Cấu trúc dữ liệu và giải thuật

Biên tập bởi:

Khoa CNTT ĐHSP KT Hưng Yên

Cấu trúc dữ liệu và giải thuật

Biên tập bởi:

Khoa CNTT ĐHSP KT Hưng Yên

Các tác giả:

Khoa CNTT ĐHSP KT Hưng Yên

Phiên bản trực tuyến:
http://voer.edu.vn/c/60bbf7d3

MỤC LỤC

Giải thuật và cấu trúc dữ liệu

GIẢI THUẬT

Khi viết một chương trình máy tính, ta thường cài đặt một phương pháp đã được nghĩ ra trước đó để giải quyết một vấn đề. Phương pháp này thường là độc lập với một máy tính cụ thể sẽ được dùng để cài đặt: hầu như nó thích hợp cho nhiều máy tính. Trong bất kỳ trường hợp nào, thì*phương pháp*, chứ không phải là bản thân chương trình máy tính là cái được nghiên cứu để học cách làm thế nào để tấn công vào bài toán. từ "Giải thuật" hay "*Thuật toán*" được dùng trong khoa học máy tính để mô tả một phương pháp giải bài toán thích hợp như là cài đặt các chương trình máy tính. Giải thuật chúng là các đối tượng nghiên cứu trung tâm trong hầu hết các lĩnh vực của Tin học.

Các chương trình máy tính thường quá tối ưu, đôi khi chúng ta không cần một thuật toán quá tối ưu, trừ khi một thuật toán được dùng lại nhiều lần. Nếu không chỉ cần một cài đặt đơn giản và cẩn thận là đủ để ta có thể tin tưởng rằng nó sẽ hoạt động tốt và nó có thể chạy chậm hơn 5 đến mười lần một phiên bản tốt, điều này có nghĩa nó có thể chạy chậm hơn vài giây, trong khi nếu ta chọn và thiết kế một cài đặt tối ưu và phức tạp ngay từ đầu thì có thể sẽ tốn nhiều phút, nhiều giờ… Do vậy ở đây ta sẽ xem xét các cài đặt hợp lý đơn giản của các thuật toán tốt nhất.

Thông thường để giải quyết một bài toán ta có lựa chọn nhiều thuật toán khác, việc lựa chọn một thuật toán tốt nhất là một vấn đề tương đối khó khăn phức tạp, thường cần đến một quá trình phân tích tinh vi của tin học.

Khái niệm Giải thuật có từ rất lâu do một nhà toán học người Arập phát ngôn, một trong những thuật toán nổi tiếng có từ thời cổ Hylạp là thuật toán Euclid (thuật toán tìm ước số chung lớn nhất của 2 số).

Phương pháp cộng, nhân, chia… hai số cũng là một giải thuật…

Trong Tin học khái niệm về giải thuật được trình bày như sau:

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

(Thuật toán là một dãy hữu hạn các bước, mỗi bước mô tả chính xác các phép toán hoặc hành động cần thực hiện, để giải quyết một vấn đề).

Đối tượng chỉ ra ở đây chính là Input và kết quả mong muốn chính là Output trong thuật toán Euclid ở trên

MỐI QUAN HỆ GIỮA CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết trên máy tính. Một bài toán thực tế bất kỳ đều bao gồm các đối tượng dữ liệu và các yêu cầu xử lý trên những đối tượng đó. Vì thế, để xây dựng một mô hình tin học phản ánh được bài toán thực tế cần chú trọng đến hai vấn đề :

Tổ chức biểu diễn các đối tượng thực tế :

Các thành phần dữ liệu thực tế đa dạng, phong phú và thường chứa đựng những quan hệ nào đó với nhau, do đó trong mô hình tin học của bài toán, cần phải tổ chức , xây dựng các cấu trúc thích hợp nhất sao cho vừa có thể phản ánh chính xác các dữ liệu thực tế này, vừa có thể dễ dàng dùng máy tính để xử lý. Công việc này được gọi là xây dựng *cấu trúc dữ liệu* cho bài toán.

Xây dựng các thao tác xử lý dữ liệu:

Từ những yêu cầu xử lý thực tế, cần tìm ra các giải thuật tương ứng để xác định trình tự các thao tác máy tính phải thi hành để cho ra kết quả mong muốn, đây là bước xây dựng *giải thuật* cho bài toán.

Tuy nhiên khi giải quyết một bài toán trên máy tính, chúng ta thường có khuynh hướng chỉ chú trọng đến việc xây dựng giải thuật mà quên đi tầm quan trọng của việc tổ chức dữ liệu trong bài toán. Giải thuật phản ánh các phép xử lý , còn đối tượng xử lý của giải thuật lại là dữ liệu, chính dữ liệu chứa đựng các thông tin cần thiết để thực hiện giải thuật. Để xác định được giải thuật phù hợp cần phải biết nó tác động đến loại dữ liệu nào (ví dụ để làm nhuyễn các hạt đậu , người ta dùng cách xay chứ không băm bằng dao, vì đậu sẽ văng ra ngoài) và khi chọn lựa cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào sẽ tác động đến nó (ví dụ để biểu diễn các điểm số của sinh viên người ta dùng số thực thay vì chuỗi ký tự vì còn phải thực hiện thao tác tính trung bình từ những điểm số đó). Như vậy trong một đề án tin học, giải thuật và cấu trúc dữ liệu có mối quan hệ chặt chẽ với nhau, được thể hiện qua công thức :

Cấu trúc dữ liệu + Giải thuật = Chương trình

Với một cấu trúc dữ liệu đã chọn, sẽ có những giải thuật tương ứng, phù hợp. Khi cấu trúc dữ liệu thay đổi thường giải thuật cũng phải thay đổi theo để tránh việc xử lý gượng ép, thiếu tự nhiên trên một cấu trúc không phù hợp. Hơn nữa, một cấu trúc dữ liệu tốt sẽ giúp giải thuật xử lý trên đó có thể phát huy tác dụng tốt hơn, vừa đáp ứng nhanh vừa tiết kiệm vật tư, giải thuật cũng dễ hiểu và đơn giản hơn.

Ví dụ 1.1: Một chương trình quản lý điểm thi của sinh viên cần lưu trữ các điểm số của 3 sinh viên. Do mỗi sinh viên có 4 điểm số ứng với 4 môn học khác nhau nên dữ liệu có dạng bảng như sau:

Sinh viên	Môn 1	Môn 2	Môn3	Môn4
SV 1	7	9	5	2
SV 2	5	0	9	4
SV 3	6	3	7	4

Chỉ xét thao tác xử lý là xuất điểm số các môn của từng sinh viên.

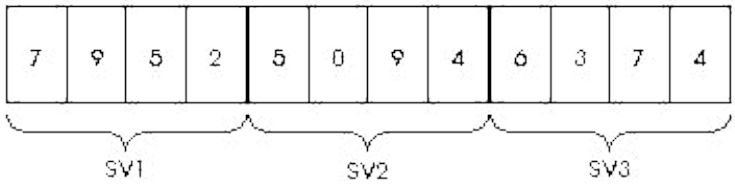
Giả sử có các phương án tổ chức lưu trữ sau:

Phương án 1 : Sử dụng mảng một chiều

Có tất cả 3(SV)*4(Môn) = 12 điểm số cần lưu trữ, do đó khai báo mảng *result* như sau :

int result [12] = {7, 9, 5, 2,5, 0, 9, 4,6, 3, 7, 4};

khi đó trong mảng *result* các phần tử sẽ được lưu trữ như sau:



Và truy xuất điểm số môn j của sinh viên i - là phần tử tại (dòng i, cột j) trong bảng - phải sử dụng một công thức xác định chỉ số tương ứng trong mảng result:

bảngđiểm(dòng i, cột j) ⇒ result[((i-1)*số cột) + j]

Ngược lại, với một phần tử bất kỳ trong mảng, muốn biết đó là điểm số của sinh viên nào, môn gì, phải dùng công thức xác định sau

result[i] ⇒ bảngđiểm (dòng((i / số cột) +1), cột (i % số cột))

Với phương án này, thao tác xử lý được cài đặt như sau :

void XuatDiem() //Xuất điểm số của tất cả sinh viên{

const int so_mon = 4;int sv,mon;for (int i=0; i<12; i+){

sv = i/so_mon; mon = i % so_mon;printf("Điểm môn %d của sv %d là: %d", mon, sv, result[i]);

} }

Phương án 2 : Sử dụng mảng 2 chiều

Khai báo mảng 2 chiều *result* có kích thước 3 dòng* 4 cột như sau :

int result[3][4] ={{ 7, 9, 5, 2},{ 5, 0, 9, 4},{ 6, 3, 7, 4 }};

khi đó trong mảng *result* các phần tử sẽ được lưu trữ như sau :

	Cột 0	Cột 1	Cột 2	Cột 3
Dòng 0	result[0][0]=7	result[0][1]=9	result[0][2]=5	result[0][3] =2
Dòng 1	result[1][0]=5	result[1][1]=0	result[1][2]=9	result[1][3]= 4
Dòng 2	result[2][0]=6	result[2][1]=3	result[2][2]=7	result[2][3]= 4

Và truy xuất điểm số môn j của sinh viên i - là phần tử tại (dòng i, cột j) trong bảng - cũng chính là phần tử nằm ở vị trí (dòng i, cột j) trong mảng

bảngđiểm(dòng i,cột j) ⇒ result[i] [j]

Với phương án này, thao tác xử lý được cài đặt như sau :

void XuatDiem() //Xuất điểm số của tất cả sinh viên

{

int so_mon = 4, so_sv =3;for (int i=0; i<so_sv; i+) for (int j=0; i<so_mon; j+) printf("Điểm môn %d của sv %d là: %d", j, i, result[i][j]);

}

NHẬN XÉT

Có thể thấy rõ phương án 2 cung cấp một cấu trúc lưu trữ phù hợp với dữ liệu thực tế hơn phương án 1, và do vậy giải thuật xử lý trên cấu trúc dữ liệu của phương án 2 cũng đơn giản, tự nhiên hơn.

CÁC TIÊU CHUẨN ĐÁNH GIÁ CẤU TRÚC DỮ LIỆU

Do tầm quan trọng đã được trình bày trong phần 1.1, nhất thiết phải chú trọng đến việc lựa chọn một phương án tổ chức dữ liệu thích hợp cho đề án. Một cấu trúc dữ liệu tốt phải thỏa mãn các tiêu chuẩn sau :

Phản ánh đúng thực tế :

Đây là tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ bài toán. Cần xem xét kỹ lưỡng cũng như dự trù các trạng thái biến đổi của dữ liệu trong chu trình sống để có thể chọn cấu trúc dữ liệu lưu trữ thể hiện chính xác đối tượng thực tế.

Ví dụ 1.2 : Một số tình huống chọn cấu trúc lưu trữ sai :

- Chọn một biến số nguyên int để lưu trữ tiền thưởng bán hàng (được tính theo công thức tiền thưởng bán hàng = trị giá hàng * 5%), do vậy sẽ làm tròn mọi giá trị tiền thưởng gây thiệt hại cho nhân viên bán hàng. Trường hợp này phải sử dụng biến số thực để phản ánh đúng kết quả của công thức tính thực tế.

- Trong trường trung học, mỗi lớp có thể nhận tối đa 28 học sinh. Lớp hiện có 20 học sinh, mỗi tháng mỗi học sinh đóng học phí $10. Chọn một biến số nguyên unsigned char (khả năng lưu trữ 0 - 255) để lưu trữ tổng học phí của lớp học trong tháng, nếu xảy ra trường hợp có thêm 6 học sinh được nhận vào lớp thì giá trị tổng học phí thu được là $260, vượt khỏi khả năng lưu trữ của biến đã chọn, gây ra tình trạng tràn, sai lệch.

Phù hợp với các thao tác trên đó:

Tiêu chuẩn này giúp tăng tính hiệu quả của đề án: việc phát triển các thuật toán đơn giản, tự nhiên hơn; chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

Ví dụ 1.3 : Một tình huống chọn cấu trúc lưu trữ không phù hợp:

Cần xây dựng một chương trình soạn thảo văn bản, các thao tác xử lý thường xảy ra là chèn, xoá sửa các ký tự trên văn bản. Trong thời gian xử lý văn bản, nếu chọn cấu trúc lưu trữ văn bản trực tiếp lên tập tin thì sẽ gây khó khăn khi xây dựng các giải thuật cập nhật văn bản và làm chậm tốc độ xử lý của chương trình vì phải làm việc trên bộ nhớ ngoài. Trường hợp này nên tìm một cấu trúc dữ liệu có thể tổ chức ở bộ nhớ trong để lưu trữ văn bản suốt thời gian soạn thảo.

LƯU Ý :

Đối với mỗi ứng dụng , cần chú ý đến thao tác nào được sử dụng nhiều nhất để lựa chọn cấu trúc dữ liệu cho thích hợp.

Tiết kiệm tài nguyên hệ thống:

Cấu trúc dữ liệu chỉ nên sử dụng tài nguyên hệ thống vừa đủ để đảm nhiệm được chức năng của nó. Thông thường có 2 loại tài nguyên cần lưu tâm nhất : CPU và bộ nhớ. Nếu tổ chức sử dụng đề án cần có những xử lý nhanh thì khi chọn cấu trúc dữ liệu yếu tố tiết kiệm thời gian xử lý phải đặt nặng hơn tiêu chuẩn sử dụng tối ưu bộ nhớ, và ngược lại.

Ví dụ 1.4: Một số tình huống chọn cấu trúc lưu trữ lãng phí:

- Sử dụng biến int (2 bytes) để lưu trữ một giá trị cho biết tháng hiện hành . Biết rằng tháng chỉ có thể nhận các giá trị từ 1-12, nên chỉ cần sử dụng kiểu char (1 byte) là đủ.

- Để lưu trữ danh sách học viên trong một lớp, sử dụng mảng 50 phần tử (giới hạn số học viên trong lớp tối đa là 50). Nếu số lượng học viên thật sự ít hơn 50, thì gây lãng phí. Trường hợp này cần có một cấu trúc dữ liệu linh động hơn mảng- ví dụ danh sách liên kết - sẽ được bàn đến trong các bài sau.

Phân tích và thiết kế bài toán

CÁC BƯỚC CƠ BẢN ĐỂ GIẢI QUYẾT BÀI TOÁN

Xác định bài toán

Input → Process → Output (Dữ liệu vào → Xử lý → Kết quả ra)

Việc xác định bài toán tức là phải xác định xem ta phải giải quyết vấn đề gì?, với giả thiết nào đã cho và lời giải cần phải đạt những yêu cầu gì. Khác với bài toán thuần tuý toán học chỉ cần xác định rõ giả thiết và kết luận chứ không cần xác định yêu cầu về lời giải, đôi khi những bài toán tin học ứng dụng trong thực tế chỉ cần tìm lời giải tốt tới mức nào đó, thậm chí là tồi ở mức chấp nhận được. Bởi lời giải tốt nhất đòi hỏi quá nhiều thời gian và chi phí.

Ví dụ 2.1:

Khi cài đặt các hàm số phức tạp trên máy tính. Nếu tính bằng cách khai triển chuỗi vô hạn thì độ chính xác cao hơn nhưng thời gian chậm hơn hàng tỉ lần so với phương pháp xấp xỉ. Trên thực tế việc tính toán luôn luôn cho phép chấp nhận một sai số nào đó nên các hàm số trong máy tính đều được tính bằng phương pháp xấp xỉ của giải tích số.

Xác định đúng yêu cầu bài toán là rất quan trọng bởi nó ảnh hưởng tới cách thức giải quyết và chất lượng của lời giải. Một bài toán thực tế thường cho bởi những thông tin khá mơ hồ và hình thức, ta phải phát biểu lại một cách chính xác và chặt chẽ để hiểu đúng bài toán.

Ví dụ 2.2:

Bài toán: Một dự án có n người tham gia thảo luận, họ muốn chia thành các nhóm và mỗi nhóm thảo luận riêng về một phần của dự án. Nhóm có bao nhiêu người thì được trình lên bấy nhiêu ý kiến. Nếu lấy ở mỗi nhóm một ý kiến đem ghép lại thì được một bộ ý kiến triển khai dự án. Hãy tìm cách chia để số bộ ý kiến cuối cùng thu được là lớn nhất.

Phát biểu lại: Cho một số nguyên dương n, tìm các phân tích n thành tổng các số nguyên dương sao cho tích của các số đó là lớn nhất.

Trên thực tế, ta nên xét một vài trường hợp cụ thể để thông qua đó hiểu được bài toán rõ hơn và thấy được các thao tác cần phải tiến hành. Đối với những bài toán đơn giản, đôi khi chỉ cần qua ví dụ là ta đã có thể đưa về một bài toán quen thuộc để giải.

Tìm cấu trúc dữ liệu biểu diễn bài toán

Khi giải một bài toán, ta cần phải định nghĩa tập hợp dữ liệu để biểu diễn tình trạng cụ thể. Việc lựa chọn này tuỳ thuộc vào vấn đề cần giải quyết và những thao tác sẽ tiến hành trên dữ liệu vào. Có những thuật toán chỉ thích ứng với một cách tổ chức dữ liệu nhất định, đối với những cách tổ chức dữ liệu khác thì sẽ kém hiệu quả hoặc không thể thực hiện được. Chính vì vậy nên bước xây dựng cấu trúc dữ liệu không thể tách rời bước tìm kiếm thuật toán giải quyết vấn đề.

Các tiêu chuẩn khi lựa chọn cấu trúc dữ liệu

- Cấu trúc dữ liệu trước hết phải biểu diễn được đầy đủ các thông tin nhập và xuất của bài toán
- Cấu trúc dữ liệu phải phù hợp với các thao tác của thuật toán mà ta lựa chọn để giải quyết bài toán.
- Cấu trúc dữ liệu phải cài đặt được trên máy tính với ngôn ngữ lập trình đang sử dụng

Đối với một số bài toán, trước khi tổ chức dữ liệu ta phải viết một đoạn chương trình nhỏ để khảo sát xem dữ liệu cần lưu trữ lớn tới mức độ nào.

Xác định thuật toán

Thuật toán là một hệ thống chặt chẽ và rõ ràng các quy tắc nhằm xác định một dãy thao tác trên cấu trúc dữ liệu sao cho: Với một bộ dữ liệu vào, sau một số hữu hạn bước thực hiện các thao tác đã chỉ ra, ta đạt được mục tiêu đã định.

Các đặc trưng của thuật toán

- Tính đơn nghĩa

Ở mỗi bước của thuật toán, các thao tác phải hết sức rõ ràng, không gây nên sự nhập nhằng, lộn xộn, tuỳ tiện, đa nghĩa.

Không nên lẫn lộn tính đơn nghĩa và tính đơn định: Người ta phân loại thuật toán ra làm hai loại: Đơn định (Deterministic) và Ngẫu nhiên (Randomized). Với hai bộ dữ liệu giống nhau cho trước làm input, thuật toán đơn định sẽ thi hành các mã lệnh giống nhau và cho kết quả giống nhau, còn thuật toán ngẫu nhiên có thể thực hiện theo những mã lệnh khác nhau và cho kết quả khác nhau. Ví dụ như yêu cầu chọn một số tự nhiên x: $a \leq x \leq b$, nếu ta viết x = a hay x = b hay x = (a + b) div 2, thuật toán sẽ luôn cho một giá trị duy nhất với dữ liệu vào là hai số tự nhiên a và b. Nhưng nếu ta viết x = a + Random(b - a + 1) thì sẽ có thể thu được các kết quả khác nhau trong mỗi lần thực hiện với input là a và b tuỳ theo máy tính và bộ tạo số ngẫu nhiên.

- Tính dừng

Thuật toán không được rơi vào quá trình vô hạn, phải dừng lại và cho kết quả sau một số hữu hạn bước.

- Tính đúng

Sau khi thực hiện tất cả các bước của thuật toán theo đúng quá trình đã định, ta phải được kết quả mong muốn với mọi bộ dữ liệu đầu vào. Kết quả đó được kiểm chứng bằng yêu cầu bài toán.

- Tính phổ dụng

Thuật toán phải dễ sửa đổi để thích ứng được với bất kỳ bài toán nào trong một lớp các bài toán và có thể làm việc trên các dữ liệu khác nhau.

Tính khả thi

- Kích thước phải đủ nhỏ: Ví dụ: Một thuật toán sẽ có tính hiệu quả bằng 0 nếu lượng bộ nhớ mà nó yêu cầu vượt quá khả năng lưu trữ của hệ thống máy tính.
- Thuật toán phải chuyển được thành chương trình: Ví dụ một thuật toán yêu cầu phải biểu diễn được số vô tỉ với độ chính xác tuyệt đối là không hiện thực với các hệ thống máy tính hiện nay
- Thuật toán phải được máy tính thực hiện trong thời gian cho phép, điều này khác với lời giải toán (Chỉ cần chứng minh là kết thúc sau hữu hạn bước). Ví dụ như xếp thời khoá biểu cho một học kỳ thì không thể cho máy tính chạy tới học kỳ sau mới ra được.

Ví dụ 2.3:

Input: 2 số nguyên tự nhiên a và b không đồng thời bằng 0

Output: Ước số chung lớn nhất của a và b

Thuật toán sẽ tiến hành được mô tả như sau: (Thuật toán Euclide)

Bước 1 (Input): Nhập a và b: Số tự nhiên

Bước 2: Nếu $b \neq 0$ thì chuyển sang bước 3, nếu không thì bỏ qua bước 3, đi làm bước 4

Bước 3: Đặt r = a mod b; Đặt a = b; Đặt b = r; Quay trở lại bước 2.

Bước 4 (Output): Kết luận ước số chung lớn nhất phải tìm là giá trị của a. Kết thúc thuật toán.

Khi mô tả thuật toán bằng ngôn ngữ tự nhiên, ta không cần phải quá chi tiết các bước và tiến trình thực hiện mà chỉ cần mô tả một cách hình thức đủ để chuyển thành ngôn ngữ lập trình. Viết sơ đồ các thuật toán đệ quy là một ví dụ.

Đối với những thuật toán phức tạp và nặng về tính toán, các bước và các công thức nên mô tả một cách tường minh và chú thích rõ ràng để khi lập trình ta có thể nhanh chóng tra cứu.

Đối với những thuật toán kinh điển thì phải thuộc. Khi giải một bài toán lớn trong một thời gian giới hạn, ta chỉ phải thiết kế tổng thể còn những chỗ đã thuộc thì cứ việc lắp ráp vào.

Tính đúng đắn của những mô-đun đã thuộc ta không cần phải quan tâm nữa mà tập trung giải quyết các phần khác.

Lập trình

Sau khi đã có thuật toán, ta phải tiến hành lập trình thể hiện thuật toán đó. Muốn lập trình đạt hiệu quả cao, cần phải có kỹ thuật lập trình tốt. Kỹ thuật lập trình tốt thể hiện ở kỹ năng viết chương trình, khả năng gỡ rối và thao tác nhanh. Lập trình tốt không phải chỉ cần nắm vững ngôn ngữ lập trình là đủ, phải biết cách viết chương trình uyển chuyển và phát triển dần dần để chuyển các ý tưởng ra thành chương trình hoàn chỉnh. Kinh nghiệm cho thấy một thuật toán hay nhưng do cài đặt vụng về nên khi chạy lại cho kết quả sai hoặc tốc độ chậm.

Thông thường, ta không nên cụ thể hoá ngay toàn bộ chương trình mà nên tiến hành theo phương pháp tinh chế từng bước (Stepwise refinement):

- Ban đầu, chương trình được thể hiện bằng ngôn ngữ tự nhiên, thể hiện thuật toán với các bước tổng thể, mỗi bước nêu lên một công việc phải thực hiện.
- Một công việc đơn giản hoặc là một đoạn chương trình đã được học thuộc thì ta tiến hành viết mã lệnh ngay bằng ngôn ngữ lập trình.
- Một công việc phức tạp thì ta lại chia ra thành những công việc nhỏ hơn để lại tiếp tục với những công việc nhỏ hơn đó.

Trong quá trình tinh chế từng bước, ta phải đưa ra những biểu diễn dữ liệu. Như vậy cùng với sự tinh chế các công việc, dữ liệu cũng được tinh chế dần, có cấu trúc hơn, thể hiện rõ hơn mối liên hệ giữa các dữ liệu.

Phương pháp tinh chế từng bước là một thể hiện của tư duy giải quyết vấn đề từ trên xuống, giúp cho người lập trình có được một định hướng thể hiện trong phong cách viết chương trình. Tránh việc mò mẫm, xoá đi viết lại nhiều lần, biến chương trình thành tờ giấy nháp.

Kiểm thử

Chạy thử và tìm lỗi

Chương trình là do con người viết ra, mà đã là con người thì ai cũng có thể nhầm lẫn. Một chương trình viết xong chưa chắc đã chạy được ngay trên máy tính để cho ra kết quả mong muốn. Kỹ năng tìm lỗi, sửa lỗi, điều chỉnh lại chương trình cũng là một kỹ năng quan trọng của người lập trình. Kỹ năng này chỉ có được bằng kinh nghiệm tìm và sửa chữa lỗi của chính mình.

Có ba loại lỗi:

- Lỗi cú pháp: Lỗi này hay gặp nhất nhưng lại dễ sửa nhất, chỉ cần nắm vững ngôn ngữ lập trình là đủ. Một người được coi là không biết lập trình nếu không biết sửa lỗi cú pháp.
- Lỗi cài đặt: Việc cài đặt thể hiện không đúng thuật toán đã định, đối với lỗi này thì phải xem lại tổng thể chương trình, kết hợp với các chức năng gỡ rối để sửa lại cho đúng.
- Lỗi thuật toán: Lỗi này ít gặp nhất nhưng nguy hiểm nhất, nếu nhẹ thì phải điều chỉnh lại thuật toán, nếu nặng thì có khi phải loại bỏ hoàn toàn thuật toán sai và làm lại từ đầu.

Xây dựng các bộ test

Có nhiều chương trình rất khó kiểm tra tính đúng đắn. Nhất là khi ta không biết kết quả đúng là thế nào?. Vì vậy nếu như chương trình vẫn chạy ra kết quả (không biết đúng sai thế nào) thì việc tìm lỗi rất khó khăn. Khi đó ta nên làm các bộ test để thử chương trình của mình.

Các bộ test nên đặt trong các file văn bản, bởi việc tạo một file văn bản rất nhanh và mỗi lần chạy thử chỉ cần thay tên file dữ liệu vào là xong, không cần gõ lại bộ test từ bàn phím. Kinh nghiệm làm các bộ test là:

Bắt đầu với một bộ test nhỏ, đơn giản, làm bằng tay cũng có được đáp số để so sánh với kết quả chương trình chạy ra.

Tiếp theo vẫn là các bộ test nhỏ, nhưng chứa các giá trị đặc biệt hoặc tầm thường. Kinh nghiệm cho thấy đây là những test dễ sai nhất.

Các bộ test phải đa dạng, tránh sự lặp đi lặp lại các bộ test tương tự.

Có một vài test lớn chỉ để kiểm tra tính chịu đựng của chương trình mà thôi. Kết quả có đúng hay không thì trong đa số trường hợp, ta không thể kiểm chứng được với test này.

Lưu ý rằng chương trình chạy qua được hết các test không có nghĩa là chương trình đó đã đúng. Bởi có thể ta chưa xây dựng được bộ test làm cho chương trình chạy sai. Vì vậy nếu có thể, ta nên tìm cách chứng minh tính đúng đắn của thuật toán và chương trình, điều này thường rất khó.

Tối ưu chương trình

Một chương trình đã chạy đúng không có nghĩa là việc lập trình đã xong, ta phải sửa đổi lại một vài chi tiết để chương trình có thể chạy nhanh hơn, hiệu quả hơn. Thông thường, trước khi kiểm thử thì ta nên đặt mục tiêu viết chương trình sao cho đơn giản, miễn sao chạy ra kết quả đúng là được, sau đó khi tối ưu chương trình, ta xem lại những chỗ nào viết chưa tốt thì tối ưu lại mã lệnh để chương trình ngắn hơn, chạy nhanh hơn. Không nên viết tới đâu tối ưu mã đến đó, bởi chương trình có mã lệnh tối ưu thường phức tạp và khó kiểm soát.

Việc tối ưu chương trình nên dựa trên các tiêu chuẩn sau:

- Tính tin cậy

Chương trình phải chạy đúng như dự định, mô tả đúng một giải thuật đúng. Thông thường khi viết chương trình, ta luôn có thói quen kiểm tra tính đúng đắn của các bước mỗi khi có thể.

- Tính uyển chuyển

Chương trình phải dễ sửa đổi. Bởi ít có chương trình nào viết ra đã hoàn hảo ngay được mà vẫn cần phải sửa đổi lại. Chương trình viết dễ sửa đổi sẽ làm giảm bớt công sức của lập trình viên khi phát triển chương trình.

- Tính trong sáng

Chương trình viết ra phải dễ đọc dễ hiểu, để sau một thời gian dài, khi đọc lại còn hiểu mình làm cái gì?. Để nếu có điều kiện thì còn có thể sửa sai (nếu phát hiện lỗi mới), cải tiến hay biến đổi để được chương trình giải quyết bài toán khác. Tính trong sáng của chương trình phụ thuộc rất nhiều vào công cụ lập trình và phong cách lập trình.

- Tính hữu hiệu

Chương trình phải chạy nhanh và ít tốn bộ nhớ, tức là tiết kiệm được cả về không gian và thời gian. Để có một chương trình hữu hiệu, cần phải có giải thuật tốt và những tiểu xảo khi lập trình. Tuy nhiên, việc áp dụng quá nhiều tiểu xảo có thể khiến chương trình trở nên rối rắm, khó hiểu khi sửa đổi. Tiêu chuẩn hữu hiệu nên dừng lại ở mức chấp nhận được, không quan trọng bằng ba tiêu chuẩn trên. Bởi phần cứng phát triển rất nhanh, yêu cầu hữu hiệu không cần phải đặt ra quá nặng.

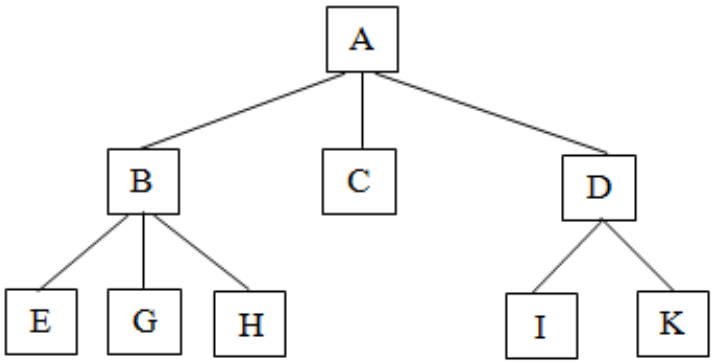
Từ những phân tích ở trên, chúng ta nhận thấy rằng việc làm ra một chương trình đòi hỏi rất nhiều công đoạn và tiêu tốn khá nhiều công sức. Chỉ một công đoạn không hợp lý sẽ làm tăng chi phí viết chương trình. Nghĩ ra cách giải quyết vấn đề đã khó, biến ý tưởng đó thành hiện thực cũng không dễ chút nào.

Những cấu trúc dữ liệu và giải thuật đề cập tới trong chuyên đề này là những kiến thức rất phổ thông, một người học lập trình không sớm thì muộn cũng phải biết tới. Chỉ hy vọng rằng khi học xong chuyên đề này, qua những cấu trúc dữ liệu và giải thuật hết sức mẫu mực, chúng ta rút ra được bài học kinh nghiệm: Đừng bao giờ viết chương trình khi mà chưa suy xét kỹ về giải thuật và những dữ liệu cần thao tác, bởi như vậy ta dễ mắc phải hai sai lầm trầm trọng: hoặc là sai về giải thuật, hoặc là giải thuật không thể triển khai nổi trên một cấu trúc dữ liệu không phù hợp. Chỉ cần mắc một trong hai lỗi đó thôi thì nguy cơ sụp đổ toàn bộ chương trình là hoàn toàn có thể, càng cố chữa càng bị rối, khả năng hầu như chắc chắn là phải làm lại từ đầu

MODUL HÓA VÀ VIỆC GIẢI QUYẾT BÀI TOÁN

Trong thực tế các bài toán được giải trên máy tính điện tử ngày càng nhiều và càng phức tạp. Các giải thuật ngày càng có qui mô lớn và khó thiết lập.

Để đơn giản hoá bài toán người ta tiến hành phân chia bài toán lớn thành các bài toán nhỏ. Có nghĩa là nếu bài toán lớn là một modul chính thì cần chia nó ra thành các modul con, đến lượt nó mỗi modul con này lại có thể chia tiếp ra thành các modul con khác ứng với các phần việc cơ bản mà người ta đã biết cách giải quyết. Việc tổ chức lời giải của bài toán có thể được thực hiện theo cấu trúc phân cấp như sau :



Chiến lược giải quyết bài toán theo kiểu như vậy gọi là chiến lược “chia để trị” (devide and conquare). Để thể hiện chiến lược này người ta sử dụng phương pháp thiết kế từ trên “*đỉnh - xuống*” (top - down design). Đó là cách phân tích tổng quát toàn bộ mọi vấn đề, xuất phát từ dữ kiện và các mục tiêu đề ra, để đề cập đến những công việc chủ yếu

rồi sau đó mới đi dần vào giải quyết các phần cụ thể một cách chi tiết hơn(gọi đó là cách thiết kế từ khái quát đến chi tiết) .

Ví dụ : Chủ tịch hội đồng xét cấp học bổng của nhà trường yêu cầu chúng ta:

" Dùng máy tính điện tử để quản lý và bảo trì các hồ sơ về học bổng của các sinh viên ở diện được tài trợ, đồng thời thường kỳ phải lập các báo cáo tổng kết để đệ trình lên Bộ"

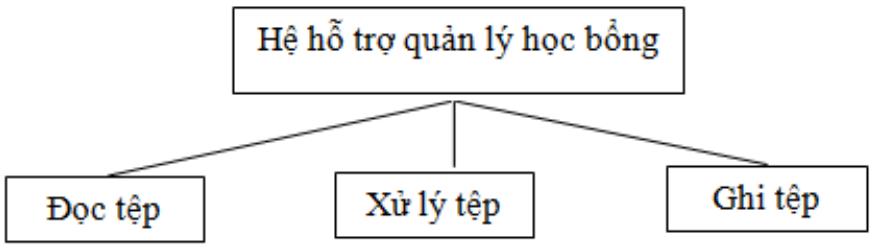
Như vậy trước hết ta phải hình dung được cụ thể hơn đầu vào và đầu ra của bài toán.

Có thể coi như ta đã có 1 tập hồ sơ (file) bao gồm các bản ghi (records) về các thông tin liên quan đến học bổng của sinh viên như : Mã SV, Điểm TB, điểm đạo đức, khoản tiền tài trợ. Và chương trình lập ra phải tạo điều kiện cho người sử dụng giải quyết được các yêu cầu sau:

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

Xuất phát từ những nhận định trên, giải thuật xử lý phải giải quyết 3 nhiệm vụ chính như sau:

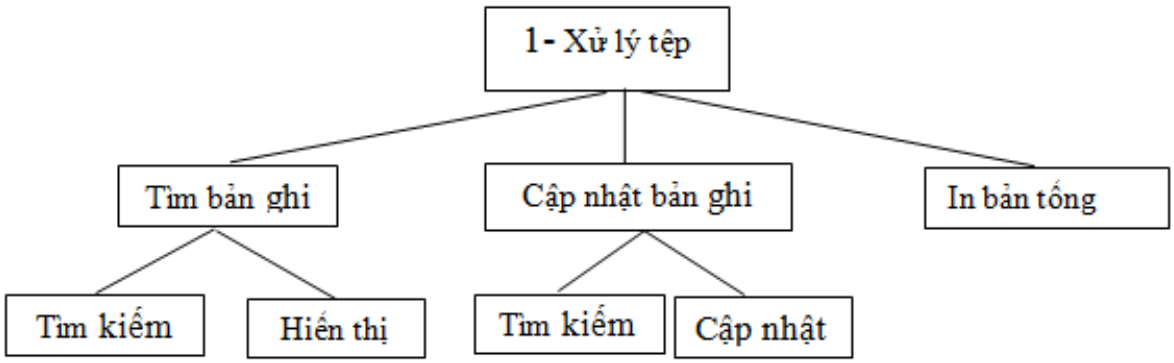
1. Những thông tin về sinh viên được học bổng, lưu trữ trên đĩa phải được đọc vào bộ nhớ trong để có thể xử lý (gọi là nhiệm vụ "đọc tệp")
2. Xử lý các thông tin này để tạo ra kết quả mong muốn (nhiệm vụ "xử lý tệp")
3. Sao chép những thông tin đã được cập nhật vào tệp trên đĩa để lưu trữ cho việc xử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



Các nhiệm vụ ở mức đầu này tương đối phức tạp thường chia thành các nhiệm vụ con. Chẳng hạn, nhiệm vụ "xử lý tệp" sẽ được phân thành 3 nhiệm vụ con tương ứng giải quyết 3 yêu cầu chính được nêu trên:

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

Những nhiệm vụ con này cũng có thể lại được chia nhỏ thành các nhiệm vụ theo sơ đồ sau:



Cách thiết kế giải thuật theo kiểu top - down này sẽ giúp cho việc giải quyết bài toán được định hướng rõ ràng, dễ dàng thực hiện và nó chính là nền tảng cho việc lập trình cấu trúc.

PHƯƠNG PHÁP TINH CHỈNH DẦN TỪNG BƯỚC (Stepwise refinement)

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

Phương pháp này được tiến hành theo sơ đồ:

CTDL → CTDL lưu trữ → Cách cài đặt DL hợp lý → CTDL tiền định.

Trong quá trình thực hiện giải thuật ban đầu chương trình được thực hiện bằng ngôn ngữ tự nhiên phản ánh ý chính của công việc cần làm. Đến các bước sau những ý đó sẽ được chi tiết hoá dần dần tương ứng với những công việc nhỏ hơn. Ta gọi đó là các bước tinh chỉnh, sự tinh chỉnh này sẽ được hướng về phía ngôn ngữ lập trình mà ta đã chọn. Càng ở các bước sau lời lẽ đặc tả các công việc xử lý sẽ được thay thế bởi các câu lệnh hướng tới câu lệnh của ngôn ngữ lập trình.

Ví dụ 2.4: Giả sử ta muốn lập chương trình sắp xếp một dãy n số nguyên khác nhau theo thứ tự tăng dần.

Giải thuật có thể được phác thảo một cách thủ công đơn giản như sau:“ Coi các phần tử của dãy số như các phần tử của một véc tơ (có cấu trúc mảng một chiều) và dãy này được lưu trữ bởi một vec tơ lưu trữ gồm n từ máy kế tiếp ở bộ nhớ trong (a1, a2, . . ., an) mỗi từ a_i lưu trữ một phần tử thứ i ($1 \le i \le n$) của dãy số. Qui ước dãy số được sắp xếp rồi vẫn để tại chỗ cũ như đã cho.

Từ các số đã cho chọn ra một số nhỏ nhất, đặt nó vào cuối dãy đã được sắp xếp. Sau đó tiến hành so sánh với số hiện đang ở vị trí đó nếu như nó khác với số này thì phải tiến hành đổi chỗ. Công việc cứ lặp lại cho đến chỉ dãy số chưa được sắp xếp trở thành rỗng”.

Bước tinh chỉnh đầu tiên được thực hiện nhờ ngôn ngữ tựa C như sau:

*For(*int i =1, i ≤ n, i++)

{

+ Xét từ a i đến a n để tìm số nhỏ nhất a j

+ Đổi chỗ giữa ai và aj }

Các bước tiến hành:

+ B1: Xét dãy đã cho. Tìm số nguyên nhỏ nhất a_j trong các số từ a_i đến a_n

+ B2: Đổi chỗ giữa a_j và a_i

Nhiệm vụ đầu có thể được thực hiện bằng cách:

“ *Thoạt tiên coi ai là* “*số nhỏ nhất*” *tạm thời; lần lượt so sánh ai với ai+1,ai+2, . . . Khi đã so sánh với an rồi thì số nhỏ nhất sẽ được xác định*.”

Để xác định ta phải chỉ ra vị trí của nó, hay nói cách khác là nắm được chỉ số của phần tử ấy thông qua một khâu trung gian:

{Bước tinh chỉnh 1}

j =i;

*For(*int k =*j*+1, k ≤ n, k++)

if (a k < a j) j=k;

{Bước tinh chỉnh 2}

B = a i ; a i = a j ; a j = B;

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

Void Sort(A,n)

{các biến i,j,k kiểu nguyên; biến trung gian B cùng kiểu A}

1. *For(*int i =1, i ≤ n, i++)

{

2- {Chọn số nhỏ nhất}

j =i;

*For(*int k =*j*+1, k ≤ n, k++)

if (A[k] < A[j]) j = k;

1. {Đổi chỗ} B = A[i]; A[i]= A[j]; A[j] = B;

}

Ví dụ 2: Cho ma trận vuông n × n các số nguyên. Hãy in ra các phần tử thuộc đường chéo song song với đường chéo chính theo thứ tự tính từ phải sang trái.

$$A = \begin{pmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{pmatrix}$$

Chọn cách in từ phải sang trái ta có kết quả:

a14

a13 a24

a12 a23 a34

a11 a22 a33 a44

a21 a32 a43

a31 a42

a41

Nửa tam giác trên các cột giảm dần từ n → 1, đưa ra các phần tử thuộc đường chéo ứng với cột j

Nửa tam giác dưới các hàng tăng từ 2 → n. Với mỗi hàng như vậy ta phải đưa ra các phần tử thuộc đường chéo tương đương với hàng i đã cho.

Ta có thể phác hoạ giải thuật như sau:

1. Nhập cấp ma trận n
2. Nhập các phần tử của ma trận A[i,j]
3. In các đường chéo song song với đường chéo chính.

Hai nhiệm vụ (1) và (2) có thể dễ dàng thể hiện bằng Pascal:

1. Cin>>*n;*

    1. *for (i = 1, i <= n, i++)*

for (j = 1, j <= n,j++)

Cout <<a[i] [j];

Nhiệm vụ 3 cần phải được phân tích rõ ràng hơn:

Về đường chéo ta có thể phân ra làm 2 loại:

+ Đường chéo ứng với cột từ n đến 1

+ Đường chéo ứng với hàng từ 2 đến n

Cho nên ta tách ra 2 nhiệm vụ con là:

    1. *for (j = n, j> = 1, j--)*

in đường chéo ứng với cột j

3.2. For (i = 2, i<=n, i++)

in đường chéo ứng với hàng i

Tới đây phải chi tiết hơn công việc " in đường chéo ứng với cột j"

Với j = n thì in một phần tử bằng cột j

Với j = n - 1 thì in 2 phần tử hàng 1 cột j

hàng 2 cột j+1

Với j = n - 2 thì in 3 phần tử hàng 1 cột j

hàng 2 cột j+1

hàng 3 cột j+2

Ta nhận thấy số lượng các phần tử được in chính là (n - j + 1), còn phần tử được in chính là A[i, j + (i - 1)] với i nhận giá trị từ 1 tới (n - j + 1)

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

```
For ( i = 1, i<= (n - j + 1), i++)
    Cout<<a[i],[j + (i - 1)]<<endl
```

Ta tận dụng khả năng của Pascal để in mỗi phần tử trong một khoảng cách 8 kí tự và mỗi đường chéo được in trên một dòng, sau đó để cách một dòng trống.

1. tương tự

For (j = 1,j<= (n - i + 1),j++)

Cout<<a ? i + (j - 1)] [j ? ,<<endl

Toàn bộ giải thuật này có thể được thể hiện bằng ngôn ngữ C như sau:

```
{
do
  Cout<< "Cho biết kích thước ma trận";Cin>>n;
while  (0<n) and (n<=max);
```

//Nhập các pt của mảng

clrscr();

//In đường chéo ma trận

For (j=n,j<=n, j--)

{

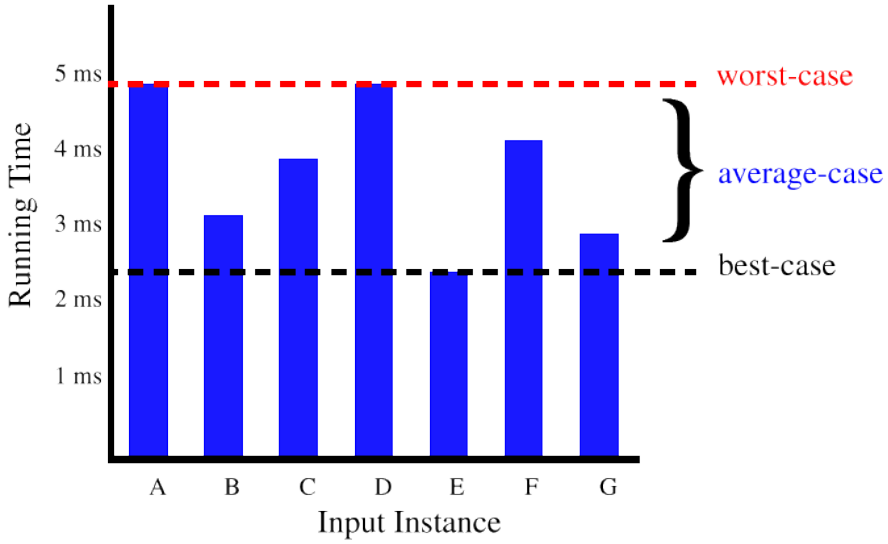
```
        For ( i = 1, i<= (n - j + 1), i++)
         Cout<<a[i],[j + (i - 1)]<<endl
        getch();
       }
For ( i = 2, i<=n, i++)
   {
    For (j = 1,j<= (n - i + 1),j++)
      Cout<<a[i + (j - 1)] [j],<<endl
       getch();
   }
}
```

Phân tích thời gian thực hiện thuật toán

ĐỘ PHỨC TẠP GIẢI THUẬT

Giới thiệu

Hầu hết các bài toán đều có nhiều thuật toán khác nhau để giải quyết chúng. Như vậy, làm thế nào để chọn được sự cài đặt tốt nhất? Đây là một lĩnh vực được phát triển tốt trong nghiên cứu về khoa học máy tính. Chúng ta sẽ thường xuyên có cơ hội tiếp xúc với các kết quả nghiên cứu mô tả các tính năng của các thuật toán cơ bản. Tuy nhiên, việc so sánh các thuật toán rất cần thiết và chắc chắn rằng một vài dòng hướng dẫn tổng quát về phân tích thuật toán sẽ rất hữu dụng.



Khi nói đến hiệu quả của một thuật toán, người ta thường quan tâm đến chi phí cần dùng để thực hiện nó. Chi phí này thể hiện qua việc sử dụng tài nguyên như bộ nhớ, thời gian sử dụng CPU, … Ta có thể đánh giá thuật toán bằng phương pháp thực nghiệm thông qua việc cài đặt thuật toán rồi chọn các bộ dữ liệu thử nghiệm. Thống kê các thông số nhận được khi chạy các dữ liệu này ta sẽ có một đánh giá về thuật toán.

Tuy nhiên, phương pháp thực nghiệm gặp một số nhược điểm sau khiến cho nó khó có khả năng áp dụng trên thực tế:

- Do phải cài đặt bằng một ngôn ngữ lập trình cụ thể nên thuật toán sẽ chịu sự hạn chế của ngữ lập trình này.
- Đồng thời, hiệu quả của thuật toán sẽ bị ảnh hưởng bởi trình độ của người cài đặt.

- Việc chọn được các bộ dữ liệu thử đặc trưng cho tất cả tập các dữ liệu vào của thuật toán là rất khó khăn và tốn nhiều chi phí.
- Các số liệu thu nhận được phụ thuộc nhiều vào phần cứng mà thuật toán được thử nghiệm trên đó. Điều này khiến cho việc so sánh các thuật toán khó khăn nếu chúng được thử nghiệm ở những nơi khác nhau.

Vì những lý do trên, người ta đã tìm kiếm những phương pháp đánh giá thuật toán hình thức hơn, ít phụ thuộc môi trường cũng như phần cứng hơn. Một phương pháp như vậy là phương pháp đánh giá thuật toán theo hướng xấp xỉ tiệm cận qua các khái niệm toán học O-lớn O(), O-nhỏ o()

Thông thường các vấn đề mà chúng ta giải quyết có một "kích thước" tự nhiên (thường là số lượng dữ liệu được xử lý) mà chúng ta sẽ gọi là N. Chúng ta muốn mô tả tài nguyên cần được dùng (thông thường nhất là thời gian cần thiết để giải quyết vấn đề) như một hàm số theo N. Chúng ta quan tâm đến trường hợp trung bình, tức là thời gian cần thiết để xử lý dữ liệu nhập thông thường, và cũng quan tâm đến trường hợp xấu nhất, tương ứng với thời gian cần thiết khi dữ liệu rơi vào trường hợp xấu nhất có thể có.

Việc xác định chi phí trong trường hợp trung bình thường được quan tâm nhiều nhất vì nó đại diện cho đa số trường hợp sử dụng thuật toán. tuy nhiên, việc xác định chi phí trung bình này lại gặp nhiều khó khăn. Vì vậy, trong nhiều trường hợp, người ta xác định chi phí trong trường hợp xấu nhất (chặn trên) thay cho việc xác định chi phí trong trường hợp trung bình. Hơn nữa, trong một số bài toán, việc xác định chi phí trong trường hợp xấu nhất là rất quan trọng. Ví dụ, các bài toán trong hàng không, phẫu thuật, …

Các bước phân tích thuật toán

Bước đầu tiên trong việc phân tích một thuật toán là xác định đặc trưng dữ liệu sẽ được dùng làm dữ liệu nhập của thuật toán và quyết định phân tích nào là thích hợp. Về mặt lý tưởng, chúng ta muốn rằng với một phân bố tùy ý được cho của dữ liệu nhập, sẽ có sự phân bố tương ứng về thời gian hoạt động của thuật toán. Chúng ta không thể đạt tới điều lý tưởng nầy cho bất kỳ một thuật toán không tầm thường nào, vì vậy chúng ta chỉ quan tâm đến bao của thống kê về tính năng của thuật toán bằng cách cố gắng chứng minh thời gian chạy luôn luôn nhỏ hơn một "chận trên" bất chấp dữ liệu nhập như thế nào và cố gắng tính được thời gian chạy trung bình cho dữ liệu nhập "ngẫu nhiên".

Bước thứ hai trong phân tích một thuật toán là nhận ra các thao tác trừu tượng của thuật toán để tách biệt sự phân tích với sự cài đặt. Ví dụ, chúng ta tách biệt sự nghiên cứu có bao nhiêu phép so sánh trong một thuật toán sắp xếp khỏi sự xác định cần bao nhiêu micro giây trên một máy tính cụ thể; yếu tố thứ nhất được xác định bởi tính chất của thuật toán, yếu tố thứ hai lại được xác định bởi tính chất của máy tính. Sự tách biệt này cho phép chúng ta so sánh các thuật toán một cách độc lập với sự cài đặt cụ thể hay độc lập với một máy tính cụ thể.

Bước thứ ba trong quá trình phân tích thuật toán là sự phân tích về mặt toán học, với mục đích tìm ra các giá trị trung bình và trường hợp xấu nhất cho mỗi đại lượng cơ bản. Chúng ta sẽ không gặp khó khăn khi tìm một chặn trên cho thời gian chạy chương trình, vấn đề ở chỗ là phải tìm ra chận trên tốt nhất, tức là thời gian chạy chương trình khi gặp dữ liệu nhập của trường hợp xấu nhất. Trường hợp trung bình thông thường đòi hỏi một phân tích toán học tinh vi hơn trường hợp xấu nhất. Mỗi khi đã hoàn thành một quá trình phân tích thuật toán dựa vào các đại lượng cơ bản, nếu thời gian kết hợp với mỗi đại lượng được xác định rõ thì ta sẽ có các biểu thức để tính thời gian chạy.

Nói chung, tính năng của một thuật toán thường có thể được phân tích ở một mức độ vô cùng chính xác, chỉ bị giới hạn bởi tính năng không chắc chắn của máy tính hay bởi sự khó khăn trong việc xác định các tính chất toán học của một vài đại lượng trừu tượng. Tuy nhiên, thay vì phân tích một cách chi tiết chúng ta thường thích ước lượng để tránh sa vào chi tiết.

Cách đánh giá thời gian thực hiện giải thuật độc lập với máy tính và các yếu tố liên quan tới máy như vậy sẽ dẫn đến khái niệm về " cấp độ lớn của thời gian thực hiện giải thuật" hay nói cách khác là "*độ phức tạp tính toán của giải thuật*"

Nếu thời gian thực hiện một giải thuật là $T(n) = cn^2$ (c = const) thì ta nói độ phức tạp tính toán của giải thuật này có cấp là n^2.

Kí hiệu : $T(n) = O(n^2)$ (kí hiệu chữ O lớn).

Định nghĩa:

Một hàm f(n) được xác định là O(g(n)) hay f(n) = O(g(n)) và được gọi là có cấp g(n) nếu tồn tại các hằng số c và n_0 sao cho :

$f(n) \leq cg(n)$ khi $n \geq n_0$

nghĩa là f(n) bị chặn trên bởi một hằng số nhân với g(n), với mọi giá trị của n từ một điểm nào đó.

Sự phân lớp các thuật toán

Như đã được chú ý trong ở trên, hầu hết các thuật toán đều có một tham số chính là N, thông thường đó là số lượng các phần tử dữ liệu được xử lý mà ảnh hưởng rất nhiều tới thời gian chạy. Tham số N có thể là bậc của một đa thức, kích thước của một tập tin được sắp xếp hay tìm kiếm, số nút trong một đồ thị .v.v... Hầu hết tất cả các thuật toán trong giáo trình này có thời gian chạy tiệm cận tới một trong các hàm sau:

Hằng số: Hầu hết các chỉ thị của các chương trình đều được thực hiện một lần hay nhiều nhất chỉ một vài lần. Nếu tất cả các chỉ thị của cùng một chương trình có tính chất nầy thì chúng ta sẽ nói rằng thời gian chạy của nó là hằng số. Điều nầy hiển nhiên là hoàn cảnh phấn đấu để đạt được trong việc thiết kế thuật toán.

logN: Khi thời gian chạy của chương trình là logarit tức là thời gian chạy chương trình tiến chậm khi N lớn dần. Thời gian chạy thuộc loại nầy xuất hiện trong các chương trình mà giải một bài toán lớn bằng cách chuyển nó thành một bài toán nhỏ hơn, bằng cách cắt bỏ kích thước bớt một hằng số nào đó. Với mục đích của chúng ta, thời gian chạy có được xem như nhỏ hơn một hằng số "lớn". Cơ số của logarit làm thay đổi hằng số đó nhưng không nhiều: khi N là một ngàn thì logN là 3 nếu cơ số là 10, là 10 nếu cơ số là 2; khi N là một triệu, logN được nhân gấp đôi. Bất cứ khi nào N được nhân đôi, logN tăng lên thêm một hằng số, nhưng logN không bị nhân gấp đôi khi N tăng tới N^2.

N: Khi thời gian chạy của một chương trình là tuyến tính, nói chung đây trường hợp mà một số lượng nhỏ các xử lý được làm cho mỗi phần tử dữ liệu nhập. Khi N là một triệu thì thời gian chạy cũng cỡ như vậy. Khi N được nhân gấp đôi thì thời gian chạy cũng được nhân gấp đôi. Đây là tình huống tối ưu cho một thuật toán mà phải xử lý N dữ liệu nhập (hay sản sinh ra N dữ liệu xuất).

NlogN: Đây là thời gian chạy tăng dần lên cho các thuật toán mà giải một bài toán bằng cách tách nó thành các bài toán con nhỏ hơn, kế đến giải quyết chúng một cách độc lập và sau đó tổ hợp các lời giải. Bởi vì thiếu một tính từ tốt hơn (có lẻ là "tuyến tính logarit"?), chúng ta nói rằng thời gian chạy của thuật toán như thế là "NlogN". Khi N là một triệu, NlogN có lẽ khoảng hai mươi triệu. Khi N được nhân gấp đôi, thời gian chạy bị nhân lên nhiều hơn gấp đôi (nhưng không nhiều lắm).

N^2: Khi thời gian chạy của một thuật toán là bậc hai, trường hợp nầy chỉ có ý nghĩa thực tế cho các bài toán tương đối nhỏ. Thời gian bình phương thường tăng dần lên trong các thuật toán mà xử lý tất cả các cặp phần tử dữ liệu (có thể là hai vòng lặp lồng nhau). Khi N là một ngàn thì thời gian chạy là một triệu. Khi N được nhân đôi thì thời gian chạy tăng lên gấp bốn lần.

N^3:Tương tự, một thuật toán mà xử lý các bộ ba của các phần tử dữ liệu (có lẻ là ba vòng lặp lồng nhau) có thời gian chạy bậc ba và cũng chỉ có ý nghĩa thực tế trong các bài toán nhỏ. Khi N là một trăm thì thời gian chạy là một triệu. Khi N được nhân đôi, thời gian chạy tăng lên gấp tám lần.

2^N: Một số ít thuật toán có thời gian chạy lũy thừa lại thích hợp trong một số trường hợp thực tế, mặc dù các thuật toán như thế là "sự ép buộc thô bạo" để giải các bài toán. Khi N là hai mươi thì thời gian chạy là một triệu. Khi N gấp đôi thì thời gian chạy được nâng lên lũy thừa hai!

Thời gian chạy của một chương trình cụ thể đôi khi là một hệ số hằng nhân với các số hạng nói trên ("số hạng dẫn đầu") cộng thêm một số hạng nhỏ hơn. Giá trị của hệ số hằng và các số hạng phụ thuộc vào kết quả của sự phân tích và các chi tiết cài đặt. Hệ số của số hạng dẫn đầu liên quan tới số chỉ thị bên trong vòng lặp: ở một tầng tùy ý của thiết kê thuật toán thì phải cẩn thận giới hạn số chỉ thị như thế. Với N lớn thì các số hạng dẫn đầu đóng vai trò chủ chốt; với N nhỏ thì các số hạng cùng đóng góp vào và sự so sánh các thuật toán sẽ khó khăn hơn. Trong hầu hết các trường hợp, chúng ta sẽ gặp các chương trình có thời gian chạy là "tuyến tính", "NlogN", "bậc ba", ... với hiểu ngầm là các phân tích hay nghiên cứu thực tế phải được làm trong trường hợp mà tính hiệu quả là rất quan trọng.

Sau đây là bảng giá trị của một số hàm đó:

Log_2n	n	$nlog_2n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1026	32768	2.147.483.648

CÁC QUY TẮC XÁC ĐỊNH ĐỘ PHỨC TẠP GIẢI THUẬT

+ *Qui tắc cộng:* Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P1 và P2 mà :

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

thì thời gian thực hiện P1 rồi P2 tiếp theo sẽ là :

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

Ví dụ : Trong một chương trình có 3 bước thực hiện mà thời gian thực hiện từng bước lần lượt là $O(n^2)$, $O(n^3)$ và $O(nlog_2n)$ thì thời gian thực hiện 2 bước đầu là $O(\max(n^2, n^3)) = O(n^3)$. Thời gian thực hiện chương trình sẽ là $O(\max(n^3, nlog_2n)) = O(n^3)$

Chú ý : Nếu $g(n) \leq f(n)$ với mọi $n \geq n_0$ thì $O(f(n)+g(n))$ cũng là $O(f(n))$.

VD : $O(n^4 + n^2) = O(n^4)$; $O(n + \log_2 n) = O(n)$.

+ *Qui tắc nhân*: Nếu $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của 2 đoạn chương trình P1 và P2 trong đó ($T_1(n) = O(f(n))$; $T_2 = (O(g(n)))$; thì thời gian thực hiện P1 và P2 lồng nhau là:

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

Ví dụ: Câu lệnh For(i = 1 ,i < n , i++) x = x + 1;

có thời gian thực hiện O(n.1) = O(n)

Câu lệnh For(i = 1, i <= n , i++)

For(j = 1 , j <= n , j++)

x = x + 1;

Có thời gian thực hiện được đánh giá là $O(n.n) = O(n^2)$

Chú ý : O(cf(n)) = O(F(n)) với c là hằng số

VD: $O(n^2/2) = O(n^2)$

Ví dụ 3.1 : Tìm độ phức tap của giải thuật tính giá trị e^x theo công thức gần đúng sau:

$e^x = 1 + x/1! + x^2/2! + \ldots + x^n/n!$ với x và n cho trước.

```
Void EXP1
{
    1. Scanf(x); S = 1;
    2. For (int i=1, i <= n, i++ )
{
p = 1;
For (int j=1, j <= i, j++ )
p = p * x/j;
```

```
S = S + p;

}

}
```

Ta có thể coi phép toán tích cực ở đây là phép : p = p * x/j;

Và nó được thực hiện : 1 + 2 + . . . + n = n(n-1)/2 lần

⇒ Thời gian thực hiện giải thuật là : $T(n) = O(n^2)$.

Cũng trường hợp tính e^x ta có thể biểu diễn giải thuật theo cách khác (dựa vào số hạng trước để tính số hạng sau):

$x^2/2! = x/1! * x/2; . . .; x^n/n! = x^{n-1} /(n - 1)! * x/n;$

Giải thuật có thể được viết :

```
Void EXP2

{

    1. Scanf(x); S = 1; p = 1;
    2. For (int i=1, i <= n, i++ )

{

p = p * x/i;

S = S + p;

}

}
```

Trường hợp này thì thời gian thực hiện giải thuật lại là : T(n) = O(n) vì phép p * x/i chỉ được thực hiện n lần.

Chú ý: Trong thực tế có những trường hợp thời gian thực hiện giải thuật không chỉ phụ thuộc vào kích thước của dữ liệu, mà còn phụ thuộc vào chính tình trạng của dữ liệu đó nữa.

Ví dụ 3.2: Cho một vec tơ V có n phần tử, xác định thời gian thực hiện giải thuật tìm trong V một phần tử có giá trị bằng X cho trước.

```
Void Tim;

{

1. Found = false; //Biến logic báo hiệu ngừng khi tìm thấy

i = 1;

2. While (i <= n and not Found )

if (V[i] = = X )

{

Found = true; k = i;

Cout << k;

}

Else

i = i + 1;

}
```

Ta coi phép toán tích cực ở đây là phép so sánh V[i] với X. Có thể thấy số lần phép toán tích cực này thực hiện phụ thuộc vào chỉ số i mà V[i] = X. Trường hợp thuận lợi nhất xảy ra khi X bằng V[1] một lần thực hiện.

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

Vậy : $T_{tốt} = O(1)$

$T_{xấu} = O(n)$

Lúc này ta phải xác định thời gian trung bình thực hiện giải thuật. Giả thiết khả năng xác suất X rơi đồng đều với mọi phần tử của V. Ta có thể xét như sau:

Gọi q là xác suất để X rơi vào một phần tử nào đó của V thì xác suất để X rơi vào phần tử V[i] là : $p_i^* = q/n$

Còn xác suất để X không rơi vào phần tử nào sẽ là 1 - q. Khi đó ta sẽ xác định được thời gian thực hiện trung bình:

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

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

$$= \sum_{i=1}^{n} q/n * n(n + 1)/2 + (1 - q)n$$

$$= q(n + 1)/2 + (1 - q)n$$

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

Nếu q = 1/2 (khả năng tìm thấy và không tìm thấy xác suất bằng nhau) thì T_{tb} = (n + 1)/4 + n/2 = (3n + 1)/4

Cả hai trường hợp đều dẫn đến cùng một kết quả là T(n) = O(n).

Mảng và dánh sách

MẢNG

Mảng một chiều, mảng nhiều chiều

Khái niệm

Mảng là một tập hợp có thứ tự gồm một số cố định các phần tử. Không có phép bổ sung phần tử hoặc loại bỏ phần tử được thực hiện.

Các phép toán thao tác trên mảng bao gồm : phép tạo lập (create) mảng, phép tìm kiếm (retrieve) một phần tử của mảng, phép lưu trữ (store) một phần tử của mảng.

Các phần tử của mảng được đặc trưng bởi chỉ số (index) thể hiện thứ tự của các phần tử đó trong mảng.

Mảng bao gồm các loại:

+ Mảng một chiều: Mảng mà mỗi phần tử a_i của nó ứng với một chỉ số i.

Ví dụ : Véc tơ a[i] trong đó 0 = 1 . . n cho biết véc tơ là mảng một chiều gồm có n phần tử.

Khai báo : kiểu phần tử A[0...n]

A: Tên biến mảng; Kiểu phần tử: Chỉ kiểu của các phần tử mảng (integer, real, . . .)

+ Mảng hai chiều: Là mảng mà mỗi phần tử a_{ij} của nó ứng với hai chỉ số i và j

Ví dụ : Ma trận A[i],[j] là mảng 2 chiều có i là chỉ số hàng của ma trận và j là chỉ số cột của ma trận.

i = 0 . . n; j = 0 . . m

n: Số hàng của ma trận; m : số cột của ma trận.

Khai báo : kiểu phần tử A[n][m];

+ Mảng n chiều : Tương tự như mảng 2 chiều.

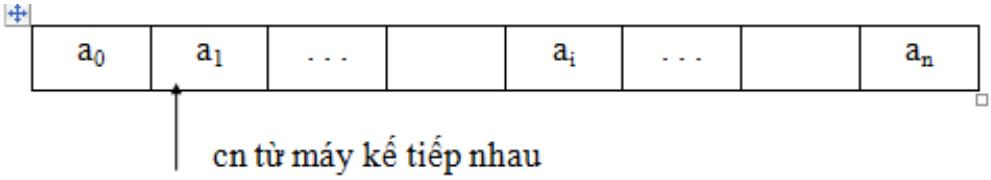
Cấu trúc lưu trữ của mảng.

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

Thông thường thì một số từ máy sẽ được dành ra để lưu trữ các phần tử của mảng. Cách lưu trữ này được gọi là cách *lưu trữ kế tiếp* (sequential storage allocation).

Trường hợp một mảng một chiều hay véc tơ có n phần tử của nó có thể lưu trữ được trong một từ máy thì cần phải dành cho nó n từ máy kế tiếp nhau. Do kích thước của véc tơ đã được xác định nên không gian nhớ dành ra cũng được ấn định trước.

Véc tơ A có n phần tử, nếu mỗi phần tử a_i ($0 \le i \le n$) chiếm c từ máy thì nó sẽ được lưu trữ trong cn từ máy kế tiếp như hình vẽ:



L_0 – Địa chỉ của phần tử a_0

Địa chỉ của a_i được tính bởi công thức:

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

trong đó :

L_0 được gọi là địa chỉ gốc - đó là địa chỉ từ máy đầu tiên trong miền nhớ kế tiếp dành để lưu trữ véc tơ (gọi là véc tơ lưu trữ).

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

Đối với mảng nhiều chiều việc lưu trữ cũng tương tự như vậy nghĩa là vẫn sử dụng một véc tơ lưu trữ kế tiếp như trên.

a_{01}	a_{11}	. . .		a_{ij}	. . .		a_{nm}

Giả sử mỗi phần tử trong ma trận n hàng m cột (mảng nhiều chiều) chiếm một từ máy thì địa chỉ của a_{ij} sẽ được tính bởi công thức tổng quát như sau:

$Loc(a_{ij}) = L_0 + j * n + i$ { theo thứ tự ưu tiên cột (column major order }

Cũng với ma trận n hàng, m cột cách lưu trữ theo thứ tự ưu tiên hàng (row major order) thì công thức tính địa chỉ sẽ là:

$Loc(a_{ij}) = L_0 + i * m + j$

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

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

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

Ví dụ : Xét mảng ba chiều B có các phần tử b_{ijk} với $1 \le i \le 2$;

$1 \le j \le 3$; $1 \le k \le 4$; được lưu trữ theo thứ tự ưu tiên hàng thì các phần tử của nó sẽ được sắp đặt kế tiếp như sau:

$b_{111}, b_{112}, b_{113}, b_{114}, b_{121}, b_{122}, b_{123}, b_{124}, b_{131}, b_{132}, b_{133}, b_{134}, b_{211}, b_{212}, b_{213}, b_{214}, b_{221}, b_{222}, b_{223}, b_{224}, b_{231}, b_{232}, b_{233}, b_{234}$.

Công thức tính địa chỉ sẽ là :

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

VD $Loc(b_{223}) = L_0 + 22$.

Xét trường hợp tổng quát với mảng A n chiều mà các phần tử là :

$A[s_1, s_2, . . ., s_n]$ trong đó $b_i \le s_i \le u_i$ (i = 1, 2, . . ., n), ứng với thứ tự ưu tiên hàng ta có:

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

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

đặc biệt $p_n = 1$.

Chú ý :

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

Ví dụ : Xét bài toán tính đa thức của x,y chẳng hạn cộng hai đa thức sau:

$(3x^2 - xy + y^2 + 2y - x)$

$+ (x^2 + 4xy - y^2 + 2x)$

$= (4x^2 + 3xy + 2y + x)$

Ta biết khi thực hiện cộng 2 đa thức ta phải phân biệt được từng số hạng, phân biệt được các biến, hệ số và số mũ.

Để biểu diễn được một đa thức với 2 biến x,y ta có thể dùng ma trận: hệ số của số hạng x^iy^j sẽ được lưu trữ ở phần tử có hàng i cột j của ma trận. Nếu ta hạn chế kích thước của ma trận là n × n thì số mũ cao nhất của x,y chỉ xử lý được với đa thức bậc n-1 thôi.

VD : Với $x^2 + 4xy - y^2 + 2x$ thì ta sẽ sử dụng ma trận 5×5 biểu diễn nó sẽ có dạng:

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

Với cách biểu diễn kiểu này thì việc thực hiện phép cộng hai đa thức chỉ là cộng ma trận mà thôi. Nhưng nó có một số hạn chế : số mũ của đa thức bị hạn chế bởi kích thước của ma trận do đó lớp các đa thức được xử lý bị giới hạn trong một phạm vi hẹp. Mặt khác ma trận biểu diễn có nhiều phần tử bằng 0, dẫn đến sự lãng phí bộ nhớ.

Cấu trúc lưu trữ mảng trên một số ngôn ngữ lập trình

Lưu trữ mảng trong ngôn ngữ lập trình C

Hay như để lưu trữ các từ khóa của ngôn ngữ lập trình C, ta cũng dùng đến một mảng để lưu trữ chúng.

Ví dụ 1: Viết chương trình cho phép nhập 2 ma trận a, b có m dòng n cột, thực hiện phép toán cộng hai ma trận a,b và in ma trận kết quả lên màn hình.

Trong ví dụ này, ta sẽ sử dụng hàm để làm ngắn gọn hơn chương trình của ta. Ta sẽ viết các hàm: nhập 1 ma trận từ bàn phím, hiển thị ma trận lên màn hình, cộng 2 ma trận.

```
#include<conio.h>

#include<stdio.h>

void Nhap(int a[][10],int M,int N)

{

int i,j;

for(i=0;i<M;i++)

for(j=0; j<N; j++){

printf("Phan tu o dong %d cot %d: ",i,j);

scanf("%d",&a[i][j]);

}

}

void InMaTran(int a[][10], int M, int N)

{

int i,j;

for(i=0;i<M;i++){

for(j=0; j< N; j++)
```

```
printf("%d ",a[i][j]);
printf("\n");
}
}
/* Cong 2 ma tran A & B ket qua la ma tran C*/
void CongMaTran(int a[][10],int b[][10],int M,int N,int c[][10]){
int i,j;
for(i=0;i<M;i++)
for(j=0; j<N; j++)
c[i][j]=a[i][j]+b[i][j];
}
int main()
{
int a[10][10], b[10][10], M, N;
int c[10][10];/* Ma tran tong*/
printf("So dong M= "); scanf("%d",&M);
printf("So cot M= "); scanf("%d",&N);
printf("Nhap ma tran A\n");
Nhap(a,M,N);
printf("Nhap ma tran B\n");
Nhap(b,M,N);
printf("Ma tran A: \n");
```

```
InMaTran(a,M,N);

printf("Ma tran B: \n");

InMaTran(b,M,N);

CongMaTran(a,b,M,N,c);

printf("Ma tran tong C:\n");

InMaTran(c,M,N);

getch();

return 0;

}
```

Lưu trữ mảng trong ngôn ngữ lập trình C#

Array là một cấu trúc dữ liệu cấu tạo bởi một số biến được gọi là những phần tử mảng. Tất cả các phần tử này đều thuộc một kiểu dữ liệu. Bạn có thể truy xuất phần tử thông qua chỉ số (index). Chỉ số bắt đầu bằng zero.

Có nhiều loại mảng (array): mảng một chiều, mảng nhiều chiều.

Cú pháp :

type[ ] array-name;

thí dụ: int[] myIntegers; // mảng kiểu số nguyên string[] myString ; // mảng kiểu chuỗi chữ Bạn khai báo mảng có chiều dài xác định với từ khoá new như sau: // Create a new array of 32 ints int[] myIntegers = new int[32]; integers[0] = 35;// phần tử đầu tiên có giá trị 35 integers[31] = 432;// phần tử 32 có giá trị 432 Bạn cũng có thể khai báo như sau:

int[] integers; integers = new int[32]; string[] myArray = {"first element", "second element", "third element"};

Làm việc với mảng (Working with Arrays)

Ta có thể tìm được chiều dài của mảng sau nhờ vào thuộc tính Length thí dụ sau : int arrayLength = integers.Length

Nếu các thành phần của mảng là kiểu định nghĩa trước (predefined types), ta có thể sắp xếp tăng dần vào phương thức gọi là static Array.Sort() method: Array.Sort(myArray);

Cuối cùng chúng ta có thể đảo ngược mảng đã có nhờ vào the static Reverse() method: Array.Reverse(myArray); string[] artists = {"Leonardo", "Monet", "Van Gogh", "Klee"}; Array.Sort(artists); Array.Reverse(artists); foreach (string name in artists) { Console.WriteLine(name); }

Mảng nhiều chiều (Multidimensional Arrays in C#)

Cú pháp :

type[,] array-name;

Thí dụ muốn khai báo một mảng hai chiều gồm hai hàng ba cột với phần tử kiểu nguyên :

int[,] myRectArray = new int[2,3];

Bạn có thể khởi gán mảng xem các ví dụ sau về mảng nhiều chiều:

```
int[,] myRectArray = new int[,]{ {1,2},{3,4},{5,6},{7,8}}; // mảng 4 hàng 2 cột
string[,] beatleName = { {"Lennon","John"},
                         {"McCartney","Paul"},
                         {"Harrison","George"},
                         {"Starkey","Richard"} };
```

chúng ta có thể sử dụng :

```
string[,,] my3DArray;
double [, ] matrix = new double[10, 10];
for (int i = 0; i < 10; i++)
{
    for (int j=0; j < 10; j++)
    matrix[i, j] = 4;
}
```

Mảng jagged

Một loại thứ 2 của mảng nhiều chiều trong C# là Jagged array. Jagged là một mảng mà mỗi phần tử là một mảng với kích thước khác nhau. Những mảng con này phải được khai báo từng mảng con một.

Thí dụ sau đây khai báo một mảng jagged hai chiều nghĩa là hai cặp [], gồm 3 hàng mỗi hàng là một mảng một chiều:

```
int[][] a = new int[3][];
a[0] = new int[4];
a[1] = new int[3];
a[2] = new int[1];
```

Khi dùng mảng jagged ta nên sử dụng phương thức GetLength() để xác định số lượng cột của mảng. Thí dụ sau nói lên điều này:

```
using System;

    namespace Wrox.ProCSharp.Basics
    {
        class MainEntryPoint
        {
            static void Main()
            {
                // Declare a two-dimension jagged array of authors' names
                  string[][] novelists = new string[3][];
                  novelists[0] = new string[] {
                  "Fyodor", "Mikhailovich", "Dostoyevsky"};
                  novelists[1] = new string[] {
                  "James", "Augustine", "Aloysius", "Joyce"};
                   novelists[2] = new string[] {
                   "Miguel", "de Cervantes", "Saavedra"};
```

```
        // Loop through each novelist in the array
          int i;
          for (i = 0; i < novelists.GetLength(0); i++)
          {
            // Loop through each name for the novelist
              int j;
              for (j = 0; j < novelists[i].GetLength(0); j++)
              {
                  // Display current part of name
                    Console.Write(novelists[i][j] + " ");
               }
            // Start a new line for the next novelist
              Console.Write("\n");
           }
}
      }
  }
```

Kết quả chương trình sau khi chạy:

```
csc AuthorNames.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.
AuthorNames
Fyodor Mikhailovich Dostoyevsky
James Augustine Aloysius Joyce
Miguel de Cervantes Saavedra
```

DANH SÁCH

Khái niệm danh sách tuyến tính

Danh sách là một tập hợp có thứ tự nhưng bao gồm một số biến động các phần tử (x_1, x_2, . . ., x_n)

nếu n = 0 ta có một danh sách rỗng.

Một danh sách mà quan hệ lân cận được hiển thị gọi là *danh sách tuyến tính* (linear list).

VD: Véc tơ chính là một trường hợp đặc biệt của danh sách tuyến tính xét tại một thời điểm nào đấy.

Danh sách tuyến tính là một danh sách hoặc *rỗng* (không có phần tử nào) hoặc có dạng (a_1, a_2, . . ., a_n) với a_i ($1 \le i \le n$) là các dữ liệu nguyên tử. Trong danh sách tuyến tính luôn tồn tại một phần tử đầu a_1, phần tử cuối a_n. Đối với mỗi phần tử a_i bất kỳ với $1 \le i \le n - 1$ thì có một phần tử a_{i+1} gọi là *phần tử sau* a_i, và với $2 \le i \le n$ thì có một phần tử $a_{i - 1}$ gọi là *phần tử trước* a_i. a_i được gọi là phần tử thứ i của danh sách tuyến tính, n được gọi là độ dài hoặc kích thước của danh sách.

Mỗi phần tử trong danh sách thường là một bản ghi (gồm một hoặc nhiều trường (fields)) đó là phần thông tin nhỏ nhất có thể tham khảo. VD: Danh sách sinh viên trong một lớp là một danh sách tuyến tính mà mỗi phần tử ứng với một sinh viên, nó bao gồm các trường:

Mã SV (STT), Họ và tên, Ngày sinh, Quê quán, . . .

Các phép toán thao tác trên danh sách :

+ Phép bổ sung một phần tử vào trong danh sách (Insert) .

+ Phép loại bỏ một phần tử trong danh sách (Delete).

+ Phép ghép nối 2 hoặc nhiều danh sách.

+ Phép tách một danh sách thành nhiều danh sách.

+ Phép sao chép một danh sách.

+ Phép cập nhật (update) danh sách.

+ Phép sắp xếp các phần tử trong danh sách theo thứ tự ấn định.

+ Phép tìm kiếm một phần tử trong danh sách theo giá trị ấn định của một trường nào đó.

Trong đó phép bổ sung và phép loại bỏ là hai phép toán thường xuyên được sử dụng trong danh sách.

Tệp cũng là một trường hợp của danh sách nó có kích thước lớn và thường được lưu trữ ở bộ nhớ ngoài. Còn danh sách nói chung thường được xử lý ở bộ nhớ trong.

Bộ nhớ trong được hình dung như một dãy các từ máy(words) có thứ tự, mỗi từ máy ứng với một địa chỉ. Mỗi từ máy chứa từ 8 ? 64 bits, việc tham khảo đến nội dung của nó thông qua địa chỉ.

+ *Cách xác định địa chỉ của một phần tử trong danh sách*: Có 2 cách xác định địa chỉ:

Cách 1: Dựa vào đặc tả của dữ liệu cần tìm . Địa chỉ này gọi là địa chỉ tính được (hay địa chỉ trực tiếp).

VD: Xác định địa chỉ của các phần tử trong véc tơ, ma trận thông qua các chỉ số.

Cách 2: Lưu trữ các địa chỉ cần thiết ở trong bộ nhớ, khi cần xác định sẽ lấy ở đó ra. Địa chỉ này được gọi là con trỏ (pointer) hay mối nối (link).

Lưu trữ kế tiếp của danh sách tuyến tính

Lưu trữ kế tiếp là phương pháp lưu trữ sử dụng mảng một chiều làm cấu trúc lưu trữ của danh sách tuyến tính nghĩa là có thể dùng một véc tơ lưu trữ V_i với $1 \leq i \leq n$ để lưu trữ một danh sách tuyến tính $(a_1, a_2, \ldots, a_n)$ trong đó phần tử a_i được chứa ở V_i.

Ưu điểm : Tốc độ truy nhập nhanh, dễ thao tác trong việc bổ sung, loại bỏ và tìm kiếm phần tử trong danh sách.

Nhược điểm: Do số phần tử trong danh sách tuyến tính thường biến động (kích thước n thay đổi) dẫn đến hiện tượng lãng phí bộ nhớ. Mặt khác nếu dự trữ đủ rồi thị việc bổ sung hay loại bỏ một phần tử trong danh sách mà không phải là phần tử cuối sẽ đòi hỏi phải dồn hoặc dãn danh sách (nghĩa là phải dịch chuyển một số phần tử để lấy chỗ bổ sung hay tiến lên để lấp chỗ phần tử bị loại bỏ) sẽ tốn nhiều thời gian

Nhu cầu xây dựng cấu trúc dữ liệu động

Với các cấu trúc dữ liệu được xây dựng từ các kiểu cơ sở như: kiểu thực, kiểu nguyên, kiểu ký tự ... hoặc từ các cấu trúc đơn giản như mẩu tin, tập hợp, mảng ... lập trình viên có thể giải quyết hầu hết các bài toán đặt ra. Các đối tượng dữ liệu được xác định thuộc những kiểu dữ liệu này có đặc điểm chung là không thay đổi được kích thước, cấu trúc

trong quá trình sống, do vậy thường cứng ngắt, gò bó khiến đôi khi khó diễn tả được thực tế vốn sinh động, phong phú. Các kiểu dữ liệu kể trên được gọi là các kiểu dữ liệu tĩnh.

Ví dụ :

1. Trong thực tế, một số đối tượng có thể được định nghĩa đệ qui, ví dụ để mô tả đối tượng "con người" cần thể hiện các thông tin tối thiểu như :

- Họ tên
- Số CMND
- Thông tin về cha, mẹ

Để biểu diễn một đối tượng có nhiều thành phần thông tin như trên có thể sử dụng kiểu bản ghi. Tuy nhiên, cần lưu ý cha, mẹ của một người cũng là các đối tượng kiểu NGƯỜI, do vậy về nguyên tắc cần phải có định nghĩa như sau:

```
typedef struct NGUOI{

char Hoten[30];

int So_CMND ;

NGUOI Cha,Me;

};
```

Nhưng với khai báo trên, các ngôn ngữ lập trình gặp khó khăn trong việc cài đặt không vượt qua được như xác định kích thước của đối tượng kiểu NGUOI.

2. Một số đối tượng dữ liệu trong chu kỳ sống của nó có thể thay đổi về cấu trúc, độ lớn, như danh sách các học viên trong một lớp học có thể tăng thêm, giảm đi ... Khi đó nếu cố tình dùng những cấu trúc dữ liệu tĩnh đã biết như mảng để biểu diễn những đối tượng đó lập trình viên phải sử dụng những thao tác phức tạp, kém tự nhiên khiến chương trình trở nên khó đọc, do đó khó bảo trì và nhất là khó có thể sử dụng bộ nhớ một cách có hiệu quả.

3. Một lý do nữa làm cho các kiểu dữ liệu tĩnh không thể đáp ứng được nhu cầu của thực tế là tổng kích thước vùng nhớ dành cho tất cả các biến tĩnh có giới hạn. Khi có nhu cầu dùng nhiều bộ nhớ hơn ta phải sử dụng các *cấu trúc dữ liệu động.*

4. Cuối cùng, do bản chất của các dữ liệu tĩnh, chúng sẽ chiếm vùng nhớ đã dành cho chúng suốt quá trình hoạt động của chương trình. Tuy nhiên, trong thực tế, có thể xảy ra trường hợp một dữ liệu nào đó chỉ tồn tại nhất thời hay không thường xuyên trong quá

trình hoạt động của chương trình. Vì vậy việc dùng các CTDL tĩnh sẽ không cho phép sử dụng hiệu quả bộ nhớ.

Do vậy, nhằm đáp ứng nhu cầu thể hiện sát thực bản chất của dữ liệu cũng như xây dựng các thao tác hiệu quả trên dữ liệu, cần phải tìm cách tổ chức kết hợp dữ liệu với những hình thức mới linh động hơn, có thể thay đổi kích thước, cấu trúc trong suốt thời gian sống. Các hình thức tổ chức dữ liệu như vậy được gọi là *cấu trúc dữ liệu động*. Bài sau sẽ giới thiệu về các cấu trúc dữ liệu động và tập trung khảo sát cấu trúc đơn giản nhất thuộc loại này là danh sách liên kết.

Danh sách nối đơn (Singlely Linked List)

KHÁI NIỆM DANH SÁCH NỐI ĐƠN

Mỗi phần tử của danh sách đơn là một cấu trúc chứa 2 thông tin :

- *Thành phần dữ liệu:* lưu trữ các thông tin về bản thân phần tử .

- *Thành phần mối liờn kết:* lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử cuối danh s ách.

Ta có định nghĩa tổng quát

typedef struct tagNode

{ Data Info; // Data là kiểu đó định nghĩa trước

struct tagNode* pNext; // con trỏ chỉ đến cấu trúc node

}NODE;

Ví dụ: Định nghĩa danh sách đơn lưu trữ hồ sơ sinh viên:

typedef struct SinhVien

{ char Ten[30];

int MaSV;

}SV;

typedef struct SinhvienNode

{ SV Info;

struct SinhvienNode* pNext;

}SVNode;

- Một phần tử trong danh sách đơn là một biến động sẽ được yêu cầu cấp phát khi cần. Và danh sách đơn chính là sự liên kết các biến động này với nhau do vậy đạt được sự linh động khi thay đổi số lượng các phần tử

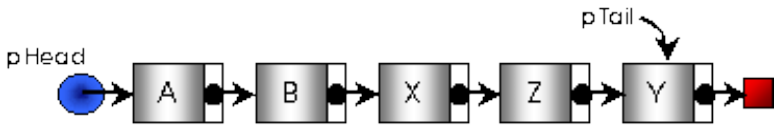
- Nếu biết được địa chỉ của phần tử đầu tiên trong danh sách đơn thỡ cú thể dựa vào thụng tin pNext của nú để truy xuất đến phần tử thứ 2 trong xâu, và lại dựa vào thông tin Next của phần tử thứ 2 để truy xuất đến phần tử thứ 3...Nghĩa là để quản lý một xâu đơn chỉ cần biết địa chỉ phần tử đầu xâu. Thường một con trỏ Head sẽ được dùng để lưu trữ địa chỉ phần tử đầu xâu, ta gọi Head là đầu xâu. Ta có khai báo:

NODE *pHead;

- Tuy về nguyên tắc chỉ cần quản lý xâu thông qua đầu xâu pHead, nhưng thực tế có nhiều trường hợp cần làm việc với phần tử cuối xâu, khi đó mỗi lần muốn xác định phần tử cuối xâu lại phải duyệt từ đầu xâu. Để tiện lợi, cú thể sử dụng thờm một con trỏ pTail giữ địa chỉ phần tử cuối xâu. Khai báo pTail như sau :

NODE *pTail;

Lúc này có xâu đơn:



MỘT SỐ PHÉP TOÁN TRÊN DANH SÁCH NỐI ĐƠN

Giả sử có các định nghĩa:

typedef struct tagNode

{

Data Info;

struct tagNode* pNext;

}NODE;

typedef struct tagList

{

NODE* pHead;

NODE* pTail;

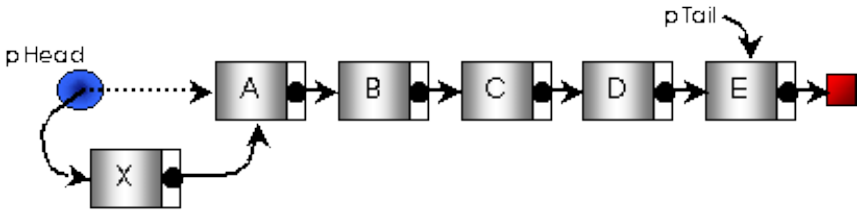
}LIST;

NODE *new_ele // giữ địa chỉ của một phần tử mới được tạo

Data x; // lưu thông tin về một phần tử sẽ được tạo

Và đã xây dựng thủ tục GetNode để tạo ra một phần tử cho danh sách với thông tin chứa trong x:

Chèn một phần tử vào danh sách:



Có 3 loại thao tác chèn new_ele vào xâu:

Cách 1: Chèn vào đầu danh sách

- Thuật toán :

Bắt đầu:

Nếu Danh sách rỗng Thì

B11 : Head = new_elelment;

B12 : Tail = Head;

Ngược lại

B21 : new_ele ->pNext = Head;

B22 : Head = new_ele ;

- Cài đặt :

void AddFirst(LIST &l, NODE* new_ele)

{

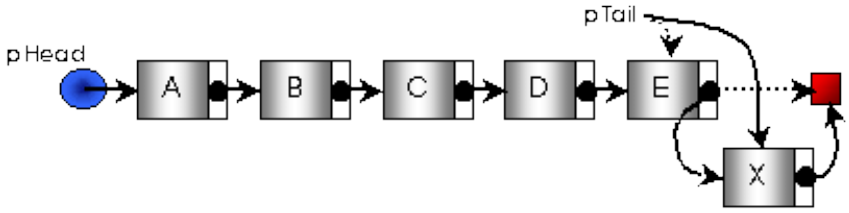
if (l.pHead==NULL) //Xâu rỗng

```
{
l.pHead = new_ele; l.pTail = l.pHead;
}
else
{
new_ele->pNext = l.pHead;
l.pHead = new_ele;
}
}
NODE* InsertHead(LIST &l, Data x)
{
NODE* new_ele = GetNode(x);

if (new_ele ==NULL) return NULL;
if (l.pHead==NULL)
{
l.pHead = new_ele; l.pTail = l.pHead;
}
else
{
new_ele->pNext = l.pHead;
l.pHead = new_ele;
```

}

return new_ele;

}



Cách 2: Chèn vào cuối danh sách

- Thuật toán :

Bắt đầu :

Nếu Danh sách rỗng Thì

B11 : Head = new_elelment;

B12 : Tail = Head;

Ngược lại

B21 : Tail ->pNext = new_ele;

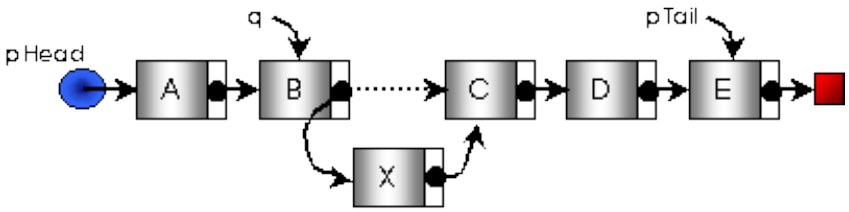
B22 : Tail = new_ele ;

- Cài đặt :

```
void AddTail(LIST &l, NODE *new_ele)
{
if (l.pHead==NULL)
{
l.pHead = new_ele; l.pTail = l.pHead;
}
```

```
else
{
l.pTail->Next = new_ele;
l.pTail = new_ele;
}
}

NODE* InsertTail(LIST &l, Data x)
{
NODE* new_ele = GetNode(x);

if (new_ele ==NULL) return NULL;
if (l.pHead==NULL)
{
l.pHead = new_ele; l.pTail = l.pHead;
}
else
{
l.pTail->Next = new_ele;
l.pTail = new_ele;
}
return new_ele;
```

}



Cách 3 : Chèn vào danh sách sau một phần tử q

- Thuật toán :

Bắt đầu :

Nếu (q != NULL) thì

B1 : new_ele -> pNext = q->pNext;

B2 : q->pNext = new_ele ;

- Cài đặt :

```
void AddAfter(LIST &l,NODE *q, NODE* new_ele)
{
if ( q!=NULL)
{
new_ele->pNext = q->pNext;
q->pNext = new_ele;
if(q == l.pTail)
l.pTail = new_ele;
}
else //chèn vào đầu danh sách
AddFirst(l, new_ele);
```

```
}

void InsertAfter(LIST &l,NODE *q, Data x)

{

NODE* new_ele = GetNode(x);


if (new_ele ==NULL) return NULL;


if ( q!=NULL)

{

new_ele->pNext = q->pNext;

q->pNext = new_ele;

if(q == l.pTail)

l.pTail = new_ele;

}

else //chèn vào đầu danh sách

AddFirst(l, new_ele);

}
```

Tìm một phần tử trong danh sách đơn

- Thuật toán :

Xâu đơn đòi hỏi truy xuất tuần tự, do đó chỉ có thể áp dụng thuật toán tìm tuyến tính để xác định phần tử trong xâu có khoá k. Sử dụng một con trỏ phụ trợ p để lần lượt trỏ đến các phần tử trong xâu. Thuật toán được thể hiện như sau :

Bước 1:

p = Head; //Cho p trỏ đến phần tử đầu danh sách

Bước 2:

Trong khi (p != NULL) và (p->pNext != k) thực hiện:

B21 : p:=p->Next;// Cho p trỏ tới phần tử kế

Bước 3:

Nếu p != NULL thì p trỏ tới phần tử cần tìm

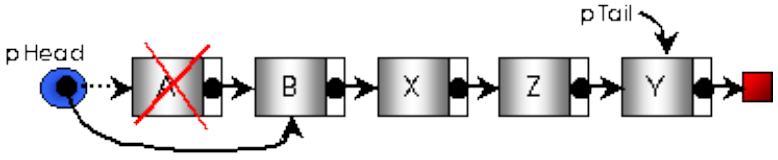
Ngược lại: không có phần tử cần tìm.

- Cài đặt :

```
NODE *Search(LIST l, Data k)
{ NODE *p;
p = l.pHead;
while((p!= NULL)&&(p->Info != x))
p = p->pNext;
return p;
}
```

Hủy một phần tử khỏi danh sách

Có 3 loại thao tác thông dụng hủy một phần tử ra khỏi xâu. Chúng ta sẽ lần lượt khảo sát chúng. Lưu ý là khi cấp phát bộ nhớ, chúng ta đã dùng hàm new. Vì vậy khi giải phóng bộ nhớ ta phải dùng hàm delete.



Hủy phần tử đầu xâu:

- Thuật toán :

Bắt đầu:

Nếu (Head != NULL) thì

B1: p = Head; // p là phần tử cần hủy

B2:

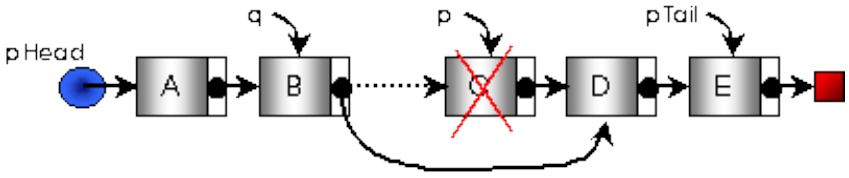
B21 : Head = Head →pNext; // tách p ra khỏi xâu

B22 : free(p); // Hủy biến động do p trỏ đến

B3: Nếu Head=NULL thì Tail = NULL; //Xâu rỗng

- Cài đặt :

```
Data RemoveHead(LIST &l)
{ NODE *p;
Data x = NULLDATA;

if ( l.pHead != NULL)
{
p = l.pHead; x = p→Info;
l.pHead = l.pHead→pNext;
delete p;
if(l.pHead == NULL) l.pTail = NULL;
}
return x;
}
```



Hủy một phần tử đứng sau phần tử q

- Thuật toán :

Bắt đầu:

Nếu (q!= NULL) thì

B1: p = q->Next; // p là phần tử cần hủy

B2: Nếu (p != NULL) thì // q không phải là cuối xâu

B21 : q->Next = p->Next; // tách p ra khỏi xâu

B22 : free(p); // Hủy biến động do p trỏ đến

- Cài đặt :

```
void RemoveAfter (LIST &l, NODE *q)
{ NODE *p;


if ( q != NULL)
{
p = q ->pNext ;
if ( p != NULL)
{
if(p == l.pTail) l.pTail = q;
q->pNext = p->pNext;
delete p;
```

}

}

else

RemoveHead(l);

}

Hủy 1 phần tử có khoá k

- Thuật toán :

Bước 1:

Tìm phần tử p có khóa k và phần tử q đứng trước nó

Bước 2:

Nếu (p!= NULL) thì // tìm thấy k

Hủy p ra khỏi xâu tương tự hủy phần tử sau q;

Ngược lại

Báo không có k;

- Cài đặt :

```
int RemoveNode(LIST &l, Data k)
{ NODE *p = l.pHead;
NODE *q = NULL;

while( p != NULL)
{
if(p->Info == k) break;
```

```
q = p; p = p->pNext;
}
if(p == NULL) return 0; //Không tìm thấy k
if(q != NULL)
{
if(p == l.pTail)
l.pTail = q;
q->pNext = p->pNext;
delete p;
}
else //p là phần tử đầu xâu
{
l.pHead = p->pNext;
if(l.pHead == NULL)
l.pTail = NULL;
}
return 1;
}
```

Duyệt danh sách

Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách như:

- Đếm các phần tử của danh sách,

- Tìm tất cả các phần tử thoả điều kiện,
- Huỷ toàn bộ danh sách (và giải phóng bộ nhớ)

Để duyệt danh sách (và xử lý từng phần tử) ta thực hiện các thao tác sau:

- Thuật toán :

Bước 1:

p = Head; //Cho p trỏ đến phần tử đầu danh sách

Bước 2:

Trong khi (Danh sách chưa hết) thực hiện

B21 : Xử lý phần tử p;

B22 : p:=p->pNext; // Cho p trỏ tới phần tử kế

- Cài đặt :

```
void ProcessList (LIST &l)
{ NODE *p;

p = l.pHead;
while (p!= NULL)
{
ProcessNode(p); // xử lý cụ thể tùy ứng dụng
p = p->pNext;
}
}
```

LƯU Ý :

Để huỷ toàn bộ danh sách, ta có một chút thay đổi trong thủ tục duyệt (xử lý) danh sách trên (ở đây, thao tác xử lý bao gồm hành động giải phóng một phần tử, do vậy phải cập nhật các liên kết liên quan) :

- Thuật toán :

Bước 1:

Trong khi (Danh sách chưa hết) thực hiện

B11:

p = Head;

Head:=Head ->pNext; // Cho p trỏ tới phần tử kế

B12:

Hủy p;

Bước 2:

Tail = NULL; //Bảo đảm tính nhất quán khi xâu rỗng

- Cài đặt :

```
void ReamoveList(LIST &l)
{ NODE *p;
while (l.pHead!= NULL)
{
p = l.pHead;
l.pHead = p->pNext;
delete p;
}
l.pTail = NULL;
```

```
}
```

Thực hành cài đặt danh sách nối đơn

THỰC HÀNH CÀI ĐẶT DANH SÁCH NỐI ĐƠN

Bài 1:

Cài đặt những thủ tục, hàm sau bằng mảng đối với danh sách liên kết đơn

First (L : List) : Position

Next (p : Position ; L : List) : Position

Previous (p : Position ; L : List) : Position

Retrieve (p : Position ; L : List)

PrintList (L : List)

Bài 2:

Cài đặt những thủ tục, hàm sau bằng con trỏ đối với danh sách liên kết đơn

First (L : List) : Position

Next (p : Position ; L : List) : Position

Previous (p : Position ; L : List) : Position

Retrieve (p : Position ; L : List)

PrintList (L : List)

Bài 3

Viết khai báo và các thủ tục cài đặt danh sách bằng mảng. Dùng các thủ tục này để viết:

1. Thủ tục nhận một dãy các số nguyên nhập từ bàn phím, lưu trữ nó trong danh sách theo thứ tự nhập vào.
2. Thủ tục 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.
3. Viết thủ tục in ra màn hình các phần tử trong danh sách theo thứ tự của nó trong danh sách.

Bài 4

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

Bài 5

Viết thủ tục sắp xếp một danh sách chứa các số nguyên, trong các trường hợp:

1. Danh sách được cài đặt bằng mảng (danh sách đặc).
2. Danh sách được cài đặt bằng con trỏ (danh sách liên kết).

Danh sách tuyến tính ngăn xếp (Stack)

ĐỊNH NGHĨA

Stack là một vật chứa (container) các đối tượng làm việc theo cơ chế LIFO (Last In First Out) nghĩa là việc thêm một đối tượng vào stack hoặc lấy một đối tượng ra khỏi stack được thực hiện theo cơ chế "Vào sau ra trước".

Các đối tượng có thể được thêm vào stack bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào sau cùng mới được phép lấy ra khỏi stack.

Thao tác thêm 1 đối tượng vào stack thường được gọi là "Push". Thao tác lấy 1 đối tượng ra khỏi stack gọi là "Pop".

Trong tin học, CTDL stack có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều sâu và quay lui, vét cạn, ứng dụng trong các bài toán tính toán biểu thức, .

Ta có thể định nghĩa CTDL stack như sau: stack là một CTDL trừu tượng (ADT) tuyến tính hỗ trợ 2 thao tác chính:

Push(o): Thêm đối tượng o vào đầu stack

Pop(): Lấy đối tượng ở đầu stack ra khỏi stack và trả về giá trị của nó. Nếu stack rỗng thì lỗi sẽ xảy ra.

Ngoài ra, stack cũng hỗ trợ một số thao tác khác:

isEmpty(): Kiểm tra xem stack có rỗng không.

Top(): Trả về giá trị của phần tử nằm ở đầu stack mà không hủy nó khỏi stack. Nếu stack rỗng thì lỗi sẽ xảy ra.

Các thao tác thêm, trích và huỷ một phần tử chỉ được thực hiện ở cùng một phía của Stack do đó hoạt động của Stack được thực hiện theo nguyên tắc LIFO (Last In First Out - vào sau ra trước).

Để biểu diễn Stack, ta có thể dùng mảng 1 chiều hoặc dùng danh sách liên kết.

CÀI ĐẶT STACK

Cài đặt Stack bằng mảng

Ta có thể tạo một stack bằng cách khai báo một mảng 1 chiều với kích thước tối đa là N (ví dụ, N có thể bằng 1000)

Như vậy stack có thể chứa tối đa N phần tử đánh số từ 0 đến N -1. Phần tử nằm ở đầu stack sẽ có chỉ số t (lúc đó trong stack đang chứa t+1 phần tử)

Để khai báo một stack, ta cần một mảng 1 chiều S, biến nguyên t cho biết chỉ số của đầu stack và hằng số N cho biết kích thước tối đa của stack.

Tạo stack S và quản lý đỉnh stack bằng biến t:

Data S [N];

int t;

Bổ sung một phần tử vào stack

****

Void PUSH (S, T, X)

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

if (T ≥ n)

{

Cout<<” Stack tràn”;

Getch();

};

2- {chuyển con trỏ}

T +=T;

3 - {Bổ sung phần tử mới X vào stack}

S[T] = X;

4- Return

- *Giải thuật loại bỏ một phần tử ra khỏi Stack :*

Giải thuật này tiến hành việc loại bỏ một phần tử ở đỉnh Stack đang trỏ bởi con trỏ T ra khỏi Stack. Phần tử bị loại bỏ sẽ được thu nhận và đưa ra. Giải thuật được viết theo dạng chơng trình con hàm như sau:

Data POP (S, T)

1 - {Xét xem Stack có Cạn (UnderFlow) không?(Cạn nghĩa là số phần tử trong Stack = 0) . Hiện tượng cạn xảy ra khi Stack đã rỗng, không còn phần tử nào để loại nữa. Lúc đó sẽ in ra thông báo Cạn và kết thúc}

if (T ≤ 0)

{

Cout << “ Stack Cạn”;

return;

}

2 - {Chuyển con trỏ}

T- = T;

3 - {Đa phần tử bị loại ra}

POP = S [T + 1];

4 - return.

Việc cài đặt stack thông qua mảng một chiều đơn giản và khá hiệu quả.

Tuy nhiên, hạn chế lớn nhất của phương án cài đặt này là giới hạn về kích thước của stack N. Giá trị của N có thể quá nhỏ so với nhu cầu thực tế hoặc quá lớn sẽ làm lãng phí bộ nhớ.

Cài đặt Stack bằng danh sách

Ta có thể tạo một stack bằng cách sử dụng một danh sách liên kết đơn (DSLK). Có thể nói, DSLK có những đặc tính rất phù hợp để dùng làm stack vì mọi thao tác trên stack đều diễn ra ở đầu stack.

Sau đây là các thao tác tương ứng cho list-stack:

- Tạo Stack S rỗng

LIST * S;

Lệnh S.pHead=l.pTail= NULL sẽ tạo ra một Stack S rỗng.

- Kiểm tra stack rỗng :

Lệnh S.pHead=l.pTail= NULL sẽ tạo ra một Stack S rỗng.

- Kiểm tra stack rỗng :

```
char IsEmpty(LIST &S)

{

if (S.pHead == NULL) // stack rỗng

return 1;

else return 0;

}
```

- Thêm một phần tử p vào stack S

```
void Push(LIST &S, Data x)
{
InsertHead(S, x);
}
```

- Trích huỷ phần tử ở đỉnh stack S

```
Data Pop(LIST &S)
{ Data x;
if(isEmpty(S)) return NULLDATA;
x = RemoveFirst(S);
return x;
}
```

- Xem thông tin của phần tử ở đỉnh stack S

```
Data Top(LIST &S)
{if(isEmpty(S)) return NULLDATA;
return l.Head->Info;
}
```

MỘT SỐ VÍ DỤ ỨNG DỤNG CỦA STACK

Cấu trúc Stack thích hợp lưu trữ các loại dữ liệu mà thứ tự truy xuất ngược với thứ tự lưu trữ, do vậy một số ứng dụng sau thường cần đến stack :

Trong trình biên dịch (thông dịch), khi thực hiện các thủ tục, Stack được sử dụng để lưu môi trường của các thủ tục.

Trong một số bài toán của lý thuyết đồ thị (như tìm đường đi), Stack cũng thường được sử dụng để lưu dữ liệu khi giải các bài toán này.

Ngoài ra, Stack cũng cũng được sử dụng trong trường hợp khử đệ qui đuôi.

- KÝ PHÁP NGHỊCH ĐẢO BA LAN PHƯƠNG PHÁP TÍNH GIÁ TRỊ BIỂU THỨC TOÁN HỌC

Khi lập trình, tính giá trị một biểu thức toán học là điều quá đỗi bình thường. Tuy nhiên, trong nhiều ứng dụng (như chương trình vẽ đồ thị hàm số chẳng hạn, trong đó chương trình cho phép người dùng nhập vào hàm số), ta cần phải tính giá trị của một biểu thức được nhập vào từ bàn phím dưới dạng một chuỗi. Với các biểu thức toán học đơn giản (như a+b) thì bạn có thể tự làm bằng các phương pháp tách chuỗi "thủ công". Nhưng để "giải quyết" các biểu thức có dấu ngoặc, ví dụ như (a+b)*c + (d+e)*f , thì các phương pháp tách chuỗi đơn giản đều không khả thi. Trong tình huống này, ta phải dùng đến Ký Pháp Nghịch Đảo Ba Lan (Reserve Polish Notation – RPN), một thuật toán "kinh điển" trong lĩnh vực trình biên dịch.

Để đơn giản cho việc minh họa, ta giả định rằng chuỗi biểu thức mà ta nhận được từ bàn phím chỉ bao gồm: các dấu mở ngoặc/đóng ngoặc; 4 toán tử cộng, trừ, nhân và chia (+, -, *, /); các toán hạng đều chỉ là các con số nguyên từ 0 đến 9; không có bất kỳ khoảng trắng nào giữa các ký tự.

Thế nào là ký pháp nghịch đảo Ba Lan?

Cách trình bày biểu thức theo cách thông thường tuy tự nhiên với con người nhưng lại khá "khó chịu" đối với máy tính vì nó không thể hiện một cách tường minh quá trình tính toán để đưa ra giá trị của biểu thức. Để đơn giản hóa quá trình tính toán này, ta phải biến đổi lại biểu thức thông thường về dạng hậu tố - postfix (cách gọi ngắn của thuật ngữ ký pháp nghịch đảo Ba Lan). Để phân biệt hai dạng biểu diễn biểu thức, ta gọi cách biểu diễn biểu thức theo cách thông thường là trung tố - infix (vì toán tử nằm ở giữa hai toán hạng).

Ký pháp nghịch đảo Ba Lan được phát minh vào khoảng giữa thập kỷ 1950 bởi Charles Hamblin - một triết học gia và khoa học gia máy tính người Úc - dựa theo công trình về ký pháp Ba Lan của nhà Toán học người Ba Lan Jan Łukasiewicz. Hamblin trình bày nghiên cứu của mình tại một hội nghị khoa học vào tháng 6 năm 1957 và chính thức công bố vào năm 1962.

Từ cái tên hậu tố các bạn cũng đoán ra phần nào là theo cách biểu diễn này, các toán tử sẽ được đặt sau các toán hạng. Cụ thể là biểu thức trung tố: 4+5 sẽ được biểu diễn thành 4 5 +.

Quá trình tính toán giá trị của biểu thức hậu tố khá tự nhiên đối với máy tính. Ý tưởng là đọc biểu thức từ trái sang phải, nếu gặp một toán hạng (con số hoặc biến) thì push toán hạng này vào ngăn xếp; nếu gặp toán tử, lấy hai toán hạng ra khỏi ngăn xếp (stack),

tính kết quả, đẩy kết quả trở lại ngăn xếp. Khi quá trình kết thúc thì con số cuối cùng còn lại trong ngăn xếp chính là giá trị của biểu thức đó.

Ví dụ: biểu thức trung tố :

5 + ((1 + 2) * 4) + 3

được biểu diễn lại dưới dạng hậu tố là (ta sẽ bàn về thuật toán chuyển đổi từ trung tố sang hậu tố sau):

5 1 2 + 4 * + 3 +

Quá trình tính toán sẽ diễn ra theo như bảng dưới đây:

Ký tự	Thao tác	Stack	Chuỗi hậu tố
3	Ghi 3 vào k.quả		3
+	Push +	+	
4	Ghi 4 vào k.quả		3 4
*	Push *	+ *	
2	Ghi 2 vào kquả		3 4 2
/	Lấy * ra khỏi stack, ghi vào k.quả, push /	+ /	3 4 2 *
(	Push (	+ / (	3 4 2 *
1	Ghi 1 vào k.quả	+ / (	3 4 2 * 1
-	Push -	+ / (-	3 4 2 * 1
5	Ghi 5 vào k.quả	+ / (-	3 4 2 * 1 5
)	Pop cho đến khi lấy được (, ghi các toán tử pop được ra k.quả	+ /	3 4 2 * 1 5 -
2	Ghi 2 ra k.quả	+ /	3 4 2 * 1 5 – 2
	Pop tất cả các toán tử ra khỏi ngăn xếp và ghi vào kết quả		3 4 2 * 1 5 – 2 / +

Dĩ nhiên là thuật toán được trỡnh bày ở đây là khá đơn giản và chưa ứng dụng được trong trường hợp biểu thức có các hàm như sin, cos,… hoặc có các biến. Tuy nhiên, việc

mở rộng thuật toán là hoàn toàn nằm trong khả năng của bạn nếu bạn đó hiểu cặn kẽ thuật toỏn cơ bản này.

+/ *Chuong trinh dinh tri mot bieu thuc postfix*

Chương trinh co cai dat stack de chua cac toan hang (operands), moi toan hang la mot ky so. Co 5 toan tu la cong (+), tru (-), nhan (*), chia (/) va luy thua ($)

Vi du: 23+ = 5.00

23* = 6.00

23+4*5/ = 4.00

23+3$ = 125.00

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <math.h>

#define TOIDA 80

#define TRUE 1

#define FALSE 0

// Khai bao stack chua cac toan hang

struct stack

{

int top;

double nodes[TOIDA];

};

// prototypes
```

```
int empty(struct stack *);
void push(struct stack *, double);
double pop(struct stack *);
double dinhtri(char[]);
int lakyso(char);
double tinh(int, double, double);
int empty(struct stack *ps)
{
if (ps->top == -1)
return(TRUE);
else
return(FALSE);
}
void push(struct stack *ps, double x)
{
if(ps->top == TOIDA-1)
printf("%s", "stack bi day");
else
ps->nodes[++(ps->top)] = x;
}
double pop(struct stack *ps)
{
```

```
if(empty(ps))
printf("%", "stack bi rong");
else
return(ps->nodes[ps->top--]);
}
// Ham dinhtri: tinh mot bieu thuc postfix
double dinhtri(char bieuthuc[])
{
int c, vitri;
double toanhang1, toanhang2, tri;
struct stack s;
s.top = -1; // khoi dong stack
for(vitri = 0; (c = bieuthuc[vitri]) != '\0'; vitri++)
if(lakyso(c)) // c la toan hang
push(&s, (double)(c-'0'));
else // c la toan tu
{
toanhang2 = pop(&s);
toanhang1 = pop(&s);
tri = tinh(c, toanhang1, toanhang2); // tinh ket qua trung gian
push(&s, tri);
}
```

```
return(pop(&s));
}
// Ham lakyso: kiem tra mot ky tu co phai la ky so hay khong
int lakyso(char kytu)
{
return(kytu >= '0' && kytu <= '9');
}
/* Ham tinh: tinh tri cua hai toan hang toanhang1 va toanhang2 qua
phep toan toantu */
double tinh(int toantu, double toanhang1, double toanhang2)
{
switch(toantu)
{
case '+':
return(toanhang1 + toanhang2);
case '-':
return(toanhang1 - toanhang2);
case '*':
return(toanhang1 * toanhang2);
case '/':
return(toanhang1 / toanhang2);
case '$':
```

```
return(pow(toanhang1, toanhang2));
default:
printf("%s", "toan tu khong hop le");
exit(1);
}
}
void main()
{
char c, bieuthuc[TOIDA];
int vitri;
clrscr();
do
{
vitri = 0;
printf("\n\nNhap bieu thuc postfix can dinh tri: ");
while ((bieuthuc[vitri++] = getchar()) != '\n') ;
bieuthuc[--vitri] = '\0';
printf("%s%s%s%5.2f", "Bieu thuc ", bieuthuc, " co tri la: ", dinhtri(bieuthuc));
printf("\n\nTiep tuc khong ? (c/k): ");
c = getche();
} while(c == 'c' || c == 'C'); }
```

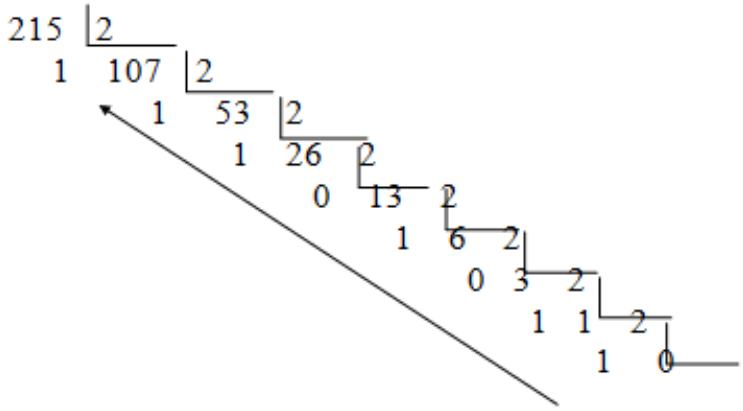
+/Chuyển đổi cơ số

Đổi một số nguyên dạng thập phân sang nhị phân để sử dụng trong máy tính điện tử.

Ví dụ: Biểu diễn số 215 như sau :

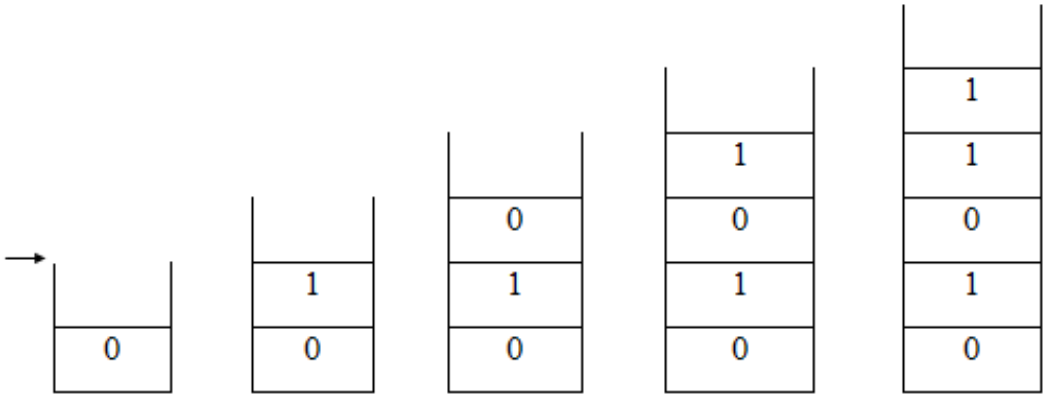
$1.2^7 + 1.2^6 + 0.2^5 + 1.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = (215)_{10}$.

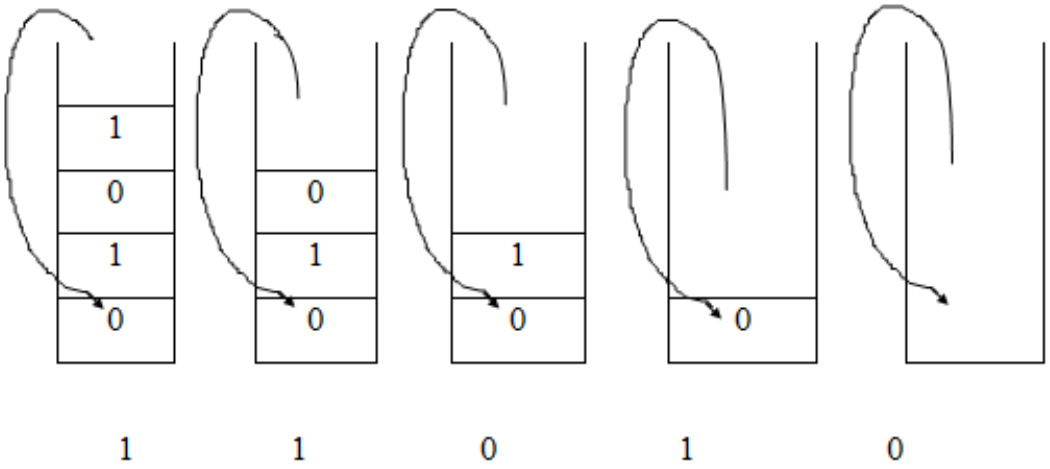
Thuật toán đổi một số nguyên dạng thập phân sang nhị phân là thực hiện phép chia liên tiếp cho 2 và lấy số dư. Các số dư là các số nhị phân theo chiều ngược lại.



⇒ 11010111 (Ở dạng số nhị phân)

Ví dụ: số $(26)_{10} \rightarrow (0\ 1\ 0\ 1\ 1) = (11010)_2$





1 1 0 1 0

Giải thuật chuyển đổi có thể được viết như sau:

Giải thuật thực hiện chuyển đổi biểu diễn cơ số 10 của một số nguyên dương n sang cơ số 2 và hiển thị biểu diễn cơ số 2 này. Giải thuật được viết dưới dạng thủ tục như sau:

Void chuyendoi;

1 - While N ≠ 0

{

R = N % 2; {tính số d R trong phép chia n cho 2}

PUSH (S, T, R);{nạp R vào đỉnh Stack}

N = N / 2;{Thay n bằng thơng của phép chia n cho 2}

}

2 - While S ≠ Ø

{

R = POP (S, T);{ Lấy R ra từ đỉnh Stack}

Cout <<R;

}

3 - return

Chương trình chi tiết như sau:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define STACKSIZE 100
#define TRUE 1
#define FALSE 0
struct stack
{
int top;
int nodes[STACKSIZE];
};
int empty(struct stack *ps)
{
if(ps->top == -1)
return(TRUE);
else
return(FALSE);
}
void push(struct stack *ps, int x)
{
```

```
if(ps->top == STACKSIZE-1)
{
printf("%s", "stack bi day");
exit(1);
}
else
ps->nodes[++(ps->top)] = x;
}
int pop(struct stack *ps)
{
if(empty(ps))
{
printf("%s", "stack bi rong");
exit(1);
}
return(ps->nodes[ps->top--]);
}
main()
{
struct stack s;
int coso, so, sodu;
char c;
```

```
clrscr();
do
{
s.top =- 1; // khoi dong stack
printf("\n\nNhap vao mot so thap phan: ");
scanf("%d", &so);
printf("%s","Muon doi so thap phan nay sang co so may: ");
scanf("%d", &coso);
while (so != 0)
{
sodu = so % coso;
push(&s, sodu); // push so du vao stack
so = so / coso;
}
printf("So da doi la: ");
while(!empty(&s))
printf("%X", pop(&s)); // pop so du ra khoi stack
printf("\n\nBan co muon tiep tuc khong? (c/k): ");
c = getche();
} while(c == 'c' || c == 'C');
}
```

Bài 8: Thực hành cài đặt danh sách Stack

Bài 1: Viết thủ tục 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.

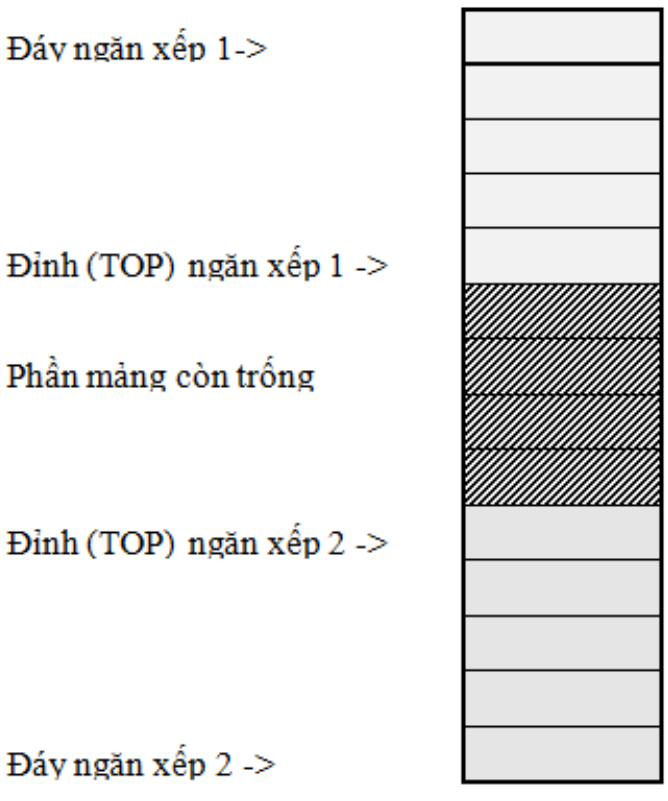
Hướng dẫn: Khi xem xét bài toán ta thấy rằng ký tự ban đầu được đọc vào lại được in ra sau cùng, các ký tự được đọc trước lại in ra sau … Đây chính là cơ chế hoạt động của Stack. Ta dùng một Stack để lưu trữ các ký tự được đọc vào bằng thủ tục Push(c, S), cho đến khi gặp ký tự @ chúng ta sẽ in ra các ký tự lần lượt được lấy ra từ Stack băng thủ tục POP(S).

Bài 2 : Cài đặt chương trình cho phép nhận vào một biểu thức gồm các số, các toán tử +, -, *, /, %, các hàm toán học sin, cos, tan, ln, exp, dấu mở, đóng ngoặc "(", ")" và tính toán giá trị của biểu thức này.

Bài 3 : Dùng ngăn xếp để viết thủ tục đổi một số thập phân sang số nhị phân.

Bài 4 : Viết thủ tục/hàm kiểm tra một chuỗi dấu ngoặc đúng (chuỗi dấu ngoặc đúng là chuỗi dấu mở đóng khớp nhau như trong biểu thức toán học).

Bài 5 : Ta có thể cài đặt 2 ngăn xếp vào trong một mảng, gọi là ngăn xếp hai đầu hoạt động của hai ngăn xếp này như sơ đồ sau:



Hình vẽ Mảng chứa 2 ngăn xếp

Hãy viết các thủ tục cần thiết để cài đặt ngăn xếp hai đầu.

Danh sách tuyến tính kiểu hàng đợi

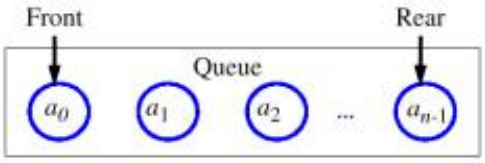
ĐỊNH NGHĨA

Hàng đợi là một vật chứa (container) các đối tượng làm việc theo cơ chế FIFO (First In First Out) nghĩa là việc thêm một đối tượng vào hàng đợi hoặc lấy một đối tượng ra khỏi hàng đợi được thực hiện theo cơ chế "Vào trước ra trước".

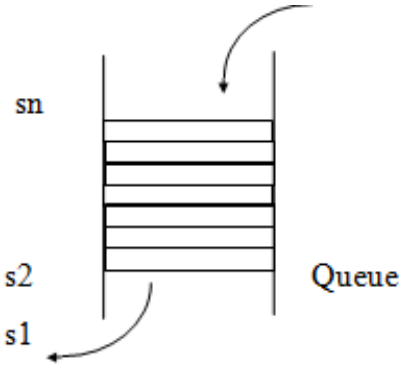
Các đối tượng có thể được thêm vào hàng đợi bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào đầu tiên mới được phép lấy ra khỏi hàng đợi.

Thao tác thêm một đối tượng vào hàng đợi và lấy một đối tượng ra khỏi hàng đợi lần lượt được gọi là "enqueue" và "dequeue".

Việc thêm một đối tượng vào hàng đợi luôn diễn ra ở cuối hàng đợi và một phần tử luôn được lấy ra từ đầu hàng đợi.



Ta hình dung nguyên tắc hoạt động của Queue như sau:



Trong tin học, CTDL hàng đợi có nhiều ứng dụng: khử đệ qui, tổ chức lưu vết các quá trình tìm kiếm theo chiều rộng và quay lui, vét cạn, tổ chức quản lý và phân phối tiến trình trong các hệ điều hành, tổ chức bộ đệm bàn phím, .

Ta có thể định nghĩa CTDL hàng đợi như sau: hàng đợi là một CTDL trừu tượng (ADT) tuyến tính. Tương tự như stack, hàng đợi hỗ trợ các thao tác:

EnQueue(o): Thêm đối tượng o vào cuối hàng đợi

DeQueue(): Lấy đối tượng ở đầu queue ra khỏi hàng đợi và trả về giá trị của nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.

IsEmpty(): Kiểm tra xem hàng đợi có rỗng không.

Front(): Trả về giá trị của phần tử nằm ở đầu hàng đợi mà không hủy nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.

Các thao tác thêm, trích và huỷ một phần tử phải được thực hiện ở 2 phía khác nhau của hàng đợi do đó hoạt động của hàng đợi được thực hiện theo nguyên tắc FIFO (First In First Out - vào trước ra trước).

Cũng như stack, ta có thể dùng cấu trúc mảng 1 chiều hoặc cấu trúc danh sách liên kết để biểu diễn cấu trúc hàng đợi.

CÀI ĐẶT QUEUE

Cài đặt Queue bằng mảng

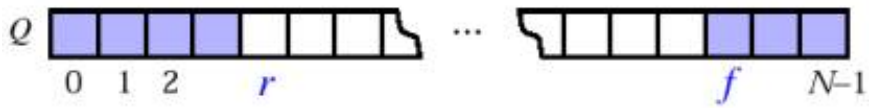
Ta có thể tạo một hàng đợi bằng cách sử dụng một mảng 1 chiều với kích thước tối đa là N (ví dụ, N có thể bằng 1000) theo kiểu xoay vòng (coi phần tử a_{n-1} kề với phần tử a_0).

Như vậy hàng đợi có thể chứa tối đa N phần tử. Phần tử nằm ở đầu hàng đợi (front element) sẽ có chỉ số f. Phần tử nằm ở cuối hàng đợi (rear element) sẽ có chỉ số r (xem hình).

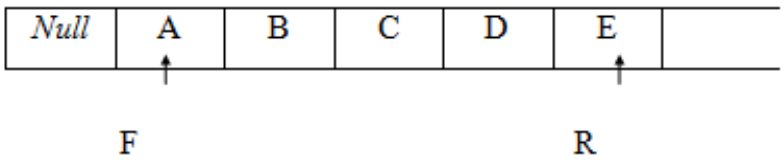
Để khai báo một hàng đợi, ta cần một mảng một chiều Q, hai biến nguyên f, r cho biết chỉ số của đầu và cuối của hàng đợi và hằng số N cho biết kích thước tối đa của hàng đợi. Ngoài ra, khi dùng mảng biểu diễn hàng đợi, ta cũng cần một giá trị đặc biệt để gán cho những ô còn trống trên hàng đợi. Giá trị này là một giá trị nằm ngoài miền xác định của dữ liệu lưu trong hàng đợi. Ta ký hiệu nó là NULLDATA như ở những phần trước.

Trạng thái hàng đợi lúc bình thường:

Trạng thái hàng đợi lúc xoay vòng:



Hoặc:



Khi queue rỗng R = F = 0. Nếu mỗi phần tử của queue được lưu trữ trong một từ máy thì khi bổ sung một phần tử vào queue R sẽ tăng lên 1, còn khi loại bỏ phần tử ra khỏi queue F sẽ tăng lên 1.

Câu hỏi đặt ra: khi giá trị f=r cho ta điều gì ? Ta thấy rằng, lúc này hàng đợi chỉ có thể ở một trong hai trạng thái là rỗng hoặc đầy. Coi như một bài tập các bạn hãy tự suy nghĩ tìm câu trả lời trước khi đọc tiếp để kiểm tra kết quả.

Hàng đợi có thể được khai báo cụ thể như sau:

Data Q[N] ;

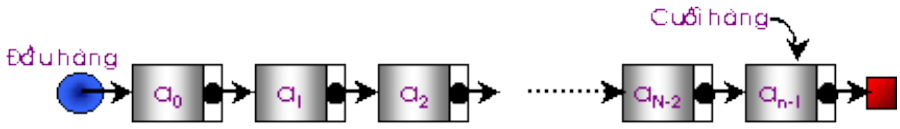
int f, r;

Cũng như strack, do khi cài đặt bằng mảng một chiều, hàng đợi có ki?hước tối đa nên ta cần xây dựng thêm một thao tác phụ cho hàng đợi:

IsFull(): Kiểm tra xem hàng đợi có đầy chưa.

Cài đặt Queue bằng danh sách

Ta có thể tạo một hàng đợi bằng cách sử dụng một DSLK đơn.



Phần tử đầu DSKL (head) sẽ là phần tử đầu hàng đợi, phần tử cuối DSKL (tail) sẽ là phần tử cuối hàng đợi.

Sau đây là các thao tác tương ứng cho array-queue:

Tạo hàng đợi rỗng:

Lệnh Q.pHead = Q.pTail = NULL sẽ tạo ra một hàng đợi rỗng.

Kiểm tra hàng đợi rỗng :

```
char IsEmpty(LIST Q)
{
if (Q.pHead == NULL) // stack rỗng
return 1;
else return 0;
}
```

Thêm một phần tử p vào cuối hàng đợi

```
void EnQueue(LIST Q, Data x)
{
InsertTail(Q, x);
}
```

- Loại bỏ phần tử ở đầu hàng đợi

```
Data DeQueue(LIST Q)
{ Data x;
if (IsEmpty(Q)) return NULLDATA;
x = RemoveFirst(Q);
return x;
}
```

Xem thông tin của phần tử ở đầu hàng đợi

```
Data Front(LIST Q)

{

if (IsEmpty(Q)) return NULLDATA;

return Q.pHead->Info;

}
```

Các thao tác trên đều làm việc với chi phí O(1).

Chương trình minh họa hàng đợi có ưu tiên, cách cài đặt các nút trên hàng đợi có độ ưu tiên giảm dần từ front tới rear.

pqinsert: Chèn nút mới vào vị trí thích hợp trên hàng đợi để đảm bảo độ ưu tiên của các nút giảm dần từ front tới rear

pqremove: Xóa nút có độ ưu tiên cao nhất, nút này là nút tại front

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#define MAXQUEUE 100

#define TRUE 1

#define FALSE 0

// Khai bao cau truc pqueue

struct pqueue

{

int rear; // front luon la 0

int nodes[MAXQUEUE]; // moi nut la mot so nguyen chi do uu tien
```

```
};
// Tac vu pqinitialize: khoi dong hang doi co uu tien
void pqinitialize(struct pqueue *ppq)
{
ppq->rear = -1;
}
// Tac vu pqempty: kiem tra hang doi co rong khong
int pqempty(struct pqueue *ppq)
{
return((ppq->rear == -1) ? TRUE : FALSE);
}
// Tac vu pqueuesize: xac dinh so nut co trong hang doi
int pqueuesize(struct pqueue *ppq)
{
return(ppq->rear+1);
}
// Tac vu pqinsert: them nut vao hang doi co uu tien
void pqinsert(struct pqueue *ppq, int x)
{
int i, j;
if(ppq->rear == MAXQUEUE-1)
printf("Hang doi bi day, khong them nut duoc");
```

```
else
{
// tim vi tri chen
for(i = 0; i < pqueuesize(ppq) && ppq->nodes[i] >= x; i++)
;
// doi cho cac nut tu nut cuoi den nut i+1 len mot vi tri
for (j = pqueuesize(ppq) ; j > i; j--)
ppq->nodes[j] = ppq->nodes[j-1];
ppq->nodes[i] = x;
ppq->rear++;
}
}
/* Tac vu pqremove: xoa nut co do uu tien cao nhat (nut o front), truong hop
nay ta phai doi cac nut tu nut thu hai den nut cuoi xuong mot vi tri */
int pqremove(struct pqueue *ppq)
{
int x, i;
if(pqempty(ppq))
printf("Hang doi bi rong, khong xoa nut duoc");
else
{
x = ppq->nodes[0]; // do uu tien cua nut can xoa
```

```
// doi cho
for (i = 0; i < ppq->rear; i++)
ppq->nodes[i] = ppq->nodes[i+1];
ppq->rear--;
return x;
}
}
// Tac vu pqtraverse: duyet hang doi co uu tien tu front den rear
void pqtraverse(struct pqueue *ppq)
{
int i;
if(pqempty(ppq))
printf("hang doi bi rong");
else
for(i = 0; i <= ppq->rear; i++)
printf("%d ", ppq->nodes[i]);
}
void main(void)
{
struct pqueue pq;
int douutien, chucnang;
char c;
```

```
clrscr();
// khoi dong hang doi
pqinitialize(&pq);
do
{
// menu chinh cua chuong trinh
printf("\n\n\t\tCHUONG TRINH MINH HOA HANG DOI CO UU TIEN\n");
printf("\nCac chuc nang cua chuong trinh:\n");
printf(" 1: Them nut vao hang doi co uu tien\n");
printf(" 2: Xoa nut co do uu tien cao nhat\n");
printf(" 3: Xoa toan bo hang doi\n");
printf(" 4: Duyet hang doi\n");
printf(" 0: Ket thuc chuong trinh\n");
printf("Chuc nang ban chon: ");
scanf("%d", &chucnang);
switch(chucnang)
{
case 1:
{
printf("\nDo uu tien cua nut moi: ");
scanf("%d", &douutien);
pqinsert(&pq, douutien);
```

```
break;
}
case 2:
{
pqremove(&pq);
break;
}
case 3:
{
printf("\nBan co chac khong (c/k): ");
c = getche();
if(c == 'c' || c == 'C')
pqinitialize(&pq);
break;
}
case 4:
{
printf("\nDuyet hang doi: ");
pqtraverse(&pq);
break;
}
}
```

```
} while(chucnang != 0);

}
```

Lưu ý, nếu không quản lý phần tử cuối xâu, thao tác dequeue sẽ có độ phức tạp O(n).

ỨNG DỤNG CỦA QUEUE

Hàng đợi có thể được sử dụng trong một số bài toán:

Bài toán sản xuất và tiêu thụ (ứng dụng trong các hệ điều hành song song).

Bộ đệm (ví dụ: Nhấn phím -> Bộ đệm -> CPU xử lý).

Xử lý các lệnh trong máy tính (ứng dụng trong HĐH, trình biên dịch), hàng đợi các tiến trình chờ được xử lý, ..

Một số ví dụ:

Chuong trinh quan ly kho

Mat hang nao nhap kho truoc se duoc xuat kho truoc

```
#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#define MAXQUEUE 100

#define TRUE 1

#define FALSE 0

// Khai bao cau truc mathang

typedef struct mathang

{

int mamh;

char tenmh[12];
```

```
};
struct queue
{
int front, rear;
mathang nodes[MAXQUEUE];
};
void initialize(struct queue *pq)
{
pq->front = pq->rear = MAXQUEUE-1;
}
int empty(struct queue *pq)
{
return((pq->front == pq->rear) ? TRUE : FALSE);
}
void insert(struct queue *pq, mathang x)
{
if(pq->rear == MAXQUEUE-1)
pq->rear = 0;
else
(pq->rear)++;
if(pq->rear == pq->front)
printf("kho hang bi day");
```

```
else
pq->nodes[pq->rear] = x;
}
mathang remove(struct queue *pq)
{
if(empty(pq))
printf("kho khong con hang");
else
{
if(pq->front == MAXQUEUE-1)
pq->front = 0;
else
(pq->front)++;
return(pq->nodes[pq->front]);
}
}
// Tac vu traverse: duyet kho hang tu front toi rear
void traverse(struct queue *pq)
{
int i;
if(empty(pq))
{
```

```
printf("\nKho khong con hang");
return;
}
if(pq->front == MAXQUEUE-1)
i = 0;
else
i = pq->front+1;
// vong lap in cac nut tu front den nut ke cuoi
while(i != pq->rear)
{
printf("\n%11d%15s", pq->nodes[i].mamh, pq->nodes[i].tenmh);
if(i == MAXQUEUE-1)
i = 0;
else
i++;
}
// in nut cuoi (nut tai rear)
printf("\n%11d%15s", pq->nodes[i].mamh, pq->nodes[i].tenmh);
}
// chuong trinh chinh
void main(void)
{
```

```
struct queue q;
int chucnang, front1;
char c;
mathang mh;
clrscr();
// khoi tao queue
initialize(&q);
do
{
printf("\n\n\t\t\tCHUONG TRINH QUAN LY KHO");
printf("\n\t\t\t(NHAP TRUOC - XUAT TRUOC)");
printf("\n\nCac chuc nang cua chuong trinh:\n");
printf(" 1: Nhap mot mat hang\n");
printf(" 2: Xuat mot mat hang\n");
printf(" 3: Xem mat hang chuan bi xuat\n");
printf(" 4: Xem mat hang moi nhap\n");
printf(" 5: Xem cac mat hang co trong kho\n");
printf(" 6: Xuat toan bo kho hang\n");
printf(" 0: Ket thuc chuong trinh\n");
printf("Chuc nang ban chon: ");
scanf("%d", &chucnang);
switch(chucnang)
```

```
{
case 1:
{
printf("\nMa mat hang: ");
scanf("%d", &mh.mamh);
printf("Ten mat hang: ");
scanf("%s", &mh.tenmh);
insert(&q, mh);
break;
}
case 2:
{
if(!empty(&q))
{
mh = remove(&q);
printf("\nMat hang xuat: Ma:%d Ten:%s", mh.mamh, mh.tenmh);
}
else
printf("\nKho khong con hang");
break;
}
case 3:
```

```
{
front1 = (q.front == MAXQUEUE-1 ? 0 : q.front+1);
printf("\nMat hang chuan bi xuat: Ma:%d Ten:%s",
q.nodes[front1].mamh, q.nodes[front1].tenmh);
break;
}
case 4:
{
printf("\nMat hang moi nhap: Ma:%d Ten:%s",
q.nodes[q.rear].mamh, q.nodes[q.rear].tenmh);
break;
}
case 5:
{
printf("\nCac mat hang co trong kho:");
printf("\n%11s%15s", "MA MAT HANG", " TEN MAT HANG");
traverse(&q);
break;
}
case 6: // xoa toan bo queue (khoi dong queue)
{
printf("\nBan co chac khong (c/k): ");
```

```
c = getche();

if(c == 'c' || c == 'C')

initialize(&q);

break;

}

}

} while(chucnang != 0);

}
```

Hàng đợi hai đầu (double-ended queue)

Hàng đợi hai đầu (gọi tắt là Deque) là một vật chứa các đối tượng mà việc thêm hoặc hủy một đối tượng được thực hiện ở cả 2 đầu của nó.

Ta có thể định nghĩa CTDL deque như sau: deque là một CTDL trừu tượng (ADT) hỗ trợ các thao tác chính sau:

InsertFirst(e): Thêm đối tượng e vào đầu deque

InsertLast(e): Thêm đối tượng e vào cuối deque

RemoveFirst(): Lấy đối tượng ở đầu deque ra khỏi deque và trả về giá trị của nó.

RemoveLast(): Lấy đối tượng ở cuối deque ra khỏi deque và trả về giá trị của nó.

Ngoài ra, deque cũng hỗ trợ các thao tác sau:

IsEmpty(): Kiểm tra xem deque có rỗng không.

First(): Trả về giá trị của phần tử nằm ở đầu deque mà không hủy nó.

Last(): Trả về giá trị của phần tử nằm ở cuối deque mà không hủy nó.

Dùng deque để cài đặt stack và queue

Ta có thể dùng deque để biểu diễn stack. Khi đó ta có các thao tác tương ứng như sau:

STT	Stack	Deque
1	Push	InsertLast
2	Pop	RemoveLast
3	Top	Last
4	IsEmpty	IsEmpty

Tương tự, ta có thể dùng deque để biểu diễn queue. Khi đó ta có các thao tác tương ứng như sau:

STT	Queue	Deque
1	Enqueue	InsertLast
2	Dequeue	RemoveFist
3	Front	First
4	IsEmpty	IsEmpty

Cài đặt deque

Do đặc tính truy xuất hai đầu của deque, việc xây dựng CTDL biểu diễn nó phải phù hợp.

Ta có thể cài đặt CTDL deque bằng danh sách liên kết đơn. Tuy nhiên, khi đó thao tác RemoveLast hủy phần tử ở cuối deque sẽ tốn chi phí O(n). Điều này làm giảm hiệu quả của CTDL. Thích hợp nhất để cài đặt deque là dùng danh sách liên kết kép. Tất cả các thao tác trên deque khi đó sẽ chỉ tốn chi phí O(1)

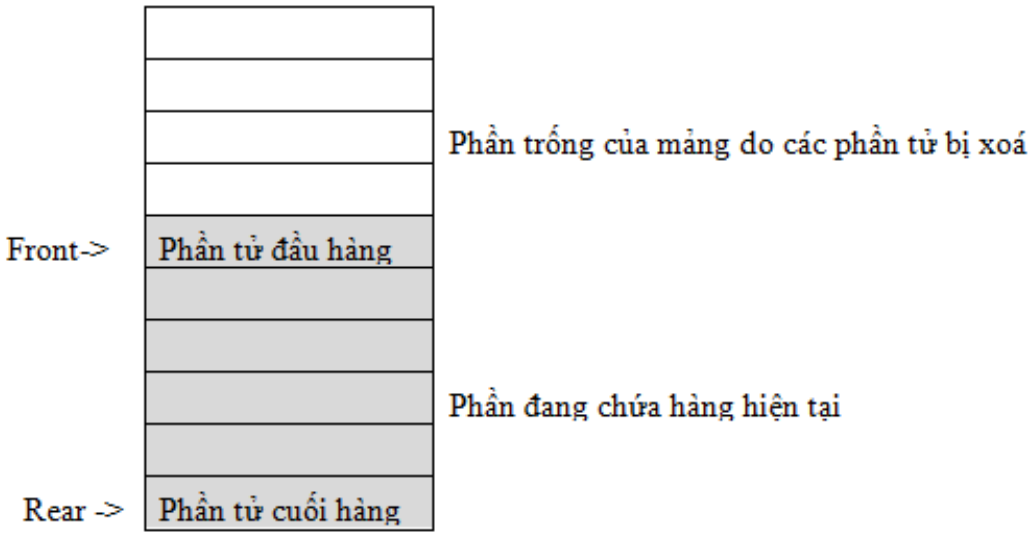
Thực hành cái đặt danh sách kiểu hàng đợi

CÀI ĐẶT 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, phần tử thứ 2 vào vị trí thứ 2 của mảng... Giả sử hàng có n phần tử, ta có front=1 và rear=n. Khi xoá một phần tử front tăng lên 1, khi thêm một phần tử rear tăng lên 1. Như vậy hàng có khuynh hướng đi xuống, đến một lúc nào đó ta không thể thêm vào hàng được nữa dù mảng còn nhiều chỗ trống (các vị trí trước front) trường hợp này ta gọi là hàng bị tràn (xem hình 2).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-1 vị trí, Cách này gọi là di chuyển tịnh tiến. Trong trường hợp này ta luôn có front <= rear.
- Xem mảng như là một vòng tròn nghĩa là khi hàng bị tràn nhưng chưa đầy ta thêm phần tử mới vào vị trí 1 của mảng, thêm một phần tử mới nữa thì thêm vào vị trí 2 (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 dưới).



Hình : hàng bị tràn khi có một phần nữa thêm vào

Cài đặt hàng với phương pháp di chuyển tịnh tiến khi hàng bị tràn.

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

Bài 1: Tạo hàng rỗng

Hướng dẫn : 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 0

Bài 2 : Kiểm tra hàng rỗng

Hướng dẫn : Trong quá trình làm việc ta có thể thêm và xoá các phần tử trong hàng. Rõ ràng, nếu ta có đưa vào hàng một phần tử nào đó thì front>0. Khi xoá một phần tử ta tăng front lên 1. Hàng rỗng nếu front>rear. Hơn nữa khi mới khởi tạo hàng, tức là front = 0, thì hàng cũng rỗng. Tuy nhiên để phép kiểm tra hàng rỗng đơn giản, ta sẽ làm một phép kiểm tra khi xoá một phần tử của hàng, nếu phần tử bị xoá là phần tử duy nhất trong hàng thì ta đặt lại front = 0. Vậy hàng rỗng khi và chỉ khi front = 0.

Bài 3: Kiểm tra hàng đầy

Hướng dẫn : Hàng đầy nếu số phần tử hiện có trong hàng bằng độ dài của mảng.

Bài 4: Xoá một phần tử của hàng

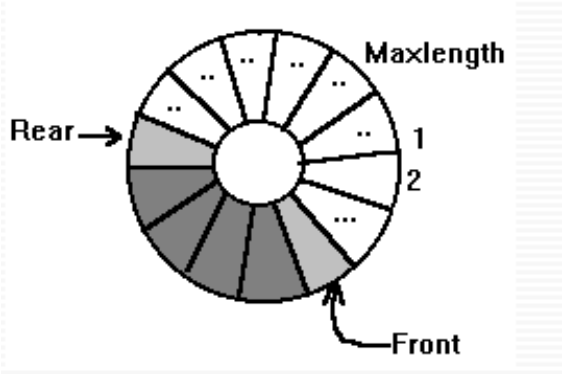
Hướng dẫn : Xoá phần tử đầu hàng ta chỉ cần cho front tăng lên 1. Nếu front > rear thì hàng thực chất đã rỗng, nên ta khởi tạo lại hàng rỗng (tức là đặt lại giá trị front = rear =0)

Bài 5: Thêm một phần tử vào hàng

Hướng dẫn : Khi thêm một phần tử vào hàng ta xét các trường hợp

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

Cài đặt hàng với phương pháp mảng xoay vòng



Hình : Cài đặt hàng bằng mảng xoay vòng

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

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

Bài 1: Tạo hàng rỗng

Hướng dẫn : 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 0

Bài 2 : Kiểm tra hàng rỗng

Hướng dẫn : Trong quá trình làm việc ta có thể thêm và xoá các phần tử trong hàng. Rõ ràng, nếu ta có đưa vào hàng một phần tử nào đó thì front>0. Khi xoá một phần tử ta tăng front lên 1. Hàng rỗng nếu front>rear. Hơn nữa khi mới khởi tạo hàng, tức là front = 0, thì hàng cũng rỗng. Tuy nhiên để phép kiểm tra hàng rỗng đơn giản, ta sẽ làm một phép kiểm tra khi xoá một phần tử của hàng, nếu phần tử bị xoá là phần tử duy nhất trong hàng thì ta đặt lại front=0. Vậy hàng rỗng khi và chỉ khi front =0.

Bài 3: Kiểm tra hàng đầy

Hướng dẫn : 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, do đó hàm được viết như sau:

Bài 4: Xoá một phần tử của hàng

Hướng dẫn : Xoá phần tử đầu hàng ta chỉ cần cho front tăng lên 1. Nếu front > rear thì hàng thực chất đã rỗng, nên ta khởi tạo lại hàng rỗng (tức là đặt lại giá trị front = rear =0)

Bài 5: Thêm một phần tử vào hàng

Hướng dẫn : Khi thêm một phần tử vào hàng ta xét các trường hợp

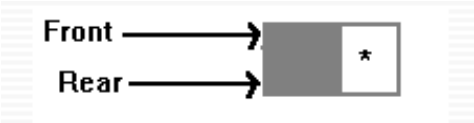
- Nếu hàng đầy thì không thêm được nữa.
- Nếu hàng chưa đầy ta phải xét xem hàng có bị tràn không. Nếu hàng bị tràn ta di chuyển tịnh tiến rồi mới nối thêm phần tử mới vào đuôi hàng, rear tăng lên 1. Đặc biệt nếu thêm vào hàng rỗng thì ta cho front=1 để front trỏ đúng phần tử đầu tiên của hàng.

CÀI ĐẶT HÀNG BẰNG DANH SÁCH LIÊN KẾT (dùng con trỏ)

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

Bài 1: Tạo hàng rỗng

Hướng dẫn : Front và rear cùng trỏ đến HEADER của hàng.



Bài 2 : Kiểm tra hàng rỗng

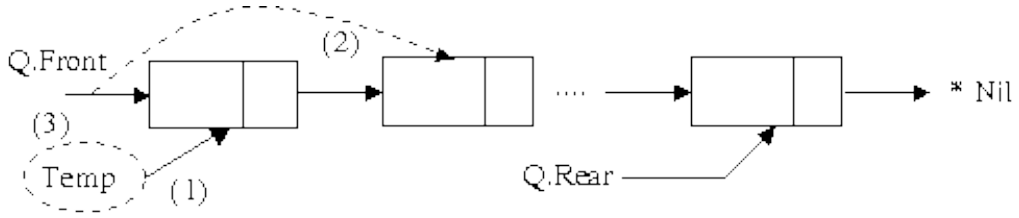
Hướng dẫn : Hàng rỗng nếu front và rear chỉ cùng một ô, ô đó chính là HEADER

Bài 3: Kiểm tra hàng đầy

Hướng dẫn : 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, do đó hàm được viết như sau:

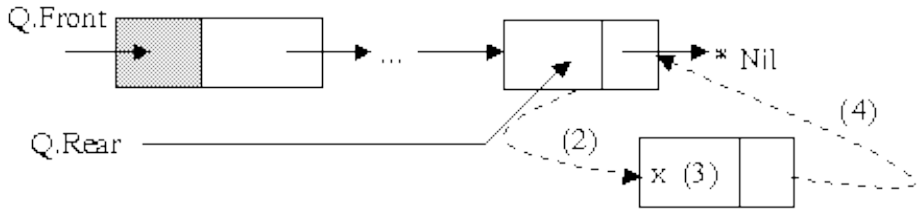
Bài 4: Xoá một phần tử của hàng

Hướng dẫn : Thực chất là xoá phần tử nằm ở đầu hàng. Ta chỉ cần cho front trỏ tới vị trí kế tiếp nó trong danh sách.



Bài 5: Thêm một phần tử vào hàng

Hướng dẫn : Thêm một phần tử vào hàng ta thêm vào sau rear (rear^.next), rồi cho rear trỏ đến phần tử mới này, xem hình. Trường next của ô mới này trỏ tới NIL.

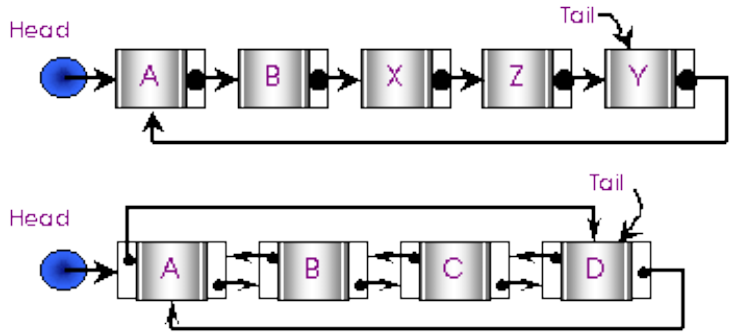


Danh sách nối vòng và nối kép

DANH SÁCH NỐI VÒNG (Circulary Linked List)

Định nghĩa và nguyên tắc

Danh sách liên kết vòng (xâu vòng) là một danh sách đơn (hoặc kép) mà phần tử cuối danh sách thay vì mang giá trị NULL, trỏ tới phần tử đầu danh sách. Để biểu diễn, ta có thể xử dụng các kỹ thuật biểu diễn như danh sách đơn (hoặc kép).



Ta có thể khai báo xâu vòng như khai báo xâu đơn (hoặc kép). Trên danh sách vòng ta có các thao tác thường gặp sau:

Tìm phần tử trên danh sách vòng

Danh sách vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kỳ trên danh sách xem như phân tử đầu xâu để kiểm tra việc duyệt đã qua hết các phần tử của danh sách hay chưa.

```
NODE* Search(LIST &l, Data x)

{

NODE *p;

p = l.pHead;

do

{

if ( p->Info == x)
```

```
return p;
p = p->pNext;
}while (p != l.pHead); // chưa đi giáp vòng
return p;
}
```

Thêm phần tử đầu xâu

```
void AddHead(LIST &l, NODE *new_ele)
{
if(l.pHead == NULL) //Xâu rỗng
{
l.pHead = l.pTail = new_ele;
l.pTail->pNext = l.pHead;
}
else
{
new_ele->pNext = l.pHead;
l.pTail->pNext = new_ele;
l.pHead = new_ele;
}
}
```

Thêm phần tử cuối xâu

```
void AddTail(LIST &l, NODE *new_ele)
```

```
{
if(l.pHead == NULL) //Xâu rỗng
{
l.pHead = l.pTail = new_ele;
l.pTail->pNext = l.pHead;
}
else
{
new_ele->pNext = l.pHead;
l.pTail->pNext = new_ele;
l.pTail = new_ele;
}
}
```

Thêm phần tử sau nút q

```
void AddAfter(LIST &l, NODE *q, NODE *new_ele)
{
if(l.pHead == NULL) //Xâu rỗng
{
l.pHead = l.pTail = new_ele;
l.pTail->pNext = l.pHead;
}
else
```

```
{
new_ele->pNext = q->pNext;
q->pNext = new_ele;
if(q == l.pTail)
l.pTail = new_ele;
}
}
```

Hủy phần tử đầu xâu

```
void RemoveHead(LIST &l)
{ NODE *p = l.pHead;

if(p == NULL) return;
if (l.pHead = l.pTail) l.pHead = l.pTail = NULL;
else
{
l.pHead = p->Next;
if(p == l.pTail)
l.pTail->pNext = l.pHead;
}
delete p;
}
```

Hủy phần tử đứng sau nút q

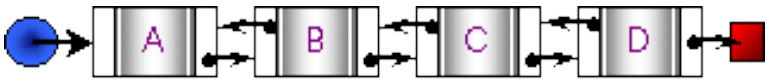
```
void RemoveAfter(LIST &l, NODE *q)
{ NODE *p;

if(q != NULL)
{
p = q ->Next ;
if ( p == q) l.pHead = l.pTail = NULL;
else
{
q->Next = p->Next;
if(p == l.pTail)
l.pTail = q;
}
delete p;
}
}
```

NHẬN XÉT

Đối với danh sách vòng, có thể xuất phát từ một phần tử bất kỳ để duyệt toàn bộ danh sách

DANH SÁCH NỐI KÉP

Danh sách liên kết kép là danh sách mà mỗi phần tử trong danh sách có kết nối với 1 phần tử đứng trước và 1 phần tử đứng sau nó.



Các khai báo sau định nghĩa một danh sách liên kết kép đơn giản trong đó ta dùng hai con trỏ: pPrev liên kết với phần tử đứng trước và pNext như thường lệ, liên kết với phần tử đứng sau:

```
typedef  struct tagDNode

{

Data Info;

struct tagDNode* pPre;  // trỏ đến phần tử đứng trước

struct tagDNode* pNext; // trỏ đến phần tử đứng sau

}DNODE;

typedef  struct tagDList

{

DNODE* pHead;  // trỏ đến phần tử đầu danh sách

DNODE* pTail; // trỏ đến phần tử cuối danh sách

}DLIST;
```

khi đó, thủ tục khởi tạo một phần tử cho danh sách liên kết kép được viết lại như sau :

```
DNODE* GetNode(Data x)

{ DNODE *p;

  // Cấp phát vùng nhớ cho phần tử

p = new DNODE;

if ( p==NULL) {

printf("Không đủ bộ nhớ");

exit(1);
```

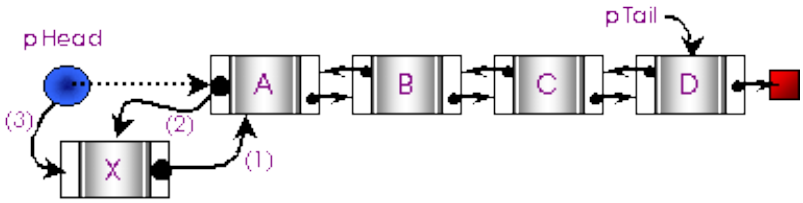
```
}

// Gán thông tin cho phần tử p

p ->Info = x;

p->pPrev = NULL;

p->pNext = NULL;

return p;

}
```

Tương tự danh sách liên kết đơn, ta có thể xây dựng các thao tác cơ bản trên danh sách liên kết kép (xâu kép). Một số thao tác không khác gì trên xâu đơn. Dưới đây là một số thao tác đặc trưng của xâu kép:

Chèn một phần tử vào danh sách:

Có 4 loại thao tác chèn new_ele vào danh sách:

Cách 1: Chèn vào đầu danh sách



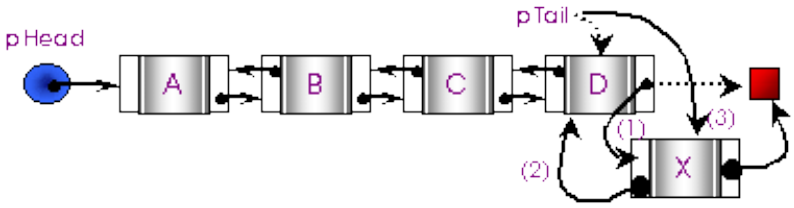
Cài đặt :

```
void AddFirst(DLIST &l, DNODE* new_ele)

{

if (l.pHead==NULL) //Xâu rỗng

{

l.pHead = new_ele; l.pTail = l.pHead;

}
```

```
else
{
new_ele->pNext = l.pHead; // (1)
l.pHead ->pPrev = new_ele; // (2)
l.pHead = new_ele;  // (3)
}
}
NODE* InsertHead(DLIST &l, Data x)
{ NODE* new_ele = GetNode(x);

if (new_ele ==NULL) return NULL;
if (l.pHead==NULL)
{
l.pHead = new_ele; l.pTail = l.pHead;
}
else
{
new_ele->pNext = l.pHead; // (1)
l.pHead ->pPrev = new_ele; // (2)
l.pHead = new_ele;  // (3)
}
return new_ele;
```

}

Cách 2: Chèn vào cuối danh sách

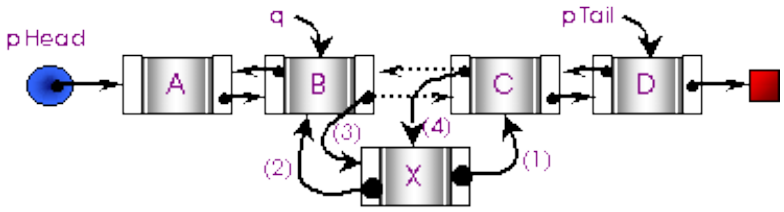


Cài đặt :

```
void AddTail(DLIST &l, DNODE *new_ele)
{
if (l.pHead==NULL)
{
l.pHead = new_ele; l.pTail = l.pHead;
}
else
{
l.pTail->Next = new_ele; // (1)
new_ele ->pPrev = l.pTail; // (2)
l.pTail = new_ele;  // (3)
}
}
NODE* InsertTail(DLIST &l, Data x)
{ NODE* new_ele = GetNode(x);
```

```
if (new_ele ==NULL) return NULL;
if (l.pHead==NULL)
{
l.pHead = new_ele; l.pTail = l.pHead;
}
else
{
l.pTail->Next = new_ele; // (1)
new_ele ->pPrev = l.pTail; // (2)
l.pTail = new_ele;  // (3)
}
return new_ele;
}
```

Cách 3 : Chèn vào danh sách sau một phần tử q



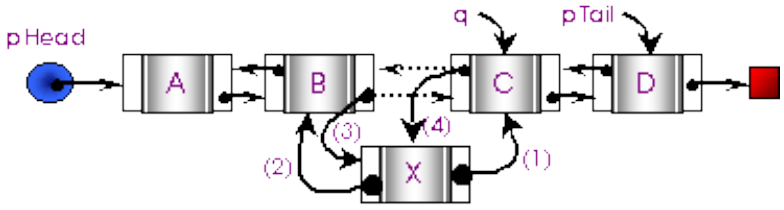
Cài đặt :

```
void AddAfter(DLIST &l, DNODE* q,DNODE* new_ele)
{ DNODE* p = q->pNext;

if ( q!=NULL)
```

```
{
new_ele->pNext = p; //(1)
new_ele->pPrev = q; //(2)
q->pNext = new_ele; //(3)
if(p != NULL)
p->pPrev = new_ele; //(4)
if(q == l.pTail)
l.pTail = new_ele;
}
else //chèn vào đầu danh sách
AddFirst(l, new_ele);
}
 void InsertAfter(DLIST &l, DNODE *q, Data x)
{
DNODE* p = q->pNext;
NODE* new_ele = GetNode(x);
  if (new_ele ==NULL) return NULL;
  if ( q!=NULL)
{
new_ele->pNext = p; //(1)
new_ele->pPrev = q; //(2)
q->pNext = new_ele; //(3)
```

```
if(p != NULL)
p->pPrev = new_ele; //(4)
if(q == l.pTail)
l.pTail = new_ele;
}
else //chèn vào đầu danh sách
AddFirst(l, new_ele);
}
```

Cách 4 : Chèn vào danh sách trước một phần tử q



Cài đặt :

```
void AddBefore(DLIST &l, DNODE q, DNODE* new_ele)
{ DNODE* p = q->pPrev;
if ( q!=NULL)
{
new_ele->pNext = q; //(1)
new_ele->pPrev = p; //(2)
q->pPrev = new_ele; //(3)
if(p != NULL)
p->pNext = new_ele; //(4)
```

```
if(q == l.pHead)
l.pHead = new_ele;
}
else //chèn vào đầu danh sách
AddTail(l, new_ele);
}
  void InsertBefore(DLIST &l, DNODE q, Data x)
{  DNODE* p = q->pPrev;
NODE* new_ele = GetNode(x);
if (new_ele ==NULL) return NULL;
  if ( q!=NULL)
{
new_ele->pNext = q; //(1)
new_ele->pPrev = p; //(2)
q->pPrev = new_ele; //(3)
if(p != NULL)
p->pNext = new_ele; //(4)
if(q == l.pHead)
l.pHead = new_ele;
}
else //chèn vào đầu danh sách
AddTail(l, new_ele);
```

```
}
```

Hủy một phần tử khỏi danh sách

Có 5 loại thao tác thông dụng hủy một phần tử ra khỏi xâu. Chúng ta sẽ lần lượt khảo sát chúng.

Hủy phần tử đầu xâu:

```
Data RemoveHead(DLIST &l)

{ DNODE *p;

Data x = NULLDATA;

if ( l.pHead != NULL)

{

p = l.pHead; x = p->Info;

l.pHead = l.pHead->pNext;

l.pHead->pPrev = NULL;

delete p;

if(l.pHead == NULL) l.pTail = NULL;

else l.pHead->pPrev = NULL;

}

return x; }
```

Hủy phần tử cuối xâu:

```
Data RemoveTail(DLIST &l)

{

DNODE *p;

Data     x = NULLDATA;
```

```
if ( l.pTail != NULL)
{
p = l.pTail; x = p->Info;
l.pTail = l.pTail->pPrev;
l.pTail->pNext = NULL;
delete p;
if(l.pHead == NULL)     l.pTail = NULL;
else l.pHead->pPrev = NULL;
}
return x;
}
```

Hủy một phần tử đứng sau phần tử q

```
void RemoveAfter (DLIST &l, DNODE *q)
{     DNODE *p;
     if ( q != NULL)
{
p = q ->pNext ;
if ( p != NULL)
{
q->pNext = p->pNext;
if(p == l.pTail) l.pTail = q;
else p->pNext->pPrev = q;
```

```
delete p;
}
}
else
RemoveHead(l);
}
```

Hủy một phần tửđứng trước phần tử q

```
void RemoveAfter (DLIST &l, DNODE *q)
{ DNODE *p;

if ( q != NULL)
{
p = q ->pPrev;
if ( p != NULL)
{
q->pPrev = p->pPrev;
if(p == l.pHead) l.pHead = q;
else p->pPrev->pNext = q;
delete p;
}
}
else
```

```
RemoveTail(l);
}
```

Hủy 1 phần tử có khoá k

```
int RemoveNode(DLIST &l, Data k)
{
DNODE *p = l.pHead;
NODE *q;

while( p != NULL)
{
if(p->Info == k) break;
p = p->pNext;
}
if(p == NULL) return 0; //Không tìm thấy k
q = p->pPrev;
if ( q != NULL)
{
p = q ->pNext ;
if ( p != NULL)
{
q->pNext = p->pNext;
if(p == l.pTail)
```

```
l.pTail = q;

else p->pNext->pPrev = q;

}

}

else //p là phần tử đầu xâu

{

l.pHead = p->pNext;

if(l.pHead == NULL)

l.pTail = NULL;

else

l.pHead->pPrev = NULL;

}

delete p;

return 1;

}
```

NHẬN XÉT:

Xâu kép về mặt cơ bản có tính chất giống như xâu đơn. Tuy nhiên nó có một số tính chất khác xâu đơn như sau:

- Xâu kép có mối liên kết hai chiều nên từ một phần tử bất kỳ có thể truy xuất một phần tử bất kỳ khác. Trong khi trên xâu đơn ta chỉ có thể truy xuất đến các phần tử đứng sau một phần tử cho trước. Điều này dẫn đến việc ta có thể dễ dàng hủy phần tử cuối xâu kép, còn trên xâu đơn thao tác này tồn chi phí O(n).

Bù lại, xâu kép tốn chi phí gấp đôi so với xâu đơn cho việc lưu trữ các mối liên kết. Điều này khiến việc cập nhật cũng nặng nề hơn trong một số trường hợp. Như vậy ta cần cân nhắc lựa chọn CTDL hợp lý khi cài đặt cho một ứng dụng cụ thể.

Thực hành cài đặt danh sách liên kết kép

THỰC HÀNH CÀI ĐẶT DANH SÁCH LIÊN KẾT KÉP

Viết chương trình lưu trữ một danh sách các số nguyên, sắp xếp danh sách theo thứ tự (tăng hoặc giảm), trộn 2 danh sách có thứ tự để được một danh sách mới có thứ tự.

Yêu cầu chi tiết:

1. Viết các khai báo cần thiết để cài đặt một danh sách các số nguyên.

2. Viết thủ tục khởi tạo một danh sách rỗng.

3. Viết hàm kiểm tra danh sách rỗng.

4. Viết thủ tục nhập một danh sách.

5. Viết thủ tục hiển thị danh sách ra màn hình.

6. Viết thủ tục sắp xếp danh sách theo thứ tự (tăng hoặc giảm).

7. Xen một phần tử mới x vào danh sách sau cho danh sách mới vẫn bảo đảm thứ tự.

8. Xóa một phần tử x ra khỏi danh sách sao cho danh sách mới vẫn bảo đảm thứ tự.

9. viết thủ tục trộn 2 danh sách đã có thứ tự thành một danh sách mới sao cho danh sách mới vẫn bảo đảm thứ tự.

Viết chương trình nhập vào một danh sách các số nguyên và thực hiện các yêu cầu sau:

- Hiển thị danh sách vừa nhập.
- Sắp xếp danh sách theo thứ tự. Hiển thị danh sách sau khi sắp xếp.
- Xen một phần tử mới vào danh sách. Hiển thị danh sách mới sau khi xen.
- Xóa một phần tử khỏi danh sách. Hiển thị danh sách mới sau khi xóa.
- Nhập 2 danh sách, sắp xếp 2 danh sách theo thứ tự, sau đó trộn 2 danh sách này để được một danh sách mới cũng có thứ tự. Hiển thị danh sách mới ra màn hình để kiểm tra.

Kiểu dữ liệu cây

CÂY VÀ CÁC KHÁI NIỆM VỀ CÂY

Định nghĩa 1:

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

Định nghĩa 2:

Cây được định nghĩa đệ qui như sau

1. Một nút là một cây và nút này cũng là gỗc của cây.

2. Giả sử T 1 , T 2 , ...,T n (n ≥ 1) là các cây có gốc tương ứng r 1 , r 2 ,..., r n . Khi đó cây T với gốc r được hình thành bằng cách cho r trở thành nút cha của các nút r 1 , r 2 ,..., r n

Một số khái niệm cơ bản

Bậc của một nút: là số con của nút đó

Bậc của một cây: là bậc lớn nhất của các nút có trên cây đó (số cây con tối đa của một nút thuộc cây). Cây có bậc n thì gọi là cây n - phân

Nút gốc: là nút có không có nút cha

Nút lá: là nút có bậc bằng 0

Nút nhánh: là nút có bậc khác 0 và không phải là nút gốc

Mức của một nút

Mức (gốc (T_0)) =1

Gọi T_1, T_2,..., T_n là các cây con của T_0.

Khi đó Mức (T_1) = Mức (T_2) = ... = Mức (T_n) = Mức (T_0) +1 Chiều cao của cây: là số mức lớn nhất có trên cây đó

Đường đi: Dãy các đỉnh $n_1, n_2, ...,n_k$ được gọi là đường đi nếu n_i là cha của n_{i+1} ($1 \leq i \leq k-1$

Độ dài của đường đi: là số nút trên đường đi -1

Cây được sắp : Trong một cây, nếu các cây con của mỗi đỉnh được sắp theo một thứ nhất định, thì cây được gọi là cây được sắp (cây có thứ tự). Chẳng hạn, hình minh hoạ hai cây được sắp khác nhau

Hình 13.1. Hai cây được sắp khác nhau

Rừng: là tập hợp hữu hạn các cây phân biệt

Hình 13.2. Rừng gồm ba cây

CÂY TỔNG QUÁT

Biểu diễn cây tổng quát

- Đối với cây tổng quát cấp m nào đó có thể sử dụng cách biểu diễn móc nối tương tự như đối với cây nhị phân. Như vậy ứng với mỗi nút ta phải dành ra m trường mối nối để trỏ tới các con của nút đó và như vậy số mối nối không sẽ rất nhiều: nếu cây có n nút sẽ có tới n(m-1) + 1"mối nối không" trong số m.n mối nối.

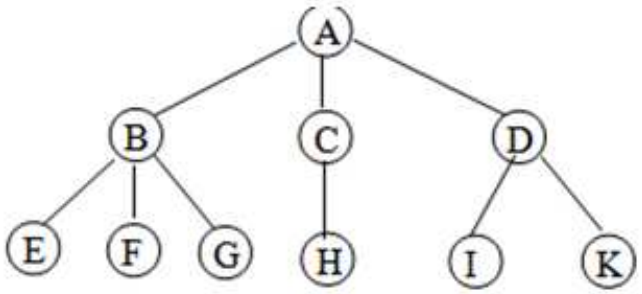
- Nếu tuỳ theo số con của từng nút mà định ra mối nối nghĩa là dùng nút có kích thước biến đổi thì sự tiết kiện không gian nhớ này sẽ phải trả giá bằng những phức tạp của quá trình xử lý trên cây.

- Để khắc phục các nhược điêm trên là dùng cách biểu diễn cây nhị phân để biểu diễn cây tổngquát.

Ta có thể biến đổi một cây bất kỳ thành một cây nhị phân theo qui tắc sau

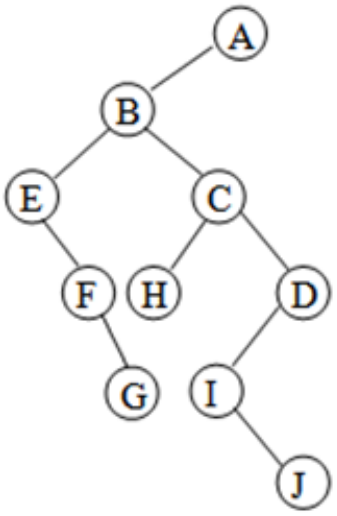
- Giữ lại nút con trái nhất làm nút con trái
- Các nút con còn lại chuyển thành các nút con phải
- Như vậy, trong cây nhị phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu. Khi đó cây nhị phân này được gọi là cây nhị phân tương đương.

Ta có thể xem ví dụ dưới đây để thấy rõ qui trình. Giả sử có cây tổng quát như hình vẽ dưới đây



Hình 5.15. Câytổng quát

Cây nhị phân tương đương sẽ như sau



Hình 5.16. Cây nhị phân tương đương

Phép duyệt cây tổng quát

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

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

2) Sự nhất quán giữa định nghĩa phép định nghĩa phép duyệt cây tổng quát với định nghĩa phép duyệt cây nhị phân. Vì cây nhị phân cũng có thể coi là cây tổng quát và ta có thể áp dụng định nghĩa phép duyệt cây tổng quát cho cây nhị phân.

Ta có thể xây dựng được định nghĩa của phép duyệt cây tổng quát T như sau

Duyệt theo thứ tự trước

a) Nếu T rỗng thì không làm gì

b) Nếu T khác rỗng thì

Thăm gốc của T

Duyệtcác cây con thứ nhất T_1 của gốc của cây T theo thứ tự trước

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

Duyệt theo thứ tự giữa

a) Nếu T rỗng thì không làm gì

b) Nếu T khác rỗng thì

Duyệtcác cây con thứ nhất T_1 của gốc của cây T theo thứ tự giữa

Thăm gốc của T

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

Duyệt theo thứ tự sau

a) Nếu T rỗng thì không làm gì

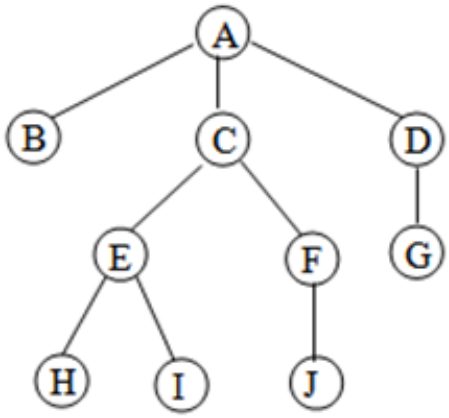
b) Nếu T khác rỗng thì

Duyệtcác cây con thứ nhất T_1 của gốc của cây T theo thứ tự sau

Duyệt các cây con còn lại T_2, T_3,...,T_n của gốc T theo thứ tự sau

Thăm gốc của T

ví dụ với cây ở hình vẽ 5.17



Hình 5.17

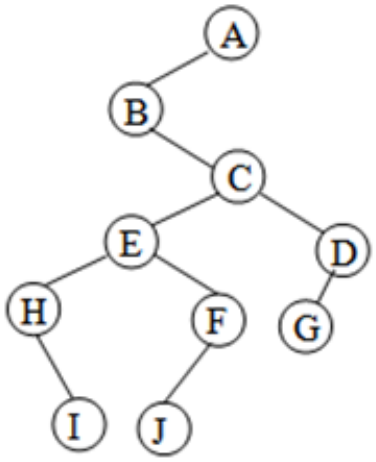
Thì dãy tên các nút được thăm sẽ là

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa : B A H E I C J F G D

Thứ tự sau : B H I E J F C G D A

Bây giờ ta dựng cây nhị phân tương đương với cây tổng quát ở hình 5.17



Hình 5.18. Cây nhị phân tương đương

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

Thứ tự trước: A B C E H I F J D G

Thứ tự giữa: B H I E J F C G D A

Thứ tự sau: I H JF E G D C B A

Nhận xét

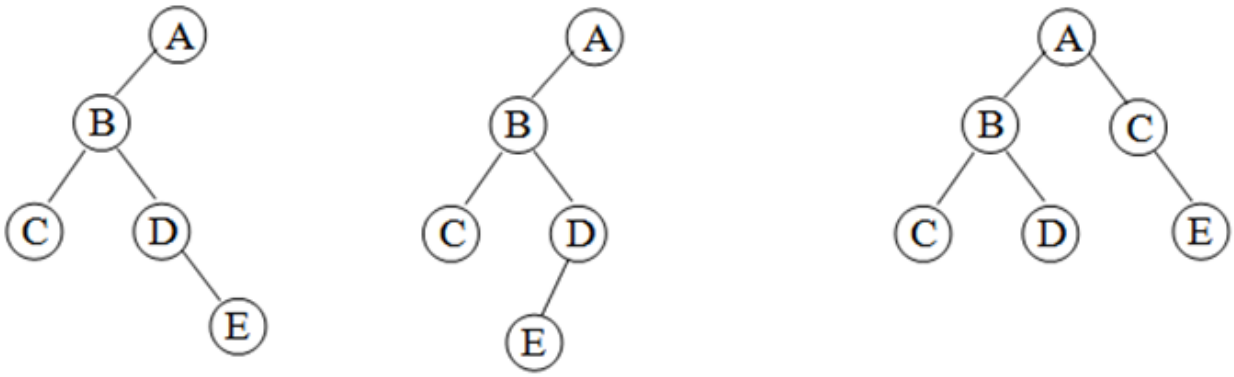
Với thứ tự trước phép duyệt cây tổng quát và phép duyệt cây nhị phân tương đương của nó đều cho một dãy tên như nhau. Phép duyệt cây tổng quát theo thứ tự sau cho dãy tên giống như dãy tên các nút được duyệt theo thứ tự giữa trong phép duyệt cây nhị phân. Còn phép duyệt cây tổng quát theo thứ tự giữa thì cho dãy tên không giống bất kỳ dãy nào đối với cây nhị phân tương đương. Do đó đối với cây tổng quát, nếu định nghĩa phép duyệt như trên người ta thường chỉ nêu hai phép duyệt theo thứ tự trước và phép duyệt theo thứ tự sau

CÂY NHỊ PHÂN

Biểu diễn cây nhị phân

Định nghĩa: *Cây nhị phân là cây mà mỗi nút có tối đa hai cây con. Đối với cây con của một nút người ta cũng phân biệt cây con trái và cây con phải.*

Như vậy cây nhị phân là cây có thứ tự.



Hình 5.3. Một số cây nhị phân

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

i) Số lượng tối đa các nút có ở mức i trên cây nhị phân là 2 i -1 (i ≥ 1)

ii) Số lượng nút tối đa trên một cây nhị phân có chiều cao h là 2 h -1(h ≥ 1)

Chứng minh

i) Sẽ được chứng minh bằng qui nạp

Bước cơ sở: với i = 1, cây nhị phân có tối đa $1 = 2^0$ nút.Vậy mệnh đề đúng với i = 1

Bước qui nạp: Giả sử kết quả đúng với mức i, nghĩa là ở mức này cây nhị phân có tối đa 2^{i-1} nút, ta chứng minh mệnh đề đúng với mức i + 1.

Theo định nghĩa cây nhị phân thì tại mỗi nút có tối đa hai cây con nên mỗi nút ở mức i có tối đa hai con. Do đó theo giả thiết qui nạp ta suy ra tại mức i+ 1 ta có

$2^{i-1} \times 2 = 2^i$ nút.

ii) Ta đã biết rằng chiều cao của cây là số mức lớn nhất có trên cây đó. Theo i) ta suy ra số nút tối đa có trên cây nhị phân với chiều cao h là :

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

Từ kết quả này có thể suy ra:

Nếu cây nhị phân có n nút thì chiều cao của no là $h = ?\log_2(n + 1)?$

(Ta qui ước : ?x? là số nguyên trên của x

[x] là số nguyên dưới của x)

1.lưu trữ kế tiếp

- Phương pháp tự nhiên nhất để biểu diễn cây nhị phân là chỉ ra đỉnh con trái và đỉnh con phải của mỗi đỉnh.

Ta có thể sử dụng một mảng để lưu trữ các đỉnh của cây nhị phân. Mỗi đỉnh của cây được biểu diễn bởi bản ghi gồm ba trường:

trường infor: mô tả thông tin gắn với mỗi đỉnh

letf : chỉ đỉnh con trái

right: chỉ đỉnh con phải.

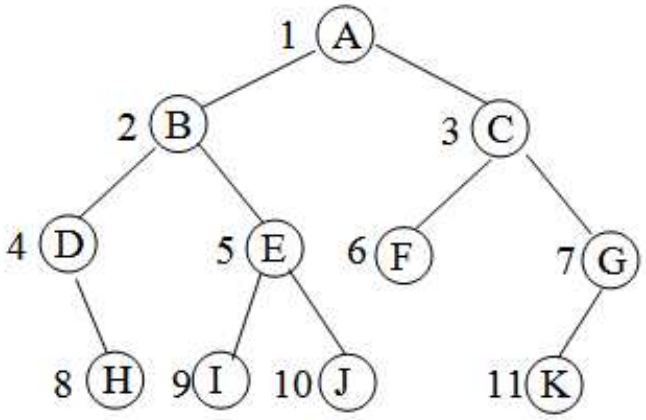
Giả sử các đỉnh của cây được được đánh số từ 1 đến max. Khi đó cấu trúc dữ liệu biểu diễn cây nhị phân được khai báo như sau:

```
const max = ...; {số thứ tự lớn nhất của nút trên cây}

type

item = ...; {kiểu dữ liệu của các nút trên cây}

Node = record

infor : item;

letf :0..max;
```

right :0..max;

end;

Tree = array[1.. max] of Node;



Hình 5.4. Một cây nhị phân

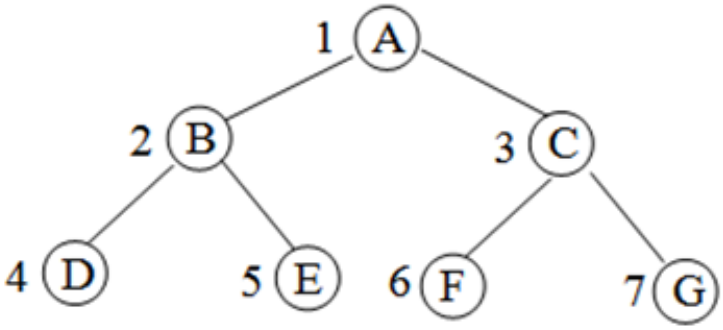
Hình 5.5. minh hoạ cấu trúc dữ liệu biểu diễn cây nhị phân trong hình 5.4

	infor	left	right
1	A	2	3
2	B	4	5
3	C	6	7
4	D	0	8
5	E	9	10
6	F	0	0
7	G	11	9
8	H	0	0
9	I	0	0
10	J	0	0
11	K	0	0

Hình 5.5. Cấu trúc dữ liệu biểu diễn cây

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

ví dụ với hình 5.6 có thể đánh số như sau:



Hình 5.6. Cây nhị phân được đánh số

Ta có nhận xét sau: con của nút thứ i là các nút thứ 2i và 2i + 1 hoặc cha của nút thứ j là [j/2]

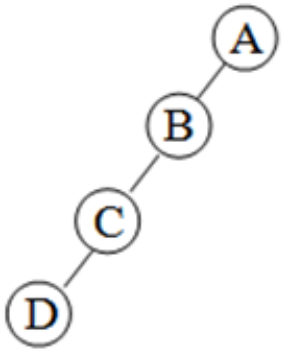
Nếu như vậy thì ta có thể lưu trữ cây này bằng một vectơ V, theo nguyên tắc: nút thứ i của cây được lưu trữ ở V[i]. Đó chính là cách lưu trữ kế tiếp đối với cây nhị phân. Với cách lưu trữ này nếu biết được địa chỉ của nút con sẽ tính được địa chỉ nút cha và ngược lại.

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

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

Nhận xét

Nếu cây nhị phân không đầy đủ thì cách lưu trữ này không thích hợp vì sẽ gây ra lãng phí bộ nhớ do có nhiều phần tử bỏ trống (ứng với cây con rỗng). Ta hãy xét cây như hình 5.7. Để lưu trữ cây này ta phải dùng mảng gồm 31 phần tử mà chỉ có 5 phần tử khác rỗng; hình ảnh lưu trữ miền nhớ của cây này như sau



Hình 5.7.Cây nhị phân đặcbiệt

A	B	∅	C	∅	∅	∅	D	∅	∅	∅	∅	∅	∅	∅	E	∅	...

(∅: chỉ chỗ trống)

Nếu cây nhị phân luôn biến động nghĩa là có phép bổ sung, loại bỏ các nút thường xuyên tác động thì cách lưu trữ này gặp phải một số nhược điểm như tốn thời gian khi phải thực hiện các thao tác này, độ cao của cây phụ thuộc vào kích thước của mảng...

2. Lưu trữ móc nối

Cách lưu trữ này khắc phục được các nhược điểm của cách lưu trữ trên đồng thời phản ánh được dạng tự nhiên của cây.

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

letf	info	right

Trường info ứng với thông tin (dữ liệu) của nútTrường left ứng với con trỏ, trỏ tới cây con trái của nút đó Trường right ứng với con trỏ, trỏ tới cây con phải của nút đó

Ta có thể khai báo như sau

type

item = ...;{kiểu dữ liệu của các nút trên cây

Tree = ^Node;

Node = record

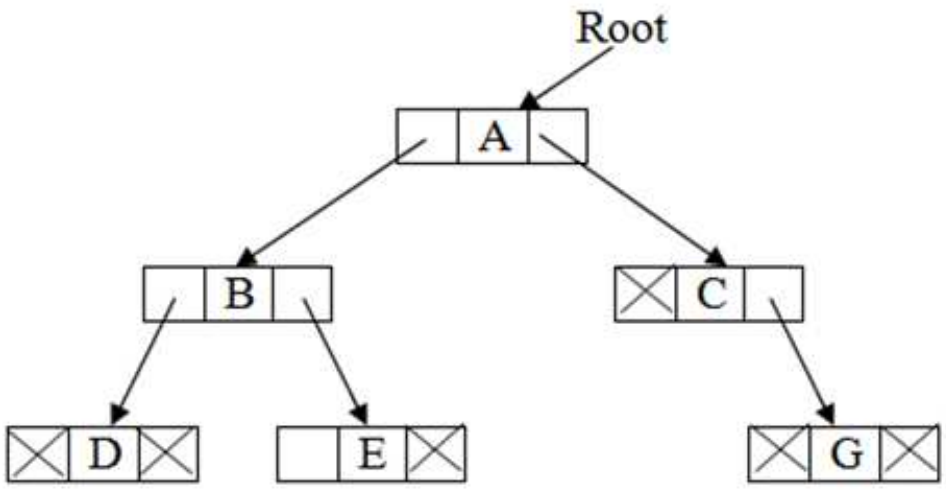
info : item;

left, right: Tree;

end;

var Root :Tree;

ví dụ: cây nhị phân hình 5.8 có dạng lưu trữ móc nối như ở hình 5.9



Hình. Cấu trúc dữ liệu biểu diễn cây

Để truy nhập vào các nút trên cây cần có một con trỏ Root, trỏ tới nút gốc của cây

Duyệt cây nhị phân

Phép xử lý các nút trên cây - mà ta gọi chung là phép thăm các nút một cách hệ thống, sao cho mỗi nút được thăm đúng một lần, gọi là phép duyệt cây. Chúng ta thường duyệt cây nhị phân theo một trong ba thứ tự: duyệt trước, duyệt giữa và duyệt sau, các phép này được định nghĩa đệ qui như sau:

Duyệt theo thứ tự trước (preorder traversal)

- Thăm gốc

- Duyệt câycon trái theo thứ trước

- Duyệt cây con phải theo thư tự trước

Duyệt theo thứ tự giữa (inorder traversal)

- Duyệt câycon trái theo thứ giữa

- Thăm gốc

- Duyệt cây con phải theo thư tự giữa

Duyệt theo thứ tự sau (postorder traversal)

- Duyệt câycon trái theo thứ sau

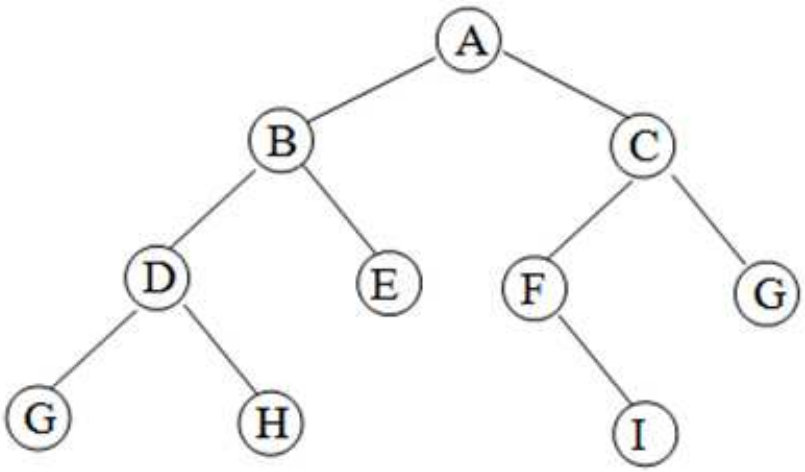
- Duyệt cây con phải theo thư tự sau

- Thăm gốc

Tương ứng với ba phép duyệt ta có ba thủ tục duyệt cây nhị phân. Sau đây là thủ tục đệ qui duyệt cây theo thứ tự trước

```
procedure Preorder (Root : Tree);

Begin

if Root <> nil then

Begin

write(Root^.info);

Preorder(Root^.left);

Preorder(Root^.right);

end;

end;
```

Một cách tương tự, ta có thể viết được các thủ tục đệ qui đi qua cây theo thứ tự giữa và theo thứ tự sau.



Với cây nhị phân ở hình vẽ này, dãy các nút được thăm trong các phép duyệt là

a) Duyệt theo thứ tự trước

A B D G H E C F I G

b) Duyệt theo thứ giữa

G D H B E A F I C G

c) Duyệt theo thứ tự sau

G H D E B I F G C A

Thực hành cài đặt cây nhị phân

THỰC HÀNH CÀI ĐẶT CÂY NHỊ PHÂN

Cài đặt cây nhị phân với các phương thức:

Khởi tạo

Thêm nút

Xóa nút

Duyệt cây

Cây nhị phân và ứng dụng

CÂY BIỂU THỨC

Cách dựng cây biểu thức

TTTH2TTTH1TH2 Đối với phép toán hai ngôi (chẳng hạn +, -, *, /) được biểu diễn bởi cây nhị phân mà gốc của nó chứa toán tử, cây con trái biểu diễn toán hạng bên trái, còn cây con bên phải biểu diễn toán hạng bên phải

(1)

Hình 5.11

a+ba + b!nn!xexpexp(x) Đối với phép toán một ngôi như "phủ định" hoặc "lấy giá trị đối", hoặc các hàm chuẩn như exp() hoặc cos()...thì cây con bên trái rỗng. Còn với các phép toán một toán hạng như phép "lấy đạo hàm" ()' hoặc hàm "giai thừa" ()! thì cây con bên phải rỗng.

<>=orabcd(a < b) or (c >= d)/a+bca/(b + c)

Hình 15.1.Một số cây biểu thức

Nhận xét

Nếu ta đuyệt cây biểu thức theo thứ tự trước thì ta được biểu thức Balan dạng tiền tố (prefix). Nếu duyệt cây nhị phân theo thứ tự sau thì ta có biểu thức Balan dạng hậu tố (postfix); còn theo thứ giữa thì ta nhận được cách viết thông thường của biểu thức (dạng trung tố).

Ví dụ

Để minh cho nhận xét này ta lấy ví dụ sau:

Cho biểu thức P = a*(b - c) + d/e

a) Hãy dựng cây biểu thức biểu diễn biểu thức trên

TH1+TH2b) Đưa ra biểu thức ở dạng tiền tố và hậu tố

Giải

a) Ta có TH1 = a*(b - c) suy ra câybiểu thức có dạng

TH2 = d/e

Xét TH1 = a*(b - c), toán hạng chưa ở dạng cơ bản ta phải phân tích để được như ở

TH3*TH4(1)

TH3 = a cây biểu thức của toán hạng này là

TH4 = b - c

b-c

Toán hạng TH4 đã ở dạng cơ bản

nên ta có ngay cây biểu thức

d/e

Cũng tương tự như vậy đối với

toán hạng TH2, cây biểu thức

tương ứng với toán hạng này là

Hình 5.13

Tổng hợp cây biểu thức của các toán hạng ta được cây biểu thức sau

*/+a-debc

Hình 15.2. Cây biểu thức

b) Bây giờ ta duyệt cây biểu thức ở hình 5.14

Duyệt theo thứ tự trước : + * a - b c / d e

Duyệt theo thứ sau: a b c - * d e / +

CÂY NHỊ PHÂN TÌM KIẾM

Cây nhị phân được sử dụng vào nhiều mục đích khác nhau. Tuy nhiên việc sử dụng cây nhị phân để lưu giữ và tìm kiếm thông tin vẫn là một trong những áp dụng quan trọng

nhất của cây nhị phân. Trong phần này chúng ta sẽ nghiên cứu một lớp cây nhị phân đặcbiệt phục vụ cho việc tìm kiếm thông tin, đó là cây nhị phân tìm kiếm.

Trong thực tế, một lớp đối tượng nào đó có thể được mô tả bởi một kiểu bản ghi, các trường của bản ghi biểu diễn các thuộc tính của đối tượng. Trong bài toán tìm kiếm thông tin, ta thường quan tâm đến một nhóm các thuộc tính nào đó của đối tượng mà thuộc tính này hoàn toàn xác định được đối tượng. Chúng ta gọi các thuộc tính này là khoá. Như vậy, khoá là một nhóm các thuộc tính của một lớp đối tượng sao cho hai đối tượng khác nhau cần phải có giá trị khác nhau trên nhóm thuộc tính đó.

Định nghĩa

Cây nhị phân tìm kiếm (CNPTK) là cây nhị phân hoặc rỗng hoặc thoả mãn đồng thời các điều kiện sau:

- Khoá của các đỉnh thuộc cây con trái nhỏ hơn khoá nút gốc
- Khoá của nút gốc nhỏ hơn khoá của các đỉnh thuộc cây con phải của của gốc
- Cây con trái và cây con phải của gốc cũng là cây nhị phân tìm kiếm

Hình 5.19 biểu diễn một cây nhị phân tìm kiếm, trong đó khoá của các đỉnh là các số nguyên.

7 24 15 2 10 20 34 55 9 12

Cài đặt cây nhị phân tìm kiếm

Mỗi nút trên cây nhị phân tìm kiếm có dạng

left	info	right

Trong đó trường Left :con trỏ chỉ tới cây con trái

Right :con trỏ chỉ tới cây con phải

Info : chứa thông tin của nút

Type

keytype =...; {kiểu dữ liệu của mỗi nút}

Tree = ^ node;

Node = record

```
Info : keytype;

Left, Right : Tree;

end;

Var T : Tree;
```

Các thao tác cơ bản trên cây nhị phân tìm kiếm

1. Tìm kiếm

Tìm kiếm trên cây là một trong các phép toán quan trọng nhất đối với cây nhị phân tìm kiếm . Ta xét bài toán sau

Bài toán: Giả sử mỗi đỉnh trên cây nhị phân tìm kiếm là một bản ghi, biến con trỏ Root chỉ tới gốc của cây và x là khoá cho trước. Vấn đề đặt ra là, tìm xem trên cây có chứa đỉnh với khoá x hay không.

Giải thuật đệ qui

```
Procedure search (Root : Tree; x : keytype; var p : tree);

{ Nếu tìm thấy thì p chỉ tới nút có trường khoá bằng x, ngược lại p=nil}

Begin

p:=root;

if p <> nil then

if x < p^.info then search (p^.left, x, p)

else

if x > p^.info then search (p^.Right, x, p)

End;
```

Giải thuật lặp

Trong thủ tục này, ta sẽ sử dụng biến địa phương found có kiểu boolean để điều khiển vòng lặp, nó có giá trị ban đầu là false. Nếu tìm kiếm thành công thì found nhận giá trị

true, vòng lặp kết thúc và đồng thời p trỏ đến nút có trường khoá bằng x. Còn nếu không tìm thấy thì giá trị của found vẫn là false và giá trị của p lànil

```
Procedure search (Root : Tree; x:keytype; var p : Tree);

Var Found : boolean;

Begin

Found:=False;

P:=Root;

while (p<>nil) and( not Found ) do

if x > p^.info then p := p^.right

else

if x < p^.info then p := p^.left

else Found = True;

End;
```

1. Đi qua CNPTK

Như ta đã biết CNPTK cũng là cây nhị phân nên các phép duyệt trên cây nhị phân cũng vẫn đúng trên CNPTK. Chỉ có lưu ý nhỏ là khi duyệt theo thứ tự giữa thì được dãy khoá theo thứ tự tăng dần.

c) Chèn một nút vào CNPTK

Việc thêm một một nút có trường khoá bằng x vào cây phải đảm bảo điều kiện ràng buộc của CNPTK. Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm vào một nút lá sẽ là tiện lợi nhất do ta có thể thực hiện quá trình tương tự như thao tác tìm kiếm. Khi kết thúc việc tìm kiếm cũng chính là lúc tìm được chỗ cần chèn.

Giải thuật đệ quy

```
Procedure Insert (var Root :Tree; x: kkeytype);

Var p : tree;
```

```
Begin

New(p);{Tạo ra nút mới}

P^.info := x;

if root=nil then

begin

Root :=p;

P^.left:= nil;

P^.right:= nil;

End

Else

with Root^ do

if x> info then insert (Right, x)

else if x < info then insert (left, x);

End;
```

Giải thuật lặp

Trong thủ tục này ta sử dụng biến con trỏ địa phương q chạy trên các đỉnh của cây bắt đầu từ gốc. Khi đang ở một đỉnh nào đó, q sẽ xuống đỉnh con trái (phải) tuỳ theo khoá ở đỉnh lớn hơn (nhỏ hơn) khoá x.

Ở tại một đỉnh nào đó khi p muốn xuống đỉnh con trái (phải) thì phải kiểm tra xem đỉnh này có đỉnh con trái (phải) không. Nếu có thì tiếp tục xuống, ngược lại thì treo đỉnh mới vào bên trái (phải) đỉnh đó. Điều kiện q = nil sẽ kết thúc vòng lặp. Quá trình này được lại lặp khi có đỉnh mới được chèn vào.

```
procedure Insert (var Root : Tree; x: keytype)

var p, q : tree;

begin
```

```
New(p);
P^.info :=x;
if Root = nil then
Begin
Root:=p;
P^.left:= nil;
P^.right:= nil;
End
Else
Begin
q:=Root;
while q <> nil do
if x < q^.info then
if q^.left <> nil then q := q^.left
else begin
q^.left :=p;
p := nil;
end
else if x > q^.info then
if q^.right <> nil then q:=q^.right
else begin
q^.right :=p;
```

q =nil;

end;

end;

end;

Nhận xét: Để dựng được CNPTK ứng với một dãy khoá đưa vào bằng cách liên tục bổ các nút ứng với từng khoá, bắt đầu từ cây rỗng. Ban đầu phải dựng lên cây với nút gốc là khoá đầu tiên sau đó đối với các khoá tiếp theo, tìm trên cây không có thì bổ sung vào.

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

thì cây nhị phân tìm kiếm dựng được sẽ có dạng ở hình 5.20

23744211366594589987

Hình 5.20. Một cây nhị phân tìm kiếm

d)Loại bỏ nút trên cây nhị phân tìm kiếm

Đối lập với phép toán chèn vào là phép toán loại bỏ. Chúng ta cần phải loại bỏ khỏi CNPTK một đỉnh có khoá x (ta gọi tắt là nút x) cho trước, sao cho việc huỷ một nút ra khỏi cây cũng phải bảo đảm điều kiện ràng buộc của CNPTK.

Có ba trường hợp khi huỷ một nút x có thể xảy ra:

- X là nút lá
- X là nút nửa lá (chỉ có một con trái hoặc con phải)
- X có đủ hai con (trường hợp tổng quát)

Trường hợp thứ nhất: chỉ đơn giản huỷ nút x vì nó không liên quan đến phần tử nào khác.

320TCây trước khi xoá20TCây sau khi xoá

Hình 5.21

Trường hợp thứ hai: Trước khi xoá nút x cần móc nối cha của x với nút con (nút con trái hoặc nút con phải) của nó

$T_2$2010318T_1

$T_2$201025318T_1

a) Cây trước khi xoáb) Cây sau khi xoá đỉnh (25)

Trường hợp tổng quát: khi nút bị loại bỏ có cả cây con trái và cây con phải, thì nút thay thế nó hoặc là nút ứng với khoá nhỏ hơn ngay sát trước nó (nút cực phải của cây con trái nó) hoặc nút ứng với khoá lớn hơn ngay sát sau nó (nút cực trái của cây con phải nó). Như vậy ta sẽ phải thay đổi một số mối nối ở các nút:

- Nút cha của nút bị loại bỏ
- Nút được chọn làm nút thay thế
- Nút cha của nút được chọn làm nút thay thế

T_2b) Cây sau khi xoá đỉnh 201810253T_1T_2a) Cây trước khi xoá đỉnh 20201025318T_1

Trong ví dụ này ta chọn nút thay thế nút bị xoá là nút cực phải của cây con trái (nút 18).

T_5ABT_4CE$T_2T_3T_1$RQTSa) Cây trước khi xoá nút trỏ bởi QT_5AEB$T_4C T_2T_1$RQTST_3b) Cây sau khi xoá nút trỏ bởi Q Sau đây là giải thuật thực hiện việc loại bỏ một nút trỏ bởi Q. Ban đầu Q chính là nối trái hoặc nối phải của một nút R trên cây nhị phân tìm kiếm, mà ta giả sử đã biết rồi.

```
procedure Del (var Q: Tree); {xoá nút trỏ bởi Q}

var T, S : Tree;

begin

P := Q; {Xử lý trường hợp nút lá và nút nửa lá}

if P^.left = nil then

Begin

Q:=P^.right ;{R^.left := P^.right}

Dispose(P);

end

else

if P^.right = nil then
```

```
begin
Q :=P^.left;
Dispore (P);
end
else { Xử lý trường hợp tổng quát}
begin
T := P^.left;
if T^.right = nil then
begin
T^.right := P^.right;
Q := T;
Dispose (P);
end
else
begin
S := T^.right; {Tìm nút thay thế, là nút cực phải của cây }
while S^.right <> nil do
begin
T := S;
S := T^.right;
end;
```

S^.right := P^.right;

T^.right := S^.left;

S^.left := P^.left;

Q:=S;

Dispose(p);

end;

end;

end;

Thủ tục xoá trường dữ liệu bằng X

- Tìm đến nút có trường dữ liệu bằng X
- Áp dụng thủ tục Del để xoá

Sau đây chúng ta sẽ viết thủ tục loại khỏi cây gốc Root đỉnh có khoá x cho trước. Đó là thủ tục đệ qui, nó tìm ra đỉnh có khoá x, sau đó áp dụng thủ tục Del để loại đỉnh đó ra khỏi cây.

procedure Delete (var Root :Tree ; x : keytype);

begin

if Root <> nil then

if x < Root^.info then Delete (Root^.left, x)

else if x > Root^.info then Delete (Root^.right, x)

else Del(Root);

end;

Nhận xét: Việc huỷ toàn bộ cây có thể thực hiện thông qua thao tác duyệt cây theo thứ sau. Nghĩa là ta sẽ huỷ cây con trái, cây con phải rồi mới huỷ nút gốc.

procedure RemoveTree (var Root: Tree);

```
begin
if Root <> nil then
begin
RemoveTree(Root^.left);
RemoveTree(Root^.right);
Dispose (Root);
end;
end;
```

Thực hành cài đặt cây nhị phân tìm kiếm

THỰC HÀNH CÀI ĐẶT CÂY NHỊ PHÂN TÌM KIẾM

Cài đặt cây nhị phân tìm kiếm với các phương thức:

Khởi tạo

Thêm nút

Xóa nút

Tìm kiếm

Kiểm tra thực hành và tổng kết module

KIỂM TRA THỰC HÀNH

Chia nhóm và tiến hành kiểm tra, đánh giá

TỔNG KẾT MODUL

Trao đổi về bài tập, bài tập thực hành

Hệ thống kiến thức modul đã học

Mở rộng kiến thức môn học

Tham gia đóng góp

Tài liệu: Cấu trúc dữ liệu và giải thuật

Biên tập bởi: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://voer.edu.vn/c/60bbf7d3

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Giải thuật và cấu trúc dữ liệu

Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://www.voer.edu.vn/m/8b6180c7

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Phân tích và thiết kế bài toán

Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://www.voer.edu.vn/m/32569f6a

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Phân tích thời gian thực hiện thuật toán

Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://www.voer.edu.vn/m/a7204439

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Mảng và dánh sách

Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://www.voer.edu.vn/m/9461a675

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Danh sách nối đơn (Singlely Linked List)

Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://www.voer.edu.vn/m/578aa05e

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Thực hành cài đặt danh sách nối đơn

Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://www.voer.edu.vn/m/1038f94e

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Danh sách tuyến tính ngăn xếp (Stack)
Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên
URL: http://www.voer.edu.vn/m/a208ce0f
Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Danh sách tuyến tính kiểu hàng đợi
Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên
URL: http://www.voer.edu.vn/m/387652b5
Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Thực hành cái đặt danh sách kiểu hàng đợi
Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên
URL: http://www.voer.edu.vn/m/27f6f7c3
Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Danh sách nối vòng và nối kép
Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên
URL: http://www.voer.edu.vn/m/556b338a
Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Thực hành cài đặt danh sách liên kết kép
Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên
URL: http://www.voer.edu.vn/m/b1a9a363
Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Kiểu dữ liệu cây
Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên
URL: http://www.voer.edu.vn/m/2ff63fcb
Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Thực hành cài đặt cây nhị phân
Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên
URL: http://www.voer.edu.vn/m/a5c4789c
Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Cây nhị phân và ứng dụng

Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://www.voer.edu.vn/m/58472201

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Thực hành cài đặt cây nhị phân tìm kiếm

Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://www.voer.edu.vn/m/c00632bd

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Module: Kiểm tra thực hành và tổng kết module

Các tác giả: Khoa CNTT ĐHSP KT Hưng Yên

URL: http://www.voer.edu.vn/m/a852a6eb

Giấy phép: http://creativecommons.org/licenses/by/3.0/

Chương trình Thư viện Học liệu Mở Việt Nam

Chương trình Thư viện Học liệu Mở Việt Nam (Vietnam Open Educational Resources – VOER) được hỗ trợ bởi Quỹ Việt Nam. Mục tiêu của chương trình là xây dựng kho Tài nguyên giáo dục Mở miễn phí của người Việt và cho người Việt, có nội dung phong phú. Các nội dung đều tuân thủ Giấy phép Creative Commons Attribution (CC-by) 4.0 do đó các nội dung đều có thể được sử dụng, tái sử dụng và truy nhập miễn phí trước hết trong trong môi trường giảng dạy, học tập và nghiên cứu sau đó cho toàn xã hội.

Với sự hỗ trợ của Quỹ Việt Nam, Thư viện Học liệu Mở Việt Nam (VOER) đã trở thành một cổng thông tin chính cho các sinh viên và giảng viên trong và ngoài Việt Nam. Mỗi ngày có hàng chục nghìn lượt truy cập VOER (www.voer.edu.vn) để nghiên cứu, học tập và tải tài liệu giảng dạy về. Với hàng chục nghìn module kiến thức từ hàng nghìn tác giả khác nhau đóng góp, Thư Viện Học liệu Mở Việt Nam là một kho tàng tài liệu khổng lồ, nội dung phong phú phục vụ cho tất cả các nhu cầu học tập, nghiên cứu của độc giả.

Nguồn tài liệu mở phong phú có trên VOER có được là do sự chia sẻ tự nguyện của các tác giả trong và ngoài nước. Quá trình chia sẻ tài liệu trên VOER trở lên dễ dàng như đếm 1, 2, 3 nhờ vào sức mạnh của nền tảng Hanoi Spring.

Hanoi Spring là một nền tảng công nghệ tiên tiến được thiết kế cho phép công chúng dễ dàng chia sẻ tài liệu giảng dạy, học tập cũng như chủ động phát triển chương trình giảng dạy dựa trên khái niệm về học liệu mở (OCW) và tài nguyên giáo dục mở (OER) . Khái niệm chia sẻ tri thức có tính cách mạng đã được khởi xướng và phát triển tiên phong bởi Đại học MIT và Đại học Rice Hoa Kỳ trong vòng một thập kỷ qua. Kể từ đó, phong trào Tài nguyên Giáo dục Mở đã phát triển nhanh chóng, được UNESCO hỗ trợ và được chấp nhận như một chương trình chính thức ở nhiều nước trên thế giới.