

LỜI NÓI ĐẦU

Cấu trúc dữ liệu và giải thuật là một trong những môn học cơ bản của sinh viên ngành Công nghệ thông tin. Các cấu trúc dữ liệu và các giải thuật được xem như là 2 yếu tố quan trọng nhất trong lập trình, đúng như câu nói nổi tiếng của Niklaus Wirth: Chương trình = Cấu trúc dữ liệu + Giải thuật (Programs = Data Structures + Algorithms). Nắm vững các cấu trúc dữ liệu và các giải thuật là cơ sở để sinh viên tiếp cận với việc thiết kế và xây dựng phần mềm cũng như sử dụng các công cụ lập trình hiện đại.

Cấu trúc dữ liệu có thể được xem như là 1 phương pháp lưu trữ dữ liệu trong máy tính nhằm sử dụng một cách có hiệu quả các dữ liệu này. Và để sử dụng các dữ liệu một cách hiệu quả thì cần phải có các thuật toán áp dụng trên các dữ liệu đó. Do vậy, cấu trúc dữ liệu và giải thuật là 2 yếu tố không thể tách rời và có những liên quan chặt chẽ với nhau. Việc lựa chọn một cấu trúc dữ liệu có thể sẽ ảnh hưởng lớn tới việc lựa chọn áp dụng giải thuật nào.

Tài liệu “Cấu trúc dữ liệu và giải thuật” bao gồm 7 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, 6 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 7 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.

Cuối mỗi phần đều có các câu hỏi và bài tập để sinh viên ôn luyện và tự kiểm tra kiến thức của mình. Cuối tài liệu có các phụ lục hướng dẫn trả lời câu hỏi, mã nguồn tham khảo và tài liệu tham khảo.

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.

Cuối cùng, mặc dù đã hết sức cố gắng nhưng chắc chắn không tránh khỏi các thiếu sót. Tác giả rất mong nhận được sự góp ý của bạn đọc và đồng nghiệp để tài liệu được hoàn thiện hơn.

Hà nội, tháng 10/2007

MỤC LỤC

CHƯƠNG I: PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT	1
1.1 GIẢI THUẬT VÀ NGÔN NGỮ DIỄN ĐẠT GIẢI THUẬT	1
1.1.1 Giải thuật.....	1
1.1.2 Ngôn ngữ diễn đạt giải thuật và kỹ thuật tinh chỉnh từng bước.....	7
1.2 PHÂN TÍCH THUẬT TOÁN.....	9
1.2.1 Ước lượng thời gian thực hiện chương trình	10
1.2.2 Tính toán thời gian thực hiện chương trình	11
1.3 TÓM TẮT CHƯƠNG 1	13
1.4 CÂU HỎI VÀ BÀI TẬP.....	13
CHƯƠNG 2: ĐỆ QUI.....	14
2.1 KHÁI NIỆM	14
2.1.1 Điều kiện để có thể viết một chương trình đệ qui.....	15
2.1.2 Khi nào không nên sử dụng đệ qui	15
2.2 THIẾT KẾ GIẢI THUẬT ĐỆ QUI	17
2.2.1 Chương trình tính hàm $n!$	17
2.2.2 Thuật toán Euclid tính ước số chung lớn nhất của 2 số nguyên dương	18
2.2.3 Các giải thuật đệ qui dạng chia để trị (divide and conquer)	18
2.2.4 Thuật toán quay lui (backtracking algorithms).....	23
2.3 TÓM TẮT CHƯƠNG 2	32
2.4 CÂU HỎI VÀ BÀI TẬP.....	32
CHƯƠNG 3: MẢNG VÀ DANH SÁCH LIÊN KẾT	34
3.1 CẤU TRÚC DỮ LIỆU KIỂU MẢNG (ARRAY).....	34
3.2 DANH SÁCH LIÊN KẾT	35
3.2.1 Khái niệm.....	35
3.2.2 Các thao tác cơ bản trên danh sách liên kết	36
3.2.3 Một số dạng khác của danh sách liên kết.....	45
3.3 TÓM TẮT CHƯƠNG 3	48
3.4 CÂU HỎI VÀ BÀI TẬP.....	48

CHƯƠNG 4: NGĂN XẾP VÀ HÀNG ĐỢI.....	49
4.1 NGĂN XẾP (STACK).....	49
4.1.1 Khái niệm.....	49
4.1.2 Cài đặt ngăn xếp bằng mảng.....	50
4.1.3 Cài đặt ngăn xếp bằng danh sách liên kết.....	52
4.1.4 Một số ứng dụng của ngăn xếp.....	55
4.2 HÀNG ĐỢI (QUEUE).....	63
4.2.1 Khái niệm.....	63
4.2.2 Cài đặt hàng đợi bằng mảng.....	64
4.2.3 Cài đặt hàng đợi bằng danh sách liên kết.....	67
4.3 TÓM TẮT CHƯƠNG 4.....	68
4.4 CÂU HỎI VÀ BÀI TẬP.....	69
CHƯƠNG 5: CẤU TRÚC DỮ LIỆU KIỂU CÂY.....	70
5.1 KHÁI NIỆM.....	70
5.2 CÀI ĐẶT CÂY.....	71
5.2.1 Cài đặt cây bằng mảng các nút cha.....	71
5.2.2 Cài đặt cây thông qua danh sách các nút con.....	72
5.3 DUYỆT CÂY.....	73
5.3.1 Duyệt cây thứ tự trước.....	73
5.3.2 Duyệt cây thứ tự giữa.....	74
5.3.3 Duyệt cây thứ tự sau.....	74
5.4 CÂY NHỊ PHÂN.....	75
5.4.1 Cài đặt cây nhị phân bằng mảng.....	76
5.4.2 Cài đặt cây nhị phân bằng danh sách liên kết.....	77
5.4.3 Duyệt cây nhị phân.....	78
5.5 TÓM TẮT CHƯƠNG 5.....	78
5.6 CÂU HỎI VÀ BÀI TẬP.....	79
CHƯƠNG 6: ĐỒ THỊ.....	80
6.1 CÁC KHÁI NIỆM CƠ BẢN.....	80
6.1.1 Đồ thị có hướng.....	80
6.1.2 Đồ thị vô hướng.....	81
6.1.3 Đồ thị có trọng số.....	81
6.2 BIỂU DIỄN ĐỒ THỊ.....	82

6.2.1 Biểu diễn đồ thị bằng ma trận kề	82
6.2.2 Biểu diễn đồ thị bằng danh sách kề	84
6.3 DUYỆT ĐỒ THỊ	84
6.3.1 Duyệt theo chiều sâu	84
6.3.2 Duyệt theo chiều rộng	86
6.3.3 Ứng dụng duyệt đồ thị để kiểm tra tính liên thông	88
6.4 TÓM TẮT CHƯƠNG 6	88
6.5 CÂU HỎI VÀ BÀI TẬP	89
CHƯƠNG 7: SẮP XẾP VÀ TÌM KIẾM	90
7.1 BÀI TOÁN SẮP XẾP	90
7.2 CÁC GIẢI THUẬT SẮP XẾP ĐƠN GIẢN	91
7.2.1 Sắp xếp chọn	91
7.2.2 Sắp xếp chèn	93
7.2.3 Sắp xếp nổi bọt	95
7.3 QUICK SORT	97
7.3.1 Giới thiệu	97
7.3.2 Các bước thực hiện giải thuật	98
7.3.3 Cài đặt giải thuật	99
7.4 HEAP SORT	100
7.4.1 Giới thiệu	100
7.4.2 Các thuật toán trên heap	101
7.5 MERGE SORT (SẮP XẾP TRỘN)	109
7.5.1 Giới thiệu	109
7.5.2 Trộn 2 dãy đã sắp	109
7.5.3 Sắp xếp trộn	113
7.6 BÀI TOÁN TÌM KIẾM	115
7.7 TÌM KIẾM TUẦN TỰ	115
7.8 TÌM KIẾM NHỊ PHÂN	115
7.9 CÂY NHỊ PHÂN TÌM KIẾM	117
7.9.1 Tìm kiếm trên cây nhị phân tìm kiếm	118
7.9.2 Chèn một phần tử vào cây nhị phân tìm kiếm	119
7.9.3 Xoá một nút khỏi cây nhị phân tìm kiếm	121
7.10 TÓM TẮT CHƯƠNG 5	122

7.11 CÂU HỎI VÀ BÀI TẬP.....	123
PHỤ LỤC I: HƯỚNG DẪN GIẢI CÂU HỎI VÀ BÀI TẬP.....	124
PHỤ LỤC II: MÃ NGUỒN THAM KHẢO.....	127
TÀI LIỆU THAM KHẢO	148

CHƯƠNG 1

PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT

Chương 1 trình bày các khái niệm về giải thuật và phương pháp tính chính 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, tính chính 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.

Đề 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 GIẢI THUẬT VÀ NGÔN NGỮ DIỄN ĐẠT GIẢI THUẬT

1.1.1 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.

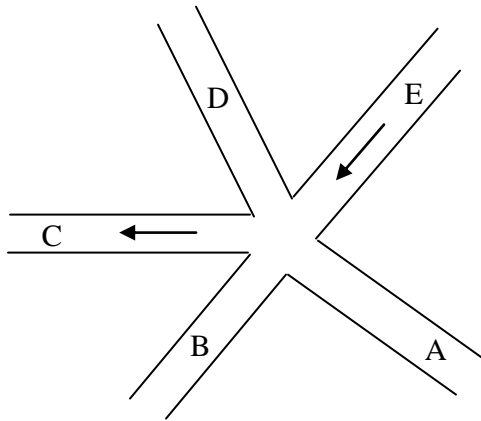
Như đã nói ở trên, mỗi lệnh trong thuật toán phải có ngữ nghĩa rõ ràng và có thể được thực thi trong một khoảng thời gian hữu hạn. Tuy nhiên, đôi khi một lệnh có ngữ nghĩa rõ ràng đối với người này nhưng lại không rõ ràng đối với người khác. Ngoài ra, thường rất khó để chứng minh một lệnh có thể được 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 gắn trạng thái của các đèn giao

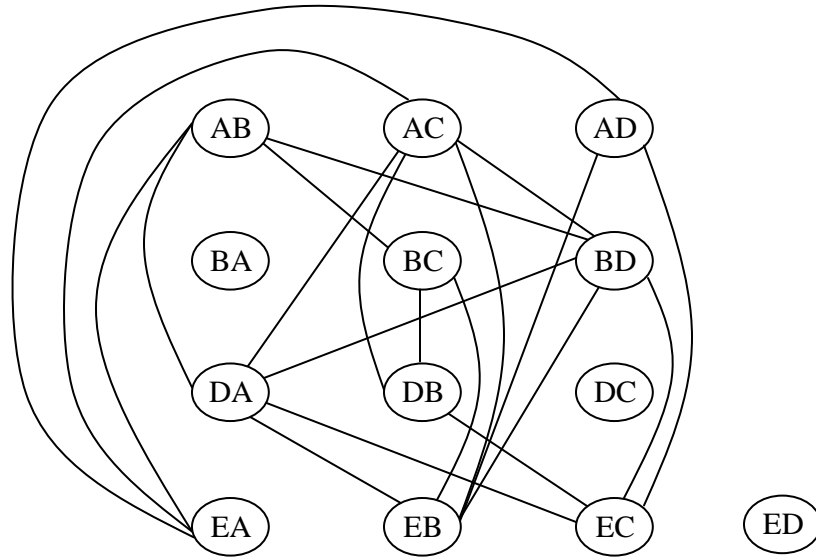
thông với mỗi nhóm vừa được phân chia. Với cách phân chia có số nhóm ít nhất, ta sẽ xây dựng được 1 hệ thống đèn giao thông có ít trạng thái nhất.

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 .

	AB	AC	AD	BA	BC	BD	DA	DB	DC	EA	EB	EC	ED
AB					1	1	1			1			
AC						1	1	1		1	1		
AD										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. Để 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”.

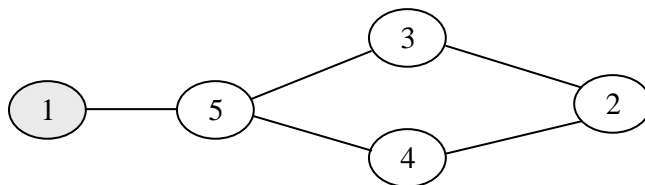
Đối với bài toán tô màu đồ thị, một thuật toán cảm tính thường được sử dụng là thuật toán “tham ăn” (greedy). Theo thuật toán này, đầu tiên ta sử dụng một màu để tô nhiều nhất số đỉnh có thể, thỏa mãn yêu cầu bài toán. Tiếp theo, sử dụng màu thứ 2 để tô các đỉnh chưa được tô trong bước 1, rồi sử dụng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

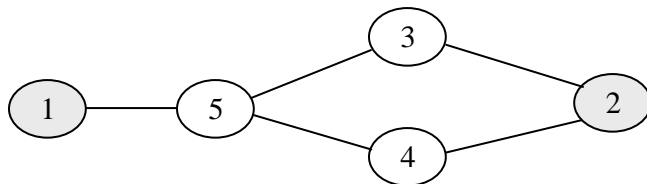
- Lựa chọn 1 đỉnh chưa được tô màu và tô nó bằng màu mới.
- Duyệt qua các đỉnh chưa được tô màu. Với mỗi đỉnh dạng này, kiểm tra xem có cạnh nào nối nó với một đỉnh vừa được tô bởi màu mới hay không. Nếu không có cạnh nào thì ta tô màu đỉnh này bằng màu mới.

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 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.

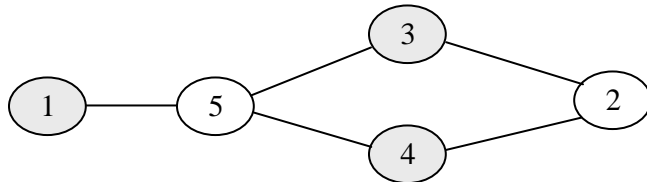
Ví dụ, xem xét đồ thị ở hình 1.3, trong đó đỉnh 1 đã được tô màu đỏ. Ta thấy rằng hoàn toàn có thể tô cả 2 đỉnh 3 và 4 là màu đỏ, với điều kiện ta không tô đỉnh số 2 màu đỏ. Tuy nhiên, nếu ta áp dụng thuật toán tham ăn theo thứ tự các đỉnh lớn dần thì đỉnh 1 và đỉnh 2 sẽ là màu đỏ, và khi đó đỉnh 3,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



a) Đồ thị ban đầu



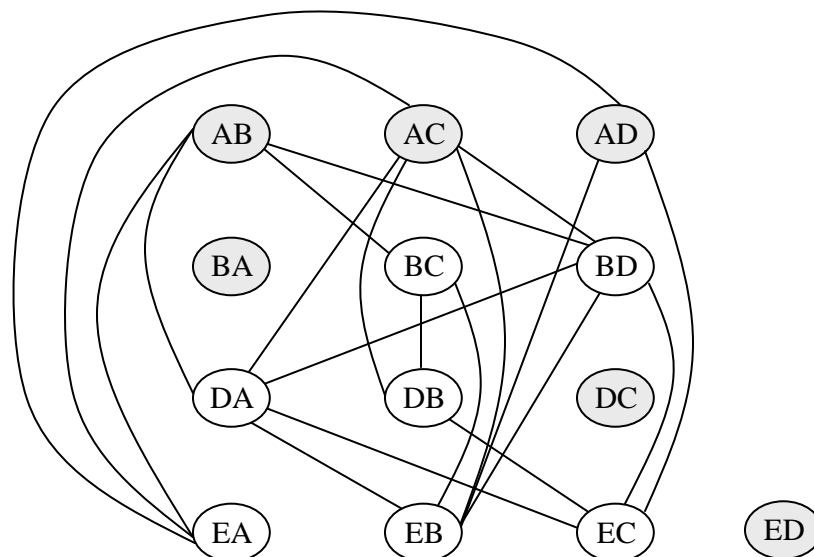
b) Tô màu theo thuật toán tham ăn



c) Một cách tô màu tốt hơn

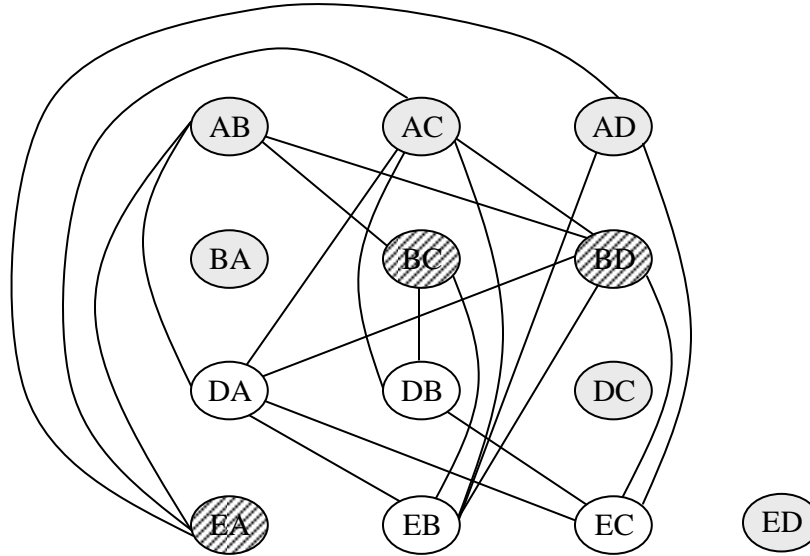
Hình 1.3 Đồ thị

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.



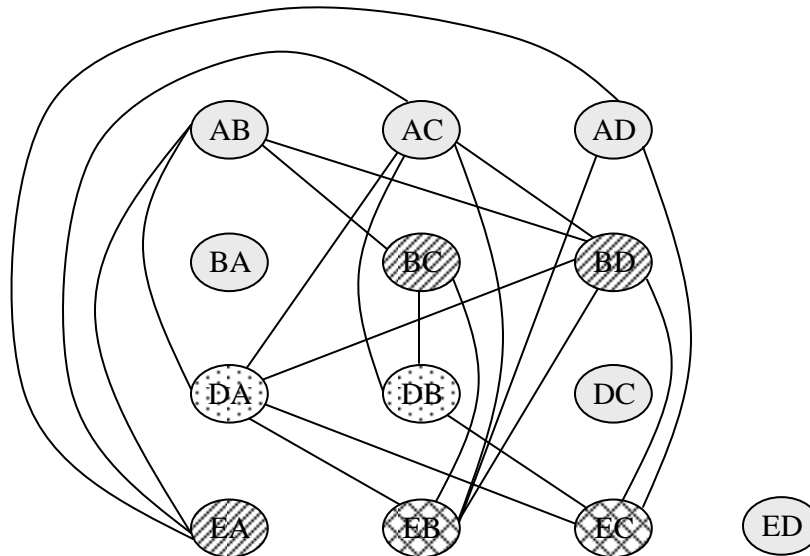
Hình 1.4 Tô màu xanh cho các đỉnh của đồ thị ngã rẽ

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.

1.1.2 Ngôn ngữ diễn đạt giải thuật và kỹ thuật tinh chỉnh từng bước

Sau khi đã xây dựng được mô hình toán học cho vấn đề cần giải quyết, tiếp theo, ta có thể hình thành một thuật toán cho mô hình đó. Phiên bản đầu tiên của thuật toán thường được diễn tả dưới dạng các phát biểu tương đối tổng quát, và sau đó sẽ được tinh chỉnh dần từng bước thành chuỗi các lệnh cụ thể, rõ ràng hơn. Ví dụ trong thuật toán tham ăn ở trên, ta mô tả bước thực hiện ở mức tổng quát là “Lựa chọn 1 đỉnh chưa được tô màu”. Với phát biểu như vậy, ta hy vọng rằng người đọc có thể nắm được ý tưởng thực hiện thao tác. Tuy nhiên, để chuyển các phát biểu đó thành chương trình máy tính, cần phải qua 1 số bước tinh chỉnh cho tới khi đạt đến mức các phát biểu đều có thể được chuyển đổi trực tiếp sang các lệnh của ngôn ngữ lập trình.

Trở lại ví dụ về bài toán tô màu đồ thị bằng thuật toán tham ăn. Ta sẽ xem xét việc mô tả thuật toán từ mức tổng quát cho tới một số mức cụ thể hơn. Tại bước nào đó, giả sử ta có đồ thị G có 1 số đỉnh đã được tô màu theo quy tắc đã nói ở trên. Thủ tục *Tham_an* dưới đây sẽ xác định 1 tập các đỉnh chưa được tô màu thuộc G mà có thể cùng được tô bởi 1 màu mới. Thủ tục này sẽ được gọi đi gọi lại nhiều lần cho tới khi tất cả các đỉnh của G đã được tô màu. Ở mức tổng quát, thủ tục được mô tả như sau:

```

void Tham_an(GRAPH: G, SET: Mau_moi)
{
    Mau_moi = Tập rỗng;
    For mỗi đỉnh v chưa được tô màu thuộc G
        If v không được nối tới đỉnh nào trong tập Mau_moi
        {
            Tô màu mới cho đỉnh v;
            Đưa v vào tập Mau_moi;
        }
}

```

Trong thủ tục trên, ta sử dụng một ngôn ngữ diễn đạt giải thuật tựa như ngôn ngữ lập trình C. Trong ngôn ngữ này, các lệnh được mô tả dưới dạng ngôn ngữ tự nhiên nhưng vẫn tuân theo cú pháp của ngôn ngữ lập trình.

Ta nhận thấy rằng các phát biểu trong thủ tục trên còn rất tổng quát, và chưa tương ứng với các lệnh trong ngôn ngữ lập trình, chẳng hạn các điều kiện kiểm tra trong câu lệnh For và If ở mức mô tả hiện tại là không thực hiện được trong C. Để thủ tục có thể thực thi được, ta cần phải tinh chỉnh một số bước để có thể chuyển đổi về chương trình trong ngôn ngữ lập trình C thông thường.

Đầu tiên, ta xem xét lệnh If ở trên. Để kiểm tra xem đỉnh v có nối tới một đỉnh nào đó trong tập Mau_moi hay không, ta xem xét từng đỉnh w trong Mau_moi và sử dụng đồ thị G để kiểm tra xem có tồn tại cạnh nối v và w không. Để lưu giữ kết quả kiểm tra, ta sử dụng một biến ton_tai. Khi đó, thủ tục được tinh chỉnh như sau:

```

void Tham_an(GRAPH: G, SET: Mau_moi)
{
    int ton_tai;
    Mau_moi = Tập rỗng;
    For mỗi đỉnh v chưa được tô màu thuộc G
    {
        ton_tai = 0;
        For mỗi đỉnh w thuộc Mau_moi
            If tồn tại cạnh nối v và w trong G
                ton_tai = 1;
        If ton_tai == 1
        {
            Tô màu mới cho đỉnh v;
            Đưa v vào tập Mau_moi;
        }
    }
}

```

Như vậy, ta có thể thấy rằng điều kiện kiểm tra trong phát biểu If đã được mô tả cụ thể hơn bằng các phát nhỏ hơn, và các phát biểu này có thể dễ dàng chuyển thành các lệnh cụ thể trong C. Tiếp theo, ta sẽ tinh chỉnh các vòng lặp For để duyệt qua các đỉnh thuộc G và thuộc Mau_moi. Để làm điều này, tốt nhất là ta thay For bằng một vòng lặp While, biến v ban đầu được gán là phần tử đầu tiên chưa tô màu trong tập G, và tại mỗi bước lặp, biến v sẽ được thay bằng phần tử chưa tô màu tiếp theo trong G. Vòng lặp F bên trong có thể thực hiện tương tự.

```
Void Tham_an (GRAPH: G, SET: Mau_moi)
{
    int ton_tai;
    int v, w
    Mau_moi = Tập rỗng;
    v = đỉnh chưa tô màu đầu tiên trong G ;
    While v != NULL
    {
        ton_tai = 0;
        w = đỉnh đầu tiên trong Mau_moi;
        While w != NULL{
            If tồn tại cạnh nối v và w trong G
                ton_tai = 1;
            w = đỉnh tiếp theo trong Mau_moi ;
        }
        If ton_tai == 1
        {
            Tô màu mới cho đỉnh v;
            Đưa v vào tập Mau_moi;
        }
        v = đỉnh chưa tô màu tiếp theo trong G;
    }
}
```

Như vậy, ta thấy các phát biểu trong thủ tục đã khá cụ thể, tuy nhiên, để chuyển đổi thành chương trình trong ngôn ngữ C thì cần tới vài bước tinh chỉnh nữa. Bạn đọc hãy xem như đó là bài tập và tự giải để hiểu rõ về ngôn ngữ diễn đạt giải thuật cũng như kỹ thuật tinh chỉnh từng bước.

1.2 PHÂN TÍCH 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.

1.2.1 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.

1.2.2 Tính toán thời gian thực hiện chương trình

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]){
4.                 // Đổi chỗ cho a[j] và a[j-1]
5.                 temp = a[j-1];
6.                 a[j-1] = a[j];
7.                 a[j] = temp;
8.             }
9.     }

```

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 thoả 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.

1.3 TÓM TẮT CHƯƠNG 1

Các kiến thức cần nhớ trong chương 1:

- 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.
- Thuật toán thường được mô tả bằng các ngôn ngữ diễn đạt giải thuật gần với ngôn ngữ tự nhiên. Các mô tả này sẽ được tinh chỉnh dần dần để đạt tới mức ngôn ngữ lập trình.
- Thời gian thực hiện thuật toán thường được coi như là 1 hàm của kích thước dữ liệu đầu vào.
- Thời gian thực hiện thuật toán thường được tính trong các trường hợp tốt nhất, xấu nhất, hoặc trung bình.
- 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 $n \geq n_0$.
- Cấp độ tăng về thời gian thực hiện của chương trình cho phép ta xác định độ lớn của bài toán mà ta có thể giải quyết.
- Quy tắc cộng cấp độ tăng: 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 $O(\max(f(n), g(n)))$.
- Quy tắc nhân cấp độ tăng: 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 $T_1(n).T_2(n) = O(f(n).(g(n)))$.

1.4 CÂU HỎI VÀ BÀI TẬP

1. Trình bày khái niệm thuật toán? Các đặc điểm của thuật toán?
2. Trong một giải vô địch bóng đá có 6 đội bóng đá vòng tròn. Các đội là A, B, C, D, E, F. Đội A đã đá với B và C. Đội B đã đá với D và F. Đội E đã đá với C và F. Mỗi đội đá mỗi tuần 1 trận. Hãy lập 1 lịch cho các đội bóng sao cho tất cả các đội đều đá đủ số trận quy định trong 1 số tuần vừa phải. Thực hiện phân tích, thiết kế thuật toán. Biểu diễn thuật toán bằng ngôn ngữ diễn đạt giải thuật, sau đó tinh chỉnh về dạng ngôn ngữ lập trình C.
3. Thời gian thực hiện một chương trình thường phụ thuộc vào các yếu tố nào? Phân tích cụ thể từng yếu tố.
4. Nói thời gian thực hiện chương trình là $T(n) = O(f(n))$ có nghĩa là gì? Cho ví dụ minh họa.
5. Nêu quy tắc cộng và nhân cấp độ tăng của hàm. Có ví dụ minh họa.
6. Nêu các quy tắc chung trong việc phân tích và đánh giá thời gian thực hiện chương trình.

CHƯƠNG 2

ĐỆ QUI

Chương 2 trình bày các khái niệm về định nghĩa đệ qui, chương trình đệ qui. Ngoài việc trình bày các ưu điểm của chương trình 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 phần 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 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 chuỗi ký tự bằng đệ qui:

- Chuỗi rỗng là 1 chuỗi ký tự.
- Một chữ cái bất kỳ ghép với 1 chuỗi sẽ tạo thành 1 chuỗi mới.

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

2.1.1 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.

2.1.2 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. Để minh họa cho điều này, chúng ta sẽ xem xét một ví dụ, trong đó cả đệ qui và lặp đều có thể được sử dụng. Tuy nhiên, ta sẽ phân tích để thấy sử dụng đệ qui trong trường hợp này gây lãng phí bộ nhớ và các tính toán không cần thiết như thế nào.

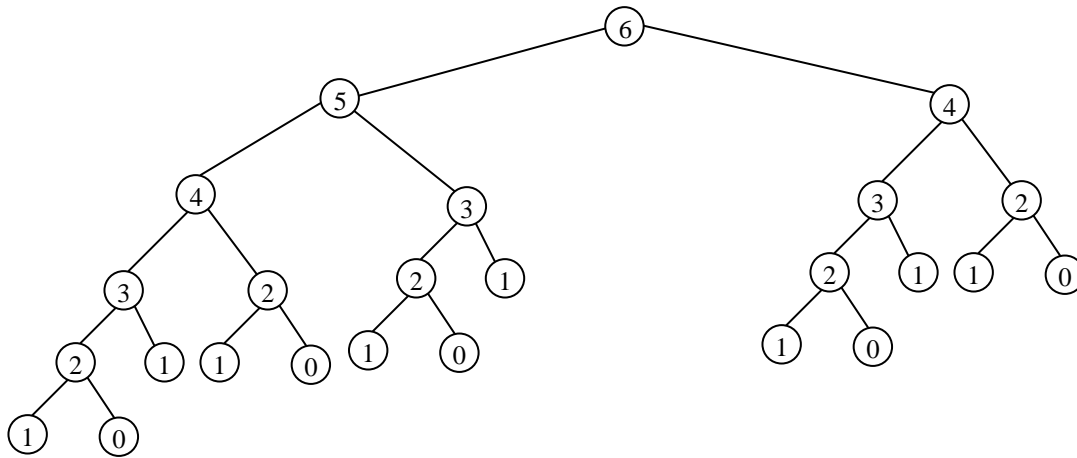
Xét bài toán tính các phần tử của dãy Fibonacci. Dãy Fibonacci được định nghĩa như sau:

- $F(0) = 0$
- $F(1) = 1$
- 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.

2.2 THIẾT KẾ GIẢI THUẬT ĐỆ QUI

2.2.1 Chương trình tính hàm n!

Theo định nghĩa đã trình bày ở phần trước, $n! = 1$ nếu $n=0$, ngược lại, $n! = (n-1)! * n$.

```

int giaithua (int n){
    if (n==0) return 1;
    else return giaithua(n-1) * n;
}

```

Trong chương trình trên, điểm dừng của thuật toán đệ qui là khi $n=0$. Khi đó, giá trị của hàm `giaithua(0)` có thể tính được ngay lập tức mà không cần gọi hàm đệ qui khác. Nếu điều kiện dừng

không thỏa mãn, sẽ có một lời gọi đệ qui hàm giai thừa với tham số là $n-1$, nhỏ hơn tham số ban đầu 1 đơn vị (tức là bài toán tính $n!$ đã được quy về bài toán đơn giản hơn là tính $(n-1)!$).

2.2.2 Thuật toán Euclid tính ước số chung lớn nhất của 2 số nguyên dương

Ước số chung lớn nhất (USCLN) của 2 số nguyên dương m, n là 1 số k lớn nhất sao cho m và n đều chia hết cho k . Một phương pháp đơn giản nhất để tìm USCLN của m và n là duyệt từ số nhỏ hơn trong 2 số m, n cho đến 1, ngay khi gặp số nào đó mà m và n đều chia hết cho nó thì đó chính là USCLN của m, n . Tuy nhiên, phương pháp này không phải là cách tìm USCLN hiệu quả. Cách đây hơn 2000 năm, Euclid đã phát minh ra một giải thuật tìm USCLN của 2 số nguyên dương m, n rất hiệu quả. Ý tưởng cơ bản của thuật toán này cũng tương tự như ý tưởng đệ qui, tức là đưa bài toán về 1 bài toán đơn giản hơn. Cụ thể, giả sử m lớn hơn n , khi đó việc tính USCLN của m và n sẽ được đưa về bài toán tính USCLN của $m \bmod n$ và n vì $\text{USCLN}(m, n) = \text{USCLN}(m \bmod n, n)$.

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.

2.2.3 Các giải thuật đệ qui dạng chia để trị (divide and conquer)

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 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:

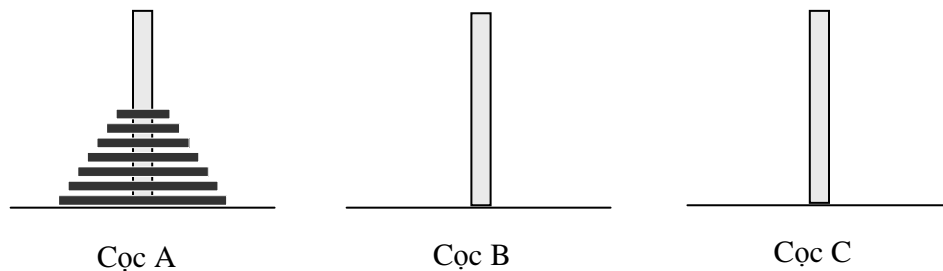
Giải sử ta cần sắp xếp 1 dãy các số theo chiều tăng dần. Tiến hành chia dãy đó thành 2 nửa sao cho các số trong nửa đầu đều nhỏ hơn các số trong nửa sau. Sau đó, tiến hành thực hiện sắp xếp trên

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.

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.

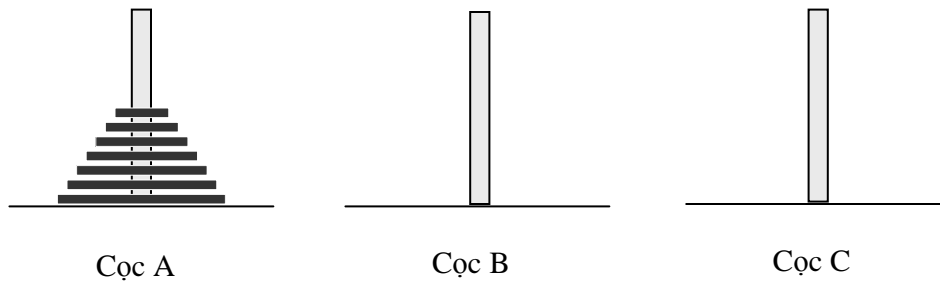
Như vậy, cả 2 đĩa đã được chuyển sang cọc đích C và không có tình huống nào đĩa lớn nằm trên đĩa nhỏ.

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

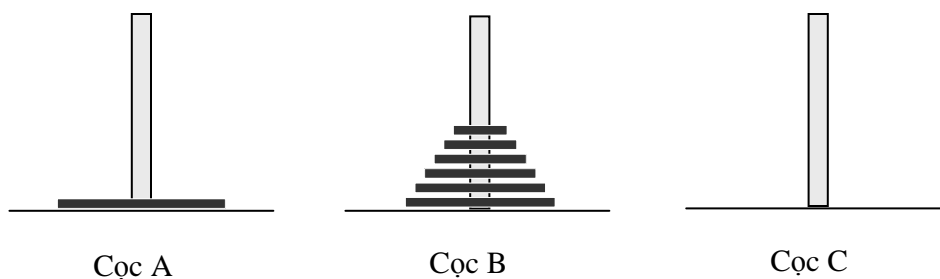
- Lấy cọc đích C làm cọc trung gian để chuyển $n-1$ đĩa bên trên sang cọc trung gian B.
- Chuyển cọc dưới cùng (cọc thứ n) sang cọc đích C.
- Lấy cọc ban đầu A làm cọc trung gian để chuyển $n-1$ đĩa từ cọc trung gian B sang cọc đích C.

Có thể minh họa quá trình chuyển này như sau:

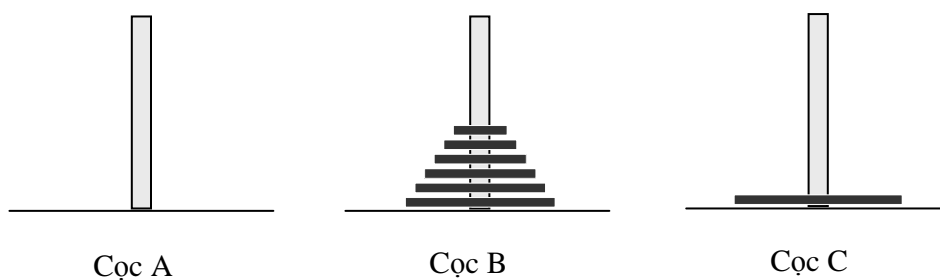
Trạng thái ban đầu:



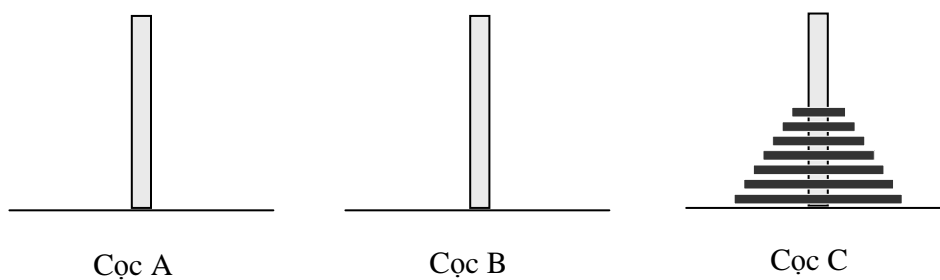
Bước 1: Chuyển $n-1$ đĩa bên trên từ cọc A sang cọc B, sử dụng cọc C làm cọc trung gian.



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



Bước 3: Chuyển $n-2$ đĩa từ cọc B sang cọc C sử dụng cọc A làm cọc trung gian.



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

Ở đây, ta thấy rằng bài toán chuyển n cọc đã được chuyển về bài toán đơn giản hơn là chuyển $n-1$ cọc. Điểm dừng của thuật toán đệ qui là khi $n=1$ và ta chuyển thẳng cọc này từ cọc ban đầu sang cọc 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 cọc a sang cọc trung gian b , và lần thứ 2 chuyển $n-1$ đĩa từ cọc trung gian b sang cọc 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 cọc a sang cọc c .
- Hàm `thaphanoi(int n, int a, int c, int b)` là hàm đệ qui thực hiện việc chuyển n đĩa từ cọc a sang cọc c , sử dụng cọc trung gian là cọc 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;
}

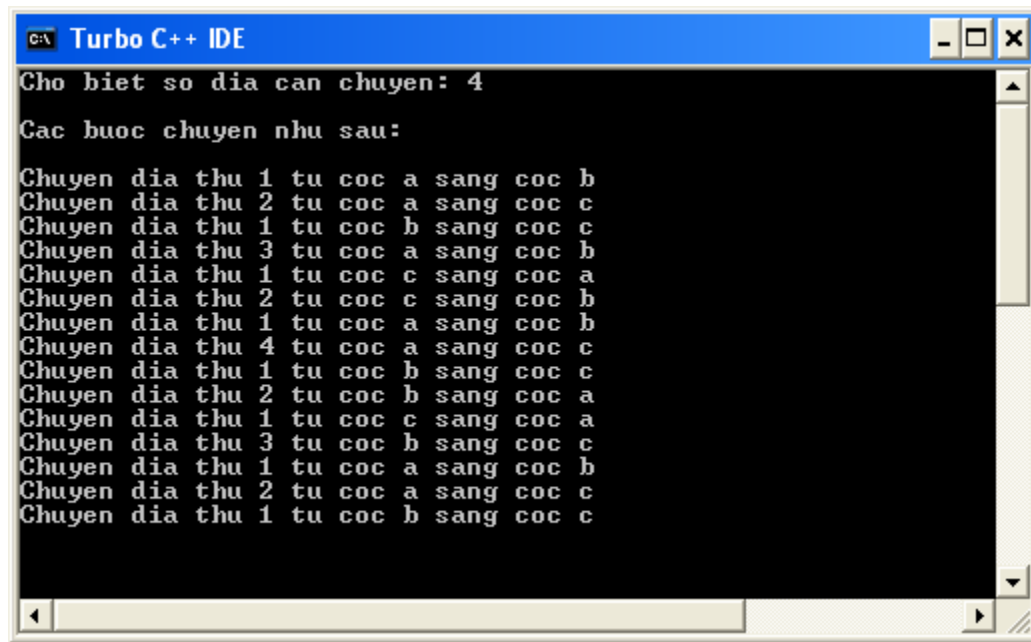
void thaphanoi(int n, char a, char c, char b){
    if (n==1) chuyen(1, a, c);
    else{
        thaphanoi(n-1, a, b, c);
        chuyen(n, a, c);
        thaphanoi(n-1, b, c, a);
    }
    return;
}
```

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- Lời 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- Lời 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
Cac 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;
}

```

2.2.4 Thuật toán quay lui (backtracking algorithms)

Như chúng ta đã biết, các thuật toán được xây dựng để giải quyết vấn đề thường đưa ra 1 quy tắc tính toán nào đó. Tuy nhiên, có những vấn đề không tuân theo 1 quy tắc, và khi đó ta phải dùng phương pháp thử - sai (trial-and-error) để giải quyết. Theo phương pháp này, quá trình thử - sai được xem xét trên các bài toán đơn giản hơn (thường chỉ là 1 phần của bài toán ban đầu). Các bài toán này thường được mô tả dưới dạng đệ qui và thường liên quan đến việc giải quyết một số hữu hạn các bài toán con.

Để hiểu rõ hơn thuật toán này, chúng ta sẽ xem xét 1 ví dụ điển hình cho thuật toán quay lui, đó là bài toán Mã đi tuần.

Cho bàn cờ có kích thước $n \times n$ (có n^2 ô). Một quân mã được đặt tại ô ban 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 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

y

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 sau:

```
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)
        {
            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];
int dx[8]={2,1,-1,-2,-2,-1,1,2};
int dy[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)
            {
                ThuNuocTiepTheo(i+1, u, v, q1);
                if ((*q1)==0) Banco[u][v]=0;
            }else (*q1)=1;
        }
        k++;
    }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]);
            else printf("%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);
}

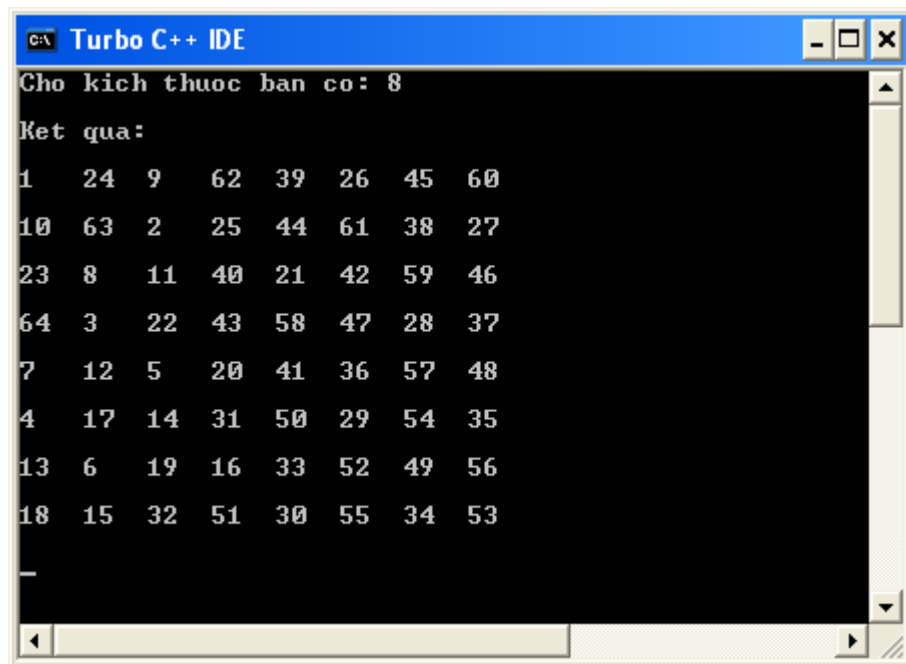
```

```

    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

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:

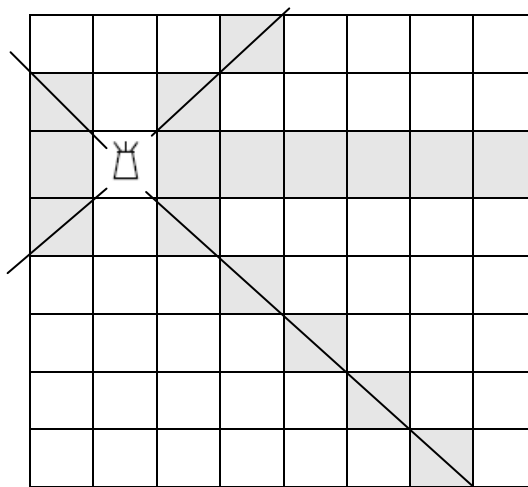
```
void DatHau(int i)
```

```

{
    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 $a[j]$, $b[i + j]$, 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];
```

Vậy 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))
}

```

2.3 TÓM TẮT CHƯƠNG 2

Các kiến thức cần nhớ trong chương 2:

- Định nghĩa bằng đệ qui: 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.
- Chương trình đệ qui: Một chương trình máy tính gọi là đệ qui nếu trong chương trình có lời gọi chính nó (có kiểm tra điều kiện dừng).
- Để viết một chương trình 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 (có hỗ trợ hàm và thủ tục).
- Nếu chương trình có thể viết dưới dạng lặp hoặc các cấu trúc lệnh khác thì không nên sử dụng đệ qui.
- Các thuật toán đệ qui dạng “chia để trị” là các thuật toán 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 qui để giải quyết.
- Thuật toán quay lui dùng để giải quyết các bài toán không tuân theo 1 quy tắc, và khi đó ta phải dùng phương pháp thử - sai (trial-and-error) để giải quyết. Theo phương pháp này, quá trình thử - sai được xem xét trên các bài toán đơn giản hơn (thường chỉ là 1 phần của bài toán ban đầu). Các bài toán này thường được mô tả dưới dạng đệ qui và thường liên quan đến việc giải quyết một số hữu hạn các bài toán con.

2.4 CÂU HỎI VÀ BÀI TẬP

1. Hãy trình bày một số ví dụ về định nghĩa theo kiểu đệ qui.
2. Một chương trình đệ qui khi gọi chính nó thì bài toán khi đó có kích thước như thế nào so với bài toán ban đầu? Để chương trình đệ qui không bị lặp vô hạn thì cần phải làm gì?

3. Hãy cho biết tại sao khi chương trình có thể viết dưới dạng lặp hoặc cấu trúc khác thì không nên sử dụng đệ qui?
4. Viết chương trình đệ qui tính tổng các số lẻ trong khoảng từ 1 đến $2n+1$.
5. Hãy cho biết các bước thực hiện chuyển đĩa trong bài toán tháp Hà nội với số lượng đĩa là 5.
6. Hoàn thiện mã nguồn cho bài toán 8 quân hậu và chạy thử cho ra kết quả.

CHƯƠNG 3

MẢNG VÀ DANH SÁCH LIÊN KẾT

Chương 3 giới thiệu về các kiểu dữ liệu danh sách, bao gồm kiểu dữ liệu cơ sở mảng và kiểu danh sách nâng cao là danh sách liên kết. Ngoài phần giới thiệu sơ lược về mảng, chương 3 tập trung vào các kiểu danh sách liên kết.

Phần danh sách liên kết đơn giới thiệu các khái niệm danh sách, các thao tác cơ bản trên danh sách như chèn phần tử, xoá phần tử, duyệt qua toàn bộ danh sách. Cuối phần là một ví dụ về sử dụng danh sách liên kết đơn để biểu diễn 1 đa thức.

Chương này cũng đề cập tới một số kiểu danh sách liên kết khác như danh sách liên kết vòng và danh sách liên kết kép.

Để học tốt chương này, sinh viên cần nắm vững lý thuyết và tìm tòi một số ví dụ khác minh hoạ cho việc sử dụng mảng và danh sách liên kết.

3.1 CẤU TRÚC DỮ LIỆU KIỂU MẢNG (ARRAY)

Có thể nói, mảng là cấu trúc dữ liệu căn bản và được sử dụng rộng rãi nhất trong tất cả các ngôn ngữ lập trình. Một mảng là 1 tập hợp cố định các thành phần có cùng 1 kiểu dữ liệu, được lưu trữ kế tiếp nhau và có thể được truy cập thông qua một chỉ số. Ví dụ, để truy cập tới phần tử thứ i của mảng a , ta viết $a[i]$. Chỉ số này phải là số nguyên không âm và nhỏ hơn kích thước của mảng (số phần tử của mảng). Trong chương trình, chỉ số này không nhất thiết phải là các hằng số hoặc biến số, mà có thể là các biểu thức hoặc các hàm.

a_1	a_2	...	a_i	a_{i+1}	...	a_n
-------	-------	-----	-------	-----------	-----	-------

Lưu ý rằng cấu trúc của bộ nhớ máy tính cũng được tổ chức thành các ô nhớ, và cũng có thể truy cập ngẫu nhiên thông qua các địa chỉ. Do vậy, việc lưu trữ dữ liệu trong mảng có sự tương thích hoàn toàn với bộ nhớ máy tính, trong đó có thể coi toàn bộ bộ nhớ máy tính như 1 mảng, và địa chỉ các ô nhớ tương ứng như chỉ số của mảng. Chính vì sự tương thích này mà việc sử dụng cấu trúc dữ liệu mảng trong các ngôn ngữ lập trình có thể làm cho chương trình hiệu quả hơn và chạy nhanh hơn.

Mảng có thể có nhiều hơn 1 chiều. Khi đó, số các chỉ số của mảng sẽ tương ứng với số chiều. Chẳng hạn, trong mảng 2 chiều a , thành phần thuộc cột i , hàng j được viết là $a[i][j]$. Mảng 2 chiều còn được gọi là ma trận (matrix).

a_{11}	a_{21}	...	a_{i1}	a_{i+11}	...	a_{m1}
a_{12}	a_{22}	...	a_{i2}	a_{i+12}	...	a_{m2}
...
a_{1j}	a_{2j}	...	a_{ij}	a_{i+1j}	...	a_{mj}
a_{1j+1}	a_{2j+1}	...	a_{ij+1}	a_{i+1j+1}	...	a_{mj+1}
...
a_{1n}	a_{2n}	...	a_{in}	a_{i+1n}	...	a_{mn}

Như đã nói ở trên, mảng là cấu trúc dữ liệu dễ sử dụng, tốc độ truy cập cao. Tuy nhiên, nhược điểm chính của mảng là không linh hoạt về kích thước. Nghĩa là khi ta đã khai báo 1 mảng thì kích thước của nó là cố định, không thể thay đổi trong quá trình thực hiện chương trình. Điều này rất bất tiện khi ta chưa biết trước số phần tử cần xử lý. Nếu khai báo mảng lớn sẽ tốn bộ nhớ và ảnh hưởng đến hiệu suất của chương trình. Nếu khai báo mảng nhỏ sẽ dẫn đến thiếu bộ nhớ. Ngoài ra, việc bố trí lại các phần tử trong mảng cũng khá phức tạp, bởi vì mỗi phần tử đã có vị trí cố định trong mảng, và để bố trí 1 phần tử sang 1 vị trí khác, ta phải tiến hành “dồn” các phần tử có liên quan.

Trong phần tiếp theo, chúng ta sẽ xem xét một cấu trúc dữ liệu khác, cũng cho phép lưu trữ 1 tập các phần tử, nhưng có kích thước và cách bố trí linh hoạt hơn. Đó là cấu trúc dữ liệu danh sách liên kết.

3.2 DANH SÁCH LIÊN KẾT

3.2.1 Khái niệm

Khác với mảng, danh sách liên kết là 1 cấu trúc dữ liệu có kiểu truy cập tuần tự. Mỗi phần tử trong danh sách liên kết có chứa thông tin về phần tử tiếp theo, qua đó ta có thể truy cập tới phần tử này.

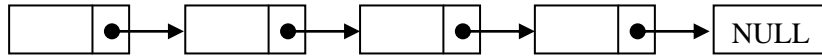
R. Sedgewick (Algorithms in Java - 2002) định nghĩa danh sách liên kết như sau:

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.

Nói “mỗi phần tử là 1 phần của 1 nút” bởi vì mỗi nút ngoài việc chứa thông tin về phần tử còn chứa thông tin về liên kết tới nút tiếp theo trong danh sách.

Có thể nói danh sách liên kết là 1 cấu trúc dữ liệu được định nghĩa kiểu đệ qui, vì trong định nghĩa 1 nút của danh sách có tham chiếu tới khái niệm nút. Thông thường, một nút thường có liên kết trở tới một nút khác, tuy nhiên nó cũng có thể trở tới chính nó.

Danh sách liên kết có thể được xem như là 1 sự bố trí tuần tự các phần tử trong 1 tập. Bắt đầu từ 1 nút, ta coi đó là phần tử đầu tiên trong danh sách. Từ nút này, theo liên kết mà nó trở tới, ta có nút thứ 2, được coi là phần tử thứ 2 trong danh sách, v.v. cứ tiếp tục như vậy cho đến hết danh sách. Nút cuối cùng có thể có liên kết là một liên kết null, tức là không trở tới nút nào, hoặc nó có thể trở về nút đầu tiên để tạo thành 1 vòng.



Hình 3.1 Danh sách liên kết

Như vậy, mặc dù cùng là cấu trúc dữ liệu bao gồm 1 tập các phần tử, nhưng giữa danh sách liên kết và mảng có 1 số điểm khác biệt sau:

- Mảng có thể được truy cập ngẫu nhiên thông qua chỉ số, còn danh sách chỉ có thể truy cập tuần tự. Trong danh sách liên kết, muốn truy cập tới 1 phần tử phải bắt đầu từ đầu danh sách sau đó lần lượt qua các phần tử kế tiếp cho tới khi đến phần tử cần truy cập.
- Việc bố trí, sắp đặt lại các phần tử trong 1 danh sách liên kết đơn giản hơn nhiều so với mảng. Bởi vì đối với danh sách liên kết, để thay đổi vị trí của 1 phần tử, ta chỉ cần thay đổi các liên kết của một số phần tử có liên quan, còn trong mảng, ta thường phải thay đổi vị trí của rất nhiều phần tử.
- Do bản chất động của danh sách liên kết, kích thước của danh sách liên kết có thể linh hoạt hơn nhiều so với mảng. Kích thước của danh sách không cần phải khai báo trước, bất kỳ lúc nào có thể tạo mới 1 phần tử và thêm vào vị trí bất kỳ trong danh sách. Nói cách khác, mảng là 1 tập có số lượng cố định các phần tử, còn danh sách liên kết là 1 tập có số lượng phần tử không cố định.

Để khai báo một danh sách trong C, ta có thể dùng cấu trúc tự trở. Ví dụ, để khai báo một danh sách liên kết mà mỗi nút chứa một phần tử là số nguyên như sau:

```
struct node {
    int item;
    struct node *next;
};
typedef struct node *listnode;
```

Đầu tiên, ta khai báo một cấu trúc `node` bao gồm 2 thành phần. Thành phần thứ nhất là 1 biến nguyên chứa dữ liệu, thành phần thứ 2 là một con trỏ chứa địa chỉ của nút kế tiếp. Tiếp theo, ta định nghĩa một kiểu dữ liệu con trỏ tới nút có tên là `listnode`.

Với các danh sách liên kết có kiểu phần tử phức tạp hơn, ta phải khai báo cấu trúc của phần tử này trước (`itemstruct`), sau đó đưa kiểu cấu trúc đó vào kiểu phần tử trong cấu trúc `node`.

```
struct node {
    itemstruct item;
    struct node *next;
};
typedef struct node *listnode;
```

3.2.2 Các thao tác cơ bản trên danh sách liên kết

Như đã nói ở trên, với tính chất động của danh sách liên kết, các nút của danh sách không được

tạo ra ngay từ đầu mà chỉ được tạo ra khi cần thiết. Do vậy, thao tác đầu tiên cần có trên danh sách là tạo và cấp phát bộ nhớ cho 1 nút. Tương ứng với nó là thao tác giải phóng bộ nhớ và hủy 1 nút khi không dùng đến nữa.

Thao tác tiếp theo cần xem xét là việc chèn 1 nút đã tạo vào danh sách. Do cấu trúc đặc biệt của danh sách liên kết, việc chèn nút mới vào đầu, cuối, hoặc giữa danh sách có một số điểm khác biệt. Do vậy, cần xem xét cả 3 trường hợp. Tương tự như vậy, việc loại bỏ 1 nút khỏi danh sách cũng sẽ được xem xét trong cả 3 trường hợp. Cuối cùng là thao tác duyệt qua toàn bộ danh sách.

Trong phần tiếp theo, ta sẽ xem xét chi tiết việc thực hiện các thao tác này, được thực hiện trên danh sách liên kết có phần tử của nút là 1 số nguyên như khai báo đã trình bày ở trên.

3.2.2.1 Tạo, cấp phát, và giải phóng bộ nhớ cho 1 nút

```
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;
```

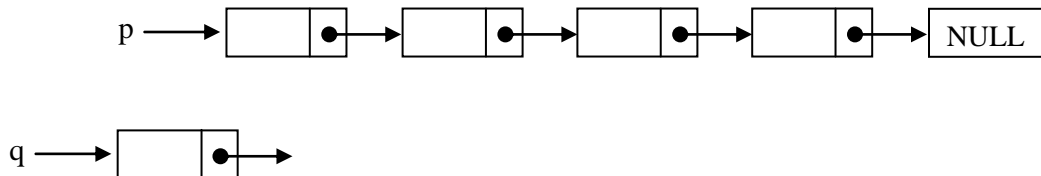
3.2.2.2 Chèn một nút vào đầu danh sách

Giả sử ta có 1 danh sách mà đầu của danh sách được trỏ tới bởi con trỏ p.

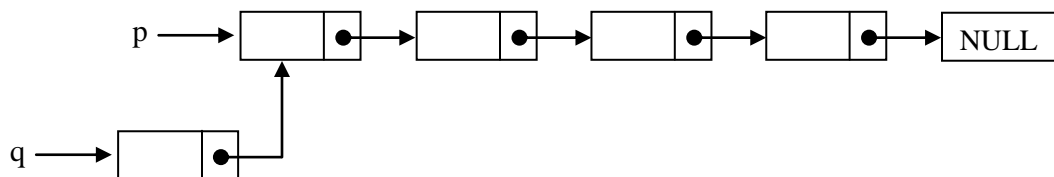


Các bước để chèn 1 nút mới vào đầu danh sách như sau:

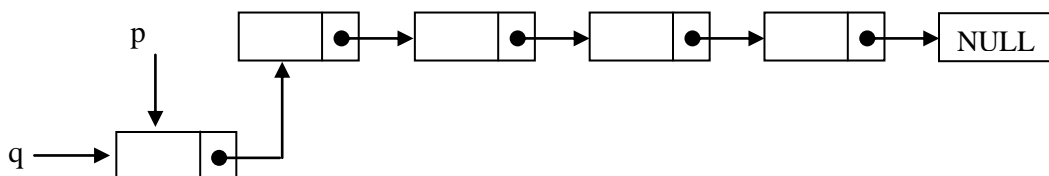
- Tạo và cấp phát bộ nhớ cho 1 nút mới. Nút này được trỏ tới bởi q.



- Sau khi gán các giá trị thích hợp cho phần tử của nút mới, cho con trỏ tiếp của nút mới trỏ đến phần tử đầu tiên của nút.



- Cuối cùng, để p vẫn trỏ đến nút đầu danh sách, ta cần cho p trỏ đến nút mới tạo.

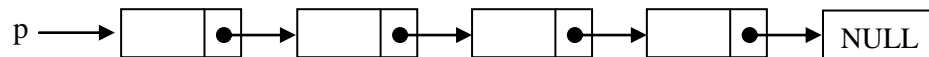


Chú ý rằng các bước trên phải làm đúng trình tự, nếu làm sai sẽ dẫn đến mất dữ liệu. Chẳng hạn, nếu ta cho con trỏ p trở đến nút mới tạo trước, thì khi đó nút mới tạo sẽ không trở tới được nút đầu danh sách cũ, vì không còn biến nào lưu trữ vị trí này nữa.

```
void Insert_Begin(listnode *p, int x){
    listnode q;
    q = (listnode)malloc(sizeof(struct node));
    q->item = x;
    q->next = *p;
    *p = q;
}
```

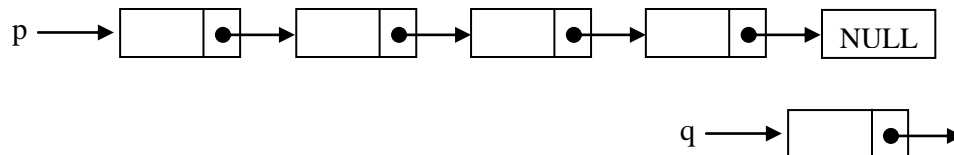
3.2.2.3 Chèn một nút vào cuối danh sách

Giả sử ta có 1 danh sách mà đầu của danh sách được trỏ tới bởi con trỏ p.

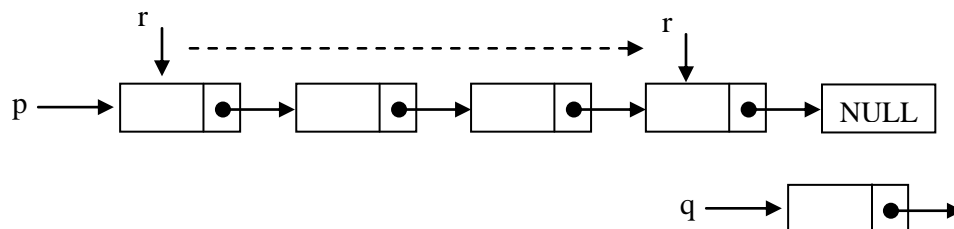


Các bước để chèn 1 nút mới vào cuối danh sách như sau (thực hiện đúng theo trình tự):

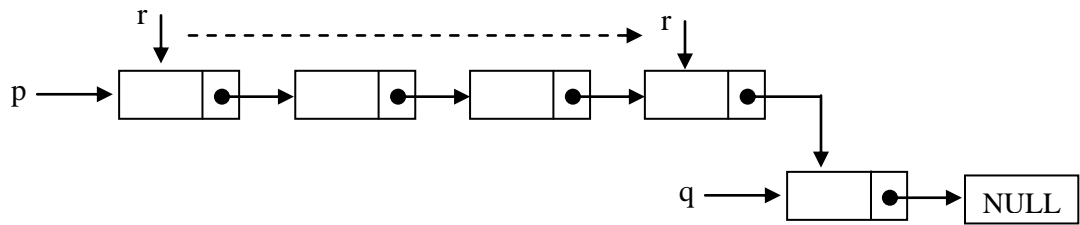
- Tạo và cấp phát bộ nhớ cho 1 nút mới. Nút này được trỏ tới bởi q.



- Dịch chuyển con trỏ tới nút cuối của danh sách. Để làm được việc này, ta phải khai báo 1 biến con trỏ mới r. Ban đầu, biến này, cũng với p, trỏ đến đầu danh sách. Lần lượt dịch chuyển r theo các nút kế tiếp cho tới khi đến cuối danh sách.



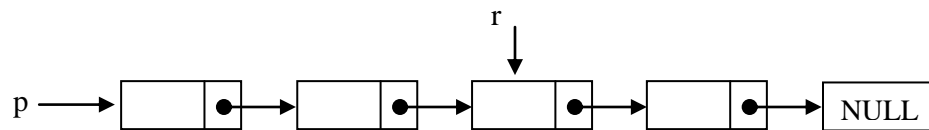
- Cho con trỏ tiếp của nút cuối (được trỏ tới bởi r) trỏ đến nút mới tạo là q, và cho con trỏ tiếp của q trỏ tới null.



```
void Insert_End(listnode *p, int x){
    listnode q, r;
    q = (listnode)malloc(sizeof(struct node));
    q->item = x;
    q->next = NULL;
    if (*p==NULL) *p = q;
    else{
        r = *p;
        while (r->next != NULL) r = r->next;
        r->next = q;
    }
}
```

3.2.2.4 Chèn một nút vào trước nút *r* trong danh sách

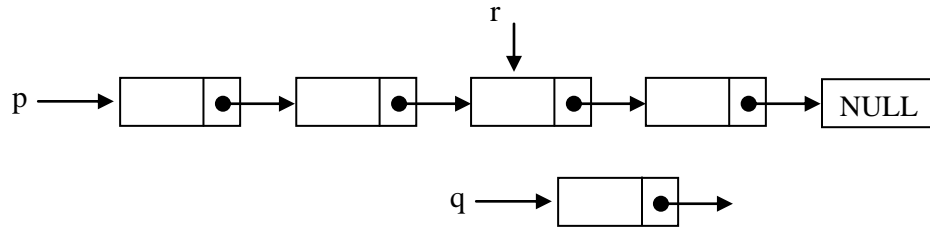
Giả sử ta có 1 danh sách mà đầu của danh sách được trỏ tới bởi con trỏ *p*, và 1 nút *r* trong danh sách.



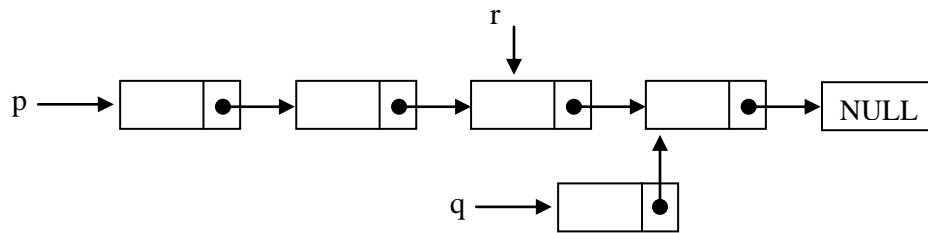
Ta giả thiết rằng nút *r* không phải là nút cuối cùng của danh sách, vì nếu như vậy, ta chỉ cần thực hiện thao tác chèn 1 nút vào cuối danh sách như đã trình bày ở trên.

Các bước để chèn 1 nút mới vào trước nút *r* trong danh sách như sau (thực hiện đúng theo trình tự):

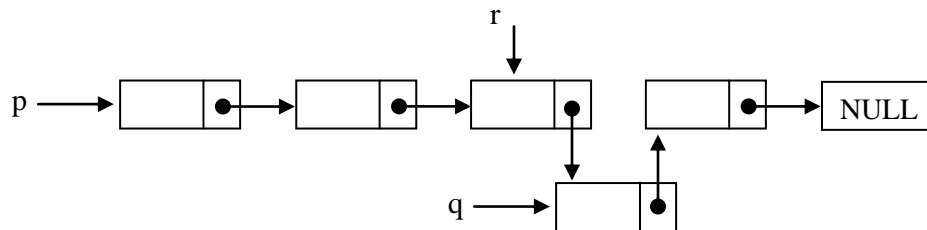
- Tạo và cấp phát bộ nhớ cho 1 nút mới. Nút này được trỏ tới bởi *q*.



- Cho con trỏ tiếp của nút mới trở đến nút kế tiếp của nút r.



- Cho con trỏ tiếp của nút r trở đến q.



```
void Insert_Middle(listnode *p, int position, int x){
    int count=1, found=0;
    listnode q, r;
    r = *p;
    while ((r != NULL)&&(found==0)){
        if (count == position){
            q = (listnode)malloc(sizeof(struct node));
            q-> item = x;
            q-> next = r-> next;
            r-> next = q;
            found = 1;
        }
        count ++;
        r = r-> next;
    }
}
```

```

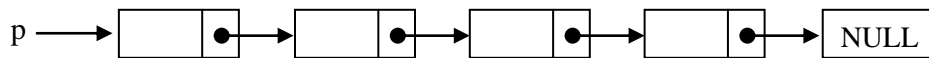
        r = r-> next;
    }
    if (found==0)
        printf("Khong tim thay vi tri can chen !");
}

```

Chú ý rằng trong hàm này, ta giả sử rằng cần phải xác định nút r trong xâu tại 1 vị trí cho trước position. Sau đó mới tiến hành chèn nút mới vào trước nút r.

3.2.2.5 Xóa một nút ở đầu danh sách

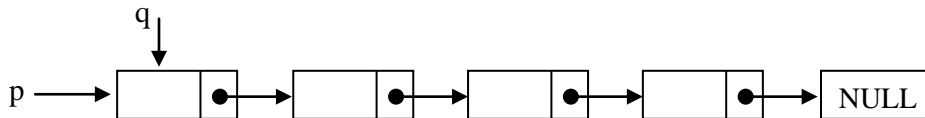
Giả sử ta có 1 danh sách mà đầu của danh sách được trỏ tới bởi con trỏ p.



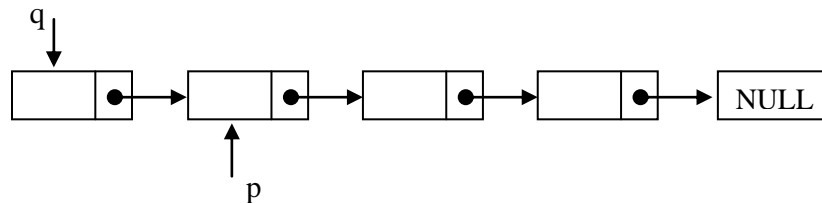
Chú ý rằng để xóa 1 nút trong danh sách thì danh sách đó không được rỗng.

Các bước để xóa 1 nút ở đầu danh sách như sau:

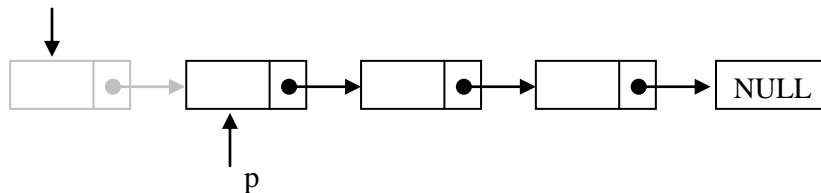
- Dùng 1 con trỏ tạm q trỏ đến đầu danh sách.



- Dịch chuyển con trỏ p qua phần tử đầu tiên đến phần tử kế tiếp.



- Ngắt liên kết của biến tạm q với nút tiếp theo, giải phóng bộ nhớ cho q.



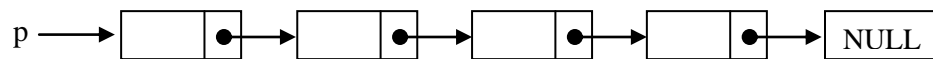
```

void Remove_Begin(listnode *p){
    listnode q;
    if (*p == NULL) return;
    q = *p;
    *p = (*p)-> next;
    q-> next = NULL;
    free(q);
}

```

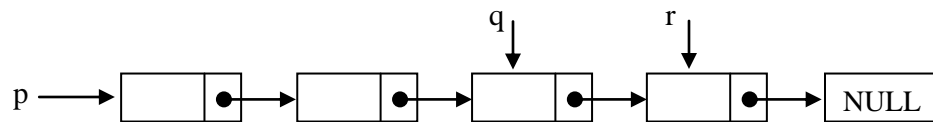
3.2.2.6 Xóa một nút ở cuối danh sách

Giả sử ta có 1 danh sách mà đầu của danh sách được trỏ tới bởi con trỏ p.

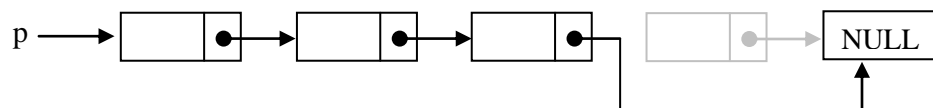


Các bước để xóa 1 nút ở cuối danh sách như sau:

- Dịch chuyển con trỏ tới nút gần nút cuối của danh sách. Để làm được việc này, ta phải dùng 2 biến tạm là q và r. Lần lượt dịch chuyển q và r từ đầu danh sách tới cuối danh sách, trong đó q luôn dịch chuyển sau r 1 nút. Khi r tới nút cuối cùng thì q là nút gần nút cuối cùng nhất.



- Cho con trỏ tiếp của nút gần nút cuối cùng nhất (đang được trỏ bởi q) trỏ tới null. Giải phóng bộ nhớ cho nút cuối cùng (đang được trỏ bởi r).



```

void Remove_End(listnode *p){
    listnode q, r;
    if (*p == NULL) return;
    if ((*p)-> next == NULL){
        Remove_Begin(*p);
        return;
    }
}

```

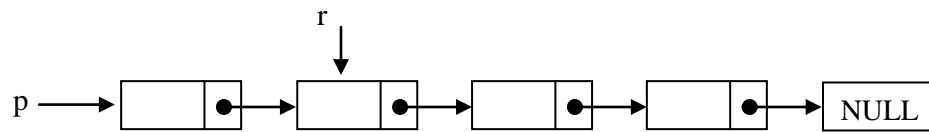
```

r = *p;
while (r-> next != NULL) {
    q = r;
    r = r-> next;
}
q-> next = NULL;
free(r);
}

```

3.2.2.7 Xóa một nút ở trước nút r trong danh sách

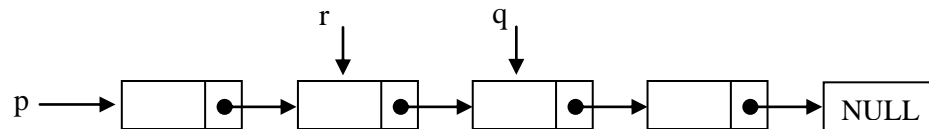
Giả sử ta có 1 danh sách mà đầu của danh sách được trỏ tới bởi con trỏ p, và 1 nút r trong danh sách.



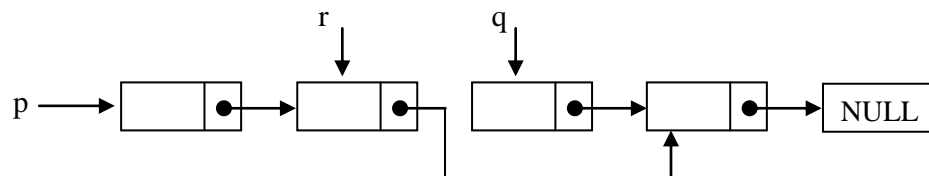
Ta giả thiết rằng nút r không phải là nút cuối cùng của danh sách, vì nếu như vậy thì sẽ không có nút đứng trước nút r.

Các bước để xóa 1 nút ở trước nút r trong danh sách như sau:

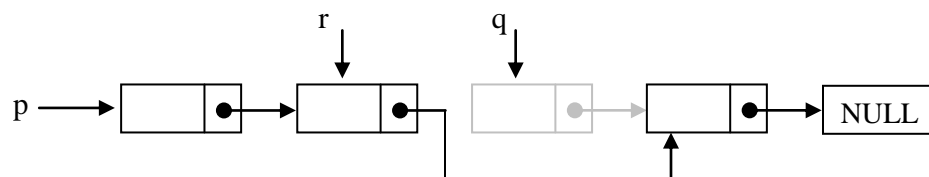
- Sử dụng 1 biến tạm q trỏ đến nút đứng trước nút r.



- Cho con trỏ tiếp của nút r trỏ tới nút đứng sau nút q.



- Ngắt liên kết của nút q và giải phóng bộ nhớ cho q.



```

void Remove_Middle(listnode *p, int position){
    int count=1, found=0;
    listnode q, r;
    r = *p;
    while ((r != NULL)&&(found==0)){
        if (count == position){
            q = r-> next;
            r-> next = q-> next;
            q-> next = NULL;
            free (q);
            found = 1;
        }
        count ++;
        r = r-> next;
    }
    if (found==0)
        printf("Khong tim thay vi tri can xoa !");
}

```

3.2.2.8 Duyệt toàn bộ danh sách

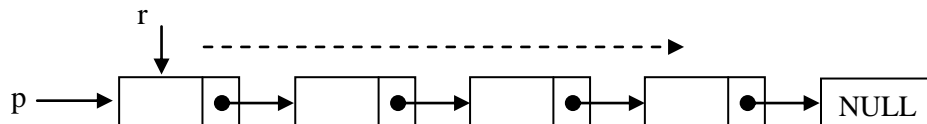
Thao tác duyệt danh sách cho phép duyệt qua toàn bộ các phần tử của danh sách, từ phần tử đầu tiên cho tới phần tử cuối cùng.

Để thực hiện thao tác này, ta cần một biến tạm r trỏ tới đầu danh sách. Từ vị trí này, theo liên kết của các nút, thực hiện duyệt qua từng phần tử trong danh sách. Trong quá trình duyệt, tại mỗi nút ta có thể thực hiện các thao tác cần thiết như lấy thông tin phần tử, sửa thông tin, so sánh, v.v.

```

r = p;
while (r-> next != null){
    //thực hiện các thao tác cần thiết
    r = r-> next;
}

```



3.2.2.9 Ví dụ về sử dụng danh sách liên kết

Trong phần này, chúng ta sẽ xem xét 1 ví dụ về việc sử dụng danh sách liên kết để biểu diễn 1 đa thức.

Giả sử ta cần biểu diễn 1 đa thức có dạng:

$$a_0 + a_1x^1 + a_2x^2 + \dots + a_nx^n$$

Trong đó a_i là hệ số của đa thức có kiểu số thực.

Ta có thể dùng mảng để lưu trữ đa thức này, tuy nhiên việc sử dụng mảng có một số nhược điểm sau:

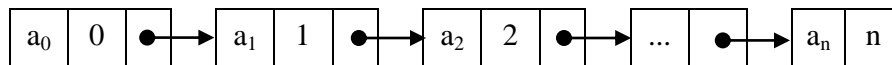
- Bậc của đa thức là chưa biết trước, do đó kích thước của mảng là chưa xác định. Ta phải khai báo một mảng với số phần tử tối đa nào đó.
- Đối với các hệ số $a_i = 0$ thì ta có thể bỏ qua số hạng a_ix^i trong biểu diễn đa thức. Tuy nhiên, nếu sử dụng mảng thì do phần tử a_i đã khai báo trong mảng nên ta vẫn phải dùng và gán cho nó giá trị 0.

Việc sử dụng danh sách liên kết để biểu thị đa thức sẽ khắc phục được các nhược điểm trên.

Để danh sách liên kết có thể biểu thị được đa thức, mỗi nút của danh sách cần có 3 thành phần:

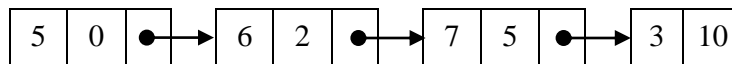
1. Trường heso kiểu thực lưu giữ giá trị của hệ số.
2. Trường mu lưu trữ giá trị l của lũy thừa x tại hệ số đó.
3. Trường con trỏ tiếp trở đến số hạng tiếp theo trong đa thức (nút tiếp theo trong danh sách).

Như vậy, danh sách này sẽ có dạng:



Trong đó, những số hạng có $a_i = 0$ thì không cần phải đưa vào danh sách.

Chẳng hạn, với đa thức $5 + 6x^2 + 7x^5 + 3x^{10}$ có thể được biểu diễn bởi danh sách liên kết như sau:



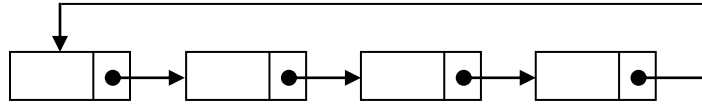
Khía báo trong C cho danh sách này như sau:

```
struct node{
    double heso;
    int luythua;
    struct node *next;
};
typedef struct node *listnode;
```

3.2.3 Một số dạng khác của danh sách liên kết

3.2.3.1 Danh sách liên kết vòng

Trong danh sách liên kết đơn, nút cuối cùng của danh sách sẽ có liên kết trở đến một giá trị null cho biết danh sách đã kết thúc. Nếu liên kết này không trở đến null mà trở về nút đầu tiên thì ta sẽ có một danh sách liên kết vòng.



Hình 3.2 Danh sách liên kết vòng

Ưu điểm của danh sách liên kết vòng là bất kỳ nút nào cũng có thể coi là đầu của danh sách. Có nghĩa là từ một nút bất kỳ, ta có thể tiến hành duyệt qua toàn bộ các phần tử của danh sách mà không cần trở về nút đầu tiên như trong danh sách liên kết thông thường.

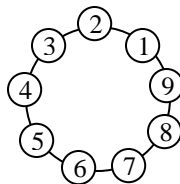
Tuy nhiên, nhược điểm của danh sách loại này là có thể không biết khi nào thì đã duyệt qua toàn bộ phần tử của danh sách. Điều này dẫn đến 1 quá trình duyệt vô hạn, không có điểm dừng. Để khắc phục nhược điểm này, trong quá trình duyệt luôn phải kiểm tra xem đã trở về nút ban đầu hay chưa. Việc kiểm tra này có thể dựa trên giá trị phần tử hoặc bằng cách thêm vào 1 nút đặc biệt.

Sau đây, chúng ta sẽ xem xét việc sử dụng danh sách liên kết vòng để giải quyết bài toán Josephus. Bài toán như sau:

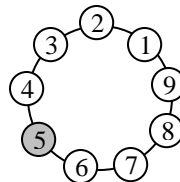
Có 1 nhóm N người muốn lựa chọn ra 1 thủ lĩnh. Các lựa chọn như sau: N người xếp thành vòng tròn. Bắt đầu từ 1 người nào đó, duyệt qua vòng và đến người thứ M thì người đó bị loại khỏi vòng. Quá trình duyệt bắt đầu lại, và người thứ M tiếp theo lại bị loại khỏi vòng. Lặp lại như vậy cho tới khi chỉ còn 1 người trong vòng và người đó là thủ lĩnh.

Chẳng hạn, với $N = 9$ người và $M = 5$, ta có quá trình duyệt và loại như sau:

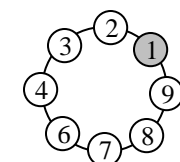
Vòng ban đầu:

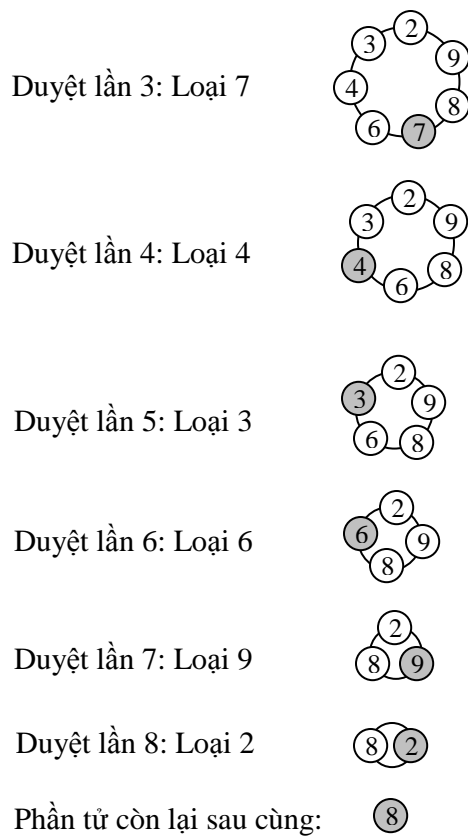


Duyệt lần 1: Loại 5



Duyệt lần 2: Loại 1

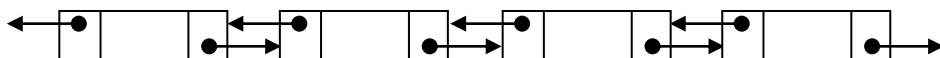




Việc sử dụng danh sách liên kết vòng có thể cung cấp 1 lời giải hiệu quả cho bài toán. Theo đó, N người sẽ lần lượt được đưa vào 1 danh sách liên kết vòng. Quá trình duyệt bắt đầu từ người đầu tiên, và đếm đến M lần duyệt thì loại nút ra khỏi danh sách và tiếp tục quá trình duyệt. Danh sách liên kết vòng sẽ thu hẹp dần, và đến khi chỉ còn 1 nút thì kết thúc quá trình duyệt. Rõ ràng là nếu sử dụng mảng cho bài toán Josephus sẽ không đem lại hiệu quả cao, vì quá trình loại bỏ 1 phần tử ra khỏi mảng phức tạp hơn nhiều so với danh sách liên kết.

3.2.3.2 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.



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;

```

3.3 TÓM TẮT CHƯƠNG 3

- Một mảng là 1 tập hợp cố định các thành phần có cùng 1 kiểu dữ liệu, được lưu trữ kế tiếp nhau và có thể được truy cập thông qua một chỉ số. Mảng có thể có một hoặc nhiều chiều.
- Mảng có ưu điểm là dễ sử dụng, tốc độ truy cập cao. Tuy nhiên, mảng có nhược điểm là không linh hoạt về kích thước và phức tạp khi bố trí lại các phần tử.
- 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, xoá 1 phần tử khỏi đầu, cuối, giữa danh sách, duyệt qua toàn bộ danh sách.
- Ngoài danh sách liên kết đơn còn một số loại danh sách liên kết khác như danh sách vòng, danh sách liên kết kép .v.v

3.4 CÂU HỎI VÀ BÀI TẬP

1. Hãy nêu các ưu và nhược điểm của danh sách liên kết so với mảng.
2. Nêu các bước để thêm một nút vào đầu, giữa, và cuối danh sách liên kết đơn.
3. Nêu các bước để xoá một nút ở đầu, giữa, và cuối danh sách liên kết đơn.
4. Viết thủ tục để in ra tất cả các phần tử của 1 danh sách liên kết đơn.
5. Viết chương trình thực hiện việc sắp xếp 1 danh sách liên kết đơn bao gồm các phần tử là các số nguyên.
6. Viết chương trình cộng 2 đa thức được biểu diễn thông qua danh sách liên kết đơn như ví dụ ở phần 3.2.2.9.
7. Viết chương trình minh hoạ việc sử dụng danh sách liên kết đơn với các chức năng:
 - a. Khởi tạo danh sách
 - b. Thêm phần tử
 - c. Xoá phần tử
 - d. In danh sách

CHƯƠNG 4

NGĂN XẾP VÀ HÀNG ĐỢI

Chương 4 trình bày về hai cấu trúc dữ liệu rất gần gũi với các hoạt động trong thực tế, đó là ngăn xếp và hàng đợi.

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.

Để học tốt chương 4, 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 2 kiểu dữ liệu này. Tự đặt ra các bài toán ứng dụng thực tế để thực hiện.

4.1 NGĂN XẾP (STACK)

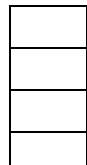
4.1.1 Khái niệm

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 kí tự. Ban đầu, ngăn xếp ở trạng thái rỗng:



Khi thực hiện lệnh bổ xung phần tử A, push(S, A), ngăn xếp có dạng:

A

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

	C
B	B
A	A

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

B
A

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

D
B
A

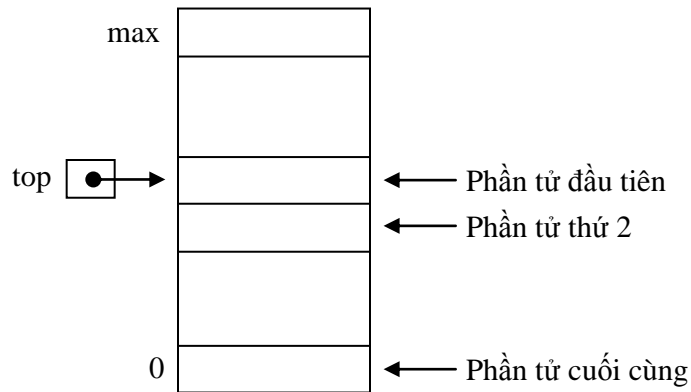
Hai lệnh $\text{pop}(S)$ tiếp theo sẽ lần lượt loại bỏ các phần tử nằm trên là D và B ra khỏi ngăn xếp:

B	
A	A

4.1.2 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(stack s){
    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

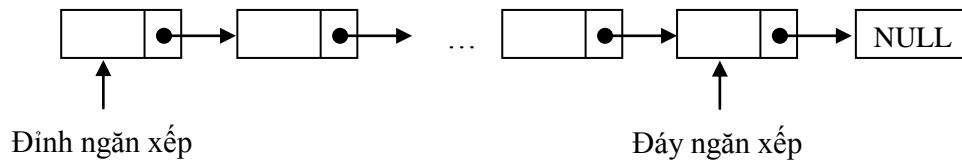
```
int Pop(stack *s){
    if (StackEmpty(*s)){
        printf("Ngan xep rong !");
    }else{
        return s-> nut[s-> top--];
    }
}
```

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.

4.1.3 Cài đặt ngăn xếp bằng danh sách liên kết

Để cài đặt ngăn xếp bằng danh sách liên kết, ta sử dụng 1 danh sách liên kết đơn. Theo tính chất của danh sách liên kết đơn, việc bổ sung và loại bỏ một phần tử khỏi danh sách được thực hiện đơn giản và nhanh nhất khi phần tử đó nằm ở đầu danh sách. Do vậy, ta sẽ chọn cách lưu trữ của ngăn xếp theo thứ tự: phần tử đầu danh sách là đỉnh ngăn xếp, và phần tử cuối cùng của danh sách là

đáy ngăn xếp. Để bổ sung 1 phần tử vào danh sách, ta tạo ra 1 nút mới và thêm nó vào đầu danh sách. Để lấy 1 phần tử khỏi ngăn xếp, ta chỉ cần lấy giá trị nút đầu tiên và loại nút ra khỏi danh sách.



Hình 4.2 Cài đặt ngăn xếp bằng danh sách liên kết

Như vậy, ta có thể thấy rằng ngăn xếp được cài đặt bằng danh sách liên kết có kích thước gần như “vô hạn” (tùy thuộc vào bộ nhớ của máy tính). Bất kỳ lúc nào ta cũng có thể thêm 1 nút mới và bổ sung vào đỉnh của ngăn xếp. Các thao tác push và pop đối với các danh sách kiểu này cũng khá đơn giản. Tuy nhiên, một số thao tác khác lại phức tạp hơn so với ngăn xếp kiểu mảng, chẳng hạn truy cập tới 1 phần tử ở giữa ngăn xếp, hoặc đếm số phần tử của ngăn xếp.

Khai báo 1 ngăn xếp bằng danh sách liên kết như sau:

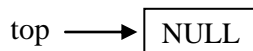
```
struct node {
    int item;
    struct node *next;
};
typedef struct node *stacknode;
typedef struct {
    stacknode top;
}stack;
```

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 null cho nút đầu ngăn xếp, cho biết ngăn xếp đang ở trạng thái rỗng.

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

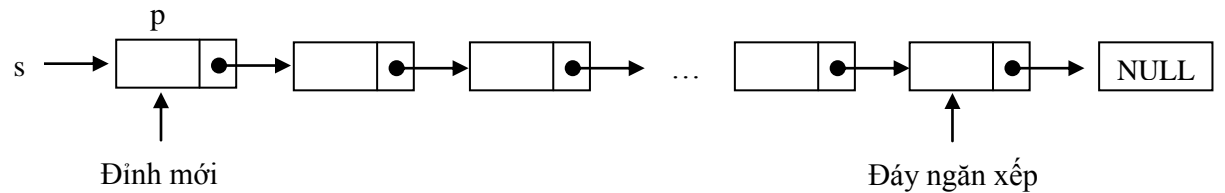


Thao tác kiểm tra ngăn xếp rỗng

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

top \longrightarrow NULL

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

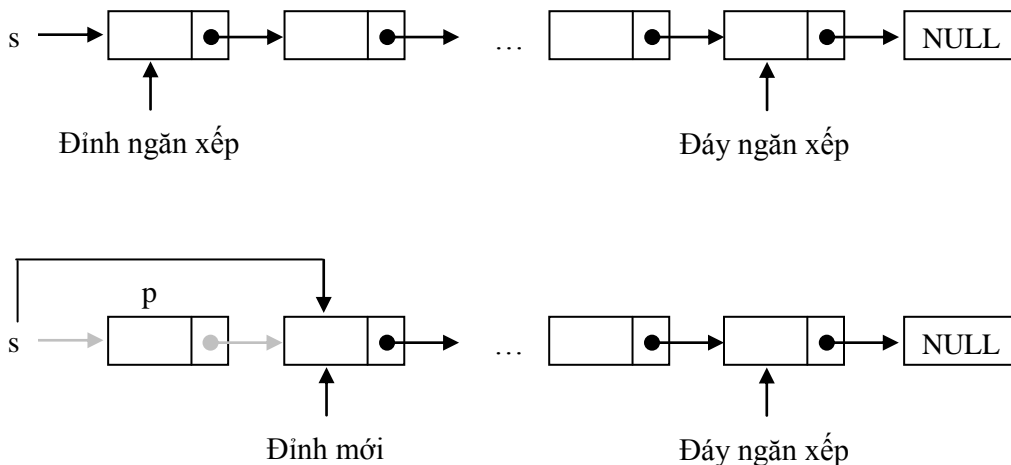


```
int Pop(stack *s){
    stacknode p;
    if (StackEmpty(*s)){
        printf("Ngan xep rong !");
    }else{
        p = s-> top;
```

```

        s-> top = s-> top-> next;
        return p->item;
    }
}

```



4.1.4 Một số 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 xâu sẽ được lấy ra đầu tiên, ..., và ký tự đầu tiên của xâu sẽ được lấy ra sau cùng. Như vậy, toàn bộ các ký tự trong xâu đã được đảo ngược thứ tự.

Mã chương trình đảo ngược xâu 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){
    stacknode p;
    p = (stacknode) malloc (sizeof(struct node));
    p-> item = c;
    p-> next = s->top;
    s->top = p;
    return;
}

char  Pop(stack *s){
    stacknode p;
    p = s-> top;
```

```

    s-> top = s-> top-> next;
    return p->item;
}

void main (void){
    char *st;
    int i;
    stack *s;
    clrscr();
    StackInitialize(s);
    printf("Nhập vào 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
3

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

1
7
3
3

Duyệt tiếp, gặp 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.

8
3
3

Duyệt tiếp, gặp 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.

24
3

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

6
24
3

Duyệt tiếp, gặp 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.

18
3

Duyệt tiếp gặp 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 3 dấu * ic, 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 2 -

Ngăn xếp:

*
*

Bước 9: Gặp dấu mở ngoặc, bỏ qua.

Biểu thức mới: 3 5 2 -

Ngăn xếp:

*
*

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

Biểu thức mới: 3 5 2 - 7

Ngăn xếp:

*
*

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

Biểu thức mới: 3 5 2 - 7

Ngăn xếp:

+
*
*

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

Biểu thức mới: 3 5 2 - 7 1

Ngăn xếp:

+
*
*

Bước 13: Gập 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 14: Gập 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: Gặp toán tử -, đưa vào ngăn xếp.

Biểu thức mới: 3 5 2 - 7 1 + *

Ngăn xếp:

-
*

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

Biểu thức mới: 3 5 2 - 7 1 + * 6

Ngăn xếp:

-
*

Bước 17: Gặp 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 - *

4.2 HÀNG ĐỢI (QUEUE)

4.2.1 Khái niệm

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.

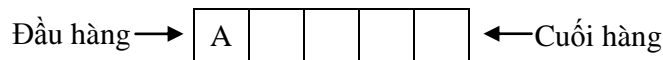
Như vậy, ta có thể định nghĩa hàng đợi là một dạng đặc biệt của danh sách mà việc lấy ra một phần tử, get, được thực hiện ở 1 đầu (gọi là đầu hàng), còn việc bổ sung 1 phần tử, put, được thực hiện ở đầu còn lại (gọi là cuối hàng).

Trở lại với ví dụ về việc bổ sung và loại bỏ các phần tử của 1 ngăn xếp các ký tự như ở phần trước, ta sẽ xem xét việc bổ sung và loại bỏ tương tự nhưng áp dụng cho hàng đợi các ký tự.

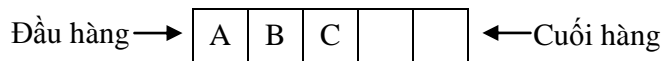
Giả sử ta có hàng đợi Q lưu trữ các ký tự. Ban đầu Q ở trạng thái rỗng:



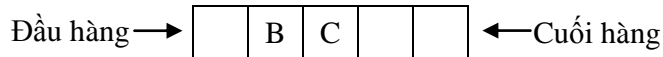
Khi thực hiện lệnh bổ sung phần tử A, put(Q, A), hàng đợi có dạng:



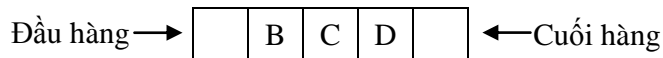
Tiếp theo là các lệnh put(Q, B), put(Q, C):



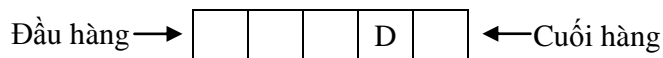
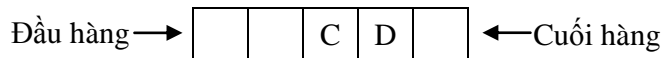
Khi thực hiện lệnh get để lấy ra 1 phần tử từ hàng đợi thì phần tử được lưu trữ lâu nhất trong hàng sẽ được lấy ra. Đó là phần tử đầu tiên ở đầu hàng.



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



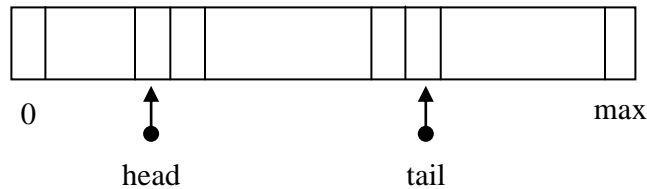
Hai lệnh get tiếp theo sẽ lần lượt lấy ra 2 phần tử ở đầu hàng là B và C.



4.2.2 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.

Khi đó, các thao tác trên hàng đợi được cài đặt như sau:

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--;
    }
}
```

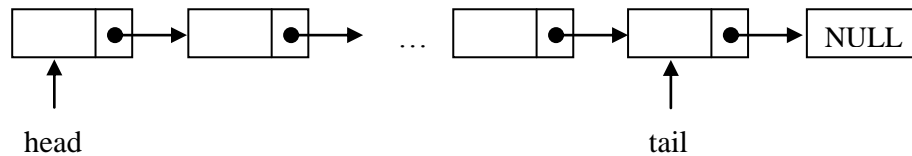
```

    }
    return x;
}

```

4.2.3 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. Khi đó, các thao tác trên hàng đợi được cài đặt như sau:

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){
    queuenode p;
    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;
    }
}

```

4.3 TÓM TẮT CHƯƠNG 4

- 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. Ngăn xếp còn được gọi là kiểu dữ liệu có nguyên tắc LIFO (Last In First Out - Vào sau ra trước).
- Ngăn xếp có thể được cài đặt bằng mảng hoặc danh sách liên kết.
- Các thao tác cơ bản trên ngăn xếp bao gồm: Khởi tạo ngăn xếp, kiểm tra ngăn xếp rỗng (đầy), thêm 1 phần tử vào ngăn xếp, loại bỏ 1 phần tử khỏi ngăn xếp.
- Hàng đợi là một cấu trúc dữ liệu gần giống với ngăn xếp, nhưng 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. Quy luật của hàng đợi là vào trước ra trước (FIFO - First In First Out).
- Hàng đợi cũng có thể được cài đặt bằng mảng hoặc danh sách liên kết. Các thao tác cơ bản cũng bao gồm: Khởi tạo hàng đợi, kiểm tra hàng đợi rỗng (đầy), thêm 1 phần tử vào hàng đợi, loại bỏ 1 phần tử khỏi hàng đợi.

4.4 CÂU HỎI VÀ BÀI TẬP

1. Để cài đặt ngăn xếp bằng mảng 1 chiều, ta cần bố trí ngăn xếp trong mảng như thế nào? Cần dùng thêm các biến phụ nào?
2. Hạn chế của cài đặt ngăn xếp bằng mảng so với danh sách liên kết là gì?
3. Để cài đặt ngăn xếp bằng danh sách liên kết cần bố trí danh sách như thế nào?
4. Hoàn thiện mã nguồn của chương trình tính biểu thức dạng hậu tố.
5. Hoàn thiện mã nguồn chương trình chuyển đổi biểu thức dạng trung tố sang hậu tố.
6. Viết chương trình đổi 1 số nguyên từ hệ thập phân sang nhị phân sử dụng ngăn xếp.
7. Sự khác biệt cơ bản giữa hàng đợi và ngăn xếp là gì?
8. Hoàn thiện mã nguồn của chương trình cài đặt ngăn xếp bằng mảng và danh sách liên kết bao gồm khai báo và các thao tác như hướng dẫn trong tài liệu.
9. Hoàn thiện mã nguồn của chương trình cài đặt hàng đợi bằng mảng và danh sách liên kết bao gồm khai báo và các thao tác như hướng dẫn trong tài liệu.

CHƯƠNG 5

CẤU TRÚC DỮ LIỆU KIỂU CÂY

Chương 5 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.

Ngoài ra, chương này còn giới thiệu một loại cây đặc biệt, có nhiều ứng dụng trong thực tiễn và nghiên cứu khoa học, đó là cây nhị phân. Loại cây đặc biệt hơn nữa là cây nhị phân tìm kiếm sẽ được giới thiệu trong chương 7.

5.1 KHÁI NIỆM

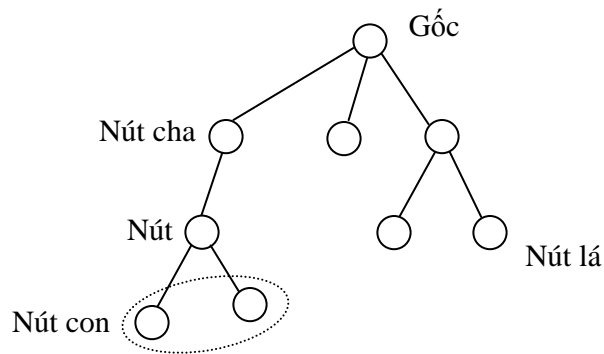
Cây là một cấu trúc dữ liệu có vai trò quan trọng trong việc phân tích và thiết kế các thuật toán. Lưu trữ và biểu diễn dữ liệu kiểu cây có thể thấy trong nhiều lĩnh vực của cuộc sống. Ví dụ một cuốn gia phả lưu trữ thông tin về các thành viên trong một dòng họ, trong đó các thành viên thức bậc khác nhau được phân thành các cấp khác nhau trong biểu diễn hình cây của gia phả. Sơ đồ tổ chức của 1 đơn vị cũng thường được biểu diễn thông qua cấu trúc cây. Các đơn vị con nằm ở cấp dưới đơn vị trực tiếp quản lý. Các đơn vị ngang hàng nằm cùng cấp. Trong lĩnh vực máy tính, cách lưu trữ dữ liệu của hệ điều hành cũng áp dụng kiểu lưu trữ cây. Các tệp tin được lưu trữ trong các cây thư mục, trong đó các thư mục con nằm trong các thư mục cha.

Cây có thể được định nghĩa như sau:

Cây là một tập hợp các nút (các đỉnh) và các cạnh, thỏa mãn một số yêu cầu nào đó. Mỗi nút của cây đều có 1 đỉnh danh và có thể mang thông tin nào đó. Các cạnh dùng để liên kết các nút với nhau. Một đường đi trong cây là một danh sách các đỉnh phân biệt mà đỉnh trước có liên kết với đỉnh sau.

Một tính chất rất quan trọng hình thành nên cây, đó là có đúng một đường đi nối 2 nút bất kỳ trong cây. Nếu tồn tại 2 nút trong cây mà có ít hoặc nhiều hơn 1 đường đi thì ta có một đồ thị (sẽ xem xét ở chương sau).

Mỗi cây thường có một nút được gọi là nút gốc. Mỗi nút đều có thể coi là nút gốc của cây con bao gồm chính nó và các nút bên dưới nó. Trong biểu diễn hình học của cây, nút gốc thường nằm ở vị trí cao nhất, tiếp theo là các nút kế tiếp.



Hình 5.1 Cây

Như vậy ta có thể thấy rằng cây bao gồm gốc và các cây con nối với gốc. Qua đó, ta có thể định nghĩa cây dưới dạng đệ qui như sau. Cây có thể là:

- Một nút đứng riêng lẻ (và nó chính là gốc của cây này).
- Hoặc một nút kết hợp với một số cây con bên dưới.

Mỗi nút trong cây (trừ nút gốc) có đúng 1 nút nằm trên nó, gọi là nút cha (parent). Các nút nằm ngay dưới nút đó được gọi là các nút con (subnode). Các nút nằm cùng cấp được gọi là các nút anh em (sibling). Nút không có nút con nào được gọi là nút lá (leaf) hoặc nút tận cùng.

Chiều cao của nút là đường đi dài nhất từ nút tới một lá. Chiều cao của cây chính là chiều cao của nút gốc. Độ sâu của 1 nút là độ dài đường đi duy nhất giữa nút gốc và nút đó.

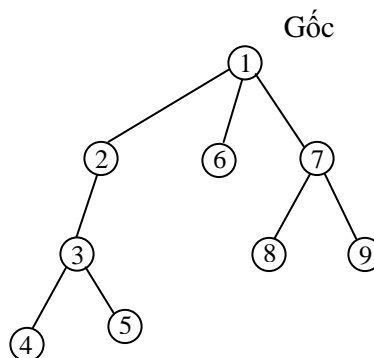
Một cây được gọi là có thứ tự nếu các nút con của 1 nút được bố trí theo thứ tự nào đó. Ngược lại gọi là cây không có thứ tự.

5.2 CÀI ĐẶT CÂY

5.2.1 Cài đặt cây bằng mảng các nút cha

Giả sử ta cần cài đặt 1 cây có n nút là các nút 1, 2, ..., n . Khi đó để biểu diễn cây bằng mảng, ta sử dụng một mảng A để lưu trữ các nút cha của các nút trong cây: $A[i] = j$ nếu j là nút cha của nút i . Nếu i là nút gốc thì ta gán giá trị $A[i] = 0$.

Cây được biểu diễn theo cách này dựa trên tính chất: Mỗi nút trong cây chỉ có duy nhất 1 nút cha. Để tìm đường đi từ 1 nút lên gốc, ta tìm nút cha của nút đó, rồi tìm nút cha của nút vừa tìm được, v.v. cho tới khi lên đến nút gốc. Hình 5.2 cho thấy biểu diễn bằng mảng của 1 cây.



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

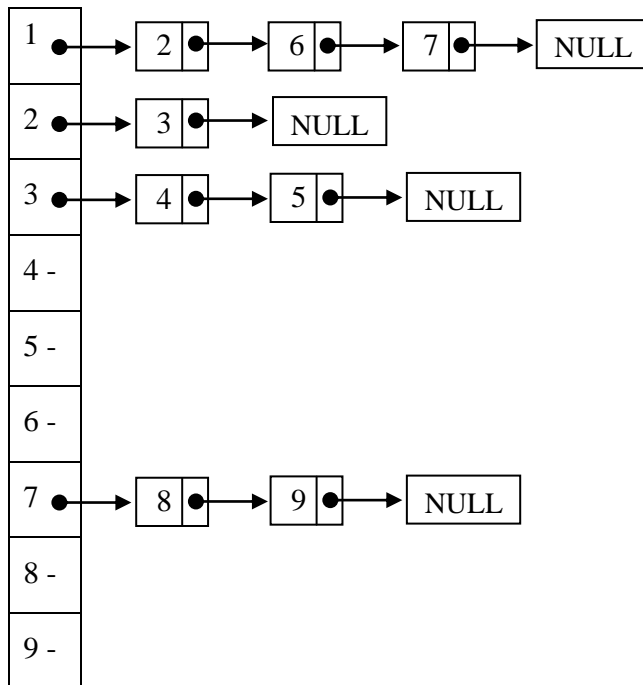
Hình 5.2 Biểu diễn cây bằng mảng các nút cha

Với phương pháp biểu diễn này, ta có thể dễ dàng tìm nút cha của 1 nút trên cây, nhưng nhược điểm là việc tìm nút con của 1 nút khá phức tạp, đặc biệt là tìm tất cả các nút con của một nút sẽ tốn rất nhiều công sức. Ngoài ra, với cách biểu diễn này, ta cũng không ấn định được thứ tự của các nút con.

5.2.2 Cài đặt cây thông qua danh sách các nút con

Cây có thể được cài đặt một cách hiệu quả hơn bằng cách tạo ra 1 danh sách các nút con cho mỗi nút của cây. Danh sách các nút con này có thể sử dụng bất kỳ loại danh sách nào như đã trình bày ở chương 3. Tuy nhiên, do số nút con của 1 nút là không xác định trước, do vậy nên dùng danh sách liên kết để biểu thị danh sách các nút con.

Quay trở lại với cây ở phần trước, biểu diễn cây theo danh sách các nút con như sau:



Hình 5.3 Cài đặt cây bằng danh sách các nút con

Rõ ràng biểu diễn cây theo phương pháp này cho phép duyệt cây dễ dàng và hợp logic hơn. Xuất phát từ gốc, ta tìm các nút con của gốc, rồi tìm các nút con của các nút vừa tìm được, v.v. cho tới khi đến các nút lá.

Khai báo cho cây theo theo phương pháp này trong C như sau:

```

#define max = 100;
struct node {
    int item;
    struct node *next;
};
typedef struct node *listnode;
typedef struct {
    int root;
    listnode subnode[max];
} tree;

```

Trong khai báo trên, thành phần `subnode[i]` là con trỏ trỏ đến danh sách các nút con của nút `i`.

5.3 DUYỆT CÂY

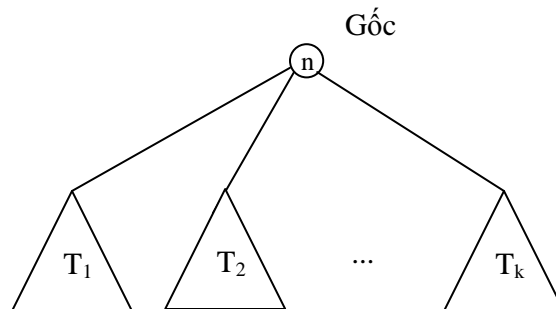
Duyệt cây là hành động duyệt qua tất cả các nút của một cây theo một trình tự nào đó. Trong quá trình duyệt, tại mỗi nút ta có thể tiến hành một thao tác xử lý nào đó. Đối với các danh sách liên kết, việc duyệt qua danh sách đơn giản là đi từ nút đầu, qua các liên kết và tới nút cuối cùng. Tuy nhiên, đối với cây, mỗi nút có thể có nhiều liên kết tới các nút con, vì vậy thứ tự duyệt qua các nút sẽ cho các phương pháp duyệt cây theo trình tự khác nhau.

Nhìn chung, có 3 trình tự duyệt cây phổ biến nhất, đó là:

- Duyệt cây theo thứ tự trước.
- Duyệt cây theo thứ tự giữa.
- Duyệt cây theo thứ tự sau.

5.3.1 Duyệt cây thứ tự trước

Giả sử ta có một cây T với gốc n và k cây con là T_1, T_2, \dots, T_k như hình vẽ.



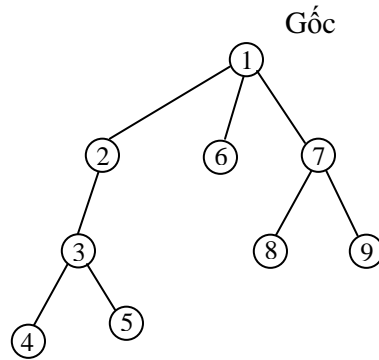
Hình 5.4 Duyệt cây thứ tự trước

Quá trình duyệt cây thứ tự trước được tiến hành theo trình tự như sau:

- Thăm nút gốc n .

- Thăm cây con T_1 theo phương pháp thứ tự trước.
- Thăm cây con T_2 theo phương pháp thứ tự trước.
- ...
- Thăm cây con T_k theo phương pháp thứ tự trước.

Chẳng hạn với cây như ở phần trước, trình tự thăm cây theo thứ tự trước như sau:



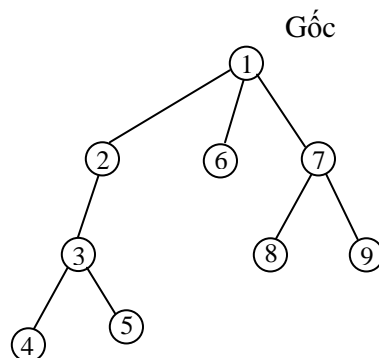
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9

5.3.2 Duyệt cây thứ tự giữa

Quá trình duyệt cây thứ tự giữa được tiến hành theo trình tự như sau:

- Thăm cây con T_1 theo phương pháp thứ tự giữa.
- Thăm nút gốc n.
- Thăm cây con T_2 theo phương pháp thứ tự giữa.
- ...
- Thăm cây con T_k theo phương pháp thứ tự giữa.

Với cây như ở phần trước, trình tự thăm cây theo thứ tự giữa như sau:



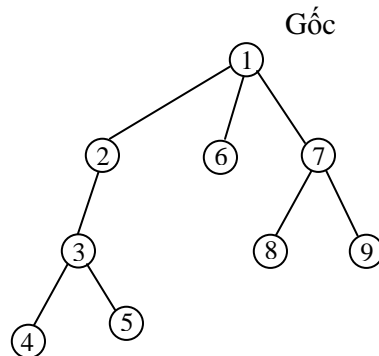
4 -> 3 -> 5 -> 2 -> 1 -> 6 -> 8 -> 7 -> 9

5.3.3 Duyệt cây thứ tự sau

Quá trình duyệt cây thứ tự sau được tiến hành theo trình tự như sau:

- Thăm cây con T_1 theo phương pháp thứ tự sau.
- Thăm cây con T_2 theo phương pháp thứ tự sau.
- ...
- Thăm cây con T_k theo phương pháp thứ tự sau.
- Thăm nút gốc n.

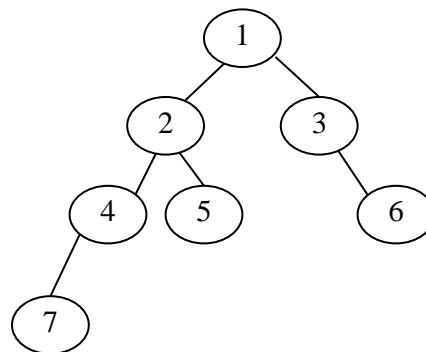
Với cây như ở phần trước, trình tự thăm cây theo thứ tự sau như sau:



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

5.4 CÂY NHỊ PHÂN

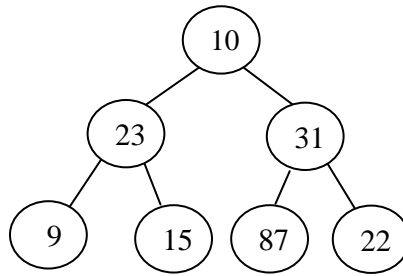
Cây nhị phân là một loại cây đặc biệt mà mỗi nút của nó chỉ có nhiều nhất là 2 nút con. Khi đó, 2 cây con của mỗi nút được gọi là cây con trái và cây con phải.



Hình 5.5 Cây nhị phân

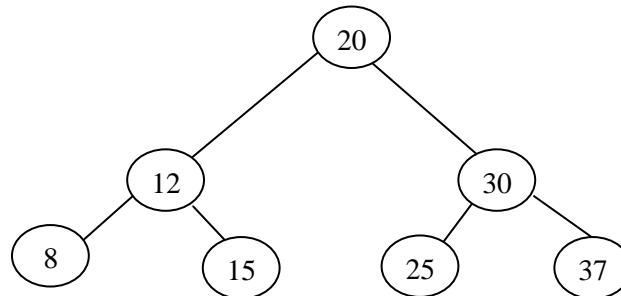
Cây nhị phân là loại cây có cấu trúc đơn giản và có nhiều ứng dụng trong tin học. Một số dạng cây nhị phân đặc biệt và được ứng dụng nhiều nhất là:

- 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

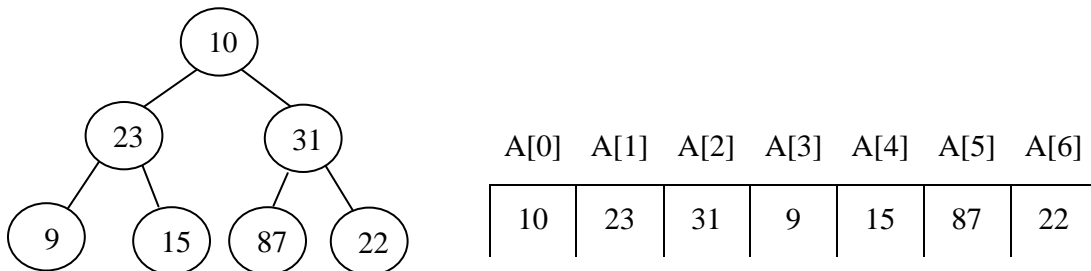
5.4.1 Cài đặt cây nhị phân bằng mảng

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 $\lfloor j/2 \rfloor$.

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];
```

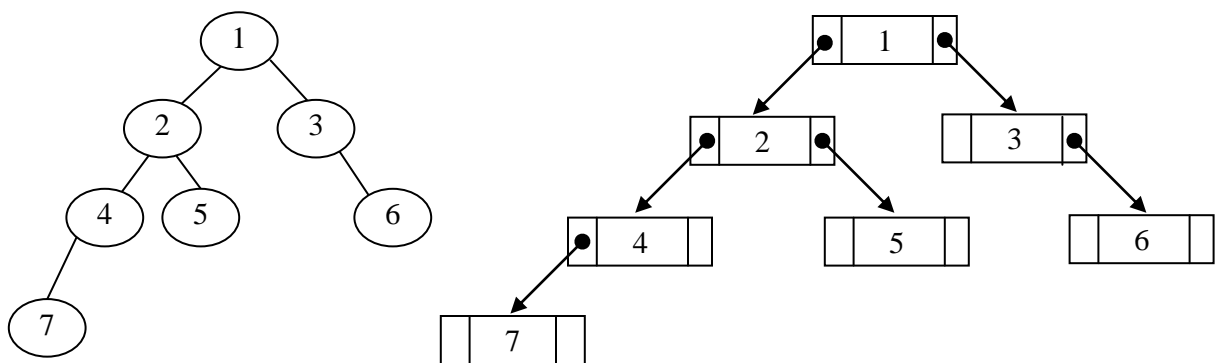
5.4.2 Cài đặt cây nhị phân bằng danh sách liên kết

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;
```

5.4.3 Duyệt cây nhị phân

Phép duyệt cây nhị phân cũng được chia làm 3 kiểu: duyệt thứ tự trước, duyệt thứ tự sau, và duyệt thứ tự cuối.

Duyệt thứ tự trước

```
void PreOrder (treenode root ) {
    if (root !=NULL) {
        printf("%d", root.item);
        PreOrder(root.left);
        PreOrder(root.right);
    }
}
```

Duyệt thứ tự giữa

```
void InOrder (treenode root ) {
    if (root !=NULL) {
        PreOrder(root.left);
        printf("%d", root.item);
        PreOrder(root.right);
    }
}
```

Duyệt thứ tự sau

```
void PostOrder (treenode root ) {
    if (root !=NULL) {
        PreOrder(root.left);
        PreOrder(root.right);
        printf("%d", root.item);
    }
}
```

5.5 TÓM TẮT CHƯƠNG 5

- Định nghĩa đệ qui của cây:

Cây có thể là:

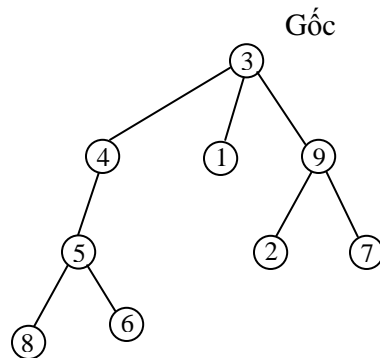
- (1) Một nút đứng riêng lẻ (và nó chính là gốc của cây này).
 - (2) Hoặc một nút kết hợp với một số con bên dưới.
- Mỗi nút trong cây (trừ nút gốc) có đúng 1 nút nằm trên nó, gọi là nút cha (parent). Các nút nằm ngay dưới nút đó được gọi là các nút con (subnode). Các nút nằm cùng cấp được gọi

là các nút anh em (sibling). Nút không có nút con nào được gọi là nút lá (leaf) hoặc nút tận cùng.

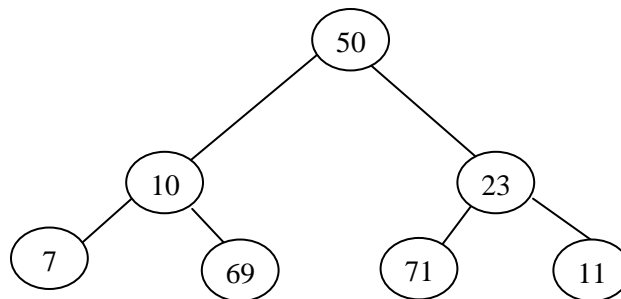
- Chiều cao của nút là đường đi dài nhất từ nút tới một lá. Chiều cao của cây chính là chiều cao của nút gốc. Độ sâu của 1 nút là độ dài đường đi duy nhất giữa nút gốc và nút đó.
- Cây có thể được cài đặt bằng mảng các nút cha hoặc thông qua danh sách các nút con.
- Duyệt cây là thao tác thăm tất cả các nút của cây, mỗi nút đúng 1 lần.
- Duyệt cây có thể theo 3 phương pháp: Duyệt thứ tự đầu, thứ tự giữa, và thứ tự cuối.
- 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.

5.6 CÂU HỎI VÀ BÀI TẬP

1. Nêu khái niệm cây và một số tính chất của cây.
2. Với cây như ở hình bên dưới, nếu cài đặt cây bằng mảng các nút cha thì giá trị của mảng sẽ như thế nào?



3. Cũng với cây ở trên, hãy cho biết biểu diễn cây theo phương pháp dùng danh sách các nút con.
4. Cho biết trình tự thăm các nút của cây ở trên khi tiến hành duyệt theo thứ tự trước, thứ tự giữa, và thứ tự sau.
5. Với cây nhị phân bên dưới, hãy biểu thị cây theo phương pháp dùng mảng và danh sách liên kết.



CHƯƠNG 6

ĐỒ THỊ

Chương 6 giới thiệu các khái niệm cơ bản về đồ thị, đồ thị có hướng, đồ thị vô hướng, đồ thị có trọng số.

Hai phương pháp biểu diễn đồ thị thông dụng nhất cũng được trình bày trong chương, đó là biểu diễn đồ thị bằng ma trận kề và danh sách kề.

Đối với thao tác duyệt đồ thị, hai phương pháp duyệt được xem xét là duyệt theo chiều rộng và duyệt theo chiều sâu.

Để học tốt chương này, ngoài việc nắm vững các thuật toán, sinh viên cần tự đặt ra cho mình các đồ thị cụ thể và thực hiện các bước thuật toán trên các đồ thị này.

6.1 CÁC KHÁI NIỆM CƠ BẢN

6.1.1 Đồ thị có hướng

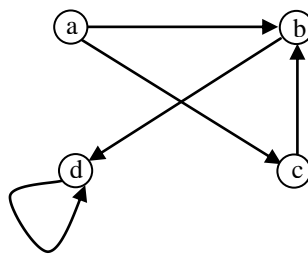
Đồ thị có hướng $G = \langle V, E \rangle$ bao gồm:

- V là một tập hữu hạn các đỉnh.
- E là một tập hữu hạn, có thứ tự các cặp đỉnh của V , gọi là các cạnh.

Ví dụ, đồ thị có hướng $G_1 = \langle V_1, E_1 \rangle$, với V_1 và E_1 được xác định như sau:

- $V_1 = \{a, b, c, d\}$
- $E_1 = \{(a, b); (a, c); (b, d); (c, b), (d, d)\}$

Khi đó, biểu diễn hình học của đồ thị này như sau:



Hình 6.1 Đồ thị có hướng

Chú ý rằng, trong đồ thị có hướng, cạnh là 1 cặp **có thứ tự** các đỉnh, vì vậy cạnh (a, c) và (c, a) là khác nhau. Ngoài ra, một đỉnh cũng có thể nối tới chính nó để tạo thành 1 cạnh.

Mỗi thành phần thuộc V được gọi là 1 đỉnh hoặc 1 nút của đồ thị, vì vậy V được gọi là tập các đỉnh của đồ thị. Mỗi thành phần thuộc E được gọi là 1 cạnh hoặc 1 cung, vì vậy E được gọi là tập các cạnh của đồ thị.

Một cạnh (u, v) của đồ thị có hướng có thể được biểu thị dạng $u \rightarrow v$. Đỉnh u khi đó được gọi là đỉnh kẻ của v . Cạnh (u, v) được gọi là cạnh xuất phát từ u . Ta ký hiệu $A(u)$ là tập các cạnh xuất phát từ u . Cạnh (u, v) cũng được gọi là cạnh đi tới v , và ta ký hiệu $I(v)$ là tập các cạnh đi tới v .

Bậc ngoài của 1 đỉnh là số các cạnh xuất phát từ đỉnh đó. Do đó, bậc ngoài của $u = |A(u)|$.

Bậc trong của 1 đỉnh là số các cạnh đi tới đỉnh đó. Do đó, bậc trong của $v = |I(v)|$.

Trong ví dụ trên, bậc ngoài của a là 2, bậc trong là 0. Bậc ngoài của b là 1, bậc trong là 2.

Định nghĩa về đường đi và độ dài đường đi, chu trình, đồ thị liên thông :

Một đường đi trong đồ thị có hướng $G(V, E)$ là một chuỗi các đỉnh

$$P = \{v_1, v_2, \dots, v_k\}$$

Trong đó, $v_i \in V$ ($i = 1..k$), và $(v_i, v_{i+1}) \in E$ ($i = 1..k-1$).

Độ dài của đường đi trong trường hợp này là $k - 1$.

Ví dụ, với đồ thị ở trên, ta có các đường đi:

$\{a, b, c\}$, $\{a, b\}$, $\{a, c\}$, $\{a, a\}$...

Chu trình là một đường đi mà đỉnh đầu và đỉnh cuối trùng nhau. Đồ thị liên thông là một đồ thị mà luôn tồn tại đường đi giữa 2 đỉnh bất kì.

6.1.2 Đồ thị vô hướng

Đồ thị vô hướng là đồ thị có các cạnh không có hướng. Hai nút ở hai đầu của cạnh có vai trò như nhau. Định nghĩa về đồ thị vô hướng như sau:

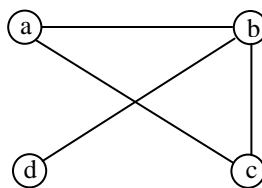
Đồ thị vô hướng $G = \langle V, E \rangle$ bao gồm:

- V là một tập hữu hạn các đỉnh.
- E là một tập hữu hạn các cặp đỉnh phân biệt của V , gọi là các cạnh.

Ví dụ, đồ thị có hướng $G_2 = \langle V_2, E_2 \rangle$, với V_2 và E_2 được xác định như sau:

- $V_2 = \{a, b, c, d\}$
- $E_2 = \{(a, b); (a, c); (b, d); (c, b)\}$

Khi đó, biểu diễn hình học của đồ thị này như sau:

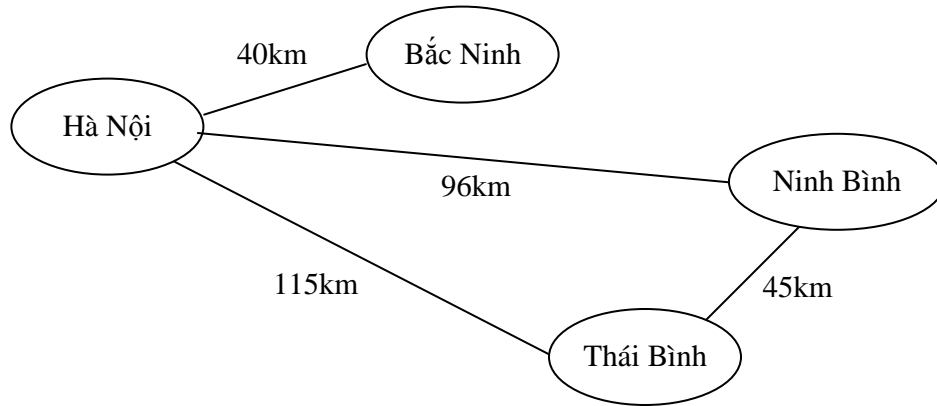


Hình 6.2 Đồ thị vô hướng

Chú ý rằng các cạnh của đồ thị là không có hướng, do vậy cạnh $(u, v) \sim$ cạnh (v, u) . Trong đồ thị vô hướng, cạnh (u, v) được coi là cạnh xuất phát và đồng thời là cạnh đi tới u hoặc v . Bậc của 1 đỉnh là tổng số cạnh xuất phát (cũng như đi tới) đỉnh đó.

6.1.3 Đồ thị có trọng số

Với các đồ thị như trình bày ở trên, mỗi cạnh của đồ thị chỉ biểu thị rằng có một liên kết nào đó từ đỉnh này tới đỉnh khác của đồ thị. Tuy nhiên, trong thực tế có rất nhiều ứng dụng của đồ thị cần thêm một số thông tin cho liên kết này. Chẳng hạn khoảng cách giữa 2 nút của đồ thị là 2 thành phố trên bản đồ, đồ thị biểu thị việc chuyển trạng thái của một loại máy dưới tác động của một số thao tác, .v.v.



Hình 6.3 Đồ thị có trọng số

6.2 BIỂU DIỄN ĐỒ THỊ

6.2.1 Biểu diễn đồ thị bằng ma trận kề

Giả sử ta có một đồ thị có hướng $G = \langle V, E \rangle$ bao gồm n đỉnh $\{v_1, v_2, \dots, v_n\}$. Phương pháp biểu diễn đồ thị bằng ma trận kề sử dụng 1 ma trận A ($n \times n$) được xác định như sau:

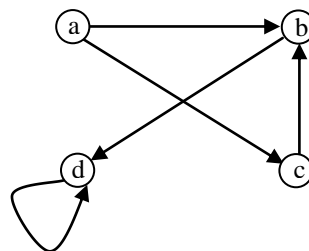
$$A_{i,j} = \begin{cases} 1 & \text{nếu } (v_i, v_j) \in E \\ 0 & \text{nếu } (v_i, v_j) \notin E \end{cases}$$

Có nghĩa là phần tử ở hàng i , cột j của ma trận A có giá trị 1 khi có một cạnh nối từ v_i đến v_j . Ngược lại, phần tử đó có giá trị 0.

Ma trận trên được gọi là ma trận kề của đồ thị có hướng (V, E) .

Ví dụ, với đồ thị có hướng (V, E) như trong hình vẽ dưới, ma trận kề của nó là:

$$A_1 = \begin{vmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



$$v_0 = a, v_1 = b$$

$$v_2 = c, v_3 = d$$

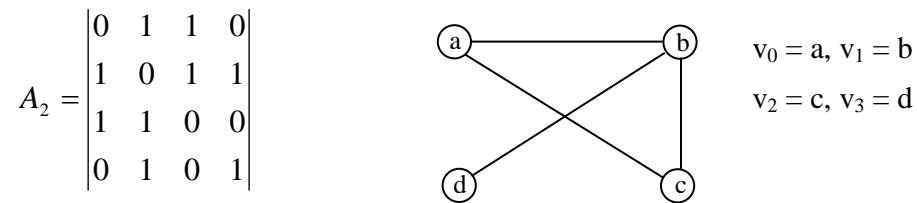
Hình 6.4 Biểu diễn đồ thị có hướng bằng ma trận kề

Rõ ràng là số phần tử có giá trị 1 của ma trận đúng bằng với số cạnh của đồ thị.

Một trong những ưu điểm nổi bật của ma trận kề là dựa vào ma trận này, ta dễ dàng xác định được các cạnh đi tới hoặc xuất phát từ 1 đỉnh cho trước. Ví dụ xét đỉnh v_i , mỗi phần tử có giá trị 1 trong hàng i của ma trận tương ứng với 1 cạnh xuất phát từ đỉnh v_i . Tương tự như vậy, mỗi phần tử có giá trị 1 ở cột i tương ứng với 1 cạnh đi tới đỉnh v_i .

Ma trận kề cũng có thể sử dụng để biểu diễn đồ thị vô hướng theo quy tắc trên. Chú ý rằng trong đồ thị vô hướng thì cạnh (u, v) và (v, u) là một nên ma trận kề của đồ thị vô hướng là ma trận đối xứng qua đường chéo, tức là $A_{i,j} = A_{j,i}$.

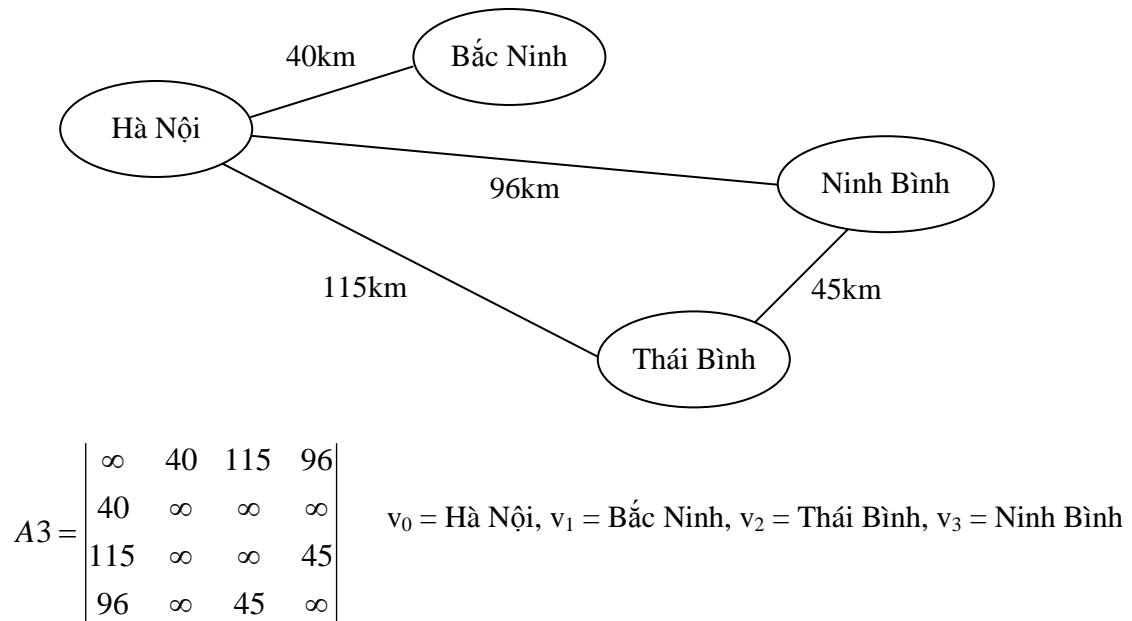
Ví dụ, với đồ thị vô hướng (V, E) như trong hình vẽ dưới, ma trận kề của nó là:



Hình 6.5 Biểu diễn đồ thị vô hướng bằng ma trận kề

Để biểu diễn đồ thị có trọng số bằng ma trận kề, ta thay các phần tử có giá trị 1 trong ma trận bằng chính trọng số của cạnh tương ứng, và với các phần tử có giá trị 0, ta thay bằng 1 giá trị ∞ cho biết không có cạnh nối 2 đỉnh tương ứng.

Chẳng hạn, với đồ thị có trọng số ở ví dụ trước, ta có thể biểu diễn bằng ma trận kề như sau:



Hình 6.6 Biểu diễn đồ thị có trọng số bằng ma trận kề

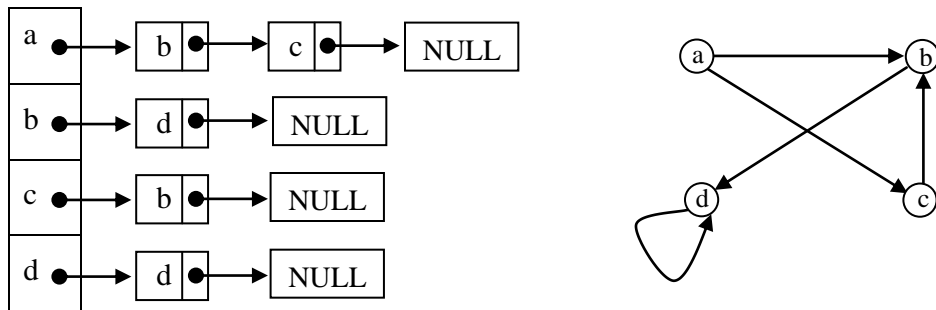
Ưu điểm của phương pháp biểu diễn đồ thị bằng ma trận kề là ta có thể dễ dàng biết được có một cạnh nối 2 đỉnh v_i, v_j hay không và có trọng số bao nhiêu. Tuy nhiên, phương pháp này có nhược điểm là kích thước ma trận kề luôn luôn là $n \times n$ bất kể đồ thị có bao nhiêu cạnh.

6.2.2 Biểu diễn đồ thị bằng danh sách kề

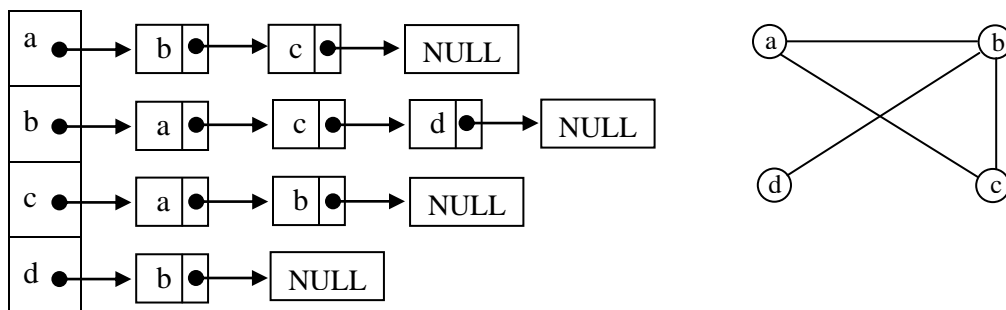
Rõ ràng là đối với các đồ thị có số cạnh ít thì việc sử dụng ma trận kề để biểu diễn đồ thị gây ra lãng phí không gian nhớ. Trong trường hợp này, người ta thường sử dụng phương pháp danh sách kề để biểu diễn đồ thị.

Phương pháp này sử dụng một danh sách liên kết cho mỗi đỉnh của đồ thị. Danh sách liên kết của một đỉnh sẽ chứa các đỉnh khác kề với nó, do vậy các danh sách này được gọi là các danh sách kề.

Ví dụ, với đồ thị có hướng như ở hình 6.1, các danh sách kề của nó là:



Còn với đồ thị vô hướng như ở hình 6.2, các danh sách kề của nó là:



Hình 6.7 Biểu diễn đồ thị bằng danh sách kề

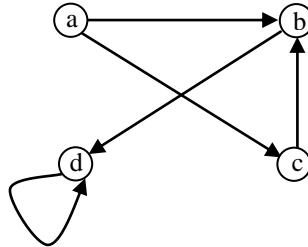
6.3 DUYỆT ĐỒ THỊ

Duyệt đồ thị là hành động thăm tất cả các đỉnh của đồ thị, mỗi đỉnh đúng 1 lần, theo một trình tự nào đó. Tương tự như duyệt cây, có nhiều phương pháp duyệt đồ thị tùy theo trình tự thăm các nút trong quá trình duyệt. Tuy nhiên, có 2 phương pháp duyệt phổ biến nhất là duyệt theo chiều sâu và duyệt theo chiều rộng.

6.3.1 Duyệt theo chiều sâu

Quá trình duyệt theo chiều sâu bắt đầu từ một đỉnh nào đó của đồ thị. Sau khi thăm đỉnh này, quá trình duyệt theo chiều sâu được lặp lại với tất cả các đỉnh kề của nó. Tuy nhiên, đồ thị có thể tồn tại các chu trình, do vậy, ta cần phải đánh dấu các đỉnh đã duyệt để tránh duyệt lại đỉnh này một lần nữa.

Với trình tự duyệt như trên, quá trình duyệt sẽ duyệt hết một “nhánh” của đồ thị rồi mới sang “nhánh” khác. Do vậy, phương pháp duyệt này được gọi là duyệt theo chiều sâu.



Hình 6.8 Duyệt đồ thị theo chiều sâu

Ví dụ, với đồ thị ở trên, quá trình duyệt theo chiều sâu bắt đầu từ đỉnh a sẽ cho thứ tự duyệt như sau:

- Sau khi thăm đỉnh a, tiến hành thăm đỉnh kề với a là b. Tiếp theo thăm đỉnh kề b là d. Đỉnh d không kề đỉnh nào, do vậy quay lại bước trước.
- Đỉnh b chỉ có 1 đỉnh kề là d đã thăm, do vậy quay trở lại bước trước.
- Đỉnh a còn đỉnh kề là c chưa thăm, do vậy tiến hành thăm đỉnh này.

Như vậy, thứ tự các đỉnh trong quá trình duyệt là: a, b, d, c

Quá trình duyệt sẽ chỉ duyệt theo các cạnh dẫn tới các đỉnh chưa thăm. Các cạnh dẫn tới các đỉnh thăm rồi sẽ được bỏ qua. Chẳng hạn, trong quá trình duyệt đồ thị trên, khi duyệt đến đỉnh c, cạnh nối tới b sẽ được bỏ qua vì đỉnh b đã được thăm rồi.

Cài đặt phương pháp duyệt theo chiều sâu như sau:

Để kiểm tra việc duyệt mỗi đỉnh đúng một lần, chúng ta sử dụng một mảng `daxet` gồm n phần tử (tương ứng với n đỉnh). Nếu đỉnh thứ i đã được duyệt, `daxet[i]=1`, ngược lại, `daxet[i]=0`. Thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh v nào đó sẽ duyệt tất cả các đỉnh liên thông với v . Thuật toán có thể được mô tả bằng thủ tục đệ quy `DeepFirstSearch`.

```

void DeepFirstSearch(int v) {
    Thăm đỉnh v;
    daxet[v] = 1;
    for mỗi đỉnh u kề với v {
        if (daxet[u]=0 )
            DeepFirstSearch(v);
    }
}

```

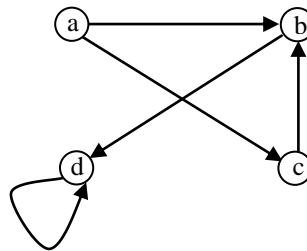
Thủ tục DeepFirstSearch sẽ thăm tất cả các đỉnh cùng thành phần liên thông với v mỗi đỉnh đúng một lần. Để đảm bảo duyệt tất cả các đỉnh của đồ thị (có thể có nhiều thành phần liên thông), chúng ta chỉ cần thực hiện :

```
for( i=1; i≤n; i++)
    daxet[i] = 0;
for( i:=1; i≤ n; i++)
    if (daxet[i]=0)
        DeepFirstSearch(i);
```

6.3.2 Duyệt theo chiều rộng

Quá trình duyệt theo chiều rộng cũng bắt đầu từ một đỉnh nào đó của đồ thị. Tiếp đến, các đỉnh kề của nó sẽ được thăm, rồi tiếp tục đến các đỉnh kề của các đỉnh vừa thăm .v.v.

Như vậy, quá trình duyệt theo chiều rộng không duyệt theo từng “nhánh” của đồ thị mà duyệt theo độ sâu của các đỉnh so với đỉnh ban đầu. Từ đỉnh bắt đầu, các đỉnh có khoảng cách với đỉnh ban đầu là 1 được duyệt, tiếp đến là các đỉnh có khoảng cách 2, v.v.



Hình 6.9 Duyệt đồ thị theo chiều rộng

Ví dụ, vẫn với đồ thị như ở phần trước, quá trình duyệt theo chiều rộng với đỉnh bắt đầu là a sẽ cho thứ tự duyệt như sau:

- Sau khi thăm đỉnh a, tiến hành thăm các đỉnh kề với a là b và c.
- Tiếp theo, thăm các đỉnh kề với b là d.
- Đỉnh kề với c là b đã được thăm rồi nên bỏ qua.

Như vậy, thứ tự các đỉnh được thăm là: a, b, c, d.

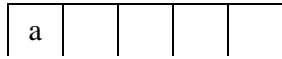
Duyệt theo chiều rộng có thể được cài đặt không đệ qui bằng cách sử dụng hàng đợi để lưu các đỉnh cần được thăm. Các bước như sau:

Đầu tiên, đưa đỉnh bắt đầu v vào hàng đợi, sau đó lặp lại quá trình sau cho đến khi hàng đợi không còn phần tử nào:

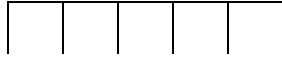
- Lấy phần tử ra khỏi hàng đợi, đưa vào biến v.
- Thăm đỉnh v.
- Với mỗi đỉnh kề với v, nếu đỉnh này chưa được thăm thì đưa vào hàng đợi.

Ví dụ, đối với đồ thị ở hình ... ở trên, các bước thực hiện như sau:

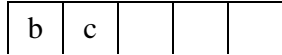
Đầu tiên, đưa đỉnh a vào hàng đợi.



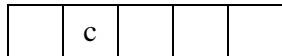
- Lấy đỉnh a ra khỏi hàng đợi, thăm đỉnh a.



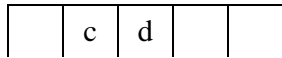
- Đưa 2 đỉnh kề với a là b và c vào hàng đợi.



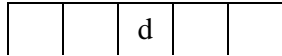
- Lấy đỉnh b ra khỏi hàng đợi, thăm đỉnh b



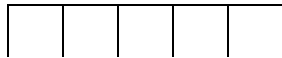
- Đưa đỉnh kề với b là d vào hàng đợi



- Lấy đỉnh c ra khỏi hàng đợi, thăm đỉnh c



- Đỉnh kề với c là b đã thăm, vì vậy không đưa vào hàng đợi. Lấy đỉnh d ra khỏi hàng đợi, thăm đỉnh d.



- Hàng đợi hết phần tử, quá trình duyệt kết thúc. Thứ tự thăm các đỉnh là: a, b, c, d

Cài đặt cho thuật toán duyệt theo chiều rộng như sau:

```
void BreadthFirstSearch(int v){
    queue =  $\phi$ ;
    Đưa v vào hàng đợi;
    daxet[v] = 1;
    while (queue  $\neq \phi$  ){
        Lấy phần tử ra khỏi hàng đợi, đưa vào biến u;
        Thăm đỉnh u;
        for mỗi đỉnh w kề với u {
            if (daxet[w]=0 ) {
                Đưa w vào hàng đợi;
                daxet[w] = 1;
            }
        }
    }
}
```

Tương tự như duyệt theo chiều sâu, thủ tục BreadthFirstSearch sẽ thăm tất cả các đỉnh cùng thành phần liên thông với v. Để thăm tất cả các đỉnh của đồ thị, chúng ta chỉ cần thực hiện:

```
for( v=1; v≤n; v++)
    daxet[v] = 0;
for(v=1; v≤n; v++)
    if (daxet[v]=0)
        BreadthFirstSearch(u);
```

6.3.3 Ứng dụng duyệt đồ thị để kiểm tra tính liên thông

Như đã nói ở trên, duyệt đồ thị (theo chiều rộng hay theo chiều sâu) sẽ thăm tất cả các đỉnh cũng thành phần liên thông với đỉnh bắt đầu duyệt. Vì vậy, ta có thể sử dụng thủ tục duyệt đồ thị để kiểm tra tính liên thông của đồ thị, hoặc thậm chí có thể đếm được số thành phần liên thông của đồ thị.

Để làm được điều này, ta thực hiện duyệt từ đầu đến cuối danh sách các đỉnh của đồ thị. Tại mỗi bước, ta kiểm tra nếu đỉnh chưa được thăm thì ta tiến hành gọi thủ tục duyệt đồ thị cho đỉnh này. Như vậy, nếu đồ thị liên thông hoàn toàn thì chỉ mất một lần gọi thủ tục duyệt cho đỉnh đầu tiên. Ngược lại, số lần gọi thủ tục duyệt chính là số thành phần liên thông của đồ thị.

```
int lt=0;
for( v=1; v≤n; v++)
    daxet[v] = 0;
for(v=1; v≤n; v++)
    if (daxet[v]=0){
        BreadthFirstSearch(u);
        lt++;
    }
if (lt ==1) printf("Do thi lien thong!");
else printf("Do thi khong lien thong, so thanh phan lien thong la %d", lt);
```

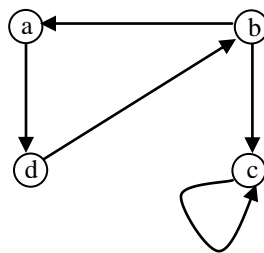
6.4 TÓM TẮT CHƯƠNG 6

- Khái niệm đồ thị có hướng:
Đồ thị có hướng $G = \langle V, E \rangle$ bao gồm:
 - (1) V là một tập hữu hạn các đỉnh.
 - (2) E là một tập hữu hạn, có thứ tự các cặp đỉnh của V, gọi là các cạnh.
- Khái niệm đồ thị vô hướng:
Đồ thị vô hướng $G = \langle V, E \rangle$ bao gồm:
 - (1) V là một tập hữu hạn các đỉnh.
 - (2) E là một tập hữu hạn các cặp đỉnh phân biệt của V, gọi là các cạnh.
- Đồ thị có thể được biểu diễn bằng ma trận kề hoặc danh sách kề.

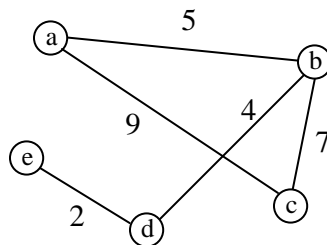
- Đồ thị biểu diễn bằng ma trận kề A có tính chất: Phần tử ở hàng i , cột j của ma trận A có giá trị 1 khi có một cạnh nối từ v_i đến v_j . Ngược lại, phần tử đó có giá trị 0.
- Biểu diễn đồ thị bằng danh sách kề: Sử dụng một danh sách liên kết cho mỗi đỉnh của đồ thị. Danh sách liên kết của một đỉnh sẽ chứa các đỉnh khác kề với nó
- Duyệt theo chiều sâu bắt đầu từ một đỉnh nào đó của đồ thị. Sau khi thăm đỉnh này, quá trình duyệt theo chiều sâu được lặp lại với tất cả các đỉnh kề của nó.
- Duyệt theo chiều rộng cũng bắt đầu từ một đỉnh nào đó của đồ thị. Tiếp đến, các đỉnh kề của nó sẽ được thăm, rồi tiếp tục đến các đỉnh kề của các đỉnh vừa thăm .v.v.

6.5 CÂU HỎI VÀ BÀI TẬP

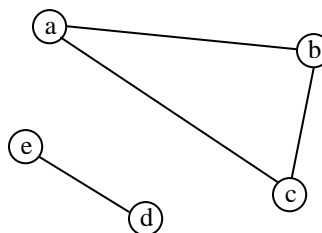
1. Cho biết biểu diễn bằng ma trận kề và danh sách kề của đồ thị bên dưới:



2. Cho biết ma trận kề của đồ thị trọng số sau:



3. Với đồ thị câu 1, cho biết trình tự thăm các đỉnh khi thực hiện duyệt theo chiều sâu bắt đầu từ đỉnh a.
4. Với đồ thị câu 2, cho biết trình tự thăm các đỉnh khi thực hiện duyệt theo chiều rộng bắt đầu từ đỉnh a.
5. Cho biết số thành phần liên thông của đồ thị bên dưới.



CHƯƠNG 7

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 7 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.

7.1 BÀI TOÁN SẮP XẾP

Sắp xếp là quá trình bố trí lại các phần tử của 1 tập hợp theo thứ tự nào đó. Mục đích chính của sắp xếp là làm cho thao tác tìm kiếm phần tử trên tập đó được dễ dàng hơn. Ví dụ về tập các đối tượng được sắp phổ biến trong thực tế là: danh bạ điện thoại được sắp theo tên, các từ trong từ điển được sắp theo vần, sách trong thư viện được sắp theo mã số, theo tên, .v.v.

Nhìn chung, có rất nhiều thao tác xử lý dữ liệu cần đến việc sắp xếp các phần tử dữ liệu theo trình tự nào đó. Trên thực tế, sắp xếp là một thao tác khá đơn giản. Tuy nhiên, như chúng ta sẽ thấy, có rất nhiều giải thuật sắp xếp khác nhau, từ đơn giản tới phức tạp. Và các kỹ thuật được sử dụng trong các giải thuật sắp xếp này được nghiên cứu và phân tích nhiều hơn là chính bản thân giải thuật sắp xếp. Các kỹ thuật này được coi là cơ sở để xây dựng nhiều giải thuật quan trọng khác. Do đó, các thuật toán sắp xếp được trình bày và phân tích kỹ trong hầu hết các tài liệu về giải thuật.

Các giải thuật sắp xếp còn là một ví dụ điển hình cho sự đa dạng của thuật toán. Cùng một mục đích, nhưng có rất nhiều cách thực hiện, mỗi cách tối ưu trên một khía cạnh nào đó, và có một số cách sắp xếp có nhiều ưu điểm hơn những cách khác. Do đó, sắp xếp cũng được sử dụng như một ví dụ điển hình trong việc phân tích thuật toán.

Thông thường, các giải thuật sắp xếp được chia làm 2 loại. Loại thứ nhất là các giải thuật được cài đặt đơn giản, nhưng không hiệu quả (phải sử dụng nhiều thao tác). Loại thứ hai là các giải thuật được cài đặt phức tạp, nhưng hiệu quả hơn về mặt tốc độ (dùng ít thao tác hơn). Đối với các tập dữ liệu ít phần tử, tốt nhất là nên lựa chọn loại thứ nhất. Đối với tập có nhiều phần tử, loại thứ hai sẽ mang lại hiệu quả hơn.

Các đối tượng dữ liệu cần sắp xếp thường có nhiều thuộc tính, và ta cần chọn một thuộc tính làm *khóa* để sắp xếp dữ liệu theo khóa này. Ví dụ, đối tượng về người thường có các thuộc tính cơ bản như họ tên, ngày sinh, giới tính, v.v., tuy nhiên họ tên thường được chọn làm khóa để sắp xếp.

Tham số để tính toán hiệu quả của giải thuật thường là thời gian thực hiện. Đối với các phương pháp sắp xếp đơn giản, thời gian thực hiện (số thao tác thực hiện) tỷ lệ với N^2 , trong đó N là số phần tử của tập. Các giải thuật sắp xếp phức tạp và tinh xảo hơn có thời gian thực hiện tỷ lệ với $N\log N$. Người ta chứng minh được rằng, không có giải thuật nào có thể có thời gian thực hiện nhỏ hơn $N\log N$. Ngoài thời gian thực hiện, dung lượng bộ nhớ bị chiếm cũng là một tham số để đánh giá tính hiệu quả của giải thuật.

Một vấn đề nữa cần phải chú ý khi thực hiện sắp xếp, đó là tính ổn định của giải thuật sắp xếp. Một giải thuật được gọi là ổn định nếu sau khi sắp xếp, nó giữ nguyên vị trí của các phần tử có cùng giá trị khóa. Chẳng hạn, với danh sách theo vần họ tên các sinh viên trong một lớp. Nếu ta tiến hành sắp danh sách này theo điểm, thì các sinh viên có cùng điểm vẫn được sắp theo vần họ tên. Hầu hết các giải thuật sắp xếp đơn giản có tính ổn định, trong khi các giải thuật tinh xảo hơn lại không có tính chất này.

7.2 CÁC GIẢI THUẬT SẮP XẾP ĐƠN GIẢN

7.2.1 Sắp xếp chọn

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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $O(N * N/2) = O(N^2/2) = O(N^2)$.

7.2.2 Sắp xếp chèn

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:

06	17	49	98
----	----	----	----

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:

06	17		49	98
----	----	--	----	----

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

06	17	32	49	98
----	----	----	----	----

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.

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

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.

32	17	49	98	06	25	53	61
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.

17	32	49	98	06	25	53	61
Đã sắp		Chưa sắp					

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

17	32	49	98	06	25	53	61
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.

06	17	32	49	98	25	53	61
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.

06	17	25	32	49	98	53	61
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.

06	17	25	32	49	53	98	61
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.

06	17	25	32	49	53	61	98
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 $O(N^2/4) = O(N^2)$.

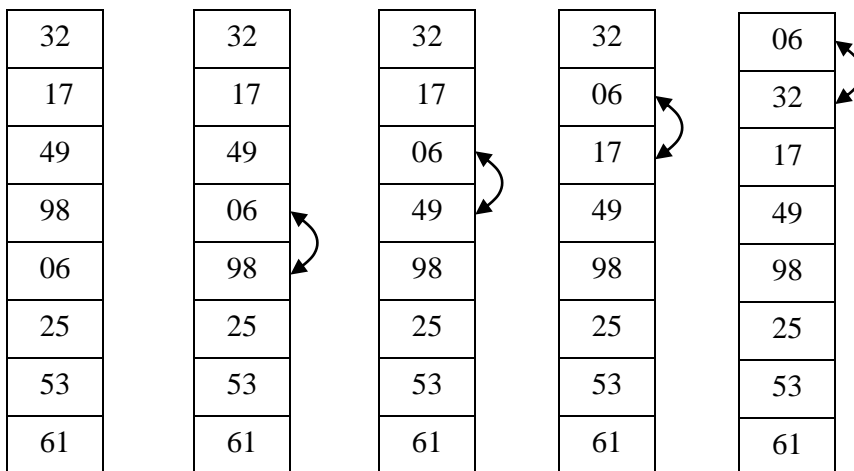
7.2.3 Sắp xếp nổi bọt

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 sắp.

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;
                no_exchange = 0;
            }
        }
        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 cũng giải thuật sắp xếp nổi bọt cũng là $O(N^2)$.

7.3 QUICK SORT

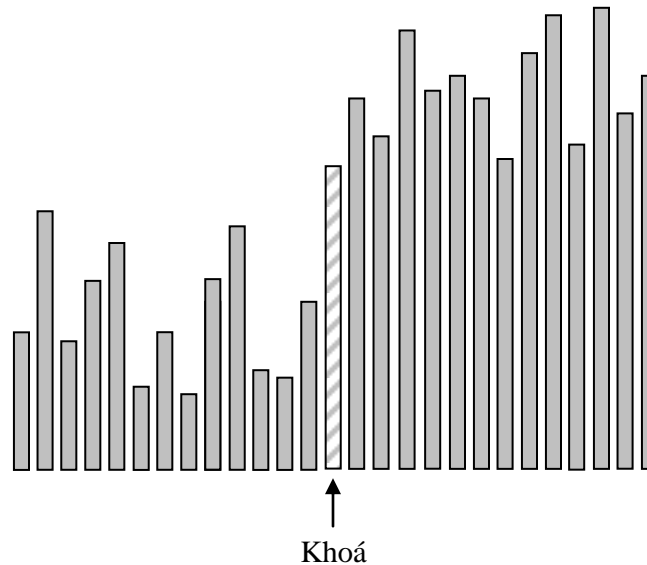
7.3.1 Giới thiệu

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.

7.3.2 Các bước thực hiện giải thuật

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.

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

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 cho nhau.

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

↑ ↑ ↑
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.

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

↑ ↑ ↑
Khoá i j

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

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

↑ ↑ ↑
Khoá j i

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.

7.3.3 Cài đặt giải thuật

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 quy, bao gồm việc chia phân đoạn thành 2 đoạn con thỏa mãn yêu cầu trên, sau đó thực hiện lời gọi đệ quy 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) {
```

```

int i,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 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:

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

$$\text{Khi đó, } T(N) \approx N \log N.$$

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.

7.4 HEAP SORT

7.4.1 Giới thiệu

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.

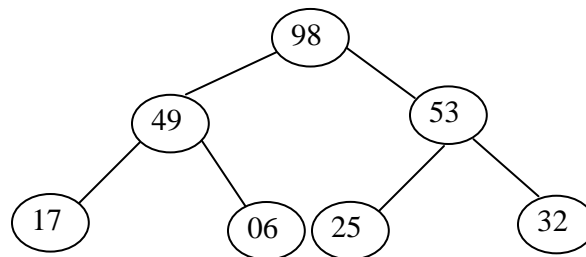
Việc thực hiện giải thuật này được chia làm 2 giai đoạn. Đầu tiên là việc tạo heap từ dãy ban đầu. Theo định nghĩa của heap thì nút cha bao giờ cũng lớn hơn các nút con. Do vậy, nút gốc của heap bao giờ cũng là phần tử lớn nhất.

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.

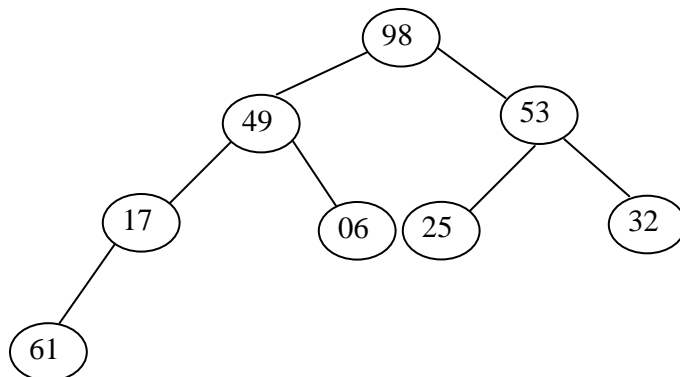
7.4.2 Các thuật toán trên heap

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.

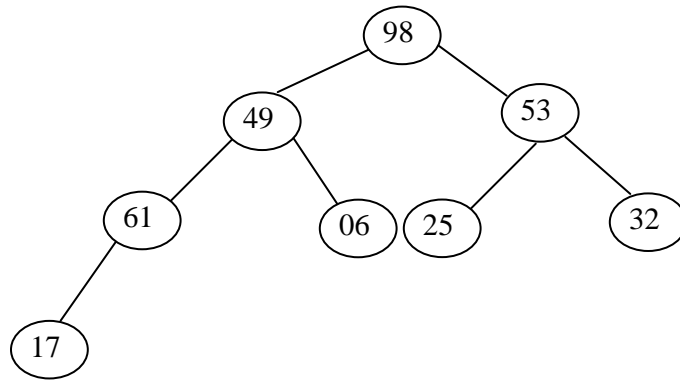
Giả sử ta đã có 1 heap như sau:



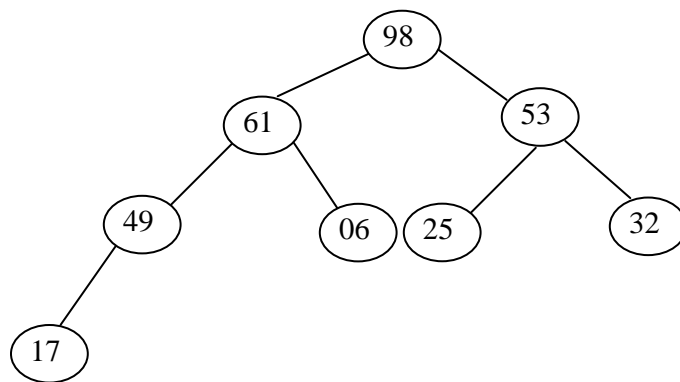
Để chèn phần tử 61 vào heap, đầu tiên, ta đặt nó vào vị trí cuối cùng trong cây.



Rõ ràng cây mới vi phạm định nghĩa heap vì nút con 61 lớn hơn nút cha 49. Tiến hành đổi vị trí 2 nút này cho nhau:



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 thỏa 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.

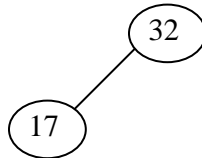
Ta hãy xem xét quá trình tạo heap với dãy số đã cho ở phần trước.

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

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

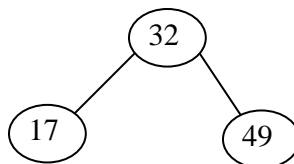


Bước 1: Tiến hành chèn 17 vào heap.

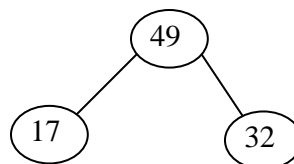


Do không vi phạm định nghĩa heap nên không thay đổi gì.

Bước 2: Tiến hành chèn 49 vào heap.

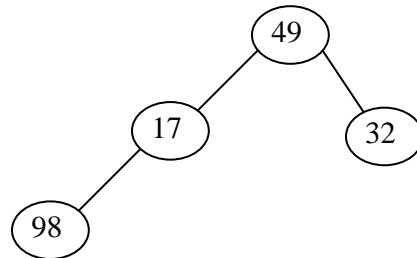


Cây này vi phạm định nghĩa heap do $49 > 32$ nên đổi vị trí 32 và 49 cho nhau.

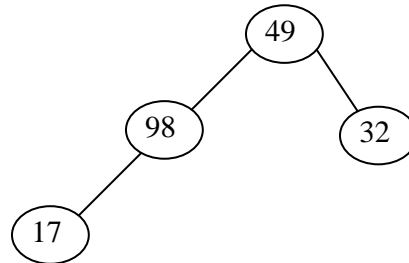


Cây mới thoả mãn định nghĩa heap.

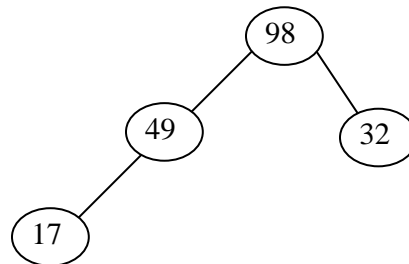
Bước 3: Tiến hành chèn 98 vào heap.



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

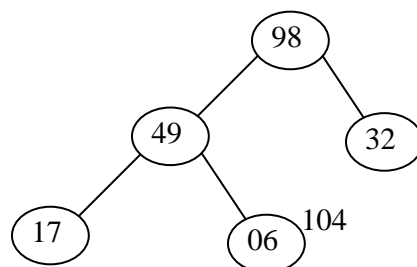


Cây mới lại vi phạm định nghĩa heap, do $98 > 49$, nên đổi vị trí 98 cho 49.



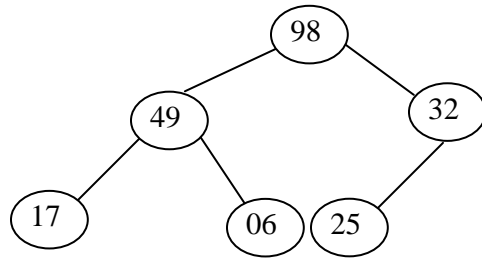
Cây này thoả mãn định nghĩa heap.

Bước 4: Tiến hành chèn 06 vào heap.



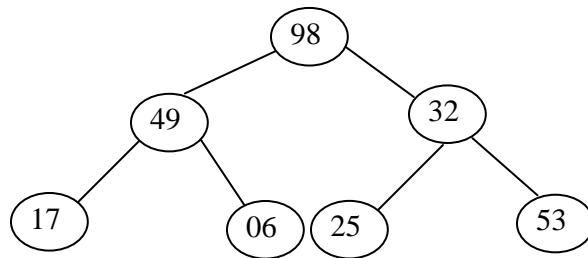
Cây này thoả mãn định nghĩa heap do $06 < 49$.

Bước 5: Tiến hành chèn 25 vào heap.

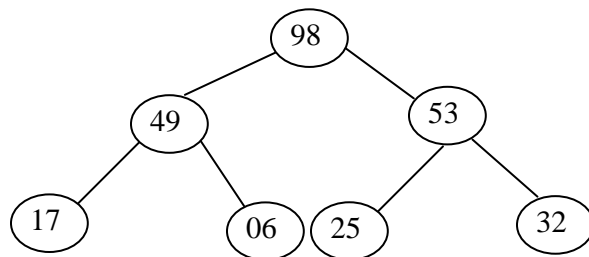


Cây này thoả mãn định nghĩa heap do $25 < 32$.

Bước 6: Tiến hành chèn 53 vào heap.

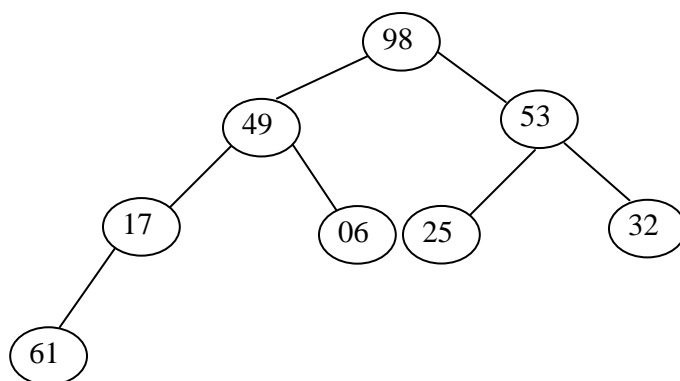


Cây này vi phạm định nghĩa heap do $53 > 32$ nên đổi vị trí 53 và 32 cho nhau.

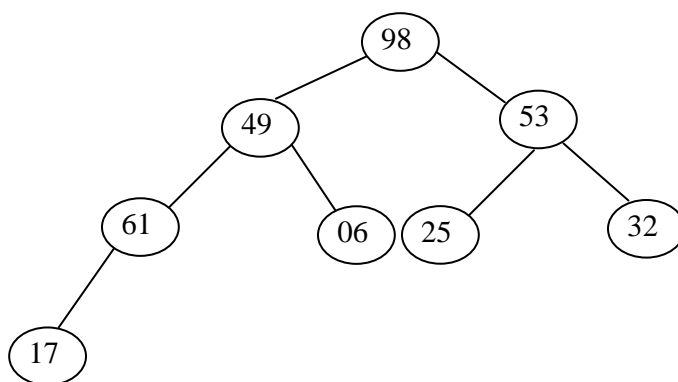


Cây mới thoả mãn định nghĩa heap.

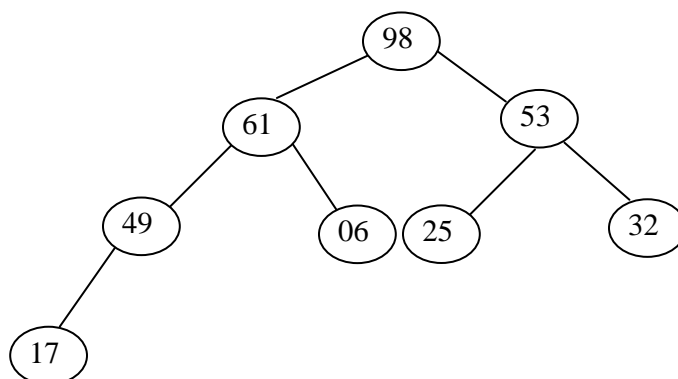
Bước 7: Tiến hành chèn 61 vào heap.



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.

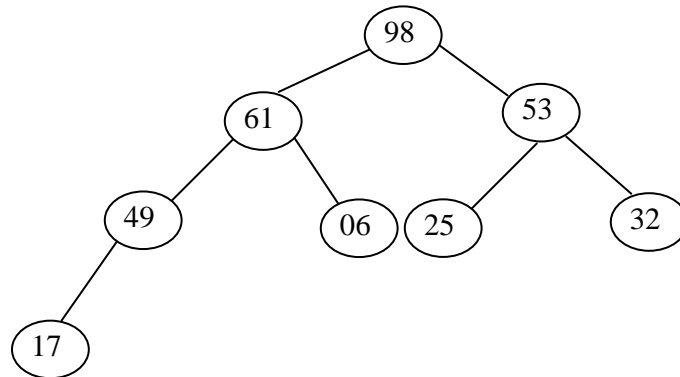


Cây này thoả mãn định nghĩa heap, và chính là heap cần tạo.

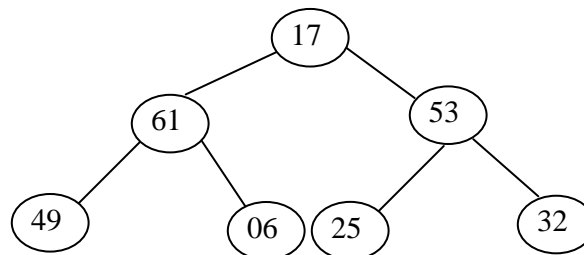
Sau khi tạo được heap, để tiến hành sắp xếp, ta cần lấy phần tử đầu và là phần tử lớn nhất của cây và thay thế nó bằng phần tử cuối của dãy. Điều này có thể làm vi phạm định nghĩa heap vì phần tử mới đưa lên gốc có thể nhỏ hơn 1 trong 2 nút con.

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.

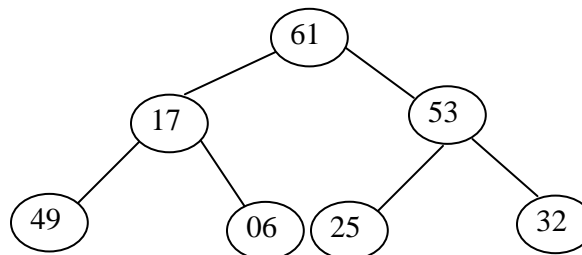
Ta xem xét ví dụ với heap vừa tạo được ở phần trước:



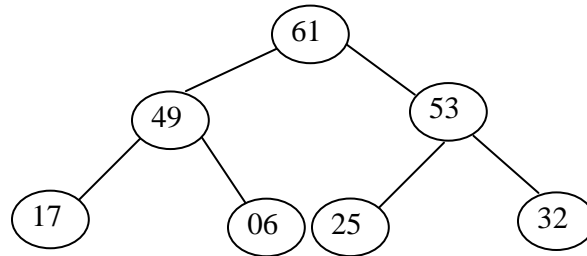
Lấy nút gốc 98 ra khỏi heap và thay thế bởi nút cuối là 17.



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 thỏa mãn định nghĩa heap như sau:

```

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 $O(N^2)$.

7.5 MERGE SORT (SẮP XẾP TRỘN)

7.5.1 Giới thiệu

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.

7.5.2 Trộn 2 dãy đã sắp

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
↓

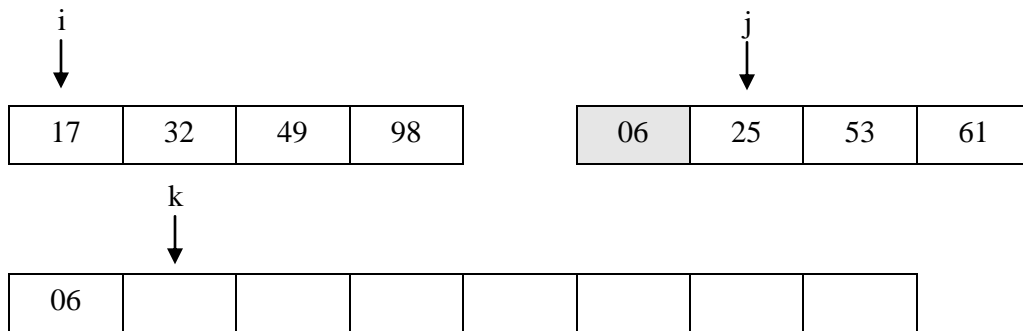
j
↓

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

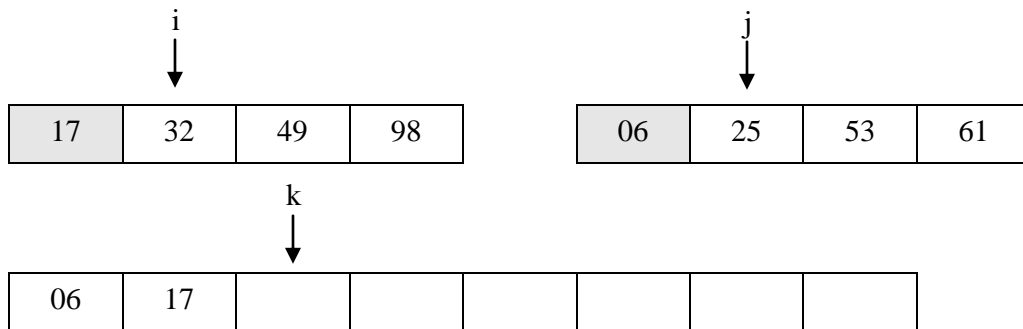
06	25	53	61
----	----	----	----

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.

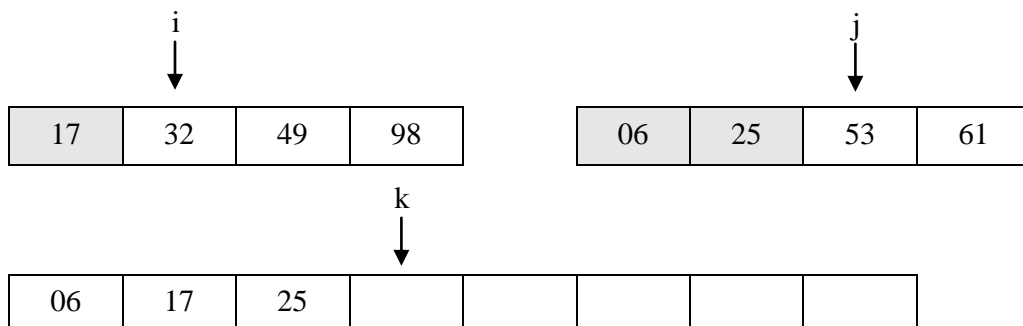
Bước 1: Phần tử tại vị trí biến duyệt j là 06 nhỏ hơn phần tử tại vị trí biến duyệt i là 17 nên ta đưa 06 xuống dãy mới và tăng j lên 1. Đồng thời, biến duyệt k cũng tăng lên 1.



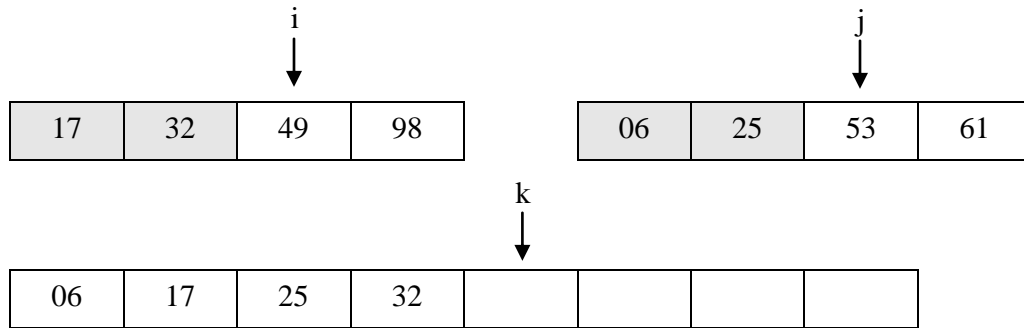
Bước 2: Phần tử tại vị trí i là 17 nhỏ hơn phần tử tại vị trí j là 25 nên ta đưa 17 xuống dãy mới và tăng i lên 1, biến duyệt k cũng tăng lên 1.



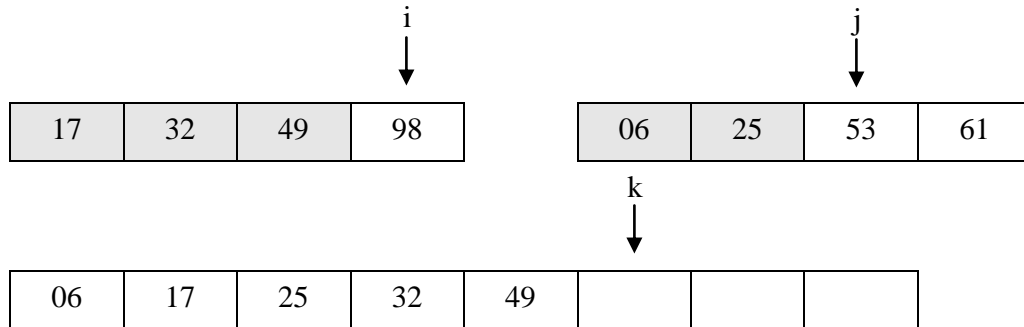
Bước 3: Phần tử tại vị trí j là 25 nhỏ hơn phần tử tại vị trí i là 32 nên ta đưa 25 xuống dãy mới và tăng j lên 1, biến duyệt k cũng tăng lên 1.



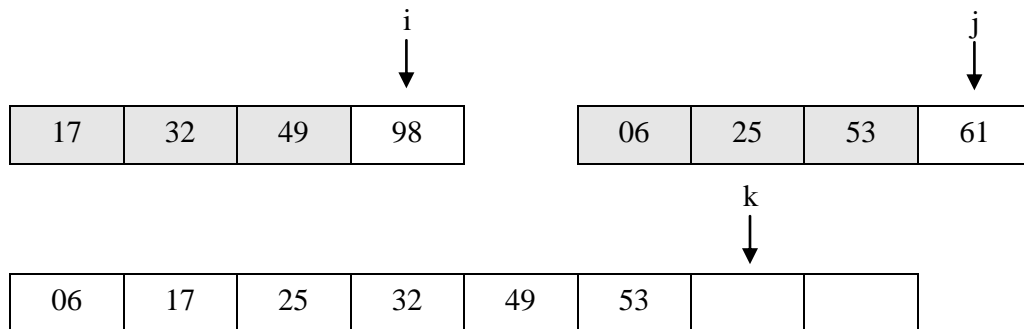
Bước 4: Phần tử tại vị trí i là 32 nhỏ hơn phần tử tại vị trí j là 53 nên ta đưa 32 xuống dãy mới và tăng i lên 1, biến duyệt k cũng tăng lên 1.



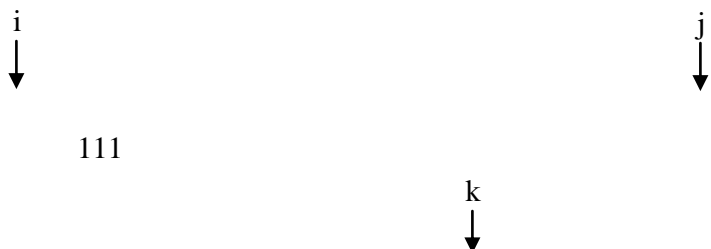
Bước 5: Phần tử tại vị trí i là 49 nhỏ hơn phần tử tại vị trí j là 53 nên ta đưa 49 xuống dãy mới và tăng i lên 1, biến duyệt k cũng tăng lên 1.



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.

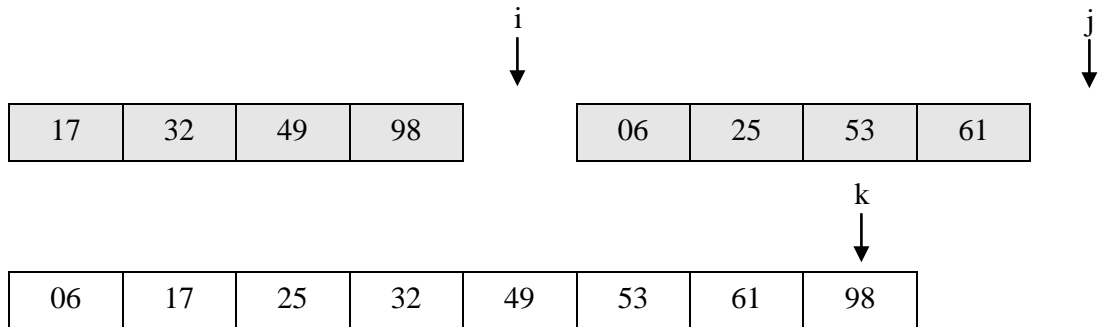


17	32	49	98
----	----	----	----

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

06	17	25	32	49	53	61	
----	----	----	----	----	----	----	--

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.



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 al 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.

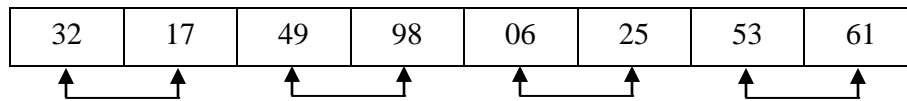
7.5.3 Sắp xếp trộn

Để tiến hành sắp xếp trộn, đầu tiên ta coi các phần tử của dãy như các dãy con 1 phần tử. Tiến hành trộn từng cặp 2 dãy con này để được các dãy con được sắp gồm 2 phần tử. Tiếp tục tiến hành trộn từng cặp dãy con 2 phần tử đã sắp để tạo thành các dãy con được sắp gồm 4 phần tử v.v. Quá trình lặp lại cho tới khi toàn bộ dãy được sắp.

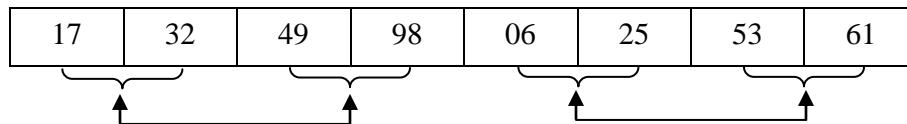
Ta xét quá trình sắp xếp trộn với dãy ở phần trước.

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

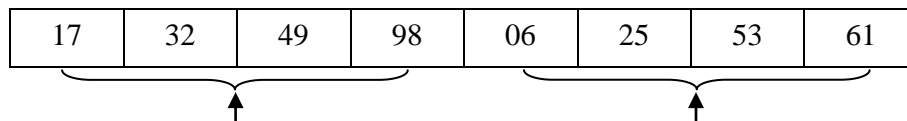
Đầu tiên, coi mỗi phần tử của dãy như 1 dãy con đã sắp gồm 1 phần tử. Tiến hành trộn từng cặp dãy con 1 phần tử với nhau:



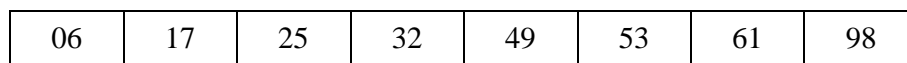
Sau bước này ta được các dãy con đã sắp gồm 2 phần tử. Tiến hành trộn các cặp dãy con đã sắp gồm 2 phần tử để tạo thành dãy con được sắp gồm 4 phần tử.



Sau bước này ta được các dãy con đã sắp gồm 4 phần tử. Tiến hành trộn 2 dãy con đã sắp gồm 4 phần tử.



Cuối cùng, ta có toàn bộ dãy đã được sắp:



Cài đặt cho thuật toán merge_sort bằng đệ qui như sau :

```
void merge_sort(int *a, int left, int right){
    int middle;
    if (right<=left) return;
    middle=(right+left)/2;
    merge_sort(a, left, middle);
    merge_sort(a, middle+1, right);
    merge(a, left, middle ,right);
}
```

Trong thủ tục này, đầu tiên ta tiến hành chia dãy cần sắp làm 2 nửa, sau đó thực hiện lời gọi đệ qui merge_sort cho mỗi nửa dãy. Hai lời gọi đệ qui này đảm bảo rằng mỗi nửa dãy này sẽ được sắp. Cuối cùng, thủ tục merge được gọi để trộn 2 nửa dãy đã sắp này.

7.6 BÀI 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.

7.7 TÌM KIẾM TUẦN TỰ

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.

7.8 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, biến 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.

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

Xét 1 ví dụ với dãy đã sắp ở trên, để tìm kiếm giá trị 61 trong dãy, ta tiến hành các bước như sau:

Bước 1: Phân chia dãy làm 2 nửa, với chỉ số phân cách là 3.

0	1	2	3	4	5	6	7
06	17	25	32	49	53	61	98

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.

4	5	6	7
49	53	61	98

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.

6	7
61	98

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.

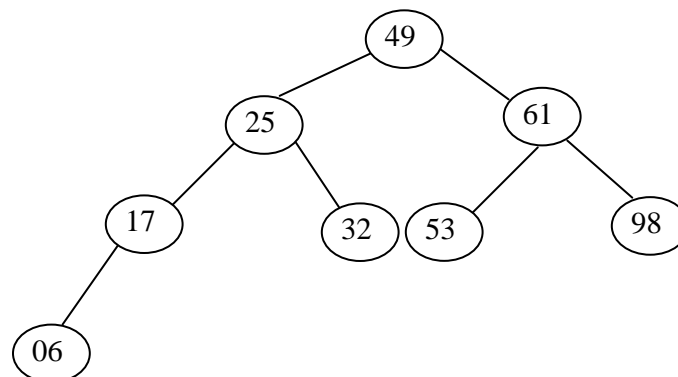
Thuật toán tìm kiếm nhị phân có thời gian thực hiện là $\lg N$. Tuy nhiên, thuật toán đòi hỏi dãy đã được sắp trước khi tiến hành tìm kiếm. Do vậy, nên áp dụng tìm kiếm nhị phân khi việc tìm kiếm phải thực hiện nhiều lần trên 1 tập phần tử cho trước. Khi đó, ta chỉ cần tiến hành sắp tập phần tử 1 lần và thực hiện tìm kiếm nhiều lần trên tập phần tử đã sắp này.

7.9 CÂY NHỊ PHÂN TÌM KIẾM

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.



Hình 7.2 Ví dụ về cây nhị phân tìm kiếm

7.9.1 Tìm kiếm trên cây nhị phân tìm kiếm

Việc tiến hành tìm kiếm trên cây nhị phân tìm kiếm cũng tương tự như phương pháp tìm kiếm nhị phân đã nói ở trên. Để tìm một nút có khoá là x , đầu tiên tiến hành so sánh nó với khoá của nút gốc. Nếu nhỏ hơn, tiến hành tìm kiếm ở cây con bên trái, nếu bằng nhau thì dừng quá trình tìm kiếm, và nếu lớn hơn thì tiến hành tìm kiếm ở cây con bên phải. Quá trình tìm kiếm trên cây con lại được lặp lại tương tự.

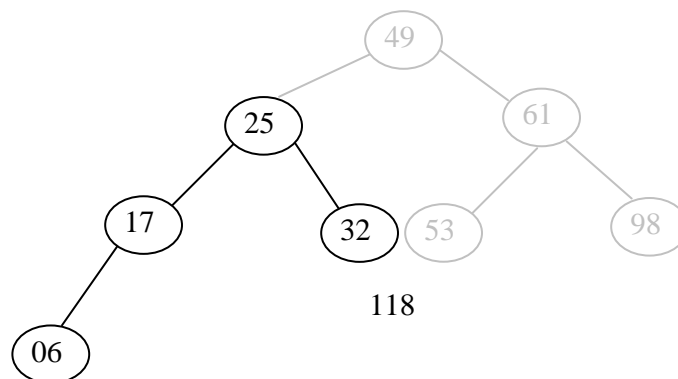
Tại mỗi bước, ta loại bỏ được một phần của cây mà chắc chắn là không chứa nút có khoá cần tìm. Phạm vi tìm kiếm luôn được thu hẹp lại và quá trình tìm kiếm kết thúc khi gặp được nút có khoá cần tìm hoặc không có nút nào như vậy (có nghĩa là cây con để tìm là cây rỗng).

```
struct node {
    int item;
    struct node *left;
    struct node *right;
}

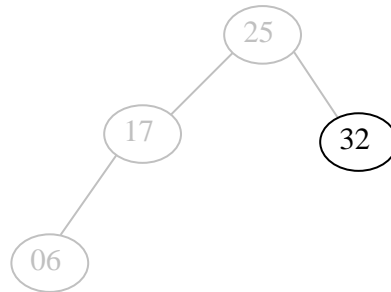
typedef struct node *treenode;
treenode tree_search(int x, treenode root){
    int found=0;
    treenode temp=root;
    while (temp!=NULL){
        if (x < temp.item) temp=temp.left;
        elseif (x > temp.item) temp=temp.right;
        else break;
    }
    return temp;
}
```

Xét cây nhị phân tìm kiếm như ở hình 7.2. Giả sử ta cần tìm nút 32 trên cây này. Quá trình tìm kiếm như sau:

Bước 1: So sánh 32 với nút gốc là 49. Do 32 nhỏ hơn 49 nên tiến hành tìm kiếm ở cây con bên trái.



Bước 2: So sánh 32 với nút gốc của cây tìm kiếm hiện tại là 25. Do 32 lớn hơn 25 nên tiến hành tìm kiếm ở cây con bên phải.



Bước 3: So sánh 32 với nút gốc của cây tìm kiếm hiện tại cũng là 32. Do 2 giá trị bằng nhau nên quá trình tìm kiếm kết thúc thành công.

Như vậy, chỉ sau 3 phép so sánh, thao tác tìm kiếm trong 1 danh sách gồm 7 phần tử đã kết thúc và cho kết quả.

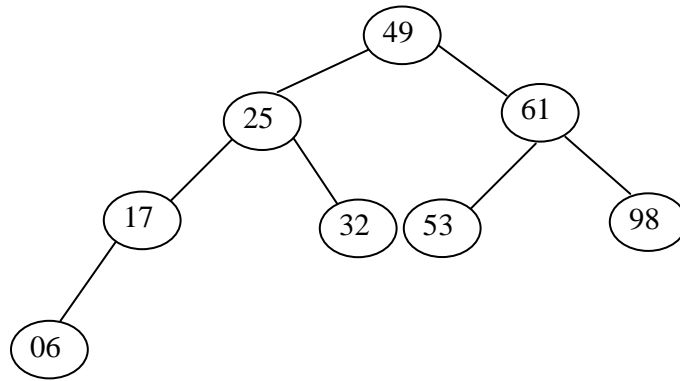
7.9.2 Chèn một phần tử vào cây nhị phân tìm kiếm

Để chèn một phần tử vào cây nhị phân tìm kiếm, đầu tiên tiến hành quá trình tìm kiếm nút cần chèn trong cây theo các bước như đã nói ở trên. Nếu tìm thấy nút trong cây, có nghĩa là cây đã tồn tại một nút có khoá bằng nút cần chèn và việc chèn thêm nút này vào cây là không hợp lệ. Nếu không tìm thấy nút trong cây thì quá trình tìm kiếm kết thúc không thành công và ta tiến hành chèn nút vào điểm kết thúc của quá trình tìm kiếm.

```
void tree_insert(int x, treenode *root){
    treenode p;
    p = (treenode) malloc (sizeof(struct node));
    p.item=x;
    p.left=p.right=NULL;
    if (root==NULL)
        root=p;
    else if (x < root->item)
        tree_insert(x, root->left)
    else
        tree_insert(x, root->right)
}
```

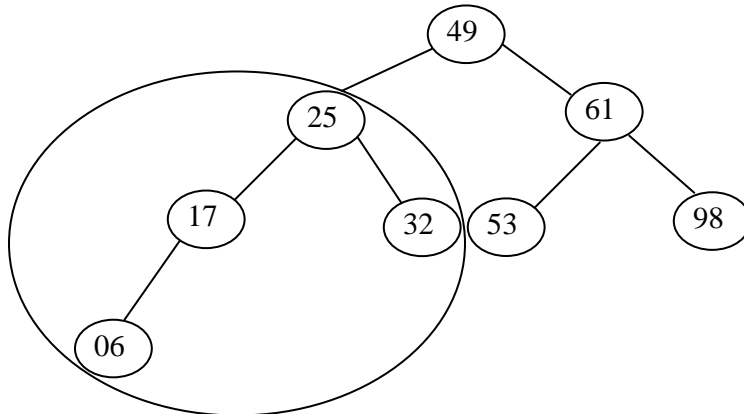
}

Thủ tục trên là một thủ tục đệ qui. Để chèn 1 phần tử x là cây có gốc là root. Tiến hành so sánh x với khoá của gốc. Nếu nó nhỏ hơn khoá của gốc thì tiến hành chèn vào cây con bên trái, ngược lại chèn vào cây con bên phải.

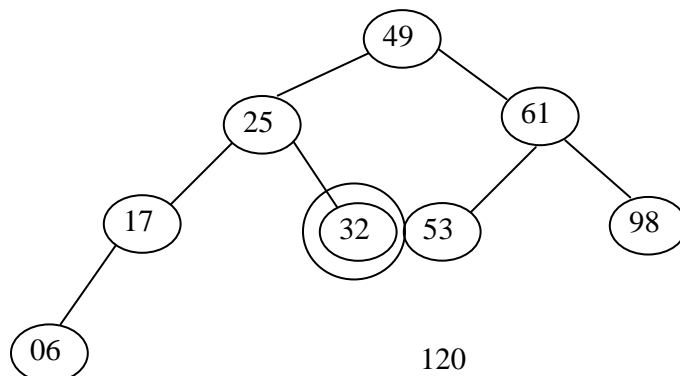


Xét cây nhị phân ở hình 7.2 Để tiến hành chèn nút có khoá là 30 vào cây, ta tiến hành các bước như sau:

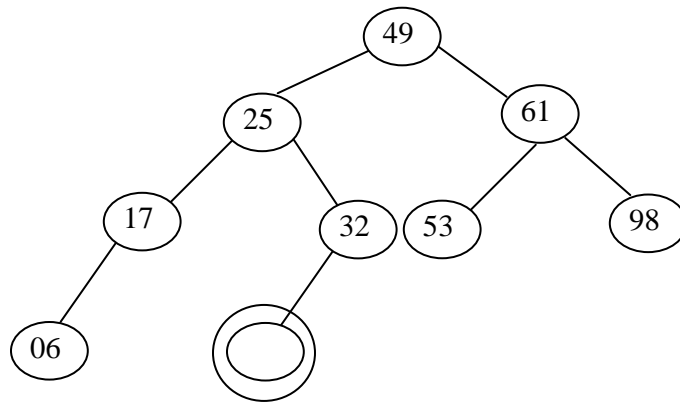
Bước 1: So sánh 30 với khoá nút gốc là 49. Do 30 nhỏ hơn 49 nên tiến hành chèn 30 vào cây con bên trái.



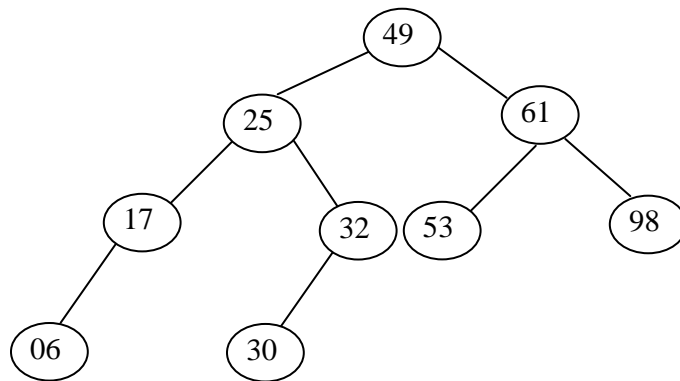
Bước 2: So sánh 30 với khoá nút gốc của cây hiện tại là 25. Do 30 lớn hơn 25 nên tiến hành chèn 30 vào cây con bên phải.



Bước 3: So sánh 30 với khoá nút gốc của cây hiện tại là 32. Do 30 nhỏ hơn 32 nên tiến hành chèn 30 vào cây con bên trái.



Bước 4: Do cây hiện tại là rỗng, do vậy đó chính là vị trí nút mới cần chèn.



Từ thủ tục chèn một nút vào cây nhị phân tìm kiếm ở trên, ta có thủ tục tạo cây nhị phân tìm kiếm từ một mảng cho trước như sau (cho trước 1 mảng a gồm n phần tử):

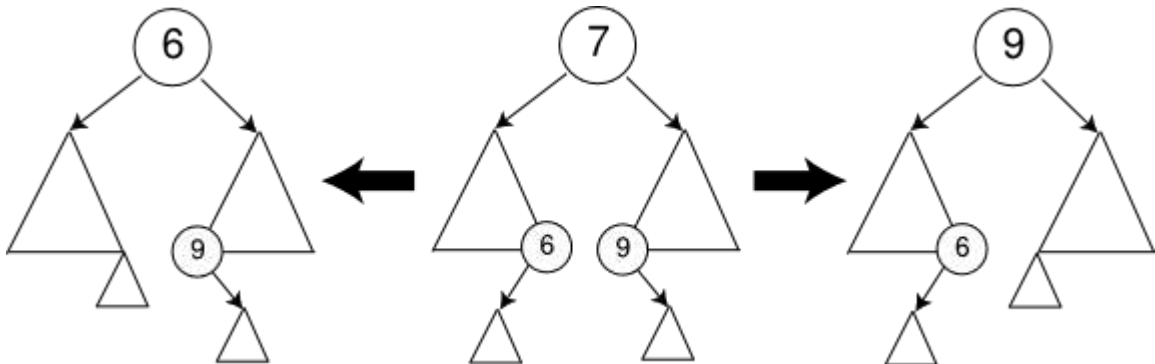
```
void tree_creation(treenode *root){
    int i;
    for (i=0; i<n; i++) tree_insert(a[i], root);
}
```

7.9.3 Xoá một nút khỏi cây nhị phân tìm kiếm

Để xoá một nút khỏi cây nhị phân, ta xét các trường hợp sau:

- Xoá 1 nút lá: Thao tác xoá 1 nút không có nút con nào là trường hợp đơn giản nhất. Chỉ cần tiến hành loại bỏ nút ra khỏi cây.
- Xoá nút có 1 nút con: Tiến hành loại bỏ nút khỏi cây và thay nó bằng nút con.

- Xóa nút có 2 nút con: Tiến hành loại bỏ nút và thay thế nó bằng nút con ngoài cùng bên trái của cây con bên phải hoặc nút con ngoài cùng bên phải của cây con bên trái.



7.10 TÓM TẮT CHƯƠNG 5

- Sắp xếp là quá trình bố trí lại các phần tử của 1 tập hợp theo thứ tự nào đó đối với 1 tiêu chí nào đó.
- Các giải thuật sắp xếp đơn giản dễ dàng trong việc cài đặt nhưng thời gian thực hiện lớn.
- Các giải thuật sắp xếp phức tạp cài đặt khó khăn hơn nhưng có thời gian chạy nhanh hơn.
- Các giải thuật sắp xếp đơn giản có thời gian thực hiện là $O(N^2)$ trong khi các giải thuật sắp xếp phức tạp có thời gian thực hiện là $O(N\log N)$.
- Quick sort là giải thuật sắp xếp dựa trên phương pháp chia để trị: Chia dãy cần sắp thành 2 phần, sau đó thực hiện việc sắp xếp cho mỗi phần độc lập với nhau. Các phần tử trong 1 phần luôn nhỏ hơn 1 giá trị khoá, còn các phần tử trong phần còn lại luôn lớn hơn giá trị khoá.
- Heap sort là 1 giải thuật sắp xếp dựa trên heap. Đó là một cây nhị phân hoàn chỉnh có tính chất khoá của nút cha bao giờ cũng lớn hơn khoá của các nút con. Quá trình sắp xếp sẽ đổi chỗ nút gốc của heap cho nút cuối, chỉnh lại heap và lại lặp lại quá trình cho tới khi dãy được sắp hoàn toàn.
- Sắp xếp trộn là phương pháp sắp xếp dựa trên việc trộn 2 danh sách đã sắp thành 1 danh sách cũng được sắp. Quá trình sắp xếp sẽ 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 toàn bộ dãy được sắp.
- Tìm kiếm là việc thu thập một số thông tin nào đó từ một khối thông tin lớn đã được lưu trữ trước đó.
- Tìm kiếm tuần tự 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.

- Tìm kiếm nhị phân sử dụng phương pháp chia để trị: Chia tập cần tìm làm 2 nửa, xác định nửa chứa bản ghi cần tìm và tập trung tìm kiếm trên nửa đó.
- Cây nhị phân tìm kiếm là 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.
- Quá trình tìm kiếm trên cây nhị phân tìm kiếm như sau: Để tìm một nút có khoá là x, đầu tiên tiến hành so sánh nó với khoá của nút gốc. Nếu nhỏ hơn, tiến hành tìm kiếm ở cây con bên trái, nếu bằng nhau thì dừng quá trình tìm kiếm, và nếu lớn hơn thì tiến hành tìm kiếm ở cây con bên phải. Quá trình tìm kiếm trên cây con lại được lặp lại tương tự.

7.11 CÂU HỎI VÀ BÀI TẬP

1. Cho dãy số:

15	37	12	58	07	24	67
----	----	----	----	----	----	----

- Nêu các bước trong quá trình thực hiện sắp xếp chọn, sắp xếp chèn, và sắp xếp nổi bọt cho dãy trên.
 - Với cách chọn khoá là phần tử đầu tiên, nêu các bước trong quá trình phân chia dãy làm 2 nửa với tính chất như trong giải thuật Quick sort.
 - Nêu các bước tạo heap từ dãy số trên.
 - Nêu các bước trong quá trình sắp xếp trộn cho dãy số trên.
 - Nêu các bước trong quá trình tạo cây nhị phân tìm kiếm từ dãy trên. Tiến hành các bước tìm phần tử 07 trong cây.
2. Hoàn thiện mã nguồn và tiến hành chạy cho ra kết quả tất cả các thuật toán sắp xếp được trình bày trong tài liệu.

Phụ lục 1

HƯỚNG DẪN GIẢI BÀI TẬP

CHƯƠNG I

1. Sinh viên tự giải.
2. Bài toán có thể được thiết kế tương tự như bài toán nút giao thông, sử dụng thuật toán tham ăn. Khi đó, xem mỗi cặp chưa đấu như là 1 nút của đồ thị và nếu 2 cặp có thể đấu trong cùng 1 tuần thì tồn tại 1 cạnh nối 2 nút tương ứng, ngược lại không tồn tại cạnh. Tiến hành tô màu cho đồ thị này và số màu được tô chính là số tuần cần tính.
3. Sinh viên tự giải.
4. Sinh viên tự giải.
5. Sinh viên tự giải.
6. Sinh viên tự giải.

CHƯƠNG II

1. Sinh viên tự tìm kiếm các ví dụ về đệ quy trong thực tế
2. Sinh viên tự giải.
3. Sinh viên tự giải.
4. Để viết chương trình tính tổng các số lẻ từ 1 đến $2n+1$, sử dụng công thức đệ quy:
$$\text{tong}(2*n+1) = 2*n+1 + \text{tong}(2*n-1)$$
5. Sinh viên tự giải.
6. Làm tương tự như bài Mã đi tuần.

CHƯƠNG III

1. Sinh viên tự giải.
2. Sinh viên tự giải.
3. Sinh viên tự giải.
4. Để in ra tất cả các phần tử của danh sách, tiến hành duyệt từ đầu đến cuối danh sách. Tại mỗi vị trí, tiến hành thao tác in giá trị thông tin của nút.
5. Để sắp xếp các phần tử trong một danh sách có thể sử dụng một trong các thuật toán được trình bày ở chương 7. Tuy nhiên, để đổi chỗ cho 2 phần tử trong danh sách, ta không cần đổi chỗ cả nút, mà chỉ cần đổi giá trị biến item của mỗi nút.

6. Để viết chương trình cộng 2 đa thức được biểu diễn thông qua danh sách liên kết, ta cần sử dụng một danh sách liên kết thứ 3 chứa đa thức tổng. Sử dụng 2 biến duyệt để duyệt từ đầu mỗi danh sách, nếu trùng bậc thì tiến hành cộng 2 hệ số và đưa kết quả xuống danh sách tổng. Nếu không trùng bậc thì tiến hành đưa hệ số của phần tử có bậc bé hơn xuống danh sách tổng, chuyển con trỏ tương ứng đến phần tử tiếp theo và lặp lại quá trình.
7. Tham khảo mã nguồn tại phụ lục 2.

CHƯƠNG IV

1. Sinh viên tự giải.
2. Sinh viên tự giải.
3. Sinh viên tự giải.
4. Sinh viên tự giải.
5. Sinh viên tự giải.
6. Thuật toán đổi số nguyên từ hệ thập phân sang hệ nhị phân sử dụng ngăn xếp như sau:
 - 1) Chia số thập phân cho 2.
 - 2) Lấy số dư đưa vào ngăn xếp.
 - 3) Nếu thương = 0 thì dừng
 - 4) Nếu thương lớn hơn 0 thì gán số đó bằng thương và quay lại bước 1
 - 5) Lần lượt lấy các số đã lưu trong ngăn xếp hiển thị ra màn hình, đó chính là dạng nhị phân của số ban đầu.
7. Sinh viên tự giải.
8. Tham khảo mã nguồn tại phụ lục 2.
9. Tham khảo mã nguồn tại phụ lục 2.

CHƯƠNG V

1. Sinh viên tự giải.
2. Sinh viên tự giải.
3. Sinh viên tự giải.
4. Sinh viên tự giải.
5. Sinh viên tự giải.

CHƯƠNG VI

1. Sinh viên tự giải.
2. Sinh viên tự giải.
3. Sinh viên tự giải.
4. Sinh viên tự giải.

5. Sinh viên tự giải.

CHƯƠNG VII

1. Sinh viên tự giải.
2. Tham khảo mã nguồn tại phụ lục 2.

Phụ lục 2

MÃ NGUỒN THAM KHẢO

DANH SÁCH LIÊN KẾT ĐƠN

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

struct node{
    int item;
    struct node *next;
};
typedef struct node *listnode;

void Insert_Begin(listnode *p, int x);
void Insert_End(listnode *p, int x);
void PrintList(listnode p);

void Insert_Begin(listnode *p, int x){
    listnode q;
    q = (listnode)malloc(sizeof(struct node));
    q-> item = x;
    q-> next = *p;
    *p = q;
    printf("\nThem nut vao dau danh sach thanh cong, bam phim bat ky
    de tiep tuc!");
    getch();
}

void Insert_End(listnode *p, int x){
    listnode q, r;
    q = (listnode)malloc(sizeof(struct node));
    q-> item = x;
    q->next=NULL;

    if (*p==NULL) *p=q;
    else{
```

```

        r = *p;
        while (r->next != NULL) r = r->next;
        r->next = q;
    }
    printf("\nThem nut vao cuoi danh sach thanh cong, bam phim bat ky
    de tiep tuc!");
    getch();
}

void Insert_Middle(listnode *p, int position, int x){
    int count=1, found=0;
    listnode q, r;
    r = *p;
    while ((r != NULL)&&(found==0)){
        if (count == position){
            q = (listnode)malloc(sizeof(struct node));
            q-> item = x;
            q-> next = r-> next;
            r-> next = q;
            found = 1;
        }
        count ++;
        r = r-> next;
    }
    if (found==0)
        printf("Khong tim thay vi tri can chen !");
    else
        printf("\nThem nut vao vi tri %d thanh cong, bam phim bat
        ky de tiep tuc!", position);
    getch();
}

void Remove_Begin(listnode *p){
    listnode q;
    if (*p == NULL) return;
    q = *p;
    *p = (*p)-> next;
    q-> next = NULL;
    free(q);
}

```

```

        printf("\nXoa nut dau danh sach thanh cong, bam phim bat ky de
        tiep tuc!");
        getch();
    }

void Remove_End(listnode *p){
    listnode q, r;
    if (*p == NULL) return;
    if ((*p)-> next == NULL){
        Remove_Begin(*p);
        return;
    }
    r = *p;
    while (r-> next != NULL){
        q = r;
        r = r-> next;
    }
    q-> next = NULL;
    free(r);
    printf("\nXoa nut cuoi danh sach thanh cong, bam phim bat ky de
    tiep tuc!");
    getch();
}

void Remove_Middle(listnode *p, int position){
    int count=1, found=0;
    listnode q, r;
    r = *p;
    while ((r != NULL)&&(found==0)){
        if (count == position){
            q = r-> next;
            r-> next = q-> next;
            q-> next = NULL;
            free (q);
            found = 1;
        }
        count ++;
        r = r-> next;
    }
}

```

```

    if (found==0)
        printf("Khong tim thay vi tri can xoa !");
    else
        printf("\nXoa nut o vi tri %d thanh cong, bam phim bat ky
        de tiep tuc!", position);
    getch();
}

void PrintList(listnode p){
    listnode q;
    q=p;
    while (q!=NULL){
        printf("%d ",q->item);
        q=q->next;
    }
    printf("\nBam phim bat ky de tiep tuc");
    getch();
}

void main(void){
    listnode *p;
    int x, chon, vitri;

    *p=NULL;

    do{
        clrscr();
        printf("CHUWONG TRINH MINH HOA SU DUNG DANH SACH LIEN KET
        DON\n\n");
        printf("Moi ban chon chuc nang:\n");
        printf("  1. Khoi tao danh sach\n");
        printf("  2. Them phan tu vao dau danh sach\n");
        printf("  3. Them phan tu vao cuoi danh sach\n");
        printf("  4. Them phan tu va giua danh sach\n");
        printf("  5. Xoa phan tu o dau danh sach\n");
        printf("  6. Xoa phan tu o cuoi danh sach\n");
        printf("  7. xoa phan tu o giua danh sach\n");
        printf("  8. In danh sach\n");
        printf("  9. Thoat khoi chuong trinh\n");
    }
}

```

```

printf("Lua chon: ");
scanf("%d",&chon);
switch (chon) {
    case 1:
        *p=NULL;
        break;
    case 2:
        clrscr();
        printf("Cho x= ");
        scanf("%d",&x);
        Insert_Begin(p, x);
        break;
    case 3:
        clrscr();
        printf("Cho x= ");
        scanf("%d",&x);
        Insert_End(p, x);
        break;

    case 4:
        clrscr();
        printf("Cho vi tri can chen: ");
        scanf("%d",&vitri);
        printf("\nCho x= ");
        scanf("%d",&x);
        Insert_Middle(p, vitri, x);
        break;
    case 5:
        clrscr();
        Remove_Begin(p);
        break;
    case 6:
        clrscr();
        Remove_End(p);
        break;
    case 7:
        clrscr();
        printf("Cho vi tri can xoa: ");
        scanf("%d",&vitri);

```

```

        Remove_Middle(p,vitri);
        break;
    case 8:
        PrintList(*p);
        break;
    default:
        break;
    }
}while (chon!=9);
return;
}

```

CÀI ĐẶT NGĂN XẾP BẰNG MẢNG

```

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

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

void StackInitialize(stack *s);
int StackEmpty(stack s);
int StackFull(stack s);
void Push(stack *s, int x);
int Pop(stack *s);

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

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

int StackFull(stack s){

```

```

        return (s.top==MAX-1);
    }
void Push(stack *s, int x){
    if (StackFull(*s)){
        printf("Ngan xep day");
        return;
    }else{
        s-> top ++;
        s-> nut[s-> top] = x;
        return;
    }
}

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

```

CÀI ĐẶT NGĂN XẾP BẰNG DANH SÁCH LIÊN KẾT

```

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

struct node {
    int 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, int x){
    stacknode p;
    p = (stacknode) malloc (sizeof(struct node));
    p-> item = x;
    p-> next = s->top;
    s->top = p;
    return;
}

int  Pop(stack *s){
    stacknode p;
    if (StackEmpty(*s)){
        printf("Ngan xep rong !");
    }else{
        p = s-> top;
        s-> top = s-> top-> next;
        return p->item;
    }
}

```

CÀI ĐẶT HÀNG ĐỢI BẰNG MẢNG

```

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

#define MAX 10
typedef struct  {
    int  head, tail, count;
    int  node[MAX];
} queue;

void QueueInitialize(queue *);

```

```

int QueueEmpty(queue);
void Put(queue *, int);
int Get(queue *);

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

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

void Put(queue *q, int x){
    if (q->count >= MAX) printf("Hang doi day !\n");
    else {
        if (q->tail==MAX-1 ) q->tail=0;
        else q->tail++;
        q->node[q->tail]=x;
        q->count++;
    }
    return;
}

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;
}

```

CÀI ĐẶT HÀNG ĐỢI BẰNG DANH SÁCH LIÊN KẾT

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

struct node {
    int item;
    struct node *next;
};

typedef struct node *queuenode;

typedef struct {
    queuenode head;
    queuenode tail;
}queue;

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

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

void Put(queue *q, int x){
    queuenode p;
    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;
}
```

```

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;
    }
}

```

CHƯƠNG TRÌNH SẮP XẾP CHỌN

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
#include<dos.h>

int *a, n, m;

void selection_sort();
void Init();
void In();

void Init(){
    int i;
    printf("\n Tao lap day so:");
    for (i=0; i<n;i++){
        a[i]=random(100);
        printf("%5d",a[i]);
    }
    delay(1000);
}

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;
    }
}

void In() {
    register int i;
    for(i=0; i<n; i++)
        printf("%5d", a[i]);
    printf("\n");
    getch();
}

void main(void) {
    clrscr();
    printf("\n Nhap n="); scanf("%d", &n);
    a=(int *) malloc(n*sizeof(int));
    Init();
    selection_sort();
    printf("\n\n Day da sap: ");
    In();
    free(a);
}

```

CHƯƠNG TRÌNH SẮP XẾP CHÈN

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
#include<dos.h>

int *a, n, m;

void Init();
void In();

```

```

void Init(){
    int i;
    printf("\n Tao lap day so:");
    for (i=0; i<n;i++){
        a[i]=random(100);
        printf("%5d",a[i]);
    }
    delay(1000);
}

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;
    }
}

void In(){
    register int i;
    for(i=0;i<n;i++)
        printf("%5d",a[i]);
    printf("\n");
    getch();
}

void main(void){
    clrscr();
    printf("\n Nhap n="); scanf("%d",&n);
    a=(int *) malloc(n*sizeof(int));
    Init();
    insertion_sort();
    printf("\n\n Day da sap: ");
}

```

```

        In();
        free(a);
    }

```

CHƯƠNG TRÌNH SẮP XẾP NỔI BỌT

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
#include<dos.h>

int *a, n, m;

void Init();
void In();

void Init(){
    int i;
    printf("\n Tao lap day so:");
    for (i=0; i<n;i++){
        a[i]=random(100);
        printf("%5d",a[i]);
    }
    delay(1000);
}

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;
                no_exchange = 0;
            }
        }
    }
}

```

```

        }
        i++;
    } while (!no_exchange && (i < n-1));
}

void In() {
    register int i;
    for (i=0; i<n; i++)
        printf("%5d", a[i]);
    printf("\n");
    getch();
}

void main(void) {
    clrscr();
    printf("\n Nhap n="); scanf("%d", &n);
    a=(int *) malloc(n*sizeof(int));
    Init();
    bubble_sort();
    printf("\n\n Day da sap: ");
    In();
    free(a);
}

```

CHƯƠNG TRÌNH SẮP XẾP NHANH (QUICK SORT)

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
#include<dos.h>

```

```

int *a, n, m;

```

```

void Init();
void In();

```

```

void Init() {
    int i;

```

```

printf("\n Tao lap day so:");
for (i=0; i<n;i++){
    a[i]=random(100);
    printf("%5d",a[i]);
}
delay(1000);
}

void quick(int left, int right) {
    int i,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);
}

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

void In(){
    register int i;
    for(i=0;i<n;i++)
        printf("%5d",a[i]);
    printf("\n");
    getch();
}

void main(void){
    clrscr();

```

```

printf("\n Nhap n="); scanf("%d",&n);
a=(int *) malloc(n*sizeof(int));
Init();
quick_sort();
printf("\n\n Day da sap: ");
In();
free(a);
}

```

CHƯƠNG TRÌNH SẮP XẾP VUN ĐỒNG (HEAP SORT)

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
#include<dos.h>

int *a, n, m;

void Init();
void In();

void Init(){
    int i;
    printf("\n Tao lap day so:");
    for (i=0; i<n;i++){
        a[i]=random(100);
        printf("%5d",a[i]);
    }
    delay(1000);
}

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++;
}

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;
}

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();
}

```

```

void In() {
    register int i;
    for(i=0; i<n; i++)
        printf("%5d", a[i]);
    printf("\n");
    getch();
}

void main(void) {
    clrscr();
    printf("\n Nhap n="); scanf("%d", &n);
    a=(int *) malloc(n*sizeof(int));
    Init();
    heap_sort();
    printf("\n\n Day da sap: ");
    In();
    free(a);
}

```

CHƯƠNG TRÌNH SẮP XẾP TRỘN (MERGE SORT)

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<alloc.h>
#include<dos.h>

int *a, *c, n, m;

void Init();
void In();

void Init(){
    int i;
    printf("\n Tao lap day so:");
    for (i=0; i<n; i++){
        a[i]=random(100);
        printf("%5d", a[i]);
    }
}

```

```

        delay(1000);
    }

void merge2(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++];
    }
}

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];
}

void merge_sort(int *a, int left, int right){
    int middle;

```

```

        if (right<=left) return;
        middle=(right+left)/2;
        merge_sort(a, left, middle);
        merge_sort(a, middle+1, right);
        merge(a, left, middle ,right);
    }

void In(){
    register int i;
    for(i=0;i<n;i++)
        printf("%5d",a[i]);
    printf("\n");
    getch();
}

void main(void){
    clrscr();
    printf("\n Nhap n="); scanf("%d",&n);
    a=(int *) malloc(n*sizeof(int));
    c=(int *) malloc(n*sizeof(int));
    Init();
    merge_sort(a, 0, n-1);
    printf("\n\n Day da sap: ");
    In();
    free(a);
}

```

TÀI LIỆU THAM KHẢO

- [1] Aho, Hopcroft & Ullman, *Data Structures and Algorithms*, Addison Wesley, 2001.
- [2] Robert Sedewick, *Algorithms in Java Third Edition*, Addison Wesley, 2002.
- [3] Niklaus Wirth, *Data Structures and Algorithms*, Prentice Hall, 2004.
- [4] Robert Sedewick, *Algorithms*, Addison Wesley, 1983.
- [5] Bruno R. Preiss, *Data Structures and Algorithms with Object-Oriented Design*, Jon Wiley & Sons, 1998.