

ĐẠI HỌC KINH DOANH VÀ CÔNG NGHỆ HÀ NỘI
KHOA CÔNG NGHỆ THÔNG TIN



GIÁO TRÌNH

CẤU TRÚC DỮ LIỆU VÀ

GIẢI THUẬT

Chủ biên : TS. Hoàng Xuân Thảo

Biên soạn: GS. Trần Anh Bảo

ThS. Hoàng Thị Kim Oanh

(Dùng cho chương trình đào tạo hệ đại học)

Lưu hành nội bộ

HÀ NỘI - 2015

MỤC LỤC

LỜI NÓI ĐẦU	7
Chương 1. Một số vấn đề cơ bản của cấu trúc dữ liệu và giải thuật	8
1.1. Thuật toán và giải thuật.....	8
1.1.1 Thuật toán và các đặc trưng của thuật toán	8
1.1.2. Thuật toán và Giải thuật	8
1.1.3. Phương pháp mô tả Thuật toán.....	15
1.1.4. Độ phức tạp của thuật toán	17
1.2 Các cấu trúc dữ liệu cơ bản	20
1.2.1 Dữ liệu kiểu số	20
1.2.2. Dữ liệu kiểu kí tự	21
1.3. Một số giải thuật cơ bản trên các kiểu dữ liệu cơ bản	22
1.3.1. Một số thuật toán cơ bản trên dữ liệu kiểu số	22
1.3.2. Một số thuật toán cơ bản trên dữ liệu kiểu ký tự.....	24
1.4. CASE STUDY	26
1.4.1. Cài đặt các thuật toán trên dữ liệu kiểu số.	26
1.4.2. Cài đặt các thuật toán trên dữ liệu kiểu ký tự.....	27
Chương 2. Duyệt và đệ qui	28
2.1. Thuật toán duyệt	28
2.1.1. Phương pháp duyệt toàn bộ	28
2.1.2. Phương pháp Heuristic	28
2.1.3. Một số bài toán sử dụng thuật toán duyệt	29
2.2. Thuật toán đệ qui	29
2.2.1. Giới thiệu về thuật toán đệ qui	29
2.2.2. Một số bài toán sử dụng thuật toán đệ qui	31
2.3. Qui hoạch động	32
2.3.1. Giới thiệu phương pháp qui hoạch động.....	33
2.3.2. Một số bài toán sử dụng phương pháp qui hoạch động.....	33
2.4. CASE STUDY	33
2.4.1. Xây dựng và cài đặt thuật toán duyệt cho một số bài toán thực tế.....	33
2.4.2. Xây dựng và cài đặt thuật toán đệ qui cho một số bài toán thực tế.....	40
2.4.3. Xây dựng , cài đặt thuật toán qui hoạch động cho một số bài toán thực tế.	43
Chương 3. Ngăn xếp, Hàng đợi, Danh sách liên kết	49

3.1. Ngăn xếp.....	49
3.1.1. Định nghĩa ngăn xếp	49
3.1.2. Biểu diễn ngăn xếp.....	51
3.1.3. Các thao tác trên ngăn xếp	51
3.1.4. Ứng dụng của ngăn xếp.....	53
3.2. Hàng đợi (QUEUE).....	63
3.3.1. Định nghĩa hàng đợi.....	63
3.3.2. Biểu diễn hàng đợi	64
3.3.3. Các thao tác trên hàng đợi.....	66
3.3.4. Ứng dụng của hàng đợi	69
3.3. Danh sách liên kết đơn	69
3.3.1. Định nghĩa danh sách liên kết đơn.....	69
3.3.2. Biểu diễn danh sách liên kết đơn.....	69
Cài đặt danh sách.....	69
Khởi tạo danh sách rỗng	70
3.3.3. Các thao tác trên danh sách liên kết đơn.....	70
Kiểm tra danh sách rỗng hay không	70
Tính độ dài danh sách	70
Tạo 1 Node trong danh sách	70
Chèn Node P vào vị trí đầu tiên.....	71
Chèn Node P vào vị trí k trong danh sách.....	71
Tìm phần tử có giá trị x trong danh sách.....	72
Xóa phần tử ở vị trí đầu tiên	72
Xóa phần tử ở vị trí k.....	72
Xóa phần tử có giá trị x.....	73
3.3.4. Ứng dụng của danh sách liên kết đơn	73
3.4. Danh sách liên kết kép	73
3.4.1. Định nghĩa danh sách liên kết kép	73
3.4.2. Biểu diễn danh sách liên kết kép	74
3.4.3. Các thao tác trên danh sách liên kết kép	74
3.4.4. Ứng dụng của danh sách liên kết kép	79
3.5. CASE STUDY	79
3.5.1. Cài đặt chương trình máy tính các thuật toán trên ngăn xếp.	79
3.5.2. Cài đặt chương trình máy tính các thuật toán trên hàng đợi.	81

3.5.3. Cài đặt chương trình máy tính các thuật toán trên danh sách liên kết.....	83
3.5.4. Tìm hiểu các thao tác trên ngăn xếp, hàng đợi và danh sách liên kết trong thư viện của ngôn ngữ lập trình C ⁺⁺	83
Chương 4. Cây nhị phân	84
4.1. Định nghĩa và phân loại cây nhị phân.....	84
4.1.1. Định nghĩa cây nhị phân.....	84
4.1.2. Phân loại các cây nhị phân	84
4.2. Biểu diễn cây nhị phân	85
4.2.1. Biểu diễn cây nhị phân bằng mảng.....	85
4.2.2. Biểu diễn cây nhị phân bằng danh sách liên kết	86
4.3. Thao tác trên cây nhị phân	86
4.3.1. Thao tác trên cây nhị phân tổng quát	86
4.3.2. Thao tác trên cây nhị phân tìm kiếm.....	87
4.3.4. Thao tác trên cây nhị phân cân bằng.....	90
4.3.3. Thao tác trên cây nhị phân đầy đủ	90
4.4. CASE STUDY	90
4.4.1. Cài đặt chương trình máy tính các thuật toán trên cây nhị phân tổng quát.	90
4.4.2. Cài đặt chương trình máy tính các thuật toán trên cây nhị phân tìm kiếm.	93
4.4.3. Cài đặt chương trình máy tính các thuật toán trên cây nhị phân cân bằng.	94
4.4.4. Tìm hiểu các thao tác trên cây nhị phân trong thư viện của ngôn ngữ lập trình C ⁺⁺	97
Chương 5. Đồ thị	98
5.1. Khái niệm và định nghĩa	98
5.2. Biểu diễn đồ thị.....	98
5.3. Các thuật toán tìm kiếm trên đồ thị	99
5.4. Một số bài toán tối ưu trên đồ thị	99
5.5. CASE STUDY	104
5.5.1. Cài đặt chương trình máy tính các thuật toán tìm kiếm trên đồ thị.....	104
Cài đặt bằng ngôn ngữ C ⁺⁺	104
5.5.2. Cài đặt chương trình máy tính các thuật toán tối ưu trên đồ thị.	106
5.5.3. Tìm hiểu thao tác trên đồ thị trong thư viện của ngôn ngữ lập trình C ⁺⁺	106
Chương 6. Sắp xếp và tìm kiếm	107
6.1. Các thuật toán sắp xếp đơn giản	107
6.1.1. Sắp xếp kiểu đổi chỗ	107
6.1.2. Sắp xếp kiểu chèn trực tiếp	110

6.1.3. Sắp xếp kiểu sủi bọt	113
6.2. Thuật toán sắp xếp nhanh	116
6.2.1. Giới thiệu thuật toán.....	116
6.2.2. Mô tả thuật toán	117
6.2.3. Kiểm nghiệm thuật toán	119
6.3.2. Mô tả thuật toán	121
6.3.3. Kiểm nghiệm thuật toán	123
6.4. Thuật toán sắp xếp kiểu hòa nhập.....	130
6.4.1. Giới thiệu thuật toán.....	130
6.4.2. Mô tả thuật toán	130
6.4.3. Kiểm nghiệm thuật toán	130
6.5. Một số thuật toán tìm kiếm.....	136
6.5.1. Tìm kiếm tuyến tính	136
6.5.2. Tìm kiếm nhị phân	136
6.5.3. Tìm kiếm theo cơ số.....	138
6.6. CASE STUDY	138
6.6.1. Cài đặt chương trình máy tính các thuật toán sắp xếp.	138
6.6.2. Cài đặt chương trình máy tính các thuật toán tìm kiếm.....	140
6.6.3. Tìm hiểu các thuật toán sắp xếp và tìm kiếm trong thư viện của ngôn ngữ lập trình C ⁺⁺	141
TÀI LIỆU THAM KHẢO	142

LỜI NÓI ĐẦU

Để đáp ứng nhu cầu học tập của các bạn sinh viên, nhất là sinh viên chuyên ngành tin học, chúng tôi đã tiến hành biên soạn các giáo trình, bài giảng chính trong chương trình học. Giáo trình được biên soạn theo đề cương chi tiết môn Cấu Trúc Dữ Liệu của sinh viên chuyên ngành tin học của Khoa Công Nghệ Thông Tin Trường Đại Học Kinh Doanh và Công Nghệ Hà nội. Mục tiêu của nó nhằm giúp các bạn sinh viên chuyên ngành có một tài liệu cô đọng dùng làm tài liệu học tập. Chúng tôi nghĩ rằng các bạn sinh viên không chuyên tin và những người quan tâm tới cấu trúc dữ liệu và giải thuật cũng sẽ tìm được trong này những điều hữu ích

Giáo trình bao gồm 6 chương, trình bày về các cấu trúc dữ liệu và các giải thuật cơ bản nhất trong tin học.

Chương 1 trình bày về phân tích và thiết kế thuật toán. Đầu tiên là cách phân tích 1 vấn đề, từ thực tiễn cho tới chương trình, cách thiết kế một giải pháp cho vấn đề theo cách giải quyết bằng máy tính. Tiếp theo, các phương pháp phân tích, đánh giá độ phức tạp và thời gian thực hiện giải thuật cũng được xem xét trong chương. Chương 2 trình bày về đệ qui, một khái niệm rất cơ bản trong toán học và khoa học máy tính. Việc sử dụng đệ qui có thể xây dựng được những chương trình giải quyết được các vấn đề rất phức tạp chỉ bằng một số ít câu lệnh, đặc biệt là các vấn đề mang bản chất đệ qui.

Chương 3, 4, 5 trình bày về các cấu trúc dữ liệu được sử dụng rất thông dụng như mảng và danh sách liên kết, ngăn xếp và hàng đợi, cây, đồ thị. Đó là các cấu trúc dữ liệu cũng rất gần gũi với các cấu trúc trong thực tiễn.

Chương 6 trình bày về các thuật toán sắp xếp và tìm kiếm. Các thuật toán này cùng với các kỹ thuật được sử dụng trong đó được coi là các kỹ thuật cơ sở cho lập trình máy tính. Các thuật toán được xem xét bao gồm các lớp thuật toán đơn giản và cả các thuật toán cài đặt phức tạp nhưng có thời gian thực hiện tối ưu.

Về nguyên tắc, các cấu trúc dữ liệu và các giải thuật có thể được biểu diễn và cài đặt bằng bất cứ ngôn ngữ lập trình hiện đại nào. Tuy nhiên, để có được các phân tích sâu sắc hơn và có kết quả thực tế hơn, tác giả đã sử dụng ngôn ngữ lập trình C để minh họa cho các cấu trúc dữ liệu và thuật toán. Do vậy, ngoài các kiến thức cơ bản về tin học, người đọc cần có kiến thức về ngôn ngữ lập trình C.

Mặc dù đã rất cố gắng nhiều trong quá trình biên soạn giáo trình nhưng chắc chắn giáo trình sẽ còn nhiều thiếu sót và hạn chế. Rất mong nhận được sự đóng góp ý kiến quý báu của sinh viên và các bạn đọc để giáo trình ngày một hoàn thiện hơn.

Chương 1. Một số vấn đề cơ bản của cấu trúc dữ liệu và giải thuật

Chương 1 trình bày các khái niệm về giải thuật và phương pháp tinh chỉnh từng bước chương trình được thể hiện qua ngôn ngữ diễn đạt giải thuật. Chương này cũng nêu phương pháp phân tích và đánh giá một thuật toán, các khái niệm liên quan đến việc tính toán thời gian thực hiện chương trình.

Trong mỗi phần đều có các minh họa cụ thể. Phần đầu đưa ra ví dụ về bài toán nút giao thông và phương pháp giải quyết bài toán từ phân tích vấn đề cho đến thiết kế giải thuật, tinh chỉnh từng bước cho tới mức cụ thể hơn. Phần 2 đưa ra một ví dụ về phân tích và tính toán thời gian thực hiện giải thuật sắp xếp nổi bọt.

Để học tốt chương này, sinh viên cần nắm vững phần lý thuyết và tìm các ví dụ tương tự để thực hành phân tích, thiết kế, và đánh giá giải thuật.

1.1. Thuật toán và giải thuật

1.1.1 Thuật toán và các đặc trưng của thuật toán

Với mỗi vấn đề cần giải quyết, ta có thể tìm ra nhiều thuật toán khác nhau. Có những thuật toán thiết kế đơn giản, dễ hiểu, dễ lập trình và sửa lỗi, tuy nhiên thời gian thực hiện lớn và tiêu tốn nhiều tài nguyên máy tính.

Ngược lại, có những thuật toán thiết kế và lập trình rất phức tạp, nhưng cho thời gian chạy nhanh hơn, sử dụng tài nguyên máy tính hiệu quả hơn. Khi đó, câu hỏi đặt ra là ta nên lựa chọn giải thuật nào để thực hiện?

Đối với những chương trình chỉ được thực hiện một vài lần thì thời gian chạy không phải là tiêu chí quan trọng nhất. Đối với bài toán kiểu này, thời gian để lập trình viên xây dựng và hoàn thiện thuật toán đáng giá hơn thời gian chạy của chương trình và như vậy những giải thuật đơn giản về mặt thiết kế và xây dựng nên được lựa chọn.

Đối với những chương trình được thực hiện nhiều lần thì thời gian chạy của chương trình đáng giá hơn rất nhiều so với thời gian được sử dụng để thiết kế và xây dựng nó. Khi đó, lựa chọn một giải thuật có thời gian chạy nhanh hơn (cho dù việc thiết kế và xây dựng phức tạp hơn) là một lựa chọn cần thiết. Trong thực tế, trong giai đoạn đầu của việc giải quyết vấn đề, một giải thuật đơn giản thường được thực hiện trước như là 1 nguyên mẫu (prototype), sau đó nó sẽ được phân tích, đánh giá, và cải tiến thành các phiên bản tốt hơn.

1.1.2. Thuật toán và Giải thuật

Trong thực tế, khi gặp phải một vấn đề cần phải giải quyết, ta cần phải đưa ra 1

phương pháp để giải quyết vấn đề đó. Khi muốn giải quyết vấn đề bằng cách sử dụng máy tính, ta cần phải đưa ra 1 giải pháp phù hợp với việc thực thi bằng các chương trình máy tính. Thuật ngữ “thuật toán” được dùng để chỉ các giải pháp như vậy.

Thuật toán có thể được định nghĩa như sau:

Thuật toán là một chuỗi hữu hạn các lệnh. Mỗi lệnh có một ngữ nghĩa rõ ràng và có thể được thực hiện với một lượng hữu hạn tài nguyên trong một khoảng hữu hạn thời gian.

Chẳng hạn lệnh $x = y + z$ là một lệnh có các tính chất trên.

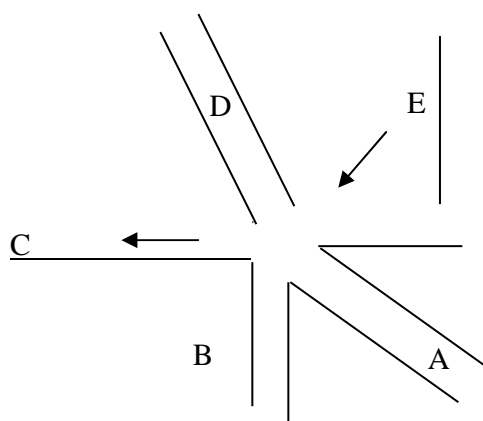
Trong một thuật toán, một lệnh có thể lặp đi lặp lại nhiều lần, tuy nhiên đối với bất kỳ bộ dữ liệu đầu vào nào, thuật toán phải kết thúc sau khi thực thi một số hữu hạn lệnh.

Như đã nói ở trên, mỗi lệnh trong thuật toán phải có ngữ nghĩa rõ ràng và có thể được thực thi trong một khoảng thời gian hữu hạn. Tuy nhiên, đôi khi một lệnh có ngữ nghĩa rõ ràng đối với người này nhưng lại không rõ ràng đối với người khác. Ngoài ra, thường rất khó để chứng minh một lệnh có thể được thực hiện trong 1 khoảng hữu hạn thời gian. Thậm chí, kể cả khi biết rõ ngữ nghĩa của các lệnh, cũng khó để có thể chứng minh là với bất kỳ bộ dữ liệu đầu vào nào, thuật toán sẽ dừng.

Tiếp theo, chúng ta sẽ xem xét một ví dụ về xây dựng thuật toán cho bài toán đèn giao thông:

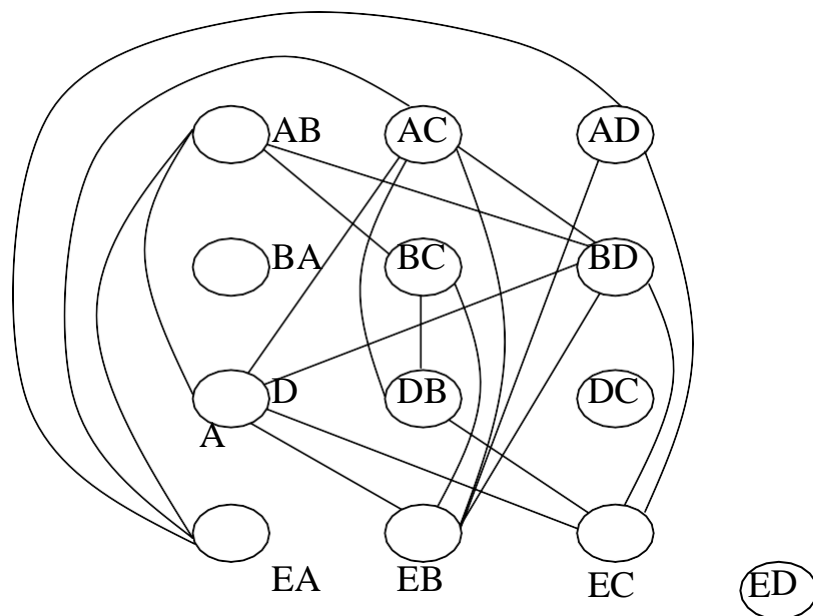
Giả sử người ta cần thiết kế một hệ thống đèn cho một nút giao thông có nhiều đường giao nhau phức tạp. Để xây dựng tập các trạng thái của các đèn giao thông, ta cần phải xây dựng một chương trình có đầu vào là tập các ngã rẽ được phép tại nút giao thông (lối đi thẳng cũng được xem như là 1 ngã rẽ) và chia tập này thành 1 số ít nhất các nhóm, sao cho tất cả các ngã rẽ trong nhóm có thể được đi cùng lúc mà không xảy ra tranh chấp. Sau đó, chúng ta sẽ gán trạng thái của các đèn giao thông với mỗi nhóm vừa được phân chia. Với cách phân chia có số nhóm ít nhất, ta sẽ xây dựng được 1 hệ thống đèn giao thông có ít trạng thái nhất.

Chẳng hạn, ta xem xét bài toán trên với nút giao thông được cho như trong hình 1.1 ở dưới. Trong nút giao thông trên, C và E là các đường 1 chiều, các đường còn lại là 2 chiều. Có tất cả 13 ngã rẽ tại nút giao thông này. Một số ngã rẽ có thể được đi đồng thời, chẳng hạn các ngã rẽ AB (từ A rẽ sang B) và EC. Một số ngã rẽ thì không được đi đồng thời (gây ra các tuyến giao thông xung đột nhau), chẳng hạn AD và EB. Hệ thống đèn tại nút giao thông phải hoạt động sao cho các ngã rẽ xung đột (chẳng hạn AD và EB) không được cho phép đi tại cùng một thời điểm, trong khi các ngã rẽ không xung đột thì có thể được đi tại cùng 1 thời điểm.



Hình 1.1 Nút giao thông

Chúng ta có thể mô hình hóa vấn đề này bằng một cấu trúc toán học gọi là đồ thị (sẽ được trình bày chi tiết ở chương 5). Đồ thị là một cấu trúc bao gồm 1 tập các điểm gọi là đỉnh và một tập các đường nối các điểm, gọi là các cạnh. Vấn đề nút giao thông có thể được mô hình hóa bằng một đồ thị, trong đó các ngã rẽ là các đỉnh, và có một cạnh nối 2 đỉnh biểu thị rằng 2 ngã rẽ đó không thể đi đồng thời. Khi đó, đồ thị của nút giao thông ở hình 1.1 có thể được biểu diễn như ở hình 1.2.



Hình 1.2 Đồ thị ngã rẽ

Ngoài cách biểu diễn trên, đồ thị còn có thể được biểu diễn thông qua 1 bảng, trong đó phần tử ở hàng i , cột j có giá trị 1 khi và chỉ khi có 1 cạnh nối đỉnh i và đỉnh j .

	AC	AD	BA	BC	BD	DA	DB	DC	EA	EB	EC	ED
AC				1	1	1			1			
AD					1	1	1		1	1		
BA												
BC	1						1			1		
BD	1	1				1				1	1	
DA	1	1			1					1	1	
DB		1		1							1	
DC												
EA	1	1	1									
EB		1	1	1	1	1						
EC			1		1	1	1					
ED												

Bảng 1.1 Biểu diễn đồ thị ngã rẽ bằng bảng

Ta có thể sử dụng đồ thị trên để giải quyết vấn đề thiết kế hệ thống đèn cho nút giao thông như đã nói.

Việc tô màu một đồ thị là việc gán cho mỗi đỉnh của đồ thị một màu sao cho không có hai đỉnh được nối bởi 1 cạnh nào đó lại có cùng một màu. Dễ thấy rằng vấn đề nút giao thông có thể được chuyển thành bài toán tô màu đồ thị các ngã rẽ ở trên sao cho phải sử dụng số màu ít nhất.

Bài toán tô màu đồ thị là bài toán đã xuất hiện và được nghiên cứu từ rất lâu. Tuy nhiên, để tô màu một đồ thị bất kỳ với số màu ít nhất là bài toán rất phức tạp. Để giải bài toán này, người ta thường sử dụng phương pháp “vét cạn” để thử tất cả các khả năng có thể. Có nghĩa, đầu tiên thử tiến hành tô màu đồ thị bằng 1 màu, tiếp theo dùng 2 màu, 3 màu, v.v. cho tới khi tìm ra phương pháp tô màu thỏa mãn yêu cầu.

Phương pháp vét cạn có vẻ thích hợp với các đồ thị nhỏ, tuy nhiên đối với các đồ thị phức tạp thì sẽ tiêu tốn rất nhiều thời gian thực hiện cũng như tài nguyên hệ thống. Ta có thể tiếp cận vấn đề theo hướng cố gắng tìm ra một giải pháp đủ tốt, không nhất thiết phải là giải pháp tối ưu. Chẳng hạn, ta sẽ cố gắng tìm một giải pháp tô màu cho đồ thị ngã rẽ ở trên với một số màu khá ít, gần với số màu ít nhất, và thời gian thực hiện việc tìm giải pháp là khá nhanh. Giải thuật tìm các giải pháp đủ tốt nhưng chưa phải tối ưu như vậy gọi là các giải thuật tìm theo “cảm tính”.

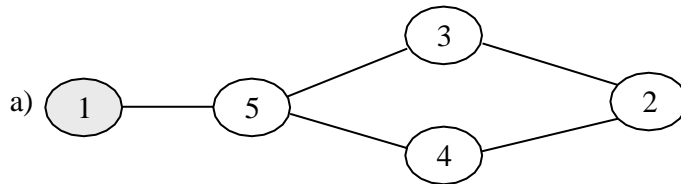
Đối với bài toán tô màu đồ thị, một thuật toán cảm tính thường được sử dụng là thuật toán “tham ăn” (greedy). Theo thuật toán này, đầu tiên ta sử dụng một màu để tô nhiều nhất số đỉnh có thể, thỏa mãn yêu cầu bài toán. Tiếp theo, sử dụng màu thứ 2 để tô các đỉnh chưa được tô trong bước 1, rồi sử dụng đến màu thứ 3 để tô các đỉnh chưa được tô trong bước 2, v.v.

Để tô màu các đỉnh với màu mới, chúng ta thực hiện các bước:

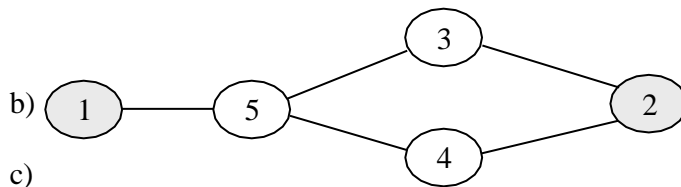
- Lựa chọn 1 đỉnh chưa được tô màu và tô nó bằng màu mới.
- Duyệt qua các đỉnh chưa được tô màu. Với mỗi đỉnh dạng này, kiểm tra xem có cạnh nào nối nó với một đỉnh vừa được tô bởi màu mới hay không. Nếu không có cạnh nào thì ta tô màu đỉnh này bằng màu mới.

Thuật toán này được gọi là “tham ăn” vì tại mỗi bước nó tô màu tất cả các đỉnh có thể mà không cần phải xem xét xem việc tô màu đó có để lại những điểm bất lợi cho các bước sau hay không. Trong nhiều trường hợp, chúng ta có thể tô màu được nhiều đỉnh hơn bằng 1 màu nếu chúng ta bớt “tham ăn” và bỏ qua một số đỉnh có thể tô màu được trong bước trước.

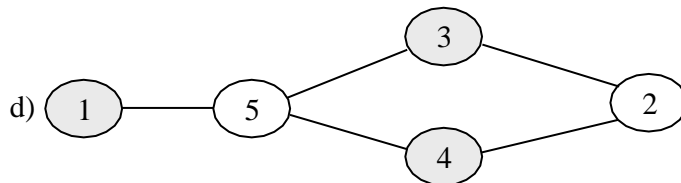
Ví dụ, xem xét đồ thị ở hình 1.3, trong đó đỉnh 1 đã được tô màu đỏ. Ta thấy rằng hoàn toàn có thể tô cả 2 đỉnh 3 và 4 là màu đỏ, với điều kiện ta không tô đỉnh số 2 màu đỏ. Tuy nhiên, nếu ta áp dụng thuật toán tham ăn theo thứ tự các đỉnh lớn dần thì đỉnh 1 và đỉnh 2 sẽ là màu đỏ, và khi đó đỉnh 3, 4 sẽ không được tô màu đỏ.



Đồ thị ban đầu



Tô màu theo thuật toán
tham ăn

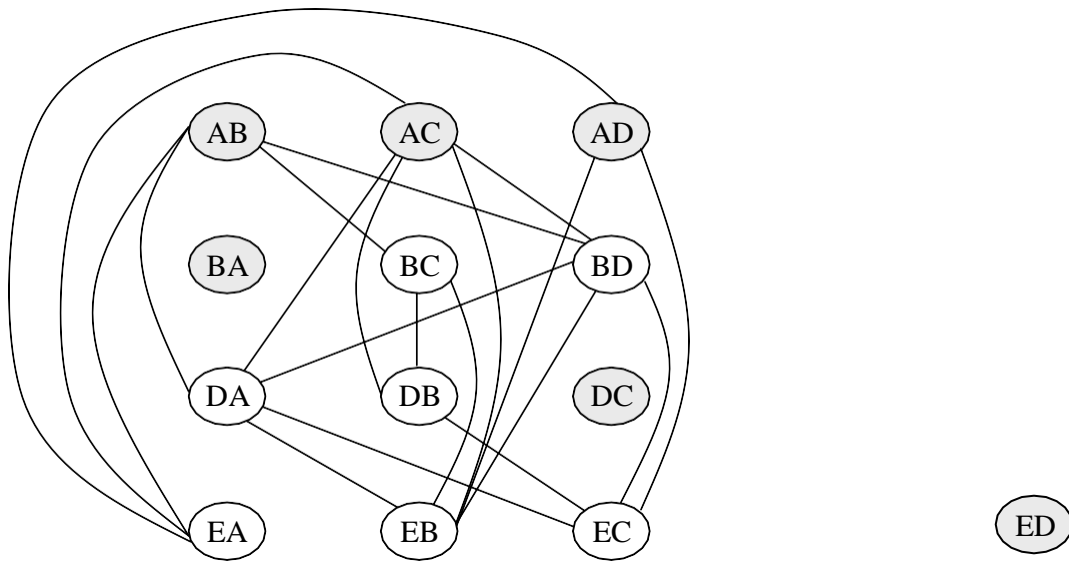


Một cách tô màu tốt hơn

Hình 1.3 Đồ thị

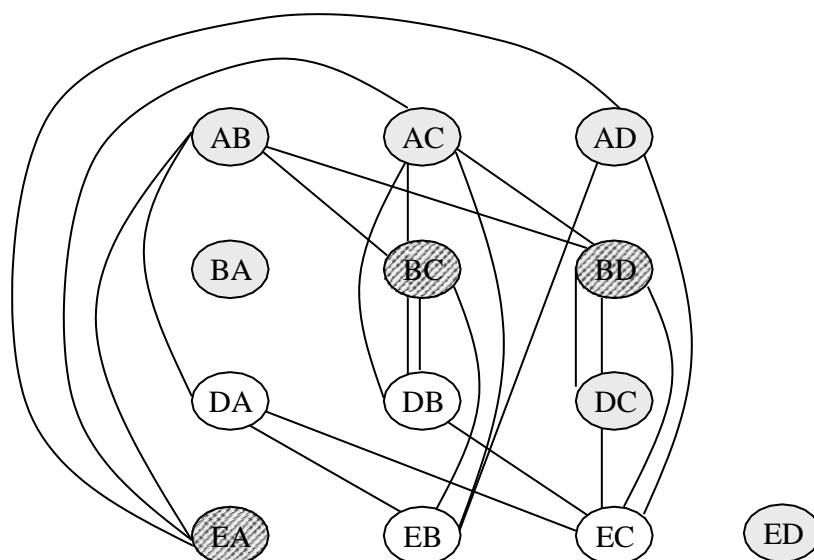
Bây giờ ta sẽ xem xét thuật toán tham ăn được áp dụng trên đồ thị các ngã rẽ ở hình 1.2 như thế nào. Giả sử ta bắt đầu từ đỉnh AB và tô cho đỉnh này màu xanh. Khi đó, ta có thể tô cho đỉnh AC màu xanh vì không có cạnh nối đỉnh này với AB. AD cũng có thể tô màu xanh vì không có cạnh nối AD với AB, AC. Đỉnh BA không có cạnh nối tới AB, AC, AD nên cũng có thể được tô màu xanh. Tuy nhiên, đỉnh BC không tô được màu xanh vì tồn tại cạnh nối BC và AB. Tương tự như vậy, BD, DA, DB không thể tô màu xanh vì tồn tại cạnh nối chúng tới một trong các đỉnh đã tô màu xanh. Cạnh DC thì có thể tô màu xanh. Cuối cùng, cạnh EA, EB, EC cũng

không thể tô màu xanh trong khi ED có thể được tô màu xanh.



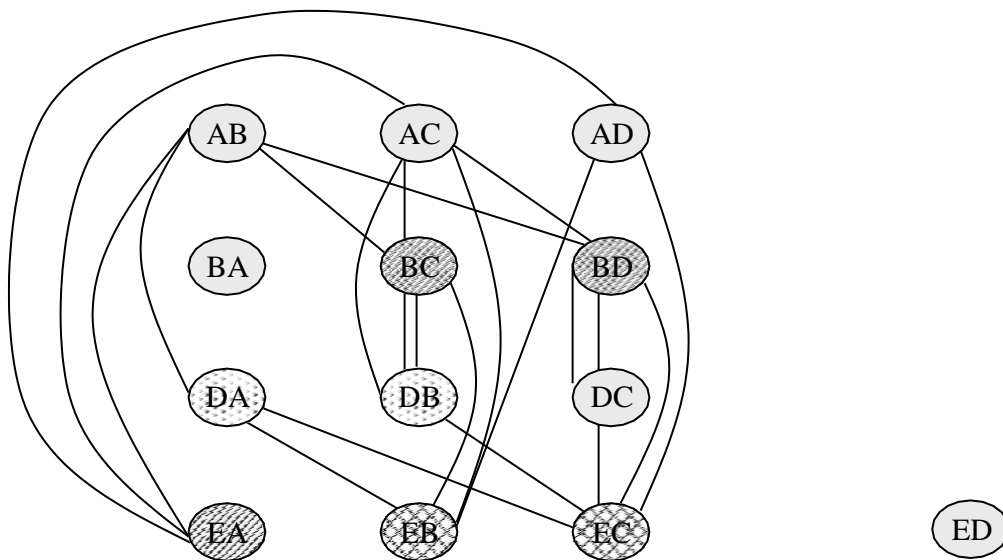
Hình 1.4 Tô màu xanh cho các đỉnh của đồ thị ngã rẽ

Tiếp theo, ta sử dụng màu đỏ để tô các đỉnh chưa được tô màu ở bước trước. Đầu tiên là BC. BD cũng có thể tô màu đỏ, tuy nhiên do tồn tại cạnh nối DA với BD nên DA không được tô màu đỏ. Tương tự như vậy, DB không tô được màu đỏ còn EA có thể tô màu đỏ. Các đỉnh chưa được tô màu còn lại đều có cạnh nối tới các đỉnh đã tô màu đỏ nên cũng không được tô màu.



Hình 1.5 Tô màu đỏ trong bước 2

Bước 3, các đỉnh chưa được tô màu còn lại là DA, DB, EB, EC. Nếu ta tô màu đỉnh DA là màu lục thì DB cũng có thể tô màu lục. Khi đó, EB, EC không thể tô màu lục và ta chọn 1 màu thứ tư là màu vàng cho 2 đỉnh này.



Hình 1.6 Tô màu lục và màu vàng cho các đỉnh còn lại

Như vậy, ta có thể dùng 4 màu xanh, đỏ, lục, vàng để tô màu cho đồ thị ngã rẽ ở hình 1.2 theo yêu cầu như đã nói ở trên. Bảng tổng hợp màu được mô tả như sau:

Màu	Ngã rẽ
Xanh	AB, AC, AD, BA, DC, ED
Đỏ	BC, BD, EA
Lục	DA, DB
Vàng	EB, EC

Bảng 1.2 Bảng tổng hợp màu

Thuật toán tham ăn không đảm bảo cho ra kết quả tối ưu là số màu ít nhất được dùng. Tuy nhiên, người ta có thể dùng một số tính chất của đồ thị để đánh giá kết quả thu được.

Trở lại với vấn đề nút giao thông, từ kết quả tô màu trên, ta có thể thiết kế hệ thống đèn giao thông theo bảng tổng hợp màu trên, trong đó mỗi trạng thái của hệ thống đèn tương ứng với 1 màu. Tại mỗi trạng thái, các ngã rẽ nằm tại hàng tương ứng với màu đó được cho phép đi, các ngã rẽ còn lại bị cấm.

1.1.3. Phương pháp mô tả Thuật toán

Ước lượng thời gian thực hiện chương trình

Thời gian chạy của 1 chương trình phụ thuộc vào các yếu tố sau:

- Dữ liệu đầu vào
- Chất lượng của mã máy được tạo ra bởi chương trình dịch
- Tốc độ thực thi lệnh của máy
- Độ phức tạp về thời gian của thuật toán

Thông thường, thời gian chạy của chương trình không phụ thuộc vào giá trị dữ liệu đầu vào mà phụ thuộc vào kích thước của dữ liệu đầu vào. Do vậy thời gian chạy của chương trình nên được định nghĩa như là một hàm có tham số là kích thước dữ liệu đầu vào. Giả sử T là hàm ước lượng thời gian chạy của chương trình, khi đó với dữ liệu đầu vào có kích thước n thì thời gian chạy của chương trình là $T(n)$. Ví dụ, đối với một số chương trình thì thời gian chạy là an hoặc an^2 , trong đó a là hằng số. Đơn vị của hàm $T(n)$ là không xác định, tuy nhiên ta có thể xem như $T(n)$ là tổng số lệnh được thực hiện trên 1 máy tính lý tưởng.

Trong nhiều chương trình, thời gian thực hiện không chỉ phụ thuộc vào kích thước dữ liệu vào mà còn phụ thuộc vào tính chất của nó. Khi tính chất dữ liệu vào thoả mãn một số đặc điểm nào đó thì thời gian thực hiện chương trình có thể là lớn nhất hoặc nhỏ nhất. Khi đó, ta định nghĩa thời gian thực hiện chương trình $T(n)$ trong trường hợp xấu nhất hoặc tốt nhất. Đó là thời gian thực hiện lớn nhất hoặc nhỏ nhất trong tất cả các bộ dữ liệu vào có kích thước n . Ta cũng định nghĩa thời gian thực hiện trung bình của chương trình trên mọi bộ dữ liệu vào kích thước n . Trong thực tế, ước lượng thời gian thực hiện trung bình khó hơn nhiều so với thời gian thực hiện trong trường hợp xấu nhất hoặc tốt nhất, bởi vì việc phân tích thuật toán trong trường hợp trung bình khó hơn về mặt toán học, đồng thời khái niệm “trung bình” không có ý nghĩa thực sự rõ ràng.

Yếu tố chất lượng của mã máy được tạo bởi chương trình dịch và tốc độ thực thi lệnh của máy cũng ảnh hưởng tới thời gian thực hiện chương trình cho thấy chúng ta không thể thể hiện thời gian thực hiện chương trình dưới đơn vị thời gian chuẩn, chẳng hạn phút hoặc giây. Thay vào đó, ta có thể phát biểu thời gian thực hiện chương trình tỷ lệ với n hoặc n^2 v.v. Hệ số của tỷ lệ là 1 hằng số chưa xác định, phụ thuộc vào máy tính, chương trình dịch, và các nhân tố khác.

Ký hiệu $O(n)$

Để biểu thị cấp độ tăng của hàm, ta sử dụng ký hiệu $O(n)$. Ví dụ, ta nói thời gian thực hiện $T(n)$ của chương trình là $O(n^2)$, có nghĩa là tồn tại các hằng số dương c và n_0 sao cho $T(n) \leq cn^2$ với $n \geq n_0$.

Ví dụ, xét hàm $T(n) = (n+1)^2$. Ta có thể thấy $T(n)$ là $O(n^2)$ với $n_0 = 1$ và $c = 4$, vì ta có $T(n)$

$= (n+1)^2 < 4n^2$ với mọi $n \geq 1$. Trong ví dụ này, ta cũng có thể nói rằng $T(n)$ là $O(n^3)$, tuy nhiên, phát biểu này “yếu” hơn phát biểu $T(n)$ là $O(n^2)$.

Nhìn chung, ta nói $T(n)$ là $O(f(n))$ nếu tồn tại các hằng số dương c và n_0 sao cho $T(n) < c.f(n)$ với $n \geq n_0$. Một chương trình có thời gian thực hiện là $O(f(n))$ thì được xem là có cấp độ tăng $f(n)$.

Việc đánh giá các chương trình có thể được thực hiện qua việc đánh giá các hàm thời gian chạy của chương trình, bỏ qua các hằng số tỷ lệ. Với giả thiết này, một chương trình với thời gian thực hiện là $O(n^2)$ sẽ tốt hơn chương trình với thời gian chạy $O(n^3)$. Bên cạnh các yếu tố hằng số xuất phát từ chương trình dịch và máy, còn có thêm hằng số từ bản thân chương trình. Ví dụ, trên cùng một chương trình dịch và cùng 1 máy, chương trình đầu tiên có thời gian thực hiện là $100n^2$, trong khi chương trình thứ 2 có thời gian thực hiện là $5n^3$. Với n nhỏ, có thể $5n^3$ nhỏ hơn $100n^2$, tuy nhiên với n đủ lớn thì $5n^3$ sẽ lớn hơn $100n^2$ đáng kể.

Một lý do nữa để xem xét cấp độ tăng về thời gian thực hiện của chương trình là nó cho phép ta xác định độ lớn của bài toán mà ta có thể giải quyết. Mặc dù máy tính có tốc độ ngày càng cao, tuy nhiên, với những chương trình có thời gian thực hiện có cấp độ tăng lớn (từ n^2 trở lên), thì việc tăng tốc độ của máy tính tạo ra sự khác biệt không đáng kể về kích thước bài toán có thể xử lý bởi máy tính trong một khoảng thời gian cố định.

1.1.4. Độ phức tạp của thuật toán

Để tính toán được thời gian thực hiện chương trình, ta cần chú ý một số nguyên tắc cộng và nhân cấp độ tăng của hàm như sau :

Giả sử $T_1(n)$ và $T_2(n)$ là thời gian chạy của 2 đoạn chương trình P_1 và P_2 , trong đó $T_1(n)$ là $O(f(n))$ và $T_2(n)$ là $O(g(n))$. Khi đó, thời gian thực hiện của 2 đoạn chương trình nối tiếp P_1, P_2 là $O(\max(f(n), g(n)))$.

Nguyên tắc cộng trên có thể sử dụng để tính thời gian thực hiện của chương trình bao gồm 1 số tuần tự các bước, mỗi bước có thể là 1 đoạn chương trình bao gồm 1 số vòng lặp và rẽ nhánh. Ví dụ, giả sử ta có 3 bước thực hiện chương trình lần lượt có thời gian chạy là $O(n^2)$, $O(n^3)$, $O(n \log n)$. Khi đó, thời gian chạy của 2 đoạn chương trình đầu là $O(\max(n^2, n^3)) = O(n^3)$, còn thời gian chạy của cả 3 đoạn chương trình là $O(\max(n^3, n \log n)) = O(n^3)$.

Nguyên tắc nhân cấp độ tăng của hàm như sau: Với giả thiết về $T_1(n)$ và $T_2(n)$ như trên, nếu 2 đoạn chương trình P_1 và P_2 không được thực hiện tuần tự mà lồng nhau thì thời gian chạy tổng thể sẽ là $T_1(n).T_2(n) = O(f(n).g(n))$.

Để minh họa cho việc phân tích và tính toán thời gian thực hiện của 1 chương trình, ta sẽ xem xét một thuật toán đơn giản để sắp xếp các phần tử của một tập hợp, đó là thuật toán sắp xếp nổi bọt (bubble sort).

Thuật toán như sau :

```
void bubble (int a[n]){
    int i, j, temp;
    1.      for (i = 0; i < n-1;i++)
    2.          for (j = n-1; j>= i+1;j--)
    3.              if (a[j-1] >a[j]){
// Đổi chỗ cho a[j] và a[j-1]
    4.                  temp =a[j-1];
    5.                  a[j-1] =a[j];

    6.                  a[j] =temp;
    }
    }
```

Trong thuật toán này, mỗi lần duyệt của vòng lặp trong (biến duyệt j) sẽ làm “nổi” lên trên phần tử nhỏ nhất trong các phần tử được duyệt.

Dễ thấy rằng kích thước dữ liệu vào chính là số phần tử được sắp, n. Mỗi lệnh gán sẽ có thời gian thực hiện cố định, không phụ thuộc vào n, do vậy, các lệnh 4, 5, 6 sẽ có thời gian thực hiện là $O(1)$, tức thời gian thực hiện là hằng số. Theo quy tắc cộng cấp độ tăng thì tổng thời gian thực hiện cả 3 lệnh là $O(\max(1, 1, 1)) = O(1)$.

Tiếp theo ta sẽ xem xét thời gian thực hiện của các lệnh lặp và rẽ nhánh. Lệnh If kiểm tra điều kiện để thực hiện nhóm lệnh gán 4, 5, 6. Việc kiểm tra điều kiện sẽ có thời gian thực hiện là $O(1)$. Ngoài ra, chúng ta chưa biết được là điều kiện có thoả mãn hay không, tức là không biết được nhóm lệnh gán có được thực hiện hay không. Do vậy, ta giả thiết trường hợp xấu nhất là tất cả các lần kiểm tra điều kiện đều thoả mãn, và các lệnh gán được thực hiện. Như vậy, toàn bộ lệnh If sẽ có thời gian thực hiện là $O(1)$.

Tiếp tục xét từ trong ra ngoài, ta xét đến vòng lặp trong (biến duyệt j). Trong vòng lặp này, tại mỗi bước lặp ta cần thực hiện các thao tác như kiểm tra đã gặp điều kiện dừng chưa và tăng biến duyệt lên 1 nếu chưa dừng. Như vậy,

mỗi bước lặp có thời gian thực hiện là $O(1)$. Số bước lặp là $n-i$, do đó theo quy tắc nhân cấp độ tăng thì tổng thời gian thực hiện của vòng lặp này là $O((n-i) \times 1) = O(n-i)$.

Cuối cùng, ta xét vòng lặp ngoài cùng (biến duyệt i). Vòng lặp này được thực hiện (n-1) lần, do đó, tổng thời gian thực hiện của chương trình là:

$\sum (n-i) = n(n-1)/2 = n^2/2 - n/2 = O(n^2)$ Như vậy, thời gian thực hiện giải thuật sắp xếp nổi bọt là tỷ lệ với n^2 .

Một số quy tắc chung trong việc phân tích và tính toán thời gian thực hiện chương trình

- Thời gian thực hiện các lệnh gán, đọc, ghi .v.v, luôn là $O(1)$.
- Thời gian thực hiện chuỗi tuần tự các lệnh được xác định theo quy tắc cộng cấp độ tăng. Có nghĩa là thời gian thực hiện của cả nhóm lệnh tuần tự được tính là thời gian thực hiện của lệnh lớn nhất.
- Thời gian thực hiện lệnh rẽ nhánh If được tính bằng thời gian thực hiện các lệnh khi điều kiện kiểm tra được thỏa mãn và thời gian thực hiện việc kiểm tra điều kiện. Thời gian thực hiện việc kiểm tra điều kiện luôn là $O(1)$.
- Thời gian thực hiện 1 vòng lặp được tính là tổng thời gian thực hiện các lệnh ở thân vòng lặp qua tất cả các bước lặp và thời gian để kiểm tra điều kiện dừng (thường là $O(1)$). Thời gian thực hiện này thường được tính theo quy tắc nhân cấp độ tăng số lần thực hiện bước lặp và thời gian thực hiện các lệnh ở thân vòng lặp. Các vòng lặp phải được tính thời gian thực hiện một cách riêng rẽ.

1.2 Các cấu trúc dữ liệu cơ bản

1.2.1 Dữ liệu kiểu số

Tùy ngôn ngữ lập trình, các kiểu dữ liệu định nghĩa sẵn có thể khác nhau đôi chút. Với ngôn ngữ C, các kiểu dữ liệu này chỉ gồm số nguyên, số thực, ký tự. Và theo quan điểm của C, kiểu ký tự thực chất cũng là kiểu số nguyên về mặt lưu trữ, chỉ khác về cách sử dụng. Ngoài ra, giá trị logic ĐÚNG (TRUE) và giá trị logic SAI (FALSE) được biểu diễn trong C như là các giá trị nguyên khác zero và zero. Trong khi đó PASCAL định nghĩa tất cả các kiểu dữ liệu cơ sở đã liệt kê ở trên và phân biệt chúng một cách chặt chẽ. Trong giới hạn giáo trình này ngôn ngữ chính dùng để minh họa sẽ là C.

Các kiểu dữ liệu định sẵn trong C gồm các kiểu sau:

Tên kiểu	Kthước	Miền giá trị	Ghi chú
Char	01 byte	-128 đến 127	Có thể dùng như số nguyên 1 byte có dấu hoặc kiểu ký tự
unsign char	01 byte	0 đến 255	Số nguyên 1 byte không dấu

Int	02 byte	-32768 đến 32767	
unsign int	02 byte	0 đến 65535	Có thể gọi tắt là unsign
Long	04 byte	-2^{32} đến $2^{31} - 1$	
unsign long	04 byte	0 đến $2^{32}-1$	
Float	04 byte	$3.4E-38$ $\frac{1}{4}$ $3.4E38$	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	08 byte	$1.7E-308$ $\frac{1}{4}$ $1.7E308$	
long double	10 byte	$3.4E-4932$ $\frac{1}{4}$ $1.1E4932$	

Một số điều đáng lưu ý đối với các kiểu dữ liệu cơ bản trong C là kiểu ký tự (char) có thể dùng theo hai cách (số nguyên 1 byte hoặc ký tự). Ngoài ra C không định nghĩa kiểu logic (boolean) mà nó đơn giản đồng nhất một giá trị nguyên khác 0 với giá trị TRUE và giá trị 0 với giá trị FALSE khi có nhu cầu xét các giá trị logic. Như vậy, trong C xét cho cùng chỉ có 2 loại dữ liệu cơ bản là số nguyên và số thực. Tức là chỉ có dữ liệu số. Hơn nữa các số nguyên trong C có thể được thể hiện trong 3 hệ cơ số là hệ thập phân, hệ thập lục phân và hệ bát phân. Nhờ những quan điểm trên, C rất được những người lập trình chuyên nghiệp thích dùng.

Các kiểu cơ sở rất đơn giản và không thể hiện rõ sự tổ chức dữ liệu trong một cấu trúc, thường chỉ được sử dụng làm nền để xây dựng các kiểu dữ liệu phức tạp khác.

1.2.2. Dữ liệu kiểu ký tự

Tuy nhiên trong nhiều trường hợp, chỉ với các kiểu dữ liệu cơ sở không đủ để phản ánh tự nhiên và đầy đủ bản chất của sự vật thực tế, dẫn đến nhu cầu phải xây dựng các kiểu dữ liệu mới dựa trên việc tổ chức, liên kết các thành phần dữ liệu có kiểu dữ liệu đã được định nghĩa. Những kiểu dữ liệu được xây dựng như thế gọi là kiểu dữ liệu có cấu trúc. Đa số các ngôn ngữ lập trình đều cài đặt sẵn một số kiểu có cấu trúc cơ bản như mảng, chuỗi, tập tin, bản ghi... và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới.

Ví dụ : 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
- Nơi sinh: chuỗi ký tự
- Điểm thi: số nguyên

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

```
int Diemthi;
```

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

```
char masv[15];
```

```
char tensv[15];
```

```
char noisinh[15];
```

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

```
typedef struct tagDate{
```

```
char ngay;
```

```
char thang;
```

```
char thang;
```

```
}Date;
```

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

```
typedef struct tagSinhVien{
```

```
char masv[15];
```

```
char tensv[15];
```

```
char noisinh[15];
```

```
int Diem thi;
```

```
}SinhVien;
```

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...Mục tiêu của việc nghiên cứu cấu trúc dữ liệu chính là tìm những phương cách thích hợp để tổ chức, liên kết dữ liệu, hình thành các kiểu dữ liệu có cấu trúc từ những kiểu dữ liệu đã được định nghĩa.

1.3. Một số giải thuật cơ bản trên các kiểu dữ liệu cơ bản

1.3.1. Một số thuật toán cơ bản trên dữ liệu kiểu số

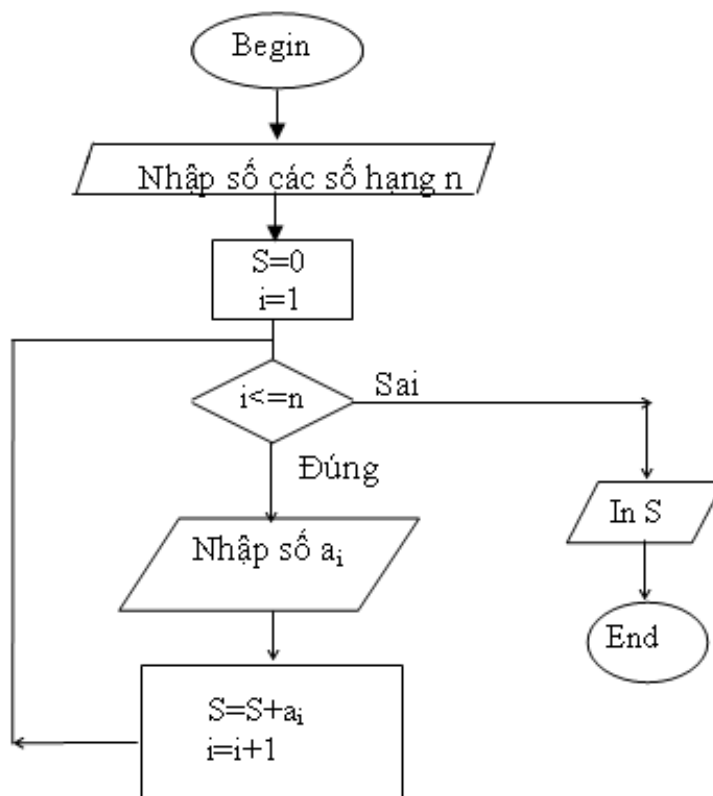
Ví dụ 1: Cần viết chương trình cho máy tính sao cho khi thực hiện chương trình đó, máy tính yêu cầu người sử dụng chương trình nhập vào các số hạng của tổng (n); nhập vào dãy các số hạng a_i của tổng. Sau đó, máy tính sẽ thực hiện việc tính tổng các số a_i này và in kết quả của tổng tính được.

Yêu cầu: Tính tổng n số $S = a_1 + a_2 + a_3 + \dots + a_n$.

Để tính tổng trên, chúng ta sử dụng phương pháp “cộng tích lũy” nghĩa là khởi đầu cho $S=0$. Sau mỗi lần nhận được một số hạng a_i từ bàn phím, ta cộng tích lũy a_i vào S (lấy giá trị được lưu trữ trong S , cộng thêm a_i và lưu trữ lại vào S). Tiếp tục quá trình này đến khi ta tích lũy được a_n vào S thì ta có S là tổng các a_i . Chi tiết giải thuật được mô tả bằng ngôn ngữ tự nhiên như sau:

- Bước 1: Nhập số các số hạng n .
- Bước 2: Cho $S=0$ (lưu trữ số 0 trong S)
- Bước 3: Cho $i=1$ (lưu trữ số 1 trong i)
- Bước 4: Kiểm tra nếu $i \leq n$ thì thực hiện bước 5, ngược lại thực hiện bước 8.
- Bước 5: Nhập a_i
- Bước 6: Cho $S=S+a_i$ (lưu trữ giá trị $S + a_i$ trong S)
- Bước 7: Tăng i lên 1 đơn vị và quay lại bước 4.
- Bước 8: In S và kết thúc chương trình.

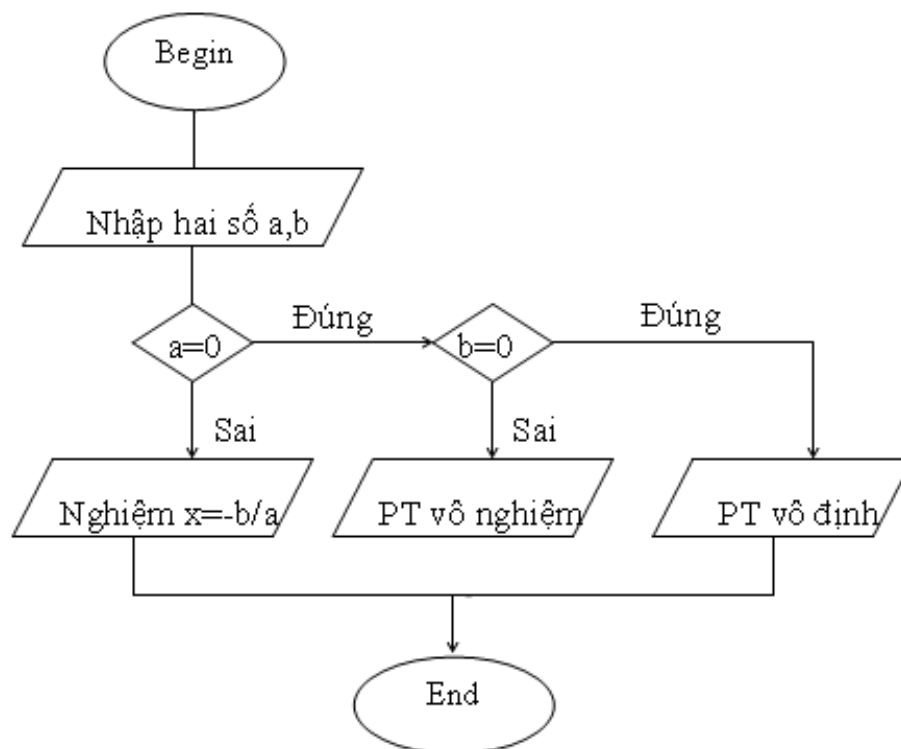
Chi tiết giải thuật bằng lưu đồ:



Ví dụ 2: Viết chương trình cho phép nhập vào 2 giá trị a, b mang ý nghĩa là các hệ số a, b của phương trình bậc nhất. Dựa vào các giá trị a, b đó cho biết nghiệm của phương trình bậc nhất $ax + b = 0$.

Mô tả giải thuật bằng ngôn ngữ tự nhiên:

- Bước 1: Nhập 2 số a và b
- Bước 2: Nếu $a = 0$ thì thực hiện bước 3, ngược lại thực hiện bước 4
- Bước 3: Nếu $b=0$ thì thông báo phương trình vô số nghiệm và kết thúc chương trình, ngược lại thông báo phương trình vô nghiệm và kết thúc chương trình.
- Bước 4: Thông báo nghiệm của phương trình là $-b/a$ và kết thúc.



1.3.2. Một số thuật toán cơ bản trên dữ liệu kiểu ký tự

Ví dụ 1: Viết chương trình cho phép nhập vào 1 số n , sau đó lần lượt nhập vào n giá trị a_1, a_2, \dots, a_n . Hãy tìm và in ra giá trị lớn nhất trong n số a_1, a_2, \dots, a_n .

Để giải quyết bài toán trên, chúng ta áp dụng phương pháp “thử và sửa”. Ban đầu giả sử a_1 là số lớn nhất (được lưu trong giá trị max); sau đó lần lượt xét các a_i còn lại, nếu a_i nào lớn hơn giá trị max thì lúc đó max sẽ nhận giá trị là a_i . Sau khi đã xét hết các a_i thì max chính là giá trị lớn nhất cần tìm.

Mô tả giải thuật bằng ngôn ngữ tự nhiên:

- Bước 1: Nhập số n
- Bước 2: Nhập số thứ nhất a_1
- Bước 3: Gán $\text{max}=a_1$

- Bước 4: Gán $i=2$
- Bước 5: Nếu $i \leq n$ thì thực hiện bước 6, ngược lại thực hiện bước 9
- Bước 6: Nhập a_i
- Bước 7: Nếu $\max < a_i$ thì gán $\max = a_i$.
- Bước 8: Tăng i lên một đơn vị và quay lại bước 5
- Bước 9: In \max - kết thúc

Phần mô tả giải thuật bằng lưu đồ, sinh viên tự làm xem như bài tập.

Ví dụ 2: Viết chương trình cho phép nhập vào 1 số n , sau đó lần lượt nhập vào n giá trị a_1, a_2, \dots, a_n . Sắp theo thứ tự tăng dần một dãy n số a_1, a_2, \dots, a_n nói trên. Có rất nhiều giải thuật để giải quyết bài toán này. Phần trình bày dưới đây là một phương pháp.

Giả sử ta đã nhập vào máy dãy n số a_1, a_2, \dots, a_n . Việc sắp xếp dãy số này trải qua $(n-1)$ lần:

- Lần 1: So sánh phần tử đầu tiên với tất cả các phần tử đứng sau phần tử đầu tiên. Nếu có phần tử nào nhỏ hơn phần tử đầu tiên thì đổi chỗ phần tử đầu tiên với phần tử nhỏ hơn đó. Sau lần 1, ta được phần tử đầu tiên là phần tử nhỏ nhất.

- Lần 2: So sánh phần tử thứ 2 với tất cả các phần tử đứng sau phần tử thứ 2. Nếu có phần tử nào nhỏ hơn phần tử thứ 2 thì đổi chỗ phần tử thứ 2 với phần tử nhỏ hơn đó. Sau lần 2, ta được phần tử đầu tiên và phần tử thứ 2 là đúng vị trí của nó khi sắp xếp.

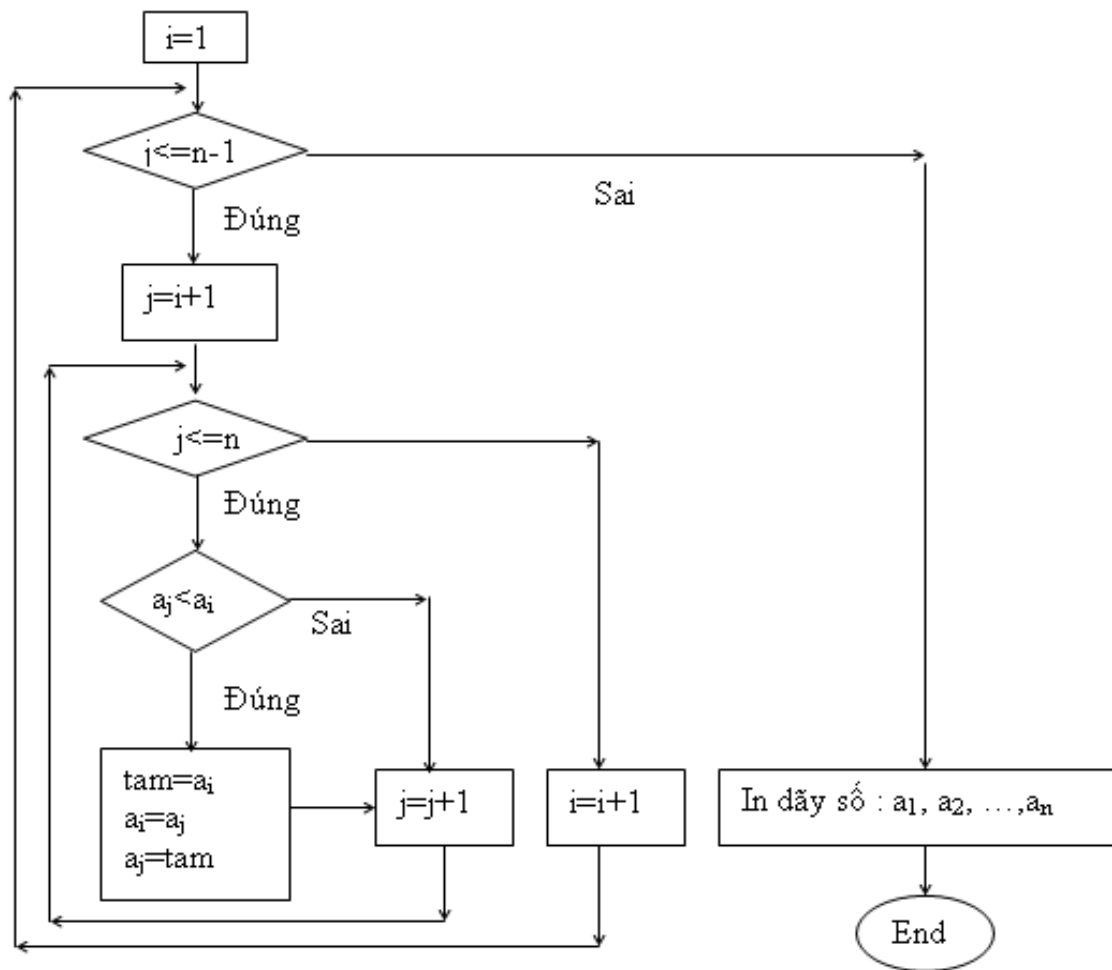
- ...

- Lần $(n-1)$: So sánh phần tử thứ $(n-1)$ với phần tử đứng sau phần tử $(n-1)$ là phần tử thứ n . Nếu phần tử thứ n nhỏ hơn phần tử thứ $(n-1)$ thì đổi chỗ 2 phần tử này. Sau lần thứ $(n-1)$, ta được danh sách gồm n phần tử được sắp thứ tự.

Mô tả giải thuật bằng ngôn ngữ tự nhiên:

- Bước 1: Gán $i=1$
- Bước 2: Gán $j=i+1$
- Bước 3: Nếu $i \leq n-1$ thì thực hiện bước 4, ngược lại thực hiện bước 8
- Bước 4: Nếu $j \leq n$ thì thực hiện bước 5, ngược lại thì thực hiện bước 7.
- Bước 5: Nếu $a_i > a_j$ thì hoán đổi a_i và a_j cho nhau (nếu không thì thôi).
- Bước 6: Tăng j lên một đơn vị và quay lại bước 4
- Bước 7: Tăng i lên một đơn vị và quay lại bước 3
- Bước 8: In dãy số a_1, a_2, \dots, a_n - Kết thúc.

Mô tả giải thuật sắp xếp bằng lưu đồ



1.4. CASE STUDY

1.4.1. Cài đặt các thuật toán trên dữ liệu kiểu số.

Thuật toán được cài đặt như sau:

```

int USCLN(int m, int
n){ if (n==0) return m;
    else return USCLN(n, m % n);
}

```

Điểm dừng của thuật toán là khi $n=0$. Khi đó đương nhiên là USCLN của m và 0 chính là m , vì 0 chia hết cho mọi số. Khi n khác 0 , lời gọi đệ qui $\text{USCLN}(n, m\%n)$ được thực hiện. Chú ý rằng ta giả sử $m \geq n$ trong thủ tục tính USCLN, do đó, khi gọi đệ qui ta gọi $\text{USCLN}(n, m\%n)$ để đảm bảo thứ tự các tham số vì n bao giờ cũng lớn hơn phần dư của phép m cho n . Sau mỗi lần gọi đệ qui, các tham số của thủ tục sẽ nhỏ dần đi, và sau 1 số hữu hạn lời gọi tham số nhỏ hơn sẽ bằng 0 . Đó chính là điểm dừng của thuật toán.

Ví dụ, để tính USCLN của 108 và 45, ta gọi thủ tục $\text{USCLN}(108, 45)$. Khi đó, các thủ tục sau sẽ lần lượt được gọi:

$\text{USCLN}(108, 45)$ 108 chia 45 dư 18, do đó tiếp theo gọi

$\text{USCLN}(45, 18)$ 45 chia 18 dư 9, do đó tiếp theo gọi

$\text{USCLN}(18, 9)$ 18 chia 9 dư 0, do đó tiếp theo gọi

$\text{USCLN}(9, 0)$ tham số thứ 2 = 0, do đó kết quả là tham số thứ nhất, tức là 9.

Như vậy, ta tìm được USCLN của 108 và 45 là 9 chỉ sau 4 lần gọi thủ tục.

1.4.2. Cài đặt các thuật toán trên dữ liệu kiểu ký tự.

Ý tưởng cơ bản của các thuật toán dạng chia để trị là phân chia bài toán ban đầu thành 2 hoặc nhiều bài toán con có dạng tương tự và lần lượt giải quyết từng bài toán con này. Các bài toán con này được coi là dạng đơn giản hơn của bài toán ban đầu, do vậy có thể sử dụng các lời gọi đệ quy để giải quyết. Thông thường, các thuật toán chia để trị chia bộ dữ liệu đầu vào thành 2 phần riêng rẽ, sau đó gọi 2 thủ tục đệ quy để với các bộ dữ liệu đầu vào là các phần vừa được chia.

Một ví dụ điển hình của giải thuật chia để trị là Quicksort, một giải thuật sắp xếp nhanh. Ý tưởng cơ bản của giải thuật này như sau:

Giải sử ta cần sắp xếp 1 dãy các số theo chiều tăng dần. Tiến hành chia dãy đó thành 2 nửa sao cho các số trong nửa đầu đều nhỏ hơn các số trong nửa sau. Sau đó, tiến hành thực hiện sắp xếp trên mỗi nửa này. Rõ ràng là sau khi mỗi nửa đã được sắp, ta tiến hành ghép chúng lại thì sẽ có toàn bộ dãy được sắp. Chi tiết về giải thuật Quicksort sẽ được trình bày trong chương 7 - Sắp xếp và tìm kiếm.

Tiếp theo, chúng ta sẽ xem xét một bài toán cũng rất điển hình cho lớp bài toán được giải bằng giải thuật đệ quy chia để trị.

Chương 2. Duyệt và đệ qui

Chương 2 trình bày các khái niệm về định nghĩa về duyệt và đệ qui, chương trình đệ qui. Ngoài việc trình bày các ưu điểm của chương trình duyệt, đệ qui, các tình huống không nên sử dụng đệ qui cũng được đề cập cùng với các ví dụ minh họa.

Chương này cũng đề cập và phân tích một số thuật toán đệ qui tiêu biểu và kinh điển như bài toán tháp Hà nội, các thuật toán quay lui.v.v

Để học tốt chương này, sinh viên cần nắm vững phân lý thuyết. Sau đó, nghiên cứu kỹ các phân tích thuật toán và thực hiện chạy thử chương trình. Có thể thay đổi một số điểm trong chương trình và chạy thử để nắm kỹ hơn về thuật toán. Ngoài ra, sinh viên cũng có thể tìm các bài toán tương tự để phân tích và giải quyết bằng chương trình.

2.1. Thuật toán duyệt

2.1.1. Phương pháp duyệt toàn bộ

2.1.2. Phương pháp Heuristic

Heuristic là các kỹ thuật dựa trên kinh nghiệm để giải quyết vấn đề, học hỏi hay khám phá nhằm đưa ra một giải pháp mà không được đảm bảo là tối ưu. Với việc nghiên cứu khảo sát không có tính thực tế, các phương pháp heuristic được dùng nhằm tăng nhanh quá trình tìm kiếm với các giải pháp hợp lý thông qua các suy nghĩ rút gọn để giảm bớt việc nhận thức vấn đề khi đưa ra quyết định. Ví dụ của phương pháp này bao gồm sử dụng một luật ngón tay cái, giả thuyết, phán đoán trực giác, khuôn mẫu hay nhận thức thông thường.

Thuật giải Heuristic là một sự mở rộng khái niệm thuật toán. Nó thể hiện cách giải bài toán với các đặc tính sau :

Thường tìm được lời giải tốt (nhưng không chắc là lời giải tốt nhất)

Giải bài toán theo thuật giải Heuristic thường dễ dàng và nhanh chóng đưa ra kết quả hơn so với giải thuật tối ưu, vì vậy chi phí thấp hơn.

Thuật giải Heuristic thường thể hiện khá tự nhiên, gần gũi với cách suy nghĩ và hành động của con người.

Có nhiều phương pháp để xây dựng một thuật giải Heuristic, trong đó người ta thường dựa vào một số nguyên lý cơ sở như sau:

Nguyên lý vét cạn thông minh :

Trong một bài toán tìm kiếm nào đó, khi không gian tìm kiếm lớn, ta thường tìm cách giới hạn lại không gian tìm kiếm hoặc thực hiện một kiểu dò tìm đặc biệt dựa vào đặc thù của bài toán để nhanh chóng tìm ra mục tiêu.

Nguyên lý tham lam (Greedy):

Lấy tiêu chuẩn tối ưu (trên phạm vi toàn cục) của bài toán để làm tiêu chuẩn chọn lựa hành động cho phạm vi cục bộ của từng bước (hay từng giai đoạn) trong quá trình tìm kiếm lời giải.

Nguyên lý thứ tự :

Thực hiện hành động dựa trên một cấu trúc thứ tự hợp lý của không gian khảo sát nhằm nhanh chóng đạt được một lời giải tốt.

2.1.3. Một số bài toán sử dụng thuật toán duyệt

2.2. Thuật toán đệ qui

2.2.1. Giới thiệu về thuật toán đệ qui

a) Khái niệm:

Đệ qui là một khái niệm cơ bản trong toán học và khoa học máy tính. Một đối tượng được gọi là đệ qui nếu nó hoặc một phần của nó được định nghĩa thông qua khái niệm về chính nó. Một số ví dụ điển hình về việc định nghĩa bằng đệ qui là:

- 1- Định nghĩa số tự nhiên:
 - 0 là số tự nhiên.
 - Nếu k là số tự nhiên thì $k+1$ cũng là số tự nhiên.

Như vậy, bắt đầu từ phát biểu “0 là số tự nhiên”, ta suy ra $0+1=1$ là số tự nhiên. Tiếp theo $1+1=2$ là số tự nhiên, v.v.

- 2- Định nghĩa xâu ký tự bằng đệ qui:
 - Xâu rỗng là 1 xâu ký tự.
 - Một chữ cái bất kỳ ghép với 1 xâu sẽ tạo thành 1 xâu mới.

Từ phát biểu “Xâu rỗng là 1 xâu ký tự”, ta ghép bất kỳ 1 chữ cái nào với xâu rỗng đều tạo thành xâu ký tự. Như vậy, chữ cái bất kỳ có thể coi là xâu ký tự. Tiếp tục ghép 1 chữ cái bất kỳ với 1 chữ cái bất kỳ cũng tạo thành 1 xâu ký tự, v.v.

- 3- Định nghĩa hàm giai thừa, $n!$.
 - Khi $n=0$, định nghĩa $0!=1$
 - Khi $n>0$, định nghĩa $n!=(n-1)! \times n$

Như vậy, khi $n=1$, ta có $1!=0! \times 1 = 1 \times 1 = 1$. Khi $n=2$, ta có $2!=1! \times 2 = 1 \times 2 = 2$, v.v.

Trong lĩnh vực lập trình, một chương trình máy tính gọi là đệ qui nếu trong chương trình có lời gọi chính nó. Điều này, thoạt tiên, nghe có vẻ hơi vô lý. Một chương trình không thể gọi mãi chính nó, vì như vậy sẽ tạo ra một vòng lặp vô hạn. Trên thực tế, một chương trình đệ qui trước khi gọi chính nó bao giờ cũng có một thao tác kiểm tra điều kiện dừng. Nếu điều kiện dừng thỏa mãn, chương trình sẽ không gọi chính nó nữa, và quá trình đệ qui chấm dứt. Trong các ví dụ ở trên, ta đều thấy có các điểm dừng. Chẳng hạn, trong ví dụ thứ nhất, nếu $k = 0$ thì có thể suy ngay k là số tự nhiên, không cần tham chiếu xem $k-1$ có là số tự nhiên hay không.

Nhìn chung, các chương trình đệ qui đều có các đặc điểm sau:

- Chương trình này có thể gọi chính nó.
- Khi chương trình gọi chính nó, mục đích là để giải quyết 1 vấn đề tương tự, nhưng nhỏ hơn.
- Vấn đề nhỏ hơn này, cho tới 1 lúc nào đó, sẽ đơn giản tới mức chương trình có thể tự giải quyết được mà không cần gọi tới chính nó nữa.

Khi chương trình gọi tới chính nó, các tham số, hoặc khoảng tham số, thường trở nên nhỏ hơn, để phản ánh 1 thực tế là vấn đề đã trở nên nhỏ hơn, dễ hơn. Khi tham số giảm tới mức cực tiểu, một điều kiện so sánh được kiểm tra và chương trình kết thúc, chấm dứt việc gọi tới chính nó.

Ưu điểm của chương trình đệ qui cũng như định nghĩa bằng đệ qui là có thể thực hiện một số lượng lớn các thao tác tính toán thông qua 1 đoạn chương trình ngắn gọn (thậm chí không có vòng lặp, hoặc không tường minh để có thể thực hiện bằng các vòng lặp) hay có thể định nghĩa một tập hợp vô hạn các đối tượng thông qua một số hữu hạn lời phát biểu. Thông thường, một chương trình được viết dưới dạng đệ qui khi vấn đề cần xử lý có thể được giải quyết bằng đệ qui. Tức là vấn đề cần giải quyết có thể đưa được về vấn đề tương tự, nhưng đơn giản hơn. Vấn đề này lại được đưa về vấn đề tương tự nhưng đơn giản hơn nữa .v.v, cho đến khi đơn giản tới mức có thể trực tiếp giải quyết được ngay mà không cần đưa về vấn đề đơn giản hơn nữa.

b) Điều kiện để có thể viết một chương trình đệ qui

Như đã nói ở trên, để chương trình có thể viết dưới dạng đệ qui thì vấn đề cần xử lý phải được giải quyết 1 cách đệ qui. Ngoài ra, ngôn ngữ dùng để viết chương trình phải hỗ trợ đệ qui. Để có thể viết chương trình đệ qui chỉ cần sử dụng ngôn ngữ lập trình có hỗ trợ hàm hoặc thủ tục, nhờ đó một thủ tục hoặc hàm có thể có lời gọi đến chính thủ tục hoặc hàm đó. Các ngôn ngữ lập trình thông dụng hiện nay đều hỗ trợ kỹ thuật này, do vậy vấn đề công cụ để tạo các chương trình đệ qui không phải là vấn đề cần phải xem xét. Tuy nhiên, cũng nên lưu ý rằng khi một thủ tục đệ qui gọi đến chính nó, một bản sao của tập các đối tượng được sử dụng trong thủ tục này như các biến, hằng, các thủ tục con, .v.v. cũng được tạo ra. Do vậy, nên hạn chế việc khai báo và sử dụng các đối tượng này trong thủ tục đệ qui nếu không cần thiết nhằm tránh lãng phí bộ nhớ, đặc biệt đối với các lời gọi đệ qui được gọi đi gọi lại nhiều lần. Các đối tượng cục bộ của 1 thủ tục đệ qui khi được tạo ra nhiều lần, mặc dù có cùng tên, nhưng do khác phạm vi nên không ảnh hưởng gì đến chương trình. Các đối tượng đó sẽ được giải phóng khi thủ tục chứa nó kết thúc.

Nếu trong một thủ tục có lời gọi đến chính nó thì ta gọi đó là đệ qui trực tiếp.

Còn trong trường hợp một thủ tục có một lời gọi thủ tục khác, thủ tục này lại gọi đến thủ tục ban đầu thì được gọi là đệ qui gián tiếp. Như vậy, trong chương trình khi nhìn vào có thể không thấy ngay sự đệ qui, nhưng khi xem xét kỹ hơn thì sẽ nhận ra.

c) Khi nào không nên sử dụng đệ qui

Trong nhiều trường hợp, một chương trình có thể viết dưới dạng đệ qui. Tuy nhiên, đệ qui không hẳn đã là giải pháp tốt nhất cho vấn đề. Nhìn chung, khi chương trình có thể viết dưới dạng lặp hoặc các cấu trúc lệnh khác thì không nên sử dụng đệ qui.

Lý do thứ nhất là, như đã nói ở trên, khi một thủ tục đệ qui gọi chính nó, tập các đối tượng được sử dụng trong thủ tục này như các biến, hằng, cấu trúc .v.v sẽ được tạo ra. Ngoài ra, việc chuyển giao điều khiển từ các thủ tục cũng cần lưu trữ các thông số dùng cho việc trả lại điều khiển cho thủ tục ban đầu.

Lý do thứ hai là việc sử dụng đệ qui đôi khi tạo ra các tính toán thừa, không cần thiết do tính chất tự động gọi thực hiện thủ tục khi chưa gặp điều kiện dừng của đệ qui.

2.2.2. Một số bài toán sử dụng thuật toán đệ qui

Xét bài toán tính các phần tử của dãy Fibonacci. Dãy Fibonacci được định nghĩa như sau:

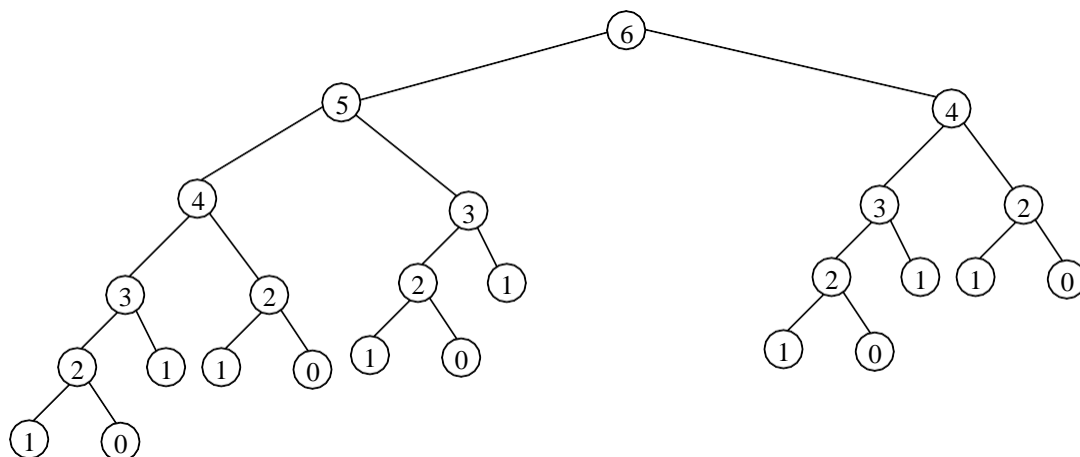
- $F(0) = 0$
- $F(1) = 1$
- Với $n > 1$ thì $F(n) = F(n-1) + F(n-2)$

Rõ ràng là nhìn vào một định nghĩa đệ qui như trên, chương trình tính phần tử của dãy Fibonacci có vẻ như rất phù hợp với thuật toán đệ qui. Phương thức đệ qui để tính dãy này có thể được viết như sau:

```
int Fibonacci(int i){  
    if (i==0) return 0; if (i==1) return 1;  
    return Fibonacci(i-1) + Fibonacci (i-2)  
}
```

Kết quả thực hiện chương trình không có gì sai. Tuy nhiên, chú ý rằng một lời gọi đệ qui Fibonacci (n) sẽ dẫn tới 2 lời gọi đệ qui khác ứng với n-1 và n-2. Hai lời gọi này lại gây ra 4 lời gọi nữa .v.v, cứ như vậy số lời gọi đệ qui sẽ tăng theo cấp số mũ. Điều này rõ ràng là không hiệu quả vì trong số các lời gọi đệ qui đó có rất nhiều lời gọi trùng nhau. Ví dụ lời gọi đệ qui Fibonacci (6) sẽ dẫn đến 2 lời gọi Fibonacci (5) và Fibonacci (4). Lời gọi Fibonacci (5) sẽ gọi Fibonacci (4) và Fibonacci (3). Ngay chỗ này, ta đã thấy có 2 lời gọi Fibonacci (4) được thực hiện.

Hình 2.1 cho thấy số các lời gọi được thực hiện khi gọi thủ tục Fibonacci(6).



Hình 2.1 Các lời gọi đệ qui được thực hiện khi gọi thủ tục Fibonacci (6)

Trong hình vẽ trên, ta thấy để tính được phần tử thứ 6 thì cần có tới 25 lời gọi ! Sau đây, ta sẽ xem xét việc sử dụng vòng lặp để tính giá trị các phần tử của dãy Fibonacci như thế nào.

Đầu tiên, ta khai báo một mảng F các số tự nhiên để chứa các số Fibonacci. Vòng lặp để tính và gán các số này vào mảng rất đơn giản như sau:

```
F[0]=0;
F[1]=1;
for (i=2; i<n-1; i++)
    F[i] = F[i-1] + F[i-2];
```

Rõ ràng là với vòng lặp này, mỗi số Fibonacci (n) chỉ được tính 1 lần thay vì được tính toán chồng chéo như ở trên.

Tóm lại, nên tránh sử dụng đệ qui nếu có một giải pháp khác cho bài toán. Mặc dù vậy, một số bài toán tỏ ra rất phù hợp với phương pháp đệ qui. Việc sử dụng đệ qui để giải quyết các bài toán này hiệu quả và rất dễ hiểu. Trên thực tế, tất cả các giải thuật đệ qui đều có thể được đưa về dạng lặp (còn gọi là “khử” đệ qui). Tuy nhiên, điều này có thể làm cho chương trình trở nên phức tạp, nhất là khi phải thực hiện các thao tác điều khiển stack đệ qui (bạn đọc có thể tìm hiểu thêm kỹ thuật khử đệ qui ở các tài liệu tham khảo khác), dẫn đến việc chương trình trở nên rất khó hiểu. Phần tiếp theo sẽ trình bày một số thuật toán đệ qui điển hình.

2.3. Qui hoạch động

2.3.1. Giới thiệu phương pháp qui hoạch động

Ý tưởng cơ bản của các thuật toán dạng chia để trị là phân chia bài toán ban đầu thành 2 hoặc nhiều bài toán con có dạng tương tự và lần lượt giải quyết từng bài toán con này. Các bài toán con này được coi là đơn giản hơn của bài toán ban đầu, do vậy có thể sử dụng các lời gọi đệ qui để giải quyết. Thông thường, các thuật toán chia để trị chia bộ dữ liệu đầu vào thành 2 phần riêng rẽ, sau đó gọi 2 thủ tục đệ qui để với các bộ dữ liệu đầu vào là các phần vừa được chia.

Một ví dụ điển hình của giải thuật chia để trị là Quicksort, một giải thuật sắp xếp nhanh. Ý tưởng cơ bản của giải thuật này như sau:

Giải sử ta cần sắp xếp 1 dãy các số theo chiều tăng dần. Tiến hành chia dãy đó thành 2 nửa sao cho các số trong nửa đầu đều nhỏ hơn các số trong nửa sau. Sau đó, tiến hành thực hiện sắp xếp trên mỗi nửa này. Rõ ràng là sau khi mỗi nửa đã được sắp, ta tiến hành ghép chúng lại thì sẽ có toàn bộ dãy được sắp. Chi tiết về giải thuật Quicksort sẽ được trình bày trong chương 7 - Sắp xếp và tìm kiếm.

Tiếp theo, chúng ta sẽ xem xét một bài toán cũng rất điển hình cho lớp bài toán được giải bằng giải thuật đệ qui chia để trị.

2.3.2. Một số bài toán sử dụng phương pháp qui hoạch động

2.4. CASE STUDY

2.4.1. Xây dựng và cài đặt thuật toán duyệt cho một số bài toán thực tế.

Cho bàn cờ có kích thước n . Một quân mã được đặt tại ô đầu có tọa độ x_0, y_0 và được phép dịch chuyển theo luật cờ thông thường. Bài toán đặt ra là từ ô ban đầu tìm được một chuỗi các nước đi của quân mã, sao cho quân mã này đi qua tất cả các ô của bàn cờ, mỗi ô đúng 1 lần.

Như đã nói ở trên, quá trình thử - sai ban đầu được xem xét ở mức đơn giản hơn. Cụ thể, trong bài toán này, thay vì xem xét việc tìm kiếm chuỗi nước đi phủ khắp bàn cờ, ta xem xét vấn đề đơn giản hơn là tìm kiếm nước đi tiếp theo của quân mã, hoặc kết luận rằng không còn nước đi kế tiếp thỏa mãn. Tại mỗi bước, nếu có thể tìm kiếm được 1 nước đi kế tiếp, ta tiến hành ghi lại nước đi này cùng với chuỗi các nước đi trước đó và tiếp tục quá trình tìm kiếm nước đi. Nếu tại bước nào đó, không thể tìm nước đi kế tiếp thỏa mãn yêu cầu của bài toán, ta quay trở lại bước trước, hủy bỏ nước đi đã lưu lại trước đó và thử sang 1 nước đi mới. Quá trình có thể phải thử rồi quay lại nhiều lần, cho tới khi tìm ra giải pháp hoặc đã thử hết các phương án mà không tìm ra giải pháp.

Quá trình trên có thể được mô tả bằng hàm sau:

```
void ThuNuocTiepTheo;
```

```
{
```

```
    Khởi tạo danh sách các nước đi kế tiếp;
```



```

do{
    Lựa chọn 1 nước đi kế tiếp từ danh sách; if Chấp
    nhận được
    {
        Ghi lại nước đi;
        if Bàn cờ còn ô trống
        {
            ThuNuocTiepTheo;
            if Nước đi không thành công
            Hủy bỏ nước đi đã lưu ở bước trước
        }
    }
}while (nước đi không thành công) && (vẫn còn nước đi)
}

```

Để thể hiện hàm 1 cách cụ thể hơn qua ngôn ngữ C, trước hết ta phải định nghĩa các cấu trúc dữ liệu và các biến dùng cho quá trình xử lý.

Đầu tiên, ta sử dụng 1 mảng 2 chiều để mô tả bàn cờ: `int Banco[n][n]`;

Các phần tử của mảng này có kiểu dữ liệu số nguyên. Mỗi phần tử của mảng đại diện cho 1 ô của bàn cờ. Chỉ số của phần tử tương ứng với tọa độ của ô, chẳng hạn phần tử `Banco[0][0]` tương ứng với ô (0,0) của bàn cờ. Giá trị của phần tử cho biết ô đó đã được quân mã đi qua hay chưa. Nếu giá trị ô = 0 tức là quân mã chưa đi qua, ngược lại ô đã được quân mã đi qua.

`Banco[x][y] = 0`: ô (x,y) chưa được quân mã đi qua `Banco[x][y] = i`: ô (x,y) đã được quân mã đi qua tại nước thứ i.

Tiếp theo, ta cần phải thiết lập thêm 1 số tham số. Để xác định danh sách các nước đi kế tiếp, ta cần chỉ ra tọa độ hiện tại của quân mã, từ đó theo luật cờ thông thường ta xác định các ô quân mã có thể đi tới. Như vậy, cần có 2 biến x, y để biểu thị tọa độ hiện tại của quân mã. Để cho biết nước đi có thành công hay không, ta cần dùng 1 biến kiểu boolean.

Nước đi kế tiếp chấp nhận được nếu nó chưa được quân mã đi qua, tức là nếu ô (u,v) được chọn là nước đi kế tiếp thì `Banco[u][v] = 0` là điều kiện để chấp nhận. Ngoài ra, hiển nhiên là ô đó phải nằm trong bàn cờ nên $0 \leq u, v < n$.

Việc ghi lại nước đi tức là đánh dấu rằng ô đó đã được quân mã đi qua. Tuy nhiên, ta cũng cần biết là quân mã đi qua ô đó tại nước đi thứ mấy. Như vậy, ta cần 1 biến i để cho biết hiện tại đang thử ở nước đi thứ mấy, và ghi lại nước đi thành công bằng cách gán giá trị `Banco[u][v]=i`.

Do i tăng lên theo từng bước thử, nên ta có thể kiểm tra xem bàn cờ còn ô trống không bằng cách kiểm tra xem i đã bằng n^2 chưa. Nếu $i < n^2$ tức là bàn cờ vẫn còn ô trống.

Để biết nước đi có thành công hay không, ta có thể kiểm tra biến boolean như đã nói ở trên. Khi nước đi không thành công, ta tiến hành hủy nước đi đã lưu ở bước trước bằng cách cho giá trị $Banco[u][v] = 0$.

Như vậy, ta có thể mô tả cụ thể hơn hàm ở trên như sau:

```
void ThuNuocTiepTheo(int i, int x, int y, int *q)
{
    int u, v, *q1;
    Khởi tạo danh sách các nước đi kế tiếp; do{
        *q1=0;
        Chọn nước đi (u,v) trong danh sách nước đi kế tiếp;
        if ((0 <= u) && (u<n) && (0 <= v) && (v<n) && (Banco[u][v]==0))
        {
            Banco[u][v]=i; if (i<n*n)
            {
                ThuNuocTiepTheo(i+1, u, v, q1) if (*q1==0) Banco[u][v]=0;
            } else *q1=1;
        }
    }while ((*q1==0) && (Vẫn còn nước đi))
    *q=*q1;
}
```

Trong đoạn chương trình trên vẫn còn 1 thao tác chưa được thể hiện bằng ngôn ngữ lập trình, đó là thao tác khởi tạo và chọn nước đi kế tiếp. Bây giờ, ta sẽ xem xét xem từ ô (x,y), quân mã có thể đi tới các ô nào, và cách tính vị trí tương đối của các ô đó so với ô (x,y) ra sao.

Theo luật cờ thông thường, quân mã từ ô (x,y) có thể đi tới 8 ô trên bàn cờ như trong hình vẽ:



	3		2	
4				1
5				8
	6		7	

x

Hình 2.4 Các nước đi của quân mã

Ta thấy rằng 8 ô mà quân mã có thể đi tới từ ô (x,y) có thể tính tương đối so với (x,y) là: (x+2, y-1); (x+1, y-2); (x-1, y-2); (x-2, y-1); (x-2, y+1); (x-1, y+2); (x+1, y+2); (x+2, y+1)

Nếu gọi dx, dy là các giá trị mà x, y lần lượt phải cộng vào để tạo thành ô mà quân mã có thể đi tới, thì ta có thể gán cho dx, dy mảng các giá trị như sau:

dx = { 2, 1, -1, -2, -2, -1, 1, 2 }

dy = { -1, -2, -2, -1, 1, 2, 2, 1 }

Như vậy, danh sách các nước đi kế tiếp (u, v) có thể được tạo ra như sau: u = x + dx[i]

v = y + dy[i] i = 1..8

Chú ý rằng, với các nước đi như trên thì (u, v) có thể là ô nằm ngoài bàn cờ. Tuy nhiên, như đã nói ở trên, ta đã có điều kiện $0 \leq u, v < n$, do vậy luôn đảm bảo ô (u, v) được chọn là hợp lệ.

Cuối cùng, hàm *ThuNuocTiepTheo* có thể được viết lại hoàn toàn bằng ngôn ngữ C như

```
s
a      void ThuNuocTiepTheo(int i, int x, int y, int*q)
u      {
:
      int k, u, v, *q1; k=0;
      do{
      *q1=0; u=x+dx[k]; v=y+dy[k];
      if ((0 <= u) && (u<n) && (0 <= v) && (v<n) && (Banco[u][v]==0))
      {
      Banco[u][v]=i; if (i<n*n)
      {
      ThuNuocTiepTheo(i+1, u, v, q1) if (*q1==0)
      Banco[u][v]=0;
      } else *q1=1;
      }
      k=k+1;
```

```

}while ((*q1==0) && (k<8));
*q=*q1;
}

```

Như vậy, có thể thấy đặc điểm của thuật toán là giải pháp cho toàn bộ vấn đề được thực hiện dần từng bước, và tại mỗi bước có ghi lại kết quả để sau này có thể quay lại và hủy kết quả đó nếu phát hiện ra rằng hướng giải quyết theo bước đó đi vào ngõ cụt và không đem lại giải pháp tổng thể cho vấn đề. Do đó, thuật toán được gọi là *thuật toán quay lui*.

Dưới đây là mã nguồn của toàn bộ chương trình Mã đi tuần viết bằng ngôn ngữ C:

```

#include<stdio.h>
> #include<conio.h>
#define maxn 10

void ThuNuocTiepTheo(int i, int x, int y,
int *q); void InBanco(int n);
void XoaBanco(int n);

int Banco[maxn][maxn];
intdx[8]={2,1,-1,-2,-2,-1,1,2};
intdy[8]={-1,-2,-2,-1,1,2,2,1};
int n=8;

void ThuNuocTiepTheo(int i, int x, int y, int *q)
{
    int k, u, v, *q1;
k=0;
    do{
        *q1=0;      u=x+dx[k];
v=y+dy[k];
        if          ((0 <= u) && (u<n) && (0 <= v) && (v<n)
&& (Banco[u][v]==0))
        {
            Banco[u][v]=i; if (i<n*n)
            {

```

```
} k++;
        ThuNuocTiepTheo(i+1, u, v,
q1); if ((*q1)==0)
Banco[u][v]=0;
        }else (*q1)=1;
```

```

}while (((*q1)==0) && (k<8));
    *q=*q1;
}

void InBanco(int n){
    int i, j;
    for (i=0;i<=n-1;i++){
        for (j=0;j<=n-1;j++){
            if(Banco[i][j]<10)printf("%d  ",Banco[i][j]);
elseprintf("%d          ",Banco[i][j]);

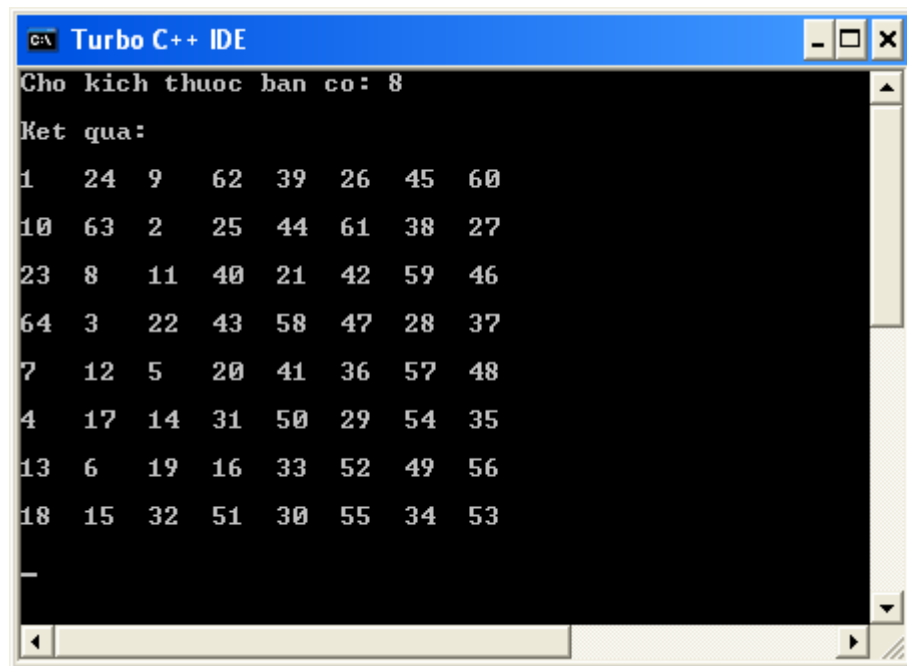
        printf("\n\n");
    }
}

void XoaBanco(int n){
    int i, j;
    for (i=0;i<=n-1;i++){
        for (j=0;j<=n-1;j++) Banco[i][j]=0;
    }
}

void main(){
    int      *q=0;
clrscr();
    printf("Cho kich thuoc ban co:
"); scanf(" %d",&n);
    XoaBanco(n);
Banco[0][0]=1;
    ThuNuocTiepTheo(2,0,0,q);
printf("\n Ket qua:  \n\n");
InBanco(n);
    getch();
return;
}

```

Và kết quả chạy chương trình với bàn cờ 8x8 và ô bắt đầu là ô (0,0):



Hình 2.5 Kết quả chạy chương trình mã đi tuần

2.4.2. Xây dựng và cài đặt thuật toán đệ qui cho một số bài toán thực tế.

Bài toán 8 quân hậu

Bài toán 8 quân hậu là 1 ví dụ rất nổi tiếng về việc sử dụng phương pháp thử - sai và thuật toán quay lui. Đặc điểm của các bài toán dạng này là không thể dùng các biện pháp phân tích để giải được mà phải cần đến các phương pháp tính toán thủ công, với sự kiên trì và độ chính xác cao. Do đó, các thuật toán kiểu này phù hợp với việc sử dụng máy tính vì máy tính có khả năng tính toán nhanh và chính xác hơn nhiều so với con người.

Bài toán 8 quân hậu được phát biểu ngắn gọn như sau: Tìm cách đặt 8 quân hậu trên 1 bàn cờ sao cho không có 2 quân hậu nào có thể ăn được nhau.

Tương tự như phân tích ở bài Mã đi tuần, ta có hàm `DatHau` để tìm vị trí đặt quân hậu tiếp theo như sau:

```

void DatHau(int i)
{
    Khởi tạo danh sách các vị trí có thể đặt quân hậu tiếp theo;
do{
    Lựa chọn vị trí đặt quân hậu tiếp theo; if Vị trí đặt
là an toàn
    {

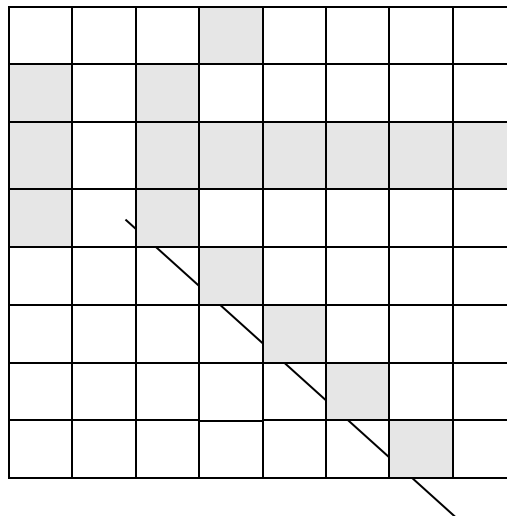
```

```

Đặt hậu; if  $i < 8$ 
{
  DatHau( $i+1$ );
  if Không thành công
  Bỏ hậu đã đặt ra khỏi vị trí
}
}
}while (Không thành công) && (Vẫn còn lựa chọn)
}

```

Tiếp theo, ta xem xét các cấu trúc dữ liệu và biến sẽ được dùng để thực hiện các công việc trong hàm. Theo luật cờ thông thường thì quân hậu có thể ăn tất cả các quân nằm trên cùng hàng, cùng cột, hoặc đường chéo. Do vậy, ta có thể suy ra rằng mỗi cột của bàn cờ chỉ có thể chứa 1 và chỉ 1 quân hậu, và từ đó ta có quy định là quân hậu thứ i phải đặt ở cột thứ i . Như vậy, ta sẽ dùng biến i để biểu thị chỉ số cột, và quá trình lựa chọn vị trí đặt quân hậu sẽ chọn 1 trong 8 vị trí trong cột cho biến chỉ số hàng j .



Hình 2.6 Các nước chiếu của quân hậu

Trong bài toán Mã đi tuần, ta sử dụng một mảng 2 chiều $Banco(i, j)$ để biểu thị bàn cờ. Tuy nhiên, trong bài toán này nếu tiếp tục dùng cấu trúc dữ liệu đó sẽ dẫn tới một số phức tạp trong việc kiểm tra vị trí đặt quân hậu có an toàn hay không, bởi vì ta cần phải kiểm tra hàng và các đường chéo đi qua ô quân hậu sẽ được đặt (không cần kiểm tra cột vì theo quy định ban đầu, có đúng 1 quân hậu được đặt trên mỗi cột). Đối với mỗi ô trong cột, sẽ có 1 hàng và 2 đường chéo đi qua nó là đường chéo trái và đường chéo phải.

Ta sẽ dùng 3 mảng kiểu boolean để biểu thị cho các hàng, các đường chéo trái, và các

đường chéo phải (có tất cả 15 đường chéo trái và 15 đường chéo phải). `int a[8];`

`int b[15],`

`c[15];` Trong đó:

`a[j] = 0;` Hàng j chưa bị chiếm bởi quân hậu nào.

`b[k] = 0;` Đường chéo trái k chưa bị chiếm bởi quân hậu nào. `c[k] = 0;` Đường chéo phải k chưa bị chiếm bởi quân hậu nào.

Chú ý rằng các ô (i, j) cùng nằm trên 1 đường chéo trái thì có cùng giá trị $i + j$, và cùng nằm trên đường chéo phải thì có cùng giá trị $i - j$. Nếu đánh số các đường chéo trái và phải từ 0 đến 14, thì ô (i, j) sẽ nằm trên đường chéo trái $(i + j)$ và nằm trên đường chéo phải $(i - j + 7)$.

Do vậy, để kiểm tra xem ô (i, j) có an toàn không, ta chỉ cần kiểm tra xem hàng j và các

đường chéo $(i + j)$, $(i - j + 7)$ đã bị chiếm chưa, tức là kiểm tra `a[i]`, `b[i + j]`, và `c[i - j + 7]`.

Ngoài ra, ta cần có 1 mảng `x` để lưu giữ chỉ số hàng của quân hậu trong cột i . `int x[8];`

Với thao tác đặt hậu vào vị trí hàng j trên cột i , ta cần thực hiện các công việc: `x[i] = j; a[j] = 1; b[i + j] = 1; c[i - j + 7] = 1;`

Với thao tác bỏ hậu ra khỏi hàng j trong cột i , ta cần thực hiện các công việc: `a[j] = 0; b[i + j] = 0; c[i - j + 7] = 0;`

Còn điều kiện để kiểm tra xem vị trí tại hàng j trong cột i có an toàn không là: `(a[j] == 0) && (b[i + j] == 0) && (c[i - j + 7] == 0)`

Như vậy, hàm `DatHau` sẽ được thể hiện cụ thể bằng ngôn ngữ C như sau:

```
void DatHau(int i, int *q)
```

```
{
```

```
int j; j=0;
```

```
do{
```

```
    *q=0;
```

```
    if ((a[j] == 0) && (b[i + j] == 0) && (c[i - j + 7] == 0))
```

```

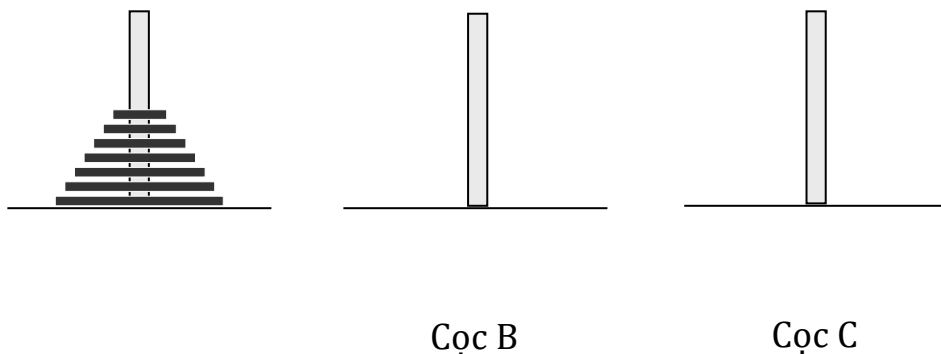
{
x[i] = j;
a[j] = 1; b[i + j] = 1; c[i - j + 7] = 1;
if (i < 7)
{
DatHau(i + 1, q); if ((*q) == 0)
{
a[j] = 0; b[i + j] = 0; c[i - j + 7] = 0;
}
} else
(*q) = 1;
}
j++;
}while ((*q) == 0 && (j < 8))
}

```

2.4.3. Xây dựng , cài đặt thuật toán qui hoạch động cho một số bài toán thực tế.

Bài toán tháp Hà nội

Có 3 chiếc cọc và một bộ n chiếc đĩa. Các đĩa này có kích thước khác nhau và mỗi đĩa đều có 1 lỗ ở giữa để có thể xuyên chúng vào các cọc. Ban đầu, tất cả các đĩa đều nằm trên 1 cọc, trong đó, đĩa nhỏ hơn bao giờ cũng nằm trên đĩa lớn hơn.



Hình 2.2 Bài toán tháp Hà nội

Yêu cầu của bài toán là chuyển bộ n đĩa từ cọc ban đầu A sang cọc đích C (có thể sử dụng cọc trung gian B), với các điều kiện:

- Mỗi lần chuyển 1 đĩa.
- Trong mọi trường hợp, đĩa có kích thước nhỏ hơn bao giờ cũng phải nằm trên đĩa có kích thước lớn hơn.

Với $n=1$, có thể thực hiện yêu cầu bài toán bằng cách chuyển trực tiếp đĩa 1 từ cọc A sang cọc C.

Với $n=2$, có thể thực hiện như sau:

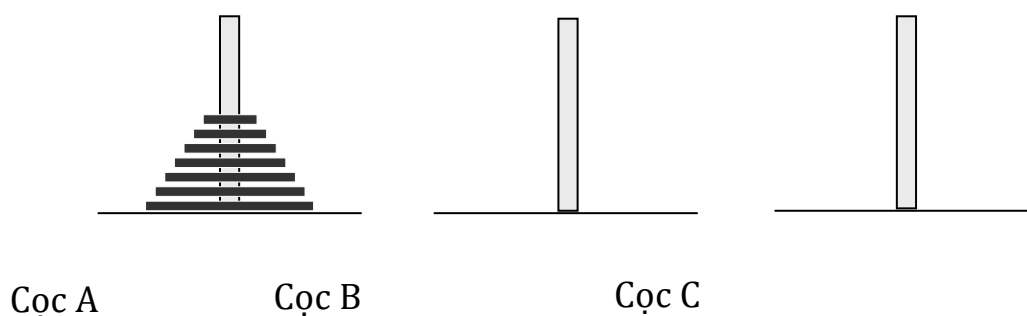
- Chuyển đĩa nhỏ từ cọc A sang cọc trung gian B.
- Chuyển đĩa lớn từ cọc A sang cọc đích C.
- Cuối cùng, chuyển đĩa nhỏ từ cọc trung gian B sang cọc đích C.

Như vậy, cả 2 đĩa đã được chuyển sang cọc đích C và không có tình huống nào đĩa lớn nằm trên đĩa nhỏ.

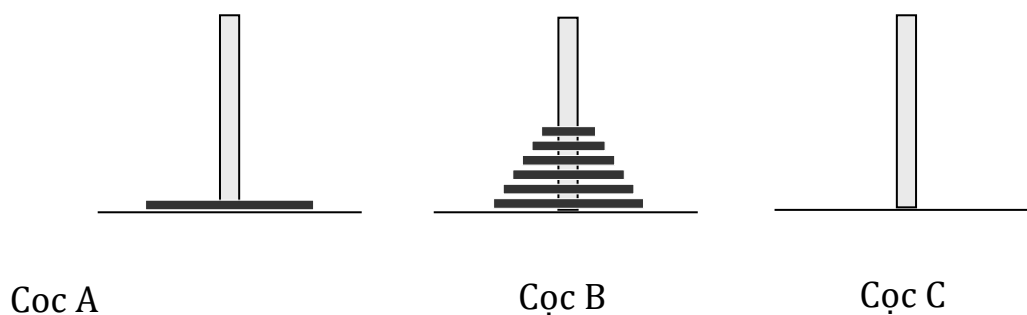
Với $n > 2$, giả sử ta đã có cách chuyển $n-1$ đĩa, ta thực hiện như sau:

- Lấy cọc đích C làm cọc trung gian để chuyển $n-1$ đĩa bên trên sang cọc trung gian B.
- Chuyển cọc dưới cùng (cọc thứ n) sang cọc đích C.
- Lấy cọc ban đầu A làm cọc trung gian để chuyển $n-1$ đĩa từ cọc trung gian B sang cọc đích C.

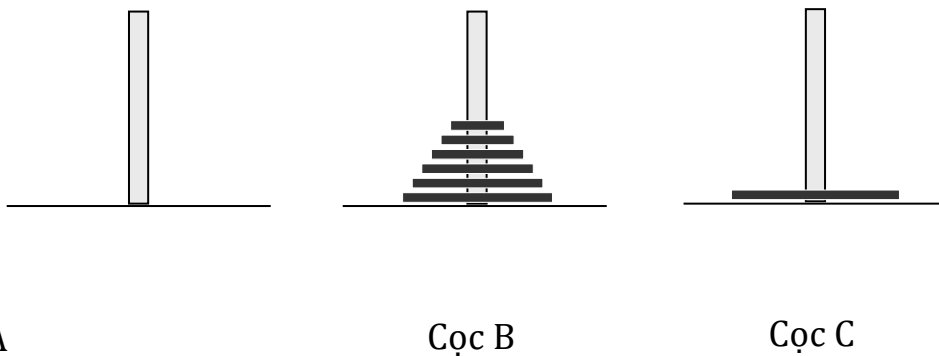
Có thể minh họa quá trình chuyển này như sau: Trạng thái ban đầu:



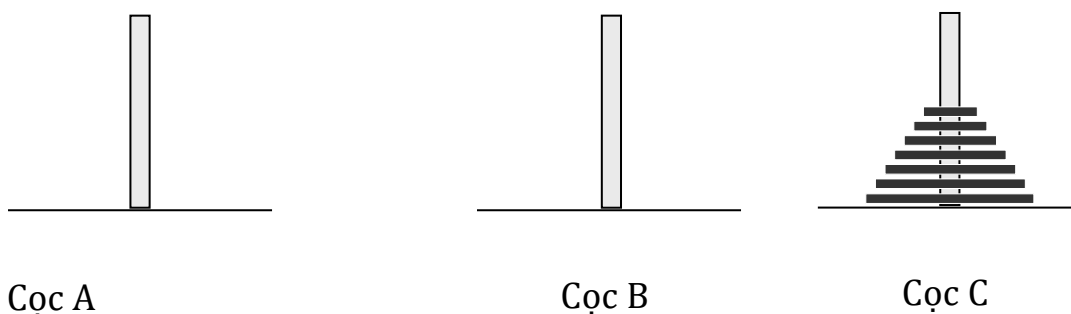
Bước 1: Chuyển $n-1$ đĩa bên trên từ cọc A sang cọc B, sử dụng cọc C làm cọc trung gian.



Bước 2: Chuyển đĩa dưới cùng từ cọc A thẳng sang cọc C.



Bước 3: Chuyển n-1 đĩa từ cột B sang cột C sử dụng cột A làm cột trung gian.



Như vậy, ta thấy toàn bộ n đĩa đã được chuyển từ cột A sang cột C và không vi phạm bất cứ điều kiện nào của bài toán.

Ở đây, ta thấy rằng bài toán chuyển n cột đã được chuyển về bài toán đơn giản hơn là chuyển n-1 cột. Điểm dừng của thuật toán đệ qui là khi $n=1$ và ta chuyển thẳng cột này từ cột ban đầu sang cột đích.

Tính chất chia để trị của thuật toán này thể hiện ở chỗ: Bài toán chuyển n đĩa được chia làm 2 bài toán nhỏ hơn là chuyển n-1 đĩa. Lần thứ nhất chuyển n-1 đĩa từ cột a sang cột trung gian b, và lần thứ 2 chuyển n-1 đĩa từ cột trung gian b sang cột đích c.

Cài đặt đệ qui cho thuật toán như sau:

- Hàm `chuyen(int n, int a, int c)` thực hiện việc chuyển đĩa thứ n từ cột a sang cột c.
- Hàm `thaphanoi(int n, int a, int c, int b)` là hàm đệ qui thực hiện việc chuyển n đĩa từ cột a sang cột c, sử dụng cột trung gian là cột b.

Chương trình như sau:

```
void chuyen(int n, char a, char c){
printf('Chuyen          dia  thu  %d tu  coc  %c sang  coc
                                     %c
```

```

\n",n,a,c);
return;
}
void thaphanoi(int n, char a, char c, char b){ if (n==1) chuyen(1, a, c);
else{
thaphanoi(n-1, a, b, c);
chuyen(n, a, c);
thaphanoi(n-1, b, c,a);
}
return;
}

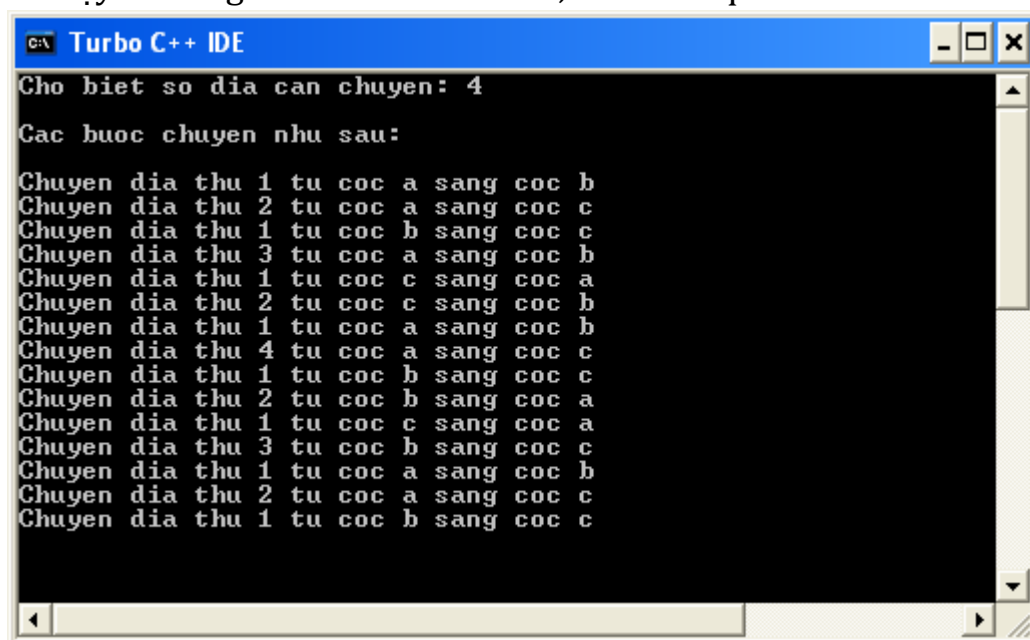
```

Hàm chuyen thực hiện thao tác in ra 1 dòng cho biết chuyển đĩa thứ mấy từ cọc nào sang cọc nào.

Hàm thaphanoi kiểm tra nếu số đĩa bằng 1 thì thực hiện chuyển trực tiếp đĩa từ cọc a sang cọc c. Nếu số đĩa lớn hơn 1, có 3 lệnh được thực hiện:

- 1- Lệnh gọi đệ qui thaphanoi(n-1, a, b, c) để chuyển n-1 đĩa từ cọc a sang cọc b, sử dụng cọc c làm cọc trung gian.
- 2- Thực hiện chuyển đĩa thứ n từ cọc a sang cọc c.
- 3- Lệnh gọi đệ qui thaphanoi(n-1, b, c, a) để chuyển n-1 đĩa từ cọc b sang cọc c, sử dụng cọc a làm cọc trung gian.

Khi chạy chương trình với số đĩa là 4, ta có kết quả như sau:



```

Turbo C++ IDE
Cho biet so dia can chuyen: 4
Gac buoc chuyen nhu sau:
Chuyen dia thu 1 tu coc a sang coc b
Chuyen dia thu 2 tu coc a sang coc c
Chuyen dia thu 1 tu coc b sang coc c
Chuyen dia thu 3 tu coc a sang coc b
Chuyen dia thu 1 tu coc c sang coc a
Chuyen dia thu 2 tu coc c sang coc b
Chuyen dia thu 1 tu coc a sang coc b
Chuyen dia thu 4 tu coc a sang coc c
Chuyen dia thu 1 tu coc b sang coc c
Chuyen dia thu 2 tu coc b sang coc a
Chuyen dia thu 1 tu coc c sang coc a
Chuyen dia thu 3 tu coc b sang coc c
Chuyen dia thu 1 tu coc a sang coc b
Chuyen dia thu 2 tu coc a sang coc c
Chuyen dia thu 1 tu coc b sang coc c

```

Hình 2.3 Kết quả chạy chương trình tháp Hà nội với 4 đĩa

Độ phức tạp của thuật toán là $2^n - 1$. Nghĩa là để chuyển n cọc thì mất $2^n - 1$ thao tác chuyển.

Ta sẽ chứng minh điều này bằng phương pháp qui nạp toán học: Với $n=1$ thì số lần chuyển là $1 = 2^1 - 1$.

Giả sử giả thiết đúng với $n-1$, tức là để chuyển $n-1$ đĩa cần thực hiện $2^{n-1} - 1$ thao tác chuyển.

Ta sẽ chứng minh rằng để chuyển n đĩa cần $2^n - 1$ thao tác chuyển.

Thật vậy, theo phương pháp chuyển của giải thuật thì có 3 bước. Bước 1 chuyển $n-1$ đĩa từ cọc a sang cọc b mất $2^{n-1} - 1$ thao tác. Bước 2 chuyển 1 đĩa từ cọc a sang cọc c mất 1 thao tác. Bước 3 chuyển $n-1$ đĩa từ cọc b sang cọc c mất $2^{n-1} - 1$ thao tác. Tổng cộng ta mất $(2^{n-1} - 1) + (2^{n-1} - 1) + 1 = 2 * 2^{n-1} - 1 = 2^n - 1$ thao tác chuyển. Đó là điều cần chứng minh.

Như vậy, thuật toán có cấp độ tăng rất lớn. Nói về cấp độ tăng này, có một truyền thuyết vui về bài toán tháp Hà nội như sau: Ngày tận thế sẽ đến khi các nhà sư ở một ngôi chùa thực hiện xong việc chuyển 40 chiếc đĩa theo quy tắc như bài toán vừa trình bày. Với độ phức tạp của bài toán vừa tính được, nếu giả sử mỗi lần chuyển 1 đĩa từ cọc này sang cọc khác mất 1 giây thì với $2^{40} - 1$ lần chuyển, các nhà sư này phải mất ít nhất 34.800 năm thì mới có thể chuyển xong toàn bộ số đĩa này !

Dưới đây là toàn bộ mã nguồn chương trình tháp Hà nội viết bằng C:

```
#include<stdio.h> #include<conio.h>
void chuyen(int n, char a, char c);
void thaphanoi(int n, char a, char c, char b);
void chuyen(int n, char a, char c){
printf("Chuyen dia thu %d tu coc %c sang coc %c \n", n, a,c);
return;
}
void thaphanoi(int n, char a, char c, char b){ if (n==1) chuyen(1, a, c);
else{
thaphanoi(n-1, a, b, c);
chuyen(n, a, c);
thaphanoi(n-1, b, c,a);
}
return;
}
void main(){
```

```
int sodia; clrscr();  
printf("Cho biet so dia can chuyen: "); scanf("%d",&sodia);  
printf("\nCac buoc chuyen nhu sau:\n\n"); thaphanoi(sodia, 'a', 'c', 'b');  
getch(); return;  
}
```

Chương 3. Ngăn xếp, Hàng đợi, Danh sách liên kết

Chương 3 trình bày về hai cấu trúc dữ liệu rất gần gũi với các hoạt động trong thực tế, đó là ngăn xếp, hàng đợi và danh sách liên kết

Phần 1 trình bày các khái niệm, định nghĩa liên quan đến ngăn xếp, khai báo ngăn xếp bằng mảng và các thao tác cơ bản như kiểm tra ngăn xếp rỗng, đưa phần tử vào ngăn xếp, lấy phần tử ra khỏi ngăn xếp. Một cách cài đặt ngăn xếp khác cũng được giới thiệu, đó là dùng danh sách liên kết. Việc sử dụng danh sách liên kết để cài đặt sẽ cho một ngăn xếp có kích thước linh hoạt hơn.

Phần 2 trình bày về hàng đợi. Tương tự như phần 1, các khái niệm, các cách cài đặt và các thao tác cơ bản trên ngăn xếp cũng được trình bày chi tiết.

Phần 3 trình bày về danh sách liên kết, tương tự như phần 1 và 2.

Để học tốt chương 3, sinh viên cần có liên hệ với các hoạt động thực tế để hình dung về ngăn xếp và hàng đợi. Nắm vững cách cài đặt và các thao tác trên 3 kiểu dữ liệu này. Tự đặt ra các bài toán ứng dụng thực tế để thực hiện.

3.1. Ngăn xếp

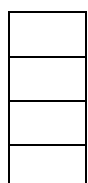
3.1.1. Định nghĩa ngăn xếp

Ngăn xếp là một dạng đặc biệt của danh sách mà việc bổ sung hay loại bỏ một phần tử đều được thực hiện ở 1 đầu của danh sách gọi là đỉnh. Nói cách khác, ngăn xếp là 1 cấu trúc dữ liệu có 2 thao tác cơ bản: bổ sung (push) và loại bỏ phần tử (pop), trong đó việc loại bỏ sẽ tiến hành loại phần tử mới nhất được đưa vào danh sách. Chính vì tính chất này mà ngăn xếp còn được gọi là kiểu dữ liệu có nguyên tắc LIFO (Last In First Out - Vào sau ra trước).

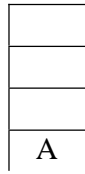
Các ví dụ về lưu trữ kiểu LIFO như của ngăn xếp là: Một chồng sách trên mặt bàn, một chồng đĩa trong hộp, v.v. Khi thêm 1 cuốn sách vào chồng sách, cuốn sách sẽ nằm ở trên đỉnh của chồng sách. Khi lấy sách ra khỏi chồng sách, cuốn nằm trên cùng sẽ được lấy ra đầu tiên, tức là cuốn mới nhất được đưa vào sẽ được lấy ra trước tiên. Tương tự như vậy với chồng đĩa trong hộp.

Ta xét 1 ví dụ minh họa sự thay đổi của ngăn xếp thông qua các thao tác bổ sung và loại bỏ đỉnh trong ngăn xếp.

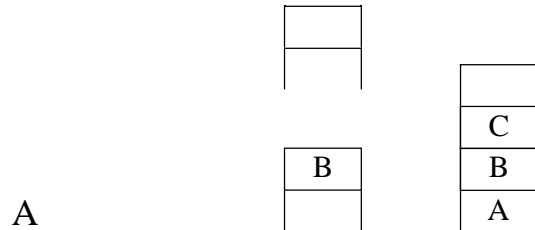
Giả sử ta có một stack S lưu trữ các ký tự. Ban đầu, ngăn xếp ở trạng thái rỗng:



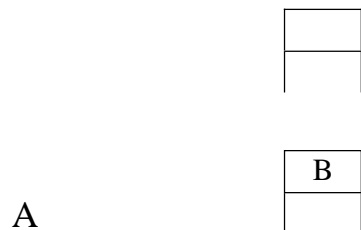
Khi thực hiện lệnh bỏ xung phần tử A, $\text{push}(S, A)$, ngăn xếp có dạng:



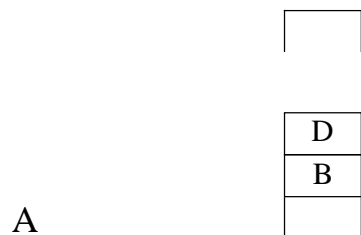
Tiếp theo là các lệnh $\text{push}(S, B)$, $\text{push}(S, C)$:



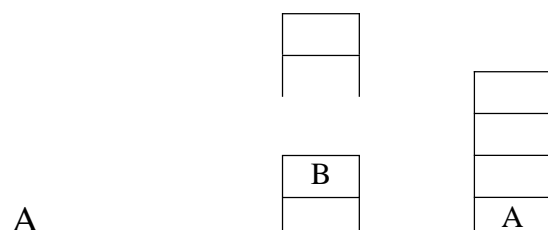
Lệnh $\text{pop}(S)$ sẽ loại bỏ phần tử mới nhất được đưa vào ra khỏi ngăn xếp, đó là C:



Lệnh $\text{push}(S, D)$ sẽ đưa phần tử D vào ngăn xếp, ngay trên phần tử B:



Hai lệnh $\text{pop}(S)$ tiếp theo sẽ lần lượt loại bỏ các phần tử nằm trên là D và B ra khỏi ngăn xếp:

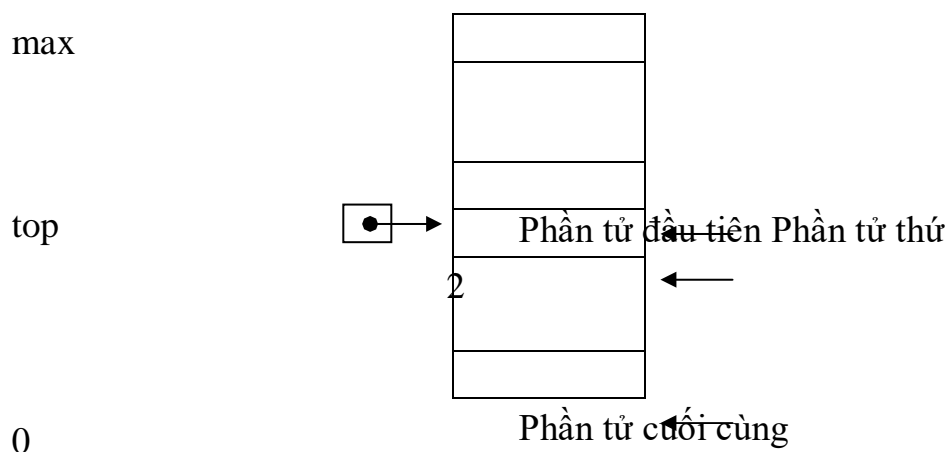


3.1.2. Biểu diễn ngăn xếp

3.1.3. Các thao tác trên ngăn xếp

Cài đặt ngăn xếp bằng mảng

Ngăn xếp có thể được cài đặt bằng mảng hoặc danh sách liên kết (sẽ được trình bày ở phần sau). Để cài đặt ngăn xếp bằng mảng, ta sử dụng một mảng 1 chiều s để biểu diễn ngăn xếp. Thiết lập phần tử đầu tiên của mảng, $s[0]$, làm đáy ngăn xếp. Các phần tử tiếp theo được đưa vào ngăn xếp sẽ lần lượt được lưu tại các vị trí $s[1]$, $s[2]$, ... Nếu hiện tại ngăn xếp có n phần tử thì $s[n-1]$ sẽ là phần tử mới nhất được đưa vào ngăn xếp. Để lưu giữ đỉnh hiện tại của ngăn xếp, ta sử dụng 1 con trỏ top . Chẳng hạn, nếu ngăn xếp có n phần tử thì top sẽ có giá trị bằng $n-1$. Còn khi ngăn xếp chưa có phần tử nào thì ta quy ước top sẽ có giá trị -1 .



Hình 4.1 Cài đặt ngăn xếp bằng mảng

Nếu có 1 phần tử mới được đưa vào ngăn xếp thì nó sẽ được lưu tại vị trí kế tiếp trong mảng và giá trị của biến top tăng lên 1. Khi lấy 1 phần tử ra khỏi ngăn xếp, phần tử của mảng tại vị trí top sẽ được lấy ra và biến top giảm đi 1.

Có 2 vấn đề xảy ra khi thực hiện các thao tác trên trong ngăn xếp. Khi ngăn xếp đã đầy, tức là khi biến top đạt tới phần tử cuối cùng của mảng thì không thể tiếp tục thêm phần tử mới vào mảng. Và khi ngăn xếp rỗng, tức là chưa có phần tử nào, thì ta không thể lấy được phần tử ra từ ngăn xếp. Như vậy, ngoài các thao tác đưa vào và lấy phần tử ra khỏi ngăn xếp, cần có thao tác kiểm tra xem ngăn xếp có rỗng hoặc đầy hay không.

Khai báo bằng mảng cho 1 ngăn xếp chứa các số nguyên với tối đa 100 phần tử

như sau:

```
#define MAX      100
typedef struct    {
    int           top;
    int           nut[MAX];
} stack;
```

Khi đó, các thao tác trên ngăn xếp được cài đặt như sau:

Thao tác khởi tạo ngăn xếp

Thao tác này thực hiện việc gán giá trị -1 cho biến top, cho biết ngăn xếp đang ở trạng thái rỗng

```
void StackInitialize(stack *s){
    s-> top = -1;
    return;
}
```

Thao tác kiểm tra ngăn xếp rỗng

```
int StackEmpty(stack s){
    return (s.top == -1);
}
```

Thao tác kiểm tra ngăn xếp đầy

```
int StackFull(stacks){
    return (s.top == MAX-1);
}
```

Thao tác bổ sung 1 phần tử vào ngăn xếp

```
void Push(stack *s, int x){
if(StackFull(*s)){
    printf("Ngan xep day !"); return;
}else{
    s-> top ++;
    s-> nut[s-> top] = x; return;
}
}
```

Thao tác lấy 1 phần tử ra khỏi ngăn xếp

```
int Pop(stack*s){
    if (StackEmpty(*s)){ printf("Ngan xep rong
!");
    }else{
        return s-> nut[s-> top--];
    }
}
```

Hạn chế của việc cài đặt ngăn xếp bằng mảng, cũng tương tự như cấu trúc dữ liệu kiểu mảng, là ta cần phải biết trước kích thước tối đa của ngăn xếp (giá trị max trong khai báo ở trên). Điều này không phải lúc nào cũng xác định được và nếu ta chọn một giá trị bất kỳ thì có thể dẫn đến lãng phí bộ nhớ nếu kích thước quá thừa so với yêu cầu hoặc nếu thiếu thì sẽ dẫn tới chương trình có thể không hoạt động được. Để khắc phục nhược điểm này, có thể sử dụng danh sách liên kết để cài đặt ngăn xếp.

3.1.4. Ứng dụng của ngăn xếp

Một số ví dụ về ứng dụng của ngăn xếp được xem xét trong phần này bao gồm:

- Đảo ngược chuỗi ký tự.
- Tính giá trị một biểu thức dạng hậu tố(postfix).
- Chuyển một biểu thức dạng trung tố sang hậu tố (infix to postfix).

Trong các ví dụ này, ta giả sử rằng đã có một ngăn xếp với các hàm thao tác được cài đặt như ở phần trước (bảng mảng hoặc danh sách).

Đảo ngược chuỗi ký tự

Bài toán đảo ngược chuỗi ký tự yêu cầu hiển thị các ký tự của 1 chuỗi ký tự theo chiều ngược lại. Tức là ký tự cuối cùng của chuỗi sẽ được hiển thị trước, tiếp theo là ký tự sát ký tự cuối, ..., và ký tự đầu tiên sẽ được hiển thị cuối cùng.

Ví dụ:

Chuỗi ban đầu: stack

Chuỗi đảo ngược: kcats

Đây là 1 ứng dụng khá đơn giản và hiệu quả của ngăn xếp, do yêu cầu của bài toán cũng khá phù hợp với tính chất của ngăn xếp.

Để giải quyết bài toán, ta chỉ cần duyệt từ đầu đến cuối chuỗi, lần lượt cho các ký tự vào ngăn xếp. Khi đó, ký tự đầu tiên của chuỗi sẽ được cho vào trước, tiếp theo đến ký tự thứ 2, ..., ký tự cuối được cho vào sau cùng. Sau khi đã cho toàn bộ ký tự của chuỗi vào ngăn xếp, lần lượt lấy các phần tử ra khỏi ngăn xếp và hiển thị trên màn hình. Theo tính chất của ngăn xếp, ký tự cho vào sau cùng sẽ được lấy ra trước tiên. Do đó, ký tự cuối cùng của chuỗi sẽ được lấy ra đầu tiên, ..., và ký tự đầu tiên của chuỗi sẽ được lấy ra sau cùng. Như vậy, toàn bộ các ký tự trong chuỗi đã được đảo ngược thứ tự.

Mã chương trình đảo ngược chuỗi ký tự

như sau: #include<stdio.h>

#include<conio.h>

```
struct node {  
char item;  
struct node *next;  
};
```

```
typedef struct node
```

```
*stacknode; typedef struct {
```

```

    stacknode top;
}stack;

void StackInitialize(stack
*s){ s-> top = NULL;
    return;
}

int StackEmpty(stack
s){ return (s.top
==NULL);
}

void Push(stack *s,
char c){ stacknodep;
    p = (stacknode) malloc (sizeof(struct
node)); p-> item = c;
    p-> next = s-
>top; s->top = p;
return;
}

char Pop(stack
*s){ stacknodep;
    p = s->top;
    s-> top = s-> top->
next; return p->item;
}

void main
(void){ char*st;
    int i;
stack *s;
clrscr();
    StackInitialize(s);

```

```

printf("Nhap vao xau ky tu:
"); gets(st);
for
(i=0;i<strlen(st);i++)
Push(s,st[i]);
printf("\Xau da dao nguoc: \n");
while (!StackEmpty(*s))
printf("%c",Pop(s)); getch();
return;
}

```

Tính giá trị của biểu thức dạng hậu tố

Một biểu thức toán học thông thường bao gồm các toán tử (cộng, trừ, nhân, chia ...), các toán hạng (các số), và các dấu ngoặc để cho biết thứ tự tính toán. Chẳng hạn, ta có thể có biểu thức toán học sau:

$$3 * ((5 - 2) * (7 + 1) - 6)$$

Như ta thấy, trong biểu thức trên, các toán tử bao giờ cũng nằm giữa 2 toán hạng. Do vậy, các viết trên được gọi là các viết dạng trung tố (infix). Để tính giá trị của biểu thức trên, ta phải tính giá trị của các phép toán trong ngoặc trước. Đôi khi, ta cần lưu các kết quả tính được này như một kết quả trung gian, sau đó lại sử dụng chúng như những toán hạng tiếp theo. Ví dụ, để tính giá trị biểu thức trên, đầu tiên ta tính $5 - 2 = 3$, lưu kết quả này. Tiếp theo tính $7 + 1 = 8$. Lấy kết quả này nhân với kết quả đã lưu là 3 được 24. Lấy $24 - 6 = 18$, và cuối cùng $18 \times 3 = 54$ là kết quả cuối cùng của biểu thức.

Trong các biểu thức dạng này, vị trí của dấu ngoặc là rất quan trọng. Nếu vị trí các dấu ngoặc thay đổi, giá trị của cả biểu thức có thể thay đổi theo.

Mặc dù đối với con người, cách trình bày biểu thức toán học theo dạng này có vẻ như là hợp lý nhất, nhưng đối với máy tính, việc tính toán những biểu thức như vậy tương đối phức tạp. Để dễ dàng hơn cho máy tính trong việc tính toán các biểu thức, người ta đưa ra một cách trình bày khác cho biểu thức toán học, đó là dạng hậu tố (postfix). Theo cách trình bày này, toán tử không nằm ở giữa 2 toán hạng mà nằm ngay phía sau 2 toán hạng. Chẳng hạn, biểu thức trên có thể được viết dưới dạng hậu tố như sau:

$$3 \ 5 \ 2 \ - \ 7 \ 1 \ + \ * \ 6 \ - \ *$$

Ta tính giá trị biểu thức viết dưới dạng này như sau:

Toán tử trừ nằm ngay sau 2 toán hạng 5 và 2 nên lấy $5 - 2 = 3$, lưu kết quả 3. Toán tử cộng nằm ngay sau 2 toán hạng 7 và 1 nên lấy $7 + 1 = 8$, lưu kết quả 8.

Toán tử nhân nằm ngay sau 2 kết quả vừa lưu nên lấy $3 \times 8 = 24$, lưu kết quả 24. Toán tử trừ nằm ngay sau toán hạng 6 và kết quả vừa lưu nên lấy $24 - 6 = 18$. Toán tử nhân nằm ngay sau kết quả vừa lưu và toán hạng 3 nên lấy $3 \times 18 = 54$ là kết quả cuối cùng của biểu thức.

Như ta thấy, biểu thức dạng hậu tố không cần dùng bất kỳ dấu ngoặc nào. Cách tính giá trị của biểu thức dạng này cần đến 1 số bước lưu kết quả trung gian để khi gặp toán tử lại lấy ra để tính toán tiếp, do vậy rất phù hợp với việc sử dụng ngăn xếp.

Thuật toán để tính giá trị của biểu thức hậu tố bằng cách sử dụng ngăn xếp như sau: Duyệt biểu thức từ trái qua phải.

- Nếu gặp toán hạng, đưa vào ngăn xếp.
- Nếu gặp toán tử, lấy ra 2 toán tử từ ngăn xếp, sử dụng toán hạng trên để tính, đưa kết quả vào ngăn xếp.

Chẳng hạn với biểu thức dạng hậu tố ở trên, các bước tính như sau:

Duyệt từ trái sang phải, gặp các toán hạng 3, 5, 2, lần lượt đưa vào ngăn xếp.

2
5
3

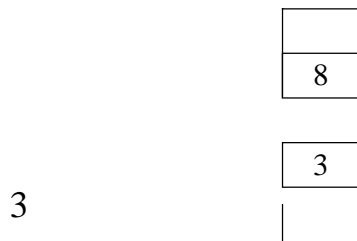
Duyệt tiếp, gặp toán tử trừ. Lấy ra 2 toán hạng từ ngăn xếp là 2 và 5, thực hiện phép trừ được kết quả 3 đưa vào ngăn xếp.

3

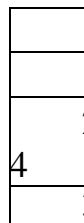
Duyệt tiếp, gặp 2 toán hạng 7, 1 lần lượt đưa vào ngăn xếp.

1
7
3

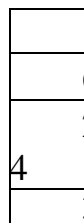
Duyệt tiếp, gấp toán tử cộng. Lấy 2 toán hạng trong ngăn xếp là 1 và 7, thực hiện phép cộng được kết quả 8. Đưa vào ngăn xếp.



Duyệt tiếp, gấp toán tử nhân. Lấy 2 toán hạng trong ngăn xếp là 8 và 3. Thực hiện phép cộng, được kết quả 24, cho vào ngăn xếp.



Duyệt tiếp, gấp toán hạng 6, cho vào ngăn xếp.



Duyệt tiếp, gấp toán tử trừ. Lấy ra 2 toán hạng trong ngăn xếp là 6 và 24. Thực hiện phép trừ, được kết quả 18, đưa vào ngăn xếp.



Duyệt tiếp gấp toán tử nhân là phần tử cuối của biểu thức. Lấy ra 2 toán hạng

trong ngăn xếp là 18 và 3. Thực hiện phép nhân được kết quả 54. Do đã hết biểu thức nên 54 là kết quả cuối cùng và chính là giá trị biểu thức.

Chuyển đổi biểu thức dạng trung tố sang hậu tố

Như vậy, ta có thể thấy rằng biểu thức dạng hậu tố có thể được tính dễ dàng nhờ máy tính thông qua ngăn xếp. Tuy nhiên, biểu thức dạng trung tố vẫn gần gũi và được sử dụng phổ biến hơn trong thực tế. Vậy bài toán đặt ra là cần phải có thuật toán biến đổi biểu thức dạng trung tố sang dạng hậu tố. Trong thuật toán này, ngăn xếp vẫn được sử dụng như một công cụ hữu hiệu để chứa các phần tử trung gian trong quá trình chuyển đổi.

Thuật toán chuyển đổi biểu thức từ dạng trung tố sang dạng hậu tố như sau:

Duyệt biểu thức từ trái qua phải.

- Nếu gặp dấu mở ngoặc: Bỏ qua
- Nếu gặp toán hạng: Đưa vào biểu thức mới.
- Nếu gặp toán tử: Đưa vào ngăn xếp.
- Nếu gặp dấu đóng ngoặc: Lấy toán tử trong ngăn xếp, đưa vào biểu thức mới.

Ta xem xét thuật toán với biểu thức ở trên (chú ý rằng ta phải điền đầy đủ các dấu ngoặc):

$$(3 * ((5 - 2) * (7 + 1)) - 6)$$

Bước 1: Gặp dấu mở ngoặc bỏ qua, gặp toán hạng 3, đưa vào biểu thức mới.
Biểu thức mới: 3

Ngăn.xếp:

Bước 2: Gặp toán tử *, đưa vào ngăn xếp.

Biểu thức mới: 3

Ngăn xếp:

*

Bước 3: Gặp dấu ngoặc bỏ qua

Biểu thức mới: 3

Ngăn xếp:

*

Bước 4: Gặp toán hạng 5, đưa vào biểu thức mới.

Biểu thức mới: 3 5

Ngăn xếp:

*

Bước 5: Gặp toán tử -, đưa vào ngăn xếp.

Biểu thức mới: 3 5

Ngăn xếp:

-
*

Bước 6: Gặp toán hạng 2, đưa vào biểu thức mới.

Biểu thức mới: 3 5 2

Ngăn xếp:

-
*

Bước 7: Gặp dấu đóng ngoặc, lấy toán tử ra khỏi ngăn xếp, đưa vào biểu thức mới.

Biểu thức mới: 3 5 2 -

Ngăn xếp:

*

Bước 8: Gặp toán tử *, đưa vào ngăn xếp.

Biểu thức mới: 3 5 ^

*
*

Ngăn xếp:

Bước 9: Gặp dấu mở ngoặc, bỏ qua.

Biểu thức mới: 3 5 2 -

Ngăn xếp:

*

*

Bước 10: Gặp toán hạng 7, đưa vào biểu thức mới.

Biểu thức mới: 3 5 2 - 7

Ngăn xếp:

*
*

Bước 11: Gặp toán tử +, đưa vào ngăn xếp.

Biểu thức mới: 3 5 2 - 7

Ngăn xếp:

+
*

*

Bước 12: Gặp toán hạng 1, đưa vào biểu thức mới.

Biểu thức mới: 3 5 2 - 7 1

Ngăn xếp:

+

*

*			
---	--	--	--

Bước 13: Gấp dấu đóng ngoặc, lấy toán tử ra khỏi ngăn xếp (+), đưa vào biểu thức mới.

Biểu thức mới: 3 5 ^ 7 1 +

Ngăn xếp:

*
*

Bước 14: Gấp dấu đóng ngoặc, lấy toán tử ra khỏi ngăn xếp (*), đưa vào biểu thức mới.

Biểu thức mới: 3 5 2 - 7 1 + *

Ngăn xếp:

*

Bước 15: Gấp toán tử -, đưa vào ngăn xếp.

Biểu thức mới: 3 5 2 - 7 1 + *

Ngăn xếp:

-
*

Bước 16: Gấp toán hạng 6, đưa vào biểu thức mới.

Biểu thức mới: 3 5 2 - 7 1 + * 6

Ngăn xếp:

*

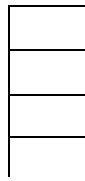
-

Bước 17: Gấp 2 dấu đóng ngoặc, lần lượt lấy các toán tử ra khỏi ngăn xếp

vào đưa vào biểu thức mới.

Biểu thức mới: 3 5 2 - 7 1 + * 6 - *

Ngăn xếp:



Vậy ta có kết quả biểu thức dạng hậu tố là:

3 5 2 - 7 1 + * 6 - *

3.2. Hàng đợi (QUEUE)

3.3.1. Định nghĩa hàng đợi

Hàng đợi là một cấu trúc dữ liệu gần giống với ngăn xếp, nhưng khác với ngăn xếp ở nguyên tắc chọn phần tử cần lấy ra khỏi tập phần tử. Trái ngược với ngăn xếp, phần tử được lấy ra khỏi hàng đợi không phải là phần tử mới nhất được đưa vào mà là phần tử đã được lưu trong hàng đợi lâu nhất.

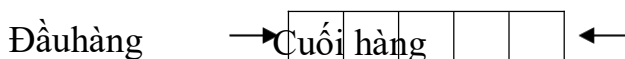
Điều này nghe có vẻ hợp với quy luật thực tế hơn là ngăn xếp ! Quy luật này của hàng đợi còn được gọi là Vào trước ra trước (FIFO - First In First Out). Ví dụ về hàng đợi có rất nhiều trong thực tế. Một dòng người xếp hàng chờ cắt tóc ở 1 tiệm hớt tóc, chờ vào rạp chiếu phim, hay siêu thị là nhưng ví dụ về hàng đợi. Trong lĩnh vực máy tính cũng có rất nhiều ví dụ về hàng đợi. Một tập các tác vụ chờ phục vụ bởi hệ điều hành máy tính cũng tuân theo nguyên tắc hàng đợi.

Hàng đợi còn khác với ngăn xếp ở chỗ: phần tử mới được đưa vào hàng đợi sẽ nằm ở phía cuối hàng, trong khi phần tử mới đưa vào ngăn xếp lại nằm ở đỉnh ngăn xếp.

Như vậy, ta có thể định nghĩa hàng đợi là một dạng đặc biệt của danh sách mà việc lấy ra một phần tử, get, được thực hiện ở 1 đầu (gọi là đầu hàng), còn việc bổ sung 1 phần tử, put, được thực hiện ở đầu còn lại (gọi là cuối hàng).

Trở lại với ví dụ về việc bổ sung và loại bỏ các phần tử của 1 ngăn xếp các

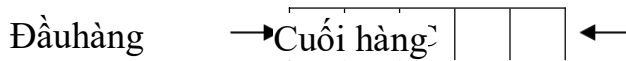
cho hàng đợi các ký tự. Giả sử ta có hàng đợi Q lưu trữ các ký tự. Ban đầu Q ở trạng thái rỗng:



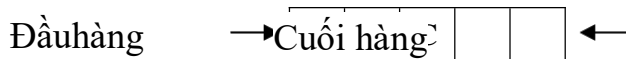
Khi thực hiện lệnh bổ sung phần tử A, put(Q, A), hàng đợi có dạng:



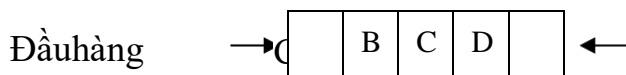
Tiếp theo là các lệnh $\text{put}(Q, B)$, $\text{put}(Q, C)$:



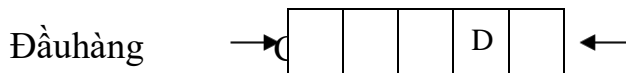
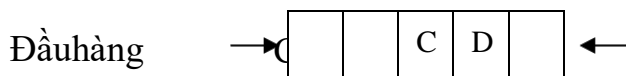
Khi thực hiện lệnh get để lấy ra 1 phần tử từ hàng đợi thì phần tử được lưu trữ lâu nhất trong hàng sẽ được lấy ra. Đó là phần tử đầu tiên ở đầu hàng.



Tiếp theo, thực hiện lệnh $\text{put}(Q, D)$ để bổ sung phần tử D. Phần tử này sẽ được bổ sung ở phía cuối của hàng.



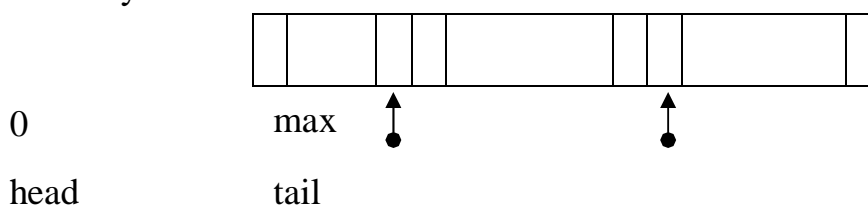
Hai lệnh get tiếp theo sẽ lần lượt lấy ra 2 phần tử ở đầu hàng là B và C.



3.3.2. Biểu diễn hàng đợi

Cài đặt hàng đợi bằng mảng

Tương tự như ngăn xếp, hàng đợi có thể được cài đặt bằng mảng hoặc danh sách liên kết. Đối với ngăn xếp, việc bổ sung và loại bỏ một phần tử đều được thực hiện ở đỉnh ngăn xếp, do vậy ta chỉ cần sử dụng 1 biến top để lưu giữ để đỉnh này. Tuy nhiên, đối với hàng đợi việc bổ sung và loại bỏ phần tử được thực hiện ở 2 đầu khác nhau, do vậy ta cần sử dụng 2 biến là head và tail để lưu giữ điểm đầu và điểm cuối của hàng đợi. Các phần tử thuộc hàng đợi là các phần tử nằm giữa điểm đầu và điểm cuối này.



Hình 4.3 Cài đặt hàng đợi bằng mảng

Để lấy ra 1 phần tử của hàng, điểm đầu tăng lên 1 và phần tử ở đầu hàng sẽ

được lấy ra. Để bổ sung 1 phần tử vào hàng đợi, phần tử này sẽ được bổ sung vào cuối hàng và điểm cuối sẽ tăng lên 1.

Ta thấy rằng biến tail luôn tăng khi bổ sung phần tử và cũng không giảm khi loại bỏ phần tử. Do đó, sau 1 số hữu hạn thao tác, biến này sẽ tiến đến cuối mảng và cho dù phần đầu mảng có thể còn trống do một số phần tử của hàng đợi đã được lấy ra, ta vẫn không thể bổ sung thêm phần tử vào hàng đợi. Để giải quyết vấn đề này, ta sử dụng phương pháp quay vòng. Khi biến tail tiến đến cuối mảng và phần đầu mảng còn trống thì ta sẽ cho biến này quay trở lại đầu mảng. Tương tự vậy, ta cũng cho biến head quay lại đầu mảng khi nó tiến tới cuối mảng.

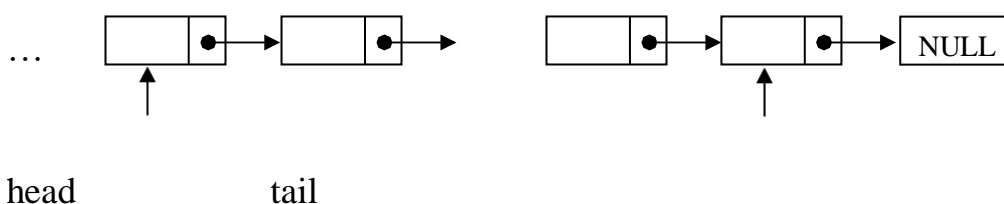
Khai báo bằng mảng cho 1 hàng đợi chứa các số nguyên với tối đa 100 phần tử như sau:

```
#define MAX 100 typedef struct {  
    int head, tail, count; int node[MAX];  
} queue;
```

Trong khai báo này, để thuận tiện cho việc kiểm tra hàng đợi đầy hoặc rỗng, ta dùng thêm 1 biến count để cho biết số phần tử hiện tại của hàng đợi.

Cài đặt hàng đợi bằng danh sách liên kết

Để cài đặt hàng đợi bằng danh sách liên kết, ta cũng sử dụng 1 danh sách liên kết đơn và 2 con trỏ head và tail lưu giữ nút đầu và nút cuối của danh sách. Việc bổ sung phần tử mới sẽ được tiến hành ở cuối danh sách và việc lấy phần tử ra sẽ được tiến hành ở đầu danh sách.



Hình 4.4 Cài đặt hàng đợi bằng danh sách liên kết

Khai báo 1 hàng đợi bằng danh sách liên kết như sau:

```
struct node {  
    int item;  
    struct node *next;  
};  
typedef struct node *queuenode; typedef struct {  
    queuenode head;  
    queuenode tail;
```



```
}queue;
```

Khai báo tương tự như ngăn xếp, tuy nhiên, hàng đợi sử dụng 2 biến là head và tail để lưu giữ điểm đầu và điểm cuối của hàng.

3.3.3. Các thao tác trên hàng đợi

3.3.3.1 Cài đặt hàng đợi bằng mảng

Thao tác khởi tạo hàng đợi

Thao tác này thực hiện việc gán giá trị 0 cho biến head, giá trị MAX -1 cho biến tail, và giá trị 0 cho biến count, cho biết hàng đợi đang ở trạng thái rỗng.

```
void
QueueInitialize(queue *q) {
q-> head = 0;
    q-> tail = MAX-1;
q-> count = 0; return;
}
```

Thao tác kiểm tra hàng đợi rỗng

Hàng đợi rỗng nếu có số phần tử nhỏ hơn hoặc bằng 0.

```
int QueueEmpty(queue
q){ return (q.count <= 0);
}
```

Thao tác thêm 1 phần tử vào hàng đợi

```
void Put(queue *q, int
x){ if (q-> count == MAX)
printf("Hang doi day !");
else{
            if (q->tail ==
MAX-1 ) q-
>tail=0;

else
(q->tail)++;
q->node[q->tail]=x; q->
count++;
}
return;
}
```

Để thêm phần tử vào cuối hàng đợi, điểm cuối tăng lên 1 (nếu điểm cuối đã ở vị trí cuối mảng thì quay vòng điểm cuối về 0). Trước khi thêm phần tử vào hàng đợi, cần kiểm tra xem hàng đợi đã đầy chưa (hàng đợi đầy khi giá trị biến count = MAX).

Lấy phần tử ra khỏi hàng đợi

Để lấy phần tử ra khỏi hàng đợi, tiến hành lấy phần tử tại vị trí điểm đầu và cho điểm đầu tăng lên 1 (nếu điểm đầu đã ở vị trí cuối mảng thì quay vòng điểm đầu về 0). Tuy nhiên, trước khi làm các thao tác này, ta phải kiểm tra xem hàng đợi có rỗng hay không.

```
int      Get(queue
*q){ int x;
    if (QueueEmpty(*q)) printf("Hang
doi rong!");
    else{
        x = q-> node[q-
> head];  if (q-
>head == MAX-1 )
            q->head=0;

        else
            (q->head)++;
        q-> count--;
    }
    return x;
}
```

3.3.3.2.Cài đặt hàng đợi bằng danh sách liên kết

Thao tác khởi tạo hàng đợi

Thao tác này thực hiện việc gán giá trị null cho nút đầu và cuối của hàng đợi, cho biết hàng đợi đang ở trạng thái rỗng.

```
void QueueInitialize(queue *q) {
    q-> head = q-> tail = NULL;
return;
}
```

Thao tác kiểm tra hàng đợi rỗng

Hàng đợi rỗng nếu nút đầu trỏ đến NULL.

```
int QueueEmpty(queue q) {
return (q.head ==NULL);
}
```

Thao tác thêm 1 phần tử vào hàng đợi

```
void Put(queue *q,
int x){ queuenodep;
    p = (queuenode) malloc (sizeof(struct
node)); p-> item = x;
    p-> next = NULL;
    q-> tail-> next = p;
    q-> tail = q-> tail-> next;
    if (q-> head == NULL) q-> head = q->
tail; return;
}
```

Để thêm phần tử vào cuối hàng đợi, tạo và cấp phát bộ nhớ cho 1 nút mới. Gán giá trị thích hợp cho nút này, sau đó cho con trỏ tiếp của nút cuối hàng đợi trỏ đến nó. Nút này bây giờ trở thành nút cuối của hàng đợi. Nếu hàng đợi chưa có phần tử nào thì nó cũng chính là nút đầu của hàng đợi.

Lấy phần tử ra khỏi hàng đợi

Để lấy phần tử ra khỏi hàng đợi, tiến hành lấy phần tử tại vị trí nút đầu và cho nút đầu chuyển về nút kế tiếp. Tuy nhiên, trước khi làm các thao tác này, ta phải kiểm tra xem hàng đợi có rỗng hay không.

```

int      Get(queue*q) {
    queuenode p;
    if          (QueueEmpty(*q)) {
printf("Ngan xep rong !"); return 0;
    }else{
        p = q-> head;
        q-> head = q-> head-> next;
return p->item;
    }
}

```

3.3.4. Ứng dụng của hàng đợi

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

Cấu trúc dữ liệu hàng đợi có thể định nghĩa như sau: Hàng đợi là một cấu trúc dữ liệu trừu tượng (ADT) tuyến tính. Tương tự như ngăn xếp, hàng đợi hỗ trợ các thao tác:

- EnQueue(o): thêm đối tượng o vào cuối hàng đợi.
- DeQueue(): lấy đối tượng ở đầu queue ra khỏi hàng đợi và trả về giá trị của nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.
- IsEmpty(): kiểm tra xem hàng đợi có rỗng không.
- Front(): trả về giá trị của phần tử nằm ở đầu hàng đợi mà không hủy nó. Nếu hàng đợi rỗng thì lỗi sẽ xảy ra.

Các thao tác thêm, trích và hủy một phần tử phải được thực hiện ở hai phía khác nhau của hàng đợi, do đó hoạt động của hàng đợi được thực hiện theo nguyên tắc FIFO. Cũng như ngăn xếp, cấu trúc mảng một chiều hoặc cấu trúc danh sách liên kết có thể dùng để biểu diễn cấu trúc hàng đợi.

3.3. Danh sách liên kết đơn

3.3.1. Định nghĩa danh sách liên kết đơn

Danh sách liên kết có thể được cài đặt bằng mảng hoặc bằng con trỏ. Loại danh sách này gọi tắt là danh sách liên kết đơn

3.3.2. Biểu diễn danh sách liên kết đơn

Cài đặt danh sách

```
typedef int item; //kieu cac phan tu dinh nghia la item
```

```
typedef struct Node //Xay dung mot Node trong danh sach
{
    item Data; //Du lieu co kieu item
    Node *next; //Truong next la con tro, tro den 1 Node tiep theo
};
typedef Node *List; //List la mot danh sach cac Node
```

Khởi tạo danh sách rỗng

```
void Init (List &L) // &L lay dia chi cua danh sach ngay khi truyen vao
ham
{
    L=NULL; //Cho L tro den NULL
}
```

Để có thể thay đổi được giá trị của đối mà ta truyền vào hàm ta thường dùng biến con trỏ (*) và trong lời gọi hàm ta cần có & trước biến tuy nhiên khi chúng ta sử dụng cách truyền địa chỉ ngay khi khởi tạo hàm thì trong lời gọi hàm ta tiên hành truyền biến bình thường mà không phải lấy địa chỉ (thêm &) trước biến nữa.
(Đây là một cách truyền địa chỉ cho biến trong hàm ở C++.)

3.3.3. Các thao tác trên danh sách liên kết đơn

Kiểm tra danh sách rỗng hay không

```
int Isempty (List L)
{
    return (L==NULL) ;
}
```

Tính độ dài danh sách

Ta dùng 1 Node để duyệt từ đầu đến cuối, vừa duyệt vừa đếm

```
int len (List L)
{
    Node *P=L; //tao 1 Node P de duyet danh sach L
    int i=0; //bien dem
    while(P!=NULL) //trong khi P chua tro den NULL
    (cuoi danh sach thi lam)
    {
        i++; //tang bien dem
        P=P->next; //cho P tro den Node tiep theo
    }
    return i; //tra lai so Node cua l
}
```

Tạo 1 Node trong danh sách

Việc tạo 1 Node chứa thông tin trong danh sách sẽ giúp ta dễ dàng chèn, xóa và quản

lý danh sách hơn. Trước tiên ta sẽ phải cấp phát vùng nhớ cho Node và sau đó gán Data vào là ok

```
Node *Make_Node (Node *P, item x) //tao 1 Node P
chua thong tin la x
{
    P = (Node *) malloc(sizeof(Node)); //Cap phat
vung nho cho P
    P->next = NULL; //Cho truong Next tro den NULL
    P->Data = x; //Ghi du lieu vao Data
    return P;
}
```

Chèn Node P vào vị trí đầu tiên

```
void Insert_first (List &L, item x) //Chen x vao vi
tri dau tien trong danh sach
{
    Node *P;
    P = Make_Node(P,x); //tao 1 Node P
    P->next = L; //Cho P tro den L
    L = P; //L tro ve P
}
```

Chèn Node P vào vị trí k trong danh sách

Trước tiên ta kiểm tra vị trí chèn có hợp lệ không, nếu hợp lệ kiểm tra tiếp chèn vào vị trí 1 hay $k > 1$. Với $k > 1$ ta thực hiện duyệt bằng Node Q đến vị trí $k-1$ sau đó cho $P \rightarrow \text{Next}$ trở đến Node $Q \rightarrow \text{Next}$, tiếp đến cho $Q \rightarrow \text{Next}$ trở đến P

```
idInsert_k (List &L, item x, int k) //chen x vao vi
tri k trong danh sach
{
    Node *P, *Q = L;
    inti=1;
    if(k<1 || k> len(L)+1) printf("Vi tri chen khong
hop le !"); //kiem tra dieu kien
    else
    {
        P = Make_Node(P,x); //tao 1 Node P
        if(k == 1) Insert_first(L,x); //chen vao vi
tri dau tien
        else//chen vao k != 1
        {
            while(Q != NULL && i != k-1) //duyet den
vi tri k-1
            {
                i++;
                Q = Q->next;
            }
        }
    }
}
```

```

        P->next = Q->next;
        Q->next = P;
    }
}

```

Tìm phần tử có giá trị x trong danh sách

Ta duyệt danh sách cho đến khi tìm thấy hoặc kết thúc và trả về vị trí nếu tìm thấy, ngược lại trả về 0

```

intSearch (List L, item x) //tim x trong danh sach
{
    Node *P=L;
    inti=1;
    while(P != NULL && P->Data != x) //duyet danh
sach den khi tim thay hoac ket thuc danh sach
    {
        P = P->next;
        i++;
    }
    if(P != NULL) return i; //tra ve vi tri tim
thay
    elsereturn 0; //khong tim thay
}

```

Xóa phần tử ở vị trí đầu tiên

```

voidDel_frist (List &L, item &x) //Xoa phan tu dau
tien
{
    x = L->Data; //lay gia tri ra neu can dung
    L = L->next; //Cho L tro den Node thu 2 trong
danh sach
}

```

Xóa phần tử ở vị trí k

Dùng P duyệt đến vị trí k-1 và tiến hành cho P->Next trở đến phần tử kế tiếp k mà bỏ qua k. Lưu ý trong hình mình quên không lưu lại giá trị cần xóa tuy nhiên các bạn cần lưu lại như khi xóa ở vị trí đầu tiên.

```

voidDel_k (List &L, item &x, intk) //Xoa Node k
trong danh sach
{
    Node *P=L;
    inti=1;
    if(k<1 || k>len(L)) printf("Vi tri xoa khong
hop le !"); //kiem tra dieu kien
    else
    {
        if(k==1) Del_frist(L,x); //xoa vi tri dau

```

```

tien
        else//xoa vi tri k != 1
        {
            while(P != NULL && i != k-1) //duyet
den vi tri k-1
            {
                P=P->next;
                i++;
            }
            P->next = P->next->next; //cho P tro
sang Node ke tiep vi tri k
        }
    }
}

```

Xóa phần tử có giá trị x

Đơn giản là ta tìm x trong danh sách bằng hàm Search và xóa tại vị trí tìm thấy mà ta nhận được

```

void Del_x (List &L, item x) //xoa phan tu x
trong danh sach
{
    while(Search(L,x)) Del_k (L,x,Search(L,x));
//trong khi van tim thay x thi van xoa
}

```

3.3.4. Ứng dụng của danh sách liên kết đơn

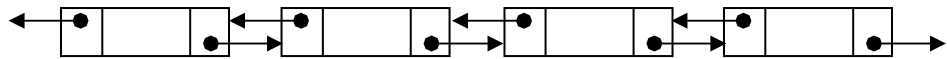
- Danh sách liên kết là 1 cấu trúc dữ liệu bao gồm 1 tập các phần tử, trong đó mỗi phần tử là 1 phần của 1 nút có chứa một liên kết tới nút kế tiếp.
- Danh sách liên kết có kiểu truy cập tuần tự, có kích thước linh hoạt và dễ dàng trong việc bố trí lại các phần tử.
- Các thao tác cơ bản trên danh sách liên kết bao gồm: Khởi tạo danh sách, chèn 1 phần tử vào đầu, cuối, giữa danh sách, xóa 1 phần tử khỏi đầu, cuối, giữa danh sách, duyệt qua toàn bộ danh sách.

3.4. Danh sách liên kết kép

3.4.1. Định nghĩa danh sách liên kết kép

Trong danh sách liên kết đơn, mỗi nút chỉ có một liên kết trỏ tới nút kế tiếp. Điều này có nghĩa danh sách liên kết đơn chỉ cho phép duyệt theo 1 chiều, trong khi thao tác duyệt chiều ngược lại đôi khi cũng rất cần thiết. Để giải quyết vấn đề này, ta có thể tạo cho mỗi nút hai liên kết: một để trỏ tới nút đứng trước, một để trỏ tới nút đứng sau. Những danh sách như vậy được gọi là danh sách liên kết kép.

3.4.2. Biểu diễn danh sách liên kết kép



Hình 3.3 Danh sách liên kết kép

Việc khai báo danh sách liên kết kép cũng tương tự như khai báo danh sách liên kết đơn, chỉ có điểm khác biệt là có thêm 1 liên kết nữa cho mỗi nút.

```
struct node {  
    itemstruct item; struct node  
*left;  
  
    struct node *right;  
};  
typedef struct node *doublelistnode;
```

3.4.3. Các thao tác trên danh sách liên kết kép

3.4.3.1 Cài đặt danh sách

Cấu trúc của 1 Node trong danh sách liên kết kép tương đối giống với DSLKD nhưng có thêm một con trỏ trở về Node trước nó

```
typedef int item;  
  
typedef struct Node //cau truc 1 Node  
{  
    item Data; //du lieu cua Node  
    Node *Left; //Con tro trai  
    Node *Right; //con tro phai  
};  
  
typedef struct DList //cau truc Cua List  
{  
    Node *Head; //con tro dau  
    Node *Tail; //con tro cuoi  
};
```

Cấu trúc của DSLKK không như DSLKD có 1 Con trỏ trỏ đến đầu DS, nhưng DSLKK ngoài con trỏ trỏ đến đầu danh sách còn có thêm 1 con trỏ trỏ đến Node cuối của danh sách

3.4.3.2 Khởi tạo và kiểm tra rỗng

Khởi tạo ta cho 2 con trỏ đầu và cuối trỏ về NULL, Khi kiểm tra rỗng thì chỉ cần xem con trỏ đầu có trỏ về NULL không là đủ

```
voidInit(DList &L)
{
    L.Head = NULL; // Con trỏ đầu trỏ den NULL
    L.Tail = NULL; // Con trỏ cuối trỏ den NULL
}

intIsEmpty (DList L) //kiem tra DS rong
{
    return(L.Head == NULL);
}
```

3.4.3.3. Độ dài danh sách

Để tìm độ dài của DSLKK ta hoàn toàn có thể làm giống như DSLKD, tức dùng con trỏ duyệt từ đầu đến cuối, nhưng trong DSLKK ta có thể dùng 2 con trỏ ở đầu và cuối để đếm

```
intLen (DList L) // Do dai danh sach
{
    Node *PH = L.Head, *PT = L.Tail; //tao Node PH (con trỏ duyệt từ đầu DS) và PT (con trỏ duyệt từ cuối DS) để duyệt danh sách L
    inti = 0; //biến đếm
    if(PH != NULL) i = 1;
    while(PH != NULL) //trong khi P chưa trỏ den NULL (cuối danh sách thì làm)
    {
        if(PH == PT) break;
        PH = PH->Right; //cho PH trỏ den Node tiếp theo
        i++;
        if(PH == PT) break;
        PT = PT->Left; //cho PT trỏ den Node trước đó
        i++;
    }
    return i; //tra lại số Node của L
}
```

3.4.3.4. Tạo 1 Node P chứa thông tin

```
Node *Make_Node (item x) //tao 1 Node P chứa thông tin là x
{
    Node *P = (Node *) malloc(sizeof(Node)); //Cấp phát vùng nhớ cho P
    P->Data = x; //Ghi dữ liệu vào Data
}
```

```

P->Left = NULL;
P->Right = NULL;
return P;
}

```

3.4.3.5. Chèn phần tử vào vị trí đầu tiên

Trước khi chèn vào đầu danh sách cần kiểm tra xem danh sách rỗng hay không. Nếu danh sách rỗng ta cho Head và Tail đều trỏ đến P. Nếu không rỗng thực hiện chèn.

```

void Insert_first (DList &L, item x) //Chen x vao vi tri dau tien trong
danh sach
{
    Node *P;
    P = Make_Node(x); //tao 1 Node P
    if(Isempty(L)) //Neu danh sach rong
    {
        L.Head = P;
        L.Tail = P;
    }
    else
    {
        P->Right = L.Head;
        L.Head->Left = P;
        L.Head = P;
    }
}

```

3.4.3.6. Chèn phần tử vào cuối danh sách tương tự như đầu danh sách

```

void Insert_last (DList &L, item x) //Chen x vao vi tri cuoi trong danh
sach
{
    Node *P;
    P = Make_Node(x); //tao 1 Node P
    if(Isempty(L)) //Neu danh sach rong
    {
        L.Head = P;
        L.Tail = P;
    }
    else
    {
        L.Tail->Right = P; //ket noi voi danh sach
        P->Left = L.Tail; //P tro ve Node truoc
        L.Tail = P; //luu lai vi tri cuoi
    }
}

```

```

    }
}

```

3.4.3.7. Chèn phần tử vào vị trí k

Trước khi chèn vào vị trí k cần kiểm tra vị trí k có phù hợp, có phải đầu danh sách hay cuối danh sách. Nếu chèn vào giữa danh sách ta thực hiện theo 4 bước

```

void Insert_k (DList &L, item x, int k) //chen x vao vi tri k trong danh
sach
{
    Node *PH = L.Head, *PT, *R;
    int i=1, l = Len(L);
    if(k<1 || k> l+1) printf("Vi tri chen khong hop le !"); //kiem tra dieu
kien
    else
    {
        R = Make_Node(x); //tao 1 Node P
        if(k == 1) Insert_first(L,x); //chen vao vi tri dau tien
        else
            if(k == l+1) Insert_last(L,x); //chen vao vi tri cuoi
            else //chen vao vi tri 1<k<l+1
            {
                while(PH != NULL && i != k-1) //duyet den vi tri k-1
                {
                    i++;
                    PH = PH->Right;
                }
                PT = PH->Right; //xac dinh vi tri k
                R->Right = PT; //(1)
                R->Left = PH; //(2)
                PH->Right = R; //(3)
                PT->Left = R; //(4)
            }
    }
}

```

3.4.3.8. Xóa phần tử đầu, cuối danh sách

```

// Lay gia tri can xoa ra, sau do bo qua 1 Node dau tien
void Del_first (DList &L, item &x) //Xoa phan tu dau tien
{
    if(!Isempty(L))
    {
        x = L.Head->Data; //lay gia tri ra neu can dung
    }
}

```

```

        L.Head = L.Head->Right; //Cho L tro den Node thu 2 trong danh
sach
    }
}

// Lay gia tri can xoa ra, sau do bo qua 1 Node cuoi
void Del_last (DList &L, item &x) //Xoa phan tu dau tien
{
    if(!Isempty(L))
    {
        x = L.Tail->Data;
        L.Tail = L.Tail->Left;
        L.Tail->Right = NULL;
    }
}

```

3.4.3.9. Xóa phần tử ở vị trí k

```

void Del_k (DList &L, item &x, int k) //Xoa Node k trong danh sach
{
    Node *PH = L.Head, *PT;
    inti=1, l = Len(L);
    if(k<1 || k> l) printf("Vi tri xoa khong hop le !"); //kiem tra dieu kien
    else
    {
        if(k == 1) Del_first(L,x); //xoa vi tri dau tien
        else
            if(k == l) Del_last(L,x); //xoa vi tri cuoi
            else//xoa vi tri 1<k<l+1
            {
                while(PH != NULL && i != k-1) //duyet den vi tri k-1
                {
                    i++;
                    PH = PH->Right;
                }
                x = PH->Right->Data;
                PT = PH->Right->Right; //xac dinh vi tri k+1
                PH->Right = PT;
                PT->Left = PH;
            }
    }
}

```

3.4.3.10. Tìm phần tử x trong DS

```
intSearch (DList L, item x) //tim x trong danh sach
{
    Node *P=L.Head;
    inti=1;
    while(P != NULL && P->Data != x) //duyet danh sach den khi tim
    thay hoac ket thuc danh sach
    {
        P = P->Right;
        i++;
    }
    if(P != NULL) return i; //tra ve vi tri tim thay
    elsereturn 0; //khong tim thay
}
```

3.4.3.11. Xóa phần tử x trong DS

```
voidDel_x (DList &L, item x) //xoa phan tu x trong danh sach
{
    intl = Search(L,x);
    while(l)
    {
        Del_k (L,x,l); //trong khi van tim thay x thi van xoa
        l = Search(L,x);
    }
}
```

3.4.4. Ứng dụng của danh sách liên kết kép

- Danh sách liên kết là 1 cấu trúc dữ liệu bao gồm 1 tập các phần tử, trong đó mỗi phần tử là 1 phần của 1 nút có chứa một liên kết tới nút kế tiếp.
- Danh sách liên kết có kiểu truy cập tuần tự, có kích thước linh hoạt và dễ dàng trong việc bố trí lại các phần tử.
- Các thao tác cơ bản trên danh sách liên kết bao gồm: Khởi tạo danh sách, chèn 1 phần tử vào đầu, cuối, giữa danh sách, xóa 1 phần tử khỏi đầu, cuối, giữa danh sách, duyệt qua toàn bộ danh sách.

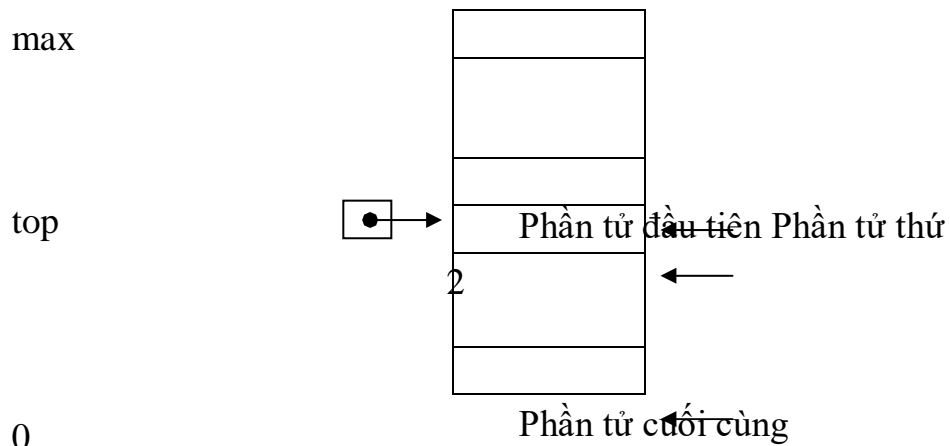
3.5. CASE STUDY

3.5.1. Cài đặt chương trình máy tính các thuật toán trên ngăn xếp.

Cài đặt ngăn xếp bằng mảng

Ngăn xếp có thể được cài đặt bằng mảng hoặc danh sách liên kết (sẽ được trình bày ở phần sau). Để cài đặt ngăn xếp bằng mảng, ta sử dụng một mảng 1 chiều s để

biểu diễn ngăn xếp. Thiết lập phần tử đầu tiên của mảng, $s[0]$, làm đáy ngăn xếp. Các phần tử tiếp theo được đưa vào ngăn xếp sẽ lần lượt được lưu tại các vị trí $s[1]$, $s[2]$, ... Nếu hiện tại ngăn xếp có n phần tử thì $s[n-1]$ sẽ là phần tử mới nhất được đưa vào ngăn xếp. Để lưu giữ đỉnh hiện tại của ngăn xếp, ta sử dụng 1 con trỏ top . Chẳng hạn, nếu ngăn xếp có n phần tử thì top sẽ có giá trị bằng $n-1$. Còn khi ngăn xếp chưa có phần tử nào thì ta quy ước top sẽ có giá trị -1 .



Hình 4.1 Cài đặt ngăn xếp bằng mảng

Nếu có 1 phần tử mới được đưa vào ngăn xếp thì nó sẽ được lưu tại vị trí kế tiếp trong mảng và giá trị của biến top tăng lên 1. Khi lấy 1 phần tử ra khỏi ngăn xếp, phần tử của mảng tại vị trí top sẽ được lấy ra và biến top giảm đi 1.

Có 2 vấn đề xảy ra khi thực hiện các thao tác trên trong ngăn xếp. Khi ngăn xếp đã đầy, tức là khi biến top đạt tới phần tử cuối cùng của mảng thì không thể tiếp tục thêm phần tử mới vào mảng. Và khi ngăn xếp rỗng, tức là chưa có phần tử nào, thì ta không thể lấy được phần tử ra từ ngăn xếp. Như vậy, ngoài các thao tác đưa vào và lấy phần tử ra khỏi ngăn xếp, cần có thao tác kiểm tra xem ngăn xếp có rỗng hoặc đầy hay không.

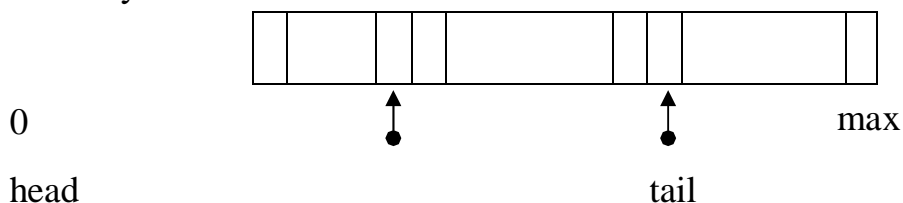
Khai báo bằng mảng cho 1 ngăn xếp chứa các số nguyên với tối đa 100 phần tử như sau:

```
#define MAX      100
typedef struct {
    int          top;
    int          nut[MAX];
} stack;
```

3.5.2. Cài đặt chương trình máy tính các thuật toán trên hàng đợi.

Cài đặt hàng đợi bằng mảng

Tương tự như ngăn xếp, hàng đợi có thể được cài đặt bằng mảng hoặc danh sách liên kết. Đối với ngăn xếp, việc bổ sung và loại bỏ một phần tử đều được thực hiện ở đỉnh ngăn xếp, do vậy ta chỉ cần sử dụng 1 biến top để lưu giữ để đỉnh này. Tuy nhiên, đối với hàng đợi việc bổ sung và loại bỏ phần tử được thực hiện ở 2 đầu khác nhau, do vậy ta cần sử dụng 2 biến là head và tail để lưu giữ điểm đầu và điểm cuối của hàng đợi. Các phần tử thuộc hàng đợi là các phần tử nằm giữa điểm đầu và điểm cuối này.



Hình 4.3 Cài đặt hàng đợi bằng mảng

Để lấy ra 1 phần tử của hàng, điểm đầu tăng lên 1 và phần tử ở đầu hàng sẽ được lấy ra. Để bổ sung 1 phần tử vào hàng đợi, phần tử này sẽ được bổ sung vào cuối hàng và điểm cuối sẽ tăng lên 1.

Ta thấy rằng biến tail luôn tăng khi bổ sung phần tử và cũng không giảm khi

loại bỏ phần tử. Do đó, sau 1 số hữu hạn thao tác, biến này sẽ tiến đến cuối mảng và cho dù phần đầu mảng có thể còn trống do một số phần tử của hàng đợi đã được lấy ra, ta vẫn không thể bổ sung thêm phần tử vào hàng đợi. Để giải quyết vấn đề này, ta sử dụng phương pháp quay vòng. Khi biến tail tiến đến cuối mảng và phần đầu mảng còn trống thì ta sẽ cho biến này quay trở lại đầu mảng. Tương tự vậy, ta cũng cho biến head quay lại đầu mảng khi nó tiến tới cuối mảng.

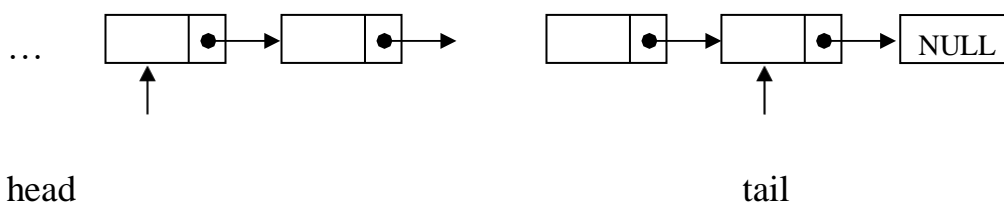
Khai báo bằng mảng cho 1 hàng đợi chứa các số nguyên với tối đa 100 phần tử như sau:

```
#define MAX 100
typedef struct {
    int head, tail, count;
    int node[MAX];
} queue;
```

Trong khai báo này, để thuận tiện cho việc kiểm tra hàng đợi đầy hoặc rỗng, ta dùng thêm 1 biến count để cho biết số phần tử hiện tại của hàng đợi.

Cài đặt hàng đợi bằng danh sách liên kết

Để cài đặt hàng đợi bằng danh sách liên kết, ta cũng sử dụng 1 danh sách liên kết đơn và 2 con trỏ head và tail lưu giữ nút đầu và nút cuối của danh sách. Việc bổ sung phần tử mới sẽ được tiến hành ở cuối danh sách và việc lấy phần tử ra sẽ được tiến hành ở đầu danh sách.



Hình 4.4 Cài đặt hàng đợi bằng danh sách liên kết

Khai báo 1 hàng đợi bằng danh sách liên kết như sau:

```
struct node {
    int item;
    struct node *next;
};
typedef struct node
*queuenode;
typedef struct {
    queuenode head;
    queuenode tail;
```

```
}queue;
```

Khai báo tương tự như ngăn xếp, tuy nhiên, hàng đợi sử dụng 2 biến là *head* và *tail* để lưu giữ điểm đầu và điểm cuối của hàng.

3.5.3. Cài đặt chương trình máy tính các thuật toán trên danh sách liên kết.

```
listnode p; // Khai báo biến p
p = (listnode)malloc(sizeof(struct node)); // cấp phát
bộ nhớ cho
p
free(p); //giải phóng bộ nhớ đã cấp phát cho nút p;
```

3.5.4. Tìm hiểu các thao tác trên ngăn xếp, hàng đợi và danh sách liên kết trong thư viện của ngôn ngữ lập trình C++.

Chương 4. Cây nhị phân

Chương 4 giới thiệu một cấu trúc dữ liệu rất gần gũi và có nhiều ứng dụng trong thực tế, đó là cấu trúc dữ liệu kiểu cây.

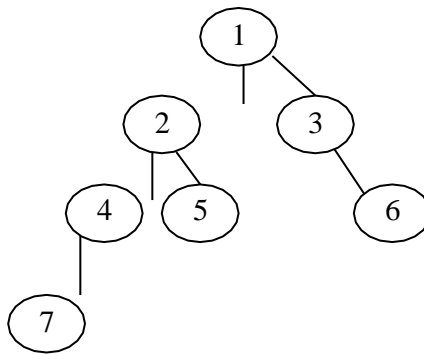
Các nội dung chính được trình bày trong chương bao gồm:

- Định nghĩa và các khái niệm về cây.
- Cài đặt cây : Cài đặt bằng mảng hoặc danh sách liên kết.
- Phép duyệt cây: Duyệt thứ tự trước, thứ tự giữa, và thứ tự sau.

4.1. Định nghĩa và phân loại cây nhị phân

4.1.1. Định nghĩa cây nhị phân

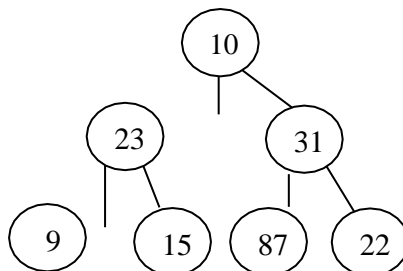
Cây nhị phân là một loại cây đặc biệt mà mỗi nút của nó chỉ có nhiều nhất là 2 nút con. Khi đó, 2 cây con của mỗi nút được gọi là cây con trái và cây con phải.



Hình 5.5 Cây nhị phân

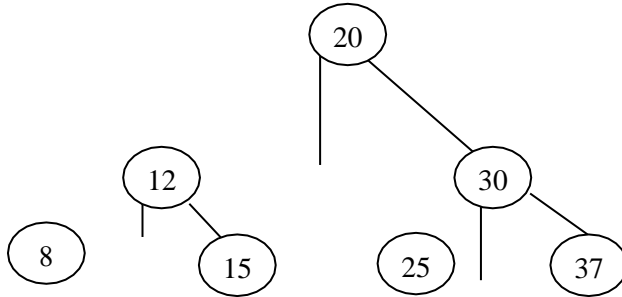
4.1.2. Phân loại các cây nhị phân

- Cây nhị phân đầy đủ: Là cây nhị phân mà mỗi nút không phải lá đều có đúng 2 nút con và các nút lá phải có cùng độ sâu.



Hình 5.6 Cây nhị phân đầy đủ

- Cây nhị phân tìm kiếm: Là cây nhị phân có tính chất khóa của nút con bên trái bao giờ cũng nhỏ hơn khóa của nút cha, còn khóa của cây con bên phải bao giờ cũng lớn hơn hoặc bằng khóa của nút cha.



Hình 5.7 Cây nhị phân tìm kiếm

4.2. Biểu diễn cây nhị phân

4.2.1. Biểu diễn cây nhị phân bằng mảng

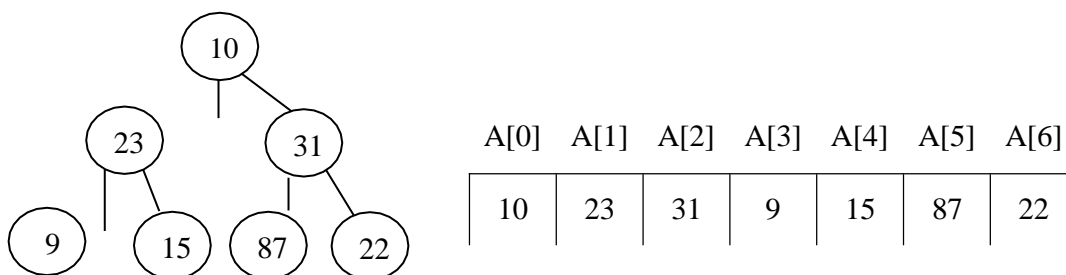
Đối với cây nhị phân đầy đủ, mỗi nút đều có đúng 2 nút con, ta có thể sử dụng 1 mảng để

biểu diễn cây theo quy tắc:

- Nút đầu tiên (nút thứ 1) của mảng là nút gốc.
- Nút thứ i ($i \geq 1$) của cây có 2 nút con là nút thứ $2i$ và $2i + 1$. Điều này đồng nghĩa với nút cha của nút j là nút $[j/2]$.

Với cách lưu trữ này, ta có thể dễ dàng tìm được các nút con của 1 nút cho trước cũng như dễ dàng tìm được nút cha của nó.

Ví dụ, cây nhị phân đầy đủ như ở phần trước có thể được biểu diễn bằng mảng A như sau:



Hình 5.8 Cài đặt cây nhị phân bằng mảng

Đối với cây nhị phân không cân bằng, do số nút con của một nút có thể < 2 nên dùng cách biểu diễn trên không thích hợp. Khi đó, ta có thể dùng một mảng các nút, mỗi nút này có 2 thành phần là nút con trái và nút con phải.

```
typedef struct {  
    int item;
```

```

    int leftchild;  int
rightchild;
    } node;
    node tree[max];

```

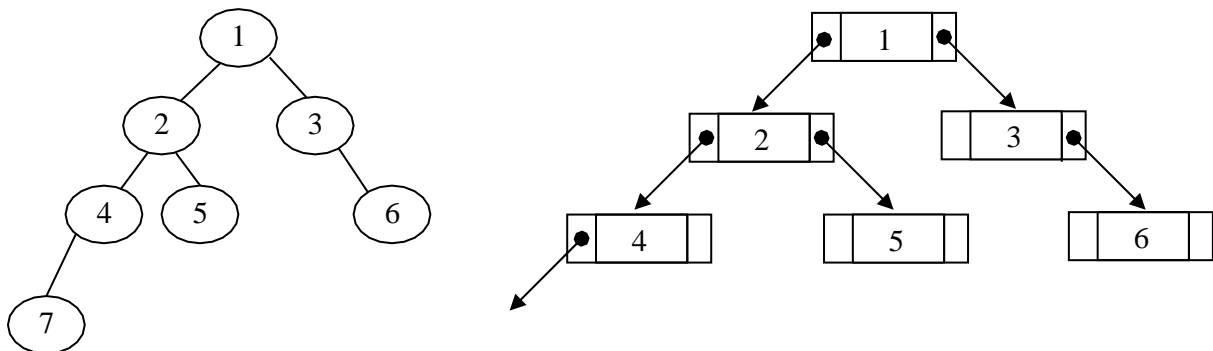
4.2.2. Biểu diễn cây nhị phân bằng danh sách liên kết

Mỗi nút trong cây nhị phân có tối đa 2 nút con, do vậy sử dụng danh sách liên kết để cài đặt cây nhị phân là một phương pháp hữu hiệu. Mỗi nút của cây nhị phân khi đó sẽ có 3 thành phần:

- Thành phần item chứa thông tin về nút.
- Con trỏ left trỏ đến nút con bên trái.
- Con trỏ right trỏ đến nút con bên phải.

Nếu nút có ít hơn 2 nút con thì một trong hai con trỏ hoặc cả 2 sẽ được gán giá trị NULL. Ngoài ra, để tăng cường khả năng di chuyển trong cây, ta có thể thêm một thành phần nữa cho nút đó là con trỏ parent trỏ đến nút cha.

Ví dụ, cây nhị phân ở hình bên dưới có thể được biểu diễn bằng danh sách liên kết như sau:



Hình 5.9 Cài đặt cây nhị phân bằng danh sách liên kết

Khai báo cây nhị phân bằng danh sách liên trên trong C như sau:

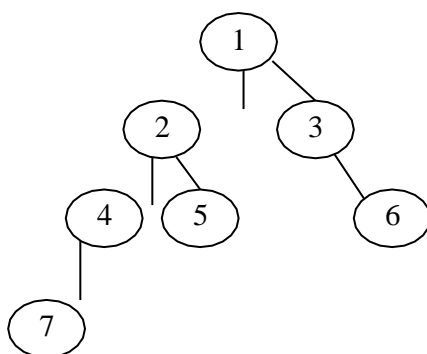
```

struct node{
    int      item;
    struct node *left; struct node *right;
}
typedef struct node *treenode; treenode root;

```

4.3. Thao tác trên cây nhị phân

4.3.1. Thao tác trên cây nhị phân tổng quát



Hình 5.5 Cây nhị phân

4.3.2. Thao tác trên cây nhị phân tìm kiếm

1.MakeEmpty()

+Chức năng: Khởi tạo cây tới trạng thái rỗng.

+Điều kiện sau : Cây tồn tại và ở trạng thái rỗng

2.Boolean IsEmpty()

+Chức năng: Kiểm tra xem liệu cây có rỗng hay không.

+Điều kiện sau : Giá trị hàm = (cây là rỗng ?).

3.Boolean IsFull()

+Chức năng: Kiểm tra xem cây có đầy hay không.

+Điều kiện sau : Giá trị hàm = (cây là đầy ?).

4.int LengthTree()

+Chức năng : Cho biết số lượng phần tử hiện có trong cây.

+Điều kiện sau : Giá trị hàm = Số lượng phần tử trong cây.

-Số lượng nút trên cây nhị phân là xác định nếu như ta biết được số lượng nút trên cây con trái và số lượng nút trên cây con phải.Khi đó

Nút trên cây = Nút trên cây con trái + nút trên cây con phải + 1.

-Điều này gợi ý cho ta cài đặt dùng thuật toán đệ quy.Sử dụng hàm CountNodes để đếm số lượng nút, sau đó hàm LengthTree() sẽ gọi và truyền con trỏ root tới CountNodes như một đối số.

Hàm CountNodes

+ Chức năng : Đếm số lượng Nút.

+ BS : Nếu cây rỗng trả lại 0.

+ RS : Trả lại CountNodes(left-> subtree) + CountNodes(right-> subtree) + 1.

5.RetrieveItem(ItemType& item, bool& found)

+Chức năng : Truy tìm trong cây phần tử có key giống với key của item.

+Điều kiện trước: Khoá của item đã được khởi tạo.

+Điều kiện sau : Nếu như tồn tại một phần tử someltem mà key của nó giống với key của item thì found = true, và item là một bản sao của someltem, nếu không found = false.

-Để tìm kiếm một phần tử item nào đó với thành phần key đã xác định trên cây nhị phân tìm kiếm. Trước hết, ta so sánh key của nó với key của nút gốc để quyết định xem liệu nút cần tìm có phải là nút gốc hay không, nếu không thì sẽ tiếp tục quá trình tìm kiếm trên cây con trái hay cây con phải.

-Việc tìm kiếm trên cây con trái hay cây con phải cũng được thực hiện tương tự cho tới khi tìm kiếm được, hoặc là không. ⇒

Như vậy, quá trình tìm kiếm có thể được thực hiện một cách đệ quy. Trường hợp cơ sở xuất hiện khi hoặc nút cần tìm là nút gốc hoặc cây là rỗng (item không được tìm thấy trên cây).

-Ta có thể cài đặt hàm đệ quy Retrieve() mà hàm RetrieveItem() gọi tới và truyền cho nó con trỏ root như một đối số.

Hàm Retrieve

+ Chức năng : Truy tìm trên cây phân tử có key giống với key của item.

+ BS : (1) key của item bằng với key của nút gốc, khi đó thiết lập

found = true và lưu item vào tree -> info(cập nhật lại).

(2) Nếu cây là rỗng thì found = false.

+ RS : Nếu key của item nhỏ hơn key của tree -> info thì

Retrieve(tree -> left, item, found), nếu không

Retrieve(tree -> right, item, found).

6.InsertItem(ItemType item)

+Chức năng : Thêm một phần tử vào cây.

+Điều kiện trước: Cây chưa đầy.

+Điều kiện sau: item được chèn vào vị trí thích hợp trên cây đảm bảo duy trì thuộc tính tìm kiếm nhị phân

-Chèn một phần tử mới vào cây không những đòi hỏi phải duy trì thuộc tính hình dạng của cây mà còn phải đảm bảo thuộc tính tìm kiếm nhị phân của nó. Ta sẽ chèn thêm một nút mới vào cây tại một vị trí thích hợp như một nút lá.

-Cũng như các thuật toán đã cài đặt ở trên, ta viết hàm đệ quy Insert để chèn một phần tử vào cây, hàm mang theo một con trỏ tree tới nút gốc của cây. Hàm InsertItem đơn giản gọi tới hàm này và truyền cho nó con trỏ root như một tham chiếu

7.DeleteItem(ItemType item)

+Chức năng : Xóa phần tử có key giống với key của item.

+Điều kiện trước : Thành phần key của item đã được khởi tạo

Có duy nhất một phần tử trên cây có key giống với key của item.

+Điều kiện sau : Không tồn tại phần tử nào trên cây có key giống với key của

item

- Trước hết, ta viết hàm Delete mang theo một con trỏ tree tới gốc của một cây để xoá đi một nút trên cây. Hàm DeleteItem đơn giản gọi tới và truyền cho hàm này con trỏ root như một tham chiếu.

- Để xoá đi một nút trên cây hàm Delete cần thực hiện hai bước sau :

+ Tìm nút cần xoá trên cây.

+ Loại bỏ nút cần xoá khỏi cây.

-Bước thứ nhất được thực hiện hoàn toàn giống như đã thực hiện khi cài đặt hàm Retrieve. Bước thứ hai phức tạp hơn xong ta có thể khái quát thành ba trường hợp cơ bản sau, tùy thuộc vào nút cần xoá:

- Nút cần xoá là một nút lá : Việc xoá nó được thực hiện hết sức đơn giản, ta chỉ cần thiết lập liên kết cha của nút đó thành NULL.

- Nút cần xoá chỉ có một con : Việc xoá được thực hiện bằng cách thiết lập liên kết giữa cha của nút cần xoá tới con của nó.

- Nút cần xoá có hai con : Trường hợp này phức tạp hơn, ta không thể áp dụng phương pháp đã thực hiện khi xoá nút với chỉ một con, vì như thế sẽ làm mất đi thuộc tính hình dạng của cây nhị phân tìm kiếm. Thay vào đó ta đưa ra giải pháp thay thế thành phần info của nút cần xoá bằng thành phần info từ một nút khác mà vẫn duy trì được thuộc tính tìm kiếm nhị phân., sau đó nút khác này sẽ được xoá đi. Như vậy, ta chỉ cần giải quyết vấn đề nút được thay thế. Đó chính là nút mà giá trị key của nó gần nhất nhưng nhỏ hơn (hoặc lớn hơn) giá trị key nút cần xoá, nút này còn được gọi là tiền nhiệm –predecessor (kế nhiệm-successor). Nút tiền nhiệm (cũng như kế nhiệm) này chỉ có thể có một nút con hoặc không có nút con nào, việc xoá nó sau đó được thực hiện đơn giản.

-Trong phân tích trên, hàm Delete sẽ cần tới một thủ tục xoá đi một nút, để chương trình được sáng sủa ta viết hàm DeleteNode để xoá đi một nút, hàm này sẽ được Delete gọi tới, và một hàm GetPredecessor để tìm người tiền nhiệm.

8.PrintOrderTree

+Chức năng : In ra các phần tử theo kiểu duyệt cây.

-Để duyệt một danh sách tuyến tính chỉ có hai cách : tiến (forward) hoặc lùi (backward). Song đối với cây nhị phân ta có nhiều cách ,phổ dụng nhất là :

+ Duyệt theo thứ tự trước (PreOrder) .

+ Duyệt theo thứ tự giữa (InOrder).

+ Duyệt theo thứ tự sau (PostOrder).

-Mỗi cách duyệt thể hiện ưu điểm riêng của nó tùy trong trường hợp cụ thể. Chẳng hạn như để in ra các phần tử theo thứ tự tăng dần ta duyệt cây theo thứ tự giữa, hay để huỷ một cây thì duyệt theo thứ tự sau là thích hợp nhất, mỗi nút sẽ được xoá đi nếu như cây con trái và cây con phải của nó đã được phá huỷ.

-Việc duyệt cây có thể được định nghĩa đệ quy một cách khái quát như sau :

PreOrder(tree)


```

Nếu tree # NULL thì :
Thăm tree->info;
PreOrder(tree->left);
PreOrder(tree->right);

```

```

InOrder(tree)
Nếu tree # NULL thì :
InOrder(tree->left);
Thăm tree->info;
InOrder(tree → left);

```

```

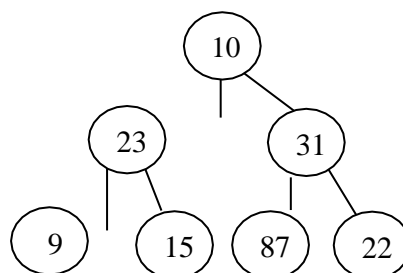
PostOrder(tree)
Nếu tree # NULL thì :
PostOrder(tree->left);
PostOrder(tree->right);
Thăm tree->info;

```

4.3.4. Thao tác trên cây nhị phân cân bằng

4.3.3. Thao tác trên cây nhị phân đầy đủ

- Cây nhị phân đầy đủ: Là cây nhị phân mà mỗi nút không phải lá đều có đúng 2 nút con và các nút lá phải có cùng độ sâu.



Hình 5.6 Cây nhị phân đầy đủ

4.4. CASE STUDY

4.4.1. Cài đặt chương trình máy tính các thuật toán trên cây nhị phân tổng quát.

Cài đặt cây nhị phân sau, sau đó duyệt theo thứ tự trước, thứ tự sau và thứ tự giữa:

```

#include <iostream>
#include <stdio.h>
#include <conio.h>

typedef struct Tree
{
    char data;

```

```

        Tree * LeftChild, * RightChild;
};

Tree * root;

void AddChildNode(Tree * root, Tree * LeftChild, Tree * RightChild)
{
    root->LeftChild = LeftChild;
    root->RightChild = RightChild;
}

int IsChild(Tree * p, Tree * q) // Kiem tra xem q co phai la hau due
cua p hay khong
{
    if (p == NULL) return 0;
    else if (p == q) return 1;
    else return IsChild(p->LeftChild, q) || IsChild(p->RightChild,
q);
}

int SizeOfNode(Tree * p) // Find the size of node p
{
    if (p == NULL) return 0;
    else return SizeOfNode(p->LeftChild) + SizeOfNode(p-
>RightChild) + 1;
}

void CreatTree()
{
    root = new Tree[14]; // Chi su dung cac phan tu tu 1 toi 13

    // Khoi tao cay
    // Khoi tao gia tri
    int i;
    for (i=1; i<= 13; i++) root[i].data = char(65+i-1);
    // Tao cac moi noi
    AddChildNode(&root[1], &root[2], &root[3]);
    AddChildNode(&root[2], &root[4], &root[5]);
    AddChildNode(&root[4], NULL, NULL);
    AddChildNode(&root[5], &root[8], NULL);
    AddChildNode(&root[8], &root[12], &root[13]);
    AddChildNode(&root[12], NULL, NULL);
    AddChildNode(&root[13], NULL, NULL);
    AddChildNode(&root[3], &root[6], &root[7]);
}

```

```

    AddChildNode(&root[6], &root[9], &root[10]);
    AddChildNode(&root[9], NULL, NULL);
    AddChildNode(&root[10], NULL, NULL);
    AddChildNode(&root[7], NULL, &root[11]);
    AddChildNode(&root[11], NULL, NULL);
}

void PreOrder(Tree * r)
{
    if (r == NULL) return;
    printf("%c", r->data);
    PreOrder(r->LeftChild);
    PreOrder(r->RightChild);
}

void InOrder(Tree * r)
{
    if (r == NULL) return;
    InOrder(r->LeftChild);
    printf("%c", r->data);
    InOrder(r->RightChild);
}

void PostOrder(Tree * r)
{
    if (r == NULL) return;
    PostOrder(r->LeftChild);
    PostOrder(r->RightChild);
    printf("%c", r->data);
}

int main()
{
    CreatTree();
    printf("Size of the node 5 is %d\n", SizeOfNode(&root[5]));
    printf("Size of the node 3 is %d\n\n", SizeOfNode(&root[3]));

    if (IsChild(&root[3], &root[10])) printf("The node 3 is the father
of the node 10\n");
    else printf("The node 3 is not the father of the node 10\n");

    if (IsChild(&root[2], &root[9])) printf("The node 2 is the father of
the node 9");
    else printf("The node 2 is not the father of the node 9");
}

```

```

printf("\n\nPreOrder:\n");
PreOrder(&root[1]);

printf("\n\nInOrder:\n");
InOrder(&root[1]);

printf("\n\nPostOrder:\n");
PostOrder(&root[1]);

getch();
}

```

4.4.2. Cài đặt chương trình máy tính các thuật toán trên cây nhị phân tìm kiếm.

```

1. // Khai bao lop ADT Cay nhi phan tim kiem.
2. #include<iostream>
3.
4. struct TreeNode;
5.
6. enum OrderType { PRE_ORDER, IN_ORDER, POST_ORDER };
7.
8. typedef int ItemType;
9. // Gia su rang thanh phan key chua trong moi nut co kieu don gian int
10. // Doi voi nhung kieu phuc tap hon ta co the dinh nghia tuong tu.
11.
12. #ifndef TREETYPE_H
13. #define TREETYPE_H
14.
15. class TreeType
16. {
17. public:
18.     TreeType(); // Constructor.
19.     ~TreeType(); // Destructor.
20.     TreeType( const TreeType& originalTree );// Copy Constructor.
21.
22.     void operator=( const TreeType& originalTree );
23.     // Ham nap chong toan tu.
24.     void MakeEmpty(); // Dua cay ve trang thai rong.
25.     bool IsEmpty() const; // Kiem tra xem cay co rong hay khong.

```

```

25.      bool IsFull() const; // Kiem tra xem cay co
      day hay khong.
26.      int LengthTree(); // Ham tra lai so phan tu
      co tren cay.
27.      int DepthTree(); // Ham tra lai chieu sau
      cua cay.
28.
      void RetrieveItem( ItemType& item, bool &found );
29.      // Ham truy luc toi phan tu tren cay co key
      giống voi key của item.
30.      void InsertItem( ItemType item ); // Chen
      mot phan tu vao cay.
31.      void DeleteItem( ItemType item);
32.      // Xoa phan tu tren cay co key giống voi key
      của item.
33.      void Print( std::ofstream& outFile);
34.      // In ra outFile cac phan tu theo thu tu
      tang dan.
35.      void PrintOrderTree( OrderType order );
36.      // In ra noi dung cac nut theo kieu duyet
      order.
37.  private:
38.      TreeNode *root;
39.  };
40.
41.  #endif

```

4.4.3. Cài đặt chương trình máy tính các thuật toán trên cây nhị phân cân bằng.

```

1. // Test thu cac ham thao tac tren Cay nhi phan can
   bang.
2. #include<iostream>
3. #include<iomanip>
4.
5. using namespace std;
6.
7. #include "TreeType.h"
8.
9. int main()
10. {
11.     TreeType myTree;
12.     ItemType item;
13.     OrderType order;
14.     int choice, option;
15.     bool found;
16.

```

```

17.         cout << setw(15) << " " << "CHUONG   TRINH
TEST THU CAY NHI PHAN TIM KIEM\n"
18.                                     << setw(15) << "
" << "=====
19.         << "\n\n\n";
20.         cout << setw(20) << " " << "1:Tao cay ADT
BST\n"
21.         << setw(20) << " " << "2:Chen phan tu
moi vao cay BST\n"
22.         << setw(20) << " " << "3:Xoa phan tu
khoi cay BST\n"
23.         << setw(20) << " " << "4:Truy tim phan
tu\n"
24.         << setw(20) << " " << "5:In ra tong so
phan tu\n"
25.         << setw(20) << " " << "6:In ra chieu
sau cua cay\n"
26.         << setw(20) << " " << "7:In ra cac phan
tu theo kieu duyet\n"
27.         << setw(20) << " " << "8:Huy cay da
duoc tao\n"
28.         << setw(20) << " " << "9:Thoat khoi ung
dung\n\n";
29.
30.         while(1)
31.         {
32.             cout << "Nhap vao lua chon cua ban:";
33.             cin >> choice;
34.
35.             switch(choice)
36.             {
37.                 case 1: int number;
38.                     cout << "Nhap so luong nut :";
39.                     cin >> number;
40.
41.                     while(number > 0)
42.                     {
43.                         cout << "Nhap gia tri :";
44.                         cin >> item;
45.
46.                         myTree.InsertItem(item);
47.                         number--;
48.                     }
49.                     break;
50.

```

```

51.         case 2: cout << "Nhap vao mot so nguyen
52.             :";
53.             cin >> item;
54.             myTree.InsertItem(item);
55.             break;
56.
57.         case 3: cout << "Nhap vao phan tu
58.             can xoa :";
59.             cin >> item;
60.             myTree.DeleteItem(item);
61.             break;
62.
63.         case 4: cout << "Nhap phan tu can
64.             tim kiem :";
65.             cin >> item;
66.             myTree.RetrieveItem(item, found);
67.             if( found == true)
68.                 cout << "Phan tu co mat tren
69.                 cay !\n";
70.             else
71.                 cout << "Phan tu khong co
72.                 tren cay !\n";
73.             break;
74.
75.         case 5: cout << "So phan tu hien co
76.             tren cay la: "
77.             << myTree.LengthTree() << endl;
78.             break;
79.
80.         case 6: cout << "Chieu sau cua cay
81.             la: "
82.             << myTree.DepthTree() << endl;
83.             break;
84.
85.         case 7: cout << setw(20) << "
86.             " << "1:Duyet theo thu tu truoc\n"
87.             << setw(20) << "
88.             " << "2:Duyet theo thu tu giua\n"
89.             << setw(20) << "
90.             " << "3:Duyet theo thu tu sau\n";
91.

```

```

85.             cout << "Nhap vao lua chon cua
ban:";
86.             cin >> option;
87.
88.             order = (OrderType) (option-1);
89.
90.             if( myTree.IsEmpty() )
91.             {
92.                 cout << "Cay rong!\n";
93.                 break;
94.             }
95.
96.             else
97.
myTree.PrintOrderTree(order) ;
98.                 cout << endl;
99.                 break;
100.            case 8: myTree.MakeEmpty() ;
101.                break;
102.            case 9: return 0;
103.
104.        }
105.    }
106.
107.    return 0;
108. }
```

4.4.4. Tìm hiểu các thao tác trên cây nhị phân trong thư viện của ngôn ngữ lập trình C++.

Chương 5. Đồ thị

5.1. Khái niệm và định nghĩa

Một đồ thị (đồ thị) là một dạng biểu diễn hình ảnh của một tập các đối tượng, trong đó các cặp đối tượng được kết nối bởi các link. Các đối tượng được nối liền nhau được biểu diễn bởi các điểm được gọi là **các đỉnh (vertices)**, và các link mà kết nối các đỉnh với nhau được gọi là **các cạnh (edges)**.

5.2. Biểu diễn đồ thị

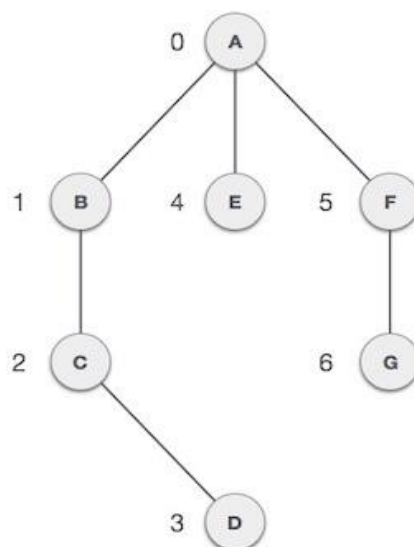
Các hình toán học có thể được biểu diễn trong cấu trúc dữ liệu. Chúng ta có thể biểu diễn một hình bởi sử dụng một mảng các đỉnh và một mảng hai chiều của các cạnh. Trước khi tiếp tục, chúng ta tìm hiểu một vài khái niệm quan trọng sau:

Đỉnh (Vertex): Mỗi nút của hình được biểu diễn như là một đỉnh. Trong ví dụ dưới đây, các hình tròn biểu diễn các đỉnh. Do đó, các điểm từ A tới G là các đỉnh. Chúng ta có thể biểu diễn các đỉnh này bởi sử dụng một mảng, trong đó đỉnh A có thể được nhận diện bởi chỉ mục 0, điểm B là chỉ mục 1, ... như hình dưới.

Cạnh (Edge): Cạnh biểu diễn một đường nối hai đỉnh. Trong hình dưới, các đường nối A và B, B và C, ... là các cạnh. Chúng ta có thể sử dụng một mảng hai chiều để biểu diễn các cạnh này. Trong ví dụ dưới, AB có thể được biểu diễn như là 1 tại hàng 0; BC là 1 tại hàng 1, cột 2, ...

Kề nhau: Hai đỉnh là kề nhau nếu chúng được kết nối với nhau thông qua một cạnh. Trong hình dưới, B là kề với A; C là kề với B, ...

Đường: Đường biểu diễn một dãy các cạnh giữa hai đỉnh. Trong hình dưới, ABCD biểu diễn một đường từ A tới D.



5.3. Các thuật toán tìm kiếm trên đồ thị

Following are basic primary operations of a Graph which are following.

Thêm đỉnh: Thêm một đỉnh vào trong đồ thị.

Thêm cạnh: Thêm một cạnh vào giữa hai đỉnh của một đồ thị.

Hiển thị đỉnh: Hiển thị một đỉnh của một đồ thị.

5.4. Một số bài toán tối ưu trên đồ thị

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

Cho đồ thị G với tập các đỉnh V và tập các cạnh E (đồ thị có hướng hoặc vô hướng). Mỗi cạnh của đồ thị có một nhãn, đó là một giá trị không âm, nhãn này còn gọi là giá (cost) của cạnh. Cho trước một đỉnh v xác định, gọi là đỉnh nguồn. Vấn đề là tìm đường đi ngắn nhất từ v đến các đỉnh còn lại của G ; tức là các đường đi từ v đến các đỉnh còn lại với tổng các giá (cost) của các cạnh trên đường đi là nhỏ nhất. Chú ý rằng nếu đồ thị có hướng thì đường đi này là đường đi có hướng.

Ta có thể giải bài toán này bằng cách xác định một tập hợp S chứa các đỉnh mà khoảng cách ngắn nhất từ nó đến đỉnh nguồn v đã biết. Khởi đầu $S = \{v\}$, sau đó tại mỗi bước ta sẽ thêm vào S các đỉnh mà khoảng cách từ nó đến v là ngắn nhất. Với giả thiết mỗi cung có một giá không âm thì ta luôn luôn tìm được một đường đi ngắn nhất như vậy mà chỉ đi qua các đỉnh đã tồn tại trong S . Để chi tiết hoá giải thuật, giả sử G có n đỉnh và nhãn trên mỗi cung được lưu trong mảng hai chiều C , tức là $C[i,j]$ là giá (có thể xem như độ dài) của cung (i,j) , nếu i và j không nối nhau thì $C[i,j] = \infty$. Ta dùng mảng 1 chiều D có n phần tử để lưu độ dài của đường đi ngắn nhất từ mỗi đỉnh của đồ thị đến v . Khởi đầu khoảng cách này chính là độ dài cạnh (v,i) , tức là $D[i] = C[v,i]$. Tại mỗi bước của giải thuật thì $D[i]$ sẽ được cập nhật lại để lưu độ dài đường đi ngắn nhất từ đỉnh v tới đỉnh i , đường đi này chỉ đi qua các đỉnh đã có trong S .

Để cài đặt giải thuật dễ dàng, ta giả sử các đỉnh của đồ thị được đánh số từ 1 đến n , tức là $V = \{1, \dots, n\}$ và đỉnh nguồn là 1. Dưới đây là giải thuật Dijkstra để giải bài toán trên.

```
void Dijkstra()
{
    S = [1]; //Tập hợp S chỉ chứa một đỉnh nguồn
    for (i=2; i<=n; i++)
        D[i-1] = C[0,i-1]; //khởi đầu các giá trị cho D
    for (i=1; i<n; i++)
    {
        Lấy đỉnh w trong V-S sao cho D[w-1] nhỏ nhất;
        Thêm w vào S;
        for (mỗi đỉnh u thuộc V-S)
            D[u-1] = min(D[u-1], D[w-1] + C[w-1,u-1]);
    }
}
```

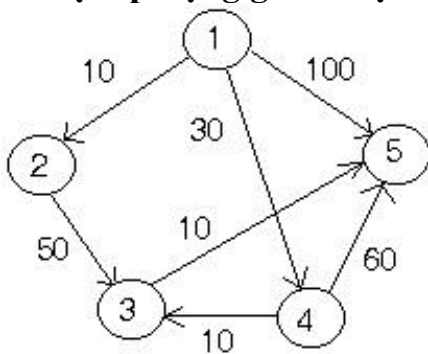
Nếu muốn lưu trữ lại các đỉnh trên đường đi ngắn nhất để có thể xây dựng lại đường đi này từ đỉnh nguồn đến các đỉnh khác, ta dùng một mảng P . Mảng này sẽ

lưu $P[u]=w$ với u là đỉnh "trước" đỉnh w trong đường đi. Lúc khởi đầu $P[u]=1$ với mọi u .

Giải thuật Dijkstra được viết lại như sau:

```
void Dijkstra()
{
    S=[1]; //S chỉ chứa một đỉnh nguồn
    for(i=2; i<=n; i++)
    {
        P[i-1]=1; //khởi tạo giá trị cho P
        D[i-1]=C[0,i-1]; //khởi đầu các giá trị cho D
    }
    for (i=1; i<n; i++)
    {
        Lấy đỉnh w trong V-S sao cho D[w-1] nhỏ nhất;
        Thêm w vào S;
        for (mỗi đỉnh u thuộc V-S)
            if (D[w-1] + C[w-1,u-1] < D[u-1])
            {
                D[u-1] = D[w-1] + C[w-1,u-1];
                P[u-1] = w;
            }
    }
}
```

Ví dụ: áp dụng giải thuật Dijkstra cho đồ thị hình V.5



Hình V.5

Kết quả khi áp dụng giải thuật

Lần lặp	S	W	D[2]	D[3]	D[4]	D[5]
Khởi đầu	{1}	-	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	40	30	90
3	{1,2,3,4}	3	10	40	30	50
4	{1,2,3,4,5}	5	10	40	30	50

Mảng P có giá trị như sau:

P	1	2	3	4	5
		1	4	1	3

Từ kết quả trên ta có thể suy ra rằng đường đi ngắn nhất từ đỉnh 1 đến đỉnh 3 là $1 \rightarrow 4 \rightarrow 3$ có độ dài là 40. đường đi ngắn nhất từ 1 đến 5 là $1 \rightarrow 4 \rightarrow 3 \rightarrow 5$ có độ dài 50.

Tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh

Giả sử đồ thị G có n đỉnh được đánh số từ 1 đến n. Khoảng cách hay giá giữa các cặp đỉnh được cho trong mảng C[i,j]. Nếu hai đỉnh i,j không được nối thì C[i,j]= ∞ . Giải thuật Floyd xác định đường đi ngắn nhất giữa hai cặp đỉnh bất kỳ bằng cách lặp k lần, ở lần lặp thứ k sẽ xác định khoảng cách ngắn nhất giữa hai đỉnh i,j theo công thức: $A_k[i,j] = \min(A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j])$. Ta cũng dùng mảng P để lưu các đỉnh trên đường đi.

```
float A[n,n], C[n,n];
int P[n,n];
void Floyd()
{
    int i,j,k;
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
        {
            A[i-1,j-1] = C[i-1,j-1];
            P[i-1,j-1]=0;
        }
    for (i=1; i<=n; i++)
        A[i-1,i-1]=0;
    for (k=1; k<=n; k++)
        for (i=1; i<=n; i++)
            for (j=1; j<=n; j++)
                if (A[i-1,k-1] + A[k-1,j-1] < A[i-1,j-1])
                {
                    A[i-1,j-1] = A[i-1,k-1] + A[k-1,j-1];
                    P[i-1,j-1] = k;
                }
}
```

Bài toán tìm bao đóng chuyển tiếp (transitive closure)

Trong một số trường hợp ta chỉ cần xác định có hay không có đường đi nối giữa hai đỉnh i,j bất kỳ. Giải thuật Floyd có thể đặc biệt hoá để giải bài toán này. Bây giờ khoảng cách giữa i,j là không quan trọng mà ta chỉ cần biết i,j có nối nhau không do đó ta cho $C[i,j]=1$ (~true) nếu i,j được nối nhau bởi một cạnh, ngược lại $C[i,j]=0$ (~false). Lúc này mảng A[i,j] không cho khoảng cách ngắn nhất giữa i,j mà nó cho biết là có đường đi từ i đến j hay không. A gọi là bao đóng chuyển tiếp của đồ thị G có biểu diễn ma trận kề là C. Giải thuật Floyd sửa đổi như trên gọi là giải thuật Warshall.

```
int A[n,n], C[n,n];
void Warshall()
{

```

```

int i,j,k;
for (i=1; i<=n; i++)
for (j=1; j<=n; j++)
A[i-1,j-1] = C[i-1,j-1];
for (k=1; k<=n; k++)
for (i=1; i<=n; i++)
for (j=1; j<=n; j++)
if (A[i-1,j-1] == 0) then
A[i-1,j-1] = A[i-1,k-1] && A[k-1,j-1];
}

```

Bài toán tìm cây bao trùm tối thiểu (minimum-cost spanning tree)

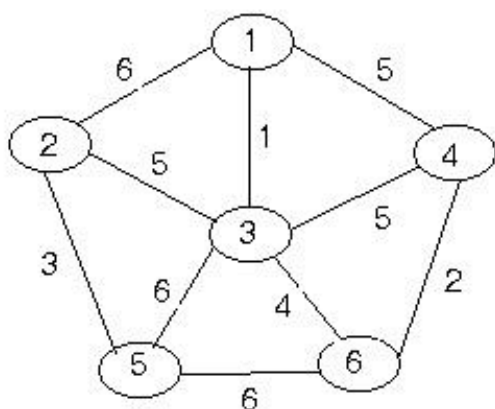
Giả sử ta có một đồ thị vô hướng $G=(V,E)$. Đồ thị G gọi là liên thông nếu tồn tại đường đi giữa hai đỉnh bất kỳ. Bài toán tìm cây bao trùm tối thiểu (hoặc cây phủ tối thiểu) là tìm một tập hợp T chứa các cạnh của một đồ thị liên thông G sao cho V cùng với tập các cạnh này cũng là một đồ thị liên thông, tức là (V,T) là một đồ thị liên thông. Hơn nữa tổng độ dài các cạnh trong T là nhỏ nhất. Một thể hiện của bài toán này trong thực tế là bài toán thiết lập mạng truyền thông, ở đó các đỉnh là các thành phố còn các cạnh của cây bao trùm là đường nối mạng giữa các thành phố.

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

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

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

Ví dụ, áp dụng giải thuật Prim để tìm cây bao trùm tối thiểu của đồ thị liên thông hình V.6.



Hình V.6

Bước khởi đầu: $U=\{1\}$, $T=\emptyset$.

Bước kế tiếp ta có cạnh $(1,3)=1$ là cạnh ngắn nhất thỏa mãn điều kiện trong giải thuật Prim nên: $U=\{1,3\}$, $T=\{(1,3)\}$.

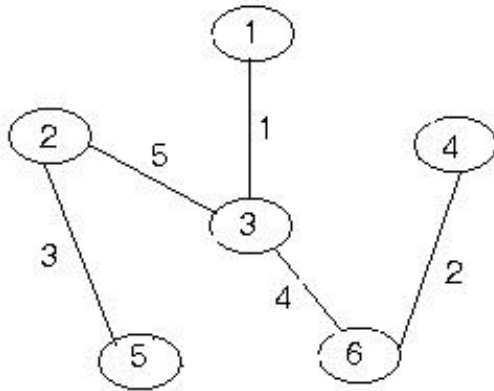
Kế tiếp thì cạnh $(3,6)=4$ là cạnh ngắn nhất thỏa mãn điều kiện trong giải thuật Prim nên: $U=\{1,3,6\}$, $T=\{(1,3),(3,6)\}$.

Kế tiếp thì cạnh $(6,4)=2$ là cạnh ngắn nhất thỏa mãn điều kiện trong giải thuật

Prim nên: $U=\{1,3,6,4\}$, $T=\{(1,3),(3,6),(6,4)\}$.

Tiếp tục, cạnh $(3,2)=5$ là cạnh ngắn nhất thoả mãn điều kiện trong giải thuật Prim nên: $U=\{1,3,6,4,2\}$, $T=\{(1,3),(3,6),(6,4),(3,2)\}$.

Cuối cùng, cạnh $(2,5)=3$ là cạnh ngắn nhất thoả mãn điều kiện trong giải thuật Prim nên: $U=\{1,3,6,4,2,5\}$, $T=\{(1,3),(3,6),(6,4),(3,2),(2,5)\}$. Giải thuật dừng và ta có cây bao trùm như trong hình V.7.



Hình V.7

Giải thuật Prim được viết lại như sau:

```
void Prim(graph G, set_of_edges *T)
{
    set_of_vertices U;           //tập hợp các đỉnh
    vertex u,v;                  //u,v là các đỉnh

    T = ∅;
    U = [1];

    while (U≠V) do               // V là tập hợp các đỉnh của G
    {
        gọi (u,v) là cạnh ngắn nhất sao cho u ∈ U và v ∈ V-U;

        U = U ∪ [v];
        T = T ∪ [(u,v)];
    }
}
```

Bài toán cây bao trùm tối thiểu còn có thể được giải bằng giải thuật Kruskal như sau:

Khởi đầu ta cũng cho $T = \emptyset$ giống như trên, ta thiết lập đồ thị khởi đầu $G'=(V,T)$.

Xét các cạnh của G theo thứ tự độ dài tăng dần. Với mỗi cạnh được xét ta sẽ đưa nó vào T nếu nó không làm cho G' có chu trình.

Ví dụ áp dụng giải thuật Kruskal để tìm cây bao trùm cho đồ thị hình V.6.

Các cạnh của đồ thị được xếp theo thứ tự tăng dần là.

$(1,3)=1, (4,6)=2, (2,5)=3, (3,6)=4, (1,4)=(2,3)=(3,4)=5, (1,2)=(3,5)=(5,6)=6$.

Bước khởi đầu $T = \emptyset$

Lần lặp 1: $T = \{(1,3)\}$

Lần lặp 2: $T = \{(1,3), (4,6)\}$

Lần lặp 3: $T = \{(1,3), (4,6), (2,5)\}$

Lần lặp 4: $T = \{(1,3), (4,6), (2,5), (3,6)\}$

Lần lặp 5:

Cạnh $(1,4)$ không được đưa vào T vì nó sẽ tạo ra chu trình $1,3,6,4,1$.

Kế tiếp cạnh $(2,3)$ được xét và được đưa vào T .

$T = \{(1,3), (4,6), (2,5), (3,6), (2,3)\}$

Không còn cạnh nào có thể được đưa thêm vào T mà không tạo ra chu trình.

Vậy ta có cây bao trùm tối thiểu cũng giống như trong hình V.7.

5.5. CASE STUDY

5.5.1. Cài đặt chương trình máy tính các thuật toán tìm kiếm trên đồ thị.

a) Tìm kiếm theo chiều sâu

Cài đặt bằng ngôn ngữ C++

So sánh với BFS, chúng ta có thể cài đặt DFS dễ dàng với một hàm đệ quy.

Tùy theo những gì bạn cần làm trên đồ thị mà bạn có thể điều chỉnh code cho phù hợp.

```
vectorg[maxn]; // vector lưu các đỉnh kề với các đỉnh
intdd[maxn]; // mảng đánh dấu

voiddfs(intu)
{
    dd[u]=1; // đánh dấu đỉnh u đã đi qua
    for(inti=0;i<g[u].size();++i) // với mỗi đỉnh v kề với u
    {
        intv=g[u][i];
        if(dd[v]==0)dfs(v); // nếu v chưa đánh dấu, tới thăm đỉnh v
    }
}
```

}

b) Tìm kiếm theo chiều rộng

Cách cài đặt giải thuật :

- fringe : là một cấu trúc kiểu hàng đợi (FIFO) lưu giữ các nút sẽ được duyệt
 - closed : cấu trúc hàng đợi queue lưu giữ các nút đã được duyệt .
 - $G(N, A)$: cây biểu diễn không gian trạng thái bài toán.
 - no : trạng thái đầu của bài toán
 - DICH : tập hợp các trạng thái đích của bài toán.
 - $NB(n)$: tập hợp các trạng thái con của của nút đang xét n
- sau đây là giải thuật :

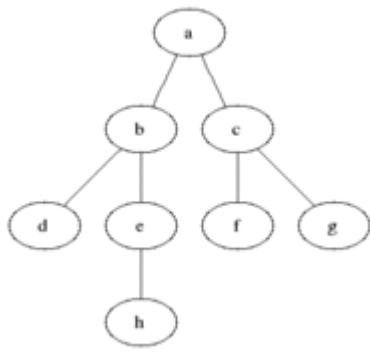
BFS (N, A, n_0, DICH)

```
{
  fringe  $\leftarrow$   $n_0$ ;
  closed  $\leftarrow$   $\emptyset$ ;
  while (fringe  $\neq$   $\emptyset$ ) do
  {
     $n \leftarrow \text{GET\_FIRST}(\text{fringe});$       // lấy phần tử đầu tiên của fringe
    closed  $\leftarrow$  closed  $\oplus$  n;
    if ( $n \in \text{DICH}$ ) then return SOLUTION(n);
    if ( $\Gamma(n) \neq \emptyset$ ) then fringe  $\leftarrow$  fringe  $\oplus$   $\Gamma(n)$ ;
  }
  return ("No solution");
}
```

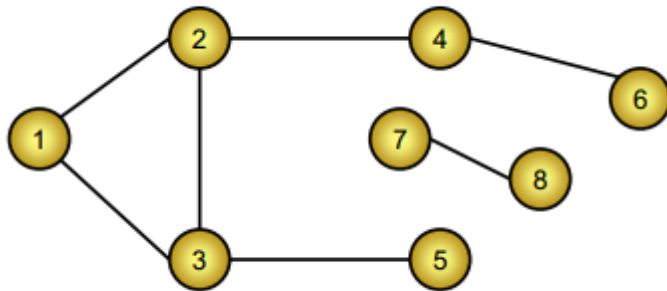
Mô tả :

1. C
hèn đỉnh gốc no vào hàng đợi
2. L
ấy ra đỉnh đầu tiên trong hàng đợi và thăm nó
 - Nếu đỉnh này chính là đỉnh đích, dừng quá trình tìm kiếm và trả về kết quả.
 - Nếu không phải thì chen tất cả các đỉnh kề với đỉnh vừa thăm nhưng chưa được thăm trước đó vào hàng đợi.
3. N
ếu hàng đợi là rỗng, thì tất cả các đỉnh có thể đến được đều đã được thăm – dừng việc tìm kiếm và trả về “không thấy”.
4. N
ếu hàng đợi không rỗng thì quay về bước 2.

ví dụ minh họa :



Giả sử ta có đồ thị :



5.5.2. Cài đặt chương trình máy tính các thuật toán tối ưu trên đồ thị.

5.5.3. Tìm hiểu thao tác trên đồ thị trong thư viện của ngôn ngữ lập trình C⁺⁺.

Chương 6. Sắp xếp và tìm kiếm

Sắp xếp và tìm kiếm là các vấn đề rất cơ bản trong tin học cũng như trong thực tiễn. Chương 6 giới thiệu các phương pháp sắp xếp và tìm kiếm thông dụng nhất, bao gồm các giải thuật từ đơn giản đến phức tạp.

Đối với các giải thuật sắp xếp, các phương pháp sắp xếp đơn giản được trình bày bao gồm: sắp xếp chọn, sắp xếp chèn, sắp xếp nổi bọt. Các phương pháp sắp xếp phức tạp và hiệu quả hơn được xem xét là giải thuật sắp xếp nhanh (quick sort), sắp xếp vun đống (heap sort) và sắp xếp trộn (merge sort). Với mỗi phương pháp sắp xếp, ngoài việc trình bày các bước thực hiện thuật toán, độ phức tạp của giải thuật cũng được tính toán và đánh giá.

Đối với các phương pháp tìm kiếm, ngoài phương pháp tìm kiếm tuần tự đơn giản, các phương pháp tìm kiếm phức tạp và hiệu quả hơn cũng được xem xét là tìm kiếm nhị phân và tìm kiếm bằng cây nhị phân tìm kiếm.

Để học tốt chương này, sinh viên cần nghiên cứu kỹ các bước thực hiện các thuật toán và lấy ví dụ cụ thể, sau đó thực hiện từng bước trên ví dụ.

6.1. Các thuật toán sắp xếp đơn giản

6.1.1. Sắp xếp kiểu đổi chỗ

Đây là một trong những giải thuật sắp xếp đơn giản nhất. Ý tưởng của giải thuật như sau:

Lựa chọn phần tử có giá trị nhỏ nhất, đổi chỗ cho phần tử đầu tiên. Tiếp theo, lựa chọn phần tử có giá trị nhỏ thứ nhì, đổi chỗ cho phần tử thứ 2. Quá trình tiếp tục cho tới khi toàn bộ dãy được sắp.

Ví dụ, các bước thực hiện sắp xếp chọn dãy số bên dưới như sau:

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

Bước 1: Chọn được phần tử nhỏ nhất là 06, đổi chỗ cho 32.

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

Bước 2: Chọn được phần tử nhỏ thứ nhì là 17, đó chính là phần tử thứ 2 nên giữ nguyên.

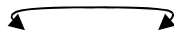
0	1	4	9	3	2	5	6
6	7	9	8	2	5	3	1

Bước 3: Chọn được phần tử nhỏ thứ ba là 25, đổi chỗ cho 49.



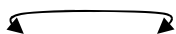
0	1	2	9	3	4	5	6
6	7	5	8	2	9	3	1

Bước 4: Chọn được phần tử nhỏ thứ tư là 32, đổi chỗ cho 98.



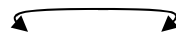
0	1	2	3	9	4	5	6
6	7	5	2	8	9	3	1

Bước 5: Chọn được phần tử nhỏ thứ năm là 49, đổi chỗ cho 98.



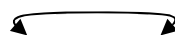
	0	1	2	3	4	9	5	6
6	7	5	2	9	8	3	1	6

Bước 6: Chọn được phần tử nhỏ thứ sáu là 53, đổi chỗ cho 98.



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

Bước 7: Chọn được phần tử nhỏ thứ bảy là 61, đổi chỗ cho 98.



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

Như vậy, toàn bộ dãy đã được sắp.

Giải thuật được gọi là sắp xếp chọn vì tại mỗi bước, một phần tử được chọn và đổi chỗ cho phần tử ở vị trí cần thiết trong dãy.

Thủ tục thực hiện sắp xếp chọn trong C như sau:

```
void selection_sort() {
    int i, j, k, temp;
    for (i = 0; i < N; i++) { k
= i;
        for (j = i+1; j < N; j++) { if
(a[j] < a[k]) k =j;
        }
        temp = a[i]; a[i] =a [k]; a[k] = temp;
    }
}
```

Trong thủ tục trên, vòng lặp đầu tiên duyệt từ đầu đến cuối dãy. Tại mỗi vị trí i, tiến hành duyệt tiếp từ i tới cuối dãy để chọn ra phần tử nhỏ thứ i và đổi chỗ cho phần tử ở vị trí i.

Thời gian thực hiện thuật toán tỷ lệ với N^2 , vì vòng lặp ngoài (biến chạy i) duyệt qua N phần tử, và vòng lặp trong duyệt trung bình $N/2$ phần tử. Do đó, độ phức tạp trung bình của thuật toán là $O(N * N/2) = O(N^2/2) = O(N^2)$.

6.1.2. Sắp xếp kiểu chèn trực tiếp

Giải thuật này coi như dãy được chia làm 2 phần. Phần đầu là các phần tử đã được sắp. Từ phần tử tiếp theo, chèn nó vào vị trí thích hợp tại nửa đã sắp sao cho nó vẫn được sắp.

Để chèn phần tử vào nửa đã sắp, chỉ cần dịch chuyển các phần tử lớn hơn nó sang trái 1 vị trí và đưa phần tử này vào vị trí trống trong dãy.

Ví dụ, nửa dãy đã sắp là:

	0	1	4	9
6	7	9	8	

Để chèn phần tử 32 vào nửa dãy này, ta tiến hành dịch chuyển các phần tử lớn hơn 32 về bên trái 1 vị trí:

	0	1		4	9
6	7		9	8	

Sau đó, chèn 32 vào vị trí trống trong nửa dãy:

	0	1	3	4	9
6	7	2	9	8	

Quay trở lại với dãy số ở phần trước, các bước thực hiện sắp xếp chèn trên dãy như sau: Dãy ban đầu: Nửa đã sắp trống, nửa chưa sắp là toàn bộ dãy.

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

Bước 1: Chèn phần tử đầu của nửa chưa sắp là 32 vào nửa đã sắp. Do nửa đã sắp là trống nên có thể chèn vào vị trí bất kỳ.

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

Đã sắp

 Chưa sắp

Bước 2: Chèn phần tử 17 vào nửa đã sắp. Dịch chuyển 32 sang phải 1 vị trí và đưa 17 vào vị trí trống.

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

Đã sắp

 Chưa sắp

Bước 3, 4: Lần lượt chèn phần tử 49, 98 vào nửa đã sắp.

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

Đã sắp

 Chưa sắp

Bước 5: Chèn phần tử 06 vào nửa đã sắp. Dịch chuyển các phần tử 17, 32, 49, 98 sang phải 1 vị trí và đưa 06 vào vị trí trống.

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

Đã sắp

 Chưa sắp

Bước 6: Chèn phần tử 25 vào nửa đã sắp. Dịch chuyển các phần tử 32, 49, 98 sang phải 1 vị trí và đưa 25 vào vị trí trống.

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

Đã sắp

Chưa sắp

Bước 7: Chèn phần tử 53 vào nửa đã sắp. Dịch chuyển phần tử 98 sang phải 1 vị trí và đưa 53 vào vị trí trống.

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

Đã sắp

Chưa sắp

Bước 8: Chèn phần tử cuối cùng 61 vào nửa đã sắp. Dịch chuyển phần tử 98 sang phải 1 vị trí và đưa 61 vào vị trí trống.

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

Đã sắp

Thủ tục thực hiện sắp xếp chèn trong C như sau:

```
void insertion_sort() {
    int i, j, k, temp;
    for (i = 1; i < N; i++) {
        temp = a[i];
        j = i - 1;
        while ((a[j] > temp) && (j >= 0)) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = temp;
    }
}
```

Thuật toán sử dụng 2 vòng lặp. Vòng lặp ngoài duyệt khoảng N lần, và vòng lặp trong duyệt trung bình N/4 lần (giả sử duyệt đến giữa nửa đã sắp thì gặp vị trí

cần chèn). Do đó, độ phức tạp trung bình của thuật toán là $O(N^2/4) = O(N^2)$.

6.1.3. Sắp xếp kiểu sủi bọt

Giải thuật sắp xếp nổi bọt được thực hiện theo nguyên tắc: Duyệt nhiều lần từ cuối lên đầu dãy, tiến hành đổi chỗ 2 phần tử liên tiếp nếu chúng ngược thứ tự. Đến một bước nào đó, khi không có phép đổi chỗ nào xảy ra thì toàn bộ dãy đã được sắp.

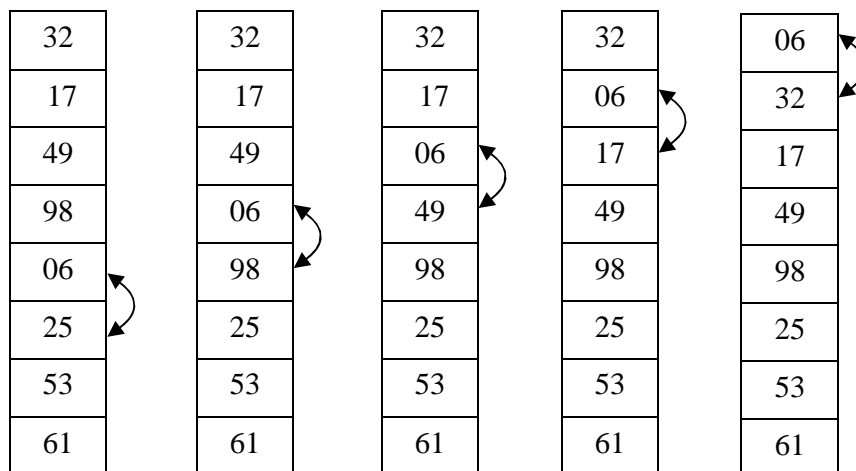
Như vậy, sau lần duyệt đầu tiên, phần tử nhỏ nhất của dãy sẽ lần lượt được đổi chỗ cho các phần tử lớn hơn và “nổi” lên đầu dãy. Lần duyệt thứ 2, phần tử nhỏ thứ 2 sẽ nổi lên vị trí thứ nhì dãy .v.v. Chú ý rằng, không nhất thiết phải tiến hành tất cả N lần duyệt, mà tới một lần duyệt nào đó, nếu không còn phép đổi chỗ nào xảy ra tức là tất cả các phần tử đã nằm đúng thứ tự và toàn bộ dãy đã được sắp.

Với dãy số như ở phần trước, các bước tiến hành giải thuật sắp xếp nổi bọt trên dãy như

sau:

Bước 1: Tại bước này, khi duyệt từ cuối dãy lên, lần lượt xuất hiện các cặp ngược thứ tự là

(06, 98), (06, 49), (06, 17), (06, 32). Phần tử 06 “nổi” lên đầu dãy.



Bước 2: Duyệt từ cuối dãy lên, lần lượt xuất hiện các cặp ngược thứ tự là (25, 98), (25, 49), (17, 32). Phần tử 17 nổi lên vị trí thứ 2.

06	06	06	06
32	32	32	17
17	17	17	32
49	49	25	25
98	25	49	49
25	98	98	98
53	53	53	53
61	61	61	61

Bước 3: Duyệt từ cuối dãy lên, lần lượt xuất hiện các cặp ngược thứ tự là (53, 98), (25, 32).

Phần tử 25 nổi lên vị trí thứ 3.

06	06	06
17	17	17
32	32	25
25	25	32
49	49	49
98	53	53
53	98	98
61	61	61

Bước 4: Duyệt từ cuối dãy lên, xuất hiện cặp ngược thứ tự là (61, 98).

06	06
17	17
25	25
32	32
49	49
53	53
98	61
61	98

Bước 5: Duyệt từ cuối dãy lên, không còn xuất hiện cặp ngược nào. Toàn bộ dãy đã được

Thủ tục thực hiện sắp xếp nổi bọt trong C như sau:

```
void bubble_sort() {
    int i, j, temp,
no_exchange; i = 1;
    do{

        no_exchange = 1;
        for (j=n-1; j >= i; j--){
            if (a[j-1] > a[j]){
temp=a[j-1];          a[j-1]=a[j];
a[j]=temp; }

            i++;
        } until (no_exchange || (i == n-1));
    }
```

Lần duyệt đầu tiên cần khoảng N-1 phép so sánh và đổi chỗ để làm nổi phần tử nhỏ nhất lên đầu. Lần duyệt thứ 2 cần khoảng N-2 phép toán, .v.v. Tổng cộng, số phép so sánh cần thực hiện là:

$$(N-1) + (N-2) + \dots + 2 + 1 = N(N-1)/2$$

Như vậy, độ phức tạp của giải thuật sắp xếp nổi bọt cũng là $O(N^2)$.

6.2. Thuật toán sắp xếp nhanh

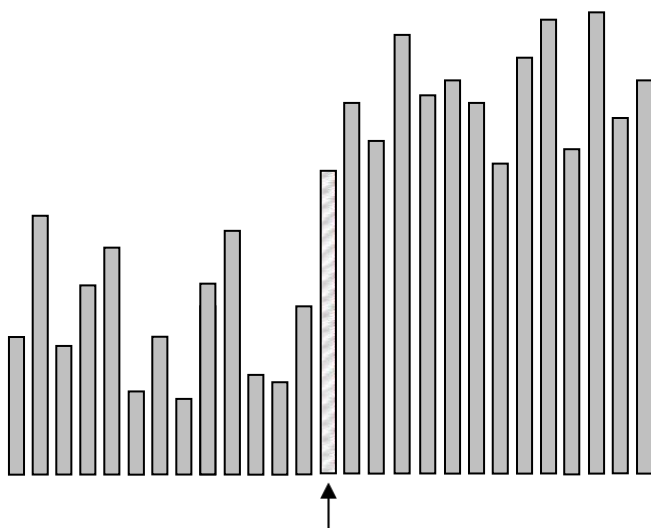
6.2.1. Giới thiệu thuật toán

Quick sort là một thuật toán sắp xếp được phát minh lần đầu bởi C.A.Hoare vào năm 1960. Đây có lẽ là thuật toán được nghiên cứu nhiều nhất và được sử dụng rộng rãi nhất trong lớp các thuật toán sắp xếp.

Quick sort là một thuật toán dễ cài đặt, hiệu quả trong hầu hết các trường hợp, và tiêu tốn ít tài nguyên hơn so với các thuật toán khác. Độ phức tạp trung bình của giải thuật là $O(N\log N)$. Nhược điểm của giải thuật này là phải cài đặt bằng đệ qui (có thể không dùng đệ qui, tuy nhiên cài đặt phức tạp hơn nhiều) và trong trường hợp xấu nhất thì độ phức tạp là $O(N^2)$. Ngoài ra, cài đặt cho Quick sort phải đòi hỏi cực kỳ chính xác. Chỉ cần một sai sót nhỏ có thể làm cho chương trình ngừng hoạt động.

Kể từ khi Quick sort ra đời lần đầu tiên, đã có rất nhiều nỗ lực nhằm cải tiến thuật toán này. Tuy nhiên, hầu hết các cải tiến này đều không mang lại hiệu quả như mong đợi, vì bản thân Quick sort là một thuật toán rất cân bằng. Một sự cải tiến ở một phần này của thuật toán có thể dẫn đến một tác dụng ngược lại ở phần kia và làm cho thuật toán trở nên mất cân bằng.

Ý tưởng cơ bản của Quick sort dựa trên phương pháp chia để trị như đã trình bày trong chương 2. Giải thuật chia dãy cần sắp thành 2 phần, sau đó thực hiện việc sắp xếp cho mỗi phần độc lập với nhau. Để thực hiện điều này, đầu tiên chọn ngẫu nhiên 1 phần tử nào đó của dãy làm khoá. Trong bước tiếp theo, các phần tử nhỏ hơn khoá phải được xếp vào phía trước khoá và các phần tử lớn hơn được xếp vào phía sau khoá. Để có được sự phân loại này, các phần tử sẽ được so sánh với khoá và hoán đổi vị trí cho nhau hoặc cho khoá nếu nó lớn hơn khoá mà lại nằm trước hoặc nhỏ hơn khoá mà lại nằm sau. Khi lượt hoán đổi đầu tiên thực hiện xong thì dãy được chia thành 2 đoạn: 1 đoạn bao gồm các phần tử nhỏ hơn khoá, đoạn còn lại bao gồm các phần tử lớn hơn khoá.



Hình 7.1 Quick sort

Áp dụng kỹ thuật như trên cho mỗi đoạn đó và tiếp tục làm như vậy cho đến khi mỗi đoạn chỉ còn 2 phần tử. Khi đó toàn bộ dãy đã được sắp.

6.2.2. Mô tả thuật toán

Để chia dãy thành 2 phần thoả mãn yêu cầu như trên, ta lấy một phần tử của dãy làm khoá (chẳng hạn phần tử đầu tiên). Tiến hành duyệt từ bên trái dãy và dừng lại khi gặp 1 phần tử lớn hơn hoặc bằng khoá. Đồng thời tiến hành duyệt từ bên phải dãy cho tới khi gặp 1 phần tử nhỏ hơn hoặc bằng khoá. Rõ ràng 2 phần tử này nằm ở những vị trí không phù hợp và chúng cần phải được đổi chỗ cho nhau. Tiếp tục quá trình cho tới khi 2 biến duyệt gặp nhau, ta sẽ chia được dãy thành 2 nửa: Nửa bên phải khoá bao gồm những phần tử lớn hơn hoặc bằng khoá và nửa bên trái là những phần tử nhỏ hơn hoặc bằng khoá.

Ta hãy xem xét quá trình phân đôi dãy số đã cho ở phần trước.

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

Chọn phần tử đầu tiên của dãy, phần tử 32, làm khoá. Quá trình duyệt từ bên trái với biến duyệt i sẽ dừng lại ở 49, vì đây là phần tử lớn hơn khoá. Quá trình duyệt từ bên phải với biến duyệt j sẽ dừng lại ở 25 vì đây là phần tử nhỏ hơn khoá. Tiến hành đổi chỗ 2 phần tử chonhau.

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



Khoá

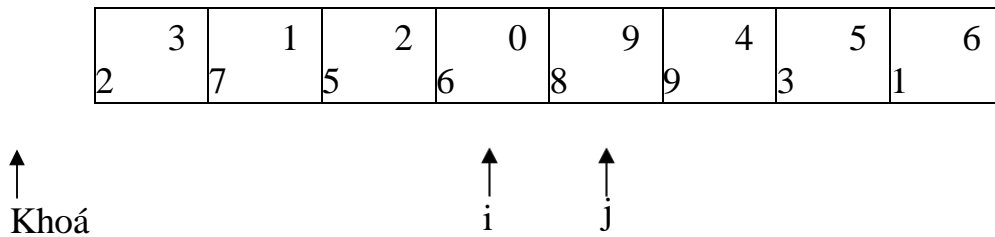


i

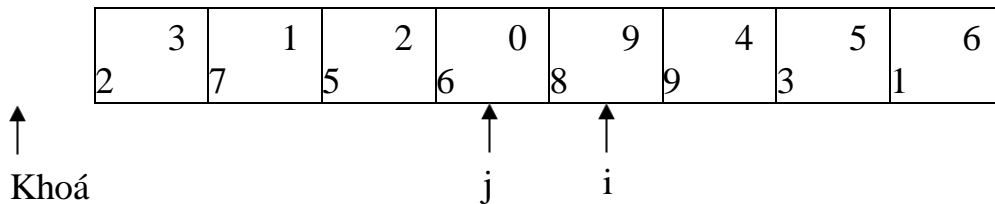


j

Quá trình duyệt tiếp tục. Biến duyệt i dừng lại ở 98, còn biến duyệt j dừng lại ở 06. Lại tiến hành đổi vị trí 2 phần tử 98 và 06.



Tiếp tục quá trình duyệt. Các biến duyệt i và j gặp nhau và quá trình duyệt dừng lại.



Như vậy, dãy đã được chia làm 2 nửa. Nửa đầu từ phần tử đầu tiên đến phần tử thứ j, bao gồm các phần tử nhỏ hơn hoặc bằng khoá. Nửa sau từ phần tử thứ i đến phần tử cuối, bao gồm các phần tử lớn hơn hoặc bằng khoá.

Quá trình duyệt và đổi chỗ được lặp lại với 2 nửa dãy vừa được tạo ra, và cứ tiếp tục như vậy cho tới khi dãy được sắp hoàn toàn.

6.2.3. Kiểm nghiệm thuật toán

Để cài đặt giải thuật, trước hết ta xây dựng một thủ tục để sắp một phân đoạn của dãy. Thủ tục này là 1 thủ tục đệ qui, bao gồm việc chia phân đoạn thành 2 đoạn con thỏa mãn yêu cầu trên, sau đó thực hiện lời gọi đệ qui với 2 đoạn con vừa tạo được. Giả sử phân đoạn được giới hạn bởi 2 tham số là left và right cho biết chỉ số đầu và cuối của phân đoạn, khi đó thủ tục được cài đặt như sau:

```
void quick(int left, int right) { inti,j;
    int x,y;
    i=left; j=right; x= a[left];
    do {
        while(a[i]<x && i<right) i++; while(a[j]>x && j>left)
j--; if(i<=j){
        y=a[i];a[i]=a[j];a[j]=y; i++;j--;
        }
    }while (i<=j);
    if (left<j) quick(left,j); if
(i<right) quick(i,right);
}
```

Tiếp theo, để thực hiện sắp xếp toàn bộ dãy, ta chỉ cần gọi thủ tục trên với tham số left là chỉ số đầu và right là chỉ số cuối của mảng.

```
void quick_sort() {  
    quick(0, n-1);  
}
```

Nhược điểm của Quick sort là hoạt động rất kém hiệu quả trên những dãy đã được sắp sẵn. Khi đó, cần phải mất N lần gọi đệ qui và mỗi lần chỉ loại được 1 phần tử. Thời gian thực hiện thuật toán trong trường hợp xấu nhất này là khoảng $N^2/2$, có nghĩa là $O(N^2)$.

Trong trường hợp tốt nhất, mỗi lần phân chia sẽ được 2 nửa dãy bằng nhau, khi đó thời gian thực hiện thuật toán $T(N)$ sẽ được tính là:

$$T(N) = 2T(N/2) + N$$

Khi đó, $T(N) \approx N \log N$.

Trong trường hợp trung bình, thuật toán cũng có độ phức tạp khoảng $2N \log N = O(N \log N)$. Như vậy, quick sort là thuật toán rất hiệu quả trong đa số trường hợp. Tuy nhiên, đối với các trường hợp việc sắp xếp chỉ phải thực hiện một vài lần và số lượng dữ liệu cực lớn thì nên thực thi một số thuật toán khác có thời gian thực hiện trong mọi trường hợp là $O(N \log N)$, sẽ xem xét ở phần sau, để đảm bảo trường hợp xấu nhất không xảy ra khi dùng quick sort.

6.3. Thuật toán sắp xếp kiểu Heap

6.3.1. Giới thiệu thuật toán

Heap sort là một giải thuật đảm bảo kể cả trong trường hợp xấu nhất thì thời gian thực hiện thuật toán cũng chỉ là $O(N \log N)$.

Ý tưởng cơ bản của giải thuật này là thực hiện sắp xếp thông qua việc tạo các heap, trong đó heap là 1 cây nhị phân hoàn chỉnh có tính chất là khóa ở nút cha bao giờ cũng lớn hơn khóa ở các nút con.

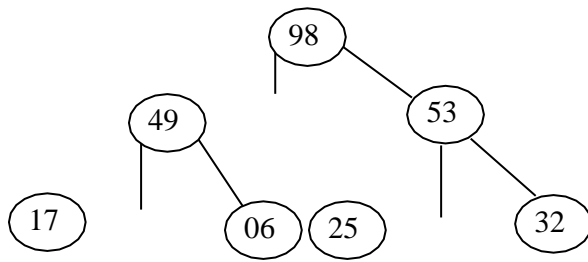
Việc thực hiện giải thuật này được chia làm 2 giai đoạn. Đầu tiên là việc tạo heap từ dãy ban đầu. Theo định nghĩa của heap thì nút cha bao giờ cũng lớn hơn các nút con. Do vậy, nút gốc của heap bao giờ cũng là phần tử lớn nhất.

Giai đoạn thứ 2 là việc sắp dãy dựa trên heap tạo được. Do nút gốc là nút lớn nhất nên nó sẽ được chuyển về vị trí cuối cùng của dãy và phần tử cuối cùng sẽ được thay vào gốc của heap. Khi đó ta có 1 cây mới, không phải heap, với số nút được bớt đi 1. Lại chuyển cây này về heap và lặp lại quá trình cho tới khi heap chỉ còn 1 nút. Đó chính là phần tử bé nhất của dãy và được đặt lên đầu.

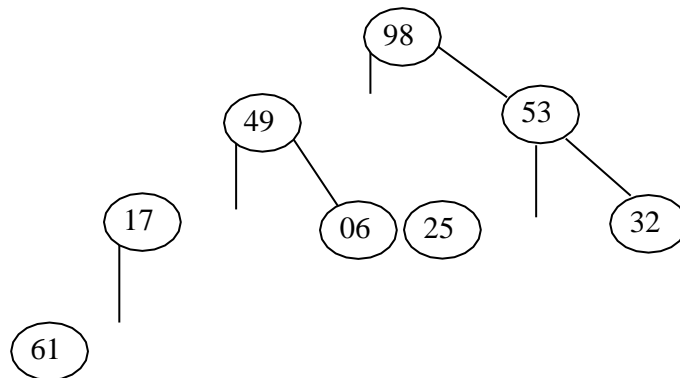
6.3.2. Mô tả thuật toán

Như vậy, việc đầu tiên cần làm là phải tạo được 1 heap từ 1 dãy phần tử cho trước. Để làm việc này, cần thực hiện thao tác chèn 1 phần tử vào 1 heap đã có. Khi đó, kích thước của heap tăng lên 1, và ta đặt phần tử mới vào cuối heap. Việc này có thể làm vi phạm định nghĩa heap vì nút mới có thể lớn hơn nút cha của nó. Vấn đề này được giải quyết bằng cách đổi vị trí nút mới cho nút cha, và nếu vẫn vi phạm định nghĩa heap thì ta lại giải quyết theo cách tương tự cho đến khi có một heap mới hoàn chỉnh.

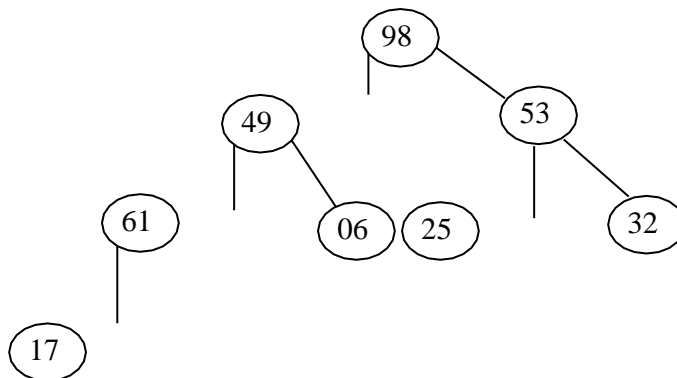
Giả sử ta đã có 1 heap như sau:



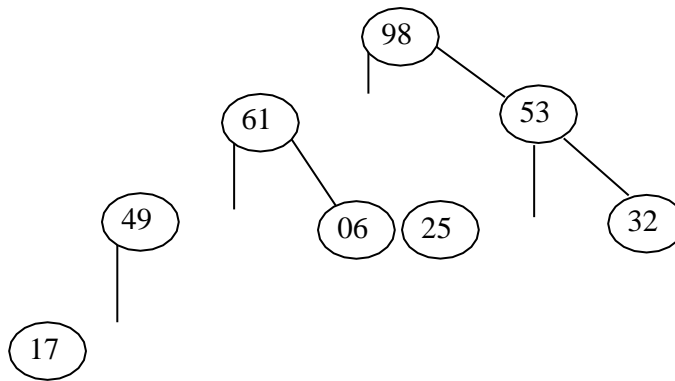
Để chèn phần tử 61 vào heap, đầu tiên, ta đặt nó vào vị trí cuối cùng trong cây.



Rõ ràng cây mới vi phạm định nghĩa heap vì nút con 61 lớn hơn nút cha 17.
Tiến hành đổi vị trí 2 nút này chonhau:



Cây này vẫn tiếp tục vi phạm định nghĩa heap do nút con 61 lớn hơn nút cha 49. Lại đổi vị trí 61 cho 49.



Do nút con 61 nhỏ hơn nút cha 98 nên cây thoả mãn định nghĩa heap. Như vậy, ta đã có một heap với nút mới được thêm vào là 61.

Để chèn một phần tử x vào 1 heap đã có k phần tử, ta gán phần tử thứ $k + 1$, $a[k]$, bằng x , rồi gọi thủ tục $\text{upheap}(k)$.

```

void    upheap(int
m){ int x;
    x=a[m];
    while    ((a[(m-1)/2]<=x)    &&
(m>0)) { a[m]=a[(m-1)/2];
    m=(m-1)/2;
    }
    a[m]=x;
    }
  
```

```

void    insert_heap(int
x){ a[m]=x;
    upheap(m);
m++;
    }
  
```

Như vậy, với heap ban đầu chỉ có 1 phần tử là phần tử đầu tiên của dãy, ta lần lượt lấy các phần tử tiếp theo của dãy chèn vào heap sẽ tạo được 1 heap gồm toàn bộ n phần tử.

6.3.3. Kiểm nghiệm thuật toán

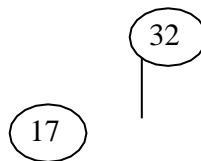
Ta hãy xem xét quá trình tạo heap với dãy số đã cho ở phần trước.

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

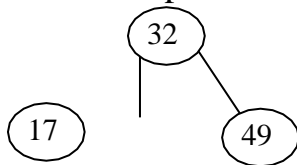
Đầu tiên, tạo 1 heap với chỉ 1 phần tử là 32:



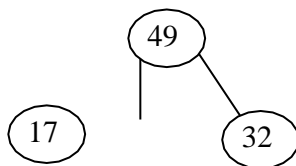
Bước 1: Tiến hành chèn 17 vào heap.



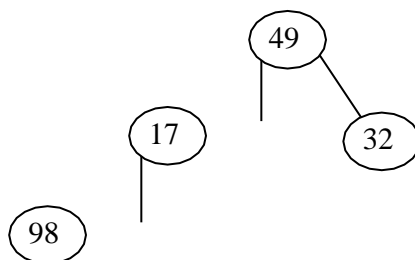
Do không vi phạm định nghĩa heap nên không thay đổi gì. Bước 2: Tiến hành chèn 49 vào heap



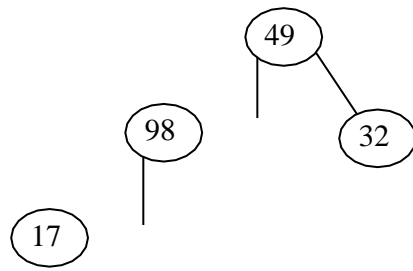
Cây này vi phạm định nghĩa heap do $49 > 32$ nên đổi vị trí 32 và 49 cho nhau.



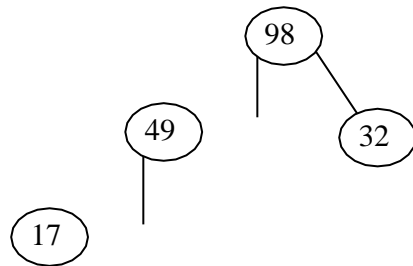
Cây mới thoả mãn định nghĩa heap. Bước 3: Tiến hành chèn 98 vào heap.



Cây này vi phạm định nghĩa heap do $98 > 17$ nên đổi vị trí 98 và 17 cho nhau.

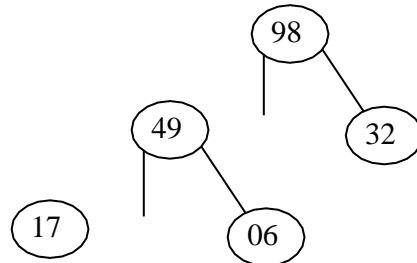


Cây mới lại vi phạm định nghĩa heap, do $98 > 49$, nên đổi vị trí 98 cho 49.

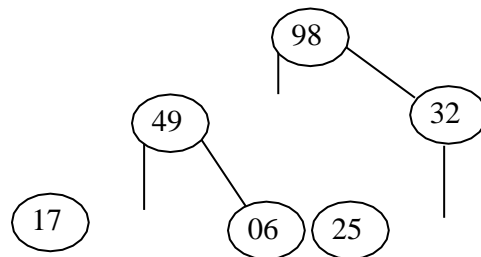


Cây này thoả mãn định nghĩa heap.

Bước 4: Tiến hành chèn 06 vào heap.

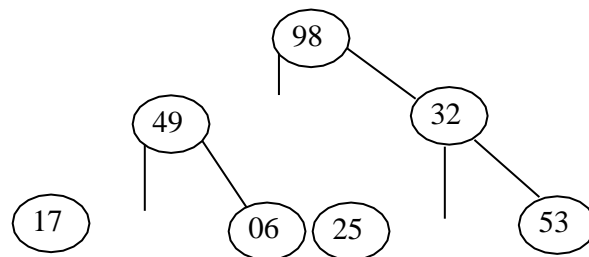


Cây này thoả mãn định nghĩa heap do



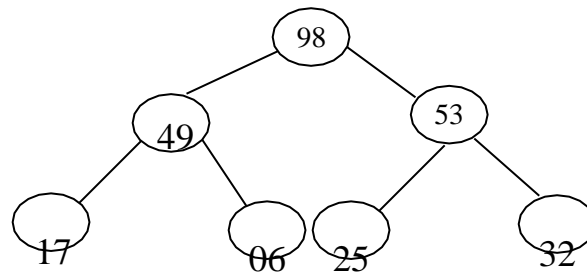
Bước 5: Tiến hành chèn 25 vào heap.

Cây này thoả mãn định nghĩa heap do



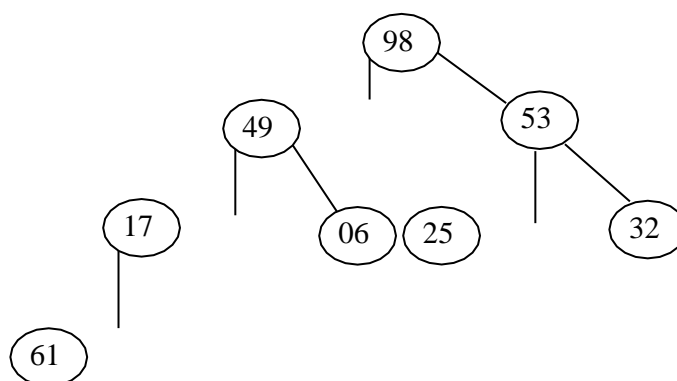
Bước 6: Tiến hành chèn 53 vào heap.

Cây này vi phạm định nghĩa heap do $53 > 32$ nên đổi vị trí 53 và 32 cho nhau.

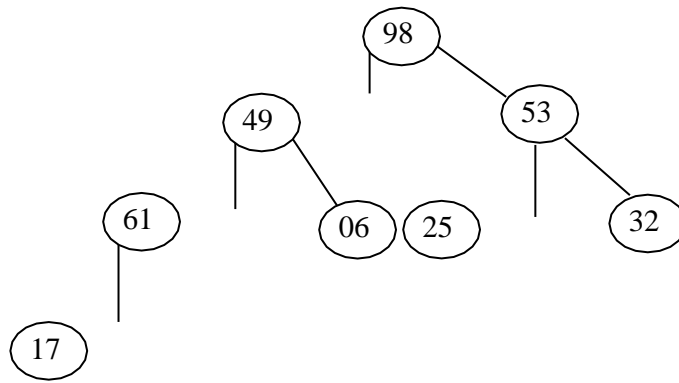


Cây mới thoả mãn định nghĩa heap.

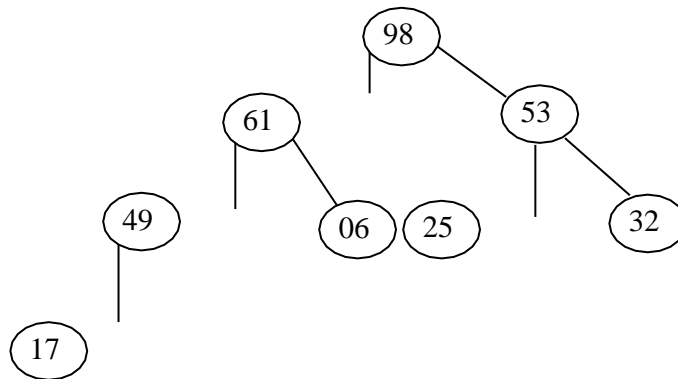
Bước 7: Tiến hành chèn 61 vào heap.



Cây này vi phạm định nghĩa heap do $61 > 17$ nên đổi vị trí 61 và 17 cho nhau.



Cây mới tiếp tục vi phạm định nghĩa heap do $61 > 49$ nên đổi vị trí 61 và 49 cho nhau.



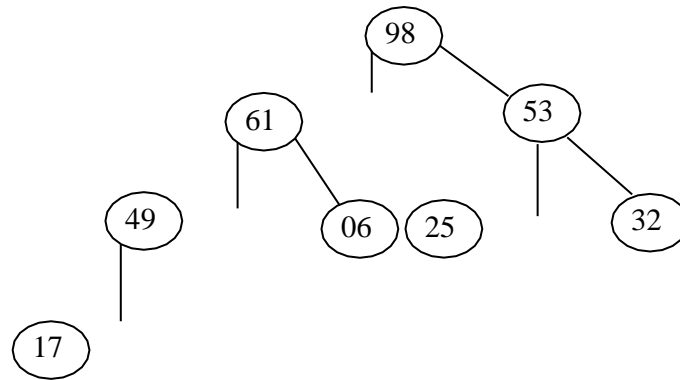
Cây này thoả mãn định nghĩa heap, và chính là heap cần tạo.

Sau khi tạo được heap, để tiến hành sắp xếp, ta cần lấy phần tử đầu và là phần tử lớn nhất của cây và thay thế nó bằng phần tử cuối của dãy. Điều này có thể làm vi phạm định nghĩa heap vì phần tử mới đưa lên gốc có thể nhỏ hơn 1 trong 2 nút con.

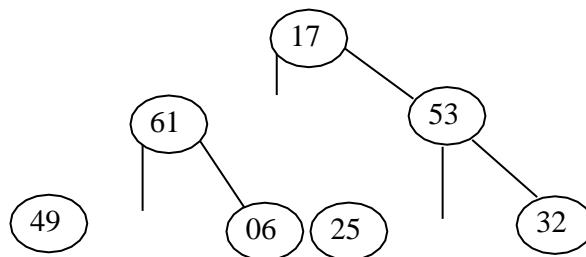
Do đó, thao tác thứ 2 cần thực hiện trên heap là tiến hành chỉnh lại heap khi có 1 nút nào đó nhỏ hơn 1 trong 2 nút con của nó. Khi đó, ta sẽ tiến hành thay thế nút này cho nút con lớn hơn.

Nếu vẫn vi phạm định nghĩa heap thì ta lại lặp lại quá trình cho tới khi nó lớn hơn cả 2 nút con hoặc trở thành nút lá.

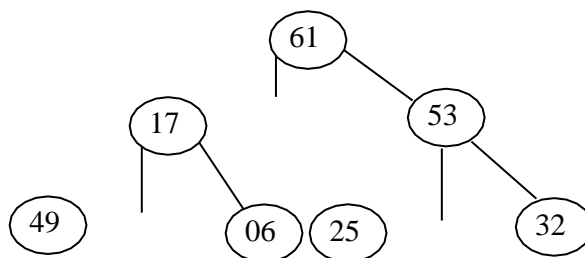
Ta xem xét ví dụ với heap vừa tạo được ở phần trước:



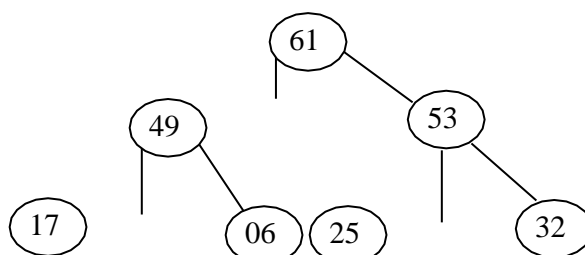
Lấy nút gốc 98 ra khỏi heap và thay thế bởi nút cuối là 17.



Cây này không thỏa mãn định nghĩa heap vì 17 nhỏ hơn cả 2 nút con là 61 và 53. Tiến hành đổi chỗ 17 cho nút con lớn hơn là 61.



Vẫn tiếp tục vi phạm định nghĩa heap do 17 nhỏ hơn nút con là 49. Tiến hành đổi chỗ 17 cho 49, ta có heap mới hoàn chỉnh.



Ta có thủ tục downheap để chỉnh lại heap khi nút k không thoả mãn định nghĩa heap nhursau:

```
void downheap(int
k){    int    j,    x;
x=a[k];
    while    (k<=(m-2)/2){
j=2*k+1;
    if (j<m-1) if (a[j]<a[j+1]) j++; if
(x>=a[j]) break;
    a[k]=a[j]; k=j;
    }
    a[k]=x;
    }
```

Trong thủ tục này, nút k sẽ được kiểm tra. Nếu nó vi phạm định nghĩa heap thì sẽ được thay bởi 1 trong 2 nút con (nút lớn hơn) tại vị trí $2*k$ và $2*k+1$. Sau khi hoán đổi vị trí, nếu vẫn tiếp tục vi phạm định nghĩa heap thì việc hoán đổi lại được thực hiện. Quá trình tiếp tục cho đến khi không còn vi phạm hoặc tới nút lá.

Cuối cùng, thủ tục heap sort thực hiện việc sắp xếp trên heap đã tạo như sau:

```
int
remove_node(){    int
temp;    temp=a[0];
a[0]=a[m];
    m--;
    downheap(0);
return temp;
    }

void
heap_sort(){    int
i;
    m=0;
    for    (i=0;    i<=n-1;    i++)
insert_heap(a[i]); m=n-1;
```



```
for (i=n-1; i>=0; i--) a[i]=remove_node();
}
```

Trong đoạn mã trên, hàm remove() sẽ trả về giá trị là nút gốc của heap. Nút này sẽ được chuyển xuống cuối heap, và nút cuối được đổi lên gốc. Kích thước của heap giảm đi 1, tiến hành gọi thủ tục downheap để chỉnh lại heap mới.

Thủ tục heap sort đầu tiên tạo 1 heap bằng cách lần lượt chèn các phần tử của dãy vào heap. Tiếp theo, các nút gốc lần lượt được lấy ra, heap mới được tạo và chỉnh lại. Quá trình kết thúc khi heap không còn phần tử nào.

Một trong những tính chất của heap sort khiến nó rất được quan tâm trong thực tế đó là thời gian thực hiện thuật toán luôn là $O(N\log N)$ trong mọi trường hợp, bất kể dữ liệu đầu vào có tính chất như thế nào. Đây cũng là ưu điểm của heap sort so với quick sort, thuật toán có thời gian thực hiện nhanh nhưng trong trường hợp xấu nhất thì thời gian lên tới $O(N^2)$.

6.4. Thuật toán sắp xếp kiểu hòa nhập

6.4.1. Giới thiệu thuật toán

Tương tự như heap sort, merge sort cũng là một giải thuật sắp xếp có thời gian thực hiện là $O(N\log N)$ trong mọi trường hợp.

Ý tưởng của giải thuật này bắt nguồn từ việc trộn 2 danh sách đã được sắp xếp thành 1 danh sách mới cũng được sắp. Rõ ràng việc trộn 2 dãy đã sắp thành 1 dãy mới được sắp có thể tận dụng đặc điểm đã sắp của 2 dãy con.

Để thực hiện giải thuật sắp xếp trộn đối với 1 dãy bất kỳ, đầu tiên, coi mỗi phần tử của dãy là 1 danh sách con gồm 1 phần tử đã được sắp. Tiếp theo, tiến hành trộn từng cặp 2 dãy con 1 phần tử kề nhau để tạo thành các dãy con 2 phần tử được sắp. Các dãy con 2 phần tử được sắp này lại được trộn với nhau tạo thành dãy con 4 phần tử được sắp. Quá trình tiếp tục đến khi chỉ còn 1 dãy con duy nhất được sắp, đó chính dãy ban đầu.

6.4.2. Mô tả thuật toán

Thao tác đầu tiên cần thực hiện khi sắp xếp trộn là việc tiến hành trộn 2 dãy đã sắp thành 1 dãy mới cũng được sắp. Để làm việc này, ta sử dụng 2 biến duyệt từ đầu mỗi dãy. Tại mỗi bước, tiến hành so sánh giá trị của 2 phần tử tại vị trí của 2 biến duyệt. Nếu phần tử nào có giá trị nhỏ hơn, ta đưa phần tử đó xuống dãy mới và tăng biến duyệt tương ứng lên 1. Quá trình lặp lại cho tới khi tất cả các phần tử của cả 2 dãy đã được duyệt và xét.

6.4.3. Kiểm nghiệm thuật toán

Giả sử ta có 2 dãy đã sắp như sau:

i
↓

17	32	49	98
----	----	----	----

j
↓

06	25	53	61
----	----	----	----

Để trộn 2 dãy, ta sử dụng một dãy thứ 3 để chứa các phần tử của dãy tổng. Một biến duyệt k dùng để lưu giữ vị trí cần chèn tiếp theo trong dãy mới.

Bước 1: Phần tử tại vị trí biến duyệt j là 06 nhỏ hơn phần tử tại vị trí biến duyệt i là 17 nên ta đưa 06 xuống dãy mới và tăng j lên 1. Đồng thời, biến duyệt k cũng tăng lên 1.

i
↓

17	32	49	98
----	----	----	----

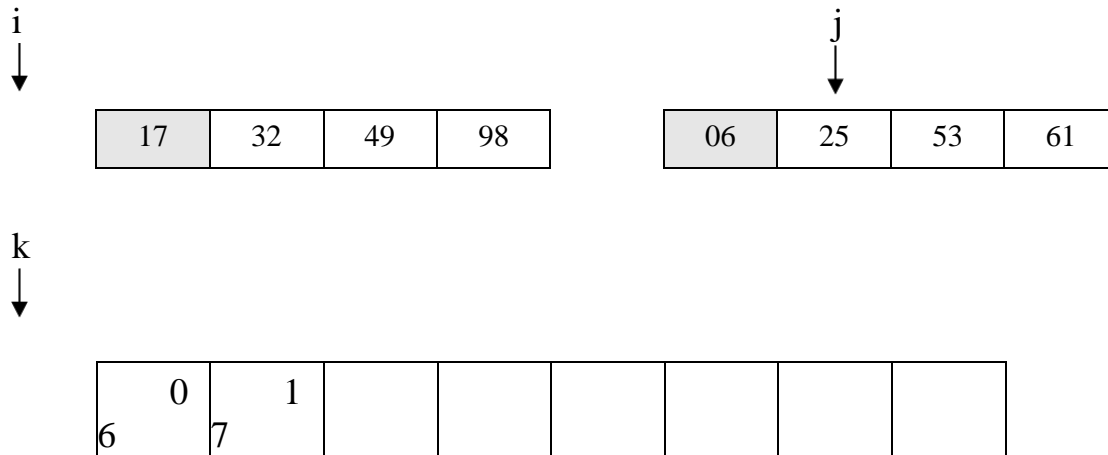
j
↓

06	25	53	61
----	----	----	----

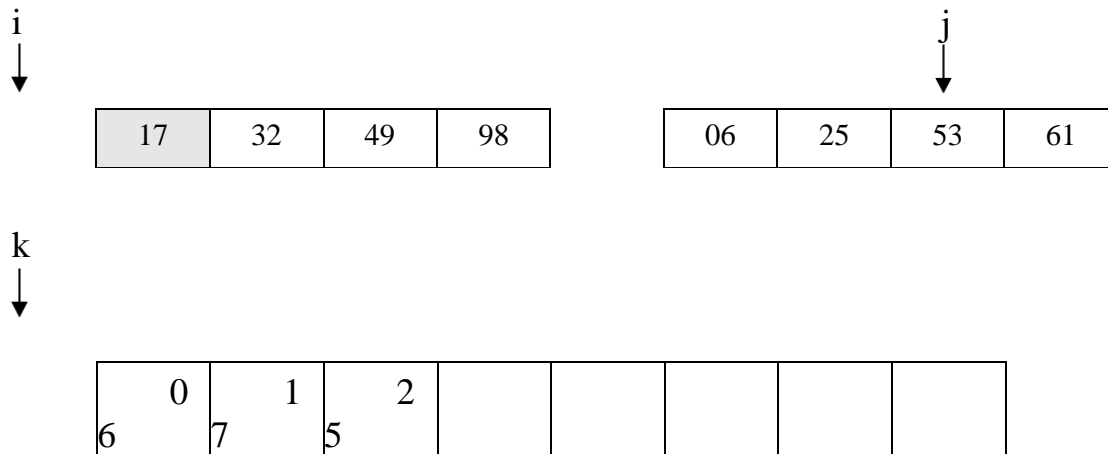
k
↓

0							
6							

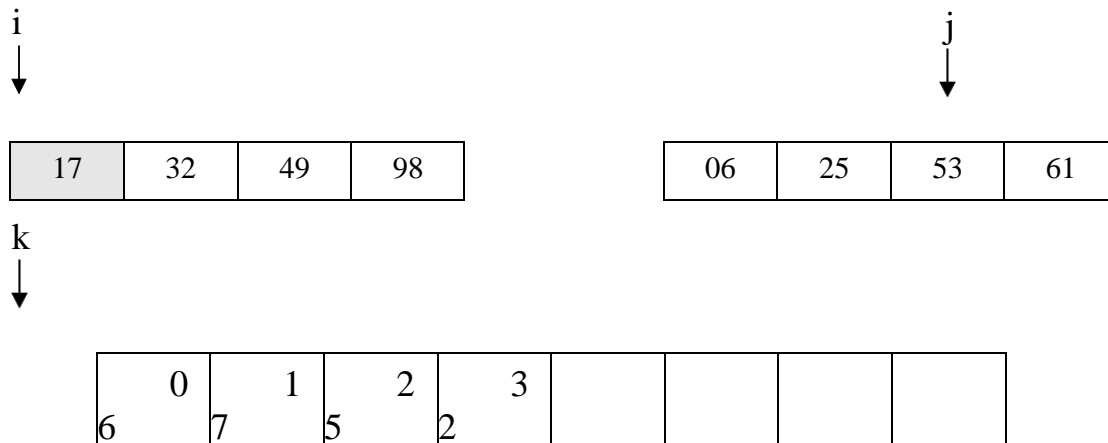
Bước 2: Phần tử tại vị trí i là 17 nhỏ hơn phần tử tại vị trí j là 25 nên ta đưa 17 xuống dãy mới và tăng i lên 1, biến duyệt k cũng tăng lên 1.



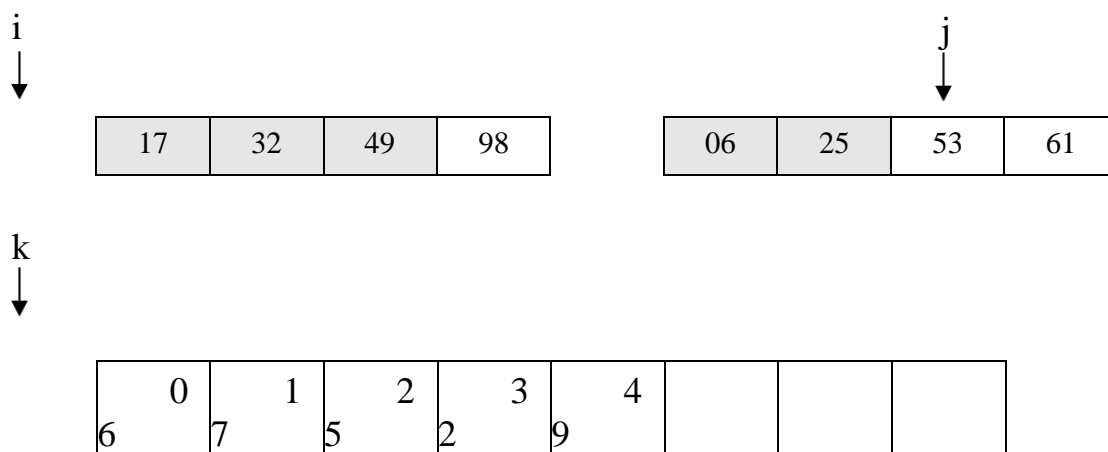
Bước 3: Phần tử tại vị trí j là 25 nhỏ hơn phần tử tại vị trí i là 32 nên ta đưa 25 xuống dãy mới và tăng j lên 1, biến duyệt k cũng tăng lên 1.



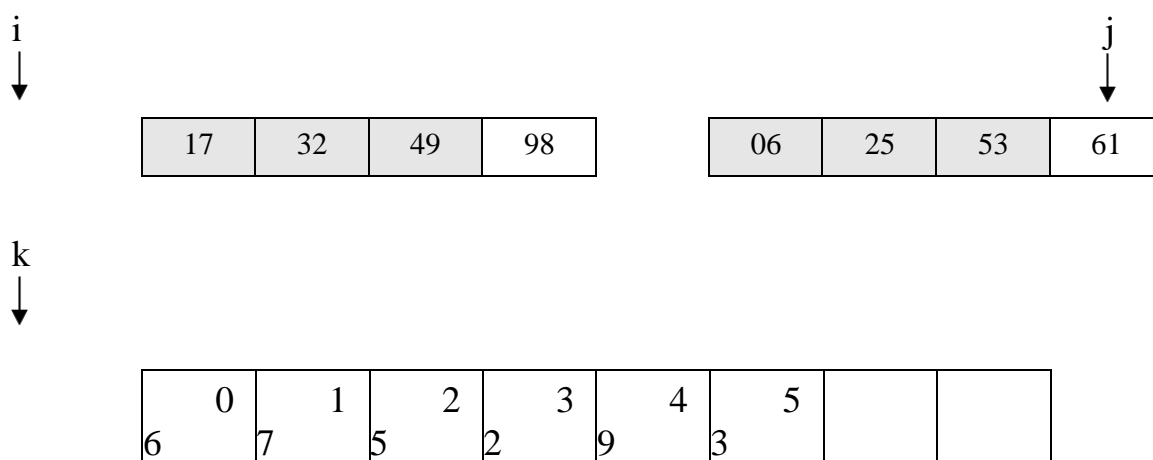
Bước 4: Phần tử tại vị trí i là 32 nhỏ hơn phần tử tại vị trí j là 53 nên ta đưa 32 xuống dãy mới và tăng i lên 1, biến duyệt k cũng tăng lên 1.



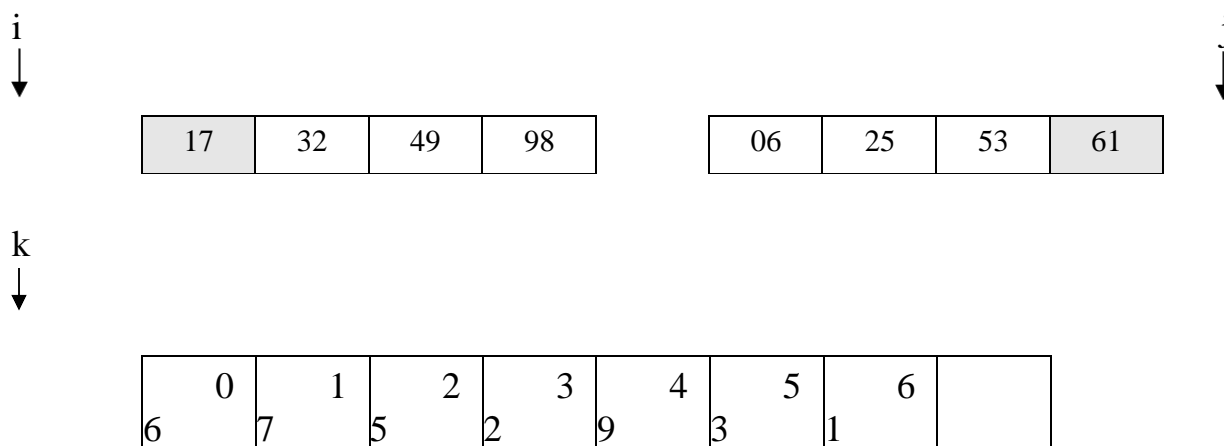
Bước 5: Phần tử tại vị trí i là 49 nhỏ hơn phần tử tại vị trí j là 53 nên ta đưa 49 xuống dãy mới và tăng i lên 1, biến duyệt k cũng tăng lên 1.



Bước 6: Phần tử tại vị trí j là 53 nhỏ hơn phần tử tại vị trí i là 98 nên ta đưa 53 xuống dãy mới và tăng j lên 1, biến duyệt k cũng tăng lên 1.



Bước 7: Phần tử tại vị trí j là 61 nhỏ hơn phần tử tại vị trí i là 98 nên ta đưa 61 xuống dãy mới và tăng j lên 1, biến duyệt k cũng tăng lên 1.



Bước 8: Biến duyệt j đã duyệt hết dãy thứ 2. Khi đó, ta tiến hành đưa toàn bộ phần còn lại của dãy 1 xuống dãy mới.



17	32	49	98
----	----	----	----

06	25	53	61
----	----	----	----

k
↓

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

Như vậy, ta có dãy mới là dãy đã sắp, bao gồm các phần tử của 2

dãy ban đầu. Thủ tục tiến hành trộn 2 dãy đã sắp như sau:

```
void merge(int *c, int cl, int *a, int al, int ar,
int *b, int bl, int br){
    int i=al, j=bl, k;
    for (k=cl; k< cl+ar-al+br-bl+1;
k++){ if (i>ar){
        c[k]=b[j++];
```

```

        continue;
    }
    if (j>br) {
        c[k]=a[i++];
continue;
    }
    if (a[i]<b[j]) c[k]=a[i++];
else c[k]=b[j++];
    }
}

```

Thủ tục này tiến hành trộn 2 dãy a và b, với các chỉ số đầu và cuối tương ứng là al, ar, bl, br. Kết quả trộn được lưu trong mảng c, có chỉ số đầu là cl.

Tuy nhiên, ta thấy rằng để thực hiện sắp xếp trộn thì 2 dãy được trộn không phải là 2 dãy riêng biệt, mà nằm trên cùng 1 mảng. Hay nói cách khác là ta trộn 2 phần của 1 dãy chứ không phải 2 dãy riêng biệt a và b như trong thủ tục trên. Ngoài ra, kết quả trộn lại được lưu ngay tại dãy ban đầu chứ không lưu ra một dãy c khác.

Do vậy, ta phải cải tiến thủ tục trên để thực hiện trộn 2 phần của 1 dãy và kết quả trộn được lưu lại ngay trên dãy đó.

```

void merge(int *a, int al, int am,
int ar){ int i=al, j=am+1, k;
    for (k=al; k<=ar;
k++){ if (i>am) {
        c[k]=a[j++];
continue;
    }
    if (j>ar) {
        c[k]=a[i++];
continue;
    }
    if (a[i]<a[j]) c[k]=a[i++];
else c[k]=a[j++];
    }
    for (k=al; k<=ar; k++) a[k]=c[k];
}

```

```
}
```

Thủ tục này tiến hành trộn 2 phần của dãy a. Phần đầu có chỉ số từ a1 đến am, phần sau có chỉ số từ am+1 đến ar. Ta dùng 1 mảng tạm c để lưu kết quả trộn, sau đó sao chép lại kết quả vào mảng ban đầu a.

6.5. Một số thuật toán tìm kiếm

6.5.1. Tìm kiếm tuyến tính

Tìm kiếm tuần tự là một phương pháp tìm kiếm rất đơn giản, lần lượt duyệt qua toàn bộ các bản ghi một cách tuần tự. Tại mỗi bước, khoá của bản ghi sẽ được so sánh với giá trị cần tìm. Quá trình tìm kiếm kết thúc khi đã tìm thấy bản ghi có khoá thoả mãn hoặc đã duyệt hết danh sách.

Thủ tục tìm kiếm tuần tự trên một mảng các số nguyên như sau:

```
int sequential_search(int *a, int
x, int n){ int i;
    for (i=0; i<n ; i
++) { if (a[i] == X)
        return(i);
    }
    return(-1);
}
```

Thủ tục này tiến hành duyệt từ đầu mảng. Nếu tại vị trí nào đó, giá trị phần tử bằng với giá trị cần tìm thì hàm trả về chỉ số tương ứng của phần tử trong mảng. Nếu không tìm thấy giá trị trong toàn bộ mảng thì hàm trả về giá trị -1.

Thuật toán tìm kiếm tuần tự có thời gian thực hiện là $O(n)$. Trong trường hợp xấu nhất, thuật toán mất n lần thực hiện so sánh và mất khoảng $n/2$ lần so sánh trong trường hợp trung bình.

6.5.2. Tìm kiếm nhị phân

Trong trường hợp số bản ghi cần tìm rất lớn, việc tìm kiếm tuần tự có thể là 1 giải pháp không hiệu quả về mặt thời gian. Một giải pháp tìm kiếm khác hiệu quả hơn có thể được sử dụng

dựa trên mô hình “chia để trị” như sau: Chia tập cần tìm làm 2 nửa, xác định nửa chứa bản ghi cần tìm và tập trung tìm kiếm trên nửa đó.

Để làm được điều này, tập các phần tử cần phải được sắp, và sử dụng chỉ số của

mảng để xác định nửa cần tìm. Đầu tiên, so sánh giá trị cần tìm với giá trị của phần tử ở giữa. Nếu nó nhỏ hơn, tiến hành tìm ở nửa đầu dãy, ngược lại, tiến hành tìm ở nửa sau của dãy. Quá trình được lặp lại tương tự cho nửa dãy vừa được xác định này.

Hàm tìm kiếm nhị phân được cài đặt như sau (giả sử dãy a đã được sắp):

```
int binary_search(int *a, int x){ int k, left =0,
right=n-1;
do{
k=(left+right)/2;
if (x<a[k]) right=k-1; else l=k+1;
}while ((x!=a[k]) && (left<=right)) if (x=a[k])
return k;
else return -1;
}
```

Trong thủ tục này, x là giá trị cần tìm trong dãy a . Hai biến `left` và `right` dùng để giới hạn phân đoạn của mảng mà quá trình tìm kiếm sẽ được thực hiện trong mỗi bước. Đầu tiên 2 biến này được gán giá trị 0 và $n-1$, tức là toàn bộ mảng sẽ được tìm kiếm.

Tại mỗi bước, biên k sẽ được gán cho chỉ số giữa của đoạn đang được tiến hành tìm kiếm. Nếu giá trị x nhỏ hơn giá trị phần tử tại k , biến $right$ sẽ được gán bằng $k-1$, cho biết quá trình tìm tại bước sau sẽ được thực hiện trong nửa đầu của đoạn. Ngược lại, giá trị $left$ được gán bằng $k+1$, cho biết quá trình tìm tại bước sau sẽ được thực hiện trong nửa sau của đoạn.

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

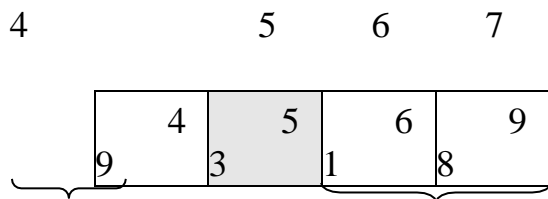
Xét 1 ví dụ với dãy đã sắp ở trên, để tìm kiếm giá trị 61 trong dãy, ta tiến hành các bước nhusau:

Bước 1: Phân chia dãy làm 2 nửa, với chỉ số phân cách là 3.

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

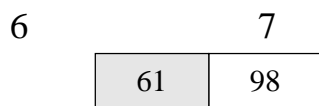
Giá trị phần tử tại chỉ số này là 32, nhỏ hơn giá trị cần tìm là 61. Do vậy, tiến hành tìm kiếm phần tử tại nửa sau của dãy.

Bước 2: Tiếp tục phân chia đoạn cần tìm làm 2 nửa, với chỉ số phân cách là 5.



Giá trị phần tử tại chỉ số này là 53, nhỏ hơn giá trị cần tìm là 61. Do vậy, tiến hành tìm kiếm phần tử tại nửa sau của đoạn.

Bước 3: Tiếp tục phân chia đoạn, với chỉ số phân cách là 6.



Giá trị phần tử tại chỉ số này là 61, bằng giá trị cần tìm. Do vậy, quá trình tìm kiếm kết thúc, chỉ số cần tìm là 6.

Thuật toán tìm kiếm nhị phân có thời gian thực hiện là $\lg N$. Tuy nhiên, thuật toán đòi hỏi dãy đã được sắp trước khi tiến hành tìm kiếm. Do vậy, nên áp dụng tìm kiếm nhị phân khi việc tìm kiếm phải thực hiện nhiều lần trên 1 tập phần tử cho trước. Khi đó, ta chỉ cần tiến hành sắp tập phần tử 1 lần và thực hiện tìm kiếm nhiều lần trên tập phần tử đã sắp này.

6.5.3. Tìm kiếm theo cơ số

Tìm kiếm bằng cây nhị phân là một phương pháp tìm kiếm rất hiệu quả và được xem như là một trong những thuật toán cơ sở của khoa học máy tính. Đây cũng là một phương pháp đơn giản và được lựa chọn để áp dụng trong rất nhiều tình huống thực tế.

Ý tưởng cơ bản của phương pháp này là xây dựng một cây nhị phân tìm kiếm. Đó là một cây nhị phân có tính chất sau: Với mỗi nút của cây, khoá của các nút của cây con bên trái bao giờ cũng nhỏ hơn và khoá của các nút của cây con bên phải bao giờ cũng lớn hơn hoặc bằng khoá của nút đó.

Như vậy, trong một cây nhị phân tìm kiếm thì tất cả các cây con của nó đều thoả mãn tính chất như vậy.

6.6. CASE STUDY

6.6.1. Cài đặt chương trình máy tính các thuật toán sắp xếp.

Thao tác đầu tiên cần thực hiện khi sắp xếp trộn là việc tiến hành trộn 2 dãy đã sắp thành 1 dãy mới cũng được sắp. Để làm việc này, ta sử dụng 2 biến duyệt từ đầu mỗi dãy. Tại mỗi bước, tiến hành so sánh giá trị của 2 phần tử tại vị trí của 2 biến duyệt. Nếu phần tử nào có giá trị nhỏ hơn, ta đưa phần tử đó xuống dãy mới và tăng biến duyệt tương ứng lên 1. Quá trình lặp lại cho tới khi tất cả các phần tử của cả 2 dãy đã được duyệt và xét.

Giả sử ta có 2 dãy đã sắp như sau:

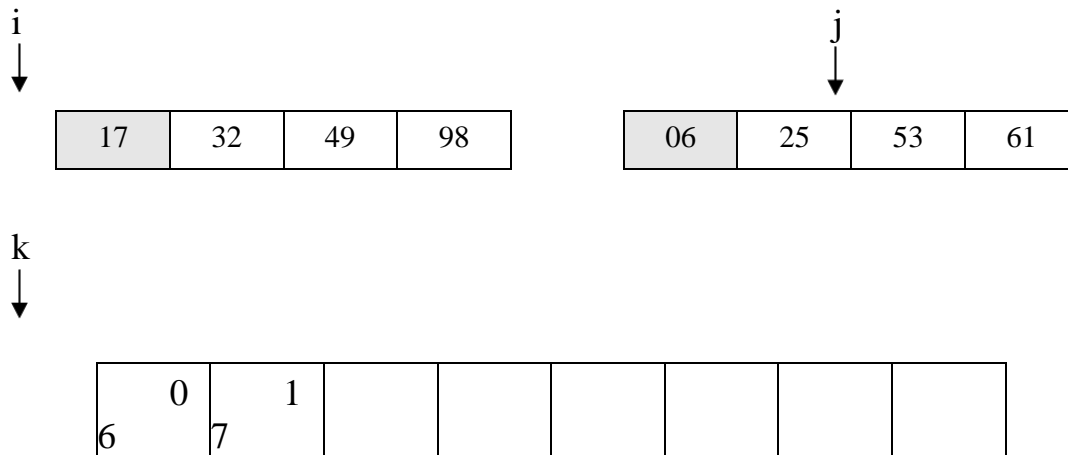
i ↓						j ↓					
	17	32	49	98			06	25	53	61	

Để trộn 2 dãy, ta sử dụng một dãy thứ 3 để chứa các phần tử của dãy tổng. Một biến duyệt k dùng để lưu giữ vị trí cần chèn tiếp theo trong dãy mới.

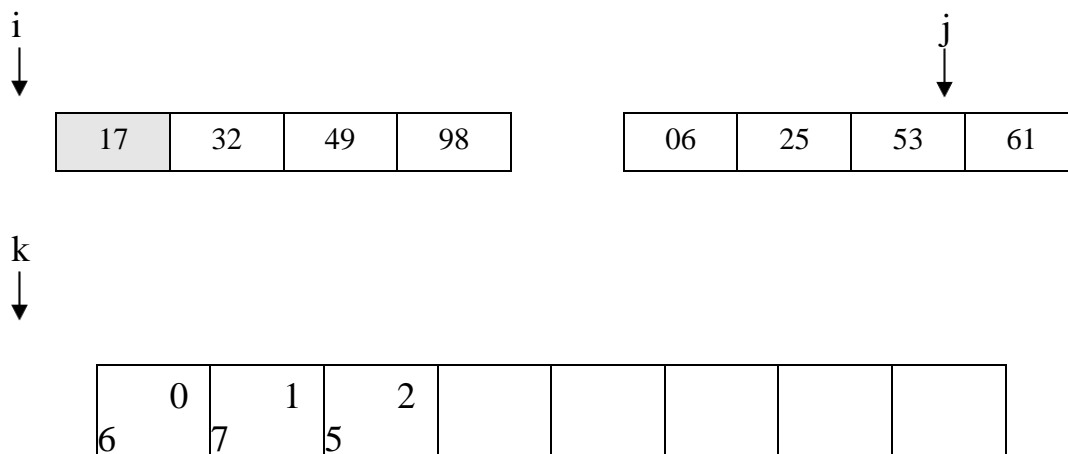
Bước 1: Phần tử tại vị trí biến duyệt j là 06 nhỏ hơn phần tử tại vị trí biến duyệt i là 17 nên ta đưa 06 xuống dãy mới và tăng j lên 1. Đồng thời, biến duyệt k cũng tăng lên 1.

i ↓							j ↓				
	17	32	49	98				06	25	53	61
k ↓											
	0										
	6										

Bước 2: Phần tử tại vị trí i là 17 nhỏ hơn phần tử tại vị trí j là 25 nên ta đưa 17 xuống dãy mới và tăng i lên 1, biến duyệt k cũng tăng lên 1.



Bước 3: Phần tử tại vị trí j là 25 nhỏ hơn phần tử tại vị trí i là 32 nên ta đưa 25 xuống dãy mới và tăng j lên 1, biến duyệt k cũng tăng lên 1.



Bước 4: Phần tử tại vị trí i là 32 nhỏ hơn phần tử tại vị trí j là 53 nên ta đưa 32 xuống dãy mới và tăng i lên 1, biến duyệt k cũng tăng lên 1.

6.6.2. Cài đặt chương trình máy tính các thuật toán tìm kiếm.

Tìm kiếm là một thao tác rất quan trọng đối với nhiều ứng dụng tin học. Tìm kiếm có thể định nghĩa là việc thu thập một số thông tin nào đó từ một khối thông tin lớn đã được lưu trữ trước đó. Thông tin khi lưu trữ thường được chia thành các bản ghi, mỗi bản ghi có một giá trị khoá để phục vụ cho mục đích tìm kiếm. Mục tiêu của việc tìm kiếm là tìm tất cả các bản ghi có giá trị khoá trùng với một giá trị cho trước. Khi tìm được bản ghi này, các thông tin đi kèm trong bản ghi sẽ được thu thập và xử lý.

Một ví dụ về ứng dụng thực tiễn của tìm kiếm là từ điển máy tính.

Trong từ điển có rất nhiều mục từ, khoá của mỗi mục từ chính là cách viết của từ. Thông tin đi kèm là định nghĩa của từ, cách phát âm, các thông tin khác như loại từ, từ đồng nghĩa, khác nghĩa v.v. Ngoài ra còn rất nhiều ví dụ khác về ứng dụng của tìm kiếm, chẳng hạn một ngân hàng lưu trữ các bản ghi thông tin về khách hàng và muốn tìm trong danh sách này một bản ghi của một khách hàng nào đó để kiểm tra số dư và thực hiện các giao dịch, hoặc một chương trình tìm kiếm duyệt qua các tệp văn bản trên máy tính để tìm các văn bản có chứa các từ khoá nào đó.

Trong phần tiếp theo, chúng ta sẽ xem xét 2 phương pháp tìm kiếm phổ biến nhất, đó là tìm kiếm tuần tự và tìm kiếm nhị phân.

6.6.3. Tìm hiểu các thuật toán sắp xếp và tìm kiếm trong thư viện của ngôn ngữ lập trình C⁺⁺.

TÀI LIỆU THAM KHẢO

1. *Data Structures and Algorithms"* của Alfred V. Aho, John E. Hopcroft và Jeffrey D. Ullman do Addison-Wesley tái bản năm 1987
2. *Trần Hạnh Nhi- Dương Anh Đức (2009)- Giáo trình cấu trúc dữ liệu và giải thuật- ĐH Quốc Gia Tp.HCM*
3. *Đỗ Xuân Lôi (2009)- Cấu trúc dữ liệu và giải thuật- ĐH Quốc Gia Hà Nội*