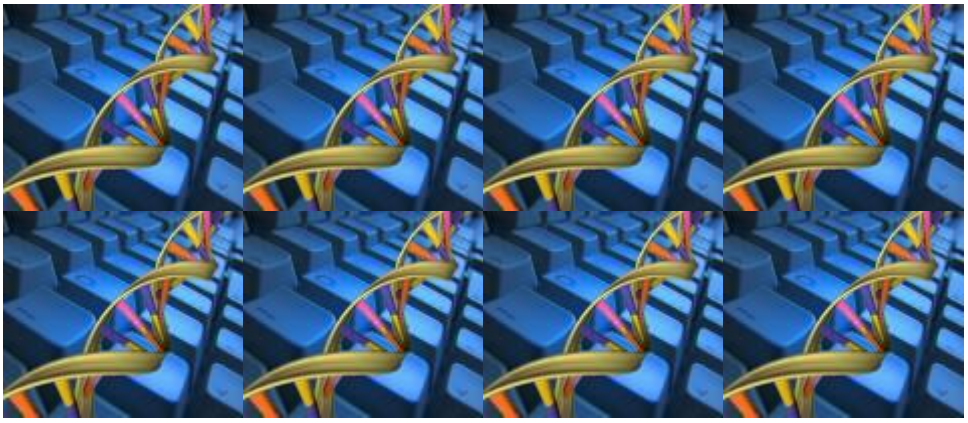
Giáo trình

CẤU TRÚC DỮ LIỆU

DATA STRUCTURES



TRẦN CAO ĐỆ

2012

MỤC LỤC

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 và Truyền thông - Trường Đại học Cần Thơ đã tiến hành biên soạn các giáo trình, bài giảng chính trong chương trình học. Giáo trình môn Cấu trúc Dữ liệu 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.

Giáo trình 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 là giúp các bạn sinh viên chuyên ngành có một tài liệu cô đọng dùng làm tài liệu học tập. Tuy vậy, các đối tượng sinh viên khác vẫn có thể sử dụng giáo trình này làm tài liệu tham khảo hay tự học. Chúng tôi nghĩ rằng các bạn sinh viên không chuyên tin và những người quan tâm tới các cấu trúc dữ liệu và giải thuật sẽ tìm được trong này những điều hữu ích.

Giáo trình được phát hành nội bộ lần đầu tiên vào năm 1998, và được chỉnh sửa, tu bổ hàng năm. Lần chỉnh sửa vào năm 2007 (và hoàn chỉnh năm 2008) đã 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. Việc cài đặt các cấu trúc dữ liệu trong một ngôn ngữ lập trình cụ thể như C hay Pascal không phải là mục đích chính của giáo trình. Tuy nhiên, hầu hết các đoạn chương trình được viết trong giáo trình đã được biên dịch trong môi trường Visual C++ 6.0.

Nội dung giáo trình môn Cấu trúc Dữ liệu này sẽ cung cấp cho sinh viên các kiến thức cơ bản về các kiểu dữ liệu trừu tượng và các phép toán cơ bản trên các kiểu dữ liệu trừu tượng 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 trong thực tế hay trong Khoa học máy tính.

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

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

- 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/C++ đó là một ngôn ngữ quen thuộc đối với sinh viên chuyên ngành Tin học.
- Kiến thức toán rời rạc.

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. 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/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/C++ đồng thời cũng nhằm minh họa 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. 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 một cách hiệu quả.

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 và bằng danh sách có hoặc không có thứ tự. Phần chính của chương này trình bày cấu trúc dữ liệu từ điển, đó là tập hợp với ba phép toán thêm, xóa 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, tức là cài đặt đồ thị. Các phép duyệt đồ thị bao gồm duyệt theo chiều rộng và duyệt theo chiều sâu một đồ thị cũng được trình bày trong chương. 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 chỗ 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.

Chúng tôi hy vọng tài liệu này đã đúc kết kinh nghiệm giảng dạy nhiều năm môn Cấu trúc dữ liệu và Giải thuật của chúng tôi. Mặc dù đã rất cố gắng trong quá trình biên soạn, nhưng giáo trình có thể vẫn còn nhiều khiếm khuyết 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.

CHƯƠNG I: MỞ ĐẦU

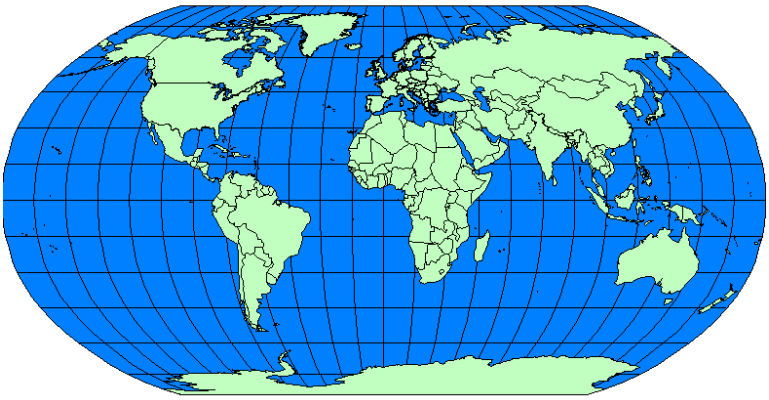
I. TỪ BÀI TOÁN ĐẾN CHƯƠNG TRÌNH

1. Mô hình hóa bài toán

Để giải một bài toán trong thực tế bằng máy tính (computer) 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: vấn đề tô màu bản đồ thế giới:

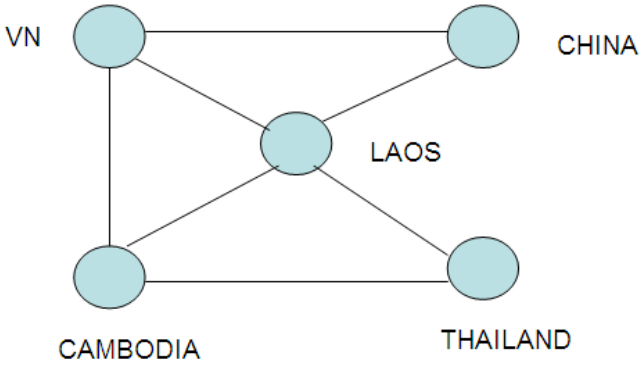
Giả sử ta cần phải tô màu cho các nước trên bản đồ thế giới (hình I.1). Trong đó mỗi nước được tô một màu và hai nước láng giềng (có biên giới chung) thì phải được tô bằng hai màu khác nhau. Hãy tìm một phương án tô màu các nước trên bản đồ sao cho số màu sử dụng là ít nhất.



Hình I.1: Bản đồ thế giới.

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 đoạn thẳng hay cong tùy thích, gọi là cung hay cạnh của đồ thị. Chẳng hạn, một phần của bản đồ thế giới được cho trong hình I.2 có thể mô hình hóa bằng một đồ thị như trong hình I.3.

Hình I.2: Một phần của bản đồ thế giới.



Hình I.3: Minh họa việc mô hình hóa cho phần bản đồ thế giới trong hình I.2.

Bài toán "Tô màu cho bản đồ thế giới" trở thành:

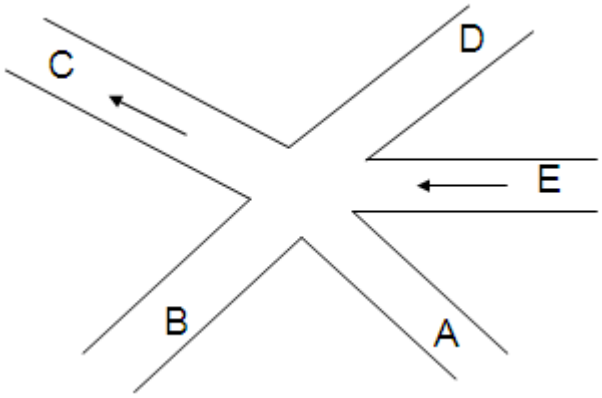
- Tìm cách tô màu cho tất cả các đỉnh đồ thị sao cho hai đỉnh có cạnh nối nhau thì phải được tô bằng hai màu khác nhau;
- Số màu được sử dụng là ít nhất.

Rõ ràng, giờ đây bài toán tô màu đặt ra ban đầu đã có vẻ "toán học" hơn và nó xác định cụ thể hơn vấn đề "phải làm gì?". Việc làm sao để giải được bài toán này, tức là "làm như thế nào?" còn cần phải được tiếp tục tháo gỡ.

Ví dụ 2: Đèn giao thông

Cho một ngã năm như hình I.4, 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.4: Ví dụ một ngã năm.

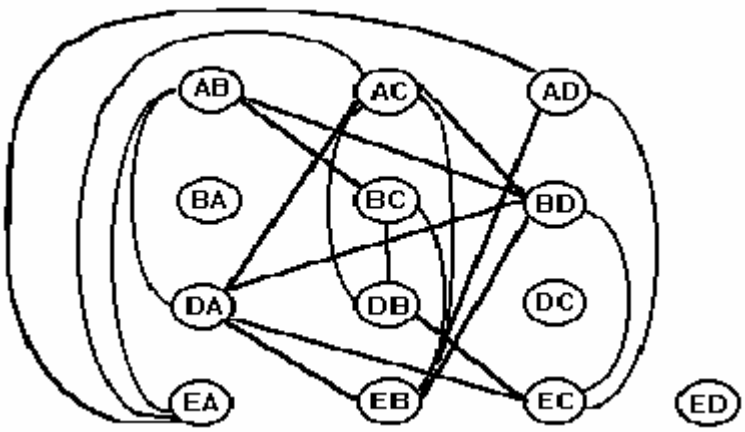
Tương tự như trong ví dụ 1, trước hết ta tìm cách mô hình hóa bài toán, tức là tìm cách trừu tượng hóa để gạt bỏ các chi tiết không cần thiết. Để giải bài toán vừa đặt ra, ít nhất ta phải biết ở ngã năm có bao nhiêu lối đi, lối đi nào với lối đi nào có hướng giao thông không cắt nhau, lối đi nào với lối đi nào có hướng giao thông cắt nhau.

Nhận thấy rằng tại ngã năm này có 13 lối đi: AB, AC, AD, BA, BC, BD, DA, DB, DC, EA, EB, EC, ED. 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 tùy thích. Ta sẽ có một sơ đồ như hình I.5. 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 được gọi là cạnh của đồ thị. Bây giờ ta phải xác định các nhóm, với số nhóm ít nhất. Mỗi

nhóm sẽ 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.



Hình I.5: Mô hình giao thông tại ngã năm trong hình I.4

Tóm lại, ta phải giải quyết bài toán sau:

Tô màu cho đồ thị ở hình I.5 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.

Đến đây, bài toán "Đèn giao thông" đã được phát biểu thành một bài toán đồ thị. Ta cũng thấy rằng 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 phương pháp hoặc một chương trình có sẵn để giải. Nếu không có một phương pháp hoặc 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 một công việc nào đó 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.

Cho đến nay, 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 trùng khớp với số màu cần dùng trong 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.

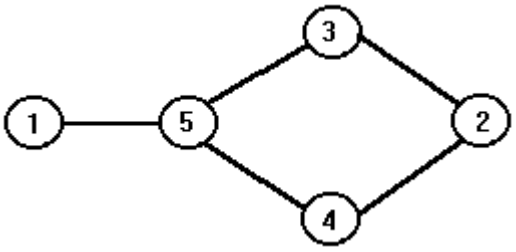
HEURISTIC cho bài toán tô màu đồ thị, thường gọi là giải thuật "háu ăn" (GREEDY) là:

- *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.6 và cách tô màu cho nó



Hình I.6: Đồ thị minh họa cách tô màu theo "háu ăn" (GREEDY).

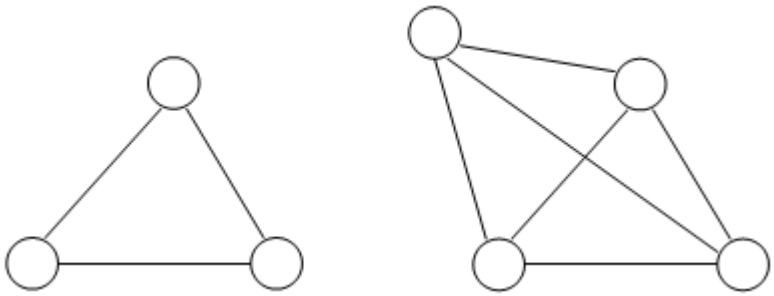
Tô theo GREEDY (xét lần lượt theo số thứ tự các đỉnh)	Tối ưu (thử tất cả các khả năng)
1: đỏ; 2: đỏ 3: xanh; 4: xanh 5: vàng	1, 3, 4 : đỏ 2, 5 : xanh

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.5 (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.5. 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:



Hình I.7: Ví dụ về các đồ thị có nối kết đầy đủ.

- 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.7 chỉ ra hai ví dụ với k=3 và k=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.5 có 4 đỉnh: AC, DA, BD, EB mà mỗi cặp đỉnh bất kỳ đều được nối nhau vậy đồ thị hình I.5 không thể tô với ít hơn 4 màu. Điều này cho phép kết luận rằng không thể dùng ít hơn 4 màu để tô cho đồ thị hình I.5. Nói cách khác, lời giải vừa tìm được ở trên, với 4 màu, cũng là 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.

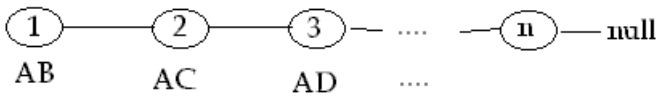
```
PROCEDURE GREEDY ( var G: GRAPH ; var Newclr: SET );
begin
{1}     Newclr := Ø;
{2}     for (mỗi đỉnh v chưa tô màu của G) do
{3}             if (v không được nối với một đỉnh nào trong Newclr) then begin
{4}                     đánh dấu v đã được tô màu;
{5}                     thêm v vào Newclr;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

Thủ tục GREEDY được viết với ngôn ngữ giả PASCAL

Trong thủ tục bằng ngôn ngữ giả ở trên, chúng ta đã dùng một số từ khóa 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 thành chương trình chạy được trên máy tính, 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:= Ø;
{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 được tinh chế mệnh đề if ở {3}



Hình I.8: Biểu diễn tập hợp các đỉnh như là một danh sách (LIST).

Bây giờ ta hãy quan tâm một chút đến các kiểu dữ liệu trừu tượng GRAPH và SET. Trong giải thuật viết bằng ngôn ngữ giả ở trên, GRAPH được dùng để chỉ « kiểu » của đồ thị đầu vào còn SET để chỉ « kiểu » của tập hợp đầu ra. Một cách nôm na ta có thể coi GRAPH như là tập hợp các đỉnh và tập hợp các cạnh của đồ thị đầu vào còn SET là tập hợp các đỉnh ở đầu ra. Để thảo luận không quá sa đà vào cài đặt, ta hãy gác lại việc biểu diễn tập hợp các cạnh để tập trung vào tập hợp các đỉnh. Có nhiều cách để biểu diễn tập hợp các đỉnh trong ngôn ngữ lập trình. Đơn giản, ta xem các tập hợp đỉnh 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.8). 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:= Ø;
        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 được 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 ngôn 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 = Ø;
/*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= Ø;
/*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;
                  }
        }
}
```

Thủ tục có thể được 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, 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. Bước đầu, 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. Hình thức hoá giải thuật bằng cách viết một thủ tục bằng ngôn ngữ giả, rồi chi tiết hoá dần ("mịn dần") 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, Java ...

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 vấn đề (hoặc đối tượng) sáng sủa hơn và dễ hiểu hơn bằng cách nắm bắt những cái cơ bản ở mức tổng thể, gạt bỏ các chi tiết thừa. 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. Ví dụ, các phép toán số học (cộng, trừ, nhân, chia) trong máy tính đã là sự trừu tượng hóa của các phép toán trong bộ nhớ trên bít, byte, dịch chuyển. Ví dụ khác, 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* tương tự như GRAPH và SET ở trên.

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 họa 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 (trừu tượng) cần thiết cùng với các phép toán 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.

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 khá giống 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 (trong PASCAL) 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 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 ngôn ngữ lập trình) 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" (primitive) của ngôn ngữ lập trình 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.

Trong giáo trình, chúng tôi dùng thuật ngữ "kiểu dữ liệu trừu tượng" để nhấn mạnh vào tính trừu tượng của nó, nhất là ở giai đoạn xây dựng khái niệm trừu tượng. Tuy nhiên, về sau, khi cấu trúc đã được cài đặt (theo một cách nào đó) thì cấu trúc trừu tượng đã được thể hiện bằng một kiểu dữ liệu phức hợp cụ thể và vì vậy nó có tính "cấu trúc dữ liệu" hơn là "trừu tượng". Chúng tôi sẽ dùng trộn lẫn hai thuật ngữ này để chỉ các kiểu dữ liệu trừu tượng và các cấu trúc xây dựng nên từ cài đặt các kiểu dữ liệu trừu tượng này; ví dụ, "kiểu dữ liệu trừu tượng danh sách" hoặc "cấu trúc dữ liệu danh sách" hay "cấu trúc danh sách" cho gọn nhẹ.

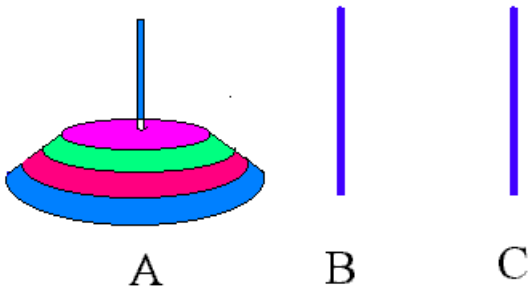
TỔNG KẾT CHƯƠNG

Chương này đã trình bày cách tiếp cận từ bài toán đến chương trình, trước hết, đó là việc mô hình hóa bài toán. Sau khi bài toán đã được mô hình hóa, ta hình thức hóa một giải thuật để giải bài toán dựa trên kiểu dữ liệu trừu tượng. Giải thuật có thể được diễn đạt bằng ngôn ngữ giả và được chi tiết hóa dần (mịn hóa). Trong khi xây dựng một giải thuật ta chỉ tập trung vào mô hình của bài toán, các phép toán trên mô hình đó ở mức trừu tượng, chưa được cài đặt trong chương trình. Về sau, khi cài đặt kiểu dữ liệu trừu tượng (mô hình và các phép toán trừu tượng) trong một ngôn ngữ lập trình, chúng ta cần lựa chọn các cấu trúc dữ liệu thích hợp (được cung cấp trong ngôn ngữ lập trình). Các cài đặt một cho kiểu dữ liệu trừu tượng có thể khác nhau tùy theo cấu trúc dữ liệu được lựa chọn. Tuy nhiên các phép toán trừu tượng là trong suốt đối với người dùng và không phụ thuộc vào các cấu trúc dữ liệu. Chính vì thế các phép toán trừu tượng có thể được dùng như là các "nguyên sơ" trong thiết kế giải thuật về sau.

BÀI TẬP

1. Nêu khái niệm kiểu dữ liệu, kiểu dữ liệu trừu tượng, cấu trúc dữ liệu. Cho ví dụ cụ thể để minh họa từng khái niệm.
2. Nêu khái niệm trừu tượng hóa.
    a. Cho ví dụ về trừu tượng hóa chương trình, ý nghĩa của việc trừu tượng hóa chương trình
    b. Cho ví dụ về trừu tượng hóa dữ liệu, ý nghĩa của việc trừu tượng hóa này.
3. Hãy dùng cách tiếp cận từ bài toán đến chương trình để giải bài toán Tháp Hà Nội:

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, có thể dùng cọc C làm trung gian. Mỗi lần thực hiện, chỉ chuyển một đĩa từ một cọc sang một cọc khác và không được đặt đĩa lớn nằm trên đĩa nhỏ (hình I.9)



Hình I.9: Minh họa bài toán Tháp Hà Nội.

a. Mô hình hóa bài toán
b. Viết giải thuật thích hợp cho bài toán này
c. Cài đặt giải thuật thành chương trình.

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 (element type). 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 Dũng
2. Nguyễn Ngọc Nương
3. Lê Thị Thanh
4. Trịnh Minh Thành
5. Nguyễn Phú Vinh

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. Chẳng hạn, "Nguyễn Ngọc Nương" có vị trí là 2; "Nguyễn Phú Vinh" ở vị trí cuối cùng trong danh sách.

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 (Position); 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 nhằm để chỉ vị trí của phần tử, nó không phải là giá trị của các phần tử trong danh sách. Vị trí không nhất thiết phải là số nguyên.

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 x vào vị trí p 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 x (có kiểu là kiểu phần tử - ElementType) 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 và giá trị trả về là NULL).
- **DELETE_LIST(p,L)** chương trình con thực hiện việc xóa 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.

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 (tức là có thể áp dụng phép so sánh >). Ta cũng giả sử hàm SWAP(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);
        }
}
```

Cần phải nhấn mạnh rằng, các phép toán trừu tượng trên (FIRST, NEXT, RETRIEVE,…) do chúng ta định nghĩa, nó chưa được cài đặt trong các ngôn ngữ lập trình. Do đó để chương trình con SORT nói trên có thể chạy được, ta phải cài đặt các phép toán này tthành 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 (tức là không được dùng), 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 giáo trình này, ta vẫn giữ đúng những tham số hình thức đã định nghĩa trong phép toán trừu tượng trong khi 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 với ngôn ngữ lập trình C, các biến trong chương trình con không được dùng sẽ được (hoặc bị) cảnh báo dạng "biến này được không dùng". Đó không phải là lỗi biên dịch của chương trình, nên độc giả không cần phải quan tâm sửa lỗi.

3. Cài đặt danh sách

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

Danh sách có thể được cài đặt 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, 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 luôn 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 trong trường hợp này "vị trí" được định nghĩa là số nguyên. Cũng cần phải thấy rằng vị trí này không nhất thiết phải là chỉ số của mảng tại nơi lưu trữ phần tử. Chẳng hạn trong ngôn ngữ C, chỉ số của mảng bắt đầu từ 0, vì vậy phần tử thứ 1 của danh sách có (vị trí là 1) nhưng nằm trong mảng tại chỉ số 0, phần tử thứ i (vị trí là i) nằm trong mảng tại chỉ số (i-1). Trong ngôn ngữ lập trình PASCAL mảng có thể bắt đầu bằng số nguyên tùy ý, ví dụ A[10..100] nên phần tử thứ nhất của danh sách nằm tại chỉ số A[10] chứ không phải là A[1]! Điểm đáng lưu tâm nhất là các phần tử phải được đặt vào mảng liên tiếp nhau, không có "ô trống" trong mảng từ đầu cho đến cuối danh sách. Chính vì vậy danh sách với cách cài đặt này còn được gọi là *danh sách đặc*, tức là "không rỗng ở giữa".

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 (danh sách đặc).

Với hình ảnh minh họa trên, 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 của 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 tượng danh sách bằng cấu trúc dữ liệu mảng. Phần tiếp theo là các mã (code) 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, tức là 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 là 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;
}
```

Các bạn mới làm quen với lập trình C cần lưu tâm tới cú pháp khai báo truyền tham số trong thủ tục trên. Dấu "&" nhằm để khai báo truyền tham số theo kiểu tham chiếu, tức là các thay đổi trong chương trình con sẽ còn hiệu lực khi chương trình con kết thúc (ở đây biến Last sẽ có giá trị 0 ngay cả sau khi chương trình con kết thúc). Thực ra đó là cú pháp của C++. Tuy nhiên ta sẽ không đề cập nhiều đến các khía cạnh của ngôn ngữ lập trình ở đây. Các đoạn chương trình trong giáo trình này được biên dịch với Visual C++ 6.0.

Kiểm tra danh sách rỗng

Danh sách rỗng là một danh sách mà độ dài của nó bằng 0. Vì vậy để kiểm tra danh sách rỗng ta cần kiểm tra độ dài này. Hàm EMPTY_LIST sẽ trả ra giá trị 0 hoặc 1 tùy theo giá trị so sánh của Last với 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, ta cần xét 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 độ dài 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. Lúc này thao tác xen sẽ không được làm và danh sách L sẽ không thay đổi.
- 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 vị trí cuối cùng của danh sách ra sau 1 vị trí.
        - Đưa phần tử mới vào vị trí p
        - Độ dài danh sách tăng 1.

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++;
                }
}
```

Lưu ý rằng : để dời các phần tử từ vị trí p đến vị trí cuối cùng của danh sách ra sau 1 vị trí, thì các phần tử phải được dời một cách có hệ thống; phần tử cuối cùng phải di

chuyển trước, kế đến là các phần tử trước nó một cách tuần tự như đã được cài đặt ở trên.

Xóa phần tử ra khỏi danh sách

Để xóa 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 phần tử có vị trí cuối cùng trong danh sách ra phía trước một vị trí và độ dài danh sách giảm đi 1 phần tử.

Thủ tục xóa phần tử có vị trí p trong danh sách L có thể được cài đặt như sau :

```
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 về trước 1 vị trí*/
                 for(Q=P;Q<L.Last;Q++)
                        L.Elements[Q-1]=L.Elements[Q];
                 L.Last--;
        }
}
```

Một lần nữa, xin lưu ý rằng các phần tử cần được di dời một cách có hệ thống. Phần tử (p+1) được di dời, kế đến là (p+2),..., cho đến phần tử cuối cùng trong danh sách.

Định vị một phần tử trong danh sách

Để định vị phần tử x trong danh sách L (tìm 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 bằng mảng ở đây thì ENDLIST(L) là 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. Nói cách khác, chương trình con sẽ tìm x một cách tuần tự cho đến khi gặp x; hoặc là đi đến sau vị trí phần tử cuối cùng nếu không gặp x.

```
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; //ngưng và trả ra vị trí P
```

```
                else
                        P ++ ; //tìm tiếp tục sang vị trí kế tiếp
}
```

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

- 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

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 các phép toán trừu tượng của danh sách)

- Hiển thị danh sách ra màn hình, hay in danh sách (PRINT_LIST(L)). Phép toán này cũng chưa có trong các phép toán trừu tượng của 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");
}
```

Bây giờ 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;
```

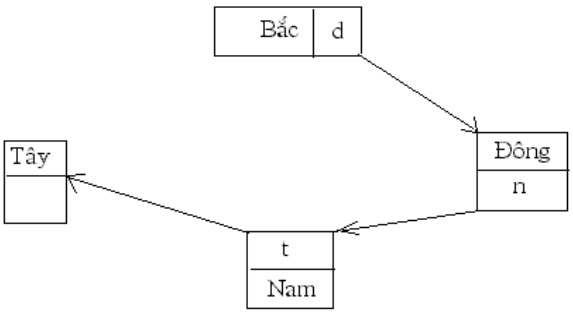
```
        Position P;
        MAKENULL_LIST(L); //Khởi tạo danh sách rỗng
        READ_LIST(L);
        printf("Danh sach vua nhap: ");
        PRINT_LIST(L); // In danh sach len man hinh
        printf("Phan tu can them: ");scanf("%d",&X);
        printf("Vi tri xen phan tu : ");scanf("%d",&P);
        INSERT_LIST(X,P,L);
        printf("Danh sach sau khi them phan tu la: ");
        PRINT_LIST(L);
        printf("Phan tu muon 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;
}
```

Ví dụ này một lần nữa cho thấy rằng kiểu dữ liệu trừu tượng cho phép trừu tượng hóa chương trình. Nó che đi các cài đặt cụ thể của chương trình với một cấu trúc dữ liệu cụ thể. Các phép toán trừu tượng sau khi được cài đặt bằng một cấu trúc dữ liệu phù hợp, nó sẽ được dùng như các nguyên sơ (primitive) khi cần thiết. Chẳng hạn như trong chương trình chính ở trên, chúng ta không cần quan tâm đến việc xen hay xóa một phần tử được thực hiện như thế nào, đơn giản chỉ cần gọi các phép toán INSERT_LIST hay DELETE_LIST với các tham số thích hợp. Từ rày về sau, các phép toán trừu tượng có thể được dùng như là chúng đã có sẵn trong ngôn ngữ lập trình.

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

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:

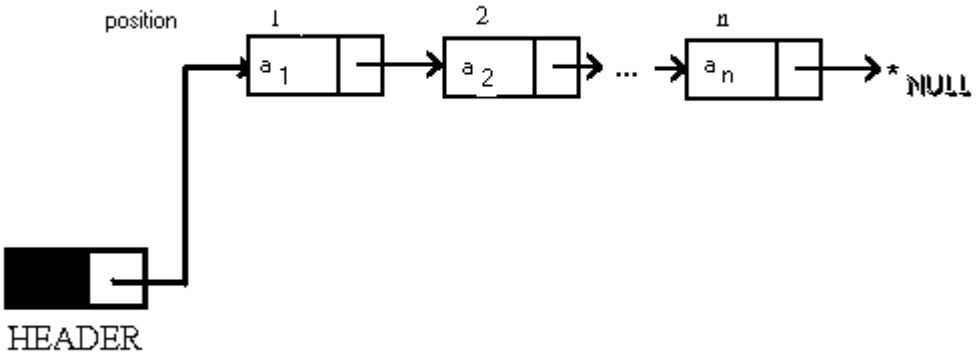
Giả sử 1 lớp có 4 bạn: Đông, Tây, Nam, Bắc có địa chỉ lần lượt là d,t,n,b. Giả sử: Đông có địa chỉ của Nam, Tây không có địa chỉ của bạn nào, Bắc giữ địa chỉ của Đông, Nam có địa chỉ của Tây (xem hình II.2). Một bạn A giữ địa chỉ của bạn B thì coi như là A « chỉ đến » B, chẳng hạn Bắc chỉ đến Đông như mũi tên trong hình II.2.



Hình II.2 : Cơ chế liên kết các phần tử (« chỉ đến »)

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 thể tìm được danh sách cả 4 bạn này thì ta cần và **chỉ cần** giữ địa chỉ của người đầu tiên (tức là 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 struct) 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.** Cách cài đặt danh sách như vậy được gọi là cách cài đặt bằng con trỏ và danh sách được cài đặt như vậy gọi là *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 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, 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 của danh sách. Nếu danh sách rỗng thì *Header->Next* sẽ là **`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, xóa các phần tử trong danh sách.

Ở đây ta cần phân biệt rõ giá trị của một phần tử (tức là nội dung) 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 (tức là con trỏ) 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 trong bảng 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 (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 xóa 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* trong ô chứa phần tử thứ (p-1). Một cách chính xác, để truy cập vào phần tử p ta cần biết địa chỉ ô nhớ chứa phần tử p. Đây là thực chất là địa chỉ máy nên ta truy cập thông qua con trỏ trỏ đến phần tử p. Con trỏ trỏ đến phần tử p nằm trong trường *Next* của ô chứa phần tử (p-1). Cho nên để đơn giản trong các phép toán, ta định nghĩa **vị trí của phần tử p là địa chỉ ô nhớ chứa con trỏ trỏ đến ô chứa phần tử thứ p.** Rõ ràng địa chỉ này chính là địa chỉ của ô chứa phần tử thứ (p-1). Có thể nói diễn dịch rằng vị trí của phần tử thứ 1 là con trỏ trỏ vào *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, 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 xác định 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;
```

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* trỏ đến **NULL.** Do đó, 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;
}
```

Xin nhắc một lần nữa, dấu "&" nhằm để khai báo truyền tham số theo kiểu tham chiếu.

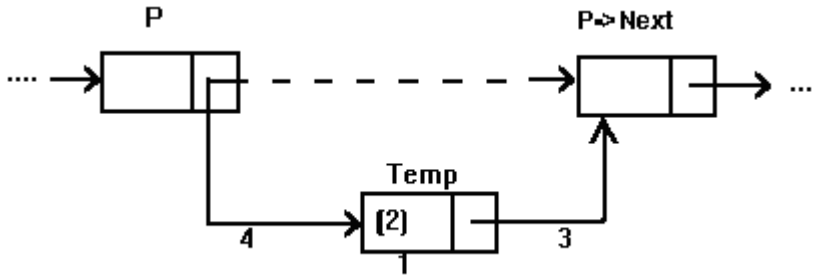
Kiểm tra một danh sách rỗng

Danh sách rỗng không chứa phần tử nào, khi đó trường *Next* trong ô *Header* trỏ tới **NULL.** Như vậy để kiểm tra danh sách rỗng ta chỉ cần xem trường *Next* trong *Header* có phải là NULL hay không. Lưu ý rằng, *Header* chỉ là một khái niệm để chỉ một con trỏ trỏ vào đầu danh sách. Trong khi cài đặt một danh sách cụ thể, ví dụ danh sách L, thì L chính là chỉ điểm đầu của danh sách.

```
int EMPTY_LIST(List L){
        return (L->Next==NULL); //L chính là Header của danh sách L
}
```

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.

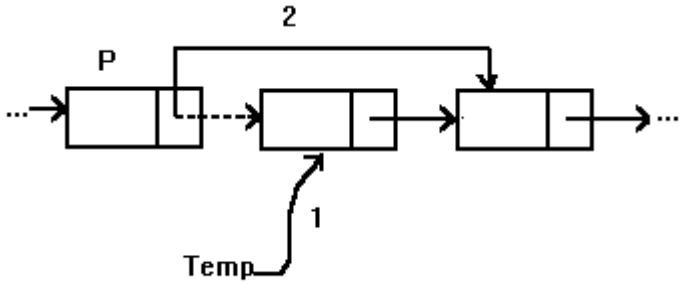


Hình II.4: Thêm một phần tử vào danh sách tại vị trí P

```
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;
}
```

Trong hàm INSERT_LIST ở trên, tham số hình thức L không được sử dụng lần nào vì vậy khi biên dịch có thể gặp cảnh báo với nội dung "biến L không được dùng". Đó không phải là lỗi biên dịch. Ta cũng có thể loại bỏ biến L trong khai báo hàm ở trên mà không gây lỗi gì cho hàm INSERT_LIST. Tuy nhiên như đã nói trước đây, trong giáo trình, ta vẫn giữ nguyên khai báo hàm như trong cấu trúc dữ liệu trừu tượng để cho cấu trúc được định nghĩa nhất quán và độc lập với các cách cài đặt khác nhau.

Xóa phần tử ra khỏi danh sách



Hình II.5: Xóa 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 xóa. Chi tiết và trình tự các thao tác để xóa 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 xóa 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 (từ vị trí của phần tử đầu tiên - tức là 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 và hàm Locate trả về vị trí P thì x chính là P->Next->Element.

```
Position LOCATE(ElementType X, List L){
        Position P = L;
        while (P->Next != NULL)
                if (P->Next->Element == X) break; // ngưng tại vị trí 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. Từ nhận xét này độc giả có thể tự viết thủ tục "tìm tất cả vị trí xuất hiện của x" hay "đếm số lần xuất hiện của x trong danh sách" bằng cách gọi hàm LOCATE ở trên.

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 là **NULL**) thì hàm không xác định.

```
ElementType RETRIEVE(Position P, List L){
        if (P->Next!=NULL)
                return P->Next->Element;
        //else return một giá trị nào đó thể hiện NULL tùy theo kiểu phần tử
}
```

a. So sánh hai phương pháp cài đặt danh sách nêu trên

Không thể kết luận phương pháp cài đặt nào hiệu quả hơn, mà nó hoàn toàn tùy thuộc vào từng ứng dụng hay tùy 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 tùy 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 xóa một phần tử tỉ lệ với số phần tử đi sau vị trí xen hoặc 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, xóa các phần tử.

c. Cài đặt bằng con nháy

Một số ngôn ngữ lập trình không 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 (tương tự 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. Theo cách cài đặt này, danh sách được cài đặt theo *cơ chế danh sách liên kết*. Điểm khác biệt là danh sách không có chỉ điểm đầu thực sự (tức là, không có *Header*), thay vào đó ta chỉ có một biến số nguyên giữ chỉ số mảng chứa phần tử đầu tiên của danh sách.

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 →

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

Chỉ số Element Next

Mảng SPACE

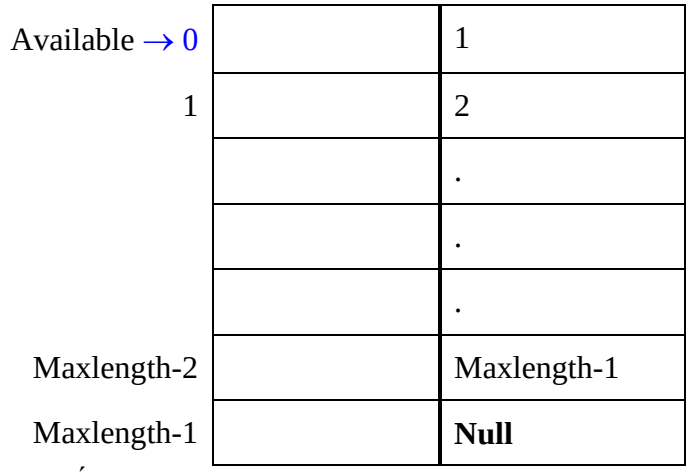
Hình II.6 Mảng đang chứa hai danh sách L_1 và L_2

Khi xen một phần tử vào danh sách ta lấy một ô trống trong mảng (ô không chứa bất kỳ phần tử nào của bất kỳ danh sách nào đang chứa trong mảng) để chứa phần tử mới này và nối kết lại các con nháy. Ngược lại, khi xóa 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 xóa một phần tử từ một danh sách ta cho ô bị xóa 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 đơn giản là cho phần tử thứ i của mảng trỏ tới phần tử thứ 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 Element; /*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

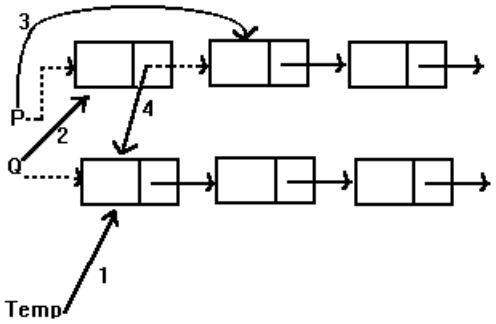
Khởi tạo cấu trúc – Thiết lập Available ban đầu

Ta cho phần tử thứ 0 của mảng trỏ đến phần tử thứ 1; phần tử thứ 1 trỏ đến phần tử thứ 2;...; 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; //chỉ điểm đầu của Available
}
```

Chuyển một ô từ danh sách này sang danh sách khác

Thực chất của việc xen hay xóa một phần tử trong danh sách cài đặt bằng con nháy là thực hiện việc chuyển một ô từ danh sách này sang danh sách khác. Chẳng hạn, khi xen thêm một phần tử vào danh sách L_1 trong hình II.6 tại 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; ngược lại, khi xóa 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 là 1 hoặc 0 tùy theo chuyển thành công hay thất bại. Hàm sẽ trả về 0 nếu chuyển không thành công (lỗi) và sẽ trả về 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. Hình II.8 trình bày các thao tác cơ bản để chuyển một ô từ một danh sách sang 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 bằng con nháy, **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.

Ở đây cần lưu ý rằng các chỉ điểm đầu danh sách (L_1, L_2 và *Available*) là các số nguyên chỉ ô chứa phần tử đầu tiên của danh sách, chứ không phải là số nguyên chỉ ô chứa con nháy trỏ đến phần tử đầu tiên của danh sách. Vì vậy, về bản chất các chỉ điểm đầu này khác với *Header* trong cài đặt danh sách liên kết. Trong cài đặt danh sách liên kết đơn ở trên, khi danh sách rỗng thì *Header* vẫn tồn tại và con trỏ *Next* của nó là NULL (*Headrer->Next* là NULL). Trong cài đặt con nháy ở đây, các chỉ điểm đầu này sẽ là *Null* nếu danh sách rỗng. Ví dụ nếu danh sách L_1 rỗng thì L_1 là *Null* chứ không phải là space[L1].Next là *Null*.

Xen một phần tử vào danh sách

Muốn xen một phần tử vào danh sách ta cần biết vị trí xen, giả sử là p; ta sẽ 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 *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].Element=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 chua X la o tro boi Space[p].Next
                        Space[Space[P].Next].Element=X;
                else printf("Loi! Khong con bo nho trong");
}
```

Xóa một phần tử trong danh sách

Muốn xóa 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 xóa 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 ca
}
```

II. NGĂN XẾP (STACK)

1. Định nghĩa ngăn xếp

Ngăn xếp (Stack) là một danh sách mà 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" - **LIFO** (last in - first out).

2. Các phép toán trên ngăn xếp

- **MAKENULL_STACK(S):** tạo một ngăn xếp rỗng.
- **TOP(S)** hàm trả về phần tử tại đỉnh ngăn xếp. Nếu ngăn xếp rỗng thì kết quả của hàm không xác định.
- **POP(S)** xóa một phần tử tại đỉnh ngăn xếp.
- **PUSH(x,S)** thêm một phần tử x vào đầu ngăn xếp.
- **EMPTY_STACK(S)** kiểm tra ngăn xếp rỗng. Hàm cho kết quả 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 sơ" 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 lý để cài đặt các "nguyên sơ" 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=getchar();
                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);
        }
}
```

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. Thực chất việc cài đặt này chỉ là thực hiện các lời gọi các phép toán đã được định nghĩa cho danh sách một cách đúng đắn để hạn chế việc vào ra trên danh sách chỉ tại một đầu.

Trước hết ta có thể khai báo ngăn xếp như là danh sách:

```
typedef List Stack;
```

Sau đó, thực hiện các lời gọi hàm trên danh sách hợp lí để thực hiện các phép toán trên ngăn xếp.

Tạo ngăn xếp rỗng

Ngăn xếp rỗng cũng có nghĩa là danh sách rỗng, vì vậy:

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

Kiểm tra ngăn xếp rỗng

Kiểm tra ngăn xếp rỗng đồng nghĩa với kiểm tra danh sách rỗng.

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

Thêm phần tử vào ngăn xếp

Thêm 1 phần tử vào ngăn xếp tức là thêm một phần tử vào đầu danh sách.

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

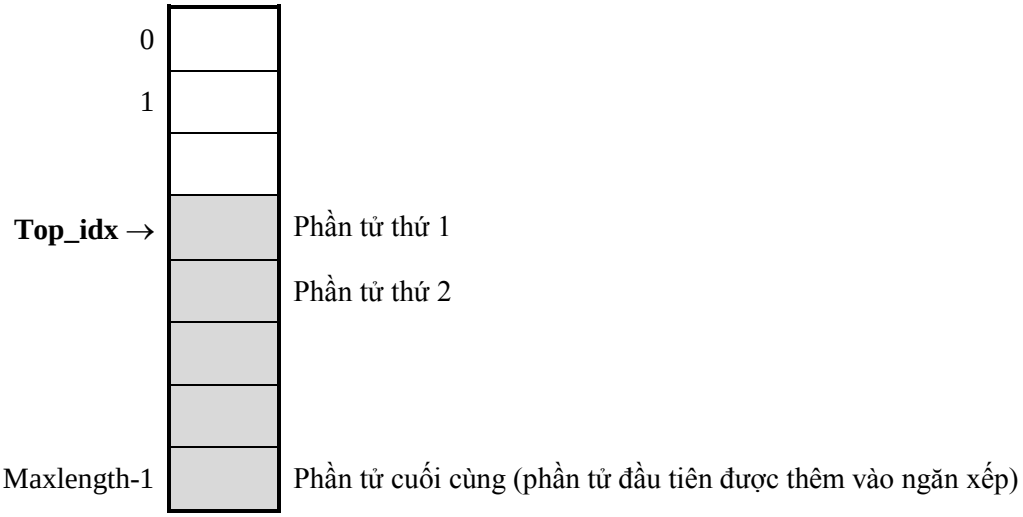
Xóa phần tử ra khỏi ngăn xếp

Xóa một phần tử trong ngăn xếp chính là xóa phần tử đầu tiên trong danh sách.

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

Rõ ràng rằng, chúng ta không cần tốn nhiều công sức cho việc cài đặt ngăn xếp. Tuy nhiên, việc cài đặt danh sách (tổng quát) sau đó làm các lời gọi "hạn chế" lại để có ngăn xếp là việc "lấy dao mổ trâu để giết ruồi". Để tăng tính đơn giản, hiệu quả của ngăn xếp ta có thể cài đặt ngăn xếp trực tiếp từ các cấu trúc dữ liệu như trình bày trong các phần sau.

b. Cài đặt bằng mảng



Hình II.9: Ngăn xếp

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. Dùng một biến số nguyên (Top_idx) giữ chỉ số của phần tử tại đỉnh ngăn xếp.

Tất nhiên việc sắp xếp phần tử vào trong mảng từ đầu nào của mảng là không quan trọng. Điều quan trọng là các phần tử phải liên tục với nhau và việc thêm hay xóa phần tử chỉ thực hiện tại đỉnh của ngăn xếp. Do vậy, sẽ rất không tự nhiên khi bảo rằng hãy loại phần tử thứ k nào đó khác với phần tử tại đỉnh ngăn xếp ra khỏi ngăn xếp! Nếu cần phải thao tác các phần tử trong danh sách tại vị trí bất kỳ thì danh sách đó không phải là ngăn xếp. Nếu đã dùng ngăn xếp thì các thao tác xen xóa phải thực hiện tại đỉnh của 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];     //mảng lưu 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, tăng tính nhất quán, 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

Ngăn xếp rỗng khi đỉnh của ngăn xếp tại vị trí Maxlength.

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

Kiểm tra ngăn xếp đầy

Ngăn xếp đầy cần phải được nói cho đúng là mảng chứa ngăn xếp đầy, tức là không có phần tử nào có thể được thêm vào ngăn xếp. Khi đó, toàn bộ mảng đã có các phần tử trong ngăn xếp chiếm giữ. Trong trường hợp này, đỉnh ngăn xếp ở tại vị trí 0. Để kiểm tra ngắn xếp có đầy hay không ta chỉ cần xem Top_idx có bằng 0 hay không.

```
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"); // có thể return một giá trị biểu diễn cho Null
}
```

Trong một số ngôn ngữ lập trình (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){
// x la gia tri tra ve, do la phan tu tai dinh cua ngan xep
if (!EMPTY_STACK(S))
        X = S.Elements[S.Top_idx];
else printf("Loi: Ngan xep rong ");
}
```

Trong thực hành, cách cài đặt như trên (cả hai cách viết trên) không phải là cách cài đặt tốt vì không thể thao tác được khi gọi hàm với ngăn xếp rỗng. Để tăng tính dễ thao tác kết quả trả về khi thực hiện lời gọi hàm, ta có thể thiết kế hàm trả về mã lỗi. Ví dụ 0 là không có lỗi, 1 là có lỗi. Hàm có thể cài đặt như sau:

```
int TOP(Stack S, Elementtype& X){
// x la gia tri tra ve, do la phan tu tai dinh cua ngan xep
// hàm TOP tra ve ma loi: 0-khong co loi; 1-loi ngan xep rong
if (!EMPTY_STACK(S)){
X = S.Elements[S.Top_idx];
Return 0;
}
```

```
else //Loi: Ngan xep rong
return 1 ;
}
```

Chương trình dùng hàm này có thể viết theo khung như sau:

```
If ( !TOP(S,x))
        //thao tác trên x
else
        // KHÔNG thao tác trên x
```

Có thể áp dụng ý tưởng cài đặt này vào các phép toán POP và PUSH. Tuy nhiên để tập trung vào cốt lõi của vấn đề, không quá sa đà vào cài đặt và sử dụng, chúng tôi định nghĩa các phép toán của ngăn xếp như ở trên (trong đoạn 2) mà không đề cập đến mã lỗi.

Xóa phần tử ra khỏi ngăn xếp

Phần tử được xóa ra khỏi ngăn xếp là phần tử nằm 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;
        }
}
```

Cách cài đặt bằng mảng mang điểm yếu cố hữu là phải ước lượng được số lượng phần tử. Vì vậy nếu muốn tăng tính linh hoạt về số lượng phần tử có thể phải viện tới con trỏ. Tất nhiên ngăn xếp sẽ cài đặt được 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.

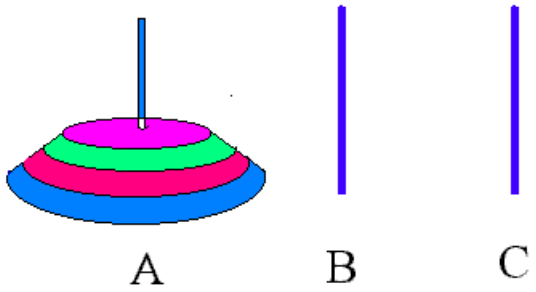
Trong khi một chương trình con từ mức i đi vào mức i+1 thì các biến cục bộ của mức i và địa chỉ của mã lệnh còn dang dở phải được lưu trữ, địa chỉ này gọi là địa chỉ trở về. Khi từ mức i+1 quay về mức i các giá trị đó được sử dụng. Như vậy những biến cục bộ và địa chỉ lưu sau được dùng trước. Tính chất 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 họa 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, có thể dùng cọc C làm trung gian. Mỗi lần thực hiện, chỉ chuyển một đĩa từ một cọc sang một cọc khác và không được đặt đĩa lớn nằm trên đĩa nhỏ (hình II.10)



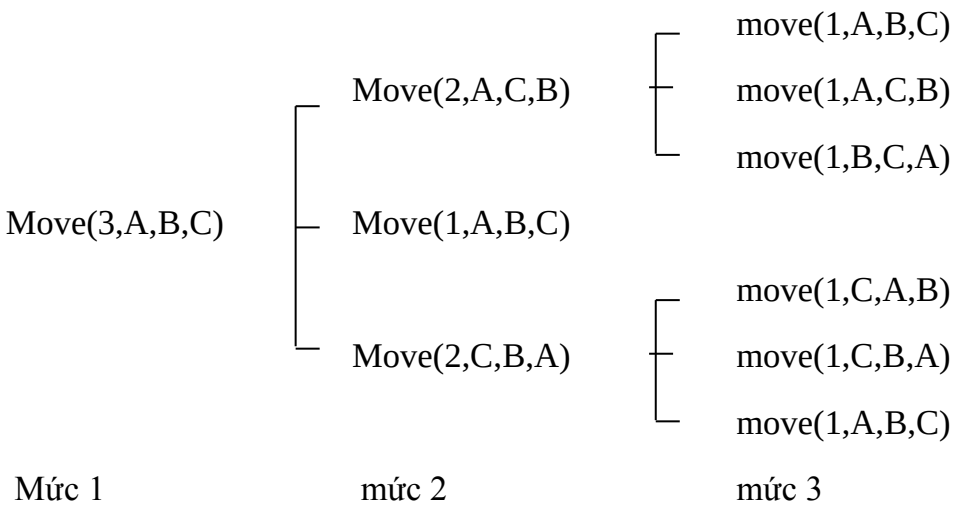
Hình II.10: Bài toán tháp Hà Nội.

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

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

Quá trình thực hiện chương trình con được minh họa 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 xóa 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:

3,A,B,C

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 xóa. 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.

Ứng dụng ngăn xếp để khử đệ qui của chương trình là một ứng dụng quan trọng của cấu trúc ngăn xếp trong khoa học máy tính. Thực chất, đệ qui chỉ là kỹ thuật trong ngôn ngữ cấp cao, cho nên khi máy tính thực hiện một chương trình đệ qui, nó phải "khử đệ qui" với hỗ trợ của ngăn xếp.

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)** xóa 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 đợi

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 xóa 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

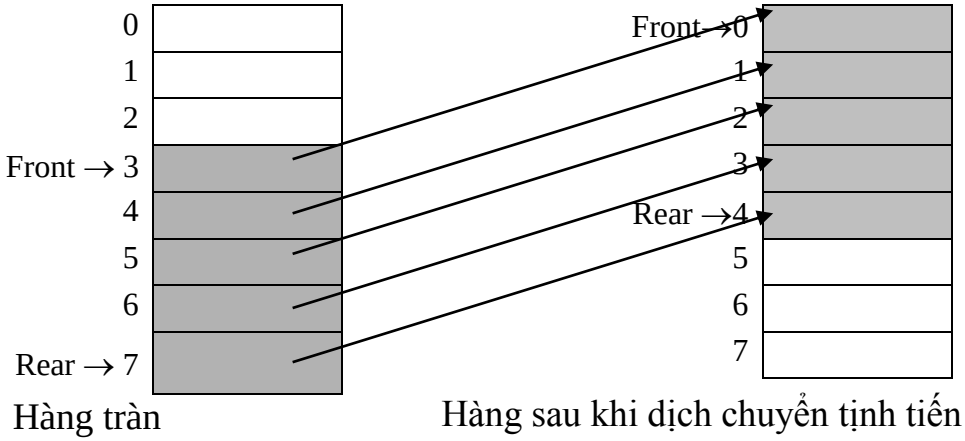
Ta dùng một mảng để chứa các phần tử của hàng. Khởi đầu, phần tử đầu tiên của hàng được đưa vào vị trí thứ 1 của mảng (vào vị trí có chỉ số 0), phần tử thứ 2 vào vị trí thứ 2 của mảng (vào vị trí có chỉ số 1),... Giả sử hàng có n phần tử, ta có front=0 và rear=n-1.

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

Như vậy hàng sẽ 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 thì 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.

b. 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];     // mảng lưu trữ các phần tử
        int Front, Rear; //chỉ số đầu và cuối 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 (tức là Null=-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 xóa 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 xóa một phần tử của hàng, nếu phần tử bị xóa 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. Số phần tử hiện có của hàng là Q.Rear – Q.Front + 1.

```
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:

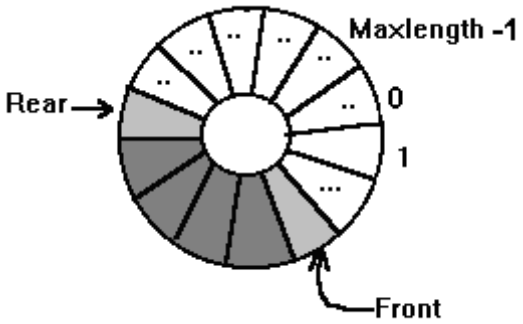
- Nếu hàng đầy thì báo lỗi không thêm được nữa. Trong trường hợp này thao tác thêm không được thực hiện và hàng sẽ không thay đổi.
- Nếu hàng chưa đầy ta phải xét xem hàng có bị tràn không. Nếu hàng bị tràn ta di chuyển tịnh tiến rồi mới nối thêm phần tử mới vào đuôi hàng (Rear tăng lên 1). Đặc biệt nếu thêm vào hàng rỗng thì ta cho Front=0 để nó 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!");
}
```

c. Cài đặt hàng với mảng xoay vòng

Ta vẫn dùng mảng để lưu các phần tử như trong cài đặt ở trên, điều khác biệt ở đây là 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.



Hình II.12 Cài đặt hàng bằng 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];     // mảng lưu trữ các phần tử
        int Front, Rear; //chỉ số đầu và đuôi hàng
 } Queue;
```

Tạo hàng rỗng

Khi hàng rỗng, cả Front và Rear không trỏ đến vị trí hợp lệ nào trong mảng vậy ta có thể cho Front và Rear đều bằng -1 (tức là Null=-1).

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

Kiểm tra hàng rỗng

Như phân tích ở trên, hàng rỗng khi và chỉ khi Front = -1.

```
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 và không xóa phần tử nào;
- 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 phần tử nào;

- 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!");
}
```

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

Cách mềm dẻo nhất là cài đặt hàng như một danh sách liên kết. 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 và Rear cùng trỏ về một vị trí đó chính là ô *Header*. Như đã nói trong khi cài đặt danh sách liên kết đơn, dù hàng rỗng (tức là danh sách liên kết đó rỗng) *Header* vẫn tồn tại và con trỏ *Next* của nó sẽ trỏ NULL.

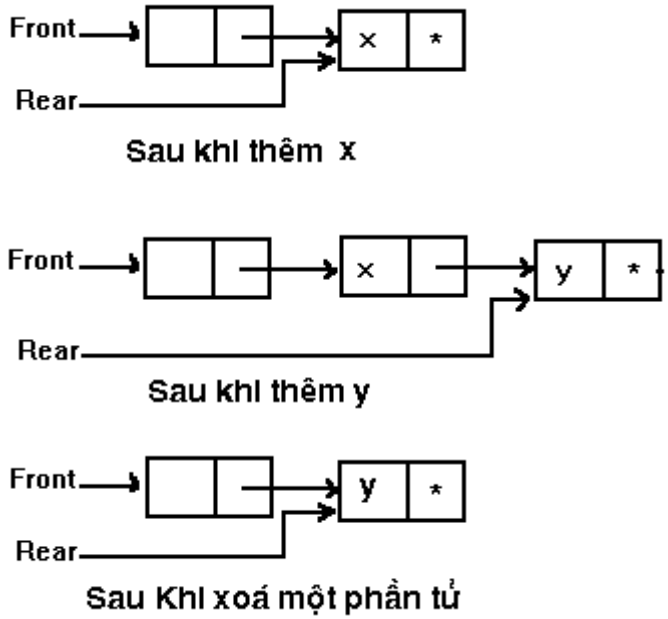
Hình II.13: Khởi tạo hàng rỗng.

```
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* (hình II.13).

```
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

Khi thêm một phần tử vào hàng ta thêm phần tử vào sau Rear (tức là 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 xóa 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 và giải phóng ô nhớ chứa phần tử đầu 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");
}
```

Thực ra trong thủ tục trên, khi hàng rỗng không cần thiết phải có một thông báo lỗi. Có thể đơn giản là hàm chẳng có việc gì làm khi hàng rỗng, vì vậy có thể bỏ dòng else kèm thông báo lỗi. Tuy nhiên, để tăng tính dễ thao tác người ta có thể thiết kế hàm có giá trị trả về để biết là thao tác thành công hay thất bại.

4. Một số ứng dụng của cấu trúc hàng

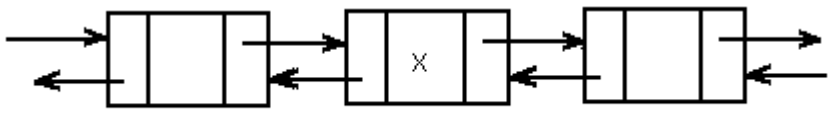
Hàng đợi là một cấu trúc dữ liệu được dùng khá phổ biến trong thiết kế giải thuật. Bất kỳ nơi nào ta cần quản lí dữ liệu, quá trình... theo kiểu vào trước-ra trước đều có thể ứng dụng hàng đợi.

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 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), xóa (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 sau đây, chúng ta không sử dụng *Header* như đã làm với danh sách liên kết đơn mà sẽ quản lý danh sách một các trực tiếp bằng một con trỏ trỏ vào một phần tử trong danh sách.

Tạo danh sách liên kết kép rỗng

Giả sử DL là con trỏ quản lí danh sách liên kết kép. Con trỏ này nhằm để trỏ vào ô chứa một phần tử nào đó hiện tại trong danh sách đang xét. Khi mới khởi tạo danh sách rỗng ta cho con trỏ này trỏ NULL. Ở đây DL khác với Header trong danh sách liên kết đơn ở chỗ nó là một con trỏ giữa địa chỉ một ô chứa phần tử của danh sách, trong khi Header là một ô giống như một ô chứa phần tử và trường Next của nó là con trỏ trỏ tới phần tử đầu tiên trong danh sách. Khi danh sách không rỗng DL sẽ trỏ vào một phần tử nào đó trong danh sách, tức là DL có giá trị khác 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 có là NULL hay không.

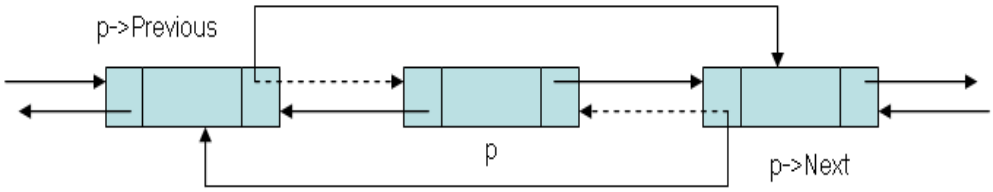
```
int EMPTY_LIST (DoubleList DL){
        return (DL==NULL);
}
```

Xóa một phần tử ra khỏi danh sách liên kết kép

Để xóa 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 bằng N**ULL**: chương trình con dừng và chẳng làm gì cả.
- Trường hợp danh sách khác rỗng, tức là DL!=**NULL**, ta phải phân biệt hai trường hợp:
  - Ô bị xóa 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;

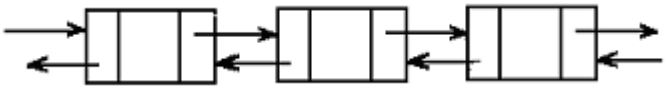
    Các đường nét đứt trong hình II.16 biểu diễn cho các con trỏ trước khi thực hiện cập nhật các nối kết.
  - Nếu xóa ô đang được trỏ bởi DL, tức là p bằng DL thì ngoài việc cập nhật lại các con trỏ để nối kết các ô trước và sau p ta còn phải cập nhật lại DL, ta có thể cho DL trỏ đến phần tử trước nó (tức là DL = p->Previous) hoặc đến phần tử sau nó (DL = p->Next) tùy 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.



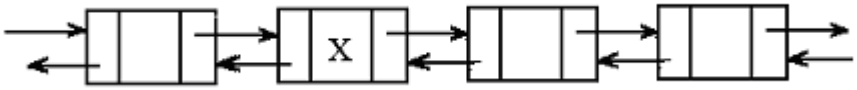
Hình II.16: Xóa phần tử tại vị trí p

```
void DELETE_LIST (Position p, DoubleList& DL){
        if (DL == NULL) printf("Danh sach rong");
        else{
                 //Xóa pt đầu tiên của danh sách nên phải thay đổi DL
                if (p==DL)
                        if (p->next!=NULL)
                                DL=DL->Next;
                        else DL=DL->Previous;
                //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



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.

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:

- Nếu danh sách rỗng, tức là DL bằng 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 khác NULL, sau khi thêm phần tử x vào vị trí p (xem hình II.17) ta có kết quả như trong hình II.18

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 này đã trình bày kiểu dữ liệu trừu tượng danh sách và dùng các cấu trúc dữ liệu khác nhau để cài đặt kiểu dữ liệu trừu tượng đó. Tùy theo bài toán cụ thể và mức độ biến động của dữ liệu mà ta lựa chọn các cấu trúc dữ liệu cho phù hợp. Phần cơ bản nhất của chương là kiểu dữ liệu trừu tượng danh sách và cách thức cài đặt kiểu dữ liệu này. Kế đến là các kiểu 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 nhằm cung cấp khả năng duyệt theo hai chiều thuận lợi.

Danh sách, ngăn xếp và hàng đợi là các kiểu dữ liệu trừu tượng cơ bản nhất trong giáo trình. Các chương sau sẽ cần dùng tới các kiểu dữ liệu này. Và như đã nói ở phần trước, các kiểu dữ liệu trừu tượng có thể được dùng như các "nguyên sơ" trong thiết kế giải thuật, tức là xem như chúng đã có sẵn để dùng mà không cần phải bận tâm về việc cài đặt chúng như thế nào. Tuy nhiên trong thực hành, để thực thi được các giải thuật trên máy tính thì các kiểu dữ liệu trừu tượng phải được cài đặt và được tích hợp một cách thích hợp vào chương trình. Các ngôn ngữ lập trình khác nhau cung cấp cơ chế tích hợp khác nhau, ví dụ trong C (hoặc C++), có thể tích hợp các mã chương trình viết sẵn (các module chương trình) bằng cơ chế *include*. Độc giả có thể củng cố thêm khả năng lập trình và tư duy về giải thuật qua các bài tập của chương.

BÀI TẬP

1. Viết khai báo và các chương trình con cài đặt danh sách chứa số nguyên 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 chứa số nguyên 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 chứa số nguyên có thứ tự bằng cách vận dụng các phép toán cơ bản trên danh sách.
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 nói trên bằng cách vận dụng các phép toán cơ bản trên danh sách.
7. Viết chương trình con loại bỏ các phần tử trùng nhau (giữ lại duy nhất 1 phần tử) trong một danh sách liên kết chứa số nguyên có thứ tự không giảm.
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 bằng cách vận dụng các phép toán cơ bản trên danh sách.
9. Viết chương trình con trộn hai danh sách liên kết chứa các số nguyên theo thứ tự tăng để được một danh sách liên kết cũng có thứ tự tăng.
10. Viết chương trình con xóa các phần tử là số nguyên lẻ khỏi danh sách liên kết chứa các số nguyên.
11. Viết chương trình con tách một danh sách liên kết chứa các số nguyên thành hai danh sách liên kết: một danh sách gồm các số chẵn còn cái kia chứa các số lẻ.

12. Một phân số a/b (với a,b là số nguyên và b khác 0) được biểu diễn như một bản ghi có hai trường: Tử số và mẫu số. Hãy viết các khai báo thích hợp để cài đặt một danh sách liên kết chứa các phần tử là phân số. Dùng các khai báo đó để viết:
    a. Thủ tục để tạo danh sách L rỗng.
    b. Thủ tục thêm một phần tử vào đầu danh sách.
    c. Thủ tục in ra các phần tử trong danh sách
    d. Viết thủ tục làm tối giản tất cả các phân số có trong danh sách (phân số tối giản thì ước chung lớn nhất của tử số và mẫu số bằng 1).
    e. Thủ tục tìm kiếm và xóa các phần tử của danh sách có tử số lớn hơn mẫu số.
    f. Thủ tục (hoặc hàm) tìm trong danh sách L hai phần tử có tích bằng 1. Nếu tìm thấy thì trả ra một cặp con trỏ trỏ đến hai phần tử đó. Nếu không tìm thấy thì trả ra cặp (NULL,NULL)
13. 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. Danh sách có chỉ điểm đầu là A biểu diễn cho Available.

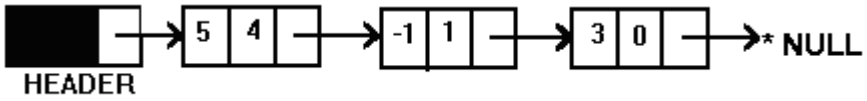
	0	w	9
	1	h	4
A→	2		8
	3		-1
	4	x	6
L1 →	5	g	1
	6	i	-1
L2→	7	y	0
	8		3
	9	u	-1
		Element	Next

Mảng 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. Viết lời gọi hàm thích hợp để xóa x,y.
    d. Vẽ lại hình ở câu b. sau khi xóa : x,y.

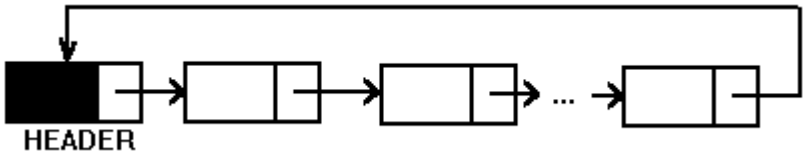
14. Đa thức $P(x)= a_nx_n+ a_{n-1}x_{n-1}+... + a_1x + a_0$ được lưu trữ trong máy tính dưới dạng một danh sách liên kết mà mỗi phần tử của danh sách là một struct có ba trường lưu giữ *hệ số, số mũ*, và trường *NEXT* trỏ đến phần tử kế tiếp. Chú ý cách lưu trữ đảm bảo thứ tự giảm dần theo số mũ của 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 các khai báo thích hợp cho việc lưu trữ này, dựa vào sự cài đặt đó viết:
    b. Chương trình con lấy đạo hàm của đa thức.
    c. Chương trình con thực hiện việc cộng hai đa thức.
    d. Chương trình con thực hiện việc nhân hai đa thức.

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



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

17. Hãy cài đặt một ngăn xếp bằng cách dùng con trỏ.
18. Hãy sử dụng cấu trúc ngăn xếp, viết chương trình:
    a. Đổi một số thập phân sang số nhị phân.
    b. Kiểm tra một chuỗi dấu ngoặc mở và đóng xem có phải là một chuỗi dấu ngoặc đúng hay không. Chuỗi dấu ngoặc đúng là chuỗi dấu mở đóng lồng nhau và khớp nhau như trong biểu thức toán học. Gợi ý: dùng ngăn xếp.
19. 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.

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

20. Khử đệ qui các hàm sau:

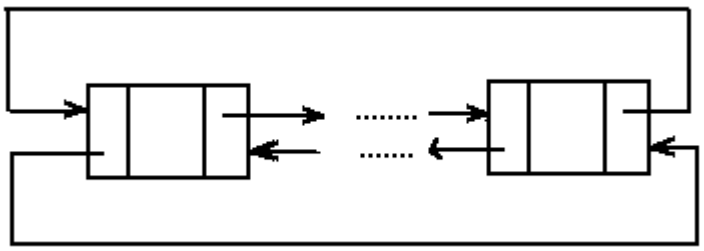
a. Hàm tính tổ hợp chập k của n phần tử

```
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));
}
```

21. Cài đặt danh sách liên kết kép chứa các số nguyên với các phép cập nhật được thực hiện như sau:
    a. Thêm phần tử mới: tại vị trí chỉ điểm đầu danh sách.
    b. Tìm kiếm phần tử x: tìm tuần tự từ chỉ điểm đầu sang hai phía cho đến khi tìm thấy phần tử cần tìm, trả ra vị trí của phần tử này (nếu có). Trong trường hợp có nhiều phần tử trùng nhau thì trả ra vị trí phần tử đầu tiên được tìm thấy. Nếu không tìm thấy thì trả kết quả NULL.
    c. Tìm kiếm và xóa phần tử x trong danh sách.
22. Danh sách liên kết kép nối vòng có dạng sau:



Hãy viết các khai báo thích hợp cùng với các phép toán cơ bản để cài đặt danh sách liên kết kép dạng nối vòng như trên.

CHƯƠNG III: CẤU TRÚC CÂY (TREES)

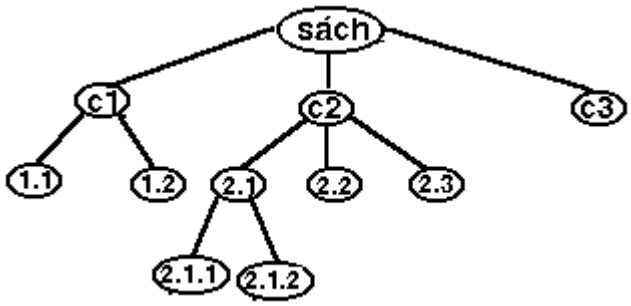
I. CÁC THUẬT NGỮ CƠ BẢN TRÊN CÂY

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 cây

- 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 cấu trúc của một quyển sách qua mục lục của nó. Mục lục này có thể xem là một cây với nút gốc biểu diễn cả quyển sách; nút con của nó biểu diễn cho các chương; mỗi chương lại có các nút con tương ứng với các mục nhỏ.



Hình III.1: Cây mục lục một quyển sách.

Ta cũng có thể xem từng chương như là một cây con: 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.

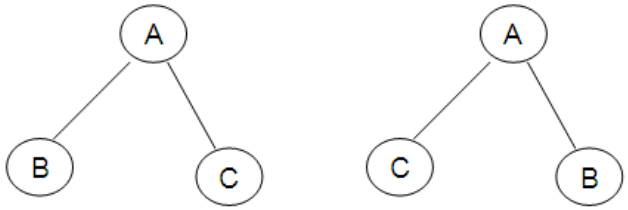
Nếu có đường đi từ nút a đến nút b thì ta nói a là *tiền bối* (ancestor) của b, còn b gọi là *hậu duệ* (descendant) của nút a. Rõ ràng *một nút thì vừa là tiền bối vừa là hậu duệ của chính nó*. Tiền bối hoặc hậu duệ của một nút khác với chính nó gọi là tiền bối hoặc hậu duệ thực sự. Trên cây *nút gốc*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

Bậc của một nút là số nút con của nút đó. Ví dụ trong hình III.1, nút C1 có bậc 2; nút C2 có bậc 3. Bậc của cây là bậc của nút có bậc cao nhất trong cây, chẳng hạn cây trong hình III.1 có bậc là 3.

2. Thứ tự các nút trong cây

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

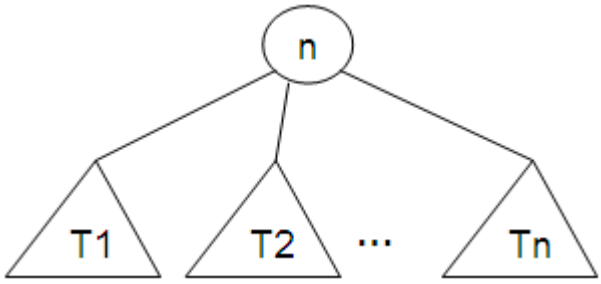


Hình III.2: Hai cây có thứ tự khác nhau.

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 ở 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

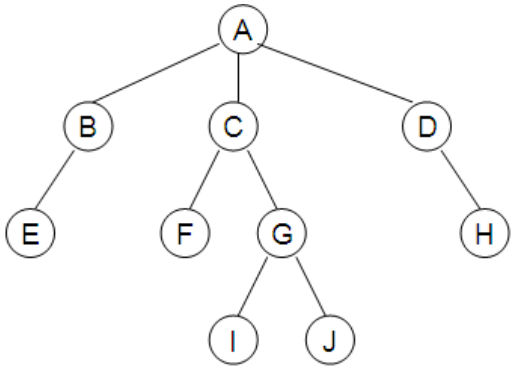
Duyệt cây là một qui tắc cho phép đi qua lần lượt tất cả các nút của cây mỗi nút đúng một lần. Danh sách liệt kê các nút (tên nút hoặc giá trị chứa bên trong nút) theo thứ tự đi qua gọi là danh sách duyệt cây. Có ba cách duyệt cây quan trọng: *Duyệt tiền tự* (preorder), *duyệt trung tự* (inorder), *duyệt 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: Một cây tổng quát.

- 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 (như hình III.3) 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 được cho trong ví dụ.

Biểu thức duyệt tiền tự: A B E C F G I J D H
trung tự: E B A F C I G J H D
hậu tự: E B F I J G C H D A

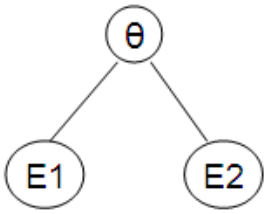
4. Cây có nhãn và cây biểu thức

Thông thường người ta 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 khi cần phân biệt rõ thì hai khái niệm này có thể không đồng. Nhãn là giá trị hay nội dung lưu trữ tại nút, còn khóa 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 sinh viên, họ tên, ngày sinh, địa chỉ,...) thì khóa có thể là mã sinh viên hoặc họ tên hoặc ngày sinh tùy theo giá trị nào ta đang quan tâm đến trong giải thuật.

Trong khoa học máy tính, một biểu thức toán học có thể được biểu diễn thành một cây: mỗi nút chứa nhãn là một toán hạng hoặc một toán tử. Qui tắc biểu diễn một biểu thức toán học trên cây như sau:

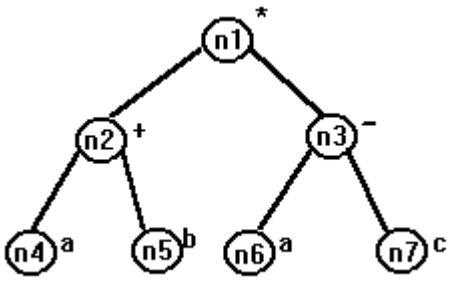
- Mỗi nút lá có nhãn biểu diễn cho một toán hạng.
- Mỗi nút trung gian biểu diễn một toán tử.



Hình III.5: Cây biểu diễn biểu thức E1θ E2.

Trong trường hợp phép toán đang xét là phép toán hai ngôi θ thì ta biểu diễn nút chứa toán tử còn hai con là hai toán hạng tương ứng bên trái và bên phải của θ, xem hình III.5.

Ví dụ: Cây biểu diễn biểu thức (a+b)*(a-c) như trong hình III.6.



Hình III.6: 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.

Khi 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.

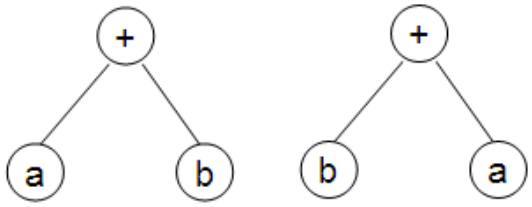
- *Biểu thức dạng hậu tố* hay biểu thức hậu tố (posfix) tương ứng với phép *duyệt hậu tự* của cây.

Ví dụ: đối với cây trong hình III.6 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 cây cho biểu thức 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 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. Như vậy nút gốc cây sẽ biểu diễn cho phép trừ.

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. Nút anh em ruột phải của n là nút nằm ngay bên phải của n và có cùng cha với n.
- 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). Đây là hàm trừu tượng, khi cài đặt ta sẽ cụ thể hóa i là bao nhiêu.
- 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

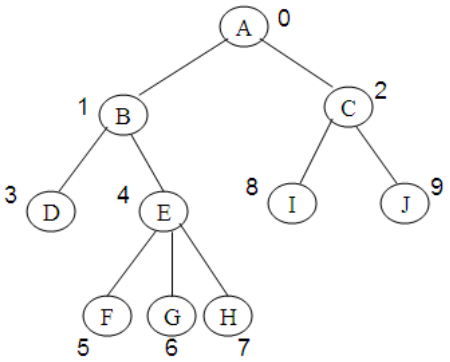
Cho cây T có n nút, ta có thể gán tên cho các nút lần lượt là 0,1, 2, .., n-1. Sau đó ta dùng một mảng một chiều A để lưu trữ cây bằng cách cho A[i] = j với j là nút cha của nút i. Nếu i là nút gốc ta cho a[i] = Null (Null có thể chọn là -1) vì nút gốc không có nút cha.

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. Ở đây ta sử dụng mảng để lưu trữ các phần tử của cây vì vậy ta cần một biến MaxNode giữ số nút hiện tại đang có trên cây. Ta cũng có thể khai báo mảng a là mảng chứa các struct có hai trường: trường Parent giữ chỉ số nút cha; trường Data giữ nhãn của nút thay vì sử dụng hai mảng song song, một mảng chứa chỉ số của nút cha còn một mảng chứa giá trị của nút.

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	…		← Nhãn của các nút trên cây
-1	0	0	1	1	4	4	4	2	2	…		← Cha của nút trên cây
0	1	2	3	4	5	6	7	8	9	…		← Chỉ số của mảng

MaxNode Maxlength

Khai báo cấu trúc dữ liệu

```
#define MAXLENGTH ... /* chỉ số tối đa của mảng */
#define NULL -1 /* đây là giá trị NULL thiết kế cho nút*/
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 */
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. Với cách lưu trữ cây như trên, ta có thể viết được các phép toán cơ bản trên cây như sau:

Khởi tạo cây rỗng

Cây rỗng thì số nút trên cây là 0.

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

Kiểm tra cây rỗng

Kiểm tra cây rỗng tức là kiểm tra xem số nút hiện có trong cây có bằng 0 hay không.

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

Xác định nút cha của nút trên cây

Trong trường hợp cây rỗng hoặc khi nút n được cho không hợp lệ, tức là (n>T.MaxNode-1) hoặc (n<0) thì kết quả trả về sẽ là NULL. Ngược lại, hàm sẽ trả ra chỉ số của nút cha của nút n tức là Parent [n].

```
Node PARENT(Node n,Tree T){
if (EMPTY_TREE(T) || (n>T.MaxNode-1) || (n<0))
        return NULL;
else
        return T.Parent[n];
}
```

Xác định nhãn của nút trên cây

Hoàn toàn tương tự như việc xác định nút cha của một nút, nhãn của một nút cũng đã được lưu trong cấu trúc dữ liệu. Hàm đọc giá trị của nút có thể viết như sau:

```
DataType LABEL_NODE(Node n,Tree T){
if (!EMPTY_TREE(T) && (n>=0) && (n<=T.MaxNode-1))
                return T.Data[n];
//else return một giá trị NULL;
}
```

Nếu nút được cho không hợp lệ hoặc cây rỗng thì hàm không xác định. Về lý thuyết hàm sẽ trả về NULL. Giá trị của NULL được ấn định là bao nhiêu thì tùy thuộc vào kiểu của dữ liệu (DataType) của nút. Ví dụ, nếu DataType là chuỗi kí tự (char*) biểu diễn cho họ tên người thì NULL có thể chọn là "***" vì không có ai có tên như vậy cả!

Hàm xác định nút gốc trong cây

Theo cách cài đặt thì gốc cây luôn nằm tại ô mảng có chỉ số 0.

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

Hàm xác định con trái nhất của một nút

Theo cách đánh số nút, nút con trái nhất của nút n (nếu có) sẽ có chỉ số từ (n+1) trở đi, vì vậy tiến hành tìm tuần tự từ vị trí (n+1) cho đến khi gặp nút đầu tiên có trường Parent là n hoặc không tìm thấy nút nào như vậy.

```
Node LEFTMOST_CHILD(Node n,Tree T){
        Node i;
        if (n<0) return NULL;
        i=n+1; /* Vị trí bắt đầu duyệt tìm nút 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

Theo cách đánh số nút, nút con anh em ruột phải của một nút n (nếu có) sẽ có chỉ số từ (n+1) trở đi, vì vậy tiến hành tìm tuần tự từ vị trí (n+1) cho đến khi gặp nút đầu tiên có trường Parent cùng giá trị với trường Parent của nút n hoặc không tìm thấy nút nào như vậy.

```
Node RIGHT_SIBLING(Node n,Tree T){
        Node i,parent;
        if (n<0) return NULL;
        parent=T.Parent[n];
        i=n+1; /* Vị trí bắt đầu duyệt tìm nút anh em ruột phải của nút n */
        while (i<=T.MaxNode-1)
                if (T.Parent[i]==parent) return i;
                else i=i+1;
        return NULL;
}
```

Thủ tục duyệt tiền tự

Để duyệt tiền tự nút n ta tiến hành duyệt nút đó tiếp đến là duyệt tiền tự tất cả các nút con của nó. Để duyệt tất cả các nút con của n, chương trình sẽ bắt đầu từ nút con trái nhất của nút n, tiếp theo là nút anh em ruột phải của nút đang xét. Giải thuật như sau:

```
Void PreOrder(Node n,Tree T){
    if (T!=NULL){
        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

Tương tự giải thuật duyệt trung tự được viết như sau:

```
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

Duyệt hậu tự một nút n: duyệt hậu tự tất cả các nút con của n rồi ghi ra nhãn của nút n. Giải thuật duyệt hậu tự được viết như sau:

```
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 (kieu nhan)");
        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 (so nguyen)",i);
                scanf("%d",&T.Parent[i]);
```

```
                printf("Nhap nhan cua nut %d (kieu nhan) ",i);
                fflush(stdin);
                scanf("%c",&T.Data[i]);
        }
}
```

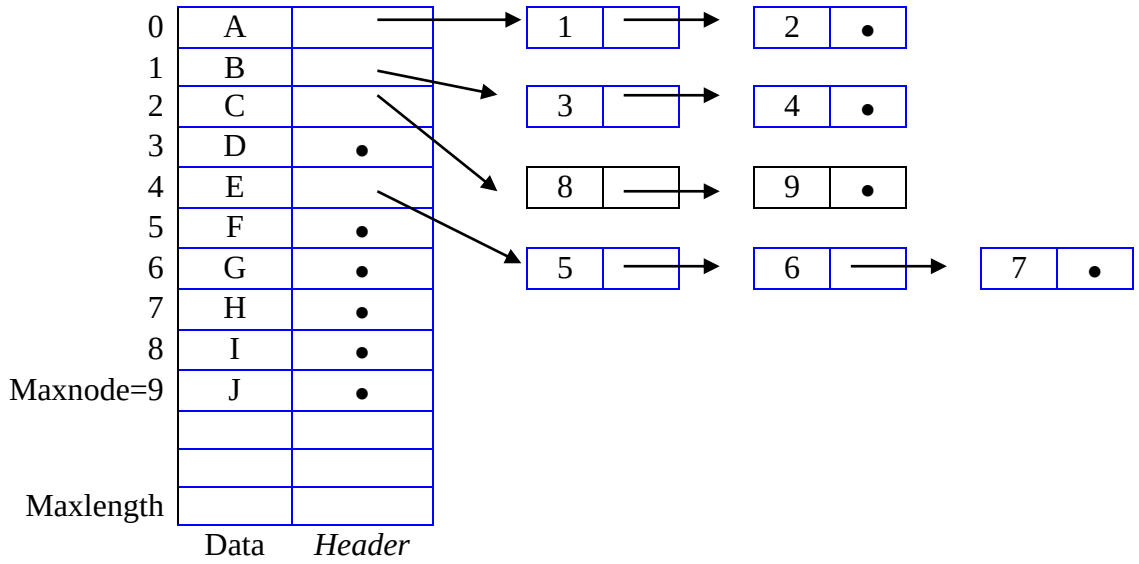
- Hàm xác định con trái nhất của một nút LEFTMOST_CHILD(n,T). Hàm này được dùng trong phép duyệt cây.
- Hàm xác định anh em ruột phải của một nút RIGHT_SIBLING (n,T). Hàm này được dùng trong phép duyệt cây.
- Các chương trình con hiển thị danh sách duyệt cây theo các phép duyệt (PreOrder, InOrder, PostOrder).

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);
}
```

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

2. Biểu diễn cây bằng danh sách các con



Hình III.9: Lưu trữ 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. Cách cài đặt như sau: các nhãn của nút được lưu trữ trong mảng một cách tuần tự từ vị trí đầu tiên của mảng. Tại mỗi phần tử của mảng (tương ứng một nút) có một con trỏ trỏ đến danh sách liên kết chứa chỉ số các nút con của nút đó.

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

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 để thực hiện 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 có hai nút con là hai nút gốc của T1 và T2 tương ứng. 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 có thể có (mảng đóng vai trò như bộ nhớ chứa tất cả các nút của tất cả các cây).

Các khai báo cần thiết là

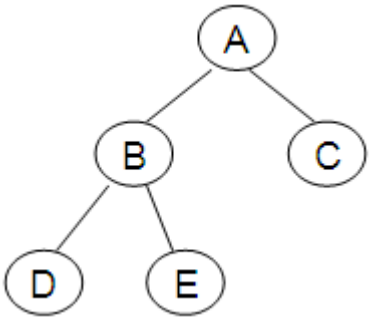
```
#define MaxLength …
#define NULL -1
typedef … DataType;
typedef struct {
        DataType Data;
        int LeftMost_Child;
        int Right_Sibling;
} Node;
Node CELLSPACE [MaxLength];
```

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 một con nháy 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 trong ví dụ.

	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	5
	8			6	

Hình III.11: Quản lí các ô trống bằng danh sách các ô trống (Available).

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;
        CELLSPACE[temp].Parent=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. Hàm thực hiện việc cấp phát một nút mới M có nhãn là v. Nút M sẽ có hai cây con là T1, T2. Như vậy T1 sẽ là con trái nhất của M. Nút gốc của cây T1 sẽ có anh ruột phải là nút gốc của cây T2.

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->Data=v;
        n->Leftmost_Child=left;
        n->Right_Sibling=NULL;
```

```
        n->Parent= NULL;
        if (left!=NULL) {
                left->Parent = n;
                left-> Right_Sibling = right;
        }
        if (right!=NULL) right->Parent=n;
        return n;
}
```

Hàm này cũng chỉ tốn một hằng thời gian để tạo cây mới. Hàm thực hiện việc cấp phát ô nhớ cho nút mới M. Nút này có nhãn là v. Hàm tiếp tục cập nhật lại các con trỏ để nút gốc của cây T1 trở thành nút con trái nhất của nút M; nút gốc của cây T2 là anh ruột phải của nút gốc cây T1.

Cần lưu ý rằng, trong kiểu dữ liệu trừu tượng cây, CREATEi chỉ là một phép toán trừu tượng. Trong cài đặt cụ thể chúng ta cần phải xác định một tập các hàm cần thiết (ví dụ CREATE2: tạo cây mới từ hai cây con; CREATE3: tạo cây mới từ 3 cây con…) để xây dựng cây theo yêu cầu. Tuy nhiên, việc xây dựng một tập các hàm như vậy là cồng kềnh và không thể thực hiện được nếu không biết giới hạn về số con của một nút. Trong thực hành, việc tạo một cây tổng quát bất kỳ có thể thực hiện dễ dàng bằng phép cài đặt con trái nhất anh em ruột phải. Ta chỉ cần viết một hàm CREATE thực hiện tạo nút mới có con trái nhất và nút anh em ruột phải cho trước như đoạn chương trình dưới đây.

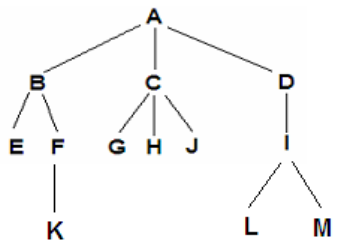
```
typedef char DataType;
typedef struct Cell{
        DataType Data;
        Cell* Leftmost_Child;
        Cell* Right_Sibling;

};
Node Create(DataType v, Tree lmc, Tree rsl){
        Node n;
        n = (Node) malloc(sizeof(Cell));
        n->Data=v;
        n->Leftmost_Child=lmc;
        n->Right_Sibling=rsl;

        return n;
}
```

Từ hàm Create này ta có thể tạo một cây tổng quát bất kỳ. Ví dụ các lời gọi sao cho phép tạo ra cây T như trong hình vẽ.

```
Tree t1 = Create('K',NULL,NULL);
Tree t2 = Create('M',NULL,NULL);
Tree t3 = Create('L',NULL,t2);
Tree t4 = Create('F',t1,NULL);
Tree t5 = Create('E',NULL,t4);
Tree t6 = Create('J',NULL,NULL);
Tree t7 = Create('H',NULL,t6);
Tree t8 = Create('G',NULL,t7);
Tree t9 = Create('I',t3,NULL);
Tree t10 = Create('D',t9,NULL);
```

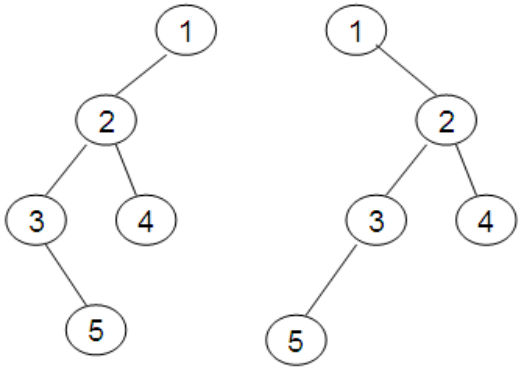


```
Tree t11 = Create('C',t8,t10);
Tree t12 = Create('B',t5,t11);
Tree T = Create('A',t12,NULL);
```

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.



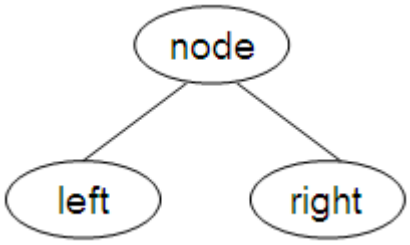
a/ b/

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).

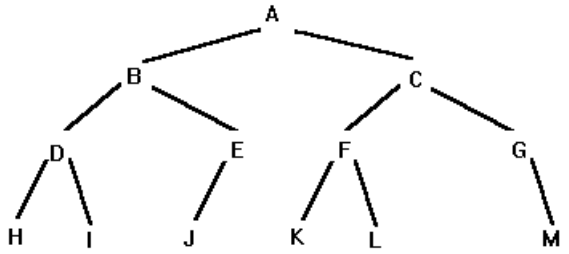


Hình III.13: Cấu trúc cây nhị phân tại một nút.

- ***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.

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: Cây được cho để minh họa các phép duyệt cây trong ví dụ.

	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

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 con trỏ trỏ gốc cây có 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

Con trái của nút n được trỏ bởi n->left. Nút này có kiểu Node* tức là Tree.

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

Xác định con phải của một nút

Con phải của nút n được trỏ bởi n->right. Nút này có kiểu Node* tức là Tree.

```
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

Số nút của cây có thể tính toán một cách đệ qui như sau:

- Nếu cây rỗng thì số nút là 0.
- Nếu cây khác rỗng thì ta đếm 1 cộng với số nút của cây con trái cộng với số nút của cây con bên phải.

```
int NB_NODES(Tree T){
        if(EMPTY_TREE(T)) return 0;
```

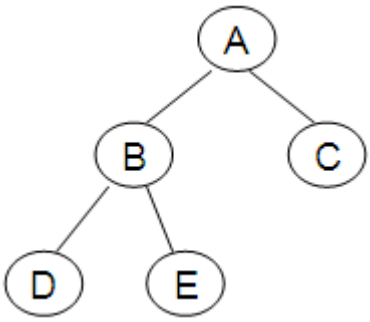
```
        else return 1+NB_NODES(LEFTCHILD(T)) + NB_NODES(RIGHTCHILD(T));
}
```

Tạo cây mới từ hai cây có sẵn

Để tạo cây mới từ hai cây con, chỉ cần cấp phát một nút gốc và cho hai con trỏ của nút này trỏ đến hai cây con đã cho.

```
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;
}
```

Ví dụ: dùng hàm Create2 tạo ra cây trong hình III.15 như sau:



Hình III.15: Cây nhị phân được cho trong ví dụ.

Các lời gọi hàm sẽ lần lượt tạo ra các nút lá, nút B, nút A:

```
Tree T1= Create2(“D”,NULL,NULL);
Tree T2= Create2(“E”,NULL,NULL);
Tree T3= Create2(“C”,NULL,NULL);
Tree T4= Create2(“B”,T1,T2);
Tree T5= Create2(“A”,T4,T3);
```

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

Thủ tục duyệt tiền tự (hoặc NLR)

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

Thủ tục duyệt trung tự (hoặc LNR)

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

Thủ tục duyệt hậu tự (hoặc LRN)

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

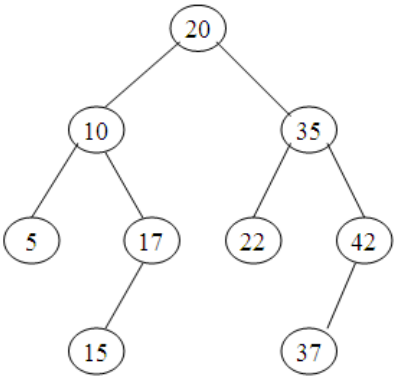
V. CÂY TÌM KIẾM NHỊ PHÂN (BINARY SEARCH TREES)

1. Định nghĩa

Cây tìm kiếm nhị phân (TKNP) là cây nhị phân mà khóa tại mỗi nút lớn hơn khóa của tất cả các nút thuộc cây con bên trái và nhỏ hơn khóa 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 khóa của nút được tính dựa trên một trường nào đó, ta gọi là *trường khóa*. Trường khóa 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.16 minh họa một cây TKNP có khóa là số nguyên (với quan hệ thứ tự trong tập số nguyên).



Hình III.16: 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 khóa.
- 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, 35, 37, 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 xóa 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 khóa, 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 khóa> 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 khóa x trên cây TKNP, ta tiến hành từ nút gốc bằng cách so sánh khóa của nút gốc với khóa x.

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

Ví dụ: tìm nút có khóa 37 trong cây ở trong hình III.16

- So sánh 37 với khóa nút gốc là 20, vì 37 > 20 vậy ta tìm tiếp trên cây con bên phải, tức là cây có nút gốc có khóa là 35.
- So sánh 37 với khóa của nút gốc là 35. Vì 37 > 35, ta tìm tiếp trên cây con bên phải, tức là cây có nút gốc có khóa là 42.
- So sánh 37 với khóa của nút gốc là 42. Vì 37 < 42, ta tìm tiếp trên cây con bên trái, tức là cây có nút gốc có khóa là 37.
- So sánh 37 với khóa nút gốc là 37. 37 = 37, vậy đến đây giải thuật dừng và ta tìm được nút chứa khóa cần tìm.

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

```
Tree Search(KeyType x,Tree Root){
        if (Root == NULL) return NULL;          //không tìm thấy khóa x
        else    if (Root->Key == x) /* tìm thấy khóa 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);
}
```

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 khóa. Do đó nếu ta muốn thêm một nút có khóa x vào cây TKNP thì trước hết ta phải tìm kiếm để xác định có nút nào chứa khóa x chưa. Nếu có thì giải thuật kết thúc (không làm gì cả!). Ngược lại, sẽ thêm một nút mới chứa khóa x này. Việc thêm một khóa 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:

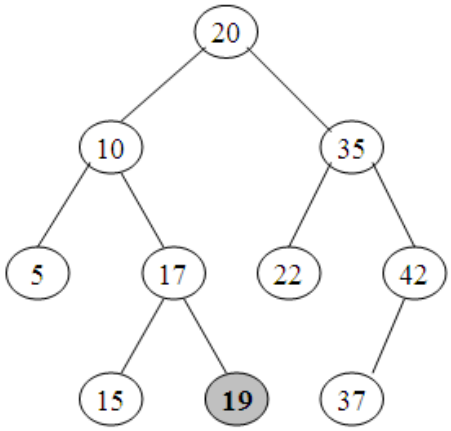
Ta tiến hành từ nút gốc bằng cách so sánh khóa của nút gốc với khóa x.

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

- Nếu x nhỏ hơn khóa của nút gốc thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên trái.

Ví dụ: thêm khóa 19 vào cây ở trong hình III.16

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



Hình III.17: Thêm khóa 19 vào cây hình III.16.

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

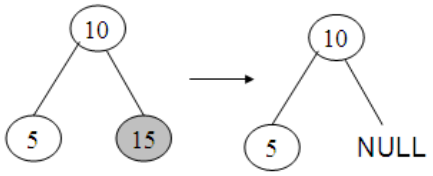
```
void InsertNode(KeyType x,Tree& Root ){
        if (Root == NULL){ /* thêm nút mới chứa khóa x */
                Root=(Node*)malloc(sizeof(Node));
                Root->Key = x;
                Root->Left = NULL;
                Root->Right = NULL;
        }
        else
        if (x < Root->Key)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

Xóa một nút có khóa cho trước ra khỏi cây TKNP

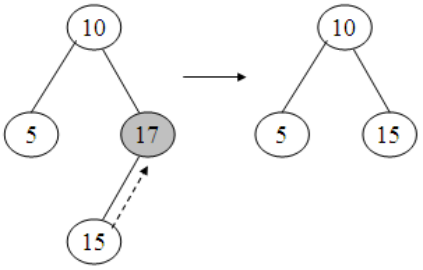
Giả sử ta muốn xóa một nút có khóa x, trước hết ta phải tìm kiếm nút chứa khóa x trên cây. Việc xóa một nút như vậy, tất nhiên, ta phải bảo đảm cấu trúc cây TKNP không bị phá vỡ. Ta có các trường hợp như hình III.18:

- Nếu không tìm thấy nút chứa khóa x thì giải thuật kết thúc.

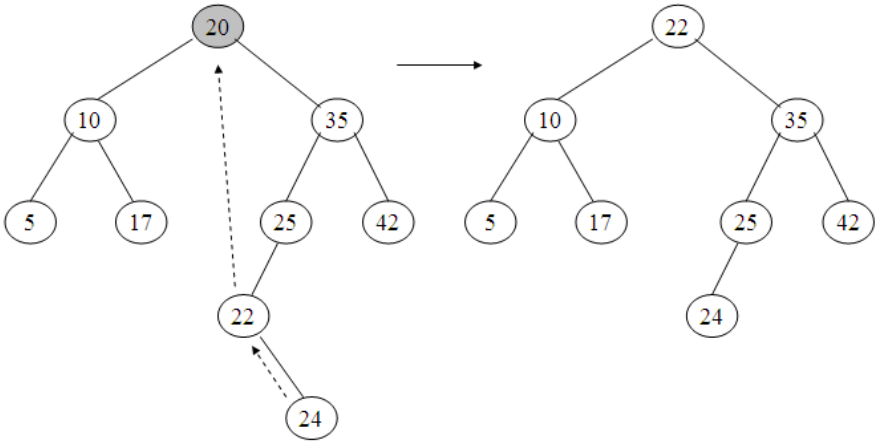
- Nếu tìm gặp nút N có chứa khóa x, ta có ba trường hợp sau (xem hình III.18)
  - Nếu N là lá ta thay nó bởi NULL, ví dụ xóa nút 15 trong hình III.18a.
  - N chỉ có một nút con ta thay nó bởi nút con của nó, ví dụ xóa nút 17 trong hình III.18b.
  - N có hai nút con ta thay nó bởi nút lớn nhất trên cây 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 khóa x bởi khóa của nút cực trái của cây con bên phải rồi ta xóa nút cực trái này. Việc xóa nút cực trái của cây con bên phải sẽ rơi vào một trong hai trường hợp nêu ở trên (xóa nút lá hoặc xóa nút chỉ có một nút con). Ví dụ xóa nút 20 trong cây hình III.18c, ta thay 20 bởi nút 22; như vậy nút 22 sẽ bị xóa và thay thế bởi nút 24.



a/ Xóa nút 15 là nút lá



b/ Xóa nút 17 là nút có một nút con



c/ xóa nút 20 là nút có hai nút con

Hình III.18: Ví dụ về giải thuật xóa nút trên cây

Giải thuật xóa một nút có khóa nhỏ nhất

Hàm dưới đây trả về khóa của nút cực trái, đồng thời xóa 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 khóa 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);
}
```

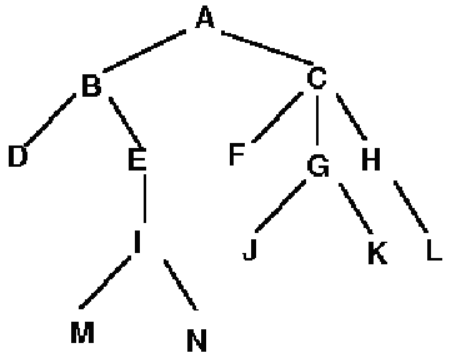
TỔNG KẾT CHƯƠNG

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. Các cách khác nhau để cài đặt cây cũng đã được trình bày như là: cài đặt cây bằng mảng, cài đặt bằng cách dùng con trỏ, cài đặt cây bằng danh sách các con và cài đặt bằng cách lưu trữ cấu trúc theo con trái nhất, anh em ruột phải. Các phép toán cơ bản trên cây cũng đã được thảo luận và cài đặt cụ thể tùy theo cấu trúc dữ liệu được chọn. 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. Cây tìm kiếm nhị phân sẽ phát huy hiệu quả tốt nhất nếu nó "cân bằng". Có nhiều cách cân bằng khác nhau để cho các thao tác trên cây TKNP đạt hiệu quả O(logn) với n là số nút trên cây. Chủ đề này sẽ mang nhiều thú vị cho độc giả muốn tìm hiểu sâu hơn về cây cân bằng.

Cây thể hiện cấu trúc phân cấp trên tập hợp các phần tử (được biểu diễn trên cây). Vì vậy cây là cấu trúc dữ liệu trừu tượng cơ bản cho các bài toán trong đó các phần tử (dữ liệu) được xử lí theo cấu trúc phân cấp hơn là xử lí một cách tuyến tính như trong danh sách.

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

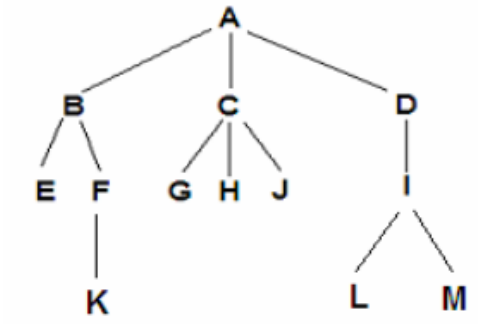
    Tương tự:

    - 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)

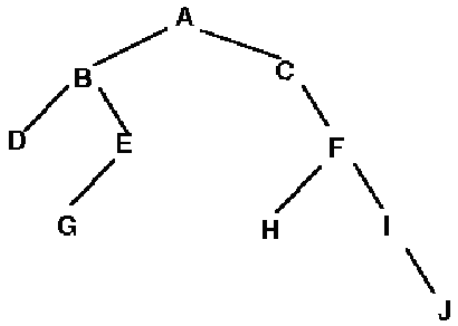
5. Hãy viết các khai báo thích hợp để cài đặt cây (tổng quát) có nhãn là kí tự bằng con trỏ theo cách biểu diễn LeftMost_Child và Right_Sibling. Dùng các khai báo đó viết:
    a. Thủ tục tạo cây mới từ ba cây con Create3(v,T1,T2,T3)
    b. Các thủ tục/ hàm để duyệt tiền tự, trung tự, hậu tự cây
    c. Hàm đếm số nút trên cây
    d. Hàm đếm số nút lá

e. Hàm kiểm tra một nút n có phải là tổ tiên của nút m cho trước hay không?

f. Viết chương trình chính thực hiện việc tạo cây như hình vẽ bên dưới và in ra các biểu thức duyệt tiền tự, trung tự, hậu tự của cây; in ra số nút trên cây và số nút lá trên cây.



6. 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 họa sự lưu trữ kế tiếp các nút cây này trong mảng.

7. 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

8. Một **cây nhị phân được cài đặt bằng con trỏ**, mỗi nút (Node) có 4 trường: trường *value* chứa nhãn của nút, trường *left* trỏ đến con trái, trường *right* trỏ đến con phải, trường *parent* trỏ đến nút cha. Hãy viết các khai báo thích hợp để cài đặt cây *nhị phân* có nhãn là số nguyên. Dùng các khai báo đó để viết:

   a. Hàm kiểm tra hai nút n, m có phải là hai nút anh em ruột hay không. Hàm phải trả ra true nếu n, m là hai nút anh em ruột (có cùng nút cha); ngược lại hàm trả về false.

   b. Hàm kiểm tra một nút n có phải là tổ tiên (ancestor) của nút m hay không. Hàm trả về true nếu n là tổ tiên của m; ngược lại hàm trả về false.

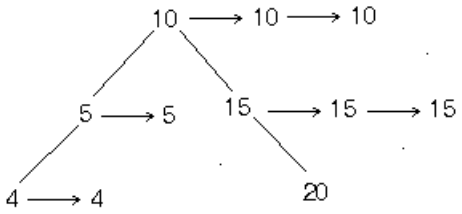
   c. Hàm trả ra số nút chỉ có một nút con trên cây.

   d. Hàm đếm số nút lá của cây.

9. 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.

10. Vẽ hình cây tìm kiếm nhị phân tạo ra từ cây rỗng bằng cách lần lượt thêm vào các khóa là các số nguyên: 54, 31, 43, 29, 65, 10, 20, 36, 78, 59.
11. Vẽ lại hình cây tìm kiếm nhị phân ở bài 10 sau khi lần lượt xen thêm các nút 15, 45, 55.
12. Vẽ lại hình cây tìm kiếm nhị phân ở bài 10 sau khi lần lượt xóa các nút 10, 20, 43, 65, 54.
13. 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.
14. Cài đặt cây TKNP có khóa là chuỗi (String) với các phép toán thêm, xóa. Bổ sung thêm các thủ tục cần thiết để có 1 chương trình hoàn chỉnh, cung cấp giao diện để người dùng có thể thêm, xóa 1 khóa và duyệt cây để kiểm tra kết quả.
15. Viết các thủ tục thêm, xóa một nút có khóa x trên cây tìm kiếm nhị phân bằng cách không đệ qui.
16. Để mở rộng khả năng xử lí các khóa 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 khóa trùng nhau đó. Chẳng hạn cây được thiết lập từ dãy khóa số nguyên 10, 15, 5, 10, 20, 4, 5, 10, 15, 15, 4, 15 như sau



   a. Trong đó các mũi tên nằm ngang chỉ các con trỏ của danh sách liên kết.
   b. 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 số bất kỳ các đối tượng trong thế giới thực. Vì vậy tập hợp 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 ở một số giải thuật, chúng ta giả sử rằng các phần tử của tập hợp có thứ tự tuyến tính. Nói cách khác, trong một số trường hợp ta sẽ xem các phần tử đang xét 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:

- **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.
- **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.
- **DIFFERENCE(A,B,C)** nhận vào 3 tham số là A, B, C; Thực hiện phép toán lấy hiệu của hai tập A với 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 hàm cho kết quả là 0 (sai).
- **MAKENULL_SET(A)** tạo tập hợp A tập rỗng
- **INSERT_SET(x,A)** thêm x vào tập hợp A
- **DELETE_SET(x,A)** xóa x khỏi tập hợp A
- **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

Đây là các phép toán được định nghĩa chung trên kiểu dữ liệu trừu tượng tập hợp. Sau này, trong các bài toán khác nhau chúng ta sẽ quan tâm tới tập hợp cùng với một số phép toán cơ bản nào đó. Tùy vào các phép toán được quan tâm và hiệu quả mong muốn mà ta sẽ cài đặt chúng bằng các cấu trúc dữ liệu thích hợp, từ đó tạo ra các cấu trúc dữ liệu khác nhau, chẳng hạn như bảng băm hoặc hàng ưu tiên.

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.

Bây giờ ta thảo luận một trường hợp đơn giản là khi toàn thể tập hợp được xét là tập hợp con của 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. Mảng kiểu boolean này còn gọi là *vectơ bít*. Cách cài đặt như sau: cho phần tử thứ i của mảng này giá trị TRUE nếu i thuộc tập hợp và cho là FALSE nếu i không thuộc tập hợp. Trong cài đặt, mảng có thể chứa số nguyên 0, 1 tương ứng với FALSE hoặc TRUE. Nếu phần tử trong mảng tại vị trí i là 1, nghĩa là i thuộc tập hợp, ngược lại, phần tử trong mảng tại vị trí thứ i 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:

Chỉ số	1	2	3	4	5	6	7	8	9	10
Giá trị	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. Giả sử các phần tử có giá trị trong phạm vi [0,maxlength-1] ta có khai báo cài đặt như sau :

```
const maxlength = …; // một số nguyên dương chỉ phạm vi các phần tử [0..maxlength-1]
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. Thủ tục 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;
}
```

Hoặc có thể viết súc tích hơn như sau:

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

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

Các phép toán như 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 nhỏ cho độc giả.

Cài đặt tập hợp bằng vectơ bít có nhược điểm chính là phải dùng mảng có kích thước đủ lớn cho một tập hợp toàn thể các phần tử của tất cả các tập hợp đang xét có thể có.

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 các phần bất kỳ, không cần phải giới hạn các phần tử trong phạm vi nhỏ.

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 ≥ 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 không 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*m phép kiểm tra (với n,m là số phần tử của A và B tương ứng)

Nếu A, B được biểu diễn bằng danh sách có thứ tự tăng thì đối với một phần tử e∈A ta chỉ tìm kiếm trong B cho đến khi gặp phần tử x ≥ 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 cấu trúc danh sách

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

Kiểm tra sự hiện diện của phần tử trong tập hợp

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, ngược lại thì hàm trả về giá trị 0.

Nếu tập hợp được cài đặt như một danh sách không có thứ tự thì phải dò tìm tuần tự từ đầu danh sách cho đến khi tìm thấy X hoặc đi đến hết danh sách mà vẫn không tìm thấy phần tử cần tìm.

```
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;
}
```

Nếu tập hợp được cài đặt như là danh sách liên kết có thứ tự thì chỉ cần tìm từ đầu cho đến khi gặp phần tử lớn hơn hoặc bằng X là đủ.

```
int MEMBER(ElementType X, SET L){
        SET P;
        P=L;
        while (P->next!=NULL)
                if (P->next->element<X) P=P->next; //đi tiếp
                else return (P->next->element==X); // dừng khi tìm thấy phần tử y>=x.
        return 0; // đã tìm hết danh sách mà không tìm thấy
}
```

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

Ta cần dùng hai con trỏ duyệt trên hai danh sách tương ứng với hai tập hợp A, B. Mỗi khi xét một cặp phần tử như vậy thì phải xét xem chúng có bằng nhau không, nếu có thì đưa chúng sang tập hợp giao. Nếu không bằng nhau thì tăng con trỏ bên có giá trị bé hơn để đi đến phần tử tiếp theo.

```
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 được viết tương tự (xem như bài tập). Phép ASSIGN(A,B) thực hiện chép 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 cần phải chú ý rằng khi xen x vào A phải đảm bảo thứ tự của danh sách A.

Thêm phần tử vào tập hợp được cài đặt như là danh sách có thứ tự tăng

Để thêm phần tử x vào tập hợp, ta tiến hành tìm kiếm từ đầu danh sách cho đến khi gặp phần tử y lớn hơn hoặc bằng x. Nếu tìm thấy phần tử như vậy thì ta sẽ thêm x vào tại vị trí đó nếu y khác x. Nếu không tìm thấy phần tử như vậy thì ta sẽ thêm phần tử x vào vị trí mà giải thuật dừng, tức là vào cuối danh sách.

```
void INSERT_SET(ElementType X, SET& L){
        SET T,P;
        P=L;
        // tìm vị trí thích hợp cho X
        while (P->next!=NULL)
         if (P->next->element <=X) P=P->next;
         else break; // P đang trỏ đến vị trí để xen phần tử X vào

        if ((P->next==NULL) || ((P->next!=NULL) && (P->next->element!=x))){
                T=(SET)malloc(sizeof(Cell));
                T->element=X;
                T->next=P->next;
                P->next=T;
        }
        // else không làm gì cả
}
```

Xóa phần tử ra khỏi tập hợp được cài đặt như là danh sách có thứ tự tăng

Xóa phần tử x khỏi tập hợp, ta cũng tiến hành dò tìm phần tử x từ đầu danh sách cho đến khi gặp phần tử y lớn hơn hoặc bằng x. Nếu y bằng x (tức là tìm thấy), thì sẽ xóa x khỏi danh sách cài đặt cho tập hợp.

```
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!=NULL)
                if(P->next->element ==X) {
                        T=P->next;
                        P->next=T->next;
                        free(T);
                }
                //else không làm gì cả
}
```

IV. TỪ ĐIỂN (DICTIONARY)

Từ điển là một kiểu dữ liệu trừu tượng tập hợp đặc biệt, trong đó ta chỉ quan tâm đến các phép toán INSERT_SET, DELETE_SET, MEMBER và MAKENULL_SET. Trong nhiều ứng dụng người ta không sử dụng đến các phép toán hợp, giao, hiệu của hai tập hợp. Ngược lại người ta cần một cấu trúc làm sao cho việc tìm kiếm, thêm và

bớt phần tử có phần hiệu quả nhất. Phép toán MAKENULL_SET chỉ nhằm để khởi tạo cấu trúc rỗng.

1. Cài đặt từ điển bằng mảng

Ta có thể cài đặt từ điển bằng mảng như là một danh sách không có thứ tự. Các phần tử của từ điển được lưu tuần tự trong mảng. Khi thêm một phần tử ta có thể thêm vào cuối danh sách; khi xóa một phần tử ta có thể thay thế phần tử bị xóa bởi phần tử cuối cùng trong danh sách; để tìm kiếm một phần tử thì phải tìm kiếm từ đầu đến cuối danh sách. Một trường Last được dùng để chỉ số phần tử hiện tại trong danh sách.

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

Ta duyệt qua các phần tử từ đầu danh sách để tìm X. Xin nhắc lại rằng phần tử thứ i sẽ nằm trong mảng tại vị trí (i-1) với cài đặt bằng C như ở đây.

```
int MEMBER(ElementType X, SET L){
        for (Position P=1; P <= L.Last; P++)
                if (L.Data[P-1] == X) return 1;
        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. Với cách cài đặt như vậy thì phép thêm chỉ mất một hằng thời gian.

```
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-1]=X;
                }
                else
                        printf ("\nPhan tu da ton tai trong tu dien");
}
```

Xóa một phần tử trong tập hợp

Để xóa một phần tử X nào đó ta phải tiến hành tìm kiếm nó trong danh sách. Vì danh sách không có thứ tự nên ta có thể thay thế phần tử bị xóa bằng phần tử cuối cùng trong danh sách để khỏi phải dời các phần tử nằm sau phần tử bị xóa lên một vị trí. Với cách cài đặt này, phép xóa thực hiện trong một hằng thời gian.

```
void DELETE_SET(ElementType X, SET& L){
        if (L.Last==0)
                printf("Tap hop rong!\n");
        else {
                //tìm kiếm phần tử X
                for (Position Q=1; (Q<=L.Last)&& (L.Data[Q-1]!=X); Q++) ;
                //xóa bằng cách gán phần tử cuối vào phần tử bị xóa
                if ( L.Data[Q-1]==X) {
                        L.Data[Q-1]=L.Data[L.Last-1];
                        L.Last--;
                }
        }
}
```

Cài đặt từ điển bằng mảng như trên đòi hỏi tốn n phép so sánh để xác định xem một phần tử có thuộc từ điển có n phần tử hay không. Khi làm việc với kiểu dữ liệu trừu tượng từ điển, việc tìm kiếm một phần tử (bằng hàm MEMBER) nói chung 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. Hơn nữa, hàm MEMBER còn được gọi từ trong thủ tục INSERT_SET để xác định xem phần tử đã tồn tại trong từ điển hay chưa trước khi xen. Do đó nếu MEMBER không được cài đặt để thực hiện có hiệu quả thì nó sẽ dẫn đến là thủ tục INSERT_SET cũng không hiệu quả. Nói cách khác, khi cài đặt từ điển ta sẽ phải thiết kế giải thuật sao cho hàm MEMBER có hiệu quả cao nhất.

2. Cài đặt từ điển bằng danh sách liên kết

Từ điển có thể được cài đặt bằng một danh sách liên kết. Danh sách có thể có thứ tự hoặc không có thứ tự.

Nếu danh sách không có thứ tự thì khi thêm một phần tử ta có thể thêm vào bất kỳ vị trí nào của danh sách (thêm vào đầu danh sách chẳng hạn). Để tìm kiếm một phần tử trong danh sách không có thứ tự ta phải duyệt toàn bộ danh sách. Khi xóa một phần tử ta cần tìm kiếm phần tử đó trong danh sách. Việc xóa phần tử trong danh sách liên kết chỉ mất một hằng thời gian, nhưng tìm kiếm mất thời gian tỷ lệ với số phần tử trong danh sách.

Nếu danh sách có thứ tự thì phép thêm, xóa phải thực hiện tìm kiếm phần tử trong danh sách có thứ tự. Phép tìm kiếm nói chung phải thực hiện trung bình là phân nửa chiều dài danh sách.

Như vậy trong trường hợp cài đặt từ điển như một danh sách liên kết, phép tìm kiếm luôn thực hiện với thời gian tỷ lệ với chiều dài của danh sách. Thời gian này cũng ngang với thời gian thực hiện các phép toán khi thực hiện cài đặt từ điển bằng

mảng. Muốn tăng hiệu quả các phép toán trên từ điển ta cần tìm các cấu trúc dữ liệu khác thích hợp hơn.

V. CẤU TRÚC BẢNG BĂM (HASH TABLE)

1. Kỹ thuật băm

Như đã phân tích ở trên, cài đặt từ điển bằng mảng đòi hỏi tốn n bước để thực hiện một phép toán đơn giản như INSERT_SET 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 nói trên và không đòi hỏi tập hợp phải là tập con của 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 liên kết. 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 phép toán là nhỏ tùy ý.

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

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 xϵA ta phải dựa vào khóa này để suy ra một giá trị số nguyên thuộc khoảng 0.. B-1 là vị trí cất giữ khóa này. Nói cách khác, ta chọn một hàm:

$h: A \rightarrow 0..B-1$

$x \rightarrow h(x)$

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

Như vậy, để áp dụng phương pháp băm một cách hiệu quả ta cần chọn hàm h sao cho ít xảy ra đụng độ hay h có thể "rải đều" các khóa vào trong mảng. Hơn nữa ta phải có cách giải quyết khi đụng độ xảy ra. Mảng được dùng để lưu trữ các phần tử trong tập hợp theo phương pháp băm nói trên gọi là bảng băm (hash table).

Cần nói thêm rằng, các phần tử trong từ điển có thể có cấu trúc phức hợp (ví dụ một bản ghi chứa thông tin sinh viên chẳng hạn), khi đó các tính toán liên quan đến hàm băm có thể thực hiện trên một trường nào đó gọi là trường khóa. Tuy nhiên, để tránh sa đà vào chi tiết này, trong các phần sau chúng tôi không phân biệt rõ phần tử và khóa của nó.

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;
}
```

Hàm thực hiện việc đổi chuỗi kí tự sang số nguyên bằng cách cộng dồn mã ASCII của tất cả các kí tự trong chuỗi. Sau đó thực hiện modulo với B để sinh ra giá trị từ 0 đến B-1.

Chẳng hạn với B=11 ta có:

h(“WINDOWS XP”) = 5

h(“EXCEL”) = 9

h(“WINWORD”) = 4

h(“NETWORK”) = 4

h(“INTERNET”) = 7

Rõ ràng với bảng băm có 11 phần tử và hàm băm như trên thì hai khóa “WINWORD” và “NETWORK” sẽ đụng độ với nhau.

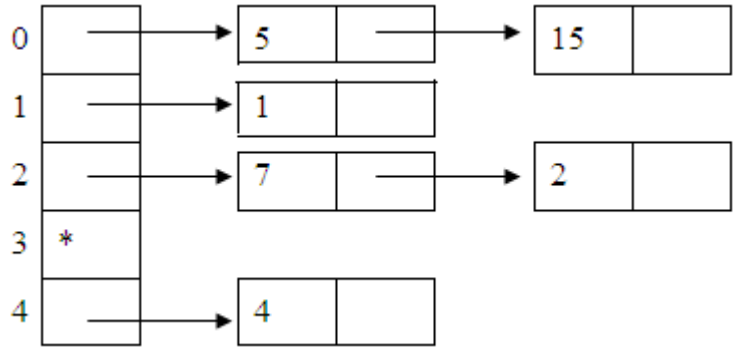
2. Bảng băm mở

Ý tưởng đơn giản để giải quyết đụng độ là tổ chức một danh sách cho một tập hợp các khóa có cùng giá trị băm, xem hình IV.1. Tức là chia một tập hợp đang xét 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 (hay lô). 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. Theo giả thiết này, 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 với danh sách liên kết trong chương 2) sẽ gây lãng phí lớn, do đó phần tử đầu của bucket k ta đặt ngay vào A[k]. Nói cách khác, A[k] là con trỏ trỏ vào phần tử đầu tiên trong danh sách liên kết, nếu danh sách rỗng thì A[k] bằng NULL.

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: Ví dụ 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 địa 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 từ điển sẽ làm việc hiệu quả (với một hằng thời gian). 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 phần tử 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 x sẽ nằm trong bucket được trỏ bởi D[h(x)], với h(x) là hàm băm. Như vậy để tìm phần tử 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 bucket 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 x vào từ điển ta phải tính bucket chứa nó, tức là phải tính h(x). Phần tử 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;
        }
}
```

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

Xóa một phần tử có khóa x trong từ điển bao gồm việc tìm ô chứa khóa và xóa 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 xóa 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 xóa 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 nằm ở ô đầu tiên trong danh sách
```

```
                        temp=D[Bucket];
                        D[Bucket]=D[Bucket]->Next;
                        free(temp);
                }
                else { // Tìm X trong các ô phía sau
                        P=D[Bucket];
                        while (P->Next!=NULL)
                                if (P->Next->Data==X) break;
                                else P=P->Next;
                        // Nếu tìm thấy
                        if (P->Next!=NULL)     {
                                temp=P->Next;
                                P->Next=temp->Next;
                                free(temp);
                        }
                }
        }
}
```

3. 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 (tức là xảy ra đụng độ). Khi thiết kế một bảng băm đóng ta phải có cách để giải quyết sự đụng độ này.

Giải quyết đụng độ

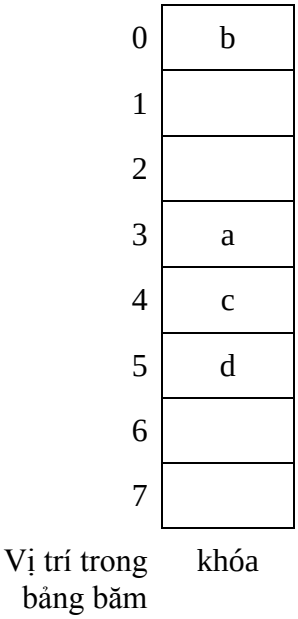
Cách giải quyết đụng độ trong bảng băm đóng gọi là **chiến lược băm lại** (rehash strategy). Chiến lược băm lại là chọn tuần tự các vị trí h_1,..., h_k theo một cách nào đó cho tới khi gặp một vị trí trống để đặt x vào. Dãy h_1,..., h_k gọi là dãy các phép thử. Một chiến lược đơn giản là **băm lại tuyến 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 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)



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 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, điều đó chỉ đúng với trường hợp ta không hề xóa đi một phần tử nào trong bảng băm. Nếu chấp nhận phép xóa thì chúng ta qui ước rằng phần tử bị xóa sẽ được thay bởi một giá trị đặc biệt, gọi là **Deleted.** Giá trị của Deleted không bằng với bất kỳ một phần tử nào trong tập hợp đang xét và nó cũng phải khác giá trị **Empty**. Như đã nói, Empty cũng là một giá trị đặc biệt cho ta biết ô trống, nó không được trùng với bất kỳ phần tử nào có thể xét đến.

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 họa 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 * (với điều kiện là các chuỗi này không phải là nội dung khóa chúng ta có thể xét)

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 modulo với B.

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

Tạo từ 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 từ 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 chỗ 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 chỗ 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 trị 0 nếu phần tử X không tồn tại trong từ 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 từ 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 một phần tử trong từ điển

```
void DELETE_SET(ElementType x, Dictionary& A){
        int bucket;
        bucket=LOCATE(x,A);
        if (A[bucket]==x)
                A[bucket]=Deleted; //đánh dấu là đã xóa nội dung trong ô
}
```

4. Các phương pháp xác định hàm băm

Phương pháp chia

"Lấy phần dư của giá trị khóa 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, nếu B=1000 thì H(x) chỉ phụ thuộc vào ba số cuối cùng của khóa 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 khóa 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 khóa do vậy các khóa có khác nhau đôi chút thì hàm băm cho kết quả khác nhau.

Phương pháp tách

Đối với các khóa dài và kích thước thay đổi người ta thường dùng phương pháp phân đoạn : phân khóa ra thành nhiều đoạn có kích thước bằng nhau từ một đầu (trừ đoạn tại đầu cuối). Nói chung, người ta thường chia 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: khóa 17046329 tách thành

329

046

017

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

b. Gấp: gấp khóa 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 được xếp lại thẳng hàng với nhau rồi có thể cộng lại rồi áp dụng phương pháp chia (modulo) để cho kết quả băm

Ví dụ: khóa 17046329 gấp hai biên vào ta có

932

046

710

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

Thông thường, người ta dùng các tính toán số học để tạo ra hàm băm có khả năng rải đều các khóa. Hàm băm có giá trị «càng ngẫu nhiên » càng tốt để tránh đụng độ. Trong các ứng dụng thực tế người ta cần nghiên cứu kỹ các khóa để đề xuất hàm băm sao cho có hiệu quả nhất : đơn giản, có khả năng sinh các khóa « rải đều ».

VI. 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; 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 giá trị thuộc tập hợp có 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 xóa 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. Phép toán DELETEMIN xem như là lấy phần tử có độ ưu tiên nhất để xử lí.

2. Cài đặt hàng ưu tiên

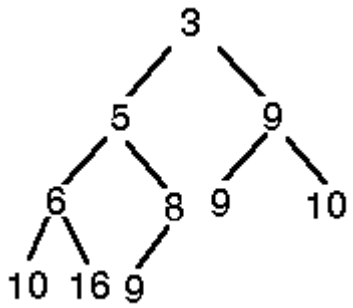
Trước hết, có thể cài đặt hàng ưu tiên bằng danh sách liên kết. Danh sách liên kết có thể có thứ tự hoặc không có thứ tự. Nếu danh sách liên kết có thứ tự thì ta có thể dễ dàng tìm phần tử nhỏ nhất, đó là phần tử đầu tiên, nhưng phép thêm vào đòi hỏi ta phải duyệt trung bình phân nửa danh sách để có một chỗ 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.

a. 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 để cài đặt hàng ưu tiên, trong đó mỗi phần tử được biểu diễn bởi một nút trên cây và độ ưu tiên của phần tử là sẽ xác định trật tự các nút trên cây theo tính chất của cây có thứ tự từng phần.

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 các 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 đó là con trái (như vậy nút đó 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".

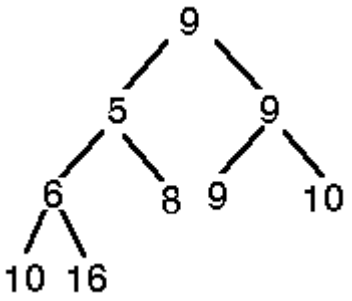
Chiến lược xây dựng lại cây như sau

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.

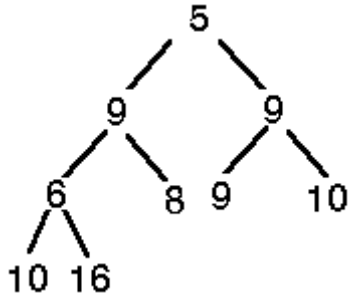
Giải thuật đẩy nút xuống như sau:

- 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, tức là 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, tức là 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 nút đó xem có phải đẩy 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.

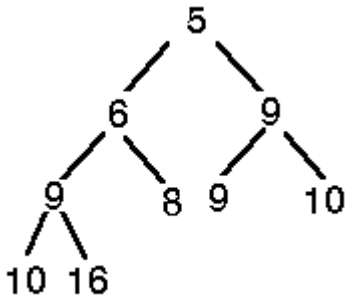
Ví dụ: thực hiện DELETEMIN với cây trong hình IV.3 trên ta loại bỏ nút 3 và thay nó bằng nút 9 (nút con của nút 8), cây có dạng sau:



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



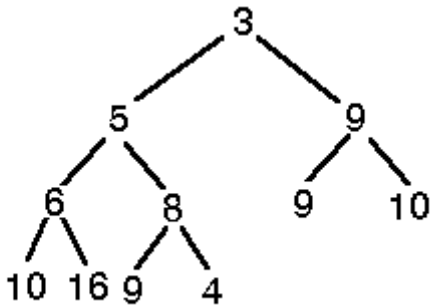
Tiếp tục "đẩy nút 9 xuống" bằng cách đổi chỗ nó với 6



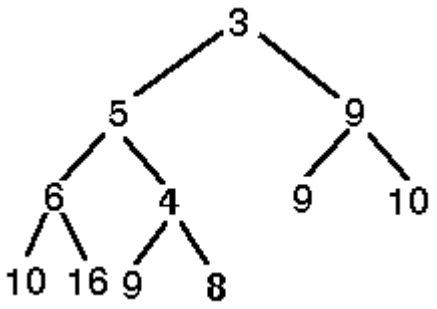
Quá trình đã kết thúc.

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 nút đang xét đã lên đến nút gốc hoặc cây đã thỏa mãn tính chất có thứ tự từng phần.

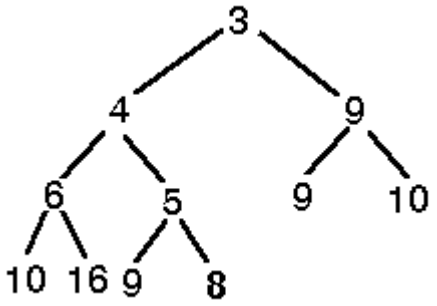
Ví dụ: thêm nút 4 vào cây trong hình IV.3, ta đặt 4 vào lá ở mức cao nhất và ngay bên phải các lá đang có mặt trên mức này ta được cây



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



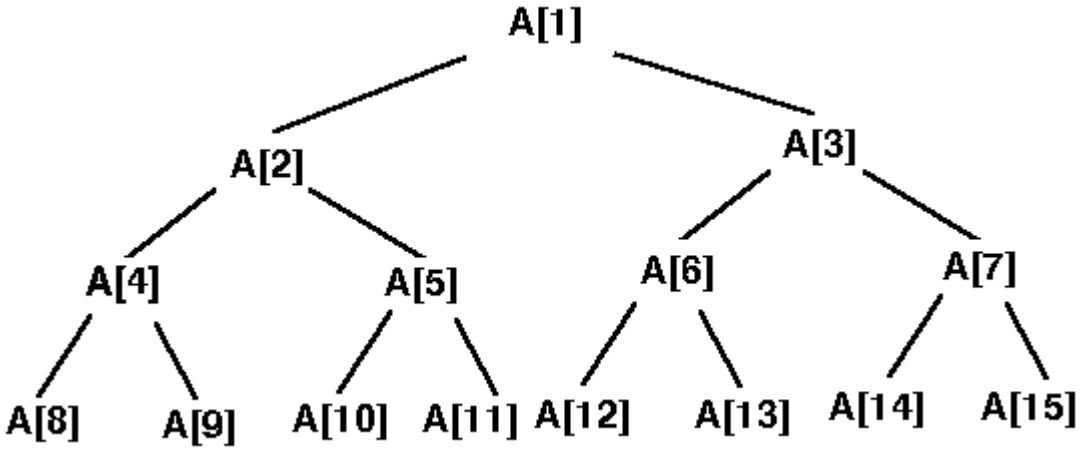
Tiếp tục cho 4 nổi lên ta có cây



Quá trình đã kết thúc.

Nhận xét rằng việc thêm một nút vào cây có thứ tự từng phần hoặc khi xóa nút nhỏ nhất (nút gốc) trên cây có thứ tự từng phần sẽ kéo theo việc tổ chức lại cây theo cách cho khóa "nổi lên" (hoặc "chìm xuống"), tức là đổi chỗ các nút cho nhau để đảm bảo tính có thứ tự từng phần. Việc đổi chỗ này có thể thực hiện nhiều nhất là bằng với chiều cao của cây. Nếu cây có n nút được tổ chức "cân bằng" thì chiều cao của cây là thấp nhất, khi đó chiều cao này bằng $O(\log_2 n)$. Với cách tổ chức cây "cân bằng" bởi các điều kiện áp đặt như trên thì thời gian thêm xóa nút trên cây có thứ tự từng phần sẽ là $O(\log_2 n)$.

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



Hình IV.4: Heap có 15 phần tử.

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

Nhận xét rằng, với cách xây dựng Heap như trên thì nút cha của nút A[i] là A[i div 2], với i>1. Cây được xây dựng lớn lên từ mức này đến mức khác bắt đầu từ gốc và tại mỗi mức cây phát triển từ trái sang phải.

Cài đặt hàng ưu tiên bằng Heap như sau:

Khai báo cài đặt

```
#define MaxLength 100 //một số nguyên chỉ độ dài mảng
typedef int ElementType; //kiểu phần tử
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;
}
```

Lưu ý rằng Last là số phần tử của cây, đó cũng chính là vị trí của phần tử cuối cùng trong Heap. Các phần tử trong Heap chiếm chỗ trong mảng từ 1 đến Last (với cài đặt bằng C thì mảng bắt đầu từ 0, nhưng vị trí số 0 này sẽ không được dùng).

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>1)&&(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, giả sử các số nguyên này cũng chính là độ ưu tiên của phần tử. 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("Xóa lần lượt các nút có khóa nhỏ nhất \n");
```

```
        while (L.Last>0)
                printf("%d\n", DeleteMin(L));
}
```

TỔNG KẾT CHƯƠNG

Chương này đã trình bày cấu trúc dữ liệu trừu tượng tập hợp. Tập hợp là một khái niệm cơ bản trong toán học vì vậy cấu trúc dữ liệu trừu tượng này đóng vai trò quan trọng trong việc mô hình hóa các bài toán. Phần chính của chương dùng để thảo luận về kiểu dữ liệu tập hợp đặc biệt đó là từ điển. Từ điển là một kiểu dữ liệu trừu tượng tập hợp đặc biệt, trong đó phép toán kiểm tra thành viên (MEMBER) được sử dụng rất thường xuyên và cần phải được cài đặt sao cho có hiệu quả cao. Bảng băm là một cấu trúc được tạo ra để đáp ứng yêu cầu này. Việc tìm kiếm một phần tử trong bảng băm chỉ mất một hằng thời gian.

Hàng ưu tiên là một kiểu dữ liệu trừu tượng tập hợp khác, trong đó phép toán tìm kiếm và loại bỏ phần tử bé nhất ("ưu tiên nhất") và thêm một phần tử vào hàng ưu tiên được quan tâm đặc biệt. Việc cài đặt hàng ưu tiên phải làm sao cho các phép toán này có hiệu quả nhất, tức là có thời gian thực hiện nhanh. Cấu trúc cây có thứ tự từng phần được trình bày để đáp ứng yêu cầu này. Các phép toán thêm vào và loại bỏ phần tử có độ ưu tiên bé nhất được thực hiện trong thời gian O(logn), trong đó n là số nút của cây, là một cài đặt hiệu quả nhất được biết hiện tại.

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 khóa 1, 8, 27, 64, 125, 216, 343 trong các trường hợp:
   a. Dùng bảng băm mở.
   b. 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 $h_i = (h(x)+i^2) \bmod B$.
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 một từ trong văn bản hoặc không có hoặc là kí tự đặc biệt và kí tự sau một từ là kí tự đặc biệt.

   Viết chương trình thành lập một từ điển gồm các từ trong văn bản và chứa trong một bảng băm mở.

   Gợi ý: 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 nếu nó chưa có trong bảng băm. 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 tiếng Anh;

   Explanation: kiểu chuỗi ký tự giải thích cho từ tương ứng;

   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 cung 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 khóa 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 khóa 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;

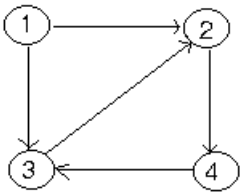
   Priority: số thực ghi nhận mức độ ưu tiên của tiến trình;

   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.
10. Một điểm trong mặt phẳng được biểu diễn bằng một cặp tọa độ (x,y), với x,y là số thực. Như vậy, có thể cài đặt điểm như là một bản ghi có hai trường x,y. Hãy cài đặt hàng ưu tiên chứa các phần tử là điểm với độ ưu tiên theo khoảng cách từ điểm (x,y) tới gốc tọa độ. Chương trình phải cho phép:
    a. Nhập vào các điểm, lưu trữ trong hàng ưu tiên
    b. In ra các điểm dạng (x,y) theo thứ tự DeleteMin.

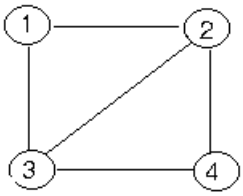
CHƯƠNG V: ĐỒ THỊ (GRAPH)

I. CÁC ĐỊNH NGHĨA

Một đồ thị G bao gồm một tập hợp V các đỉnh và một tập hợp E các cung (hay còn được gọi là cạnh), ký hiệu G=(V,E). Các đỉnh còn được gọi là nút (node) hay điểm (point). Một cung là một cặp đỉnh và thường được biểu diễn bằng một đường (thẳng/cong) nối giữa hai đỉnh, hai đỉnh này có thể trùng nhau. Hai đỉnh có cung nối nhau gọi là hai đỉnh kề (adjacency). Nếu cặp đỉnh (biểu diễn cho một cung) 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 tất cả 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

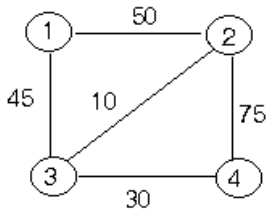
Thông thường trong một đồ thị, các đỉnh biểu diễn cho các đối tượng còn các cung biểu diễn mối quan hệ (relationship) giữa các đối tượng đó. Một ví dụ trực quan: các đỉnh biểu diễn cho các thành phố còn các cung biểu diễn cho đường giao thông nối giữa hai thành phố.

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, khi 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 được 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 được 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 đơn có độ dài 3. Trong hình V.1b thì 1,3,4,2,1 là một chu trình đơn 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 của đồ thị, khi đó 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 tùy

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.
- Xóa một đỉnh.
- Xóa 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 tùy 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 thực 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 tùy 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 cung 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	1	2	3	4
1	true	true	true	**False**
2	true	true	true	True
3	true	true	true	True
4	**false**	true	true	True

Ta cũng có thể biểu diễn true là 1 còn false là 0. Ví dụ, đồ thị hình V.1a có biểu diễn ma trận kề như sau:

j / i	1	2	3	4
1	1	1	1	0
2	0	1	0	1
3	0	1	1	0
4	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= …; //số đỉnh của đồ thị
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][i] == 1)
                        return i;
        return Null;
 }
```

Xin lưu ý rằng các đỉnh của đồ thị được đánh số 1..n, trong khi biểu diễn ma trận kề cài đặt bằng C chỉ số mảng bắt đầu từ 0. Vì vậy cung (i,j) sẽ chứa trong a[i-1][j-1] thay vì chứa trong a[i][j]. Trong trường hợp này ta có thể coi như các đỉnh của đồ thị được đánh số từ 0..(n-1); như vậy các giải thuật sẽ không cần thay đổi để thay a[i][j] bởi a[i-1][j-1].

```
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][j] == 1)
                        return j;
        return Null;
}
```

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

```
int VERTEX(int i){
        return 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);
}
```

Nếu 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 x thì a[i][j]=x. Ví dụ ma trận kề của đồ thị hình V.2 là:

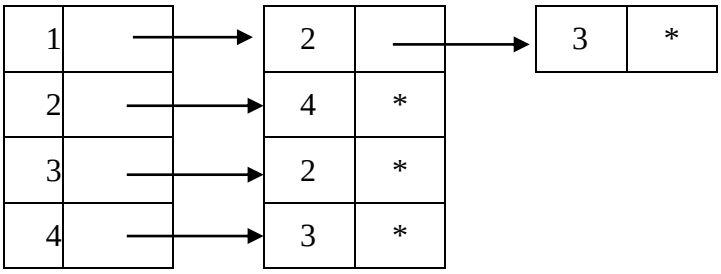
j / i	1	2	3	4
1		50	45	
2	50		10	75
3	45	10		30
4		75	30	

Ở đây, các cặp đỉnh không có cạnh nối thì ta để trống, nhưng trong các ứng dụng cụ thể ta sẽ 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, nhãn là các giá trị biểu diễn cho khoảng cách giữa hai thành phố. Nếu các cặp thành phố không có cạnh nối ta gán cho nó khoảng cách bằng ∞ (vô cùng), còn khoảng cách từ một đỉnh đến chính nó được cho bằng 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, chẳng hạn là một danh sách theo thứ tự các đỉnh. 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

Cách biểu diễn này gọi là biểu diễn theo danh sách các đỉnh kề. Nó cho phép tìm kiếm nhanh chóng các đỉnh kề với một đỉnh đang xét. Cách biểu diễn này cũng rất thích hợp cho biểu diễn đồ thị trong trường hợp ma trận kề là một ma trận thưa, tức là đồ thị có nhiều đỉnh nhưng không có nhiều cạnh.

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)

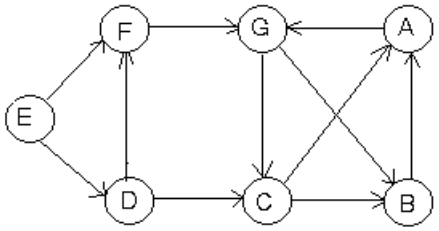
Giả sử ta có đồ thị G=(V,E) với các đỉnh ban đầu được đánh dấu là chưa duyệt (unvisited). Từ một đỉnh v nào đó ta bắt đầu duyệt như sau: đánh dấu v đã duyệt, với mỗi đỉnh w chưa duyệt kề với v, ta thực hiện đệ qui quá trình trên cho w. Sở dĩ cách duyệt này có tên là duyệt theo chiều sâu vì nó sẽ duyệt 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. mark[i] chứa giá trị unvisited để chỉ đỉnh i chưa duyệt; ngược lại nếu đỉnh i đã được duyệt thì mark[i] chứa giá trị visited.

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

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

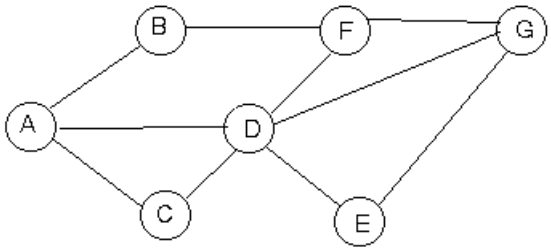
```
void dfs(vertex v){                    // v ∈ [1..n]
        vertex w;
        mark[v]=visited;
        for (mỗi đỉnh w là đỉnh kề với v)
                if (mark[w] == unvisited)
                        dfs(w);
}
```



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 đều 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 theo đúng thứ tự. 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] = 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] == 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] = visited;
        ENQUEUE(v,Q);
        while !(EMPTY_QUEUE(Q)){
                x = FRONT(Q);
                DEQUEUE(Q);
                for (mỗi đỉnh y kề với x)
                        if (mark[y] == unvisited) {
                                mark[y] = 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 họa 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

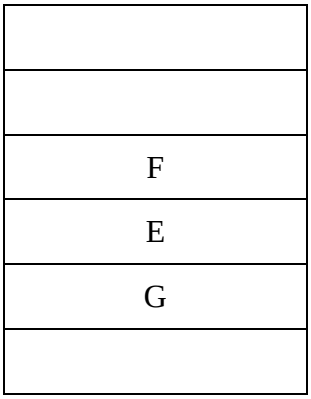
Kế đến B được lấy ra khỏi hàng và các đỉnh kề với B mà chưa được duyệt, đó là F, sẽ được duyệt, và F được đưa vào hàng đợi.

C
D
F

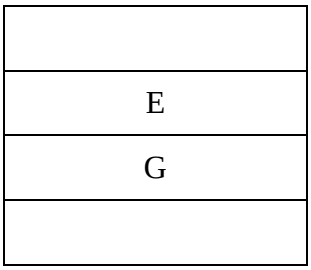
Kế đến thì C được lấy ra khỏi hàng và các đỉnh kề với C mà chưa được duyệt sẽ được duyệt. Không có đỉnh nào như vậy, nên bước này không có thêm đỉnh nào được duyệt.

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.



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.



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 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... Phần này được xem như tìm hiểu và cài đặt một số giải thuật cho các bài toán cơ bản trên đồ thị.

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

Cho đồ thị G (đồ thị có hướng hoặc vô hướng) với tập các đỉnh V và tập các cạnh E. 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. 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.

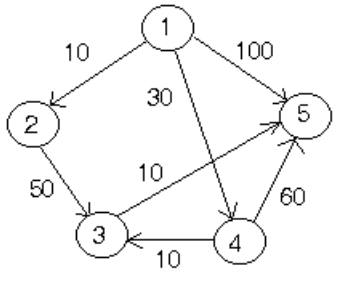
```
void Dijkstra(){
        S = [1];            //Tập hợp S chỉ chứa một đỉnh nguồn
        for (i =2; i<=n; i++)
                D[i] = C[1][i];   //khởi đầu các giá trị cho D
        for (i=2; i<=n; i++){
                Lấy đỉnh w trong V-S sao cho D[w] nhỏ nhất;
                Thêm w vào S;
  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

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;            //khởi tạo giá trị cho P
                D[i] =C[1][i];    //khởi đầu các giá trị cho D
        }
        for (i=2; i<=n; i++){
                Lấy đỉnh w trong V-S sao cho D[w] nhỏ nhất;
                Thêm w vào S;
                for (mỗi đỉnh u thuộc V-S)
                        if (D[w] + C[w][u] < D[u]) {
                                D[u] =D[w] + C[w][u];
                        P[u] =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:

$$A_k[i][j]=\min(A_{k-1}[i][j], A_{k-1}[i][k]+A_{k-1}[k][j]).$$

(Kí hiệu $A_k[i][j]$ để chỉ khoảng cách ngắn nhất giữa hai đỉnh i,j ở lần lặp thứ k).

Ta cũng dùng mảng P để lưu các đỉnh trên đường đi như trong giải thuật Dijkstra. Cụ thể P[i][j]=k nếu Ak-1[i][k]+Ak-1[k][j] < Ak-1[i][j].

```
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][j] = C[i][j];
                        P[i][j]=1;
                }
        for (i=1; i<=n; i++)
                A[i][i]=0;
        for (k=1; k<=n; k++)
                for (i=1; i<=n; i++)
                        for (j=1; j<=n; j++)
```

```
                        if (A[i][k] + A[k][j] < A[i][j]){
                                A[i][j] = A[i][k] + A[k][j];
                                P[i][j] = 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 (tức là true) nếu i,j được nối nhau bởi một cạnh, ngược lại C[i][j]=0 (tức là 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][j] = C[i][j];
        for (k=1; k<=n; k++)
                for (i=1; i<=n; i++)
                        for (j=1; j<=n; j++)
                                if (A[i][j] == 0)
                                        A[i][j] =A[i][k] && A[k][j];
}
```

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.

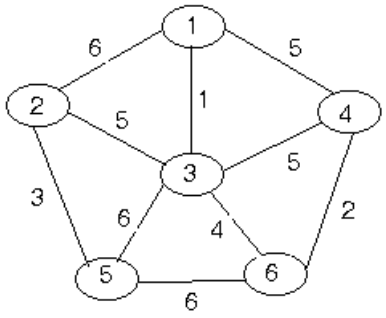
a. Giải thuật Prim

Giả sử G có n đỉnh được đánh số 1..n. Giải thuật Prim để giải bài toán này như sau:

Bắt đầu, ta khởi tạo tập U bằng 1 đỉnh nào đó, đỉnh 1 chẳng hạn, U = {1}, T=U

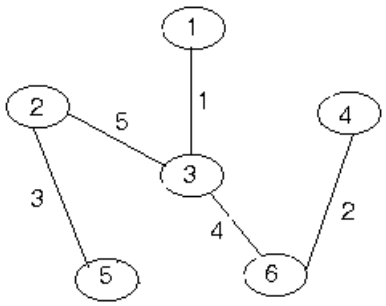
Sau đó ta lặp lại cho đến khi U=V, tại mỗi bước lặp ta chọn cạnh nhỏ nhất (u,v) sao cho u ∈ U và v ∈ V-U. Thêm v vào U và (u,v) vào T. Khi giải thuật kết thúc thì (U,T) là một cây phủ tối thiểu.

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)];
        }
}
```

b. Giải thuật Kruskal

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.

Bây giờ 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.

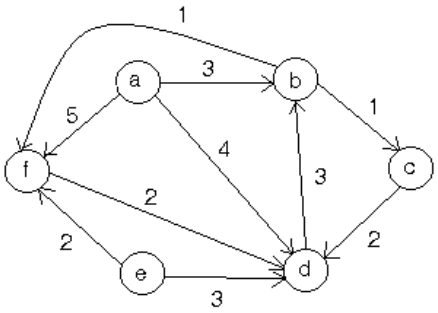
TỔNG KẾT CHƯƠNG

Chương này trình bày về cấu trúc dữ liệu trừu tượng đồ thị. Cách cài đặt đồ thị bằng ma trận kề và cài đặt bằng danh sách các con được trình bày. Nói chung, cài đặt bằng ma trận kề là đơn giản và hiệu quả trong nhiều bài toán. Tuy nhiên, nếu ma trận kề là một ma trận thưa thì sẽ lãng phí bộ nhớ rất nhiều. Trong trường hợp này nên cài đặt đồ thị bằng danh sách các đỉnh kề như đã đề cập trong chương.

Các phép toán trên đồ thị và các phép duyệt đồ thị đã được thảo luận. Hai cách duyệt đồ thị quan trọng là duyệt theo chiều rộng và duyệt theo chiều sâu đã được trình bày. Cuối cùng, các bài toán quan trọng trên đồ thị như bài toán tìm đường đi ngắn nhất từ một đỉnh đến tất cả các đỉnh khác của đồ thị, bài toán tìm bao đóng chuyển tiếp, bài toán tìm cây bao trùm tối thiểu cũng được thảo luận; các giải thuật để giải các bài toán đó đã được trình bày và hướng dẫn cách cài đặt. Do thời lượng lên lớp không cho phép trình bày nhiều hơn nữa các bài toán trên đồ thị. Độc giả quan tâm đến các bài toán khác trong lí thuyết đồ thị có thể tham khảo thêm các tài liệu liên quan đến lí thuyết đồ thị hoặc các tài liệu toán rời rạc.

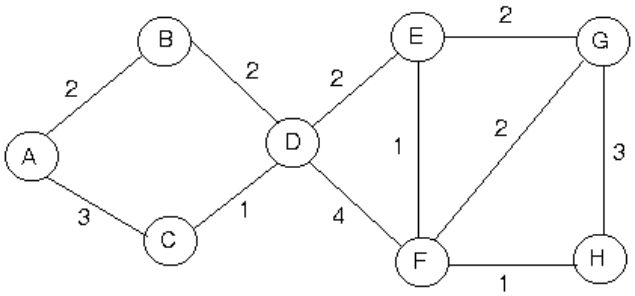
BÀI TẬP

1. Viết biểu diễn đồ thị V.8 bằng:
    a. Ma trận kề.
    b. Danh sách các đỉnh kề.



Hình V.8

2. Duyệt đồ thị hình V.8 (xét các đỉnh theo thứ tự a,b,c...)
    a. Theo chiều rộng bắt đầu từ a.
    b. Theo chiều sâu bắt đầu từ f
3. Áp dụng giải thuật Dijkstra cho đồ thị hình V.8, với đỉnh nguồn là a.



Hình V.9

4. Viết biểu diễn đồ thị V.9 bằng:
    a. Ma trận kề.
    b. Danh sách các đỉnh kề.
5. Duyệt đồ thị hình V.9 (xét các đỉnh theo thứ tự A,B,C...)
    a. Theo chiều rộng bắt đầu từ A.
    b. 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
    a. Giải thuật Prim.
    b. Giải thuật Kruskal.
8. Cài đặt đồ thị có hướng bằng ma trận kề rồi viết các giải thuật:
    a. Duyệt theo chiều rộng.

b. Duyệt theo chiều sâu.
c. Tìm đường đi ngắn nhất từ một đỉnh cho trước (Dijkstra).
d. 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:
   a. Duyệt theo chiều rộng.
   b. Duyệt theo chiều sâu.
10. Cài đặt đồ thị vô hướng bằng ma trận kề rồi viết các giải thuật:
   a. Duyệt theo chiều rộng.
   b. Duyệt theo chiều sâu.
   c. Tìm đường đi ngắn nhất từ một đỉnh cho trước (Dijkstra).
   d. Tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh (Floyd).
   e. Tìm cây bao trùm tối thiểu (Prim, Kruskal).
   f. 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:
   a. Duyệt theo chiều rộng.
   b. 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:
   a. Nhập tọa độ n đỉnh của đồ thị.
   b. 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:
      i. Đường đi ngắn nhất từ một đỉnh cho trước (Dijkstra).
      ii. Đường đi ngắn nhất giữa tất cả các cặp đỉnh (Floyd).
      iii. Cây bao trùm tối thiểu (Prim, Kruskal).
      iv. Thể hiện đồ thị lên màn hình đồ họa, các cạnh thuộc cây bao trùm tối thiểu được vẽ bằng một màu khác với các cạnh khác.

TÀI LIỆU THAM KHẢO

[1] Aho, A. V. , J. E. Hopcroft, J. D. Ullman; *Data Structure and Algorihtms*, Addison–Wesley; 1987.

[2] Đỗ Xuân Lôi; *Cấu trúc dữ liệu và giải thuật*, Nhà xuất bản Khoa học và kỹ thuật; 1995.

[3] Lê Xuân Trường; *Giáo trình cấu trúc dữ liệu bằng ngôn ngữ C++*, Nhà xuất bản Thống kê; 1999.

[4] Nicklaus Wirth; Data structures + Algorithms = Programs; Prentice Hall; 1983. Bản dịch tiếng việt " *Cấu trúc dữ liệu + giải thuật= Chương trình*", Nhà xuất bản khoa học và kỹ thuật; 1993.

[5] Nguyễn Đình Tê, Hoàng Đức Hải; *Giáo trình lý thuyết và bài tập ngôn ngữ C*, Nhà xuất bản Giáo dục; 1998.

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

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

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

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