TRƯỜNG CAO ĐẲNG NGHỀ CÔNG NGHIỆP HÀ NỘI
Chủ biên: Vũ Thị Kim Phượng
Đồng tác giả: Nguyễn Thị Nhung

GIÁO TRÌNH

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

(Lưu hành nội bộ)

Hà Nội năm 2012

Tuyên bố bản quyền

Giáo trình này sử dụng làm tài liệu giảng dạy nội bộ trong trường cao đẳng nghề Công nghiệp Hà Nội

Trường Cao đẳng nghề Công nghiệp Hà Nội không sử dụng và không cho phép bất kỳ cá nhân hay tổ chức nào sử dụng giáo trình này với mục đích kinh doanh.

Mọi trích dẫn, sử dụng giáo trình này với mục đích khác hay ở nơi khác đều phải được sự đồng ý bằng văn bản của trường Cao đẳng nghề Công nghiệp Hà Nội

LỜI NÓI ĐẦU

Giáo trình "Cấu trúc dữ liệu và giải thuật" biên soạn dựa theo đề cương chương trình môn học *Cấu trúc dữ liệu và giải thuật* thuộc chương trình đào tạo Cao đẳng nghề *Quản trị mạng* của trường Cao đẳng nghề Công nghiệp Hà nội, ban hành năm 2011, với số tiết là 90h.

Giáo trình gồm 7 chương, đề cập đến những kiến thức cơ bản về cấu trúc dữ liệu và các giải thuật có liên quan. Từng chương trong giáo trình cũng cố gắng gắn kết và phát triển nội dung có liên quan ở các môn học trước hay ở các chương trong giáo trình với nhau, giúp sinh viên nâng cao về kỹ thuật lập trình, về chọn cấu trúc dữ liệu phù hợp và xây dựng các giải thuật giải các bài toán cơ bản.

Giáo trình cố gắng trình bày để phục vụ cho đối tượng sinh viên năm thứ hai vừa học qua một ngôn ngữ lập trình. Trong mỗi chương đều có ví dụ diễn giải làm rõ những định nghĩa, khái niệm và đặc biệt với mỗi giải thuật đều có mô tả và cài đặt giải thuật hoặc ví dụ áp dụng. Cuối mỗi chương là những câu hỏi về lý thuyết và bài tập ở mức độ dễ, vừa, giúp sinh viên củng cố kiến thức.

Cùng với giáo trình này, giáo viên có thể yêu cầu sinh viên tự đọc một số phần, như vậy sẽ có nhiều thời gian giảng kỹ những phần chính, khó hoặc luyện được nhiều bài tập. Bên cạnh đó cũng giúp sinh viên rèn luyện khả năng tự học của bản thân.

Nhóm tác giả chân thành cảm ơn những đồng nghiệp trong khoa Công nghệ thông tin trường Cao đẳng nghề Công nghiệp Hà nội đã tham gia xây dựng đề cương chi tiết giáo trình, đọc bản thảo và đóng góp những ý kiến quý báu.

Nhóm tác giả mong muốn nhận được những ý kiến đóng góp của bạn đọc để nâng cao chất lượng giáo trình cho lần tái bản sau.

Mọi ý kiến đóng góp xin gửi về:

Vũ Thị Kim Phượng

Email: vkphuong2010@gmail.com

Hà Nội, ngày..... tháng....năm 2012

Tham gia biên soạn giáo trình

1. Vũ Thị Kim Phượng – Chủ biên
2. Nguyễn Thị Nhung – Thành viên

MỤC LỤC

MỤC TIÊU:

Kiến thức:

Trình bày được các khái niệm về cấu trúc dữ liệu và giải thuật, kiểu dữ liệu, kiểu dữ liệu trừu tượng (danh sách, cây, đồ thị).

Trình bày được các phép toán cơ bản tương ứng với các cấu trúc dữ liệu và các giải thuật.

Kỹ năng:

Biết cách tổ chức dữ liệu hợp lý, khoa học cho một chương trình đơn giản.

Biết áp dụng thuật toán hợp lý đối với cấu trúc dữ liệu tương ứng để giải quyết bài toán trên máy tính.

Áp dụng được các phương pháp sắp xếp, tìm kiếm cơ bản trong các bài toán khi cần.

NỘI DUNG:

Số TT	Tên chương, mục	Thời gian			
		Tổng số	Lý thuyết	Thực hành	Kiểm tra* (LT hoặcTH)
I	**Tổng quan về Cấu trúc dữ liệu và giải thuật**	**6**	**4**	**2**	**0**
	Khái niệm cấu trúc dữ liệu và giải thuật. Mối quan hệ giữa CTDL và giải thuật.	2	1	1	
	Các kiểu dữ liệu cơ bản	0.5	0.5	0	
	Các kiểu dữ liệu có cấu trúc	0.5	0.5	0	
	Các kiểu dữ liệu trừu tượng	1	1	0	
	Giải thuật và đánh giá độ phức tạp của giải thuật.	2	1	1	

II	**Đệ qui và giải thuật đệ qui**	**6**	**3**	**2**	**1**
	Khái niệm đệ qui	0.5	0.5	0	
	Giải thuật đệ qui và chương trình đệ qui	0.5	0.5	0	
	Các bài toán đệ qui căn bản	5	2	2	1
III	**Danh sách**	**30**	**15**	**14**	**1**
	Danh sách và các phép toán cơ bản trên danh sách	2	2	0	0
	Cài đặt danh sách theo cấu trúc mảng	4	2	2	0
	Cài đặt danh sách theo cấu trúc danh sách liên kết (đơn, kép)	12	6	6	0
	Cài đặt danh sách theo các cấu trúc đặc biệt (ngăn xếp, hàng đợi)	12	5	6	1
IV	**Các phương pháp sắp xếp cơ bản**	**24**	**12**	**11**	**1**
	Định nghĩa bài toán sắp xếp.	1	1	0	0
	Phương pháp chọn (Selection sort).	4	2	2	0
	Phương pháp chèn (Insertion sort).	4	2	2	0
	Phương pháp đổi chỗ (Interchange sort).	4	2	2	0
	Phương pháp nổi bọt (Bubble sort).	4	2	2	0
	Phương pháp sắp xếp nhanh (Quick sort).	7	3	3	1
V	**Tìm kiếm.**	**6**	**2**	**3**	**1**
	Tìm kiếm tuyến tính.	2	1	1	0
	Tìm kiếm nhị phân.	4	1	2	1
VI	**Cây.**	**10**	**5**	**4**	**1**
	Khái niệm về cây và cây nhị phân.	2	2	0	0
	Biểu diễn cây nhị phân và cây tổng quát.	4	2	2	0
	Bài toán duyệt cây nhị phân.	4	1	2	1
VII	**Đồ thị.**	**8**	**4**	**4**	**0**

	Khái niệm về đồ thị. Biểu diễn đồ thị. Các phép duyệt đồ thị.	2 2 4	1 1 2	1 1 2	
	Cộng	**90**	**45**	**40**	**5**

* Ghi chú: *Thời gian kiểm tra lý thuyết được tính vào giờ lý thuyết, kiểm tra thực hành được tính bằng giờ thực hành.*

Yêu cầu về đánh giá hoàn thành môn học:

- Về kiến thức: Đánh giá kiến thức qua bài kiểm tra viết, trắc nghiệm

  đạt được các yêu cầu sau:

  - Hiểu được mối quan hệ giữa cấu trúc dữ liệu và giải thuật.
  - Phân tích được các kiểu dữ liệu, giải thuật, sự kết hợp chúng để

tạo thành một chương trình máy tính.

  - Biết cách tổ chức dữ liệu hợp lý, khoa học cho một chương trình

đơn giản.

  - Biết áp dụng thuật toán hợp lý đối với cấu trúc dữ liệu tương

thích để giải quyết bài toán thực tế.

  - Biết và áp dụng được các phương pháp sắp xếp, tìm kiếm đơn

giản.

- Về kỹ năng:
  - Đánh giá kỹ năng thực hành của sinh viên:
  - Dùng ngôn ngữ lập trình bất kỳ nào đó thể hiện trên máy tính các

bài toán cần kiểm nghiệm về: đệ qui, danh sách, cây, đồ thị, sắp xếp, tìm kiếm...

- Về thái độ: Cẩn thận, tỉ mỉ, thao tác chuẩn xác, tự giác trong học tập.

CHƯƠNG 1
TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Mục tiêu:

- Trình bày được khái niệm vê`cấú truć dư~liệu, giải thuật, mối quan
  hệ giữa cấu trúc dữ liệu và giải thuật. Đánh giá được độ phức tạp của giải thuật.
- Trình bày được các kiểu dữ liệu cơ bản, các kiểu dữ liệu cấu trúc
  và kiểu dữ liệu trừu tượng.

1. Khái niệm cấu trúc dữ liệu và giải thuật, cấu trúc lưu trữ và cấu trúc dữ liệu.

1.1. Khái niệm cấu trúc dữ liệu và giải thuật

Algorithms + Data Structures = Programs

" Giải thuật + Cấu trúc dữ liệu = Chương trình "

Đo´la`nhan đê`cuốn sách được xuất bản năm 1975, bởi nha`khoa học maý tính Thụy sy~Niklaus Wirth Emil, cuốn sách đa~được công nhận rộng raĩ va`vẫn coǹ hữu dụng đến ngaỳ nay. Năm vưng cấú truć dư~liệu va`giải thuật la`cơ sở giuṕ sinh viên có khả năng đi sâu thêm vào các môn học chuyên ngành.

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

Dữ liệu (Data): Là đối tượng của giải thuật để khi tác động bởi các thao tác của giải thuật ta nhận được kết quả mong muốn.

Giải thuật chỉ phản ánh các phép xử lí, còn đối tượng để xử lí trên MTĐT, chính là dữ liệu (data) chúng biểu diễn các thông tin cần thiết cho bài toán: Các dữ kiện đưa vào, các kết quả trung gian va`kêt́ quả đâù ra của bai toań.

Ví dụ 1.1: Chương trình tìm ước chung lớn nhất của 2 số nguyên dương a và b.

Dữ kiện đưa vào (input): ***a, b*** nguyên dương

Phép xử lý (Process) : Dựa theo thuật toán Euclid, thuật toán nổi tiếng nhất có từ thời cổ đại.

Bước 1: Tìm ***r***, là phần dư của phép chia a cho b.

Bước 2:

Nếu ***r*** = 0.

Thì: Gán giá trị của ***b*** cho ***E*** (E←b) và dừng lại

Nếu ngược lại (***r*** ≠ 0).

Thì: Gán giá trị ***b*** cho ***a*** (a←b).

Gán giá trị ***r*** cho ***b*** (b←r) và quay lại bước 1.

Kết quả ra (Output): ***E,*** Ước chung lớn nhất của ***a*** và ***b.***

Cấu trúc dữ liệu (Data Structures): Cách sắp xếp, tổ chức dữ liệu, tạo quan hệ nội tại giữa các phần tử dữ liệu, tạo thuận lợi cho các phép xử lý và nâng cao hiệu quả của chúng.

Bản thân các phần tử của dữ liệu có mối quan hệ với nhau, ngoài ra nếu lại biết "tổ chức" theo các cấu trúc thích hợp thì việc thực hiện các phép xử lí trên các dữ liệu càng thuận lợi hơn, đạt hiệu quả cao hơn.

Ví dụ 1.2: Viết chương trình thực hiện công việc sau:

a. Nhập vào từ bàn phím ***n*** số nguyên bất kỳ

b. Tính tổng các số vừa nhập và đưa kết quả ra màn hình

Dữ kiện đưa vào (input): ***so***, ***tong*** là 2 biến số nguyên và ***n*** là số lượng số nguyên.

Phép xử lý (Process) : Thực hiện ***n*** lần công việc sau:

- Nhập giá trị cho biến ***so***
- Cộng giá trị biến ***so*** vào biến ***tong***

Kết quả ra (Output): ***tong,*** tổng ***n*** số nguyên vừa nhập

Với 2 yêu cầu (a, b) của bài toán, ta chỉ cần một biến ***so*** để lưu giá trị từng số nguyên nhập vào và cộng gộp dần giá trị ngay vào một biến ***tong.***

Ví dụ 1.3: Viết chương trình thực hiện công việc sau:

a. Nhập vào từ bàn phím n số nguyên bất kỳ.

b. Tính tổng các số vừa nhập và đưa kết quả ra màn hình.

c. Săp xêp daỹ sô´theo chiều tăng dần va`đưa daỹ đa~săp xêp ra màn hình.

Dư~kiện đưa vaò (input):

tong la`biến sô´nguyên.

M la`môt biến mảng kiểu phần tử là kiểu sô´nguyên.

n là số lượng số nguyên.

Phép xử ly´(Process:

Bươć 1: Thực hiện ***n*** lần công việc sau:

- Nhập gia´trị cho tưng phần tử mảng ***M[i]***

Bươć 2: Thực hiện ***n*** lần công việc sau:

- Cộng gia´trị tưng biến ***M[i]*** vaò biến ***tong***

Bươć 3: Thực hiện săp xêp daỹ sô´theo chiêù tăng dần

Kêt quả ra (Output):

- ***tong***, tổng ***n*** sô´nguyên vưà nhập
- ***M***, Daỹ sô´đa~săp xêp theo chiều tăng dần

Ơ vi´dụ nay co´thêm yêu câù thư´3 (c), ta không thể dưng một biến ***so*****,** hay khai baó ***n*** biến ***so*** được (vi`không biêt ***n*** la`bao nhiêu 10, 100 hay 10000,…). Phải cần một biến mảng ***M*** để lưu gia´trị ***n*** sô´nguyên nhập vaò tư`bàn phím va`daỹ sô´ nguyên đa~đươc săp xêp. (*nhiều ngôn ngư~ lâp trình đều định nghĩa săn kiểu dư~liệu mảng (array): gôm một tập hợp hưu hạn cac phần tử co´cưng kiểu dư~liệu, ta chỉ cần khai baó tên kiểu mảng, sô´lượng phần tử va`kiểu dư~liệu của phần tử khi cần sử dụng*.)

So sanh 2 vi´dụ trên ta nhận thấy co´sự khac biệt sau :

Ví dụ	**Dữ kiện đưa vào**	**Phép xử lý**	**Kết quả đưa ra**
Ví dụ 1.2	Chỉ cần một biến ***so*** để lưu giư~ tưng sô´nguyên	Việc tinh tổng được thực hiện ngay sau mỗi lần nhập sô´nguyên	Gia´trị biến ***tong***
Ví dụ 1.3	Phải cần biến ***mảng M*** để lưu	Co´thể tach riêng việc tinh tổng sau	Gia´trị biến ***tong*** va` biến ***mảng M*** chứa ***n***

	giữ ***n*** số nguyên	khi nhập giá trị cho ***n*** số nguyên	số nguyên đã được sắp xếp tăng dần

Tóm lại, giữa cấu trúc dữ liệu và giải thuật có mối quan hệ mật thiết, không thể nói tới giải thuật mà không nghĩ tới: *Giải thuật đó được tác động trên dữ liệu nào,* còn khi xét tới dữ liệu thì cũng phải hiểu: *Dữ liệu ấy cần được tác động bởi giải thuật gì để đưa tới kết quả mong muốn.* Với một cấu trúc dữ liệu đã chọn ta sẽ có giải thuật xử lý tương ứng. Cấu trúc dữ liệu thay đổi, giải thuật cũng có thể thay đổi theo.

1.2. Cấu trúc dữ liệu và cấu trúc lưu trữ

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

Phân biệt lưu trữ trong và lưu trữ ngoài:

Lưu trữ trong: Là lưu trữ ở bộ nhớ trong.

Lưu trữ ngoài: Là lưu trữ ở bộ nhớ ngoài (đĩa từ, đĩa quang,.....).

Ví dụ 1.4: CTDL kiểu mảng và **Stack** cùng được lưu trữ trong bộ nhớ bởi vectơ lưu trữ

a[0]	**A[1]**	**a[2]**	**....**	**.....**	**a[n-1]**

a[n-1]
....
....
a[1]
a[0]

Đỉnh

Đáy

2. Cấu trúc dữ liệu

Trong môĩ bài toán, Lựa chọn một CTDL thích hợp để tổ chức dữ liệu vào và trên cơ sở đó xây dựng được giải thuật xử ly´hữu hiệu đưa tới kết quả mong muốn cho bài toán, đó là một khâu rất quan trọng. Muốn vậy cần nắm vững đặc điểm và các phép toán cơ bản của từng kiểu dữ liệu được sử dụng trong mỗi ngôn ngữ lập trình là yêu cầu cần thiết.

2.1. Các kiểu dữ liệu cơ bản

Các loại dữ liệu cơ bản là các loại dữ liệu đơn giản, cơ sở. Chúng thường là các giá trị vô hướng như các số nguyên, số thực, các ký tự, các giá trị logic ... Các loại dữ liệu này, do tính thông dụng và đơn giản của mình, thường được các ngôn ngữ lập trình (NNLT) cấp cao xây dựng sẵn như một thành phần của ngôn ngữ để giảm nhẹ công việc cho người lập trình. Thông thường, các kiểu dữ liệu cơ bản bao gồm:

- *Kiểu có thứ tự rời rạc:* số nguyên, ký tự, logic , liệt kê, miền con …
- *Kiểu không rời rạc*: số thực.

Ví dụ: Các kiểu dữ liệu định sẵn trong C gồm:

Tên kiểu	Sô´bytes	Miền giá trị	Ghi chú
Char	1	-128 đến 127	Có thể dùng như số nguyên 1 byte có dấu hoặc kiểu ký tự
unsigned char	1	0 đến 255	Số nguyên 1 byte không dấu
Enum	2	**-32,768** đến **32,767**	
Int	2	-32738 đến 32767	
unsigned int	2	0 đến 65335	Có thể gọi tắt là unsigned
Long	4	-2^{32} đến 2^{31} -1	
unsigned long	4	0 đến 2^{32}-1	
Float	4	3.4E-38 ÷ 3.4E38	Giới hạn chỉ trị tuyệt đối.Các giá trị <3.4E-38 được coi = 0. Tuy nhiên kiểu float chỉ có 7 chữ số có nghĩa.
Double	8	1.7E-308 ÷ 1.7E308	
long double	10	3.4E-4932 ÷ 1.1E4932	

2.2. Các kiểu dữ liệu cấu trúc.

Đó là CTDL tiền định rất hay dùng đã được cài đặt sẵn trong các ngôn ngữ lập trình, người lập trình chỉ việc dùng như: Tập hợp, mảng, bản ghi, tệp,...và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới. *(Khi nghiên cứu đến một ngôn ngữ nào đó cần phải nghiên cứu kỹ cac kiểu dữ liệu câu truc của nó.)*

a. *Kiểu tập hợp* : Một tập hợp bao gồm một số các đối tượng nào đó có

cùng bản chất, được mô tả bởi cùng một kiểu, kiểu này là kiểu cơ bản (kiểu vô hướng đếm được hay đoạn con, liệt kê), không được là kiểu số thực. Các đối tượng này được gọi là các phần tử của tập hợp. Số lượng phần tử của tập hợp thông thường là từ 0 (gọi là tập rỗng) đến tối đa là 255 phần tử.

b. *Kiểu Mảng* : Là một tập hợp gồm một số cố định các phần tử có cùng kiểu dữ liệu. Mỗi phần tử của mảng ngoài giá trị còn được đặc trưng bởi chỉ số (Index) thể hiện thứ tự của phần tử đó trong mảng (Vectơ là mảng 1 chiều, mỗi phần tử ai của nó ứng với một chỉ số i. Ma trận là mảng 2 chiều mỗi phần tử aij của nó ứng với 2 chỉ số i và j,..).

c. *Kiểu bản ghi (kiểu cấu trúc):* là một tập hợp các phần tử dữ liệu (field), mỗi phần tử dữ liệu có thể được mô tả bởi một kiểu dữ liệu khác nhau nhưng có liên kết với nhau, dùng để mô tả một đối tượng (Record).

d. *Tệp tin (File):* Là một tập hợp các dữ liệu có liên quan với nhau và có cùng kiểu dữ liệu được nhóm lại tạo thành một dãy. Chúng thường được chứa trong một thiết bị nhớ ngoài của máy tính với một cái tên nào đó.

Ví dụ 1.5 : Để mô tả một đối tượng sinh viên, cần quan tâm đến các thông tin sau:

- Mã sinh viên: chuỗi ký tự.
- Tên sinh viên: chuỗi ký tự.
- Ngày sinh: kiểu ngày tháng.
- Điểm thi: số thực.

Các kiểu dữ liệu cơ sở cho phép mô tả một số thông tin như :

float Diemthi;

Các thông tin khác đòi hỏi phải sử dụng các kiểu có cấu trúc như :

```
char masv[15];
char tensv[25];
```

Để thể hiện thông tin về ngày tháng năm sinh cần phải xây dựng một kiểu bản ghi.

```
typedef struct Date
{
    unsigned char ngay;
    unsigned char thang;
    unsigned int nam;
};
```

Cuối cùng, ta có thể xây dựng kiểu dữ liệu thể hiện thông tin về một sinh viên:

```
typedef struct SinhVien
{
    char masv[15];
    char tensv[25];
    Date ngsinh;
    float Diemthi;
};
```

Giả sử đã có cấu trúc phù hợp để lưu trữ một sinh viên, nhưng thực tế lại cần quản lý nhiều sinh viên, lúc đó nảy sinh nhu cầu xây dựng kiểu dữ liệu mới (kiểu mảng bản ghi,…).

2.3. Các kiểu dữ liệu trừu tượng.

Do người sử dụng tự tạo lập để giải quyết bài toán riêng của mình mà CTDL tiền định, cơ sở không phù hợp hoặc không đủ linh hoạt để giải quyết các bài toán đó.

Như: Danh sách liên kết, cây, đồ thị,... Chúng ta sẽ tìm hiểu ở các chương sau của giáo trình.

2.4. Cać tiêu chuẩn đanh gia´cấú truć dư˜liệu.

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.

- *Phù hợp với các thao tác trên đó:* Tiêu chuẩn này giúp tăng tính hiệu

quả của giải thuật, giúp việc phát triển các giải thuật đơn giản, tự nhiên hơn; chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

- *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. Tiêu chuẩn này nên cân nhắc tùy vào tình huống cụ thể khi thực hiện đề án . 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.

2.5. Các thao tác cơ bản trên một cấu trúc dữ liệu.

Mỗi khi chọn một CTDL phải nghĩ ngay tới các phép toán tác động trên cấu trúc đó. Và ngược lại, nói tới phép toán thì phải chú ý tới phép toán đó được tác động trên cấu trúc nào. Cho nên cũng không có gì lạ khi người ta quan niệm: Nói tới CTDL là bao hàm luôn cả phép toán tác động trên cấu trúc ấy.

Thông thường mỗi cấu trúc dữ liệu đều có các phép toán (thao tác) sau:

- Tạo lập
- Huỷ bỏ
- Chèn một phần tử vào CTDL
- Loại bỏ một phần tử khỏi CTDL
- Tìm kiếm một phần tử
- Duyệt CTDL

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

Ví dụ: Phép loại bỏ và phép bổ sung một phần tử rất hữu hiệu với cấu trúc danh sách liên kết nhưng lại rất bất tiện với cấu trúc mảng.

3. Giải thuật và đánh giá độ phức tạp của giải thuật

3.1. Giải thuật.

Mọi chương trình khi được cai đặt trong maý tính, ngươì sử dụng chỉ cần cung cấp dữ liệu vaò (input), maý tính tự động xử lý va`đưa ra kêt́ quả (output). Để coś kêt́ quả đâù ra, ngươì lập trình phải cung cấp cho maý tính một giải thuật (Cać pheṕ xử lý).

Trong thực têś, coś bai toán đơn giản (dêx), nhưng coś bai toán phức tạp (khó) để tìm được lơì giải đa~khoś nhưng diễn tả lơì giải đoś sao cho tương minh, mạch lạc, ro~ràng để nhiêù ngươì coś thể hiểu được cũng không đơn giản. Thông thương giải thuật được biểu diễn băng: Ngôn ngư~tự nhiên, lưu đô`giải thuật, …

3.2. Biểu diễn giải thuật.

3.2.1. Bằng ngôn ngữ tự nhiên.

Dùng ngôn ngữ tự nhiên diễn tả ngắn gọn giải thuật. Cách này chỉ áp dụng với những giải thuật đơn giản.

Ví dụ 1.6: Thuật giải nấu cơm có thể viết như sau:

Bước 1: Lấy gạo theo định lượng cần thiết.

Bước 2: Vo gạo và đổ gạo + nước vào nồi vơí lượng vừà đủ.

Bước 3: Căm điện, đun sôi cạn nước (khoảng 15 phuť).

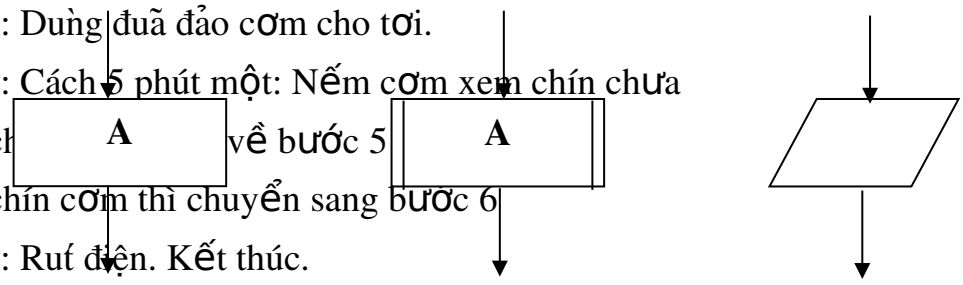
Bước 4: Dung đuã đảo cơm cho tơi.

Bước 5: Cách 5 phút một: Nếm cơm xem chín chưa

Nếu ch về bước 5

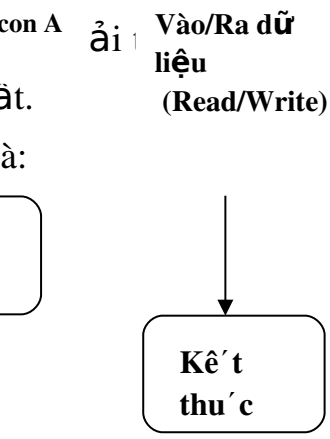
Nếu chín cơm thì chuyển sang bước 6

Bước 6: Ruť điện. Kết thúc.

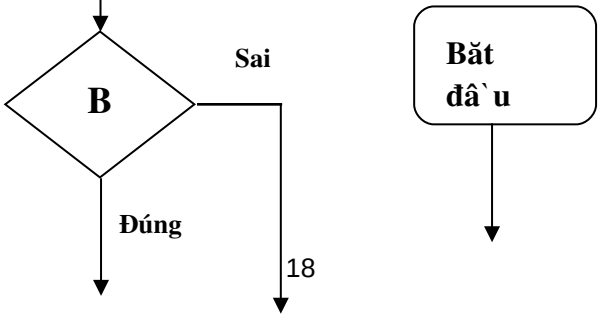


3.2.2. Bằng lưu đồ giải thuật.

Dùng r b ải này giải thuật được minh hoạ một cách trực quan nhất.

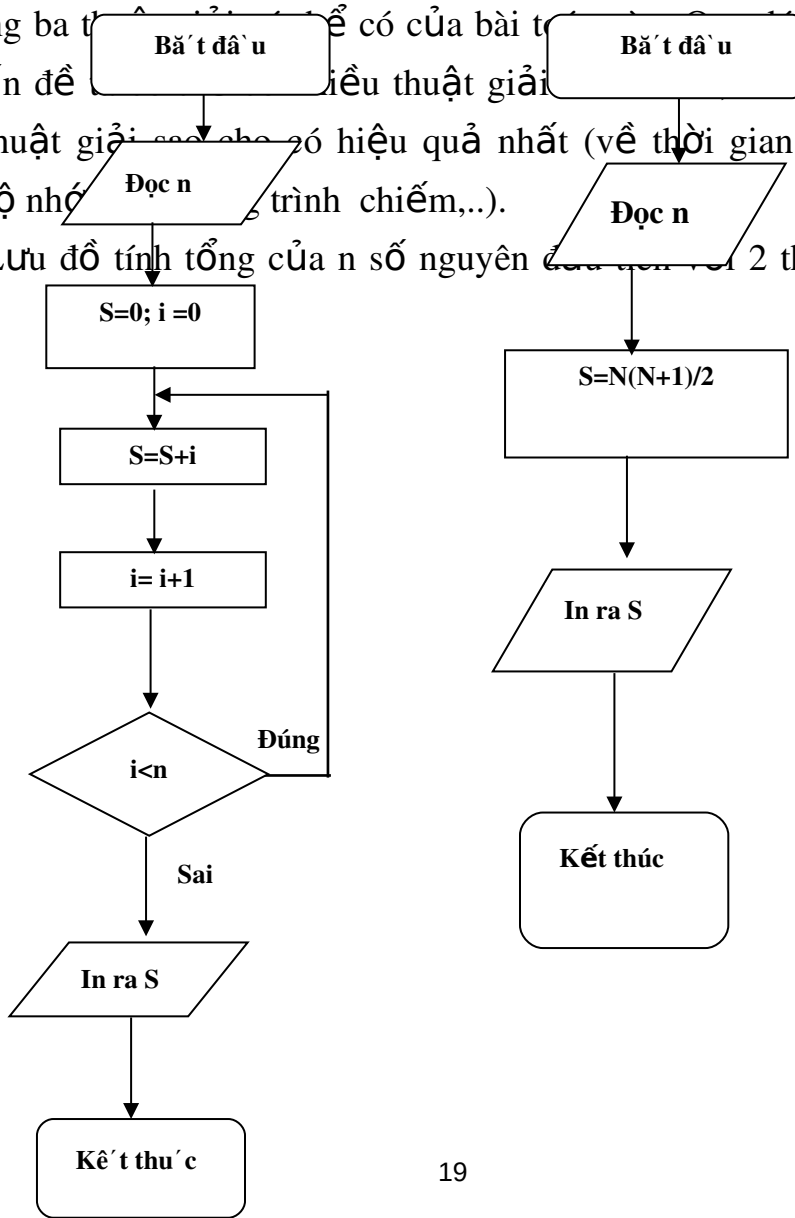


Các hình cơ bản để xây dựng lưu đồ giải thuật là:



Ví dụ 1.7: Tính tổng của n số nguyên đầu tiên. Thuật giải dưới đây chỉ là hai trong ba t[illegible]ể có của bài t[illegible]cho thấy với một vấn đề [illegible]iều thuật giải[illegible]ó ta phải xây dựng thuật giải sao cho có hiệu quả nhất (về thời gian chạy chương trình, bộ nhớ [illegible] trình chiếm,..).

Lưu đồ tính tổng của n số nguyên [illegible] 2 thuật giải khác nhau:



3.2.3. Bằng ngôn ngữ diễn đạt giải thuật (mã giả).

Như chúng ta đã biết, với những bài toán đơn giản thì chỉ cần biểu diễn giải thuật bằng ngôn ngữ tự nhiên, với bài toán lớn, phức tạp việc dùng lưu đồ khối để diễn tả giải thuật có những hạn chế nhất định (như khuôn khổ giấy, màn hình có hạn làm ảnh hưởng đến tầm quan sát của mắt, hoặc có giải thuật phức tạp gồm nhiều vòng lặp lồng nhau,…khi đó sẽ dẫn đến nhiều hạn chế). Chúng ta càng không nên dùng một ngôn ngữ cụ thể để diễn đạt giải thuật vì:

- Phải luôn tuân thủ các nguyên tắc chặt chẽ về cú pháp của ngôn ngữ

đó, khiến cho việc trình bày về giải thuật và CTDL có thiên hướng nặng nề, gò bó.

- Phải phụ thuộc vào CTDL tiền định của ngôn ngữ nên có lúc không

thể hiện đầy đủ các ý về cấu trúc mà ta mong muốn giới thiệu.

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

- Giải thuật cần độc lập với ngôn ngữ cài đặt để có thể cài đặt nó bằng bất cứ ngôn ngữ nào.

Một giải pháp cho vấn đề này là dùng ngôn ngữ diễn đạt giải thuật có đủ khả năng diễn đạt được giải thuật trên các cấu trúc đề cập đến với một mức độ linh hoạt nhất định , không quá gò bó , không câu nệ nhiều về cú pháp nhưng cũng gần gũi với các ngôn ngữ chuẩn để việc chuyển đổi, khi cần thiết được dễ dàng. Với cách này giải thuật vừa gần gũi với người, vừa gần gũi với ngôn ngữ lập trình chuẩn.

Ngôn ngữ dùng để diễn đạt giải thuật thường được chọn là C hoặc Pascal vì 2 ngôn ngữ này hay được sử dụng như là ngôn ngữ lập trình căn bản, và được gọi là ngôn ngữ tựa C hoặc tựa Pascal.

Ví dụ 1.8: Giải thuật tìm USCLN của 2 số a, b theo thuật toán Euclid

```
 USCLN (a, b 2 số nguyên)
{    số nguyên  r;
     r ← a%b;
     while (r ≠ 0)
          { a← b;
            b← r;
            r← a%b;
          }
     return (b);
}
```

Đoạn mã giả trên diễn đạt cho giải thuật tìm USCLN, có thiên về cú pháp ngôn ngữ C nhưng nó vẫn ở dạng thô, chưa sử dụng chính xác các câu lệnh trong C.

Như vậy, các cách diễn tả giải thuât ở trên chỉ có ý nghĩa giữa người với người, để máy tính hiểu và thực hiện các thao tác của giải thuật cần phải cài đặt chúng bằng những câu lệnh, cú pháp của một ngôn ngữ lập trình cụ thể.

3.3. Một số đặc trưng của giải thuật

- Tính đơn nghĩa: Ở mỗi bước của giải thuật, các thao tác phải hết

  sức rõ ràng, không gây nên sự nhập nhằng, lộn xộn, tùy tiện, đa nghĩa.

(Cần phân biệt với tính đơn định: Với hai bộ dữ liệu đầu vào giống nhau cho trước, giải thuật sẽ thi hành các mã lệnh giống nhau và cho kết quả giống nhau).

- Tính dừng: Sau một số hữu hạn bước thực hiện các thao tác sơ cấp đã chỉ ra thì giải thuật phải đi đến kết thúc để trả ra kết quả mong muốn. Không được rơi vào quá trình vô hạn.

- Tính đúng đắn: Với mọi bộ dữ liệu đầu vào, sau khi kết thúc giải thuật ta phải thu được kết quả mong muốn. Kết quả đó được kiểm chứng bằng yêu cầu bài toán.

- Tính phổ dụng: Giải thuật phải dễ sửa đổi để thích ứng với bất kỳ bài toán nào trong một lớp bài toán và có thể làm việc trên các dữ liệu cụ thể khác nhau.

- Tính hiệu quả: Trong số nhiều giải thuật cùng giải một bài toán, tính hiệu quả được đánh giá là giải thuật có thời gian thực hiên nhanh nhất và tốn ít bộ nhớ nhất.

3.4. Đánh giá độ phức tạp của giải thuật

3.4.1. Đặt vấn đề

Thông thường, một bài toán có nhiều giải thuật (lời giải). Chọn một giải thuật đưa tới kết quả nhanh là một đòi hỏi thực tế. Nhưng, căn cứ vào đâu để đánh giá giải thuật này nhanh hơn giải thuật kia?

Thời gian thực hiện giải thuật phụ thuộc vào rất nhiều yếu tố:

Yếu tố đầu tiên đó là kích thước của dữ liệu đưa vào, dữ liệu càng lớn thì càng tốn nhiều thời gian: Để sắp xếp một dãy số thì số lượng các số thuộc dãy số đó ảnh hưởng rất lớn tới thời gian thực hiện giải thuật. Nếu gọi n là số lượng này (kích thước của dữ liệu vào) thì thời gian thực hiện T của một giải thuật phải được biểu diễn như một hàm của n: T(n).

Tốc độ xử lý của máy tính, ngôn ngữ viết chương trình và chương trình dịch ngôn ngữ ấy cũng ảnh hưởng tới thời gian thực hiện, nhưng những yếu tố này không đồng đều với mọi loại máy trên đó cái đặt giải thuật, vì vậy không thể dựa vào chúng khi xác lập T(n). Như vậy, T(n) không thể được biểu diễn thành đơn vị thời gian bằng giây, bằng phút,... được. Tuy nhiên, không phải vì thế mà không thể so sánh được các giải thuật về mặt tốc độ.

Nếu như thời gian thực hiện của một giải thuật là $T_1(n)=cn^2$ và thời gian thực hiện một giải thuật khác là $T_2(n)=kn$ với c và k là một hằng số nào đó, thì khi n khá lớn, thời gian thực hiện giải thuật sau rõ ràng ít hơn so với giải thuật trước. Nếu nói thời gian thực hiện giải thuật T(n) tỉ lệ với n^2 hay tỉ lệ với n cũng cho ta ý niệm về tốc độ thực hiện giải thuật đó khi n khá lớn (với n nhỏ thì việc xét T(n) không có ý nghĩa). Cách đánh giá thời gian thực hiện giải thuật độc lập với máy tính và các yếu tố liên quan tới máy như vậy sẽ dẫn tới khái niệm về "cấp độ lớn của thời gian thực hiện giải thuật" hay còn gọi là "Độ phức tạp tính toán của giải thuật" .

3.4.2. Độ phức tạp tính toán của giải thuật.

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

$T(n)=O(n^2)$ (ký hiệu chữ O lớn).

Một cách tổng quát có thể định nghĩa:

Một hàm f(n) được xác định là O(g(n)): f(n)=O(g(n)) và được gọi là có cấp g(n) nếu tồn tại các hằng số c và n_o sao cho f(n) cg(n) khi n no. Nghĩa là f(n) bị chặn trên bởi một hằng số nhân với g(n), với mọi giá trị của n từ một điểm nào đó. Thông thường các hàm thể hiện độ phức tạp tính toán của giải thuật có dạng: $\log_2 n$, n, $n\log_2 n$, n^2, n^3, 2^n, n!, n^n .

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

3.4.3. Xác định độ phức tạp tính toán của giải thuật.

Xác định độ phức tạp tính toán của một giải thuật bất kì có thể dẫn tới những bài toán phức tạp. Trong thực tế, đối với một số giải thuật ta cũng có thể xác định được bằng một số quy tắc đơn giản sau:

a. Qui tắc tổng:

Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P_1 và P_2.

Trong đó: $T_1(n)=O(f(n))$; $T_2(n)=O(g(n))$.

Thì thời gian thực hiện P_1 rồi P_2 tiếp theo sẽ là: $T_1(n) + T_2(n)= O(\max(f(n),g(n)))$.

Ví dụ 1.9:

Trong một chương trình có 3 bước thực hiện P1, P2, P3

Thời gian thực hiện từng bước lần lượt là: $O(n^2)$, $O(n^3)$ và $O(n\log_2 n)$

Thì thời gian thực hiện P1 rồi đến P2 là: $O(\max(n^2, n^3))= O(n^3)$.

Thời gian thực hiện chương trình sẽ là : $O(n^3, n\log_2 n)= O(n^3)$.

Mở rộng của qui tắc tổng:

Nếu g(n) f(n) với mọi n no thì O(f(n) + g(n)) cũng là O(f(n)).

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

b. Qui tắc nhân:

Trong một chương trình có 2 đoạn P_1 và P_2 lồng nhau,

Nếu tương ứng với P_1: $T_1(n) =O(f(n))$;

Tương ứng với P_2: $T_2(n)=O(g(n))$

Thì thời gian thực hiện của 2 đoạn P_1 và P_2 lồng nhau sẽ là:

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

Ví dụ 1.10:

Câu lệnh x=x+1; Có thời gian thực hiện bằng c (hằng số) nên được đánh giá là O(1).

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

c. Một số nguyên tắc:

- *Bỏ hằng*: O(cf(n))=O(f(n)) với c là hằng số.
  Ví dụ: $O(n^2/2)=O(n^2)$.
- *Phép toán tích cực*: là phép toán thuộc giải thuật mà số lần thực hiện

nó không kém gì các phép toán khác. (tất nhiên phép toán tích cực không phải là duy nhất). Khi đánh giá thời gian thực hiện giải thuật ta chỉ cần dựa vào phép toán tích cực.

- *Tình trạng dữ liệu vào:* Thời gian thực hiện giải thuật không những

chỉ phụ thuộc vào kích thước của dữ liệu vào mà còn phụ thuộc vào tình trạng của dữ liệu đó nữa. Khi phân tích thời gian thực hiện giải thuật ta sẽ phải xét tới: T(n) trong trường hợp thuận lợi nhất? T(n) trong trường hợp xấu nhất là thế nào? Và T(n) trong trường hợp trung bình? Việc xác định T(n) trung bình thường khó vì sẽ phải dùng tới những công cụ đặc biệt, hơn nữa tình trạng trung bình có thể có nhiều cách quan niệm. Trong trường hợp T(n)tb khó xác định người ta thường đánh giá giải thuật qua giá trị xấu nhất của T(n).

Ví dụ 1.11:

Xét bài toán tìm một phần tử X có giá trị cho trước ở trong dãy A gồm n phần tử. Ta coi phép toán tích cực ở đây là phép toán so sánh ai với X.

Trường hợp thuận lợi nhất xảy ra khi X=a1 (Một lần thực hiện so sánh).

T(n)tốt=O(1).

Trường hợp xấu nhất xảy ra khi X=an (hoặc không tìm thấy).

T(n)xấu=O(n).

Thời gian trung bình được đánh giá: T(n)tb=T(n)xấu=O(n).

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 1

1) Cho ví dụ minh hoạ mối quan hệ giữa CTDL và giải thuật.

2) Cho ví dụ minh hoạ mối quan hệ giữa CTDL và cấu trúc lưu trữ.

3) Hãy nêu ba CTDL tiền định của ngôn ngữ lập trình đã học.

4) Để quản lý hồ sơ của nhân viên một cơ quan, biết rằng thông tin về một hồ sơ gồm: Mã hồ sơ, họ đệm, tên, ngày sinh (ngày, tháng, năm), giới tính, địa chỉ (số nhà, đường phố, phường(xã), quận (huyện), thành phố (tỉnh)), nghề nghiệp, trình độ, năm tuyển dụng, hệ số lương.

- Hãy khai báo cấu trúc dữ liệu phù hợp để lưu trữ được thông tin về

  các hồ sơ nhân viên của cơ quan.

- Viết hàm nhập thông tin từng hồ sơ, kết thúc nhập khi mã hồ sơ rỗng.
- Viết hàm hiện thông tin hồ sơ của từng nhân viên ra màn hình.

5) Hãy nêu các đặc trưng của một giải thuật, cho ví dụ minh họa.

6) Viết lưu đồ giải thuật của ví dụ 1.1, 1.2 và mở rộng ví dụ 1.2 (tính tổng các số >0 và tích các số<0), dùng ngôn ngữ C cài đặt các lưu đồ trên.

7) Hãy dùng ngôn ngữ tựa C để diễn đạt cho giải thuật sắp xếp một dãy số nguyên theo thứ tự tăng dần từ nhỏ đến lớn. Dùng ngôn ngữ C Cài đặt giải thuật sắp xếp này.

8) Hãy đánh giá độ phức tạp tính toán của các hàm sau:

```
void TGvuong(int n)
{
  for (int i=0; i<n; i++)
   { for (int j=0; j<=i; j++)
        printf("*");
     printf("\n");  }
}
void TGvuongN (int n)
{
```

```
    for (int i=n; i>=0; i--)
     { for (int j=0; j<=i; j++)
          printf("*");
       printf("\n");  }
   }

void taoM(int d[][3],int n)
   { randomize();
    for (int i=0;i<n; i++)
      for (int j=0; j<n; j++)
        d[i][j]=random(20);
   }

   void inM(int r[][3],int n)
   { for (int i=0;i<n; i++)
      { printf("\n");
       for (int j=0; j<n; j++)
          printf("%3d",r[i][j]);
 }
       void SXmang(int a[], int n)
          {
           int temp;
           for (int i= 0;i<n;i++)
            for (int j=i+1; j<n; j++)
                  if (a[i]>a[j])
                          {temp=a[i]; a[i]=a[j]; a[j]=temp;}
           }
   void loaitien (int n)
   { int sc,d=0;
     for (int i=1; i<=n/50; i++)
      for (int j=1; j<=n/20; j++)
      for (int k=1; k<=n/10; k++)
          { sc=(i*50)+(j*20)+(k*10);
```

```
        if (sc==200) {d++;
        printf("\n can: %d to 50\,%d to 20\, %d to 10",i,j,k);
        printf("\n");  }}
printf("tong so cach la: %d",d);}
```

CHƯƠNG 2

ĐỆ QUI VÀ GIẢI THUẬT ĐỆ QUI

Mục tiêu:

- Trình bày được khái niệm về đệ quy.
- Trình bày được giải thuật và chương trình sử dụng giải thuật đệ quy.
- So sánh giải thuật đệ quy với các giải thuật khác để rút ra tính ưu việt hoặc nhược điểm của giải thuật.
- Trình bày một số bài toán đệ qui căn bản.

1. Khái niệm đệ qui

Một đối tượng được gọi là đệ qui nếu nó bao gồm chính nó như một bộ phận hoặc nó được định nghĩa dưới dạng của chính nó.

Định nghĩa đệ qui trong toán học:

- Định nghĩa số tự nhiên:

  1 là một số tự nhiên.

  x là số tự nhiên nếu x-1 là một số tự nhiên

- Định nghĩa n giai thừa: N!

  N! = 1 nếu n=0 hoặc n=1

  N!= n × (n-1)! nếu n>1

Hình ảnh đệ qui trong đời sống hàng ngày:

Búp bê Nga (là một loại búp bê đặc trưng của Nga). "Diện mạo" của các búp bê trong cùng một bộ thường cùng thuộc một chủ đề. Một bộ gồm những búp bê rỗng ruột có kích thước từ lớn đến nhỏ. Con búp bê lớn sẽ chứa đựng trong lòng con búp bê nhỏ hơn nó một chút, cứ thế, con lớn nhất sẽ chứa tất cả những con búp bê còn lại trong bộ. Mỗi khi nhấc một con phía ngoài ra ta lại thấy một con nhỏ hơn,… và con nhỏ nhất được nằm trong cùng.

2. Giải thuật đệ qui và chương trình đệ qui.

2.1. Giải thuật đệ qui.

Nếu lời giải của bài toán P được thực hiện bằng lời giải của một bài toán P', có dạng giống như P, thì đó là một lời giải đệ qui. Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ qui.

Nhưng điểm mấu chốt cần lưu ý là: P' tuy có dạng giống như P, nhưng theo một nghĩa nào đó, nó phải nhỏ hơn P.

Ví dụ 2.1: Xét bài toán tính giai thừa của số nguyên dương n.

Giải thuật đệ qui :

Input: n, là một số nguyên dương.

Process:

Bước 1: Kiểm tra n:

Nếu: n=0 hoặc n=1 thì gán N!←1 và kết thúc

Nếu: n≠0 và n≠1 chuyển sang bước 2

Bước 2: Tính giai thừa của n theo công thức: N! ←n*(n-1)! Và quay lại bước 1

Output: N!, là giai thừa của n

Giả sử n=4, giải thuật tính giai thừa của n được thể hiện cụ thể như sau:

n≠0 và n≠1 chuyển sang bước 2

4!=4*(4-1)! Quay lại bước 1

n≠0 và n≠1 chuyển sang bước 2

3!=3*(3-1)! Quay lại bước 1

n≠0 và n≠1 chuyển sang bước 2

2!=2*(2-1)! Quay lại bước 1

n=1 1!=1 Kết thúc

Nhận xét:

- Sau mỗi lần kiểm tra n ≠ 0 hoặc n ≠ 1 thì n lại giảm đi một giá trị

và sẽ lại được thực hiện bằng một chiến thuật như đã dùng trước đó.

- Có một trường hợp đặc biệt, khác với mọi trường hợp trước, sẽ đạt được sau nhiều lần giảm n đi một giá trị, đó là trường hợp n=0 hoặc n=1. Lúc đó việc giảm n sẽ ngừng lại. Trường hợp đặc biệt này được gọi là trường hợp suy biến.

Ta thể hiện giải thuật tính N! này dưới dạng một hàm hay chương trình con đệ qui.

2.2. Chương trình con đệ qui.

Hàm tính giai thừa:

```
unsigned long FACTORIAL (unsigned int n)
 { if ((n==0)  || (n==1))      return 1;
   else  return  n*FACTORIAL(n-1);
  }
```

Hàm như trên được gọi là hàm đệ qui. Có thể nêu ra mấy đặc điểm sau:

- Hàm đệ qui có lời gọi đến chính hàm đó. Ở đây hàm FACTORIAL có lời gọi tới hàm FACTORIAL.
- Mỗi lần có lời gọi lại hàm thì kích thước của bài toán đã thu nhỏ hơn trước. Ở đây khi có lời gọi hàm FACTORIAL thì kích thước n được giảm đi một giá trị so với trước khi có lời gọi.
- Có một trường hợp đặc biệt, trường hợp suy biến. Ở đây chính là trường hợp (n==0) hoặc (n==1). Khi trường hợp này xảy ra thì bài toán còn lại sẽ được giải quyết theo một cách khác hẳn và việc gọi đệ qui cũng kết thúc. Chính tình trạng kích thước bài toán cứ giảm dần sẽ đảm bảo cho trường hợp suy biến này đạt tới được.

2.3. Đặc điểm của một chương trình con đệ qui:

Một chương trình con được gọi là đệ qui đồng thời phải thỏa mãn 3 đặc điểm sau:

- Chương trình con (CTC) đệ qui co´lơi gọi đến chính no´.
- Môĩ lâ`n co´lơi gọi lại CTC thi`kích thước của bai toán đa~thu nhỏ hơn trước.
- Co´một trương hợp đặc biệt, trương hợp suy biến. Đây con gọi la`*điều kiện dưng của chương trình con đệ qui.*

3. Thiết kế giải thuật đệ qui

3.1. Giải thuật đệ qui đơn giản

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

Ví dụ 2.2: Hàm Euclid-USCLN(a,b): Ước số chung lớn nhất của 2 số nguyên

$$USCLN(a,b) = \begin{cases} a & \text{nếu } b=0 \\ USCLN(b, a \% b) & \text{nếu } b <> 0 \end{cases}$$

Hàm USCLN(a,b) được viết dưới dạng hàm đệ qui như sau:

```
unsigned int USCLN(unsigned int a, unsigned int b)
{
        if (b==0) return a;
        else    return USCLN(b, a % b);
}
```

Ví dụ 2.3: Dãy số FIBONACCI

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

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

Giả sử bắt đầu từ một cặp mới ra đời thì đến tháng thứ n sẽ có bao nhiêu cặp?

Ví dụ với n=6, ta thấy:

Tháng thứ 1: 1 cặp (ban đầu).

Tháng thứ 2: 1 cặp (cặp ban đầu vẫn chưa đẻ).

Tháng thứ 3: 2 cặp (đã có thêm một cặp con).

Tháng thứ 4: 3 cặp (cặp con vẫn chưa đẻ).

Tháng thứ 5: 5 cặp (cặp con bắt đầu đẻ).

Tháng thứ 6: 8 cặp (Cặp con đẻ tiếp).

Vậy ta xét đến việc tính số cặp thỏ ở tháng thứ n: F(n).

Ta thấy nếu mỗi cặp thỏ ở tháng thứ (n-1) đều sinh con thì F(n) = 2*(n-1) nhưng không phải như vậy. Trong các cặp thỏ ở tháng thứ (n-1) chỉ có những cặp đã có ở tháng thứ (n-2) mới sinh con ở tháng thứ n được thôi.

Do đó: F(n) = F(n-2) + F(n-1)

Vì vậy có thể tính F(n) theo công thức sau:

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

Dãy số thể hiện F(n) ứng với các gía trị của n có dạng sau:

n	1	2	3	4	5	6	7	8	9	…
F(n)	1	1	2	3	5	8	13	21	34	…

Dãy trên gọi là dãy số Fibonacci. Nó là mô hình của rất nhiều hiện tượng tự nhiên và cũng được sử dụng nhiều trong tin học.

Hàm đệ qui sau thể hiện giải thuật tính F(n)

```
unsigned int  F(unsigned int n)
    { if (n ≤ 2)  return (1);
      else  return  (F(n-2) + F(n-1));
    }
```

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

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

Nhưng không phải lúc nào tính đệ qui trong trong cách giải bài toán cũng thể hiện rõ nét và đơn giản như vậy.

3.2. Nguyên tắc thiết kế một giải thuật đệ qui:

Để thiết kế một giải thuật đệ qui ta cần trả lời các câu hỏi sau:

- Có thể định nghĩa được bài toán dưới dạng một bài toán cùng loại, nhưng "nhỏ" hơn không? Và nếu được thì nhỏ hơn như thế nào?
- Như thế nào là kích thước của bài toán được giảm đi ở mỗi lần gọi

đệ qui?

- Trường hợp đặc biệt nào của bài toán sẽ được coi là trường hợp suy biến?

Ví dụ 2.4: Viết chương trình đảo ngược chữ số của một số (ví dụ:12345 →54321), yêu cầu sử dụng thuật toán đệ qui.

- Trả lời các câu hỏi:
  - *Có thể định nghĩa được bài toán dưới dạng một bài toán cùng loại, nhưng "nhỏ" hơn không?* Có, vì nguyên tắc đảo ngược các chữ số của một số là tách lần lượt từng chữ số từ phải sang trái và viết lại từng chữ số theo chiều ngược lại (từ trái qua phải).
  - *Và nếu được thì nhỏ hơn như thế nào?* Nhỏ hơn 10 lần.
  - *Như thế nào là kích thước của bài toán được giảm đi ở mỗi lần gọi đệ qui ?* Mỗi lần gọi đệ qui thì giá trị **so** được giảm đi 10 lần (so=so/10)
  - *Trường hợp đặc biệt nào của bài toán sẽ được coi là trường hợp suy biến?* Trường hợp **so** chỉ còn một chữ số (so<10).
- Giải thuật đảo số như sau:

Input: so, là một số nguyên dương.

Process:

Bước 1:

- Nếu *so* < 10 thì hiển thị chữ số đó ra màn hình. Kết thúc chương trình
- Nếu *so* >=10 thì chuyển sang bước 2

Bước 2:

- Lấy số bị chia ta chia cho 10, được số dư hiển thị ra màn hình
- Giảm giá trị *so* đi 10 lần, quay lại bước 1

Output: Số được đảo ngược.

- Hàm Daoso dạng đệ qui:

```
void  Daoso (unsigned int so)
{
      if (so < 10)   printf("%d",so);
      else
        {     printf("%d", so % 10);
              Daoso(so /10);
         }
```

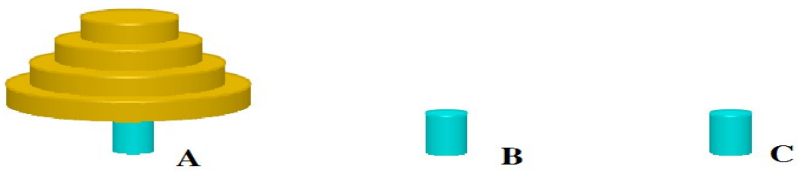
{

Ví dụ 2.5: Bài toán tháp Hà nội

Ở ví dụ trên ta có thể giải quyết bài toán bằng một giải thuật khác đơn giản hơn nhiều (như giải thuật lặp), nhưng trên thực tế có nhiều bài toán mà việc giải quyết nó bằng cách dùng thuật toán đệ qui tự nhiên, dễ hiểu và tường minh hơn, như bài toán Tháp Hà Nội.

Bài toán : Có một chồng n đĩa ở cọc nguồn (đĩa to ở dưới, nhỏ ở trên) ta cần chuyển sang cọc đích thông qua các luật sau:

- Khi di chuyển một đĩa, nó phải đặt vào một trong ba cọc (Thêm cọc trung gian) đã cho.
- Mỗi lần chỉ có thể chuyển một đĩa, và phải là đĩa ở trên cùng
- Đĩa lớn hơn không bao giờ được phép nằm trên đĩa nhỏ hơn



o

Cách giải quyết theo **giải thuật đệ quy** như sau:

- Đặt tên các cọc là A, B, C. Những tên này có thể chuyển ở các bước khác nhau (ở đây: A = Cọc Nguồn, C = Cọc Đích, B = Cọc Trung Gian).
- Gọi n là tổng số đĩa.
- Đánh số đĩa từ 1 (nhỏ nhất, trên cùng) đến n (lớn nhất, dưới cùng).

Trường hợp n=1:

Thực hiện yêu cầu bài toán bằng cách chuyển trực tiếp đĩa 1 từ cọc A sang cọc C.

Trường hợp n=2:

Chuyển đĩa thứ nhất (ở trên) từ cọc A sang cọc trung gian B.

Chuyển đĩa thứ hai (ở dưới) từ cọc A sang cọc đích C.

Chuyển đĩa thứ nhất từ cọc trung gian B sang cọc đích C.

Kết quả thu được thỏa mãn đầu bài.

Trường hợp n>2:

Giả sử ta đã có cách chuyển n-1 đĩa, ta thực hiện như sau:

1. Chuyển **n-1** đĩa trên cùng ở **cọc nguồn (A)** sang **cọc trung gian (B)**, dùng **cọc đích (C)** làm cọc phụ.
2. Chuyển đĩa thứ **n** ở **cọc nguồn (A)** sang **cọc đích (C).**

3. Chuyển **n-1** đĩa từ **cọc trung gian (B)** sang **cọc đích (C)**, dùng **cọc nguồn (A)** làm cọc phụ.

Như vậy, bài toán tháp Hà nội tổng quát với n đĩa đã dẫn đến được bài toán tương tự với kích thức nhỏ hơn, nghĩa là từ chuyển n đĩa từ cọc A sang cọc C được chuyển về bài toán chuyển n-1 đĩa từ cọc A sang cọc B,… Điểm dừng của giải thuật đệ qui khi n=1 và ta chuyển thẳng đĩa này từ cọc A sang cọc đích C.

Giải thuật đệ qui như sau:

- Hàm chuyen(int n, char A, char C) thực hiện chuyển đĩa thứ n từ cọc A sang cọc C.
- Hàm thapHNdq(int n, char A, char C, char B) là hàm đệ qui thực hiện việc chuyển n đĩa từ cọc nguồn A sang cọc đích C và sử dụng cọc trung gian B.

Cài đặt giải thuật đệ qui bằng ngôn ngữ C:

```
void chuyen(int n,  char A, char C)
{
        printf("chuyen dia thu %d tu coc %c sang coc %c\n",n,A,C);
}
void thapHNdq(int n, char A, char C, char B)
{
        if (n==1) chuyen(1, A, C);
        else
        {
                thapHNdq(n-1, A, B, C);
                chuyen(n, A, C);
                thapHNdq(n-1, B, C, A);
        }
 }
```

3.3. Nguyên tắc thực hiện một hàm đệ qui trong máy tính:

Bước 1: Mở đâù

Bảo lưu tham số, biến cục bộ va`địa chỉ quay lui.

Bước 2: Thân

- Nêú tiêu chuẩn cơ sở ứng với trương̀ hợp suy biến đa˜đạt được thi`thực hiện phân tính kêt́ thuć va`chuyển sang bươć 3

- Nếu không thi`thực hiện việc tính tưng phân va`chuyển sang bước 1 (khởi tạo một lơi gọi đệ qui).

Bước 3: Kết thuć

Khôi phục lại tham sô,́ biến cục bộ, địa chỉ quay lui va`chuyển tơí địa chỉ quay lui naỳ.

4. Nhận xét giải thuật đệ qui

Nhược điểm:

Tốn bộ nhớ và chạy chậm vì:

- Khi một hàm đệ qui gọi chính nó, tập các đối tượng được sử dụng trong hàm được tạo ra như tham sô,́ biến cục bộ. Ngoài ra, việc chuyển giao điều khiển từ các hàm cũng cần lưu trữ thông số (gọi la`địa chỉ quay lui), dùng cho việc trả lại điều khiển cho hàm ban đầu.
- Việc sử dụng đệ qui đôi khi tạo ra các phép toán thừa, không cần thiết do tính chất tự động gọi thực hiện hàm khi chưa gặp điều kiện dừng của đệ qui (ví dụ: return (F(n-2) + F(n-1))).

Ưu điểm:

- Giải thuật đệ quy đẹp (gọn gàng), dễ chuyển thành chương trình.
- Nhiều giải thuật rất dễ mô tả dạng đệ qui nhưng lại rất khó mô tả với giải thuật không đệ qui (bài toán tháp Hà nội), và có những giải thuật đệ qui thực sự có hiệu lực cao (như giải thuật sắp xếp nhanh - Quick Sort)
- Về mặt định nghĩa, công cụ đệ qui đã cho phép xác định một tập vô hạn các đối tượng bằng một phát biểu hữu hạn. Như trong định nghĩa văn phạm, định nghĩa cú pháp ngôn ngữ, định nghĩa một số cấu trúc dữ liệu,...

Trong thực tế, tất cả các giải thuật đệ qui đều có thể đưa về dạng lặp (còn gọi là "khử" đệ qui). Do đó, chỉ sử dụng đệ qui khi các giải thuật không đệ qui thay thế trở nên phức tạp hoặc chương trình trở nên rất khó hiểu.

Ví dụ 2.6: Giải thật Fibonacci không đệ qui (dùng phương pháp lặp):

```
int F(int n)
{   int f1, f2, fn;
    f1=1;  f2=1;
    for (int i=3; i<=n; i++)
```

```
        {   fn=f1+f2;
            f1=f2;
            f2= fn;
        }
    return fn;
}
```

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 2

1) Thực hiện công việc sau:
   - Viết giải thuật đệ qui để đảo ngược một xâu ký tự, ví dụ xâu "abcde" thành "edcba".
   - Hãy chỉ rõ các đặc điểm của giải thuật đệ qui ở giải thuật trên.
   - Viết hàm đệ qui theo giải thuật trên.
   - Viết hàm khử đệ qui bằng phương pháp lặp cho giải thuật trên.
2) Viết hàm khử đệ qui bằng phương pháp lặp cho **ví dụ 2.4.**
3) Hoàn thiện **ví dụ 2.5** thành một chương trình bằng ngôn ngữ C, chạy và kiểm tra kết quả với n=4.

CHƯƠNG 3
DANH SÁCH

Mục tiêu:

- Trình bày khái niệm và các phép toán cơ bản trên danh sách;
- Trình bay cách sử dụng các loại danh sách về cách tổ chức và các thao tác xử lý cơ bản trên cấu trúc danh sách.
- Giải được các bài toán sử dụng danh sách.

1. Danh sách và các phép toán cơ bản trên danh sách

1.1. Khái niệm danh sách tuyến tính

Cấu trúc dữ liệu rất quen thuộc ở mọi ngôn ngữ lập trình là cấu trúc Mảng (array). Mảng là một tập có thứ tự gồm một số cố định các phần tử có cùng 1 kiểu dữ liệu, được lưu trữ kế tiếp nhau và được truy cập thông qua một chỉ số. Rất ít dùng phép bổ sung hay loại bỏ phần tử đối với mảng. Thường chỉ có phép tạo lập (Create) mảng, tìm kiếm (Retrieve) một phần tử của mảng, lưu trữ (store) một phần tử của mảng.

Danh sách có hơi khác với mảng ở chỗ: Nó là một tập có thứ tự nhưng bao gồm một số biến động các phần tử (số lượng các phần tử luôn thay đổi). Phép bổ sung và loại bỏ một phần tử là phép thường xuyên tác động lên danh sách.

Một danh sách mà quan hệ lân cân giữa các phần tử được hiển thị ra thì được gọi là danh sách tuyến tính (Linear list). Véc tơ chính là trường hợp đặc biệt của danh sách tuyến tính, đó là hình ảnh của danh sách tuyến tính xét tại một thời điểm nào đó (giống như mảng, chỉ khác là kích thước của danh sách có giá trị thay đổi). Ngoài phép bổ sung và loại bỏ thường xuyên tác động còn có các phép: Phép ghép, tách, sắp xếp, tìm kiếm,...)

1.2. Cài đặt danh sách theo cấu trúc mảng.

Véc tơ là trường hợp đặc biệt của danh sách tuyến tính. Có thể dùng véc tơ lưu trữ để lưu trữ danh sách tuyến tính. Nếu có một danh sách tuyến tính $(a_0,a_1,...,a_{n-1})$ ta có thể lưu trữ bằng véctơ lưu trữ $(v_0,v_1,...,v_{n-1})$ với n là biến động, m là biến cố định (m>= max của n).

V_0	V_1	V_2	V_3		…	V_{n-1}

Hình 3.1: Véc tơ lưu trữ V

Trong cài đặt danh sách bằng mảng (còn gọi là lưu trữ kế tiếp), giả sử độ dài tối đa của danh sách (maxlist) là một số n nào đó, các phần tử của danh sách có kiểu dữ liệu Item (Item có thể là các kiểu dữ liệu đơn giản hoặc kiểu dữ liệu có cấu trúc). Mỗi phần tử của danh sách được biểu diễn bằng một bản ghi gồm 2 trường. Trường thứ nhất element là mảng các Item có kích thước maxlist (kích thước của danh sách), trường thứ hai count chứa số lượng phần tử thực sự hiện có trong danh sách.

Phần tử thứ 0	Phần tử thứ 1	Phần tử thứ 2	…	Phần tử thứ maxlist -1
count 1	count 2	count 3		count (maxlist)

Hình 3.2: Danh sách được cài đặt bằng mảng

1.2.1. Khai báo cấu trúc của danh sách bằng mảng:

```
const       int maxlist=100
typedef struct list
{ Item      element[maxlist];
  int       count;
};
```

Ví dụ 3.1: Khai báo một danh sách lưu trữ các số nguyên

```
const    int maxlist=100
typedef  int          Item   ;
   typedef struct list
   { Item      element[maxlist];
     int       count;
   };
```

Ví dụ 3.2: Khai báo một danh sách kế tiếp lưu trữ các bản ghi sinhvien

```
   const       int maxlist=100
        //định nghĩa bản ghi SinhVien
  struct SinhVien
    { char Hten [35];
      float   LaptrinhCB, KientrucMT, MangMT, DiemTB;
    };
    typedef     SinhVien    Item;
```

```
typedef struct list
{ Item      element[maxlist];
  int       count;
};
```

1.2.2. Các thao tác cơ bản của danh sách được cài đặt bằng mảng (danh sách kế tiếp)

a. Khởi tạo một danh sách rỗng

Theo khai báo cấu trúc danh sách cài đặt bằng mảng, biến count chứa số lượng phần tử của một danh sách. Để khởi tạo một danh sách rỗng ta chỉ cần gán cho biến count giá trị 0.

Thao tác này được sử dụng đầu tiên trước tất cả các thao tác khác đối với danh sách.

```
void    initializeList (list   *L)
{
        L->count=0;
}
```

a. Kiểm tra một danh sách có rỗng không.

Danh sách rỗng tức là số lượng phần tử của danh sách bằng không (cũng giống như trường hợp khởi tạo danh sách).

Khi danh sách rỗng ta không thể xóa một phần tử khỏi danh sách.

```
int         EmptyList(list     L)
{
        (return L.count==)0;
    }
```

a. Kiểm tra một danh sách có rỗng không.

Danh sách đầy tức là số lượng phần tử của danh sách bằng maxlist, là số lượng phần tử lớn nhất mà danh sách này có thể lưu trữ.

Khi danh sách đầy ta không thể bổ sung một phần tử mới vào danh sách.

```
int         FullList(list         L)
{
        return (L.count== maxlist);
}
```

a. Tìm kiếm một phần tử trong danh sách.

Muốn tìm kiếm một phần tử trong danh sách ta phải dựa vào giá trị một trường khóa của phần tử.

Giải thuật tìm kiếm được thực hiện bởi phép toán so sánh khóa tìm kiếm với giá trị trường khóa của từng phần tử và kết quả trả ra là vị trí phần tử được tìm thấy hoặc giá trị -1 nếu không tìm thấy.

❖ Giải thuật:

Input:

L *// là biến có kiểu list chứa danh sách*

x */* là khóa tìm kiếm và có kiểu dữ liệu phù hợp với trường khóa của danh sách*/*

Process:

Bước 1: Xét các trường hợp.

- Nếu danh sách rỗng (hàm **Empty**==1):

Thông báo danh sách rỗng và return(-1)

- Nếu danh sách không rỗng thì chuyển sang bước 2.

Bước 2: Tìm kiếm.

- So sánh khóa tìm kiếm x với giá trị trường khóa của từng phần tử trong danh sách, bắt đầu từ phần tử đầu tiên cho đến khi tìm thấy hoặc kết thúc danh sách.
- Nếu tìm thấy return (pos: vị trí phần tử)
- Nếu không tìm thấy return (-1).

Output: **pos:** Là vị trí phần tử có giá trị trường khóa trùng với **x** hoặc giá trị -1 nếu không tìm thấy.

❖ Cài đặt giải thuật

```
int    searchElement (list   L,  kiểu trường khóa      x)
{
int i=0;
   if  (Empty (L))
    {
          printf("Danh sach rỗng!");
          return(-1);
        }
   else
```

```
{   while  ((L.element [i].trường khóa==x) && (i<L.count))
            i++;
    if (i<L.count)
            return (i);
    else
            return(-1);
 }
}
```

a. Chèn (bổ sung) một phần tử mới vào danh sách.

Phần tử mới có thể được bổ sung vào các vị trí: Đầu tiên, phía trong hoặc cuối của danh sách.

Để có chỗ bổ sung một phần tử mới, ta phải dịch chuyển các phần tử từ vị trí pos cần bổ sung xuống cuối danh sách để bổ sung phần tử mới.

❖ Giải thuật:

Input:

L *// là con trỏ trỏ có kiểu list*

pos *// chứa vị trí cần chèn*

x *// là biến có kiểu Item và chứa giá trị phần tử mới*

Process:

Bước 1: Xét các trường hợp:

- Nếu danh sách đầy (hàm **Full**==1):

Thông báo danh sách đầy và kết thúc.

- Nếu ví trí pos không hợp lệ (pos<0 hoặc pos>=maxlist)

Thông báo vị trí không hợp lệ và kết thúc

- Nếu danh sách không đầy và ví trí pos hợp lệ thì chuyển sang bước 2.

Bước 2: Chèn phần tử mới.

- Dịch chuyển các phần tử từ vị trí pos xuống cuối danh sách.
- Chèn phần tử mới vào đúng vị trí pos.
- Tăng giá trị biến count thêm một.

Output: Phần tử mới **x** đã được chèn vào vị trí pos trong danh sách nếu các điều kiện thỏa mãn.

❖ Cài đặt giải thuật

```
void    insertElement (list *L, int pos, Item x)
{
int i;
   if  (Full (*L))
    printf("Danh sach day!");
   else
       if ((pos <0) || ( pos>= maxlist))
           printf("\n Vi tri chen khong phu hop");
       else
    {
                for  (i=L->count;i>=pos;i--)
                        L-> element [i+1]= L-> element [i];
                L-> element [pos]=x;
                L->count++;
         }
}
```

a. Xóa (loại bỏ) một phần tử khỏi danh sách.

Phần tử bị xóa có thể ở các vị trí: Đầu tiên, phía trong hoặc cuối của danh sách.

Để xóa một phần tử ta chỉ cần dịch chuyển các phần tử từ cuối danh sách ngược lên đến vị trí pos.

❖ Giải thuật:

Input:

L *// là con trỏ trỏ có kiểu list*

pos *// chứa vị trí cần chèn*

x *// là biến có kiểu Item và chứa giá trị phần tử bị xóa*

Process:

Bước 1: Xét các trường hợp

- Nếu danh sách rỗng (hàm **Empty**==1):

Thông báo danh sách rỗng và kết thúc

- Nếu ví trí pos không hợp lệ (pos<0 hoặc pos>maxlist)

Thông báo vị trí không hợp lệ và kết thúc

- Nếu danh sách không rỗng và ví trí pos hợp lệ thì chuyển sang bước 2.

  Bước 2: Xóa phần tử và đưa giá trị phần tử này vào biến x.
- Đưa giá trị phần tử pos vào biến x.
- Dịch chuyển các phần tử từ vị trí cuối danh sách lên vị trí pos.
- Giảm giá trị biến count đi một.

Output: Phần tử pos đã được xóa và x là giá trị phần tử pos trong danh sách nếu các điều kiện thỏa mãn.

❖ Cài đặt giải thuật.

```
void    deleteElement (list *L, int pos, Item *x)
{
int i;
   if  (Empty (*L))
    printf("Danh sach rong!");
   else
       if ((pos <0) || ( pos>= maxlist))
           printf("\n Vi tri xoa khong phu hop!");
       else
       { *x=L-> element [pos];
                for  (i=pos;i<L->count;i++)
                        L-> element [i]= L-> element [i+1];
                L->count++;
         }
}
```

a. Sắp xếp các phần tử trong danh sách.

Muốn sắp xếp các phần tử trong danh sách ta phải dựa vào giá trị một trường khóa của phần tử.

Giải thuật sắp xếp được thực hiện bởi phép toán so sánh giá trị trường khóa sắp xếp của 2 phần tử với nhau (phần tử i với phần tử j), nếu không đúng trật tự sắp xếp thì đổi chỗ hai phần tử này cho nhau.

❖ Giải thuật:

Input:

- L là một danh sách cần sắp xếp theo thứ tự tăng dần của trường Khóa.
- count là số phần tử trong danh sách L.

Process:

Lặp lại công việc sau từ chi chỉ số i=0 cho đến chỉ số i=count-2

Lặp lại công việc sau từ chỉ số j=i+1 cho đến chỉ số cuối j=count-1

- So sánh trường khóa của phần tử Li với Lj.
- Nếu giá trị trường khóa Li>Lj thì hoán đổi vị trí phần tử Li và Lj cho nhau.

Output: L là danh sách đã được sắp xếp theo chiều thuận của trường khóa (chiều tăng đối với số, chiều từ điển đối với chuỗi).

❖ Cài đặt giải thuật.

```
void sortList(list   *L)
{     int i, j, temp;
       for (  i=0; i<L->count-1;i++)
       for (  j=i+1;j< L->count ;j++)
                  //sắp xếp theo chiều thuận
           if (L->Element[i].trường khóa>L->Element[j].trường khóa)
           {       temp= L->Element[i];
                   L->element[i]= L->element[j];
                   L->element[j]=temp ;
           }
}
```

a. Chương trình quản lý điểm sinh viên.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
const    int maxlist=100;
          //định nghĩa cấu trúc SinhVien
struct SinhVien
{  char masv[10];
   char Hten [35];
   float   LaptrinhCB, KientrucMT, MangMT, DiemTB;
};
```

```
//Định nghĩa cấu trúc danh sách
typedef         SinhVien     Item;
typedef struct list
{ Item          element[maxlist];
  int           count;
};
// ham khoi tao danh sach rong
void     initializeList (list   *L)
{
          L->count=0;
}
// ham kiem tra danh sach co rong khong?
int         EmptyList(list     L)
{
        return (L.count==0);
}
// ham kiem tra danh sach co rong khong?
int         FullList(list         L)
{
        return (L.count== maxlist);
}
//hàm tìm kiếm sinh viên theo masv
int    searchElement (list   L,  char x[])
{
int i=0;
while   (i<L.count)
           if (stricmp(x,L.element[i].masv)==0)
                  return(i);
           else
                  i++;
     return(-1);
}
        //hàm bổ sung một sinh viên mới vào danh sách
void    insertElement (list *L, int pos, Item x)
```

```
{
        int i;
          if  (FullList (*L))
                        printf("Danh sach day!");
          else
              if ((pos <0) || ( pos>= maxlist))
                  printf("\n Vi tri chen khong phu hop");
              else
                {
                        for  (i=L->count;i>=pos;i--)
                                L-> element [i+1]= L-> element [i];
                        L-> element [pos]=x;
                        L->count++;
                 }
}
 //hàm xóa một sinh viên khỏi danh sách
void    deleteElement (list *L, int pos, Item *x)
{
          int i;
      if  (EmptyList (*L))
                printf("Danh sach rong!");
      else
           if ((pos <0) || ( pos>= maxlist))
                 printf("\n Vi tri xoa khong phu hop!");
          else
        {*x=L-> element [pos];
                 for  (i=pos;i<L->count;i++)
                        L-> element [i]= L-> element [i+1];
                 L->count--;
          }
}
         //ham sap xep theo hten
void sortListHten(list  *L)
{    int i, j;
```

```
    SinhVien  temp;
    for (  i=0; i<L->count-1;i++)
      for (  j=i+1;j< L->count ;j++)
        if (stricmp(L->element[i].Hten,L->element[j].Hten)>0)
         {        temp=L->element[i];
                   L->element[i]= L->element[j];
                  L->element[j]=temp ;
         }
}
 //sap xep theo diem tb giam dan
void    sortListDTB(list  *L)
{     int i, j;
      SinhVien  temp;
      for (  i=0; i<L->count-1;i++)
          for (  j=i+1;j< L->count ;j++)
            if  (L->element[i].DiemTB<L->element[j].DiemTB)
           {       temp=L->element[i];
                   L->element[i]= L->element[j];
                   L->element[j]=temp ;
            }
}
        // ham nhap thong tin mot sinh vien
    void  NhapSinhVien (SinhVien *p)
     {   float f;
     char ht[35],ma[10];
     fflush(stdin);                    //ham xoa vung dem ban phim
    printf("\n ma sinh vien: ");  gets(ma);strcpy(p->masv,ma);
    fflush(stdin);
    printf("\n Ho ten: ");  gets(ht);strcpy(p->Hten,ht);
 //Nhap diem cho sinh viên
    printf("Diem Lap trinh CB: ");  scanf("%f",&f); p->LaptrinhCB=f;
    printf("Diem Kien truc MT: ");  scanf("%f",&f); p->KientrucMT=f;
    printf("Diem Mang MT: ");       scanf("%f",&f); p->MangMT=f;
 //Tinh diem trung bình
```

```
(*p).DiemTB=( p-> LaptrinhCB+ p-> KientrucMT+ p-> MangMT )/3;
}
//Dinh nghia ham hien thong tin mot ban ghi sinh vien
    void HienSinhVien (SinhVien sv)
      {
  printf ("\n Ma sinh vien  %10s", sv.masv);
  printf ("\n Sinh vien  %35s", sv.Hten);
  printf ("\n Diem Lap trinh CB : %4.1f", sv.LaptrinhCB);
  printf ("\n Diem Kien truc MT : %4.1f", sv.KientrucMT);
  printf ("\n Diem Mang MT : %4.1f", sv.MangMT);
  printf ("\n Diem TB : %4.1f", sv. DiemTB);
  getch();
    }
// hàm tạo danh sách sinh vien
    void CreateList (list  *L)
    {    int i=0;
         char kt;
         Item sv;
  do
   {       printf("nhap phan tu thu %d:",i+1); NhapSinhVien(&sv);
           L->element[i]=sv;
           L->count++;i++;
           printf("ban nhap tiep khong c/k? "); fflush(stdin);kt=getchar();
       }
  while (kt!='k');
}
//hàm in danh sách sinh vien
void PrintList(list L)
{    int i;
  if (EmptyList(L))
      {     printf("Danh sach rong");
            return;
      }
    for (i=0; i<L.count;i++)
```

```
            HienSinhVien(L.element[i]);
      }
          //ham menu
     int menu()
      {    int chon;
        clrscr();
        printf("\n MOT SO THUAT TOAN VE DANH SACH \n\n");
        printf(" Nhan so tuong ung de chon chuc nang:\n");
        printf(" 1. Khoi tao danh sach \n");
        printf(" 2. Nhap cac phan tu cho danh sach\n");
        printf(" 3. In cac phan tu cua danh sach\n");
        printf(" 4. Tim mot phan tu cua danh sach\n");
        printf(" 5. Bo sung mot phan tu moi vao danh sach \n");
        printf(" 6. Xoa mot phan tu cua mang\n");
        printf(" 7 Sap xep danh sach theo ho ten\n");
        printf(" 8 Sap xep danh sach theo diem tb giam dan");
        printf(" 0. Thoat khoi chuong trinh\n");
        printf("  Nhap so tuong ung: "); scanf("%d",&chon);
        return (chon);
      }
          //ham chính main
        void main()
          {  list *L;
int chon,pos;
char  ht1[35];
char kt;
Item  sv;
do
            {  chon=menu();
               switch (chon)
                {
                    case 1:
        initializeList(L);
                          break;
```

```
        case 2:
                CreateList(L);
                break;
         case 3:
                printf("\n Danh sach sinh vien:\n\n") ;
                PrintList(*L);
                getch();
                break;
        case 4:
                 printf ("\n Nhap HT can tim: "); fflush(stdin);
                gets(ht1);
                 pos=searchElement(*L,ht1);
                if (pos!=-1)
                printf("\n Tim thay sinh vien tai vi tri % ",pos);
                else
          printf("khong tim thay sinh vien %s ",ht1);
                getch();
                 break;
         case 5:
                printf ("\n Nhap SV moi: ");
                NhapSinhVien(&sv);
                printf ("\n Nhap vi tri de chen : ");
  scanf("%d",&pos);
                insertElement(L,pos,sv);
                printf(" Da bo sung phan tu \n");
getch();
                break;
        case 6:
                 printf ("\n Nhap vi tri de xoa : ");
                scanf("%d",&pos);
                deleteElement(L,pos,&sv);
                 printf(" Da xoa phan tu \n");
  getch();
                break;
```

```
                case 7:
                        sortListHten(L);
                         printf(" Da sap xep theo ho ten\n");
                  getch();
                        break;
                case 8:
                        sortListDTB(L);
                         printf(" Da sap xep theo ho ten\n");
                  getch();
                        break;
                default:    printf ("\n ban chon sai roi! ");


            }
        } while (chon!=0);


    }
```

1.2.1. Nhận xét.

Nhược điểm:

- Thông thường maxlist chỉ là dự đoán, ước chừng (nhỏ thì thiếu, lớn thì thừa, lãng phí).
- Phép bổ sung hay loại bỏ tốn thời gian (vì phải dịch chuyển các phần tử).

Ưu điểm: Truy nhập tới các phần tử nhanh.

1.3. Danh sách liên kết.

Sử dụng con trỏ hoặc mối nối để tổ chức danh sách tuyến tính , mà ta gọi là danh sách móc nối (danh sách liên kết), chính là một giải pháp nhằm khắc phục các nhược điểm của cách cài đặt danh sách bằng mảng.

1.3.1. Cài đặt theo cấu trúc danh sách liên kết đơn.

1.3.1.1. Nguyên tắc:

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

info	link

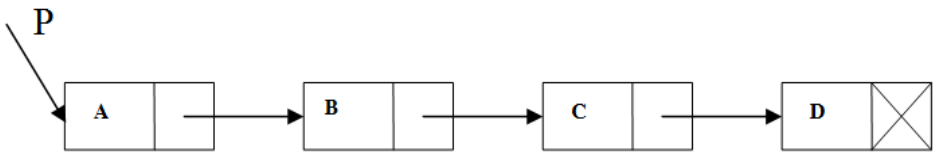
Trường **info** chứa thông tin của mỗi nút trong danh sách.

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

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

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

Nếu dùng mũi tên để chỉ mối nối, ta sẽ có hình ảnh một danh sách nối đơn như sau:



Hình 3.2: Hình ảnh một danh sách nối đơn

Dấu⊠ chỉ mối nối không và khi danh sách rỗng thì P=NULL.

Để lưu trữ danh sách liên kết đơn trong ngôn ngữ C, có thể dùng cấu trúc tự trỏ như sau:

```
struct node
{       ElementType     info;
        struct node*    link;
};
typedef struct node*  listnode;
```

Giải thích:

- Node: Là một cấu trúc gồm 2 trường (phần tử):
  - Info: là trường chứa dữ liệu của một node và có kiểu dữ liệu ElementType.
  - ElementType: là một kiểu dữ liệu bất kỳ trong ngôn ngữ C, nó có thể là các kiểu dữ liệu cơ sở như số nguyên (int), số thức (float),… hay kiểu dữ liệu bản ghi (cấu trúc),…

- link: là trường chứa địa chỉ của một node đứng ngay sau nó trong danh sách.
- struct node*: Là một kiểu con trỏ node.

- listnode: Là một kiểu dữ liệu con trỏ node.

Ví dụ 3.3: Khai báo một danh sách liên kết đơn mà mỗi nút chứa một số nguyên.

```
typedef  int     ElementType;
struct node
{       ElementType       info;
        struct node*      link;
};
typedef struct node*  listnode;
```

Ví dụ 3.4: Khai báo một danh sách liên kết đơn mà mỗi nút chứa một bản ghi sinhvien.

```
//định nghĩa bản ghi SinhVien
  struct SinhVien
    { char Hten [35];
      float   LaptrinhCB, KientrucMT, MangMT, DiemTB;
    };
    typedef      SinhVien    ElementType;
    struct node
    {       ElementType       info;
            struct node*
            link;
    };
   typedef struct node*  listnode;
```

1.3.1.2. Các phép toán cơ bản với danh sách liên kết đơn.

Đặc điểm của danh sách liên kết là sử dụng mối nối và con trỏ để tổ chức danh sách. Do đó ngoài các thao tác cơ bản của danh sách nói chung, ta cần thêm thao tác tạo và cấp phát bộ nhớ cho một nút mới.

a. Khởi tạo một danh sách rỗng.

Là thao tác đầu tiên và rất quan trọng đối với danh sách, nếu thiếu thao tác này chương trình sẽ gây lỗi khi thực hiện.

Khởi tạo một danh sách rỗng tức là gán giá trị NULL cho con trỏ chứa địa chỉ nút đầu tiên của danh sách.

```
void   initializeList (listnode   *P)
{
            *P=NULL;
}
```

b. Tạo và cấp phát bộ nhớ cho một nút.

Đặc điểm của danh sách liên kết là sử dụng biến con trỏ và cấp phát động, mỗi khi cần lưu thông tin vào một nút mới ta phải xin máy cấp phát số ô nhớ đủ cho một nút thông qua các câu lệnh sau:

❖ Giải thuật:

Input:

x *// giá trị trường info của nút mới*

Process:

Bước 1: Tạo nút mới.

- Xin cấp phát bộ nhớ cho nút mới (con trỏ **q**).
- Gán giá trị **x** cho trường **info** của con trỏ **q**.
- Gán giá trị **NULL** cho trường **link** của con trỏ **q.**

Bước 2: Trả kết quả (return (**q**)).

Output: Nút mới được tạo.

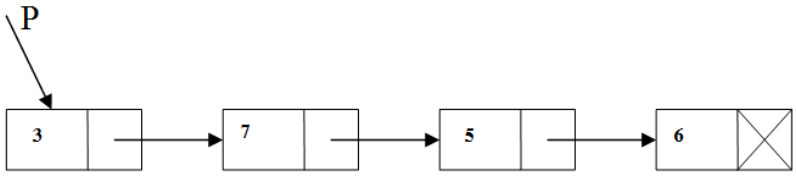
❖ Cài đặt giải thuật

```
// ham tao nut moi q
listnode newnode (ElementType      x)
{       listnode      q;
        q=(listnode)calloc (1,sizeof(struct node));
        q->info = x;
        q->link = NULL;
        return (q);
}
```

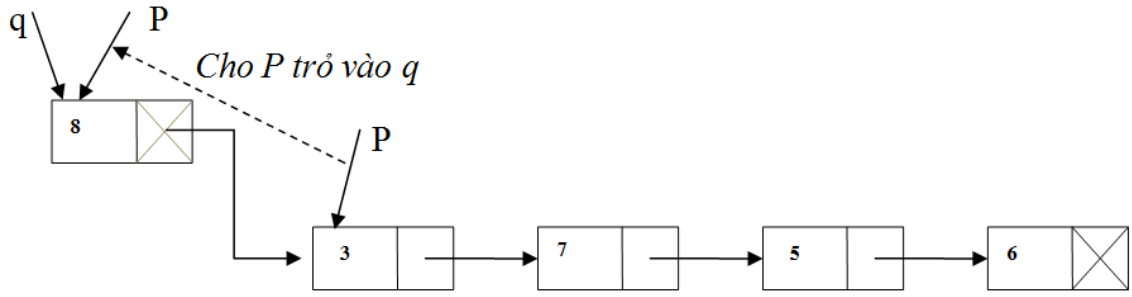
c. Chèn một nút mới vào trước nút đầu tiên của danh sách.

Mỗi danh sách liên kết luôn phải có một con trỏ lưu trữ địa chỉ của nút đầu tiên, ta không thể truy cập vào danh sách nếu không biết địa chỉ của nút đầu tiên này. Để chèn nút mới vào trước nút đầu tiên ta chỉ cần cho trường link

của nút mới *q* trỏ vào *P* (con trỏ chứa địa chỉ nút đầu tiên), sau đó cho con trỏ *P* trỏ vào nút mới *q*.



Hình 3.3a: Hình ảnh danh sách liên kết đơn được trỏ bởi con trỏ P



Hình 3.3b: Hình ảnh bổ sung nút mới q vào trước nút đầu tiên
(q trở thành nút đầu tiên sau phép bổ sung,
Ký hiệu NULL mờ được thay bằng địa chỉ khi bổ sung)

❖ Giải thuật:

Input:

P *// là con trỏ trỏ vào nút đầu tiên của danh sách*

x *// giá trị trường info của nút mới*

Process:

Bước 1: Tạo và cấp phát bộ nhớ cho nút mới **q.**

Bước 2: Xét các trường hợp.

- Nếu danh sách rỗng (**P**==NULL):

q là nút đầu tiên và duy nhất của danh sách (Cho **P** trỏ vào **q).**

- Nếu danh sách không rỗng (**P**!= NULL): Ghép nút q vào đầu danh sách bằng cách:

Cho trường link của con trỏ **q** trỏ vào **P.**

Cho con trỏ **P** trỏ vào **q.**

Output: Nút mới **q** là nút đầu tiên của danh sách.

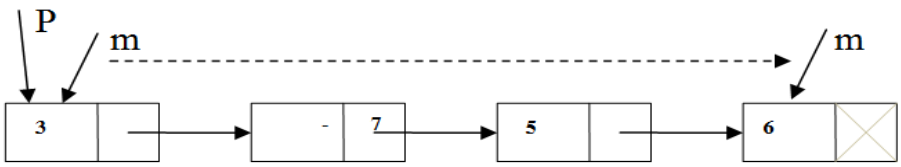
❖ Cài đặt giải thuật.

```
//ham chen truoc nut dau.
```

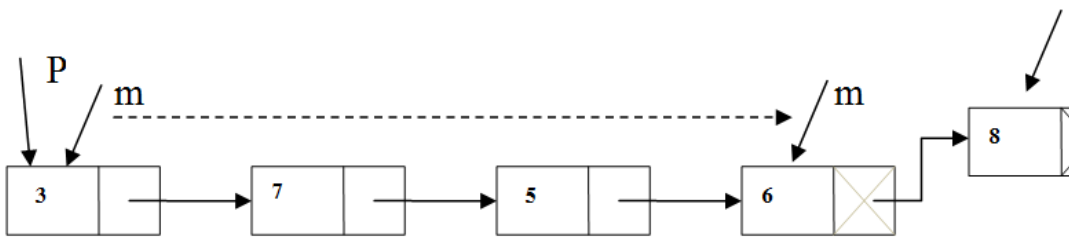
```
void Insertfirst ( listnode *P, ElementType   x)
  {             listnode   q;
                q= newnode(x);
                if (*P==NULL)
                        *P=q;
                else
                {       q->link=*P;
                        *P=q;
                }
  }
```

d. Chèn một nút mới vào sau nút cuối cùng của danh sách.

Muốn chèn một nút mới vào cuối danh sách ta phải tìm tới địa chỉ của nút cuối cùng, thao tác này cần một con trỏ phụ *m* bắt đầu từ nút đầu tiên và dịch chuyển dần qua từng nút cho tới nút cuối cùng (nút có trường link bằng NULL). Để gắn nút mới vào danh sách bằng cách cho trường link của con trỏ *m* trỏ vào nút mới *q*.



Hình 3.4a: Hình ảnh con trỏ phụ m tìm tới nút cuối cùng



Hình 3.4b: Hình ảnh bổ sung nút mới q vào sau nút cuối cùng
(ký hiệu NULL mờ được thay bằng địa chỉ sau khi bổ sung)

❖ Giải thuật:

Input:

P *// là con trỏ trỏ vào nút đầu tiên của danh sách*

x *// giá trị trường info của nút mới*

Process:

Bước 1: Tạo và cấp phát bộ nhớ cho một nút mới **q.**

Bước 2: Xét các trường hợp

- Nếu danh sách rỗng (**P**==NULL):

q là nút duy nhất của danh sách (Cho **P** trỏ vào **q)**

- Nếu danh sách không rỗng (**P**!= NULL):
  - Tìm tới nút cuối cùng của danh sách:

    Cho con trỏ phụ **m** tìm tới nút cuối cùng của danh sách
  - Gắn **q** vào cuối danh sách:

    Cho trường link của con trỏ **m** trỏ vào **q**

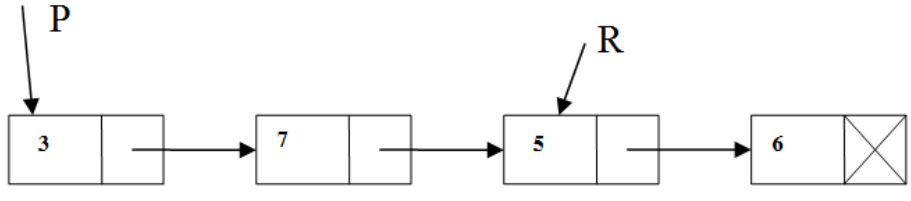
    Output: Nút mới **q** là nút cuối cùng của danh sách

❖ Cài đặt giải thuật.

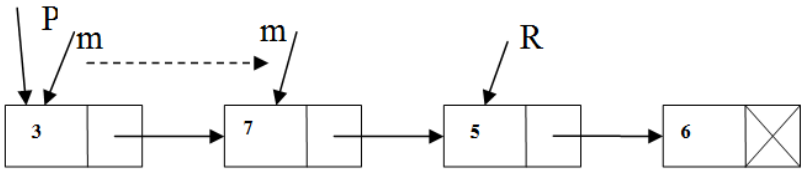
```
// ham chen sau nut cuoi
void InsertEnd ( listnode  *P, ElementType  x)
  {     listnode   q, m;
        q= newnode(x);
        if (*P==NULL)
           *P=q;
        else
        {       m=*P;
                while (m->link != NULL)
                        m=m->link;
                m->link=q;
        }
  }
```

e. Chèn một nút mới vào trước nút **R** trong danh sách.

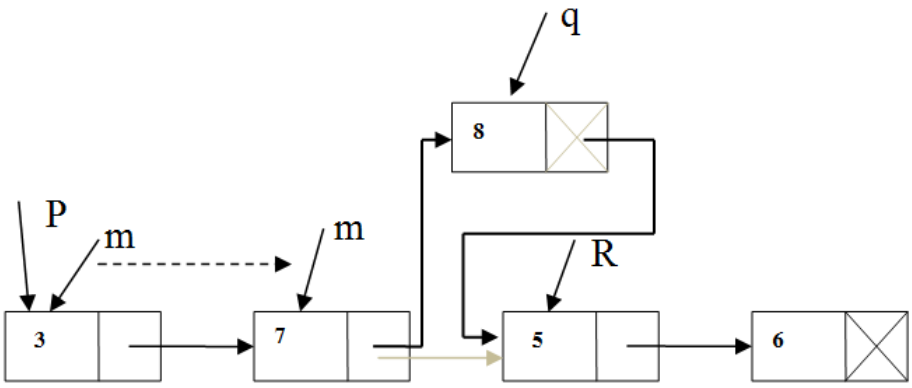
Thao tác này cần một con trỏ phụ *m,* bắt đầu từ nút đầu tiên và dịch chuyển dần qua các nút cho tới nút ngay trước *R* và gắn nút mới vào danh sách bằng cách cho trường link của con trỏ *m* trỏ vào nút mới *q,* rồi cho trường link của con trỏ *q* trỏ vào *R* .



Hình 3.5a: Hình ảnh danh sách liên kết có R trỏ vào nút bất kỳ



Hình 3.5b: Hình ảnh cho con trỏ phụ m tìm tới nút trước nút R



Hình 3.5c: Hình ảnh bổ sung nút mới q vào trước nút bất kỳ R
(mũi tên mờ, ký hiệu NULL mờ sẽ được thay bằng mũi tên đậm khi bổ sung)

❖ Giải thuật

Input:

P *// là con trỏ trỏ vào nút đầu tiên của danh sách*

R *// là con trỏ trỏ vào nút bất kỳ trong danh sách*

x *// giá trị trường info của nút mới*

Process:

Bước 1: Xét các trường hợp

- Nếu danh sách rỗng (P==NULL)

• Thông báo danh sách rỗng và kết thúc

- Nếu danh sách không rỗng (P!= NULL) chuyển sang bước 2.

Bước 2:

- Nếu **R** trỏ vào nút đầu tiên (**R==P**):

Chèn nút **q** vào đầu danh sách: Gọi hàm Insertfirst(p,x)

- Nếu **R** không trỏ vào nút đầu tiên (**R!=P**)

• Tạo và cấp phát bộ nhớ cho một nút mới **q**

• Tìm tới nút ngay trước nút **R** trong danh sách:

Cho con trỏ phụ **m** tìm tới nút ngay trước nút **R**

- Gắn **q** vào danh sách:

Cho trường **link** của con trỏ **q** trỏ vào **R**

Cho trường **link** của con trỏ **m** trỏ vào **q**

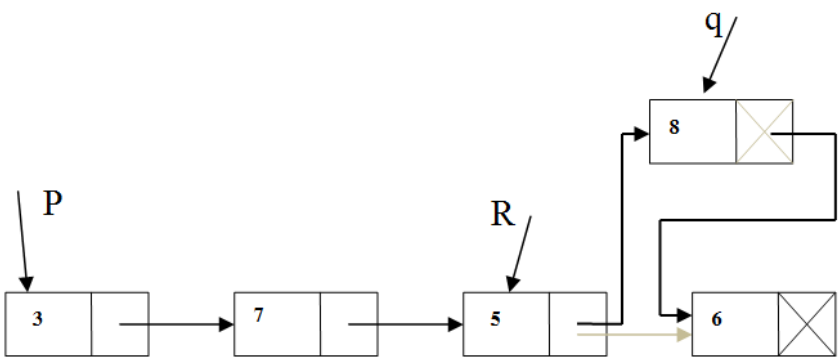
Output: Nút mới **q** được chèn vào danh sách

❖ Cài đặt giải thuật

```
//ham chen nút mới q vao truoc nut bat ky R
void InsertBefore ( listnode    *P, listnode    R,  ElementType  x)
  {             listnode   q, m;
        if (*P==NULL)
                printf("Danh sach rong");
        else
                if (R == *P)
                        Insertfirst(P,x);
                else
                {       q= newnode(x);
                        m=*P;
                        while (m->link != r)
                                m=m->link;
                        q->link=R;
                        m->link=q;
                }
    }
```

f. Chèn một nút mới vào sau nút **R** trong danh sách

Để chèn nút mới vào sau nút R ta chỉ cần gán giá trị trường link của *R* cho trường link của con trỏ *q,* sau đó cho link của *R* trỏ vào *q*.



Hình 3.6: Hình ảnh bổ sung nút mới q vào sau nút bất kỳ R
(mũi tên m được thay bằng các mũi tên đậm khi bổ sung nút mới q)

❖ Giải thuật

Input:

P *// là con trỏ trỏ vào nút đầu tiên của danh sách*

R *// là con trỏ trỏ vào nút bất kỳ trong danh sách*

x *//x giá trị trường info của nút mới*

Process:

Bước 1: Xét các trường hợp

- Danh sách rỗng (P==NULL)
  - Thông báo danh sách rỗng và kết thúc
- Danh sách không rỗng (P!=NULL), chuyển sang bước 2.

Bước 2: Chèn nút mới.

- Tạo và cấp phát bộ nhớ cho một nút mới **q**.
- Chèn nút mới **q** vào sau **R**.
  - Cho trường **link** của **q** trỏ vào trường link của của **R**.
  - Cho trường **link** của con trỏ **r** trỏ vào **q.**

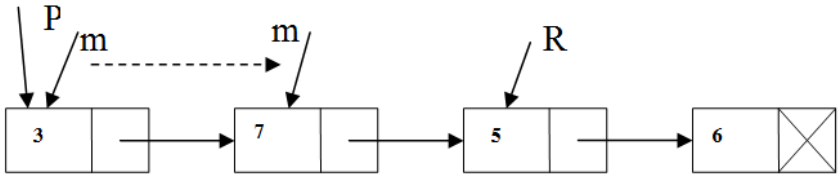
Output: Nút mới **q** được chèn vào danh sách **.**

❖ Cài đặt giải thuật.

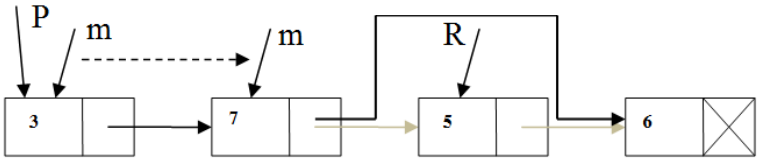
```
//ham chen sau mot nut R bat ky
void InsertAfter (listnode  *P,  listnode    R,  ElementType  x)
  { listnode   q;
        if (*P==NULL)
              printf("Danh sach rong");
        else
        {     q= newnode(x);
              q->link = R->link;
              R->link = q;
        }
  }
```

g. Xóa một nút trỏ bởi con trỏ **R** trong danh sách:

Thao tác này cần một con trỏ phụ *m,* bắt đầu từ nút đầu tiên dịch chuyển dần qua từng nút cho tới nút ngay trước *R*, tiếp đó cho trường link của *m* trỏ vào link của *R*.



Hình 3.7a: Hình ảnh con trỏ phụ m tìm tới nút trước R



Hình 3.7b: Hình ảnh xóa nút trỏ bởi con trỏ R
(các mũi tên mờ sẽ được thay bằng mũi tên đậm khi loại bỏ nút R)

❖ Giải thuật.

Input:

P *// là con trỏ trỏ vào nút đầu tiên của danh sách*

R *// là con trỏ trỏ vào nút cần xóa trong danh sách*

Process:

Bước 1: Xét các trường hợp

- Nếu danh sách rỗng (**P**==NULL):

  Thông báo danh sách rỗng và kết thúc.

- Nếu danh sách không rỗng (**P!**=NULL): Chuyển sang bước 2.

Bước 2: Xét các trường hợp

- Nếu **R** trỏ vào nút đầu tiên của danh sách (**R==P**):
  - Xóa nút **R:** Cho con trỏ **R** trỏ vào trường **link** của **R**
  - Giải phóng bộ nhớ cho nút **R;**
- Nếu **R** không trỏ vào nút đầu tiên (**R**!=**P**)
  - Tìm tới nút ngay trước nút **R**

    Cho con trỏ phụ **m** tìm tới nút ngay trước nút **R**.
  - Ngắt nút **R** khỏi danh sách

    Cho trường **link** của con trỏ **m** trỏ vào link của **R.**

- Giải phóng bộ nhớ cho **R.**

Output: Nút mới **R** đã được xóa khỏi danh sách**.**

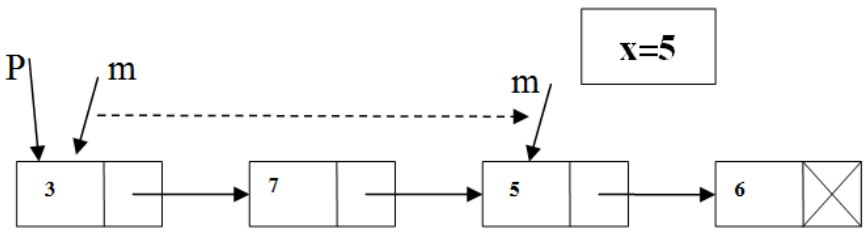
❖ Cài đặt giải thuật.

```
//ham xoa mot nut R
void    DeleteNode ( listnode  *P, listnode R)
  {         listnode    m;
            if (*P == NULL)
                  printf (" DANH SACH RONG");
            else
                  if (R == *P)
                  {       *P=(*P)->link;
                          free(R);
                  }
                  else
                  {       m=*P;
                          while (m->link != R)
                                m=m->link;
                          m->link=R->link;
                          free(R);
                  }
  }
```

h. Tìm một nút trong danh sách.

Thông thường trường info của mỗi nút cũng là một bản ghi (gồm nhiều trường). Muốn tìm kiếm một nút trong danh sách ta phải dựa vào giá trị một trường gọi là trường khóa của nút.

Giải thuật tìm kiếm được thực hiện bởi phép toán so sánh khóa tìm kiếm với giá trị trường khóa của từng nút, bắt đầu từ nút đầu tiên cho tới nút cần tìm hoặc đã hết danh sách mà không thấy. Kết quả trả ra là địa chỉ của nút được cần tìm hoặc giá trị NULL nếu không tìm thấy.



Hình 3.8: Hình ảnh tìm nút có giá trị trường khóa của info bằng x

❖ Giải thuật:

Input:

P *// là con trỏ trỏ vào nút đầu tiên của danh sách*

x */* là khóa tìm kiếm và có kiểu dữ liệu phù hợp với trường khóa của các nút trong danh sách*/*

Process:

Bước 1: Xét các trường hợp

- Nếu danh sách rỗng (**P**==NULL):

  Thông báo danh sách rỗng và kết thúc

- Nếu danh sách không rỗng (**P!**=NULL): Chuyển sang bước 2

Bước 2: Tìm nút có giá trị trường khóa của **info** bằng **x**

- Lặp lại công việc sau cho tới khi tìm thấy hoặc hết danh sách (từ nút đầu tiên):
  - Nếu giá trị trường khóa của info trùng với x thì return (địa chỉ của nút này);
  - Nếu khác thì dịch chuyển sang nút đứng sau.
- Không tìm thấy: return (NULL)

Output: Địa chỉ của nút được tìm thấy hoặc giá trị NULL nếu không tìm thấy.

❖ Cài đặt giải thuật

Giả thiết, hàm tìm kiếm xem trong danh sách nối đơn lưu trữ những số nguyên, nếu có một nút mà giá trị trường info bằng x (là một số cần tìm) thì hàm trả ra địa chỉ ô nhớ của nút đó, ngược lại hàm trả ra giá trị NULL.

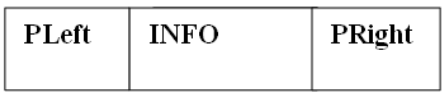
```
//ham tim nut co gia tri truong info=x
listnode    SearchNode ( listnode  P, int x)
  {             listnode     m;
                if (P == NULL)
                        printf (" DANH SACH RONG");
                else
                {       m=P;
```

```
        while (m !=NULL)
          if (x==m->info)
              return (m);
          else
              m=m->link;
      }
    return (NULL);
}
```

1.3.1. Cài đặt theo cấu trúc danh sách liên kết kép

1.3.2.1. Nguyên tắc

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



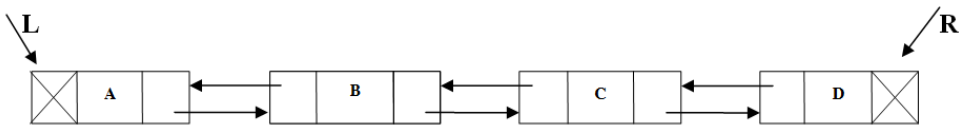
Trường INFO chứa thông tin của mỗi nút trong danh sách.

Trường PLeft chứa địa chỉ của nút đứng ngay trước, riêng nút đầu tiên không có nút đứng trước nên PLeft có giá trị NULL.

Trường Pright chứa địa chỉ của nút đứng ngay sau, riêng nút cuối cùng không có nút đứng sau nên PRight có giá trị NULL.

Để có thể truy nhập vào mọi nút trong danh sách ta cần phải có hai con trỏ L và R. Con trỏ L trỏ tới nút đầu tiên và con trỏ R trỏ tới nút cuối cùng.

Nếu dùng mũi tên để chỉ mối nối, ta sẽ có hình ảnh một danh sách nối đơn như sau:



Hình 3.8: Hình ảnh danh sách nối kép

Dấu⊠ chỉ mối nối không và khi danh sách rỗng thì L=R=NULL

Để lưu trữ danh sách liên kết kép trong ngôn ngữ C, có thể dùng cấu trúc tự trỏ như sau:

```
struct node
{       ElementType     info;
        struct node*    PLeft;
        struct node*    PRight;
};
typedef struct node*  DoubleListnode;
```

Giải thích:

- node: Là một cấu trúc gồm 3 trường gần giống như danh sách liên kết đơn, chỉ khác là có hai trường PLeft và PRight có kiểu struct node* chứa địa chỉ của nút đứng ngay trước và ngay sau nó trong danh sách.
- DoubleListnode: Là một kiểu dữ liệu con trỏ node.

Ví dụ 3.5: Khai báo một danh sách liên kết kép mà mỗi nút chứa một số nguyên.

```
typedef  int          ElementType;
struct node
{       ElementType             info;
        struct node*            PLeft;
        struct node*            PRight;
};
typedef struct node*  DoubleListnode;
```

1.3.2.2. Các phép toán cơ bản với danh sách liên kết kép

a. Khởi tạo một danh sách rỗng

Là thao tác đầu tiên và rất quan trọng đối với danh sách, nếu thiếu thao tác này chương trình sẽ gây lỗi khi thực hiện.

Khởi tạo một danh sách rỗng tức là gán giá trị NULL cho con trỏ L và R (con trỏ trỏ vào đầu danh sách và con trỏ trỏ vào cuối danh sách).

```
void   initializeListD (DoubleListnode  *L, DoubleListnode *R,)
{
        *L=*R=NULL;
}
```

a. Tạo và cấp phát bộ nhớ cho một nút

Muốn tạo và cấp phát bộ nhớ cho một nút mới ta cần xin máy cấp phát số ô nhớ đủ cho một nút thông qua các câu lệnh sau:

❖ Giải thuật:

Input:

x *// giá trị trường info của nút mới*

Process:

Bước 1: Tạo nút mới

- Xin cấp phát bộ nhớ cho nút mới (con trỏ **q**).
- Gán giá trị **x** cho trường **info** của con trỏ **q**.
- Gán giá trị **NULL** cho trường **Pleft** và **Pright** của con trỏ **q.**

Bước 2: Trả kết quả (return (**q**)).

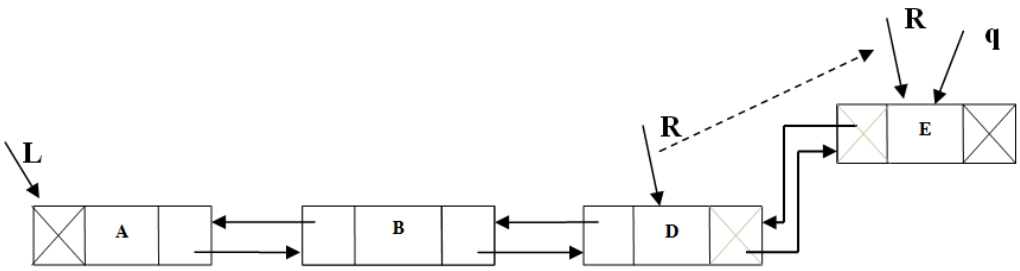
Output: Nút mới được tạo.

❖ Cài đặt giải thuật

```
// ham tao nut moi.
DoubleListnode newnode (ElementType   x)
{    DoubleListnode       q;
        q=(DoubleListnode)calloc (1,sizeof(struct node));
        q->info = x;
        q->PLeft = (q->PRight=NULL);
        return (q);
}
```

a. Chèn một nút mới vào sau nút cuối cùng của danh sách

Thao tác này chỉ cần cho cho trường PRight của con trỏ *R* trỏ vào *q,* tiếp theo cho trường PLeft của *q* trỏ vào *R***,** cuối cùng cho *R* trỏ vào *q*



Hình 3.9: Hình ảnh chèn nút mới q vào sau nút cuối

(ký hiệu NULL mờ sẽ được thay thế bởi các địa chỉ khi bổ sung nút mới q)

❖ Giải thuật:

Input:

L, R /* *là hai con trỏ trỏ vào nút đầu tiên và cuối*

cùng của danh sách/*

x *// giá trị trường info của nút mới*

Process:

Bước 1: Tạo và cấp phát bộ nhớ cho một nút mới **q**

Bước 2: Xét các trường hợp

- Nếu danh sách rỗng (**L**==R==NULL):

q là nút duy nhất của danh sách (Cho **L** và **R** trỏ vào **q)**

- Nếu danh sách không rỗng (**L**!= NULL): Gắn **q** vào cuối danh sách.
  - Cho trường **PRight** của con trỏ **R** trỏ vào **q**
  - Cho trường **PLeft** của **q** trỏ vào **R**
  - Cho **R** trỏ vào **q**

Output: Nút mới **q** là nút cuối cùng của danh sách

❖ Cài đặt giải thuật

```
// ham chen sau nut cuoi
void InsertEnd ( DoubleListnode  *L, DoubleListnode *R,
    ElementType  x)
  { DoubleListnode   q;
    q= newnode(x);
    if ((*L==NULL) && (*R==NULL))
       *L=(*R=q);
    else
          {       (*R)->PRight=q;
                  q->PLeft=*R;
                  *R=q;
          }
  }
```

a. Chèn một nút mới vào sau nút **m** trong danh sách.

Để chèn nút mới *q* vào sau nút *m,* ta cần một số thao tác theo trình tự sau:

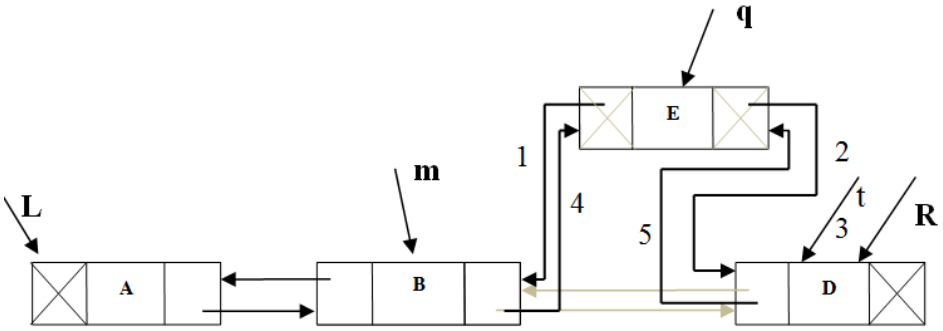
Cho trường PLeft của *q* trỏ vào *m* (1).

Cho trường PRight của con trỏ *q* trỏ vào PRight của *m* (2).

Dùng con trỏ phụ *t* trỏ vào PRight của *m* (3).

Cho trường PRight của con trỏ *m* trỏ vào *q* (4).

Cho Pleft của *t* trỏ vào *q* (5).



Hình 3.10: Hình ảnh chèn nút mới q vào sau nút m
(mũi tên mờ thể hiện các địa chỉ sẽ bị gỡ bỏ khi chèn nút mới q)

❖ Giải thuật

Input:

L, R*// là hai con trỏ trỏ vào nút đầu tiên và cuối cùng của danh sách.*

m *// là con trỏ trỏ vào nút bất kỳ trong danh sách.*

x *//x giá trị trường info của nút mới.*

Process:

Bước 1: Xét các trường hợp

- Danh sách rỗng (L==R==NULL)

  Thông báo danh sách rỗng và kết thúc.

- Danh sách không rỗng (L!=NULL), chuyển sang bước 2

Bước 2: Chèn nút mới

- Tạo và cấp phát bộ nhớ cho một nút mới **q.**
- Chèn nút mới **q** vào sau **m.**
  - Cho trường **PLeft** của **q** trỏ vào **m**
  - Cho trường **PRight** của con trỏ **q** trỏ vào **PRight** của **m**
  - Dùng con trỏ phụ **t** trỏ vào **PRight** của **m**
  - Cho trường **PRight** của con trỏ **m** trỏ vào **q**
  - Cho **Pleft** của **t** trỏ vào **q**

Output: Nút mới **q** được chèn vào danh sách**.**

Cài đặt giải thuật.

```
//ham chen sau mot nut m bat ky
void InsertAfter (DoubleListnode  *L, DoubleListnode  *R,
                          DoubleListnode m,ElementType  x)
{    DoubleListnode   q, t;
        if ((*L==NULL) &&(*R==NULL))
            printf("Danh sach rong");
        else
        {     q= newnode(x);
            q->PLeft = m;
            q->PRight = m->PRight;
            t=m->PRight;
            m-> PRight= q;
            t->PLeft = q;
        }
}
```

a. Xóa một nút trỏ bởi con trỏ **m** trong danh sách.

Để xóa nút *m,* ta cần một số thao tác theo trình tự sau:

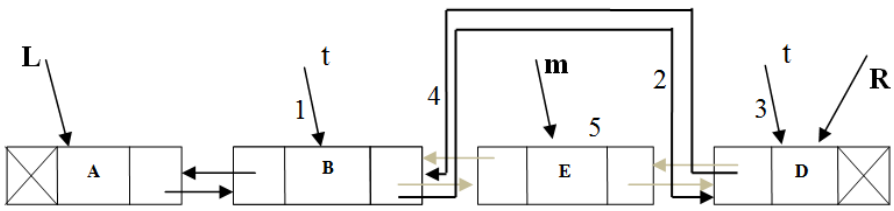
Cho con trỏ phụ *t* trỏ vào trường PLeft của *m* (1).

Cho con trỏ PRight của *t* trỏ vào PRight *m* (2).

Cho con trỏ phụ *t* trỏ vào trường PRight của *m* (3).

Cho con trỏ PLeft của *t* trỏ vào Pleft *m* (4).

Giải phóng bộ nhớ cho *m* (5).



Hình 3.11: Hình ảnh xóa nút trỏ bởi con trỏ m

(các mũi tên mờ thể hiện các địa chỉ sẽ được thay thế bằng các mũi tên đậm)

❖ Giải thuật

Input:

L, R */* là hai con trỏ trỏ vào nút đầu tiên và nút cuối cùng của danh sách*/*

m *// là con trỏ trỏ vào nút cần xóa trong danh sách*

Process:

Bước 1: Xét các trường hợp

- Nếu danh sách rỗng (**L**==R==NULL):

  Thông báo danh sách rỗng và kết thúc.

- Nếu danh sách không rỗng (**L!**=NULL): Chuyển sang bước 2.

Bước 2: Xét các trường hợp

- Nếu danh sách chỉ có một nút, m trỏ vào nút đó

  (L==R==m):

  Gán giá trị NULL Cho con trỏ L và R

- Nếu **m** trỏ vào nút đầu tiên của danh sách (**m==L**):
  - Cho con trỏ **L** trỏ vào trường **PRight** của **L**.
  - Cho con trỏ **PLeft** của **L** trỏ vào NULL.
  - Giải phóng bộ nhớ cho nút **m.**
- Nếu **m** trỏ vào nút cuối cùng của danh sách (**m==R**):
  - Cho con trỏ **R** trỏ vào trường **PLeft** của **R**.
  - Cho con trỏ **PRight** của **R** trỏ vào NULL.
  - Giải phóng bộ nhớ cho nút **m.**
- Nếu **m** trỏ vào nút bất kỳ trong danh sách
  - Cho con trỏ phụ **t** trỏ vào trường PLeft của **m.**
  - Cho con trỏ PRight của **t** trỏ vào PRight **m.**
  - Cho con trỏ phụ **t** trỏ vào trường PRight của **m.**
  - Cho con trỏ PLeft của **t** trỏ vào Pleft **m.**
  - Giải phóng bộ nhớ cho **m.**

Output: Nút mới **m** đã được xóa khỏi danh sách **.**

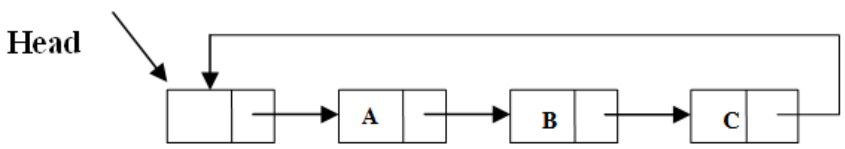
❖ Cài đặt giải thuật.

```
//ham xoa mot nut m
void DeleteNode ( DoubleListnode  *L, DoubleListnode  *R,
                  DoubleListnode  m)
{    DoubleListnode  t;
```

```
        if ((*L ==NULL) && (*R==NULL))
                printf (" DANH SACH RONG");
        else
                if ((L==R)&& (*R==m))
                {       L=(R=NULL);
                        free(L);
                }
                else
                {       t=m->PLeft;
                        t->PRight=m->PRight;
                        t=m->PRight;
                        t->PLeft=m->PLeft;
                        free(m);
                }
  }
```

1.3.1. Cài đặt theo cấu trúc danh sách liên kết nối vòng.

Là một kiểu cải tiến của danh sách nối đơn và khác ở chỗ là trường Link của nút cuối cùng không chứa giá trị NULL mà là địa chỉ của nút đầu tiên. Việc cải tiến này giúp ta có thể truy nhập vào bất cứ nút nào trong danh sách và từ một nút bất kỳ (không cần phải từ nút đầu tiên).. Trong danh sách nối vòng nút nào cũng có thể coi là nút đầu tiên và cũng có thể coi là nút cuối cùng. Với phép ghép, tách cũng có những thuận lợi nhất định.



Hình 3.12: Hình ảnh danh sách nối vòng

Nhược điểm: Trong khi xử lý nếu không lưu ý sẽ rơi vào chu trình không kết thúc (vì không biết chỗ kết thúc danh sách).

Khắc phục bằng cách đưa thêm vào một nút đặc biệt gọi là nút đầu danh sách (list head node). Trường INFO của nút này không chứa dữ liệu của phần tử nào (chỉ chưa dấu hiệu đánh dấu) và con trỏ Head bây giờ trỏ tới nút đầu danh sách này, nó cho phép ta truy nhập vào danh sách. Bằng các này nếu

danh sách rỗng thì vẫn còn lại dấu tích (hay về mặt hình thức thì danh sách không bao giờ rỗng. Với quy ước LINK(HEAD)=HEAD.

2. Cài đặt danh sách theo các cấu trúc đặc biệt (ngăn xếp, hàng đợi).

2.1. Ngăn xếp (Stack).

2.1.1. Khái niệm.

Là một kiểu danh sách tuyến tính đặc biệt mà phép bổ sung và phép loại bỏ luôn thực hiện ở một đầu gọi là đỉnh (Top). Nguyên tắc "vào sau ra trước" của Stack đã đưa tới một tên gọi khác: Danh sách kiểu LIFO (Last In First Out). Stack có thể rỗng hoặc bao gồm một số phần tử.

2.1.1. Các thao cơ bản của Stack.

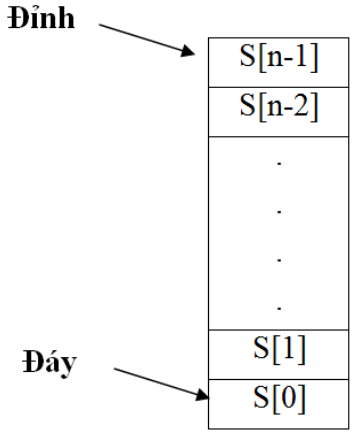
- Tạo lập stack.
- Bổ sung một phần tử vào Stack.
- Loại bỏ một phần tử khỏi Stack.

2.1.2. Cài đặt Stack bằng mảng.

Trong cài đặt Stack bằng mảng, giả sử độ dài tối đa của Stack (maxsize) là một số n nào đó, các phần tử của Stack có kiểu dữ liệu Item (Item có thể là các kiểu dữ liệu đơn giản hoặc kiểu dữ liệu có cấu trúc). Mỗi phần tử của Stack được biểu diễn bằng một bản ghi gồm 2 trường. Trường thứ nhất top chứa địa chỉ phần tử đỉnh của Stack, trường thứ hai info là mảng các Item có kích thước maxsize (kích thức của Stack).

Cụ thể: Dùng một véctơ lưu trữ S gồm n phần tử nhớ kế tiếp.

- Top chứa địa chỉ của phần tử đỉnh của Stack →Phải biết top khi muốn truy nhập (top sẽ có giá trị biến đổi khi stack hoạt động)
- Khi stack rỗng top=0. Nếu mỗi phần tử của stack ứng với một từ máy thì khi một phần tử mới được bổ sung vào Stack, top sẽ tăng lên 1 và ngược lại top sẽ giảm đi 1.



Hình 3.13: Hình ảnh Stack cài đặt bằng mảng

a. Khai báo cấu trúc Stack bằng ngôn ngữ C.

```
const   int maxsize=100          //kích thước tối đa của Stack
typedef struct Stack
{  int     top;                  //top chứa địa chỉ của phần tử đỉnh
ItemType info[maxsize];      //info chứa nội dung của một phần tử
};
```

Giải thích:

Stack: Là một cấu trúc gồm 2 trường:

- top: Là một số nguyên lưu trữ địa chỉ của phần tử đỉnh.
- info: Lưu dữ liệu của Stack và có kiểu dữ liệu mảng ItemType.
- ItemType: là một kiểu dữ liệu bất kỳ trong ngôn ngữ C, nó có thể là các kiểu dữ liệu cơ sở như số nguyên (int), số thức (float),… hay kiểu dữ liệu bản ghi (cấu trúc),…

Ví dụ 3.6 : Khai báo cấu trúc Stack để lưu trữ 100 số nguyên.

```
const   int maxsize=100          //kích thước tối đa của Stack
typedef      int     ItemType;   //TtemType có kiểu int
typedef struct StackI
{  int          top;             //top chứa địa chỉ của phần tử đỉnh
ItemType info[maxsize];      //info chứa nội dung của một phần tử
};
```

a. Thao tác khởi tạo Stack.

Thao tác đầu tiên là khởi tạo một Stack rỗng, tức là gán giá trị **-1** cho trường **top.**

```
void    Initialize (Stack   *S)
 {
        S ->top=-1;
 };
```

b. Kiểm tra xem stack có rỗng không.

Khi lấy một phần tử ra khỏi Stack cần kiểm tra xem Stack có rỗng không. Nếu Stack rỗng việc lấy phần tử ra khỏi Stack sẽ được kết thúc.

Hàm Empty trả ra giá trị 1 (TRUE) nếu Stack rỗng và giá trị 0 (FALSE) nếu Stack không rỗng.

```
int  Empty(Stack  S)
{
    return (S.top==-1);
}
```

c. Kiểm tra xem stack có đầy không.

Khi bổ sung một phần tử vào Stack cần kiểm tra xem Stack có đầy không (tức là biến top đã đạt tới kích thước tối đa của Stack chưa). Nếu Stack đầy việc bổ sung phần tử vào Stack sẽ kết thúc.

Hàm Full trả ra giá trị 1 (TRUE) nếu Stack đầy và giá trị 0 (FALSE) nếu Stack không đầy.

```
int  Full (Stack  S)
{
     return (S.top= =maxsize-1);
}
```

d. Thao tác đẩy (bổ sung) một phần tử vào Stack.

Ta cần xét các trường hợp:

- Trường hợp Stack đầy:
  Thông báo Stack đầy và kết thúc
- Trường hợp Stack không đầy:
  - Tăng giá trị biến top thêm 1 phần tử
  - Đưa giá trị mới vào biến info

Cài đặt giải thuật:

```
void     Push ( Stack  *S, ItemType    x)
 {
          if  (Full(*S))
                   printf("\n Stack day");
          else
          {
                    S->top ++;
                    S->info[S->top] = x;
          }
   }
```

e. Lấy một phần tử khỏi Stack

Ta cần xét các trường hợp:

- Trường hợp Stack rỗng:

  Thông báo Stack rỗng và kết thúc
- Trường hợp Stack không rỗng:
  - Lấy giá trị từ biến info ra
  - Giảm giá trị biến top đi 1 phần tử

Cài đặt giải thuật:

```
void    Pop( Stack *S, ItemType *x)
{
        if  (Empty(*S))
                printf("\n Stack rong");
        else
          {     *x= S->info[S->top];
                S->top--;
 }
}
```

Hạn chế của việc cài đặt Stack bằng mảng là ta phải dự đoán trước kích thước tối đa của ngăn xếp (maxsize), Nếu dự đoán ít thì thiếu, dự đoán nhiều thì lãng phí ô nhớ. Để khắc phục hạn chế này ta có thể sử dụng danh sách liên kết để cài đặt ngăn xếp.

f. Ví dụ sử dụng các thao tác của Stack để viết chương trình chuyển đổi một số hệ 10 sang hệ 2.

```
#include <stdio.h>
#include <conio.h>
//#include <stdio.h>
const    int maxsize=100;          //kích thước tối đa của Stack
typedef         int     ItemType;          //ItemType có kiểu int
typedef struct Stack
{   int          top;                //top chứa địa chỉ phầnn tử đỉnh
   ItemType   info[maxsize];      //info chứa nội dung phần tử
};
//khoi tao stack rong
void     Initialize (Stack    *S)
```

```
  {
  S ->top=-1;
  };
//kiem tra stack co rong khong
int  Empty(Stack  S)
{
    return (S.top==-1);
}
//kiem tra stack co day khong
int  Full (Stack  S)
{
     return (S.top==maxsize-1);
}
//day 1 phan tu vào stack
void     Push ( Stack  *S, ItemType    x)
 {
          if  (Full(*S))
                   printf("\n Stack day");
          else
          {
                    S->top ++;
                    S->info[S->top] = x;
          }
   }
// lay 1 phan tu khoi stack
void      Pop( Stack *S, ItemType *x)
{
  if  (Empty(*S))
                   printf("\n Stack rong");
  else
    {     *x= S->info[S->top];
           S->top--;
     }
}
```

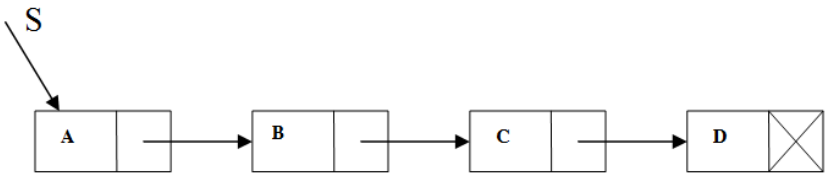
```
//đổi số hệ 10 sang hệ 2 và đẩy vào Stack
void doiso(Stack *S, ItemType  so)
{
  while (so!=0)
  {  Push (S,so%2);
     so=so/2;
  }
}
//Lấy từng chữ số hệ 2 ra khỏi Stack và in ra màn hình
void inso(Stack *S )
{ ItemType so;
  while (!Empty(*S))
  {
    Pop (S,&so);
    printf("%d", so);
  }
// hàm chính main
}
void main()
{    //int so;
  ItemType so;
     Stack *S;
     Initialize (S);
     printf("nhap so:"); scanf("%d", &so);
     doiso(S,so);
     inso(S );
   getch();
}
```

2.1.1. Cái đặt Stack bằng danh sách liên kết đơn.

Đặc điểm của Stack là việc truy nhập chỉ được thực hiện ở một đầu. Điều này khá gần gũi với danh sách liên kết đơn, việc bổ sung và loại bỏ một phần tử được thực hiện khá đơn giản khi nó ở đầu danh sách.

Để cài đặt Stack bằng danh sách liên kết, ta dùng một con trỏ top luôn trỏ vào đầu danh sách và qui ước nút đầu danh sách là đỉnh, nút cuối cùng của danh

sách là đáy Stack. Khác với cài đặt Stack bằng mảng ta coi như Stack có kích thước vô hạn (chỉ phụ thuộc vào dung lượng bộ nhớ của máy tính), chỉ cần kiểm tra tình trạng Stack rỗng khi loại bỏ một phần tử khỏi Stack.



Hình 3.14: Hình ảnh một Stack cài đặt bằng danh sách nối đơn

a. Khai báo cấu trúc Stack.

```
struct node
{       ElementType         info;
        struct node*        link;
};
typedef struct node*  Stacknode;
typedef struct
{       Stacknode    top;
} Stack;
```

Giải thích:

- node: Là một cấu trúc gồm 2 trường (phần tử):
  - info: là trường chứa dữ liệu của một node và có kiểu dữ liệu ElementType.
  - ElementType: là một kiểu dữ liệu bất kỳ trong ngôn ngữ C, nó có thể là các kiểu dữ liệu cơ sở như số nguyên (int), số thức (float),… hay kiểu dữ liệu bản ghi (cấu trúc),…
  - link: là trường chứa địa chỉ của một node đứng ngay sau nó trong danh sách.
- struct node* , Stacknode: Là một kiểu dữ liệu con trỏ node
- Stack: Là một kiểu cấu trúc mà trường top có kiểu Stacknode được dùng để chứa địa chỉ của nút đầu tiên của Stack.

Ví dụ 3.7: Khai báo một Stack mà mỗi nút chứa một số nguyên

```
typedef  int          ElementType;
struct node
{       ElementType             info;
        struct node*            link;
```

```
};
typedef struct node* Stacknode;
typedef struct
{       Stacknode    top;
} Stack;
```

b. Khởi tạo Stack.

Khởi tạo một Stack rỗng, tức là gán giá trị **NULL** cho trường **top**

```
void     Initialize (Stack    *S)
{
        S ->top=NULL;
};
```

c. Kiểm tra xem stack có rỗng không.

Hàm Empty trả ra giá trị 1 (TRUE) nếu Stack rỗng và giá trị 0 (FALSE) nếu Stack không rỗng.

```
int  Empty(Stack  S)
{
   return (S.top==NULL);
}
```

d. Đẩy (bổ sung) một phần tử vào Stack.

Thao tác này bao gồm những công việc sau:

- Xin cấp phát ô nhớ cho một nút mới **q.**
- Đưa giá trị mới vào trường info của con trỏ **q.**
- Gắn nút **q** vào đầu danh sách.
- Cho trường top của con trỏ **S** trỏ vào **q.**

Cài đặt giải thuật:

```
void     GetNode ( Stack  *S, ItemType    x)
{       Stacknode    q;
        q=( Stacknode) malloc (sizeof(struct node));
        q->info=x;
        q->link=S->top;
        S->top=q;
}
```

e. Lấy một phần tử khỏi Stack.

Ta cần xét các trường hợp:

- Trường hợp Stack rỗng:
  Thông báo Stack rỗng và kết thúc
- Trường hợp Stack không rỗng:
  - Cho con trỏ phụ **q** trỏ vào nút đầu tiên của Stack
  - Lấy giá trị từ biến **info** ra
  - Cho trường **top** của con trỏ **S** trỏ vào trường **link** của **q**
  - Giải phóng ô nhớ cho **q**

Cài đặt giải thuật:

```
void    RemoveNode( Stack *S, ItemType *x)
{       Stacknode    q;
        if  (Empty(*S))
                printf("\n Stack rong");
        else
          {     q=S->top;
                x=q->info;
        S->top=q->link;
        free(q);
          }
}
```

2.1.2. Ứng dụng của Stack.

- Stack thường được dùng để giải quyết các vấn đề có cơ chế LIFO.
- Stack thường được dùng để giải quyết các vấn đề trong trình biên dịch của

các ngôn ngữ lập trình như:

- kiểm tra cú pháp của các câu lệnh trong ngôn ngữ lập trình.
- Xử lý các biểu thức toán học: kiểm tra tính hợp lệ của các dấu trong

ngoặc một biểu thức, chuyển biểu thức từ dạng trung tố (infix) sang dạng hậu tố (postfix), tính giá trị của biểu thức dạng hậu tố.

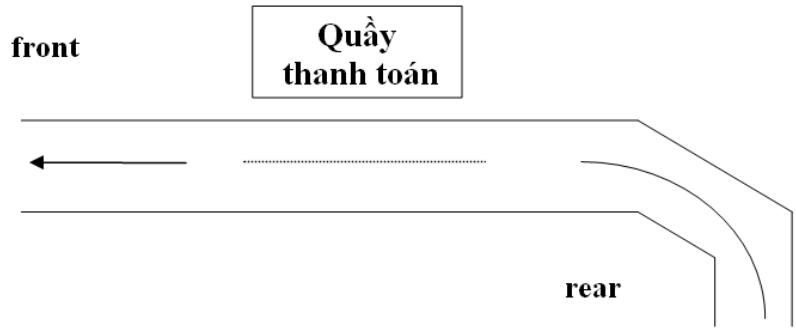
- Xử lý việc gọi các chương trình con.

- Stack thường được sử dụng để chuyển một giải thuật đệ qui thành giải thuật không đệ qui (khử đệ qui).

2.2. Hàng đợi (Queue)

2.2.1. Khái niệm

Khác với Stack Queue là một DSTT mà phép bổ sung được thực hiện ở một đầu gọi là lối sau (rear) và phép loại bỏ thực hiện ở một đầu khác gọi là lối trước (front). Cơ cấu của queue giống như một hàng đợi vào ở một đầu, ra ở đầu khác nghĩa là vào trước thì ra trước như: Quầy bán vé, xếp hàng lên xuống máy bay, một chồng hồ sơ hay một dãy các lệnh đang chờ xử lý, … . Vì vậy Queue còn được gọi là một danh sách kiểu FIFO (First in First out)



2.2.2. Các thao cơ bản của Queue.

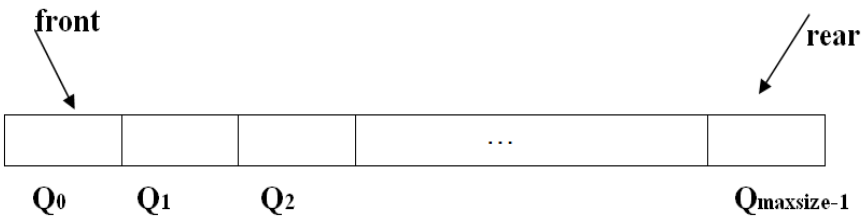
- Tạo lập Queue.
- Bổ sung một phần tử vào Queue.
- Loại bỏ một phần tử khỏi Queue.

2.2.3. Cài đặt Queue bằng mảng.

Khi phép bổ sung và loại bỏ thường xuyên tác động vào Queue, đến một lúc nào đó ta không thể thêm vào Queue được nữa dù mảng còn nhiều phần tử trống (các vị trí trước front) trường hợp này ta gọi là Queue bị tràn. Trong trường hợp toàn bộ mảng đã chứa các phần tử của Queue ta gọi là Queue bị đầy. Để khắc phục tình trạng Queue bị tràn, người ta tổ chức Q dạng mảng vòng tròn, nghĩa là Q_0 đứng ngay sau Q_{n-1}.

Trong cài đặt Queue bằng mảng cũng gần giống với Stack, chỉ khác là mỗi phần tử của Queue được biểu diễn bằng một bản ghi gồm 4 trường. Ba trường front, rear, count chứa địa chỉ phần tử đầu tiên, địa chỉ phần tử cuối cùng và số lượng phần tử thực sự của Queue, trường thứ tư info là mảng các Item có kích thước maxsize (kích thức của Queue).

Cụ thể: Dùng một véctơ lưu trữ Q gồm maxsize phần tử nhớ kế tiếp.



Hình 3.15 : Hình ảnh véc tơ lưu trữ Queue

- front chứa địa chỉ phần tử đầu tiên của Queue → Phải biết front khi muốn

lấy ra một phần tử.

- rear chứa địa chỉ phần tử cuối cùng của Queue → Phải biết rear khi muốn

bổ sung một phần tử.

- Khi *Queue rỗng* front = rear = -1, count=0. Nếu mỗi phần tử của Queue ứng với một từ máy thì khi bổ sung hay loại bỏ một phần tử thì font và rear cũng sẽ tăng thêm 1, còn count tăng theem1 khi bổ sung và giảm đi 1 khi loại bỏ một phần tử.

a. Khai báo cấu trúc Queue bằng ngôn ngữ C

```
const   int maxsize=100
typedef struct Queue
{  int                front, rear, count;
ItemType        info[maxsize];
};
```

Ví dụ 3.8 : Khai báo cấu trúc Stack để lưu trữ 100 số nguyên.

```
const   int maxsize=100                 //kích thước tối đa của Stack
typedef     int     ItemType;           //ItemType có kiểu int
typedef struct QueueInt
{  int                front, rear, count;
ItemType        info[maxsize];
};
```

a. Thao tác khởi tạo Queue.

Thao tác đầu tiên là khởi tạo một Stack rỗng, tức là gán giá trị **-1** cho trường **front** và **rear**

```
void    Initialize (QueueInt    *Q)
```

```
{
        Q ->front=-1;
      Q ->rear=-1;
      Q->count=0;
};
```

b. Kiểm tra xem Queue có rỗng không.

Queue rỗng khi biến count có giá trị nhỏ hơn hay bằng 0.

Hàm Empty trả ra giá trị 1 (TRUE) nếu Queue rỗng và giá trị 0 (FALSE) nếu Queue không rỗng.

```
int  Empty(Queue  Q)
{
   return (Q.count <= 0);
}
```

c. Kiểm tra xem Queue có đầy không.

Tình trạng Queue đầy khi **count = maxsize-1,** ta không thể bổ sung phần tử mới vào Queue được.

Hàm Full trả ra giá trị 1 (TRUE) nếu Queue đầy và giá trị 0 (FALSE) nếu Queue không đầy.

```
int  Full (Queue  Q)
{
    return ((Q.count)= = maxsize-1);
}
```

d. Thao tác bổ sung một phần tử vào Queue.

Ta cần xét các trường hợp:

- Trường hợp Queue đầy:

  Thông báo Queue đầy và kết thúc
- Trường hợp Queue không đầy:
  - Nếu Queue tràn thì gán giá trị 0 cho biến rear.
  - Nếu Queue không tràn thì tăng giá trị biến rear thêm 1.
  - Đưa giá trị mới vào biến info.
  - Tăng biến count thêm 1.

Cài đặt giải thuật:

```
void     Insert(Queue *Q, ItemType    x)
{
```

```
        if  (Full(*Q))
                printf("\n Queue day");
          else
           {     if (Q->rear==maxsize-1)
                        Q->rear=0;
                  Else
                        Q->rear++;
                  Q->info[Q->rear] = x;
                  Q->count++;
           }
    }
```

e. Loại bỏ một phần tử khỏi Queue.

Ta cần xét các trường hợp:

- Trường hợp Queue rỗng:

  Thông báo Queue rỗng và kết thúc

- Trường hợp Queue rỗng sau khi loại bỏ:

  Gán giá trị -1 cho front

- Trường hợp Queue không rỗng:
  - Lấy giá trị từ biến info ra
  - Nếu front=maxsize-1 thì gán giá trị 0 cho front
  - Ngược lại giảm front đi 1
  - Giảm count đi 1.

Cài đặt giải thuật:

```
    void    Delete(Queue *Q, ItemType *x)
    {       if  (Empty(*Q))
                    printf("\n Queue rong");
            else
              {     *x= Q->info[Q->front];
                    if (Q->count=1)
                          Initialize (Q);
                    else
                         {        Q-> front++;
                                  Q->count--;
                          }
```

```
        }
    }
```

Hạn chế của việc cài đặt Queue bằng mảng cũng giống như Stack tức là ta phải dự đoán trước kích thước tối đa của Queue (maxsize), Nếu dự đoán ít thì thiếu, dự đoán nhiều thì lãng phí ô nhớ. Để khắc phục hạn chế này ta có thể sử dụng danh sách liên kết để cài đặt Queue.

2.2.2. Cái đặt Queue bằng danh sách liên kết đơn.

Đặc điểm của Queue là loại bỏ ở một đầu và bổ sung ở đầu khác. Để cài đặt Queue bằng danh sách liên kết, ta dùng hai con trỏ front và rear, một luôn trỏ vào đầu danh sách, một luôn trỏ vào cuối danh sách. Ta cũng chỉ cần kiểm tra tình trạng Queue rỗng khi loại bỏ một phần tử khỏi Queue mà không cần kiểm tra tình trạng Queue đầy.

a. Khai báo cấu trúc Queue.

```
struct node
{       ElementType     info;
        struct node*    link;
};
typedef struct node*  Queuenode;
typedef struct
{       Queuenode  front, rear;
} Queue;
```

Giải thích:

- node: Là một cấu trúc gồm 2 trường (phần tử):
    - info: là trường chứa dữ liệu của một node và có kiểu dữ liệu ElementType.
    - ElementType: là một kiểu dữ liệu bất kỳ trong ngôn ngữ C, nó có thể là các kiểu dữ liệu cơ sở như số nguyên (int), số thức (float),… hay kiểu dữ liệu bản ghi (cấu trúc),…
    - link: là trường chứa địa chỉ của một node đứng ngay sau nó trong danh sách.
- struct node* , Queuenode: Là một kiểu dữ liệu con trỏ node.
- Queue Là một kiểu cấu trúc mà con trỏ fron luôn trỏ vào đầu danh sách và con trỏ rear luôn trỏ vào cuối danh sách.

Ví dụ 3.9: Khai báo một Queuemà mỗi nút chứa một số nguyên.

```
typedef  int              ElementType;
struct node
{       ElementType       info;
        struct node*      link;
};
typedef struct node*  Queuenode;
typedef struct
{       Queuenode  front, rear;
} Queue;
```

b. Thao tác khởi tạo Queue.

Khởi tạo một Queue rỗng, tức là gán giá trị **NULL** cho trường **front** và **rear.**

```
void    Initialize (Queue    *Q)
{
        Q ->front=NULL;
        Q ->rear=NULL;
};
```

c. Kiểm tra xem Queue có rỗng không.

Với Queue để loại bỏ một nút sẽ được thực hiện ở một đầu và ta gọi là đầu danh sách, con trỏ front luôn trỏ vào nút này, do đó khi Queue rỗng con trỏ front sẽ có giá trị NULL.

Hàm Empty trả ra giá trị 1 (TRUE) nếu Queue rỗng và giá trị 0 (FALSE) nếu Queue không rỗng.

```
int  Empty(Queue  Q)
{
   return (Q.front==NULL);
}
```

d. Bổ sung một nút vào Queue.

Thao tác này bao gồm những công việc sau:

- Xin cấp phát ô nhớ cho một nút mới **p.**
- Đưa giá trị mới vào trường info của con trỏ **p.**
- Gán giá trị NULL cho trường link của **p.**
- Gắn nút **p** vào cuối danh sách.
- Cho trường rear của con trỏ **Q** trỏ vào **p.**

- Nếu **p** là nút duy nhất của danh sách thì cho con trỏ front trỏ vào nút này.

Cài đặt giải thuật:

```
void    InsertNode ( Queue  *Q, ItemType    x)
{       Queuenode  p;
        p=( Queuenode) malloc (sizeof(struct node));
        p->info=x;
        p->link=NULL;
        Q->rear->link=p;
        Q->rear= Q->rear->link;
        if ( Empty(*Q))
                Q->front=Q->rear;
}
```

e. Xóa một nút khỏi Queue.

Ta cần xét các trường hợp:

- Trường hợp Queue rỗng:

  Thông báo Queue rỗng và kết thúc.

- Trường hợp Queue không rỗng:
  - Lấy giá trị từ biến **info** ra
  - Nếu Q rỗng sau khi loại bỏ thì gọi hàm Initialize (Q)
  - Ngược lại cho trường **front** của con trỏ **Q** trỏ vào trường **link** của **front.**
  - Giải phóng ô nhớ cho nút này.

Cài đặt giải thuật:

```
void    RemoveNode( Queue *Q, ItemType *x)
{       Queuenode  p;
        if  (Empty(*S))
                printf("\n Stack rong");
        else
                  {     p=Q->front;
                x=p->info;
                if ( Q->front == Q->rear)
                        Initialize (Q);
        else
```

```
            Q=Q->front->link;
        free(p);
          }
    }
```

2.2.3. Ứng dụng của Queue.

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

- Quản lý in trên mạng, nhiều máy tính yêu cầu in đồng thời và ngay cả một

  máy tính cũng yêu cầu in nhiều lần, khi đó các lệnh yêu cầu in phải được xếp vào một hàng đợi, lệnh nào yêu cầu trước được thực hiện trước.

- Các giải thuật duyệt theo chiều rộng một đồ thị có hướng hoặc vô

hướng cũng dùng hàng đợi để quản lý các nút đồ thị. Các giải thuật đổi biểu thức trung tố thành hậu tố, tiền tố.

Một tập các lệnh chờ thực hiện bởi hệ điều hành máy tính cũng tuân theo nguyên tắc của Queue…

CÂU HỎI VÀ BÀI TẬP CHƯƠNG 3

1) Hãy viết chương trình thực hiện các công việc sau:
   a. Sử dụng cấu trúc mảng để định nghĩa một danh sách lưu trữ các số nguyên.
   b. Viết hàm nhập danh sách số nguyên từ bàn phím, lưu trữ các số nguyên trong danh sách theo thứ tự nhập vào.
   c. Viết hàm nhập danh sách số nguyên từ bàn phím, lưu trữ các số nguyên trong danh sách theo thứ tự ngược với thứ tự nhập vào.
   d. Viết hàm in ra màn hình các số nguyên trong danh sách.
   e. Viết hàm tìm kiếm một số nguyên có giá trị được nhập từ bàn phím.
   f. Viết hàm xóa khỏi danh sách một số nguyên có giá trị được nhập từ bàn phím (gợi ý sử dụng hàm tìm kiếm ở câu trên để tìm vị trí phần tử).
   g. Viết hàm bổ sung một số nguyên vào sau phần tử có vị trí i trong danh sách.
   h. Viết hàm bổ sung một số nguyên vào trước phần tử có vị trí i trong danh sách.
   i. Tạo một menu cho phép lựa chọn thực hiện các công việc trên.

2) Sử dụng danh sách liên kết đơn để viết lại chương trình ở câu 1.
   Lưu ý: Hàm bổ sung ở mục g và h có thể thây vị trí i bằng địa chỉ một nút trong danh sách.

3) Hãy viết chương trình thực hiện các công việc sau:
   a. Định nghĩa một cấu trúc danh sách sinh viên bằng mảng, thông tin về mỗi sinh viên gồm:

```
struct Sinhvien
 {
        char Masv[10];
        char HoTen[30];
        float Diem;
        char Loai[10];
  };
```

   b. Viết hàm cập nhập thông tin cho trường loai theo nguyên tắc.

Điểm	Xếp loại
Từ 9 đến 10	Giỏi
Từ 7 đến 8	Khá

Từ 5 đến 6 Trung bình

Nhỏ hơn 5 yếu

c. Viết hàm nhập danh sách sinh viên từ bàn phím, lưu trữ các sinh viên trong danh sách theo thứ tự nhập vào.
d. Viết hàm nhập danh sách sinh viên từ bàn phím, lưu trữ các sinh viên trong danh sách theo thứ tự ngược với thứ tự nhập vào.
e. Viết hàm in ra màn hình các phần tử trong danh sách theo thứ tự của nó trong danh sách theo dạng sau:

HO VA TEN	DIEM	XEPLOAI
Nguyen Van A	7	Kha
Ho Thi B	5	Trung binh
Dang Kim C	4	Yeu

...

f. Viết hàm tìm kiếm một sinh viên có mã sinh viên được nhập từ bàn phím
g. Viết hàm xóa khỏi danh sách một sinh viên có mã sinh viên được nhập từ bàn phím (gợi ý sử dụng hàm tìm kiếm ở câu trên để tìm)
h. Viết hàm bổ sung một sinh viên mới vào sau một sinh viên được chỉ ra trong danh sách.
i. Viết hàm bổ sung một sinh viên mới vào trước một sinh viên được chỉ ra trong danh sách.
j. Tạo một menu cho phép lựa chọn thực hiện các công việc trên.

4) Sử dụng danh sách liên kết đơn để viết lại chương trình ở câu 3

5) Viết hàm nhập vào từ bàn phím các số nguyên, lưu trữ nó trong một danh sách có thứ tự tăng dần, theo cách sau: Với mỗi số được nhập vào hàm phải tìm vị trí thích hợp để chèn nó vào danh sách cho đúng thứ tự. Viết hàm cho trường hợp danh sách được cài đặt bằng mảng và cài đặt bằng danh sách liên kết.

6) Viết hàm loại bớt các phần tử trùng nhau chỉ giữ lại 1 phần tử trong một danh sách có thứ tự không giảm trong 2 trường hợp: Cài đặt bằng mảng và cài đặt bằng danh sách liên kết.

7) Viết hàm đảo ngược một danh sách trong cả 2 trường hợp là lưu trữ bằng mảng và lưu trữ bằng danh sách liên kết.

8) Viết hàm xóa khỏi danh sách các số nguyên những số nguyên chẵn, trong cả 2 trường hợp là lưu trữ bằng mảng và lưu trữ bằng danh sách liên kết.

9) Hãy sử dụng các thao tác của Stack để viết chương trình chuyển đổi một số hệ 10 sang một hệ khác (hệ 2, hệ 8, hệ 16). Cài đặt Stack theo mảng và theo danh sách liên kết.

10) Viết hàm đảo ngược một Stack.

11) Viết hàm đảo ngược một Queue.

CHƯƠNG 4
CÁC PHƯƠNG PHÁP SẮP XẾP CƠ BẢN

Mục tiêu:

- Trình bày được khái niệm bài toán sắp xếp;
- Mô phỏng được giải thuật, cách cài đặt, cách đánh giá giải thuật của một số phương pháp sắp xếp cơ bản;
- Giải được các bài toán sắp xếp sử dụng các phương pháp sắp xếp
  đã khảo sát.

1. Định nghĩa bài toán sắp xếp

1.1. Đặt vấn đề

Sắp xếp là quá trình bố trí lại các phần tử của một tập đối tượng nào đó, theo một thứ tự ấn định. Như thứ tự tăng dân (hay giảm dần) đối với một dãy số, thứ tự từ điển đối với các chữ,...

Trong cuộc sống thường xuyên xuất hiện tình huống đòi hỏi dữ liệu phải được sắp xếp theo một trật tự, như muốn tra từ điển, muốn tìm kiếm một số điện thoại, hay đơn giản hơn sắp xếp danh sách học sinh, sinh viên của một lớp học,…

Đối với các ứng dụng tin học, yêu cầu sắp xếp dữ liệu lưu trữ trong máy tính để tìm kiếm cho thuận lợi, sắp xếp các kết quả xử lý để in ra trên bảng biểu v.v...

Có hai phương pháp sắp xếp, phương pháp *sắp xếp trong:* Là các phương pháp tác động trên một tập các bản ghi lưu trữ đồng thời ở bộ nhớ trong hay còn gọi là *bảng* (table). Phương pháp *sắp xếp ngoài:* Là các phương pháp tác động trên một tập lớn các bản ghi lưu trữ ở bộ nhớ ngoài dưới dạng *tệp* (file).

1.2. Định nghĩa bài toán.

Bài toán được đặt ra ở đây là sắp xếp đối với một bảng gồm n bản ghi được lưu trữ ở bộ nhớ trong (sắp xếp trong). Mỗi bản ghi (đối tượng) bao gồm một số trường (thuộc tính) chứa dữ liệu có thể rất khác nhau, như bảng

điểm của sinh viên một lớp bao gồm các bản ghi lưu trữ các thông tin về mã sinh viên, họ tên sinh viên, điểm môn 1, môn 2,…, điểm trung bình.

Tuy nhiên, không phải toàn bộ các trường dữ liệu trong bản ghi đều được xem xét đến trong quá trình sắp xếp, mà thông thường chỉ một trường nào đó (hoặc một vài trường nào đó – nhưng trường hợp này ta sẽ không đề cập đến) được chú ý tới thôi và được gọi là *trường khoá*. Sắp xếp sẽ được tiến hành dựa vào giá trị của trường khoá này.

Một cách tổng quát, giải thuật sắp xếp bao gồm hai thao tác cơ bản là so sánh giá trị khóa của các bản ghi trong bảng và bố trí lại vị trí các bản ghi sao đúng thứ tự ấn định. Ở bảng điểm, với trường khóa là điểm trung bình khi sắp xếp ta so sánh các điểm trung bình của các bản ghi và bố trí lại các bản ghi sao cho đúng thứ tự tăng dần hay giảm dần của điểm trung bình. Với trường khóa là họ tên sinh viên, ta so sánh các chuỗi họ tên, nhưng thực chất của so sánh chuỗi ký tự là so sánh giá trị mã của ký tự tương ứng trong hai chuỗi với nhau, mà giá trị mã của ký tự cũng là một con số.

Để giúp làm đơn giản các giải thuật sắp xếp ta tạm coi bài toán sắp xếp trên một bảng gồm n bản ghi chỉ có một trường và đó là trường khóa. Như vậy, thao tác đổi chỗ bản ghi chỉ được thực hiện với trường khóa và giá trị khóa là các số nguyên, còn thứ tự sắp xếp là thứ tự tăng dần.

Thực tế có nhiều giải thuật sắp xếp, mỗi giải thuật đều được tối ưu trên một khía cạnh nào đó, có những giải thuật có những ưu điểm hơn những giải thuật khác. Trong khuôn khổ giáo trình sẽ giới thiệu 4 giải thuật sắp xếp đơn giản và một giải thuật sắp xếp nhanh (quick sort).

2. Phương pháp sắp xếp chèn (Insertion sort).

2.1. Ý tưởng giải thuât Insertion sort.

Dựa theo kinh nghiệm của những người chơi bài. Khi có i-1 lá bài đã được sắp xếp đang ở trên tay, nếu rút thêm lá bài thứ i nữa thì cần so sánh lá bài mới i lần lượt với lá bài thứ (i- 1) , thứ (i – 2) ... để tìm ra "chỗ" thích hợp và "chèn" nó vào chỗ đó.

2.2. Mô tả giải thuật.

Input:

- K là một dãy khóa cần sắp xếp theo thứ tự tăng dần
- n là số lượng khóa trong dãy K
- Các khóa được đánh số từ K_0, K_1,…K_i,…K_{n-1}

Process:

Bước 1: Tạm coi khóa K_0 đã được sắp xếp.

Bước 2: Thực hiện sắp xếp

Lặp lại công việc sau từ khóa K_1 (i=1) cho đến hết dãy (K_{n-1})

- Bước 2.1: Gán giá trị K_i cho biến temp (là biến tạm) và i-1 cho j
- Bước 2.2: Lặp lại công việc sau cho tới khi (j<0) hoặc (temp >= Kj)
    - Chuyển giá trị K_j sang $K_{(j+1)}$
    - Giảm j đi một đơn vị
- Bước 2.3: Chuyển giá trị temp vào $K_{(j+1)}$.

Output: K là dãy khóa đã được sắp xếp theo chiều tăng dần

2.3. Cài đặt giải thuật.

```
void   InsertionSort (int    K[ ], int   n)
{ int   i, j, temp;
        for  (i=1 ; i<n; i++)
          {     temp=K[i];
                j = i-1;
                while  ((j>=0)) && (temp< K[j]))
                  {
                        K[j+1] = K[j];
                        j--;
                  }
                K[j+1]=temp ;
        }
```

2.4. Biểu diễn giải thuật.

Mô tả giải thuật với dãy khóa:

K: 6 4 9 3 7

n = 5

Trong bảng mô tả các số mờ là những số đã bị thay thế bằng giá trị mới (đậm).

temp=4

Dãy khóa K		6	4	9	3	7
Chỉ số (n=5)	-1	0	1	2	3	4
Lần 1: i=1 và j=i-1=0		4 6	6 4	9	3	7
	j<0 K[j+1]←temp Kết thúc lần 1	j	i	temp< K[j]: K[j+1] ← K[j]; j--;		

temp=9

Dãy khóa K		4	6	9	3	7
Chỉ số (n=5)	-1	0	1	2	3	4
Lần 2: i=2 và j=i-1=1		4	6	9	Coi dãy chỉ có 3 phần tử	
	Temp> K[j]: kết thúc lần 2		j	i		

temp=3

Dãy khóa K		3 4	4 6	6 9	9 3	7
Chỉ số (n=5)	-1	0	1	2	3	4
Lần 3: i=3 và j=i-1=2		4	6	9	3	Coi dãy chỉ có 4 phần tử
	j<0 K[j+1]←temp Kết thúc lần 3	j	j	j	i	
		temp< K[j]: K[j+1] ← K[j]; j--;				

temp=7

Dãy khóa K		3	4	6	9	7
Chỉ số (n=5)	-1	0	1	2	3	4
Lần 1: i=1 và j=i-1=0		3	4	6	7 9	9 7
	temp>K[j]: K[j+1] ← K[j]; Kết thúc lần 4			j	j	i
				temp< K[j]: K[j+1] ← K[j]; j--;		
Dãy khóa K đã được sắp xếp		3	4	6	7	9

3. Phương pháp sắp xếp chọn (Selection sort).

3.1. Ý tưởng giải thuật Selection sort.

Xuất phát từ khóa đầu dãy (K0), So sánh khóa này với các khóa đứng sau, nếu gặp khóa nhỏ hơn thì lấy khóa nhỏ hơn này so sánh với các khóa tiếp theo, cứ như vậy cho đến hết dãy (Kn-1). Đổi chỗ khóa nhỏ nhất với khóa đầu dãy.

Lặp lại tương tự với các phần tử tiếp theo (K1) cho đến khóa sát cuối (Kn-2)

Kết thúc ta được dãy đã sắp xếp.

3.2. Mô tả giải thuật.

Input:

- K là một dãy khóa cần sắp xếp theo thứ tự tăng dần.
- n là số lượng khóa trong dãy K.
- Các khóa được đánh số từ K0, K1,…Ki,…Kn-1

Process:

Lặp lại công việc sau từ khóa Ki (i=0) cho đến khóa sát cuối (Ki=n-2)

Bước 1: Gán giá trị i cho m (biến m lưu chỉ số j khi Kj< Ki)

Bước 2: Lặp lại công việc sau từ (j=1) cho đến hết dãy (j=n-1)

- So sánh Kj với Km
- Nếu Kj<Km thì gán giá trị j cho m

Bước 3: Hoán đổi giá trị của Ki và Km.

Output: K là dãy khóa đã được sắp xếp theo chiều tăng dần

3.3. Cài đặt giải thuật

```
void    SelectionSort (int    K[ ], int    n)
{ int   i, j, m, temp;
        for  (i=0 ; i<n-1; i++)
          {      m = i ;
                 for  (j=i+1 ; j<n; j++)
                   if   (K[j] < K[m])
                          m=j;
                 temp=K[i]; K[i]=K[m]; K[m]=temp ;
          }
}
```

3.4. Biểu diễn giải thuật.

Mô tả giải thuật với dãy khóa:

K: 6 4 9 3 7

n = 5

Trong bảng mô tả các số mờ là những số đã bị thay thế bằng giá trị mới (đậm).

Dãy khóa K	6	4	9	3	7
Chỉ số (n=5)	0	1	2	3	4
Lượt 1: i=0; m=i và j=i+1=1;	3 6	4	9	6 3	7
	i	j	j	j	j
	m ----→	m ----------→		m	
	K[m] > K[j]: m ← j; j++; j=n-1: Kết thúc lượt 1; i≠m: K[m] ⇔ K[i]				

Dãy khóa K	3	4	9	6	7
Chỉ số (n=5)	0	1	2	3	4
Lượt 2: i=1; m=i và j=i+1=2;	3	4	9	6	7
		i	j	j	j
		m			
	K[m] < K[j]: j++; j=n-1: Kết thúc lượt 2;				

Dãy khóa K	3	4	9	6	7
Chỉ số (n=5)	0	1	2	3	4
Lượt 3: i=2; m=i và j=i+1=3;	3	4	6 9	9 6	7
			i	j	j
			m ----→	m	
	K[m] > K[j]: m ← j; j++; j=n-1: Kết thúc lượt 3; i≠m: K[m] ⇔ K[i]				

Dãy khóa K	**3**	**4**	**6**	**9**	**7**
Chỉ số (n=5)	**0**	**1**	**2**	**3**	**4**
Lượt 4: i=3; m=i và j=i+1=4;	**3**	**4**	**6**	**7** 9	**9** 7
				i	**j**
				m ------→	**m**
	K[m] > K[j]: m ← j; j++; j=n-1: Kết thúc lượt 4; **i≠m: K[m] ⇔ K[i]**				
	Dãy khóa K đã được sắp xếp				
	3	**4**	**6**	**9**	**7**

4. Phương pháp sắp xếp đổi chỗ (Interchange sort).

4.1. Ý tưởng của giải thuật Interchange sort.

Xuất phát từ khóa đầu dãy (K0), So sánh khóa K0 với các khóa đứng sau, nếu gặp khóa nhỏ hơn thì hoán đổi giá trị cho nhau rồi lại tiếp tục so sánh K0 với các khóa tiếp theo, cứ như vậy cho đến hết dãy (Kn-1), sau lượt đầu, khóa nhỏ nhất được chuyển về vị trí K0

Lặp lại tương tự với các phần tử tiếp theo (K1) cho đến khóa sát cuối (Kn-2), ta được phần tử thứ 2 (K1) là khóa nhỏ thứ 2,…Kết thúc ta được dãy đã sắp xếp.

4.2. Mô tả giải thuật.

Input:

- K là một dãy khóa cần sắp xếp theo thứ tự tăng dần
- n là số lượng khóa trong dãy K
- Các khóa được đánh số từ K0, K1,…Ki,…Kn-1

Process:

Lặp lại công việc sau từ khóa Ki (i=0) cho đến khóa sát cuối (Ki=n-2)

Lặp lại công việc sau từ (j=i+1) cho đến hết dãy (j=n-1)

- So sánh Ki với Kj
- Nếu Ki>Kj thì hoán đổi giá trị của Ki và Kj cho nhau

Output: K là dãy khóa đã được sắp xếp theo chiều tăng dần.

4.3. Cài đặt giải thuật.

```
void InterchangeSort(int   K[], int   n)
```

```
{     int I, j, temp;
      for (  i=0; i<n-1;i++)
      for (  j=i+1;j<n ;j++)
         if (K[i]>K[j])
         {      temp=K[j];
                K[j]=K[j-1];
                K[j-1]=temp ;
         }
}
```

4.4. Biểu diễn giải thuật

Mô tả giải thuật với dãy khóa:

K: 6 4 9 3 8 2 7 5

n = 8

Trong bảng mô tả các số mờ là những số đã bị thay thế giá trị mới.

Dãy khóa K	6	4	9	3	7
Chỉ số (n=5)	0	1	2	3	4
Lượt 1: i=0; và j=i+1=1;	3 4 6	6 4	9	4 3	7
	i	j	j	j	j
	K[i] > K[j]: K[i] ⇔ K[j]			K[i] ⇔ K[j]	j=n-1: Kết thúc lượt 1;

Dãy khóa K	3	6	9	4	7
Chỉ số (n=5)	0	1	2	3	4
Lượt 2: i=1; và j=i+1=2;	3	4 6	9	6 4	7
		i	j	j	j
				K[i] > K[j]: K[i] ⇔ K[j]	j=n-1: Kết thúc lượt 2;

Dãy khóa K	3	4	9	6	7
Chỉ số (n=5)	0	1	2	3	4
Lượt 3: i=2; và j=i+1=3;	3	4	6 9	9 6	7
			i	j	j
				K[i] > K[j]: K[i] ⇔ K[j]	j=n-1: Kết thúc lượt 3;

Dãy khóa K	3	4	6	9	7
Chỉ số (n=5)	0	1	2	3	4
Lượt 4: i=3; và j=i+1=4;	3	4	6	7 9	9 7
				i	j
					K[i] > K[j]: K[i] ⇔ K[j]
	Kết thúc lượt 3				
Dãy khóa K đã săp xếp	3	4	6	7	9

5. Phương pháp sắp xếp nổi bọt (Bubble sort).

5.1. Ý tưởng giải thuật Bubble sort.

Xuất phát từ khóa cuối dãy (Kn-1), So sánh khóa này với các khóa đứng trước, nếu gặp khóa lớn hơn thì đổi chỗ 2 khóa này cho nhau. Như vậy trong lượt đầu (i=0) khoá có giá trị nhỏ nhất sẽ chuyển lên đỉnh. Đến lượt thứ hai (i=1) khoá có giá trị nhỏ thứ hai sẽ được chuyển lên vị trí thứ hai, ... cho đến lượt cuối cùng (i=n-2). Kết thúc ta được dãy đã sắp xếp.

Nếu hình dung dẫy khoá được đặt thẳng đứng thì sau từng lượt sắp xếp các giá trị khoá nhỏ sẽ " nổi " dần lên giống như các bọt nước nổi lên trong nồi nước đang sôi. Vì vậy phương pháp này thường được gọi bằng cái tên khá đặc trưng là sắp xếp kiểu nổi bọt (Buble sort).

5.2. Mô tả giải thuật.

Input:

- K là một dãy khóa cần sắp xếp theo thứ tự tăng dần
- n là số lượng khóa trong dãy K
- Các khóa được đánh số từ K0, K1,…Ki,…Kn-1

Process:

Lặp lại công việc sau từ khóa Ki (i=0) cho đến khóa sát cuối (Ki=n-2)

Lặp lại công việc sau từ khóa Kj (j=n-1) cho đến hết dãy (Kj=i+1)

- So sánh Kj với Kj+1
- Nếu Kj < Kj+1 thì Hoán đổi giá trị của Kj và Kj+1 cho nhau

Output: K là dãy khóa đã được sắp xếp theo chiều tăng dần.

5.3. Cài đặt giải thuật.

```
void  BubleSort (int  K[ ], int  n)
```

```
{       int   i, j, temp;
        for  (i=0 ; i<n-1; i++)
          for  (j=n-1 ; j<=i+1; j--)
                if   (K[j] < K[j-1])
                    {       temp=K[j]; K[j]=K[j-1]; K[j-1]=temp ; }
    }
```

5.4. Biểu diễn giải thuật.

Mô tả giải thuật với dãy khóa:

K: 6 4 9 3 8 2 7 5

n = 8

Trong bảng mô tả các số mờ là những số đã bị thay thế giá trị mới.

Dãy khóa K	Chỉ số i và j		Lượt 2	
3		0	3	
6	i=1	1	4 6	
4	j=i+1	2	6 4	K[j]<K[j-1]: K[j]⇔K[j-1]
9	↑ j	3	7 9	
7	j=n-1	4	9 7	K[j]<K[j-1]: K[j]⇔K[j-1]

Dãy khóa K	Chỉ số i và j		Lượt 1	
6	i=0	0	3 6	
4	j=i+1	1	6 3 4	K[j]<K[j-1]: K[j]⇔K[j-1]
9	↑ j	2	4 3 9	K[j]<K[j-1]: K[j]⇔K[j-1]
3	↑ j	3	9 3	K[j]<K[j-1]: K[j]⇔K[j-1]
7	j=n-1	4	7	

Dãy khóa K	Chỉ số i và j		Lượt 3	
3		0	3	
4		1	4	
6	i=2	2	6	Dãy đã được sắp xếp
7	j=i+1	3	7	
9	j=n-1	4	9	

Nhận xét:

Đối với một giải thuật, khi xét tới hiệu lực của nó người ta thường dựa vào hai tiêu chí chính là: Sự chiếm dụng bộ nhớ của giải thuật và đặc biệt là thời gian thực hiện của giải thuật. Tuy nhiên, cách đánh giá thời gian thực hiện giải thuật lại chủ yếu dựa vào phép toán tích cực nhất của giải thuật, mà giải thuật sắp xếp là phép toán so sánh giá trị khóa, còn phép toán chuyển đổi vị trí bản ghi cho đúng trật tự sắp xếp lại ít được đề cập đến, trên thực tế, thời gian thực hiện giải thuật sắp xếp cũng bị ảnh hưởng nhiều bởi tình trạng dãy khóa ban đầu (nếu dãy khóa có tình trạng ngược với chiều sắp xếp thì sẽ tốn thời gian hơn rất nhiều so với dãy khóa có tình trạng gần giống với chiều sắp xếp).

Do đó, để đánh giá thời gian thực hiện giải thuật sắp xếp, người ta sẽ đánh giá độ phức tạp tính toán của T(n) ở các trường hợp xấu nhất, tốt nhất và trung bình.

Người ta đã chứng minh được nếu chỉ dựa vào kết quả đánh giá T(n) ở trường hợp trung bình thì giải thuật Insertion sort tỏ ra hiệu quả hơn 3 giải thuật kia. Tuy nhiên, với n khá lớn thì độ phức tạp tính toán của 4 giải thuật trên đều có cấp $O(n^2)$.

6. Phương pháp sắp xếp nhanh (Quick sort).

So với các giải thuật sắp xếp ở trên, Quick sort là một giải thuật khá tốt, ra đời năm 1960 bởi C.A.R Hoare nhà khoa học máy tính người Anh. Nhược điểm của giải thuật là phải cài đặt bằng đệ qui và độ phức tạp tính toán ở trường hợp xấu nhất là $O(n^2)$, nhưng người ta đã chứng minh được trường hợp trung bình là $O(log_2(n))$ và đặc biệt khi n khá lớn Quick sort tỏ ra hiệu quả hơn hẳn các giải thuật trên.

6.1. Ý tưởng giải thuật Quick sort.

Quick sort còn được gọi với cái tên sắp xếp kiểu phân đoạn (partition sort). Ý tưởng cơ bản của Quick sort là dựa trên phép phân chia dãy khóa thành hai dãy con, muốn vậy cần có một khóa đặc biệt làm *khóa chốt*. Sắp xếp được thực hiện bằng cách so sánh từng khóa trong dãy với khóa chốt và được đổi vị trí cho nhau hoặc cho khóa chốt, để những khóa nhỏ hơn khóa chốt được chuyển về trước khóa chốt và nằm trong phân đoạn thứ nhất, các khóa lớn hơn khóa chốt được chuyển về phía sau khóa chốt và thuộc phân đoạn thứ hai. Khi việc đổi chỗ thực hiện xong thì dãy khóa được chia thành hai phân đoạn: Một đoạn gồm những khóa nhỏ hơn hoặc bằng khóa chốt, một đoạn gồm những khóa lớn hơn hoặc bằng khóa chốt.

Áp dụng kỹ thuật trên với mỗi phân đoạn khi nó được hình thành cho tới khi mỗi đoạn chỉ còn một phần tử và ta thu được dãy khóa đã sắp xếp.

Kỹ thuật chọn khóa chốt cũng là một vấn đề khá quan trọng của Quick sort. Có một số cách chọn khóa chốt như sau:

- Chọn khóa đầu tiên hoặc khóa cuối cùng của dãy khóa làm khóa chốt.
- Chọn khóa đứng giữa dãy làm khóa chốt.
- Chọn khóa trung vị trong 3 khóa: Đầu tiên, giữa và cuối cùng làm khóa chốt.
- …

Mỗi cách chọn khóa sẽ dẫn đến một giải thuật cụ thể khác nhau. Tuy nhiên, hiệu lực của mỗi giải thuật còn phụ thuộc vào tình trạng dữ liệu của dãy khóa ban đầu.

Sau đây chúng ta sẽ tìm hiểu giải thuật Quick sort với cách chọn khóa chốt là khóa đầu tiên của dãy còn các trường hợp khác bạn đọc tự tham khảo coi như bài luyện tập.

6.2. Mô tả giải thuật.

Input:

- K là một dãy khóa gồm n bản ghi cần sắp xếp theo thứ tự tăng dần
- Các khóa được đánh số từ K0, K1,…Ki,…Kn-1
- left, right là hai biến lưu chỉ số khóa đầu và cuối của một đoạn: Khởi

đầu left có giá trị bằng 0, còn right có giá trị bằng (n-1).

Process:

Bước 1: Khởi gán

- Biến **key** chứa giá trị khóa chốt (khóa đầu tiên của đoạn):

  key=K[left];
- Hai biến chỉ số i, j được dùng để lựa chọn khóa trong quá trình xử

lý và còn để kiểm soát chỉ số khóa đầu tiên vào chỉ số khóa cuối cùng của đoạn: Gán i= left; j=right;

Bước 2: Phân đoạn.

Các khóa trong dãy sẽ được so sánh với khóa chốt và sẽ đổi vị trí cho nhau, cho chốt nếu nó nằm không đúng vị trí: Mọi khóa nhỏ hơn khóa chốt phải được xếp ở vị trí trước chốt (đầu dãy) và mọi khóa lớn hơn khóa chốt phải được xếp ở vị trí sau chốt (cuối dãy). Việc đổi chỗ thực hiện song thì dãy được phân thành 2 đoạn.

Thực hiện lặp lại công việc sau cho đến khi (i>j)

- Tăng chỉ số i cho tới khi khóa K[i]>=key hoặc i>right (*i không được vượt quá chỉ số phần tử cuối của đoạn*).
- Giảm chỉ số j cho tới khi khóa K[j] <=key hoặc j<left (*j không được vượt quá chỉ số phần tử đầu của đoạn*).
- Nêú i<j thi`đổi vị trí K[i] ⇔ K[j]
- Tăng i; giảm j

Bước 3: Sắp xếp đoạn trước

Nếu (left < j) thì gọi hàm QUICK_SORT(K,left, j);

Bước 4: Sắp xếp đoạn sau

Nếu (i< right) thì gọi hàm QUICK_SORT(K, i, right);

Output: K là dãy khóa đã được sắp xếp theo chiều tăng dần

6.3. Cài đặt giải thuật.

```
void QuickSort(int   K[], int   left , int   right)
{int     i, j, key,temp;
         key= K[left];
         i = left; j = right;
         do
         {
                  while ((K[i] < key) && (i <= right)) i++;
                  while ((K[j] > key) && (j >=left)) j--;
                  if(i <=j)
```

```
        {
                temp=K[i];k[i]=k[j];k[j]=temp;
                 i++ ; j--;
         }
    } while(i <= j);
    if(left<j)
            QuickSort(K, left, j);
    if(i<right)
            QuickSort(K, i, right);
}
```

6.4. Biểu diễn giải thuật.

Mô tả giải thuật với dãy khóa:

K: 6 4 9 3 7

n = 5

Trong bảng mô tả các số mờ là những số đã bị thay thế giá trị mới.

Key=K[left]=6

Dãy khóa K		**6**	**4**	**9**	**3**	**7**
Chỉ số (n=5)	-1	0	1	2	3	4
Lượt 1: QuickSort(K, left=0, right=n-1) i=left; j=right;		**3** 6	**4**	**9**	**6** 3	**7**
		left				**right**
		i			j	**j**
		K[i]=key	**i<j: K[j]⇔K[j]; i++, j--**		K[j]<key	K[j]>key: j--
			(j) i	j (i)	j	**j**
			K[i]<key: i++;	K[j]>key;		
			K[j]<key	K[j]>key: j--		
	i>j: dãy chia thành 2 phân đoạn: left và j; i và right					
		(3	**4)**	**(9**	**6**	**7)**

Key=K[left]=3

Dãy khóa K		**(3**	**4)**	**(9**	**6**	**7)**
Chỉ số (n=5)	-1	0	1	2	3	4
Lượt 2: j<left: QuickSort(K, left=0, j=1) i=left; j=right		**3**	**4**			
		left	**right**			
		i	**j** (i)			
		K[i]=key	K[j]>key: j--			
	(j)	**j**	K[j]=key: i=j đổi giá trị K[i] ⇔K[j]: i++; j--;			
	i>j : Dãy chia thành 2 phân đoạn: left và j; i và right					
		(3)	**(4)**			
	j không nhỏ hơn left; i không nhỏ hơn right ⇨khóa đã đúng vị trí					
		3	**4**	**(9**	**6**	**7)**

Key=K[left]=9

Dãy khóa K		3	4	(9	6	7)
Chỉ số (n=5)	-1	0	1	2	3	4
Lượt 3: i<right:				7 9	6	9 7
QuickSort(K, i=2, right=4)				**left**		**Right**
				i		**j**
i=left; j=right				K[i]=key		K[j]<key
	i<j : Đổi giá trị K[i] ⇔K[j]: i++; j--;					
					(i) i	(i)
					K[i]<key: i++	K[i]=key
	i>j : Dãy chia thành 2 phân đoạn: left và j; i và right					
				(7	6)	(9)
	i không nhỏ hơn right ⇨khóa đã đúng vị trí					

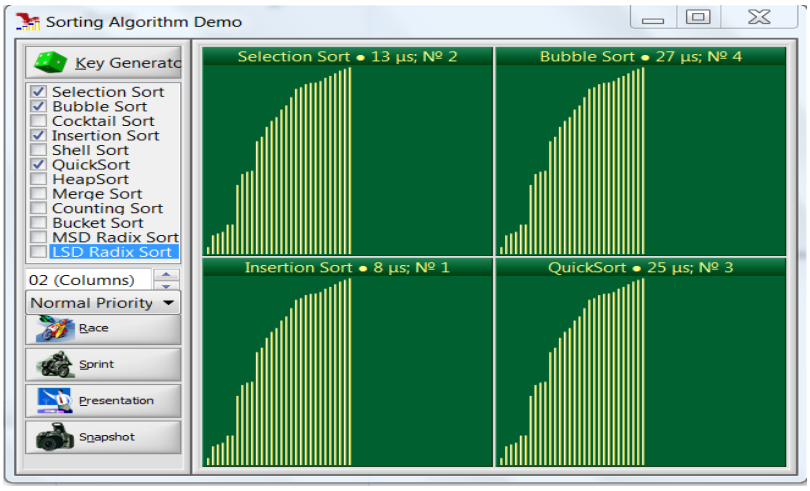
Key=K[left]=7

Dãy khóa K		3	4	(7	6)	9
Chỉ số (n=5)	-1	0	1	2	3	4
Lượt 4: j<left:				6 7	7 6	9
QuickSort(K, left=2, j=3)				**left**	**right**	
				i	**j**	
i=left; j=right				K[i]=key	K[j]<key	
	i<j : Đổi giá trị K[i] ⇔K[j]: i++; j--;					
				(j)	(i)	
	i>j : Dãy chia thành 2 phân đoạn: left và j; i và right					
				(6)	(7)	
	j không nhỏ hơn left; i không nhỏ hơn right ⇨khóa đã đúng vị trí					
Dãy khóa K đã sắp xếp		3	4	6	7	9

Bài toán sắp xếp rất thông dụng và đóng vai trò quan trọng trong thực tế cuộc sống cũng như nhiều ứng dụng trong ngành công nghệ thông tin. Giải thuật sắp xếp cũng rất phong phú, lựa chọn một giải thuật phù hợp khi cần cũng là điều đáng quan tâm. Các giải thuật sắp xếp đơn giản ở trên chỉ nên sử dụng chúng khi n nhỏ, Quick sort hiệu quả hơn khi n khá lớn, nhưng trường hợp xấu nhất vẫn có độ phức tạp tính toán là $O(n^2)$. Có nhiều giải thuật sắp xếp khác như Heap sort, merge sort,…đã đạt được hiệu quả cao, $O(n\log_2 n)$ cả khi n lớn và trường hợp xấu nhất, bạn đọc có thể tìm hiểu thêm các giải thuật này để thấy được sự phong phú của nó.

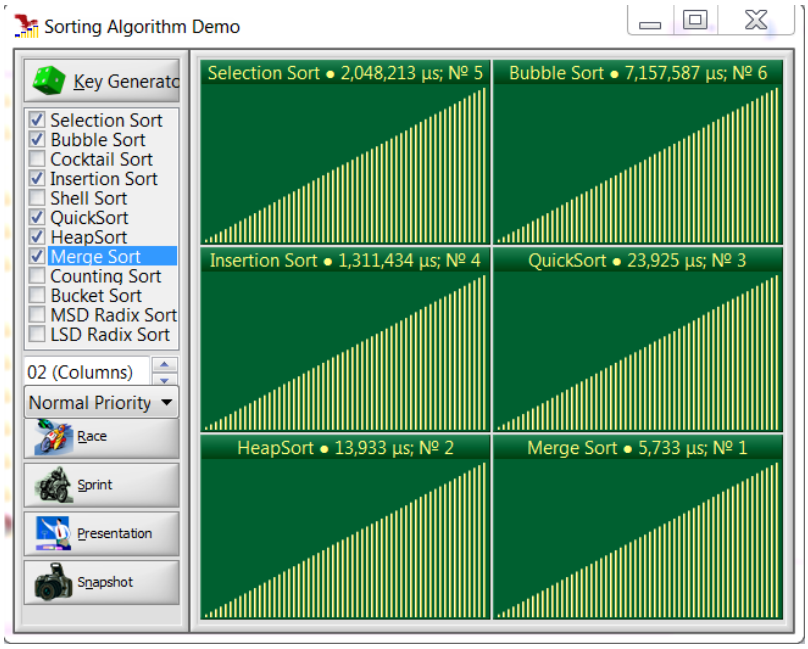
Sau đây là hình ảnh minh họa so sánh một số giải thuật sắp xếp với cùng dãy khóa những đã có khác biệt đáng kể về thời gian.

- Với n= 30



Insertion Sort có thời gian ít nhất, Bubble Sort có thời gian lớn nhất và QuickSort cũng không nhanh hơn Bubble Sort mấy.

- Với n=30000



Khi n khá lớn Merge Sort và heapSort tỏ ra nhanh nhất, QuickSort đạt mức trung bình, Insertion Sort đứng sau QuickSort, còn Bubble Sort và Selection Sort là chậm nhất.

CÂU HỎI VÀ BÀI TẬP CUỐI CHƯƠNG 4

1) Viết chương trình thực hiện công việc sau:
   a. Viết hàm tạo ngẫu nhiên một dãy số nguyên khoảng 100 số (gồm các số nguyên từ 1 đến 100.
   b. Viết hàm hiện dãy số ra màn hình.
   c. Viết 5 hàm sắp xếp theo 5 giải thuật khác nhau theo chiều giảm dần của dãy số.
   d. Viết hàm menu và hàm main cho phép chọn lựa thực hiện các công việc cho tới khi muốn kết thúc.
   e. Bổ sung phần tính thời gian thực hiện cho mỗi giải thuật để đánh giá, so sánh, chạy lại chương trình với các dãy khóa khi n=20; n=200 và n=2000.
2) Viết chương trình thực hiện công việc sau:
   a. Viết hàm tạo một mảng chứa danh sách họ tên sinh viên của một lớp, mỗi sinh viên gồm 2 trường là họ đệm và tên.
   b. Viết hàm hiện danh sách họ tên sinh viên ra màn hình (mỗi họ tên trên một dòng).
   c. Viết 5 hàm sắp xếp theo 5 giải thuật khác nhau theo chiều từ điển của tên sinh viên.
   d. Viết hàm menu và hàm main cho phép chọn lựa thực hiện các công việc cho tới khi muốn kết thúc.
3) Viết chương trình thực hiện công việc sau:
   a. Viết hàm tạo ngẫu nhiên một dãy số nguyên khoảng 50 số (gồm các số nguyên từ 1 đến 100.
   b. Viết hàm hiện dãy số ra màn hình.
   c. Viết hàm sắp xếp bằng giải thuật QuickSort với khóa chốt là phần tử đầu tiên của dãy.
   d. Viết hàm sắp xếp bằng giải thuật QuickSort với khóa chốt là phần tử cuối cùng của dãy, theo chiều tăng dần của dãy số.
   e. Viết hàm sắp xếp bằng giải thuật QuickSort với khóa chốt là phần tử ở giữa dãy, theo chiều tăng dần của dãy số.
   f. Viết hàm sắp xếp bằng giải thuật QuickSort với khóa chốt là phần tử trung vị trong 3 khóa: Đầu tiên, giữa và cuối cùng làm khóa chốt, theo chiều tăng dần của dãy số.

g. Viết hàm menu và hàm main cho phép chọn lựa thực hiện các công việc cho tới khi muốn kết thúc.

CHƯƠNG 5
TÌM KIẾM

Mục tiêu:

- Hiểu được giải thuật, cài đặt được giải thuật và đánh giá được độ phức tạp của giải thuật tìm kiếm tuyến tính, tìm kiếm nhị phân.
- Sử dụng giải thuật tìm kiếm tuyến tính, tìm kiếm nhị phân phù hợp với yêu cầu của bài toán trong thực tế.

1. Bài toán tìm kiếm

Nhu cầu tìm kiếm thường xuyên xảy ra trong thực tế cuộc sống cũng như trong xử lý tin học. Bài toán tìm kiếm tổng quát là tìm một bản ghi có giá trị trường khóa bằng X cho trước trong một bảng gồm n bản ghi.

Việc tìm kiếm được thực hiện bằng cách so sánh X (khóa tìm kiếm) với giá trị của trường khóa. Giải thuật sẽ hoàn thành khi có một trong hai tình huống sau xảy ra:

- Tìm thấy: Tìm được bản ghi có giá trị khóa tương ứng bằng X.
- Không tìm thấy: Không tìm được bản ghi có giá trị khóa tương ứng bằng X.

Khác với sắp xếp, khóa tìm kiếm được coi như là đặc điểm nhận dạng của mỗi bản ghi. Để giúp làm đơn giản ta tạm coi các giải thuật tìm kiếm được thực hiện trên một bảng gồm n bản ghi chỉ có một trường khóa và đó là trường duy nhất, còn giá trị khóa là các số nguyên.

Các giải thuật tìm kiếm được xét trong chương này là 2 giải thuật thông dụng: Tìm kiếm tuần tự và tìm kiếm nhị phân và được thực hiện ở bộ nhớ trong (tìm kiếm trong).

2. Tìm kiếm tuyến tính

2.1. Ý tưởng giải thuật

Là kỹ thuật tìm kiếm rất đơn giản và cổ điển. Nội dung có thể tóm tắt như sau:

Bắt đầu từ bản ghi thứ nhất, lần lượt so sánh khóa tìm kiếm với khóa tương ứng của các bản ghi trong bảng, cho tới khi tìm được bản ghi mong muốn hoặc đã hết bảng mà chưa thấy.

2.2. Mô tả giải thuật.

Input:

- K là một dãy khóa gồm n bản ghi
- Các khóa được đánh số từ K_0, K_1,…K_i,…K_{n-1}
- X là khóa tìm kiếm

Process:

Bước 1: Khởi gán

- Bổ sung khóa X vào cuối dãy khóa (để công việc tìm kiếm luôn thấy X)
- Khởi gán giá trị 0 cho biến chỉ số i

Bước 2: Tìm khóa X trong dãy khóa K

Lặp lại công việc sau cho đến khi tìm thấy X

So sánh khóa X với từng khóa Ki

Bước 3: Xét trường hợp tìm thấy hoặc không tìm thấy

Nếu i<n thì return (i)

Ngược lại thì return(-1)

Output: i là chỉ số của khóa có giá trị trùng với X, hoặc -1 nếu không tìm thấy X

2.3. Cài đặt giải thuật.

```
int  Sequen-Search(int K[],  int  n, int   X)
{       int  i=0;
        K[n]=X;
        While (X != K[i])
                i++;
        if (i<n)        return (i);
        else            return (-1);
}
```

2.4. Biểu diễn giải thuật.

Mô tả giải thuật với dãy khóa:

K: 6 4 9 3 8 2 7 5

n = 8; tìm kiếm với x=2 và x =1

Tìm với x=2 Bổ sung K[n]=x;

D									
C									
B									K[i]= x;
	K[i] ≠ x; tăng I cho tới khi K[i]=x;								return (i=5): Tìm thấy
Tìm với x=1 Bổ sung K[n]=x;									
D									
C									
B									
	K[i] ≠ x; tăng i cho tới khi K[i]=x; Vì i=n return (-1): Không tìm thấy								

3. Tìm kiếm nhị phân.

3.1. Ý tưởng giải thuật.

Tương tự như cách thức ta đã làm khi tra tìm số điện thoại của một cơ quan, trong bảng danh mục điện thoại hay khi ta tìm một từ trong tự điển. Giải thuật tìm kiếm nhị phân là luôn chọn khóa "ở giữa" dãy khóa đang xét để thực hiện so sánh với khóa tìm kiếm. Tìm kiếm sẽ dừng nếu khóa tìm kiếm bằng khóa ở giữa dãy, tìm kiếm lặp lại tương tự với nửa trước (bên trái) nếu khóa tìm kiếm nhỏ hơn khóa ở giữa dãy, cũng lặp lại tương tự với nửa sau (bên phải) nếu khóa tìm kiếm lớn hơn khóa ở giữa. Quá trình tìm kiếm được tiếp tục cho tới khi tìm thấy khóa mong muốn hoặc dãy khóa xét đó trở nên rỗng (không thấy).

Như vậy, điều kiện để có thể thực hiện được giải thuât tìm kiếm nhị phân là dãy khóa phải được sắp xếp tăng dần (hoặc giảm dần) với số và thứ tự từ điển đối với chuỗi ký tự.

3.2. Mô tả giải thuật.

Input:

- K là một dãy khóa gồm n bản ghi đã được sắp xếp theo thứ tự tăng

dần.

- Các khóa được đánh số từ $K_0, K_1,\ldots K_i,\ldots K_{n-1}$.
- X là khóa tìm kiếm.

Process:

Bước 1: Khởi gán.

- Khởi gán giá trị 0 cho biến chỉ số left.
- Khởi gán giá trị n-1 cho biến chỉ số right.

Bước 2: Tìm khóa X trong dãy khóa K.

- Lặp lại công việc sau cho đến khi left > right.
  - Khởi gán giá trị nguyên của (left +right)/2 cho biến chỉ số mid.
  - Nếu $X = K_{mid}$ thì return (mid).
  - Nếu $X < K_{mid}$ thì tìm ở dãy trước (right =mid-1).
  - Nếu $X > K_{mid}$ thì tìm ở dãy sau (left= mid +1).
- Nếu không tìm thấy (left > right) thì return(-1).

Output: mid là chỉ số của khóa có giá trị trùng với X, hoặc -1 nếu không tìm thấy X.

3.3. Cài đặt giải thuật.

```
int  BinarySearch (int K[], int  n, int X)
{               int   mid; left=0; right=n-1;
                do{
                        mid=(left+right)/2;
                        if (X== K[mid])    return (mid);
                        else
                          if (X< K[mid]) right=mid-1;
                          else  left=mid+1;
        }    while(left<=right);
        return -1;
}
```

3.4. Biểu diễn giải thuật.

Mô tả giải thuật với dãy khóa:

K: 6 4 9 3 8 2 7 5

n = 8; tìm kiếm với x=2 ; x =1 và x=12

Tìm với x=2

D		2								
C		0								
		left								
		mid=(left+right)/2				**x< K[mid]: right=mid-1**				
B						mid=(left+right)/2				
L						**x< K[mid]: right=mid-1**				
R										
						mid=(left+right)/2				
		K[mid]=x ⇨ return (mid): Tìm thấy x								
Tìm với x=1										
D		2								
C										
	3.10.	**left**								
		mid=(left+right)/2				**x< K[mid]: right=mid-1**				
B						mid=(left+right)/2				
L						**x< K[mid]: right=mid-1**				
R						Mid=(left+right)/2				
						x< K[mid]: right=mid-1				
		right< left return (-1): không tìm thấy x								
Tìm với x=12										
D	3.11.	2								
C										

	3.12.	left								
			mid=(left+right)/2							
			x> K[mid]: left=mid+1							
			mid=(left+right)/2							
B			**x> K[mid]: left=mid+1**							
L			mid=(left+right)/2							
R			**x> K[mid]: left=mid+1**							
			mid=(left+right)/2							
			x> K[mid]: left=mid+1							
			left >right □return (-1): không tìm thấy x							

Nhận xét:

Chúng ta dễ dàng nhận thấy độ phức tạp tính toán của giải thuật tìm kiếm tuần tự là O(n) và người ta cũng chứng minh được độ phức tạp tính toán của giải thuật tìm kiếm nhị phân là $O(\log_2 n)$. Rõ ràng kiếm nhị phân tỏ ra tối ưu hơn tìm kiếm tuần tự. Nhưng kiếm nhị phân lại đòi hỏi dãy khóa phải được sắp xếp rồi, do đó, cũng cần phải kể đến độ phức tạp tính toán của giải thuật sắp xếp nữa, đây cũng là nhược điểm của tìm kiếm nhị phân.

CÂU HỎI VÀ BÀI TẬP CUỐI CHƯƠNG 5

1) Viết chương trình thực hiện công việc sau:
   a. Viết hàm tạo ngẫu nhiên một dãy số nguyên khoảng 100 số (gồm các số nguyên từ 1 đến 100.
   b. Viết hàm hiện dãy số ra màn hình.
   c. Viết hàm tìm kiếm theo giải thuật tìm kiếm tuần tự với x (là một số) được nhập từ bàn phím. Nếu tìm thấy đưa ra thông báo vị trí của phần tử trùn với x, ngược lại đưa ra thông báo không tìm thấy.
   d. Viết hàm tìm kiếm theo giải thuật tìm kiếm nhị phân với x (là một số) được nhập từ bàn phím. Nếu tìm thấy đưa ra thông báo vị trí của phần tử trùng với x, ngược lại đưa ra thông báo không tìm thấy (*yêu cầu sắp xếp dãy số tăng dần trước khi thực hiện tìm kiếm*).
   e. Viết hàm menu và hàm main cho phép chọn lựa thực hiện các công việc cho tới khi muốn kết thúc.
2) Viết chương trình thực hiện công việc sau:
   a. Viết hàm tạo một mảng chứa danh sách họ tên sinh viên của một lớp, mỗi sinh viên gồm 2 trường là họ đệm và tên.
   b. Viết hàm hiện danh sách họ tên sinh viên ra màn hình (mỗi họ tên trên một dòng)
   c. Viết hàm tìm kiếm theo giải thuật tìm kiếm tuần tự với x (là một tên sinh viên) được nhập từ bàn phím. Nếu tìm thấy đưa ra thông báo vị trí của phần tử trùng với x, ngược lại đưa ra thông báo không tìm thấy.
   d. Viết hàm tìm kiếm theo giải thuật tìm kiếm nhị phân với x (là một tên sinh viên) được nhập từ bàn phím. Nếu tìm thấy đưa ra thông báo vị trí của phần tử trùng với x, ngược lại đưa ra thông báo không tìm thấy (yêu cầu *sắp xếp theo thứ tự từ điển của trường khóa tên sinh viên trước khi thực hiện tìm kiếm*).
   e. Viết hàm menu và hàm main cho phép chọn lựa thực hiện các công việc cho tới khi muốn kết thúc.

CHƯƠNG 6
CÂY

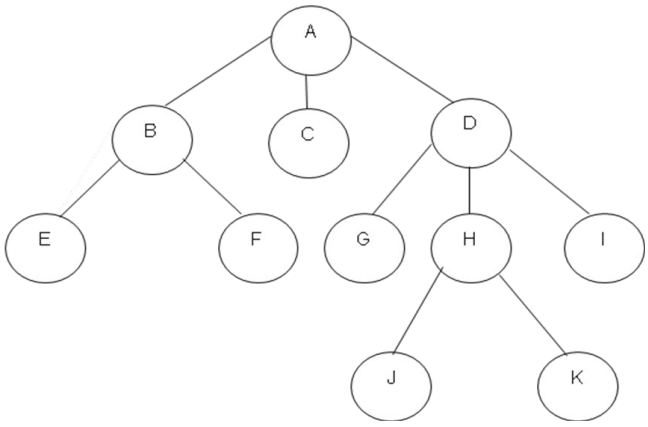
Mục tiêu:

- Trình bày được khái niệm về cây, cây nhị phân;
- Cài đặt được cây trên máy tính bằng các cấu trúc mảng và cấu trúc
  danh sách liên kết;
- Giải được bài toán duyệt cây nhị phân.

1. Khái niệm về cây

1.1. Khái niệm cây

Cây là một cấu trúc phi tuyến . Một cây là một tập hợp hữu hạn các nút trong đó có một nút đặc biệt gọi là gốc (Root). Giữa các nút có mối quan hệ phân cấp gọi là "quan hệ cha con" .



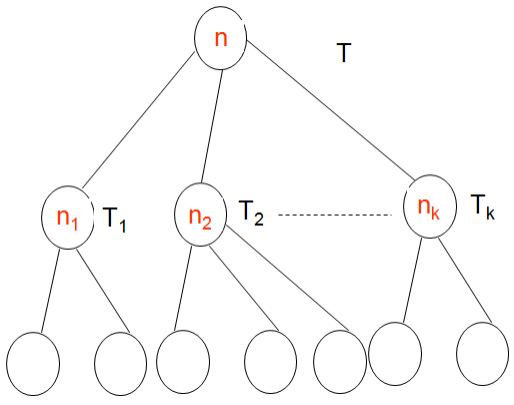
Hình 6.1: Cây tổng quát

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

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

Nếu n là một nút và $T_1,T_2,.....,T_k$ là các cây . Với $n_1,n_2,....,n_k$ lần lượt là các gốc thì cây mới T sẽ được tạo lập bằng cách cho n trở thành "cha " của các nút $n_1,n_2,....,n_k$. Nghĩa là trên cây mới này n là gốc còn $T_1,T_2,...,T_k$ là các cây con của gốc, lúc đó $n_1,n_2,....,n_k$ là con của nút n.

Để tiện, người ta cho phép một cây không có nút nào, mà ta gọi là cây rỗng (null tree).



Hình 6.2: Định nghĩa cây nhị phân

1.2. Một số khái niệm của cây

- ***Cấp (degree):*** Số các con của một nút gọi là cấp của nút đó (hình 6.1: A có 3 con là cấp 3).
  - Nút có cấp bằng không (nút không có con) gọi là lá (Leaf) (hinh 6.1: E,F,C,.. là lá)
  - Cấp cao nhất của nút có trên cây thì gọi là cấp của cây. (hình 6.1: Cấp là 3).
- ***Mức (Level):*** Gốc của cây có số mức là một.

  Nếu nút cha có số mức là i thì nút con có số mức là i+1 (hình 6.1: A có mức là 1, D có mức là 2, G có mức là 3,...).
- ***Chiều cao (Hight) hay chiều sâu (Depth)*** của một cây là số mức lớn nhất

  của nút có trên cây đó. (hình 6.1: có số mức cao nhất là 4 nên cây có chiều cao cũng là 4).
- Đường đi (Path): Nếu ta có một dãy các nút n1,n2,...,nk trên cây và thỏa mãn ni là cha của ni+1 thì dãy đó được gọi là đường đi trên cây đó. Độ dài của đường đi (path length): Là số nút trên đường đi trừ đi một.
- Nếu thứ tự các cây con của một nút được coi trọng thì cây đang xét là cây

  có thứ tự (ordered tree), ngược lại là cây không có thứ tự (Unordered tree), Thường thứ tự các cây con của một nút được đặt từ trái sang phải.
- Nếu có một tập hữu hạn các cây phân biệt thì ta gọi tập đó là ***một rừng***

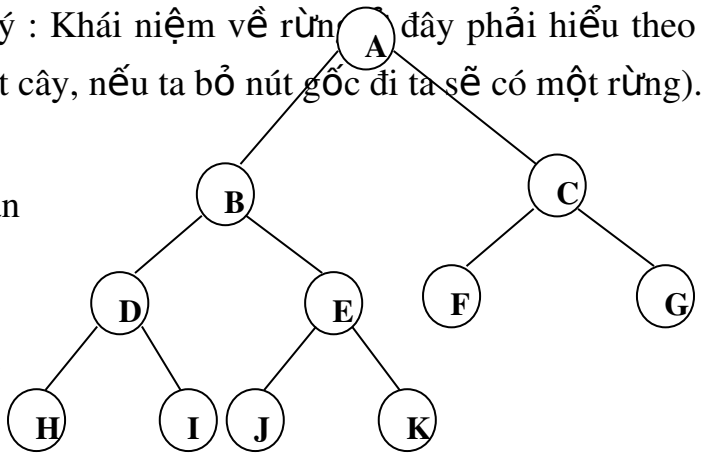
  (forest) (chú ý : Khái niệm về rừng ở đây phải hiểu theo nghĩa riêng: Có một cây, nếu ta bỏ nút gốc đi ta sẽ có một rừng).

2. Là một dạng quan trọng của cấu trúc cây. n

Đặc điểm là mọi nút trên cây chỉ có tối đa là hai con.

Đối với cây con của một nút người ta cũng phân biệt cây con trái và cây con phải.

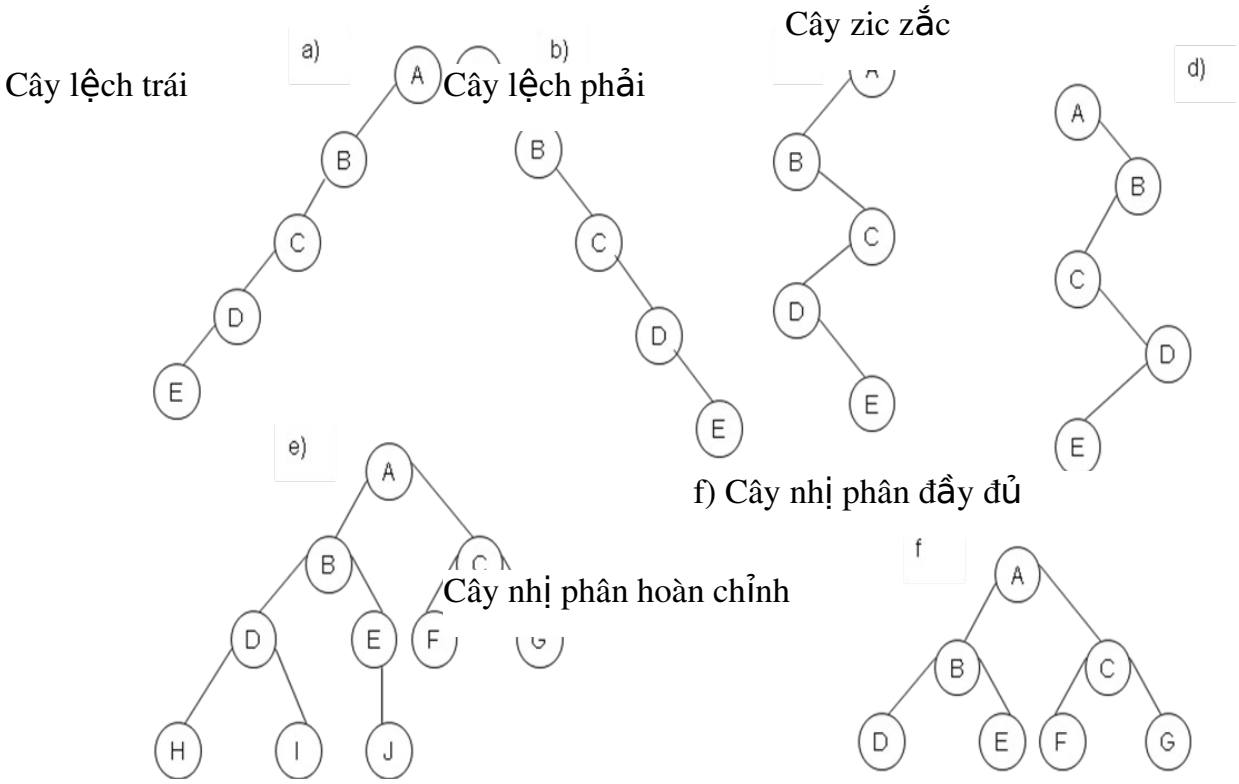
Cây nhị phân được gọi là



Hình 6.3: Cây nhị phân

2.2. Một số tính chất của cây nhị phân

- Các đặc điểm nêu ở mục 1.2 cũng đúng với cây nhị phân
- Số lượng tối đa các nút ở mức i trên một cây nhị phân là $2^{(i-1)}$. (i>=1)
- Số lượng tối đa các nút trên một cây nhị phân có chiều cao h là: $2^h - 1$ (h>=1)
- Một số dạng đặc biệt của cây nhị phân



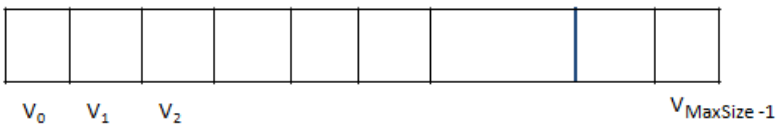
Hình 6.4: Cây nhị phân đặc biệt

- Cây nhị phân hoàn chỉnh: Các nút ứng với các mức trừ mức cuối cùng đạt tối đa (e, f là cây nhị phân hoàn chỉnh).
- Cây nhị phân hoàn chỉnh có chiều cao nhỏ nhất.
- Cây nhị phân suy biến (a, b, c, d) có chiều cao lớn nhất.

2.3. Biểu diễn cây nhị phân.

2.3.1. Lưu trữ cây bằng véc tơ kế tiếp (lưu trữ kế tiếp):

Dùng một véctơ lưu trữ V gồm MaxSize phần tử nhớ kế tiếp để lưu trữ các nút trên cây.



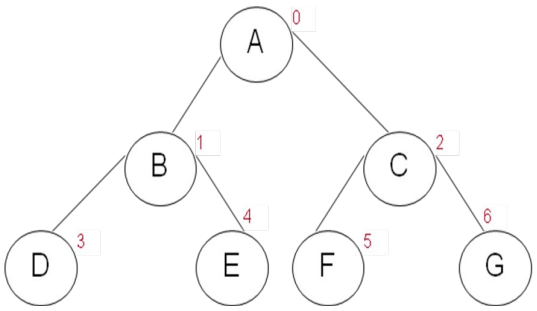
Hình 6.5: Véc tơ lưu trữ cây nhị phân

a. Nguyên tắc lưu trữ

- Cha có số thứ tự là i thì con có số thứ tự là 2i và 2i+1
- Con có số thứ tự là j thì cha có số thứ tự là j/2 (lấy số nguyên dưới)
- Quan hệ cha con được xác định qua số thứ tự này.
- Dùng véc tơ V để lưu trữ các nút trên cây theo nguyên tắc nút thứ i

  của cây được lưu trữ bởi véc tơ V[i].

❖ Xét một cây nhị phân hoàn chỉnh đầy đủ: Đánh số thứ tự cho các nút trên cây theo trình tự lần lượt từ mức 1 trở lên, hết mức này đến mức khác và trong mỗi mức thì từ trái sang phải **ABCDEFG-0123456**

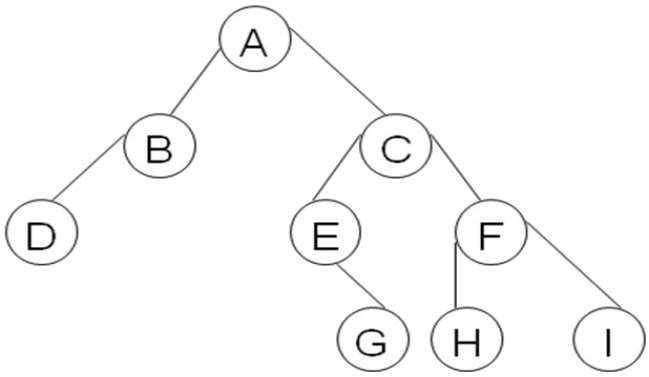


Hình 6.6: Cây nhị đầy đủ

Nút 0 được lưu trữ bởi phần tử V0, nút 1 được lưu trữ bởi phần tử V1 …

Hình 6.7: Lưu trữ kế tiếp với cây nhị phân đầy đủ

❖ Xét một cây nhị phân hoàn chỉnh không đầy đủ



Hình 6.8: Cây nhị phân hoàn chỉnh không đầy đủ

Véc tơ lưu trữ tương ứng:

Hình 6.9: Lưu trữ kế tiếp với cây nhị phân hoàn chỉnh

Nhận xét:

- Ưu điểm:
  - Với cách lưu trữ này quan hệ cha con hoàn toàn được xác định thông qua chỉ số của véc tơ lưu trữ.
  - Với cây nhị phân đầy đủ thì cách lưu trữ này khá thuận lợi
  - Truy nhập trực tiếp vào nút của cây thông qua chỉ số phần tử
- Nhược điểm:
  - Với cây nhị phân không đầy đủ và nhất là các cây nhị phân suy biến thì gây ra lãng phí bộ nhớ vì xuất hiện rất nhiều nút rỗng.
  - Nếu cây luôn biến động (thường xuyên có phép bổ sung, loại bỏ tác động thì cách lưu trữ này càng thể hiện nhiều nhược điểm).

Như vậy, thường chỉ dùng cách lưu trữ kế tiếp đối với cây nhị phân đầy đủ hoặc hoàn chỉnh.

b. Khai báo cấu trúc.

Để cài đặt cây nhị phân tổng quát bằng véc tơ lưu trữ kế tiếp, ta dùng một mảng các nút có độ dài tối đa là MaxSize phần tử. Mỗi nút của cây được biểu diễn bằng một bản ghi gồm 3 trường. Trường thứ nhất info kiểu Item (Item có thể là các kiểu dữ liệu đơn giản hoặc kiểu dữ liệu có cấu trúc),

trường thứ hai, thứ ba là Lchild và Rchild chứa chỉ số của nút con trái và nút con phải.

Khai báo cấu trúc của danh sách bằng mảng:

```
const       int MaxSize=100
typedef struct node
{ Item info;
  int Lchild, Rchild;
};
        node  tree[MaxSize];
```

2.3.2. Lưu trữ cây bằng danh sách liên kết:

Để khắc phục nhược điểm của lưu trữ kế tiếp, người ta thường dùng danh sách liên kết để lưu trữ với cây nhị phân.

a. Nguyên tắc lưu trữ

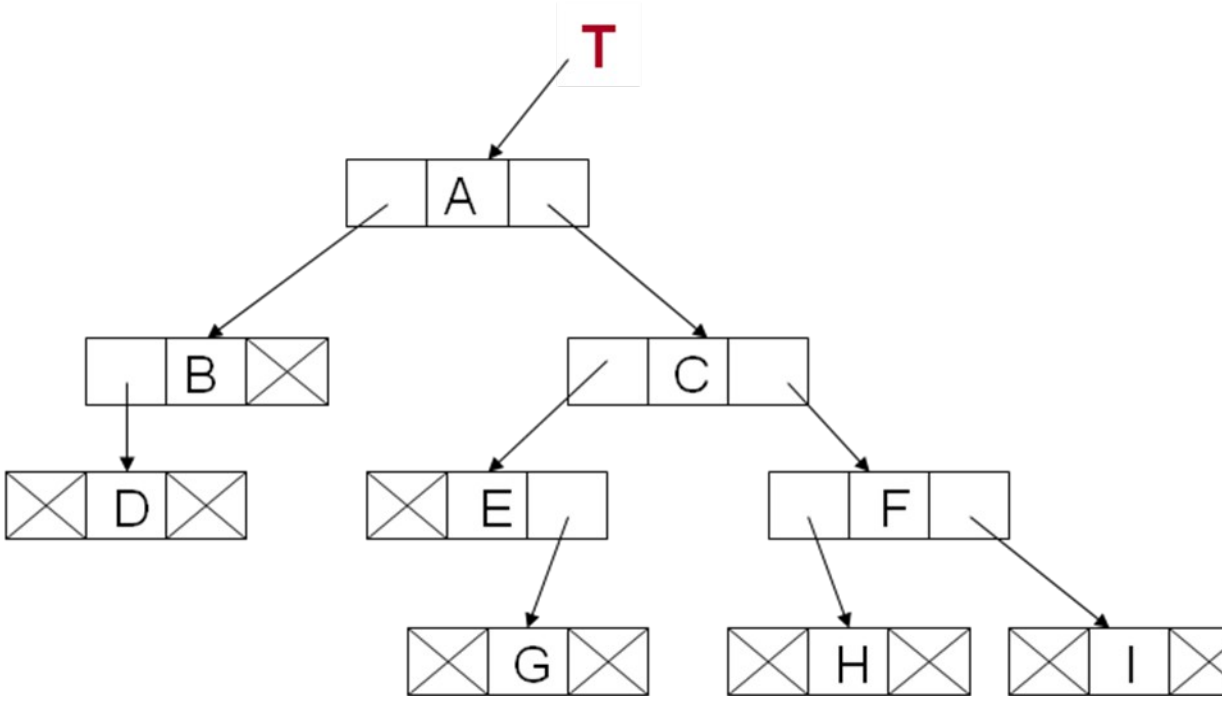
- Chỉ lưu trữ những nút tồn tại trên cây
- Với cách lưu trữ móc nối sẽ khắc phục được nhược điểm nêu trên và vừa phản ánh được dáng tự nhiên của cây.
- Dùng một con trỏ T luôn trỏ vào nút gốc của cây để giúp ta truy nhập vào cây. Khi cây rỗng T=NULL
- Quy cách:

Hình 6.10: Lưu trữ một nút trên cây nhị phân

- Trường info: Lưu trữ các thông tin tương ứng trên cây
- Trường Lchild: Trỏ tới cây con trái tương ứng của nút
- Trường Rchild: Trỏ tới cây con phải tương ứng của nút

❖ Giả thiết ta có một cây nhị phân hoàn chỉnh như hình 6.8

Hình ảnh lưu trữ móc nối



Hình 6.11: Lưu trữ móc nối với cây nhị phân hình 6.9

Nhận xét:

- Ưu điểm: Đỡ tốn bộ nhớ
- Nhược điểm: Truy nhập tuần tự bắt đầu từ nút cha tới các nút con

b. Khai báo cấu trúc

Cài đặt cây nhị phân tổng quát bằng danh sách liên kết, mỗi nút của cây được biểu diễn bằng một bản ghi gồm 3 trường. Trường thứ nhất info kiểu ElementType (ElementType có thể là các kiểu dữ liệu đơn giản hoặc kiểu dữ liệu có cấu trúc), trường thứ hai, thứ ba là Lchild và Rchild có kiểu dữ liệu con trỏ node chứa địa chỉ của nút con trái và nút con phải.

Đặc biệt cần có thêm con trỏ T để luôn chứa địa chỉ nút gốc của cây. T bằng NULL khi cây rỗng.

Khai báo cấu trúc của cây bằng danh sách liên kết:

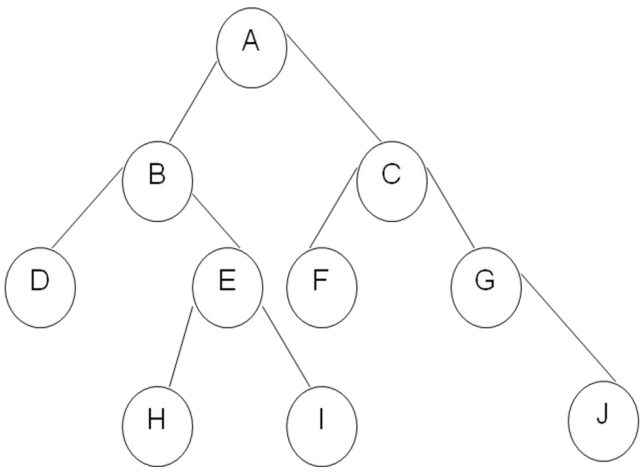
```
struct node
{ ElementType    info;
  node           *Lchild;
  node           *Rchild;
};
typedef  struct node*  TreeNode;
TreeNode         T;
```

3. Các phép duyệt cây nhị phân.

Phép xử lý mỗi nút của cây theo một thứ tự nào đó được gọi là phép duyệt cây. Thông thường có 3 phép duyệt cây:

- Duyệt cây theo thứ tự trước.
- Duyệt cây theo thứ tự giữa.
- Duyệt cây theo thứ tự sau.

Ở mỗi phép duyệt cây ta phải thăm lần lượt các nút sao cho mỗi nút chỉ được thăm một lần. Giả sử với có cây nhị phân sau ta sẽ có dãy các nút khác nhau tương ứng với từng phép duyệt cây.



Hình 6.12: Cây nhị phân hoàn chỉnh

3.1. Duyệt cây theo thứ tự trước **(Preorder traversal).**

3.1.1. Phép duyệt.

Phép duyệt được thực hiện bằng việc thăm gốc trước tiên, cụ thể theo các thứ tự sau:

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

3.1.2. Giải thuật:

Với cấu trúc cây nhị phân cài đặt bằng danh sách liên kết như sau:

```
typedef     char  ElementType;
struct node
{ ElementType    info;
  node             *Lchild;
  node             *Rchild;
};
typedef  struct node*  TreeNode;
```

```
        TreeNode        T;
```

Giải thuật PreOrder được cài đặt đệ qui như sau:

```
// Tham số hình thức T là một con trỏ chứa địa chỉ nút gốc của cây
void  PreOrder (TreeNode  T)
{ if (T != NULL)
   {  printf("% c", T->info);          //in ra màn hình giá trị trường info
      PreOrder (T-> Lchild)
      PreOrder (T-> Rchild)
  }
 }
```

Dãy các nút của cây nhị phân hình 6.12 tương ứng với phép duyệt cây theo thứ tự trước là: ABDEHICFGJ.

3.2. Duyệt cây theo thứ tự giữa (**Inorder traversal**).

3.2.1. Phép duyệt.

Phép duyệt được thực hiện bằng việc thăm gốc sau khi đã thăm hết các nút của cây con trái, cụ thể như sau:

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

3.2.1. Giải thuật:

Giải thuật đệ qui của InOrder được cài tương tự như sau:

```
// Tham số hình thức T là một con trỏ chứa địa chỉ nút gốc của cây
void  InOrder (TreeNode  T)
{ if (T != NULL)
   {  InOrder (T-> Lchild)
     printf("% c", T->info);          //in ra màn hình giá trị trường info
      InOrder (T-> Rchild)
  }
 }
```

Dãy các nút của cây nhị phân hình 6.12 tương ứng với phép duyệt cây theo thứ tự giữa là: DBHEIAFCGJ.

3.3. Duyệt cây theo thứ tự sau (**Postorder traversal**).

3.3.1. Phép duyệt.

Phép duyệt được thực hiện bằng việc thăm gốc sau cùng, khi đã thăm hết các nút của cây con trái và các nút của cây con phải theo thứ tự sau:

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

3.3.1. Giải thuật:

Giải thuật đệ qui của PostOrder được cài tương tự như sau:

```
// Tham số hình thức T là một con trỏ chứa địa chỉ nút gốc của cây
void  PostOrder (TreeNode  T)
{ if (T != NULL)
   {  PostOrder (T-> Lchild)
       PostOrder (T-> Rchild)
    printf("% c", T->info);        //in ra màn hình giá trị trường info
  }
 }
```

Dãy các nút của cây nhị phân hình 6.12 tương ứng với phép duyệt cây theo thứ tự sau là: DHIEBFJGCA

Các phép duyệt cây được cài đặt khá dễ dàng bằng giải thuật đệ qui vì nó phản ánh đúng dạng các định nghĩa của phép duyệt cây. Phép thăm mỗi nút ở đây là hiển thị giá trị trường info của nút đó.

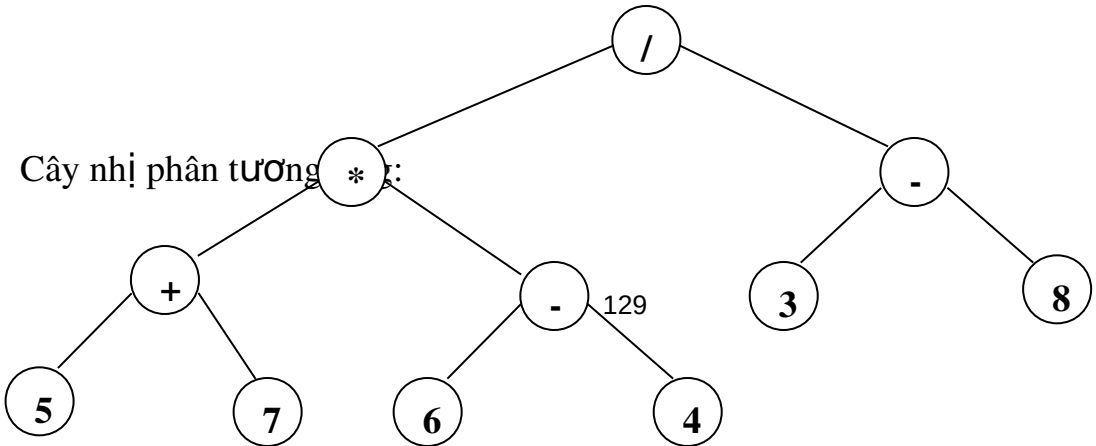
3.4. Ví dụ áp dụng.

Sử dụng cấu trúc cây nhị phân để lưu trữ một biểu thức số học, với qui định là các nút cha chứa toán tử còn các nút lá chứa toán hạng.

Giả sử ta có biểu thức:

$$\left(\frac{(5+7)*(6-4)}{(3-8)}\right)$$

Cây nhị phân tương ứng:



Hình 6.13 : Cây nhị phân biểu diễn biểu thức số học

Tương ứng với các phép duyệt cây ta có các biểu thức số học theo các dạng ký pháp Ba Lan:

- Duyệt cây theo thứ tự trước (dạng tiền tố): / * + 5 7- 6 4 - 3 8
- Duyệt cây theo thứ tự giữa (dạng trung tố): 5 + 7 * 6 – 4 / 3 - 8
- Duyệt cây theo thứ tự sau (dạng hậu tố): 5 7 + 6 4 - * 3 8 - /

Các biểu thức dạng ký pháp Ba Lan do nhà logic toán Jan Łukasiewicz người Ba Lan đề xuất khoảng năm 1920. Ký pháp Ba Lan là một cách viết một biểu thức đại số rất thuận lợi cho việc thực hiện các phép toán. Về lý thuyết, ký pháp dạng tiền tố và ký pháp dạng hậu tố có thể thực hiện các phép toán theo thứ tự từ trái sang phải mà không phải dùng tới dấu ngoặc để thể hiện độ ưu tiên các phép toán, còn ký pháp dạng trung tố thì không thể.

Thứ tự các phép toán trong các biểu thức dạng ký pháp Ba Lan tương ứng với các phép duyệt cây hình 6.13 như sau:

- Biểu thức dạng tiền tố thì dấu toán tử trước hai toán hạng. Thứ tự các phép

  toán được thực hiện là:

  Bắt đầu từ trái sang phải ta gặp phép toán: + 5 7 (lấy 5+7=12)

  Tiếp đến phép toán: - 6 4 (lấy 6 – 4=2)

  Rồi đến phép toán: * 12 2 (lấy 12 * 2 = 24)

  Tiếp theo đến phép toán: - 3 8 (lấy 3 – 8 = -5)

  Cuối cùng là phép toán: 24 -5 / (lấy 24/-5= **-4.8**)
- Biểu thức dạng hậu tố thì dấu toán tử ở sau hai toán hạng. Thứ tự các phép toán được thực hiện từ trái sang phải như sau:

  Đầu tiên là phép toán: 5 7 + (lấy 5+7 =12).

Tiếp đến phép toán: 6 4 – (lấy 6-4 = 2).

Rồi đến phép toán 12 2 * (lấy 12 * 2=24).

Tiếp theo là phép toán 3 8 – (lấy 3 – 8 = -5).

Cuối cùng là phép toán 24 -5 / (lấy 24/-5= **-4.8**).

- Biểu thức dạng trung tố thì dấu toán tử ở giữa hai toán hạng. Thứ tự các phép toán được thực hiện là:

5 + 7 * 6 – 4 / 3 - 8

Đầu tiên là phép toán: 5+7 =12.

Tiếp theo là phép toán: 12*6 =72.

Rồi đến phép toán: 72-4= 68.

Tiếp đến phép toán: 68/3 = 22.7.

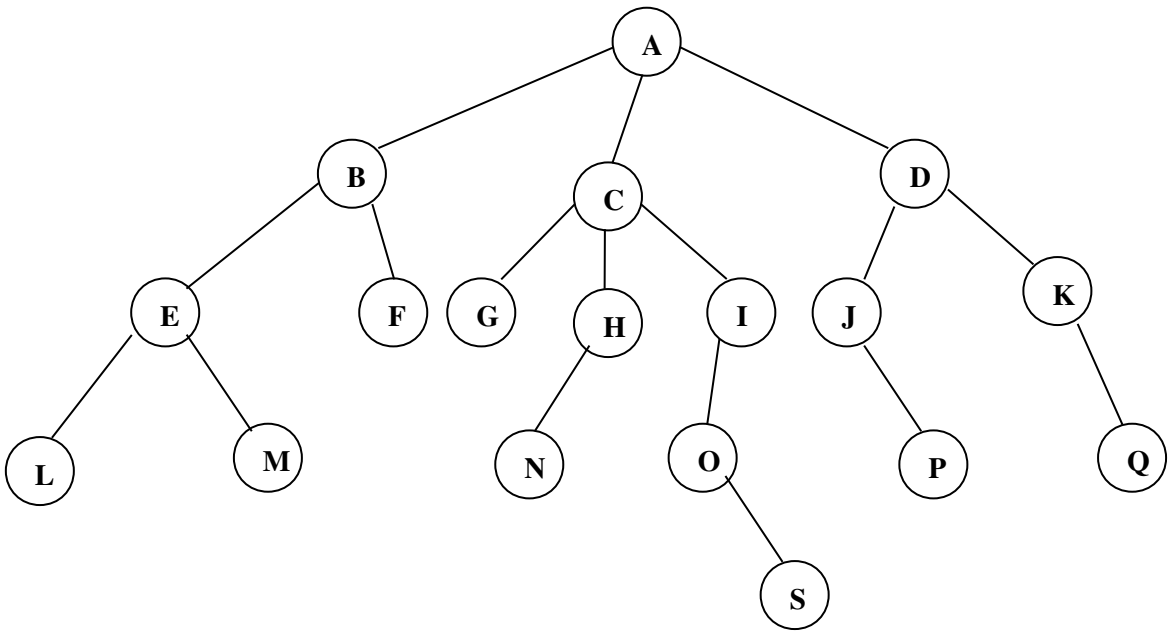
Cuối cùng là phép toán: 22.7 -8 =**14.7.**

Nhận xét:

- Biểu thức dạng trung tố cho kết quả sai vì theo nguyên tắc là toán tử ở giữa hai toán hạng, do đó mỗi khi thấy một toán tử ở giữa hai toán hạng là một phép toán được thực hiện (cần phải có các dấu ngoặc tròn để thay đổi thứ tự các phép toán như chúng ta vẫn quen dùng).
- Biểu thức dạng tiền tố và hậu tố không cần phải có cặp ngoặc tròn vẫn thực hiện đúng biểu thức.

CÂU HỎI VÀ BÀI TẬP CUỐI CHƯƠNG 6

1) Hãy trình bày định nghĩa đệ qui của cấu trúc dữ liệu cây.
2) Hãy nêu những tính chất của cấu trúc dữ liệu cây tổng quát và cây nhị phân
3) Hãy trình bày hai cách biểu diễn cây nhị phân là lưu trữ kế tiếp và lưu trữ bằng danh sách liên kết? So sánh ưu, nhược điểm của hai cách biểu diễn này và cho ví dụ minh họa.
4) Cho cây hình 6.14 sau:



Hình 6.14 : Cây tổng quát

a. Hãy liệt kê các nút lá
b. Hãy liệt kê các nút nhánh
c. Cha của nút J là nút nào ?
d. Con của nút E là nút nào?
e. Mức của K, của M là bao nhiêu?
f. Cấp của cây là bao nhiêu?
g. Chiều cao của cây là bao nhiêu?
h. Độ dài đường đi từ A tới F, Q, S là bao nhiêu?
i. Có bao nhiêu đường đi có độ dài 3 trên cây này?

5) Vẽ cây nhị phân biểu diễn các biểu thức sau và viết chúng dưới dạng tiền tố, hậu tố:

a. $$ax - \frac{by}{2} * dx + \frac{ax^2}{by^2 - z}$$

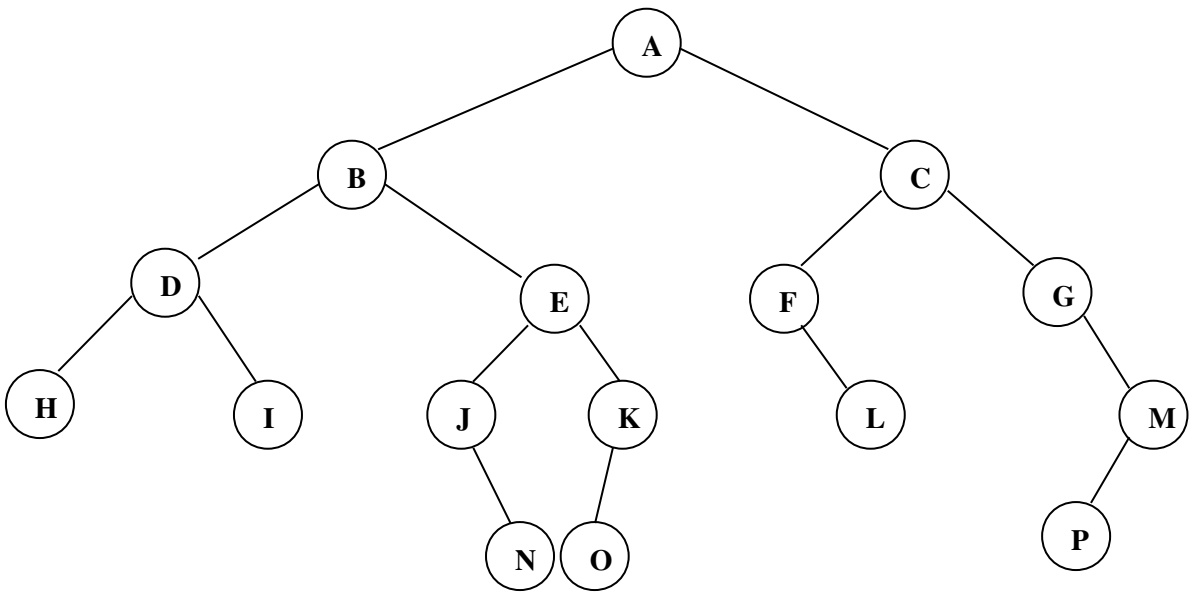
b. $$x + y - \frac{z}{3} * \frac{2x + 3y}{4x^2}$$

c. $$(a + b) + c * (d + e) - f * (g - h)$$

d. $$\frac{(a - b * c)}{\left(d + \frac{e}{f}\right)}$$

e. $$\frac{a}{(b * c)} + d * e - a * c$$

6) Cho cây nhị phân hình 6.15 sau:



Hình 6.15 : Cây nhị phân

a. Hãy viết ra các nút khi duyệt cây theo thứ tự trước.

b. Hãy viết ra các nút khi duyệt cây theo thứ tự giữa.

c. Hãy viết ra các nút khi duyệt cây theo thứ tự sau.

d. Hãy minh họa bộ nhớ khi thực hiện lưu trữ kế tiếp với cây này.

e. Hãy minh họa bộ nhớ khi thực hiện lưu trữ bằng danh sách liên kết với cây này.

7) Viết chương trình để tính giá trị của biểu thức khi cho biểu thức tiền tố

8) Chứng minh rằng: Nếu biết biểu thức duyệt tiền tố và trung tố của một cây nhị phân thì ta dựng được cây này. Điều đó đúng nữa không? Khi biết biểu thức duyệt:

a. Tiền tố và hậu tố.

b. Trung tố và hậu tố.

CHƯƠNG 7
ĐỒ THỊ

Mục tiêu:

- Hiểu được khái niệm về đồ thị;
- Cài đặt được đồ thị trên máy tính bằng các cấu trúc mảng và cấu trúc danh sách liên kết;
- Thực hiện được các phép duyệt đồ thị.

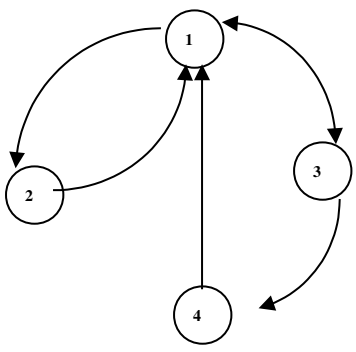
1. Khái niệm về đồ thị

1.1. Định nghĩa

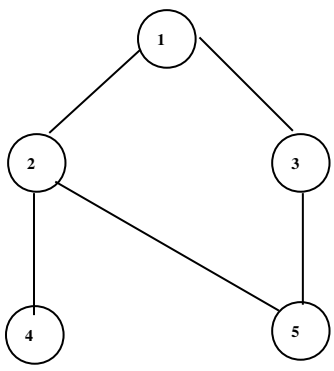
Một đồ thi G(V,E) bao gồm một tập hữu hạn V các nút, hay đỉnh (Vertices) và một tập hữu hạn E các cặp đỉnh mà ta gọi là cung (Edges)

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

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



Hình 7.1: Đồ thị định hướng



Hình 7.2: Đồ thị không định hướng (đồ thị vô hướng)

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

1.2. Các khái niệm.

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

- *Đường đi (path):* Một đường đi từ đỉnh vp đến đỉnh vq trong đồ thị G là

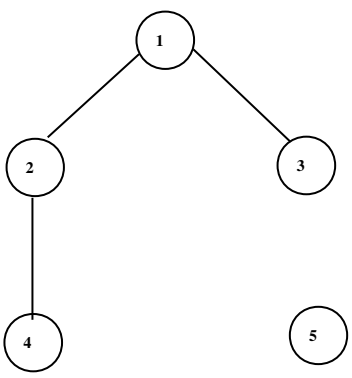
  một dãy các đỉnh v_p, v_{i0}, v_{i1}, ..., v_{in-1}, v_q mà (v_p,v_{i0}), (v_{i0},v_{i1}),... (v_{in-1},v_q) là các cung trong E(G). Số lượng các cung trên đường đi ấy gọi là *độ dài của đường đi (path length).*

  Ví dụ: Hình 7.1: 1,3,4 là một đường đi từ đỉnh 1 đến đỉnh 4, nó có độ dài 2.

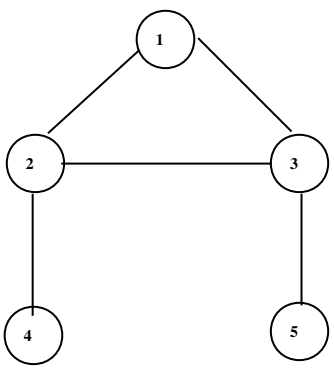
  Hình 7.2: Đường đi từ đỉnh 1 đến đỉnh 4 là 1, 3, 5, 2, 4 nó có độ dài bằng 4; hoặc 1, 2, 4 có độ dài là 2.
- *Đường đi đơn (simple path):* Là đường đi mà mọi đỉnh trên đó, trừ đỉnh

  đầu tiên và đỉnh cuối cùng, đều khác nhau.
- *Một chu trình (Cycle):* Là một đường đi mà đỉnh đầu và đỉnh cuối trùng

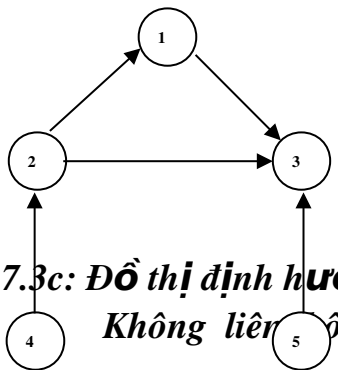
  với nhau (ví dụ : Hình 7.1 đường đi 1, 3, 4, 1 là một chu trình).
- Đối với đồ thị định hướng, để cho rõ, thường người ta thêm vào các từ "định hướng" sau các thuật ngữ trên (ví dụ: Đường đi định hướng từ vi tới vj).
- *Liên thông:* Trong đồ thị G hai đỉnh v_i và v_j gọi là liên thông nếu có một đường đi từ vi tới vj (dĩ nhiên với đồ thị không định hướng thì đồng thời cũng có đường đi từ vj tới vi)**.**



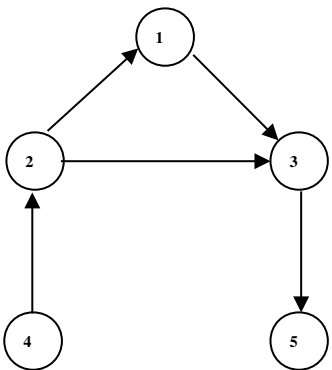
Hình 7.3a: Đồ thị không định hướng, không liên thông



Hình 7.3b: Đồ thị không định hướng, liên thông

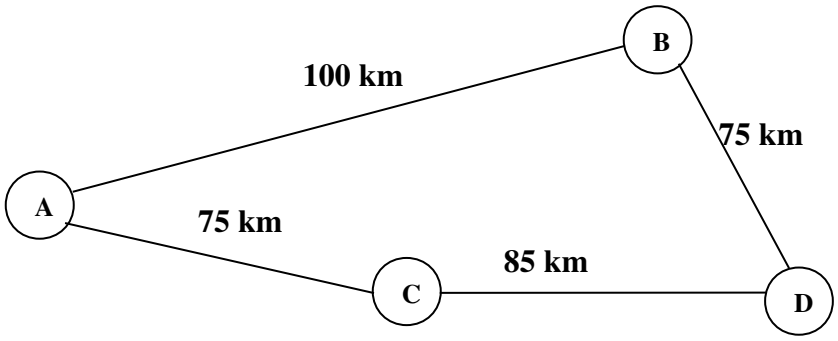


Hình 7.3c: Đồ thị định hướng, Không liên thông



Hình 7.3d: Đồ thị định hướng, liên thông

- *Đồ thị có trọng số:* Trong thực tế có rất nhiều ứng dụng của đồ thị đòi hỏi phải có thông tin trên các cạnh (cung) của đồ thị để biểu thị khoảng cách, giá thành,... Thông tin đó là những con số và được gọi là trọng số, đồ thị này được gọi là đồ thị có trọng số. Ví dụ: Bản vẽ dự toán về chi phí cho tuyến đường cần xây dựng từ tỉnh A đến tỉnh D có đi qua các tỉnh B hoặc C được thể hiện như sau:



Hình 7.4: Đồ thị có trọng số

2. Biểu diễn đồ thị

2.1. Biểu diễn bằng ma trận kề

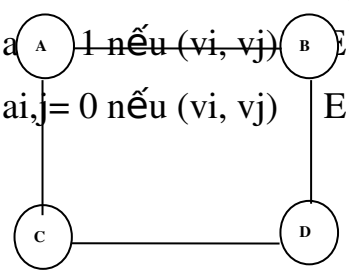
Dùng một ma trận có kiểu logic để biểu diễn các đỉnh và các cung của đồ thị

Giả sử ta có một đồ thị có hướng G=<V, E> gồm n đỉnh {v_0, v_1,…, v_{n-1}}.

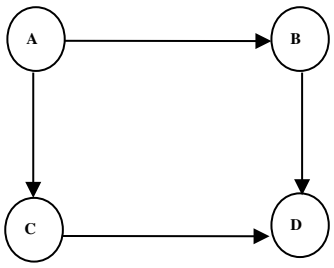
Giá trị của ma trận Ai,j được xác định theo nguyên tắc sau:

ai,j= 1 nếu (vi, vj) E: Nghĩa là có một cung nối từ vi đến v

ai,j= 0 nếu (vi, vj) E: Nghĩa v_0=A v_1=B vi và vj không có cung nối.
v_2=C v_3=D



Hình 7.5: Ma trận kề của đồ thị vô hướng



v_0=A v_1=B
v_2=C v_3=D

$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Hình 7.6: Ma trận kề của đồ thị định hướng

- Nhận xét:
  - Ở đồ thị vô hướng thì ma trận kề biểu diễn nó có các phần tử đối
    xứng qua đường chéo chính, tức là Ai,j = Aj,i
  - Nhìn vào ma trận kề biểu diễn đồ thị định hướng ta có thể biết được các cung đi tới hoặc xuất phát từ một đỉnh cho trước. Ví dụ tại đỉnh v_1=B ta biết chỉ có một cung đi từ đỉnh v_1 đến đỉnh v_3=D.
- Ưu điểm:
  - Đơn giản, trực quan.
  - Dễ dàng xác định được có hay không một cung đi từ đỉnh i đến đỉnh j.
  - Dễ cài đặt trên máy tính.
- Nhược điểm:
  - Tốn bộ nhớ: Bất kể đồ thị có bao nhiêu cạnh, mỗi đồ thị cũng cần
    một ma trận vuông với kích thước n x n phần tử (ô nhớ). Tuy nhiên có thể khắc phục điều này bằng cách chỉ lưu trữ một nửa trên hoặc nửa dưới của ma trận nhưng chỉ với đồ thị vô hướng.
  - Tốn thời gian: Với một đỉnh vi nhiều khi ta phải xét tất cả các đỉnh
    vj khác để tìm các đỉnh kề với nó.
- Ví dụ: Viết chương trình đưa ra màn hình tất cả các đỉnh kề với từng

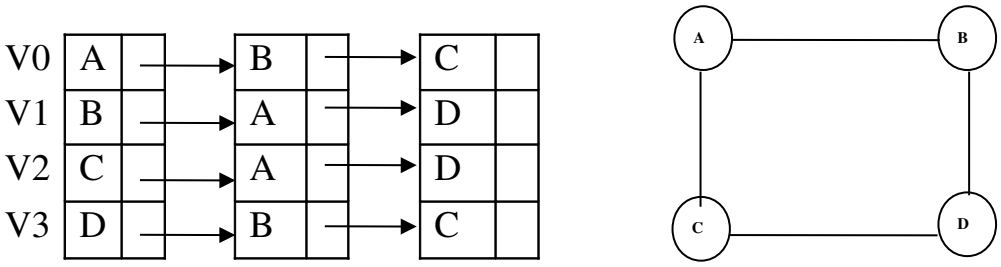
đỉnh trong một đồ thị bất kỳ.

```
#include <conio.h>
#include <stdio.h>
 const n=5;
void main()
{  int i,j,k; clrscr();
   int a[n][n];
   printf("nhap ma tran ke tuong ung\n");
   for ( i=0; i<n ;i++)
    for (j=0; j<n; j++)
     {printf("nhap a%d:",i);scanf("%d",&a[i][j]);
     }  clrscr();
    printf("cac dinh ke:\n ");
    for ( i=0; i<5 ;i++)
      {
    printf("\n dinh ke voi %d la: ",i);
        for ( j=0; j<5 ;j++)
       if (a[i][j] == 1)
          printf("%3d;",j);
  }
    getch();
}
```

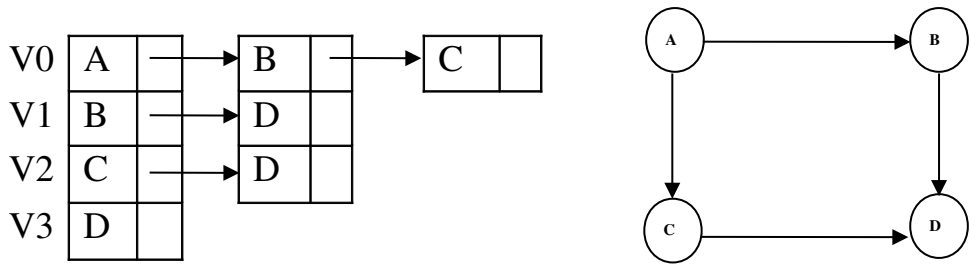
2.2. Biểu diễn đồ thị bằng danh sách kề.

Phương pháp này dùng n danh sách liên kết cho n đỉnh của đồ thị, mỗi danh sách liên kết của một đỉnh sẽ chứa tất cả các đỉnh khác kề với nó. Do đó các danh sách này còn được gọi là danh sách kề.

Như vậy để lưu trữ thông tin về một đồ thị có n đỉnh ta cần một mảng G gồm n phần tử, mỗi phần tử Gi là một con trỏ trỏ tới danh sách các đỉnh kề với đỉnh i, mỗi nút gồm 2 trường là Vertex chứa chỉ số của đỉnh kề với nó và trường Link chứa địa chỉ của nút tiếp theo trong danh sách.



Hình 7.7: Ma trận danh sách kề của đồ thị vô hướng



Hình 7.8: Ma trận danh sách kề của đồ thị định hướng

3. Các phép duyệt đồ thị

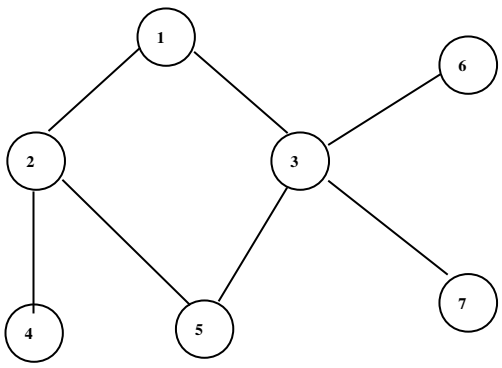
Tương tự như cấu trúc dữ liệu cây, khi xử lý bài toán đồ thị, ta cần thăm các đỉnh của đồ thị một cách có hệ thống để đảm bảo rằng mỗi đỉnh chỉ được thăm đúng một lần, việc này được gọi là duyệt đồ thị. Có hai phép duyệt đồ thị phổ biến đó là duyệt theo chiều sâu và duyệt theo chiều rộng.

Giả sử ta có đồ thị G=(V,E) và một đỉnh v trong V(G), ta cần thăm tất cả các đỉnh thuộc G mà có liên thông với G.

3.1. Duyệt theo chiều sâu (Depth First Search).

3.1.1. Phép duyệt.

Xuất phát từ một đỉnh v được thăm, tiếp theo đỉnh w chưa được thăm kề với v được chọn và một phép duyệt theo chiều sâu xuất phát từ w lại được thực hiện với các đỉnh w_1 kề với w,… khi hết một nhánh thì được chuyển sang nhánh khác. Phép duyệt theo nguyên tắc này được gọi là duyệt theo chiều sâu. Phép duyệt sẽ kết thúc khi không có một đỉnh nào kề với đỉnh đã được thăm mà chưa được thăm



Hình 7.9: Đồ thị không định hướng, liên thông

Giả thiết, với đồ thị hình 7.9 và đỉnh xuất phát là v_1, thì phép duyệt theo chiều sâu sẽ được một dãy các đỉnh sau:

v_1, v_2 (*cũng có thể v_3*), v_4 (*cũng có thể v_5*), v_5, v_3, v_7 do v_7 không có đỉnh kề nào chưa được thăm nên phải quay lại v_3 và cuối cùng là v_6.

3.1.1. Giải thuật.

Để kiểm tra việc duyệt mỗi đỉnh đúng một lần ta dùng một mảng mart gồm n phần tử tương ứng với n đỉnh của đồ thị. Khởi đầu gán giá trị 0 cho tất cả các phần tử của mảng, mỗi khi có một đỉnh i được thăm ta sẽ gán mart[i]=1.

Giải thuật được mô tả bằng hàm đệ qui DFS (Depth First Search) như sau:

```
void  DFS(int v)
{    thăm đỉnh v: mart[v]=1
          for mỗi đỉnh w kề với v
       if (mart[w]==0)
          {  mart[w]=1;
                   DFS(w);}
```

3.1.1. Ứng dụng của giải thuật.

Nhiều giải thuật sử dụng giải thuật duyệt theo chiều sâu:

- Xác định *các thành phần liên thông* của đồ thị
- *Sắp xếp tô pô* cho đồ thị.
- Xác định *các thành phần liên thông mạnh* của đồ thị có hướng
- …

3.2. Duyệt theo chiều rộng (Bredth First Search).

Xuất phát từ một đỉnh v được thăm đầu tiên, nhưng khác với DFS tiếp theo là các đỉnh chưa được thăm mà kề với v sẽ được thăm kế tiếp nhau và một phép duyệt theo chiều rộng xuất phát từ các đỉnh kề với v vừa được thăm lại được thực hiện,… khi hết các đỉnh kề với một đỉnh đã được thăm thì được chuyển sang đỉnh kề khác. Phép duyệt theo nguyên tắc này được gọi là duyệt theo chiều rộng.

Phép duyệt sẽ kết thúc khi không có một đỉnh nào kề với đỉnh đã được thăm mà chưa được thăm.

Giả thiết, với đồ thị hình 7.9 và đỉnh xuất phát là v_1, thì phép duyệt theo chiều rộng sẽ được một dãy các đỉnh sau: v1, v2, v3 rồi đến v4, v5 tiếp theo v7, v6.

3.2.1. Giải thuật.

Ta cũng dùng một mảng mart gồm n phần tử tương ứng với n đỉnh của đồ thị. Khởi đầu gán giá trị 0 cho tất cả các phần tử của mảng, mỗi khi có một đỉnh i được thăm ta sẽ gán mart[i]=1.

Giải thuật BFS (Bredth First Search) có thể cài đặt không đệ qui bằng cách dùng thêm một Queue để lưu các đỉnh cần được thăm.

Giải thuật được mô tả bằng hàm BFS như sau:

```
void  BFS(int v)
    {
        Khởi tạo hàng đợi Q
        chèn v vào hàng đợi Q
        đánh dấu đã thăm v: mart[i]←1.
         while (Q ≠  )
           {  lấy đỉnh v ra khỏi Q
              for mỗi đỉnh w kề với v
                 if (mart[w]==0)
                    đưa v vào hàng đợi Q
                    đánh dấu đã thăm w: mart[w]←1.
            }
     }
```

3.2.1. Ứng dụng của giải thuật.

Nhiều giải thuật sử dụng giải thuật duyệt theo chiều rộng:

- Tìm tất cả các đỉnh trong trong một *thành phần liên thông*
- Bài toán tìm đường đi ngắn nhất
- …

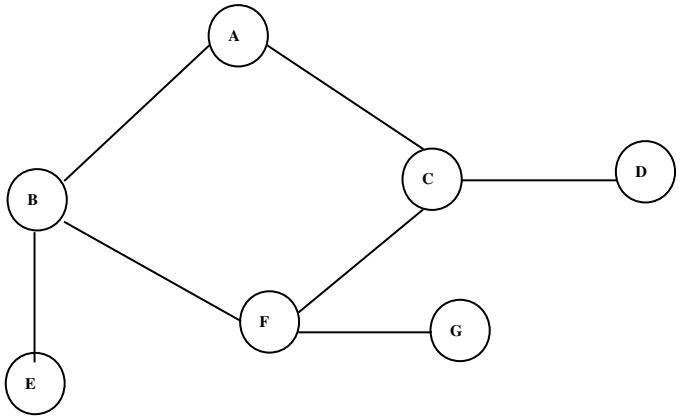
CÂU HỎI VÀ BÀI TẬP CUỐI CHƯƠNG 7

1) Trình bày khái niệm đồ thị định hướng, đồ thị không định hướng

2) Trình bày các khái niệm đường đi, chu trình, liên thông

3) Hãy nêu một số cách biểu diễn đồ thị mà mình biết và đưa ra nhận xét về ưu điểm và hạn chế của từng cách.

4) Cho đồ thị không định hướng G_1 như hình 7.10 sau:



Hình 7.10: Đồ thị G_1

a. Hãy cho biết đồ thị thuộc loại nào (định hướng, vô hướng, liên thông, không liên thông)?

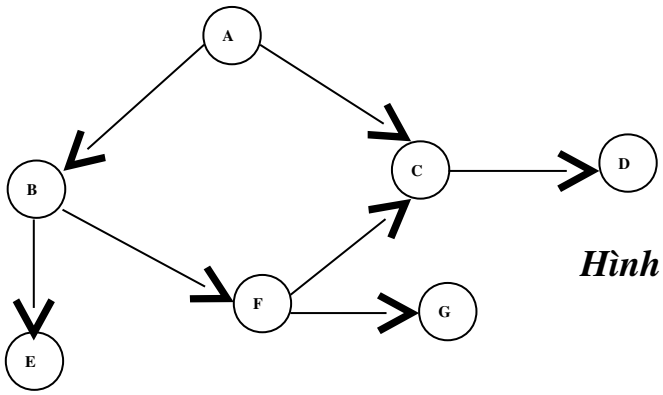
b. Hãy lập ma trận lân cận của G_1.

c. Hãy lập Danh sách các đỉnh kề của G_1.

d. Duyệt đồ thị G_1 theo chiều sâu bắt đầu từ A, B, D.

e. Duyệt đồ thị G_1 theo chiều rộng bắt đầu từ A, B, D.

5) Cho đồ thị định hướng G_2 như hình 7.11 sau:



Hình 7.11: Đồ thị G_2

a. Hãy cho biết đồ thị thuộc loại nào (định hướng, vô hướng, liên thông, không liên thông)?

b. Hãy lập ma trận kề (lân cận) của G1

c. Hãy lập Danh sách các đỉnh kề của G1

d. Duyệt đồ thị G1 theo chiều sâu bắt đầu từ A.

e. Duyệt đồ thị G1 theo chiều sâu bắt đầu từ A.

6) Cài đặt đồ thị G_1 bằng ma trận kề rồi viết các giải thuật:

a. Duyệt theo chiều sâu.

b. Duyệt theo chiều rộng.

7) Cài đặt đồ thị G_2 bằng danh sách các đỉnh kề rồi viết các giải thuật:

a. Đưa ra màn hình các đỉnh kề với từng đỉnh của đồ thị

b. Duyệt theo chiều sâu.

c. Duyệt theo chiều rộng.

8) Hãy viết một ứng dụng có sử dụng giải thuật duyệt theo chiều rộng để kiểm tra tính liên thông của một đồ thị bất kỳ và đưa ra thông báo về số lượng thành phần liên thông của đồ thị

PHỤ LỤC

Phụ lục 1
Biến con trỏ và cấp phát động

1. Khái niệm biến tĩnh, biến động và biến con trỏ:

- *Biến tĩnh:* Là những biến khi khai báo, một lượng ô nhớ cho các biến

này sẽ được cấp phát mà không cần biết trong quá trình thực thi chương trình có sử dụng hết lượng ô nhớ này hay không. Mặt khác, các biến tĩnh (nhất là các biến toàn cục) sẽ tồn tại trong suốt thời gian thực thi chương trình, gồm cả những biến mà chương trình chỉ sử dụng 1 lần rồi bỏ.

Một số hạn chế có thể gặp phải khi sử dụng các biến tĩnh:

- Cấp phát ô nhớ dư, gây ra lãng phí ô nhớ.
- Cấp phát ô nhớ thiếu, chương trình thực thi bị lỗi.

- *Biến động:* Biến động được tạo ra và khởi tạo giá trị khi chương trình

hoạt động. Đặc biệt là biến động được thu hồi bộ nhớ ngay khi có lệnh yêu cầu giải phóng vùng nhớ nó đang chiếm giữ để có thể sử dụng vào việc khác.

Đặc điểm của biến động:

- Chỉ phát sinh trong quá trình thực hiện chương trình chứ không phát sinh lúc bắt đầu chương trình.
- Khi chạy chương trình, kích thước của biến, vùng nhớ và địa chỉ

vùng nhớ được cấp phát cho biến có thể thay đổi.

- Sau khi sử dụng xong có thể giải phóng để tiết kiệm chỗ trong bộ

nhớ.

- *Biến con trỏ:* Biến động không có địa chỉ nhất định nên ta không thể truy cập đến chúng được. Vì thế, ở các ngôn ngữ lập trình như ngôn ngữ C đã cung cấp cho ta một loại biến đặc biệt để khắc phục tình trạng này, đó là biến con trỏ (pointer) với các đặc điểm:

- Biến con trỏ không chứa dữ liệu mà chỉ chứa địa chỉ của dữ liệu (

hay chứa địa chỉ của ô nhớ chứa dữ liệu), nó dùng để truy cập biến thông qua địa chỉ biến và chương trình tham chiếu biến gián tiếp qua địa chỉ này.

- Kích thước của biến con trỏ không phụ thuộc vào kiểu dữ liệu, luôn có kích thước cố định (2 bytes hoặc 4 bytes,…).
-

2. Khai báo biến con trỏ :

2.1. Khai báo biến con trỏ có kiểu:

- Cú pháp: <Kiểu dữ liệu> * <Tên biến >;

  Hoặc <Kiểu dữ liệu>* <Tên biến >;

- Giải thích :

<Kiểu dữ liệu>: Là tên một kiểu dữ liệu trong ngôn ngữ C, *<kiểu dữ liệu> ở đây là kiểu dữ liệu được trỏ tới, không phải là kiểu của bản thân con trỏ.*

* <Tên biến >: Dấu * trước tên biến thể hiện biến này là biến con trỏ. Nó chứa địa chỉ của các biến có cùng <Kiểu dữ liệu> mà nó sẽ trỏ đến.

Ví dụ 1:

int *pa, *pb;

Khai báo 2 biến pa, pb là 2 biến con trỏ, sẽ chứa địa chỉ các biến có kiểu int mà nó trỏ đến.

***Ví dụ 2*:**

float *px, *py;

Khai báo 2 biến px, py là 2 biến con trỏ, sẽ chứa địa chỉ các biến có kiểu float mà nó trỏ đến.

2.2. Khai báo biến con trỏ không kiểu:

Nếu chưa muốn khai báo kiểu dữ liệu mà biến con trỏ sẽ trỏ đến, ta sử dụng:

- Cú pháp:

void *<tên biến con trỏ>;

- Giải thích:
  - Từ khóa void thay cho tên một kiểu dữ liệu bất kỳ. Sau đó, nếu ta

muốn con trỏ *<tên biến con trỏ>* chỉ đến kiểu dữ liệu gì cũng được.

  - Tác dụng của khai báo này là chỉ dành ra một số byte nhất định (2

bytes hoặc 4 bytes) trong bộ nhớ để cấp phát cho biến con trỏ *<tên biến con trỏ>*.

- *Ví dụ 3:*

```
void main()
    {    int    a;
         float  b;
         void   *p, *q;
         p=&a;              //p trỏ tới địa chỉ biến a có kiểu int
         q=&b;              //q trỏ tới địa chỉ biến b có kiểu float
         printf("a=%d", *(int *)p);      //sử dụng phép ép kiểu
         printf("a=%d", *(float *)q);
    }
```

3. Các phép toán trên biến con trỏ

3.1. Toán tử địa chỉ &:

Như ta đã biết, ngôn ngữ C qui định dấu & trước tên một biến là làm việc với địa chỉ của biến đó. Khi muốn biến con trỏ trỏ tới địa chỉ một biến tĩnh ta thực hiện phép gán sau:

- Cú pháp: <tên biến con trỏ>=&<tên biến>;
- Giải thích:
  - &<*tên biến*> tức là, làm việc với địa chỉ của <*tên biến*>
  - Thông qua phép gán này biến con trỏ <*tên biến con trỏ*> sẽ trỏ tới

địa chỉ của <*tên biến*>. ***Nghĩa là pa tương đương với &a.***

- *Ví dụ 4 :*

```
int a, *pa, *p;
pa=&a;      //sau phép gán này biến con trỏ pa sẽ trỏ tới địa chỉ biến a
p=pa;       //biến p cũng trỏ tới địa chỉ biến a
```

- *Lưu ý*:
  - Kiểu dữ liệu của biến tĩnh và kiểu dữ liệu của biến con trỏ trong

phép gán cần phải phù hợp với nhau. Ví dụ sau đây chương trình dịch sẽ báo lỗi do không tương thích kiểu dữ liệu:

```
float a;            //a là biến có kiểu số thực
int *pa;        //pa là biến con trỏ có kiểu số nguyên
...
pa=&a;      /*phép gán sai vì kiểu dữ liệu 2 biến không tương thích, muốn phép gán đúng ta sử dụng phép ép kiểu */
```

  - Phép gán <*tên biến con trỏ*>=&<*tên biến*>*;* là phép toán bắt buộc

vì C qui định một biến con trỏ trước khi trỏ tới một giá trị, thì cần phải trỏ tới một địa chỉ cụ thể, nếu không C sẽ báo lỗi.

  - Khi gán địa chỉ của một biến cho một biến con trỏ, mọi sự thay đổi

trên nội dung ô nhớ mà con trỏ trỏ tới sẽ làm giá trị của biến thay đổi theo (thực chất nội dung ô nhớ và biến chỉ là một). Ta sẽ hiểu rõ hơn ở ví dụ 3.5.

3.2. Toán tử tham chiếu *:

Dấu * trước tên một biến khi khai báo để chỉ biến đó là biến con trỏ. Nhưng dấu * trước tên biến con trỏ là để truy xuất trực tiếp đến giá trị được lưu trữ ở địa chỉ mà biến con trỏ trỏ tới.

- *Ví dụ:* *pa=a;
- Giải thích:
  - * *pa* tức là, truy xuất tới giá trị mà *biến con trỏ pa trỏ tới*
  - Giá trị biến a sẽ bị thay đổi theo giá trị mà *biến con trỏ pa trỏ tới*. Ví dụ *pa=a+9, sau phép gán này giá trị biến a cũng tăng thêm 9 đơn vị.
- *Ví dụ 5 :*

```
#include <stdio.h>
#include <stdlib.h>
void  main()
{       int a= 100, *pa, b ;
        b=a;            //sao lưu giá trị biến a sang biến b
        pa=&a;  /* cho biến con trỏ pa trỏ tới địa chỉ của biến a, đây
                   là  phép gán bắt buộc trước lệnh  *pa=a+9;  */
        *pa=a+9;      /*sau phép gán giá trị biến con trỏ pa trỏ tới giá
                   trị 109 Giá trị của biến a cũng bị thay đổi theo */
              //Các lệnh in giá trị của các biến
        printf (" \n Địa chỉ của biến b: %p", &b);
        printf (" \n Giá trị của biến b : %d",  b);
        printf (" \n Địa chỉ của biến a: %p", &a);
        printf (" \n Giá trị của biến a: %d",  a);
        printf (" \n Địa chỉ của biến con trỏ pa: %p", &pa);
        printf (" \n Giá trị của biến con trỏ pa: %p", pa);
        printf (" \n Giá trị  biến pa trỏ tới: %d", *pa);
        getch();
}//kết thúc hàm main
```

- Kết quả chạy chương trình:

Địa chỉ của biến b: **FFF0**

Giá trị của biến b: 100

Địa chỉ của biến a: **FFF4**

Giá trị của biến a: **109**

Địa chỉ của biến con trỏ pa: FFF2

Giá trị của biến con trỏ pa: **FFF4**

Giá trị biến pa trỏ tới: **109**

- Mô tả kết quả trên trong bộ nhớ :

Câu lệnh	**Địa chỉ biến tĩnh**	**Giá trị biến tĩnh**
int a=100;	**FFF4**	100
int b;	**FFF0**	Chưa xác định
b=a;	**FFF0**	100

Câu lệnh	**Địa chỉ con trỏ**	**Địa chỉ con trỏ trỏ tới**	**Giá trị con trỏ trỏ tới**
int *pa;	FFF2	Có thể chưa xác định	Có thể chưa xác định
pa=&a;	FFF2	FFF4	100
*pa=a+9;	FFF2	FFF4	109

- ***Lưu ý:***
  - Sự liên quan giữa biến con trỏ và biến tĩnh:

Loại biến	**Địa chỉ**	**Địa chỉ trỏ tới**	**Giá trị**	**Ghi chú**
Biến tĩnh x	&x	&x	x	- Dấu & trước tên biến chỉ địa chỉ của biến - p=&x: Cho con trỏ p trỏ tới địa chỉ của biến x (p tương đương với &x) - *p tương đương với x, các lệnh dùng được với x cũng có thể
Biến con trỏ p	&p	p=&x	*p	

				dùng được với *p

• Dùng lệnh printf với mã %p để in ra màn hình (hoặc máy in) địa chỉ biến con trỏ và địa chỉ biến con trỏ trỏ tới. Không nên dùng lệnh scanf để nhập giá trị cho biến con trỏ.

3.3. Phép chuyển (ép) kiểu:

Phép gán thường thực hiện cho hai con trỏ cùng kiểu. Muốn gán các con trỏ khác kiểu phải dùng phép ép kiểu theo cú pháp sau:

Cú pháp: (Kiểu dữ liệu mới) *<tên biến con trỏ>;

Hoặc *(Kiểu dữ liệu mới *) <tên biến con trỏ không kiểu>;

Ví dụ 6:

```
float *p1, x =9.5;
void *p;
int *p2=NULL;
                        // p1 trỏ tới ô nhớ x có chứa giá trị 9.5
p1 = &x;
printf("*p1= %f\n", *p1);
p=p1;                          // p, p1 cùng trỏ vào địa chỉ biến x
                        //in ra giá trị con trỏ p trỏ tới
printf("*p= %f\n",*(float *)p);
*p2 = (int)* p1;                    //  *p2 trỏ tới giá trị 9
printf("*p2= %d",*p2);
```

Ví dụ 7:

```
int a, b, *p;
char c;
p = &a;
*p = 97;                    // sau lệnh gán này biến a=97
b = *p;                     // sau lệnh gán này biến b=97
c = (char )* p;             // sau lệnh gán này biến c = 'a'
```

3.4. *Toán tử cộng, trừ con trỏ với một số nguyên và phép tăng giả.*

Ta có thể cộng (+), trừ (-) một biến con trỏ với 1 số nguyên **n** nào đó; kết quả trả về là 1 con trỏ. Con trỏ này chỉ đến vùng nhớ cách vùng nhớ của con trỏ hiện tại **n** phần tử.

Ví dụ 8:

```
int  a[100], *pa;
pa = &a[10];          //pa trỏ tới địa chỉ phần tử a[10]
pa ++;                //pa trỏ tới địa chỉ phần tử a[10+1]
pa +5;                //pa trỏ tới địa chỉ phần tử a[10+5]
pa --;                //pa trỏ tới địa chỉ phần tử a[10-1]
pa -3;                //pa trỏ tới địa chỉ phần tử a[10-3]
```

- Phép tăng (++), giảm (--) không có nghĩa là cho biến con trỏ trỏ sang byte bên cạnh, mà thực chất là sang phần tử bên cạnh. Biến con trỏ char truy nhập 1 byte, con trỏ int truy nhập 2 byte, con trỏ float truy nhập tới 4 byte, ,…

Ví dụ 9:

```
int *pa;
pa = (int*) malloc(20); /* Cấp phát vùng nhớ 20 byte=10 số nguyên*/
int *pb, *pc;
pb = pa + 7;
pc = pb - 3;
```

- Phép trừ 2 biến con trỏ cùng kiểu sẽ trả về 1 giá trị nguyên (int). Đây chính là khoảng cách (số phần tử) giữa 2 con trỏ đó (trong ví dụ trên pc-pa=4).
- Không có phép cộng 2 biến con trỏ với nhau
- *Các phép toán trong mục này không thực hiện với biến con trỏ void, biến con trỏ hàm.*

3.5. Toán tử so sánh:

- ***Đối với biến con trỏ (p)***: Sử dụng toán tử ==, != kiểm tra xem 2 biến con

trỏ có cùng trỏ vào một địa chỉ hay không (đương nhiên 2 biến con trỏ đó phải cùng kiểu dữ liệu với nhau), hoặc một biến con trỏ có trỏ vào giá trị hằng NULL không.

p1==p2 nếu địa chỉ p1 trỏ tới bằng địa chỉ p2 trỏ tới.

p1!=p2 nếu địa chỉ p1 trỏ tới khác với địa chỉ p2 trỏ tới.

Sử dụng toán tử > , >=, <, <= kiểm tra về độ cao thấp giữa 2 địa chỉ, biến con trỏ có giá trị nhỏ hơn thì trỏ vào địa chỉ thấp hơn.

p1<p2 nếu địa chỉ p1 trỏ tới thấp hơn địa chỉ p2 trỏ tới.

p1>p2 nếu địa chỉ p1 trỏ tới cao hơn địa chỉ p2 trỏ tới.

Con trỏ void có thể đem so sánh với tất cả con trỏ khác.

- ***Đối với giá trị biến con trỏ trỏ tới (*p):*** Các toán tử so sánh được thực

hiện hay không là phụ thuộc vào kiểu dữ liệu mà biến con trỏ trỏ tới.

3.6. Hằng con trỏ:

- *Hằng con trỏ NULL*: Là một giá trị đặc biệt của biến con trỏ, bất kỳ biến

con trỏ nào nếu được gán giá trị NULL *(*<tên biến con trỏ>=NULL)* để báo rằng biến con trỏ này không trỏ vào đâu cả (giống như lệnh gán biến so=0).

- *Con trỏ chỉ đến đối tượng hằng:* Là những con trỏ mà chỉ có thể gán giá
-
- trị cho con trỏ một lần duy nhất, nhưng được phép dùng tham chiếu để thay đổi giá trị của biến.

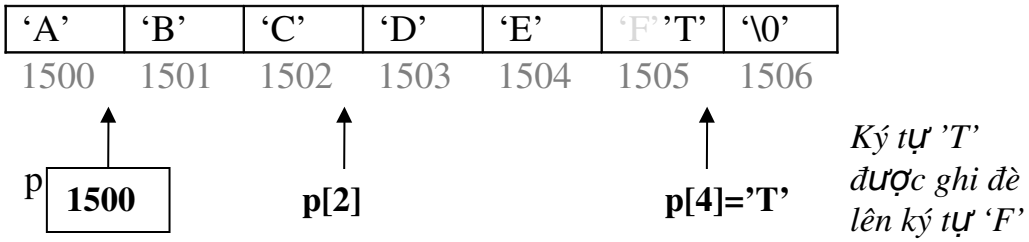
Kiể u * const p = giá trị;

Ví dụ:

```
char xau1[] = "ABCDEF";
char * const p = xau1;   //hoặc char*  const p="ABCDEF";
p++;                     /* sai, vì p     xau1, không thay đổi được vùng
                            nhớ mà p trỏ tới*/
```

p[2]++; /*đúng vì p[2] xau1[2], hoàn toàn có thể thay đổi giá trị vùng nhớ mà p trỏ đến*/

p[5]='T';



- *Con trỏ hằng:* Là những con trỏ mà chỉ *trỏ vào 1 vùng nhớ cố định*, có

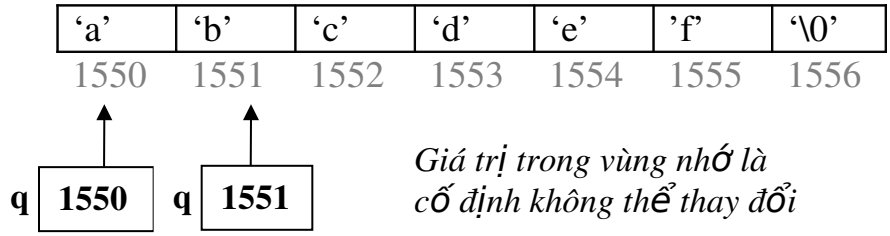
thể thay đổi địa chỉ mà nó trỏ đến, nhưng lại không thể tham chiếu để thay đổi giá trị của biến mà nó trỏ đến.

Kiểu const *p = giá trị hằng;

hoặc Const kiểu *p = giá trị hằng;

Ví dụ:

```
char xau2[] = "abcdef";
const  char* q = xau2;     // hay const char * p = xau2;
q++;                       // đúng, *q[1]=='b'; *q=="bcdef";
q[2]++;              /* sai, không thay đổi được giá trị trong
                           vùng nhớ  */
q[2]='H';                  //sai
```



- *Lưu ý :*

• Địa chỉ của một biến được xem như một con trỏ hằng, do đó nó không được phép gán, tăng hoặc giảm.

Ví dụ 10:

```
int a, b, *p;
p = & a;
p ++;                    // đúng
( & a) ++;               /* sai vì địa chỉ của một biến được coi
                            là con trỏ hằng*/
```

- Con trỏ không trỏ đến biến khác được, cũng không dùng tham chiếu

để thay đổi giá trị của biến được.

```
int x=100;
const int *const px = &x;
px++;                    //sai, không trỏ sang biến khác được
*px=5;                   /*sai, không thay đổi được giá trị của biến
                           được trỏ đến*/
```

3.7. Cấp phát vùng nhớ cho biến con trỏ:

Trước khi sử dụng biến con trỏ, ta nên cấp phát vùng nhớ cho biến con trỏ để quản lý địa chỉ. Việc cấp phát được thực hiện nhờ các hàm malloc(), calloc(), realloc() trong thư viện alloc.h. (hay stdlib.h). hoặc cho biến con trỏ trỏ vào địa chỉ của biến tĩnh: p=&a; mục 1.3.1

Khi sử dụng các hàm trên ta phải ép kiểu vì nguyên mẫu các hàm này trả về con trỏ kiểu void nếu cấp phát thành công và trả về NULL nếu cấp phát không thành công.

Hàm malloc(): Hàm thường dùng để cấp phát bộ nhớ động cho biến con trỏ có kiểu dữ liệu cơ sở, cấu trúc. Lưu ý là vùng nhớ được cấp phát có thể đang lưu giữ giá trị cũ mà chưa bị xóa.

Cú pháp: void *malloc(số ô nhớ cần cấp phát);

Ép kiểu:

(Kiểu dữ liệu *) malloc(số ô nhớ cần cấp phát);

Hoặc: void *malloc(n * sizeof(Kiểu dữ liệu);

(Kiểu dữ liệu *) malloc(n* sizeof(Kiểu dữ liệu);

Ví dụ 11:

```
int   *p;
p=(int*) malloc(20);          /*cấp phát vùng nhớ kích thước 20 bytes
                               cho 10 số nguyên */
```

Hoặc

```
p = (int*)malloc(sizeof(int));          /* Cấp phát vùng nhớ có kích
                                        thước bằng với kích thước của
                                        một số nguyên */
p = (int*)malloc(10*sizeof(int));       /*cấp phát vùng nhớ kích thước
                                        20 bytes cho 10 số nguyên */
```

a. *Hàm calloc()*: Hàm thường dùng để cấp phát bộ nhớ động cho kiểu dữ liệu do người dùng tự định nghĩa, ít dùng với kiểu cơ sở. Đặc biệt vùng nhớ động được cấp phát bởi hàm calloc sẽ được set to zeros (đưa về giá trị 0).

Cú pháp: void *calloc(n, sizeof(Kiểu dữ liệu);

```
struct sinhvien
{       char   masv[10];
        char   htsv[30];
        …
};
        sinhvien   *q;
                                        /* Cấp phát vùng nhớ có thể
                                        chứa được 10  bản ghi sinhvien
                                        */
                                        q=(struct      sinhvien*)calloc(10,
                                        sizeof(struct sinhvien));
```

b. *Hàm realloc():* Được dùng để cấp phát lại bộ nhớ động.

Cú pháp: void *realloc(tên biến con trỏ, số byte mới);

```
        int   *q;
```

```
q = (int*)malloc(sizeof(int));
….
q= (int*)realloc(q, 20);          /* Cấp phát vùng nhớ có thể
                                  chứa được 10 số nguyên*/
        …
free(q);
```

Lưu ý:

- Khi cấp phát lại thì nội dung vùng nhớ trước đó vẫn tồn tại.
- Kết quả trả về của hàm là địa chỉ đầu tiên của vùng nhớ mới. Địa chỉ này có thể khác với địa chỉ được chỉ ra khi cấp phát ban đầu.

c. *Hàm free():* Được dùng để giải phòng vùng nhớ động đã cấp phát, khi không còn dùng đến nữa.

Cú pháp: void free(void *<biến con trỏ>)

Ví dụ: free(q);

Lưu ý: mỗi khi không dùng đến biến động cần phải giải phóng ngay vùng nhớ đã cấp phát.

4. Mối liên quan giữa con trỏ, hàm, mảng, chuỗi và cấu trúc.

4.1. Biến con trỏ là tham số hình thức của hàm.

Mặc nhiên, việc truyền tham số cho hàm trong C là truyền theo giá trị, nghĩa là giá trị của tham số thực không bị thay đổi trước và sau khi gọi hàm.

Muốn sau khi kết thúc hàm giá trị của tham số thực sự thay đổi theo sự thay đổi của tham số hình thức thì ta phải *khai báo tham số hình thức là biến con trỏ (hoặc thêm dấu & vào trước tham số hình thức)* và được gọi là tham số hình thức biến (hay tham biến). Đây gọi là truyền biến.

Đặc điểm của truyền biến:

- Là tham số thực sự truyền địa chỉ vùng nhớ của mình cho tham số

hình thức biến.

- Mọi sự thay đổi trên vùng nhớ được quản lý bởi *tham số hình thức*

biến của hàm sẽ ảnh hưởng đến *vùng nhớ đang được quản lý bởi tham số thực sự tương ứng*.

- Mỗi hàm chỉ có thể trả ra một giá trị (bằng lệnh return). Để hàm có

thể trả ra nhiều giá trị ta lên dùng cách truyền biến.

Ví dụ 12: Xét chương trình sau đây:

```
/* Khoi tao 2 so */
#include <stdio.h>
#include <conio.h>
void truyenBien (int *, int *);
void main(void)
{
        int x=6, y=4;
        printf(“Gia tri bien x va y TRUOC khi goi ham:\n”);
        printf("x = %d, y = %d\n", x, y);
        truyenBien(&x, &y);
        printf(“Gia tri bien x va y SAU khi goi ham:\n”);
        printf("x = %d, y = %d\n", x, y);
        getch();
}
void truyenBien(int *px, int *py)
{
        *px = 3;                    //gan 3 cho noi dung cua px
        *py = 5;                    //gan 5 cho noi dung cua py
        printf(“Gia tri bien x va y trước khi KET THUC ham:\n”);
        printf("x = %d, y = %d\n", *px, *py);
}
```

Kết quả thực hiện chương trình:

```
Gia tri bien x va y TRUOC khi goi ham:
x=6 y=4
Gia tri bien x va y truoc khi KET THUC ham:
x=3 y=5
Gia tri bien x va y SAU khi goi ham:
x=3 y=5
```

4.2. Biến con trỏ là kiểu kết quả hàm trả về :

Cú pháp: Kiểu *hàm (danh sách tham số).

Ví dụ: Hàm strstr() lấy ra một phần của chuỗi s1, bắt đầu từ chuỗi s2. Theo qui cách làm việc của hàm thì kết quả hàm trả ra phải được gán cho một biến con trỏ có kiểu char.

```
//char *strstr(const char *s1, const char *s2);
#include <stdio.h>
#include <conio.h>
#include <string.h>
void main(void)
{
        char *str1 = "Borland International", *str2 = "Inter", *ptr;
        ptr = strstr(str1, str2);
        printf("The substring is: %s\n", ptr);
        getch();
}
```

4.3. Sự tương quan giữa con trỏ và mảng.

a. Con trỏ và mảng một chiều.

Với khai báo:

```
int a[10], *p;
```

Thực chất C qui định a tương đương với &a[0], nghĩa là tên mảng là địa chỉ, hay con trỏ trỏ tới phần tử đầu tiên của mảng.

Ta có thể viết: p=a; hay p=&a[0];

Khi đó để truy xuất tới phần tử thứ i của mảng A, ta có các cách sau:

a[i] ⇔ *(a + i) ⇔ *(p + i)

& a[i] ⇔ (a + i) ⇔ (p +i)

Chúng ta cần lưu ý là chỉ có tên của mảng mới mang giá trị địa chỉ (tức con trỏ), còn phần tử của mảng (a[i]) chỉ là biến bình thường. Cụ thể, a hay a[] là biến con trỏ còn a[0], a[1],… là các giá trị của biến.

Ví dụ 13: Hàm nhập mảng và hiện mảng:

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
        //định nghĩa hàm Nhapmang
void Nhapmang(int  *p,int n)
{       int  i;
        for (i = 0; i< n; i++)
        {       printf("\n nhap a[%d]:",i);
                scanf ("%d", p+i );
        }
}
        //định nghĩa hàm Hienmang
void Hienmang(int *p, int n)
{       int i;
        for (i = 0; i<n;i++)
                printf ("%4d", *(p+i));
}
        //định nghĩa hàm chính main
void main(void)
{       int a[100];
        int n=10;
        Nhapmang(a,n);  //Biến mảng tương đương biến con trỏ
        Hienmang(a,n);
        getch();
```

}

- *Sự khác nhau giữa con trỏ và mảng*:

Biến con trỏ thì có thể tăng, giảm hoặc gán còn biến mảng là một con trỏ hằng do đó không thể tăng, giảm hoặc gán.

Ta có thể lấy địa chỉ của con trỏ nhưng không thể lấy địa chỉ của mảng vì bản thân mảng đã là địa chỉ .

Khi ta khai báo một mảng thì chương trình dịch sẽ cấp phát một vùng nhớ cho nó. Còn biến con trỏ khi được khai báo chỉ được cấp một nhớ mà nội dung của nó thường chưa xác định: ⇒ phải có p = a để p chỉ tới a

Khi khai báo mảng ta phải chỉ ra số lương phần tử, nếu thừa nhiều sẽ lãng phí bộ nhớ, Nếu thiếu chương trình sẽ lỗi. Ta có thể khắc phục điều này bằng cách tạo một mảng bằng con trỏ và phải xin cấp phát một vùng nhớ bằng hàm malloc (), nếu thiếu có thể dùng hàm realloc() để xin cấp phát thêm ô nhớ.

*/*Viết lại hàm main của ví dụ 3.12 bằng cách dùng biến con trỏ thay mảng*/*

```
void main(void)
{       int *p, n=10;
p =(int*) malloc (10 * sizeof (int));
Nhapmang(p,n);
Hienmang(p,n);
getch();
}
```

b. Con trỏ và mảng hai chiều.

Chúng ta đã biết, C quan niệm mảng hai chiều như mảng (một chiều) của mảng.

Ví dụ 14: float a[2][3];

Khi đó ta có các phần tử:

a	trỏ tới a[0][0];	a[0];
a+1	trỏ tới a[0][1];	a[1];
a+2	trỏ tới a[0][2];	a[2];

a +3 trỏ tới a[1][0]; a[3];

…

c. Mảng 2 chiều và biến con trỏ:

Ví dụ 15: float a[2][3], *p;

Khi đó:

p=(float *)a; *//với mảng 2 chiều phải dùng phép ép kiểu*

(Không nên dùng: p=a; vì đây là phép gán được dùng với mảng một chiều)

Sau phép gán trên thì:

p	trỏ tới địa chỉ	a[0][0]
p+1	trỏ tới địa chỉ	a[0][1]
p +2	trỏ tới địa chỉ	a[0][2]
…		
Và *p	trỏ tới giá trị	a[0][0]
*(p+1)	trỏ tới giá trị	a[0][1]
…		

Ví dụ 16 : Hàm hiện mảng 2 chiều:

```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>
void hienMT(int *p)
{
        for (int i=0;i<6;i++)
                printf("%d  %c", *(p+i),(i==2)? '\n':' ');
}
void main()
{       clrscr();
        int *p,a[2][3]={1,2,3,4,5,6};
        printf("\n");
```

```
        hienMT((int *)a);          //phải dùng phép ép kiểu
        getch();
    }
```

4.4. Con trỏ và chuỗi ký tự

C quan niệm chuỗi ký tự là một mảng ký tự, nhưng con trỏ ký tự có hơi khác với mảng ký tự.

Ví dụ 17 : Khai báo:

```
char  s[50];      // Hoặc char s[] = "ABCD";
  char *p;              // Hoặc char *p = "ABCD";
```

Phép gán :

```
s = "ABCD";   /* sai (s là một hằng địa chỉ), phải dùng
              strcopy(s,"ABCD")*/
p="ABCD";     //đúng (p là một con trỏ)
```

Truy cập phần tử:

```
s[0]=='A'      tương đương          *(p+0)=='A'
s[1]=='B'      tương đương          *(p+1)=='B'
```

Đọc, viết chuỗi ký tự:

```
gets(s);   tương đương    gets(p);   //đọc chuỗi ký tự từ bàn phím
puts(s);   tương đương    puts(p);         //viết chuỗi ký tự ra màn hình
```

hoặc

```
printf("%s",s);   tương đương      printf("%s",p);
                                   //viết chuỗi ký tự ra màn hình
```

- Con trỏ và mảng chuỗi ký tự: C quan niệm mảng chuỗi ký tự là một mảng 2 chiều có kiểu char, nhưng con trỏ chuỗi có hơi khác với mảng chuỗi.

  Ví du 18:

Khai báo: Mảng ten có 5 phần tử, mỗi phần tử chứa tố đa 8 ký tự

```
        // mảng  gồm 5 phần tử có độ dài tối đa là 8 ký tự
char  ten[5][9];
```

*/*hoặc* char ten[5][9]= { "Minh", "Tuan", "Binh", *"Nam", "Ngan" };*/*

// mảng gồm 5 con trỏ có kiểu char

char * ds[5];

*//Hoặc char *ds={ "Minh", "Tuan", "Binh", "Nam", "Ngan" };*

Phép gán:

Ten[0]= "Minh"; tương đương **ds[0]= "Minh";**

Lưu ý: Khởi tạo mảng mà không dùng con trỏ thì lượng ô nhớ cấp phát cho mỗi phần tử của mảng đều bằng với số ký tự của chuỗi dài nhất; Trong khi đó, nếu khởi tạo bằng mảng con trỏ thì lượng ô nhớ sẽ cấp phát vừa đủ cho từng phần tử của mảng.

Truy cập tới phần tử mảng:

Ten[0]== "Minh" tương đương **ds[0]=="Minh"**

Ten[1]== "Tuan" tương đương **ds[1]=="Tuan"**

…

Ví dụ 19: Dùng mảng chuỗi ten chứa dữ liệu, cho mảng con trỏ p trỏ vào mảng chuỗi ten

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
#include <alloc.h>
void NhapMchuoi( char xau[][10],char *p[], int n);
void InMchuoi( char *p[], int n );
void SXMchuoi( char *p[], int n);
int Menu();
void main(void)
{
    int n,chon;
```

```
    char ten[10][10];
    char *p[10];
    do
      { chon=Menu();
        switch (chon)
        {
          case 1:
            printf("\n nhap so luong ten <=10:");
                scanf("%d",&n);
            NhapMchuoi(ten,p,n);
            break;
          case 2:
            InMchuoi(p,n);
            getch();
            break;
         case 3:
            SXMchuoi(p,n); break;
       default: printf("\n Ban chon chua dung!");
     }
   } while (chon!=0); free(p);
}
int Menu()
{  int chon;
   clrscr();
   printf("\n MOT SO THUAT TOAN VE MANG CHUOI\n\n");
   printf(" 1. Nhap ho ten\n");
   printf(" 2. Hien danh sach ho ten\n");
   printf(" 3. Sap xep ho ten\n");
   printf(" 0. Thoat khoi chuong trinh\n");
   printf(" Nhan so tuong ung de chon chuc nang:\n");
   scanf("%d",&chon);
```

```
    return (chon);
}
void NhapMchuoi(char xau[][10],char *p[], int n)
{  int i; char x[10];
   fflush (stdin);
   for (i=0; i < n ; i++)
          {
               printf("\n Nhap chuoi thu %d ",i);
               gets(xau[i]);p[i]=xau[i];
          }
}
void InMchuoi( char *p[], int n)
{
        int i;
        for (i=0; i < n ; i++)
            puts(p[i]);
}
void SXMchuoi( char *p[], int n)
{       int i, j;
        char *tg;
        for (i=0; i<n-1; i++)
             for (j=i+1; j<n; j++)
                       if (stricmp (p[i],p[j]) >0)
                       {
                              tg=p[i];
                              p[i]=p[j];
                              p[j]=tg;
                       }
}
```

Ví dụ 20: Dùng mảng con trỏ chuỗi p để chứa xâu ký tự

/*thay hàm nhapMCTchuoi vào chương trinh trên và chạy thử, nhận xét kết quả*/

```
void NhapMCTchuoi(char *p[], int n)
    {   char x[10];
        int i;
        fflush (stdin);
        for (i=0; i < n ; i++)
          {  p[i]=(char*)malloc(10 * sizeof(char));
             printf("\n Nhap chuoi thu %d ",i);
             gets(x); strcpy(p[i],x);
          }
    } /*lưu ý dùng hàm free(p) giải phóng bộ nhớ động trước khi kết thúc
    chương trình */
```

4.5. Con trỏ và kiểu cấu trúc.

a. Con trỏ cấu trúc (con trỏ bản ghi):

- Trong C có 2 cách định nghĩa cấu trúc một bản ghi đó là dùng struct và typedef struct :

```
struct Date
 {
      unsigned char day;
      unsigned char month;
      unsigned int year;
 };
Hoặc typedef  struct
 {
      unsigned char day;
      unsigned char month;
      unsigned int year;
 } Date;
```

- Định nghĩa 2 bản ghi lồng nhau. Giả sử để lưu trữ thông tin về một sinh viên có sử dụng kiểu Date lưu thông tin ngày sinh như sau:

```
struct SinhVien
{
   char Hten [35];
   struct Date Ngsinh;          /*sử dụng kiểu Date được định nghĩa
                                 ở cách 1*/
   float   LaptrinhCB, KientrucMT, MangMT, DiemTB;
 }sv1, sv2;                     //sv1, sv2 là 2 biến bản ghi SinhVien
```

Hoặc

```
typedef struct  SinhVien
{
   char Hten [35];
   Date Ngsinh;          /*sử dụng kiểu Date được định nghĩa ở
                         cách 2*/
   float TinDC, LaptrinhCB, KientrucMT, MangMT, DiemTB;
} s1, s2 ;               /*s1, s2 và SinhVien là các  kiểu bản ghi
                         có cấu trúc giống nhau*/
```

- Khai báo biến bản ghi và biến con trỏ bản ghi:

Theo cách 1:

```
struct SinhVien     sv, *p;
```

Theo cách 2:

```
SinhVien     sv, *p;
```

Việc khai báo một biến con trỏ kiểu bản ghi cũng tương tự như khai báo một biến con trỏ có kiểu dữ liệu khác, tức là đặt thêm dấu * vào trước tên biến.

Cú pháp: **struct** <Tên cấu trúc> * <Tên biến con trỏ>;
//theo định nghĩa cách 1

Hay <Tên cấu trúc> * <Tên biến con trỏ>;
//theo định nghĩa cách 2

- Truy cập tới từng phần tử (từng trường) bản ghi:

Đối với biến bản ghi: Dùng dấu chấm (.) để phân cách giữa tên biến bản ghi và tên biến trường

<Tên biến bản ghi>.<Tên biến trường> ;

Ví dụ: sv.Hten=”Nguyen Tuan Nghia”

Đối với biến con trỏ bản ghi: Dùng dấu -> (là dấu – và dấu >) để phân cách giữa tên biến con trỏ bản ghi và tên biến trường

<Tên biến con trỏ bản ghi> -> <Tên biến trường>;

Hay (*<Tên biến con trỏ bản ghi>) . <Tên biến trường>;

Ví dụ:

```
p=&sv;              // cho con trỏ p trỏ vào vùng nhớ của biến sv
printf(“ %s”,p->Hten);              //truy nhập qua con trỏ
Hay
printf(“ %s”,(*p).Hten);          //truy nhập qua vùng nhớ con trỏ
```

Ta cũng có thể sử dụng phép toán * để truy cập vùng dữ liệu đang được quản lý bởi con trỏ cấu trúc để lấy thông tin cần thiết.

- Cấp phát động cho biến con trỏ bản ghi: Cũng dùng hàm malloc() hoặc

calloc để cấp phát.

```
p=(struct SinhVien*)malloc(sizeof(struct SinhVien));
hay
p=(SinhVien*)malloc(sizeof(SinhVien));
```

Ví dụ 21: Sử dụng con trỏ bản ghi viết hàm nhập và hàm xuất giá trị của một bản ghi sinh viên.

```
//Khai báo tiền xử lý
  #include<string.h>
  #include<conio.h>
  #include<stdio.h>
  #include<alloc.h>
//định nghĩa bản ghi Date
```

```
    typedef struct
       { unsigned char day;
         unsigned char month;
         unsigned int year;
        } Date;
  //định nghĩa bản ghi SinhVien
     struct SinhVien
       { char Hten [35];
         Date Ngsinh;
         float   LaptrinhCB, KientrucMT, MangMT, DiemTB;
          };
  // Định nghĩa hàm nhapSinhVien
  void  NhapSinhVien (struct SinhVien *p)
     {   char ht[35]; unsigned  char d; unsigned int n; float diem;
         fflush(stdin);                          //hàm xóa vùng đệm bàn phím
         printf("\n Ho ten: ");  gets(ht) ; strcpy(p->Hten,ht);
             //Nhập ngày, tháng, năm sinh của sinh viên
         printf("ngay sinh: ");    scanf("%u", &d); p->Ngsinh.day=d;
         printf("thang sinh: ");   scanf("%u", &d); p->Ngsinh.month=d;
         printf("nam sinh: ");     scanf("%u", &n); p->Ngsinh.year=n;
             //Nhập điểm cho sinh viên
printf("Diem Lap trinh CB: ");  scanf("%f", &diem); p->LaptrinhCB=diem;
printf("Diem Kien truc MT: ");  scanf("%f", &diem); p->KientrucMT)=diem;
printf("Diem Mang MT: ");        scanf("%f",&diem);  p->MangMT=diem;
       //Tính điểm trung bình
   p->DiemTB=( p-> LaptrinhCB+ p-> KientrucMT+ p-> MangMT )/3;
}
//Định nghĩa hàm hiện thông tin về một bản ghi sinh viên
void HienSinhVien (struct SinhVien *p)
  {             //hiện thông tin sinh viên
       printf ("\n Sinh vien  %35s", p->Hten);
```

```
        printf ("\n sinh ngay %2d thang %2d nam %4d “, p->Ngsinh.day,
                                        p-> Ngsinh.month,  p->Ngsinh.year);
        printf ("\n Diem Lap trinh CB : %4.1f”,  p->LaptrinhCB);
        printf ("\n Diem Kien truc MT : %4.1f”,  p->KientrucMT);
        printf ("\n Diem Mang MT : %4.1f”,  p->MangMT);
        printf ("\n Diem TB : %4.1f”,  p->DiemTB);
        getch();
  }
//Định nghĩa hàm main()
int main(void)
{       clrscr;
        struct SinhVien *p, sv;          /*khai báo biến con trỏ kiểu bản ghi
                                         SinhVien*/
        p=&sv;                        //cho con trỏ p trỏ vào vùng nhớ biến sv
              // hoặc p=(struct SinhVien*)malloc(sizeof(struct SinhVien));
        NhapSinhVien(p) ;                //gọi hàm NhapSinhVien
        printf(“\n HIEN THONG TIN CHO MOT SINH VIEN:”);
        HienSinhVien (p);                // Gọi hàm hienSinhVien
        //hoặc free(p);
        return(0);
}//kết thúc hàm main.
```

b. Con trỏ mảng bản ghi

- Khai báo biến mảng bản ghi và biến con trỏ mảng bản ghi:

Theo cách 1:

```
struct SinhVien     ds[100], *p;
```

Theo cách 2:

```
 SinhVien     ds[100], *p;
```

Khai báo biến con trỏ mảng bản ghi cũng giống như khai báo một biến con trỏ bản ghi, tức là đặt thêm dấu * vào trước tên biến.

- Truy cập tới phần tử của mảng bản ghi và trường:

Đối với biến mảng bản ghi:

<Tên biến bản ghi>[chỉ số phần tử].<Tên biến trường> ;

Ví dụ: ds[1].Hten=”Nguyen Tuan Nghia”;

ds[2]=ds[1];

Đối với biến con trỏ mảng bản ghi

<Tên biến con trỏ bản ghi>[chỉ số phần tử] . <Tên biến trường>;

Hay (*(<Tên biến con trỏ bản ghi> + < chỉ số phần tử>)) . <Tên biến trường>;

Hay (<Tên biến con trỏ bản ghi> + < chỉ số phần tử>) -> <Tên biến trường>;

Ví dụ:

```
p=ds;                 // cho con trỏ p trỏ vào vùng nhớ của biến ds
printf(“ %s”,p[1].Hten);              //truy nhập qua con trỏ
Hay
printf(“ %s”,(*(p+1)).Hten);       //truy nhập qua vùng nhớ con trỏ
hay
(p+i)-> LaptrinhCB= 9;         //truy cập theo kiểu bản ghi
```

Ta phải thêm chỉ số phần tử vào sau tên biến bản ghi

- Cấp phát động cho biến con trỏ bản ghi: Cũng dùng hàm malloc() hoặc calloc để cấp phát

```
p=(struct SinhVien*)malloc(n*sizeof(struct SinhVien));
hay
p=(SinhVien*)malloc(n*sizeof(SinhVien));
```

n là số lượng bản ghi yêu cầu được cấp phát

Ví dụ 22: Sử dụng con trỏ mảng bản ghi viết hàm nhập và hàm xuất giá trị của một bản ghi sinh viên

```
//Khai báo tiền xử lý
#include<string.h>
#include<conio.h>
#include<stdio.h>
#include<alloc.h>
```

```
//định nghĩa bản ghi Date
    typedef struct
      { unsigned char day;
        unsigned char month;
        unsigned int year;
       } Date;
  //định nghĩa bản ghi SinhVien
     struct SinhVien
      { char Hten [35];
        Date Ngsinh;
        float   LaptrinhCB, KientrucMT, MangMT, DiemTB;
        };
  // Định nghĩa hàm nhapDSSV
  void  NhapDSSV (struct SinhVien *p, int n)
     { char ht; unsigned char d; unsigned int nam; float diem;
       for (int i=0; i<n; i++)
         {    fflush(stdin);                    //hàm xóa vùng đệm bàn phím
              printf("\n Ho ten: ");  gets(ht); strcpy(p[i].Hten,ht);
              //Nhập ngày, tháng, năm sinh của sinh viên
           printf("ngay sinh: ");    scanf("%d",&d);  p[i].Ngsinh.day=d;
           printf("thang sinh: ");   scanf("%d", &d ); p[i]. Ngsinh.month=d;
           printf("nam sinh: ");     scanf("%d",&nam);
                                     p[i].Ngsinh.year=nam;
              //Nhập điểm cho sinh viên
           printf("Diem Lap trinh CB: ");  scanf("%f", &diem);
                                           p[i].LaptrinhCB=diem;
           printf("Diem Kien truc MT: ");  scanf("%f", &diem);
                                           p[i].KientrucMT=diem;
           printf("Diem Mang MT: ");        scanf("%f", &diem);
                                           p[i].MangMT=diem;
              //Tính điểm trung bình
```

```
   p[i].DiemTB=( p[i]. LaptrinhCB+ p[i]. KientrucMT+ p[i]. MangMT )/3;
}//kết thúc hàm nhapSinhVien
//Định nghĩa hàm hiện thông tin về một mảng bản ghi sinh viên
void HienSinhVien (struct SinhVien *p, int n)
  { for (int i=0; i<n; i++)
                //hiện thông tin sinh viên
      { printf ("\n Sinh vien  %35s", (*(p+i)).Hten);
        printf ("\n sinh ngay %2d thang %2d nam %4d ",p[i].Ngsinh.day,
                                        p[i]. Ngsinh.month, [i].Ngsinh.year);
        printf ("\n Diem Lap trinh CB : %4.1f", (p+i)->LaptrinhCB);
        printf ("\n Diem Kien truc MT : %4.1f", (p+i)->KientrucMT);
        printf ("\n Diem Mang MT : %4.1f", (p+)->MangMT);
        printf ("\n Diem TB : %4.1f", (p+i)->DiemTB);
        getch();
  }//kết thúc hàm
//Định nghĩa hàm main()
int main(void)
{               int n;        clrscr();
        struct SinhVien *p;   //khai báo biến con trỏ mảng bản ghi SinhVien
        printf("nhap so luong sv:"); scanf("%d",&n);
        p=(struct SinhVien*)calloc(n,sizeof(struct SinhVien));
        NhapDSSV(p,n) ;                 //gọi hàm NhapDSSV
        printf("\n HIEN THONG TIN CHO MOT SINH VIEN:");
        HienDSSV (p,n);                 // Gọi hàm hienDSSV
        free(p);
        return(0);
}//kết thúc hàm main.
```

Ví dụ 23: **Sử dụng các hàm nhập và hiện một sinh viên để viết hàm nhập và hiện một danh sách sinh viên**

```
//định nghĩa hàm nhập một danh sách sinh viên
```

```
void nhapdssv(SinhVien *p, int n)
{ for (int i=0; i<n; i++)
        NhapSinhVien((p+i));
}
//định nghĩa hàm hiện một danh sách sinh viên
void hiendssv(SinhVien *p, int n)
{ for (int i=0; i<n; i++)
        HienSinhVien((p+i));
}
//định nghĩa hàm main
int main(void)
{       clrscr;
        SinhVien *p;
        int n=3;
        p=(SinhVien*)malloc(3*sizeof(SinhVien));
        nhapdssv(p,n) ;                 //gọi hàm NhapSinhVien
        printf("\n HIEN THONG TIN CHO MOT SINH VIEN:");
        hiendssv(p,n);                  // Gọi hàm hienSinhVien
        return(0); free(p);
}//kết thúc hàm main.
```

PHỤ LỤC 2

1) Chương trình quản lý điểm sinh viên được cài đặt bằng danh sách liên kết đơn.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>
//Dinh nghia ban ghi  Date
  typedef struct
  { unsigned char day;
     unsigned char month;
     unsigned int year;
   } Date;
   //Dinh nghia ban  ghi SinhVien
typedef struct
{ char Hten [35];
  Date Ngsinh;
  float   LaptrinhCB, KientrucMT, MangMT, DiemTB;
}SinhVien;
typedef  SinhVien  ElementType;
struct node
{       ElementType                 info;
        struct node*          link;
};
typedef struct node*  listnode;
// Dinh nghia ham nhapSinhVien
void  NhapSinhVien (SinhVien *p)
{ float f; char ht[35];
  unsigned char d;
  unsigned int n;
```

```
 fflush(stdin);                    /ham xoa vung dem ban phim
 printf("\n Ho ten: ");
gets(ht);strcpy(p->Hten,ht);
    //Nhap ngay, thang, nam sinh cua sinh viên
printf("ngay sinh: ");
scanf("%d",&d); p->Ngsinh.day=d;
printf("thang sinh: ");
scanf("%d",&d); p->Ngsinh.month=d;
printf("nam sinh: ");
scanf("%d",&n); p->Ngsinh.year=n;
        //Nhap diem cho sinh viên
printf("Diem Lap trinh CB: ");
scanf("%f",&f); p->LaptrinhCB=f;
printf("Diem Kien truc MT: ");
scanf("%f",&f);p->KientrucMT=f;
printf("Diem Mang MT: ");
scanf("%f",&f); p->MangMT=f;
        //Tinh diem trung bình
p->DiemTB=(p->LaptrinhCB+p->KientrucMT+p->MangMT)/3;
}
//Dinh nghia ham hien thong tin mot ban ghi sinh vien
void HienSinhVien (SinhVien sv)
{               //hien thong tin sinh viên
   printf ("\n Sinh vien  %35s", sv.Hten);
   printf ("\n sinh ngay %2d thang %2d nam %4d ",
sv.Ngsinh.day,sv.Ngsinh.month, sv.Ngsinh.year);
   printf ("\n Diem Lap trinh CB : %4.1f", sv.LaptrinhCB);
   printf ("\n Diem Kien truc MT : %4.1f", sv.KientrucMT);
   printf ("\n Diem Mang MT : %4.1f", sv.MangMT);
   printf ("\n Diem TB : %4.1f", sv. DiemTB);
   getch();
```

```
}

// hàm khởi tạo danh sách rỗng
void   Initialize (listnode *p)
  {
        *P=NULL;
  }
// ham tao nut moi
listnode newnode (ElementType         sv)
{       listnode       q;
        q=(listnode)calloc (1,sizeof(struct node));
        q->info=sv;
        q->link = NULL;
        return (q);
}
//ham chen truoc nut dau
void Insertfirst ( listnode *p, ElementType   x)
   {            listnode   q;
                q= newnode(x);
                if (*p==NULL)
                        *p=q;
                else
                        {       q->link=*p;
                                *p=q;
                        }
   }
// ham chen truoc nut cuoi
void InsertEnd ( listnode  *p, ElementType   x)
   {    listnode   q, m;
        q= newnode(x);
        if (*p==NULL)
```

```
            *p=q;
        else
        {       m=*p;
                while (m->link != NULL)
                        m=m->link;
                m->link=q;
        }
  }
//ham chen truoc mot nut bat ky
void InsertBefor ( listnode *p, listnode r,  ElementType  x)
  {             listnode   q, m;
                if (r == *p)
                        Insertfirst(p,x);
                else
                {       q= newnode(x);
                        m=*p;
                        while (m->link != r)
                                m=m->link;
                        q->link=r;
                        m->link=q;
                }
  }
//ham chen sau mot nut bat ky
void InsertAfter ( listnode  *p, listnode r,  ElementType  x)
  {     listnode   q;
        if (*p==NULL)
          printf("Danh sach rong");
        else
        {       q= newnode(x);
                q->link = r->link;
                r->link = q;
```

```
        }
    }
                //ham xoa mot nut r
void    DeleteNode ( listnode  *p, listnode r)
    {           listnode    m;
                if (*p == NULL)
                        printf (" DANH SACH RONG");
                else
                        if (r == *p)
                        {       *p=(*p)->link;
                                free(r);
                        }
                        else
                        {       m=*p;
                        while (m->link != r)
                                m=m->link;
                        m->link=r->link;
                        free(r);
        }
    }
//ham tim nut co gia tri truong info=x
listnode    SearchNode ( listnode  p, char x[])
    {           listnode    m;
                if (p == NULL)
                        printf (" DANH SACH RONG");
                else
                {   m=p;
                        while (m !=NULL)
                          if (stricmp(x,m->info.Hten)==0)
                                return (m);
                          else
```

```
                    m=m->link;
                }
                return (NULL);
   }

//ham tao dsnd
void CreateList(listnode  *p)
{  char kt;
   SinhVien   sv;
   do
    {
     NhapSinhVien (&sv);
     InsertEnd(p,sv);
          printf("ban nhap tiep khong c/k: ");
           fflush(stdin); kt=getchar();
    }while (kt!='k');
}
//ham duyet dsnd
void Browselist(listnode p)
{ listnode q;
  SinhVien sv;
  q=p;
  printf("Danh sach noi don:\n");
  while (q!=NULL)
   {
        sv=q->info;
        HienSinhVien(sv);
        q=q->link;
   }
  getch();
```

```
}
//ham menu
int menu()
 {  int chon;
    clrscr();
    printf("\n MOT SO THUAT TOAN VE DANH SACH NOI DON\n\n");
    printf(" Nhan so tuong ung de chon chuc nang:\n");
    printf(" 1. Khoi tao danh sach \n");
    printf(" 2. Nhap cac phan tu cho danh sach\n");
    printf(" 3. In cac phan tu cua danh sach\n");
    printf(" 4. Tim mot phan tu cua danh sach\n");
    printf(" 5. Bo sung mot phan tu vao sau mot phan tu \n");
    printf(" 6. Bo sung mot phan tu vao truoc mot phan tu\n");
    printf(" 7. Xoa mot phan tu cua mang\n");
    printf(" 0. Thoat khoi chuong trinh\n");
    printf("  Nhap so tuong ung: ");
         scanf("%d",&chon);
    return (chon);
 }
//ham chinh main
 void main()
 {listnode *p,q;
  int chon;
  char  ht[30];
  char kt;
  ElementType  sv;
  do
  { chon=menu();
          switch (chon)
    {
      case 1:
```

```
                Initialize(p);
                break;
 case 2:
                CreateList(p);
        break;
case 3:
            printf("\n Cac phan tu co trong danh sach:\n\n") ;
            Browselist(*p);
             getch();
             break;
case 4:
            printf ("\n Nhap HT can tim: ");
                    fflush(stdin); gets(ht1);
             q=SearchNode(*p,ht1);
             if (q!=NULL)
                                printf("\n Tim thay phan tu trong Danh sach");
            else
                                printf("khong tim thay phan tu trong danh
                        sach");
             getch();
             break;
case 5:
            printf ("\n Nhap SV moi: ");
                    NhapSinhVien(&sv);
            printf ("\n Nhap HT de tim vt chen sau: ");
                    fflush(stdin);gets(ht1);
            q=SearchNode(*p,ht1);
            if (q==NULL)
                                printf("\n Khong tim thay phan tu trong
                        DS\n");
            else
```

```
            {
    //q la con tro chua vi tri nut tim thay o tren
                        InsertAfter(p,q,sv)
                        printf(" Da bo sung phan tu \n");
                        getch();
             }
            break;
 case 6:
                    printf ("\n Nhap HT moi: "); NhapSinhVien(&sv);
       printf ("\n Nhap HT de tim vt chen truoc: ");
                    fflush(stdin);    gets(ht1);
            q=SearchNode(*p,ht1);
            if (q==NULL)
                        printf("\n Khong tim thay phan tu trong DS\n");
            else
            {
    //q la con tro chua vi tri nut tim thay o tren
                        InsertBefor(p,q,sv);
                        printf(" Da bo sung phan tu \n");
             }  getch();
            break;

case 7:
              printf ("\n Nhap ht Sinh vien can xoa: "); fflush(stdin);
              gets(ht1);
          q=SearchNode(*p,ht1);
          if (q==NULL)
                        printf("\n Khong tim thay phan tu can xoa\n");
          else
               {        DeleteNode(p,q);
                        printf(" Da xoa phan tu \n");
```

```
                } getch();
            break;

    default:    printf ("\n ban chon sai roi! ");
          }
   } while (chon!=0);
}//ket thuc ham main
```

2) Chương trình chuyển đổi một số hệ 10 sang hệ 2. Sử dụng các thao tác của Stack cài đặt bằng danh sách liên kết đơn để viết chương trình

```
#include <stdio.h>
#include <conio.h>
#include<alloc.h>
        //Đinh nghĩa kiểu phần tử
typedef  int  ElementType;
struct node
{       ElementType                 info;
        struct node*          link;
};
typedef struct node* Stacknode;
typedef struct
{
        Stacknode    top;
} Stack;
        //Định nghĩa hàm khởi tạo Stack rong
void   Initialize (Stack    *S)
 {
        S ->top=NULL;
 };
        //Định nghĩa hàm kiểm tra Stack có rỗng không?
int  Empty(Stack  S)
{
```

```
    return (S.top==NULL);
}
        //Định nghĩa hàm đẩy một nút vào Stack
void   GetNode ( Stack  *S, ElementType  x)
 {          Stacknode          q;
                q=( Stacknode) malloc (sizeof(struct node));
                q->info=x;
                q->link=S->top;
                S->top=q;
   }
        //Định nghĩa hàm lấy một nút khỏi Stack
void    RemoveNode( Stack *S, ElementType *x)
{       Stacknode    q;
        if  (Empty(*S))
                        printf("\n Stack rong");
        else
          {     q=S->top;
                *x=q->info;
                S->top=q->link;
                free(q);
          }
}
/*Định nghĩa hàm chuển đổi số hệ 10 sang hệ 2 và đưa từng chữ số hệ 2
vào Stack*/
void doiso(Stack *S, ElementType x)
{
        while (x!=0)
    {
                GetNode ( S, x%2);
                x=x/2;
    }
```

```
}
/*Định nghĩa hàm lấy từng chữ số hệ 2 và in ra màn hình*/
void inra (Stack *S)
{       ElementType x;
        while (!Empty(*S))
    {
                RemoveNode(S,&x);
                printf("%d",x);
    }
}
        //Định nghĩa hàm main
void main()
{
    Stack *S;
    ElementType x;
    Initialize (S);
    printf("nhap so he 10:");scanf("%d",&x);
    doiso(S,x);
    inra (S);
    getch();

 }
}
```

3) Chương trình cài đặt các giải thuật sắp xếp và tìm kiếm với danh sách sinh viên được cài đặt bằng mảng

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
const  int maxlist=100;
                //đinh nghia ban ghi SinhVien
struct SinhVien
```

```
{  char masv[10];
   char Hten [35];
   float   LaptrinhCB, KientrucMT, MangMT, DiemTB;
};
typedef        SinhVien      Item;
typedef struct list
{ Item          element[maxlist];
   int             count;
};
        // ham khoi tao danh sach rong
void   initializeList (list   *L)
{
                L->count=0;
}
        // ham kiem tra danh sach co rong khong?
int        EmptyList(list      L)
{
        return (L.count==0);
}
        // ham kiem tra danh sach co day khong?
int        FullList(list          L)
{
        return (L.count==maxlist);
}
//ham bo sung sinh vien x vao vị tri pos
void    insertElement (list *L, int pos, Item x)
{
        int i;
        if  (FullList (*L))
                printf("Danh sach day!");
    else
```

```
    if ((pos <0) || ( pos>=maxlist ))
            printf("\n Vi tri chen khong phu hop");
        else
        {
            for  (i=L->count;i>=pos;i--)
                L-> element [i+1]= L-> element [i];
            L-> element [pos]=x;
            L->count++;
        }
}
        // ham tao danh sach
void  NhapSinhVien (SinhVien *p)
  { float f; char ht[35],ma[10];
        unsigned char d; unsigned int n;
        fflush(stdin);          //ham xoa vung dem ban phim
        printf("\n ma sinh vien: ");
        gets(ma);strcpy(p->masv,ma);
        printf("\n Ho ten: ");
        fflush(stdin);gets(ht);strcpy(p->Hten,ht);
            //Nhap diem cho sinh viên
        printf("Diem Lap trinh CB: ");
        scanf("%f",&f); p->LaptrinhCB=f;
        printf("Diem Kien truc MT: ");
        scanf("%f",&f); p->KientrucMT=f;
        printf("Diem Mang MT: ");
        scanf("%f",&f); p->MangMT=f;
            //Tinh diem trung bình
        p->DiemTB=( p-> LaptrinhCB+ p-> KientrucMT+ p-> MangMT )/3;
}
//ham hien thong tin mot ban ghi sinh vien
void HienSinhVien (SinhVien sv)
```

```
  {
        printf ("\n Ma sinh vien  %10s", sv.masv);
        printf ("\n Sinh vien  %35s", sv.Hten);
        printf ("\n Diem Lap trinh CB : %4.1f", sv.LaptrinhCB);
        printf ("\n Diem Kien truc MT : %4.1f", sv.KientrucMT);
        printf ("\n Diem Mang MT : %4.1f", sv.MangMT);
        printf ("\n Diem TB : %4.1f", sv. DiemTB);
        getch();
  }
// ham tao danh sach sinh vien
void CreateList (list  *L)
{ int i=0;
  char kt;
  Item sv;
  do
   {    printf("nhap phan tu thu %d:",i+1);
        NhapSinhVien(&sv);
        L->element[i]=sv;
        L->count++;i++;
        printf("ban nhap tiep khong c/k? ");
        fflush(stdin);kt=getchar();
   }
  while (kt!='k');
}
//ham in danh sach sinh vien ra man hinh
void PrintList(list L)
{ int i;
  if (EmptyList(L))
   {  printf("Danh sach rong");
      return;
   }
```

```
  for (i=0; i<L.count;i++)
     HienSinhVien(L.element[i]);
 }
// ham InsertionSort sap xep danh sach theo ho ten
void   InsertionSort (list *L)
{ int   i, j;
  Item  temp;
        for  (i=1 ; i<L->count; i++)
          {     temp=L->element[i];
                j = i-1;
                while  ((j>=0) &&
                (stricmp(temp.Hten,L->element[j].Hten)>0))
                  {
                        L->element[j+1] = L->element[j];
                        j--;
                  }
                L->element[j+1]=temp ;
          }
}
//ham SelectionSort sap xep theo ho ten
void    SelectionSort (list *L)
{ int   i, j, m;
        Item  temp;
        for  (i=0 ; i<L->count-1; i++)
          {     m = i ;
                for  (j=i+1 ; j<L->count; j++)
                  if  (stricmp(L->element[i].Hten,L->element[j].Hten)>0)
                                m=j;
                 temp=L->element[i];
                 L->element[i]=L->element[m];
                 L->element[m]=temp ;
```

```
            }
}
// ham InterchangeSort  sap xep theo ho ten
void InterchangeSort(list  *L)
{       int i, j;
        SinhVien  temp;
        for (  i=0; i<L->count-1;i++)
                for (  j=i+1;j< L->count ;j++)
                        if (stricmp(L->element[i].Hten,L->element[j].Hten)>0)
                {   temp=L->element[i];
                                L->element[i]= L->element[j];
                                 L->element[j]=temp ;
                 }
}
//ham  BubleSort  sap xep theo ho ten
void    BubleSort (list   *L)
{       int   i, j;
        Item  temp;
        for  (i=0 ; i<L->count-1; i++)
          for  (j=L->count-1 ; j<=i+1; j--)
                  if   (stricmp(L->element[i].Hten,L->element[j].Hten)>0)
                       {    temp=L->element[j];
                                L->element[j]=L->element[j-1];
                                L->element[j-1]=temp ;
                       }
}
// ham QuickSort sap xep theo ho ten
void QuickSort(list *L, int   left , int   right)
{       int      i, j;
        Item  key,temp;
        key= L->element[left];
```

```
        i = left; j = right;
        do
        {       while ((stricmp(L->element[i].Hten, key.Hten)<0) && (i <=
                                                                    right))
                        i++;
                while ((stricmp(L->element[j].Hten, key.Hten)>0) && (j >=left))
                        j--;
                if(i <=j)
                        {
                         temp=L->element[i];
                         L->element[i]=L->element[j];
                         L->element[j]=temp;
                         i++ ; j--;
                        }
        } while(i <= j);
    if(left<j)
                QuickSort(L, left, j);
    if(i<right)
                QuickSort(L, i, right);
}
//ham  QuickSort sap xep theo masv
void QuickSortMSV(list *L, int   left , int   right)
{       int     i, j;
        Item  key,temp;
        key= L->element[left];
        i = left; j = right;
        do
        {
        while ((stricmp(L->element[i].masv,key.masv)<0) && (i <= right))
                i++;
        while ((stricmp(L->element[j].masv,key.masv)>0) && (j >=left))
```

```
                j--;
        if(i <=j)
         {
                temp=L->element[i];
                L->element[i]=L->element[j];
                L->element[j]=temp;
                i++ ; j--;
          }
    } while(i <= j);
    if(left<j)
                QuickSort(L, left, j);
    if(i<right)
                QuickSort(L, i, right);
}
// ham Sequen-Search  tim kiem theo masv
int  Sequen_Search(list L, char   X[])
{    int  i=0;
    for (i=0; i<L.count; i++)
        if (stricmp(X,L.element[i].masv)==0)
           return (i);
    return (-1);
}
/* ham Sequen_Search theo cach 2
int  Sequen_Search1(list L, char   X[])
{    int  i;
    strcpy(L.element[L.count].masv,X);
    while (stricmp(X,L.element[i].masv)!=0)
         i++;
     if (i<L.count)
           return (i);
     else
```

```
            return (-1);
}*/
//ham BinarySearch  tim kiem theo masv
int  BinarySearch (list L, char X[])
{   int   mid, left=0, right=L.count-1;
    do
    {
      mid=(left+right)/2;
      if (stricmp(X,L.element[mid].masv)==0)
                        return (mid);
      else
                if (stricmp(X,L.element[mid].masv)<0)
                        right=mid-1;
                else
                        left=mid+1;
    } while(left<=right);
                return -1;
}
        //ham menu
int menu()
 {  int chon;
    clrscr();
    printf("\n MOT SO GIAI TOAN VE SAP XEP VA TIM KIEM\n\n");
    printf(" Nhan so tuong ung de chon chuc nang:\n");
    printf(" 1. Khoi tao danh sach \n");
    printf(" 2. Nhap cac phan tu cho danh sach\n");
    printf(" 3. In cac phan tu cua danh sach\n");
    printf(" 4. ham InsertionSort  sap xep theo ho ten\n");
    printf(" 5. ham SelectionSort  sap xep theo ho ten\n");
    printf(" 6. ham InterchangeSort  sap xep theo ho ten\n");
    printf(" 7.ham  BubleSort  sap xep theo ho ten  \n");
```

```
  printf(" 8.ham QuickSort sap xep theo ho ten \n");
  printf(" 9 ham Sequen-Search  timf kiem theo masv\n");
  printf(" 10 ham BinarySearch  tim kiem theo masv \n");
  printf(" 0. Thoat khoi chuong trinh\n");
  printf("  Nhap so tuong ung: "); scanf("%d",&chon);
  return (chon);
}
      //ham chinh main
void main()
{list *L,*M;
int chon,pos, n;
char  ma[10];
do
{ chon=menu();
  switch (chon)
  {
   case 1:
            initializeList(L);
            break;
   case 2:
             CreateList(L);M=L;
             break;
   case 3:
             printf("\n Cac phan tu co trong danh sach:\n\n") ;
             PrintList(*L);
             getch();
             break;
   case 4:
             L=M;
             InsertionSort (L);
                  printf(" Da sap xep theo ho ten\n");
```

```
            getch();
        break;
case 5:
            L=M;
            SelectionSort (L);
            printf(" Da sap xep theo ho ten\n");
            getch();
            break;
case 6:
        L=M;
        InterchangeSort (L);
        printf(" Da sap xep theo ho ten\n");
            getch();
        break;
case 7:
        L=M;
        BubleSort (L);
        printf(" Da sap xep theo ho ten\n");
            getch();
        break;
case 8:
        L=M;
        QuickSort (L,0,L->count-1);
        printf(" Da sap xep theo ho ten\n");
            getch();
        break;
case 9:
        L=M;
        printf(" Nhap ma sinh vien can tim: ");
        fflush(stdin); gets(ma);
        pos= Sequen_Search(*L,ma);
```

```
            if (pos==-1)
                        printf("khong tim thay masv %s\n", ma);
            else
                        printf(" Da tim thay masv %s tai vi tri
                                                                %d\n",ma,pos);
                 getch();
            break;
    case 10:
            QuickSortMSV(L,0,L->count-1 );
            printf(" Nhap ma sinh vien can tim: ");
            fflush(stdin); gets(ma);
            pos= BinarySearch (*L,ma);
            if (pos==-1)
                        printf("khong tim thay masv %s\n", ma);
            else
                        printf(" Da tim thay masv %s tai vi tri %d\n",ma,pos);
                getch();
           break;
    default:    printf ("\n ban chon sai roi! ");
        }
  } while (chon!=0);
}
```

TÀI LIỆU THAM KHẢO

- Giáo trình cấu trúc dữ liệu và giải thuật- Đỗ Xuân Lôi - Nhà xuất bản Giáo dục, 1993
- Giáo trình cấu trúc dữ liệu- Trần Hạnh Nhi- Trường đại học Khoa hoc tự nhiên, tp. Hồ Chí Minh, 2003
- Cấu Trúc Dữ Liệu Và Thuật Toán - **PGS. TS. Hoàng Nghĩa Tý** - **Xây Dựng**, 2002
- Bài Tập Nâng Cao Cấu Trúc Dữ Liệu Cài Đặt Bằng C - **Gia Việt**(Biên dịch), **ESAKOV.J** , **WEISS T**,- Nhà xuất bản **Thống kê**
- 455 Bài Tập Cấu Trúc Dữ Liệu - Ứng Dụng Và Cài Đặt Bằng C++ - Minh Trung (Biên dịch), TS. Khuất Hữu Thanh(Biên dịch), **Chu Trọng Lương**(Tác giả) - **Thống kê** .
- Cẩm Nang Thuật Toán (Tập1,2) - **Robert Sedgewick**, **Trần Đan Thư** (Biên dịch), **Bùi thị Ngọc Nga** (Biên dịch) - **Khoa học và kỹ thuật**
- Giải Một Bài Toán Trên Máy Tính Như Thế Nào - **GS. TSKH. H**oàng Kiếm - **Giáo dục**, 2005.