ĐẠI HỌC CẦN THƠ

CẤU TRÚC DỮ LIỆU
DATA STRUCTURES

TS. TRẦN CAO ĐỆ

08-2007

LỜI NÓI ĐẦU

Để đáp ứng nhu cầu học tập của các bạn sinh viên, nhất là sinh viên chuyên ngành tin học, Khoa Công Nghệ Thông Tin Trường Đại Học Cần Thơ chúng tôi đã tiến hành biên soạn các giáo trình, bài giảng chính trong chương trình học. Giáo trình môn Cấu Trúc Dữ Liệu này được biên soạn cơ bản dựa trên quyển "Data Structures and Algorithms" của Alfred V. Aho, John E. Hopcroft và Jeffrey D. Ullman do Addison-Wesley tái bản năm 1987. Giáo trình này cũng được biên soạn dựa trên kinh nghiệm giảng dạy nhiều năm môn Cấu Trúc Dữ Liệu và Giải Thuật của chúng tôi.

Tài liệu này được soạn theo đề cương chi tiết môn Cấu Trúc Dữ Liệu của sinh viên chuyên ngành tin học của Khoa Công Nghệ Thông Tin Trường Đại Học Cần Thơ. Mục tiêu của nó nhằm giúp các bạn sinh viên chuyên ngành có một tài liệu cô đọng dùng làm tài liệu học tập, nhưng chúng tôi cũng không loại trừ các đối tượng khác tham khảo, tự học. Chúng tôi nghĩ rằng các bạn sinh viên không chuyên tin và những người quan tâm tới cấu trúc dữ liệu và giải thuật sẽ tìm được trong này những điều hữu ích.

Giáo trình này được phát hành lần đầu tiên vào năm 1998, các lần chỉnh sửa bổ sung vào năm 1999 và 2001. Đây là lần chỉnh sửa vào năm 2007 nhằm chuyển các cài đặt bằng Pascal sang C cho sát hơn với sinh viên chuyên ngành - hiện đang được trang bị ngôn ngữ C như là ngôn ngữ lập trình căn bản. Tuy nhiên độc giả quen thuộc với ngôn ngữ Pascal vẫn có thể dùng bản phát hành năm 2001. Mặc dù đã rất cố gắng nhiều trong quá trình biên soạn giáo trình nhưng chắc chắn giáo trình sẽ còn nhiều thiếu sót và hạn chế. Rất mong nhận được sự đóng góp ý kiến quý báu của sinh viên và các bạn đọc để giáo trình ngày một hoàn thiện hơn.

Tái bản lần thứ 3

Cần thơ, ngày 10 tháng 08 năm 2007

Tác giả

Trần Cao Đệ

MỤC LỤC

PHẦN TỔNG QUAN

1. Mục đích yêu cầu

Môn học cấu trúc dữ liệu cung cấp cho sinh viên một khối lượng lớn các kiến thức cơ bản về các kiểu dữ liệu trừu tượng và các phép toán cơ bản trên kiểu dữ liệu đó. Sau khi học xong môn này, sinh viên sẽ:

- Nắm được khái niệm kiểu dữ liệu, kiểu dữ liệu trừu tượng.
- Nắm vững và cài đặt được các kiểu dữ liệu trừu tượng cơ bản như danh sách, ngăn xếp, hàng đợi, cây, tập hợp, bảng băm, đồ thị bằng một ngôn ngữ lập trình căn bản.
- Vận dụng được các kiểu dữ liệu trừu tượng để giải quyết bài toán đơn giản trong thực tế.

2. Đối tượng sử dụng

Giáo trình này được biên soạn để giảng dạy cho sinh viên bậc đại học chuyên ngành :

- Tin học
- Toán-Tin
- Lý-Tin
- Điện tử - Viễn thông và tự động hóa.

Giáo trình này còn có thể dùng để giảng dạy như là môn học cơ bản trong tất cả các ngành gần với Tin học

3. Nội dung giáo trình

Nội dung giáo trình gồm 5 chương và được thiết kế trong khuôn khổ 3 tín chỉ (45 tiết) của một môn học trong chương trình đại học, trong đó có khoảng 30 tiết giảng lý thuyết và 15 tiết bài tập mà giáo viên sẽ hướng dẫn cho sinh viên trên lớp. Bên cạnh tài liệu này còn có tài liệu *Thực hành cấu trúc dữ liệu*, do vậy nội dung giáo trình hơi chú trọng về các cấu trúc dữ liệu và các giải thuật trên các cấu trúc dữ liệu đó hơn là các chương trình hoàn chỉnh trong ngôn ngữ lập trình C.

Chương 1: Trình bày cách tiếp cận từ một bài toán đến chương trình, nó bao gồm mô hình hoá bài toán, thiết lập cấu trúc dữ liệu theo mô hình bài toán, viết giải thuật giải quyết bài toán và các bước tinh chế giải thuật đưa đến cài đặt cụ thể trong một ngôn ngữ lập trình

Chương 2: Trình bày kiểu dữ liệu trừu tượng danh sách, các cấu trúc dữ liệu để cài đặt danh sách. Ngăn xếp và hàng đợi cũng được trình bày trong chương này như là hai cấu trúc danh sách đặc biệt. Ứng dụng ngăn xếp để khử đệ qui của chương trình được coi là một ứng dụng quan trọng của cấu trúc ngăn xếp. Ngoài ra chúng tôi còn gợi ý một số ứng dụng của cấu trúc hàng đợi. Cuối chương dành để trình bày cấu trúc danh sách liên kết kép cho những bài toán cần duyệt danh sách theo hai chiều xuôi, ngược một cách thuận lợi.

Chương này có nhiều cài đặt tương đối chi tiết để các bạn sinh viên mới tiếp cận với lập trình có cơ hội nâng cao khả năng lập trình trong ngôn ngữ C đồng thời cũng nhằm minh hoạ việc cài đặt một kiểu dữ liệu trừu tượng trong một ngôn ngữ lập trình cụ thể.

Chương 3: Giới thiệu về kiểu dữ liệu trừu tượng cây, khái niệm cây tổng quát, các phép duyệt cây tổng quát và cài đặt cây tổng quát. Kế đến chúng tôi trình bày về cây nhị phân, các cách cài đặt cây nhị phân và ứng dụng cây nhị phân để xây dựng mã Huffman. Cuối cùng, chúng tôi trình bày cây tìm kiếm nhị phân như là một ứng dụng của cây nhị phân để lưu trữ và tìm kiếm dữ liệu.

Chương 4: Nói về kiểu dữ liệu trừu tượng tập hợp, các cách đơn giản để cài đặt tập hợp như cài đặt bằng vectơ bít hay bằng danh sách có hoặc không có thứ tự. Phần chính của chương này trình bày cấu trúc dữ liệu tự điển, đó là tập hợp với ba phép toán thêm, xoá và tìm kiếm phần tử, cùng với các cấu trúc thích hợp cho nó như là bảng băm và hàng ưu tiên.

Chương 5: Trình bày kiểu dữ liệu trừu tượng đồ thị, các cách biểu diễn đồ thị hay là cài đặt đồ thị. Ở đây chúng tôi cũng trình bày các phép duyệt đồ thị bao gồm duyệt theo chiều rộng và duyệt theo chiều sâu một đồ thị. Do hạn chế về thời lượng lên lớp nên chúng tôi không tách riêng ra để trình bày đồ thị có hướng, đồ thị vô hướng nhưng chúng tôi sẽ phân biệt nó ở những chổ cần thiết. Chương này đề cập một số bài toán thường gặp trên đồ thị như là bài toán tìm đường đi ngắn nhất, bài toán tìm cây phủ tối thiểu.…

Trong chương trình đại học, Lý thuyết đồ thị được nghiên cứu sâu hơn trong các môn học như Toán rời rạc và Lý thuyết đồ thị, vì vậy chương này chỉ giới thiệu một số bài toán đơn giản nhất để sinh viên vận dụng cách cài đặt đồ thị và các phép toán trên đồ thị để lập trình giải các bài toán đó.

4. Kiến thức tiên quyết

Để học tốt môn *Cấu trúc dữ liệu* này, sinh viên cần phải có các kiến thức cơ bản sau:

- Kiến thức và kỹ năng lập trình căn bản. Việc dùng một ngôn ngữ lập trình cụ thể không phải là một yêu cầu bắt buộc. Giáo trình này minh họa các cài đặt cụ thể bằng ngôn ngữ C đó là một ngôn ngữ quen thuộc đối với sinh viên chuyên ngành Tin học.
- Kiến thức toán rời rạc.

5. Danh mục tài liệu tham khảo

[1] **Aho, A. V. , J. E. Hopcroft, J. D. Ullman,** "*Data Structure and Algorihtms*", Addison–Wesley; 1983.

[2] **Michel T. Goodrich, Roberto Tamassia, David Mount,** "*Data Structures and Algorithms in C++*", Weley International Edition; 2004.

[3] **Đỗ Xuân Lôi,** "*Cấu trúc dữ liệu và giải thuật*", Nhà xuất bản khoa học và kỹ thuật, Hà nội, 1995.

[4] **N. Wirth** " *Cấu trúc dữ liệu + giải thuật= Chương trình*", 1983.

[5] **Nguyễn Trung Trực**, "*Cấu trúc dữ liệu*", ĐHBK TP.HCM, 1990.

[6] **Lê Minh Trung,** "*Lập trình nâng cao bằng Pascal với các cấu trúc dữ liệu* "; 1997.

[7] **Ngô Trung Việt,** "*Ngôn ngữ lập trình C và C++ Bài giảng- Bài tập – Lời giải mẫu*"; NXB Giao thông vận tải, 2000.

[8**] Nguyễn Đình Tê, Hoàng Đức Hải**, " *Giáo trình lý thuyết và bài tập ngôn ngữ C*" , NXB Giáo dục, 1998.

[9] **Lê Xuân Trường**, " *Giáo trình cấu trúc dữ liệu bằng ngôn ngữ C++*", NXB thống kê, 1999.

[10] **Nguyễn Thanh Thủy, Nguyễn Quang Huy**," *Bài tập lập trình ngôn ngữ C*", NXB Khoa học kỹ thuật, 1999.

[11] http://courses.cs.hcmuns.edu.vn/ctdl1/Ctdl1/index.html

[12] http://www.cs.ualberta.ca/~holte/T26/top.realTop.html

[13] http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/ds_ToC.html

CHƯƠNG I: MỞ ĐẦU

I. TỪ BÀI TOÁN ĐẾN CHƯƠNG TRÌNH

1. Mô hình hóa bài toán

Để giải một bài toán trong thực tế bằng máy tính ta phải bắt đầu từ việc xác định bài toán. Nhiều thời gian và công sức bỏ ra để xác định bài toán cần giải quyết, tức là phải trả lời rõ ràng câu hỏi "phải làm gì?" sau đó là "làm như thế nào?". Thông thường, khi khởi đầu, hầu hết các bài toán là không đơn giản, không rõ ràng. Để giảm bớt sự phức tạp của bài toán thực tế, ta phải hình thức hóa nó, nghĩa là phát biểu lại bài toán thực tế thành một bài toán hình thức (hay còn gọi là mô hình toán). Có thể có rất nhiều bài toán thực tế có cùng một mô hình toán.

Ví dụ 1: Tô màu bản đồ thế giới.

Ta cần phải tô màu cho các nước trên bản đồ thế giới. Trong đó mỗi nước đều được tô một màu và hai nước láng giềng (có biên giới chung) thì phải được tô bằng hai màu khác nhau. Hãy tìm một phương án tô màu các nước trên bản đồ sao cho số màu sử dụng là ít nhất.

Mô hình hóa bài toán này có thể hiểu là: tìm cách biểu diễn bài toán một cách trừu tượng hơn để gạt bỏ các chi tiết không cần thiết. Ví dụ ta chỉ cần ghi lại tất cả các nước trên bản đồ và mối quan hệ "láng giềng" giữa hai nước, tức là chỉ cần biết nước nào với nước nào có biên giới chung. Một cách mô hình hóa là: vẽ mỗi nước như một điểm; nếu hai nước có chung biên giới ta sẽ vẽ một đường nối hai điểm tương ứng. Vậy bản đồ thế giới và mối quan hệ láng giềng giữa các nước đã được biểu diễn bằng một đồ thị (graph): mỗi đỉnh là một nước, hai nước có biên giới chung thì hai điểm tương ứng sẽ được nối với nhau bởi một cung.

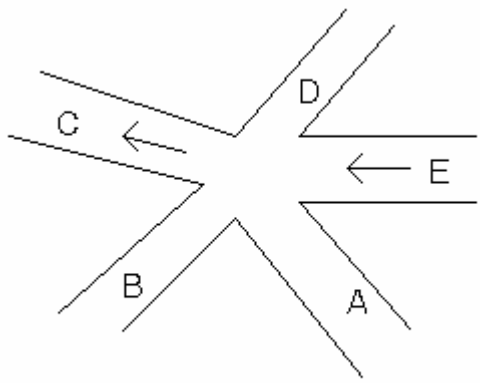
Bài toán tô màu cho bản đồ thế giới trở thành:

- Tìm cách tô màu cho tất cả các đỉnh đồ thị sao cho hai đỉnh có cạnh nối nhau thì phải được tô bằng hai màu khác nhau
- Số màu được sử dụng là ít nhất.

Ví dụ 2: Đèn giao thông

Cho một ngã năm như hình I.1, trong đó C và E là các đường một chiều theo chiều mũi tên, các đường khác là hai chiều. Hãy thiết kế một bảng đèn hiệu điều khiển giao thông tại ngã năm này một cách hợp lý, nghĩa là: phân chia các lối đi tại ngã năm này thành các nhóm, mỗi nhóm gồm các lối đi có thể cùng đi đồng thời nhưng không xảy ra tai nạn giao thông (các hướng đi không cắt nhau), và số lượng nhóm là ít nhất có thể được.

Ta có thể xem đầu vào (input) của bài toán là tất cả các lối đi tại ngã năm này, đầu ra (output) của bài toán là các nhóm lối đi có thể đi đồng thời mà không xảy ra tai nạn giao thông, mỗi nhóm sẽ tương ứng với một pha điều khiển của đèn hiệu, vì vậy ta phải tìm kiếm lời giải với số nhóm là ít nhất để giao thông không bị tắc nghẽn vì phải chờ đợi quá lâu.



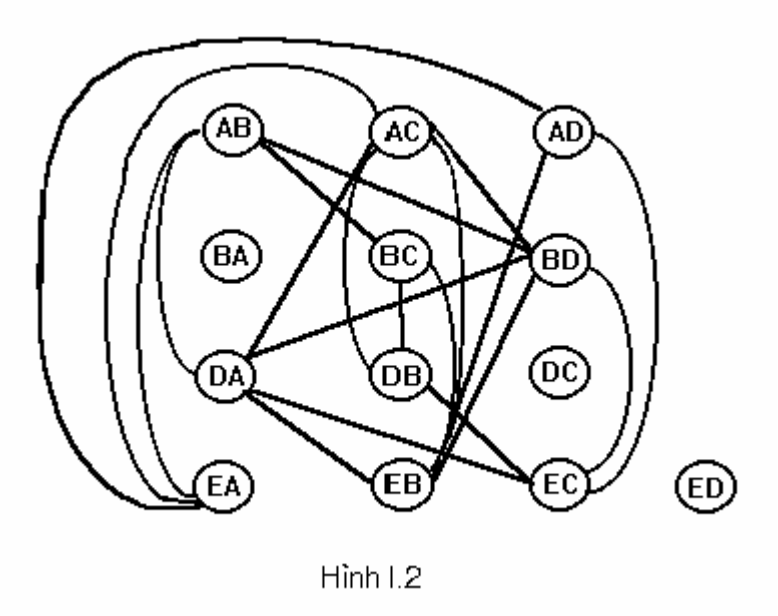
Hình I.1

Tương tự như trong ví dụ 1, trước hết ta tìm cách mô hình hóa bài toán, tức là tìm cách trừu tượng hóa để gạt bỏ các chi tiết không cần thiết. Để giải bài toán vừa đặt ra, ít nhất ta phải biết ở ngã năm có bao nhiêu lối đi, lối đi nào với lối đi nào có hướng giao thông không cắt nhau, lối đi nào với lối đi nào có hướng giao thông cắt nhau. Ta nhận thấy rằng tại ngã năm này có 13 lối đi: AB, AC, AD, BA, BC, BD, DA, DB, DC, EA, EB, EC, ED. Tất nhiên, để có thể giải được bài toán ta phải tìm một cách nào đó để thể hiện mối liên quan giữa các lối đi này. Lối nào với lối nào không thể đi đồng thời, lối nào và lối nào có thể đi đồng thời. Ví dụ cặp AB và EC có thể đi đồng thời, nhưng AD và EB thì không, vì các hướng giao thông cắt nhau. Ở đây ta sẽ dùng một sơ đồ trực quan như sau: tên của 13 lối đi được viết lên mặt phẳng, hai lối đi nào nếu đi đồng thời sẽ xảy ra đụng nhau (tức là hai hướng đi cắt qua nhau) ta nối lại bằng một đoạn thẳng, hoặc cong, hoặc ngoằn ngoèo tuỳ thích. Ta sẽ có một sơ đồ như hình I.2. Như vậy, trên sơ đồ này, hai lối đi có cạnh nối lại với nhau là hai lối đi không thể cho đi đồng thời.

Với cách biểu diễn như vậy ta đã có một đồ thị, tức là ta đã mô hình hoá bài toán giao thông ở trên theo mô hình toán là đồ thị; trong đó mỗi lối đi trở thành một đỉnh của đồ thị, hai lối đi không thể cùng đi đồng thời được nối nhau bằng một đoạn ta gọi là cạnh của đồ thị. Bây giờ ta phải xác định các nhóm, với số nhóm ít nhất, mỗi nhóm gồm các lối đi có thể đi đồng thời, nó ứng với một pha của đèn hiệu điều khiển giao thông. Giả sử rằng, ta dùng màu để tô lên các đỉnh của đồ thị này sao cho:

- Các lối đi cho phép cùng đi đồng thời sẽ có cùng một màu: Dễ dàng nhận thấy rằng hai đỉnh có cạnh nối nhau sẽ không được tô cùng màu.
- Số nhóm là ít nhất: ta phải tính toán sao cho số màu được dùng là ít nhất.

Tóm lại, ta phải giải quyết bài toán sau:



Hình I.2

Tô màu cho đồ thị ở hình I.2 sao cho:

- Hai đỉnh có cạnh nối nhau thì phải được tô bằng hai màu khác nhau
- Số màu được sử dụng là ít nhất.

Hai bài toán thực tế "tô màu bản đồ thế giới" và "đèn giao thông" xem ra rất khác biệt nhau nhưng sau khi mô hình hóa, chúng thực chất chỉ là một, đó là bài toán "tô màu đồ thị".

Đối với một bài toán đã được hình thức hoá, chúng ta có thể tìm kiếm cách giải trong thuật ngữ của mô hình đó và xác định có hay không một chương trình có sẵn để giải. Nếu không có một chương trình như vậy thì ít nhất chúng ta cũng có thể tìm được những gì đã biết về mô hình và dùng các tính chất của mô hình để xây dựng một giải thuật tốt.

2. Giải thuật (algorithms)

Khi đã có mô hình thích hợp cho một bài toán ta cần cố gắng tìm cách giải quyết bài toán trong mô hình đó. Khởi đầu là tìm một giải thuật, đó là một chuỗi hữu hạn các chỉ thị (instruction) mà mỗi chỉ thị có một ý nghĩa rõ ràng và thực hiện được trong một lượng thời gian hữu hạn.

Knuth (1973) định nghĩa giải thuật là một chuỗi hữu hạn các thao tác để giải một bài toán nào đó. Các tính chất quan trọng của giải thuật là:

- *Hữu hạn* (finiteness): giải thuật phải luôn luôn kết thúc sau một số hữu hạn bước.
- *Xác định* (definiteness): mỗi bước của giải thuật phải được xác định rõ ràng và phải được thực hiện chính xác, nhất quán.
- *Hiệu quả* (effectiveness): các thao tác trong giải thuật phải được thực hiện trong một lượng thời gian hữu hạn.

Ngoài ra một giải thuật còn phải có đầu vào (input) và đầu ra (output).

Nói tóm lại, một giải thuật phải giải quyết xong công việc khi ta cho dữ liệu vào. Có nhiều cách để thể hiện giải thuật: dùng lời, dùng lưu đồ, ... Và một lối dùng rất phổ biến là dùng ngôn ngữ giả, đó là sự kết hợp của ngôn ngữ tự nhiên và các cấu trúc của ngôn ngữ lập trình.

Ví dụ: Thiết kế giải thuật để giải bài toán *" tô màu đồ thị"* trên

Bài toán tô màu cho đồ thị không có giải thuật tốt để tìm *lời giải tối ưu*, tức là, không có giải thuật nào khác hơn là "*thử tất cả các khả năng*" hay "*vét cạn*" tất cả các trường hợp có thể có, để xác định cách tô màu cho các đỉnh của đồ thị sao cho *số màu dùng là ít nhất*. Thực tế, ta chỉ có thể "vét cạn" trong trường hợp đồ thị có số đỉnh nhỏ, trong trường hợp ngược lại ta không thể "vét cạn" tất cả các khả năng trong một lượng thời gian hợp lý, do vậy ta phải suy nghĩ cách khác để giải quyết vấn đề:

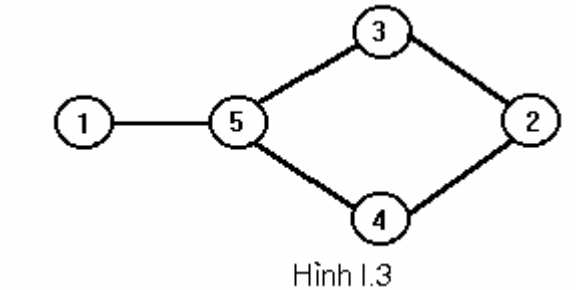
- *Thêm thông tin vào bài toán* để đồ thị có một số tính chất đặc biệt và dùng các tính chất đặc biệt này ta có thể dễ dàng tìm lời giải, hoặc
- *Thay đổi yêu cầu bài toán một ít* cho dễ giải quyết, nhưng lời giải tìm được chưa chắc là lời giải tối ưu. Một cách làm như thế đối với bài toán trên là "*Cố gắng tô màu cho đồ thị bằng ít màu nhất một cách nhanh chóng*". Ít màu nhất ở đây có nghĩa là số màu mà ta tìm được không phải luôn luôn là số màu của lời giải tối ưu (ít nhất), nhưng trong đa số trường hợp thì nó sẽ trùng với đáp số của lời giải tối ưu và nếu có chênh lệch thì nó "không chênh lệch nhiều" so với lời giải tối ưu, bù lại ta không phải "vét cạn" mọi khả năng có thể! Nói khác đi, ta không dùng giải thuật "vét cạn" mọi khả năng để tìm lời giải tối ưu mà tìm một giải pháp để đưa ra lời giải hợp lý một cách khả thi về thời gian. Một giải pháp như thế gọi là một HEURISTIC.

HEURISTIC cho bài toán tô màu đồ thị, thường gọi là giải thuật "háu ăn" (GREEDY) là:

- *Chọn một đỉnh chưa tô màu và tô nó bằng một màu mới C nào đó*.
- *Duyệt danh sách các đỉnh chưa tô màu. Đối với một đỉnh chưa tô màu, xác định xem nó có kề với một đỉnh nào được tô bằng màu C đó không. Nếu không có, tô nó bằng màu C đó*.

Ý tưởng của Heuristic này là hết sức đơn giản: dùng một màu để tô cho nhiều đỉnh nhất có thể được (các đỉnh được xét theo một thứ tự nào đó), khi không thể tô được nữa với màu đang dùng thì dùng một màu khác. Như vậy ta có thể "hi vọng" là số màu cần dùng sẽ ít nhất.

Ví dụ: Đồ thị hình I.3 và cách tô màu cho nó



Hình I.3

Tô theo GREEDY (xét lần lượt theo số thứ tự các đỉnh)	Tối ưu (thử tất cả các khả năng)
1: đỏ; 2: đỏ 3: xanh;4: xanh 5: vàng	1,3,4 : đỏ 2,5 : xanh

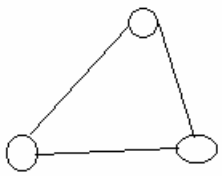
Rõ ràng cách tô màu trong giải thuật "háu ăn" không luôn luôn cho lời giải tối ưu nhưng nó được thực hiện một cách nhanh chóng.

Trở lại bài toán giao thông ở trên và áp dụng HEURISTIC Greedy cho đồ thị trong hình I.2 (theo thứ tự các đỉnh đã liệt kê ở trên), ta có kết quả:

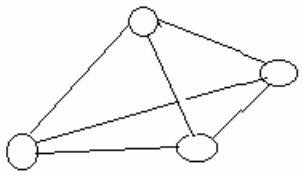
- Tô màu xanh cho các đỉnh: AB,AC,AD,BA,DC,ED
- Tô màu đỏ cho các đỉnh: BC,BD,EA
- Tô màu tím cho các đỉnh: DA,DB
- Tô màu vàng cho các đỉnh: EB,EC

Như vậy ta đã tìm ra một lời giải là dùng 4 màu để tô cho đồ thị hình I.2. Như đã nói, lời giải này không chắc là lời giải tối ưu. Vậy liệu có thể dùng 3 màu hoặc ít hơn 3 màu không? Ta có thể trở lại mô hình của bài toán và dùng tính chất của đồ thị để kiểm tra kết quả. Nhận xét rằng:

- Một đồ thị có k đỉnh và mỗi cặp đỉnh bất kỳ đều được nối nhau thì phải dùng k màu để tô. Hình I.4 chỉ ra hai ví dụ với k=3 và k=4.



phải dùng 3 màu để tô cho đồ thị này



phải dùng 4 màu để tô cho đồ thị này

Hình I.4

- Một đồ thị trong đó có k đỉnh mà mỗi cặp đỉnh bất kỳ trong k đỉnh này đều được nối nhau thì không thể dùng ít hơn k màu để tô cho đồ thị.

Đồ thị trong hình I.2 có 4 đỉnh: AC,DA,BD,EB mà mỗi cặp đỉnh bất kỳ đều được nối nhau vậy đồ thị hình I.2 không thể tô với ít hơn 4 màu. Điều này khẳng định rằng lời giải vừa tìm được ở trên trùng với lời giải tối ưu.

Như vậy ta đã giải được bài toán giao thông đã cho. Lời giải cho bài toán là 4 nhóm, mỗi nhóm gồm các lối có thể đi đồng thời, nó ứng với một pha điều khiển của đèn hiệu. Ở đây cần nhấn mạnh rằng, sở dĩ ta có lời giải một cách rõ ràng chặt chẽ như vậy là vì chúng ta đã giải bài toán thực tế này bằng cách mô hình hoá nó theo một mô hình thích hợp (mô hình đồ thị) và nhờ các kiến thức trên mô hình này (bài toán tô

màu và heuristic để giải) ta đã giải quyết được bài toán. Điều này khẳng định vai trò của việc mô hình hoá bài toán.

3. Ngôn ngữ giả và tinh chế từng bước (Pseudo-language and stepwise refinement)

Một khi đã có *mô hình thích hợp* cho bài toán, ta cần hình *thức hoá một giải thuật* trong thuật ngữ của mô hình đó. Khởi đầu là viết những mệnh đề tổng quát rồi tinh chế dần thành những chuỗi mệnh đề cụ thể hơn, cuối cùng là các chỉ thị thích hợp trong một ngôn ngữ lập trình. Chẳng hạn với heuristic GREEDY, giả sử đồ thị là G, heuristic sẽ xác định một tập hợp **Newclr** các đỉnh của G được tô cùng một màu, mà ta gọi là màu mới C ở trên. Để tiến hành tô màu hoàn tất cho đồ thị G thì Heuristic này phải được gọi lặp lại cho đến khi toàn thể các đỉnh đều được tô màu.

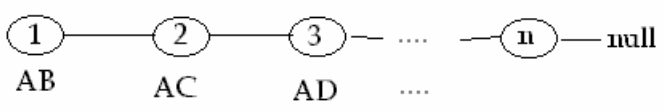
```
PROCEDURE GREEDY ( var G: GRAPH ; var Newclr: SET );
begin
{1}     Newclr := ∅;
{2}     for (mỗi đỉnh v chưa tô màu của G) do
{3}             if (v không được nối với một đỉnh nào trong Newclr) then begin
{4}                     đánh dấu v đã được tô màu;
{5}                     thêm v vào Newclr;
                end;
end;
```

Thủ tục GREEDY với ngôn ngữ giả PASCAL

Trong thủ tục bằng ngôn ngữ giả ở trên, chúng ta đã dùng một số từ khoá của ngôn ngữ PASCAL xen lẫn các mệnh đề tiếng Việt. Điều đặc biệt nữa là ta dùng các kiểu GRAPH, SET có vẻ xa lạ, chúng là các "kiểu dữ liệu trừu tượng" mà sau này chúng ta sẽ viết bằng các khai báo thích hợp trong ngôn ngữ lập trình cụ thể. Dĩ nhiên, để cài đặt thủ tục này ta phải cụ thể hoá dần những mệnh đề bằng tiếng việt ở trên cho đến khi mỗi mệnh đề tương ứng với một đoạn mã thích hợp của ngôn ngữ lập trình. Chẳng hạn mệnh đề **if** ở {3} có thể chi tiết hoá hơn nữa như sau:

```
PROCEDURE GREEDY ( var G: GRAPH ; var Newclr: SET );
begin
{1}     Newclr:= ∅;
{2}     for (mỗi đỉnh v chưa tô màu của G) do begin
{3.1}           found:=false;
{3.2}           for (mỗi đỉnh w trong Newclr) do
{3.3}                   if (có cạnh nối giữa v và w) then
{3.4}                           found:=true;
{3.5}           if found=false then begin
{4}                     đánh dấu v đã được tô màu;
{5}                     thêm v vào Newclr;
                end;
        end;
end;
```

Thủ tục GREEDY đã tinh chế mệnh đề if ở {3}



Hình I.5: Biểu diễn tập hợp các đỉnh như là một danh sách (LIST)

GRAPH và SET ta coi như tập hợp. Có nhiều cách để biểu diễn tập hợp trong ngôn ngữ lập trình, để đơn giản ta xem các tập hợp như là một danh sách (LIST) các số nguyên biểu diễn chỉ số của các đỉnh và kết thúc bằng một giá trị đặc biệt NULL (hình I.5). Với những qui ước như vậy ta có thể tinh chế giải thuật GREEDY một bước nữa như sau:

```
PROCEDURE GREEDY ( var G: GRAPH ; var Newclr: LIST );
var     found:boolean;
        v,w :integer;
begin
        Newclr:= ∅;
        v:= đỉnh đầu tiên chưa được tô màu trong G;
        while v<>null do begin
                found:=false;
                w:=đỉnh đầu tiên trong newclr;
                while( w<>null) and (not found) do begin
                        if có cạnh nối giữa v và w then
                                found:=true;
                        else w:= đỉnh kế tiếp trong newclr;
                end;
                if found=false then begin
                        đánh dấu v đã được tô màu;
                        thêm v vào Newclr;
                end;
                v:= đỉnh chưa tô màu kế tiếp trong G;
        end;
end;
```

Thủ tục GREEDY đã tinh chế thêm

Việc dùng ngôn ngữ giả nhằm mục đích phát họa ý tưởng của giải thuật, tránh sa đà vào cú pháp của ngôn ngữ. Tuy nhiên, ở các bước tinh chế về sau, thủ tục ngôn ngữ giả càng gần giống với chương trình trong một ngôn ngữ lập trình. Vì vậy, việc chọn ngôn ngữ giả tựa PASCAL hay tựa C hay tựa một một ngữ lập trình nào khác là tùy thuộc vào thói quen của người sử dụng, vào sự quen thuộc với ngôn ngữ lập trình.

Nếu người dùng quen thuộc với ngôn ngữ C có thể viết thủ tục với ngôn ngữ giả tựa C như sau :

```
void GREEDY ( GRAPH& G, SET& Newclr ){
/*1*/ Newclr = ∅;
/*2*/ for (mỗi đỉnh v chưa tô màu của G)
/*3*/  if (v không được nối với một đỉnh nào trong Newclr){
/*4*/                    đánh dấu v đã được tô màu;
/*5*/                    thêm v vào Newclr;
                }
}
```

Thủ tục GREEDY với ngôn ngữ giả C

Thủ tục tinh chế được viết tựa C như sau :

```
void GREEDY ( GRAPH& G, SET& Newclr )
{
/*1*/     Newclr= ∅;
/*2*/     for (mỗi đỉnh v chưa tô màu của G) {
/*3.1*/           int found=0;
/*3.2*/           for (mỗi đỉnh w trong Newclr)
/*3.3*/                   if (có cạnh nối giữa v và w)
/*3.4*/                           found=1;
/*3.5*/           if (!found) {
/*4*/                     đánh dấu v đã được tô màu;
/*5*/                     thêm v vào Newclr;
                }
        }
}
```

Tinh chế thêm một bước nữa:

```
void GREEDY ( GRAPH& G, LIST& Newclr ){
    Newclr= ∅;
    int v= đỉnh đầu tiên chưa được tô màu trong G;
    while (v<>null) {
        int found=0;
        int w=đỉnh đầu tiên trong newclr;
        while( w<>null) && (!found)
              If (có cạnh nối giữa v và w)  found=1;
              else w= đỉnh kế tiếp trong newclr;
        if (!found) {
          Đánh dấu v đã được tô màu;
          Thêm v vào Newclr;
        }
        v= đỉnh chưa tô màu kế tiếp trong G;
    }
}
```

4. Tóm tắt

Từ những thảo luận trên chúng ta có thể tóm tắt các bước tiếp cận với một bài toán bao gồm:

1. Mô hình hoá bài toán bằng một mô hình toán học thích hợp.
2. Tìm giải thuật trên mô hình này. Giải thuật có thể mô tả một cách không hình thức, tức là nó chỉ nêu phương hướng giải hoặc các bước giải một cách tổng quát.
3. Phải hình thức hoá giải thuật bằng cách viết một thủ tục bằng ngôn ngữ giả, rồi chi tiết hoá dần ("mịn hoá") các bước giải tổng quát ở trên, kết hợp với việc dùng các kiểu dữ liệu trừu tượng và các cấu trúc điều khiển trong ngôn ngữ lập trình để mô tả giải thuật. Ở bước này, nói chung, ta có một giải thuật tương đối rõ ràng, nó gần giống như một chương trình được viết trong ngôn ngữ lập trình, nhưng nó không phải là một chương trình chạy được vì trong khi viết giải thuật ta không chú trọng nặng đến cú pháp của ngôn ngữ và các *kiểu dữ liệu* còn ở mức trừu tượng chứ không phải là các khai báo cài đặt kiểu trong ngôn ngữ lập trình.
4. Cài đặt giải thuật trong một ngôn ngữ lập trình cụ thể (Pascal,C,...). Ở bước này ta dùng các cấu trúc dữ liệu được cung cấp trong ngôn ngữ, ví dụ Array, Record,... để thể hiện các kiểu dữ liệu trừu tượng, các bước của giải thuật được thể hiện bằng các lệnh và các cấu trúc điều khiển trong ngôn ngữ lập trình được dùng để cài đặt giải thuật.

Tóm tắt các bước như sau:

Mô hình toán học	Kiểu dữ liệu trừu tượng	Cấu trúc dữ liệu
Giải thuật không hình thức	Chương trình ngôn ngữ giả	Chương trình Pascal, C,...

II. KIỂU DỮ LIỆU TRỪU TƯỢNG (ABSTRACT DATA TYPE -ADT)

1. Khái niệm trừu tượng hóa

Trong tin học, trừu tượng hóa nghĩa là đơn giản hóa, làm cho nó sáng sủa hơn và dễ hiểu hơn. Trừu tượng hóa là che đi những chi tiết, làm nổi bật cái tổng thể. Trừu tượng hóa có thể thực hiện trên hai khía cạnh là *trừu tượng hóa dữ liệu* và *trừu tượng hóa chương trình*.

2. Trừu tượng hóa chương trình

Trừu tượng hóa chương trình là sự định nghĩa các chương trình con để tạo ra các phép toán trừu tượng (sự tổng quát hóa của các phép toán nguyên thủy). Chẳng hạn ta có thể tạo ra một chương trình con Matrix_Mult để thực hiện phép toán nhân hai ma trận. Sau khi Matrix_mult đã được tạo ra, ta có thể dùng nó như một phép toán nguyên thủy (giống như phép nhân hai số nguyên).

Trừu tượng hóa chương trình cho phép phân chia chương trình thành các chương trình con. Sự phân chia này sẽ che dấu tất cả các lệnh cài đặt chi tiết trong các chương

trình con. Ở cấp độ chương trình chính, ta chỉ thấy lời gọi các chương trình con và điều này được gọi là sự bao gói.

Ví dụ như một chương trình nhân hai ma trận được viết bằng trừu tượng hóa có thể là:

```
void Main() {
        Input_Matrix(A);
        Input_Matrix(B);
        Matrix_mult(A,B,C);
        Print_Matrix(C);
}
```

Trong chương trình trên, Input_Matrix, Matrix_mult và Print_Matrix là các phép toán trừu tượng. Chúng che dấu bên trong rất nhiều lệnh phức tạp mà ở cấp độ chương trình chính ta không nhìn thấy được. Còn A,B,C là các biến thuộc kiểu dữ liệu trừu tượng *Matrix* mà ta sẽ xét sau.

? Hàm Main() ở trên (được viết theo cách gọi các phép toán trừu tượng) có lệ thuộc vào cách cài đặt kiểu dữ liệu (kiểu Matrix) hay không?

3. Trừu tượng hóa dữ liệu

Trừu tượng hóa dữ liệu là định nghĩa các kiểu dữ liệu trừu tượng.

Một kiểu dữ liệu trừu tượng (ADT) là một mô hình toán học cùng với một tập hợp các phép toán (operator) trừu tượng được định nghĩa trên mô hình đó. Ví dụ tập hợp số nguyên cùng với các phép toán hợp, giao, hiệu là một kiểu dữ liệu trừu tượng.

Trong một ADT các phép toán có thể thực hiện trên các đối tượng (toán hạng) không chỉ thuộc ADT đó, cũng như kết quả không nhất thiết phải thuộc ADT. Tuy nhiên phải có ít nhất một toán hạng hoặc kết quả phải thuộc ADT đang xét.

ADT là sự tổng quát hoá của các kiểu dữ liệu nguyên thuỷ.

Để minh hoạ ta có thể xét bản phác thảo cuối cùng của thủ tục GREEDY. Ta đã dùng một danh sách (LIST) các số nguyên và các phép toán trên danh sách **Newclr** là:

- Tạo một danh sách rỗng.
- Lấy phần tử đầu tiên trong danh sách và trả về giá trị null nếu danh sách rỗng.
- Lấy phần tử kế tiếp trong danh sách và trả về giá trị null nếu không còn phần tử kế tiếp.
- Thêm một số nguyên vào danh sách.

Nếu chúng ta viết các chương trình con thực hiện các phép toán này, thì ta dễ dàng thay các mệnh đề hình thức trong giải thuật bằng các câu lệnh đơn giản

Câu lệnh	Mệnh đề hình thức
MAKENULL(newclr)	newclr= ∅
w=FIRST(newclr)	w=phần tử đầu tiên trong newclr
w=NEXT(w,newclr)	w=phần tử kế tiếp trong newclr
INSERT(v,newclr)	Thêm v vào newclr

Điều này cho thấy sự thuận lợi của ADT, đó là ta có thể định nghĩa một kiểu dữ liệu tuỳ ý cùng với các phép toán cần thiết trên nó rồi chúng ta dùng như là các đối tượng nguyên thuỷ. Hơn nữa chúng ta có thể cài đặt một ADT bằng bất kỳ cách nào, chương trình dùng chúng cũng không thay đổi, chỉ có các chương trình con biểu diễn cho các phép toán của ADT là thay đổi.

Cài đặt ADT là sự thể hiện các phép toán mong muốn (các phép toán trừu tượng) thành các câu lệnh của ngôn ngữ lập trình, bao gồm các khai báo thích hợp và các thủ tục thực hiện các phép toán trừu tượng. Để cài đặt, ta chọn một **cấu trúc dữ liệu** thích hợp có trong ngôn ngữ lập trình hoặc là một cấu trúc dữ liệu phức hợp được xây dựng lên từ các kiểu dữ liệu cơ bản của ngôn ngữ lập trình.

? Sự khác nhau giữa kiểu dữ liệu và kiểu dữ liệu trừu tượng là gì?

III. KIỂU DỮ LIỆU - CẤU TRÚC DỮ LIỆU VÀ KIỂU DỮ LIỆU TRỪU TƯỢNG (DATA TYPES, DATA STRUCTURES, ABSTRACT DATA TYPES)

Mặc dù các thuật ngữ kiểu dữ liệu (hay kiểu - data type), cấu trúc dữ liệu (data structure), kiểu dữ liệu trừu tượng (abstract data type) nghe như nhau, nhưng chúng có ý nghĩa rất khác nhau.

Kiểu dữ liệu là một tập hợp các giá trị và một tập hợp các phép toán trên các giá trị đó. Ví dụ kiểu Boolean là một tập hợp có 2 giá trị TRUE, FALSE và các phép toán trên nó như OR, AND, NOT …. Kiểu Integer là tập hợp các số nguyên có giá trị từ -32768 đến 32767 cùng các phép toán +, -, *, /, div, mod …

Kiểu dữ liệu có hai loại là kiểu dữ liệu sơ cấp và kiểu dữ liệu có cấu trúc. Kiểu dữ liệu sơ cấp là kiểu dữ liệu mà giá trị dữ liệu của nó là đơn nhất. Ví dụ: kiểu Boolean, Integer…. Kiểu dữ liệu có cấu trúc là kiểu dữ liệu mà giá trị dữ liệu của nó là sự kết hợp của của nhiều giá trị khác nhau, cùng kiểu hoặc khác kiểu. Ví dụ: ARRAY là một kiểu dữ liệu có cấu trúc, chứa nhiều giá trị cùng kiểu; RECORD là một kiểu dữ liệu có cấu trúc chứa nhiều giá trị có kiểu khác nhau.

Các ngôn ngữ lập trình khác nhau cung cấp các kiểu dữ liệu cơ bản khác nhau. Hơn nữa, từ các kiểu dữ liệu cơ bản này có thể tạo ra các kiểu dữ liệu có cấu trúc phức hợp

hơn. Các kiểu dữ liệu có cấu trúc cơ bản (cung cấp bởi NNLT) và các cấu trúc phức hợp (được tạo ra từ các kiểu dữ liệu cơ bản) gọi chung là các **cấu trúc dữ liệu**.

Một kiểu dữ liệu trừu tượng là một mô hình toán học cùng với một tập hợp các phép toán trên nó. Có thể nói kiểu dữ liệu trừu tượng là một kiểu dữ liệu do chúng ta định nghĩa ở mức khái niệm (conceptual), nó chưa được cài đặt cụ thể bằng một ngôn ngữ lập trình.

Khi cài đặt một kiểu dữ liệu trừu tượng trong một ngôn ngữ lập trình cụ thể, chúng ta phải thực hiện hai công việc:

1. Biểu diễn kiểu dữ liệu trừu tượng (ở mức khái niệm) bằng một cấu trúc dữ liệu hoặc một kiểu dữ liệu trừu tượng khác đã được cài đặt.

2. Viết các chương trình con thực hiện các phép toán trên kiểu dữ liệu trừu tượng.

Sau khi đã cài đặt, kiểu dữ liệu trừu tượng có thể được xem như một kiểu "nguyên sơ" của NNLT và các phép toán trên kiểu dữ liệu trừu tượng có thể xem như là các phép toán "nguyên sơ" – tức là đã được cài đặt và có sẵn để dùng trong khi giải các bài toán khác. Ví dụ, sau khi cài đặt LIST (mô hình cùng với các phép toán trên LIST) thì LIST có thể dùng cài đặt GRAPH và SET trong thủ tục GREEDY ở trên.

TỔNG KẾT CHƯƠNG

Trong chương này, chúng ta cần phải nắm vững các vấn đề sau:

1. Các bước phân tích và lập trình để quyết một bài toán thực tế.
2. Hiểu rõ khái niệm về kiểu dữ liệu, kiểu dữ liệu trừu tượng và cấu trúc dữ liệu.

CHƯƠNG II: CÁC KIỂU DỮ LIỆU TRỪU TƯỢNG CƠ BẢN (BASIC ABSTRACT DATA TYPES)

I. KIỂU DỮ LIỆU TRỪU TƯỢNG DANH SÁCH (LIST)

1. Khái niệm danh sách

Mô hình toán học của danh sách là một tập hợp hữu hạn các phần tử có cùng một kiểu, mà tổng quát ta gọi là kiểu phần tử (Elementtype). Ta biểu diễn danh sách như là một chuỗi các phần tử của nó: $a_1, a_2, . . ., a_n$,với n ≥ 0. Nếu n=0 ta nói *danh sách rỗng* (empty list). Nếu n > 0 ta gọi a_1 là phần tử đầu tiên và a_n là phần tử cuối cùng của danh sách. Số phần tử của danh sách ta gọi là *độ dài* của danh sách.

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

Ví dụ: Tập hợp họ tên các sinh viên của lớp TINHOC 28 được liệt kê trên giấy như sau:

1. Nguyễn Trung Cang
2. Nguyễn Ngọc Chương
3. Lê Thị Lệ Sương
4. Trịnh Vũ Thành
5. Nguyễn Phú Vĩnh

là một danh sách. Danh sách này gồm có 5 phần tử, mỗi phần tử có một vị trí trong danh sách theo thứ tự xuất hiện của nó.

2. Các phép toán trên danh sách

Để thiết lập kiểu dữ liệu trừu tượng danh sách (hay ngắn gọn là danh sách) ta phải định nghĩa các phép toán trên danh sách. Và như chúng ta sẽ thấy trong toàn bộ giáo trình, không có một tập hợp các phép toán nào thích hợp cho mọi ứng dụng (application). Vì vậy ở đây ta sẽ định nghĩa một số phép toán cơ bản nhất trên danh sách. Để thuận tiện cho việc định nghĩa ta giả sử rằng danh sách gồm các phần tử có kiểu là kiểu phần tử (ElementType); vị trí của các phần tử trong danh sách có kiểu là kiểu vị trí; và vị trí sau phần tử cuối cùng trong danh sách L là ENDLIST(L). Cần nhấn mạnh rằng khái niệm vị trí (position) là do ta định nghĩa, nó không phải là giá trị của các phần tử trong danh sách. **Vị trí có thể là đồng nhất với vị trí lưu trữ phần tử hoặc không**.

Các phép toán được định nghĩa trên danh sách là:

INSERT_LIST(x,p,L): xen phần tử x (kiểu ElementType) tại vị trí p (kiểu position) trong danh sách L. Tức là nếu danh sách là $a_1, a_2, . , a_{p-1}, a_p ,. . , a_n$ thì sau khi

xen ta có kết quả $a_1, a_2, \ldots, a_{p-1}, x, a_p, \ldots, a_n$. Nếu vị trí p không tồn tại trong danh sách thì phép toán không được thực hiện.

LOCATE(x,L) thực hiện việc định vị phần tử có nội dung x đầu tiên trong danh sách L. Locate trả kết quả là vị trí (kiểu position) của phần tử x trong danh sách. Nếu x không có trong danh sách thì vị trí sau phần tử cuối cùng của danh sách được trả về, tức là ENDLIST(L).

RETRIEVE(p,L) lấy giá trị của phần tử ở vị trí p (kiểu position) của danh sách L; nếu vị trí p không có trong danh sách thì kết quả không xác định (có thể thông báo lỗi hoặc trả về NULL).

DELETE_LIST(p,L) chương trình con thực hiện việc xoá phần tử ở vị trí p (kiểu position) của danh sách. Nếu vị trí p không có trong danh sách thì phép toán không được thực hiện và danh sách L sẽ không thay đổi

NEXT(p,L) cho kết quả là vị trí của phần tử (kiểu position) đi sau phần tử p; nếu p là phần tử cuối cùng trong danh sách L thì NEXT(p,L) cho kết quả là ENDLIST(L). Next không xác định nếu p không phải là vị trí của một phần tử trong danh sách.

PREVIOUS(p,L) cho kết quả là vị trí của phần tử đứng trước phần tử p trong danh sách. Nếu p là phần tử đầu tiên trong danh sách thì Previous(p,L) không xác định. Previous cũng không xác định trong trường hợp p không phải là vị trí của phần tử nào trong danh sách.

FIRST(L) cho kết quả là vị trí của phần tử đầu tiên trong danh sách. Nếu danh sách rỗng thì ENDLIST(L) được trả về.

EMPTY_LIST(L) cho kết quả TRUE nếu danh sách có rỗng, ngược lại nó cho giá trị FALSE.

MAKENULL_LIST(L) khởi tạo một danh sách L rỗng.

Trong thiết kế các giải thuật sau này chúng ta dùng các phép toán trừu tượng đã được định nghĩa ở đây như là các phép toán nguyên thủy.

? Muốn thêm phần tử vào đầu danh sách cần gọi các phép toán nào?
Muốn thêm phần tử vào cuối danh sách cần gọi các phép toán nào?

Ví dụ: Dùng các phép toán trừu tượng trên danh sách, viết một chương trình con nhận một tham số là danh sách rồi sắp xếp danh sách theo thứ tự tăng dần. Giả sử các phần tử trong danh sách thuộc kiểu có thứ tự. ta cũng giả sử hàm WAP(p,q) thực hiện việc đổi chỗ hai phần tử tại vị trí p và q trong danh sách, chương trình con sắp xếp được viết như sau:

```
void SORT(LIST& L){
        Position p= FIRST(L);
        //vị trí phần tử đầu tiên trong danh sách
        while (p!=ENDLIST(L)){
                Position  q=NEXT(p,L);
                //vị trí phần tử đứng ngay sau phần tử p
                while (q!=ENDLIST(L)){
                        if (RETRIEVE(p,L) > RETRIEVE(q,L))
                                swap(p,q); // hoán chuyển nội dung phần tử
                        q=NEXT(q,L);
                }
                p=NEXT(p,L);
        }
}
```

Tuy nhiên, cần phải nhấn mạnh rằng, đây là các phép toán trừu tượng do chúng ta định nghĩa, nó chưa được cài đặt trong các ngôn ngữ lập trình. Do đó để cài đặt giải thuật thành một chương trình chạy được, ta phải cài đặt các phép toán thành các chương trình con trong chương trình. Hơn nữa, trong khi cài đặt cụ thể, một số tham số hình thức trong các phép toán trừu tượng không đóng vai trò gì trong chương trình con cài đặt chúng, do vậy ta có thể bỏ qua nó trong danh sách tham số của chương trình con. Ví dụ: phép toán trừu tượng INSERT_LIST(x,p,L) có 3 tham số hình thức: phần tử muốn thêm x, vị trí thêm vào p và danh sách được thêm vào L. Nhưng khi cài đặt danh sách bằng con trỏ (danh sách liên kết đơn), tham số L là không cần thiết vì với cấu trúc này chỉ có con trỏ tại vị trí p phải thay đổi để nối kết với ô chứa phần tử mới. Trong bài giảng này, ta vẫn giữ đúng những tham số trong cách cài đặt để làm cho chương trình đồng nhất và trong suốt đối với các phương pháp cài đặt của cùng một kiểu dữ liệu trừu tượng. Tuy nhiên trong thực hành đôi khi ta gặp một số lời cảnh báo dạng "*biến này được không dùng*".

3. Cài đặt danh sách

a. Cài đặt danh sách bằng mảng (danh sách đặc)

Ta có thể cài đặt danh sách bằng mảng như sau: *dùng một mảng để lưu giữ liên tiếp các phần tử của danh sách từ vị trí đầu tiên của mảng*. Với cách cài đặt này, dĩ nhiên, ta phải ước lượng số phần tử của danh sách để khai báo số phần tử của mảng cho thích hợp. Dễ thấy rằng số phần tử của mảng phải được khai báo không ít hơn số phần tử của danh sách. Nói chung là mảng còn thừa một số chỗ trống. Mặt khác ta phải lưu giữ độ dài hiện tại của danh sách, độ dài này cho biết danh sách có bao nhiêu phần tử và cho biết phần nào của mảng còn trống như trong hình II.1. *Ta định nghĩa vị trí của một phần tử trong danh sách là số thứ tự của phần tử trong danh sách.* Như vậy phần tử thứ i sẽ được lưu trong mảng tại chỉ số i-1(**vì phần tử đầu tiên trong mảng có chỉ số là 0**).

Chỉ số	0	1	…	Last-1	…	Maxlength-1
Nội dung phần tử	Phần tử thứ 1	Phần tử thứ 2	…	Phần tử cuối cùng trong danh sách	…	

Hình II.1: Cài đặt danh sách bằng mảng

Với hình ảnh minh họa trên, ta cần các khai báo cần thiết là

```
#define MaxLength ...
//Số nguyên thích hợp để chỉ độ dài của mảng
typedef ... ElementType;//kiểu của phần tử trong danh sách
typedef int Position; //kiểu vị trí cuả các phần tử
typedef struct {
    ElementType Elements[MaxLength];    //mảng chứa các phần tử của danh sách
    Position Last;    //giữ độ dài danh sách
} List;
```

Trên đây là sự biểu diễn kiểu dữ liệu trừu trượng danh sách bằng cấu trúc dữ liệu mảng. Phần tiếp theo là cài đặt các phép toán cơ bản trên danh sách.

Khởi tạo danh sách rỗng

Danh sách rỗng là một danh sách không chứa bất kỳ một phần tử nào (hay độ dài danh sách bằng 0). Theo cách khai báo trên, trường Last chỉ vị trí của phần tử cuối cùng trong danh sách và đó cũng độ dài hiện tại của danh sách, vì vậy để khởi tạo danh sách rỗng ta chỉ việc gán giá trị trường Last này bằng 0.

```
void MAKENULL_LIST(List& L){
    L.Last=0;
}
```

?

1. Hãy trình bày cách gọi thực thi chương trình con tạo danh sách rỗng trên?
2. Hãy giải thích cách khai báo tham số hình thức trong chương trình con và cách truyền tham số khi gọi chương trình con đó?

Kiểm tra danh sách rỗng

Danh sách rỗng là một danh sách mà độ dài của nó bằng 0.

```
int EMPTY_LIST(List L){
    return L.Last==0;
}
```

Xen một phần tử vào danh sách

Khi xen phần tử có nội dung x vào tại vị trí p của danh sách L thì sẽ xuất hiện các khả năng sau:

- Mảng đầy: mọi phần tử của mảng đều chứa phần tử của danh sách, tức là phần tử cuối cùng của danh sách nằm ở vị trí cuối cùng trong mảng. Nói cách khác, độ dài của danh sách bằng chỉ số tối đa của mảng; Khi đó không còn chỗ cho phần tử mới, vì vậy việc xen là không thể thực hiện được, chương trình con gặp lỗi.
- Ngược lại ta tiếp tục xét:
  - Nếu p không hợp lệ (p>last+1 hoặc p<1) thì chương trình con gặp lỗi;
  - Nếu vị trí p hợp lệ thì ta tiến hành xen theo các bước sau:
    - Dời các phần tử từ vị trí p đến cuối danh sách ra sau 1 vị trí.
    - Độ dài danh sách tăng 1.
    - Đưa phần tử mới vào vị trí p

Chương trình con xen phần tử x vào vị trí p của danh sách L có thể viết như sau:

```
void INSERT_LIST(ElementType X, Position P, List& L){
    if (L.Last==MaxLength)
            printf("Danh sach day");
    else    if ((P<1) || (P>L.Last+1))
                    printf("Vi tri khong hop le");
             else{
                     Position Q;
                    /*  Dời các phần tử từ vị trí p (chỉ số trong mảng là
                    (p-1) đến cuối danh sách sang phải 1 vị trí */
                     for (Q=L.Last;Q>P-1;Q--)
                             L.Elements[Q]=L.Elements[Q-1];
                    //Đưa x vào vị trí p
                     L.Elements[P-1]=X;
                    //Tăng độ dài danh sách lên 1
                     L.Last++;
             }
}
```

Xóa phần tử ra khỏi danh sách

Xoá một phần tử ở vị trí p ra khỏi danh sách L ta làm công việc ngược lại với xen.

- Trước tiên ta kiểm tra vị trí phần tử cần xóa xem có hợp lệ hay không. Nếu p>L.last hoặc p<1 thì đây không phải là vị trí của phần tử trong danh sách.

- Ngược lại, vị trí đã hợp lệ thì ta phải dời các phần tử từ vị trí p+1 đến cuối danh sách ra trước một vị trí và độ dài danh sách giảm đi 1 phần tử (do đã xóa bớt 1 phần tử).

```
void DELETE_LIST(Position P,List& L){
    if ((P<1) || (P>L.Last))
        printf("Vi tri khong hop le");
    else {
        Position Q;
        /*Dời các phần tử từ vị trí p+1 (chỉ số trong mảng
        là p) đến cuối danh sách sang trái 1 vị trí*/
        for(Q=P-1;Q<L.Last-1;Q++)
            L.Elements[Q]=L.Elements[Q+1];
        L.Last--;
    }
}
```

Định vị một phần tử trong danh sách

Để định vị phần tử đầu tiên có nội dung x trong danh sách L, ta tiến hành dò tìm từ đầu danh sách. Nếu tìm thấy x thì vị trí của phần tử tìm thấy được trả về, nếu không tìm thấy thì hàm trả về vị trí sau vị trí của phần tử cuối cùng trong danh sách, tức là ENDLIST(L) (với cài đặt này thì ENDLIST(L)= L.Last+1). Trong trường hợp có nhiều phần tử cùng giá trị x trong danh sách thì vị trí của phần tử được tìm thấy đầu tiên được trả về.

```
Position LOCATE(ElementType X, List L){
    Position P = 1; //vị trí phần tử đầu tiên
    /*trong khi chưa tìm thấy và chưa kết thúc
    danh sách thì xét phần tử kế tiếp*/
    while (P != L.Last + 1)
        if (L.Elements[P-1] == X)
            return P;
        else
            P ++ ;
}
```

Các phép toán khác dễ dàng cài đặt:

- FIRST(L) trả về 1
- RETRIEVE(P,L) trả về L.Elements[P-1]
- ENDLIST(L) trả về L.Last+1
- NEXT(P,L) trả về P+1

? Hãy giải thích tại sao nội dung phần tử tại vị trí P trên danh sách L là L.Elements[P-1]?

Các phép toán khác cũng dễ dàng cài đặt nên xem như bài tập dành cho bạn đọc.

Ví dụ : Vận dụng các phép toán trên danh sách đặc để viết chương trình nhập vào một danh sách các số nguyên và hiển thị danh sách vừa nhập ra màn hình. Thêm phần tử có nội dung x vào danh sách tại ví trí p (trong đó x và p được nhập từ bàn phím). Xóa phần tử đầu tiên có nội dung x (nhập từ bàn phím) ra khỏi danh sách.

Hướng giải quyết :

Giả sử ta đã cài đặt đầy đủ các phép toán cơ bản trên danh sách. Để thực hiện yêu cầu như trên, ta cần thiết kế thêm các phép toán sau:

- Nhập danh sách từ bàn phím (READ_LIST(L)) (Phép toán này chưa có trong kiểu danh sách)

- Hiển thị danh sách ra màn hình (in danh sách) (PRINT_LIST(L)) (Phép toán này chưa có trong kiểu danh sách).

```
void READ_LIST(List& L){
        int n,x;
        printf("Nhap so phan tu cua danh sach: ");
        scanf("%d",&n);
        for (int i=1; i<=n; i++){
                printf("nhap phan tu thu %d: ",i);
                scanf("%d",&x);
                INSERT_LIST(x, i, L);
        }
}

void PRINT_LIST(List& L){
        for(int i=1; i<=L.Last; i++)
                printf("%d ",Retrieve(i,L));
        printf("\n");
}
```

Thực ra thì chúng ta chỉ cần sử dụng các phép toán MAKENULL_LIST, INSERT_LIST, DELETE_LIST, LOCATE và READ_LIST, PRINT_LIST vừa nói trên là có thể giải quyết được bài toán. Để đáp ứng yêu cầu đặt ra, ta viết chương trình chính để nối kết những chương trình con lại với nhau như sau:

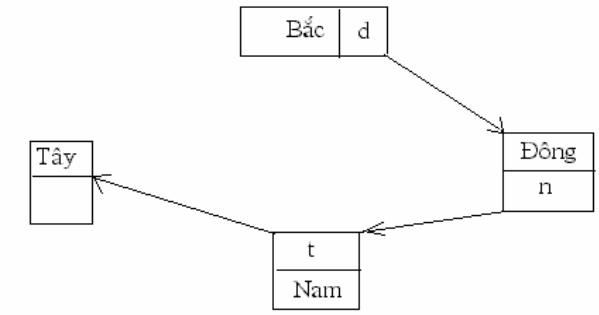
```
int main(){
        List L;
        ElementType X;
        Position P;
        MAKENULL_LIST(L); //Khởi tạo danh sách rỗng
        READ_LIST(L);
        printf("Danh sach vua nhap: ");
        PRINT_LIST(L); // In danh sach len man hinh
        printf("Phan tu can them: ");scanf("%d",&X);
        printf("Vi tri can them: ");scanf("%d",&P);
        INSERT_LIST(X,P,L);
```

```
        printf("Danh sach sau khi them phan tu la: ");
        PRINT_LIST(L);
        printf("Noi dung phan tu can xoa: ");scanf("%d",&X);
        P=LOCATE(X,L);
        DELETE_LIST(P,L);
        printf("Danh sach sau khi xoa %d la: ",X);
        PRINT_LIST(L);
        return 0;
}
```

b. Cài đặt danh sách bằng con trỏ (danh sách liên kết)

Cách khác để cài đặt danh sách là dùng con trỏ để liên kết các ô chứa các phần tử. Trong cách cài đặt này các phần tử của danh sách được lưu trữ trong các ô, mỗi ô có thể chỉ đến ô chứa phần tử kế tiếp trong danh sách. Bạn đọc có thể hình dung cơ chế này qua ví dụ sau:

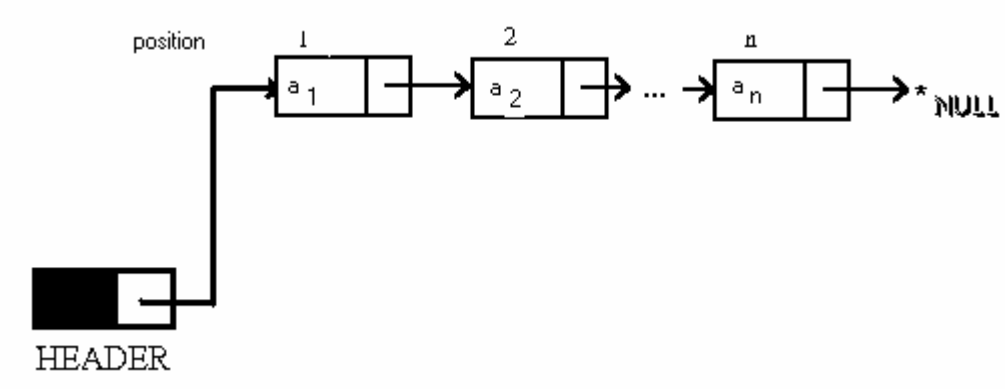
Giả sử 1 lớp có 4 bạn: Đông, Tây, Nam, Bắc có địa chỉ lần lượt là d,t,n,b. Giả sử: Đông có địa chỉ của Nam, Tây không có địa chỉ của bạn nào, Bắc giữ địa chỉ của Đông, Nam có địa chỉ của Tây (xem hình II.2).



Hình II.2

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

Trong cài đặt, để một ô có thể *chỉ đến* ô khác ta cài đặt mỗi ô là một mẩu tin (record hay s**truct**) có hai trường: trường Element giữ giá trị của các phần tử trong danh sách; trường next là một *con trỏ* giữ địa chỉ của ô kế tiếp. Trường next của phần tử cuối trong danh sách chỉ đến một giá trị đặc biệt là **NULL**. Cấu trúc như vậy gọi là danh sách cài đặt bằng con trỏ hay *danh sách liên kết đơn* hay ngắn gọn là *danh sách liên kết*.



Hình II.3 Danh sách liên kết đơn

Để quản lý danh sách ta chỉ cần một biến giữ địa chỉ ô chứa phần tử đầu tiên của danh sách, tức là một con trỏ trỏ đến phần tử đầu tiên trong danh sách. Biến này gọi là *chỉ điểm đầu danh sách (Header)* . Để đơn giản hóa vấn đề, trong chi tiết cài đặt, Header là một biến cùng kiểu với các ô chứa các phần tử của danh sách và nó cũng được cấp phát ô nhớ y như một ô chứa phần tử của danh sách (hình II.3). Tuy nhiên Header là một ô đặc biệt nên nó không chứa phần tử nào của danh sách, trường dữ liệu của ô này là rỗng, chỉ có trường con trỏ Next trỏ tới ô chứa phần tử đầu tiên thật sự của danh sách. Nếu danh sách rỗng thì Header->Next trỏ tới **NULL**. Việc cấp phát ô nhớ cho Header như là một ô chứa dữ liệu bình thường nhằm tăng tính đơn giản của các giải thuật thêm, xoá các phần tử trong danh sách.

Ở đây ta cần phân biệt rõ giá trị của một phần tử và địa chỉ của nó trong cấu trúc trên. Ví dụ giá trị của phần tử đầu tiên của danh sách trong hình II.3 là a_1, trong khi địa chỉ của ô chứa nó nằm ở trường next của ô Header. Giá trị và địa chỉ của các phần tử của danh sách trong hình II.3 như sau:

Phần tử thứ	Giá trị	Địa chỉ chứa trong trường next của ô
1	a_1	HEADER
2	a_2	1
...	...	...
N	a_n	(n-1)
Sau phần tử cuối cùng	Không xác định	n và n->Next có giá trị là **NULL**

Như đã thấy trong bảng trên, để biết được địa chỉ của phần tử thứ i ta phải truy xuất vào trường next của ô chứa phần tử thứ (i-1). Khi thêm hoặc xoá một phần tử thứ p trong danh sách liên kết, ta phải cập nhật lại con trỏ trỏ tới ô chứa phần tử thứ p để nó trỏ tới phần tử thứ p mới, tức là phải cập nhật lại trường Next của ô chứa phần tử thứ (p-1). Do đó, ta định nghĩa địa chỉ của ô chứa phần tử thứ (p-1) là **vị trí** (position) của

phần tử thứ p. Có thể nói nôm na rằng vị trí của phần tử a_i là địa chỉ của ô đứng ngay phía trước ô chứa a_i. Hay chính xác hơn, ta nói, vị trí của phần tử thứ i là con trỏ trỏ tới ô có trường next trỏ tới ô chứa phần tử a_i. Như vậy vị trí của phần tử thứ 1 là con trỏ trỏ đến Header, vị trí phần tử thứ 2 là con trỏ trỏ ô chứa phần tử a_1, vị trí của phần tử thứ 3 là con trỏ trỏ ô a_2, ..., vị trí phần tử thứ n là con trỏ trỏ ô chứa a_{n-1}. Vị trí sau phần tử cuối trong danh sách, tức là ENDLIST, chính là con trỏ trỏ ô chứa phần tử a_n (xem hình II.3).

Theo định nghĩa này ta có, nếu P là vị trí của phần tử thứ p trong danh sách thì giá trị của phần tử thứ p này nằm trong trường Element của ô được trỏ bởi P->Next. Nói cách khác **P->Next->Element chứa nội dung của phần tử ở vị trí p trong danh sách**.

Các khai báo cần thiết là

```
typedef ... ElementType; //kiểu của phần tử trong danh sách
  typedef struct Node{
            ElementType Element;//Chứa nội dung của phần tử
            Node* Next; /*con trỏ chỉ đến phần tử kế tiếp trong danh sách*/
};
typedef Node* Position; // Kiểu vị trí
typedef Position List;
```

? Trong khai báo trên, tại sao phải đặt tên kiểu Node trước khi đưa ra các trường trong kiểu đó?
Cách khai báo sau còn đúng không?

```
typedef struct
  { ElementType Element;
    Node* Next;
  } Node;
```

Tạo danh sách rỗng

Như đã nói ở phần trên, ta dùng Header như là một biến con trỏ có kiểu giống như kiểu của một ô chứa một phần tử của danh sách. Tuy nhiên trường Element của Header không bao giờ được dùng, chỉ có trường Next dùng để trỏ tới ô chứa phần tử đầu tiên của danh sách. Vậy nếu như danh sách rỗng thì ô Header vẫn phải tồn tại và ô này có trường next chỉ đến **NULL** (do không có một phần tử nào). Vì vậy khi khởi tạo danh sách rỗng, ta phải cấp phát ô nhớ cho HEADER và cho con trỏ trong trường next của nó trỏ tới **NULL**.

```
void MAKENULL_LIST(List& Header){
      (Header)=(Node*)malloc(sizeof(Node));
      Header->Next= NULL;
}
```

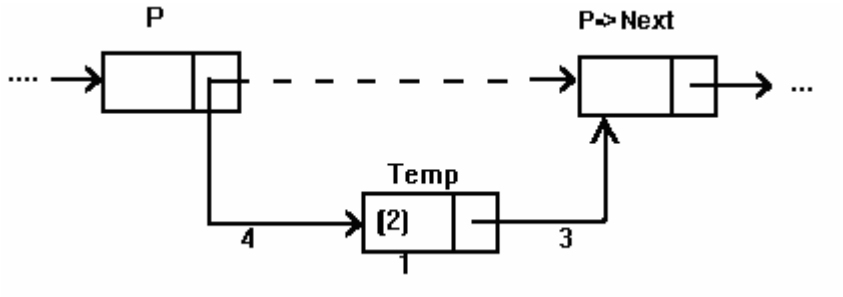
Kiểm tra một danh sách rỗng

Danh sách rỗng nếu như trường Next trong ô Header trỏ tới **NULL**.

```
int EMPTY_LIST(List L){
      return (L->Next==NULL);
}
```

Xen một phần tử vào danh sách

Xen một phần tử có giá trị x vào danh sách L tại vị trí p ta phải cấp phát một ô mới để lưu trữ phần tử mới này và nối kết lại các con trỏ để đưa ô mới này vào vị trí p. Sơ đồ nối kết và thứ tự các thao tác được cho trong hình II.4.

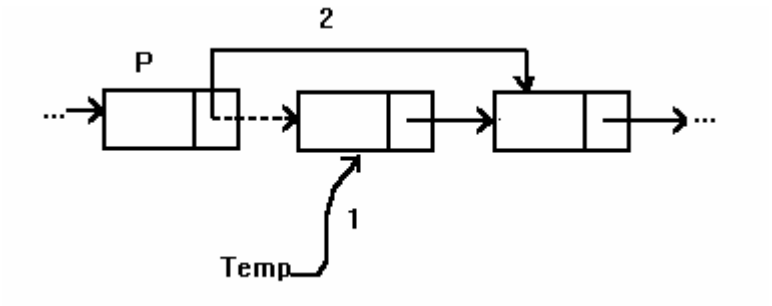


Hình II.4: Thêm một phần tử vào danh sách tại vị trí p

```
void INSERT_LIST(ElementType X, Position P, List& L){
      Position T=(Node*)malloc(sizeof(Node));
      T->Element=X;
      T->Next=P->Next;
      P->Next=T;
}
```

? Tham số L (danh sách) trong chương trình con trên có bỏ được không? Tại sao?

Xóa phần tử ra khỏi danh sách



Hình II.5: Xoá phần tử tại vị trí p

Tương tự như khi xen một phần tử vào danh sách liên kết, muốn xóa một phần tử khỏi danh sách ta cần biết vị trí p của phần tử muốn xóa trong danh sách L. Nối kết lại các con trỏ bằng cách cho p trỏ tới phần tử đứng sau phần tử thứ p. Trong các ngôn ngữ lập trình không có cơ chế thu hồi vùng nhớ tự động như ngôn ngữ Pascal, C thì ta phải thu hồi vùng nhớ của ô bị xóa một các tường minh trong giải thuật. Tuy nhiên vì tính đơn giản của giải thuật cho nên đôi khi chúng ta không đề cập đến việc thu hồi vùng nhớ cho các ô bị xoá. Chi tiết và trình tự các thao tác để xoá một phần tử trong danh sách liên kết như trong hình II.5. Chương trình con có thể được cài đặt như sau:

```
void DELETE_LIST(Position P, List& L){
    Position T;
    if (P->Next!=NULL){
        T=P->Next; /*giữ ô chứa phần tử bị xoá để thu hồi vùng nhớ*/
        P->Next=T->Next; /*nối kết con trỏ trỏ tới phần tử thứ p+1*/
        free(T); //thu hồi vùng nhớ
    }
}
```

Định vị một phần tử trong danh sách liên kết

Để định vị phần tử x trong danh sách L ta tiến hành tìm từ đầu danh sách (ô header) nếu tìm thấy thì vị trí của phần tử đầu tiên được tìm thấy sẽ được trả về nếu không thì ENDLIST(L) được trả về. Nếu x có trong danh sách, hàm Locate trả về vị trí p thì x = p->Next->Element.

```
Position LOCATE(ElementType X, List L){
    Position P = L;
    while (P->Next != NULL)
        if (P->Next->Element == X)  break; //return P;
        else P = P->Next;
    return P;
}
```

Thực chất, khi gọi hàm LOCATE ở trên ta có thể truyền giá trị cho L là bất kỳ giá trị nào. Nếu L là Header thì chương trình con sẽ tìm x từ đầu danh sách. Nếu L là một vị trí p bất kỳ trong danh sách thì hàm LOCATE sẽ tiến hành định vị phần tử x từ vị trí p.

Lấy giá trị của phần tử

Giá trị (nội dung) của phần tử đang lưu trữ tại vị trí p trong danh sách L là p->Next->Element Do đó, hàm sẽ trả về giá trị p->Next->Element nếu phần tử có tồn tại, ngược lại phần tử không tồn tại (p->Next=**NULL**) thì hàm không xác định

```
ElementType RETRIEVE(Position P, List L){
    if (P->Next!=NULL)
        return P->Next->Element;
}
```

? Hãy thiết kế hàm Locate bằng cách sử dụng các phép toán trừu tượng cơ bản trên danh sách?

c. So sánh hai phương pháp cài đặt

Không thể kết luận phương pháp cài đặt nào hiệu quả hơn, mà nó hoàn toàn tuỳ thuộc vào từng ứng dụng hay tuỳ thuộc vào các phép toán trên danh sách. Tuy nhiên ta có thể tổng kết một số ưu nhược điểm của từng phương pháp làm cơ sở để lựa chọn phương pháp cài đặt thích hợp cho từng ứng dụng:

- Cài đặt bằng mảng đòi hỏi phải xác định số phần tử của mảng, do đó nếu không thể ước lượng được số phần tử trong danh sách thì khó áp dụng cách cài đặt này một cách hiệu quả vì nếu khai báo thiếu chỗ thì mảng thường xuyên bị đầy, không thể làm việc được còn nếu khai báo quá thừa thì lãng phí bộ nhớ.

- Cài đặt bằng con trỏ thích hợp cho sự biến động của danh sách, danh sách có thể rỗng hoặc lớn tuỳ ý chỉ phụ thuộc vào bộ nhớ tối đa của máy. Tuy nhiên ta phải tốn thêm vùng nhớ cho các con trỏ (trường Next).

- Cài đặt bằng mảng thì thời gian xen hoặc xoá một phần tử tỉ lệ với số phần tử đi sau vị trí xen/ xóa. Trong khi cài đặt bằng con trỏ các phép toán này mất chỉ một hằng thời gian.

- Phép truy nhập vào một phần tử trong danh sách, chẳng hạn như PREVIOUS, chỉ tốn một hằng thời gian đối với cài đặt bằng mảng, trong khi đối với danh sách cài đặt bằng con trỏ ta phải tìm từ đầu danh sách cho đến vị trí trước vị trí của phần tử hiện hành. Nói chung *danh sách liên kết thích hợp với danh sách có nhiều biến động*, tức là ta thường xuyên thêm, xoá các phần tử.

? Cho biết ưu khuyết điểm của danh sách đặc và danh sách liên kết?

d. Cài đặt bằng con nháy

Một số ngôn ngữ lập trình không có cung cấp kiểu con trỏ. Trong trường hợp này ta có thể "giả" con trỏ để cài đặt danh sách liên kết. Ý tưởng chính là: dùng mảng để chứa các phần tử của danh sách, các "con trỏ" sẽ là các biến số nguyên (`int`) để giữ chỉ số của phần tử kế tiếp trong mảng. Để phân biệt giữa "con trỏ thật" và "con trỏ giả" ta gọi các con trỏ giả này là *con nháy* (cursor). Như vậy, để cài đặt danh sách bằng con nháy ta cần một mảng mà mỗi phần tử xem như là một ô gồm có hai trường: trường Element như thông lệ giữ giá trị của phần tử trong danh sách (có kiểu Elementtype)

trường Next là con nháy để chỉ tới vị trí trong mảng của phần tử kế tiếp. Chẳng hạn hình II.6 biểu diễn cho mảng SPACE đang chứa hai danh sách L_1, L_2. Để quản lí các danh sách ta cũng cần một con nháy chỉ đến phần tử đầu của mỗi danh sách (giống như Header trong danh sách liên kết). Phần tử cuối cùng của danh sách ta cho chỉ tới giá trị đặc biệt **`Null`**, có thể xem **`Null`** = -1 với một giả thiết là mảng SPACE không có vị trí nào có chỉ số -1.

Trong hình II.6, danh sách L_1 gồm 3 phần tử : f, o ,r. Chỉ điểm đầu của L_1 là con nháy có giá trị 5, tức là nó trỏ vào ô lưu giữ phần tử đầu tiên của L_1, trường Next của ô này có giá trị 1 là ô lưu trữ phần tử kế tiếp (tức là o). Trường Next tại ô chứa o là 4 là ô lưu trữ phần tử kế tiếp trong danh sách (tức là r). Cuối cùng trường Next của ô này chứa **`Null`** nghĩa là danh sách không còn phần tử kế tiếp.

Phân tích tương tự ta có L_2 gồm 4 phần tử : w, i, n, d.

Chỉ điểm của danh sách thứ nhất L_1 →

Chỉ điểm của danh sách thứ hai L_2 →

Chỉ số	Elements	Next
0	D	**Null**
1	O	4
2		
3	N	0
4	R	**Null**
5	F	1
6	I	3
7	W	6
8		
9		

Mảng SPACE

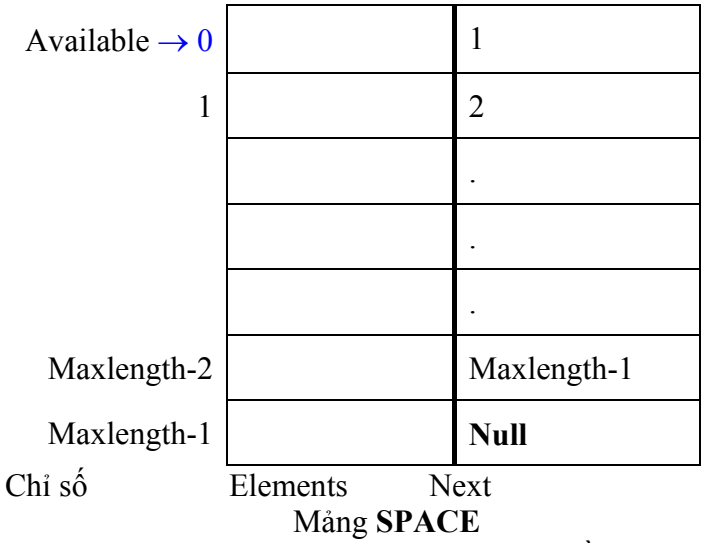
Hình II.6 Mảng đang chứa hai danh sách L_1 và L_2

Khi xen một phần tử vào danh sách ta lấy một ô trống trong mảng để chứa phần tử mới này và nối kết lại các con nháy. Ngược lại, khi xoá một phần tử khỏi danh sách ta nối kết lại các con nháy để loại phần tử này khỏi danh sách, điều này kéo theo số ô trống trong mảng tăng lên 1. Vấn đề là làm thế nào để quản lí các ô trống này để biết ô nào còn trống ô nào đã dùng? một giải pháp là *liên kết tất cả các ô trống vào một danh sách đặc biệt gọi là Available.* Khi xen một phần tử vào danh sách ta lấy ô trống đầu Available để chứa phần tử mới này. Khi xoá một phần tử từ danh sách ta cho ô bị xoá nối vào đầu Available. Tất nhiên khi mới khởi đầu việc xây dựng cấu trúc thì mảng

chưa chứa phần tử nào của bất kỳ một danh sách nào. Lúc này tất cả các ô của mảng đều là ô trống, và như vậy, tất cả các ô đều được liên kết vào trong Available. Việc khởi tạo Available ban đầu có thể thực hiện bằng cách cho phần tử thứ i của mảng trỏ tới phần tử i+1.

Các khai báo cần thiết cho danh sách

```
#define MaxLength ... //Chiều dài mảng
#define Null -1 //Gia tri Null
typedef ... ElementType; /*kiểu của các phần tử trong danh sách*/
typedef struct{
        ElementType Elements; /*trường chứa phần tử trong danh sách*/
        int Next; //con nháy trỏ đến phần tử kế tiếp
} Node;
Node Space[MaxLength]; //Mang toan cuc
int Available; //chỉ điểm đầu danh sách ô trống
```



Hình II.7: Khởi tạo Available ban đầu

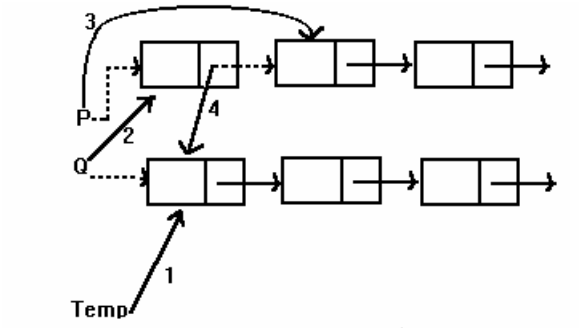
Khởi tạo cấu trúc – Thiết lập Available ban đầu

Ta cho phần tử thứ 0 của mảng trỏ đến phần tử thứ 1,..., phần tử cuối cùng trỏ **`Null`**. Chỉ điểm đầu của Available là 0 như trong hình II.7

```
void Initialize(){
        int i;
        for(i=0;i<MaxLength-1;i++)
                Space[i].Next=i+1;
        Space[MaxLength-1].Next=Null;
        Available=0;
}
```

Chuyển một ô từ danh sách này sang danh sách khác

Ta thấy thực chất của việc xen hay xoá một phần tử là thực hiện việc chuyển một ô từ danh sách này sang danh sách khác. Chẳng hạn muốn xen thêm một phần tử vào danh sách L_1 trong hình II.6 vào một vị trí p nào đó ta phải chuyển một ô từ Available (tức là một ô trống) vào L_1 tại vị trí p; muốn xoá một phần tử tại vị trí p nào đó trong danh sách L_2, chẳng hạn, ta chuyển ô chứa phần tử đó sang Available, thao tác này xem như là giải phóng bộ nhớ bị chiếm bởi phần tử này. Do đó tốt nhất ta viết một hàm thực hiện thao tác chuyển một ô từ danh sách này sang danh sách khác và hàm cho kết quả kiểu **int** tùy theo chuyển thành công hay thất bại (là 0 nếu chuyển không thành công, 1 nếu chuyển thành công). Hàm **Move** sau đây thực hiện chuyển ô được trỏ tới bởi con nháy P vào danh sách khác được trỏ bởi con nháy Q như trong hình II.8. Hình II.8 trình bày các thao tác cơ bản để chuyển một ô từ một danh sách sanh danh sách khác.



Hình II.8: Chuyển 1 ô từ danh sách này sang danh sách khác (các liên kết vẽ bằng nét đứt biểu diễn cho các liên kết cũ - trước khi giải thuật bắt đầu)

- Dùng con nháy temp để trỏ ô được trỏ bởi Q.
- Cho Q trỏ tới ô mới.
- Cập nhật lại con nháy P bằng cách cho nó trỏ tới ô kế tiếp.
- Nối con nháy trường Next của ô được chuyển (ô mà Q đang trỏ) trỏ vào ô mà temp đang trỏ.

```
int Move(int& p, int& q){
    if (p==Null)
        return 0; //Khong co o de chuyen
    else
    {
        int temp=q;
        q=p;
        p=Space[q].Next;
        Space[q].Next=temp;
        return 1; //Chuyen thanh cong
    }
}
```

Trong cách cài đặt này, **khái niệm vị trí** tương tự như khái niệm vị trí trong trường hợp cài đặt bằng con trỏ. Tức là, vị trí của phần tử thứ i trong danh sách là chỉ số của ô trong mảng chứa con nháy trỏ đến ô chứa phần tử thứ i.

Ví dụ xét danh sách L_1 trong hình II. 6, vị trí của phần tử thứ 2 trong danh sách (phần tử có giá trị o) là 5, không phải là 1; vị trí của phần tử thứ 3 (phần tử có giá trị r) là 1, không phải là 4. Vị trí của phần tử thứ 1 (phần tử có giá trị f) được định nghĩa là Null=-1, vì không có ô nào trong mảng chứa con nháy trỏ đến ô chứa phần tử f.

Xen một phần tử vào danh sách

Muốn xen một phần tử vào danh sách ta cần biết *vị trí xen*, gọi là *p*, *rồi ta chuyển ô đầu của Available vào vị trí này*. Chú ý rằng vị trí của phần tử đầu tiên trong danh sách được định nghĩa là -1 (Null), do đó nếu p=Null thì thực hiện việc thêm vào đầu danh sách.

```
void INSERT_LIST(ElementType X, int P, int& L){
    if (P==Null) {//Xen dau danh sach
        if (Move(Available,L))
            Space[L].Elements=X;
        else printf("Loi! Khong con bo nho trong");
    }
    else  //Chuyen mot o tu Available vao vi tri P
        if (Move(Available,Space[P].Next))
            // O nhan X la o tro boi Space[p].Next
            Space[Space[P].Next].Elements=X;
        else printf("Loi! Khong con bo nho trong");
}
```

Xoá một phần tử trong danh sách

Muốn *xoá một phần tử tại vị trí p* trong danh sách ta chỉ cần *chuyển ô chứa phần tử tại vị trí này vào đầu Available*. Tương tự như phép thêm vào, nếu p=Null thì xoá phần tử đầu danh sách.

```
void DELETE_LIST(int p, int& L){
    if (p== Null) //Neu la o dau tien
    {
        if (!Move(L,Available))
            printf("Loi trong khi xoa");
        // else Khong lam gi ca
    }
    else
        if (!Move(Space[p].Next,Available))
            printf("Loi trong khi xoa");
        //else Khong lam gi
}
```

II. NGĂN XẾP (STACK)

1. Định nghĩa ngăn xếp

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

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

2. Các phép toán trên ngăn xếp

- **MAKENULL_STACK(S):** tạo một ngăn xếp rỗng.
- **TOP(S)** xem như một hàm trả về phần tử tại đỉnh ngăn xếp. Nếu ngăn xếp rỗng thì hàm không xác định. Lưu ý rằng ở đây ta dùng từ "hàm" để ngụ ý là TOP(S) có trả kết quả ra. Nó có thể không đồng nhất với khái niệm hàm trong ngôn ngữ lập trình như C chẳng hạn, vì có thể kiểu phần tử không thể là kiểu kết quả ra của hàm trong C.
- **POP(S)** chương trình con xoá một phần tử tại đỉnh ngăn xếp.
- **PUSH(x,S)** chương trình con thêm một phần tử x vào đầu ngăn xếp.
- **EMPTY_STACK(S)** kiểm tra ngăn xếp rỗng. Hàm cho kết quả 1 (true) nếu ngăn xếp rỗng và 0 (false) trong trường hợp ngược lại.

Như đã nói từ trước, khi thiết kế giải thuật ta có thể dùng các phép toán trừu tượng như là các "nguyên thủy" mà không cần phải định nghĩa lại hay giải thích thêm. Tuy nhiên để giải thuật đó thành chương trình chạy được thì ta phải chọn một cấu trúc dữ liệu hợp lí để cài đặt các "nguyên thủy" này.

Ví dụ: Viết chương trình con Edit nhận một chuỗi kí tự từ bàn phím cho đến khi gặp kí tự @ thì kết thúc việc nhập và in kết quả theo thứ tự ngược lại.

```
void Edit(){
    Stack S;
    char c;
    MAKENULL_STACK(S);
    do{// Lưu từng ký tự vào ngăn xếp
        c=getche();
        PUSH(c,S);
    }while (c!='@');
        printf("\nChuoi theo thu tu nguoc lai\n");
    // In ngan xep
    while (!EMPTY_STACK(S)){
        printf("%c\n",TOP(S));
        POP(S);
    }
```

```
}
```

? Ta có thể truy xuất trực tiếp phần tử tại vị trí bất kỳ trong ngăn xếp được không?

3. Cài đặt ngăn xếp:

a. Cài đặt ngăn xếp bằng danh sách:

Do ngăn xếp là một danh sách đặc biệt nên ta có thể sử dụng kiểu dữ liệu trừu tượng danh sách để cài đặt ngăn xếp. Như vậy, ta có thể khai báo ngăn xếp như sau:

```
typedef List Stack;
```

Khi chúng ta đã dùng danh sách để cài đặt ngăn xếp thì ta nên sử dụng các phép toán trên danh sách để cài đặt các phép toán trên ngăn xếp. Sau đây là phần cài đặt ngăn xếp bằng danh sách.

Tạo ngăn xếp rỗng

```
void MAKENULL_STACK(Stack& S){
    MAKENULL_LIST(S);
}
```

Kiểm tra ngăn xếp rỗng

```
int EMPTY_STACK(Stack S){
    return EMPTY_LIST(S);
}
```

Thêm phần tử vào ngăn xếp

```
void PUSH(Elementtype X, Stack& S){
    INSERT_LIST (x, First (S), S);
}
```

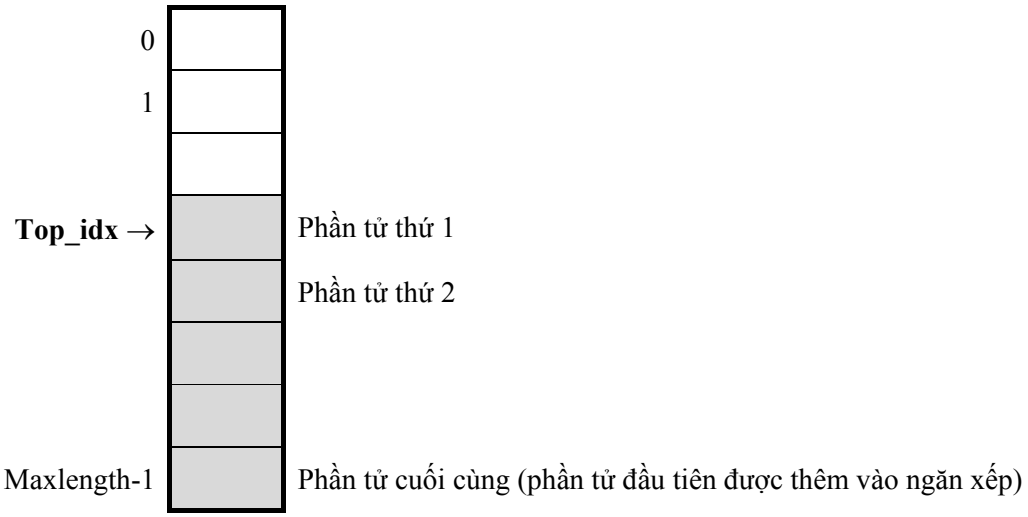
Xóa phần tử ra khỏi ngăn xếp

```
void POP (Stack& S){
    DELETE_LIST (First (S), S);
}
```

Tuy nhiên để tăng tính hiệu quả của ngăn xếp ta có thể cài đặt ngăn xếp trực tiếp từ các cấu trúc dữ liệu như các phần sau.

b. Cài đặt bằng mảng

Dùng một mảng để lưu trữ liên tiếp các phần tử của ngăn xếp. Các phần tử đưa vào ngăn xếp bắt đầu từ vị trí có chỉ số cao nhất của mảng, xem hình II.9. Ta còn phải dùng một biến số nguyên (Top_idx) giữ chỉ số của phần tử tại đỉnh ngăn xếp.



Hình II.9 Ngăn xếp

Khai báo ngăn xếp

```
#define MaxLength ... //độ dài của mảng
typedef ... ElementType; //kiểu các phần tử trong ngăn xếp
typedef struct {
    ElementType Elements[MaxLength];
    //Lưu nội dung của các phần tử
    int Top_idx; //giữ vị trí đỉnh ngăn xếp
} Stack;
```

Tạo ngăn xếp rỗng

Ngăn xếp rỗng là ngăn xếp không chứa bất kỳ một phần tử nào, do đó đỉnh của ngăn xếp không được phép chỉ đến bất kỳ vị trí nào trong mảng. Để tiện cho quá trình thêm và xóa phần tử ra khỏi ngăn xếp, khi tạo ngăn xếp rỗng ta cho đỉnh ngăn xếp nằm ở vị trí Maxlength.

```
void MAKENULL_STACK(Stack& S){
    S.Top_idx=MaxLength;
}
```

Kiểm tra ngăn xếp rỗng

```
int EMPTY_STACK(Stack S){
    return S.Top_idx==MaxLength;
}
```

Kiểm tra ngăn xếp đầy

```
int FULL_STACK(Stack S){
    return S.Top_idx==0;
}
```

Lấy nội dung phần tử tại đỉnh của ngăn xếp

Hàm trả về nội dung phần tử tại đỉnh của ngăn xếp khi ngăn xếp không rỗng. Nếu ngăn xếp rỗng thì hàm hiển thị câu thông báo lỗi.

```
ElementType TOP(Stack S){
    if (!EMPTY_STACK(S))
        return S.Elements[S.Top_idx];
    else printf("Loi! Ngan xep rong");
}
```

Chú ý: trong một số NNLT (ví dụ PASCAL) kết quả trả về của hàm không thể là một kiểu phức hợp như Record/struct. Trong trường hợp đó phải cài đặt kết quả trả về qua tham số, tức là viết hàm TOP như sau:

```
void TOP(Stack S, Elementtype& X){
//Trong đó x sẽ lưu trữ nội dung phần tử tại đỉnh của ngăn xếp
    if (!EMPTY_STACK(S))
        X = S.Elements[S.Top_idx];
    else printf("Loi: Ngan xep rong ");
}
```

Xóa phần tử ra khỏi ngăn xếp

Phần tử được xóa ra khỏi ngăn xếp là tại đỉnh của ngăn xếp. Do đó, khi xóa ta chỉ cần dịch chuyển đỉnh của ngăn xếp xuống 1 vị trí (Top_idx tăng 1 đơn vị)

```
void POP(Stack& S){
    if (!EMPTY_STACK(S))
        S.Top_idx=S.Top_idx+1;
    else printf("Loi! Ngan xep rong!");
}
```

Thêm phần tử vào ngăn xếp

Khi thêm phần tử có nội dung x (kiểu ElementType) vào ngăn xếp S (kiểu Stack), trước tiên ta phải kiểm tra xem ngăn xếp có còn chỗ trống để lưu trữ phần tử mới không. Nếu không còn chỗ trống (ngăn xếp đầy) thì báo lỗi; Ngược lại, dịch chuyển Top_idx lên trên 1 vị trí và đặt x vào tại vị trí đỉnh mới.

```
void PUSH(ElementType X, Stack& S){
    if (FULL_STACK(S))
        printf("Loi! Ngan xep day!");
    else{
        S.Top_idx=S.Top_idx-1;
        S.Elements[S.Top_idx]=X;
    }
}
```

Tất nhiên ta cũng có thể cài đặt ngăn xếp bằng con trỏ, trường hợp này xin dành cho bạn đọc xem như một bài tập nhỏ.

4. Ứng dụng ngăn xếp để loại bỏ đệ qui của chương trình

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

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

Bước 1: Lưu các biến cục bộ và địa chỉ trở về.

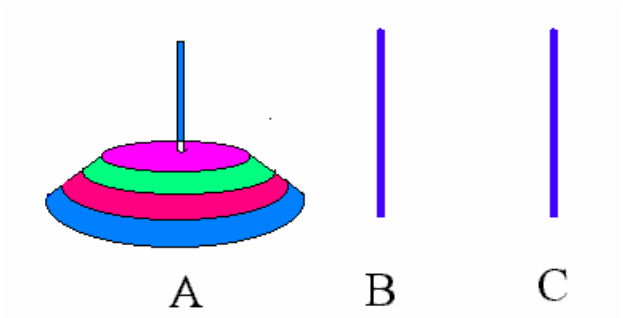
Bước 2: Nếu thoả điều kiện ngừng đệ qui thì chuyển sang bước 3. Nếu không thì tính toán từng phần và quay lại bước 1 (đệ qui tiếp).

Bước 3: Khôi phục lại các biến cục bộ và địa chỉ trở về.

Ví dụ sau đây minh hoạ việc dùng ngăn xếp để loại bỏ chương trình đệ qui của bài toán "tháp Hà Nội" (Tower of Hanoi).

Bài toán "tháp Hà Nội" được phát biểu như sau:

Có ba cọc A,B,C. Khởi đầu cọc A có một số đĩa xếp theo thứ tự nhỏ dần lên trên đỉnh. Bài toán đặt ra là phải chuyển toàn bộ chồng đĩa từ A sang B. Mỗi lần thực hiện chuyển một đĩa từ một cọc sang một cọc khác và không được đặt đĩa lớn nằm trên đĩa nhỏ (hình II.10)



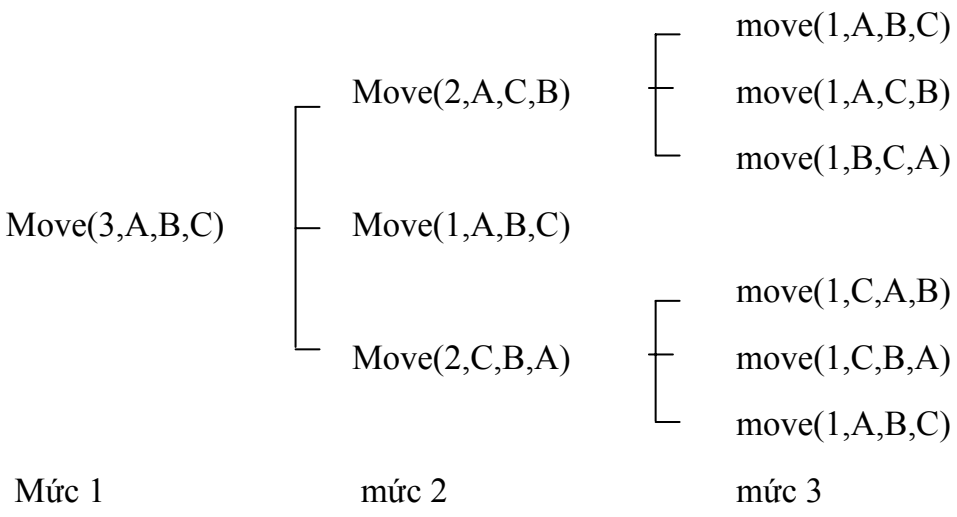
Hình II.10: Bài toán tháp Hà Nội

Chương trình con đệ qui để giải bài toán tháp Hà Nội như sau:

```
void Move(int N, int A, int B, int C) {
    //n: số đĩa, A,B,C: cọc nguồn , đích và trung gian
    if (n==1)
            printf("Chuyen 1 dia tu %c sang %c\n",A,B);
    else {
            Move(n-1, A,C,B);
            //chuyển n-1 đĩa từ cọc nguồn sang cọc trung gian
```

```
            Move(1,A,B,C);
            //chuyển 1 đĩa từ cọc nguồn sang cọc đích
            Move(n-1,C,B,A);
            //chuyển n-1 đĩa từ cọc trung gian sang cọc đích
        }
}
```

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



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

Mỗi khi chương trình con đệ qui được gọi, ứng với việc đi từ mức i vào mức i+1, ta phải lưu trữ các biến cục bộ của chương trình con ở bước i vào ngăn xếp. Ta cũng phải lưu "địa chỉ mã lệnh" chưa được thi hành của chương trình con ở mức i. Tuy nhiên khi lập trình bằng ngôn ngữ cấp cao thì đây không phải là địa chỉ ô nhớ chứa mã lệnh của máy mà ta sẽ tổ chức sao cho khi mức i+1 hoàn thành thì lệnh tiếp theo sẽ được thực hiện là lệnh đầu tiên chưa được thi hành trong mức i.

Tập hợp các biến cục bộ của mỗi lần gọi chương trình con xem như là một mẩu tin hoạt động (activation record).

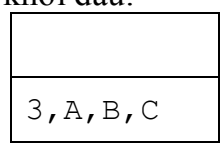
Mỗi lần thực hiện chương trình con tại mức i thì phải xoá mẩu tin lưu các biến cục bộ ở mức này trong ngăn xếp.

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

```
//Kiểu cấu trúc lưu trữ biến cục bộ
typedef struct{
	int N;
	int A, B, C;
} ElementType;
// Chương trình con MOVE không đệ qui
void Move(ElementType X){
	ElementType Temp, Temp1;
	Stack S;
	MAKENULL_STACK(S);
	PUSH(X,S);
	do
	{
		Temp=TOP(S); //Lay phan tu dau
		POP(S); //Xoa phan tu dau
		if (Temp.N==1)
			printf("Chuyen 1 dia tu %c
				sang %c\n",Temp.A,Temp.B);
		else
		{
			// Luu cho loi goi Move(n-1,C,B,A)
			Temp1.N=Temp.N-1;
			Temp1.A=Temp.C;
			Temp1.B=Temp.B;
			Temp1.C=Temp.A;
			PUSH(Temp1,S);
			// Luu cho loi goi Move(1,A,B,C)
			Temp1.N=1;
			Temp1.A=Temp.A;
			Temp1.B=Temp.B;
			Temp1.C=Temp.C;
			PUSH(Temp1,S);
			//Luu cho loi goi Move(n-1,A,C,B)
			Temp1.N=Temp.N-1;
			Temp1.A=Temp.A;
			Temp1.B=Temp.C;
			Temp1.C=Temp.B;
			PUSH(Temp1,S);
		}
	} while (!EMPTY_STACK(S));
}
```

Minh họa cho lời gọi Move(x) với 3 đĩa, tức là x.N=3.

Ngăn xếp khởi đầu:



Ngăn xếp sau lần lặp thứ nhất:

2,A,C,B
1,A,B,C
2,C,B,A

Ngăn xếp sau lần lặp thứ hai

1,A,B,C
1,A,C,B
1,B,C,A
1,A,B,C
2,C,B,A

Các lần lặp 3,4,5,6 thì chương trình con xử lý trường hợp chuyển 1 đĩa (ứng với trường hợp không gọi đệ qui), vì vậy không có mẩu tin nào được thêm vào ngăn xếp. Mỗi lần xử lý, phần tử đầu ngăn xếp bị xoá. Ta sẽ có ngăn xếp như sau.

2,C,B,A

Tiếp tục lặp bước 7 ta có ngăn xếp như sau:

1,C,A,B
1,C,B,A
1,A,B,C

Các lần lặp tiếp tục chỉ xử lý việc chuyển 1 đĩa (ứng với trường hợp không gọi đệ qui). Chương trình con in ra các phép chuyển và dẫn đến ngăn xếp rỗng.

? Viết chương trình chính gọi thực hiện chuyển ba đĩa với ba cọc A,B,C?

III. HÀNG ĐỢI (QUEUE)

1. Định Nghĩa

Hàng đợi, hay ngắn gọn là hàng (queue) cũng là một danh sách đặc biệt mà phép thêm vào chỉ thực hiện tại một đầu của danh sách, gọi là cuối hàng (REAR), còn phép loại bỏ thì thực hiện ở đầu kia của danh sách, gọi là đầu hàng (FRONT).

Xếp hàng mua vé xem phim là một hình ảnh trực quan của khái niệm trên, người mới đến thêm vào cuối hàng còn người ở đầu hàng mua vé và ra khỏi hàng. Vì vậy hàng còn được gọi là cấu trúc **FIFO** (first in - first out) hay "vào trước - ra trước".

Bây giờ chúng ta sẽ thảo luận một vài phép toán cơ bản nhất trên hàng

2. Các phép toán cơ bản trên hàng

- **MAKENULL_QUEUE(Q)** khởi tạo một hàng rỗng.
- **FRONT(Q)** hàm trả về phần tử đầu tiên của hàng Q.
- **ENQUEUE(x,Q)** thêm phần tử x vào cuối hàng Q.
- **DEQUEUE(Q)** xoá phần tử tại đầu của hàng Q.
- **EMPTY_QUEUE(Q)** hàm kiểm tra hàng rỗng.
- **FULL_QUEUE(Q)** kiểm tra hàng đầy.

3. Cài đặt hàng

Như đã trình bày trong phần ngăn xếp, ta hoàn toàn có thể dùng danh sách để biểu diễn cho một hàng và dùng các phép toán đã được cài đặt của danh sách để cài đặt các phép toán trên hàng. Tuy nhiên làm như vậy có khi sẽ không hiệu quả, chẳng hạn dùng danh sách cài đặt bằng mảng ta thấy lời gọi INSERT_LIST(x,ENDLIST(Q),Q) tốn một hằng thời gian trong khi lời gọi DELETE_LIST(FIRST(Q),Q) để xoá phần tử đầu hàng (phần tử ở vị trí 0 của mảng) ta phải tốn thời gian tỉ lệ với số các phần tử trong hàng để thực hiện việc dời toàn bộ hàng lên một vị trí. Để cài đặt hiệu quả hơn ta phải có một suy nghĩ khác dựa trên tính chất đặc biệt của phép thêm và loại bỏ một phần tử trong hàng.

a. Cài đặt hàng bằng mảng

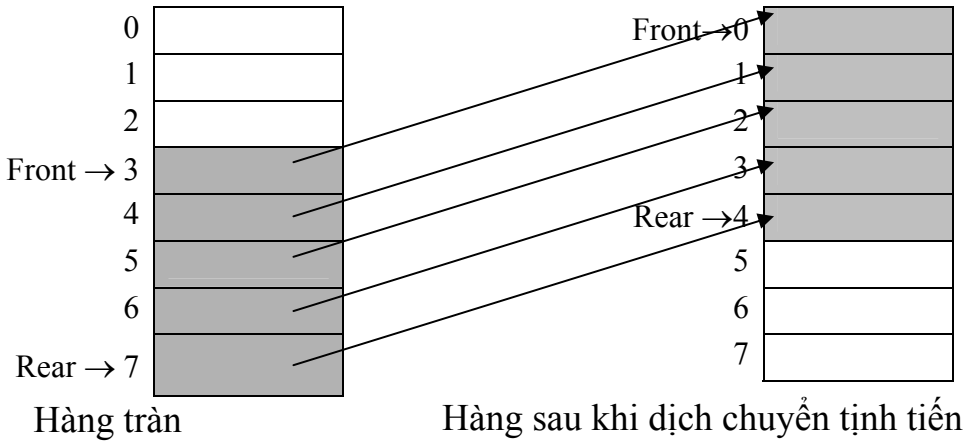
Ta dùng một mảng để chứa các phần tử của hàng, khởi đầu phần tử đầu tiên của hàng được đưa vào vị trí thứ 1 của mảng (vào vị trí có chỉ số 0), phần tử thứ 2 vào vị trí thứ 2 của mảng (vào vị trí có chỉ số 1),... Giả sử hàng có n phần tử, ta có front=0 và rear=n-1.

- Khi xoá một phần tử front tăng lên 1,
- Khi thêm một phần tử rear tăng lên 1.

Như vậy hàng có khuynh hướng đi xuống, đến một lúc nào đó ta không thể thêm vào hàng được nữa (rear=maxlength-1) dù mảng còn nhiều chỗ trống (các vị trí trước front) trường hợp này ta gọi là *hàng bị tràn* (xem hình II.11).Trong trường hợp toàn bộ mảng đã chứa các phần tử của hàng ta gọi là *hàng bị đầy*.

Cách khắc phục hàng bị tràn

- Dời toàn bộ hàng lên front vị trí, cách này gọi là di chuyển tịnh tiến. Trong trường hợp này ta luôn có front<=rear (hình II.11).
- Xem mảng như là một vòng tròn nghĩa là khi hàng bị tràn nhưng chưa đầy ta thêm phần tử mới vào vị trí 0 của mảng, thêm một phần tử mới nữa thì thêm vào vị trí 1 (nếu có thể)...Rõ ràng cách làm này front có thể lớn hơn rear. Cách khắc phục này gọi là dùng mảng xoay vòng (xem hình II.12).



Hình II.11 : Minh họa việc di chuyển tịnh tiến các phần tử khi hàng bị tràn

Cài đặt hàng bằng mảng theo phương pháp tịnh tiến

Để quản lí một hàng ta chỉ cần quản lí đầu hàng và cuối hàng. Có thể dùng 2 biến số nguyên chỉ vị trí đầu hàng và cuối hàng

Các khai báo cần thiết

```
#define MaxLength ... //chiều dài tối đa của mảng
typedef ... ElementType;
        //Kiểu dữ liệu của các phần tử trong hàng
    typedef struct {
        ElementType Elements[MaxLength];
        //Lưu trữ nội dung các phần tử
        int Front, Rear; //chỉ số đầu và đuôi hàng
    } Queue;
```

Tạo hàng rỗng

Lúc này front và rear không trỏ đến vị trí hợp lệ nào trong mảng vậy ta có thể cho front và rear đều bằng -1.

```
void MAKENULL_QUEUE(Queue& Q){
    Q.Front=-1;
    Q.Rear=-1;
}
```

Kiểm tra hàng rỗng

Trong quá trình làm việc ta có thể thêm và xóa các phần tử trong hàng. Rõ ràng, nếu ta có đưa vào hàng một phần tử nào đó thì front>-1. Khi xoá một phần tử ta tăng front lên 1. Hàng rỗng nếu front>rear. Hơn nữa khi mới khởi tạo hàng, tức là front = -1, thì hàng cũng rỗng. Tuy nhiên để phép kiểm tra hàng rỗng đơn giản, ta sẽ làm một phép kiểm tra khi xoá một phần tử của hàng, nếu phần tử bị xoá là phần tử duy nhất trong hàng thì ta đặt lại front=-1. Vậy hàng rỗng khi và chỉ khi front =-1.

```
int EMPTY_QUEUE(Queue Q){
    return Q.Front==-1;
}
```

Kiểm tra đầy

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

```
int FULL_QUEUE(Queue Q){
    return (Q.Rear-Q.Front+1)==MaxLength;
}
```

Xóa phần tử ra khỏi hàng

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

```
void DEQUEUE(Queue& Q){
    if (!EMPTY_QUEUE(Q)){
        Q.Front=Q.Front+1;
        if (Q.Front>Q.Rear) MAKENULL_QUEUE(Q); //Dat lai hang rong
    }
    else printf("Loi: Hang rong!");
}
```

Thêm phần tử vào hàng

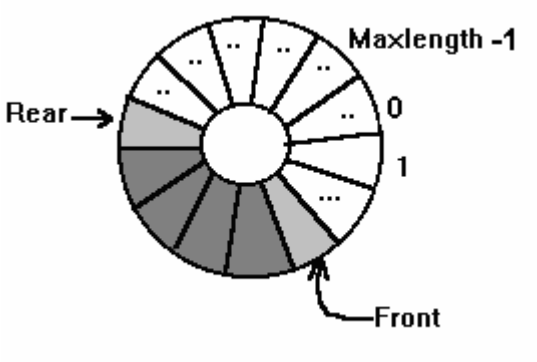
Một phần tử khi được thêm vào hàng sẽ nằm kế vị trí Rear cũ của hàng. Khi thêm một phần tử vào hàng ta phải xét các trường hợp sau:

- Nếu hàng đầy thì báo lỗi không thêm được nữa.

- Nếu hàng chưa đầy ta phải xét xem hàng có bị tràn không. Nếu hàng bị tràn ta di chuyển tịnh tiến rồi mới nối thêm phần tử mới vào đuôi hàng (rear tăng lên 1). Đặc biệt nếu thêm vào hàng rỗng thì ta cho front=0 để front trỏ đúng phần tử đầu tiên của hàng.

```
void ENQUEUE(ElementType X,Queue& Q){
    if (!FULL_QUEUE(Q)){
        if (EMPTY_QUEUE(Q)) Q.Front=0;
        if (Q.Rear==MaxLength-1){
        //Di chuyen tinh tien ra truoc Front -1 vi tri
            for(int i=Q.Front;i<=Q.Rear;i++)
                Q.Elements[i-Q.Front]=Q.Elements[i];
            //Xac dinh vi tri Rear moi
            Q.Rear=MaxLength – Q.Front-1;
            Q.Front=0;
        }
        //Tang Rear de luu noi dung moi
        Q.Rear=Q.Rear+1;
        Q.Elements[Q.Rear]=X;
    }
    else printf("Loi: Hang day!");
}
```

b. Cài đặt hàng với mảng xoay vòng



Hình II.12 Cài đặt hàng bằng mảng xoay vòng

Với phương pháp này, khi hàng bị tràn, tức là rear=maxlength-1, nhưng chưa đầy, tức là front>0, thì ta thêm phần tử mới vào vị trí 0 của mảng và cứ tiếp tục như vậy vì từ 0 đến front-1 là các vị trí trống. Vì ta sử dụng mảng một cách xoay vòng như vậy nên phương pháp này gọi là phương pháp dùng mảng xoay vòng.

Các phần khai báo cấu trúc dữ liệu, tạo hàng rỗng, kiểm tra hàng rỗng giống như phương pháp di chuyển tịnh tiến.

Khai báo cần thiết

```
#define MaxLength ... //chiều dài tối đa của mảng
typedef ... ElementType;
        //Kiểu dữ liệu của các phần tử trong hàng
    typedef struct {
        ElementType Elements[MaxLength];
        //Lưu trữ nội dung các phần tử
        int Front, Rear; //chỉ số đầu và đuôi hàng
    } Queue;
```

Tạo hàng rỗng

Lúc này front và rear không trỏ đến vị trí hợp lệ nào trong mảng vậy ta có thể cho front và rear đều bằng -1.

```
void MAKENULL_QUEUE(Queue& Q){
    Q.Front=-1;
    Q.Rear=-1;
}
```

Kiểm tra hàng rỗng

```
int EMPTY_QUEUE(Queue Q){
    return Q.Front==-1;
}
```

Kiểm tra hàng đầy

Hàng đầy nếu toàn bộ các ô trong mảng đang chứa các phần tử của hàng. Với phương pháp này thì Front có thể lớn hơn Rear. Ta có hai trường hợp hàng đầy như sau:

- Trường hợp Q.Rear=Maxlength-1 và Q.Front =0

- Trường hợp Q.Front =Q.Rear+1.

Để đơn giản ta có thể gom cả hai trường hợp trên lại theo một công thức như sau:

(Q.rear-Q.front +1) mod Maxlength =0

Thủ tục kiểm tra hàng rỗng:

```
int FULL_QUEUE(Queue Q){
    return (Q.Rear-Q.Front+1) % MaxLength==0;
}
```

Xóa một phần tử ra khỏi hàng

Khi xóa một phần tử ra khỏi hàng, ta xóa tại vị trí đầu hàng và có thể xảy ra các trường hợp sau:

- Nếu hàng rỗng thì báo lỗi không xóa;
- Ngược lại, nếu hàng chỉ còn 1 phần tử thì khởi tạo lại hàng rỗng;
- Nếu hàng có nhiều hơn 1 phần tử, thay đổi giá trị của Q.Front (Nếu Q.front != Maxlength-1 thì đặt lại Q.front = Q.Front +1; Ngược lại Q.front=0)

```
void DEQUEUE(Queue& Q){
    if (!EMPTY_QUEUE(Q)){
    //Nếu hàng chỉ chứa một phần tử thì khởi tạo hàng lại
        if (Q.Front==Q.Rear) MAKENULL_QUEUE(Q);
        else Q.Front=(Q.Front+1) % MaxLength; //tăng Front lên 1 đơn vị
    }
    else printf("Loi: Hang rong!");
}
```

Thêm một phần tử vào hàng

Khi thêm một phần tử vào hàng thì có thể xảy ra các trường hợp sau:

- Trường hợp hàng đầy thì báo lỗi và không thêm;

- Ngược lại, thay đổi giá trị của Q.rear (Nếu Q.rear =maxlength-1 thì đặt lại Q.rear=0; Ngược lại Q.rear =Q.rear+1) và đặt nội dung vào vị trí Q.rear mới.

```
void ENQUEUE(ElementType X,Queue& Q){
    if (!FULL_QUEUE(Q)){
        if (EMPTY_QUEUE(Q)) Q.Front=0;
        Q.Rear=(Q.Rear+1) % MaxLength;
        Q.Elements[Q.Rear]=X;
    }
    else printf("Loi: Hang day!");
}
```

? Cài đặt hàng bằng mảng vòng có ưu điểm gì so với bằng mảng theo phương pháp tịnh tiến? Trong ngôn ngữ lập trình có kiểu dữ liệu mảng vòng không?

c. Cài đặt hàng bằng danh sách liên kết (cài đặt bằng con trỏ)

Cách tự nhiên nhất là dùng hai con trỏ Font và Rear để trỏ tới phần tử đầu và cuối hàng. Hàng được cài đặt như một danh sách liên kết có Header là một ô thực sự, xem hình II.13.

Khai báo cần thiết

```
typedef ... ElementType; //kiểu phần tử của hàng
typedef struct Node{
        ElementType Element;
        Node* Next; //Con trỏ chỉ ô kế tiếp
};
typedef Node* Position;
typedef struct{
        Position Front, Rear;
                //là hai trường chỉ đến đầu và cuối của hàng
} Queue;
```

Khởi tạo hàng rỗng

Khi hàng rỗng Front va Rear cùng trỏ về 1 vị trí đó chính là ô header

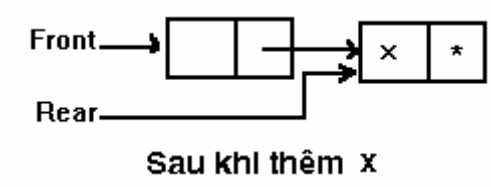
Hình II.13: Khởi tạo hàng rỗng

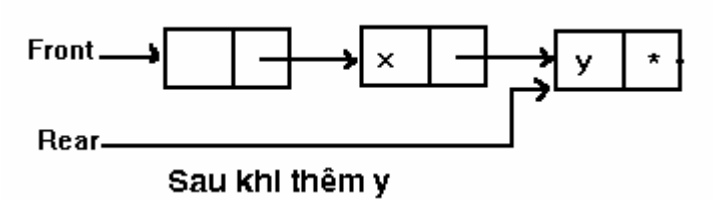
```
void MAKENULL_QUEUE(Queue& Q){
        Position Header=(Node*)malloc(sizeof(Node)); //Cấp phát Header
        Header->Next=NULL;
        Q.Front=Header;
        Q.Rear=Header;
}
```

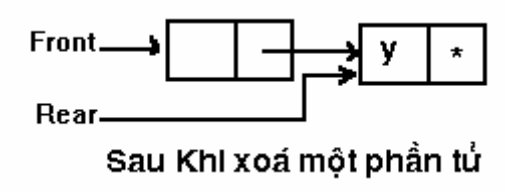
Kiểm tra hàng rỗng

Hàng rỗng nếu Front và Rear chỉ cùng một vị trí là ô Header.

```
int EMPTY_QUEUE(Queue Q){
        return (Q.Front==Q.Rear);
}
```







Hình II.14 Hàng sau khi thêm và xóa phần tử

Thêm một phần tử vào hàng

Thêm một phần tử vào hàng ta thêm vào sau Rear (Rear->Next), rồi cho Rear trỏ đến phần tử mới này, xem hình II.14. Trường Next của ô mới này trỏ tới **NULL**.

```
void ENQUEUE(ElementType X, Queue& Q){
    Q.Rear->Next=(Node*)malloc(sizeof(Node));
    Q.Rear=Q.Rear->Next;
    //Dat gia tri vao cho Rear
    Q.Rear->Element=X;
    Q.Rear->Next=NULL;
}
```

Xóa một phần tử ra khỏi hàng

Thực chất là xoá phần tử nằm ở vị trí đầu hàng do đó ta chỉ cần cho Front trỏ tới vị trí kế tiếp của nó trong hàng.

```
void DEQUEUE(Queue& Q){
    if (!EMPTY_QUEUE(Q)){
            Position T=Q.Front;
            Q.Front=Q.Front->Next;
            free(T);
    }
    else printf("Loi : Hang rong");
}
```

4. Một số ứng dụng của cấu trúc hàng

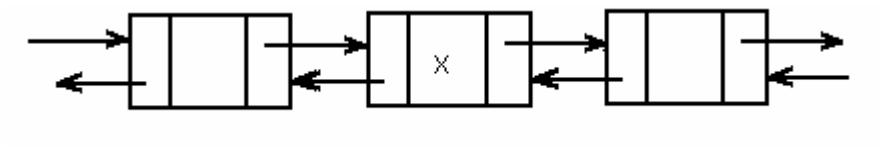
Hàng đợi là một cấu trúc dữ liệu được dùng khá phổ biến trong thiết kế giải thuật. Bất kỳ nơi nào ta cần quản lí dữ liệu, quá trình... theo kiểu vào trước-ra trước đều có thể ứng dụng hàng đợi.

Ví dụ rất dễ thấy là quản lí in trên mạng, nhiều máy tính yêu cầu in đồng thời và ngay cả một máy tính cũng yêu cầu in nhiều lần. Nói chung có nhiều yêu cầu in dữ liệu, nhưng máy in không thể đáp ứng tức thời tất cả các yêu cầu đó nên chương trình quản lí in sẽ thiết lập một hàng đợi để quản lí các yêu cầu. Yêu cầu nào mà chương trình quản lí in nhận trước nó sẽ giải quyết trước.

Một ví dụ khác là duyệt cây theo mức được trình bày chi tiết trong chương sau. Các giải thuật duyệt theo chiều rộng một đồ thị có hướng hoặc vô hướng cũng dùng hàng đợi để quản lí các nút đồ thị. Các giải thuật đổi biểu thức trung tố thành hậu tố, tiền tố.

IV. DANH SÁCH LIÊN KẾT KÉP (DOUBLE - LISTS)

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



Hình II.15 Hình ảnh một danh sách liên kết kép

Các khai báo cần thiết

```
typedef ... ElementType;
    //kiểu nội dung của các phần tử trong danh sách
typedef struct Node{
    ElementType Element; //lưu trữ nội dung phần tử
    //Hai con trỏ trỏ tới phần tử trước và sau
    Node* Previous;
    Node* Next;
};
typedef Node* Position;
typedef Position DoubleList;
```

Để quản lí một danh sách liên kết kép ta có thể dùng một con trỏ trỏ đến một ô bất kỳ trong cấu trúc. Hoàn toàn tương tự như trong danh sách liên kết đơn đã trình bày trong phần trước, con trỏ để quản lí danh sách liên kết kép có thể là một con trỏ có kiểu giống như kiểu phần tử trong danh sách và nó có thể được cấp phát ô nhớ (tương tự như Header trong danh sách liên kết đơn) hoặc không được cấp phát ô nhớ. Ta cũng có thể xem danh sách liên kết kép như là danh sách liên kết đơn, với một bổ sung duy nhất là có con trỏ previous để nối kết các ô theo chiều ngược lại. Theo quan điểm này thì chúng ta có thể cài đặt các phép toán thêm (insert), xoá (delete) một phần tử hoàn toàn tương tự như trong danh sách liên kết đơn và con trỏ Header cũng cần thiết như trong danh sách liên kết đơn, vì nó chính là vị trí của phần tử đầu tiên trong danh sách.

Tuy nhiên, nếu tận dụng khả năng duyệt theo cả hai chiều thì ta *không cần phải cấp phát bộ nhớ cho Header* và vị trí (position) của một phần tử trong danh sách có thể định nghĩa như sau: *Vị trí của phần tử a_i là con trỏ trỏ tới ô chứa a_i, tức là địa chỉ ô nhớ chứa a_i. Nói nôm na, vị trí của a_i là ô chứa a_i.* Theo định nghĩa vị trí như vậy các phép toán trên danh sách liên kết kép sẽ được cài đặt hơi khác với danh sách liên đơn. Trong cách cài đặt này, chúng ta không sử dụng ô đầu mục (Header) như danh sách liên kết đơn mà sẽ quản lý danh sách một các trực tiếp.

Tạo danh sách liên kết kép rỗng

Giả sử DL là con trỏ quản lí danh sách liên kết kép thì khi khởi tạo danh sách rỗng ta cho con trỏ này trỏ **NULL** (không cấp phát ô nhớ cho DL), tức là gán DL=**NULL**.

```
void MAKENULL_LIST (DoubleList& DL){
 DL= NULL;
}
```

Kiểm tra danh sách liên kết kép rỗng

Rõ ràng, danh sách liên kết kép rỗng khi và chỉ khi chỉ điểm đầu danh sách không trỏ tới một ô xác định nào cả. Do đó ta sẽ kiểm tra DL = **NULL**.

```
int EMPTY_LIST (DoubleList DL){
      return (DL==NULL);
}
```

Xóa một phần tử ra khỏi danh sách liên kết kép

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

- Danh sách rỗng, tức là DL=**NULL**: chương trình con dừng.

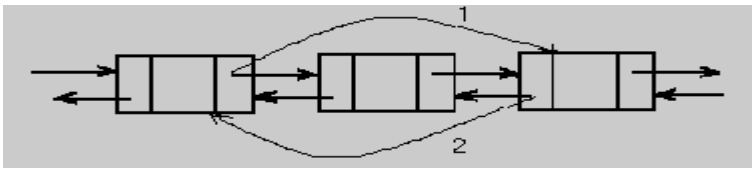
- Trường hợp danh sách khác rỗng, tức là DL!=**NULL**, ta phải phân biệt hai trường hợp

Ô bị xoá không phải là ô được trỏ bởi DL, ta chỉ cần cập nhật lại các con trỏ để nối kết ô trước p với ô sau p, các thao tác cần thiết là (xem hình II.16):

Nếu (p->previous!=**NULL**) thì p->previous->next=p->next;

Nếu (p->next!=**NULL**) thì p->next->previous=p->previous;

Xoá ô đang được trỏ bởi DL, tức là p=DL: ngoài việc cập nhật lại các con trỏ để nối kết các ô trước và sau p ta còn phải cập nhật lại DL, ta có thể cho DL trỏ đến phần tử trước nó (DL = p->Previous) hoặc đến phần tử sau nó (DL = p->Next) tuỳ theo phần tử nào có mặt trong danh sách. Đặc biệt, nếu danh sách chỉ có một phần tử tức là p->Next=**NULL** và p->Previous=**NULL** thì DL=**NULL**.



p->Previous *p* *p->Next*

Hình II.16 Xóa phần tử tại vị trí p

```
void DELETE_LIST (Position p, DoubleList& DL){
    if (DL == NULL) printf("Danh sach rong");
    else{
        if (p==DL){  //Xóa phần tử đầu tiên của danh sách nên phải thay đổi DL
            if (p->next!=NULL)
                DL=DL->Next;
            else     DL=DL->Previous;
            free(p);
        }
        else{
            //noi ket lai cac con tro
            if (p->Previous != NULL) p->Previous->Next=p->Next;
            if (p->Next != NULL)  p->Next->Previous=p->Previous;
            free(p);
        }
    }
}
```

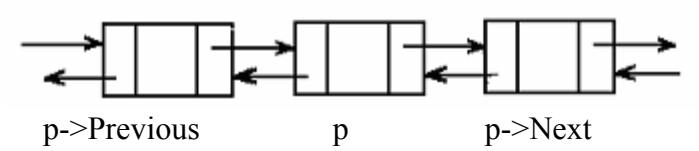
Thêm phần tử vào danh sách liên kết kép

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

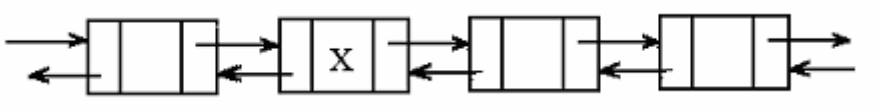
Danh sách rỗng, tức là DL = **NULL**: trong trường hợp này ta không quan tâm đến giá trị của p. Để thêm một phần tử, ta chỉ cần cấp phát ô nhớ cho nó, gán giá trị x vào trường Element của ô nhớ này và cho hai con trỏ previous, next trỏ tới **NULL** còn DL trỏ vào ô nhớ này, các thao tác trên có thể viết như sau:

```
DL=(Node*)malloc(sizeof(Node));
DL->Element = x;
DL->Previous=NULL;
DL->Next =NULL;
```

Nếu DL!=**NULL**, sau khi thêm phần tử x vào vị trí p ta có kết quả như hình II.18



Hình II.17: Danh sách trước khi thêm phần tử x



Hình II.18: Danh sách sau khi thêm phần tử x vào tại vị trí p

(phần tử tại vị trí p cũ trở thành phần tử "sau" của x)

Lưu ý: các kí hiệu p, p->Next, p->Previous trong hình II.18 để chỉ các ô trước khi thêm phần tử x, tức là nó chỉ các ô trong hình II.17.

Trong trường hợp p=DL, ta có thể cập nhật lại DL để DL trỏ tới ô mới thêm vào hoặc để nó trỏ đến ô tại vị trí p cũ như nó đang trỏ cũng chỉ là sự lựa chọn trong chi tiết cài đặt.

```
void INSERT_LIST (ElementType X,Position p, DoubleList& DL){
    if (DL == NULL){
        DL=(Node*)malloc(sizeof(Node));
        DL->Element = X;
        DL->Previous =NULL;
        DL->Next =NULL;
    }
    else{
        Position temp = (Node*)malloc(sizeof(Node));
        temp->Element=X;
        temp->Next=p;
        temp->Previous=p->Previous;
        if (p->Previous!=NULL)
            p->Previous->Next=temp;
        p->Previous=temp;
    }
}
```

TỔNG KẾT CHƯƠNG

Chương mô tả các cấu trúc dữ liệu trừu tượng và các giải thuật cài đặt các phép toán này. Tùy theo bài toán cụ thể và mức độ thay đổi của dữ liệu mà ta lựa chọn các cấu trúc dữ liệu cho phù hợp. Phần cơ bản nhất của chương là cấu trúc danh sách và cách thức cài đặt cấu trúc này. Kế đến là các cấu trúc danh sách đặc biệt: Ngăn xếp và hàng đợi. Danh sách liên kết kép được xem như một sự biến tấu của danh sách liên kết đơn.

BÀI TẬP

1. Viết khai báo và các chương trình con cài đặt danh sách bằng mảng. Dùng các chương trình con này để viết:

a. Chương trình con nhận một dãy các số nguyên nhập từ bàn phím, lưu trữ nó trong danh sách theo thứ tự nhập vào.

b. Chương trình con nhận một dãy các số nguyên nhập từ bàn phím, lưu trữ nó trong danh sách theo thứ tự ngược với thứ tự nhập vào.

c. Viết chương trình con in ra màn hình các phần tử trong danh sách theo thứ tự của nó trong danh sách.

2. Tương tự như bài tập 1. nhưng cài đặt bằng con trỏ.

3. Viết chương trình con sắp xếp một danh sách chứa các số nguyên, trong các trường hợp:

a. Danh sách được cài đặt bằng mảng (danh sách đặc).

b. Danh sách được cài đặt bằng con trỏ (danh sách liên kết).

4. Viết chương trình con thêm một phần tử trong danh sách đã có thứ tự sao cho ta vẫn có một danh sách có thứ tự bằng cách vận dụng các phép toán cơ bản trên danh sách

5. Viết chương trình con tìm kiếm và xóa một phần tử trong danh sách có thứ tự.

6. Viết chương trình con nhận vào từ bàn phím một dãy số nguyên, lưu trữ nó trong một danh sách có thứ tự không giảm, theo cách sau: với mỗi phần tử được nhập vào chương trình con phải tìm vị trí thích hợp để xen nó vào danh sách cho đúng thứ tự. Viết chương trình con trên cho trường hợp danh sách được cài đặt bằng mảng và cài đặt bằng con trỏ.

7. Viết chương trình con loại bỏ các phần tử trùng nhau (giữ lại duy nhất 1 phần tử) trong một danh sách có thứ tự không giãm, trong hai trường hợp: cài đặt bằng mảng và cài đặt bằng con trỏ.

8. Viết chương trình con nhận vào từ bàn phím một dãy số nguyên, lưu trữ nó trong một danh sách có thứ tự tăng không có hai phần tử trùng nhau, theo cách sau: với mỗi phần tử được nhập vào chương trình con phải tìm kiếm xem nó có trong danh sách chưa, nếu chưa có thì xen nó vào danh sách cho đúng thứ tự. Viết chương trình con trên cho trường hợp danh sách được cài đặt bằng mảng và cài đặt bằng con trỏ.

9. Viết chương trình con trộn hai danh sách liên kết chứa các số nguyên theo thứ tự tăng để được một danh sách cũng có thứ tự tăng.

10. Viết chương trình con xoá khỏi danh sách lưu trữ các số nguyên các phần tử là số nguyên lẻ, cũng trong hai trường hợp: cài đặt bằng mảng và bằng con trỏ.

11. Viết chương trình con tách một danh sách chứa các số nguyên thành hai danh sách: một danh sách gồm các số chẵn còn cái kia chứa các số lẻ.

12. Hình dưới đây biểu diễn cho mảng SPACE có 10 phần tử dùng để biểu diễn danh sách bằng con nháy (cursor) và hai danh sách L1 ; L2 đang có trong mảng

	0	w	9
	1	h	4
A→	2		8
	3		-1
	4	x	6
L_1 →	5	g	1
	6	i	-1
L_2 →	7	y	0
	8		3
	9	u	-1
		Element	Next

SPACE

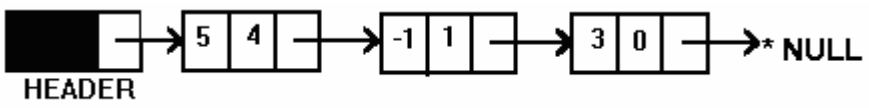
a. Hãy liệt kê các phần tử trong mỗi danh sách L1, L2.

b. Vẽ lại hình đã cho lần lượt sau các lời gọi INSERT_LIST('o',1,L1), INSERT_LIST('m',6,L1), INSERT_LIST('k',9,L2).

c. Vẽ lại hình ở câu b. sau khi xoá : x,y.

13. Đa thức $P(x)= a_nx_n+ a_{n-1}x_{n-1}+... + a_1x + a_0$ được lưu trữ trong máy tính dưới dạng một danh sách liên kết mà mỗi phần tử của danh sách là một struct có ba trường lưu giữ hệ số, số mũ, và trường NEXT trỏ đến phần tử kế tiếp. Chú ý cách lưu trữ đảm bảo thứ tự giảm dần theo số mũ của từng hạng tử của đa thức.

Ví dụ: đa thức $5x^4 - x + 3$ được lưu trữ trong danh sách có 3 phần tử như sau:



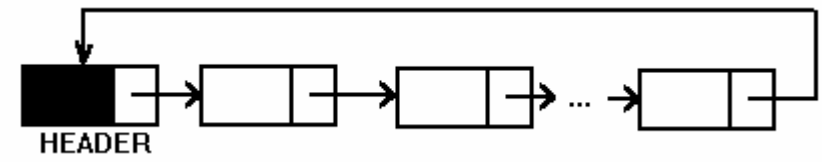
a. Hãy viết chương trình thực hiện được sự lưu trữ này.

b. Dựa vào sự cài đặt ở trên, viết chương trình con thực hiện việc cộng hai đa thức.

c. Viết chương trình con lấy đạo hàm của đa thức.

14. Để lưu trữ một số nguyên lớn, ta có thể dùng danh sách liên kết chứa các chữ số của nó. Hãy tìm cách lưu trữ các chữ số của một số nguyên lớn theo ý tưởng trên sao cho việc cộng hai số nguyên lớn là dễ dàng thực hiện. Viết chương trình con cộng hai số nguyên lớn.

15. Để tiện cho việc truy nhập vào danh sách, người ta tổ chức danh sách liên kết có dạng sau, gọi là danh sách nối vòng:



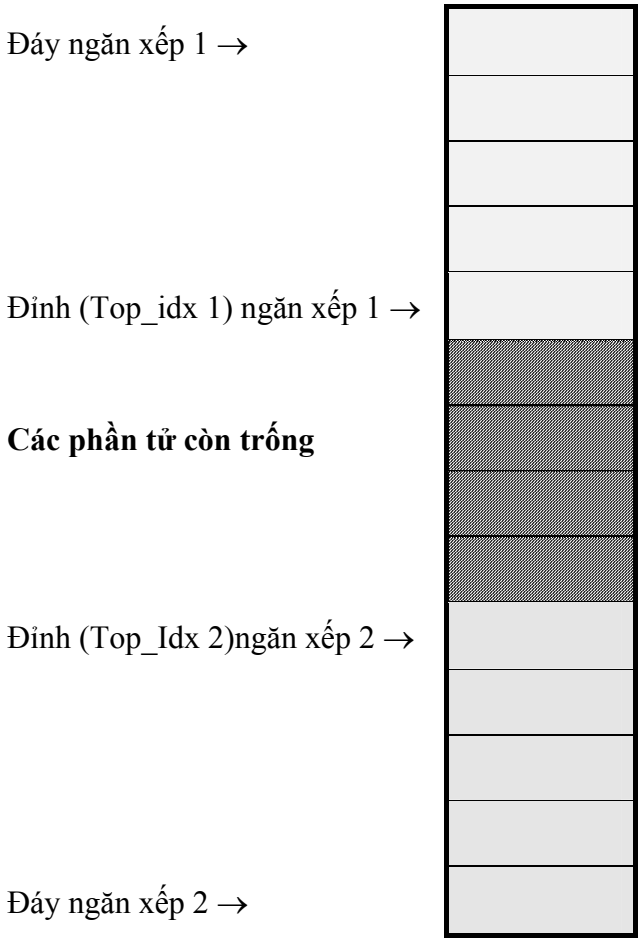
Hãy viết khai báo và các chương trình con cơ bản để cài đặt một danh sách nối vòng.

16. Hãy cài đặt một ngăn xếp bằng cách dùng con trỏ.

a. Dùng ngăn xếp để viết chương trình con đổi một số thập phân sang số nhị phân.

b. Viết chương trình con/hàm kiểm tra một chuỗi dấu ngoặc đúng (chuỗi dấu ngoặc đúng là chuỗi dấu mở đóng khớp nhau như trong biểu thức toán học).

17. Ta có thể cài đặt 2 ngăn xếp vào trong một mảng, gọi là ngăn xếp hai đầu hoạt động của hai ngăn xếp này như sơ đồ sau:



Hình vẽ mảng chứa 2 ngăn xếp

Hãy viết các chương trình con cần thiết để cài đặt ngăn xếp hai đầu.

18. Mô phỏng việc tạo buffer in một file ra máy in.

Buffer xem như là một hàng, khi ra lệnh in file máy tính sẽ thực hiện một cách lặp quá trình sau cho đến hết file:

a. Đưa nội dung của tập tin vào buffer cho đến khi buffer đầy hoặc hết file.

b. In nội dung trong buffer ra máy in cho tới khi hàng rỗng.

c. Hãy mô phỏng quá trình trên để in một file văn bản lên từng trang màn hình.

19. Khử đệ qui các hàm sau:

a. Hàm tính tổ hợp chập k của n phần tử

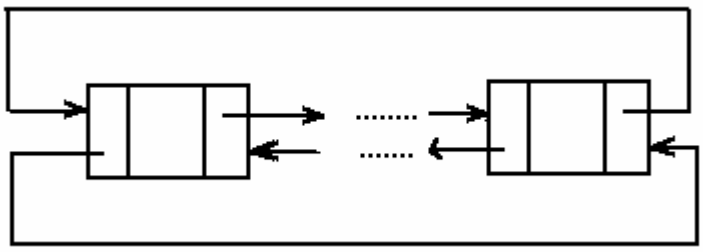
```
int TH(int k, int n){
// với giả thiết 0<=k<=n

        if ((k==0) || (k==n))
                return 1;
        else
                return (TH(k-1,n-1)+TH(k,n-1));
}
```

b. Hàm tính dãy Fibonaci theo n

```
int Fibo(int n){
//với giả thiết n>=0
        if ((n==0) || (n=1))
                return 1;
        else
                return (Fibo(n-2)+Fibo(n-1));
}
```

20. Cài đặt danh sách liên kết kép với các phép toán khởi tạo danh sách rỗng, thêm xoá một phần tử.

21. Danh sách liên kết kép nối vòng có dạng sau:



Hãy cài đặt danh sách liên kết kép dạng nối vòng như trên.

CHƯƠNG III CẤU TRÚC CÂY (TREES)

I. CÁC THUẬT NGỮ CƠ BẢN TRÊN CÂY

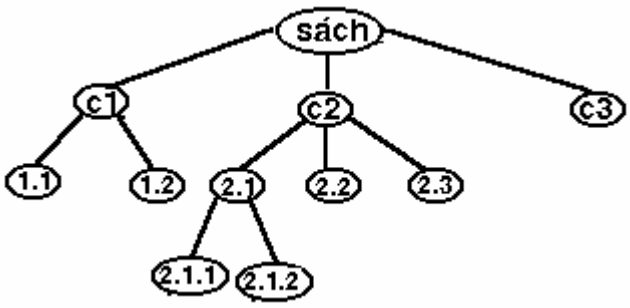
Cây là một tập hợp các phần tử gọi là nút (nodes) trong đó có một nút được phân biệt gọi là nút gốc (root). Trên tập hợp các nút này có một quan hệ, gọi là mối quan hệ *cha - con* (parenthood), để xác định hệ thống cấu trúc trên các nút. Mỗi nút, trừ nút gốc, có duy nhất một nút cha. Một nút có thể có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

1. Định nghĩa

➢ Một nút đơn độc là một cây. Nút này cũng chính là nút gốc của cây.

➢ Giả sử ta có n là một nút đơn độc và k cây T_1,.., T_k với các nút gốc tương ứng là n_1,.., n_k thì có thể xây dựng một cây mới bằng cách cho nút n là cha của các nút n_1,.., n_k. Cây mới này có nút gốc là nút n và các cây T_1,.., T_k được gọi là các cây con. Tập rỗng cũng được coi là một cây và gọi là cây rỗng kí hiệu ∅.

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



Hình III.1 - Cây mục lục một quyển sách

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

Nếu n_1,.., n_k là một chuỗi các nút trên cây sao cho n_i là nút cha của nút n_{i+1}, với i=1..k-1, thì chuỗi này gọi là một *đường đi trên cây* (hay ngắn gọn là đường đi) từ n_1 đến n_k. *Độ dài đường đi* được định nghĩa bằng số nút trên đường đi trừ 1. Như vậy độ dài đường đi từ một nút đến chính nó bằng không.

Nếu có đường đi từ nút a đến nút b thì ta nói a là *tiền bối* (ancestor) của b, còn b gọi là *hậu duệ* (descendant) của nút a. Rõ ràng *một nút vừa là tiền bối vừa là hậu duệ của chính nó*. Tiền bối hoặc hậu duệ của một nút khác với chính nó gọi là tiền bối hoặc hậu duệ thực sự. Trên cây *nút gốc* không có tiền bối thực sự. Một nút không có hậu duệ

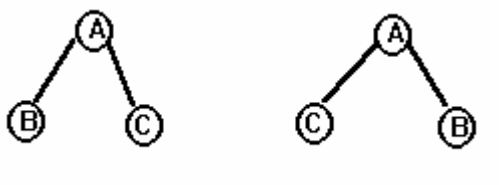
thực sự gọi là *nút lá* (leaf). Nút không phải là lá ta còn gọi là *nút trung gian* (interior). Cây con của một cây là một nút cùng với tất cả các hậu duệ của nó.

Chiều cao của một nút là độ dài đường đi lớn nhất từ nút đó tới lá. *Chiều cao của cây* là chiều cao của nút gốc. *Độ sâu của một nút* là độ dài đường đi từ nút gốc đến nút đó. Các nút có cùng một độ sâu i ta gọi là các nút có cùng một mức i. Theo định nghĩa này thì nút gốc ở mức 0, các nút con của nút gốc ở mức 1.

Ví dụ: đối với cây trong hình III.1 ta có nút C2 có chiều cao 2. Cây có chiều cao 3. nút C3 có chiều cao 0. Nút 2.1 có độ sâu 2. Các nút C1,C2,C3 cùng mức 1.

2. Thứ tự các nút trong cây

Nếu ta phân biệt thứ tự các nút con của cùng một nút thì cây gọi là cây có thứ tự, thứ tự qui ước từ trái sang phải. Như vậy, nếu kể thứ tự thì hai cây sau là hai cây khác nhau:

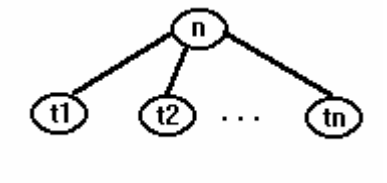


Hình III.2: Hai cây có thứ tự khác nhau

Trong trường hợp ta không phân biệt rõ ràng thứ tự các nút thì ta gọi là cây không có thứ tự. Các nút con cùng một nút cha gọi là các nút anh em ruột (siblings). Quan hệ "trái sang phải" của các anh em ruột có thể mở rộng cho hai nút bất kỳ theo qui tắc: nếu a, b là hai anh em ruột và a bên trái b thì các hậu duệ của a là "bên trái" mọi hậu duệ của b.

3. Các thứ tự duyệt cây quan trọng

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

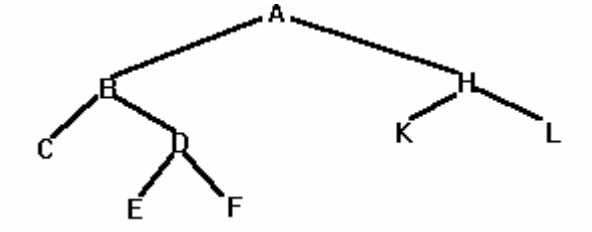


Hình III.3

- Cây rỗng thì danh sách duyệt cây là rỗng và nó được coi là biểu thức duyệt tiền tự, trung tự, hậu tự của cây.

- Cây chỉ có một nút thì danh sách duyệt cây gồm chỉ một nút đó và nó được coi là biểu thức duyệt tiền tự, trung tự, hậu tự của cây.
- Ngược lại: giả sử cây T có nút gốc là n và có các cây con là T_1,..,T_n thì:
  - ✧ Biểu thức duyệt tiền tự của cây T là liệt kê nút n kế tiếp là biểu thức duyệt tiền tự của các cây T_1, T_2, .., T_n theo thứ tự đó.
  - ✧ Biểu thức duyệt trung tự của cây T là biểu thức duyệt trung tự của cây T_1 kế tiếp là nút n rồi đến biểu thức duyệt trung tự của các cây T_2,.., T_n theo thứ tự đó.
  - ✧ Biểu thức duyệt hậu tự của cây T là biểu thức duyệt hậu tự của các cây T_1, T_2,.., T_n theo thứ tự đó rồi đến nút n.

Ví dụ cho cây như trong hình III.4



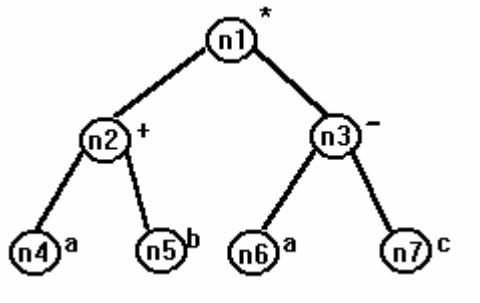
Hình III.4 Cây nhị phân

Biểu thức duyệt tiền tự: A B C D E F H K L
trung tự: C B E D F A K H L
hậu tự: C E F D B K L H A

4. Cây có nhãn và cây biểu thức

Ta thường lưu trữ kết hợp một nhãn (label) hoặc còn gọi là một giá trị (value) với một nút của cây. Như vậy nhãn của một nút không phải là tên nút mà là giá trị được lưu giữ tại nút đó. Nhãn của một nút đôi khi còn được gọi là khóa của nút, tuy nhiên hai khái niệm này là không đồng nhất. Nhãn là giá trị hay nội dung lưu trữ tại nút, còn khoá của nút có thể chỉ là một phần của nội dung lưu trữ này. Chẳng hạn, mỗi nút cây chứa một bản ghi (record/struct) về thông tin của sinh viên (mã SV, họ tên, ngày sinh, địa chỉ,...) thì khoá có thể là mã SV hoặc họ tên hoặc ngày sinh tuỳ theo giá trị nào ta đang quan tâm đến trong giải thuật.

Ví dụ: Cây biểu diễn biểu thức (a+b)*(a-c) như trong hình III.5.

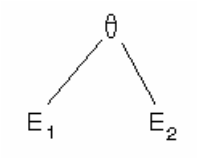


Hình III.5: Cây biểu diễn biểu thức (a+b)*(a-c)

Ở đây n_1, n_2,.., n_7 là các tên nút và *,+,-,a,b,c là các nhãn.

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

- Mỗi nút lá có nhãn biểu diễn cho một toán hạng.
- Mỗi nút trung gian biểu diễn một toán tử.



Hình III.6: Cây biểu diễn biểu thức E1θ E2

Giả sử nút n biểu diễn cho một toán tử hai ngôi θ (chẳng hạn + hoặc *), nút con bên trái biểu diễn cho biểu thức E_1, nút con bên phải biểu diễn cho biểu thức E_2 thì nút n biểu diễn biểu thức $E_1\theta\ E_2$, xem hình III.6. Nếu θ là phép toán một ngôi thì nút chứa phép toán θ chỉ có một nút con, nút con này biểu diễn cho toán hạng của θ.

Khi chúng ta duyệt một cây biểu diễn một biểu thức toán học và liệt kê nhãn của các nút theo thứ tự duyệt thì ta có:

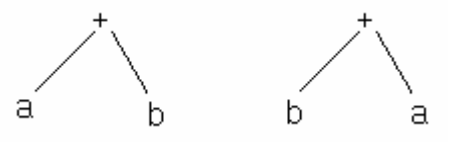
- *Biểu thức dạng tiền tố* hay biểu thức tiền tố (prefix) tương ứng với phép *duyệt tiền tự* của cây.
- *Biểu thức dạng trung tố* hay biểu thức trung tố (infix) tương ứng với phép *duyệt trung tự* của cây.
- *Biểu thức dạng hậu tố* hay biểu thức hậu tố (posfix) tương ứng với phép *duyệt hậu tự* của cây.

Ví dụ: đối với cây trong hình III.5 ta có:

- Biểu thức tiền tố: *+ab-ac
- Biểu thức trung tố: a+b*a-c
- Biểu thức hậu tố: ab+ac-*

Chú ý rằng

- Các biểu thức này không có dấu ngoặc.
- Các phép toán trong biểu thức toán học có thể có tính giao hoán nhưng khi ta biểu diễn biểu thức trên cây thì phải tuân thủ theo biểu thức đã cho. Ví dụ biểu thức a+b, với a,b là hai số nguyên thì rõ ràng a+b=b+a nhưng hai cây biểu diễn cho hai biểu thức này là khác nhau (vì cây có thứ tự).



Hình III.7 - Cây cho biểu thức a+b và b+a.

- Chỉ có cây ở phía bên trái của hình III.7 mới đúng là cây biểu diễn cho biểu thức a+b theo qui tắc trên.
- Nếu ta gặp một dãy các phép toán có cùng độ ưu tiên thì ta sẽ kết hợp từ trái sang phải. Ví dụ a+b+c-d = ((a+b)+c)-d.

II. KIỂU DỮ LIỆU TRỪU TƯỢNG CÂY

Để hoàn tất định nghĩa kiểu dữ liệu trừu tượng cây, cũng như đối với các kiểu dữ liệu trừu tượng khác, ta phải định nghĩa các phép toán trừu tượng cơ bản trên cây, các phép toán này được xem là các phép toán "nguyên thủy" để ta thiết kế các giải thuật sau này.

Các phép toán trên cây

- Hàm **PARENT(n,T)** cho nút cha của nút n trên cây T, nếu n là nút gốc thì hàm cho giá trị NULL. Trong cài đặt cụ thể thì NULL là một giá trị nào đó do ta chọn, nó phụ thuộc vào cấu trúc dữ liệu mà ta dùng để cài đặt cây.
- Hàm **LEFTMOST_CHILD(n,T)** cho nút con trái nhất của nút n trên cây T, nếu n là lá thì hàm cho giá trị NULL.
- Hàm **RIGHT_SIBLING(n,T)** cho nút anh em ruột phải nút n trên cây T, nếu n không có anh em ruột phải thì hàm cho giá trị NULL.
- Hàm **LABEL_NODE(n,T)** cho nhãn tại nút n của cây T.
- Hàm **ROOT(T)** trả ra nút gốc của cây T. Nếu Cây T rỗng thì hàm trả về NULL.
- Hàm **CREATEi(v,T1,T2,..,Ti)**,với i=0..n, thủ tục tạo cây mới có nút gốc là n được gán nhãn v và có i cây con T1,..,Ti. Nếu n= 0 thì thủ tục tạo cây mới chỉ gồm có 1 nút đơn độc là n có nhãn v. Chẳng hạn, giả sử ta có hai

cây con T1 và T2, ta muốn thiết lập cây mới với nút gốc có nhãn là v thì lời gọi thủ tục sẽ là CREATE2(v,T1,T2).

- Hàm **EMPTY_TREE**(T) trả về true nếu cây rỗng, ngược lại nó trả về false.

III. CÀI ĐẶT CÂY

1. Cài đặt cây bằng mảng

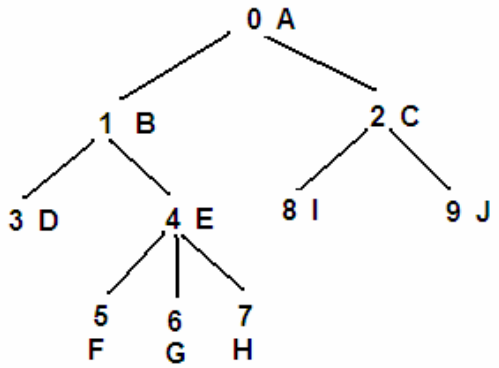
Cho cây T có n nút, ta có thể gán tên cho các nút lần lượt là 0,1, 2, .., n-1. Sau đó ta dùng một mảng một chiều A để lưu trữ cây bằng cách cho A[i] = j với j là nút cha của nút i. Nếu i là nút gốc ta cho a[i] = Null (Null có thể chọn là -1) vì nút gốc không có cha.

Nếu cây T là cây có nhãn ta có thể dùng thêm một mảng một chiều thứ hai L chứa các nhãn của cây bằng cách cho L[i] = x với x là nhãn của nút i, hoặc khai báo mảng a là mảng của các struct có hai trường: trường Parent giữ chỉ số nút cha; trường Data giữ nhãn của nút và một trường MaxNode giữ số nút hiện tại đang có trên cây.

Với cách lưu trữ như thế, hàm PARENT(n,T) tốn chỉ một hằng thời gian trong khi các hàm đòi hỏi thông tin về các con không làm việc tốt vì phải dò tìm các con trong mảng. Chẳng hạn cho một nút i tìm nút con trái nhất của nút i là không thể xác định được. Để cải thiện tình trạng này ta qui ước việc đặt tên cho các nút (đánh số thứ tự) như sau:

- Đánh số theo thứ tự tăng dần bắt đầu tại nút gốc.
- Nút cha được đánh số trước các nút con.
- Các nút con cùng một nút cha được đánh số lần lượt từ trái sang phải, chẳng hạn đánh số như cây trong hình III.8.

ví dụ:



Hình III.8 Hình ảnh một cây tổng quát

Cây trong hình III.8 được biểu diễn trong mảng như sau:

A	B	C	D	E	F	G	H	I	J	…	
-1	0	0	1	1	4	4	4	2	2	…	
0	1	2	3	4	5	6	7	8	9	…	

Nhãn của các nút trên cây

Cha của nút trên cây

Chỉ số của mảng

MaxNode Maxlength

Khai báo cấu trúc dữ liệu

```
#define MAXLENGTH ... /* chỉ số tối đa của mảng */
#define NULL -1
typedef ... DataType;
typedef int Node;
typedef struct {
        /* Lưu trữ nhãn (dữ liệu) của nút trong cây */
        DataType Data[MAXLENGTH];
        /* Lưu trữ cha của các nút trong cây theo nguyên tắc:
             Cha của nút i sẽ lưu ở vị trí i trong mảng */
        Node Parent[MAXLENGTH];
        /* Số nút thực sự trong cây */
        int MaxNode;
} Tree;
 Tree T;
```

Sự lưu trữ như vậy còn gọi là sự lưu trữ kế tiếp và cách lưu trữ cây như trên, ta có thể viết được các phép toán cơ bản trên cây như sau

Khởi tạo cây rỗng

```
void MAKENULL_TREE (Tree& T){
 T.MaxNode=0;
}
```

Kiểm tra cây rỗng

```
int EMPTY_TREE(Tree T){
 return T.MaxNode == 0;
}
```

Xác định nút cha của nút trên cây

```
Node PARENT(Node n,Tree T){
if (EMPTY_TREE(T) || (n>T.MaxNode-1))
   return NULL;
else return T.Parent[n];
}
```

Xác định nhãn của nút trên cây

```
DataType LABEL_NODE(Node n,Tree T){
if (!EMPTY_TREE(T) && (n<=T.MaxNode-1))
      return T.Data[n];
}
```

Hàm xác định nút gốc trong cây

```
Node ROOT(Tree T){
      if (!EMPTY_TREE(T)) return 0;
      else return NULL;
}
```

Hàm xác định con trái nhất của một nút

```
Node LEFTMOST_CHILD(Node n,Tree T){
Node i;
if (n<0) return NULL;
i=n+1;/* Vị trí nút đầu tiên hy vọng là con của nút n */
while (i<=T.MaxNode-1)
      if (T.Parent[i]==n) return i;
      else i=i+1;
return NULL;
}
```

Hàm xác định anh em ruột phải của một nút

```
Node RIGHT_SIBLING(Node n,Tree T){
Node i,parent;
if (n<0) return NULL;
parent=T.Parent[n];
i=n+1;
while (i<=T.MaxNode-1)
      if (T.Parent[i]==parent) return i;
      else i=i+1;
return NULL;
}
```

Thủ tục duyệt tiền tự

```
void PreOrder(Node n,Tree T){
    printf("%c ",LABEL_NODE(n,T));
    Node i=LEFTMOST_CHILD(n,T);
    while (i!=NULL){
      PreOrder(i,T);
      i=RIGHT_SIBLING(i,T);
    }
}
```

Thủ tục duyệt trung tự

```
void InOrder(Node n,Tree T){
    Node i=LEFTMOST_CHILD(n,T);
    if (i!=NULL) InOrder(i,T);
    printf("%c ",LABEL_NODE(n,T));
    i=RIGHT_SIBLING(i,T);
    while (i!=NULL){
        InOrder(i,T);
        i=RIGHT_SIBLING(i,T);
    }
}
```

Thủ tục duyệt hậu tự

```
void PostOrder(Node n,Tree T){
    Node i=LEFTMOST_CHILD(n,T);
    while (i!=NULL){
     PostOrder(i,T);
     i=RIGHT_SIBLING(i,T);}
    printf("%c ",LABEL_NODE(n,T));
}
```

Ví dụ: Viết chương trình nhập dữ liệu vào cho cây từ bàn phím như tổng số nút trên cây; ứng với từng nút thì phải nhập nhãn của nút, cha của một nút. Hiển thị danh sách duyệt cây theo các phương pháp duyệt tiền tự, trung tự, hậu tự của cây vừa nhập.

Hướng giải quyết: Với những yêu cầu đặt ra như trên, chúng ta cần phải thiết kế một số chương trình con sau:

- Tạo cây rỗng MAKENULL_TREE(T)
- Nhập dữ liệu cho cây từ bàn phím READTREE(T). Trong đó có nhiều cách nhập dữ liệu vào cho một cây nhưng ở đây ta có thể thiết kế thủ tục này đơn giản như sau:

```
void READTREE(Tree& T){
    int i;
    MAKENULL_TREE(T);
    //Nhập số nút của cây cho đến khi số nút nhập vào là hợp lệ
    do {
        printf("Cay co bao nhieu nut?");
        scanf("%d",&T.MaxNode);
    } while ((T.MaxNode<1) || (T.MaxNode>MAXLENGTH));
    printf("Nhap nhan cua nut goc ");
    fflush(stdin);
    scanf("%c",&T.Data[0]);
    T.Parent[0]=NULL; /* nut goc khong co cha */
    for (i=1;i<=T.MaxNode-1;i++){
        printf("Nhap cha cua nut %d ",i);
        scanf("%d",&T.Parent[i]);
        printf("Nhap nhan cua nut %d ",i);
        fflush(stdin);
```

```
            scanf("%c",&T.Data[i]);
        }
    }
```

- Hàm xác định con trái nhất của một nút LEFTMOST_CHILD(n,T). Hàm này được dùng trong phép duyệt cây.
- Hàm xác định anh em ruột phải của một nút RIGHT_SIBLING (n,T). Hàm này được dù trong phép duyệt cây.
- Các chương trình con hiển thị danh sách duyệt cây theo các phép duyệt.

Với những chương trình con được thiết kế như trên, ta có thể tạo một chương trình chính để thực hiện theo yêu cầu đề bài như sau:

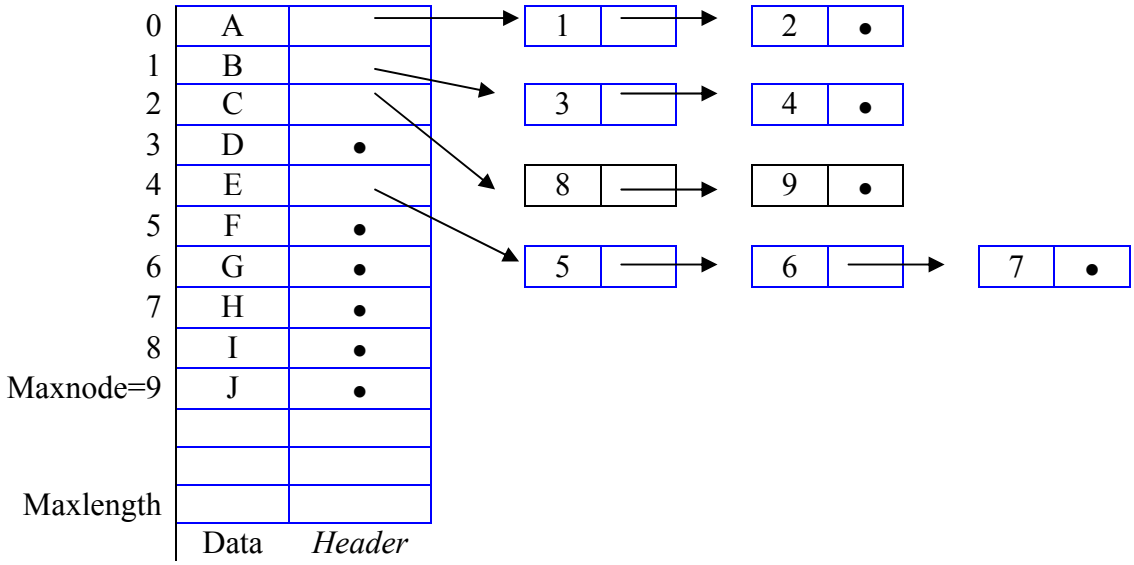
```
void main(){
    printf("Nhap du lieu cho cay tong quat\n");
    READTREE(T);
    printf("Danh sach duyet tien tu cua cay vua nhap la\n");
    PreOrder(ROOT(T),T);
    printf("\nDanh sach duyet trung tu cua cay vua nhap la\n");
    InOrder(ROOT(T),T);
    printf("\nDanh sach duyet hau tu cua cay vua nhap la\n");
    PostOrder(ROOT(T),T);
    getch();
}
```

Hàm ROOT chỉ đơn giản là trả về 0 (chỉ số mảng lưu nút gốc).

2. Biểu diễn cây bằng danh sách các con

Một cách biểu diễn khác cũng thường được dùng là biểu diễn cây dưới dạng mỗi nút có một danh sách các nút con. Danh sách có thể cài đặt bằng bất kỳ cách nào chúng ta đã biết, tuy nhiên vì số nút con của một nút là không biết trước nên dùng danh sách liên kết sẽ thích hợp hơn.

Ví dụ: Cây ở hình III.8 có thể lưu trữ dưới dạng như trong hình III.9



Hình III.9 Lưu trữ cây bằng danh sách các con

Có thể nhận xét rằng các hàm đòi hỏi thông tin về các con làm việc rất thuận lợi, nhưng hàm PARENT lại không làm việc tốt. Chẳng hạn tìm nút cha của nút 8 đòi hỏi ta phải duyệt tất cả các danh sách chứa các nút con.

3. Biểu diễn theo con trái nhất và anh em ruột phải

Các cấu trúc đã dùng để mô tả cây ở trên có một số nhược điểm, nó không trợ giúp phép tạo một cây lớn từ các cây nhỏ hơn, nghĩa là ta khó có thể cài đặt phép toán CREATEi bởi vì mỗi cây con đều có một mảng chứa các Header riêng. Chẳng hạn CREATE2(v,T1,T2) chúng ta phải chép hai cây T1, T2 vào mảng thứ ba rồi thêm một nút n có nhãn v và hai nút con là gốc của T1 và T2. Vì vậy nếu chúng ta muốn thiết lập một cấu trúc dữ liệu trợ giúp tốt cho phép toán này thì tất cả các nút của các cây con phải ở trong cùng một vùng (một mảng). Ta thay thế mảng các Header bằng mảng CELLSPACE chứa các bản ghi (struct) có ba trường Data, Leftmost_Child, Right_Sibling. Trong đó Data giữ nhãn của nút, Leftmost_Child là một con nháy chỉ đến con trái nhất của nút, còn Right_Sibling là con nháy chỉ đến nút anh ruột phải. Hơn nữa mảng này giữ tất cả các nút của tất cả các cây.

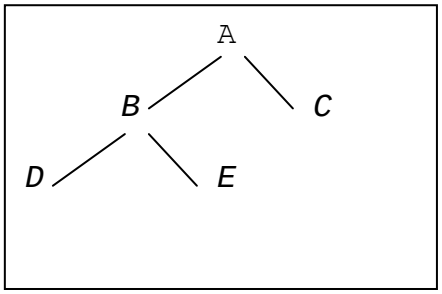
Các khai báo cần thiết là

```
#define MaxLength ...
#define NULL -1
typedef ... DataType;
typedef struct {
    DataType Data;
    int LeftMost_Child;
    int Right_Sibling;
} Node;
Node CELLSPACE [MaxLength];
```

Với cấu trúc này các phép toán đều thực hiện dễ dàng trừ PARENT, PARENT đòi hỏi phải duyệt toàn bộ mảng. Nếu chúng ta muốn cải tiến cấu trúc để trợ giúp phép toán này ta có thể thêm trường thứ 4 Parent là một con nháy chỉ tới nút cha (xem hình III.11). Các khai báo cần thiết là :

```
#define NULL -1
#define MaxNode …
typedef … DataType;
typedef struct {
    DataType Data;
    int LeftMost_Child;
    int Right_Sibling;
    int Parent;
} Node;
Node CELLSPACE [MaxLength];
```

Để cài đặt cây theo cách này chúng ta cũng cần quản lí các ô trống theo cách tương tự như cài đặt danh sách bằng con nháy, tức là liên kết các ô trống vào một danh sách có chỉ điểm đầu là Available. Ở đây mỗi ô chứa 3 con nháy nên ta chỉ cần chọn 1 để trỏ đến ô kế tiếp trong danh sách, chẳng hạn ta chọn con nháy Right_Sibling. Ví dụ cây trong hình III.10 có thể được cài đặt như trong hình III.11. Các ô được tô đậm là các ô trống, tức là các ô nằm trong danh sách Available.



Hình III.10 Hình ảnh cây tổng quát

	Chỉ số	Data	Leftmost_Child	Right_Sibling	Parent
	0			null	
	1	D	Null	4	3
Available →	2			8	
	3	B	1	7	5
	4	E	Null	null	3
Root →	5	A	3	null	Null
	6			0	
	7	C	Null	null	3
	8			6	

Hình III.11

Hàm CREATE2 tạo cây mới từ hai cây con có thể viết như sau:

```
int CREATE2(DataType v, int T1, int T2){
        // T1, T2 là hai con nháy trỏ tới hai nút gốc của hai cây con
        // Hàm trả ra con nháy trỏ tới nút gốc mới

        //lấy ô đầu danh sách available để tạo một nút mới trên cây
        int temp=Available ; //available như là một biến toàn cục
        Available = CELLSPACE[Available].Right_Sibling;

        //cập nhật các giá trị cho nút mới
        CELLSPACE[temp].LeftMost_Child=T1;
        CELLSPACE[temp].Labels=v;
        CELLSPACE[temp].Right_Sibling=NULL;

        // cập nhật lại parent cho T1 và T2
        if (T1!=NULL){
                CELLSPACE[T1].Right_Sibling=T2;
                CELLSPACE[T1].Parent = temp;
        }

        if (T2 !=NULL)
        CELLSPACE[T2].Parent = temp;

        //trả ra giá trị nút gốc của cây mới
        return temp;
}
```

Hàm này chỉ mất một hằng thời gian để tạo cây mới.

4. Cài đặt cây bằng con trỏ

Hoàn toàn tương tự như cài đặt ở trên nhưng các con nháy Leftmost_Child, Right_Sibling và Parent được thay bằng các con trỏ.
Các khai báo như sau:

```
typedef int DataType;
typedef struct Cell{
        DataType Data;
        Cell* Leftmost_Child;
        Cell* Right_Sibling;
        Cell* Parent;
};

typedef Cell* Node;
typedef Node Tree;
```

Với cấu trúc như vậy hàm Create2 có thể viết như sau:

```
Node Create2(DataType v, Tree left, Tree right){
        Node n;
        n = (Node) malloc(sizeof(Cell));
        n->value=v;
        n->Leftmost_Child=left;
        n->Right_Sibling=right;
        n->Parent= NULL;
        if (left!=NULL)  left->Parent = n;
        if (right!=NULL) right->Parent=n;
        return  n;
}
```

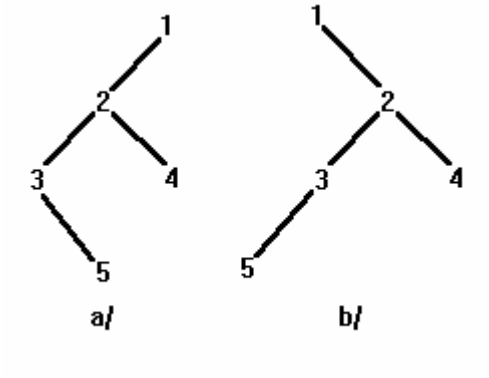
Hàm này cũng chỉ tốn một hằng thời gian để tạo cây mới.

? Hãy so sách các ưu khuyết điểm của các cách cài đặt cây.

IV. CÂY NHỊ PHÂN (BINARY TREES)

1. Định nghĩa

Cây nhị phân là cây rỗng hoặc là cây mà mỗi nút có tối đa hai nút con. Hơn nữa các nút con của cây được phân biệt thứ tự rõ ràng, một nút con gọi là nút con trái và một nút con gọi là nút con phải. Ta qui ước vẽ nút con trái bên trái nút cha và nút con phải bên phải nút cha, mỗi nút con được nối với nút cha của nó bởi một đoạn thẳng. Ví dụ các cây trong hình III.12.



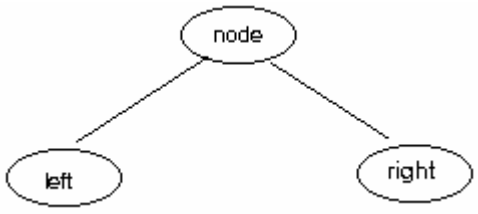
Hình III.12: Hai cây có thứ tự giống nhau nhưng là hai cây nhị phân khác nhau

Chú ý rằng, trong cây nhị phân, một nút con chỉ có thể là nút con trái hoặc nút con phải, nên có những cây có thứ tự giống nhau nhưng là hai cây nhị phân khác nhau. Ví dụ hình III.12 cho thấy hai cây có thứ tự giống nhau nhưng là hai cây nhị phân khác nhau. Nút 2 là nút con trái của cây a/ nhưng nó là con phải trong cây b/. Tương tự nút 5 là con phải trong cây a/ nhưng nó là con trái trong cây b/.

2. Duyệt cây nhị phân

Ta có thể áp dụng các phép duyệt cây tổng quát để duyệt cây nhị phân. Tuy nhiên vì cây nhị phân là cấu trúc cây đặc biệt nên các phép duyệt cây nhị phân cũng đơn giản hơn. Có ba cách duyệt cây nhị phân thường dùng (xem kết hợp với hình III.13):

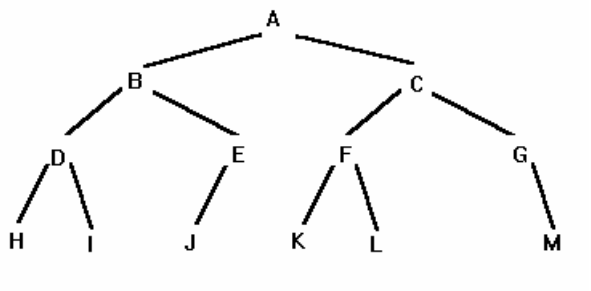
- ***Duyệt tiền tự (Node-Left-Right):*** duyệt nút gốc, duyệt tiền tự con trái rồi duyệt tiền tự con phải.
- ***Duyệt trung tự (Left-Node-Right):*** duyệt trung tự con trái rồi đến nút gốc sau đó là duyệt trung tự con phải.
- ***Duyệt hậu tự (Left-Right-Node)***: duyệt hậu tự con trái rồi duyệt hậu tự con phải sau đó là nút gốc.



Hình III.13

Chú ý rằng danh sách duyệt tiền tự, hậu tự của cây nhị phân trùng với danh sách duyệt tiền tự, hậu tự của cây đó khi ta áp dụng phép duyệt cây tổng quát. Nhưng danh sách duyệt trung tự thì khác nhau.

Ví dụ



Hình III.14

	Các danh sách duyệt cây nhị phân	Các danh sách duyệt cây tổng quát
Tiền tự:	ABDHIEJCFKLGM	ABDHIEJCFKLGM
Trung tự:	HDIBJEAKFLC**GM**	HDIBJEAKFLC**MG**
Hậu tự:	HIDJEBKLFMGCA	HIDJEBKLFMGCA

?

1. Danh sách duyệt tiền tự và hậu tự của cây nhị phân luôn luôn giống với danh sách duyệt của cây tổng quát?
2. Danh sách duyệt trung tự của cây nhị phân sẽ khác với các duyệt tổng quát chỉ khi cây nhị phân bị khuyết con trái?

3. Cài đặt cây nhị phân

Tương tự cây tổng quát, ta cũng có thể cài đặt cây nhị phân bằng con trỏ bằng cách thiết kế mỗi nút có hai con trỏ, một con trỏ trỏ nút con trái, một con trỏ trỏ nút con phải, trường Data sẽ chứa nhãn của nút.

```
typedef int DataType;
typedef struct Node{
            DataType Data;
            Node* left;
            Node* right;
 };
typedef Node* Tree;
```

Với cách khai báo như trên ta có thể thiết kế các phép toán cơ bản trên cây nhị phân như sau :

Tạo cây rỗng

Cây rỗng là một cây là không chứa một nút nào cả. Như vậy khi tạo cây rỗng ta chỉ cần cho cây trỏ tới giá trị NULL.

```
void MAKENULL_TREE(Tree& T){
```

```
    T=NULL;
}
```

Kiểm tra cây rỗng

```
int EMPTY_TREE(Tree T){
    return T==NULL;
}
```

Xác định con trái của một nút

```
Tree LEFTCHILD(Tree n){ //n co kieu Node*, tuc la Tree
    if (n!=NULL) return n->left;
    else return NULL;
}
```

Xác định con phải của một nút

```
Tree RIGHTCHILD(Tree n){
    if (n!=NULL) return n->right;
    else return NULL;
}
```

Kiểm tra nút lá

Nếu nút là nút lá thì nó không có bất kỳ một con nào cả nên khi đó con trái và con phải của nó cùng bằng NULL.

```
int ISLEAF(Tree n){
    if(n!=NULL)
        return(LEFTCHILD(n)==NULL)&&(RIGHTCHILD(n)==NULL);
    else return 0;
}
```

Xác định số nút của cây

```
int NB_NODES(Tree T){
    if(EMPTY_TREE(T)) return 0;
    else return 1+NB_NODES(LEFTCHILD(T)) + NB_NODES(RIGHTCHILD(T));
}
```

Tạo cây mới từ hai cây có sẵn

```
Tree Create2(DataType v,Tree l,Tree r){
Tree N;
N=(Node*)malloc(sizeof(Node));
N->Data=v;
N->left=l;
N->right=r;
return N;
}
```

Các thủ tục duyệt cây: tiền tự, trung tự, hậu tự

Thủ tục duyệt tiền tự

```
void PreOrder(Tree T){
        printf("%c ",T->Data);
  if (LEFTCHILD(T)!=NULL) PreOrder(LEFTCHILD(T));
  if (RIGHTCHILD(T)!=NULL)PreOrder(RIGHTCHILD(T));
}
```

Thủ tục duyệt trung tự

```
void InOrder(Tree T){
       if (LEFTCHILD(T)!=NULL)InOrder(LEFTCHILD(T));
  printf("%c ",T->Data);
  if (RIGHTCHILD(T)!=NULL) InOrder(RIGHTCHILD(T));
}
```

Thủ tục duyệt hậu tự

```
void PosOrder(Tree T){
       if (LEFTCHILD(T)!=NULL) PosOrder(LEFTCHILD(T));
  if (RIGHTCHILD(T)!=NULL)PosOrder(RIGHTCHILD(T));
  printf("%c ",T->Data);
}
```

? Hãy biểu diễn cách gọi hàm Create2 để tạo một cây nhị phân cho trước.

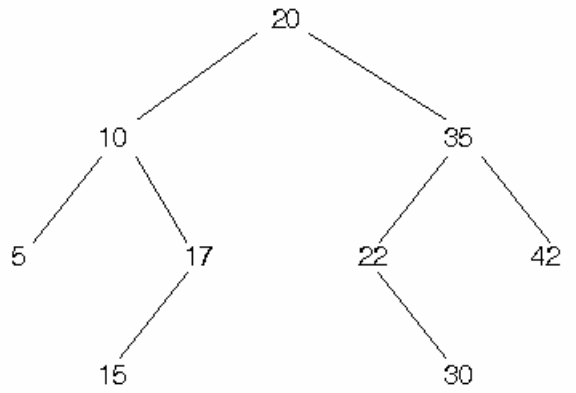
V. CÂY TÌM KIẾM NHỊ PHÂN (BINARY SEARCH TREES)

1. Định nghĩa

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

Lưu ý: dữ liệu lưu trữ tại mỗi nút có thể rất phức tạp như là một bản ghi (record/struct) chẳng hạn, trong trường hợp này khoá của nút được tính dựa trên một trường nào đó, ta gọi là trường khoá. Trường khoá phải chứa các giá trị có thể so sánh được, tức là nó phải lấy giá trị từ một tập hợp có thứ tự.

Ví dụ: hình III.15 minh hoạ một cây TKNP có khoá là số nguyên (với quan hệ thứ tự trong tập số nguyên).



Hình III.15: Ví dụ cây tìm kiếm nhị phân

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

Nhận xét:

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

2. Cài đặt cây tìm kiếm nhị phân

Cây TKNP, trước hết, là một cây nhị phân. Do đó ta có thể áp dụng các cách cài đặt như đã trình bày trong phần cây nhị phân. Sẽ không có sự khác biệt nào trong việc cài đặt cấu trúc dữ liệu cho cây TKNP so với cây nhị phân, nhưng tất nhiên, sẽ có sự khác biệt trong các giải thuật thao tác trên cây TKNP như tìm kiếm, thêm hoặc xoá một nút trên cây TKNP để luôn đảm bảo tính chất của cây TKNP.

Một cách cài đặt cây TKNP thường gặp là cài đặt bằng con trỏ. Mỗi nút của cây như là một mẩu tin (struct) có ba trường: một trường chứa khoá, hai trường kia là hai con trỏ trỏ đến hai nút con (nếu nút con vắng mặt ta gán con trỏ bằng NULL)

Khai báo như sau

```
typedef <kiểu dữ liệu của khoá> KeyType;
typedef struct Node{
        KeyType Key;
        Node* Left;
        Node* Right;
};
typedef Node* Tree;
```

Khởi tạo cây TKNP rỗng

Ta cho con trỏ quản lý nút gốc (Root) của cây bằng NULL.

```
void MAKENULL_TREE(Tree& Root){
 Root=NULL;
 }
```

Tìm kiếm một nút có khóa cho trước trên cây TKNP

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

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

Ví dụ: tìm nút có khoá 30 trong cây ở trong hình III.15

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

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

```
Tree Search(KeyType x,Tree Root){
    if (Root == NULL) return NULL;   //không tìm thấy khoá x
    else if (Root->Key == x) /* tìm thấy khoá x */
            return Root;
            else    if (Root->Key < x) //tìm tiếp trên cây bên phải
                    return Search(x,Root->Right);
            else    //tìm tiếp trên cây bên trái
                    return Search(x,Root->Left);
}
```

? Cây tìm kiếm nhị phân được tổ chức như thế nào để quá trình tìm kiếm được hiệu quả nhất?

Nhận xét: *giải thuật này sẽ rất hiệu quả về mặt thời gian nếu cây TKNP được tổ chức tốt, nghĩa là cây tương đối "cân bằng". Về chủ đề cây cân bằng các bạn có thể tham khảo thêm trong các tài liệu tham khảo của môn này.*

Thêm một nút có khóa cho trước vào cây TKNP

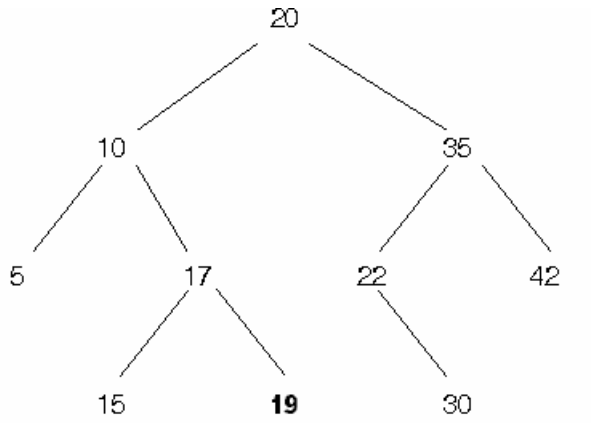
Theo định nghĩa cây tìm kiếm nhị phân ta thấy trên cây tìm kiếm nhị phân không có hai nút có cùng một khoá. Do đó nếu ta muốn thêm một nút có khoá x vào cây TKNP thì trước hết ta phải tìm kiếm để xác định có nút nào chứa khoá x chưa. Nếu có thì giải thuật kết thúc (không làm gì cả!). Ngược lại, sẽ thêm một nút mới chứa khoá x này. Việc thêm một khoá vào cây TKNP là việc tìm kiếm và thêm một nút, tất nhiên, phải đảm bảo cấu trúc cây TKNP không bị phá vỡ. Giải thuật cụ thể như sau:

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

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

Ví dụ: thêm khoá 19 vào cây ở trong hình III.15

- So sánh 19 với khoá của nút gốc là 20, vì 19 < 20 vậy ta xét tiếp đến cây bên trái, tức là cây có nút gốc có khoá là10.
- So sánh 19 với khoá của nút gốc là 10, vì 19 > 10 vậy ta xét tiếp đến cây bên phải, tức là cây có nút gốc có khoá là 17.
- So sánh 19 với khoá của nút gốc là 17, vì 19 > 17 vậy ta xét tiếp đến cây bên phải. Nút con bên phải bằng NULL, chứng tỏ rằng khoá 19 chưa có trên cây, ta thêm nút mới chứa khoá 19 và nút mới này là con bên phải của nút có khoá là 17, xem hình III.16



Hình III.16: Thêm khoá 19 vào cây hình III.15

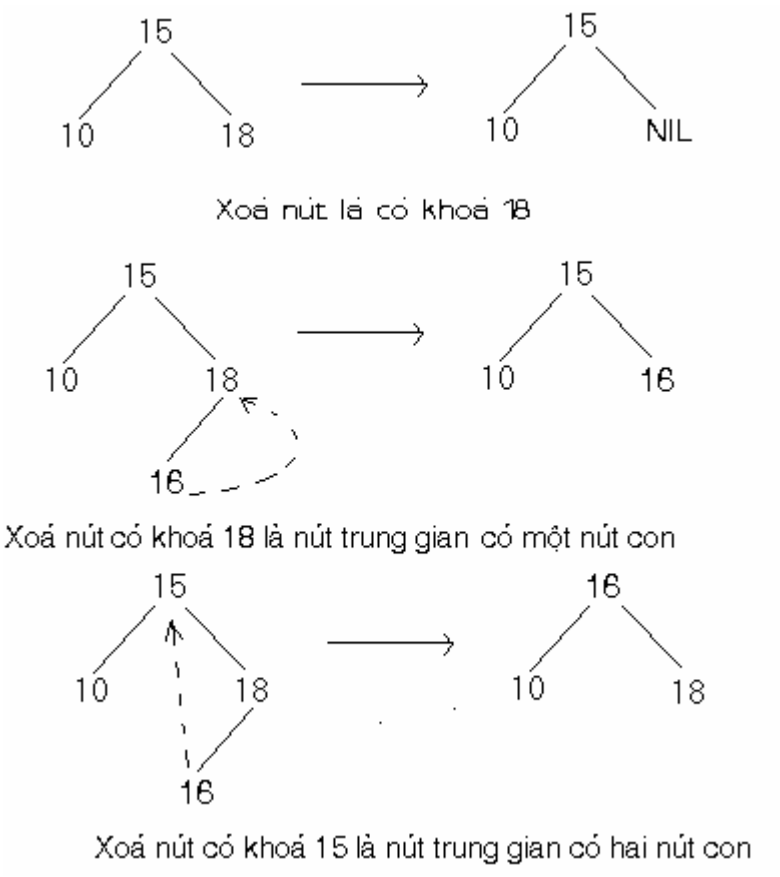
Thủ tục sau đây tiến hành việc thêm một khoá vào cây TKNP.

```
void InsertNode(KeyType x,Tree& Root ){
 if (Root == NULL){ /* thêm nút mới chứa khoá x */
            Root=(Node*)malloc(sizeof(Node));
            Root->Key = x;
            Root->Left = NULL;
            Root->Right = NULL;
      }
 else
 if (x < Root->Key) InsertNode(x,Root->Left);
      else if (x>Root->Key)InsertNode(x,Root->Right);
}
```

Xóa một nút có khóa cho trước ra khỏi cây TKNP

Giả sử ta muốn xoá một nút có khoá x, trước hết ta phải tìm kiếm nút chứa khoá x trên cây. Việc xoá một nút như vậy, tất nhiên, ta phải bảo đảm cấu trúc cây TKNP không bị phá vỡ. Ta có các trường hợp như hình III.17:

- Nếu không tìm thấy nút chứa khoá x thì giải thuật kết thúc.
- Nếu tìm gặp nút N có chứa khoá x, ta có ba trường hợp sau (xem hình III.17)
  - Nếu N là lá ta thay nó bởi NULL.
  - N chỉ có một nút con ta thay nó bởi nút con của nó.
  - N có hai nút con ta thay nó bởi nút lớn nhất trên cây con trái của nó (nút cực phải của cây con trái) hoặc là nút bé nhất trên cây con phải của nó (nút cực trái của cây con phải). Trong giải thuật sau, ta thay x bởi khoá của nút cực trái của cây con bên phải rồi ta xoá nút cực trái này. Việc xoá nút cực trái của cây con bên phải sẽ rơi vào một trong hai trường hợp trên.



Hình III.17 Ví dụ về giải thuật xóa nút trên cây

Giải thuật xoá một nút có khoá nhỏ nhất

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

```
KeyType DeleteMin (Tree& Root ){
    KeyType k;
    if (Root->Left == NULL){
            k=Root->Key;
            Root = Root->Right;
            return k;
    }
    else return DeleteMin(Root->Left);
}
```

Thủ tục xóa một nút có khoá cho trước trên cây TKNP

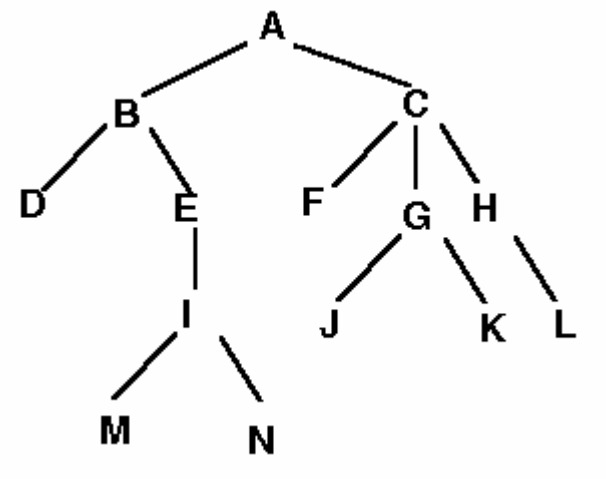
```
void DeleteNode(KeyType x,Tree& Root){
if (Root!=NULL)
        if (x < Root->Key) DeleteNode(x,Root->Left);
        else    if (x > Root->Key) DeleteNode(x,Root->Right);
                else
                        if ((Root->Left==NULL) && (Root->Right==NULL))
                                Root=NULL;
                        else
                                if (Root->Left == NULL) Root = Root->Right;
                                else    if (Root->Right==NULL) Root = Root->Left;
                                        else Root->Key = DeleteMin(Root->Right);
}
```

TỔNG KẾT CHƯƠNG

Chương này giới thiệu một số khái niệm cơ bản về cây tổng quát, cây nhị phân, cách biểu diễn biểu thức toán học, các biểu thức duyệt cây. Chương này cũng đề cập đến cách lưu trữ cây trong bộ nhớ như cài đặt cây bằng mảng, con trỏ, danh sách các con, con trái nhất, anh em ruột phải và cách cài đặt các phép toán cơ bản trên các dạng cây khác nhau theo từng cách cài đặt. Cuối chương trình bày cấu trúc cây tìm kiếm nhị phân như một ứng dụng cây nhị phân trong tổ chức dữ liệu phục vụ cho tìm kiếm nhanh.

BÀI TẬP

1. Trình bày các biểu thức duyệt tiền tự, trung tự, hậu tự của cây sau:



2. Duyệt cây theo mức là duyệt bắt đầu từ gốc, rồi duyệt các nút nằm trên mức 1 theo thứ tự từ trái sang phải, rồi đến các nút nằm trên mức 2 theo thứ tự từ trái sang phải...Và cứ như vậy.

a. Hãy liệt kê các nút theo thứ tự duyệt theo mức của cây trong bài 1.

b. Viết thủ tục duyệt cây theo mức. (Gợi ý: dùng hàng đợi)

3. Vẽ cây biểu diễn cho biểu thức ((a+b)+c*(d+e)+f)*(g+h)

Trình bày biểu thức tiền tố và hậu tố của biểu thức đã cho.

4. Viết chương trình để tính giá trị của biểu thức khi cho:

a. Biểu thức tiền tố

b. Biểu thức hậu tố.

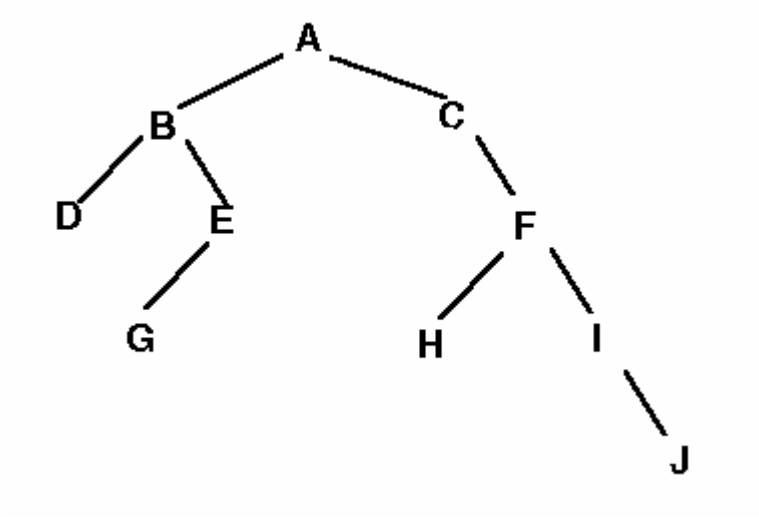
Ví dụ:

- đầu vào (input): * + 6 4 5
- thì đầu ra (output) sẽ là: 50 vì biểu thức trên là dạng tiền tố của (6+4) * 5

Tương tự:

- đầu vào (input): 6 4 5 + *
- thì đầu ra (output) sẽ là: 54 vì biểu thức trên là dạng hậu tố của 6 * (4+5)

5. Cho cây nhị phân



a. Hãy trình bày các duyệt: tiền tự (node-left-right), trung tự (left-node-right), hậu tự (left-right-node).

b. Minh hoạ sự lưu trữ kế tiếp các nút cây này trong mảng.

6. Chứng minh rằng: nếu biết biểu thức duyệt tiền tự và trung tự của một cây nhị phân thì ta dựng được cây này.

Điều đó đúng nữa không? Khi biết biểu thức duyệt:

a. Tiền tự và hậu tự

b. Trung tự và hậu tự

7. Nêu các trường hợp mà các giải thuật trên cây TKNP:

- Có hiệu quả nhất
- Kém hiệu quả nhất

Từ đó nêu ra các hướng tổ chức cây TKNP để đạt được hiệu quả cao về thời gian thực hiện giải thuật.

8. a- Vẽ hình cây tìm kiếm nhị phân tạo ra từ cây rỗng bằng cách lần lượt thêm vào các khoá là các số nguyên: 54, 31, 43, 29, 65, 10, 20, 36, 78, 59.

b- Vẽ lại hình cây tìm kiếm nhị phân ở câu a/ sau khi lần lượt xen thêm các nút 15, 45, 55.

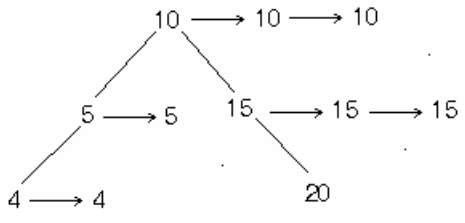
c- Vẽ lại hình cây tìm kiếm nhị phân ở câu a/ sau khi lần lượt xoá các nút 10, 20, 43, 65, 54.

9. Hãy dựng cây tìm kiếm nhị phân ứng với dãy khóa (thứ tự tính theo qui tắc so sánh chuỗi (string)): HAIPHONG, CANTHO, NHATRANG, DALAT, HANOI, ANGIANG, MINHHAI, HUE, SAIGON, VINHLONG. Đánh dấu đường đi trên cây khi tìm kiếm khóa DONGTHAP.

10. Cài đặt cây TKNP có khoá là chuỗi (String) với các phép toán thêm, xoá. Bổ sung thêm các thủ tục cần thiết đề có 1 chương trình hoàn chỉnh, cung cấp giao diện để người dùng có thể thêm, xoá 1 khoá và duyệt cây để kiểm tra kết quả.

11. Viết các thủ tục thêm, xoá một nút có khoá x trên cây tìm kiếm nhị phân bằng cách không đệ qui.

12. Để mở rộng khả năng xử lí các khoá trùng nhau trên cây tìm kiếm nhị phân, ta có thể tổ chức cây tìm kiếm nhị phân như sau: tại mỗi nút của cây ta tổ chức một danh sách liên kết chứa các khoá trùng nhau đó. Chẳng hạn cây được thiết lập từ dãy khoá số nguyên 10, 15, 5, 10, 20, 4, 5, 10, 15, 15, 4, 15 như sau



Trong đó các mũi tên nằm ngang chỉ các con trỏ của danh sách liên kết.

Hãy viết khai báo cấu trúc dữ liệu và các thủ tục/hàm để cài đặt cây TKNP mở rộng như trên.

CHƯƠNG IV TẬP HỢP

I. KHÁI NIỆM TẬP HỢP

Tập hợp là một khái niệm cơ bản trong toán học. Tập hợp được dùng để mô hình hoá hay biểu diễn một nhóm bất kỳ các đối tượng trong thế giới thực cho nên nó đóng vai trò rất quan trọng trong mô hình hoá cũng như trong thiết kế các giải thuật.

Khái niệm tập hợp cũng như trong toán học, đó là sự tập hợp các thành viên (members) hoặc phần tử (elements). Tất cả các phần tử của tập hợp là khác nhau. Tập hợp có thể có thứ tự hoặc không có thứ tự, tức là, có thể có quan hệ thứ tự xác định trên các phần tử của tập hợp hoặc không. Tuy nhiên, trong chương này, chúng ta giả sử rằng các phần tử của tập hợp có thứ tự tuyến tính, tức là, các phần tử thuộc một tập hợp S có quan hệ < và = thoả mản hai tính chất:

- Với mọi a,b $\in$ S thì a<b hoặc b<a hoặc a=b
- Với mọi a,b,c $\in$ S, a<b và b<c thì a<c

II. KIỂU DỮ LIỆU TRỪU TƯỢNG TẬP HỢP

Cũng như các kiểu dữ liệu trừu tượng khác, các phép toán kết hợp với mô hình tập hợp sẽ tạo thành một kiểu dữ liệu trừu tượng là rất đa dạng. Tùy theo nhu cầu của các ứng dụng mà các phép toán khác nhau sẽ được định nghĩa trên tập hợp. Ở đây ta đề cập đến một số phép toán thường gặp nhất như sau

- Thủ tục **UNION(A,B,C)** nhận vào 3 tham số là A,B,C; Thực hiện phép toán lấy hợp của hai tập A và B và trả ra kết quả là tập hợp C = A $\cup$B.
- Thủ tục **INTERSECTION(A,B,C)** nhận vào 3 tham số là A,B,C; Thực hiện phép toán lấy giao của hai tập A và B và trả ra kết quả là tập hợp C = A $\cap$ B.
- Thủ tục **DIFFERENCE(A,B,C)** nhận vào 3 tham số là A,B,C; Thực hiện phép toán lấy hợp của hai tập A và B và trả ra kết quả là tập hợp C = A\B
- Hàm **MEMBER(x,A)** cho kết quả kiểu logic (đúng/sai) tùy theo x có thuộc A hay không. Nếu x $\in$ A thì hàm cho kết quả là 1 (đúng), ngược lại cho kết quả 0 (sai).
- Thủ tục **MAKENULL_SET(A)** tạo tập hợp A tập rỗng
- Thủ tục **INSERT_SET(x,A)** thêm x vào tập hợp A
- Thủ tục **DELETE_SET(x,A)** xoá x khỏi tập hợp A
- Thủ tục **ASSIGN(A,B)** gán A cho B (tức là B:=A)
- Hàm **MIN(A)** cho phần tử bé nhất trong tập A
- Hàm **EQUAL(A,B)** cho kết quả TRUE nếu A=B ngược lại cho kết quả FALSE

III. CÀI ĐẶT TẬP HỢP

1. Cài đặt tập hợp bằng vector Bit

Hiệu quả của một cách cài đặt tập hợp cụ thể phụ thuộc vào các phép toán và kích thước tập hợp. Hiệu quả này cũng sẽ phụ thuộc vào tần suất sử dụng các phép toán trên tập hợp. Chẳng hạn nếu chúng ta thường xuyên sử dụng phép thêm vào và loại bỏ các phần tử trong tập hợp thì chúng ta sẽ tìm cách cài đặt hiệu quả cho các phép toán này. Còn nếu phép tìm kiếm một phần tử xảy ra thường xuyên thì ta có thể phải tìm cách cài đặt phù hợp để có hiệu quả tốt nhất.

Ở đây ta xét một trường hợp đơn giản là khi toàn thể tập hợp của chúng ta là tập hợp con của một tập hợp các số nguyên nằm trong phạm vi nhỏ từ 1.. n chẳng hạn thì chúng ta có thể dùng một mảng kiểu Boolean có n phần tử để cài đặt tập hợp (ta gọi là vectơ bít), bằng cách cho phần tử thứ i của mảng này giá trị TRUE nếu i thuộc tập hợp hoặc cho mảng lưu kiểu 0-1. Nếu nội dung phần tử trong mảng tại vị trí i là 1 nghĩa là i tồn tại trong tập hợp và ngược lại, nội dung là 0 nghĩa là phần tử i đó không tồn tại trong tập hợp.

Ví dụ: Giả sử các phần tử của tập hợp được lấy trong các số nguyên từ 1 đến 10 thì mỗi tập hợp được biểu diễn bởi một mảng một chiều có 10 phần tử với các giá trị phần tử thuộc kiểu logic. Chẳng hạn tập hợp A={1,3,5,8} được biểu diễn trong mảng có 10 phần tử như sau:

1	2	3	4	5	6	7	8	9	10
1	0	1	0	1	0	0	1	0	0

Cách biểu diễn này chỉ thích hợp trong điều kiện là mọi thành viên của tất cả các tập hợp đang xét phải có giá trị nguyên hoặc có thể đặt tương ứng duy nhất với số nguyên nằm trong một phạm vi nhỏ. Có thể dễ dàng nhận thấy khai báo cài đặt như sau

```
const maxlength = 100;
// giá trị phần tử lớn nhất trong tập hợp số nguyên không âm
typedef int SET [maxlength];
```

Tạo một tập hợp rỗng

Để tạo một tập hợp rỗng ta cần đặt tất cả các nội dung trong tập hợp từ vị trí 0 đến vị trí maxlength-1 đều bằng 0. Câu lệnh được viết như sau :

```
void MAKENULL_SET(SET a){
 int i;
 for(i=0;i<maxlength;i++)
   a[i]=0;
}
```

Biểu diễn tập hợp bằng vectơ bít tạo điều kiện thuận lợi cho các phép toán trên tập hợp. Các phép toán này có thể cài đặt dễ dàng bằng các phép toán Logic trong ngôn

ngữ lập trình. Chẳng hạn thủ tục UNION(A,B,C) và thủ tục INTERSECTION được viết như sau :

```
void UNION (SET a,SET b,SET c)
{
 int i;
 for (i=0;i<maxlength;i++)
  if ((a[i]==1)||(b[i]==1)) c[i]=1;
  else c[i]=0;
}
void INTERSECTION (SET a,SET b, SET c)
{
 int i;
 for (i=0;i<maxlength;i++)
  if ((a[i]==1)&&(b[i]==1)) c[i]=1;
  else c[i]=0;
}
```

Các phép toán giao, hiệu,... được viết một cách tương tự. Việc kiểm tra một phần tử có thuộc tập hợp hay không, thủ tục thêm một phần tử vào tập hợp, xóa một phần tử ra khỏi tập hợp cũng rất đơn giản và xem như bài tập.

2. Cài đặt bằng danh sách liên kết

Tập hợp cũng có thể cài đặt bằng danh sách liên kết, trong đó mỗi phần tử của danh sách là một thành viên của tập hợp. Không như biểu diễn bằng vectơ bít, sự biểu diễn này dùng kích thước bộ nhớ tỉ lệ với số phần tử của tập hợp chứ không phải là kích thước đủ lớn cho toàn thể các tập hợp đang xét. Hơn nữa, ta có thể biểu diễn một tập hợp bất kỳ. Mặc dù thứ tự của các phần tử trong tập hợp là không quan trọng nhưng nếu một danh sách liên kết có thứ tự nó có thể trợ giúp tốt cho các phép duyệt danh sách. Chẳng hạn nếu tập hợp A được biểu diễn bằng một danh sách có thứ tự tăng thì hàm MEMBER(x,A) có thể thực hiện việc so sánh x một cách tuần tự từ đầu danh sách cho đến khi gặp một phần tử $y \geq x$ chứ không cần so sánh với tất cả các phần tử trong tập hợp.

Một ví dụ khác, chẳng hạn ta muốn tìm giao của hai tập hợp A và B có n phần tử. Nếu A,B biểu diễn bằng các danh sách liên kết chưa có thứ tự thì để tìm giao của A và B ta phải tiến hành như sau:

```
for (mỗi x thuộc A ) {
    Duyệt danh sách B xem x có thuộc B không. Nếu có thì x thuộc giao của hai tập hợp A và B;
}
```

Rõ ràng quá trình này có thể phải cần đến n x m phép kiểm tra (với n,m là độ dài của A và B).

Nếu A,B được biểu diễn bằng danh sách có thứ tự tăng thì đối với một phần tử $e \in A$ ta chỉ tìm kiếm trong B cho đến khi gặp phần tử $x \geq e$. Quan trọng hơn nếu f đứng ngay sau e trong A thì để tìm kiếm f trong B ta chỉ cần tìm từ phần tử x trở đi chứ không phải từ đầu danh sách lưu trữ tập hợp B.

Khai báo

```
typedef … ElementType;
typedef struct Cell {
      ElementType element;
      Cell* next;
};
typedef Cell* SET;
```

Thủ tục INTERSECTION(A,B,C) trong trường hợp cài tập hợp đặt bằng danh sách liên kết có thứ tự tăng

```
void INTERSECTION( SET Aheader, SET Bheader, SET C){
   SET  Acurrent, Bcurrent, Ccurrent;
   C = (SET)malloc(sizeof(Cell));
   Acurrent=Aheader->next;
   Bcurrent=Bheader->next;
   Ccurrent=C;
   while ((Acurrent!=NULL) && (Bcurrent!=NULL)) {
     if (Acurrent->element=Bcurrent->element){
               Ccurrent->next= (SET)malloc(sizeof(Cell));
               Ccurrent=Ccurrent->next;
               Ccurrent->element=Acurrent->element;
               Acurrent=Acurrent->next;
               Bcurrent=Bcurrent->next;
     }
     else if (Acurrent->element<Bcurrent->element)
                        Acurrent=Acurrent->next;
               else  Bcurrent=Bcurrent->next;
   }
   Ccurrent->next=NULL;
}
```

Phép toán hiệu có thể viết tương tự (xem như bài tập). Phép ASSIGN(A,B) chép các các phần tử của tập A sang tập B, chú ý rằng ta không được làm bằng lệnh gán đơn giản B=A vì nếu làm như vậy hai danh sách biểu diễn cho hai tập hợp A,B chỉ là một nên sự thay đổi trên tập này kéo theo sự thay đổi ngoài ý muốn trên tập hợp kia. Toán tử MIN(A) chỉ cần trả ra phần tử đầu danh sách (tức là A->next->element). Toán tử DELETE_SET là hoàn toàn giống như DELETE_LIST. Phép INSERT_SET(x,A) cũng tương tự INSERT_LIST tuy nhiên ta phải chú ý rằng khi xen x vào A phải đảm bảo thứ tự của danh sách.

Thêm phần tử vào tập hợp tổ chức như danh sách có thứ tự tăng

```
void INSERT_SET(ElementType X, SET& L){
        SET T,P;
        P=L;
        while (P->next!=NULL)
         if (P->next->element <=X) P=P->next;
         else break;
        // P dang luu tru vi tri de xen phan tu X vao
        T=(SET)malloc(sizeof(Cell));
        T->element=X;
        T->next=P->next;
        P->next=T;
}
```

Xoá phần tử ra khỏi tập hợp tổ chức như danh sách có thứ tự tăng

```
void DELETE_SET(ElementType X, SET& L)
{
        SET T,P;
        P=L;
        while (P->next!=NULL)
                if (P->next->element<X) P=P->next;
                else break;
                if(P->next->element ==X) {
                        T=P->next;
                        P->next=T->next;
                        free(T);
                 }
}
```

Kiểm tra sự hiện diện của phần tử trong tập hợp (không cần có thứ tự)

Hàm kiểm tra xem phần tử X có thuộc tập hợp hay không. Hàm trả về giá trị 1 nếu phần tử X đó thuộc tập hợp và ngược lại, hàm trả về giá trị 0.

```
int MEMBER(ElementType X, SET L){
        SET P;
        P = L->next;
        while (P != NULL)
                if (P->element == X) return 1;
                else P = P->next;
        return 0;
}
```

IV. TỪ ĐIỂN (DICTIONARY)

Từ điển là một kiểu dữ liệu trừu tượng tập hợp đặc biệt, trong đó chúng ta chỉ quan tâm đến các phép toán INSERT_SET, DELETE_SET, MEMBER và MAKENULL_SET. Sở dĩ chúng ta nghiên cứu từ điển là do trong nhiều ứng dụng không sử dụng đến các phép toán hợp, giao, hiệu của hai tập hợp. Ngược lại ta cần

một cấu trúc làm sao cho việc tìm kiếm, thêm và bớt phần tử có phần hiệu quả nhất gọi là từ điển. Chúng ta cũng chấp nhận MAKENULL_SET như là phép khởi tạo cấu trúc.

1. Cài đặt từ điển bằng mảng

Thực chất việc cài đặt từ điển bằng mảng hoàn toàn giống với việc cài đặt danh sách đặc không có thứ tự.

Khai báo

```
#define MaxLength ... // So phan tu toi da
typedef ... ElementType; // Kieu du lieu trong tu dien
typedef int Position;
typedef struct {
    ElementType Data[MaxLength];
    Position Last;
} SET;
```

Khởi tạo cấu trúc rỗng

```
void MAKENULL_SET(SET& L){
    L.Last=0;
}
```

Hàm kiểm tra thành viên của tập hợp

```
int MEMBER(ElementType X, SET L){
    Position P=1;
    while (P <= L.Last)
        if (L.Data[P] == X) return 1;
        else P++;
    return 0;
}
```

Thêm một phần tử vào tập hợp

Vì danh sách không có thứ tự nên ta có thể thêm phần tử mới vào cuối danh sách.

```
void INSERT_SET(ElementType X, SET& L){
    if (L.Last==MaxLength)
        printf("Mang day");
    else
        if (Member(X,L)==0)    {
            L.Last++;
            L.Data[L.Last]=X;
        }
    else
        printf ("\nPhan tu da ton tai trong tu dien");
}
```

Xóa một phần tử trong tập hợp

Để xoá một phần tử nào đó ta phải tiến hành tìm kiếm nó trong danh sách. Vì danh sách không có thứ tự nên ta có thay thế phần tử bị xoá bằng phần tử cuối cùng trong danh sách để khỏi phải dời các phần tử nằm sau phần tử bị xoá lên một vị trí.

```
void DELETE_SET(ElementType X, SET& L){
      if (L.Last==0) //EmptySET(L)
      printf("Tap hop rong!");
      else  {
      Position Q=1;
      while ((Q<=L.Last)&& (L.Data[Q]!=X)) Q++;
      if ( L.Data[Q]==X) {
                        for (int i=Q;i<L.Last;i++)
                              L.Data[i]=L.Data[i+1];
                        L.Last--;
                }
 }
}
```

Cài đặt tự điển bằng mảng đòi hỏi tốn n phép so sánh để xác định xem một phần tử có thuộc từ điển n phần tử hay không thông qua hàm MEMBER. Trên từ điển, việc tìm kiếm một phần tử được xác định bằng hàm MEMBER sẽ thường xuyên được sử dụng. Do đó, nếu hàm MEMBER thực hiện không hiệu quả sẽ làm giảm đi ý nghĩa của từ điển (vì nói đến từ điển là phải tìm kiếm nhanh chóng). Mặt khác hàm MEMBER còn được gọi từ trong thủ tục INSERT_SET và nó cũng dẫn đến là thủ tục này cũng không hiệu quả.

V. CẤU TRÚC BẢNG BĂM (HASH TABLE)

Cài đặt từ điển bằng mảng đòi hỏi tốn n bước để thực hiện một phép toán đơn giản như INSERTSET hoặc MEMBER trên từ điển có n phần tử. Cài đặt bằng danh sách cũng có cùng tốc độ này. Cài đặt bằng vectơ bít chỉ mất một hằng thời gian cho mỗi phép toán trên từ điển, nhưng phải giới hạn trong một phạm vi nhỏ của tập hợp số nguyên. Có một kỹ thuật khác rất quan trọng và được dùng rộng rãi để cài đặt từ điển đó là BĂM (hashing).

Băm đòi hỏi trung bình chỉ một hằng thời gian cho mỗi phép toán trên và không đòi hỏi tập hợp phải là tập con của bất kỳ một tập hợp toàn thể hữu hạn nào. Trong trường hợp xấu nhất phương pháp này đòi hỏi thời gian tỉ lệ với kích thước của tập hợp như là cài đặt bằng mảng hoặc bằng danh sách. Tuy nhiên, bằng việc thiết kế cẩn thận chúng ta có thể làm cho xác suất băm đòi hỏi lớn hơn một hằng thời gian đối với mỗi toán hạng là nhỏ tuỳ ý.

Băm (hashing) là một phương pháp tính toán trực tiếp vị trí của mảng lưu trữ một phần tử của tập hợp dựa giá trị của phần tử này, tức là tính toán "địa chỉ" trực tiếp từ khoá.

Giả sử ta có một mảng gồm B phần tử được đánh số 0..B-1 và một tập hợp A muốn lưu trữ vào trong mảng này. Như vậy với mỗi phần tử x ϵA ta phải dựa vào khoá của nó để suy ra một giá trị số nguyên thuộc khoảng 0..B- 1 là vị trí cất giữ khoá này. Nói cách khác, ta chọn một hàm:

$$h: A \rightarrow 0..B\text{-}1$$

$$x \rightarrow h(x)$$

Khi đó h(x) là vị trí lưu giữ phần tử x trong mảng. Do vậy khi cần tìm kiếm phần tử x ta chỉ cần tính h(x). Giá trị h(x) gọi là giá trị băm của khoá x và hàm h gọi là hàm băm. Nói chung sẽ có nhiều khoá có cùng giá trị băm. Tức là với hai khoá x,y khác nhau nhưng h(x)=h(y) vậy x,y có cùng một vị trí trong mảng, trường hợp này ta gọi là đụng độ (collision).

Như vậy để áp dụng phương pháp băm ta cần chọn hàm h sao cho ít xảy ra đụng độ hay h có thể "rải đều" các khoá vào trong mảng. Hơn nữa ta phải có cách giải quyết khi đụng độ xảy ra. Mảng được dùng ở trên gọi là bảng băm (hash table).

Ví dụ: Hàm h dưới đây biến đổi một chuỗi các ký tự thành số nguyên thuộc đoạn 0..B-1.

```
#define B …
typedef char*  ElementType;
int h(ElementType x){
int i,sum=0;
for (i=0;i<strlen(x);i++) sum=sum+x[i];
 return sum % B;
}
```

Chẳng hạn với B=11 ta có:

h(“WINDOWS XP”) = 5

h(“EXCEL”) = 9

h(“WINWORD”) = 4

h(“NETWORK”) = 4

h(“INTERNET”) = 7

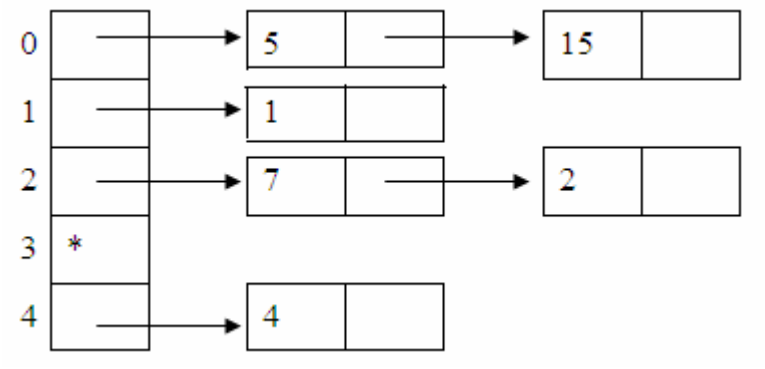
2. Bảng băm mở

Ý tưởng đơn giản để giải quyết đụng độ là tổ chức một danh sách cho một tập hợp các khoá có cùng giá trị băm, xem hình IV.1. Tức là chia một tập hợp (có thể không hữu hạn) thành một số hữu hạn các lớp. Nếu bảng băm có B phần tử được đánh số từ 0.. B-1 và h là hàm băm thì lớp thứ k là một danh sách gồm tất cả các phần tử x sao cho h(x)=k. Danh sách có thể tổ chức theo bất kỳ cách nào, nhưng vì số lượng các khóa trong mỗi lớp là không thể biết trước do vậy mỗi lớp ta tổ chức một danh sách liên kết các phần tử của chúng. Mỗi lớp gọi là một bucket. Giả sử bảng băm là mảng A thì Bucket thứ k có chỉ điểm đầu là A[k].

Ta hy vọng rằng các bucket sẽ có số phần tử bằng nhau, như vậy độ dài mỗi bucket là nhỏ nhất. Tức là nếu tập hợp có N phần tử thì trung bình mỗi bucket có N/B phần tử. Nếu có thể ước lượng được N rồi chọn B đủ lớn thì mỗi bucket chỉ có 1 hoặc 2 phần tử, như vậy các phép toán trên từ điển trung bình chỉ mất một hằng số rất nhỏ các bước, hằng số này độc lập đối với N, B. Chú ý rằng mỗi bucket sẽ chứa số phần tử không nhiều vì vậy nếu ta dùng B phần tử của bảng băm làm B chỉ điểm thực sự (như đã dùng trong danh sách liên kết) sẽ gây lãng phí lớn, do đó phần tử đầu của bucket k ta đặt ngay vào A[k].

Ví dụ : Cho tập hợp A = {1,5,7,2,4,15}

Bảng băm là mảng gồm 5 phần tử và hàm băm h(x) = x % 5; Ta có bảng băm lưu trữ A như sau :



Bảng băm chứa các chỉ điểm đầu của danh sách

Danh sách của mỗi bucket

Hình IV.1: Bảng băm mở

Sử dụng bảng băm mở để cài đặt từ điển

Theo cách tổ chức bảng băm mở, với mỗi phần tử x ta đều có thể tính toán trực tiếp địc chỉ của nó trong bảng. Có thể có một vài phần tử có cùng bucket, nhưng số này là nhỏ (nếu ta chọn hàm băm tốt, nó "rải đều" các khóa). Vậy các phép toán thêm, xóa, tìm kiếm một phần tử của tự điển sẽ làm việc hiệu quả (thời gian hằng). Dưới đây là các thủ tục cài đặt từ điển bằng bảng băm mở với giả thiết rằng các phần tử trong từ điển có kiểu ElementType và hàm băm là H.

Khai báo

```
#define B ...
typedef ... ElementType;
typedef struct Node{
      ElementType Data;
      Node* Next;
};
typedef Node* Position;
```

```
typedef Position Dictionary[B];
```

Khởi tạo bảng băm mở rỗng

Lúc này tất cả các bucket là rỗng nên ta gán tất cả các con trỏ trỏ đến đầu các danh sách trong mỗi bucket là NULL.

```
void MAKENULL_SET(Dictionary& D){
    for(int i=0;i<B;i++)
     D[i]=NULL;
}
```

Kiểm tra một thành viên trong từ điển được cài bằng bảng băm mở

Để kiểm tra xem một khoá x nào đó có trong từ điển hay không, ta tính địa chỉ của nó trong bảng băm. Theo cấu trúc của bảng băm thì khoá x sẽ nằm trong bucket được trỏ bởi D[h(x)], với h(x) là hàm băm. Như vậy để tìm khoá x trước hết ta phải tính h(x) sau đó duyệt danh sách của bucket được trỏ bởi D[h(x)]. Giải thuật như sau:

```
int MEMBER(ElementType X, Dictionary D){
    Position P = D[H(X)];
    //Duyet tren ds thu H(X)
    while (P!=NULL)
            if (P->Data==X) return 1;
            else P=P->Next;
    return 0;
}
```

Thêm một phần tử vào từ điển được cài bằng bảng băm mở

Để thêm một phần tử có khoá x vào từ điển ta phải tính bucket chứa nó, tức là phải tính h(x). Phần tử có khoá x sẽ được thêm vào bucket được trỏ bởi D[h(x)]. Vì ta không quan tâm đến thứ tự các phần tử trong mỗi bucket nên ta có thể thêm phần tử mới ngay đầu bucket này. Giải thuật như sau:

```
void INSERT_SET(ElementType X, Dictionary& D)
{
    int Bucket;
    Position P;
    if (!MEMBER(X,D))
    {
            Bucket=H(X);
            P=D[Bucket]; //giu lai D[Bucket] hien tai
            D[Bucket] = (Node*)malloc(sizeof(Node)); //Cap phat o nho moi cho D[Bucket]
            D[Bucket]->Data=X;
            D[Bucket]->Next=P;
    }
}
```

Xoá một phần tử trong từ điển được cài bằng bảng băm mở

Xoá một phần tử có khoá x trong từ điển bao gồm việc tìm ô chứa khoá và xoá ô này. Phần tử x, nếu có trong từ điển, sẽ nằm ở bucket D[h(x)]. Có hai trường hợp cần phân biệt. Nếu x nằm ngay đầu bucket, sau khi xoá x thì phần tử kế tiếp sau x trong bucket sẽ trở thành đầu bucket. Nếu x không nằm ở đầu bucket thì ta duyệt bucket này để tìm và xoá x. Trong trường hợp này ta phải định vị con trỏ duyệt tại "ô trước" ô chứa x để cập nhật lại con trỏ Next của ô này. Giải thuật như sau:

```
void DELETE_SET(ElementType X, Dictionary& D){
        int Bucket, Done;
        Position P,temp;
        Bucket=H(X);
        // Neu danh sach ton tai
        if (D[Bucket]!=NULL)    {
                if (D[Bucket]->Data==X){ // X o dau danh sach
                        temp=D[Bucket];
                        D[Bucket]=D[Bucket]->Next;
                        free(temp);
                }
                else { // Tim X
                        P=D[Bucket];
                        while (P->Next!=NULL)
                                if (P->Next->Data==X) break;
                                else P=P->Next;
                        // Neu tim thay
                        if (P->Next!=NULL)      {
                                temp=P->Next;
                                P->Next=temp->Next;
                                free(temp);
                        }
                }
        }
}
```

2.2. Cài đặt từ điển bằng bảng băm đóng

Bảng băm đóng

Bảng băm đóng lưu giữ các phần tử của từ điển ngay trong mảng chứ không dùng mảng làm các chỉ điểm đầu của các danh sách liên kết. Bucket thứ i chứa phần tử có giá trị băm là i, nhưng vì có thể có nhiều phần tử có cùng giá trị băm nên ta sẽ gặp trường hợp sau: ta muốn đưa vào bucket i một phần tử x nhưng bucket này đã bị chiếm bởi một phần tử y nào đó (đụng độ). Như vậy khi thiết kế một bảng băm đóng ta phải có cách để giải quyết sự đụng độ này.

Giải quyết đụng độ

Cách giải quyết đụng độ đó gọi là **chiến lược băm lại** (rehash strategy). Chiến lược băm lại là chọn tuần tự các vị trí h_1,..., h_k theo một cách nào đó cho tới khi gặp một vị

trí trống để đặt x vào. Dãy h_1,..., h_k gọi là dãy các phép thử. Một chiến lược đơn giản là **băm lại tuyến tính, trong đó dãy các phép thử có dạng :**

$$h_i(x)=(h(x)+i) \bmod B$$

Ví dụ B=8 và các phần tử của từ điển là a,b,c,d có giá trị băm lần lượt là: h(a)=3, h(b)=0, h(c)=4, h(d)=3. Ta muốn đưa các phần tử này lần lượt vào bảng băm.

Khởi đầu bảng băm là rỗng, có thể coi mỗi bucket chứa một giá trị đặc biệt Empty, Empty không bằng với bất kỳ một phần tử nào mà ta có thể xét trong tập hợp các phần tử muốn đưa vào bảng băm.

Ta đặt a vào bucket 3, b vào bucket 0, c vào bucket 4. Xét phần tử d, d có h(d)=3 nhưng bucket 3 đã bị a chiếm ta tìm vị trí $h_1(x)$= (h (x)+1) mod B = 4, vị trí này cũng đã bị c chiếm, tiếp tục tìm sang vị trí h_2 (x)= (h(x)+2) mod B= 5 đây là một bucket rỗng ta đặt d vào (xem hình IV.2)

0	b
1	
2	
3	a
4	c
5	d
6	
7	

Hình IV.2: Giải quyết đụng độ trong bảng băm đóng bằng chiến lược băm lại tuyến tính

Trong bảng băm đóng, phép kiểm tra một thành viên(thủ tục MEMBER (x,A)) phải xét dãy các bucket h(x),h_1(x),h_2(x),... cho đến khi tìm thấy x hoặc tìm thấy một vị trí trống. Bởi vì nếu h_k(x) là vị trí trống được gặp đầu tiên thì x không thể được tìm gặp ở một vị trí nào xa hơn nữa. Tuy nhiên, nói chung điều đó chỉ đúng với trường hợp ta không hề xoá đi một phần tử nào trong bảng băm. Nếu chúng ta chấp nhận phép xoá thì chúng ta qui ước rằng phần tử bị xóa sẽ được thay bởi một giá trị đặc biệt, gọi là **Deleted**, giá trị Deleted không bằng với bất kỳ một phần tử nào trong tập hợp đang xét vào nó cũng phải khác giá trị **Empty**. Empty cũng là một giá trị đặc biệt cho ta biết ô trống.

Ví dụ

- Tìm phần tử e trong bảng băm trên, giả sử h(e)=4. Chúng ta tìm kiếm e tại các vị trí 4,5,6. Bucket 6 là chứa Empty, vậy không có e trong bảng băm.

- Tìm d, vì h(d)=3 ta khởi đầu tại vị trí này và duyệt qua các bucket 4,5. Phần tử d được tìm thấy tại bucket 5.

Sử dụng bảng băm đóng để cài đặt từ điển

Dưới đây là khai báo và thủ tục cần thiết để cài đặt từ điển bằng bảng băm đóng. Để dễ dàng minh hoạ các giá trị Deleted và Empty, giả sử rằng ta cần cài đặt từ điển gồm các chuỗi 10 kí tự. Ta có thể qui ước:

Empty là chuỗi 10 dấu + và Deleted là chuỗi 10 dấu *.

Khai báo

```
#define B 101
#define Deleted "++++++++++"          //giá trị giả định cho ô đã bị xóa
#define Empty "**********"            //giá trị giả định cho ô trống
typedef char*  ElementType;           //kiểu phần tử của bảng băm
typedef ElementType Dictionary [B];   // bảng băm
```

Hàm băm

Hàm băm đổi chuỗi thành số rồi lấy mod B.

```
int h(ElementType x){
        int i,sum=0;
        for (i=0;i<strlen(x);i++)
                sum=sum+x[i];
        return sum % B;
}
```

Tạo tự điển rỗng

```
void MAKENULL_SET(Dictionary D){
        for (int i=0;i<B; i++)
                D[i]=Empty;
}
```

Kiểm tra sự tồn tại của phần tử trong tự điển

- Hàm Locate duyệt từ điển từ vị trí H(x) cho đến khi tìm thấy x hoặc tìm thấy Empty đầu tiên. Nó trả về chỉ số của mảng tại chổ dừng.

- Hàm Locate1 duyệt từ điển từ vị trí H(x) cho đến khi tìm thấy x hoặc tìm thấy Empty hay Deleted đầu tiên. Nó trả về chỉ số của mảng tại chổ dừng

```
int LOCATE(ElementType x, Dictionary A){
        int inital,i;
        inital=h(x);
        i=0;
        while ((i<B) && ( A[(inital+i) %  B]!=x) && (A[(inital+i) % B]!= Empty))
                i++;
        return (inital+i) % B;
```

```
}

int LOCATE1(ElementType x, Dictionary A){
        int inital,i;
        inital=h(x);
        i=0;
        while    ((i<B) && ( A[(inital+i) % B]!=x) && (A[(inital+i) % B]!= Empty)
                 && (A[(inital+i) % B]!= Deleted))
                 i++;
        return (inital+i) %  B ;
}
```

Hàm trả về giá tri 0 nếu phần tử X không tồn tại trong tự điển; Ngược lại, hàm trả về giá trị 1;

```
int MEMBER(ElementType x, Dictionary A){
        return A[LOCATE(x,A)] == x;
}
```

Thêm phần tử vào tự điển

```
void INSERT_SET(ElementType x,Dictionary A){
        int bucket;
        if (A[LOCATE(x,A)]!=x) // chưa có x trong bảng
        bucket= LOCATE1 (x,A);
        if ((A[bucket]==Empty) || (A[bucket]==Deleted))
                A[bucket]=x;
        else printf ("loi: bang bam day");
}
```

Xóa từ ra khỏi tự điển

```
void DELETE_SET(ElementType x,Dictionary A){
        int bucket;
        bucket=LOCATE(x,A);
        if (A[bucket]==x)
                A[bucket]=Deleted; //danh dau o bi xoa
}
```

3. Các phương pháp xác định hàm băm

Phương pháp chia

"Lấy phần dư của giá trị khoá khi chia cho số bucket" . Tức là hàm băm có dạng:

H(x)= x mod B

Phương pháp này rõ ràng là rất đơn giản nhưng nó có thể không cho kết quả ngẫu nhiên lắm. Chẳng hạn B=1000 thì H(x) chỉ phụ thuộc vào ba số cuối cùng của khoá

mà không phụ thuộc vào các số đứng trước. Kết quả có thể ngẫu nhiên hơn nếu B là một số nguyên tố.

Phương pháp nhân

"Lấy khoá nhân với chính nó rồi chọn một số chữ số ở giữa làm kết quả của hàm băm".

Ví dụ

x	x^2	h(x) gồm 3 số ở giữa
5402	29181604	181 hoặc 816
0367	00134689	134 346
1246	01552516	552 525
2983	08898289	898 982

Vì các chữ số ở giữa phụ thuộc vào tất cả các chữ số có mặt trong khoá do vậy các khoá có khác nhau đôi chút thì hàm băm cho kết quả khác nhau.

Phương pháp tách

Đối với các khoá dài và kích thước thay đổi người ta thường dùng phương pháp phân đoạn, tức là phân khoá ra thành nhiều đoạn có kích thước bằng nhau từ một đầu (trừ đoạn tại đầu cuối), nói chung mỗi đoạn có độ dài bằng độ dài của kết quả hàm băm. Phân đoạn có thể là tách hoặc gấp:

a. Tách: tách khóa ra từng đoạn rồi xếp các đoạn thành hàng được canh thẳng một đầu rồi có thể cộng chúng lại rồi áp dụng phương pháp chia để có kết quả băm.

ví dụ: khoá 17046329 tách thành

329

046

017

cộng lại ta được 392. 392 mod 1000 = 392 là kết quả băm khoá đã cho.

b. Gấp: gấp khoá lại theo một cách nào đó, có thể tương tự như gấp giấy, các chữ số cùng nằm tại một vị trí sau khi gấp dược xếp lại thẳng hàng với nhau rồi có thể cộng lại rồi áp dụng phương pháp chia (mod) để cho kết quả băm

Ví dụ: khoá 17046329 gấp hai biên vào ta có

932

046

710

Cộng lại ta có 1679. 1679 mod 1000= 679 là kết quả băm khoá đã cho.

V. HÀNG ƯU TIÊN (PRIORITY QUEUE)

1. Khái niệm hàng ưu tiên

Hàng ưu tiên là một kiểu dữ liệu trừu tượng tập hợp đặc biệt, trong đó mỗi phần tử có một độ ưu tiên nào đó.

Độ ưu tiên của phần tử thường là một số, theo đó, phần tử có độ ưu tiên nhỏ nhất sẽ được '*ưu tiên*' nhất. Một cách tổng quát thì độ ưu tiên của một phần tử là một phần tử thuộc tập hợp được xếp theo thứ tự tuyến tính.

Trên hàng ưu tiên chúng ta chỉ quan tâm đến các phép toán: MAKENULL để tạo ra một hàng rỗng, INSERT để thêm phần tử vào hàng ưu tiên và DELETEMIN để xoá phần tử ra khỏi hàng với phần tử được xóa có độ ưu tiên bé nhất.

Ví dụ: tại bệnh viện, các bệnh nhân xếp hàng để chờ phục vụ nhưng không phải người đến trước thì được phục vụ trước mà họ có độ ưu tiên theo tình trạng khẩn cấp của bệnh.

2. Cài đặt hàng ưu tiên

Chúng ta có thể cài đặt hàng ưu tiên bằng danh sách liên kết. Danh sách liên kết có thể dùng có thứ tự hoặc không có thứ tự. Nếu danh sách liên kết có thứ tự thì ta có thể dễ dàng tìm phần tử nhỏ nhất, đó là phần tử đầu tiên, nhưng phép thêm vào đòi hỏi ta phải duyệt trung bình phân nửa danh sách để có một chổ xen thích hợp. Nếu danh sách chưa có thứ tự thì phép thêm vào có thể thêm vào ngay đầu danh sách, nhưng để tìm kiếm phần tử nhỏ nhất thì ta cũng phải duyệt trung bình phân nửa danh sách.

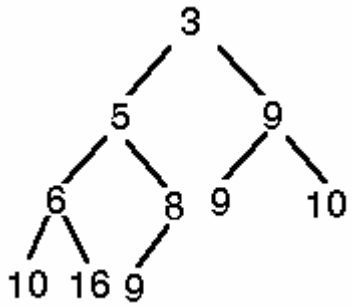
Ta không thể cài đặt hàng ưu tiên bằng bảng băm vì bảng băm không thuận lợi trong việc tìm kiếm phần tử nhỏ nhất. Một cách cài đặt hàng ưu tiên khá thuận lợi đó là cài đặt bằng cây có thứ tự từng phần.

2.1. Cài đặt hàng ưu tiên bằng cây có thứ tự từng phần

Định nghĩa cây có thứ tự từng phần

Cây có thứ tự từng phần là cây nhị phân mà giá trị tại mỗi nút đều nhỏ hơn hoặc bằng giá trị của hai con.

Ví dụ:



Hình IV.3: Cây có thứ tự từng phần

Nhận xét: Trên cây có thứ tự từng phần, nút gốc là nút có giá trị nhỏ nhất.

Từ nhận xét này, ta thấy có thể sử dụng cây có thứ tự từng phần đề cài đặt hàng ưu tiên và trong đó mỗi phần tử được biểu diễn bởi một nút trên cây mà độ ưu tiên của phần tử là giá trị của nút.

Để việc cài đặt được hiệu quả, ta phải cố gắng sao cho cây tương đối 'cân bằng'. Nghĩa là mọi nút trung gian (trừ nút là cha của nút lá) đều có hai con; Đối với các nút cha của nút lá có thể chỉ có một con và trong trường hợp đó ta quy ước là con trái (không có con phải).

Để thực hiện DELETEMIN ta chỉ việc trả ra nút gốc của cây và loại bỏ nút này. Tuy nhiên nếu loại bỏ nút này ta phải xây dựng lại cây với yêu cầu là cây phải có thứ tự từng phần và phải "cân bằng".

Chiến lược xây dựng lại cây như sau

Lấy nút lá tại mức cao nhất và nằm bên phải nhất thay thế cho nút gốc, như vậy cây vẫn "*cân bằng*" nhưng nó không còn đảm bảo tính thứ tự từng phần. Như vậy để xây dựng lại cây từng phần ta thực hiện việc "**đẩy nút này xuống dưới**" tức là ta đổi chỗ nó với nút con nhỏ nhất của nó, nếu nút con này có độ ưu tiên nhỏ hơn nó.

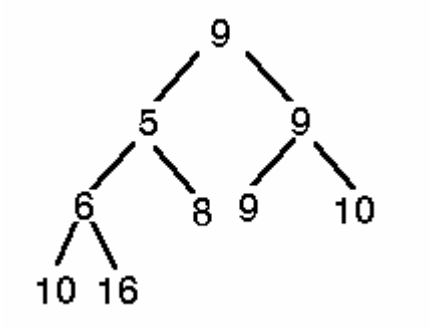
Giải thuật đẩy nút xuống như sau:

- Nếu giá trị của nút gốc lớn hơn giá trị con trái và giá trị con trái lớn hơn hoặc bằng giá trị con phải thì đẩy xuống bên trái. (Hoán đổi giá trị của nút gốc và con trái cho nhau)

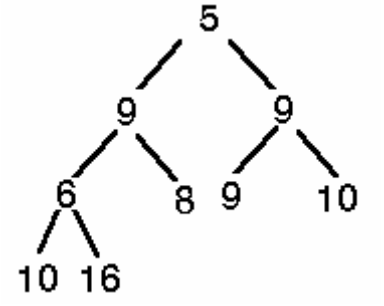
- Nếu giá trị của nút gốc lớn hơn giá trị con phải và giá trị con phải nhỏ hơn giá trị con trái thì đẩy xuống bên phải. (Hoán đổi giá trị của nút gốc và con phải cho nhau)

- Sau khi đẩy nút gốc xuống một con nào đó (trái hoặc phải) thì phải tiếp tục xét con đó xem có phải dẩy xuống nữa hay không. Quá trình đẩy xuống này sẽ kết thúc khi ta đã đẩy đến nút lá hoặc cây thỏa mãn tính chất có thứ tự từng phần.

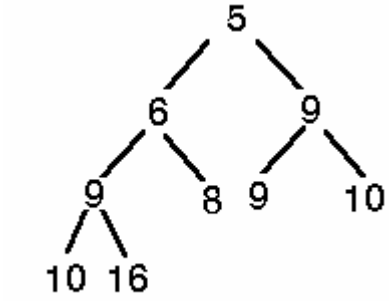
Ví dụ: thực hiện DELETEMIN với cây trong hình IV.3 trên ta loại bỏ nút 3 và thay nó bằng nút 9 (nút con của nút 8), cây có dạng sau



Ta "đẩy nút 9 tại gốc xuống" nghĩa là ta đổi chỗ nó với nút 5



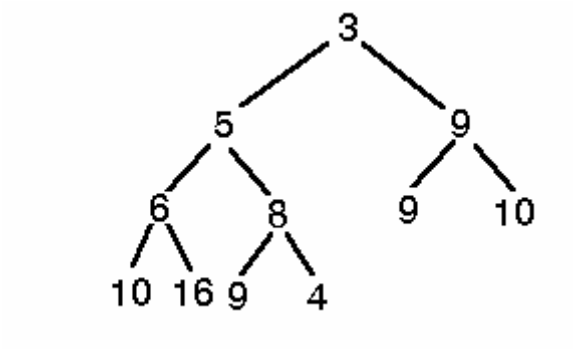
Tiếp tục "đẩy nút 9 xuống" bằng cách đổi chỗ nó với 6



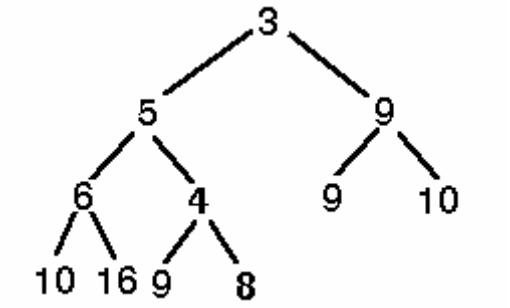
Quá trình đã kết thúc.

Xét phép toán INSERT, để thêm một phần tử vào cây ta bắt đầu bằng việc tạo một nút mới là lá nằm ở mức cao nhất và ngay bên phải các lá đang có mặt trên mức này. Nếu tất cả các lá ở mức cao nhất đều đang có mặt thì ta thêm nút mới vào bên trái nhất ở mức mới. Tiếp đó ta cho nút này "**nổi dần lên**" bằng cách đổi chỗ nó với nút cha của nó nếu nút cha có độ ưu tiên lớn hơn. Quá trình nổi dần lên cũng là quá trình đệ quy. Quá trình đó sẽ dừng khi đã nổi lên đến nút gốc hoặc cây thỏa mãn tính chất có thứ tự từng phần.

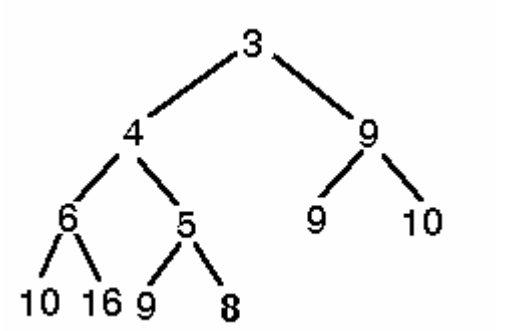
Ví dụ: thêm nút 4 vào cây trong hình IV.3, ta đặt 4 vào lá ở mức cao nhất và ngay bên phải các lá đang có mặt trên mức này ta được cây



Cho 4 "nổi lên" bằng cách đổi chỗ với nút cha



Tiếp tục cho 4 nổi lên ta có cây

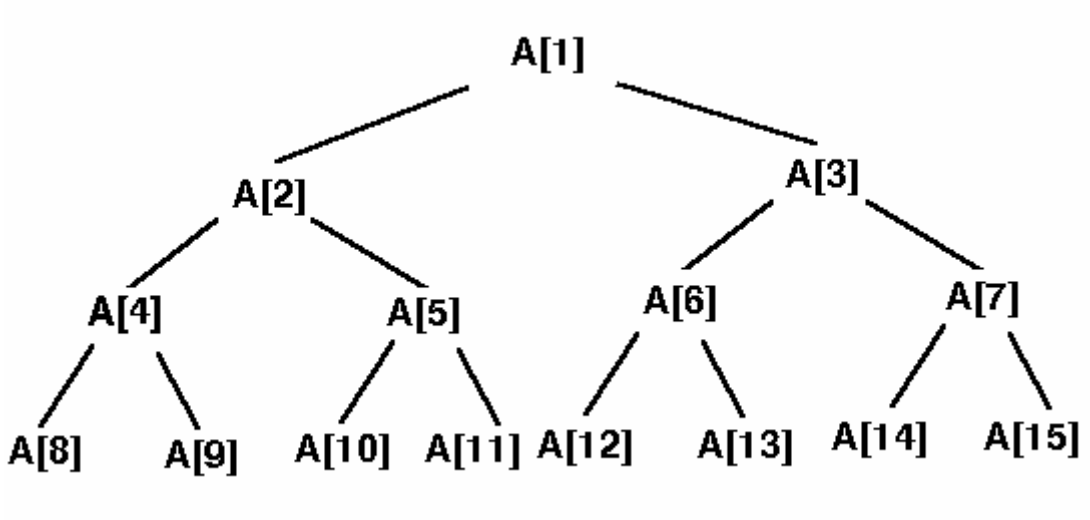


Quá trình đã kết thúc

2.2. Cài đặt cây có thứ tự từng phần bằng mảng.

Trong thực tế các cây có thứ tự từng phần như đã bàn bạc ở trên thường được cài đặt bằng mảng hơn là cài đặt bằng con trỏ. Cây có thứ tự từng phần được biểu diễn bằng mảng như vậy gọi là HEAP. Nếu cây có n nút thì ta chứa n nút này vào n ô đầu của mảng A nào đó, A[1] chứa gốc cây. Nút A[i] sẽ có con trái là A[2i] và con phải là A[2i+1]. Việc biểu diễn này đảm bảo tính '*cân bằng*' như chúng ta đã mô tả trên.

Ví dụ: HEAP có 15 phần tử ta sẽ có cây như trong hình IV.4



Hình IV.4

Nói cách khác nút cha của nút A[i] là A[i div 2], với i>1. Như vậy cây được xây dựng lớn lên từ mức này đến mức khác bắt đầu từ đỉnh (gốc) và tại mỗi mức cây phát triển từ trái sang phải. Cài đặt hàng ưu tiên bằng mảng như sau:

Khai báo cài đặt

```
#define MaxLength 100
typedef int ElementType;
typedef struct{
   ElementType Data[MaxLength];
   int Last;
} PriorityQueue;
```

Giả sử p là hàm trả về độ ưu tiên của khóa, để đơn giản giả sử p(x)=x.

```
int p(ElementType x){
        return x;
}
```

Hàm khởi tạo hàng ưu tiên rỗng

```
void MakeNullPriorityQueue(PriorityQueue& L){
 L.Last=0;
}
```

Thêm một phần tử vào hàng ưu tiên hay thêm một nút vào cây có thứ tự từng phần

```
void InsertPriorityQueue(ElementType X, PriorityQueue& L){
  ElementType temp;
  if (L.Last>MaxLength-1)
        printf("Hang day");
  else  {
        L.Last++;
        L.Data[L.Last]=X;
        int i= L.Last;
```

```
        while ((i>0)&&(p(L.Data[i])<p(L.Data[i/2]))) {
                temp=L.Data[i];
                L.Data[i]=L.Data[i/2];
                L.Data[i/2]=temp;
                i=i/2;
        }
  }
}
```

Xóa phần tử có độ ưu tiên bé nhất

```
ElementType DeleteMin(PriorityQueue& L){
  int i,j;
  ElementType temp;

   if (L.Last==0)
        printf("\nHang rong!");
   else {
        ElementType minimum= L.Data[1];
        L.Data[1]=L.Data[L.Last];
        L.Last--;
        // Qua trinh day xuong
        i=1;
        while (i<=L.Last/2) {
                // Tim nut be nhat trong hai nut con cua i
                if ((p(L.Data[2*i])<p(L.Data[2*i+1])) || (2*i==L.Last))
                        j=2*i;
                else j=2*i+1;

                if (p(L.Data[i])>p(L.Data[j])){
                        // Doi cho hai phan tu
                        temp=L.Data[i];
                        L.Data[i]=L.Data[j];
                        L.Data[j]=temp;
                        i=j; // Bat dau o muc moi
                }
                else break;
        }//while
    return minimum;
  }//else
}
```

Áp dụng: Viết chương trình gọi các hàm trên để thực hiện việc tạo một hàng ưu tiên từ 1 dãy số nguyên. Sau khi hoàn tất việc nhập, hãy in lần lượt các khóa khi thực hiện hàm DeleteMin.

```
void main(){
int n,x;
PriorityQueue L;
MakeNullPriorityQueue(L);
printf("hang co may phan tu");
```

```
scanf("%d",&n);
for (int i=1;i<=n ; i++){
        printf("nhap phan tu thu %d ",i);
        scanf("%d",&x);
        InsertPriorityQueue(x,L);
}
printf("Xoá lần lượt các nút có khóa nhỏ nhất \n");
while (L.Last>0)
        printf("%d\n", DeleteMin(L));
getch();
}
```

BÀI TẬP

1. Viết các khai báo cấu trúc dữ liệu và các thủ tục/hàm cho các phép toán trên tập hợp để cài đặt tập hợp kí tự (256 kí tự ASCII) bằng vectơ bít.

2. Viết các khai báo cấu trúc dữ liệu và các thủ tục/hàm cho các phép toán trên tập hợp để cài đặt tập hợp các số nguyên bằng danh sách liên kết có thứ tự.

3. Giả sử bảng băm có 7 bucket, hàm băm là h(x)= x mod 7. Hãy vẽ hình biểu diễn bảng băm khi ta lần lượt đưa vào bảng băm rỗng các khoá 1,8, 27, 64, 125, 216, 343 trong các trường hợp:

- Dùng bảng băm mở.
- Bảng băm đóng với chiến lược giải quyết đụng độ là phép thử tuyến tính.

4. Cài đặt bảng băm đóng, với chiến lược băm lại là phép thử cầu phương. Tức là hàm băm lại lần thứ i có dạng $h_i = (h(x)+i2) \bmod B$.

5. Giả sử trong một tập tin văn bản ta có các kí tự đặc biệt sau:

   BLANK=32 là mã ASCII của kí tự trống

   CR = 13 là mã ASCII kí tự kết thúc dòng

   LF = 10 là mã ASCII kí tự kết xuống dòng

   EOF= 26 là mã ASCII kí tự kết thúc tập tin

Một **từ** (word) trong văn bản được định nghĩa là một chuỗi gồm các kí tự không chứa kí tự đặc biệt nào. Hơn nữa kí tự trước chuỗi trong văn bản hoặc không có hoặc là kí tự đặc biệt và kí tự sau chuỗi là kí tự đặc biệt.

Viết chương trình thành lập một từ điển gồm các từ trong văn bản bằng một bảng băm mở. Bằng cách đọc từng kí tự của một tập tin văn bản cho đến hết văn bản, khi đọc phải thiết lập từ để khi gặp kí tự đặc biệt (hết từ) thì đưa từ đó vào bảng băm đưa vào bảng băm nếu nó chưa có trong bảng. Hàm băm có thể chọn như hàm băm chuỗi 10 kí tự trong bài học.

6. Viết chương trình dùng cấu trúc bảng băm mở để cài đặt một từ điển tiếng Anh đơn giản. Giả sử mỗi mục từ trong từ điển chỉ gồm có từ tiếng Anh và phần giải nghĩa của từ này. Cấu trúc mỗi mục từ như sau:

Mẩu tin item gồm có 2 trường:

Word: kiểu chuỗi ký tự để lưu từ khóa cần tra;

Explanation: kiểu chuỗi ký tự giải thích cho từ khóa;

Tạo giao diện đơn giản để người dùng nhập các từ vào từ điển. Lưu trữ từ điển trong bảng băm và tạo một giao diện đơn giản cho người dùng có thể tra từ. Chương

trình phải cạnh cấp cơ chế lưu các từ đã có trong từ điển lên đĩa và đọc lại từ đĩa một từ điển có sẵn.

7. Vẽ cây có thứ tự từng phần được thiết lập bằng cách lần lượt đưa vào cây rỗng các khoá 5,9,6,4,3,1,7

8. Ta thấy rằng nếu ta lần lượt thực hiện DELETEMIN trên cây có thứ tự từng phần thì ta sẽ có một dãy các khoá có thứ tự tăng. Hãy dùng ý tưởng này để sắp xếp 1 dãy các số nguyên.

9. Giả lập việc quản lí các tiến trình thời gian thực (real-time processes):

Giả sử hệ điều hành phải quản lí nhiều tiến trình khác nhau, mỗi tiến trình có một độ ưu tiên khác nhau được tính theo một cách nào đó. Để đơn giản ta giả sử rằng mỗi tiến trình được quản lí như là một struct có hai trường:

Name: chuỗi ký tự;

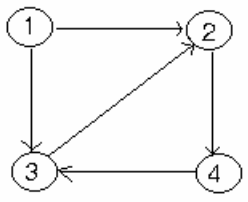
Priority: số thực ghi nhận mức độ ưu tiên của tiến trình;

Hãy cài đặt một hàng ưu tiên để quản lí các tiến trình này. Độ ưu tiên của các tiến trình dựa trên giá trị trường priority.

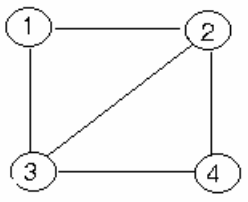
CHƯƠNG V ĐỒ THỊ (GRAPH)

I. CÁC ĐỊNH NGHĨA

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



Hình V.1a



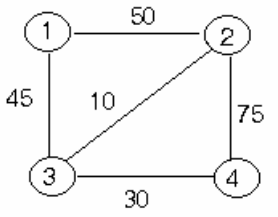
Hình V.1b

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

Một đường đi (path) trên đồ thị là một dãy tuần tự các đỉnh v_1, v_2,..., v_n sao cho (v_i,v_{i+1}) là một cung trên đồ thị (i=1,...,n-1). Đường đi này là đường đi từ v_1 đến v_n và đi qua các đỉnh v_2,...,v_{n-1}. Đỉnh v_1 còn gọi là đỉnh đầu, v_n gọi là đỉnh cuối. Độ dài của đường đi này bằng (n-1). Trường hợp đặc biệt dãy chỉ có một đỉnh v thì ta coi đó là đường đi từ v đến chính nó có độ dài bằng không. Ví dụ dãy 1,2,4 trong đồ thị V.1a là một đường đi từ đỉnh 1 đến đỉnh 4, đường đi này có độ dài là hai.

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

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



Hình V.2

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

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

II. KIỂU DỮ LIỆU TRỪU TƯỢNG ĐỒ THỊ

Các phép toán được định nghĩa trên đồ thị là rất đơn giản như là:

- Đọc nhãn của đỉnh.
- Đọc nhãn của cạnh.
- Thêm một đỉnh vào đồ thị.
- Thêm một cạnh vào đồ thị.
- Xoá một đỉnh.
- Xoá một cạnh.
- Lần theo (navigate) các cung trên đồ thị để đi từ đỉnh này sang đỉnh khác.

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

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

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

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

III. BIỂU DIỄN ĐỒ THỊ

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

1. Biểu diễn đồ thị bằng ma trận kề

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

j / i	0	1	2	3
0	true	true	true	**false**
1	true	true	true	true
2	true	true	true	true
3	**false**	true	true	true

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

j / i	0	1	2	3
0	1	1	1	0
1	0	1	0	1
2	0	1	1	0
3	0	0	0	1

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

```
const Null=0;
const n= …;
int a[n][n];        //mảng biểu diễn ma trận kề
```

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

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

Còn VERTEX(i) chỉ đơn giản là trả ra chính i.

Vòng lặp trên các đỉnh kề với v có thể cài đặt như sau

```
i=FIRST(v);
while (i!=Null)
{     w = VERTEX(i);
      //thao tác trên w
      i =NEXT(v,i);
}
```

Trên đồ thị có nhãn thì ma trận kề có thể dùng để lưu trữ nhãn của các cung chẳng hạn cung giữa i và j có nhãn a thì A[i,j]=a. Ví dụ ma trận kề của đồ thị hình V.2 là:

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

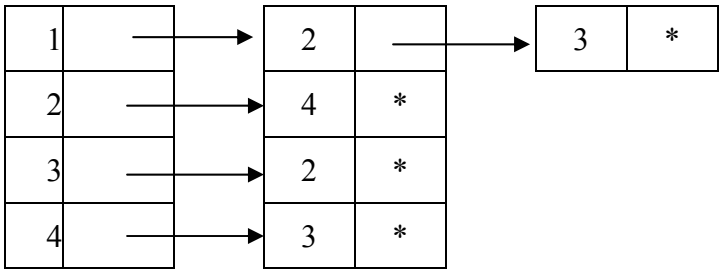
Ở đây các cặp đỉnh không có cạnh nối thì ta để trống, nhưng trong các ứng dụng ta có thể phải gán cho nó một giá trị đặc biệt nào đó để phân biệt với các giá trị có nghĩa khác. Chẳng hạn như trong bài toán tìm đường đi ngắn nhất, các giá trị số nguyên biểu diễn cho khoảng cách giữa hai thành phố thì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

Cách biểu diễn đồ thị bằng ma trận kề cho phép kiểm tra một cách trực tiếp hai đỉnh nào đó có kề nhau không. Nhưng nó phải mất thời gian duyệt qua toàn bộ mảng để xác

định tất cả các cạnh trên đồ thị. Thời gian này độc lập với số cạnh và số đỉnh của đồ thị. Ngay cả số cạnh của đồ thị rất nhỏ chúng ta cũng phải cần một mảng nxn phần tử để lưu trữ. Do vậy, nếu ta cần làm việc thường xuyên với các cạnh của đồ thị thì ta có thể phải dùng cách biểu diễn khác cho thích hợp hơn.

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

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



Mảng HEAD

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

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

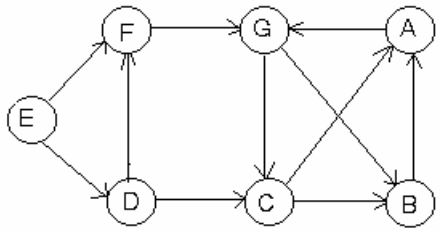
1. Duyệt theo chiều sâu (depth-first search)

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

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

Thủ tục dfs ở trong giải thuật ở trên có thể được viết như sau:

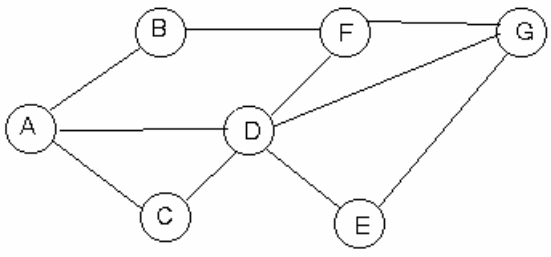
```
void dfs(vertex v)                // v ∈ [1..n]
{
vertex w;
      mark[v-1]=visited;
      for (mỗi đỉnh w là đỉnh kề với v)
            if (mark[w-1] == unvisited)
                  dfs(w);
}
```



Hình V.3

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

Ví dụ duyệt theo chiều sâu đồ thị hình V.4 bắt đầu từ đỉnh A: Duyệt A, A có các đỉnh kề là B,C,D; theo thứ tự đó thì B được duyệt. B có 1 đỉnh kề chưa duyệt là F, nên F được duyệt. F có các đỉnh kề chưa duyệt là D,G; theo thứ tự đó thì ta duyệt D. D có các đỉnh kề chưa duyệt là C,E,G; theo thứ tự đó thì C được duyệt. Các đỉnh kề với C đều đã được duyệt nên giải thuật được tiếp tục duyệt E. E có một đỉnh kề chưa duyệt là G, vậy ta duyệt G. Lúc này tất cả các nút đều đã được duyệt nên đồ thị đã được duyệt xong. Vậy thứ tự các đỉnh được duyệt là ABFDCEG.



Hình V.4

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

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

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

Thủ tục bfs được viết như sau:

```
void bfs(vertex v)              // v ∈ [1..n]
{
QUEUE of vertex Q;
vertex x,y;

      mark[v-1] = visited;
      ENQUEUE(v,Q);
      while !(EMPTY_QUEUE(Q)){
            x = FRONT(Q);
            DEQUEUE(Q);
            for (mỗi đỉnh y kề với x)
                  if (mark[y-1] == unvisited)
                  {
                        mark[y-1] = visited; {duyệt y}
                        ENQUEUE(y,Q);
                  }
      }
}
```

Ví dụ duyệt theo chiều rộng đồ thị hình V.3. Giả sử bắt đầu duyệt từ A. A chỉ có một đỉnh kề G, nên ta duyệt G. Kế đến duyệt tất cả các đỉnh kề với G; đó là B,C. Sau đó duyệt tất cả các đỉnh kề với B, C theo thứ tự đó. Các đỉnh kề với B, C đều đã được duyệt, nên ta tiếp tục duyệt các đỉnh chưa được duyệt. Các đỉnh chưa được duyệt là D, E, F. Duyệt D, kế đến là F và cuối cùng là E. Vậy thứ tự các đỉnh được duyệt là: AGBCDFE.

Ví dụ duyệt theo chiều rộng đồ thị hình V.4. Giả sử bắt đầu duyệt từ A. Duyệt A, kế đến duyệt tất cả các đỉnh kề với A; đó là B, C, D theo thứ tự đó. Kế tiếp là duyệt các đỉnh kề của B, C, D theo thứ tự đó. Vậy các nút được duyệt tiếp theo là F, E,G. Có thể minh hoạ hoạt động của hàng trong phép duyệt trên như sau:

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

A

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

B
C
D

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

C
D
F

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

D
F

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

F
E
G

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

E
G

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

V. MỘT SỐ BÀI TOÁN TRÊN ĐỒ THỊ

Phần này sẽ giới thiệu với các bạn một số bài toán quan trọng trên đồ thị, như bài toán tìm đường đi ngắn nhất, bài toán tìm bao đóng chuyển tiếp, cây bao trùm tối thiểu... Các bài toán này cùng với các giải thuật của nó đã được trình bày chi tiết trong giáo trình về Qui Hoạch Động, vì thế ở đây ta không đi vào quá chi tiết các giải thuật này. Phần này chỉ xem như là phần nêu các ứng dụng cùng với giải thuật để giải quyết các bài toán đó nhằm giúp bạn đọc có thể vận dụng được các giải thuật vào việc cài đặt để giải các bài toán nêu trên.

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

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

Ta có thể giải bài toán này bằng cách xác định một tập hợp S chứa các đỉnh mà khoảng cách ngắn nhất từ nó đến đỉnh nguồn v đã biết. Khởi đầu S={v}, sau đó tại mỗi bước ta sẽ thêm vào S các đỉnh mà khoảng cách từ nó đến v là ngắn nhất. Với giả thiết mỗi cung có một giá không âm thì ta luôn luôn tìm được một đường đi ngắn nhất như vậy mà chỉ đi

qua các đỉnh đã tồn tại trong S. Để chi tiết hoá giải thuật, giả sử G có n đỉnh và nhãn trên mỗi cung được lưu trong mảng hai chiều C, tức là C[i,j] là giá (có thể xem như độ dài) của cung (i,j), nếu i và j không nối nhau thì C[i,j]=∞. Ta dùng mảng 1 chiều D có n phần tử để lưu độ dài của đường đi ngắn nhất từ mỗi đỉnh của đồ thị đến v. Khởi đầu khoảng cách này chính là độ dài cạnh (v,i), tức là D[i]=C[v,i]. Tại mỗi bước của giải thuật thì D[i] sẽ được cập nhật lại để lưu độ dài đường đi ngắn nhất từ đỉnh v tới đỉnh i, đường đi này chỉ đi qua các đỉnh đã có trong S.

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

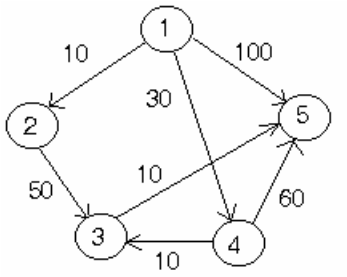
```
void Dijkstra()
{
    S = [1]; //Tập hợp S chỉ chứa một đỉnh nguồn
    for (i =2; i<=n; i++)
        D[i-1] = C[0,i-1]; //khởi đầu các giá trị cho D
    for (i=1; i<n; i++)
    {
        Lấy đỉnh w trong V-S sao cho D[w-1] nhỏ nhất;
        Thêm w vào S;
        for (mỗi đỉnh u thuộc V-S)
      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

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

Giải thuật Dijkstra được viết lại như sau:

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

Ví dụ: áp dụng giải thuật Dijkstra cho đồ thị hình V.5



Hình V.5

Kết quả khi áp dụng giải thuật

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

Mảng P có giá trị như sau:

P	1	2	3	4	5
		1	4	1	3

Từ kết quả trên ta có thể suy ra rằng đường đi ngắn nhất từ đỉnh 1 đến đỉnh 3 là

$1 \rightarrow 4 \rightarrow 3$ có độ dài là 40. đường đi ngắn nhất từ 1 đến 5 là $1 \rightarrow 4 \rightarrow 3 \rightarrow 5$ có độ dài 50.

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

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

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

void Floyd(){
int i,j,k;
```

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

3. Bài toán tìm bao đóng chuyển tiếp (transitive closure)

Trong một số trường hợp ta chỉ cần xác định có hay không có đường đi nối giữa hai đỉnh i,j bất kỳ. Giải thuật Floyd có thể đặc biệt hoá để giải bài toán này. Bây giờ khoảng cách giữa i,j là không quan trọng mà ta chỉ cần biết i,j có nối nhau không do đó ta cho C[i,j]=1 (~true) nếu i,j được nối nhau bởi một cạnh, ngược lại C[i,j]=0 (~false). Lúc này mảng A[i,j] không cho khoảng cách ngắn nhất giữa i,j mà nó cho biết là có đường đi từ i đến j hay không. A gọi là bao đóng chuyển tiếp của đồ thị G có biểu diễn ma trận kề là C. Giải thuật Floyd sửa đổi như trên gọi là giải thuật Warshall.

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

4. Bài toán tìm cây bao trùm tối thiểu (minimum-cost spanning tree)

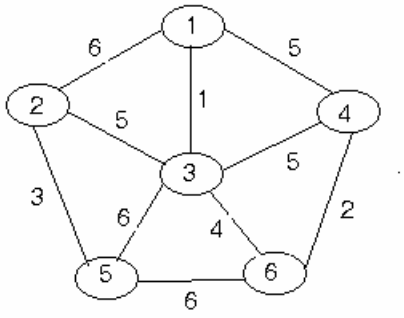
Giả sử ta có một đồ thị vô hướng G=(V,E). Đồ thị G gọi là liên thông nếu tồn tại đường đi giữa hai đỉnh bất kỳ. Bài toán tìm cây bao trùm tối thiểu (hoặc cây phủ tối thiểu) là tìm một tập hợp T chứa các cạnh của một đồ thị liên thông G sao cho V cùng với tập các cạnh này cũng là một đồ thị liên thông, tức là (V,T) là một đồ thị liên thông. Hơn nữa tổng độ dài các cạnh trong T là nhỏ nhất. Một thể hiện của bài toán này trong thực tế là bài toán thiết lập mạng truyền thông, ở đó các đỉnh là các thành phố còn các cạnh của cây bao trùm là đường nối mạng giữa các thành phố.

Giả sử G có n đỉnh được đánh số 1..n. Giải thuật Prim để giải bài toán này như sau:

Bắt đầu, tập ta khởi tạo tập U bằng 1 đỉnh nào đó, đỉnh 1 chẳng hạn, U = {1}, T=U

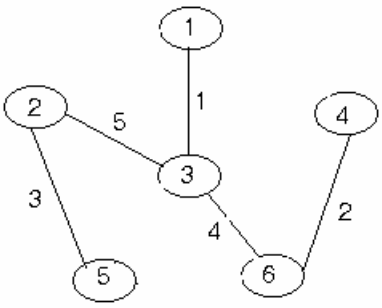
Sau đó ta lặp lại cho đến khi U=V, tại mỗi bước lặp ta chọn cạnh nhỏ nhất (u,v) sao cho u ∈ U và v ∈ V-U. Thêm v vào U và (u,v) vào T. Khi giải thuật kết thúc thì (U,T) là một cây phủ tối tiểu.

Ví dụ, áp dụng giải thuật Prim để tìm cây bao trùm tối thiểu của đồ thị liên thông hình V.6.



Hình V.6

- Bước khởi đầu: U={1}, T=∅.
- Bước kế tiếp ta có cạnh (1,3)=1 là cạnh ngắn nhất thoả mãn điều kiện trong giải thuật Prim nên: U={1,3}, T={(1,3)}.
- Kế tiếp thì cạnh (3,6)=4 là cạnh ngắn nhất thoả mãn điều kiện trong giải thuật Prim nên: U={1,3,6}, T={(1,3),(3,6)}.
- Kế tiếp thì cạnh (6,4)=2 là cạnh ngắn nhất thoả mãn điều kiện trong giải thuật Prim nên: U={1,3,6,4}, T={(1,3),(3,6),(6,4)}.
- Tiếp tục, cạnh (3,2)=5 là cạnh ngắn nhất thoả mãn điều kiện trong giải thuật Prim nên: U={1,3,6,4,2}, T={(1,3),(3,6),(6,4),(3,2)}.
- Cuối cùng, cạnh (2,5)=3 là cạnh ngắn nhất thoả mãn điều kiện trong giải thuật Prim nên: U={1,3,6,4,2,5}, T={(1,3),(3,6),(6,4),(3,2),(2,5)}. Giải thuật dừng và ta có cây bao trùm như trong hình V.7.



Hình V.7

Giải thuật Prim được viết lại như sau:

```
void Prim(graph G, set_of_edges& T)
{
    set_of_vertices U;          //tập hợp các đỉnh
    vertex u,v;                 //u,v là các đỉnh
    T = ∅;
    U = [1];
    while (U≠V) do  // V là tập hợp các đỉnh của G
```

```
        {
    gọi (u,v) là cạnh ngắn nhất sao cho u ∈ U và v ∈ V-U;
                U = U ∪ [v];
                T = T ∪ [(u,v)];
        }
    }
```

Bài toán cây bao trùm tối thiểu còn có thể được giải bằng giải thuật Kruskal như sau:

Khởi đầu ta cũng cho T= ∅ giống như trên, ta thiết lập đồ thị khởi đầu G'=(V,T).

Xét các cạnh của G theo thứ tự độ dài tăng dần. Với mỗi cạnh được xét ta sẽ đưa nó vào T nếu nó không làm cho G' có chu trình.

Ví dụ áp dụng giải thuật Kruskal để tìm cây bao trùm cho đồ thị hình V.6.

Các cạnh của đồ thị được xếp theo thứ tự tăng dần là.

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

- Bước khởi đầu T= ∅
- Lần lặp 1: T={(1,3)}
- Lần lặp 2: T={(1,3),(4,6)}
- Lần lặp 3: T={(1,3),(4,6),(2,5)}
- Lần lặp 4: T={(1,3),(4,6),(2,5),(3,6)}
- Lần lặp 5:

  Cạnh (1,4) không được đưa vào T vì nó sẽ tạo ra chu trình 1,3,6,4,1.

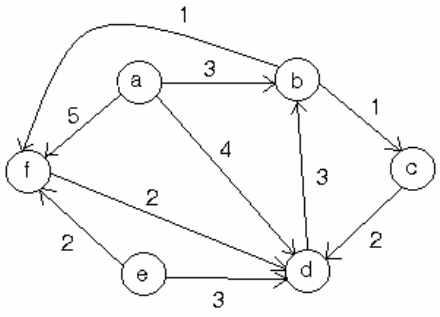
  Kế tiếp cạnh (2,3) được xét và được đưa vào T.

T={(1,3),(4,6),(2,5),(3,6),(2,3)}

Không còn cạnh nào có thể được đưa thêm vào T mà không tạo ra chu trình. Vậy ta có cây bao trùm tối thiểu cũng giống như trong hình V.7.

BÀI TẬP

1. Viết biểu diễn đồ thị V.8 bằng:
   - Ma trận kề.
   - Danh sách các đỉnh kề.



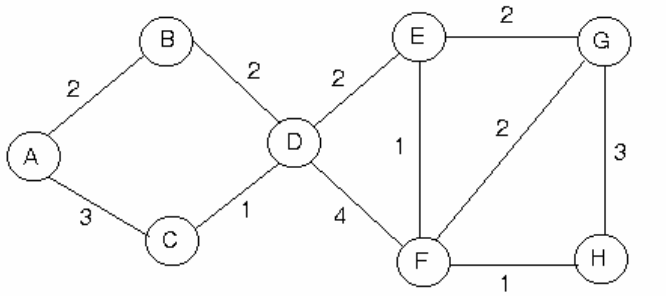
Hình V.8

2. Duyệt đồ thị hình V.8 (xét các đỉnh theo thứ tự a,b,c...)

   - Theo chiều rộng bắt đầu từ a.

   - Theo chiều sâu bắt đầu từ f

3. Áp dụng giải thuật Dijkstra cho đồ thị hình V.8, với đỉnh nguồn là



Hình V.9

a.

4. Viết biểu diễn đồ thị V.9 bằng:
   Ma trận kề.
   Danh sách các đỉnh kề.
5. Duyệt đồ thị hình V.9 (xét các đỉnh theo thứ tự A,B,C...)
   Theo chiều rộng bắt đầu từ A.
   Theo chiều sâu bắt đầu từ B.
6. Áp dụng giải thuật Dijkstra cho đồ thị hình V.9, với đỉnh nguồn là A.
7. Tìm cây bao trùm tối thiểu của đồ thị hình V.9 bằng
   Giải thuật Prim.
   Giải thuật Kruskal.
8. Cài đặt đồ thị có hướng bằng ma trận kề rồi viết các giải thuật:
   Duyệt theo chiều rộng.
   Duyệt theo chiều sâu.
   Tìm đường đi ngắn nhất từ một đỉnh cho trước (Dijkstra).
   Tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh (Floyd).

9. Cài đặt đồ thị có hướng bằng danh sách các đỉnh kề rồi viết các giải thuật:
   Duyệt theo chiều rộng.
   Duyệt theo chiều sâu.
10. Cài đặt đồ thị vô hướng bằng ma trận kề rồi viết các giải thuật:
   Duyệt theo chiều rộng.
   Duyệt theo chiều sâu.
   Tìm đường đi ngắn nhất từ một đỉnh cho trước (Dijkstra).
   Tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh (Floyd).
   Tìm cây bao trùm tối thiểu (Prim, Kruskal).
   Cài đặt thuật toán Greedy cho bài toán tô màu đồ thị (Gợi ý: xem giải thuật trong chương 1)
11. Cài đặt đồ thị vô hướng bằng danh sách các đỉnh kề rồi viết các giải thuật:
   Duyệt theo chiều rộng.
   Duyệt theo chiều sâu.
12. Hãy viết một chương trình, trong đó, cài đặt đồ thị vô hướng bằng cấu trúc ma trận kề rồi viết các thủ tục/hàm sau:

Nhập toạ độ n đỉnh của đồ thị.

Giả sử đồ thị là đầy đủ, tức là hai đỉnh bất kỳ đều có cạnh nối, và giả sử giá của mỗi cạnh là độ dài của đoạn thẳng nối hai cạnh. Hãy tìm:

Đường đi ngắn nhất từ một đỉnh cho trước (Dijkstra).

Đường đi ngắn nhất giữa tất cả các cặp đỉnh (Floyd).

Cây bao trùm tối thiểu (Prim, Kruskal).

Thể hiện đồ thị lên màn hình đồ hoạ, các cạnh thuộc cây bao trùm tối thiểu được vẽ bằng một màu khác với các cạnh khác.