đã có thứ tự sắp xếp rồi!), không khác gì một danh sách tuyến tính mà tìm kiếm trên cây đó chính là tìm kiếm tuần tự với chi phí có cấp O(n). Điều này cũng dễ đưa ta tới một khuynh hướng lo ngại khiến ta thiếu tin tưởng vào phương pháp tìm kiếm này.

Thực ra, người ta cũng đã chứng minh được số lượng trung bình các phép so sánh trong tìm kiếm trên cây nhị phân tìm kiếm chỉ là

$$C_{tb} = 1{,}386 \log_2 n$$

Như vậy cấp độ lớn của thời gian thực hiện trung bình giải thuật BST cũng chỉ là $O(\log_2 n)$ còn nếu xét chi tiết ra thì chi phí tìm kiếm trung bình ở đây chỉ lớn hơn khoảng 39% so với chi phí tìm kiếm trên cây cân đối.

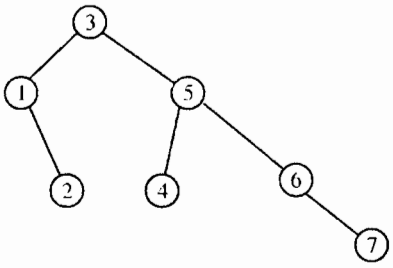
Tới đây cũng có thể xuất hiện thêm câu hỏi là: tại sao không tìm cách dựng lên một cây nhị phân tìm kiếm luôn cân đối để có thể đạt được chi phí tối thiểu? Ta có thể tìm thấy câu trả lời qua việc xét ví dụ đơn giản sau. Giả sử ta có cây nhị phân tìm kiếm cân đối ở hình 10.3.



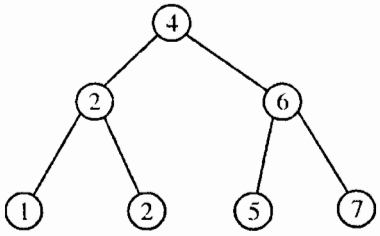
Hình 10.3

Nếu bây giờ xuất hiện khoá X = 7 thì sao?

Ta thấy phép tìm kiếm với X như trên sẽ không được thoả và ta phải bổ sung nút có khoá bằng 7 vào cây trên (Hình 10.4). Ta thấy cây sẽ không còn cân đối nữa. Lẽ tất nhiên khi đó ta phải "tái cân đối" lại để có cây cân đối với 7 nút như hình 10.5.



Hình 10.4



Hình 10.5

Nhìn vào hình, ta thấy hầu như không còn một nút nào mà mối nối được giữ nguyên như cũ. Như vậy, việc tái cân đối đã đòi hỏi phải sửa lại khá nhiều mối nối, nghĩa là sẽ tốn khá nhiều thời gian! Do đó khi phép bổ sung thường xuyên được thực hiện thì cách làm đó sẽ trở nên không thực tế nữa. Chính điều ấy sẽ dẫn tới câu trả lời cho câu hỏi đặt ra.

10.4.4 Loại bỏ trên cây nhị phân tìm kiếm

Khi có một nút ứng với một khoá nào đó (được chỉ định) bị loại ra khỏi cây nhị phân tìm kiếm thì vấn đề gì sẽ xảy ra?

Việc xử lý sẽ không còn đơn giản như khi bổ sung nữa vì để đảm bảo được phần cây còn lại vẫn là một cây nhị phân tìm kiếm ta sẽ phải "sửa" lại cây, nghĩa là tìm "nút thay thế nó" để nhận các con trỏ mà trước đây trỏ tới nó và chữa các con trỏ cần thiết khác.

Nếu nút bị loại bỏ là nút lá, ta không cần tìm nút thay thế nữa. Mối nối cũ trỏ tới nó (từ nút cha nó) sẽ được thay bởi mối nối không.

Nếu nút bị loại bỏ là nút "nửa lá", nghĩa là nó chỉ có cây con trái hoặc cây con phải thì nút thay thế nó chính là nút gốc cây con trái hoặc cây con phải đó. Mối nối cũ trỏ tới nó nay sẽ trỏ tới nút thay thế này.

Trường hợp tổng quát: khi nút bị loại bỏ có cả cây con trái lẫn cây con phải, thì nút thay thế nó hoặc là nút ứng với khoá nhỏ hơn ngay sát trước nó (nút cực phải của cây con trái nó) hoặc là nút ứng với khoá lớn hơn ngay sát sau nó (nút cực trái của cây con phải nó). Như vậy sẽ phải thay đổi một số mối nối, cùng lắm thì cũng chỉ ở các nút:

- Nút cha của nút bị loại bỏ;
- Nút được chọn làm nút "thay thế";
- Nút cha của nút được chọn làm nút thay thế.

Hình 10.6. minh hoạ các trường hợp trên.

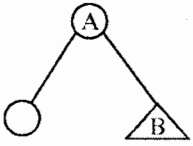
Ở đây, trong trường hợp tổng quát nút thay thế được chọn là nút cực phải của cây con trái.

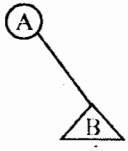
O Chỉ nút bị loại bỏ

Δ chỉ cây con

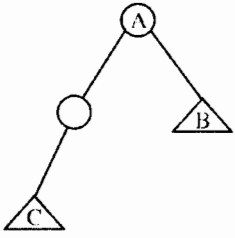
Trước **Sau**

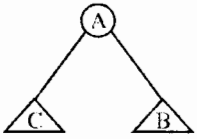
1) Trường hợp nút lá



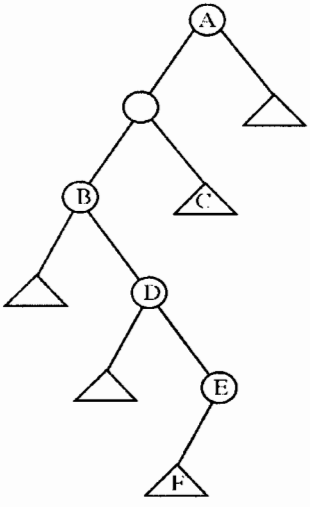


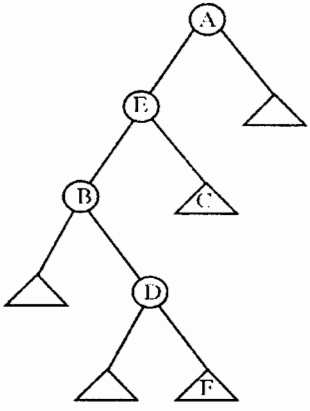
2) Trường hợp nút nửa lá





3) Trường hợp tổng quát





Hình 10.6

Sau đây là giải thuật thực hiện loại bỏ một nút trỏ bởi Q. Thoạt đầu Q chính là nối trái hoặc nối phải của một nút R trên cây nhị phân tìm kiếm, mà ta giả sử đã biết rồi.

Procedure BSTDEL (Q)

{Trong thủ tục này ta phải hiểu Q là đại diện cho LPTR(R) hoặc RPTR(R). Giả sử nó chính là LPTR(R) thì câu lệnh Q := RPTR(P) tương đương với LPTR(R) : = RPTR(P)}

1. {Xử lý trường hợp nút lá và nửa lá}

```
P := Q;
if LPTR(P) = null then begin
        Q := RPTR(P);
        call dispose(P);
                        end;
if RPTR(P) = null then begin
        Q := LPTR(P);
        call dispose(P);
                        end;
```

2. {Xử lý trường hợp tổng quát}

```
T := LPTR(P);
if RPTR(T) = null then begin
        RPTR(T) := RPTR(P);
        call dispose(P);
                        end;
S := RPTR(T); [Tìm nút thay thế là nút cực phải của cây con trái}
while RPTR(S) ≠ null do begin
        T := S;
        S := RPTR(T)
                        end;
RPTR(S) := RPTR(P)
RPTR(T) := LPTR(S);
LPTR(S):= LPTR(P);
Q := S;
call dispose(P);
```

3. **return**

Qua giải thuật trên ta thấy khi loại bỏ một nút ra khỏi cây nhị phân tìm kiếm, việc sửa lại cây đòi hỏi tối đa phải sửa bốn mối trên ba nút. Như vậy

chi phí về sửa đổi này cũng không đáng kể. Nếu cho biết giá trị khoá của nút cần loại bỏ trên cây nhị phân tìm kiếm thì trước hết phải tìm ra nút cần loại rồi mới thực hiện loại bỏ và tương tự như khi tìm kiếm rồi bổ sung, phép tìm kiếm rồi loại bỏ cũng có chi phí trung bình về thời gian ở cấp $O(\log_2 n)$.

10.5 Cây nhị phân cân đối AVL (AVL balanced binary tree)

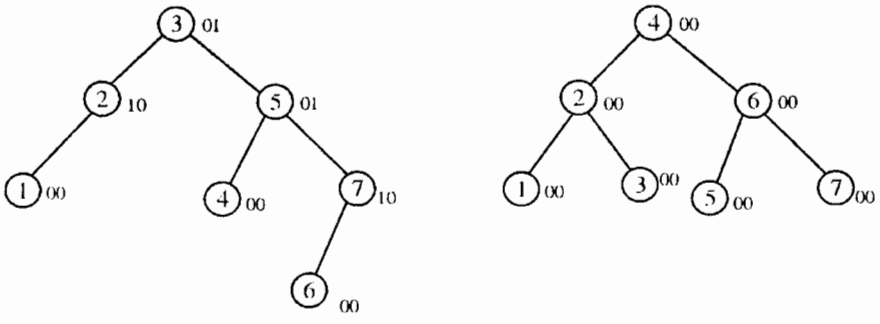
Cây nhị phân tìm kiếm với ưu điểm thực hiện dễ dàng phép bổ sung và loại bỏ đã tỏ ra khá thuận tiện cho việc xử lý các bảng luôn biến động. Tuy nhiên nếu để cây phát triển tự nhiên thì khuynh hướng "suy biến" có thể xuất hiện và điều đó đã làm cho người sử dụng lo ngại. Còn nếu muốn luôn luôn đạt được chi phí tối thiểu thì đòi hỏi cây phải luôn được "cân đối" (như cây nhị phân hoàn chỉnh hoặc cây nhị phân gần đầy). Nhưng như ta đã biết, việc sửa lại cây cho cân đối nếu tiến hành thường xuyên sẽ gây tổn phí khá nhiều thời gian và công sức. Vì vậy cần phải đi tới một giải pháp dung hoà: giảm bớt sự chặt chẽ của tính "cân đối" để tránh được khả năng suy biến của cây.

Năm 1962, P.M. Adelson-Velski E.M. Landis đã mở đầu phương hướng giải quyết này bằng cách đưa ra một dạng cây cân đối mới mà sau này được mang tên họ, đó là cây nhị phân tìm kiếm cân đối AVL (ta gọi là cây AVL cho gọn).

10.5.1 Định nghĩa và tính chất

Cây nhị phân tìm kiếm được gọi là cân đối AVL nếu như đối với mọi nút của nó chiều cao của hai cây con tương ứng chỉ chênh nhau một đơn vị.

Ví dụ: Cây cân đối AVL với 7 nút.



a) *b)*

Hình 10.7

Các chữ số ghi bên cạnh mỗi nút trong hình 10.7 phản ánh tính cân đối của từng nút, ta gọi là *hệ số cân đối* (balance factor)

00 ứng với : Cây con trái và cây con phải có chiều cao bằng nhau (cân bằng)

01 ứng với : Cây con phải có chiều cao lớn hơn 1 (lệch phải)

10 ứng với : Cây con trái có chiều cao lớn hơn 1 (lệch trái)

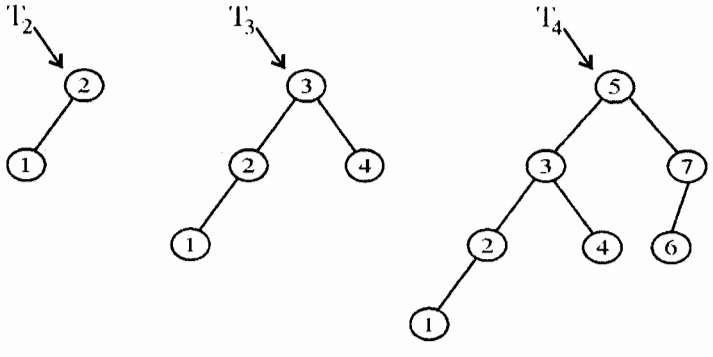
Ta cũng thấy cây nhị phân hoàn chỉnh hoặc gần đầy bao giờ cũng là cây cân đối AVL nhưng cây AVL thì chưa chắc đã là cây nhị phân hoàn chỉnh hoặc gần đầy (như hình 10.7a). Điều đó nói lên: tính cân đối của cây AVL có giảm nhẹ hơn so với tính cân đối của cây nhị phân hoàn chỉnh hoặc gần đầy.

Trong công trình của mình, Adelson - Velski và Landis đã khẳng định được rằng chiều cao của cây cân đối AVL chỉ vượt hơn khoảng 45% so với chiều cao của cây nhị phân hoàn chỉnh. Cụ thể là nếu gọi chiều cao của cây cân đối AVL có n nút là $h_{AVL}(n)$ thì

$$\log_2(n+1) \le h_{AVL}(n) \le 1{,}4404 \log_2(n+2) - 0{,}328 \qquad (1)$$

Rõ ràng cận dưới của nó chính là chiều cao của cây nhị phân hoàn chỉnh hoặc gần đầy có n nút, còn cận trên của nó là chiều cao của loại cây AVL nào?

Chú ý rằng nếu so các cây AVL có chiều cao như nhau, thì loại cây nào có số nút ít nhất chính là loại cây AVL có chiều cao lớn nhất trong số các cây AVL có số nút như nhau. Vì vậy muốn tìm loại cây này ta thử dựng cây AVL có chiều cao cố định là h, còn số nút tối thiểu, gọi là *cây cực tiểu* (mintrees). Giả sử gọi cây đó là T_h. Ta thấy ngay T_0 là một cây rỗng, T_1 chỉ có một nút. Để có cây T_h với h >1 thì ta phải có nút gốc với hai cây con cũng là cây cực tiểu. Do đó một trong hai cây con phải có chiều cao là h-1 và cây kia có chiều cao là h-2. Hình 10.8. cho ta cây cực tiểu với h = 2,3,4.



Hình 10.8

Nếu gắn vào mỗi nút một số tự nhiên và coi chúng như đóng vai trò của khoá trên cây nhị phân tìm kiếm, thì thấy: dãy số kể từ nút cực trái lên tới gốc, chính là dãy số Fibonacci. Như với cây T_4 ở hình 10.8 dãy đó là 1, 2, 3, 5. Vì vậy cây cực tiểu còn có tên là cây Fibonacci. Có thể định nghĩa nó như sau:

a) Cây rỗng là cây Fibonacci với chiều cao bằng 0;

b) Một nút là cây Fibonacci với chiều cao bằng 1;

c) Nếu T_{h-1} và T_{h-2} là cây Fibonacci với chiều cao (h-1) và (h-2) thì $T_h = <T_{h-1}, x, T_{h-2}>$ là cây Fibonacci với chiều cao h.

Chiều cao lớn nhất của cây AVL có n nút, đã nêu ở (1) chính là chiều cao của cây Fibonacci có n nút.

Số nút trên cây Fibonacci T_h được tính theo công thức truy hồi sau đây:

$$N_0 = 0, N_1 = 1$$

$$N_h = N_{h-1} + 1 + N_{h-2}$$

* Theo (1) ở trên thì thấy rằng nếu cây nhị phân tìm kiếm mà luôn luôn có dạng cân đối AVL, thì chi phí cho tìm kiếm đối với nó, ngay trong trường hợp xấu nhất vẫn là $O(\log_2 n)$

Nhưng tới đây, có thể đặt ra một câu hỏi: để đảm bảo được tính cân đối AVL trên cây nhị phân tìm kiếm thì phép bổ sung và loại bỏ có gây ra tốn phí thời gian không? Các phần sau đây sẽ cho ta câu trả lời.

10.5.2 Bổ sung trên cây AVL

Ta hãy xét tới các vấn đề sẽ phải giải quyết khi bổ sung một nút mới vào cây nhị phân tìm kiếm cân đối AVL.

Trước hết, cần chú ý là qui cách một nút của cây bây giờ có khác một chút so với qui cách đã nêu ở 10.4.2. Đó là nó có thêm một trường BIT để ghi nhận hệ số cân đối của nút (chỉ cần 2 bit).

Việc đi theo đường tìm kiếm trên cây để thấy được khoá mới chưa có sẵn trên cây và biết được "chỗ" để bổ sung nó vào, tất nhiên được thực hiện tương tự như đã nêu (thủ tục BST ở 10.4.2). Tuy nhiên có một điều khác là đường đi này phải được ghi nhận lại để phục vụ cho việc xem xét và chỉnh lý hệ số cân đối của nút trên đường đi đó, bị tác động bởi phép bổ sung. Sau khi nút mới được bổ sung, có ba tình huống có thể xảy ra với các nút tiền bối của nó. Để tiện trình bày, ta giả sử phép bổ sung được thực hiện vào phía trái (còn vào phía phải thì cũng tương tự, người đọc sẽ tự suy thêm).

Như vậy ba tình huống đó có thể nêu cụ thể như sau:

a) Tình huống 1: Cây con phải đã cao hơn 1 (lệch phải) sau phép bổ sung chiều cao hai cây con bằng nhau.

b) Tình huống 2: Chiều cao hai cây con vốn đã bằng nhau, sau phép bổ sung cây con trái cao hơn 1 (lệch trái).

c) Tình huống 3: Cây con trái đã cao hơn 1 (lệch trái), sau phép bổ sung nó cao hơn 2: tính "cân đối AVL" bị phá vỡ!

Đối với tình huống 1, chỉ cần chỉnh lý các hệ số cân đối ở nút đang xét.

Đối với tình huống 2, chiều cao của cây có gốc là nút đang xét bị thay đổi, nên không những phải chỉnh lý hệ số cân đối ở nút đang xét mà còn phải chỉnh lý hệ số cân đối ở các nút tiền bối của nó. Dọc trên đường đi đã ghi nhận khi tìm kiếm, cũng phải xem xét để biết có xảy ra tình huống nào trong ba tình huống đã nêu đối với nút đó không và có biện pháp xử lý thích hợp. Như vậy có khi cần phải lần ngược lại tới tận gốc cây.

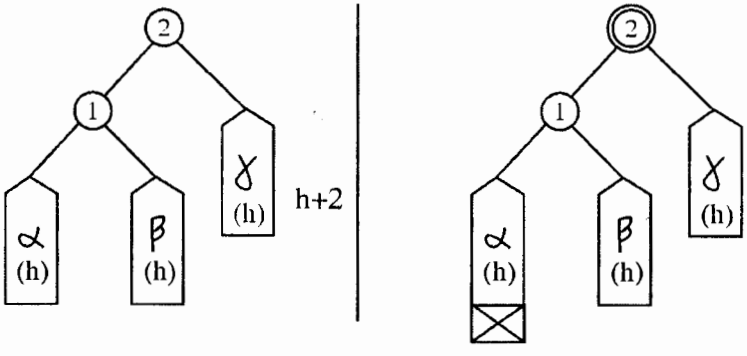
Còn đối với tình huống 3 thì đòi hỏi phải sửa lại cây con mà nút đang xét là nút gốc (ta sẽ gọi là "nút bất thường" critical node), để nó cân đối lại. Có hai trường hợp phải xử lý khác nhau. Hình 10.9 mô tả các trường hợp đó.

a) Trường hợp 1 (LL): Nút mới bổ sung làm tăng chiều cao cây con trái của nút con trái nút bất thường. Như ở hình 10.9a, nút mới bổ sung làm tăng chiều cao của cây α, cây con trái của nút 1.

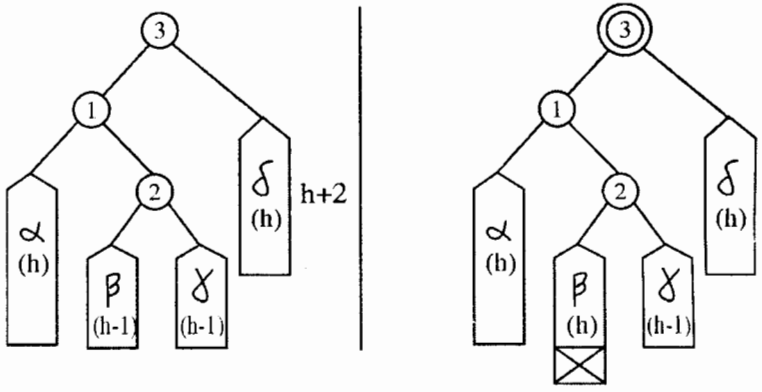
b) Trường hợp 2 (LR): Nút mới bổ sung làm tăng chiều cao cây con phải của nút con trái nút bất thường. Như ở hình 10.9b, nút mới bổ sung làm tăng chiều cao của cây β, cây con phải của nút 1.

Trước lúc bổ sung Sau lúc bổ sung

a)



b)



α(h) chỉ cây con α có chiều cao bằng h

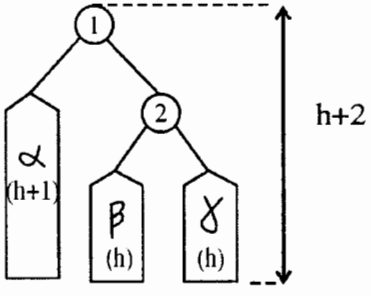
chỉ nút bất thường ứng với khoá bằng 2

chỉ nút mới bổ sung

Hình 10.9

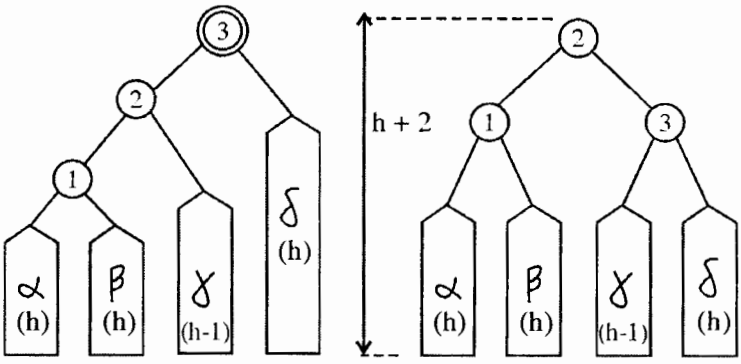
Đối với trường hợp 1 để tái cân đối ta phải thực hiện một phép quay từ trái sang phải để đưa nút (1) lên vị trí gốc cây con, nút (2) sẽ trở thành con phải của nó, và β được gắn vào thành con trái của (2). Ta thấy cách làm này tương tự như cách áp dụng luật kết hợp trong đại số: thay (αβ)γ bằng α(βγ).

Người ta gọi phép này là phép *quay đơn* (single rotation). Xem hình 10.10.



Hình 10.10

Còn đối với trường hợp 2 thì phải thực hiện một *phép quay kép* (double rotation), đó là việc phối hợp hai phép quay đơn: quay trái đối với cây con trái ((1), (2)) và quay phải đối với cây ((3), (2)) như hình 10.11.

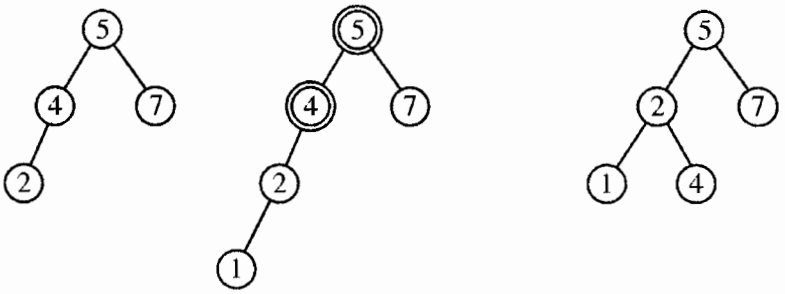


Sau phép quay đơn thứ nhất Sau phép quay đơn thứ hai

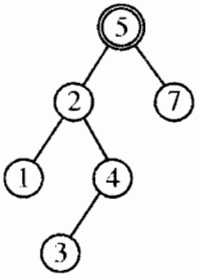
Hình 10.11

Ví dụ sau đây minh hoạ cụ thể tình huống và các phép xử lý tương ứng (hình 10.12).

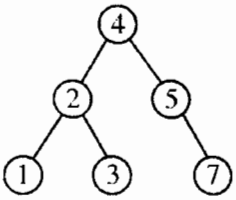
Nếu để ý, ta thấy: sau khi thực hiện phép quay để tái cân đối cây con mà nút gốc là nút bất thường thì chiều cao của cây con đó vẫn giữ nguyên như trước lúc bổ sung, nghĩa là phép quay không làm ảnh hưởng gì tới chiều cao các cây có liên quan tới cây con này. Ngoài ra tính chất của cây nhị phân tìm kiếm cũng luôn luôn được đảm bảo.



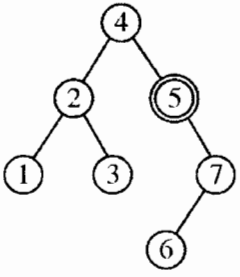
a) Cây ban đầu b) Bổ sung thêm nút (1): mất cân đối (trường hợp 1) c) Tái cân đối bằng phép quay đơn



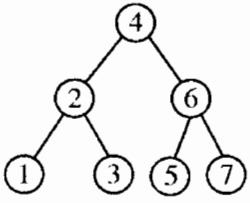
d) Bổ sung thêm nút (3): mất cân đối (trường hợp 2)



e) Tái cân đối bằng phép quay kép



f) Bổ sung thêm nút (6): mất cân đối (trường hợp 2)



g) Tái cân đối bằng phép quay kép

Hình 10.12

Chú thích: Sở dĩ trường hợp f) lại rơi vào tình huống mất cân đối theo trường hợp 2 vì nút (6) bổ sung vào đã làm tăng chiều cao cây con trái của nút con phải nút bất thường (5): RL (suy tương tự điều đã nêu trên, nhưng ứng với phía phải).

Để thực hiện phép quay ta cũng chỉ phải sửa một số ít mối nối (quay đơn cần 2, quay kép cần 4). Điều đó nói lên tính đơn giản của các phép biến đổi này và cũng cho ta thấy việc dựng cây nhị phân tìm kiếm AVL cũng không đến nỗi phức tạp. Tuy nhiên hai câu hỏi sau đây đặc biệt có liên quan tới phép bổ sung:

1. Nếu mọi n! hoán vị của n khoá, xảy ra đồng khả năng thì chiều cao trung bình của cây AVL dựng được sẽ là bao nhiêu?

2. Xác suất để thao tác bổ sung kéo theo việc tái cân đối sẽ là bao nhiêu?

Sự phân tích toán học đối với giải thuật bổ sung trên cây AVL này vẫn còn là một bài toán mở. Nhưng những kiểm định thực nghiệm cho thấy rằng: chiều cao trung bình của cây AVL được tạo lập cũng xấp xỉ bằng $\log_2 n$. Điều đó có nghĩa là dáng điệu của cây nhị phân AVL cũng tốt như

cây nhị phân hoàn chỉnh hoặc gần đầy tuy rằng nó dễ bảo trì hơn nhiều. Thực nghiệm cũng còn cho thấy trung bình cứ hai lần bổ sung thì có một lần phải tái cân đối và phép quay đơn, quay kép đều đồng khả năng.

Chú ý: Phép loại bỏ cũng được thực hiện tương tự với chi phí không khác gì lắm so với phép bổ sung. Phép loại bỏ có thể cũng gây ra mất cân đối và phải tái cân đối bằng các phép quay như đã làm khi bổ sung.

10.6 Cây nhị phân tìm kiếm tối ưu (Optimal binary search tree)

Trước kia ta đã xét tới việc tổ chức cây nhị phân tìm kiếm dựa trên giả thiết là hiện tượng đối trị tìm kiếm (khoá tìm kiếm) trùng với một khoá nào đó đều đồng khả năng, nghĩa là xác suất để $X = k_i \; \forall i \in [1,n]$ đều bằng $\frac{1}{n}$.

Tuy nhiên trong thực tế, có nhiều trường hợp đối trị tìm kiếm xuất hiện với các khả năng khác nhau, nói chung $p_i \neq p_j$ nếu $i \neq j \; \forall i, j \in [1,n]$ và $\sum_{i=1}^{n} p_i = 1$.

Chẳng hạn, qua thống kê các từ khoá được dùng trong hàng nghìn chương trình dịch, người ta thấy xác suất xuất hiện các từ khoá không phải như nhau. Do đó, nếu định tổ chức cây nhị phân tìm kiếm tương ứng với các từ khoá này, ví dụ để phục vụ cho việc xác định xem một từ nào đó trong chương trình có phải là từ khoá không - một khâu trong việc tổ chức chương trình dịch - thì không thể thực hiện như trước được nữa. Rõ ràng là đối với các khoá có xác suất xuất hiện lớn, thì cần phải được tìm thấy nhanh hơn, cho nên quan niệm về chi phí tìm kiếm trên cây bây giờ phải thay đổi cho thích hợp.

Như ta đã biết, để xét chi phí tìm kiếm đặc trưng bởi số lượng các phép so sánh cần thiết khi tìm kiếm trên cây, ta phải chú ý tới độ dài đường đi trên cây. Do đó ở đây ta cũng dựa vào khái niệm này, nhưng có thay đổi chút ít để làm sao có thể phản ánh được trên cây nhị phân tìm kiếm bây giờ, đối với các khoá mà xác suất tìm kiếm lớn hơn phải tương ứng với đường đi ngắn hơn. Muốn thế người ta phải gắn với một khoá (nút) một trọng số, đó chính là xác suất xuất hiện của khoá này trong tìm kiếm. Điều đó có nghĩa là khoá mà xác suất xuất hiện lớn được coi là "nặng hơn", ngược lại sẽ là "nhẹ hơn". Từ đấy dẫn tới khái niệm độ dài đường đi có trọng số của cây (weighted path length): đó là tổng độ dài đường đi có trọng số ứng với mọi nút trên cây, được định nghĩa cụ thể như sau:

$$P = \sum_{i=1}^{n} p_i h_i \qquad (10.1)$$

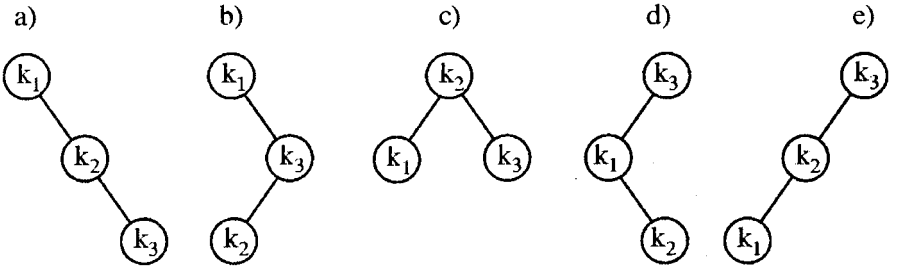
với p_i là xác suất để khoá k_i xuất hiện

h_i là số mức của nút tương ứng với k_i

(h_i = (độ dài đường đi từ gốc tới nút i) + 1, đó cũng chính là số lượng các phép so sánh cần thiết để tìm thấy k_i)

Mà mục đích của chúng ta bây giờ là xác định được cây nhị phân tìm kiếm sao cho P có giá trị nhỏ nhất.

Ví dụ: Cho các khoá $k_1 < k_2 < k_3$ với xác suất tương ứng $p_1 = \frac{1}{7}$, $p_2 = \frac{2}{7}$; $p_3 = \frac{4}{7}$ cây nhị phân tìm kiếm với khoá này có thể dựng như sau:



Hình 10.13

Độ dài đường đi có trong số đối với chúng lần lượt là:

$$P(a) = 1.\frac{1}{7} + 2.\frac{2}{7} + 3.\frac{4}{7} = \frac{17}{7}$$

$$P(b) = 1.\frac{1}{7} + 2.\frac{4}{7} + 3.\frac{2}{7} = \frac{15}{7}$$

$$P(c) = 1.\frac{2}{7} + 2.\frac{1}{7} + 3.\frac{4}{7} = \frac{12}{7}$$

$$P(d) = 1.\frac{4}{7} + 2.\frac{1}{7} + 3.\frac{4}{7} = \frac{12}{7}$$

$$P(e) = 1.\frac{4}{7} + 2.\frac{2}{7} + 3.\frac{1}{7} = \frac{11}{7}$$

Như vậy ta thấy cây (e) là cây suy biến nhưng ở đây nó lại đạt được yêu cầu (chứ không phải cây nhị phân đầy đủ (c)!)

Tuy nhiên cách đặt vấn đề như trên mới chỉ xét tới trường hợp tìm kiếm *thỏa mãn, còn nếu xét toàn diện hơn thì phải đề cập tới cả trường hợp tìm kiếm* không thỏa. Vì vậy ta đặt lại vấn đề như sau:

Có 2n + 1 xác suất:

$p_1, p_2, ..., p_n$ và $q_0, q_1, q_2, ..., q_n$, trong đó:

p_i $(1 \le i \le n)$ là xác suất để đối trị tìm kiếm bằng k_i

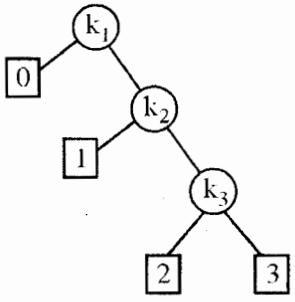
q_j $(0 \le j \le n)$ là xác suất để đối trị tìm kiếm lớn hơn k_j nhưng nhỏ hơn k_{j+1}

(Qui ước: q_0 là xác suất để đối trị tìm kiếm nhỏ hơn k_1, q_n là xác suất để đối trị tìm kiếm lớn hơn k_n).

Như vậy:

$$p_1 + p_2 + ... + p_n + q_0 + q_1 + 1_2 + ...+ q_n = 1$$

Để dễ hình dung ta đưa vào khái niệm "nút ngoài" của cây nhị phân tìm kiếm. Ví dụ: bình thường một nút lá thì có hai con rỗng, nhưng bây giờ ta nói: nút lá có 2 con, mỗi con là một nút ngoài: trên hình vẽ ta thể hiện nút ngoài bằng một hình vuông. Chẳng hạn cây (a) ở hình (10.13) được thể hiện như hình 10.14:



Hình 10.14

Bây giờ, với các nút cũ của cây ta sẽ gọi là "nút trong" như cây ở hình 10.14 có ba nút trong và bốn nút ngoài. Đối với nút trong chỉ số của nút dựa theo thứ tự các khoá như đã qui ước. Còn với nút ngoài thì dựa theo nút trong (như đã nêu với q_j). Như vậy một phép tìm kiếm thoả sẽ dẫn đến một nút trong, một phép tìm kiếm không thoả sẽ dẫn tới một nút ngoài.

Bây giờ yêu cầu đặt ra là: xác định một cây nhị phân tìm kiếm ứng với các khoá $k_1, k_2, ..., k_n$ mà $k_1 < k_2 < ... < k_n$ sao cho

$$P = \sum_{i=1}^{n} p_i h_i + \sum_{j=0}^{n} q_j h'_j \qquad (10.2)$$

(với : h_i là số mức của nút trong thứ i, h'_j là số mức của nút ngoài thứ j trừ đi 1)

có giá trị tối thiểu.

Cây đáp ứng yêu cầu đó được gọi là *cây nhị phân tìm kiếm tối ưu.*

Thực ra khi xác định cây nhị phân tìm kiếm tối ưu không nhất thiết phải đòi hỏi tổng các p_i, q_j phải bằng 1. Trong thực tế người ta thường thay các xác suất này bằng số lần xuất hiện các đối trị tìm kiếm mà người ta dễ dàng xác định được qua thực nghiệm. Ngoài ra cũng để cho tiện người ta thay h'_j bằng h_j là số mức của nút ngoài thứ j. Vì vậy người ta thường tính P dưới dạng:

$$P = \sum_{i-1}^{n} a_i h_i + \sum_{j=0}^{n} b_j h_j \qquad (10.3)$$

(với a_i là số lần xuất hiện đối trị tìm kiếm X bằng k_i

b_j là số lần xuất hiện đối trị tìm kiếm X lớn hơn k_j và nhỏ hơn k_{j+1}, riêng b_0 là số lần $X < k_1$ và b_n là số lần $X > k_n$).

Người ta chứng minh được rằng số lượng các cây nhị phân n nút, có dạng khác nhau, là $\frac{1}{n+1}C_{2n}^{n}$ mà xấp xỉ thì coi như bằng $\frac{4^n}{\sqrt{\pi}n^{3/2}}$ khi n khá lớn. Do đó việc chọn cây nhị phân tìm kiếm tối ưu bằng cách lựa từ trong các cây đó một cây nào độ dài đường đi có trọng số nhỏ nhất, quả là một công việc khó thực hiện khi n khá lớn. Vậy giải quyết bằng cách nào?

Nguyên tắc sau đây sẽ giúp ta vượt qua được trở ngại này, đó là: "đối với một cây nhị phân tìm kiếm tối ưu thì bất kỳ một cây con nào của nó cũng phải là cây nhị phân tìm kiếm tối ưu".

Như vậy nghĩa là nếu $T_{1,n}$ là cây nhị phân tìm kiếm tối ưu ứng với $k_1, k_2, ..., k_n$ có k_i là gốc, thì cây con trái của nó $T_{1,i-1}$ ứng với các khoá $k_1, k_2, ..., k_{i-1}$ và cây con phải của nó $T_{i+1,n}$ ứng với $k_{i+1}, ..., k_n$ phải là cây nhị phân tìm kiếm tối ưu. Đối với các cây con của những cây này cũng có tính chất như vậy. Nhưng cái khó bây giờ lại thể hiện ở chỗ là ta không thể nào xác định được gốc cây nhị phân tìm kiếm tối ưu, nếu không thử mọi khả năng có thể; và ứng với mỗi khả năng ấy phải dựng được cây con trái và cây con phải tối ưu của nó. Tuy nhiên tất cả những điều đó sẽ được giải quyết nếu như ta biết được các cây con tối ưu kể từ cây có 1 nút, 2 nút, đến (n-1) nút ứng với các khoá kế tiếp nhau của cây (theo thứ tự của khoá như đã qui ước). Từ đó dẫn tới một giải thuật cho phép dựng được cây nhị phân tìm kiếm tối ưu lớn dần lên bắt đầu từ một nút lá (từ cây con gồm một nút) mà ta gọi là kỹ thuật dựng cây nhị phân tìm kiếm tối ưu theo kiểu "từ đáy lên" (bottom up)

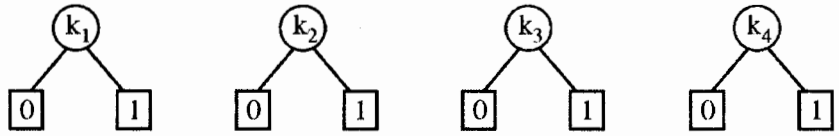
Sau đây ta sẽ minh hoạ cách làm này bằng một ví dụ cụ thể:

Giả sử ta muốn dựng một cây nhị phân tìm kiếm ứng với 4 khoá k_1, k_2, k_3, k_4 ($k_1 < k_2 < k_3 < k_4$) mà

$a_1 = 1, \quad a_2 = 6, \quad a_3 = 4, \quad a_4 = 5$

$b_0 = 3, \quad b_1 = 8, \quad b_2 = 4, \quad b_3 = 2, \quad b_4 = 1$

Trước hết ta xét cây nhị phân tìm kiếm tối ưu gồm một nút, ta có 4 cây (hình 10.15).



Hình 10.15

Bây giờ xét cây nhị phân hai nút ứng với hai khoá kế tiếp, chẳng hạn ứng với k_1, k_2. Ta thấy nó chỉ có thể như ở hình 10.16.

a) b)

hoặc

Hình 10.16

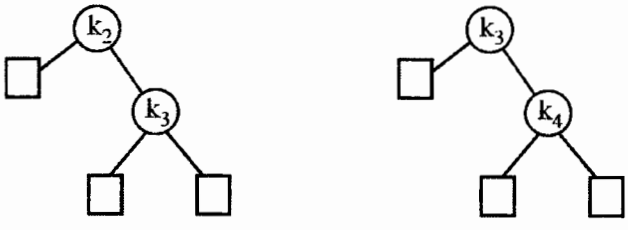
Vì các cây con của chúng đều là cây tối ưu cả nên chúng đều có khả năng là cây tối ưu, nhưng nếu muốn biết cụ thể cây nào thì phải so sánh độ dài đường đi có trọng số của chúng. Theo (10.3) ta có:

$$P_a = 1.1 + 2.6 + 2.3 + 3.8 + 3.4 = 55$$

$$P_b = 1.6 + 2.1 + 3.3 + 3.8 + 2.4 = 49$$

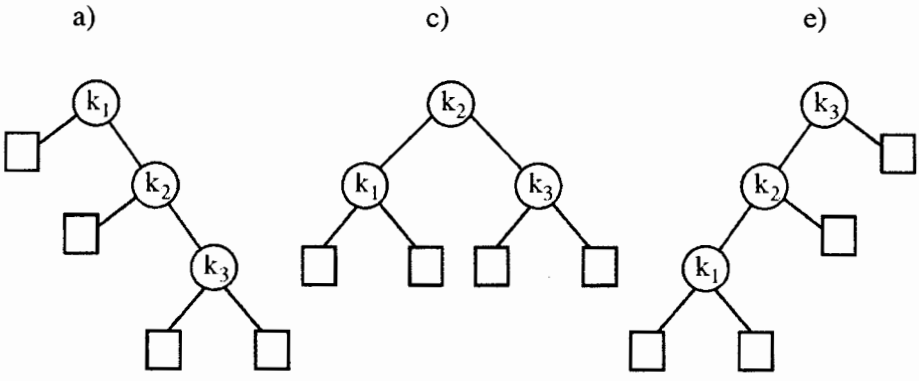
Vậy cây b) ở hình 10.16 là cây nhị phân tìm kiếm tối ưu ứng với hai khoá k_1, k_2.

Tương tự, với các khoá k_2, k_3 và k_3, k_4 cây tối ưu sẽ là:



Hình 10.17

Với cây ba nút ứng với ba khoá kế tiếp k_1, k_2, k_3 đáng nhẽ ta phải xét tới năm trường hợp như ở hình 10.13, nhưng theo nguyên tắc trên thì ta chỉ cần chú ý đến những cây có các cây con (ứng với mọi nút đó) là cây nhị phân tìm kiếm tối ưu (cây tối ưu một nút/ hai nút, như đã nêu). Do đó chỉ còn phải xét ba cây sau:



Hình 10.18

Ta có:

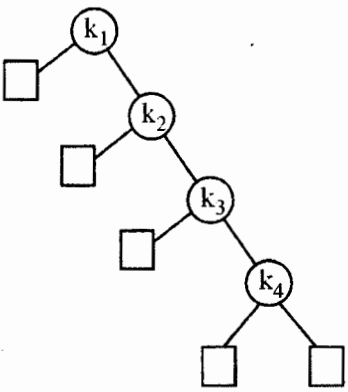
P(a) = 1.1 + 2.6 + 3.4 + 2.3 + 3.8 + 4.4 + 4.2 = 79

P(c) = 1.6 + 2.1 + 2.4 + 3.3 + 3.8 + 3.4 + 3.2 = 67

P(e) = 1.4 + 2.6 + 3.1 + 4.3 + 4.8 + 3.4 + 2.2 = 79

Vậy cây nhị phân tìm kiếm tối ưu là cây c)

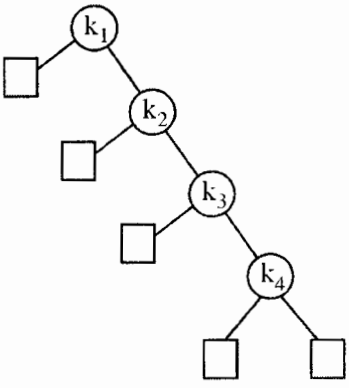
Tương tự, ứng với 3 khoá k_2, k_3, k_4 ta cũng có cây tối ưu là:



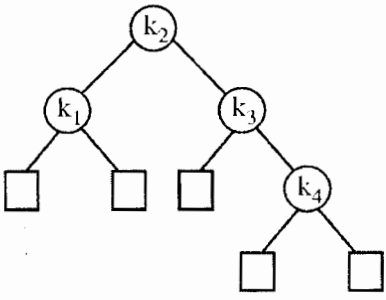
Hình 10.19

Cuối cùng với 4 khoá k_1, k_2, k_3, k_4 thì đáng nhẽ ta phải xét $\frac{1}{n+1}C_8^4 = \frac{8!}{5!} = 14$ trường hợp, nhưng ta cũng chỉ cần xét bốn trường hợp sau:

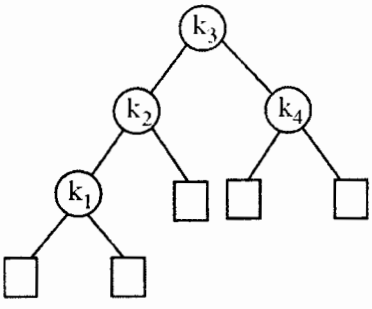
a)



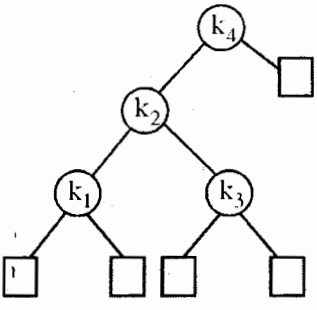
b)



c)

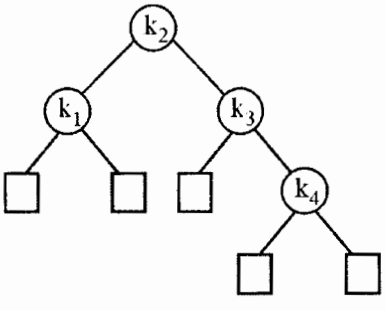


d)



Hình 10.20

Từ đó ta xác định được cây tối ưu là:



Hình 10.21

Người ta chứng minh được rằng với phương pháp dựng cây nhị phân tìm kiếm tối ưu như trên chi phí về không gian nhớ tỉ lệ với n^2: $O(n^2)$, về thời gian thì là $O(n^3)$. D. Knuth trong [8] đã chứng minh được rằng: nếu gọi T_{ij} là cây nhị phân tìm kiếm ứng với các giá trị khoá kế tiếp $k_i < k_{i+1} <...< k_j$ thì gốc của cây này không bao giờ vượt ra ngoài phạm vi các khoá giới hạn bởi các khoá gốc của cây T_{ij-1} và T_{i+1j} nghĩa là nếu gọi r_{ij} là chỉ số của khoá gốc của cây nhị phân tìm kiếm tối ưu T_{ij} thì:

$$r_{ij-1} \le r_{ij} \le r_{i+1j} \qquad (10.4)$$

Từ đó cho phép ta khi xác định khoá ứng với gốc cây nhị phân tìm kiếm tối ưu không phải thử trong số (j - i) khoá nữa, mà chỉ cần thử trong số ($r_{i+1j} - r_{ij-1} + 1$) thôi. Với cách này chi phí về thời gian sẽ giảm xuống chỉ còn là $O(n^2)$.

Như đối với ví dụ trên, bắt đầu từ khi xét cây nhị phân tìm kiếm tối ưu với ba khoá ta thấy ngay số lần thử đã giảm đi rõ rệt.

Chẳng hạn với k_1, k_2, k_3 ta thấy khoá gốc của cây tối ưu bây giờ chỉ cần xét trong các khoá giới hạn bởi khoá gốc của cây tối ưu ứng với k_1, k_2 và k_2, k_3. Nhưng, đối với hai trường hợp này, khoá gốc là k_2, do đó khoá gốc của cây tối ưu tương ứng với k_1, k_2, k_3 phải là k_2 và nếu vậy thì cây tối ưu sẽ có dạng cây c) ở hình 10.18. Đối với trường hợp k_1, k_2, k_3, k_4 cũng có thể suy ra: khoá gốc của cây tối ưu chính là k_2 và ta xác định được cây tối ưu này ở hình 10.21.

Tuy nhiên, nếu so sánh với các phương pháp đã nêu ở các phần trên, ta thấy chi phí tìm kiếm ở đây vẫn còn khá lớn. Vì vậy một số tác giả (như Bruno J; Coffman E.G; Walker W.A; Gotlief CC.v.v...) đã đi tới nghiên cứu các dạng cây nhị phân tìm kiếm "tiệm tối ưu" (nearly optimal binary search tree) và cũng đã đạt được những kết quả đáng chú ý.

10.7 Tìm kiếm dựa vào giá trị khoá

10.7.1 Giới thiệu phương pháp

Đây là một phương pháp tìm kiếm khác hẳn các phương pháp đã nêu trên. Nó không dựa trên cơ sở của phép so sánh giá trị khoá của các bản ghi mà dựa vào chính bản thân giá trị của từng khoá.

Bằng một qui tắc biến đổi nào đó, từ giá trị của khoá ta tính ra một địa chỉ (địa chỉ tương đối). Địa chỉ này sẽ dùng để lưu trữ bản ghi tương ứng, đồng thời cũng để tìm kiếm bản ghi ấy. Như vậy nghĩa là ta thiết lập một *hàm địa chỉ* (address function) h(k) thực hiện phép ánh xạ tập các giá trị

của k lên tập các địa chỉ tương đối, nghĩa là các số nguyên từ 0 đến m - 1 mà ta gọi là *bảng địa chỉ* (address table), m được gọi là độ dài hoặc kích thước của bảng. Như vậy ta luôn có: $0 \leq h(k) < m$. Giá trị của h(k) sẽ được sử dụng khi lưu trữ cũng như khi tìm kiếm bản ghi ứng với k.

Có thể hình dung sơ bộ phương pháp qua một ví dụ đơn giản sau đây:

Xét các bản ghi có khoá tương ứng là các số nguyên gồm không quá 4 chữ số thập phân, chẳng hạn 5402, 0367, 1246, 2983,... Giả sử kích thước của bảng địa chỉ là m= 1000 nghĩa là các địa chỉ tính được phải nằm trong khoảng từ 0 đến 999. Ta chọn qui tắc tính địa chỉ như sau: "Lấy ba chữ số cuối cùng của khoá làm địa chỉ". Như vậy ứng với các khoá nêu trên ta sẽ có kết quả:

Giá trị khoá	Địa chỉ
5402	402
0367	367
1246	246
2983	983

Khi lưu trữ, bản ghi ứng với khoá chẳng hạn 5402 sẽ được đưa vào một ô gồm một số byte trong bộ nhớ có địa chỉ thực là A_0+402. Đến khi tìm kiếm thì địa chỉ A_0 + 402 lại được xác định để sử dụng. Để tiện trình bày, từ đây ta qui ước $A_0 = 0$ nghĩa là tạm coi địa chỉ tương đối như địa chỉ thực.

Rõ ràng với phương pháp này, các khoá có giá trị khác nhau cũng có thể cùng ứng với một địa chỉ; ví dụ 5402, 7402, 0402 đều cùng một địa chỉ 402 cả. Lúc đó ta nói: có *hiện tượng đụng độ* (collision).

Lẽ dĩ nhiên một hàm h(k) được gọi là tốt khi nó phân bố đều các địa chỉ tính được trên bảng địa chỉ cho phép và hiện tượng đụng độ không xảy ra. Về mặt lý thuyết, ta không có cách để xây dựng được một hàm địa chỉ "hoàn hảo" như vậy. Nhưng trên thực tế người ta cũng đã xác định được một số phương pháp có hiệu lực cho phép tạo được các địa chỉ gần ngẫu nhiên (near random) khiến cho hiện tượng đụng độ ít xảy ra hơn. Do hiện tượng đụng độ vẫn có khả năng xuất hiện nên vẫn phải tìm cách khắc phục nó.

Một biện pháp khắc phục đụng độ đơn giản nhanh chóng sẽ ảnh hưởng tới hiệu lực của toàn bộ phương pháp. Cho nên khi xét tới phương pháp này không những cần chú ý đến cách xây dựng hàm địa chỉ h(k) mà còn phải xét cả tới phương pháp khắc phục đụng độ nữa.

Do cách thực hiện phương pháp phần nào có hình ảnh giống như việc rải các hạt một cách ngẫu nhiên vào các ô, cho nên người ta còn gọi nó bởi một cái tên khác là *kỹ thuật lưu trữ rải* (scatter storage technique) và tương ứng với thuật ngữ này các khái niệm có liên quan cũng có tên gọi tương tự như *hàm rải* (scatter function), *địa chỉ rải* (scatter address), *bảng rải* (scatter table).

10.7.2 Hàm rải

Ngoài yêu cầu cơ bản là rải đều các địa chỉ trên bảng địa chỉ cho phép, đối với hàm rải còn phải chú ý thêm một đặc điểm nữa, không kém phần quan trọng, đó là sự tính toán đơn giản của nó. Vì vậy các phương pháp xây dựng hàm rải hiện nay đều dựa trên các phép tính số học quen thuộc.

1) Phương pháp chia

Nguyên tắc của nó rất đơn giản:"lấy số dư của phép chia giá trị khoá cho kích thước m của bảng rải để làm địa chỉ rải" nghĩa là:

$$h(k) = k \bmod m \qquad (10.5)$$

Như trong ví dụ trên, ta lấy ba chữ số cuối cùng của giá trị khoá làm địa chỉ rải tức là lấy phần dư của phép chia giá trị khoá cho 1000, chẳng hạn

$$246 = 1246 \bmod 1000$$

Tất nhiên với phương pháp này, có thể có một số giá trị nào đó của m tạo ra được h(k) tốt hơn giá trị khác của nó. Ví dụ: nếu m là số chẵn thì h(k) sẽ chẵn khi k chẵn và lẻ khi k lẻ, như vậy với giá trị m này h(k) sẽ không được "ngẫu nhiên" lắm. Trường hợp m là luỹ thừa của cơ số của hệ đếm đang dùng, ví dụ như đối với hệ đếm thập phân mà m = 1000 như ở trên, thì cũng không tốt vì lúc này h(k) chính là con số bao gồm các chữ số ở bên phải của khoá và như vậy thì các chữ số ở bên trái của khoá không có ảnh hưởng gì tới h(k) cả, do đó đối với các giá trị khoá mà chỉ khác nhau ở các chữ số nằm bên trái sẽ xảy ra hiện tượng đụng độ!

Thông thường người ta chọn m^* là số nguyên tố nhỏ hơn và gần m thay cho m, nghĩa là lúc này

$$h(k) = k \bmod m^* \qquad (10.6)$$

Như ví dụ trên, nếu $m^* = 997$ ta sẽ có kết quả:

Giá trị khoá	Địa chỉ
5402	417
0367	367
1246	249
2983	989

Bây giờ nếu có thêm các khoá 7402, 0402 thì địa chỉ rải tương ứng của chúng sẽ là 423, 402 nghĩa là không trùng với địa chỉ 417 tương ứng với khoá 5402 ở trên.

Dĩ nhiên h(k) xác định bởi (10.6) thì giá trị của nó chỉ thoả mãn $0 \le h(k) < m^*$ thôi.

Phương pháp này là một trong những phương pháp đơn giản, khá phổ dụng.

2) Phương pháp nhân

Giá trị khoá được nhân với chính nó, sau đó lấy con số bao gồm một số chữ số ở "giữa" kết quả để làm "địa chỉ rải"

Ví dụ:

k	k^2	h(k) gồm 3 chữ số
5402	29181604	181 hoặc 816
0367	00134689	134 hoặc 346
1246	01552516	552 hoặc 525
2893	08898289	898 hoặc 982

Rõ ràng các chữ số "ở giữa" kết quả phụ thuộc vào mọi chữ số có mặt trong khoá vì vậy dù cho các khoá có khác nhau chút ít thì địa chỉ rải tạo ra thường vẫn khác nhau, chẳng hạn:

k	k^2	h(k)
7402	54789604	789 hoặc 896
5401	29170801	170 hoặc 708
5301	28100601	100 hoặc 006

3) Phương pháp phân đoạn (partitioning)

Nếu khoá có kích thước lớn, kích thước thay đổi thì người ta áp dụng phương pháp phân đoạn. Trước hết giá trị khoá phân thành nhiều đoạn bằng nhau (có thể trừ đoạn đầu hoặc đoạn cuối) thường mỗi đoạn có độ dài bằng độ dài địa chỉ. Muốn vậy người ta áp dụng các kỹ thuật như:

a) *Tách* (spliting): tách các đoạn ra, xếp mỗi đoạn một hàng, dóng thẳng theo đầu trái hoặc đầu phải.

b) *Gấp* (folding): gấp các đoạn lại theo đường biên tương tự như gấp giấy. Các chữ số rơi vào cùng một chỗ được đặt thành hàng dóng thẳng với nhau.

Sau khi các đoạn đã được tách hoặc gấp, chúng sẽ được phối hợp với nhau theo một cách nào đấy. Ví dụ chúng được cộng lại. Từ kết quả thu được lấy một đoạn dài bằng địa chỉ để làm địa chỉ rải hoặc lại áp dụng với nó các kỹ thuật tạo địa chỉ như đã nêu. Giả sử có khoá: 17046329.

Bằng phương pháp tách, ta phân ra các đoạn 3 chữ số kể từ đầu phải, rồi cộng lại. Ta có:

329
046
017
392

392 được coi như địa chỉ rải ứng với khoá đó.

Còn bằng phương pháp gấp, ta sẽ có:

046
923
710
1679

Ta có thể lấy 167 hoặc 679 làm địa chỉ rải.

Với phương pháp này ta cũng thấy các chữ số của khoá đều được tham gia vào việc tạo nên địa chỉ rải tương ứng với nó.

Chú thích: Kỹ thuật tạo địa chỉ rải của phương pháp phân đoạn phần nào giống như cách làm món thịt băm: thịt băm nhỏ rồi trộn lẫn với nhau.

Vì vậy, từ năm 1968 trở lại đây, một từ tiếng Anh đã được dùng phổ biến để chỉ phương pháp tìm kiếm này, đó là từ hashing hoặc hash method mà ta gọi là *phương pháp "băm"* (hash nghĩa là băm). Từ đó cũng xuất hiện các thuật ngữ có liên quan đến *hàm băm* (hash function), *địa chỉ băm* (has address) *bảng băm* (hash table).

10.7.3 Khắc phục đụng độ

Có khá nhiều phương pháp cụ thể thực hiện có hiệu lực việc khắc phục hiện tượng đụng độ. Có thể phân chúng làm hai loại:

Loại thứ nhất gọi là phương pháp *địa chỉ mở* (open addressing), nó dựa trên nguyên tắc: đối với bản ghi mà vị trí ứng với địa chỉ rải của nó đã bận (nghĩa là đã bị chiếm: có đụng độ) thì người ta tìm một vị trí khác còn bỏ ngỏ (mở) để thay thế. Đến khi tìm kiếm cũng theo một cách tương tự.

Loại thứ hai dùng tới kỹ thuật móc nối để tạo nên những danh sách móc nối, mỗi danh sách tương ứng với các bản ghi có cùng một địa chỉ rải. Do đó nó được gọi là phương pháp *móc xích* (chaining).

Để tiện cho việc giới thiệu hai phương pháp này, trước hết ta qui ước: bảng được xét bao gồm n bản ghi ứng với n khoá khác nhau và kích thước của bảng địa chỉ rải là m. Các ví dụ minh hoạ sẽ được xây dựng cụ thể trên bảng gồm n= 7 bản ghi mà khoá tương ứng lần lượt là:

221, 643, 512, 326, 495, 108, 069 (10.7)

và m = 8. Địa chỉ rải sẽ được tính theo qui tắc

$$h(k) = k \bmod m^*$$

với m=7, nghĩa là với dãy khoá trên các địa chỉ rải sẽ lần lượt là:

4, 6, 1, 4, 5, 3, 6 (10.8)

1) Phương pháp địa chỉ mở

Phương pháp địa chỉ mở đơn giản nhất là phương pháp *thử tuyến tính* (linear probing). Ở đây hiện tượng đụng độ được khắc phục bằng cách: xem xét vị trí bên cạnh vị trí bận, nếu nó trống thì đưa bản ghi mới vào, nếu nó đã bận thì tiếp tục tìm kiếm tuần tự, nếu đã tới cuối bảng (hoặc đầu bảng) thì phải quay về đầu (hoặc cuối) cho tới khi hoặc tìm được một chỗ trống hoặc đã quay lại vị trí cũ mà không tìm thấy một chỗ trống nào, khi đó ta nói là đã rơi vào tình trạng TRÀN bảng (overflow), nghĩa là không có chỗ cho bản ghi đang xét nữa.

Ví dụ với dãy khoá (10.7) sau khi thực hiện phân bố theo địa chỉ rải ở (10.8) và thực hiện khắc phục đụng độ bằng phương pháp thử tuyến tính thì bảng rải sẽ có dạng như hình 10.22.

V_0	069
V_1	512
V_2	
V_3	108
V_4	221
V_5	326
V_6	643
V_7	496

Hình 10.22

Ta thấy đối với ba khoá đầu việc phân bố thực hiện bình thường theo địa chỉ rải (với mỗi khoá chỉ cần một phép thử để xem vị trí ứng với địa chỉ rải của nó đang trống hay đã bận). Khi khoá 326 được xét thì có đụng độ. V_4 đã bận nên phải chuyển sang V_5 (2 phép thử). Ở đây ta qui ước khi tìm chỗ mới ta dò theo chiều tăng của địa chỉ khi khoá 495 được xét, đụng độ lại xảy ra, V_5, V_6 đều đã bận phải chuyển đến V_7 (3 phép thử). Đối với khoá 108 thì bình thường (1 phép thử) nhưng đến khoá 069 thì lại xuất hiện đụng

độ: V_6, V_7 đều đã bận nên phải quay lên V_0 (3 phép thử). Tất nhiên để có thể phân biệt được chỗ trống hay chỗ đã bận thì ở mỗi vị trí phải dành ra một trường BIT để đánh dấu, chẳng hạn: BIT = 0 thì chỗ còn trống, BIT = 1 thì chỗ đã bận.

Sau đây là giải thuật:

```
Procedure LINEAR_PROBE(V, X,m)
```

{Ở đây V_i (i= 0, ..., m) là nút ứng với địa chỉ rải i, nó gồm có hai trường KEY và BIT để lưu trữ giá trị khoá và dấu tương ứng. Giải thuật này thực hiện tìm kiếm hoặc lưu trữ khi có một khoá mới X được đưa vào}

```
1. {Xác định địa chỉ rải của X}
    i := h(X); j := i;
2. {Xác định chỗ của X}
    while BIT (V[j]) ≠ 0 and KEY (V[j]) ≠ X do
        begin j:= (j +1) mod m;
            if j = i then begin
            write ('không thoả');
            return
            end
        end;
3.  if BIT (V[j]) = 0 then begin
        write('đã lưu trữ');
        BIT (V[j]) ;= 1;
        KEY (V[j]) := X;
        return
                    end;
        else begin
        write ('đã thấy')
        return
            end
```

{Khi tìm kiếm "không thoả" nghĩa là không thấy, khi lưu trữ "không thoả" nghĩa là "TRÀN"}

Một nhược điểm chính của phương pháp thử tuyến tính là hiện tượng *tụ hội* (clustering) các khoá chung quanh các khoá không xảy ra đụng độ, làm cho tốc độ xử lý càng chậm lại khi bảng rải càng gần đầy. Có thể thấy nguyên nhân của hiện tượng này là ở chỗ: đối với hai khoá khi dãy các

phép thử tương ứng với chúng đã dẫn về một vị trí thì từ đó trở đi các phép thử tiếp lặp lại hệt như nhau chứ không tản ra các vị trí khác nhau. Tất nhiên để tránh hiện tượng này thì khi "tìm một địa chỉ mở" ta không thể theo $h_i=(H+i) \bmod m$ với i = 1, 2, ..., m-1 và $h_0 = H = h(X)$, mà phải làm theo $h_i = (H + G(i)) \bmod m$ trong đó G(i) là một hàm ngẫu nhiên nào đó mà giá trị của nó lần lượt là các số nguyên khác nhau trong khoảng [1, m-1] để có khả năng sao cho nếu như: với hai khoá k_1 và k_2 khác nhau mà

$$h(k_1) + G(i) = h(k_2) + G(j) \qquad i \neq j$$

thì thường

$$h(k_1) + G(i+1) \neq h(k_2) + G(j+1)$$

Với G(i) như vậy ta sẽ đi tới một phương pháp tổng quát hơn, gọi là phương pháp *thử ngẫu nhiên* (random probing). Tuy nhiên trong thực tế xử lý, việc tạo ra các giá trị của hàm ngẫu nhiên ngay trong quá trình thực hiện cũng dẫn tới những điều cồng kềnh phức tạp. Do đó, để dung hoà: vừa đạt được kết quả tốt hơn phương pháp thử tuyến tính, vừa đảm bảo thực hiện được đơn giản, người ta thường dùng $G(i) = i^2$. Lúc đó ta gọi là phương pháp *thử cầu phương* (quadractic probing).

Ta thấy: $G(i) = i^2$ thì

$$d_i = G(i+1) - G(i) = 2i+1 \qquad (10.9)$$

$$\Delta d_i = [G(i+2) - G(i+1)] - [G(i+1) - G(i)] = 2$$

Vậy

$$G(i+1) = G(i) + d_i$$

$$d_{i+1} = d_i + 2 \qquad (10.10)$$

Công thức (10.9) cho thấy d_i phụ thuộc vào i nghĩa là ứng với các khoá có địa chỉ rải khác nhau hai phép thử liên tiếp không thể nào xảy ra trùng nhau được do đó có thể tránh được hiện tượng tụ hội. Còn công thức (10.10) cho phép ta dễ dàng xác định được giá trị của G(i) với i = 1, 2,..., m-1 bắt đầu từ $G(0) = 0$, $d_0 = 1$ mà chỉ dùng tới phép cộng.

Tuy nhiên, phương pháp thử cầu phương cũng có một nhược điểm đó là: các phép thử liên tiếp không thể cho phép xét được quá m/2 vị trí khác nhau. Đó là vì $i^2 = (m-i)^2 \bmod m$, nếu m là số nguyên tố thì mới có khả năng đạt tới $\frac{m}{2}$. Nhược điểm trên có khả năng dẫn tới tình trạng khi lưu trữ các vị trí thử đều chật cả, ta sẽ tưởng như đã tràn bảng nhưng thực tình lúc đó vẫn còn chỗ trống. Trong thực hành sự bất tiện này không đáng lo ngại lắm vì rất hiếm khi ta phải thực hiện m/2 phép thử. Ngoài những nhược điểm nêu trên, nói chung phương pháp địa chỉ mở không có khả năng khắc phục được hiện tượng TRÀN bảng và việc loại bỏ khoá cũng gây ra những điều phiền phức đòi hỏi có giải pháp khắc phục khác nữa.

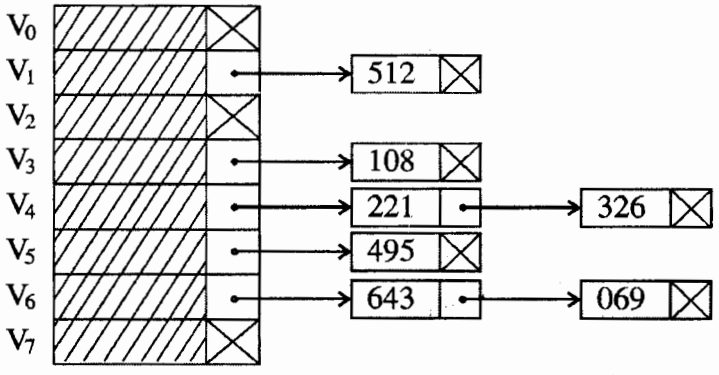
2) Phương pháp móc xích

Một kỹ thuật khá tự nhiên thường được dùng trong phương pháp móc xích là kỹ thuật *móc nối ngoài* (extermal linking). Miền nhớ tương ứng với bảng rải, không dùng để chứa các khoá mà chỉ chứa các nút đầu danh sách của các danh sách móc nối ứng với một địa chỉ rải. Như với dãy khoá (10.7) nêu trên, các danh sách sẽ được tổ chức như ở hình 10.23

Mỗi nút của danh sách móc nối có 2 trường:

Trường KEY chứa giá trị khoá

Trường LINK chứa con trỏ trỏ tới nút tiếp theo



Hình 10.23

Rõ ràng phương pháp này đòi hỏi một không gian nhớ lớn, không phải chỉ do ở chỗ thêm trường LINK ở mỗi nút mà còn ở chỗ phải dùng tới n + m nút ứng với n khoá cho.

Ngoài miền nhớ chính V_0, V_1, ..., V_{m-1}, ứng với bảng rải, n nút của danh sách thường được lấy từ một miền nhớ phụ gọi là *miền tràn* (overflow area).

Ta nêu giải thuật như sau:

Procedure EXTERNAL_LINK (V,X,m)

{Giải thuật này thực hiện tìm kiếm khoá X trong danh sách móc xích, nếu không thấy thì đưa X vào danh sách ứng với địa chỉ h(X) của nó. Các nút trong danh sách có hai trường như đã nêu}

1.{ Xác định vị trí danh sách}

P := V[h(X)];

2. {Tìm vào danh sách}

while LINK(P) ≠ null **and** KEY(P) ≠ X **do**

P:= LINK(P);

3. {Thấy X} **if** KEY(P) = X **then begin write** ('đã thấy');

return;

end

{Ta qui ước V[i] cũng có hai trường KEY và LINK và KEY(V[i]) luôn khác X}.

4. {Lưu trữ X}

Lấy một nút trống Q từ miền tràn;

KEY(Q) := X; LINK(Q) := null;

LINK(P) := Q;

return

Rõ ràng là với phương pháp này do sử dụng danh sách móc nối là loại cấu trúc động nên hiện tượng TRÀN dễ dàng được khắc phục và các tình huống rắc rối do phép loại bỏ cũng sẽ không xuất hiện nữa. Chỉ có một điều ngại là đối với danh sách móc nối thì phải thực hiện tìm kiếm tuần tự. Nhưng trong thực tế thường độ dài các danh sách này không lớn.

Để tiết kiệm không gian nhớ người ta đi tới một cách làm khác dựa trên kỹ thuật *móc nối trong* (internal linking) nghĩa là không dùng thêm một không gian nhớ nào ngoài m nút V_i với i = 1, 2,..., m đã cho (trừ khi để khắc phục hiện tượng TRÀN). Cụ thể là: khi một khoá được đưa vào thì địa chỉ rải của nó được xác định và nút tương ứng được thử. Nếu nút đó trống, khoá sẽ được lưu trữ ở đấy, nó sẽ là phần tử đầu tiên của một danh sách móc nối. Nếu nút đó đã bận thì có hai khả năng:

1) Địa chỉ rải của khoá mới đưa vào trùng với địa chỉ rải của khoá cũ đang ở đấy (ta tạm gọi là khoá hiện diện) thế thì khoá mới này cùng thuộc một danh sách với khoá hiện diện. Phép tìm kiếm trong danh sách tương ứng sẽ được thực hiện. Nếu trong danh sách đó có khoá bằng khoá đưa vào: phép tìm kiếm được thoả. Nếu không có thì phải chọn một nút trống trong miền V làm nút chứa khoá mới và bổ sung nút đó vào danh sách đang xét: Việc lưu trữ khoá mới đã xong.

2) Địa chỉ rải của khoá đưa vào không trùng với địa chỉ rải của khoá hiện diện. Như vậy khoá hiện diện là một phần tử của một danh sách móc nối ứng với một địa chỉ rải khác.

Ta phải tìm một chỗ trống khác để chuyển nó sang và lấy lại chỗ hiện tại cho khoá mới. Công việc này rõ ràng không phải là đơn giản. Ta không những phải sao chép nội dung của các trường ở nút cũ sang nút mới mà còn

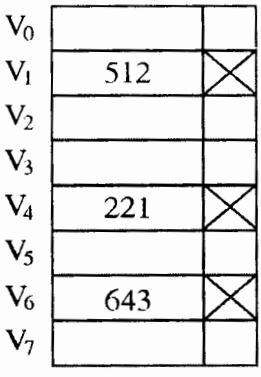
phải sửa mối nối ở nút trước kia đã trỏ tới nút cũ này thành mối nối trỏ tới nút mới chuyển đó. Dĩ nhiên phải tìm được nút đang trỏ tới nút cũ này. Để tìm ngược như vậy, danh sách phải được tổ chức dưới dạng nối kép hoặc nối vòng. Còn nếu nối thẳng thì phải tính lại địa chỉ rải của khoá hiện diện để theo đó lần qua các nút trong danh sách mà tìm đến nút cần thiết.

Hình 10.24 minh hoạ kỹ thuật móc nối trong, ứng với dãy khoá (10.7) nhưng thứ tự đưa vào hơi thay đổi chút ít như sau:

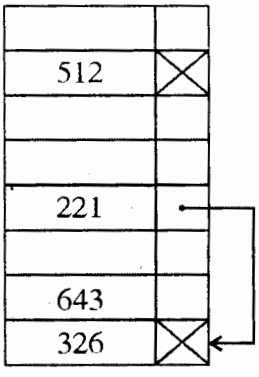
224	643	512	326	069	495	108
4	6	1	4	6	5	3

Ở đây, giả sử khi tìm chỗ trống trong miền V, ta lần ngược từ dưới lên trên.

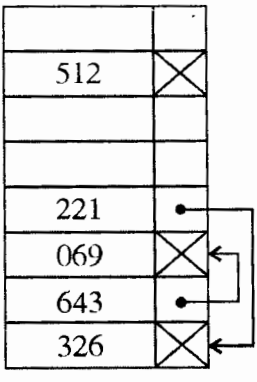
Những khó khăn xuất hiện do việc chuyển khoá cũ để lấy chỗ cho khoá mới cũng cần được xem xét để khắc phục, nhưng ta sẽ không tiếp tục đi sâu thêm nữa.



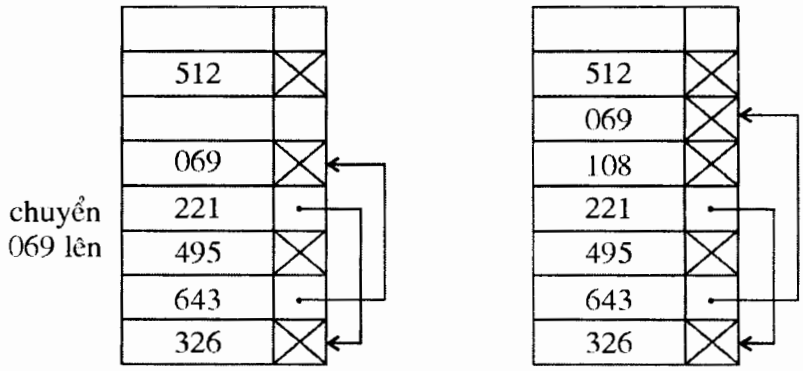
a) Khi 3 khoá đầu được đưa vào



b) Khi khoá 326 được đưa vào



c) Khi khoá 069 được đưa vào



d) Khi khoá 495 được đưa vào e) Khi khoá 108 được đưa vào

Hình 10.24

10.7.4 Phân tích đánh giá phương pháp

Trở lại ví dụ minh hoạ phương pháp thử tuyến tính nêu ở 10.7.3. a) ta thấy: để thực hiện lưu trữ cũng như tìm kiếm các khoá của dãy 10.7 ta đã phải tiến hành tổng cộng 12 phép thử. Có trường hợp khi đưa khoá mới vào, chẳng hạn khoá 495 ta đã phải thực hiện tới 3 phép thử vì liên tiếp xảy ra đụng độ. Có một vấn đề sẽ đặt ra: khi một khoá mới được đưa vào (để lưu trữ hoặc tìm kiếm) số phép thử phải thực hiện có nhiều không? (vì nó ảnh hưởng tới thời gian thực hiện của phương pháp). Nó sẽ phụ thuộc vào điều kiện gì? Thực ra điều ta mong muốn là: tính trung bình, số các phép thử phải nhỏ. Lập luận xác suất sau đây xác nhận rằng quả thật nó cũng khá nhỏ.

Ta hãy xét vào phương pháp thử ngẫu nhiên. Ta giả thiết là các khoá xuất hiện đều đồng khả năng và hàm rải H(k) phân bố chúng đều (ngẫu nhiên) trên bảng rải.

Nếu thêm một khoá vào bảng có kích thước m đã chứa sẵn k đối tượng thì xác suất tìm thấy một vị trí trống ngay lần đầu tiên sẽ là: $1 - \frac{k}{m} = \frac{m-k}{m}$.

Đó là xác suất p_1: chỉ cần một phép thử (phép so sánh). Xác suất để cần đúng một phép thử thứ hai sẽ bằng xác suất của một đụng độ trong lần thứ nhất nhân với xác suất gặp vị trí trống trong lần tiếp theo, nghĩa là:

$$p_2 = \frac{k}{m}.\frac{m-k}{m-1}$$

Tương tự ta sẽ có:

$$p_3 = \frac{k}{m}.\frac{k-1}{m-1}.\frac{m-k}{m-2}$$

...

$$p_{(i)} = \frac{k}{m}.\frac{k-1}{m-1}.\frac{k-2}{m-2}...\frac{k-i+2}{m-i+2}.\frac{m-k}{m-i+1}$$

Vậy thì kỳ vọng của số phép thử cần thiết khi đưa thêm vào khoá thứ $k+1$ sẽ là:

$$E_{k+1} = \sum_{i=1}^{k+1} ip_i = 1.\frac{m-k}{m} + 2.\frac{k}{m}.\frac{m-k}{m-1} + ...$$

$$... + (k+1)\left(\frac{k}{m}.\frac{k-1}{m-1}\frac{k-2}{m-2}...\frac{1}{m-k+1}\right)...$$

$$= \frac{m+1}{m-k+1}$$

Vì số phép thử cần thiết khi đưa một khoá vào lưu trữ sẽ đồng nhất với số phép thử cần thiết khi tìm kiếm nó, nên kết quả trên có thể dùng để tính số trung bình E các phép thử cần thiết để truy nhập vào một khoá ngẫu nhiên trong một bảng rải.

Vẫn ký hiệu m là kích thước bảng và n là số khoá hiện có trong bảng. Ta sẽ có:

$$E = \frac{1}{n}\sum_{k=1}^{n} E_k = \frac{m+1}{n}\sum_{k=1}^{n}\frac{1}{m-k+2}$$

$$= \frac{m+1}{n}(H_{m+1} - H_{m-n+1})$$

$H_m = 1 + \frac{1}{2} + ... + \frac{1}{m}$ là hàm điều hoà.

H_m có thể tính xấp xỉ:

$H_m \approx \ln(m) + \gamma$ với γ là hằng số Euler $\quad (\gamma = 0{,}577216)$

Nếu thay $\alpha = \frac{n}{m+1}$, ta sẽ được:

$$E = \frac{1}{\alpha}(\ln(m+1) - \ln(m-n+1))$$

$$= \frac{1}{\alpha}\ln\frac{m+1}{(m+1-n)} = -\frac{1}{\alpha}(1-\alpha)$$

α như vậy coi như xấp xỉ $\frac{n}{m}$: (thương của số các vị trí đã bị chiếm giữ và vị trí vốn có) α được gọi là *hệ số tải* (load factor). Như vậy kỳ vọng của số phép thử đó phụ thuộc vào hệ số tải này. Bảng liệt kê sau đây sẽ cho thấy giá trị của E tương ứng với giá trị của α (Bảng A)

α	E
0.1	1.05
0.25	1.15
0.5	1.39
0.75	1.85
0.90	2.56
0.95	3.15
0.99	4.66

Bảng A

Ta thấy ngay khi bảng đã đầy tới 90% tính trung bình cũng chỉ cần 2.56 phép thử để hoặc đưa khoá vào lưu trữ ở một vị trí trống hoặc tìm được khoá cho.

Sự phân tích tương tự cũng cho biết kết quả sau (bảng B) đối với phép thử tuyến tính:

$$E = \frac{1 - \frac{\alpha}{2}}{1 - \alpha}$$

α	E
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.90	5.50
0.95	10.50

Bảng B

Còn đối với phương pháp móc nối ngoài thì: E có giá trị ở bảng C

$$E = 1 + \frac{\alpha}{2}$$

α	E
0.1	1.05
0.25	1.12
0.5	1.25
0.75	1.37
0.90	1.45
0.95	1.47

Bảng C

Các kết quả trên chứng tỏ rằng tìm kiếm bằng phương pháp biến đổi khoá là khá tốt. Điều đó cũng khiến ta yên tâm khi sử dụng phương pháp này.

BÀI TẬP CHƯƠNG 10

10.1. Giả sử bảng (tệp) được tổ chức dưới dạng một danh sách nối đơn, mỗi bản ghi ứng với một nút. Quy cách mối nối như sau

KEY	INFO	LINK

Trường KEY: ghi giá trị khoá của bảng ghi

INFO: Chứa các dữ liệu khác

LINK: Chứa con trỏ tới nút tiếp theo.

Gọi L là con trỏ tới nút đầu tiên của danh sách. Hãy lập giải thuật tìm một bản ghi có giá trị khoá bằng X. Có thể áp dụng kinh nghiệm tương tự như giải thuật SEQUEN-SEARCH ở 10.2 vào giải thuật tìm kiếm hay không?

10.2. Tính "được sắp xếp" của một bảng đã tạo ra thuận lợi gì cho tìm kiếm?

10.3. Hãy dùng cây nhị phân để biểu diễn nội dung của giải thuật tìm kiếm nhị phân ứng với n = 9.

10.4. Hãy dựng cây nhị phân tìm kiếm ứng với dãy khoá cho như sau (áp dụng giải thuật BST) HAIPHONG, CANTHO, NHATRANG, DALAT, THAINGUYEN, HANOI, DANANG, HUE, VINH, NAMDINH, SAIGON.

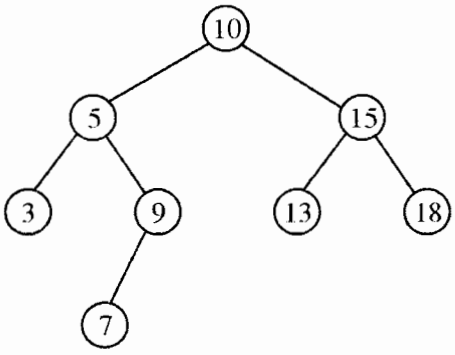
Đánh dấu đường đi trên cây này khi thực hiện tìm kiếm khoá HONGAY

10.5. Có thể dựa vào cây nhị phân tìm kiếm ứng với dãy khoá cho trong bài 4.10, để thực hiện sắp xếp dãy khoá đó theo thứ tự từ điển được không?

10.6. Với các khoá cho như trong bài 10.4, dãy các khoá đọc vào phải thế nào để khi áp dụng giải thuật BST ta có một cây nhị phân hoàn chỉnh (cân đối), một cây nhị phân suy biến?

10.7. Cho cây nhị phân tìm kiếm:

Hãy lần lượt thực hiện phép loại bỏ các nút ứng khoá 13, 15, 5 và 10; Hãy nêu rõ tình huống xảy ra đối với từng trường hợp và cách giải quyết.



Minh hoạ qua hình vẽ.

10.8. a) Dựa vào giải thuật BST, hãy dựng cây nhị phân tìm kiếm ứng với dãy khoá cho sau đây:

23, 5, 14, 3, 8, 10, 21, 1, 7, 4

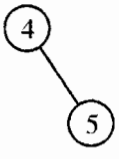
b) Với số lượng 10 khoá như trên thì cây nhị phân tìm kiếm có chiều cao nhỏ nhất là bao nhiêu?

c) Các khoá phải xuất hiện như thế nào để cây nhị phân tìm kiếm dựng được có chiều cao nhỏ nhất?

10.9. Với cây nhị phân tìm kiếm đã dựng ở phần a) thuộc bài tập trên, hãy loại nút mà khoá có giá trị bằng 5 ra khỏi cây đó và vẽ lại cây kết quả.

10.10. Hãy dựng cây Fibonacci với chiều cao h = 5, số nút trên cây này là bao nhiêu?

10.11. Cho cây nhị phân tìm kiếm AVL



Lần lượt bổ sung vào cây đó các khoá 7, 2, 1, 3, 6. Hãy nêu rõ các tình huống xảy ra ứng với từng trường hợp và cách giải quyết. Minh hoạ qua hình vẽ.

10.12. Bổ sung lần lượt các khoá sau đây vào một cây AVL, thoạt đầu còn rỗng: 7, 10, 11, 14, 12, 3, 1, 8, 5, 4, 6, 9, 13.

Nêu rõ các tình huống đòi hỏi phải tái cân đối và cách xử lý. Minh hoạ dạng cây qua hình vẽ.

10.13. Với cây AVL đã dựng xong trong bài 10.12, hãy minh hoạ dạng cây và cách xử lý để cây luôn cân đối AVL, khi lần lượt loại nút ứng với các giá trị khoá bằng 6 và 8.

10.14. Để xác định cây nhị phân tìm kiếm tối ưu ứng với ba khoá k_1, k_2, k_3 ($k_1 < k_2 < k_3$) mà $a_1 = 1$, $a_2 = 3$, $a_3 = 5$, $b_0 = 2$, $b_1 = 4$, $b_2 = 6$, $b_3 = 8$ thì có thể thực hiện theo những cách nào? Hãy thể hiện cụ thể các cách làm đó.

10.15. Cho các khoá $k_1 < k_2 < k_3 < k_4 < k_5$ mà đối với k_1, k_2, k_3 k_4 thì các số liệu cho giống như trong ví dụ của bài giảng (xem 10.6), bây giờ bổ sung thêm $a_5 = 3$, $b_5 = 5$.

Hãy xác định cây nhị phân tìm kiếm tối ưu.

10.16. Giả sử ta xét các khoá là các dòng chữ cái La tinh. Người ta mã hoá chúng theo quy tắc: mỗi chữ cái được mã bởi hai chữ số thập phân ứng với thứ tự của nó trong bộ chữ cái La tinh (từ 01 đến 26). Chẳng hạn khoá TRUNG được mã bởi 20 18 21 14 07.

a) Cho các khoá TIN, HOC, BACH, KHOA. Hãy dựa vào mã của chúng để tính địa chỉ rải theo các phương pháp chia, nhân; cho biết bảng rải có kích thước m = 500.

b) Cho khoá TRUONG DAI HOC BACH KHOA, hãy tính địa chỉ rải theo phương pháp phân đoạn

- Bằng cách tách
- Bằng cách gấp

Giả sử lấy độ dài mỗi đoạn là 8 chữ số và địa chỉ rải gồm 4 chữ số.

10.17. Giả sử địa chỉ rải ứng với khoá k được tính theo công thức: h(k) = k mod 7 và kích thước bảng rải m= 7. Cho dãy các khoá được đưa vào lần lượt là

1, 8, 27, 125, 216, 343

Hãy dùng hình vẽ minh hoạ tình trạng miền rải sau khi mỗi khoá được đưa vào và nói rõ cách giải quyết đụng độ theo phương pháp thử tuyến tính.

10.18. Với bài toán như trên nhưng giải quyết theo phương pháp móc xích (móc nối ngoài và móc nối trong). Minh hoạ qua hình vẽ.

Chương 11

SẮP XẾP VÀ TÌM KIẾM NGOÀI

11.1 Mô hình của xử lý ngoài

Trong các giải thuật bàn luận trước đây, ta đều giả thiết rằng các dữ liệu đưa vào máy không "quá lớn" nghĩa là nó có thể lưu trữ đồng thời được ở bộ nhớ trong, trong quá trình xử lý.

Nhưng, sẽ như thế nào nếu như bây giờ ta muốn quản lý tất cả phiếu nhân sự của mọi công dân trong cuộc điều tra dân số hoặc muốn lưu trữ mọi thông tin về thuế của toàn quốc? Rõ ràng là khối lượng dữ liệu trong các bài toán này đã vượt quá dung lượng bộ nhớ trong của máy tính điện tử, chúng ta phải lưu trữ ở bộ nhớ ngoài trước khi xử lý.

Các hệ máy tính điện tử đều có các thiết bị nhớ ngoài như băng từ, đĩa từ, trống từ. Các thiết bị này có đặc điểm truy nhập hoàn toàn khác so với bộ nhớ trong. Chẳng hạn, với băng từ: dữ liệu sẽ được ghi nhận trên băng dọc theo chiều dài của nó, chúng được đọc - ghi bởi đầu từ. Khi có lệnh đọc ghi, băng từ được kéo chạy qua đầu từ. Như vậy sẽ phải xuất hiện các đoạn băng bị bỏ qua khi khởi động cũng như khi giảm dần tốc độ để dừng, nghĩa là xuất hiện các "khoảng trống" cần thiết trên băng. Vì vậy băng từ phải chia thành từng *khối* (block) và giữa các khối có các *đoạn phân cách khối* (interblock gap). Kích thước của khối được quy định theo từng hệ máy, từ 512 byte đến 4096 byte. Mỗi khối đều có một địa chỉ đó là địa chỉ tuyệt đối của nó trên phương tiện nhớ ngoài. Tệp sẽ được lưu trữ trong một hoặc nhiều khối, mỗi khối bao gồm trọn vẹn một bản ghi (trong khối có thể có các byte thừa không dùng cho bản ghi nào). Bản ghi cũng có địa chỉ, có thể đó là địa chỉ tuyệt đối của byte đầu tiên của bản ghi đó hoặc là địa chỉ của khối chứa nó cộng thêm với số byte trong khối đứng trước byte đầu tiên của bản ghi ấy mà thường được gọi là *độ dời* (offset).

Với đĩa từ hoặc trống từ cách tổ chức cũng tương tự như vậy. Chỉ có khác là: dữ liệu được phân theo từng khối lưu trữ trên các rãnh đồng tâm trên mặt đĩa hay trống, đầu từ sẽ được đặt tới từng rãnh khi đọc và khi ghi dữ liệu. Trong quá trình xử lý việc chuyển giao dữ liệu từ bộ nhớ ngoài vào

bộ nhớ trong và ngược lại được thực hiện theo từng khối thông qua *vùng đệm* (buffer). Đó là một miền nhớ trong, được dành riêng, có kích thước như kích thước của một khối.

Ta cũng luôn luôn hiểu ngầm rằng chính hệ quản lý tệp sẽ đảm nhiệm công việc giúp người sử dụng truy nhập vào các khối của tệp hoặc các bản ghi thông qua tên tệp hoặc tên các trường của bản ghi.

11.2 Đánh giá các phép xử lý ngoài

Đối với thiết bị nhớ ngoài thì thời gian tìm một khối và đọc nó vào bộ nhớ trong hoặc ngược lại, thường khá lớn so với thời gian xử lý dữ liệu. Ví dụ ta có một khối gồm 1000 số nguyên trên một đĩa quay với tốc độ 1000 vòng/phút. Thời gian để đầu từ đặt vào rãnh chứa khối đó (thời gian tìm rãnh - seek time) cộng với thời gian để khối đó được đặt dưới đầu từ (latency time) phải tốn khoảng 100miligiây. Trong khi đó với thời gian này máy có thể thực hiện được 100.000 lệnh, nghĩa là thừa đủ để tính tổng của 1000 số nguyên trên hoặc thậm chí sắp xếp chúng theo giải thuật sắp xếp kiểu phân đoạn (partition sort hay Quick sort) ở bộ nhớ trong.

Vì vậy khi đánh giá thời gian của một giải thuật tác động trên các dữ liệu tổ chức dưới dạng tệp thì điều quan trọng là phải chú ý tới số lần đọc - ghi các khối - ta gọi chung là *phép truy nhập khối* - (Block access).

Kích thước của khối được ấn định bởi hệ điều hành mà ta sử dụng, do đó ta không thể làm giảm thời gian thực hiện giải thuật bằng cách tăng kích thước của khối lên mà phải giảm số lần truy nhập khối bằng cách khác.

Với nhận định như vậy ta sẽ đi vào các giải thuật xử lý ngoài mà đầu tiên là sắp xếp ngoài.

11.3 Sắp xếp ngoài

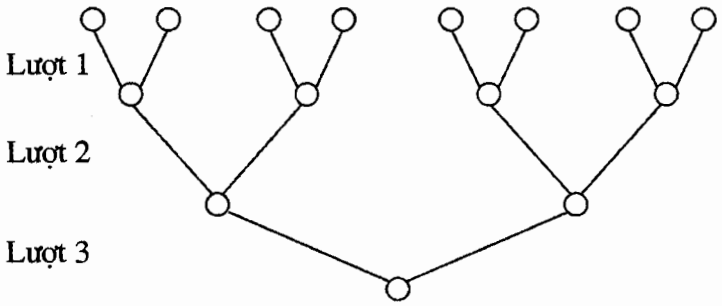
Quá trình sắp xếp ngoài bao gồm hai giai đoạn chủ yếu:

- Giai đoạn sắp xếp trong để tạo nên các mạch ban đầu.
- Giai đoạn hoà nhập các mạch để tạo nên mạch mới, ngày càng lớn hơn, cho tới khi chỉ còn một mạch bao gồm toàn bộ bản ghi của tệp.

11.3.1 Phép hoà nhập K đường (K way merge)

Như ta đã biết: Phép hoà nhập 2 đường cho phép ta hợp nhất hai mạch thành một mạch lớn hơn. Rõ ràng nếu hoà nhập 2 đường với m mạch ban đầu

thì số lượng sẽ là $\lceil \log_2 m \rceil$. Chẳng hạn với m= 8 thì mất 3 lượt (hình 11.1).



Hình 11.1

Số lượng hoà nhập sẽ giảm đi hơn nữa nếu ta dùng phép hoà nhập k đường với k >2. Phép này cũng tương tự như hoà nhập 2 đường, chỉ khác ở chỗ là khoá chọn ra để đưa vào mạch mới không phải chọn lấy từ 2 mạch mà từ k mạch. Với m mạch ban đầu thì hoà nhập k đường sẽ chỉ cần $\lceil \log_k m \rceil$ lượt. Sự giảm bớt số lượt sẽ làm giảm bớt số lần truy nhập khối dẫn tới giảm bớt thời gian chi phí cho sắp xếp. Tuy nhiên, không phải cứ tăng k lên một cách tuỳ tiện. Hoà nhập k đường sẽ kéo theo số các vùng đệm cùng tỷ lệ với k. Tối thiểu phải có k vùng đệm vào (input buffer) và một vùng đệm ra (output buffer). Nhưng do bộ nhớ trong có hạn nên khi số buffer tăng sẽ làm kích thước của buffer bị giảm đi. Điều đó lại kéo theo việc giảm kích thước của khối, như vậy ở mỗi lượt hoà nhập, phép truy nhập khối lại tăng lên!

11.3.2 Sắp xếp trên băng từ

Sắp xếp trên băng từ được nhắc tới đầu tiên do tính chất truyền thống của nó cũng như tính điển hình của nó trong phương pháp sắp xếp ngoài.

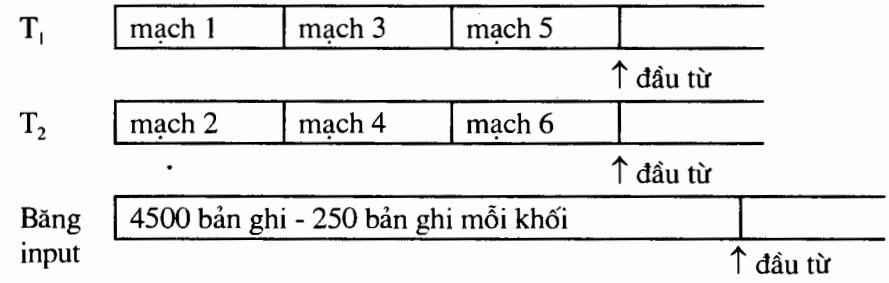
Sắp xếp trên băng từ đặc biệt chịu ảnh hưởng của việc phân bổ các mạch trên băng. Như ta đã biết, băng từ là phương tiện nhớ truy nhập tuần tự. Nếu đầu từ đang ở đầu băng mà khối cần đọc tiếp lại ở cuối băng thì thời gian tìm khối sẽ khá lớn. Do đó yêu cầu đặt ra là: các khối dữ liệu phải được phân bố sao cho chúng được đọc tuần tự trong quá trình hoà nhập k đường.

Sau đây là một ví dụ cụ thể:

Giả sử có 1 tệp gồm 4500 bản ghi R_1, R_2,..., R_{4500} cần được sắp xếp trên một MTĐT mà bộ nhớ trong chỉ chứa được 800 bản ghi.

Bộ nhớ ngoài có 4 băng T_1, T_2, T_3, T_4. Mỗi khối trên băng từ gồm 250 bản ghi. Giả sử tệp input cần phải được giữ lại để còn sử dụng cho yêu cầu khác. Thoạt đầu nó được ghi trên một băng. Sau khi dữ liệu đã được đọc hết, băng từ này sẽ được tháo ra và thay bằng một băng làm việc. Để cho đơn giản ta cũng đặt giả thiết: các mạch ban đầu đã được tạo lập xong bằng một phương pháp sắp xếp trong nào đó, và đã được ghi luân phiên trên hai băng từ T_1 và T_2. Như vậy là 3 khối của băng dữ liệu vào được đọc vào bộ nhớ trong để sắp xếp mới cho ra một mạch (250 x 3 = 750 bản ghi). Ta sẽ có 6 mạch (750 x 6 = 4500). Sau đây ta sẽ nêu từng bước hoạt động của quá trình hoà nhập theo phép hoà nhập 2 đường (kèm theo hình vẽ minh hoạ).

Khởi đầu:

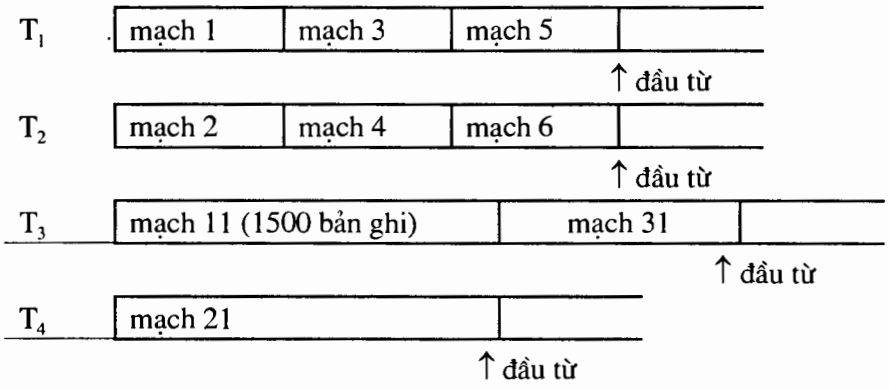


Bước 1

Cuộn các băng T_1, T_2 và băng input (nghĩa là để đầu từ đặt vào đầu các băng đó) tháo băng input ra và thay bằng một băng làm việc, chẳng hạn T_4.

Bước 2

Hoà nhập 2 đường, các mạch ở T_1 và T_2 và phân bố mạch lớn hơn (bây giờ mỗi mạch đã có 1500 bản ghi) luân phiên lên băng T_3, T_4.



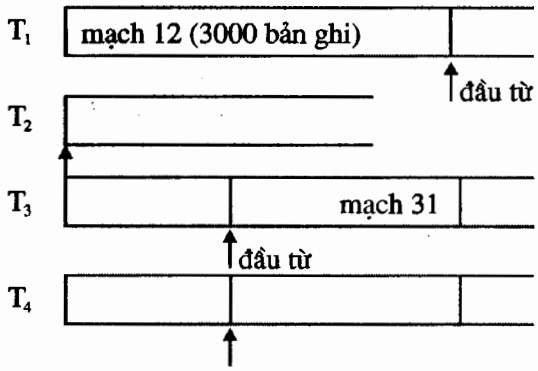
mạch 11 của T_3 được tạo bằng hoà nhập mạch 1 của T_1 với mạch 2 của T_2.
mạch 21 của T_4 được tạo bằng hoà nhập mạch 3 của T_1 với mạch 4 của T_2.
mạch 31 của T_1 được tạo bằng hoà nhập mạch 5 của T_1 với mạch 6 của T_2.

Bước 3

Cuộn các băng T_1, T_2, T_3, T_4

Bước 4

Hoà nhập mạch 11 của T_3 với mạch 21 của T_4 ghi tên T_1



Bước 5

Cuộn các băng T_1 và T_4

Bước 6

Hoà nhập mạch 12 của T_1 với mạch 31 của T_3 và ghi lên T_2. Ta đã có tệp được sắp xếp.

Bước 7

Cuộn lại cả 4 băng. Băng T_2 ghi kết quả.

Ta có thể nêu mấy nhận xét như sau:

1- Với cách tổ chức luân phiên như trên ta đã thường xuyên sử dụng 2 băng từ vào và 2 băng từ ra.

2- Khi thực hiện hoà nhập, dữ liệu trên băng được đọc tuần tự cho đến hết (không phải kéo băng đi, kéo băng lại).

3- Ngoài thời gian thực hiện sắp xếp trong (khi tạo mạch) và thời gian thay băng dữ liệu vào bằng băng làm việc T_4 thì còn phải kể đến:

- Thời gian truy nhập khối
- Thời gian hoà nhập các bản ghi từ "vùng đệm vào" đưa sang "vùng đệm ra"
- Thời gian cuốn lại các băng từ.

Với 4 băng từ thì cách sắp xếp như trên là tương đối hợp lý. Nó phản ánh một trường hợp cụ thể của một phương pháp sắp xếp ngoài được gọi là sắp xếp kiểu hoà nhập nhiều đường cân đối (Balanced multiway merge) mà ta sẽ nêu sau đây.

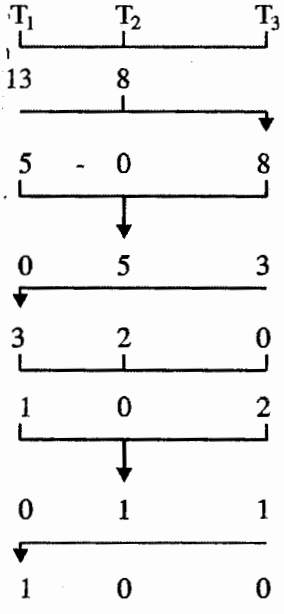
11.3.3 Sắp xếp kiểu hoà nhập nhiều đường cân đối

Đó là phương pháp sắp xếp mà số lượng băng từ để chứa các mạch đưa vào hoà nhập mà ta gọi là các băng từ vào hay băng input - và để chứa các mạch đưa ra sau khi hoà nhập - mà ta gọi là băng từ ra hay băng output - là bằng nhau, nghĩa là nếu có 2k băng từ thì k băng được dùng làm băng input và k băng được dùng làm băng output. Như vậy ở đây rõ ràng sẽ sử dụng phép hoà nhập k đường. Các mạch trên k băng input sẽ lần lượt được hoà nhập lại và được ghi luân phiên lên k băng output. Sau mỗi lượt (khi các mạch trên băng input đã cạn) thì vai trò của băng input và băng output lại đổi cho nhau. Quá trình cứ tiếp tục cho đến khi chỉ còn một mạch duy nhất trên một băng.

11.3.4 Sắp xếp kiểu hoà nhập nhiều pha (polyphase merge sort)

Phương pháp sắp xếp kiểu hoà nhập nhiều đường cân đối có ưu điểm là đơn giản và đồng đều đối với mọi lượt sắp xếp; nhưng nó đã hạn chế việc sử dụng cao hơn nữa số lượng băng từ hiện có để làm băng input. Phương pháp sắp xếp kiểu hoà nhập nhiều pha, do R.L. Gilstar nêu ra năm 1960 đã khắc phục được nhược điểm này. Nó đã nâng mức sử dụng băng từ input lên đến mức tối đa, nghĩa là (m - 1) băng từ trong số m băng từ hiện có. Như vậy là có thể áp dụng được phương pháp hoà nhập (m-1) đường chứ không phải là $\lceil m/2 \rceil$ đường nữa.

Thoạt đầu các mạch trên các băng input sẽ được hoà nhập thành mạch mới và được ghi vào băng output. Quá trình sẽ tiếp tục cho tới khi một băng input cạn, lúc đó băng này lại trở thành băng output còn băng output lúc này lại đóng vai trò băng input, cùng nhập cuộc với các băng input cũ để thực hiện hoà nhập. Như vậy rõ ràng để phép hoà nhập được thực hiện "nhịp nhàng" lúc nào cũng có (m-1) băng input và một băng output thì việc phân bố các mạch trên băng input lúc khởi đầu không thể tuỳ tiện được. Ta sẽ thấy qua vài ví dụ sau đây:



Hình 11.2

Giả sử ta chỉ có 3 băng T_1, T_2, T_3 mà trên T_1 chứa 13 mạch, trên T_2 chứa 8 mạch còn T_3 dùng làm băng output. Ta thấy ở lượt đầu 8 mạch của T_2 hoà nhập với 8 mạch của T_1 theo phép hoà nhập 2 đường (sẽ còn lại 5 mạch) để thành 8 mạch mới, ghi lên T_3. Ở lượt thứ hai 5 mạch của T_1 hoà nhập với 5 mạch của T_3 (sẽ còn 3 mạch) để cho 5 mạch mới ghi tên T_2. Ở lượt thứ ba 3 mạch của T_3 hoà nhập với 3 mạch của T_2 (sẽ còn 2 mạch) để cho 3 mạch mới ghi lên T_1.v.v... Cuối cùng ta sẽ có một mạch lớn ứng với tệp. Có thể minh hoạ qua hình 11.2.

* Bây giờ xét thêm trường hợp 6 băng từ T_1, T_2, T_3, T_4, T_5, T_6 mà các mạch lúc ban đầu được phân bố như sau:

16 mạch trên T_1

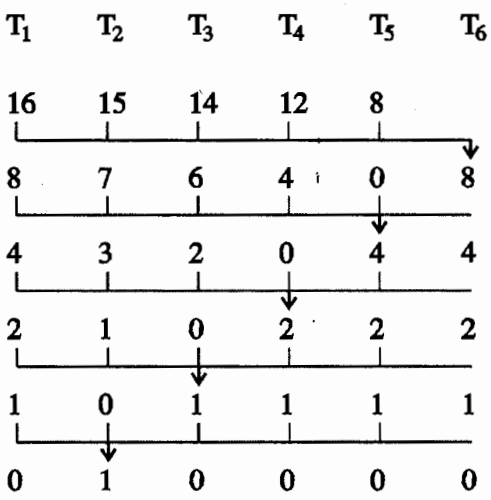
15 mạch trên T_2

14 mạch trên T_3

12 mạch trên T_4

8 mạch trên T_5

Số mạch trên các băng qua quá trình hoà nhập 5 đường, sẽ biến động tương ứng như hình 11.3.



Hình 11.3

Qua các ví dụ trên ta thấy: các mạch ban đầu phân bố trên các băng input tỏ ra rất phù hợp. Nhưng dĩ nhiên đó là cách phân bố đã được lựa chọn trước. Để có thể hiểu được quy tắc của cách phân bố các mạch ban đầu đó, ta hãy xét quá trình theo trình tự ngược lại. Để cho tiện, ta gọi lượt

hoà nhập cuối cùng là lượt hoà nhập ở mức 0, các lượt trước đó là ở mức 1, mức 2, mức 3, ... và gọi ℓ là biến chỉ số mức. Dưới đây là bảng thống kê các mạch ở các mức khác nhau trên các băng input (với M_i^ℓ chỉ: số mạch trên băng input thứ i ở lượt ứng với số mức ℓ.

a) Trường hợp 2 băng input (bảng 11.1)

Bảng 11.1

ℓ	M_1^ℓ	M_2^ℓ	$\sum M_i^\ell$
0	1	0	1
1	1	1	2
2	2	1	3
3	3	2	5
4	5	3	8
5	8	5	13
6	13	8	21

b) Trường hợp 5 băng input

Bảng 11.2

ℓ	M_1^ℓ	M_2^ℓ	M_3^ℓ	M_4^ℓ	M_5^ℓ	$\sum M_i^\ell$
0	1	0	0	0	0	1
1	1	1	1	1	1	5
2	2	2	2	2	1	9
3	4	4	4	3	2	17
4	8	8	7	6	4	33
5	16	15	14	12	8	65

Từ bảng 11.1, ta rút ra mối quan hệ

$$M_2^{(\ell+1)} = M_1^{(\ell)}$$

$$M_1^{(\ell+1)} = M_1^{(\ell)} + M_2^{(\ell)} = M_1^{(\ell)} + M_1^{(\ell-1)}$$

và

$$M_1^{(o)} = 1$$

$$M_2^{(o)} = 0$$

Nếu đặt $M_1^{(\ell)} = F_i$ thì ta có:

$F_{i+1} = F_i + F_{i-1}$ với $i \geq 1$

$F_1 = 1$

$F_0 = 0$

Ta thấy đây chính là quan hệ của các số Fibonacci trong dãy Fibonacci 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

Đối với trường hợp 6 băng từ (5 băng input), cũng thấy mối quan hệ tương tự.

$M_5^{(\ell+1)} = M_1^{(\ell)}$

$M_4^{(\ell+1)} = M_1^{(\ell)} + M_5^{(\ell)} = M_1^{(\ell)} + M_1^{(\ell+1)}$

$M_3^{(\ell+1)} = M_4^{(\ell)} + M_1^{(\ell)} = M_1^{(\ell)} + M_1^{(\ell-1)} + M_1^{(\ell-2)}$

$M_2^{(\ell+1)} = M_1^{(\ell)} + M_3^{(\ell)} = M_1^{(\ell)} + M_1^{(\ell-1)} + M_1^{(\ell-2)} + M_1^{(\ell-3)}$

$M_1^{(\ell+1)} = M_1^{(\ell)} + M_2^{(\ell)} = M_1^{(\ell)} + M_1^{(\ell-1)} + M_1^{(\ell-2)} + M_1^{(\ell-3)} + M_1^{(\ell-4)}$

và

$M_1^{\circ} = 1, M_2^{\circ} = M_3^{\circ} = M_4^{\circ} = M_5^{\circ} = 0$

và cũng suy ra:

$F_{i+1} = F_i + F_{i-1} + F_{i-2} + F_{i-3} + F_{i-4}$ với $i \geq 4$

$F_4 = 1$

$F_i = 0$ với $0 \leq i < 4$

Các số có quan hệ như trên gọi là số Fibonacci cấp 4.

Một cách tổng quát, các số Fibonacci cấp p được định nghĩa:

$F_{i+1}^{(p)} = F_i^{(p)} + F_{i-1}^{(p)} + \ldots + F_{i-p}^{(p)}$ với $i \geq p$

$F_p^{(p)} = 1.$

$F_i^{(p)} = 0$ với $0 \leq i < p$

Như vậy các số Fibonacci mà ta đã biết trước đây chính là các số Fibonacci cấp 1.

Với khái niệm tổng quát này có thể đi tới nhận định: Để phương pháp hoà nhập nhiều pha đối với m băng từ được thuận lợi, thì số các mạch ban đầu phân bố trên (m-1) băng input phải lần lượt là tổng của (m-1), (m-2), ... 2, 1 số Fibonacci cấp m - 2, kế tiếp nhau. Cụ thể là trên băng từ T_i, ứng với mức *l*, số mạch phân bố sẽ là:

$$M_i^{(\ell)} = F_{\ell+p-1}^{(p)} + F_{\ell+p-2}^{(p)} + \ldots + F_{\ell+i-2}^{(p)}$$

Như vậy nghĩa là tổng các mạch ban đầu cần có phải là

$$S_m^{(\ell)} = \sum_{i=1}^{m-\ell} M_i^{(\ell)}$$

Sau đây là bảng giá trị $S_m^{(\ell)}$ ứng với một số giá trị cụ thể của m và l

Bảng 11.3

ℓ /m	3	4	5	6	7	8
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	9	13	17	21	25
4	8	17	25	33	41	49
5	13	31	49	65	81	97
6	21	57	94	129	161	198
7	34	105	181	253	321	385
8	55	193	349	497	636	769
9	89	355	673	977	1261	1531
10	144	653	1297	1921	2501	3049

Tuy nhiên, những điều nói trên rõ ràng là chỉ phù hợp với trường hợp mà tổng số các mạch hiện có đúng bằng $S_m^{(l)}$. Còn với trường hợp mà số các mạch ban đầu không bằng đúng các tổng "lý tưởng" đó thì sao? Ta có thể nghĩ rằng ta sẽ thay các mạch thiếu hụt bằng các mạch rỗng, nghĩa là các "mạch giả". Nhưng nếu như vậy thì làm sao có thể biết được các mạch giả trong quá trình hoà nhập? Vấn đề này rất quan trọng. Nếu như ta biết trên băng T_i có mạch giả thì khi hoà nhập ta vẫn coi như nó có tham dự, nhưng nếu như không biết thì ta có thể sẽ nhầm tưởng là băng từ đó đã cạn và sẽ cho nó chuyển quá sớm thành băng output; điều đó sẽ làm cho hoà nhập nhiều pha thực hiện lệch lạc không đúng như dự kiến nữa!

Việc khắc phục các khó khăn đã nêu sẽ được giải quyết trong giải thuật cụ thể (xem [8]), ta sẽ không đi sâu vào.

11.3.5 Sắp xếp trên đĩa từ

Khác với băng từ, đối với đĩa từ thời gian truy nhập vào các khối khác nhau trên đĩa không chênh lệch nhau bao nhiêu (vì vậy có khi người ta coi đĩa từ là phương tiện nhớ truy nhập "trực tiếp"). Cho nên vấn đề phân bố các mạch không phải đặt ra. Sắp xếp trên đĩa từ chỉ còn phải chú ý tới việc giảm bớt số lượng truy nhập khối. Việc sử dụng phép hoà nhập k đường với k thích hợp là một giải pháp, ngoài ra việc tăng độ dài cho các mạch cũng được chú ý tới.

*** Tăng độ dài cho mạch**

Trước đây khi nói tới các mạch ban đầu ta chỉ giới thiệu một cách đơn giản như một quá trình sắp xếp trong đối với từng mẻ dữ liệu đưa từ bộ nhớ ngoài và độ dài các mạch tạo ra phụ thuộc hoàn toàn vào dung lượng cho phép của bộ nhớ trong (như trong ví dụ đã nêu ở 11.3.2.). Quá trình này có thể coi như độc lập với quá trình phân bố mạch trên phương tiện nhớ ngoài. Tuy nhiên trong điều kiện mà việc tổ chức vào, ra và xử lý trong có thể tiến hành đồng thời (song song) thì thường người ta kết hợp cả hai quá trình sắp xếp và phân bố. Điều đó cho phép áp dụng một kỹ thuật tạo mạch mới dựa trên việc *lựa chọn có thay thế* (selection by replacement).

Giả sử vùng đệm ở bộ nhớ trong cho phép sắp xếp được m khoá (bản ghi) và dữ liệu đang chứa trên đĩa gốc DI. Mạch sẽ được tạo như sau:

Đọc m khoá từ DI vào vùng đệm. Chọn khoá nhỏ K_{min} và ghi lên đĩa DO. Gọi KDI là khoá của DI đang được xét tiếp. Nếu KDI > K_{min} nó sẽ được đưa vào thay thế cho K_{min} và được coi như sẽ thuộc mạch đang tạo. Quá trình chọn khoá nhỏ lại tiếp tục và bước tiếp theo cũng tiến hành tương tự. Nếu KDI < K_{min}, trong một lượt nào đó, nó sẽ vẫn được đưa vào thay thế K_{min} nhưng bị coi như thuộc mạch sau và để lại xét sau, khi các khoá thuộc mạch đang tạo đã được xử lý xong. Chừng nào mà các khoá thay thế bị để lại đã vào đầy vùng đệm cho phép (kích thước m) hoặc dữ liệu vào đã hết thì một mạch mới sẽ bắt đầu được tạo lập.

Ví dụ: Xét dãy khoá sau với m = 7

50 83 59 06 98 85 25 42 17 62 66 08 34 71 21 76

Nội dung của bộ nhớ trong và các khoá được đưa ra mạch mới sẽ diễn biến như trong bảng 11.4

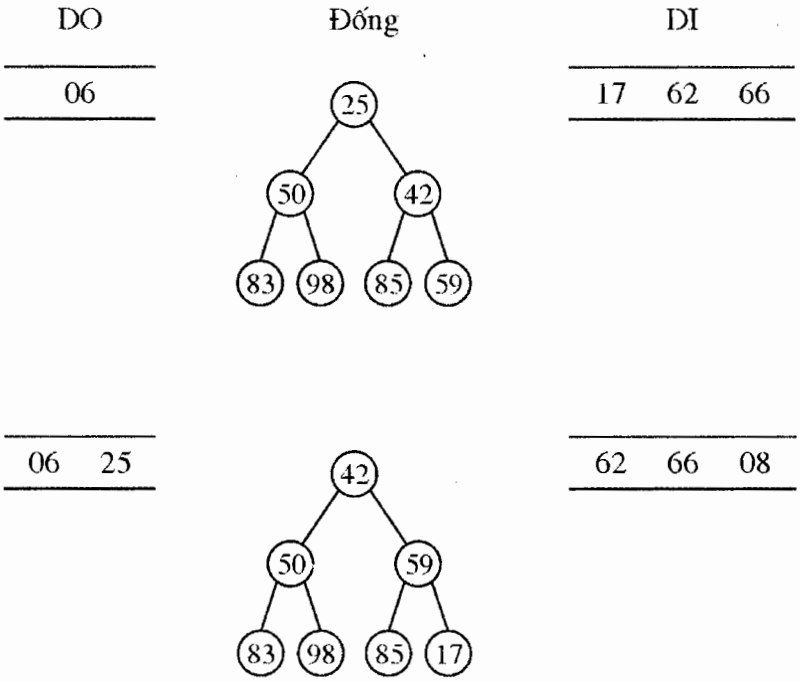
Nội dung bộ nhớ trong							Khoá đưa ra mạch mới
50	83	59	06	98	85	25	06
50	83	59	42	98	85	25	25
50	83	59	42	98	85	(17)	42
50	83	59	62	98	85	(17)	50
66	83	59	62	98	85	(17)	59
66	83	(08)	62	98	85	(17)	62
66	83	(08)	(34)	98	85	(17)	66
71	83	(08)	(34)	98	85	(17)	71
(21)	83	(08)	(34)	98	85	(17)	83
(21)	(76)	(08)	(34)	98	85	(17)	85
(21)	(76)	(08)	(34)	98		(17)	98 hết mạch cũ
21	76	08	34			17	08 bắt đầu mạch mới
21	76		34			17	17
21	76		34				21
	76		34				34
	76						76

Dấu _ chỉ khoá được chọn đưa ra mạch;

Dấu ( ) chỉ khoá bị giữ lại cho mạch tiếp theo.

Với cách này rõ ràng là ta có thể tạo lập được các mạch có độ dài thường không phải là m, mà lớn hơn m. Như ở ví dụ trên, mạch đầu đã có độ dài bằng 11 > 7. Các kết quả thực nghiệm đã cho thấy: trung bình độ dài của các mạch có giá trị là 2m. Như vậy nếu ta có N khoá thì số mạch tạo thành không phải là $S = \lceil N/m \rceil$ mà thường là $s = \lceil N/2m \rceil$ nghĩa là giảm đi khoảng một nửa, điều đó sẽ góp phần làm giảm số lượt hoà nhập, nâng cao hơn tốc độ xử lý của toàn bộ quá trình xử lý ngoài.

Để triển khai kỹ thuật này người ta thường sử dụng sắp xếp trong kiểu vun đống (Heap sort) với khái niệm đống hơi khác với định nghĩa như đã nêu ở 9.4.1 đôi chút: đống ở đây có "khoá cha" nhỏ hơn "khoá con" (chứ không phải lớn hơn) như vậy khoá nhỏ nhất được chọn chính là khoá nằm ở đỉnh đống. Mỗi lần khoá này được chọn đưa ra mạch mới, thì khoá thay thế được đưa vào và đống sẽ được "vun lại". Lúc đó đống sẽ như một "đường hầm" mà các khoá được đưa vào từ DI nếu "chui qua" được, thì khi ra sẽ trở thành khoá của mạch đang được tạo, nếu còn đọng lại trong đó thì sẽ được xét khi tạo mạch tiếp theo. Có thể minh hoạ qua hình 11.4.



Hình 11.4

11.4 Lưu trữ và tìm kiếm ngoài

Trường cũng như bản ghi có thể có kích thước cố định hoặc kích thước biến đổi. Tuy nhiên để cho đơn giản ta cứ coi như chúng có kích thước cố định; còn với kích thước biến đổi, kỹ thuật có thể khác đôi chút, ta sẽ xem xét lúc khác. Đối với tệp, các phép toán cơ bản thường tác động lên chúng vẫn là:

- Tìm kiếm một bản ghi theo một giá trị khoá nào đó.
- Bổ sung một bản ghi vào tệp.
- Loại bỏ khỏi tệp một bản ghi nào đấy.
- Cập nhật một số bản ghi trong tệp (chủ yếu là tìm đến bản ghi đó rồi sửa đổi một số thông tin cần thiết).

Cần chú ý là: có thể có con trỏ trỏ tới bản ghi của tệp, đặc biệt là trong các hệ cơ sở dữ liệu. Như vậy thì các bản ghi, khi đó, phải coi như bị "găm" vào một chỗ cố định vì nếu di chuyển chúng đi chỗ khác thì các con trỏ trỏ tới nó, từ một chỗ nào đó sẽ không còn có tác dụng nữa. Trong trường hợp này, việc loại bỏ một bản ghi thông thường sẽ gây nguy hiểm. Người ta thường dành ra ở một bản ghi một trường 1 bit để đánh dấu vào đó khi loại

bỏ bản ghi ấy. Như vậy bản ghi chỉ bị loại bỏ về mặt lôgic thôi chứ chỗ của nó vẫn để đấy, không dùng lại cho bản ghi khác chừng nào những con trỏ trỏ tới nó chưa bị "treo".

11.4.1 Một cách tổ chức đơn giản: Tệp tuần tự

Một cách đơn giản để tổ chức tệp là lưu trữ tuần tự các bản ghi của nó trong một số khối cần thiết. Các khối có thể được móc nối với nhau bởi con trỏ hoặc một bảng các địa chỉ của chúng được lưu trữ ở một chỗ quy định nào đấy, có thể trên một hoặc nhiều khối.

Tìm một bản ghi có giá trị khoá cho trước được thực hiện bằng cách xem xét giá trị khoá của từng bản ghi trong tệp. Dĩ nhiên chi phí thời gian cho phép này sẽ khá cao nếu như tệp lớn!

Bổ sung một bản ghi được thực hiện bằng cách đưa nó vào khối cuối cùng nếu ở đó còn chỗ, nếu không thì phải xin thêm một khối mới để đưa bản ghi đó vào.

Loại bỏ được tiến hành bằng cách đánh dấu vào "bit loại bỏ". Tuy nhiên nếu chắc chắn rằng các bản ghi của tệp không bao giờ bị "găm" vào một chỗ, thì có thể tổ chức quản lý các không gian trống (ứng với bản ghi đã bị loại) đó để tái sử dụng.

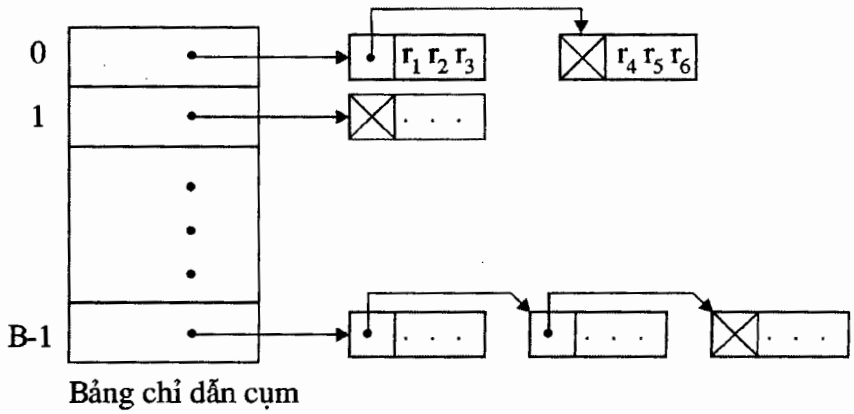
Nhược điểm của cách tổ chức tệp theo kiểu tuần tự này là tốc độ chậm. Trung bình phải đọc tới nửa số khối vào bộ nhớ trong, trong quá trình tìm kiếm.

May thay, còn có cách tổ chức tệp khác cho phép ta truy nhập vào một bản ghi của nó mà chỉ cần đọc một phần nhỏ của tệp thôi. Sau đây ta sẽ xét tới một số cách tổ chức này.

11.4.2 Tệp băm (hashed files)

Cách tổ chức tệp băm ở đây cũng tương tự như cách tổ chức móc nối ngoài mà ta đã xét trước đây (xem 10.7), chỉ có khác một điều là tương ứng với mỗi nút bây giờ là một khối chứ không phải là một bản ghi. Các bản ghi của tệp được phân vào từng cụm (bucket) là ứng với một địa chỉ băm. Nó hoặc rỗng hoặc bao gồm một số khối móc nối với nhau, mỗi khối chứa một số cố định bản ghi. Ở đầu mỗi khối đều có con trỏ trỏ tới khối tiếp theo trong cụm. Có một bảng ghi chỉ dẫn cụm (bucket directory) chứa B con trỏ, mỗi con trỏ ứng với một cụm, đó chính là địa chỉ của khối đầu tiên thuộc cụm đó. B cụm này lần lượt ứng với B địa chỉ băm: 0, 1, 2,..., (B-1). Nếu x là giá trị khoá của một bản ghi nào đó của tệp thì hàm băm h(x) sẽ cho địa chỉ băm của x tương ứng với một trong B địa chỉ nói trên.

Nếu bảng chỉ dẫn cụm có kích thước nhỏ thì nó có thể được lưu trữ ở bộ nhớ trong, nếu không thì phải lưu trữ tuần tự trên một số khối cần thiết ở bộ nhớ ngoài; còn khối của bảng chỉ dẫn cụm (có chứa con trỏ trỏ tới khối đầu tiên của cụm i) sẽ được đọc vào bộ nhớ trong, khi địa chỉ băm i đã được tính.



Hình 11.5

* **Tìm kiếm** một bản ghi có giá trị khoá bằng x:

Tính h(x), ta sẽ được địa chỉ băm của cụm, giả sử là i. Tìm trong bảng chỉ dẫn cụm để biết con trỏ trỏ tới khối đầu tiên của cụm i (nếu có). Tìm trong khối đó xem có thấy bản ghi nào ứng với giá trị khoá x không. Nếu không thấy thì theo con trỏ ở đầu khối này để tới khối tiếp theo và cứ như thế cho tới khi thấy được bản ghi mong muốn, hoặc đã tới khối cuối cùng của cụm mà vẫn không thấy.

* **Bổ sung** một bản ghi có khoá bằng x được thực hiện: tìm tương tự như trên, nếu thấy bản ghi có khoá như vậy thì đã có một sự nhầm lẫn (vì ta giả thiết các giá trị khoá đều khác nhau). Nếu không thấy thì bổ sung bản ghi này vào khối đầu tiên trong cụm mà còn chỗ trống (chỗ trống này có thể được ghi nhận lại trong quá trình tìm kiếm). Nếu không còn chỗ trống nào trong mọi khối của cụm h(x) thì phải xin một khối mới, móc khối đó vào cụm, đặt con trỏ null vào đầu khối đó (vì nó trở thành khối cuối cùng của cụm). Sau đó bổ sung bản ghi vào khối mới này để nó trở thành bản ghi đầu tiên của khối ấy.

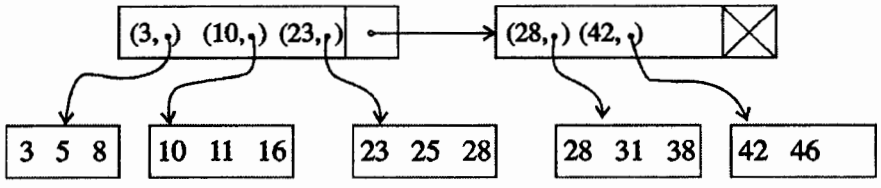
* **Loại bỏ** cũng được thực hiện sau khi đã tìm thấy bản ghi cần thiết, với các cách như đã nêu trong 11.4.1. Tất nhiên trong trường hợp các bản ghi không bị găm vào một chỗ, mà bản ghi bị loại bỏ lại là bản ghi duy nhất của một khối thì khối đó có thể được hệ quản lý tệp giải phóng khỏi cụm để tái sử dụng sau này.

11.4.3 Tệp có chỉ dẫn (indexed files)

Một kỹ thuật khác cũng hay được dùng để tổ chức tệp là tập "chỉ dẫn" cho tệp. Nó tương tự như bảng mục lục cho cuốn sách, giúp ta chỉ việc tra cứu ở đó mà tìm đến được các chương, các mục cần thiết.

Bảng chỉ dẫn thưa (Sparse index) là một thể hiện của kỹ thuật này. Có thể coi nó như một tệp phụ mà mỗi bản ghi là một cặp (K,b) với K là giá trị khoá và b là địa chỉ của khối mà bản ghi đầu tiên có giá trị khoá là K, trong tệp chính. Các cặp này được sắp xếp thứ tự theo giá trị khoá (ta vẫn giả sử giá trị khoá là số và sắp xếp theo thứ tự tăng dần).

Ví dụ:



Hình 11.6

Ở đây ta giả sử: mỗi khối chứa được 3 bản ghi (với tệp bản ghi được đại diện bởi khoá, với tệp (bảng) chỉ dẫn bản ghi được đặc trưng bởi bộ (k,b) trong hình vẽ b được thể hiện bởi mũi tên).

Trong trường hợp các bản ghi không bị găm vào một chỗ cố định, thì phép toán sẽ được thực hiện như sau:

* Để **tìm kiếm** một bản ghi ứng với giá trị khoá x, trước hết tìm trong tệp chỉ dẫn xem có cặp (x, b') không. Thường ta tìm thấy cặp (z, b) mà $z \le x$ và đứng sau nó là cặp (y, c) với $y > x$. Vậy thì x, nếu có, sẽ nằm trong khối b.

Có thể sử dụng nhiều phương pháp tìm kiếm như đã nêu, đối với tệp chỉ dẫn. Đơn giản là tìm kiếm tuần tự, nhưng chỉ nên dùng khi tệp tìm kiếm có kích thước nhỏ. Tốt nhất vẫn là sử dụng tìm kiếm nhị phân, vì tệp chỉ dẫn đã được sắp xếp. Giả sử tệp chỉ dẫn đã được lưu trữ trên các khối $b_1, b_2, ..., b_n$. Để tìm khoá x, ta chọn khối ở giữa: $b_{\lceil n/2 \rceil}$, chẳng hạn, ta so sánh x với giá trị khoá y của cặp đầu tiên trong khối đó. Nếu $x < y$ ta lặp lại phương pháp với các khối $b_1, b_2, ..., b_{\lceil n/2 \rceil}$. Nếu $x>y$ nhưng x nhỏ hơn khoá của cặp đầu tiên của khối $b_{\lceil n/2 \rceil+1}$ thì tìm trong $b_{\lceil n/2 \rceil}$ (tìm tuần tự) nếu không thì lặp lại phương pháp với $b_{\lceil n/2 \rceil} + b_{\lceil n/2 \rceil+2}, ..., b_n$. Với phép tìm kiếm nhị phân này ta sẽ truy nhập tối đa $\lfloor \log_2(n+1) \rfloor$ khối của tệp chỉ dẫn.

Bổ sung một bản ghi mới, ứng với giá trị khoá x, vào tệp, được bắt đầu bằng việc tìm kiếm khối b_i sẽ chứa bản ghi mới đó. Nếu b_i còn chỗ thì đặt x vào khối ấy theo đúng thứ tự sắp xếp, tất nhiên phải dịch chuyển một số bản ghi cũ trong b_i để lấy chỗ cho x, trong trường hợp cần thiết. Nếu sau phép bổ sung, bản ghi mới x trở thành bản ghi đầu tiên của b_i thì phải chỉnh lý lại cặp chỉ dẫn tương ứng ở tệp chỉ dẫn. Nếu b_i không còn chỗ cho bản ghi mới x thì phải áp dụng chiến thuật khác. Đơn giản nhất là xem b_{i+1} , nếu còn chỗ trống để bản ghi cuối của b_i có thể chuyển đến đầu b_{i+1} được thì thực hiện phép chuyển này và bổ sung x vào vị trí tương ứng trong b_i. Dĩ nhiên trong tệp mục lục phải có phép chỉnh tương ứng đối với b_{i+1} cũng như b_i (khi x là bản ghi đầu tiên của b_i). Nếu b_{i+1} cũng đầy hoặc b_i đã là khối cuối (i = m) thì một khối mới sẽ được tạo ra, bản ghi mới được bổ sung vào khối đó và khối này được đặt sau b_i theo đúng thứ tự như quy định. Tất nhiên ở tệp chỉ dẫn cũng phải có sự bổ sung tương ứng.

Phép **loại bỏ** cũng sẽ dẫn tới tình huống phải xử lý tương tự.

Cũng cần chú ý thêm đôi nét về việc khởi tạo nên tệp có chỉ dẫn. Trước hết phải sắp xếp các bản ghi theo giá trị khoá và phân bố chúng lần lượt vào các khối theo thứ tự ấn định.

Ta có thể "nhét" đầy bản ghi vào từng khối nhưng cũng có thể để thừa ra một số chỗ trống để tạo thuận lợi cho phép bổ sung sau này,

Sau đó, ta tạo nên tệp chỉ dẫn bằng cách truy nhập vào từng khối và tìm khoá đầu tiên trong khối đó, từ đó thiết lập nên các bản ghi (cặp (k,b)) cho tệp chỉ dẫn. Cũng tương tự như đối với tệp chính, ở đây khi ghi bản ghi vào các khối, có thể để dành chỗ dự trữ cho việc bổ sung nghĩa là không cần "nhét" đầy khối.

Một cách lập chỉ dẫn khác gọi là "chỉ dẫn đầy" (dense index) lại cho phép truy nhập tới từng bản ghi. Như vậy trong tệp chỉ dẫn đầy các bản ghi là các cặp (k, p) với p là con trỏ trỏ tới bản ghi tương ứng với giá trị khoá k trong tệp chính. Các cặp này cũng được sắp xếp theo giá trị khoá tương tự như tệp chỉ dẫn thưa. Tuy nhiên, bản ghi của tệp chính thì lại không được sắp xếp, nó được ghi nhận ngẫu nhiên. Như vậy bổ sung một bản ghi mới vào tệp sẽ được thực hiện vào khối cuối cùng, nếu khối đó còn chỗ. Nếu nó đầy rồi thì một khối mới sẽ được tạo lập để bổ sung nó vào. Tất nhiên kèm theo đó phải bổ sung thêm chỉ dẫn ứng với bản ghi này vào tệp chỉ dẫn. Loại bỏ một bản ghi ở tệp chính cũng kéo theo loại bỏ chỉ dẫn tương ứng với bản ghi ấy trong tệp chỉ dẫn đầy.

11.4.4 Cây tìm kiếm ngoài

Cấu trúc cây đã giới thiệu có thể được sử dụng để biểu diễn tệp ở bộ nhớ ngoài. Tuy nhiên B-cây (B-trees) là một cấu trúc đặc biệt có nhiều

thuận lợi hơn. Để hiểu B-cây, trước hết ta hãy xét qua về cây *tìm kiếm nhiều đường* (multiway search trees).

1) Cây tìm kiếm nhiều đường

Cây tìm kiếm cấp m là sự mở rộng của cây nhị phân tìm kiếm, trong đó mỗi nút có tối đa m con.

Sự mở rộng đặc tính này của cây nhị phân tìm kiếm vào cây này thể hiện ở chỗ:

Nếu n_1 và n_2 là hai con của một nút nào đó mà n_1 ở bên trái của n_2 , thì khoá ứng với n_1 và các con của nó đều nhỏ hơn khoá ứng với n_2.

Các giải thuật tìm kiếm, bổ sung, loại bỏ ở đây cũng được mở rộng tương tự như cây nhị phân tìm kiếm.

Tuy nhiên, có một điều cần chú ý là các bản ghi của tệp ở đây được lưu trữ theo từng khối của thiết bị nhớ ngoài. Vì vậy trên cây tìm kiếm nhiều đường, tương ứng với một nút là một khối. Mỗi nút trong của một cây sẽ chứa con trỏ trỏ tới m con của nó, kèm theo (m-1) khoá để phân biệt các con này. Các nút lá thì tương ứng với các khối chứa các bản ghi của tệp chính. Như vậy các nút trong (hay nút nhánh) của cây chính là các khối của tệp chỉ dẫn (indexe files). Trong các tệp chỉ dẫn thứa hoặc tệp chỉ dẫn đầy nêu ở trên, các khối của tệp đó được tổ chức theo kiểu móc nối kế tiếp nhau, còn ở đây các khối của tệp chỉ dẫn lại được tổ chức theo kiểu phân cấp dưới dạng cây.

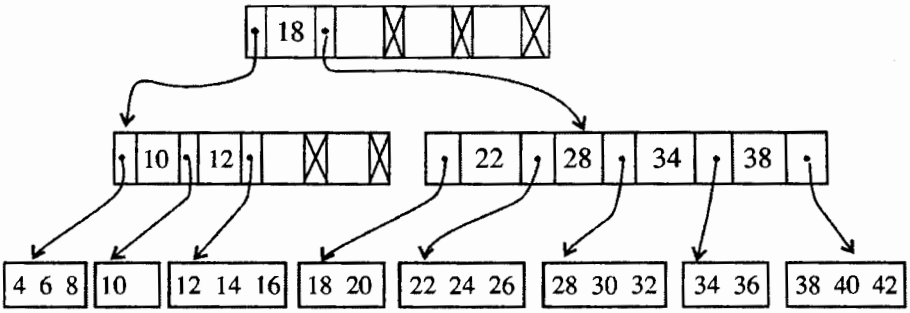
Ta đã biết nếu một cây nhị phân có n nút biểu diễn một tệp thì cần trung trình $\log_2 n$ lượt truy nhập khối để tìm một bản ghi của tệp; còn đối với cây tìm kiếm m đường thì trung bình chỉ cần $\log_m n$ lượt truy cập khối thôi.

2) B-cây

B-cây là một loại cây tìm kiếm cấp m, cân đối, với các tính chất như sau:

1. Gốc cây hoặc một nút lá ít nhất cũng có 2 con.
2. Mỗi nút, trừ nút gốc có từ $\lceil m/2 \rceil$ đến m con.
3. Mỗi đường đi từ gốc tới lá có độ dài như nhau.

Hình sau đây cho ta một B-cây cấp 5 với giả thiết: một khối ứng với lá chứa tối đa 3 bản ghi. Khối ứng với nút trong chứa được 5 con trỏ và 4 khoá.



Hình 11.7

Ta thấy: các nút trong của B-cây chính là các khối của tệp chỉ dẫn. Các chỉ dẫn được phân theo mức ngày càng chi tiết hơn. Mỗi nút trong trên B-cây có dạng: $(p_o, k_1, p_1, k_2, p_2, ..., k_n, p_n)$ với p_i là con trỏ trỏ tới con thứ i của nút mà k_i là khoá đầu ($1 \le i \le n$). Các khoá trong một nút được sắp xếp theo thứ tự tăng dần. Mọi khoá trong cây con, trỏ bởi p_o thì đều nhỏ hơn k_1.

Với $1 \le i < n$ mọi khoá trong cây con được trỏ bởi p_i thì có giá trị lớn hơn hoặc bằng k_i và nhỏ hơn k_{i+1}. Mọi khoá trong cây con được trỏ bởi p_n đều lớn hơn k_n. Các nút lá là ứng với các khoá (bản ghi) của tệp chính. Bây giờ ta xét đến các phép toán trên B-cây.

1) Tìm kiếm

Để tìm một bản ghi r với khoá x, ta phải "vạch ra" đường đi từ gốc đến nút lá có chứa r, nếu nó có trong tệp. Muốn vậy ta sẽ phải liên tiếp trút nội dung dạng $(p_o, k_1, p_1, ..., k_n, p_n)$ của các nút trong trên B-cây, kể từ nút gốc, vào bộ nhớ trong và xác định con trỏ trỏ tới nút tiếp theo trên đường đi bằng cách so sánh x với các khoá $k_1, k_2..., k_n$ trên nút đang xét. Nếu $k_i \le x < k_{i+1}$ ta sẽ theo con trỏ p_i, trút nội dung của nút trỏ bởi nó vào bộ nhớ trong và lặp lại quá trình so sánh như trên. Nếu $x < k_1$ thì có khả năng tìm thấy x theo p_o, nếu $x \ge k_n$ thì lại theo p_n.

Khi quá trình đưa ta tới một nút lá, ta sẽ tìm bản ghi r ứng với x trong khối ứng với lá đó bằng phép tìm kiếm tuần tự hoặc tìm kiếm nhị phân (sau khi trút nội dung của khối đó vào bộ nhớ trong).

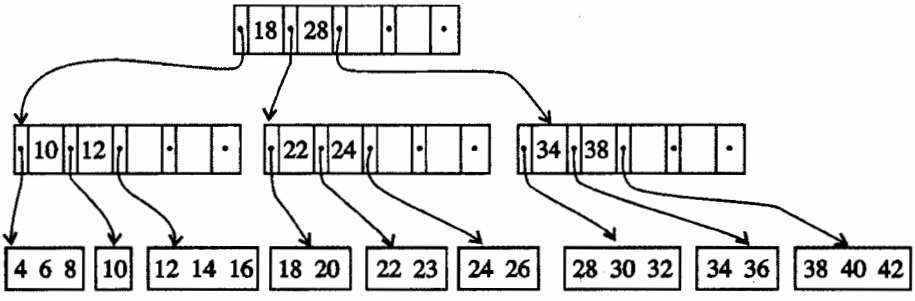
2) Bổ sung

Để bổ sung bản ghi r với khoá x vào B-cây trước hết ta phải tìm kiếm để định vị được nút lá L mà r sẽ thuộc vào đó.

Nếu có chỗ ở L cho r, ta sẽ bổ sung r vào L theo đúng thứ tự của nó. Trong trường hợp này không phải chỉnh lý gì ở nút tiền bối của L.

Nếu không có chỗ trống r trong L phải "xin" hệ quản lý tệp một khối mới L' và chuyển nửa cuối của L sang L', rồi bổ sung r vào vị trí của nó trong L hoặc L'. Giả sử nút P là cha của nút L thì phải đặt vào P cặp khoá và con trỏ (k', l') ứng với nút ở L': k' và l' được bổ sung vào ngay sau cặp (khoá, con trỏ) ứng với nút L; giá trị của k' là giá trị khoá nhỏ nhất trong L'. Nếu P đã có đủ m con trỏ rồi thì việc bổ sung (k', l') vào P sẽ dẫn tới việc phải tách P ra làm hai, tương tự như trường hợp đối với L ở trên. Ảnh hưởng của phép bổ sung này có thể truyền ứng lên trên qua các nút tiền bối của L, tới tận gốc, nghĩa là phải tạo ra một nút gốc mới với hai nửa của gốc cũ làm hai con của nó. Đó cũng là trường hợp mà một nút có nhỏ hơn m/2 con.

Hình sau đây mô tả B-cây nêu ở trên, sau phép bổ sung khoá 23 vào cây đó.



Hình 11.8

3) Loại bỏ

Để loại bỏ bản ghi r với khoá là x, trước hết cũng phải tìm nút lá L chứa r và loại bỏ r ra khỏi L nếu nó có.

Nếu r là bản ghi đầu tiên của L thì phải tìm đến nút P cha của L để chỉnh lại giá trị của khoá đầu tiên của L, đã được đặt tại đó.

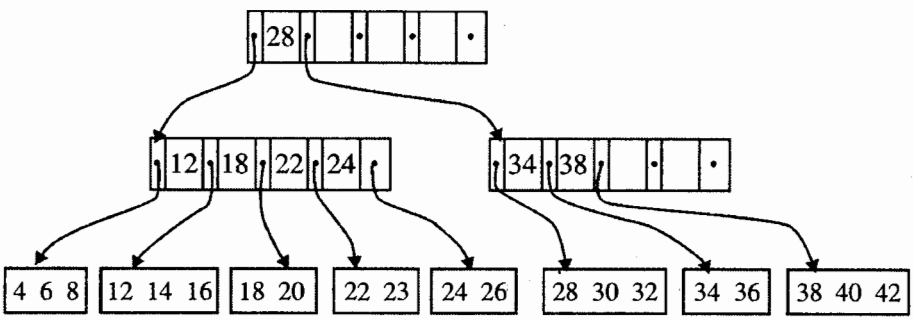
Nhưng, nếu L là con thứ nhất của P thì khoá đầu tiên của L không được đặt ở P, việc chỉnh lý ở đây không cần nữa. Tuy nhiên, rất có thể khoá đầu tiên đó lại xuất hiện ở một nút tiền bối nào đó của P. Vì vậy ta vẫn phải truyền ứng sự chỉnh lý giá trị khoá này ngược lên trên dọc theo đường đi đã được vạch ra từ gốc tới L.

Nếu sau phép loại bỏ, L trở thành rỗng thì phải trả L cho hệ quản lý tệp và phải chỉnh lý cặp (khoá, con trỏ) ứng với L trong P để phản ánh sự thay đổi này. Nếu bây giờ số con của P lại nhỏ hơn m/2 ta phải xem xét ngay nút P' ở sát bên trái P (hoặc sát bên phải) ở cùng một mức với P trong cây. Nếu P' có ít nhất là [m/2] + 1 con, ta sẽ phân bố đều các cặp (khoá, con trỏ) giữa P và P', giữ đúng thứ tự, để sao cho hai nút đó sẽ có ít nhất là [m/2]

con. Ta sẽ phải sửa lại các khoá và con trỏ tương ứng với P và P' ở chỗ cha của P và nếu cần thiết thì phải truyền ứng ảnh hưởng của sự thay đổi này đối với cả tiền bối khác của P nữa.

Nếu P' chỉ có đúng [m/2] con thôi, ta sẽ ghép P và P' lại thành một nút 2[m/2]-1 con (tối đa cũng chỉ là m con). Ta sẽ phải chỉnh lại khoá và con trỏ ứng với P' ở nút cha của P'. Nếu ảnh hưởng này lại truyền ứng ngược lên, dọc theo đường đi đã vạch cho tới tận gốc, thì có thể phải ghép hai con của gốc lại. Trong trường hợp này, kết quả của phép ghép tạo nên một nút gốc mới, và nút gốc cũ có thể trả về cho hệ quản lý tệp. Chiều cao của B-cây sẽ giảm đi 1.

Hình sau mô tả B-cây sau ghép loại bỏ khoá 10.



Hình 11.19

4) Phân tích đánh giá các phép toán đối với B-cây

Giả sử ta có một tệp n bản ghi trên B-cây cấp m. Mỗi nút lá chứa trung bình b bản ghi thì có $\lceil n/b \rceil$ lá. Đường đi dài nhất có thể có trên cây xảy ra nếu mỗi nút trong chỉ số con ít nhất có thể; nghĩa là m/2 con. Trong trường hợp này sẽ có $\lceil n/b \rceil / \frac{m}{2} = 2[n/b]/m$ cha của lá, $4\lceil n/b \rceil / m^2$ cha của cha của lá.v.v...

Nếu có j nút dọc đường đi từ gốc tới lá thì $2^{j-1}\lceil n/b \rceil / m^{j-1} \geq 1$ vì nếu không thì sẽ có ít hơn 1 nút ở mức của gốc.

Vậy $\lceil n/b \rceil \geq \left(\frac{m}{2}\right)^{j-1}$ và $j \leq 1 + \log_{m/2}\lceil n/b \rceil$

Như vậy nghĩa là số các phép truy nhập khối trong tìm kiếm tối đa cũng chỉ là $1 + \log_{m/2}\lceil n/b \rceil$

Nếu $n = 10^6$, $b = 10$, $m = 100$ thì số phép truy nhập khối sẽ là $j \leq 3,5$.

Đối với phép bổ sung hoặc loại bỏ, j khối sẽ bị truy nhập để xác định nút lá cần xem xét. Số lượng các khối cần truy nhập thêm để hoàn thành phép toán và truyền ứng ảnh hưởng của nó qua các nút có liên quan trên B-cây rất khó tính chính xác. Đa số trường hợp chỉ có một khối đó là khối ứng với nút lá chứa bản ghi được xét, nó cần được viết lại. Do đó người ta coi $2+\log_{m/2}[n/b]$ là số lượng trung bình các khối cần truy nhập khi bổ sung hoặc loại bỏ.

BÀI TẬP CHƯƠNG 11

11.1. Hãy thực hiện hợp nhất 5 đường với mạch sau:

45	54	67			
06	21	32	47	79	
17	20	25	50	86	92
34	59	63	71		
27	39	62			

11.2. Với 4 băng từ (3 băng vào, 1 băng ra) nếu thực hiện sắp xếp kiểu nhiều pha thì phân bố của các mạch trên băng từ sẽ thế nào? Hãy lập một bảng tương tự như bảng 11.4 để minh hoạ.

11.3. Áp dụng kỹ thuật làm tăng độ dài mạch theo phép lựa chọn thay thế với vùng đệm có kích thước m = 3, đối với dãy khoá đưa vào như sau:

14 26 03 15 06 35 19 28 22 40 12 09 33 20 01 17 37

Tổng cộng ta sẽ tạo được mấy mạch? Kích thước lớn nhất của mạch là bao nhiêu?

11.4. Giả sử có một tệp gồm 1.000.000.000 bản ghi, mỗi bản ghi chiếm 100 byte. Mỗi khoá có kích thước 1000 byte và con trỏ của mỗi khối chiếm 4 byte. Người ta tổ chức tệp này theo kiểu tệp băm. Như vậy phải dùng bao nhiêu khối cho các cụm? Cho bảng chỉ dẫn cụm?

Nếu tổ chức tệp theo kiểu B-cây thì thế nào?

11.5. Làm thế nào để tìm được phần tử (khoá) có giá trị lớn thứ k trong:

a) Tệp chỉ dẫn thưa?

b) Tệp kiểu B - cây?

TÀI LIỆU THAM KHẢO

[1] **Đỗ Xuân Lôi**: *"Cấu trúc dữ liệu"*. Đại học Bách khoa Hà Nội xuất bản - 1976.

[2] **Đỗ Xuân Lôi**: *"Sắp xếp và tìm kiếm dữ liệu trên máy tính điện tử"*. Đại học Bách khoa Hà Nội xuất bản - 1980.

[3] **Nguyễn Xuân Huy:** *"Thuật toán"*. Nhà xuất bản Thống kê - 1988.

[4] **Đoàn Nguyên Hải - Nguyễn Trung Trực - Nguyễn Anh Dũng:** *"Lập trình căn bản"*. Trung tâm điện toán - Đại học Bách khoa TP. HCM. 1991.

[5] **Jacques Arsac:** *"Nhập môn lập trình"*. Bản dịch của Nguyễn Chí Công - Đinh Văn Phong - Trần Ngọc Trí. Trung tâm hệ thống thông tin ISC - 1991.

[6] **J. Courtin, L. Kowarski:** *"Initiation à l'algorithmique et aux structures de données"*. Nhà xuất bản DUNOD - 1990.

[7] **Larry N. Hoff, Sanford Leesstma**: *"Lập trình nâng cao bằng PASCAL với cấu trúc dữ liệu"*. Bản dịch của Lê Minh Trung - Công ty SCITEC xuất bản 1991.

[8] **Donald Knuth**: *"The art of computer programming":* vol 1: Fundamental algorithms; vol 3: Sorting and searching. Addision Wesley Publishing Company 1973.

[9] **Niklaus Wirth**: *"Algorithms + data structures = programs"*. Prentice - Hall INC - 1976.

[10] **Mark Elson**: *"Data Structures"*. Science Research Associates INC - 1975.

[11] **Ellis Horowitz, Sartaj Sahani**: *"Fundamentals of data structures"*. Computer Science Press INC- 1976.

[12] **Robert J. Baron, Linde G. Shapiro**: *"Data structures and their implementation"*. Van Vostrand Reingold company - 1980.

[13] **A.V. Aho, J.E. Hopcroft, J.D. Ulmann:** *"Data structures and algorithms"*. Addison Wesley - 1983.

[14] Jean Paul Tremblay, Paul G. Sorenson: *"An introduction to data structures with applications"*. McGraw - Hill INC - 1984.

[15] **Paul Helman, Robert Veroff:** *"Intermediate problem solving and data structures"*. The Benjamin/cummings publishing company. INC - 1986.

MỤC LỤC