

**BAN CƠ YẾU CHÍNH PHỦ
HỌC VIỆN KỸ THUẬT MẬT MÃ**

ThS. NGUYỄN VĂN PHÁC

**GIÁO TRÌNH
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

Hà Nội, 2013

**BAN CƠ YẾU CHÍNH PHỦ
HỌC VIỆN KỸ THUẬT MẬT MÃ**

ThS. NGUYỄN VĂN PHÁC

**GIÁO TRÌNH
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

Hà Nội, 2013

MỤC LỤC

Chương 1. GIỚI THIỆU CHUNG	1
1.1. Mối quan hệ giữa cấu trúc dữ liệu và giải thuật.....	1
1.2. Cấu trúc dữ liệu và các vấn đề liên quan	2
1.2.1. Việc lựa chọn và nghiên cứu các cấu trúc dữ liệu	2
1.2.2. Cấu trúc dữ liệu và các phép toán.....	2
1.2.3. Cấu trúc dữ liệu và cấu trúc lưu trữ	3
1.2.4. Cấu trúc dữ liệu trong các ngôn ngữ lập trình bậc cao	3
1.3. Ngôn ngữ diễn đạt giải thuật.....	5
Câu hỏi chương 1	6
Chương 2. THIẾT KẾ VÀ PHÂN TÍCH GIẢI THUẬT.....	7
2.1. Các phương pháp thiết kế giải thuật	7
2.1.1. Phương pháp thiết kế Top - Down.....	7
2.1.2. Phương pháp tinh chỉnh từng bước (Stepwise refinement)	9
2.2. Phân tích giải thuật.....	16
2.2.1. Đặt vấn đề	16
2.2.2. Phân tích thời gian thực hiện giải thuật	17
2.2.2.1. Độ phức tạp tính toán của giải thuật	18
2.2.2.2. Xác định độ phức tạp tính toán	19
2.2.2.3. Thời gian chạy của các câu lệnh	19
Câu hỏi và bài tập chương 2.....	25
Chương 3. ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY.....	27
3.1. Khái niệm đệ quy	27
3.2. Giải thuật đệ quy	27
3.3. Thiết kế một số giải thuật đệ quy.....	30
3.3.1. Hàm tính $n!$	30
3.3.2. Dãy số FIBONACCI.....	31
3.3.3. Bài toán “Tháp Hà Nội” (Tower of Hanoi)	32
3.3.4. Bài toán 8 quân hậu và giải thuật quay lui (Back tracking).....	35

3.4. Hiệu lực của đệ quy.....	41
3.5. Đệ quy và quy nạp toán học.....	42
Bài tập chương 3	44
Chương 4. MẢNG VÀ DANH SÁCH	46
4.1. Mảng.....	46
4.1.1. Định nghĩa mảng	46
4.1.2. Cấu trúc lưu trữ của mảng.....	46
4.2. Danh sách	47
4.2.1. Định nghĩa danh sách.....	47
4.2.2. Cài đặt danh sách bằng mảng.....	48
4.3. Ngăn xếp (Stack).....	48
4.3.1. Định nghĩa.....	48
4.3.2. Cài đặt ngăn xếp bằng mảng.....	49
4.3.3. Một số ví dụ về ứng dụng của ngăn xếp	51
4.3.3.1. Biểu thức dấu ngoặc cân xứng.....	51
4.3.3.2. Định giá biểu thức số học	53
4.3.3.3. Chuyển đổi biểu thức dạng trung tố sang dạng hậu tố.....	56
4.4.4. Ngăn xếp và việc cài đặt hàm đệ quy	59
4.4. Hàng đợi (Queue).....	65
4.4.1. Định nghĩa.....	65
4.4.2. Cài đặt hàng đợi bằng mảng.....	65
4.5. Danh sách móc nối đơn (singly linked list)	68
4.5.1. Nguyên tắc	68
4.5.2. Một số phép toán.....	69
4.5.2.1. Bổ sung một nút mới vào danh sách móc nối đơn.....	69
4.5.2.2. Loại bỏ một nút ra khỏi danh sách móc nối đơn.....	70
4.5.2.3. Ghép hai danh sách móc nối đơn	71
4.5.3. Ví dụ áp dụng	71

4.5.3.1. Biểu diễn đa thức	71
4.5.3.2. Giải thuật cộng đa thức	72
4.6. Danh sách móc nối vòng (circularly linked list).....	75
4.7. Danh sách móc nối kép (Doubly linked list)	76
4.8. Ngăn xếp và hàng đợi móc nối	79
Bài tập chương 4	80
Chương 5. CÂY	84
5.1. Định nghĩa và các khái niệm.....	84
5.2. Cây nhị phân (Binary tree).....	87
5.2.1. Định nghĩa và tính chất	87
5.2.2. Cài đặt cây nhị phân.....	89
5.2.2.1. Cài đặt cây nhị phân bằng mảng	89
5.2.2.2. Cài đặt cây nhị phân bằng móc nối.....	90
5.2.3. Phép duyệt cây nhị phân (traversing binary tree)	91
Bài tập chương 5	96
Chương 6. SẮP XẾP	100
6.1. Đặt vấn đề	100
6.2. Một số phương pháp sắp xếp đơn giản	101
6.2.1. Sắp xếp kiểu lựa chọn (Selection sort)	101
6.2.2. Sắp xếp kiểu thêm dần (Insertion sort)	102
6.2.3. Sắp xếp kiểu đổi chỗ (Exchange sort).....	104
6.2.4. Phân tích và so sánh ba phương pháp	106
6.3. Sắp xếp kiểu phân đoạn (Partition Sort) hay sắp xếp nhanh (Quick Sort)	107
6.3.1. Giới thiệu phương pháp	107
6.3.2. Ví dụ và giải thuật.....	107
6.3.3. Nhận xét và đánh giá.....	110
6.3.3.1. Vấn đề chọn “chốt”	110

6.3.3.2. Vấn đề phối hợp với cách sắp xếp khác.....	111
6.3.3.3. Đánh giá giải thuật.....	111
6.4. Sắp xếp kiểu vun đống (Heap sort).....	113
6.4.1. Giới thiệu phương pháp	113
6.4.1.1. Định nghĩa “đống”	113
6.4.1.2. Phép tạo đống.....	114
6.4.1.3. Sắp xếp kiểu vun đống.....	117
6.4.2. Nhận xét và đánh giá.....	119
6.5. Sắp xếp theo kiểu hoà nhập (Merge-sort)	120
6.5.1. Phép hoà nhập hai đường.....	120
6.5.2. Sắp xếp kiểu hoà nhập hai đường trực tiếp.....	122
6.5.3. Phân tích đánh giá	124
6.6. Những nhận xét cuối cùng	125
Bài tập chương 6	125
Chương 7. TÌM KIẾM	127
7.1. Bài toán tìm kiếm.....	127
7.2. Tìm kiếm tuần tự (Sequential Searching)	127
7.2.1. Ý tưởng và giải thuật.....	127
7.2.2. Đánh giá giải thuật.....	128
7.3. Tìm kiếm nhị phân	128
7.3.1. Ý tưởng và giải thuật.....	129
7.3.2. Phân tích và đánh giá	131
7.4. Cây nhị phân tìm kiếm.....	132
7.4.1. Định nghĩa cây nhị phân tìm kiếm.....	132
7.4.2. Giải thuật tìm kiếm	133
7.4.3. Phân tích đánh giá.....	135
7.4.4. Phép loại bỏ trên cây nhị phân tìm kiếm.....	136
Bài tập chương 7	140
Tài liệu tham khảo.....	141

DANH MỤC CÁC HÌNH

Hình 2.1. Mô hình chia bài toán lớn thành bài toán nhỏ	7
Hình 2.2. Chia bài toán chính thành 3 bài toán nhỏ.....	8
Hình 2.3. Chia bài toán nhỏ thành các bài toán nhỏ hơn	9
Hình 2.4. Các tuyến đường tại một chốt giao thông	10
Hình 2.5. Đồ thị biểu diễn sự giao cắt của các tuyến đường	11
Hình 2.6. Đồ thị minh hoạ cho giải thuật "tham ăn"	12
Hình 2.7. Đồ thị tăng trưởng giá trị của một số hàm số.....	18
Hình 3.1. Giải thuật tìm từ trong từ điển.....	28
Hình 3.2. Hình ảnh ban đầu bài toán chuyển tháp	33
Hình 3.3. Các bước chuyển tháp	34
Hình 3.4. Mô tả quân hậu ở vị trí: cột j, hàng i, đường chéo i-j, i+j.....	37
Hình 3.5. Hình ảnh một cách đặt 8 quân hậu.....	41
Hình 4.1. Hình ảnh của ngăn xếp.....	48
Hình 4.2. Hình ảnh cài đặt ngăn xếp bằng mảng có n phần tử	49
Hình 4.3. Cách tổ chức hai ngăn xếp dùng chung một mảng	51
Hình 4.4. Cách tổ chức ba ngăn xếp dùng chung một mảng	51
Hình 4.5. Các trạng thái của ngăn xếp khi đọc biểu thức $((() ()) ())$	53
Hình 4.6. Diễn biến việc thực hiện giải thuật định giá biểu thức	56
Hình 4.7. Hình ảnh diễn biến của ngăn xếp được dùng để tính $n!$	63
Hình 4.8. Hình ảnh của hàng đợi khi thêm, bớt phần tử.....	66
Hình 4.9. Tổ chức hàng đợi theo kiểu vòng.....	66
Hình 5.1. Hình ảnh cây thư mục trên ổ đĩa	84
Hình 5.3. Các tập bao nhau	85
Hình 5.4. Biểu diễn cây của các tập bao nhau	85
Hình 5.5. Hai “cây có thứ tự” khác nhau	86
Hình 5.6. Các cây nhị phân khác nhau.....	87

Hình 5.7. Các dạng đặc biệt của cây nhị phân	88
Hình 5.8. Cách đánh số cây nhị phân đầy đủ.....	89
Hình 5.9. Cây nhị phân bất kỳ	91
Hình 5.10. Hình ảnh cài đặt móc nối của cây nhị phân	91
Hình 6.1. Minh hoạ một cây nhị phân là đồng.....	113
Hình 6.2. Dựng cây nhị phân hoàn chỉnh từ mảng	114
Hình 6.3. Cây nhị phân chưa là đồng nhưng có hai con đã là đồng	115
Hình 6.4. Quá trình tạo thành đồng của cây nhị phân.....	115
Hình 6.5. Minh hoạ quá trình tạo thành đồng của cây.....	117
Hình 6.6. Minh hoạ quá trình sắp xếp HEAP-SORT.....	119
Hình 7.1. Ví dụ về cây nhị phân tìm kiếm	132
Hình 7.2. Dựng cây nhị phân tìm kiếm từ dãy khoá đã cho	135
Hình 7.3. Ví dụ về cây nhị phân cân đối.....	136
Hình 7.4. Cây sau khi bổ sung thêm giá trị khoá.....	136
Hình 7.5. Cây sau khi tái cân đối	136

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 nâng cao thêm về kỹ thuật lập trình, về phương pháp giải các bài toán trên máy tính, từ đó tiếp cận với việc thiết kế và xây dựng phần mềm cũng như sử dụng các công cụ lập trình hiện đại, đồng thời cũng giúp sinh viên có khả năng đi sâu thêm vào các môn học chuyên ngành như: cơ sở dữ liệu, trí tuệ nhân tạo, hệ chuyên gia, chương trình dịch...

Giáo trình *cấu trúc dữ liệu và giải thuật* này được biên soạn với thời lượng 2 tín chỉ (tương đương 45 tiết). Do thời lượng có hạn nên không thể trình bày hết các kiến thức về cấu trúc dữ liệu và giải thuật, mà chỉ nhằm giới thiệu những kiến thức cơ bản về cấu trúc dữ liệu và các giải thuật có liên quan. Trước khi học môn này sinh viên đã được học các môn “Tin học đại cương” và “Lập trình căn bản”, nên đã được biết các khái niệm về giải thuật (thuật toán), đã có những kiến thức cơ bản về lập trình bằng ngôn ngữ C. Bởi vậy trong giáo trình này có sử dụng những kiến thức đó mà không cần phải giới thiệu lại.

Nội dung giáo trình bao gồm 7 chương:

Chương 1: Trình bày mối quan hệ giữa cấu trúc dữ liệu và giải thuật, các vấn đề liên quan đến cấu trúc dữ liệu, ngôn ngữ dùng để diễn đạt giải thuật.

Chương 2: Trình bày các phương pháp thiết kế giải thuật, cách đánh giá độ phức tạp tính toán của giải thuật.

Chương 3: Trình bày các vấn đề về giải thuật đệ quy, cách thiết kế một số giải thuật đệ quy.

Chương 4: Trình bày các cấu trúc dữ liệu được sử dụng rất thông dụng như: mảng, danh sách, ngăn xếp, hàng đợi, danh sách móc nối đơn, danh sách móc nối vòng, danh sách móc nối kép, cùng với các phép toán trên các cấu trúc dữ liệu này và một số ví dụ ứng dụng.

Chương 5: Trình bày các khái niệm cơ bản về cây, trong đó tập trung trình bày một lớp cây đặc biệt là *cây nhị phân*, các cách cài đặt và các phép duyệt cây nhị phân.

Chương 6: Trình bày các giải thuật sắp xếp từ đơn giản đến phức tạp gồm: sắp xếp kiểu lựa chọn (Selection sort), sắp xếp kiểu thêm dần (Insertion sort), sắp xếp kiểu nổi bọt (Bubble sort), sắp xếp nhanh (Quick - sort), sắp xếp kiểu vun đống (Heap - sort), sắp xếp kiểu hoà nhập (Merge - sort), trong đó cũng trình bày việc đánh giá độ phức tạp tính toán của các giải thuật và sự so sánh giữa các giải thuật sắp xếp.

Chương 7: Trình bày giải thuật tìm kiếm tuần tự, giải thuật tìm kiếm nhị phân, khái niệm cây nhị phân tìm kiếm, cùng với giải thuật tìm kiếm có bổ sung và phép loại bỏ trên cây nhị phân tìm kiếm. Trong đó cũng trình bày việc đánh giá độ phức tạp tính toán của các giải thuật và sự so sánh giữa các giải thuật tìm kiếm.

Mặc dù đã hết sức cố gắng, nhưng chắc chắn không thể tránh khỏi những sai sót khi biên soạn. Bởi vậy tác giả rất mong muốn nhận được những ý kiến đóng góp để giáo trình này được hoàn thiện hơn.

Tác giả

CHƯƠNG 1

GIỚI THIỆU CHUNG

1.1. Môi quan hệ giữa cấu trúc dữ liệu và giải thuật

Để giải quyết một bài toán trên máy tính điện tử ta phải thực hiện một số bước, trong đó có một bước quan trọng là tìm ra giải thuật. Giải thuật (còn gọi là thuật toán) là một hệ thống chặt chẽ và rõ ràng các qui tắc nhằm xác định một dãy các thao tác trên những đối tượng, sao cho sau một số bước hữu hạn thực hiện các thao tác đó ta thu được kết quả mong muốn.

Nhưng giải thuật chỉ phản ánh các phép xử lý, còn đối tượng để xử lý trên máy tính điện tử chính là dữ liệu (data), chúng biểu diễn các thông tin cần thiết cho bài toán như: các dữ liệu vào, các kết quả trung gian, các dữ liệu ra... . Không thể nói tới giải thuật mà không nghĩ tới giải thuật đó được tác động trên dữ liệu nào. Còn khi xét tới dữ liệu thì cũng phải hiểu dữ liệu ấy cần được tác động bởi giải thuật gì để đưa tới kết quả mong muốn.

Bản thân các phần tử của dữ liệu thường có mối quan hệ với nhau, ngoài ra nếu biết tổ chức chúng theo các cấu trúc dữ liệu thích hợp thì việc thực hiện các phép xử lý trên các dữ liệu sẽ càng thuận lợi hơn, đạt hiệu quả cao hơn. Với một cấu trúc dữ liệu đã chọn ta sẽ có giải thuật xử lý tương ứng. Cấu trúc dữ liệu thay đổi thì giải thuật cũng thay đổi theo. Ta sẽ thấy rõ điều đó qua ví dụ sau:

Ví dụ: Giả sử ta có một danh sách các trường đại học và cao đẳng trên cả nước (danh sách chưa được sắp xếp), mỗi trường có các thông tin sau: Tên trường, địa chỉ, số điện thoại phòng đào tạo. Ta muốn viết một chương trình trên máy tính điện tử để khi cho biết “tên trường” máy sẽ hiện ra màn hình cho ta: “địa chỉ” và “số điện thoại phòng đào tạo” của trường đó. Đây là một bài toán mà phép xử lý cơ bản là “tìm kiếm” sẽ được tìm hiểu kỹ trong chương 7.

- Một cách đơn giản là cứ duyệt tuần tự các tên trường trong danh sách cho tới khi tìm thấy tên trường cần tìm, thì sẽ đối chiếu ra “địa chỉ” và “số điện thoại phòng đào tạo” của trường đó. Cách tìm tuần tự này rõ ràng chỉ chấp nhận được khi danh sách ngắn, còn danh sách dài thì rất mất thời gian.

- Nếu ta biết tổ chức lại danh sách bằng cách sắp xếp theo thứ tự từ điển của tên trường, thì có thể áp dụng một giải thuật tìm kiếm khác tốt hơn, tương tự như ta vẫn thường làm khi tra từ điển. Cách tìm này nhanh hơn cách trên rất nhiều nhưng rõ ràng không thể áp dụng được với dữ liệu chưa được sắp xếp.

- Nếu lại biết tổ chức thêm một bảng mục lục chỉ dẫn theo chữ cái đầu tiên của tên trường, thì khi tìm “địa chỉ” và “số điện thoại phòng đào tạo” của Học viện Kỹ thuật Mật mã, ta sẽ bỏ qua được các tên trường mà chữ cái đầu không phải là “H”.

Như vậy, giữa cấu trúc dữ liệu và giải thuật có mối quan hệ mật thiết. Có thể coi chúng như hình với bóng, không thể nói tới cái này mà không nhắc tới cái kia.

Chính điều đó đã dẫn tới việc, cần nghiên cứu các cấu trúc dữ liệu (data structures) đi đôi với việc xác lập các giải thuật xử lý trên các cấu trúc ấy.

1.2. Cấu trúc dữ liệu và các vấn đề liên quan

1.2.1. Việc lựa chọn và nghiên cứu các cấu trúc dữ liệu

Trong một bài toán, dữ liệu bao gồm một tập các phần tử cơ sở, mà ta gọi là *dữ liệu cơ bản (basic data)*. Chúng có thể là một chữ số, một ký tự ... , nhưng cũng có thể là một số, một từ ..., điều đó tùy thuộc vào từng bài toán.

Trên cơ sở của các dữ liệu cơ bản, các cách kiến tạo có thể để liên kết chúng lại với nhau, sẽ dẫn tới các cấu trúc dữ liệu khác nhau.

Lựa chọn một cấu trúc dữ liệu thích hợp để tổ chức dữ liệu vào và trên cơ sở đó xây dựng được giải thuật xử lý hữu hiệu để đưa tới kết quả mong muốn cho bài toán, đó là một khâu rất quan trọng.

Ban đầu, khi ứng dụng của máy tính điện tử chỉ mới có trong phạm vi các bài toán khoa học kỹ thuật thì ta chỉ gặp các cấu trúc dữ liệu đơn giản như: vectơ, ma trận Nhưng khi các ứng dụng đã mở rộng sang các lĩnh vực khác mà ta thường gọi là các *bài toán phi số (non numerical problems)*, với đặc điểm thể hiện ở chỗ: khối lượng dữ liệu lớn, đa dạng, biến động; phép xử lý thường không phải là các phép số học ... thì các cấu trúc dữ liệu này không đủ đặc trưng cho các mối quan hệ mới của dữ liệu nữa. Bởi vậy ta cần phải đi sâu nghiên cứu vào các cấu trúc dữ liệu phức tạp hơn.

1.2.2. Cấu trúc dữ liệu và các phép toán

Đối với các bài toán phi số, đi đôi với các cấu trúc dữ liệu mới cũng xuất hiện các phép toán mới tác động trên các cấu trúc ấy. Thông thường có các phép toán như: Phép tạo lập hoặc huỷ bỏ một cấu trúc, phép truy nhập vào từng phần tử của cấu trúc, phép bổ sung hoặc loại bỏ một phần tử trên cấu trúc ...

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

Vì vậy, khi chọn một cấu trúc dữ liệu ta phải nghĩ ngay tới các phép toán tác động trên cấu trúc ấy. Và ngược lại, nói tới phép toán thì lại phải chú ý tới phép đó được tác động trên cấu trúc dữ liệu nào. Cho nên người ta thường quan niệm: nói tới cấu trúc dữ liệu là bao hàm luôn cả phép toán tác động trên các cấu trúc ấy.

1.2.3. Cấu trúc dữ liệu và cấu trúc lưu trữ

Các cách biểu diễn một cấu trúc dữ liệu trong bộ nhớ máy tính điện tử được gọi là cấu trúc lưu trữ (storage structures). Đó chính là cách cài đặt cấu trúc ấy trên máy tính điện tử, và trên cơ sở cấu trúc lưu trữ này mà thực hiện các phép xử lý. Ta cần phân biệt giữa cấu trúc dữ liệu và cấu trúc lưu trữ tương ứng. Có thể có nhiều cấu trúc lưu trữ khác nhau cho cùng một cấu trúc dữ liệu, cũng như có thể có những cấu trúc dữ liệu khác nhau mà được thể hiện trong bộ nhớ bởi cùng một kiểu cấu trúc lưu trữ.

Ví dụ: Cấu trúc lưu trữ kế tiếp (mảng) và cấu trúc lưu trữ móc nối đều có thể được dùng để cài đặt cấu trúc dữ liệu ngăn xếp (STACK). Mặt khác, các cấu trúc dữ liệu như: danh sách, ngăn xếp và cây đều có thể cài đặt trên máy thông qua cấu trúc lưu trữ móc nối.

1.2.4. Cấu trúc dữ liệu trong các ngôn ngữ lập trình bậc cao

Trong các ngôn ngữ lập trình bậc cao, các dữ liệu được phân thành các kiểu dữ liệu. Kiểu dữ liệu của một biến được xác định bởi một tập các giá trị mà biến đó có thể nhận và các phép toán có thể thực hiện trên các giá trị đó. Ví dụ, kiểu dữ liệu “int” trong ngôn ngữ C có miền giá trị từ: -32768 đến 32767, các phép toán có thể thực hiện trên các giá trị này là: các phép toán số học, các phép toán thao tác bit, các phép toán logic và các phép toán so sánh.

Mỗi ngôn ngữ lập trình cung cấp cho chúng ta một số *kiểu dữ liệu cơ bản* (*basic data types*). Trong các ngôn ngữ lập trình khác nhau, các kiểu dữ liệu cơ bản có thể khác nhau. Các ngôn ngữ lập trình như: Pascal, C / C++ ... có các kiểu dữ liệu cơ bản rất phong phú.

Ví dụ: Ngôn ngữ C có các kiểu dữ liệu cơ bản sau:

Các kiểu ký tự (char, signed char, unsigned char)

Các kiểu nguyên (int, unsigned int, long int, unsigned long int)

Các kiểu thực (float, double, long double)

Kiểu liệt kê (enum)

Gọi là các kiểu dữ liệu cơ bản, vì các dữ liệu của các kiểu này sẽ được sử dụng như là các thành phần cơ sở để kiến tạo nên các dữ liệu có cấu trúc phức tạp. Các kiểu dữ liệu đã cài đặt sẵn (build-in types) mà ngôn ngữ lập trình cung cấp thường không đủ cho người sử dụng. Trong nhiều ứng dụng, người lập trình cần phải tiến hành các thao tác trên các dữ liệu phức hợp. Vì vậy, mỗi ngôn ngữ lập trình cung cấp cho người sử dụng một số quy tắc cú pháp để tạo ra các kiểu dữ liệu mới từ các kiểu cơ bản hoặc các kiểu khác đã được xây dựng. Chẳng hạn, ngôn ngữ C cung cấp cho người lập trình các quy tắc để xác định các kiểu dữ liệu mới như: kiểu mảng, kiểu cấu trúc (struct), kiểu móc nối, ...

Các kiểu dữ liệu được tạo thành từ nhiều kiểu dữ liệu khác (các kiểu này có thể là kiểu cơ bản hoặc kiểu dữ liệu đã được xây dựng) được gọi là *kiểu dữ liệu có cấu trúc*. Các dữ liệu thuộc kiểu dữ liệu có cấu trúc được gọi là các cấu trúc dữ liệu. Ví dụ, các mảng, các cấu trúc, các danh sách móc nối ... trong ngôn ngữ C, là các cấu trúc dữ liệu.

Từ các kiểu cơ bản, bằng cách sử dụng các qui tắc cú pháp để kiến tạo các kiểu dữ liệu, người lập trình có thể xây dựng nên các kiểu dữ liệu mới thích hợp cho từng vấn đề. Các kiểu dữ liệu mà người lập trình xây dựng nên được gọi là các kiểu dữ liệu được xác định bởi người sử dụng (user-defined data types).

Như vậy, một cấu trúc dữ liệu là một dữ liệu phức hợp, gồm nhiều thành phần dữ liệu, mỗi thành phần hoặc là dữ liệu cơ sở (số nguyên, số thực, ký tự,...) hoặc là một cấu trúc dữ liệu đã được xây dựng. Các thành phần dữ liệu tạo nên một cấu trúc dữ liệu được liên kết với nhau theo một cách nào đó.

Trong các ngôn ngữ lập trình như: Pascal, C/C++ , có ba phương pháp để liên kết các dữ liệu:

1. Liên kết các dữ liệu cùng kiểu tạo thành mảng dữ liệu.
2. Liên kết các dữ liệu (không nhất thiết cùng kiểu) tạo thành cấu trúc trong C/C++ , hoặc bản ghi trong Pascal.
3. Sử dụng con trỏ để liên kết dữ liệu. Chẳng hạn, sử dụng con trỏ ta có thể tạo nên các danh sách móc nối, hoặc sử dụng con trỏ để biểu diễn cây...

1.3. Ngôn ngữ diễn đạt giải thuật

Mặc dù vấn đề ngôn ngữ không được đặt ra ở giáo trình này, nhưng để diễn đạt các giải thuật được trình bày trong giáo trình ta không thể không lựa chọn một ngôn ngữ, ta có thể nghĩ ngay tới việc lựa chọn ngôn ngữ C, nhưng như vậy sẽ gặp mấy hạn chế sau:

- Phải luôn tuân thủ các quy tắc chặt chẽ về cú pháp của ngôn ngữ C, khiến cho việc trình bày về giải thuật và cấu trúc dữ liệu trở nên nặng nề, gò bó.

- Phải phụ thuộc vào các kiểu dữ liệu đã cài đặt sẵn của ngôn ngữ C, nên có lúc không thể hiện được đầy đủ các ý về cấu trúc mà ta muốn biểu đạt.

Vì vậy, để dễ dàng cho việc diễn đạt giải thuật trên các cấu trúc dữ liệu, ta sẽ dùng một ngôn ngữ “thô hơn” được gọi tạm là “ngôn ngữ tựa C”, với một mức độ linh hoạt nhất định, không quá gò bó, không cầu kỳ nhiều về cú pháp nhưng cũng gần gũi với ngôn ngữ C, để khi cần thiết ta có thể dễ dàng chuyển đổi thành chương trình C chạy được trên máy tính.

Do sinh viên đã được học ngôn ngữ lập trình C, nên dưới đây chỉ trình bày một số điểm khác của “ngôn ngữ tựa C” so với ngôn ngữ C.

Một số đặc điểm của “ngôn ngữ tựa C”:

- Không có chỉ thị tiền biên dịch `#include`
- Sử dụng thêm ký hiệu `//` để đặt lời chú thích ở phía sau (trên một dòng)
- Không cần khai báo biến, mảng trước khi sử dụng.
- Không gò bó trong việc mô tả các cấu trúc dữ liệu, mà chỉ cần diễn đạt được ý tưởng.

- Tên các đối tượng không phân biệt hoa thường.
- Không cần khai báo kiểu dữ liệu của hàm. Các đối của hàm cũng không cần khai báo kiểu dữ liệu.

- Hàm `scanf` không cần có đặc tả cho các đối và không cần viết địa chỉ của các đối. Ví dụ, để nhập dữ liệu cho các biến `a`, `b`, `c`, thì ta chỉ cần viết: `scanf(a,b,c)`.

- Hàm `printf` không cần có đặc tả chuyển dạng cho các đối được in ra. Ví dụ, để in ra màn hình các biến `a`, `b`, `c`, thì ta chỉ cần viết: `printf(a,b,c)`.

Ngoài những điểm lưu ý ở trên ra thì các câu lệnh còn lại được tuân theo cú pháp và cách thực hiện của ngôn ngữ C.

Ví dụ 1: Chương trình tính $n!$ có thể được viết bằng ngôn ngữ tựa C như sau:

Giai_thua(n) // Hàm tính giai thừa

```
{
    tg=1;
    for (i=1; i<=n; i++)
        tg=tg*i;
    return tg;
}

main()
{
    printf("nhap n:"); scanf(n);
    printf("n!=",giai_thua(n));
}
```

Câu hỏi chương 1

1.1. Tìm thêm các ví dụ minh họa mối quan hệ giữa cấu trúc dữ liệu và giải thuật.

1.2. Cấu trúc dữ liệu và cấu trúc lưu trữ khác nhau ở điểm nào.

1.3. Các cấu trúc dữ liệu đã cài đặt sẵn trong ngôn ngữ lập trình có đủ đáp ứng mọi yêu cầu về tổ chức dữ liệu hay không?

Có thể có cấu trúc dữ liệu do người dùng định ra không

1.4. Một chương trình C có phải là một tập dữ liệu có cấu trúc không?

CHƯƠNG 2

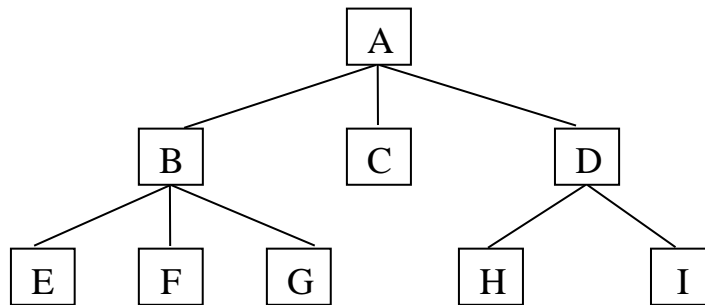
THIẾT KẾ VÀ PHÂN TÍCH GIẢI THUẬT

2.1. Các phương pháp thiết kế giải thuật

2.1.1. Phương pháp thiết kế Top - Down

Ngày nay công nghệ thông tin đã và đang được ứng dụng trong mọi lĩnh vực của cuộc sống, bởi vậy các bài toán giải được trên máy tính điện tử rất đa dạng và phức tạp, các giải thuật và chương trình để giải chúng cũng có qui mô ngày càng lớn, nên càng khó khi ta muốn tìm hiểu và thiết lập chúng.

Tuy nhiên, ta cũng thấy rằng mọi việc sẽ đơn giản hơn nếu như có thể phân chia bài toán lớn thành các bài toán nhỏ hơn. Điều đó cũng có nghĩa là nếu coi bài toán của ta như một mô-đun chính thì cần chia nó thành các mô-đun con, và dĩ nhiên, với tinh thần như thế, đến lượt nó, mỗi mô-đun con này lại được chia tiếp cho tới những mô-đun ứng với các phần việc cơ bản mà ta đã biết cách giải quyết. Như vậy việc tổ chức lời giải của bài toán sẽ được thể hiện theo một cấu trúc phân cấp có dạng như sau:



Hình 2.1. Mô hình chia bài toán lớn thành bài toán nhỏ

Cách giải quyết bài toán theo tinh thần như vậy được gọi là chiến thuật “chia để trị” (divide and conquer). Để thể hiện chiến thuật đó, người ta dùng cách thiết kế “đỉnh-xuống” (top-down design). Đó là cách phân tích tổng quát toàn bộ vấn đề, xuất phát từ dữ kiện và các mục tiêu đặt ra để đề cập đến những công việc chủ yếu trước, rồi sau đó mới đi dần vào giải quyết các phần việc cụ thể một cách chi tiết hơn, cũng vì vậy mà người ta còn gọi cách thiết kế này là cách thiết kế từ khái quát đến chi tiết.

Ví dụ: Để viết chương trình quản lý bán hàng chạy trên máy tính, với các yêu cầu là: hàng ngày phải nhập các hoá đơn bán hàng, hoá đơn nhập hàng, tìm kiếm các hoá đơn đã nhập để xem hoặc sửa lại; In các hoá đơn cho khách hàng; tính

doanh thu, lợi nhuận trong khoảng thời gian bất kỳ; Tính tổng hợp kho, tính doanh số của từng mặt hàng, từng khách hàng.

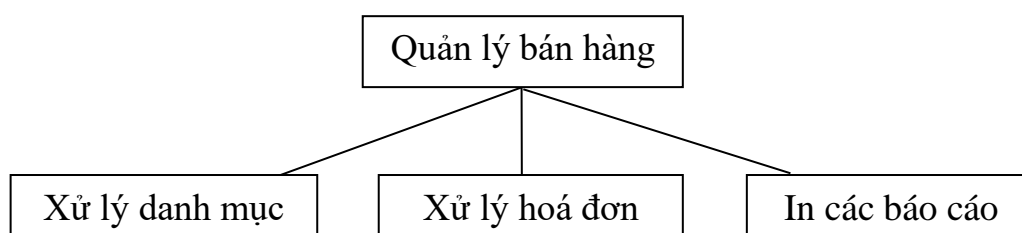
Xuất phát từ rất nhiều các yêu cầu trên ta không thể có ngay giải thuật để xử lý, mà nên chia bài toán thành 3 nhiệm vụ chính cần giải quyết như sau:

1) Xử lý các danh mục để quản lý và theo dõi các thông tin về hàng hoá và khách hàng.

2) Xử lý dữ liệu về các hoá đơn bán hàng, hoá đơn nhập hàng.

3) In các báo cáo về doanh thu, lợi nhuận.

Có thể hình dung cách thiết kế này theo sơ đồ cấu trúc sau:



Hình 2.2. Chia bài toán chính thành 3 bài toán nhỏ

Các nhiệm vụ ở mức đầu này thường vẫn còn tương đối phức tạp, nên cần phải chia tiếp thành các nhiệm vụ con. Chẳng hạn nhiệm vụ “Xử lý danh mục” được chia thành hai là “Danh mục hàng hoá” và “Danh mục khách hàng”.

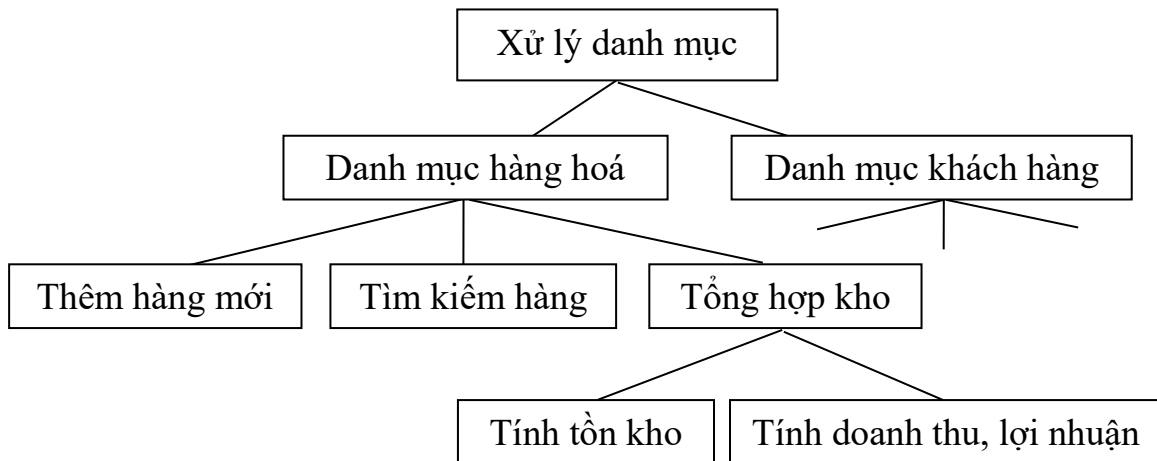
Trong “Danh mục hàng hoá” lại có thể chia thành các nhiệm vụ nhỏ hơn như:

1) Thêm hàng mới

2) Tìm kiếm hàng

3) Tổng hợp kho

Những nhiệm vụ con này cũng có thể chia thành các nhiệm vụ nhỏ hơn. Ta có thể hình dung theo sơ đồ cấu trúc sau:



Hình 2.3. Chia bài toán nhỏ thành các bài toán nhỏ hơn

Cách thiết kế giải thuật theo kiểu top-down như trên giúp cho việc giải quyết bài toán được định hướng rõ ràng, tránh sa đà ngay vào các chi tiết phụ. Nó cũng là nền tảng cho việc lập trình có cấu trúc.

Thông thường, đối với các bài toán lớn, việc giải quyết nó phải do nhiều người cùng làm. Chính phương pháp mô-đun hoá sẽ cho phép tách bài toán ra thành các phần độc lập, tạo điều kiện cho các nhóm giải quyết phần việc của mình mà không ảnh hưởng gì đến nhóm khác. Với chương trình được xây dựng trên cơ sở của các giải thuật được thiết kế theo cách này, thì việc tìm hiểu cũng như sửa chữa, chỉnh lý sẽ dễ dàng hơn.

Trong thực tế, việc phân bài toán thành các bài toán con như thế không phải là việc dễ dàng. Chính vì vậy mà có những bài toán, nhiệm vụ phân tích và thiết kế giải thuật giải bài toán còn mất nhiều thời gian và công sức hơn cả nhiệm vụ lập trình.

2.1.2. Phương pháp tinh chỉnh từng bước (Stepwise refinement)

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

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

vậy, ở các giai đoạn trung gian, người ta thường dùng pha tạp cả ngôn ngữ tự nhiên lẫn ngôn ngữ lập trình, mà người ta gọi là *giả ngôn ngữ* (pseudo language) hay giả mã (pseudo code). Như vậy nghĩa là quá trình thiết kế giải thuật và phát triển chương trình sẽ được thể hiện dần dần, từ dạng ngôn ngữ tự nhiên, qua giả ngôn ngữ, rồi đến ngôn ngữ lập trình, và đi từ mức “làm cái gì” đến mức “làm như thế nào”, ngày càng sát với các chức năng ứng với các câu lệnh của ngôn ngữ lập trình đã chọn.

Trong quá trình này dữ liệu cũng được “tinh chế” dần dần từ dạng cấu trúc dữ liệu đến dạng cấu trúc lưu trữ (cài đặt) cụ thể trên máy.

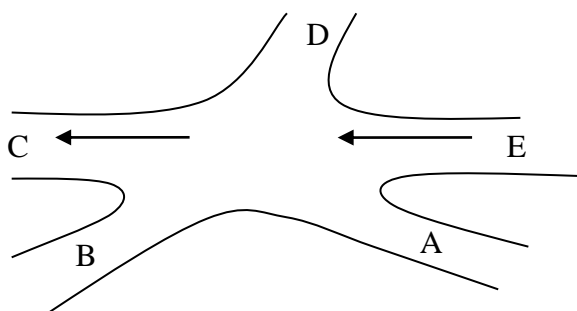
Sau đây ta xét ví dụ để minh họa cho phương pháp này:

Ví dụ: Giả sử ta cần thiết lập một qui trình điều khiển đèn giao thông ở một chốt giao thông phức tạp, có nhiều tuyến đường.

Như vậy nghĩa là phải điều khiển đèn sao cho trong một khoảng thời gian ấn định, thì chỉ một số tuyến đường được thông, trong khi một số tuyến khác bị cấm để tránh xảy ra ùn tắc.

Đối với bài toán này ta thấy: Ta nhận đầu vào là một số tuyến đường cho phép tại chốt giao thông đó và phải phân hoạch tập này thành một số ít nhất các nhóm, sao cho mọi tuyến trong một nhóm đều có thể cho thông đồng thời mà không xảy ra ùn tắc. Ta sẽ gán mỗi pha của việc điều khiển đèn giao thông với một nhóm trong phân hoạch này và việc tìm một phân hoạch với số nhóm ít nhất sẽ dẫn tới một qui trình điều khiển đèn với số pha ít nhất. Điều đó có nghĩa là thời gian chờ đợi tối đa để được thông tuyến cũng ít nhất.

Ví dụ, giả sử ở một đầu mối có các tuyến đường sau :



Hình 2.4. Các tuyến đường tại một chốt giao thông

Ở đây C và E là đường một chiều (1 tuyến) còn các đường khác đều hai chiều (2 tuyến).

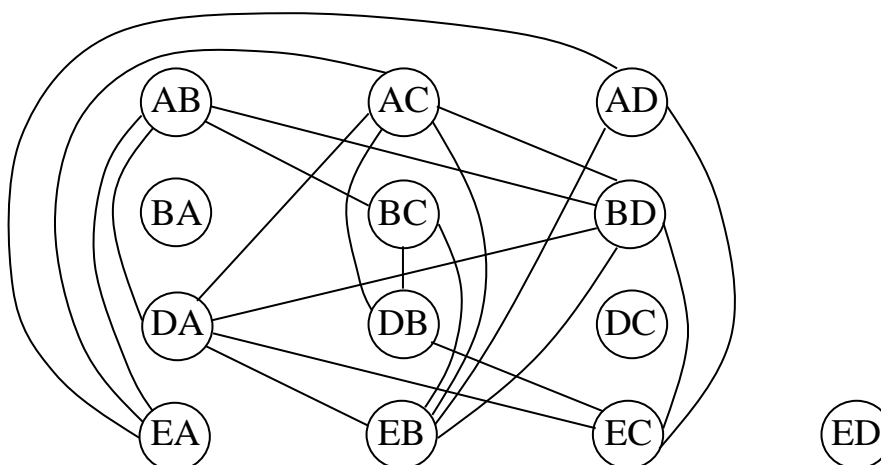
Như vậy sẽ có 13 tuyến có thể thực hiện qua đầu mối này. Những tuyến như AB (từ A tới B), EC có thể thông đồng thời. Nhưng tuyến như AD và EB thì không

thể thông đồng thời được vì chúng giao nhau, có thể gây ra ùn tắc (ta sẽ gọi là các tuyến xung khắc). Như vậy đèn tín hiệu phải báo sao cho AD và EB không thể được thông cùng một lúc, trong khi đó lại cho phép AB và EC chẳng hạn, được thông đồng thời.

Ta có thể mô hình bài toán này dựa vào một khái niệm, vốn đã được đề cập tới trong toán học, đó là *đồ thị* (*graph*).

Đồ thị bao gồm một tập các điểm gọi là đỉnh (*vertices*) và các đường nối các đỉnh gọi là cung (*edges*). Như vậy đối với bài toán "điều khiển đèn giao thông" này thì có thể hình dung một đồ thị mà các đỉnh biểu thị cho các tuyến đường, còn cung là nối một cặp đỉnh ứng với 2 tuyến đường xung khắc.

Với một đầu mối giao thông như ở hình 2.4, thì đồ thị biểu diễn nó sẽ như sau:



Hình 2.5. Đồ thị biểu diễn sự giao cắt của các tuyến đường

* Bây giờ ta sẽ tìm lời giải cho bài toán này dựa trên mô hình đồ thị đã nêu.

Ta sẽ đưa thêm khái niệm "tô màu cho đồ thị". Đó là việc gán màu cho mỗi đỉnh của đồ thị sao cho không có hai đỉnh nào nối với nhau bởi một cung lại cùng một màu.

Với khái niệm này, nếu ta hình dung mỗi màu đại diện cho một pha điều khiển đèn báo (cho thông một số tuyến và cấm một số tuyến khác), thì bài toán đang đặt ra chính là bài toán: tô màu cho đồ thị ứng với các tuyến đường ở một đầu mối giao thông như đã qui ước ở trên (xem hình 2.5), sao cho phải dùng ít màu nhất.

Bài toán *tô màu cho đồ thị* đã được nghiên cứu từ nhiều thập kỷ nay. Tuy nhiên bài toán tô màu cho đồ thị bất kỳ, với số màu ít nhất lại thuộc vào một lớp khá rộng các bài toán, được gọi là "bài toán NP đầy đủ", mà đối với chúng thì

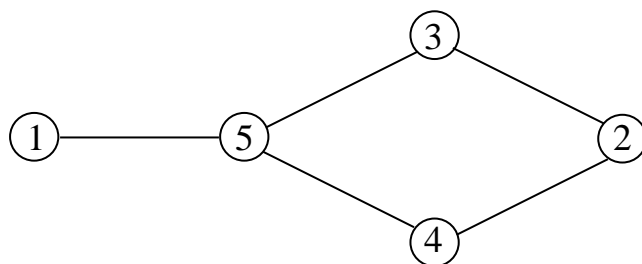
những lời giải hiện có chủ yếu thuộc loại "cố hết mọi khả năng". Trong trường hợp bài toán tô màu này thì "cố hết mọi khả năng" có nghĩa là cố gán màu cho các đỉnh, trước hết phải bằng một màu đã, khi không thể được nữa thì mới dùng tới màu thứ hai, thứ ba ... Cho tới khi đạt được mục đích. Nghe ra thì có vẻ đơn giản, nhưng đối với bài toán loại này, đây lại chính là một giải thuật có hiệu lực thực tế. Vấn đề khó nhất ở đây vẫn là khả năng tìm được lời giải tối ưu cho bài toán.

Nếu đồ thị nhỏ ta có thể cố thử theo mọi phương án, để có thể đi tới một lời giải tối ưu. Nhưng với đồ thị lớn thì cách này sẽ tốn rất nhiều thời gian, trên thực tế khó chấp nhận được. Cũng có thể có trường hợp do dựa vào một số thông tin phụ của bài toán mà việc tìm được lời giải tối ưu không cần phải thử tới mọi khả năng. Nhưng đó chỉ là những trường hợp hiếm gặp. Còn một cách khác nữa là thay đổi cách nhìn nhận về lời giải của bài toán đi một chút: Thay vì đi tìm lời giải tối ưu, ta đi tìm một lời giải "tốt" theo nghĩa mà thực tế chấp nhận được. Một giải thuật "tốt" như vậy (tuy lời giải không phải là tối ưu) được gọi là giải thuật *heuristic*.

Một giải thuật heuristic hợp lý đối với bài toán tô màu cho đồ thị đã nêu là giải thuật "tham ăn" (greedy algorithm) có nội dung như sau: Đầu tiên, ta cố tô màu cho các đỉnh nhiều hết mức có thể bằng một màu. Với các đỉnh còn lại (chưa được tô) lại làm hết mức có thể với màu thứ hai, và cứ như thế.

Để tô màu cho các đỉnh với màu mới, ta sẽ thực hiện các bước sau:

- 1- Chọn một đỉnh chưa được tô màu nào đó và tô cho nó bằng màu mới.
- 2- Tìm trong danh sách các đỉnh chưa được tô màu, với mỗi đỉnh đó xác định xem có một cung nào nối nó với một đỉnh đã được tô bằng màu mới chưa. Nếu chưa có thì tô đỉnh đó bằng màu mới.



Hình 2.6. Đồ thị minh họa cho giải thuật "tham ăn"

Cách tiếp cận này có cái tên "tham ăn" vì nó thực hiện tô màu một đỉnh nào đó mà nó có thể tô được, không hề chú ý gì đến tình thế bất lợi có thể xuất hiện khi làm điều đó. Chẳng hạn, với đồ thị như hình 2.6 thì như sau:

Nếu nó tô màu xanh cho đỉnh ① thì theo thứ tự nó sẽ tìm đến đỉnh ② và cũng tô luôn màu xanh. Như vậy thì đỉnh ③ và ④ sẽ phải tô màu đỏ, rồi ⑤ sẽ phải tô

màu vàng chẳng hạn, như vậy là phải dùng tới 3 màu. Còn nếu biết cân nhắc hơn, ta tô các đỉnh ① ③ ④ màu xanh thì chỉ cần tô màu đỏ cho các đỉnh ② và ⑤ nghĩa là bớt được một màu.

Bây giờ ta xét đến việc áp dụng giải thuật "tham ăn" cho đồ thị ở hình 2.5:

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 cung nối đỉnh này với AB. AD cũng có thể tô màu xanh vì không có cung nối AD với AB, AC. Đỉnh BA không có cung 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 cung nối BC và AB. Tương tự như vậy, BD, DA, DB không thể tô màu xanh vì tồn tại cung nối chúng tới một trong các đỉnh đã tô màu xanh. Cung DC thì có thể tô màu xanh. Cuối cùng, cung EA, EB, EC cũng không thể tô màu xanh trong khi ED có thể được tô màu xanh.

Tiếp theo, ta sử dụng màu đỏ để tô các đỉnh chưa được tô màu ở bước trước. Đầu tiên là BC và BD cùng có thể tô màu đỏ, tuy nhiên do tồn tại cung 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 cung nối tới các đỉnh đã tô màu đỏ nên cũng không được tô màu.

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

Như vậy, ta có thể dùng 4 màu: xanh, đỏ, tím, vàng để tô màu cho đồ thị ở hình 2.5 theo yêu cầu như đã nói ở trên. Bảng tổng hợp màu tô được mô tả như sau:

Bảng 2.1. Các màu được tô cho các đỉnh

Tô màu	Cho tuyến	Số tuyến
Xanh	AB, AC, AD, BA, DC, ED	6
Đỏ	BC, BD, EA	3
Tím	DA, DB	2
Vàng	EB, EC	2

Nếu liên hệ với quy trình điều khiển đèn ở chốt giao thông như hình 2.4 thì những kết quả trên là tương đương với 4 pha. Ở mỗi pha, chỉ các tuyến ứng với một màu trong bảng trên là được thông, còn các tuyến ứng với màu khác sẽ bị cấm.

Điều đó cũng có nghĩa là cùng lắm mới phải chờ đến pha thứ tư. Giả sử mỗi pha mất 1 phút thì sau 3 phút tuyến EB và EC mới được thông, sau 4 phút tuyến AB, AC,... mới lại được phục hồi và cứ như thế.

Như vậy là ta đã chọn được mô hình cho bài toán điều khiển đèn giao thông và xác định được giải thuật xử lý dựa trên mô hình ấy.

Còn bây giờ ta bắt đầu đi vào các bước tinh chỉnh giải thuật "tham ăn", bằng giả ngôn ngữ, để rồi hướng tới chương trình bằng ngôn ngữ C.

Giả sử ta gọi đồ thị được xét là G. Hàm "Tham_an" sau đây sẽ xác định chi tiết hơn các đỉnh sẽ được tô cùng một màu mới, dưới dạng các phần tử của một tập Mau_moi. Hàm này sẽ được gọi đi gọi lại nhiều lần cho tới khi tất cả các đỉnh của G đều được tô màu. Ở mức tổng quát, Hàm được mô tả như sau:

```
Tham_an(G, 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;
        }
}
```

Ta nhận thấy rằng các phát biểu trong hàm 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 hàm 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 cung nối v và w không. Để lưu giữ kết quả kiểm tra, ta sử dụng một biến Ton_tai. Khi đó, hàm được tinh chỉnh lại như sau:


```

Tham_an(G, Mau_moi)
{
    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 cung nối v và w trong G
                Ton_tai = 1;
        if (ton_tai == 0) //v không được nối tới đỉnh nào trong tập Mau_moi
        {
            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 biểu 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 for bên trong có thể thực hiện tương tự.

```

Tham_an(G, Mau_moi)
{
    Mau_moi = Tập rỗng;
    v = đỉnh chưa tô màu đầu tiên trong G ;
    while (v != NULL) // Duyệt các đỉnh chưa tô màu
    {
        ton_tai = 0;
        w = đỉnh đầu tiên trong Mau_moi;
    }
}

```

```

while (w != NULL && ton_tai==0)
{
    if tồn tại cung nối v và w trong G
        ton_tai = 1;
        w = đỉnh tiếp theo trong Mau_moi ;
    }
if (ton_tai == 0) //v không được nối tới đỉnh nào trong tập Mau_moi
{
    Tô màu mới cho đỉnh v;
    Đưa v vào tập Mau_moi;
}
v = đỉnh chưa tô màu tiếp theo trong G;
}
}

```

Cứ như vậy, qua nhiều bước tinh chỉnh ta sẽ cụ thể thêm các phép xử lý, đưa ra cách cài đặt đối với cấu trúc dữ liệu đồ thị. Vì chương trình ứng với giải thuật này khá dài nên ta tạm dừng tại đây.

2.2. Phân tích giải thuật

2.2.1. Đặt vấn đề

Khi ta đã xây dựng được giải thuật và chương trình tương ứng để giải một bài toán, thì có thể có hàng loạt yêu cầu về phân tích được đặt ra. Chẳng hạn: yêu cầu phân tích *tính đúng đắn* của giải thuật, liệu nó có thể hiện được đúng lời giải của bài toán không? Thông thường người ta có thể cài đặt chương trình thể hiện giải thuật đó trên máy tính, và thử nghiệm nó ở một số bộ dữ liệu nào đấy rồi so sánh kết quả thử nghiệm với kết quả mà ta đã biết. Nhưng cách thử này chỉ phát hiện được tính sai chứ chưa thể đảm bảo được tính đúng của giải thuật, Với các công cụ toán học người ta cũng có thể chứng minh được tính đúng đắn của giải thuật, nhưng công việc này không phải là dễ dàng, ta cũng không đặt vấn đề đi sâu thêm ở đây.

Loại yêu cầu thứ hai là về tính đơn giản của giải thuật. Thông thường ta vẫn mong muốn có được một giải thuật đơn giản, nghĩa là dễ hiểu, dễ lập trình, dễ chỉnh lý. Nhưng cách đơn giản để giải một bài toán chưa hẳn lúc nào cũng là cách tốt. Thường thường nó hay gây ra tốn phí thời gian hoặc bộ nhớ khi thực hiện. Đối

với chương trình chỉ để dùng một vài lần thì tính đơn giản này cần được coi trọng, vì như ta đã biết công sức và thời gian để xây dựng được chương trình giải một bài toán thường rất lớn so với thời gian thực hiện chương trình đó. Nhưng nếu chương trình sẽ được sử dụng nhiều lần, nhất là đối với loại bài toán mà khối lượng dữ liệu được đưa vào khá lớn, thì thời gian thực hiện rõ ràng phải được chú ý. Lúc đó yêu cầu đặt ra lại là tốc độ, hơn nữa nếu khối lượng dữ liệu lớn mà dung lượng bộ nhớ lại có giới hạn, thì không thể bỏ qua yêu cầu về tiết kiệm bộ nhớ được.

Sau đây ta sẽ chú ý đến việc phân tích thời gian thực hiện giải thuật, một trong các tiêu chuẩn để đánh giá hiệu lực của giải thuật vốn hay được đề cập tới.

2.2.2. Phân tích thời gian thực hiện giải thuật

Với một bài toán, không phải chỉ có một giải thuật. Chọn một giải thuật để đưa tới kết quả nhanh là một đòi hỏi thực tế. Nhưng căn cứ vào đâu để có thể nói được giải thuật này nhanh hơn giải thuật kia?

Có thể thấy ngay: thời gian thực hiện một giải thuật (hay chương trình thể hiện giải thuật đó) phụ thuộc vào rất nhiều yếu tố. Một yếu tố cần chú ý trước tiên đó là kích thước của dữ liệu đưa vào. Chẳng hạn thời gian sắp xếp một dãy số phải chịu ảnh hưởng của số lượng các số thuộc dãy số đó. Nếu gọi n là số lượng này (kích thước của dữ liệu vào) thì thời gian thực hiện T của một giải thuật phải được biểu diễn như một hàm của n : $T(n)$.

Các kiểu lệnh và tốc độ xử lý của máy tính, ngôn ngữ viết chương trình và chương trình dịch ngôn ngữ ấy đều ảnh hưởng tới thời gian thực hiện, nhưng những yếu tố này không đồng đều với mọi loại máy trên đó cài đặt giải thuật, vì vậy không thể dựa vào chúng khi xác lập $T(n)$. Điều đó cũng có nghĩa là $T(n)$ không thể được biểu diễn thành đơn vị thời gian bằng giây, bằng phút... được. Tuy nhiên, không phải vì thế mà không thể so sánh được các giải thuật về mặt tốc độ. Nếu như thời gian thực hiện của một giải thuật là $T_1(n) = cn^2$ và thời gian thực hiện của một giải thuật khác là $T_2(n) = kn$ (với c và k là một hằng số nào đó), thì khi n khá lớn, thời gian thực hiện giải thuật T_2 rõ ràng ít hơn so với giải thuật T_1 . Và như vậy thì nếu nói thời gian thực hiện giải thuật $T(n)$ tỉ lệ với n^2 hay tỉ lệ với n cũng cho ta ý niệm về tốc độ thực hiện giải thuật đó khi n khá lớn (với n nhỏ thì việc xét $T(n)$ không có ý nghĩa). Cách đánh giá thời gian thực hiện giải thuật độc lập với máy tính và các yếu tố liên quan tới máy như vậy sẽ dẫn tới khái niệm về “cấp độ lớn của thời gian thực hiện giải thuật” hay còn gọi là “*độ phức tạp tính toán của giải thuật*”.

2.2.2.1. Độ phức tạp tính toán của giải thuật

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

$$T(n) = O(n^2) \text{ (ký hiệu chữ O lớn)}$$

Một cách tổng quát có thể định nghĩa:

Một hàm $f(n)$ được xác định là $O(g(n))$

$$f(n) = O(g(n))$$

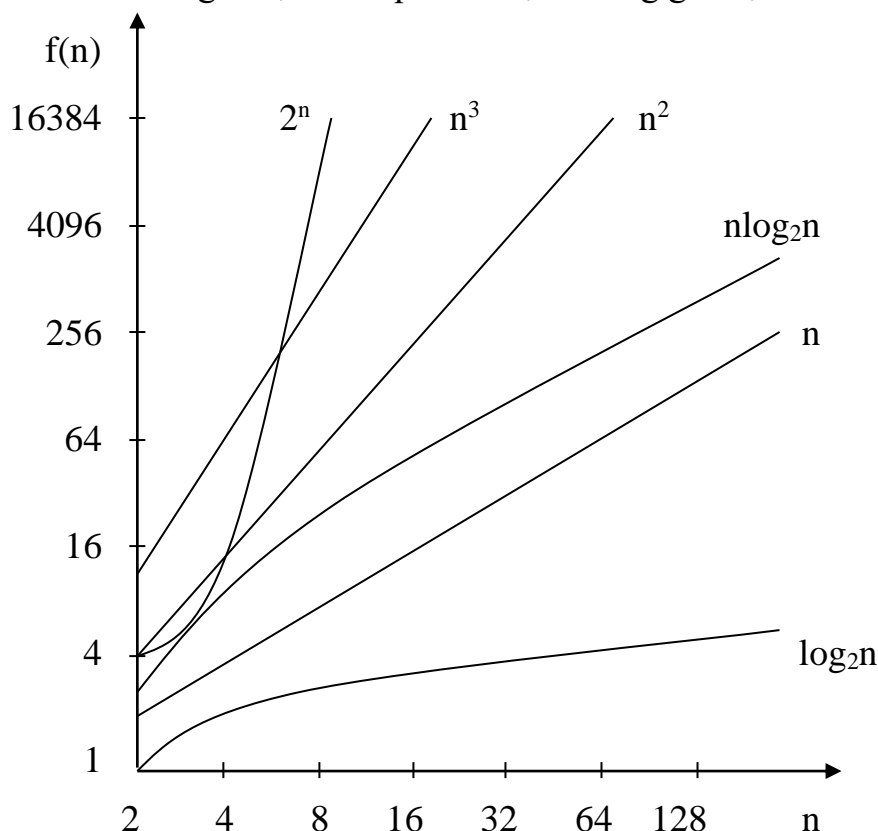
và được gọi là có cấp $g(n)$ nếu tồn tại các hằng số c và n_0 sao cho:

$$f(n) \leq cg(n) \text{ khi } n \geq n_0$$

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

Thông thường ta chọn $f(n)$ là các hàm đơn giản để biểu diễn độ phức tạp tính toán của giải thuật như: $\log_2 n$, n , $n \log_2 n$, n^2 , n^3 , 2^n , $n!$, n^n .

Chúng được sắp xếp theo thứ tự tăng dần. Ta có thể thấy rõ sự tăng trưởng giá trị của một số hàm theo giá trị của n qua đồ thị và bảng giá trị dưới đây:



Hình 2.7. Đồ thị tăng trưởng giá trị của một số hàm số

Bảng 2.2. Bảng tăng trưởng giá trị của một số hàm số

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	2.147.483.648

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

2.2.2.2. Xác định độ phức tạp tính toán

Xác định độ phức tạp tính toán của một giải thuật bất kỳ có thể dẫn tới những bài toán phức tạp. Tuy nhiên các kỹ thuật đưa ra trong mục này cho phép đánh giá được thời gian chạy của hầu hết các giải thuật mà ta gặp trong thực tế.

Quy tắc tổng: Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P_1 và P_2 mà $T_1(n) = O(f(n))$; $T_2(n) = O(g(n))$, thì thời gian thực hiện P_1 rồi P_2 tiếp theo sẽ là: $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$.

Ví dụ: trong một chương trình có 3 bước thực hiện mà thời gian thực hiện từng bước lần lượt là $O(n^2)$, $O(n^3)$ và $O(n \log_2 n)$ thì thời gian thực hiện 2 bước đầu là $O(\max(n^2, n^3)) = O(n^3)$. Thời gian thực hiện chương trình sẽ là: $O(\max(n^3, n \log_2 n)) = O(n^3)$.

Một ứng dụng khác của quy tắc này là nếu $g(n) \leq f(n)$ với mọi $n \geq n_0$ thì $O(f(n) + g(n))$ cũng là $O(f(n))$. Chẳng hạn: $O(n^4 + n^2) = O(n^4)$ và $O(n + \log_2 n) = O(n)$.

Phần chứng minh quy tắc tổng được xem như bài tập.

2.2.2.3. Thời gian chạy của các câu lệnh

Các giải thuật được đưa ra trong giáo trình này được trình bày dưới dạng “ngôn ngữ tựa C”. Dựa vào qui tắc tổng, việc đánh giá thời gian chạy của giải thuật được quy về đánh giá thời gian chạy của từng câu lệnh.

1) Lệnh gán

Lệnh gán có dạng: $X = \langle \text{biểu thức} \rangle$

Thời gian chạy của lệnh gán là thời gian thực hiện biểu thức. Trường hợp hay gặp nhất là biểu thức chỉ chứa các phép toán sơ cấp, và nó có thời gian thực hiện bằng hằng số c , nên được đánh giá là $O(1)$. Nếu biểu thức chứa các lời gọi hàm thì ta phải tính đến thời gian thực hiện hàm, và do đó trong trường hợp này thời gian thực hiện biểu thức có thể không phải là $O(1)$.

2) Lệnh lựa chọn

Lệnh lựa chọn if-else có dạng:

```
if (điều kiện)
    <lệnh 1>;
else
    <lệnh 2>;
```

Trong đó, điều kiện là một biểu thức cần được đánh giá, nếu điều kiện đúng thì lệnh 1 được thực hiện, nếu không thì lệnh 2 được thực hiện. Giả sử thời gian đánh giá điều kiện là $T_0(n)$, thời gian thực hiện lệnh 1 là $T_1(n)$, thời gian thực hiện lệnh 2 là $T_2(n)$. Khi đó thời gian thực hiện lệnh lựa chọn if-else sẽ là thời gian lớn nhất trong các thời gian: $T_0(n) + T_1(n)$ và $T_0(n) + T_2(n)$.

Trường hợp hay gặp là *thời gian kiểm tra điều kiện* chỉ cần bằng hằng số c , nên cũng được đánh giá là $O(1)$. Khi đó nếu $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ và $f(n)$ tăng nhanh hơn $g(n)$ thì thời gian chạy của lệnh if-else là $O(f(n))$; còn nếu $g(n)$ tăng nhanh hơn $f(n)$ thì lệnh if-else cần thời gian $O(g(n))$.

Thời gian chạy của lệnh lựa chọn switch được đánh giá tương tự như lệnh if-else, chỉ cần lưu ý rằng, lệnh if-else có hai khả năng lựa chọn, còn lệnh switch có thể có nhiều hơn hai khả năng lựa chọn.

3) Các câu lệnh lặp

Các câu lệnh lặp gồm: for, while, do-while

Để đánh giá thời gian thực hiện một câu lệnh lặp, trước hết ta cần đánh giá số tối đa các lần lặp, giả sử đó là $L(n)$. Sau đó đánh giá thời gian chạy của mỗi lần lặp, chú ý rằng thời gian thực hiện thân của một lệnh lặp ở các lần lặp khác nhau có thể khác nhau, giả sử thời gian thực hiện thân lệnh lặp ở lần thứ i ($i=1,2,\dots, L(n)$) là $T_i(n)$. Mỗi lần lặp, chúng ta cần kiểm tra điều kiện lặp, giả sử thời gian kiểm tra là $T_0(n)$. Như vậy thời gian chạy của lệnh lặp là:

$$\sum_{i=1}^{L(n)} (T_0(n) + T_i(n))$$

Công đoạn khó nhất trong đánh giá thời gian chạy của một lệnh lặp là đánh giá số lần lặp. Trong nhiều lệnh lặp, đặc biệt là trong các lệnh lặp for, ta có thể thấy ngay số lần lặp tối đa là bao nhiêu. Nhưng cũng không ít các lệnh lặp, từ điều kiện lặp để suy ra số tối đa các lần lặp, ta cần phải tiến hành các suy diễn không đơn giản.

Trường hợp hay gặp là: kiểm tra điều kiện lặp (thông thường là đánh giá một biểu thức) chỉ cần thời gian $O(1)$, thời gian thực hiện các lần lặp là như nhau và giả sử ta đánh giá được là $O(f(n))$; khi đó, nếu đánh giá được số lần lặp là $O(g(n))$, thì thời gian chạy của lệnh lặp là $O(g(n).f(n))$.

Ví dụ 1: Giả sử ta có mảng A các số thực, cỡ n và ta cần tìm xem mảng có chứa số thực x không. Điều đó có thể thực hiện bởi giải thuật tìm kiếm tuần tự như sau:

- (1) $i = 0;$
- (2) while ($i < n \ \&\& \ x \neq A[i]$)
- (3) $i++;$

Lệnh gán (1) có thời gian chạy là $O(1)$. Lệnh lặp (2)-(3) có số tối đa các lần lặp là n, đó là trường hợp x chỉ xuất hiện ở thành phần cuối cùng của mảng $A[n-1]$ hoặc x không có trong mảng. Thân của lệnh lặp là lệnh (3) có thời gian chạy $O(1)$. Do đó, lệnh lặp có thời gian chạy là $O(n)$. Giải thuật gồm lệnh gán và lệnh lặp với thời gian là $O(1)$ và $O(n)$, nên thời gian chạy của nó là $O(n)$.

Ví dụ 2: Giải thuật tạo ra ma trận đơn vị A cấp n;

- (1) for ($i = 0; i < n; i++$)
- (2) for ($j = 0; j < n; j++$)
- (3) $A[i][j] = 0;$
- (4) for ($i = 0; i < n; i++$)
- (5) $A[i][i] = 1;$

Giải thuật gồm hai lệnh lặp for. Lệnh lặp for đầu tiên (các dòng (1)-(3)) có thân lại là một lệnh lặp for ((2)-(3)). Số lần lặp của lệnh for ((2)-(3)) là n, thân của nó là lệnh (3) có thời gian chạy là $O(1)$, do đó thời gian chạy của lệnh lặp for này là $O(n)$. Lệnh lặp for ((1)-(3)) cũng có số lần lặp là n, thân của nó có thời gian đã

đánh giá là $O(n)$, nên thời gian của lệnh lặp for ((1)-(3)) là $O(n^2)$. Tương tự lệnh for ((4)-(5)) có thời gian chạy là $O(n)$. Sử dụng qui tắc tổng, ta suy ra thời gian chạy của giải thuật là $O(n^2)$.

Chú ý 1: Dựa vào quy tắc tổng khi đánh giá thời gian thực hiện giải thuật, ta chỉ cần chú ý tới các bước tương ứng với một phép toán mà ta gọi là phép toán tích cực (active operation), đó là phép toán thuộc giải thuật mà thời gian thực hiện nó không ít hơn thời gian thực hiện các phép khác (tất nhiên phép toán tích cực có thể không phải là duy nhất) hay nói một cách khác: số lần thực hiện nó không kém gì các phép khác.

Ví dụ: Xét giải thuật tính giá trị của e^x theo công thức gần đúng:

$$e^x \approx 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!} \text{ với } x \text{ và } n \text{ cho trước}$$

EXP1()

```
{
    //tính từng số hạng rồi cộng lại
    scanf(x);
    S = 1;
    for (i = 1; i <= n; i++)
    {
        p = 1;
        for (j = 1; j < i; j++)
            p = p*x/j;
        S = S + p;
    }
}
```

Ta thấy phép toán tích cực ở đây là phép gán:

$$p = p*x/j$$

Ta thấy nó được thực hiện:

$$1 + 2 + \dots + n = \frac{n(n+1)}{2} \text{ lần}$$

Vì vậy, thời gian thực hiện giải thuật này được đánh giá là $T(n) = O(n^2)$

Ta có thể viết giải thuật theo một cách khác.

EXP2()

{/*dựa vào số hạng trước để tính số hạng sau

$$\text{theo cách } \frac{x^2}{2!} = \frac{x}{1!} \times \frac{x}{2}; \dots; \frac{x^n}{n!} = \frac{x^{n-1}}{(n-1)!} \times \frac{x}{n} \quad */$$

```
scanf(x);
```

```
S = 1;
```

```
p = 1;
```

```
for (i = 1; i < n; i++)
```

```
{
```

```
    p = p*x/i;
```

```
    S = S + p;
```

```
}
```

```
}
```

Bây giờ thời gian thực hiện lại là:

$$T(n) = O(n)$$

Vì phép gán $p = p*x/i$ chỉ thực hiện n lần.

Tại đây ta thấy phép gán $S = S + p$ cũng được coi là phép toán tích cực vì có số lần thực hiện bằng với phép gán $p = p*x/i$, như vậy đoạn chương trình trên có đến hai phép toán tích cực.

Chú ý 2: Có những trường hợp thời gian thực hiện giải thuật không phải chỉ phụ thuộc vào kích thước của dữ liệu vào mà còn phụ thuộc vào chính tình trạng của dữ liệu đó nữa.

Chẳng hạn: Khi sắp xếp một dãy số theo thứ tự tăng dần, nếu gặp dãy số đưa vào đã có đúng thứ tự sắp xếp rồi thì sẽ khác với trường hợp dãy số đưa vào chưa có thứ tự hoặc có thứ tự ngược lại. Lúc đó khi phân tích thời gian thực hiện giải thuật ta sẽ phải xét tới: đối với mọi dữ liệu vào có kích thước n thì $T(n)$ trong trường hợp thuận lợi nhất sẽ thế nào? Rồi $T(n)$ trong trường hợp xấu nhất? và $T(n)$ trung bình? Việc xác định $T(n)$ trung bình thường khó vì sẽ phải dùng tới những công cụ toán đặc biệt, hơn nữa tính trung bình có thể có nhiều cách quan niệm. Trong các trường hợp mà $T(n)$ trung bình khó xác định người ta thường đánh giá giải thuật qua giá trị xấu nhất của $T(n)$. Để thể thấy rõ hơn ta xét giải thuật sau:

SEARCH()

```
/* Cho mảng A có n phần tử, có chỉ số từ 0 đến n-1, giải thuật này thực
   hiện tìm trong A một phần tử có giá trị bằng x cho trước */
Found = 0; //Found là biến kiểm tra để ngừng tìm khi đã thấy
i = 0;
while (i < n && !Found)
    if (A[i] == x)
    {
        Found = 1; //true
        k = i;
        printf(k);
    }
    else
        i = i + 1;
}
```

Ta coi phép toán tích cực ở đây là phép so sánh $A[i]$ với x . Có thể thấy số lần phép toán tích cực này thực hiện phụ thuộc vào chỉ số i mà $A[i] = x$.

Trường hợp thuận lợi nhất xảy ra khi x bằng $A[0]$: một lần thực hiện.

Trường hợp xấu nhất: khi x bằng $A[n-1]$ hoặc không tìm thấy: n lần thực hiện.

Vậy: $T_{\text{tốt}} = O(1)$

$T_{\text{xấu}} = O(n)$

Còn thời gian trung bình sẽ được đánh giá thế nào?

Muốn trả lời ta phải biết được xác suất mà x rơi vào một phần tử nào đó của A . Nếu ta giả thiết khả năng này là đồng đều với mọi phần tử của A (đồng khả năng) thì có thể xét như sau:

Gọi q là xác suất để x rơi vào một phần tử nào đó của A , thì xác suất để x rơi vào phần tử là $A[i] = p_i = q/n$.

Còn xác suất để x không rơi vào phần tử nào (không tìm thấy) sẽ là $1-q$.

Khi đó, thời gian thực hiện trung bình sẽ là:

$$\begin{aligned}
T_{tb}(n) &= \sum_{i=1}^n p_i i + (1-q)n \\
&= \sum_{i=1}^n \frac{q}{n} i + (1-q)n \\
&= \frac{q}{n} \frac{n(n+1)}{2} + (1-q)n \\
&= q \frac{(n+1)}{2} + (1-q)n
\end{aligned}$$

Nếu $q = 1$ (nghĩa là luôn tìm thấy)

$$\text{thì } T_{tb}(n) = \frac{n+1}{2}$$

Nếu $q=1/2$ (khả năng tìm thấy và không tìm thấy bằng nhau)

$$\text{thì } T_{tb}(n) = \frac{n+1}{4} + \frac{n}{2} = \frac{3n+1}{4}$$

Nói chung $T_{tb}(n) = O(n)$

Nếu ta lấy trường hợp xấu nhất để đánh giá thì thấy cũng là $O(n)$.

Câu hỏi và bài tập chương 2

2.1. Việc chia bài toán ra thành các bài toán nhỏ có những thuận lợi gì?

2.2. Nêu nguyên tắc của phương pháp thiết kế từ đỉnh xuống (thiết kế kiểu top-down). Cho ví dụ minh họa.

2.3. Người ta có thể thực hiện giải bài toán theo kiểu từ đáy lên (thiết kế kiểu bottom-up) nghĩa là đi từ các vấn đề cụ thể trước, sau đó mới ghép chúng lại thành một vấn đề lớn hơn. Cho ví dụ thực tế thể hiện cách thiết kế này và thử nhận xét so sánh nó với cách thiết kế kiểu Top-down.

2.4. Tóm tắt ý chủ đạo của phương pháp tính chính từng bước.

2.5. Có 6 đội bóng: A, B, C, D, E, F thi đấu để tranh giải vô địch (vòng đầu)

Đội A đã đấu với B và C

Đội B đã đấu với D và F

Đội E đã đấu với C và F

Mỗi đội chỉ đấu với đội khác 1 trận trong 1 tuần. Hãy lập lịch thi đấu sao cho các trận còn lại sẽ được thực hiện trong một số ít tuần nhất.

2.6. Hãy nêu một giải thuật mà độ phức tạp tính toán của nó là $O(1)$.

2.7. Với các đoạn chương trình dưới đây hãy xác định độ phức tạp tính toán của giải thuật bằng ký pháp chữ O lớn, trong trường hợp xấu nhất.

- a) Sum = 0;
 for (i = 1; i <=n; i++)
 {
 scanf (x);
 Sum = Sum + x;
 }
- b) for (i = 1; i <=n; i++)
 for (j = 1; j <=n; j++)
 {
 c[i][j] = 0;
 for (k = 1; k <=n; k++)
 c[i][j] = c[i][j] + a[i][k]*b[k][j]
 }
- c) for (i = 1; i <n; i++)
 for (j = i; j <n; j++)
 if (x[j] > x[j+1])
 {
 Temp = x[j];
 x[j] = x[j+1];
 x[j+1] = Temp;
 }

CHƯƠNG 3

ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY

(Recursive algorithm)

3.1. Khái niệm đệ quy

Ta nói một đối tượng là đệ quy nếu nó bao gồm chính nó như một bộ phận hoặc nó được định nghĩa dưới dạng của chính nó.

Ví dụ:

- Trên tivi trong những chương trình ca nhạc được truyền hình trực tiếp mà giữa sân khấu có đặt tivi màn ảnh rộng, có lúc ta thấy có những hình ảnh đệ quy đó là: ca sĩ đứng hát trước màn hình tivi, trên màn hình này lại có chính hình ảnh của ca sĩ ấy đứng hát trước màn hình tivi nhưng kích thước thì nhỏ hơn và cứ như thế...

- Trong toán học ta cũng hay gặp các định nghĩa đệ quy như:

1) Định nghĩa số tự nhiên:

a) 1 là một số tự nhiên.

b) x là số tự nhiên nếu $x-1$ là số tự nhiên.

2) Định nghĩa hàm n giai thừa: $n!$

a) $0! = 1$

b) Nếu $n > 0$ thì $n! = n(n-1)!$

3.2. Giải thuật đệ quy

Nếu lời giải của một bài toán T được thực hiện bằng lời giải của một bài toán T' , có dạng giống như T , thì đó là một lời giải đệ quy. Giải thuật tương ứng với lời giải như vậy gọi là *giải thuật đệ quy*.

Mới nghe thì có vẻ hơi lạ, nhưng điểm mấu chốt cần lưu ý là: T' tuy có dạng giống như T , nhưng theo một nghĩa nào đó, nó phải “nhỏ” hơn T .

Hãy xét bài toán tìm một từ trong một quyển từ điển. Có thể nêu giải thuật như sau:

if (từ điển là một trang)

 tìm từ trong trang này;

```

else
{
    Mở từ điển vào trang “giữa”;
    Xác định xem nửa nào của từ điển chứa từ cần tìm;
    if (từ đó nằm ở nửa trước của từ điển)
        tìm từ đó trong nửa trước;
    else
        tìm từ đó trong nửa sau;
}

```

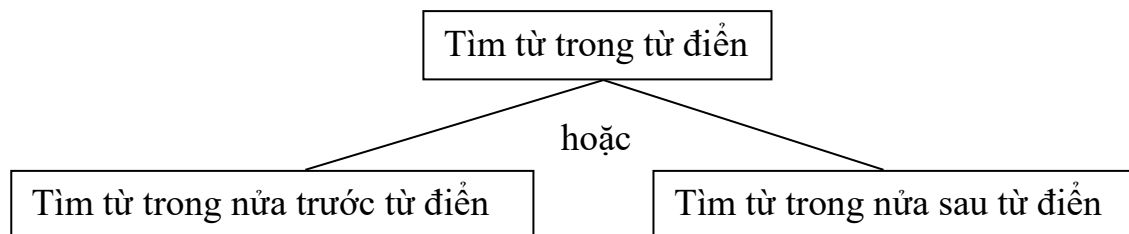
Tất nhiên giải thuật trên mới chỉ được nêu dưới dạng thô, còn nhiều chỗ chưa cụ thể, chẳng hạn:

Tìm từ trong một trang thì làm thế nào?

Thế nào là mở từ điển vào trang giữa?

Làm thế nào để biết từ đó nằm ở nửa nào của từ điển?...

Trả lời rõ những câu hỏi trên không phải là khó, nhưng ta sẽ không sa vào các chi tiết này mà muốn tập trung vào việc xét chiến thuật của lời giải. Có thể hình dung chiến thuật tìm kiếm này một cách khái quát như hình 3.1.



Hình 3.1. Giải thuật tìm từ trong từ điển

Ta thấy có hai điểm chính cần lưu ý:

a) Sau mỗi lần từ điển được tách đôi thì một nửa thích hợp sẽ lại được tìm kiếm bằng một chiến thuật như đã dùng trước đó.

b) Có một trường hợp đặc biệt, khác với mọi trường hợp trước, sẽ đạt được sau nhiều lần tách đôi, đó là trường hợp từ điển chỉ còn duy nhất một trang. Lúc đó việc tách đôi ngừng lại và bài toán đã trở thành đủ nhỏ để ta có thể giải quyết trực tiếp bằng cách tìm từ mong muốn trên trang đó chẳng hạn, bằng cách tìm tuần tự từ

trên xuống dưới. Trường hợp đặc biệt này được gọi là *trường hợp suy biến* (degenerate case).

Có thể coi đây là một chiến thuật “chia để trị” (divide and conquer). Bài toán được tách thành bài toán nhỏ hơn và bài toán nhỏ hơn lại được giải quyết với chiến thuật chia để trị như trước, cho tới khi xuất hiện trường hợp suy biến.

Bây giờ ta hãy thể hiện giải thuật tìm kiếm này dưới dạng một hàm.

SEARCH (dict, word)

```
{/* dict được coi là đầu mỗi để truy nhập được vào từ điển đang xét
```

```
word chỉ từ cần tìm */
```

```
if (từ điển chỉ còn là một trang)
```

```
    tìm từ word trong trang này;
```

```
else
```

```
{
```

```
    Mở từ điển vào trang “giữa”;
```

```
    Xác định xem nửa nào của từ điển chứa từ word;
```

```
    if (word nằm ở nửa trước của từ điển)
```

```
        SEARCH (dict 1, word);
```

```
    else
```

```
        SEARCH (dict 2, word);
```

```
}
```

```
// dict 1 và dict 2 là đầu mỗi để truy nhập được vào nửa trước
```

```
// và nửa sau của từ điển
```

```
}
```

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

1) Trong hàm đệ quy có lời gọi đến chính hàm đó. Ở đây: trong hàm SEARCH có gọi SEARCH.

2) Mỗi lần có lời gọi lại hàm thì kích thước của bài toán đã thu nhỏ hơn trước. Ở đây khi gọi lại SEARCH thì kích thước từ điển chỉ còn bằng một “nửa” trước đó.

3) Có một trường hợp đặc biệt: trường hợp suy biến. Đó chính là trường hợp mà từ điển chỉ còn một trang. Khi trường hợp này xảy ra thì bài toán còn lại sẽ được giải quyết theo một cách khác hẳn và gọi đệ quy cũng kết thúc. Chính tình trạng kích thước của bài toán cứ giảm dần sẽ đảm bảo dẫn tới trường hợp suy biến.

Một số ngôn ngữ bậc cao như: PASCAL, C/C++... cho phép viết các hàm đệ quy. Nếu hàm chứa lời gọi đến chính nó, như hàm SEARCH ở trên thì nó được gọi là *đệ quy trực tiếp* (directly recursive). Cũng có dạng hàm chứa lời gọi đến hàm khác mà ở hàm này lại chứa lời gọi đến nó. Trường hợp này gọi là *đệ quy gián tiếp* (indirectly recursive).

3.3. Thiết kế một số giải thuật đệ quy

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

Ta có thể thấy điều này qua một số bài toán sau:

3.3.1. Hàm tính $n!$

Định nghĩa đệ quy của $n!$ có thể nhắc lại như sau:

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{Factorial}(n-1) & \text{if } n > 0 \end{cases}$$

Giải thuật đệ quy được viết dưới dạng hàm như sau:

```
FACTORIAL(n)
{
    if (n == 0) // Trường hợp suy biến, kết thúc gọi đệ quy
        return 1;
    else // Trường hợp tổng quát, gọi đệ quy
        return n*FACTORIAL(n-1);
}
```

Đối chiếu với 3 đặc điểm của đệ quy nêu trên ta thấy:

- Lời gọi tới chính nó ở đây nằm trong câu lệnh return đứng sau else.
- Ở mỗi lần gọi đệ quy đến FACTORIAL, thì giá trị của n giảm đi 1. Ví dụ, FACTORIAL (4) gọi đến FACTORIAL (3), gọi đến FACTORIAL (2), gọi đến

FACTORIAL (1), gọi đến FACTORIAL (0), FACTORIAL (0) chính là trường hợp suy biến, nó được tính theo cách đặc biệt $\text{FACTORIAL}(0) = 1$.

3.3.2. Dãy số FIBONACCI

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

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

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

Ví dụ, với $n = 6$, ta thấy:

Tháng thứ 1: 1 cặp (cặp ban đầu)

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

Tháng thứ 3: 2 cặp (đã có thêm 1 cặp con)

Tháng thứ 4: 3 cặp (cặp đầu vẫn đẻ thêm)

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

Tháng thứ 6: 8 cặp (cặp con vẫn đẻ tiếp)

Bây giờ ta tính tới việc tính số cặp thỏ ở tháng thứ n : $F(n)$

Ta thấy nếu mỗi cặp thỏ ở tháng thứ $(n-1)$ đều sinh con thì:

$$F(n) = 2 \cdot (n-1).$$

Nhưng không phải như vậy. Trong các cặp thỏ ở tháng thứ $(n-1)$ chỉ có những cặp thỏ đã có ở tháng thứ $(n-2)$ mới sinh con ở tháng thứ n được thôi.

$$\text{Do đó: } F(n) = F(n-2) + F(n-1)$$

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

$$F(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ F(n-2) + F(n-1) & \text{if } n > 2. \end{cases}$$

Dãy số thể hiện $F(n)$ ứng với các giá trị của n như sau:

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

Dãy số $F(n)$ được gọi là *dãy số Fibonacci*. Nó là mô hình của rất nhiều hiện tượng tự nhiên và cũng được sử dụng nhiều trong tin học.

Sau đây là hàm đệ quy thể hiện giải thuật tính $F(n)$.

```
F(n)
{
    if (n <= 2)
        return 1;
    else
        return F(n-2) + F(n-1)
}
```

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

Chú ý:

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

Nhưng không phải lúc nào tính đệ quy trong cách giải bài toán cũng thể hiện rõ nét và đơn giản như vậy. Thế thì vấn đề gì cần lưu tâm khi thiết kế một giải thuật đệ quy? Có thể thấy câu trả lời qua việc giải đáp các câu hỏi sau:

1) Có thể định nghĩa được bài toán dưới dạng một bài toán cùng loại, nhưng “nhỏ” hơn, như thế nào?

2) Như thế nào là kích thước của bài toán được giảm đi ở mỗi lần gọi đệ quy?

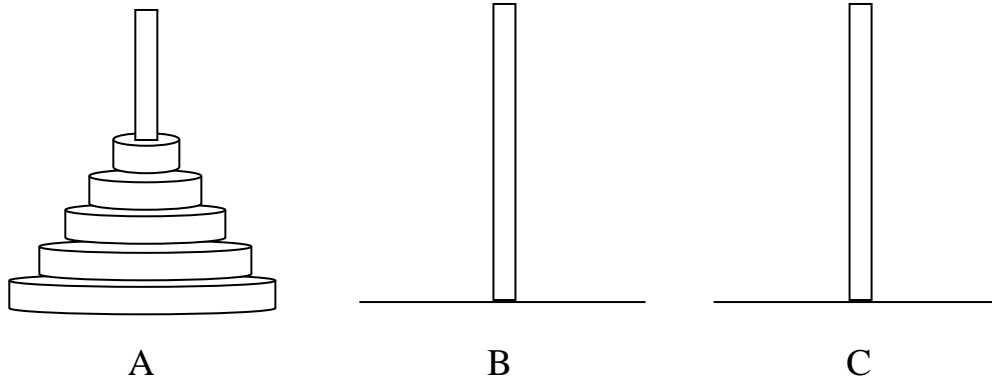
3) Trường hợp đặc biệt nào của bài toán sẽ được coi là trường hợp suy biến?

Sau đây ta xét thêm một số bài toán phức tạp hơn:

3.3.3. Bài toán “Tháp Hà Nội” (Tower of Hanoi)

Đây là một bài toán mang tính chất một trò chơi, nội dung như sau:

Có n đĩa, kích thước nhỏ dần, đĩa có lỗ ở giữa (như đĩa hát). Có thể xếp chồng chúng lên nhau xuyên qua một cọc, to dưới nhỏ trên để cuối cùng có một chồng đĩa như hình tháp (hình 3.2).



Hình 3.2. Hình ảnh ban đầu bài toán chuyển tháp

Yêu cầu đặt ra là:

Chuyển chồng đĩa từ cọc A sang cọc khác, chẳng hạn sang cọc C, theo những điều kiện:

- 1- Mỗi lần chỉ được chuyển một đĩa.
- 2- Không khi nào có tình huống đĩa to ở trên đĩa nhỏ (dù là tạm thời).
- 3- Được phép sử dụng một cọc trung gian, chẳng hạn cọc B để đặt tạm đĩa.

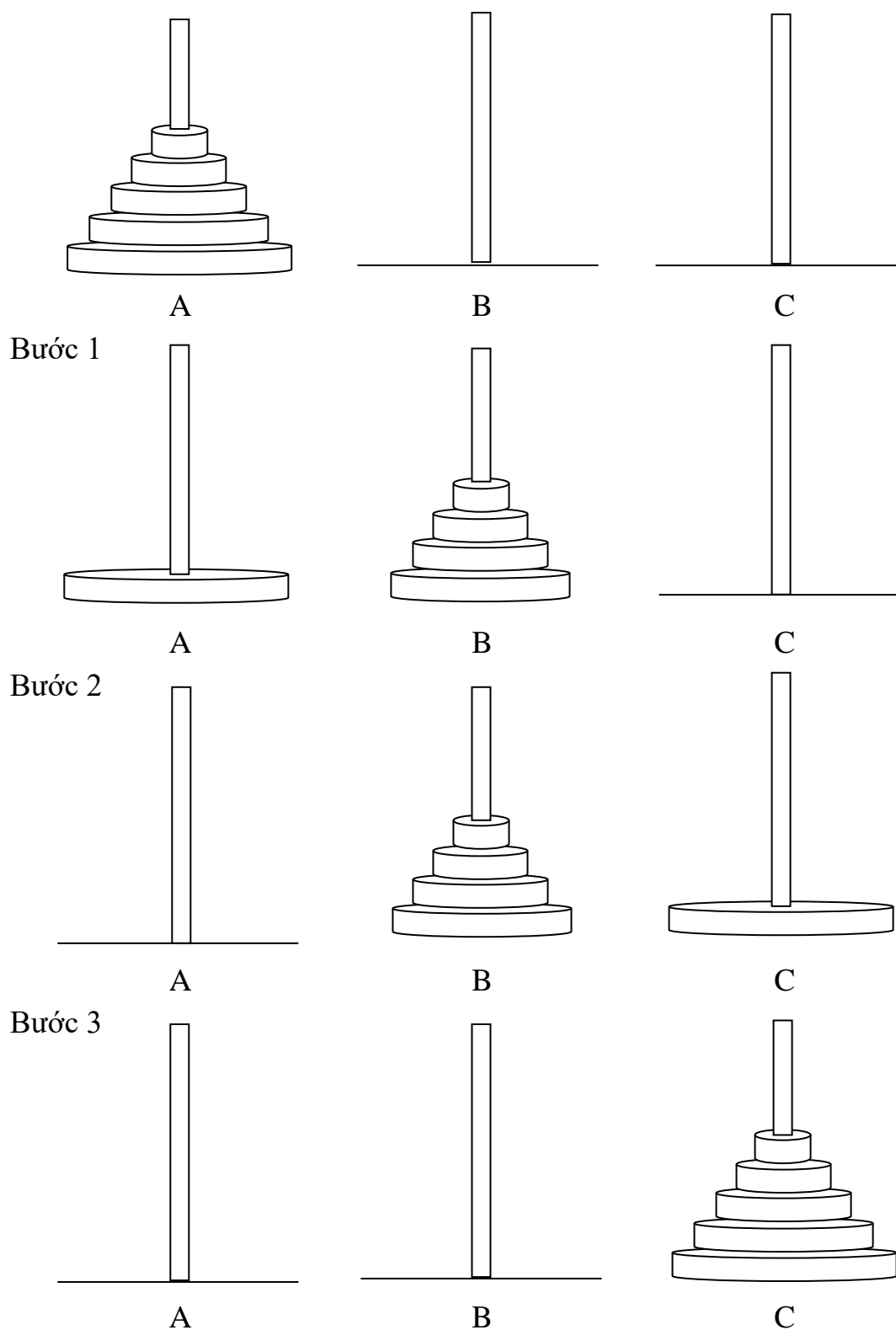
Để đi tới cách giải tổng quát, trước hết xét vài trường hợp đơn giản sau:

- Trường hợp một đĩa:
 - Chuyển đĩa từ cọc A sang cọc C.
- Trường hợp hai đĩa:
 - Chuyển đĩa thứ nhất từ cọc A sang cọc B.
 - Chuyển đĩa thứ hai từ cọc A sang cọc C.
 - Chuyển đĩa thứ nhất từ cọc B sang cọc C.

Ta thấy với trường hợp n đĩa ($n > 2$) nếu coi $(n-1)$ đĩa ở trên, đóng vai trò như đĩa thứ nhất thì có thể xử lý giống như trường hợp 2 đĩa được, nghĩa là:

- Chuyển $(n-1)$ đĩa trên từ cọc A sang cọc B.
- Chuyển đĩa thứ n từ cọc A sang cọc C.
- Chuyển $(n-1)$ đĩa từ cọc B sang cọc C.

Lược đồ thể hiện 3 bước này như sau:



Hình 3.3. Các bước chuyển tháp

Như vậy, bài toán “Tháp Hà Nội” tổng quát với n đĩa đã được dẫn đến bài toán tương tự với kích thước nhỏ hơn, chẳng hạn từ chỗ chuyển n đĩa từ cọc A sang cọc C nay là chuyển $(n-1)$ đĩa từ cọc A sang cọc B. Và ở mức này thì giải thuật lại là:

Chuyển (n-2) đĩa trên từ cọc A sang cọc C.

Chuyển đĩa thứ n-1 từ cọc A sang cọc B.

Chuyển (n-2) đĩa từ cọc C sang cọc B.

Và cứ như thế cho tới khi trường hợp suy biến xảy ra, đó là trường hợp ứng với bài toán chỉ chuyển 1 đĩa.

Vậy thì các đặc điểm của đệ quy trong giải thuật đã được xác định và ta có thể viết giải thuật đệ quy của bài toán “Tháp Hà Nội” như sau:

```
Thap_Ha_Noi (n, A, B, C)
{
    if (n == 1)
        printf("Chuyển đĩa 1 từ A sang C");
    else
    {
        Thap_Ha_Noi (n-1, A, C, B);
        printf("Chuyển đĩa n từ A sang C");
        Thap_Ha_Noi (n-1, B, A, C);
    }
}
```

3.3.4. Bài toán 8 quân hậu và giải thuật quay lui (Back tracking)

Như ta đã biết, bàn cờ quốc tế là một bảng hình vuông gồm có 8 hàng, 8 cột. Quân hậu là một quân cờ có thể ăn được bất kỳ quân nào nằm trên cùng một hàng, cùng một cột hay cùng một đường chéo. Bài toán đặt ra là: hãy xếp 8 quân hậu trên bàn cờ sao cho không có quân hậu nào có thể ăn được quân hậu nào. Điều đó có nghĩa là trên mỗi hàng, mỗi cột, mỗi đường chéo chỉ có thể có một quân hậu thôi.

Dĩ nhiên ta không nên tìm lời giải bằng cách xét mọi trường hợp ứng với mọi vị trí của 8 quân hậu trên bàn cờ, rồi lọc ra các trường hợp chấp nhận được. Khuynh hướng “thử từng bước”, mới nghe có vẻ hơi lạ, nhưng lại thể hiện một giải pháp hiện thực: nó cho phép tìm ra tất cả các cách sắp xếp để không có quân hậu nào ăn được nhau (số lượng cách sắp xếp này là 92).

Nét đặc trưng của phương pháp này là ở chỗ các bước đi tới lời giải hoàn toàn được làm thử. Nếu có một lựa chọn được chấp nhận thì ghi nhớ các thông tin cần thiết và tiến hành bước thử tiếp theo. Nếu trái lại không có một lựa chọn nào thích

hợp cả thì làm lại bước trước, xoá bớt các ghi nhớ và quay về chu trình thử với các lựa chọn còn lại. Hành động này được gọi là *quay lui*, và các giải thuật thể hiện phương pháp này gọi là các *giải thuật quay lui*.

Đối với bài toán 8 quân hậu: do mỗi cột chỉ có thể có một quân hậu nên lựa chọn đối với quân hậu thứ j , ứng với cột j , là đặt nó vào hàng nào để đảm bảo “an toàn” nghĩa là không cùng hàng, cùng đường chéo với $(j-1)$ quân hậu đã xếp trước đó. Rõ ràng để đi tới các lời giải ta phải thử tất cả các trường hợp sắp xếp quân hậu đầu tiên tại cột 1. Với mỗi vị trí như vậy ta lại phải giải quyết bài toán 7 quân hậu với phần còn lại của bàn cờ, nghĩa là ta đã “quay lại bài toán cũ” nhưng với quy mô nhỏ hơn. Ta có thể phác thảo hàm thử như sau:

```
Try(j)
{
    Khởi phát việc chọn vị trí cho quân hậu thứ j;
    do
    {
        Thực hiện phép chọn tiếp theo;
        if (an toàn)
        {
            Đặt quân hậu; // Ghi nhớ các thông tin cần thiết
            if (j<8)
            {
                Try(j+1);
                if (không thành công)
                    cất quân hậu; // Xoá bớt các ghi nhớ
            }
        }
    }
    while (chưa thành công và còn chỗ);
}
```

Để đi tới một hàm chi tiết hơn ta phải chuẩn bị dữ liệu biểu diễn các thông tin cần thiết bao gồm:

Dữ liệu biểu diễn nghiệm.

Dữ liệu biểu diễn lựa chọn.

Dữ liệu biểu diễn điều kiện.

Như trên đã nêu, đối với quân hậu thứ j , vị trí của nó chỉ chọn trong cột thứ j . Vậy tham biến j trở thành chỉ số cột và việc chọn lựa được tiến hành trên 8 giá trị của chỉ số hàng i .

Để lựa chọn i được chấp nhận, thì hàng i và hai đường chéo qua ô (i, j) phải còn tự do (không có quân hậu nào khác ở trên đó). Chú ý rằng trong hai đường chéo thì đường chéo theo chiều \uparrow có các ô $(i;j)$ mà tổng $i+j$ không đổi, còn đường chéo theo chiều \downarrow có các ô $(i;j)$ mà $i-j$ không đổi (hình 3.4).

Do đó ta sẽ chọn các mảng nguyên một chiều để biểu diễn các tình trạng này:

$a[i] = 1$ (đúng) có nghĩa là chưa có quân hậu nào chiếm hàng i .

$b[i+j] = 1$ có nghĩa là chưa có quân hậu nào chiếm đường chéo $i+j$

$c[i-j] = 1$ có nghĩa là chưa có quân hậu nào chiếm đường chéo $i-j$

ta biết:

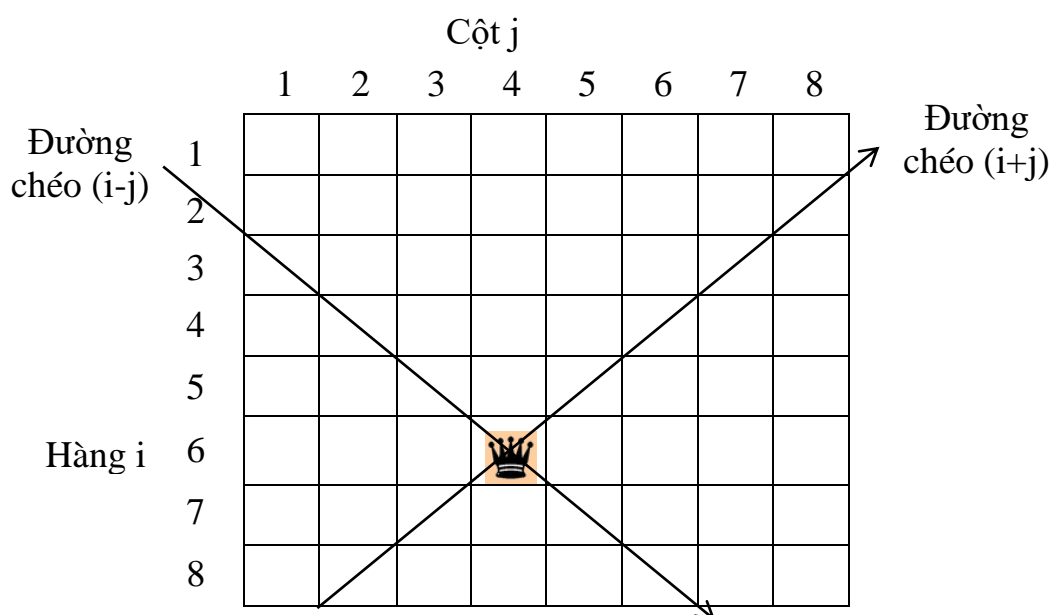
$$1 \leq i, j \leq 8$$

nên suy ra:

$$1 \leq i \leq 8$$

$$2 \leq i+j \leq 16$$

$$-7 \leq i-j \leq 7$$



Hình 3.4. Mô tả quân hậu ở vị trí: cột j , hàng i , đường chéo $i-j$, $i+j$

Như vậy điều kiện để lựa chọn i được chấp nhận là: $a[i] \&\& b[i+j] \&\& c[i-j]$ có giá trị 1 (đúng).

Ta dùng mảng nguyên một chiều x để ghi lại các lựa chọn được chấp nhận. Khi đó:

- Việc đặt quân hậu được thực hiện bởi:

$x[j] = i; a[i] = 0; b[i+j] = 0; c[i-j] = 0;$

- Còn cất quân hậu thì bởi:

$a[i] = 1; b[i+j] = 1; c[i-j] = 1;$

Hàm Try bây giờ có thể viết như sau:

Try (j)

```
{
    i = 0;
    do
    {
        i = i+1;
        q = 0; // Biến q để kiểm tra có thành công hay không
        if (a[i] && b[i+j] && c[i-j])
        {
            x[j] = i;
            a[i] = b[i+j] = c[i-j] = 0;
            if (j < 8)
            {
                Try(j+1);
                if (not q)
                {
                    a[i] = b[i+j] = c[i-j] = 1;
                }
            }
            else // j=8, đã tìm được một cách đặt cho 8 quân hậu
                q = 1; // ghi nhận thành công
        }
    }
```



```

    }
    while (q || i==8);
}

```

Với hàm Try này, chương trình cho một lời giải của bài toán 8 quân hậu như sau:

```

main()
{ // khởi tạo tình trạng ban đầu khi chưa quân hậu nào được đặt
  for (i = 1; i<=8; i++) a[i] = 1;
  for (i = 2; i<=16; i++) b[i] = 1;
  for (i = -7; i<=7; i++) c[i] = 1;
  Try(1); // tìm một lời giải
  if (q)
    for (i = 1; i<=8; i++) printf(x[i]);
}

```

Còn nếu ta muốn có tất cả các lời giải của bài toán thì cần sửa đổi đôi chút trong hàm Try như sau:

- Điều kiện kết thúc của quá trình chọn rút lại còn $i==8$, nên câu lệnh do while được thay đổi bởi câu lệnh for. Việc in kết quả được thực hiện ngay sau khi có một lời giải, vì vậy ta viết dưới dạng một hàm để gọi nó ngay trong hàm Try.

```

In_kqua(x)
{
  for (i = 1; i<=8; i++) printf(x[i]);
}

```

Như vậy, dạng mới của Try sẽ là:

```

Try(j)
{
  for (i = 1; i<=8; i++)
    if (a[i] && b[i+j] && c[i-j])
    {
      x [j] = i;

```

```

    a[i] = b[i+j] = c[i-j] = 0;
    if (j<8)
        Try(j+1);
    else
        In_kqua(x);
    a[i] = b[i+j] = c[i-j] = 1;
}
}

```

Và chương trình cho toàn bộ các lời giải của bài toán sẽ là:

```

main()
{
    for (i = 1; i<=8; i++) a[i] = 1;
    for (i = 2; i<=16; i++) b[i] = 1;
    for (i = -7; i<=7; i++) c[i] = 1;
    Try(1);
}

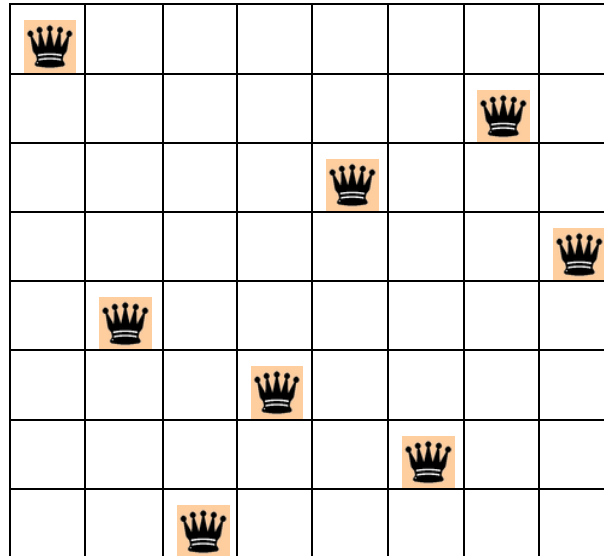
```

Bảng sau đây trình bày lời giải của 12 trong số 92 lời giải của bài toán:

Bảng 3.1. Lời giải của 10 trong số 92 lời giải của bài toán 8 quân hậu

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
1	5	8	6	3	7	2	4
1	6	8	3	7	4	2	5
1	7	4	6	8	2	5	3
1	7	5	8	2	4	6	3
2	4	6	8	3	1	7	5
2	5	7	1	3	8	6	4
2	5	7	4	1	8	6	3
2	6	1	7	4	8	3	5
2	6	8	3	1	4	7	5
2	7	3	6	8	5	1	4

Hình ảnh của nghiệm đầu tiên như sau:



Hình 3.5. Hình ảnh một cách đặt 8 quân hậu

3.4. Hiệu lực của đệ quy

Qua các ví dụ trên ta có thể thấy: đệ quy là một công cụ để giải các bài toán. Có những bài toán, bên cạnh giải thuật đệ quy vẫn có những giải thuật lặp khá đơn giản và hữu hiệu. Chẳng hạn giải thuật lặp tính $n!$ có thể viết:

```
Giai_thua(n)
{
    tg=1;
    For (i=1; i<=n; i++)
        tg=tg*i;
    return tg;
}
```

Hay giải thuật lặp tính số Fibonacci có thể viết:

```
Fibonacci(n)
{
    if (n <= 2)
        return 1;
    else
```

```

{
    Fib1 = 1; Fib2 = 1;
    For (i=3; i<=n; i++)
    {
        Fibn = Fib1 +Fib2;
        Fib1 = Fib2;
        Fib2 = Fibn;
    }
}

return Fibn;
}

```

Tuy vậy, đệ quy vẫn có vai trò xứng đáng của nó. Có những bài toán việc nghĩ ra lời giải đệ quy thuận lợi hơn nhiều so với lời giải lặp, và có những giải thuật đệ quy thực sự cũng có hiệu lực cao, chẳng hạn giải thuật sắp xếp nhanh (Quick sort) mà ta sẽ xét tới trong chương 6.

Một điều nữa cần nói thêm là: về mặt định nghĩa, công cụ đệ quy đã cho phép xác định một tập vô hạn các đối tượng bằng một phát biểu hữu hạn. Ta sẽ thấy vai trò của công cụ này trong định nghĩa văn phạm, định nghĩa cú pháp ngôn ngữ, định nghĩa một số cấu trúc dữ liệu ...

Chú thích: Khi thay các giải thuật đệ quy bằng các giải thuật không tự gọi chúng, như giải thuật lặp nêu trên, ta gọi là *khử đệ quy*.

3.5. Đệ quy và quy nạp toán học

Có một mối quan hệ khăng khít giữa đệ quy và quy nạp toán học. Cách giải đệ quy một bài toán dựa trên việc định rõ lời giải cho trường hợp suy biến, rồi thiết kế làm sao để lời giải của bài toán được suy từ lời giải của bài toán nhỏ hơn, cùng một loại như thế.

Tương tự như vậy, quy nạp toán học chứng minh một tính chất nào đó ứng với số tự nhiên cũng bằng cách chứng minh tính chất ấy đúng đối với một trường hợp cơ sở (thường ứng với $n = 1$), và rồi chứng minh tính chất ấy đúng với n bất kỳ, nếu nó đã đúng với các số tự nhiên nhỏ hơn n .

Do đó, ta sẽ không lấy làm lạ khi thấy quy nạp toán học được dùng để chứng minh các tính chất có liên quan đến giải thuật đệ quy. Chẳng hạn, sau đây ta sẽ chứng minh tính đúng đắn của giải thuật đệ quy FACTORIAL và biểu thức đánh giá số lần di chuyển đĩa trong giải thuật tháp HANOI.

1) Tính đúng đắn của giải thuật FACTORIAL:

Ta muốn chứng minh rằng hàm FACTORIAL(n) như đã nêu trên sẽ cho giá trị:

$$\text{FACTORIAL}(0) = 1$$

$$\text{FACTORIAL}(n) = n*(n-1)*\dots*2*1 \text{ (nếu } n>0\text{)}$$

Ta thấy trong hàm có chỉ thị:

if (n == 0) return 1

Chúng tỏ hàm đã đúng với $n = 0$ (cho ra giá trị bằng 1)

Giả sử nó đã đúng với $n = k$, nghĩa là với FACTORIAL(k) hàm cho ra giá trị bằng:

$$k*(k-1)*\dots*2*1$$

Bây giờ ta sẽ chứng minh nó đúng với $n = k+1$

Với $n = k+1 > 0$ chỉ thị đứng sau else

return $n * \text{FACTORIAL}(n-1)$

Sẽ được thực hiện. Vậy với $n = k+1$ thì giá trị cho ra là:

$$(k+1)*\text{FACTORIAL}(k)$$

nghĩa là $(k+1)*k*(k-1)*\dots*2*1$ và việc chứng minh đã hoàn tất.

2) Đánh giá giải thuật tháp Hà Nội

Ta xem xét với n đĩa thì số lần di chuyển đĩa sẽ là bao nhiêu? (ta coi phép di chuyển đĩa trong giải thuật này là phép toán tích cực).

Giả sử ta gọi Moves (n) là số đó, ta có Moves (1) = 1.

Khi $n > 1$ thì Moves (n) không thể tính trực tiếp như vậy, nhưng ta biết rằng: nếu biết Moves (n-1) thì ta sẽ tính được Moves (n):

$$\text{Moves}(n) = \text{Moves}(n-1) + \text{Moves}(1) + \text{Moves}(n-1) \text{ (dựa vào giải thuật)}$$

Vậy ta đã xác định được mối quan hệ truy hồi của cách tính:

$$\text{Moves}(1) = 1$$

$$\text{Moves}(n) = 2 * \text{Moves}(n-1) + 1 \text{ nếu } n > 1$$

Ví dụ: $\text{Moves}(3) = 2 * \text{Moves}(2) + 1$

$$= 2 * [2 * \text{Moves}(1) + 1] + 1$$

$$= 2 * [2 * 1 + 1] + 1$$

$$= 7$$

Tuy nhiên công thức truy hồi này không cho ta tính trực tiếp theo n được.
Nếu để ý ta thấy:

$$\text{Moves}(1) = 1 = 2^1 - 1$$

$$\text{Moves}(2) = 3 = 2^2 - 1$$

$$\text{Moves}(3) = 7 = 2^3 - 1$$

Vậy phải chăng: $\text{Moves}(n) = 2^n - 1$ với n là số tự nhiên bất kỳ?

Ta sẽ chứng minh công thức tính này là đúng bằng quy nạp toán học:

Với $n = 1$, $\text{Moves}(1) = 2^1 - 1 = 1$, công thức đã đúng.

Giả sử công thức đã đúng với $n = k$, nghĩa là:

$$\text{Moves}(k) = 2^k - 1$$

Với $n = k+1$ theo công thức truy hồi ta có:

$$\begin{aligned} \text{Moves}(k+1) &= 2 * \text{Moves}(k) + 1 \\ &= 2 * (2^k - 1) + 1 \\ &= 2^{k+1} - 2 + 1 = 2^{k+1} - 1 \end{aligned}$$

Như vậy công thức đã đúng với $k + 1$. Do đó có thể kết luận công thức vẫn đúng với mọi n .

Bài tập chương 3

3.1. Xét định nghĩa đệ quy:

$$\text{Acker}(m,n) = \begin{cases} n+1 & \text{nếu } m = 0 \\ \text{Acker}(m-1,1) & \text{nếu } n = 0 \\ \text{Acker}(m-1, \text{Acker}(m,n-1)) & \text{với các trường hợp khác.} \end{cases}$$

Hàm này được gọi là hàm Ackermann. Nó có đặc điểm là giá trị của nó tăng rất nhanh, ứng với giá trị nguyên của m và n .

- Hãy xác định Acker (1,2)
- Viết một hàm đệ quy thực hiện tính giá trị hàm này.

3.2. Hàm $C(n,k)$ với n, k là các giá trị nguyên không âm và $k \leq n$, được định nghĩa:

$$C(n, n) = 1$$

$$C(n, 0) = 1$$

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \text{ nếu } 0 < k < n$$

a) Viết một hàm đệ quy thực hiện tính giá trị $C(n,k)$ khi biết n, k .

b) Chứng minh rằng hàm này cho ra đúng giá trị $C(n,k) = \frac{n!}{(n-k)!k!}$

3.3. Hãy nêu rõ các bước thực hiện khi có lời gọi HANOI(3, A, B, C)

3.4. Viết một hàm đệ quy thực hiện in ngược một dòng ký tự cho trước.

Ví dụ, cho dòng “TURBO C” thì in ra “C OBRUT”.

3.5. Viết một hàm đệ quy nhằm in ra tất cả các hoán vị của n phần tử của một dãy số $a = \{a_1, a_2, \dots, a_n\}$. Ví dụ, với $n = 3$, $a_1 = 1$, $a_2 = 2$, $a_3 = 3$ thì in ra:

1 2 3; 1 3 2; 2 1 3; 2 3 1; 3 1 2; 3 2 1;

3.6. Có n người và n công việc. Biết rằng nếu người i làm việc j thì đạt hiệu suất $a[i][j]$. Hãy viết hàm đệ quy để lập phương án phân công mỗi người một việc khác nhau, sao cho tổng hiệu suất là lớn nhất.

CHƯƠNG 4

MẢNG VÀ DANH SÁCH

4.1. Mảng

Cấu trúc dữ liệu đầu tiên mà ta nói tới là cấu trúc *mảng*, đây là một cấu trúc rất quen thuộc, nó có mặt hầu hết trong các ngôn ngữ lập trình.

4.1.1. Định nghĩa mảng

Mảng là một tập có thứ tự gồm một số cố định n phần tử thuộc cùng một kiểu dữ liệu (n được gọi là *độ dài* hay *kích thước* của mảng). Ngoài giá trị, mỗi phần tử của mảng còn được đặc trưng bởi chỉ số (index), thể hiện thứ tự của phần tử đó trong mảng.

Đối với mảng thường có các phép toán sau:

- Tạo lập mảng
- Duyệt qua các phần tử mảng
- Tìm kiếm một phần tử của mảng
- Sắp xếp các phần tử trong mảng theo một thứ tự nào đó ...

Vì số phần tử của mảng là cố định, nên không có phép bổ sung phần tử mới vào mảng hoặc loại bỏ một phần tử ra khỏi mảng.

Vector là mảng một chiều, mỗi phần tử a_i của nó ứng với một chỉ số i . Chẳng hạn: phần tử của vector A , kí hiệu là A_i hoặc $A[i]$ với i là chỉ số.

Ma trận là mảng hai chiều, mỗi phần tử ứng với hai chỉ số, ví dụ: phần tử của ma trận B , kí hiệu là B_{ij} hoặc $B[i][j]$, với i gọi là chỉ số hàng, j gọi là chỉ số cột.

Tương tự người ta cũng mở rộng ra: mảng ba chiều, mảng bốn chiều, ..., mảng n chiều.

4.1.2. Cấu trúc lưu trữ của mảng

Khi khai báo mảng trong chương trình, các phần tử của mảng sẽ được chương trình dịch tự động lưu trữ trong các ô nhớ kế tiếp nhau trong bộ nhớ máy tính, và có thể được truy nhập trực tiếp đến thông qua một chỉ số.

Ta đã biết cấu trúc của bộ nhớ trong của máy tính được tổ chức thành các ô nhớ kế tiếp nhau, và cũng có thể truy nhập 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.

Cũng chính vì sự tương thích này mà ta có thể coi cấu trúc dữ liệu mảng và cấu trúc lưu trữ (cài đặt) của mảng là một, và còn được gọi là *cấu trúc lưu trữ kế tiếp* hay *cấu trúc lưu trữ tuần tự*.

4.2. Danh sách

4.2.1. Định nghĩa danh sách

Danh sách có hơi khác với mảng ở chỗ: nó là một tập có thứ tự nhưng bao gồm một số biến động các phần tử. Tập hợp các người đến chờ khám bệnh cho ta hình ảnh một danh sách. Họ sẽ được khám theo một thứ tự. Số người có lúc tăng lên (do có người mới đến), có lúc giảm đi (do có người đã đến lượt khám, hoặc do có người bỏ về vì không chờ được lâu).

Một danh sách mà quan hệ lân cận giữa các phần tử được hiển thị ra thì được gọi là *danh sách tuyến tính* (linear list).

Vectơ chính là trường hợp đặc biệt của danh sách tuyến tính, đó là hình ảnh của danh sách tuyến tính xét tại một thời điểm nào đấy.

Như vậy danh sách tuyến tính là một dạng danh sách hoặc *rỗng* (không có phần tử nào) hoặc có dạng (a_1, a_2, \dots, a_n) với a_i ($1 \leq i \leq n$) là các phần tử dữ liệu.

Trong danh sách tuyến tính luôn tồn tại một phần tử đầu a_1 , phần tử cuối a_n . Đối với mỗi phần tử a_i bất kỳ với $1 \leq i \leq n-1$ thì có một phần tử a_{i+1} gọi là *phần tử sau* a_i , và với $2 \leq i \leq n$ thì có một phần tử a_{i-1} gọi là *phần tử trước* a_i , a_i được gọi là phần tử thứ i của danh sách tuyến tính, n được gọi là độ dài hoặc kích thước của danh sách, nó có giá trị thay đổi.

Mỗi phần tử trong một danh sách thường là một cấu trúc (struct) gồm một hoặc nhiều thành phần. Ví dụ: danh mục điện thoại là một danh sách tuyến tính, mỗi phần tử của nó là một đơn vị thuê bao có cấu trúc gồm ba thành phần:

- + Tên đơn vị hoặc tên chủ hộ thuê bao
- + Địa chỉ
- + Số điện thoại

Đối với danh sách thường có phép bổ sung thêm phần tử mới và phép loại bỏ một phần tử cũ, ngoài ra có thể còn có các phép như:

- Tìm kiếm trong danh sách một phần tử mà một thành phần nào đó có một giá trị ấn định.

- Cập nhật một phần tử.
- Sắp xếp các phần tử trong danh sách theo một thứ tự ấn định.
- Ghép hai hoặc nhiều danh sách thành một danh sách lớn.
- Tách một danh sách thành nhiều danh sách con.
- Sao chép một danh sách...

4.2.2. Cài đặt danh sách bằng mảng

Ta có thể dùng mảng một chiều làm cấu trúc lưu trữ của danh sách tuyến tính, mỗi phần tử mảng dùng để lưu trữ một phần tử của danh sách.

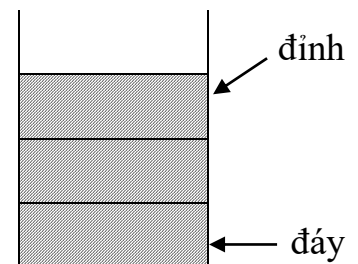
Nhưng do số phần tử của danh sách tuyến tính thường biến động, nghĩa là kích thước n thường thay đổi, nên việc lưu trữ chỉ có thể đảm bảo được nếu biết được $m = \max(n)$. Nhưng điều này thường không dễ xác định chính xác được mà chỉ có thể dự đoán. Vì vậy nếu $\max(n)$ lớn thì khả năng lãng phí bộ nhớ càng nhiều vì có thể có hiện tượng “giữ chỗ để đầy” mà không dùng tới. Hơn nữa, ngay khi đã dự trữ đủ rồi thì việc bổ sung hay loại bỏ phần tử trong danh sách, mà không phải là phần tử cuối sẽ đòi hỏi phải dồn hoặc dẫn danh sách, nghĩa là dịch chuyển một số phần tử lùi xuống (để lấy chỗ bổ sung) hoặc tiến lên (để lấp chỗ của phần tử đã loại bỏ), và điều này sẽ gây tốn phí thời gian không ít nếu các phép toán này được thực hiện thường xuyên. Việc cài đặt danh sách bằng móc nối (được trình bày trong các mục bên dưới) sẽ khắc phục các nhược điểm này.

Tuy nhiên với cách lưu trữ này, như đã nêu ở trên, ưu điểm về tốc độ truy nhập lại rất rõ.

4.3. Ngăn xếp (Stack)

4.3.1. Định nghĩa

Ngăn xếp là một dạng đặc biệt của danh sách tuyến tính mà việc bổ sung hay loại bỏ một phần tử đều được thực hiện ở 1 đầu của danh sách gọi là *đỉnh* (Top). 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 danh sách kiểu LIFO (Last In First Out - Vào sau ra trước).



Hình 4.1. Hình ảnh của ngăn xếp

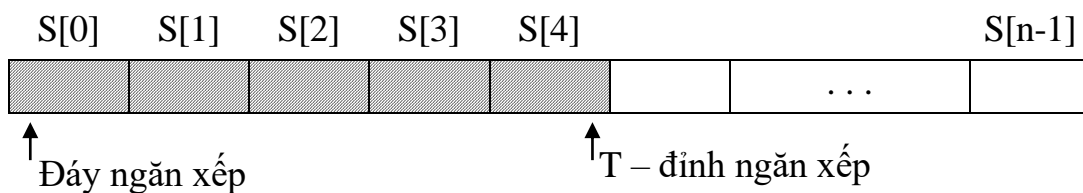
Ví dụ: Chúng ta có thể xem một chồng sách là một

ngăn xếp. Trong chồng sách, các quyển sách đã được sắp xếp theo thứ tự trên - dưới, quyển sách nằm trên cùng được xem là ở đỉnh của chồng sách. Chúng ta có thể dễ dàng đặt một quyển sách mới lên đỉnh chồng sách và lấy quyển sách ở đỉnh ra khỏi chồng sách. Và như thế quyển sách được lấy ra khỏi chồng sách là quyển sách được đặt vào chồng sách sau cùng.

4.3.2. Cài đặt ngăn xếp bằng mảng

Ngăn xếp có thể được cài đặt bằng mảng hoặc móc nối (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 biến T , được gọi là *biến trỏ* để trỏ vào đỉnh ngăn xếp. Chẳng hạn, nếu ngăn xếp có k phần tử thì T sẽ có giá trị bằng $k-1$, nếu ngăn xếp có 1 phần tử thì T sẽ có giá trị bằng 0, còn khi ngăn xếp chưa có phần tử nào thì ta quy ước T sẽ có giá trị -1 .

Có thể thấy cấu trúc lưu trữ của ngăn xếp như hình sau:



Hình 4.2. Hình ảnh cài đặt ngăn xếp bằng mảng có n phần tử

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 trỏ T sẽ tăng lên 1. Khi lấy 1 phần tử ra khỏi ngăn xếp, phần tử của mảng tại vị trí T sẽ được lấy ra và biến trỏ T sẽ giảm đi 1.

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

Sau đây là các phép bổ sung và loại bỏ đối với ngăn xếp được mô tả bằng các hàm:

PUSH (S, T, X)

```
{
/* Hàm này thực hiện bổ sung phần tử X vào ngăn xếp lưu trữ bởi mảng
   S có n phần tử. T là biến trỏ tới đỉnh ngăn xếp */
if (T ≥ n-1)
    printf(“Ngăn xếp TRÀN”);
else
{
    T = T + 1; // tăng T lên vị trí mới
    S[T] = X; // bổ sung phần tử mới X
}
}
```

POP (S, T)

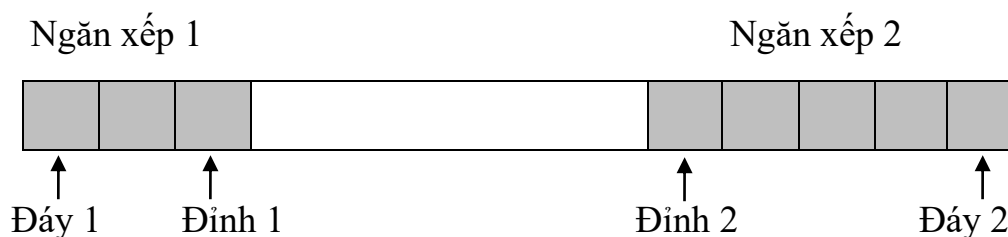
```
{
/* Hàm này thực hiện việc loại bỏ phần tử ở đỉnh ngăn xếp S đang được
   trỏ bởi T. Phần tử bị loại sẽ được thu nhận và đưa ra */
if (T < 0)
    printf(“Ngăn xếp CẠN”);
else
{
    T = T - 1; // giảm T đi một vị trí
    return S[T+1]; // đưa phần tử bị loại ra
}
}
```

Xử lý với nhiều ngăn xếp

Có những trường hợp cùng một lúc ta phải xử lý nhiều ngăn xếp. Như vậy có thể xảy ra tình trạng một ngăn xếp này đã bị TRÀN trong khi không gian dự trữ cho ngăn xếp khác vẫn còn chỗ trống (TRÀN cục bộ).

Làm thế nào để khắc phục được?

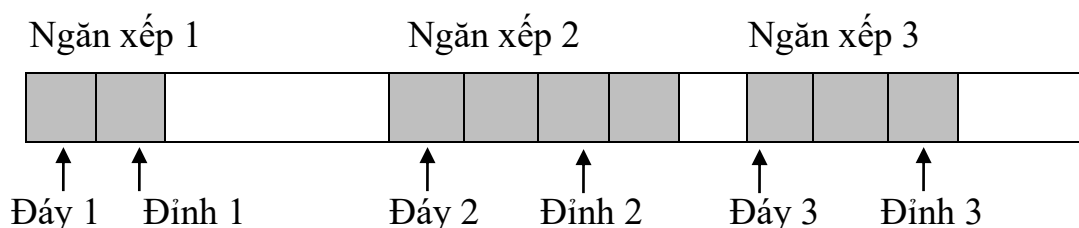
Nếu là hai ngăn xếp thì có thể giải quyết dễ dàng. Ta không qui định kích thước tối đa cho từng ngăn xếp nữa mà không gian nhớ dành ra sẽ được sử dụng chung. Ta sẽ đặt hai ngăn xếp ở hai đầu sao cho hướng phát triển của chúng ngược nhau, như ở hình 4.2.



Hình 4.3. Cách tổ chức hai ngăn xếp dùng chung một mảng

Như vậy có thể một ngăn xếp này dùng lẫn sang quá nửa không gian dự trữ nếu như ngăn xếp kia chưa dùng đến. Do đó hiện tượng TRÀN chỉ xảy ra khi toàn bộ không gian nhớ dành cho chúng đã được dùng hết.

Nhưng nếu số lượng ngăn xếp từ 3 trở lên thì không thể làm theo kiểu như vậy được, mà phải có một giải pháp linh hoạt hơn nữa. Chẳng hạn, nếu có 3 ngăn xếp, lúc đầu không gian nhớ có thể chia đều cho cả 3, như ở hình 4.3.



Hình 4.4. Cách tổ chức ba ngăn xếp dùng chung một mảng

Nhưng nếu có một ngăn xếp nào đó phát triển nhanh bị TRÀN trước mà ngăn xếp khác vẫn còn chỗ thì phải dọn chỗ cho nó bằng cách đẩy ngăn xếp đứng trước sang phải hoặc lùi chính ngăn xếp đó sang trái trong trường hợp có thể. Như vậy thì đáy của các ngăn xếp phải được phép di động và dĩ nhiên các giải thuật bổ sung hoặc loại bỏ phần tử đối với các ngăn xếp hoạt động theo kiểu này cũng phải thay đổi theo.

4.3.3. Một số ví dụ về ứng dụng của ngăn xếp

4.3.3.1. Biểu thức dấu ngoặc cân xứng

Ngăn xếp được sử dụng nhiều trong các chương trình dịch (compiler). Trong mục này chúng ta sẽ trình bày vấn đề: sử dụng ngăn xếp để kiểm tra tính cân xứng của các dấu ngoặc trong chương trình nguồn.

Trong chương trình ở dạng mã nguồn, chẳng hạn chương trình viết bằng ngôn ngữ C, chúng ta sử dụng nhiều các dấu mở ngoặc “(” và đóng ngoặc “)”, mỗi dấu “(” cần phải có một dấu “)” tương ứng đi sau. Tương tự, trong các biểu thức số học hoặc logic, chúng ta cũng sử dụng các dấu mở ngoặc “(” và đóng ngoặc “)”, mỗi dấu “(” cần phải tương ứng với một dấu “)”. Nếu chúng ta loại bỏ tất cả các ký hiệu khác, chỉ giữ lại các dấu mở ngoặc và đóng ngoặc thì chúng ta sẽ có một dãy các dấu mở ngoặc và đóng ngoặc mà ta gọi là *biểu thức dấu ngoặc* và nó cần phải cân xứng. Để minh họa, ta xét biểu thức số học:

$$(((a - b) * (5 + c) + x) / (x + y))$$

Nếu loại bỏ đi tất cả các toán hạng và các dấu phép toán thì ta nhận được biểu thức dấu ngoặc:

$$((() ()) ())$$

1 2 3 4 5 6 7 8 9 10

Biểu thức dấu ngoặc trên là cân xứng, các cặp dấu ngoặc tương ứng là:

$$1 - 10, 2 - 7, 3 - 4, 5 - 6 \text{ và } 8 - 9.$$

Biểu thức dấu ngoặc $(())$ là không cân xứng. Vấn đề đặt ra là làm thế nào để cho biết một biểu thức dấu ngoặc là cân xứng hay không cân xứng.

Sử dụng ngăn xếp, chúng ta dễ dàng thiết kế được giải thuật kiểm tra tính cân xứng của biểu thức dấu ngoặc. Có thể coi một biểu thức dấu ngoặc như một xâu ký tự được tạo thành từ hai ký tự mở ngoặc và đóng ngoặc. Ngăn xếp được sử dụng để lưu các dấu mở ngoặc. Giải thuật gồm các bước sau:

1) Khởi tạo một ngăn xếp rỗng.

2) Đọc lần lượt các ký tự trong biểu thức dấu ngoặc:

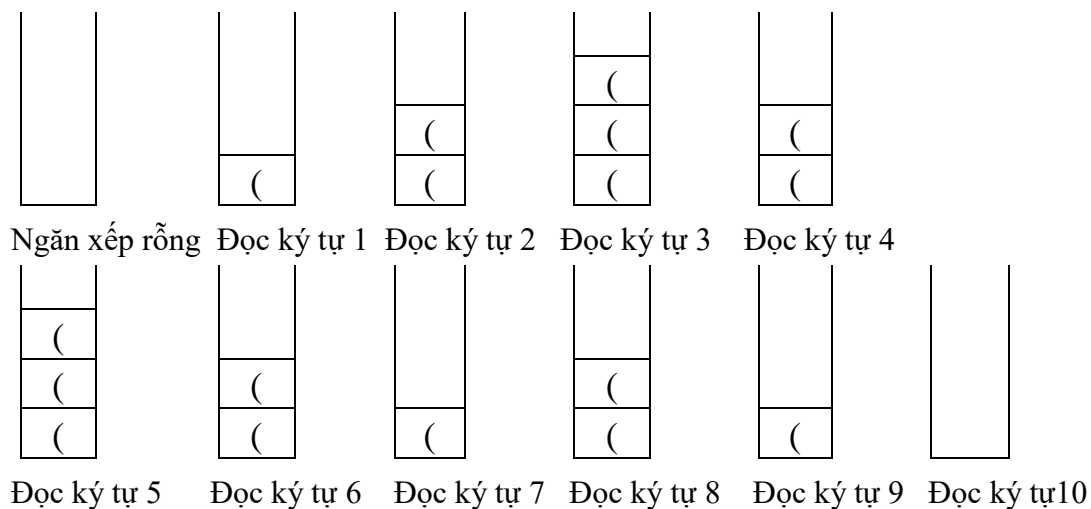
a) Nếu ký tự là dấu mở ngoặc thì đẩy nó vào ngăn xếp.

b) Nếu ký tự là dấu đóng ngoặc thì:

- Nếu ngăn xếp rỗng thì thông báo biểu thức dấu ngoặc không cân xứng và dừng.
- Nếu ngăn xếp không rỗng thì loại dấu mở ngoặc ở đỉnh ngăn xếp.

3) Sau khi ký tự cuối cùng trong biểu thức dấu ngoặc đã được đọc, nếu ngăn xếp rỗng thì thông báo biểu thức dấu ngoặc cân xứng.

Để thấy được giải thuật trên làm việc như thế nào, ta xét biểu thức dấu ngoặc đã đưa ra ở trên: $((())) ()$. Biểu thức dấu ngoặc này là một xâu gồm 10 ký tự. Ban đầu ngăn xếp rỗng. Đọc ký tự đầu tiên, nó là dấu mở ngoặc và được đẩy vào ngăn xếp. Ký tự thứ hai và ba cũng là dấu mở ngoặc, nên cũng được đẩy vào ngăn xếp, và như vậy đến đây ngăn xếp chứa ba dấu mở ngoặc. Ký tự thứ tư là dấu đóng ngoặc, do đó dấu mở ngoặc ở đỉnh ngăn xếp bị loại. Ký tự thứ năm là dấu mở ngoặc, nó lại được đẩy vào ngăn xếp. Tiếp tục, sau khi ký tự cuối cùng được đọc, ta thấy ngăn xếp rỗng, do đó biểu thức dấu ngoặc đã xét là cân xứng. Hình 4.4 minh hoạ các trạng thái của ngăn xếp tương ứng với mỗi ký tự được đọc.



Hình 4.5. Các trạng thái của ngăn xếp khi đọc biểu thức $((())) ()$

4.3.3.2. Định giá biểu thức số học

Trong các ngôn ngữ lập trình, biểu thức số học được viết như dạng thông thường của toán học nghĩa là: mỗi ký hiệu của phép toán hai ngôi được đặt giữa hai toán hạng, có thể thêm dấu ngoặc để chỉ sự ưu tiên. Bởi vậy dạng biểu thức số học này được gọi là dạng *trung tố* (infix).

Chẳng hạn biểu thức: $5 * (7 - 3)$

Dấu ngoặc là cần thiết vì nếu viết: $5 * 7 - 3$ thì theo qui ước về thứ tự ưu tiên của phép toán (mà các ngôn ngữ lập trình đều chấp nhận), biểu thức trên nghĩa là: $(5 * 7) - 3$.

Nhà logic học Ba Lan Lukasiewicz đã đưa ra thêm hai dạng biểu thức số học là: dạng *hậu tố* (postfix) và dạng *tiền tố* (prefix), mà được gọi là dạng ký pháp Ba Lan.

Ở dạng hậu tố các phép toán đi sau các toán hạng. Như biểu thức trên sẽ có dạng: $5\ 7\ 3\ -\ *$

Còn ở dạng tiền tố thì phép toán đi trước toán hạng: $* 5 - 7 3$

Ông cũng khẳng định rằng đối với các dạng ký pháp này thì dấu ngoặc là không cần thiết.

Nhiều bộ dịch khi định giá biểu thức số học thường thực hiện: trước hết chuyển các biểu thức dạng trung tố có dấu ngoặc sang hậu tố, sau đó mới tạo ra các chỉ thị máy để định giá biểu thức ở dạng hậu tố. Việc biến đổi từ dạng trung tố sang hậu tố không quá khó khăn (ta sẽ trình bày dưới đây), còn việc định giá theo dạng hậu tố thì dễ dàng hơn, “máy móc” hơn so với dạng trung tố.

Để minh họa ta thử xét cách định giá của biểu thức hậu tố sau:

$$1\ 5 + 8\ 4\ 1 - - *$$

tương ứng với biểu thức thông thường dạng trung tố:

$$(1 + 5) * (8 - (4 - 1))$$

Biểu thức này được đọc từ trái sang phải cho tới khi tìm ra một phép toán. Hai toán hạng được đọc cuối cùng, trước phép toán này, sẽ được kết hợp với nó. Trong ví dụ của chúng ta thì phép toán đầu tiên được đọc là $+$ và 2 toán hạng tương ứng với nó là 1 và 5, sau khi kết hợp biểu thức con $(1+5)$ này có giá trị là 6, thay vào ta có biểu thức rút gọn:

$$6\ 8\ 4\ 1 - - *$$

Lại đọc từ trái sang phải, phép toán tiếp theo là $-$ và ta xác định được 2 toán hạng của nó là 4 và 1. Thực hiện phép toán $(4-1)$ ta có dạng rút gọn:

$$6\ 8\ 3 - *$$

Lại tiếp tục ta đi tới:

$$6\ 5 *$$

và cuối cùng thực hiện phép toán $*$ ta có kết quả là 30.

Phương pháp định giá biểu thức hậu tố như trên đòi hỏi phải lưu trữ các toán hạng cho tới khi một phép toán được đọc, tại thời điểm này hai toán hạng cuối cùng phải được tìm ra và kết hợp với phép toán này. Như vậy ở đây đã xuất hiện cơ chế hoạt động: “vào sau ra trước” nghĩa là ta sẽ phải sử dụng tới ngăn xếp để lưu trữ các toán hạng. Cứ mỗi lần đọc được một phép toán thì hai giá trị sẽ được lấy ra từ ngăn xếp để áp đặt phép toán đó lên chúng và kết quả lại được đẩy vào ngăn xếp.

Giải thuật sau đây thể hiện các ý trên:

DINH_GIA_BIEU_THUC()

```

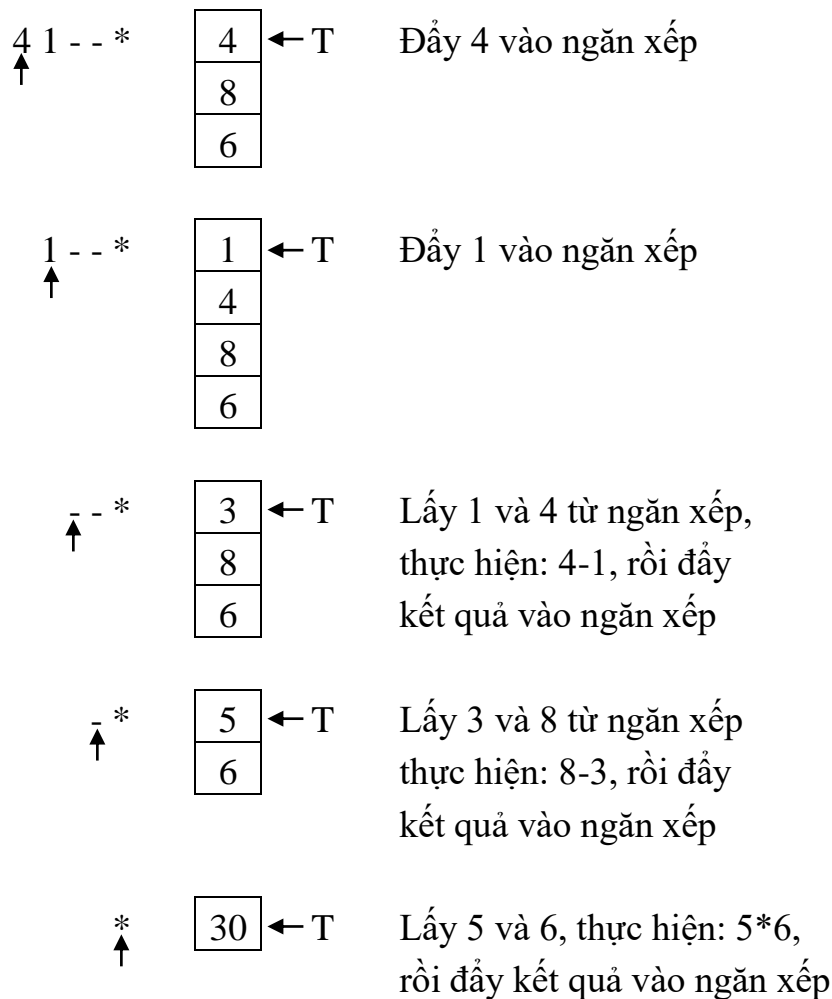
{ //Giải thuật này sử dụng 1 ngăn xếp S, trả bởi T, lúc đầu T = -1
do
{
    Đọc phần tử X tiếp theo trong biểu thức;
    if (X là toán hạng)
        PUSH (S, T, X);
    else
    {
        Y = POP (S, T);
        Z = POP (S, T);
        //tác động phép toán X vào Y và Z, rồi gán kết quả cho W
        W = Z X Y;
        PUSH (S, T, W);
    }
}
while (chưa gặp dấu kết thúc biểu thức);
R = POP (S, T);
printf(R);
}

```

Sau đây là hình ảnh diễn biến việc thực hiện giải thuật này với biểu thức:

1 5 + 8 4 1 - - *

<i>Biểu thức</i>	<i>Stack</i>	<i>Chú thích</i>
1 5 + 8 4 1 - - * ↑	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">1</div> ← T	Đẩy 1 vào ngăn xếp
5 + 8 4 1 - - * ↑	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">5</div> ← T <div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">1</div>	Đẩy 5 vào ngăn xếp
+ 8 4 1 - - * ↑	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">6</div> ← T	Lấy 5 và 1 từ ngăn xếp cộng lại, rồi đẩy kết quả vào ngăn xếp
8 4 1 - - * ↑	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">8</div> ← T <div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">6</div>	Đẩy 8 vào ngăn xếp



Hình 4.6. Diễn biến việc thực hiện giải thuật định giá biểu thức

4.3.3.3. Chuyển đổi biểu thức dạng trung tố sang dạng hậu tố

Để chuyển biểu thức dạng trung tố thành biểu thức dạng hậu tố, trước hết chúng ta cần chú ý tới thứ tự thực hiện các phép toán trong biểu thức dạng trung tố, đó là: các phép toán $*$ và $/$ cần được thực hiện trước các phép toán $+$ và $-$, chẳng hạn: $1 + 2 * 3 = 1 + 6 = 7$. Do đó chúng ta nói rằng các phép toán $*$ và $/$ có quyền ưu tiên cao hơn các phép toán $+$ và $-$, các phép toán $*$ và $/$ cùng mức ưu tiên, các phép toán $+$ và $-$ cùng mức ưu tiên. Mặt khác nếu các phép toán cùng mức ưu tiên đứng kề nhau, thì thứ tự thực hiện chúng là từ trái qua phải, chẳng hạn: $8 - 2 + 3 = 6 + 3 = 9$. Các cặp dấu ngoặc được đưa vào biểu thức dạng trung tố để thay đổi thứ tự thực hiện các phép toán, các phép toán trong cặp dấu ngoặc cần được thực hiện trước, chẳng hạn: $8 - (2 + 3) = 8 - 5 = 3$. Vì vậy vấn đề then chốt trong việc chuyển biểu thức dạng trung tố thành biểu thức dạng hậu tố là: xác định được thứ tự thực hiện các phép toán trong biểu thức dạng trung tố, và đặt các phép toán đó vào đúng vị trí của chúng trong biểu thức dạng hậu tố.

Ý tưởng của giải thuật chuyển biểu thức dạng trung tố thành biểu thức dạng hậu tố là như sau:

- Ta xem biểu thức dạng trung tố như dãy các thành phần, mỗi thành phần là toán hạng hoặc ký hiệu phép toán hoặc dấu mở ngoặc hoặc dấu đóng ngoặc. Giải thuật có đầu vào là biểu thức dạng trung tố đã cho và đầu ra là biểu thức dạng hậu tố.

- Sử dụng một ngăn xếp S để lưu các phép toán và dấu mở ngoặc. Đọc lần lượt từng thành phần của biểu thức dạng trung tố từ trái sang phải, cứ mỗi lần gặp toán hạng, chúng ta viết nó vào biểu thức dạng hậu tố. Khi đọc tới một phép toán (phép toán hiện thời), chúng ta xét các phép toán ở đỉnh ngăn xếp. Nếu các phép toán ở đỉnh ngăn xếp có quyền ưu tiên cao hơn hoặc bằng phép toán hiện thời, thì chúng được kéo ra khỏi ngăn xếp và được viết vào biểu thức dạng hậu tố. Khi mà phép toán ở đỉnh ngăn xếp có quyền ưu tiên thấp hơn phép toán hiện thời, thì phép toán hiện thời được đẩy vào ngăn xếp. Bởi vì các phép toán trong cặp dấu ngoặc cần được thực hiện trước tiên, do đó chúng ta xem dấu mở ngoặc như phép toán có quyền ưu tiên cao hơn mọi phép toán khác. Vì vậy, mỗi khi gặp dấu mở ngoặc thì nó được đẩy vào ngăn xếp, và nó chỉ bị loại khỏi ngăn xếp khi ta đọc tới dấu đóng ngoặc tương ứng, khi đó tất cả các phép toán trong cặp dấu ngoặc đó đã được viết vào biểu thức dạng hậu tố.

Giải thuật sau đây thể hiện các ý trên:

1. Khởi tạo một ngăn xếp rỗng.

2. Đọc lần lượt các thành phần trong biểu thức dạng trung tố:

2.1. Nếu thành phần được đọc là toán hạng thì viết nó vào biểu thức dạng hậu tố.

2.2. Nếu thành phần được đọc là phép toán (phép toán hiện thời), thì thực hiện các bước sau:

- Nếu ngăn xếp không rỗng thì: nếu phần tử ở đỉnh ngăn xếp là phép toán có quyền ưu tiên cao hơn hay bằng phép toán hiện thời, thì phép toán đó được kéo ra khỏi ngăn xếp và viết vào biểu thức dạng hậu tố. Lặp lại bước này.
- Nếu ngăn xếp rỗng hoặc phần tử ở đỉnh ngăn xếp là dấu mở ngoặc hoặc phép toán ở đỉnh ngăn xếp có quyền ưu tiên thấp hơn phép toán hiện thời, thì phép toán hiện thời được đẩy vào ngăn xếp.

2.3. Nếu thành phần được đọc là dấu mở ngoặc thì nó được đẩy vào ngăn xếp.

2.4. Nếu thành phần được đọc là dấu đóng ngoặc thì thực hiện các bước sau:

- (Bước lặp) Loại các phép toán ở đỉnh ngăn xếp và viết vào biểu thức dạng hậu tố cho tới khi đỉnh ngăn xếp là dấu mở ngoặc.
- Loại dấu mở ngoặc khỏi ngăn xếp.

3. Sau khi toàn bộ biểu thức dạng trung tố được đọc, loại các phép toán ở đỉnh ngăn xếp và viết vào biểu thức dạng hậu tố cho tới khi ngăn xếp rỗng.

Ví dụ: Xét biểu thức dạng trung tố :

$$a * (b - c + d) + e / f$$

Đầu tiên ký hiệu được đọc là toán hạng a, nó được viết vào biểu thức dạng hậu tố. Tiếp theo ký hiệu phép toán * được đọc, và ngăn xếp rỗng, nên nó được đẩy vào ngăn xếp. Đọc thành phần tiếp theo, đó là dấu mở ngoặc nên nó được đẩy vào ngăn xếp. Tới đây ta có trạng thái ngăn xếp và biểu thức dạng hậu tố như sau:

(
*

Biểu thức dạng hậu tố: a

Đọc tiếp toán hạng b, nó được viết vào biểu thức dạng hậu tố. Thành phần tiếp theo là phép -, lúc này đỉnh ngăn xếp là dấu mở ngoặc, nên ký hiệu - được đẩy vào ngăn xếp. Ta có:

-
(
*

Biểu thức dạng hậu tố: a b

Thành phần được đọc tiếp theo là toán hạng c, nó được viết vào biểu thức dạng hậu tố. Ký hiệu được đọc tiếp là phép +. Phép toán ở đỉnh ngăn xếp là phép - cùng quyền ưu tiên với phép +, nên nó được kéo ra khỏi ngăn xếp và viết vào đầu ra. Lúc này, đỉnh ngăn xếp là dấu mở ngoặc, nên phép + được đẩy vào ngăn xếp và ta có:

+
(
*

Biểu thức dạng hậu tố:

$a \ b \ c \ -$

Đọc đến toán hạng d, nó được viết vào biểu thức dạng hậu tố. Đọc tiếp ký hiệu tiếp theo: dấu đóng ngoặc. Lúc này ta cần loại lần lượt các phép toán ở đỉnh ngăn xếp và viết chúng vào biểu thức dạng hậu tố, cho tới khi đỉnh ngăn xếp là dấu mở ngoặc thì loại nó. Ta có:

*

Biểu thức dạng hậu tố:

$a \ b \ c \ - \ d \ +$

Ký hiệu được đọc tiếp theo là phép +. Đỉnh ngăn xếp là phép *, nó có quyền ưu tiên cao hơn phép +, do đó nó được loại khỏi ngăn xếp và viết vào biểu thức dạng hậu tố. Đến đây ngăn xếp rỗng, nên phép toán hiện thời + được đẩy vào ngăn xếp và ta có:

+

Biểu thức dạng hậu tố:

$a \ b \ c \ - \ d \ + \ *$

Đọc tiếp toán hạng e, nó được viết vào đầu ra. Ký hiệu được đọc tiếp là phép chia / , nó có quyền ưu tiên cao hơn phép + ở đỉnh ngăn xếp, nên phép / được đẩy vào ngăn xếp. Ký hiệu cuối cùng được đọc là toán hạng f, nó được viết vào đầu ra, ta có:

/
+

Biểu thức dạng hậu tố:

$a \ b \ c \ - \ d \ + \ * \ e \ f$

Loại các phép toán ở đỉnh ngăn xếp và viết vào đầu ra cho tới khi ngăn xếp rỗng, chúng ta nhận được biểu thức dạng hậu tố kết quả là:

$a \ b \ c \ - \ d \ + \ * \ e \ f \ / \ +$

4.4.4. Ngăn xếp và việc cài đặt hàm đệ quy

Một trong các ứng dụng quan trọng của ngăn xếp là sử dụng ngăn xếp để chuyển giải thuật đệ quy thành giải thuật không đệ quy. Với nhiều bài toán, có thể tồn tại cả giải thuật đệ quy và giải thuật không đệ quy để giải quyết. Trong nhiều trường hợp, giải thuật đệ quy kém hiệu quả cả về thời gian và bộ nhớ. Việc loại bỏ

các lời gọi đệ quy trong các hàm đệ quy có thể cải thiện đáng kể thời gian chạy và bộ nhớ đòi hỏi.

Khi một hàm đệ quy được gọi tới từ chương trình chính, ta nói: hàm được thực hiện ở mức 1 (hay độ sâu 1 của tính đệ quy). Nhưng khi thực hiện ở mức 1 thì lại gặp lời gọi tới chính nó, nghĩa là phải đi sâu vào mức 2 và cứ như thế cho tới một mức k nào đấy. Rõ ràng mức k phải được hoàn thành xong thì mức $(k-1)$ mới được thực hiện. Lúc đó ta nói: việc thực hiện được quay về mức $(k-1)$.

Khi từ một mức i , đi sâu vào mức $(i+1)$ thì có thể có một số đối số, biến cục bộ hay địa chỉ (gọi là địa chỉ quay lui) ứng với mức i cần phải được bảo lưu để khi quay về tiếp tục sử dụng.

Còn khi quay từ mức i về mức $(i-1)$, các đối số, biến cục bộ và địa chỉ ứng với mức $i-1$ lại phải được khôi phục để sử dụng.

Như vậy, trong quá trình thực hiện, những đối số, biến cục bộ hay địa chỉ bảo lưu sau lại được khôi phục trước. Tính chất “vào sau ra trước” này dẫn tới việc sử dụng ngăn xếp trong cài đặt hàm đệ quy. Mỗi khi có lời gọi tới chính nó thì ngăn xếp sẽ được nạp để bảo lưu các giá trị cần thiết. Còn mỗi khi thoát ra khỏi một mức thì phần tử ở đỉnh ngăn xếp sẽ được lấy ra để khôi phục lại các giá trị cần thiết cho mức tiếp theo.

Có thể tóm tắt các bước này như sau:

1) Mở đầu

Bảo lưu đối số, biến cục bộ và địa chỉ quay lui.

2) Thân

- Nếu tiêu chuẩn cơ sở (base criterion) ứng với trường hợp suy biến đã đạt được thì thực hiện phần tính kết thúc (final computation) và chuyển sang bước 3.
- Nếu không thì thực hiện đoạn tính từng phần (partial computation) và chuyển sang bước 1 (khởi tạo một lời gọi đệ quy).

3) Kết thúc

Khôi phục lại tham số, biến cục bộ và địa chỉ quay lui và chuyển tới địa chỉ quay lui này.

Sau đây là hàm thể hiện cách cài đặt hàm đệ quy, có dùng ngăn xếp cho bài toán tính $n!$ và bài toán “tháp Hà Nội”.

FACTORIAL() // Bài toán tính $n!$

{/* Cho số nguyên n , giải thuật này thực hiện tính $n!$. Ở đây sử dụng một ngăn xếp A mà đỉnh được trỏ bởi T . Mỗi phần tử của A là một cấu trúc gồm có 2 thành phần:

N - ghi giá trị của n ở mức hiện hành.

RETADD - ghi địa chỉ quay lui.

Lúc đầu ngăn xếp A rỗng: $T = -1$.

Một biến cấu trúc TEMREC được dùng làm biến trung chuyển, nó cũng có 2 thành phần:

PARAM tương ứng với N

ADDRESS tương ứng với RETADD

Ở đây đặt giả thiết: lúc đầu TEMREC đã chứa các giá trị cần thiết, nghĩa là PARAM chứa giá trị n đã cho, ADDRESS chứa địa chỉ ứng với lời gọi trong chương trình mà ta gọi là ĐCC (viết tắt của địa chỉ chính).

Các giải thuật PUSH và POP nêu ở 4.3.2. sẽ được sử dụng ở đây. */

1- // Bảo lưu giá trị của N và địa chỉ quay lui

PUSH (A, T, TEMREC)

2- // Tiêu chuẩn cơ sở đã đạt chưa?

if ($T.N == 0$)

{

$GIAI_THUA = 1$;

goto Bước 4;

}

else

{

$PARAM = T.N - 1$;

ADDRESS=Bước 3;

}

goto Bước 1;

3- // Tính $N!$

$GIAI_THUA = T.N * GIAI_THUA$;

4- // Khôi phục giá trị trước của N và địa chỉ quay lui

TEMREC = POP (A, T);

goto ADDRESS;

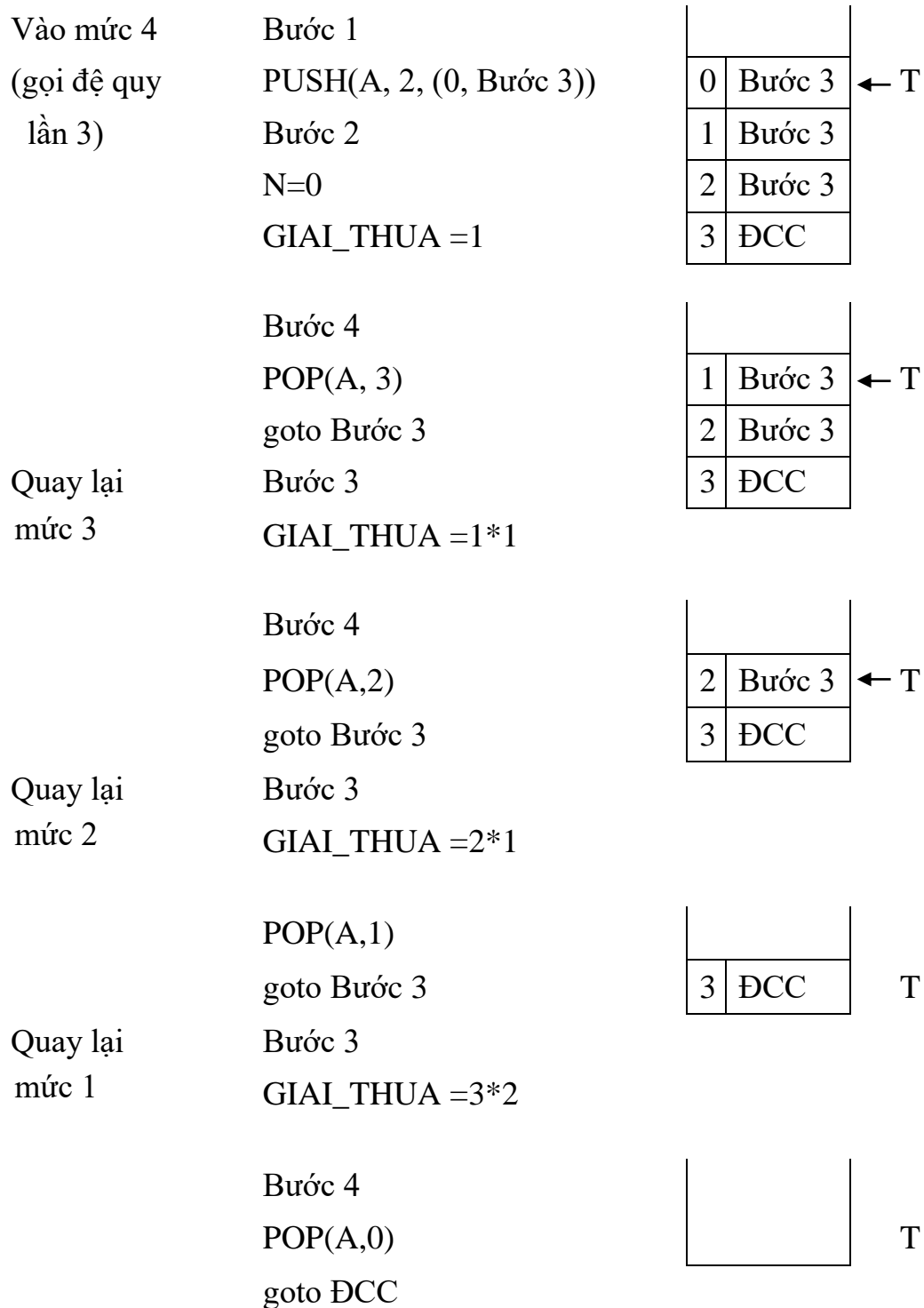
5- // Kết thúc

Return GIAI_THUA

}

Sau đây là hình ảnh minh họa tình trạng của ngăn xếp A, trong quá trình thực hiện giải thuật (mà ta gọi là vết (trace) của việc thực hiện giải thuật), ứng với $n = 3$.

Số mức	Các bước thực hiện	Nội dung của A								
Vào mức 1 (lời gọi chính)	Bước 1 PUSH(A, -1, (3, ĐCC)) Bước 2 $N \neq 0$ PARA=2 ADDRESS=Bước 3	<table><tr><td colspan="2"></td></tr><tr><td>3</td><td>ĐCC</td></tr></table> ← T			3	ĐCC				
3	ĐCC									
Vào mức 2 (gọi đệ quy lần 1)	Bước 1 PUSH(A, 0, (2, Bước 3)) Bước 2 $N \neq 0$ PARA=1 ADDRESS=Bước 3	<table><tr><td colspan="2"></td></tr><tr><td>2</td><td>Bước 3</td></tr><tr><td>3</td><td>ĐCC</td></tr></table> ← T			2	Bước 3	3	ĐCC		
2	Bước 3									
3	ĐCC									
Vào mức 3 (gọi đệ quy lần 2)	Bước 1 PUSH(A, 1, (1, Bước 3)) Bước 2 $N \neq 0$ PARA=0 ADDRESS=Bước 3	<table><tr><td colspan="2"></td></tr><tr><td>1</td><td>Bước 3</td></tr><tr><td>2</td><td>Bước 3</td></tr><tr><td>3</td><td>ĐCC</td></tr></table> ← T			1	Bước 3	2	Bước 3	3	ĐCC
1	Bước 3									
2	Bước 3									
3	ĐCC									



Hình 4.7. Hình ảnh diễn biến của ngăn xếp được dùng để tính $n!$

HANOI_TOWER() // Bài toán tháp Hà Nội

{/* Gọi N là số lượng đĩa, SN chỉ cộc xuất phát, IN chỉ cộc trung chuyển.
DN chỉ cộc đích.

Giải thuật này thực hiện việc chuyển chồng N đĩa từ cộc SN sang cộc DN. Ở đây sử dụng một ngăn xếp ST, mỗi phần tử của nó là một cấu trúc gồm có 5 thành phần tương ứng với: N, SN, IN, DN và RETADD để chứa các giá trị của: N, SN, IN, DN và địa chỉ quay lui. Một biến cấu trúc TEMREC cũng có các thành phần tương ứng với các thành phần nêu trên, lần lượt gọi là NVAL, SNVAL, INVAL, DNVAL và ADDRESS. TEMREC được dùng làm biến trung chuyển, khởi đầu nó ghi nhận các giá trị ban đầu của: N, SN, IN, DN và RETADD.

Ngăn xếp ST có đỉnh được trỏ bởi T, lúc đầu ngăn xếp rỗng thì T= -1

*/

1- // Bảo lưu tham số và địa chỉ quay lui

PUSH (ST, T, TEMREC);

2- /* Kiểm tra giá trị dừng của N, nếu chưa đạt thì chuyển N -1 đĩa từ cộc xuất phát sang cộc trung chuyển */

if (T.N == -1)

goto T.RETADD

else

{

NVAL = T.N - 1;

SNVAL = T.SN;

INVAL = T.IN;

DNVAL = T.DN;

ADDRESS = Bước 3;

goto Bước 1;

}

3- /* Chuyển đĩa thứ N từ cộc xuất phát sang cộc đích và (N-1) đĩa từ cộc trung chuyển sang cộc đích */

TEMREC = POP (ST,T);

```

printf (‘Chuyển đĩa thứ ’, N, ‘ từ cọc ’, SN, ‘ đến cọc ’, DN);
NVAL = T.N – 1;
SNVAL = T.IN;
INVAL = T.SN;
DNVAL = T.DN;
ADDRESS = Bước 4;
goto Bước 1;
4- // Quay lại mức trước
TEMREC = POP (ST,T);
goto T.RETADD;
}

```

4.4. Hàng đợi (Queue)

4.4.1. Định nghĩa

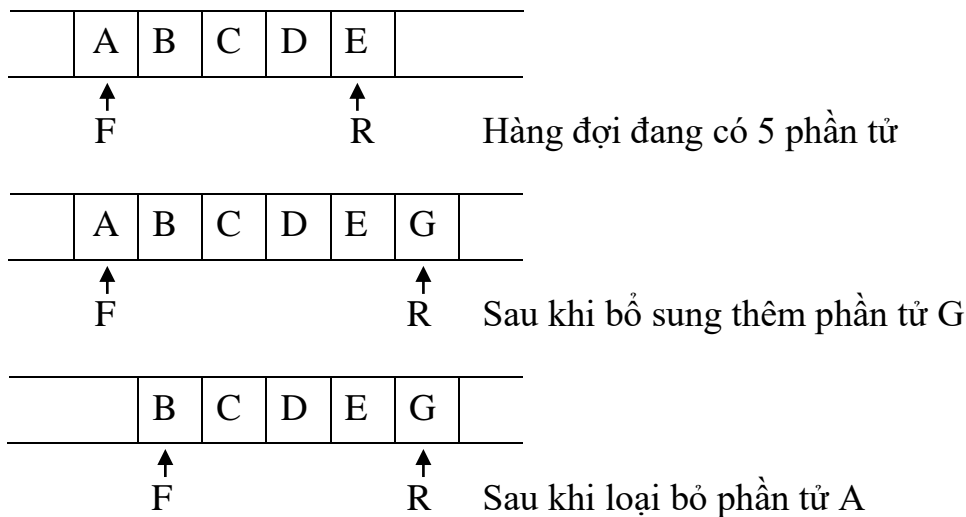
Khác với ngăn xếp, hàng đợi là kiểu danh sách tuyến tính mà phép bổ sung được thực hiện ở một đầu, gọi là *lối sau* (rear) và phép loại bỏ thực hiện ở một đầu khác, gọi là *lối trước* (front).

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. Quy luật này của hàng đợi được gọi là: Vào trước ra trước (FIFO - First In First Out). Ví dụ về hàng đợi có rất nhiều trong thực tế như: Một dòng người xếp hàng chờ vào rạp chiếu phim, hay siêu thị. Trong lĩnh vực máy tính cũng có nhiều ví dụ về hàng đợi: Một tập các tác vụ chờ được 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 đợi, trong khi phần tử mới đưa vào ngăn xếp lại nằm ở đỉnh (đầu) ngăn xếp.

4.4.2. Cài đặt hàng đợi bằng mảng

Có thể dùng một mảng Q có n phần tử làm cấu trúc lưu trữ của hàng đợi. Để có thể truy nhập vào hàng đợi ta phải dùng hai biến trỏ: R trỏ tới lối sau và F trỏ tới lối trước. Khi hàng đợi rỗng thì: $R = F = -1$. Khi bổ sung một phần tử vào hàng đợi, R sẽ tăng lên 1, còn khi loại bỏ phần tử ra khỏi hàng đợi, F sẽ tăng lên 1.



Tuy nhiên với cách tổ chức này có thể xuất hiện tình huống là: các phần tử của hàng đợi sẽ di chuyển khắp mảng khi thực hiện bổ sung và loại bỏ, dẫn tới trường hợp không còn chỗ trống ở phía cuối để bổ sung phần tử mới, trong khi ở phần đầu lại còn chỗ trống nhưng không bổ sung vào được. Do đó người ta phải khắc phục bằng cách coi không gian nhớ dành cho hàng đợi như được tổ chức theo kiểu vòng tròn, nghĩa là với mảng Q thì $Q[0]$ được coi như đứng sau $Q[n-1]$ như ở hình 4.9.

Hình 4.9. Tổ chức hàng đợi theo kiểu vòng

QUEUE_INSERT (F, R, Q, n, X) // Hàm bổ sung phần tử vào hàng đợi

Ban đầu khi hàng đợi rỗng thì $F = R = -1^*/$

```

if (R == n-1) // Chinh lại con trỏ R
    R = 0
else
    R = R + 1;
if (F == R) // Kiểm tra hàng đợi TRÀN
    printf("Hàng đợi TRÀN");
else // Bỏ sung X vào
{
    Q[R] = X;
    if (F == -1) // Điều chỉnh con trỏ F khi bỏ sung lần đầu
        F = 0;
}
}

```

QUEUE_DELETE (F, R, Q, n) // Hàm loại bỏ phần tử của hàng đợi

```

/* Cho F và R trỏ tới lối trước và lối sau của hàng đợi kiểu vòng tròn,
   được lưu trữ bởi mảng Q có n phần tử. Hàm này thực hiện việc loại bỏ
   một phần tử ở lối trước của hàng đợi và đưa nội dung của phần tử đó ra
   */

```

```

if (F == -1) // Kiểm tra hàng đợi CẠN
    printf("Hàng đợi CẠN");
else // Loại bỏ
{
    Y = Q[F]; // Đưa phần tử cần loại bỏ ra biến Y
    if (F == R) // Hàng đợi chỉ có một phần tử
        F = R = -1;
    else // Chinh lại con trỏ F sau phép loại bỏ
        if (F == n-1)
            F = 0

```

```

else
    F = F + 1
return Y;
}
}

```

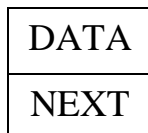
4.5. Danh sách móc nối đơn (singly linked list)

Như đã nêu trong mục 4.2.2, việc dùng mảng để cài đặt danh sách tuyến tính đã bộc lộ rõ nhược điểm trong trường hợp thường xuyên thực hiện các phép bổ sung hoặc loại bỏ phần tử.

Việc sử dụng con trỏ hoặc móc nối để tổ chức (cài đặt) danh sách tuyến tính, mà ta gọi là danh sách móc nối, chính là một giải pháp nhằm khắc phục các nhược điểm đó. Trong mục này và các mục sau, ta sẽ xét tới một số dạng của danh sách móc nối.

4.5.1. Nguyên tắc

Ở đây mỗi phần tử của danh sách được lưu trữ trong một phần tử mà ta gọi là nút (node). Các nút này có thể nằm bất kỳ ở chỗ nào trong bộ nhớ trong. Trong mỗi nút, ngoài phần thông tin về phần tử, còn có chứa địa chỉ của phần tử đứng sau nó trong danh sách. Cấu trúc của mỗi nút có thể hình dung như sau:



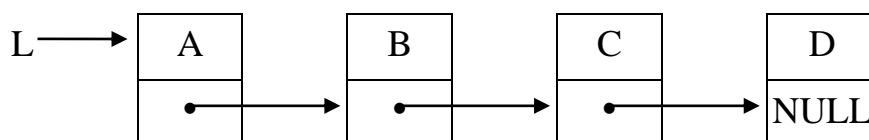
Thành phần DATA chứa dữ liệu của phần tử.

Thành phần NEXT chứa địa chỉ đến nút tiếp theo.

Riêng nút cuối cùng, vì không có nút đứng sau nó nên giá trị của thành phần NEXT của nút này phải chứa một giá trị đặc biệt được gọi là NULL (trống, không có giá trị).

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

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



4.5.2. Một số phép toán

Các giải thuật sau đây thể hiện một số phép toán thường được tác động vào danh sách móc nối đơn.

4.5.2.1. Bổ sung một nút mới vào danh sách móc nối đơn

Ở đây để đơn giản, ta qui ước, khi viết:

$P = \text{malloc}()$

là cấp phát vùng nhớ cho một nút mới (tạo nút mới) và cho con trỏ P trỏ vào nút mới này (tức là P chứa địa chỉ của nút mới).

$\text{INSERT}(L, Q, X)$

/ Cho L là con trỏ, trỏ tới nút đầu tiên của một danh sách móc nối đơn, Q là con trỏ trỏ tới một nút đang có trong danh sách.*

*Giải thuật này thực hiện bổ sung vào sau nút trỏ bởi Q một nút mới mà thành phần DATA nhận giá trị X */*

$P = \text{malloc}();$

$P \rightarrow \text{DATA} = X;$

$P \rightarrow \text{NEXT} = \text{NULL};$

if ($L == \text{NULL}$) // Danh sách rỗng

$L = P;$

else

{

$P \rightarrow \text{NEXT} = Q \rightarrow \text{NEXT};$

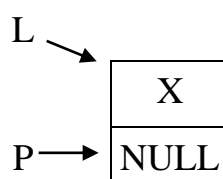
$Q \rightarrow \text{NEXT} = P;$

}

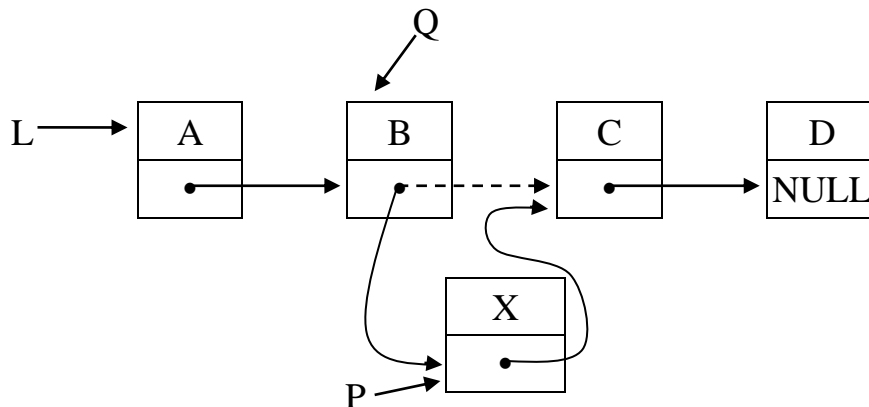
}

* Phép bổ sung được minh họa bởi hình sau:

a) Nếu $L = \text{NULL}$, thì sau phép bổ sung ta có:



b) Nếu $L \neq \text{NULL}$



4.5.2.2. Loại bỏ một nút ra khỏi danh sách móc nối đơn

DELETE(L, Q)

```
{
    /* Cho danh sách móc nối đơn trở bởi L. Giải thuật này thực hiện loại bỏ
       nút trở bởi Q ra khỏi danh sách đó */
    if (L == NULL) // Trường hợp danh sách rỗng
        printf ("Danh sách rỗng");
    else
        if (Q == L) // Loại bỏ nút đầu tiên của danh sách
        {
            L = Q->NEXT; // Loại bỏ nút trở bởi Q
            free(Q); // giải phóng vùng nhớ của nút mà Q trở vào
        }
        else
        {
            // Tìm đến nút đứng trước nút trở bởi Q
            P = L;
            while (P->NEXT != Q)
                P = P->NEXT;
            P->NEXT = Q->NEXT; // Loại bỏ nút trở bởi Q
            free(Q);
        }
}
```


4.5.2.3. Ghép hai danh sách móc nối đơn

COMBINE (P, Q)

{/* Cho hai danh sách móc nối đơn lần lượt trở bởi P và Q. Giải thuật này thực hiện ghép hai danh sách trên thành một danh sách (ghép Q vào cuối P) */

if (Q != NULL) // Q khác rỗng mới tiến hành ghép

if (P == NULL) // Trường hợp danh sách trở bởi P rỗng

P = Q;

else

{

// Tìm đến nút cuối danh sách P

P1 = P;

while (P1->NEXT != NULL)

P1 = P1->NEXT;

P1->NEXT = Q; // Ghép danh sách Q vào cuối P

}

Chú ý: Rõ ràng với các danh sách tuyến tính mà kích thước luôn biến động trong quá trình xử lý hay thường xuyên có các phép bổ sung và loại bỏ tác động, thì cách tổ chức móc nối như trên tỏ ra thích hợp. Tuy nhiên cách cài đặt này cũng có những nhược điểm nhất định sau:

- Chỉ có phần tử đầu tiên trong danh sách được truy nhập trực tiếp, còn các phần tử khác chỉ được truy nhập sau khi đã đi qua các phần tử đứng trước nó.

- Tốn bộ nhớ hơn do phải có thêm thành phần NEXT ở mỗi nút để lưu trữ địa chỉ nút tiếp theo.

4.5.3. Ví dụ áp dụng

4.5.3.1. Biểu diễn đa thức

Bây giờ ta xét tới bài toán cộng hai đa thức của x được tổ chức dưới dạng danh sách móc nối.

Biểu diễn đa thức

Đa thức sẽ được biểu diễn dưới dạng danh sách móc nối đơn, với mỗi nút có qui cách như sau:

COEF
EXP
NEXT

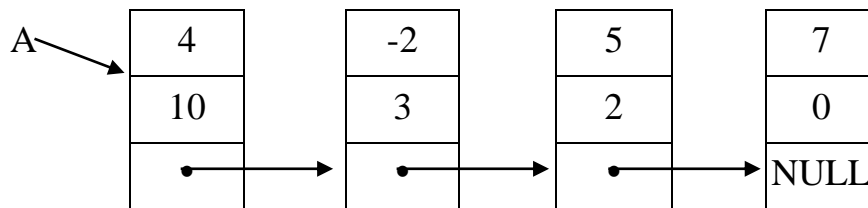
Thành phần COEF chứa hệ số khác không của một số hạng trong đa thức.

Thành phần EXP chứa số mũ tương ứng.

Thành phần NEXT chứa địa chỉ tới nút tiếp theo.

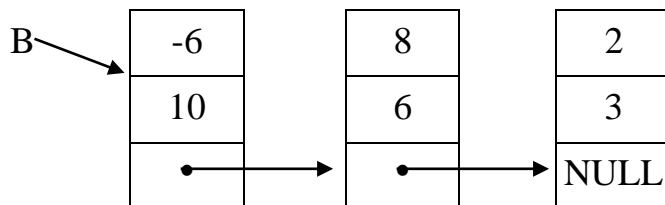
Như vậy, với đa thức:

$A(x) = 4x^{10} - 2x^3 + 5x^2 + 7$ sẽ được biểu diễn bởi:



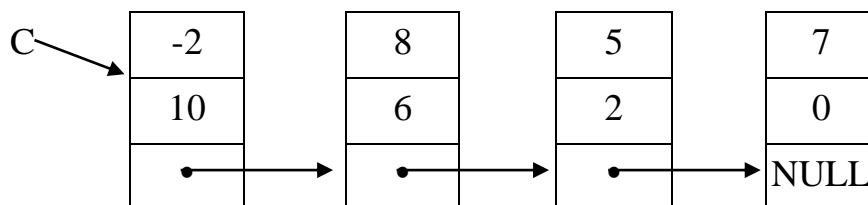
Còn đa thức:

$B(x) = -6x^{10} + 8x^6 + 2x^3$ sẽ có dạng:



Phép cộng hai đa thức này sẽ cho đa thức:

$$C(x) = A(x) + B(x) = -2x^{10} + 8x^6 + 5x + 6$$



4.5.3.2. Giải thuật cộng đa thức

Trước hết cần phải thấy rằng để thực hiện cộng $A(x)$ với $B(x)$ ta phải tìm đến từng số hạng của hai đa thức đó, nghĩa là phải dùng hai con trỏ P và Q để duyệt qua hai danh sách tương ứng với $A(x)$ và $B(x)$ trong quá trình tìm này.

Ta sẽ thấy có những tình huống như sau:

- a) Nếu $P \rightarrow EXP = Q \rightarrow EXP$, ta sẽ phải thực hiện cộng giá trị COEF ở hai nút đó, nếu giá trị tổng khác không thì phải tạo ra nút mới thể hiện số hạng tổng đó và gắn vào danh sách ứng với $C(x)$.
- b) Nếu $P \rightarrow EXP > Q \rightarrow EXP$ (hoặc ngược lại thì cũng tương tự), thì phải sao chép nút P và gắn vào danh sách của $C(x)$.
- c) Nếu một danh sách kết thúc trước, thì phần còn lại của danh sách kia sẽ được sao chép và gắn dần vào danh sách của $C(x)$.

Mỗi lần một nút mới được tạo ra đều phải gắn vào “đuôi” của $C(x)$. Như vậy phải thường xuyên nắm được nút đuôi này bằng một con trỏ D.

Công việc này được lặp lại nhiều lần, vì vậy cần được thể hiện bằng một chương trình con gọi là: ATTACH(H, M, D). Nó thực hiện: tạo một nút mới, đưa vào thành phần COEF của nút này giá trị H, đưa vào thành phần EXP giá trị M và gắn nút mới đó vào sau nút trỏ bởi D.

ATTACH(H, M, D)

```
{  
    P=malloc();  
    P->COEF= H;  
    P->EXP= M;  
    if (C != NULL) // Đã có đuôi  
        D->NEXT = P; // Gắn P vào đuôi  
    else // Chưa có đuôi  
        C=P;  
    D= P; // Nút mới này trở thành đuôi  
}
```

Sau đây là hàm cộng hai đa thức:

PADD(A, B, C)

```
{  
    P = A; Q = B;  
    C=NULL;  
    while (P!=NULL && Q!=NULL) //Cả hai đa thức vẫn còn phần tử
```

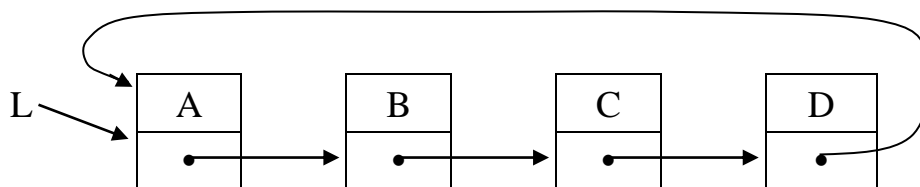
```

if (P->EXP == Q->EXP)
{
    H = P->COEF + Q->COEF;
    if (H !=0)
        ATTACH(H, P->EXP, D);
    P = P->NEXT; Q = Q->NEXT;
}
else
    if (P->EXP > Q->EXP)
    {
        ATTACH(P->COEF, P->EXP, D);
        P = P->NEXT;
    }
    else
    {
        ATTACH(Q->COEF, Q->EXP, D);
        Q = Q->NEXT;
    }
if (Q==NULL) // Danh sách ứng với B(x) đã hết
while (P != NULL)
{
    ATTACH(P->COEF, P->EXP, D);
    P = P->NEXT;
}
else // Danh sách ứng với A(x) đã hết
while (Q != NULL)
{
    ATTACH(Q->COEF, Q->EXP, D);
    Q = Q->NEXT;
}
D->NEXT = NULL; // Kết thúc danh sách C
}

```

4.6. Danh sách móc nối vòng (circularly linked list)

Một cải tiến của danh sách móc nối đơn là kiểu danh sách móc nối vòng. Nó khác với danh sách móc nối đơn ở chỗ: thành phần NEXT của nút cuối cùng không phải chứa giá trị NULL mà là địa chỉ của nút đầu tiên của danh sách. Hình ảnh của nó như sau:

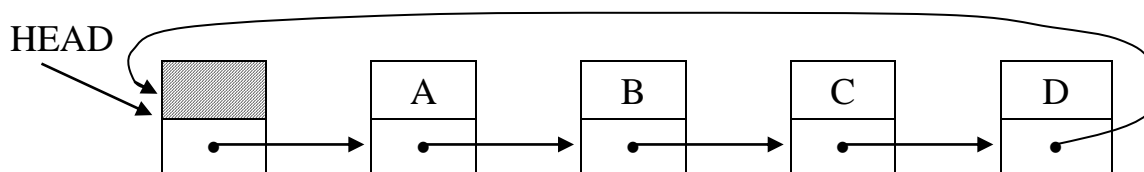


Cải tiến này làm cho truy nhập vào các nút trong danh sách được linh hoạt hơn. Ta có thể truy nhập vào mọi nút trong danh sách bắt đầu từ một nút nào cũng được, không nhất thiết phải từ nút đầu tiên. Điều đó cũng có nghĩa là nút nào cũng có thể coi là nút đầu tiên và con trỏ L trở tới nút nào cũng được.

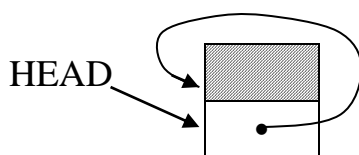
Như vậy đối với danh sách móc nối vòng, chỉ cần cho biết con trỏ tới nút muốn loại bỏ ta vẫn thực hiện được vì vẫn tìm được đến nút đứng trước nó. Với phép ghép, phép tách cũng có những thuận lợi nhất định.

Tuy nhiên, danh sách móc nối vòng có một nhược điểm rất rõ là trong xử lý, nếu không cẩn thận thì sẽ dẫn tới một chu trình không kết thúc. Sở dĩ như vậy là vì không biết được chỗ kết thúc của danh sách.

Điều này có thể khắc phục được bằng cách đưa thêm vào một nút đặc biệt gọi là “nút đầu danh sách” (list head node). Thành phần DATA của nút này không chứa dữ liệu của phần tử nào và con trỏ HEAD bây giờ trở tới nút đầu danh sách này, cho phép ta truy nhập vào danh sách.



Việc dùng thêm nút đầu danh sách đã khiến cho danh sách về mặt hình thức, không bao giờ rỗng. Lúc đó ta có hình ảnh:



Với quy ước $HEAD \rightarrow NEXT = HEAD$

Sau đây là đoạn giải thuật bổ sung một nút vào thành nút đầu tiên của một danh sách móc nối vòng có “nút đầu danh sách” trỏ bởi HEAD.

```
P=malloc();
```

```
P->DATA= X;
```

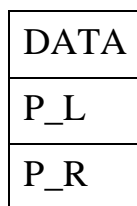
```
HEAD->NEXT = P;
```

```
P->NEXT = HEAD;
```

Chú ý: Việc dùng “nút đầu danh sách” cũng còn có những tiện lợi khác, chẳng hạn có thể dùng nút này để lưu một số thông tin về danh sách, như: độ dài, giá trị lớn nhất, giá trị nhỏ nhất trong danh sách... ,vì vậy ngay đối với danh sách móc nối đơn, trong một số trường hợp người ta cũng dùng “nút đầu danh sách”.

4.7. Danh sách móc nối kép (Doubly linked list)

Với các kiểu danh sách như đã nêu ta chỉ có thể duyệt qua danh sách theo một chiều. Trong một số ứng dụng đôi khi vẫn xuất hiện yêu cầu đi ngược lại. Để có được cả hai khả năng, cần phải đặt ở mỗi nút hai con trỏ, một trỏ tới nút đứng trước nó và một trỏ tới nút đứng sau nó. Như vậy cấu trúc của một nút sẽ như sau:

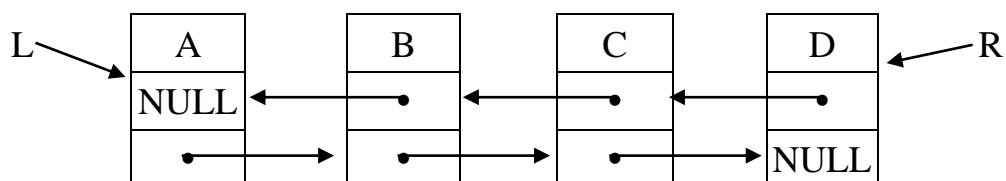


Với P_L là con trỏ trái, trỏ tới nút đứng trước.

Còn P_R là con trỏ phải, trỏ tới nút đứng sau.

Còn thành phần DATA vẫn giống như trên.

Khi đó danh sách móc nối sẽ có dạng:



Ta gọi đó là một danh sách móc nối kép. Ở đây P_L của nút cực trái và P_R của nút cực phải chứa giá trị NULL. Tất nhiên để có thể truy nhập vào danh sách theo cả hai chiều thì phải dùng hai con trỏ: con trỏ L trỏ tới nút cực trái và con trỏ R trỏ tới nút cực phải. Khi danh sách rỗng thì: L = R = NULL.

Bây giờ ta xét một vài giải thuật tác động trên danh sách móc nối kép được tổ chức như đã nêu:

DOUBLE_IN (L, R, Q, X)

```
/* Cho con trỏ L và R lần lượt trỏ tới nút cực trái và nút cực phải của một
danh sách móc nối kép, Q là con trỏ tới một nút trong danh sách này.
Giải thuật này thực hiện bổ sung một nút mới, mà dữ liệu chứa ở X, vào
trước nút trỏ bởi Q*/
```

```
P= malloc(); // 1- Tạo nút mới
```

```
P->DATA=X;
```

```
P->P_R= P->P_L= NULL;
```

```
if (R == NULL) // 2- Trường hợp danh sách rỗng
```

```
    L = R = P;
```

```
else
```

```
    if (Q == L) // 3- Q trỏ tới nút cực trái
```

```
    {
```

```
        P->P_R = Q;
```

```
        Q->P_L = P;
```

```
        L = P;
```

```
    }
```

```
else // 4- Bổ sung vào giữa
```

```
{
```

```
    P->P_L = Q->P_L;
```

```
    P->P_R = Q;
```

```
    Q->P_L = P;
```

```
    P->P_L->P_R = P;
```

```
}
```

```
}
```

DOUBLE_DEL (L, R, Q)

```
{
```

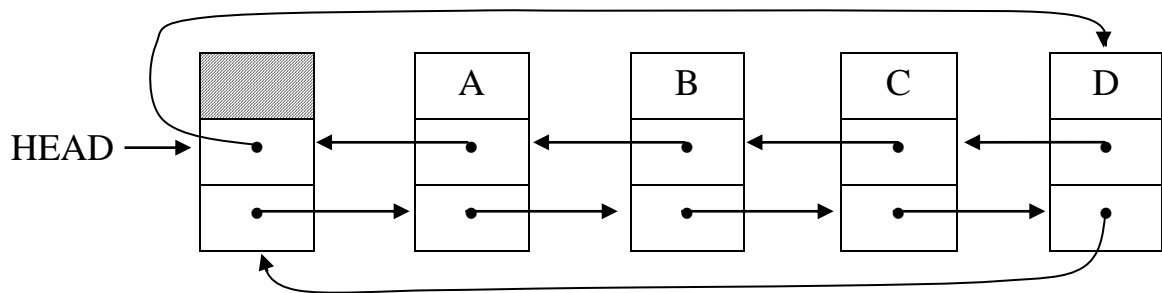
```
/* Cho L và R là hai con trỏ trái và phải của danh sách móc nối kép, Q trỏ
tới một nút trong danh sách. Giải thuật này thực hiện việc loại nút trỏ
bởi Q ra khỏi danh sách */
```

```

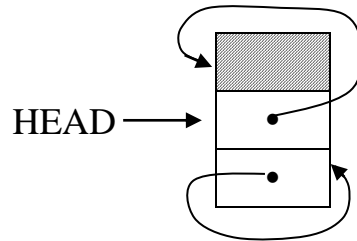
if (R == NULL) // 1- Trường hợp danh sách rỗng
    printf("Danh sách rỗng");
else // 2- Loại bỏ
{
    if (L == R) // Danh sách chỉ có một nút và Q trở tới nút đó
        L = R = NULL;
    else
        if (Q == L) // Nút cực trái bị loại
        {
            L = L->P_R;
            L->P_L = NULL;
        }
        else
            if (Q == R) // Nút cực phải bị loại
            {
                R = R->P_L;
                R->P_R = NULL;
            }
            else // Loại bỏ nút bên trong
            {
                Q->P_L->P_R = Q->P_R;
                Q->P_R->P_L = Q->P_L;
            }
            free(Q);
        }
    }
}

```

Chú ý: Nếu ta đưa nút đầu danh sách vào và tổ chức danh sách móc nối kép theo kiểu lại nối vòng có dạng như hình dưới đây, thì trong hai giải thuật nêu trên sẽ không còn có tình huống ứng với nút cực trái và nút cực phải nữa.



Như vậy nghĩa là nút đầu danh sách đóng cả hai vai trò vừa là nút cực trái, vừa là nút cực phải. Trường hợp danh sách rỗng thì chỉ còn nút đầu danh sách.



Lúc đó: $HEAD \rightarrow P_R = HEAD$

$HEAD \rightarrow P_L = HEAD$

Với danh sách kiểu này giải thuật `DOUBLE_IN` ở trên chỉ có bước 1 (1- Tạo nút mới) và bước 4 (4- Bỏ sung vào giữa). Còn giải thuật `DOUBLE_DEL` chỉ còn một bước:

```
Q->P_L->P_R = Q->P_R;
Q->P_R->P_L = Q->P_L;
free(Q);
```

4.8. Ngăn xếp và hàng đợi móc nối

Như ta đã biết, đối với ngăn xếp, việc truy nhập đều thực hiện ở một đầu (đỉnh). Vì vậy việc cài đặt ngăn xếp bằng cách dùng danh sách móc nối là khá tự nhiên. Chẳng hạn với danh sách móc nối đơn trở bởi `L` thì có thể coi `L` như con trở tới đỉnh ngăn xếp. Bỏ sung một nút vào ngăn xếp chính là bỏ sung một nút vào thành nút đầu tiên của danh sách; loại bỏ một nút ra khỏi ngăn xếp chính là loại bỏ nút đầu tiên của danh sách đang trở bởi `L`. Trong hàm bỏ sung với ngăn xếp móc nối ta không phải kiểm tra hiện tượng `TRÀN` như đối với ngăn xếp kế tiếp, vì ngăn xếp móc nối không hề bị giới hạn về kích thước, nó chỉ phụ thuộc vào giới hạn của bộ nhớ trong của máy tính (chỉ khi bộ nhớ trong của máy tính dành cho việc cấp phát động đã hết).

Đối với hàng đợi thì loại bỏ ở một đầu, bỏ sung lại ở đầu khác. Nếu coi danh sách móc nối đơn như một hàng đợi và coi `L` trở tới lối trước thì việc loại bỏ một nút sẽ không khác gì với ngăn xếp, nhưng bỏ sung một nút thì phải thực hiện vào “đuôi” nghĩa là phải lần tìm đến nút cuối cùng. Nếu tổ chức hàng đợi móc nối có dạng như danh sách móc nối kép có 2 con trở `L` và `R` như đã nói ở trên, thì sẽ không phải tìm “đuôi” nữa. Lúc đó coi lối trước được trở bởi `L`, còn lối sau được trở bởi `R` (hoặc ngược lại cũng được).

Bài tập chương 4

4.1. Cho hệ phương trình đại số tuyến tính có dạng:

$$a_{11}x_1 = b_1$$

$$a_{11}x_1 + a_{12}x_2 = b_2$$

...

$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n$$

Để tiết kiệm, người ta lưu trữ ma trận hệ số hệ phương trình dưới dạng mảng một chiều. Như vậy chỉ cần $n(n+1)/2$ phần tử chứ không cần tới n^2 phần tử.

Hãy lập giải thuật giải hệ phương trình trên ứng với cấu trúc lưu trữ như đã định.

4.2. a) Nếu dùng mảng một chiều có độ dài $n+1$ để biểu diễn một đa thức.

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 + a_0$$

theo dạng: $(n, a_n, a_{n-1}, \dots, a_1, a_0)$

với phần tử đầu tiên biểu diễn bậc của $P(x)$, $n+1$ phần tử sau lần lượt biểu diễn các hệ số của $P(x)$; thì cách lưu trữ này có ưu, nhược điểm gì?

b) Nếu người ta chỉ lưu trữ mũ và hệ số của các số hạng khác không thôi theo kiểu, chẳng hạn như với đa thức: $x^{1000} + 1$ thì mảng biểu diễn có dạng: $(2, 1000, 1, 0, 1)$

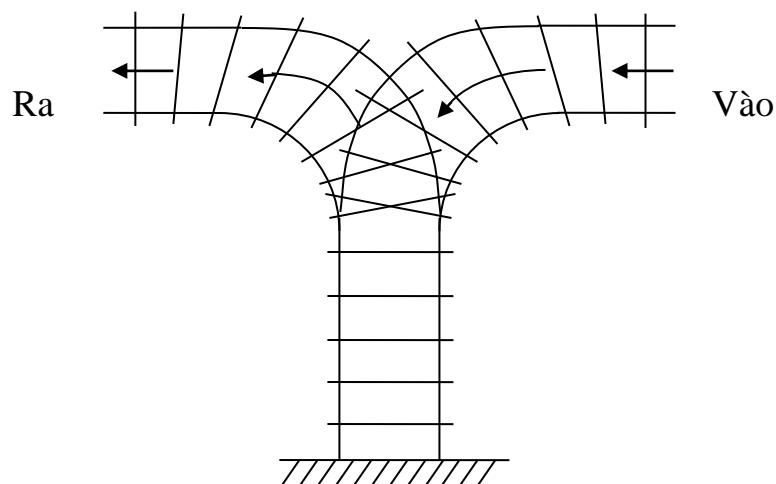
hay với đa thức: $x^7 + 3x^4 - 5x + 3$

thì có biểu diễn: $(4, 7, 1, 4, 3, 1, -5, 0, 3)$

cách lưu trữ mới này có ưu, nhược điểm gì?

c) Hãy lập giải thuật cộng hai đa thức được lưu trữ dưới dạng như câu b)

4.3. Xét một cơ cấu đường tàu vào kho sửa chữa như hình sau:



Giả sử ở đường vào có 4 đầu tàu được đánh số 1, 2, 3, 4. Gọi V là phép đưa một đầu tàu vào kho sửa chữa, R là phép đưa một đầu tàu từ kho ra. Nếu ta thực hiện dãy VVRVVRRR thì thứ tự các đầu tàu lúc ra sẽ là 2 4 3 1 (kho sửa chữa có cơ cấu như một Ngăn xếp). Như vậy có thể coi như ta đã làm một phép hoán vị thứ tự đầu tàu.

Xét trường hợp có 6 đầu tàu: 1, 2, 3, 4, 5, 6 có thể thực hiện một dãy các phép V và R thế nào để đổi thứ tự đầu tàu ở đường ra là: 3 2 5 6 4 1? 1 5 4 6 2 3?

4.4. Hãy lập giải thuật bổ sung và loại bỏ đối với hai ngăn xếp hoạt động theo hướng ngược chiều nhau trên một mảng một chiều S có n phần tử (đáy của ngăn xếp 1 sẽ là S[0], còn đáy của ngăn xếp 2 S[n-1]).

4.5. Hãy dựng hình ảnh minh hoạ tình trạng của ngăn xếp ST trong quá trình thực hiện giải thuật HANOI_TOWER (trong mục 4.4.4) ứng với 3 đĩa.

4.6. Viết biểu thức sau đây dưới dạng hậu tố: $(A*B)/(C+D)$

Dựng hình ảnh của ngăn xếp được sử dụng để định giá trị biểu thức hậu tố này ứng với A = 20; B = 4, C = 9, D = 7.

4.7. Cho bàn cờ tướng với một số quân nằm ở các vị trí tùy ý và hai vị trí A và B bất kỳ trên bàn cờ. Sử dụng hàng đợi, hãy lập giải thuật tìm đường đi ngắn nhất (nếu có) của con mã từ vị trí A đến vị trí B.

4.8. Lập các giải thuật thực hiện các phép sau đây đối với danh sách móc nối đơn mà nút đầu tiên của nó được trỏ bởi L.

- a) Tính số lượng các nút của danh sách.
- b) Tìm tới nút thứ k trong danh sách, nếu có nút thứ k thì in ra dữ liệu của nút đó.
- c) Bổ sung một nút vào sau nút thứ k.
- d) Loại bỏ nút đứng trước nút thứ k.
- e) Cho thêm con trỏ Q trỏ tới một nút có trong danh sách nói trên và một danh sách móc nối đơn khác có nút đầu tiên trỏ bởi P. Hãy chèn danh sách P này vào sau nút trỏ bởi Q.
- f) Tách thành 2 danh sách mà danh sách sau trỏ bởi Q (cho như ở câu e).
- g) Đảo ngược danh sách đã cho (tạo một danh sách L' mà các nút móc nối theo thứ tự ngược lại so với L).

4.9. Cho một danh sách móc nối đơn có nút đầu danh sách trỏ bởi P. Giá trị của thành phần DATA trong các nút giả sử là các số khác nhau, và các nút đã được sắp xếp theo thứ tự tăng dần của giá trị này. Hãy lập giải thuật:

a) Bổ sung một nút mới, mà thành phần DATA có giá trị bằng X, vào danh sách sao cho vẫn đảm bảo thứ tự tăng dần.

b) Loại bỏ nút mà thành phần DATA có giá trị bằng K cho trước.

4.10. Lập giải thuật thực hiện các phép sau đây đối với danh sách móc nối vòng:

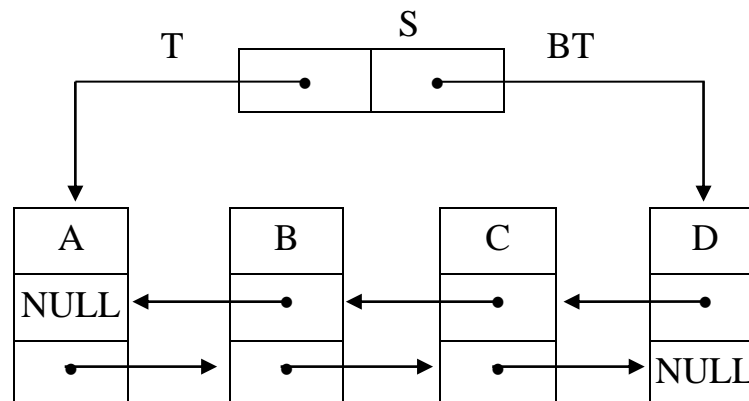
a) Ghép hai danh sách móc nối vòng có nút “đầu danh sách” lần lượt trỏ bởi P và Q, thành một danh sách mà nút đầu danh sách trỏ bởi P.

b) Lập “bản sao” của một danh sách móc nối vòng có nút đầu danh sách trỏ bởi L.

4.11. Cho L và R là 2 con trỏ trỏ tới nút cực trái và cực phải của một danh sách móc nối kép. Hãy lập giải thuật bổ sung và loại bỏ để danh sách đó hoạt động như một hàng đợi.

4.12. Cho danh sách móc nối kép có nút đầu danh sách và cách cấu trúc như trong chú ý của mục 4.7. P là con trỏ, trỏ tới nút đầu danh sách đó. Hãy lập giải thuật loại bỏ tất cả các nút mà thành phần DATA của nó có giá trị bằng K cho trước.

4.13. Nếu tổ chức ngăn xếp theo kiểu danh sách móc nối đơn như đã nêu trong mục 4.5 thì ta có thể duyệt qua ngăn xếp đó (nghĩa là thăm lần lượt các phần tử của ngăn xếp) từ đỉnh tới đáy một cách dễ dàng. Nhưng nếu muốn duyệt theo chiều ngược lại thì rất khó. Để giải quyết yêu cầu này người ta tổ chức ngăn xếp theo kiểu danh sách móc nối kép. Có hai con trỏ trỏ tới đỉnh và đáy của ngăn xếp. Con trỏ T trỏ tới đỉnh ngăn xếp đặt ở thành phần TOP của một nút S, con trỏ BT trỏ tới đáy ngăn xếp đặt ở thành phần BOTTOM của S. Có thể minh hoạ như sau:



Hãy lập giải thuật:

- a) BTRVERSE: thực hiện phép duyệt qua ngăn xếp từ đáy lên
- b) PUSH: thực hiện phép bổ sung một phần tử vào ngăn xếp.
- c) POP: thực hiện phép loại bỏ một phần tử ra khỏi ngăn xếp.

4.14. Cho một đa thức $P(x)$ được tổ chức dưới dạng một danh sách móc nối như đã nêu trong mục 4.5.3.1. Gọi P là con trỏ trỏ tới danh sách đó.

Hãy lập giải thuật tính giá trị của $P(x)$ với giá trị x cho biết.

CHƯƠNG 5

CÂY

Các cấu trúc dữ liệu được nghiên cứu trong chương trước như: mảng, danh sách, ngăn xếp, hàng đợi, là các cấu trúc dữ liệu tuyến tính. Trong chương này chúng ta sẽ nghiên cứu một loại cấu trúc dữ liệu không tuyến tính: cấu trúc dữ liệu cây. Cây là một trong các cấu trúc dữ liệu quan trọng nhất trong khoa học máy tính. Hầu hết các hệ điều hành đều tổ chức các thư mục để lưu trữ file dưới dạng cây. Cây được sử dụng trong thiết kế chương trình dịch, xử lý ngôn ngữ tự nhiên, đặc biệt là trong các vấn đề tổ chức dữ liệu để tìm kiếm và cập nhật dữ liệu được hiệu quả.

5.1. Định nghĩa và các khái niệm

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

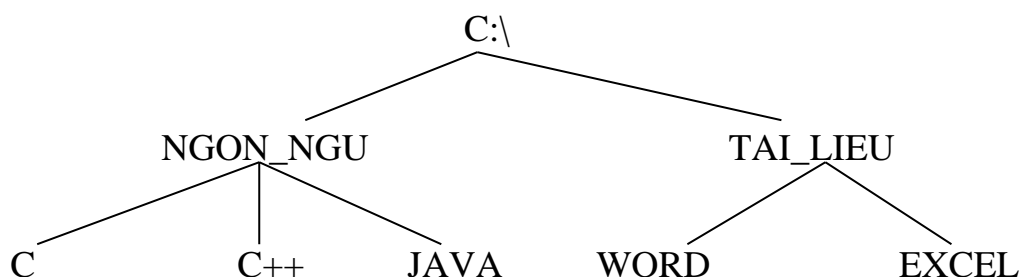
Có thể định nghĩa cây một cách đệ quy như sau:

1. Một nút là một cây. Nút đó cũng là gốc của cây ấy.
2. Nếu n là một nút và T_1, T_2, \dots, T_k là các cây với n_1, n_2, \dots, n_k lần lượt là các gốc, thì một cây mới T sẽ được tạo lập bằng cách cho nút n trở thành *cha* của các nút n_1, n_2, \dots, n_k ; nghĩa là: trên cây này n là gốc còn T_1, T_2, \dots, T_k là các *cây con* (subtree) của gốc. Lúc đó n_1, n_2, \dots, n_k là *con* của nút n .

Người ta qui ước: Một cây không có nút nào, được gọi là *cây rỗng* (null tree).

Các ví dụ về cây:

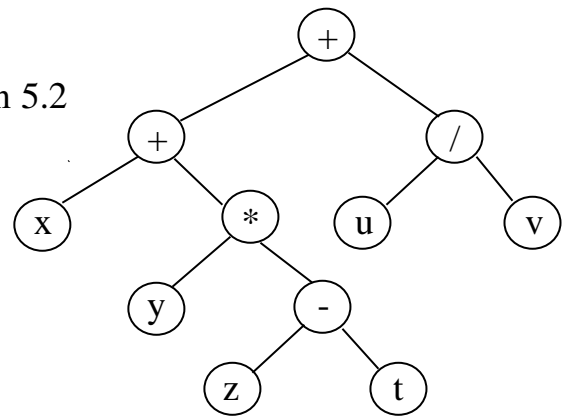
- 1) Cách tổ chức các thư mục trên ổ đĩa là một cây, chẳng hạn:



Hình 5.1. Hình ảnh cây thư mục trên ổ đĩa

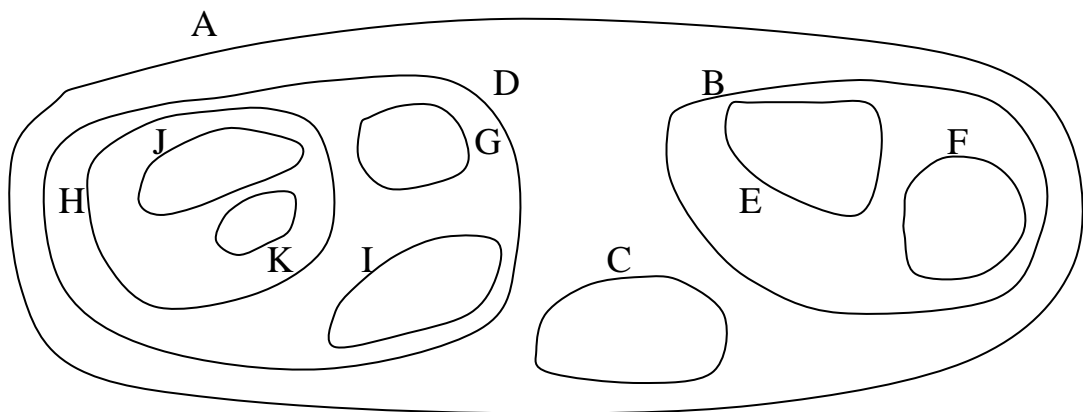
2) Biểu thức số học $x + y * (z - t) + u/v$

có thể biểu diễn dưới dạng cây như hình 5.2

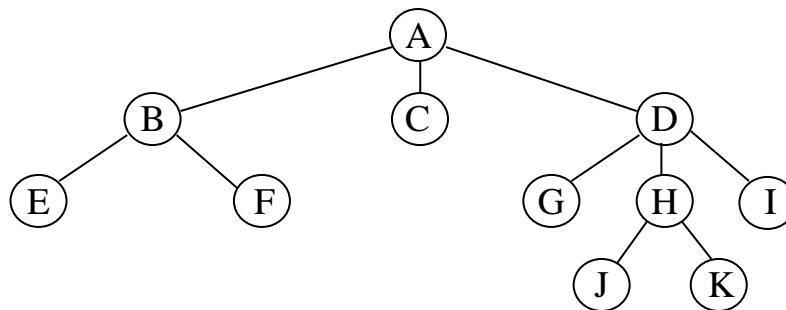


Hình 5.2. Cây biểu thức số học

3) Các tập bao nhau ở hình 5.3 có thể biểu diễn dưới dạng cây như hình 5.4



Hình 5.3. Các tập bao nhau



Hình 5.4. Biểu diễn cây của các tập bao nhau

Đối với cây, chẳng hạn xét cây ở hình 5.4 thì:

Nút A được gọi là gốc của cây.

B, C, D là gốc của các cây con của A.

A là cha của B, C, D; B, C, D là con của A.

* Số các con của một nút gọi là *cấp* (degree) của nút đó. Ví dụ cấp của A là 3, cấp của H là 2.

* Nút có cấp bằng không gọi là *lá* (leaf) hay *nút tận cùng* (terminal node), như các nút : E, F, C, K, I ...

- * Nút không là lá được gọi là *nút nhánh* (branch node)
- * Cấp cao nhất của nút trên cây gọi là *cấp của cây* đó. Ví dụ cây ở hình 5.4 là cây cấp 3.
- * Gốc của cây có *mức* (level) là 1. Nếu nút cha có mức là i thì nút con có mức là $i + 1$.

Ví dụ: nút A có số mức là 1

D có số mức là 2

G có số mức là 3

J có số mức là 4

- * *Chiều cao* (height) hay *chiều sâu* (depth) của một cây là số mức lớn nhất của nút có trên cây đó.

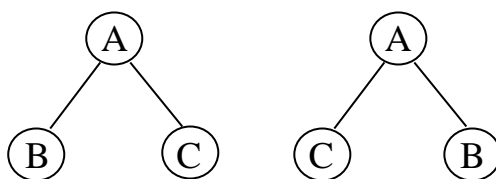
Cây ở hình 5.2 có chiều cao là 5.

Cây ở hình 5.4 có chiều cao là 4.

- * Nếu n_1, n_2, \dots, n_k là dãy các nút mà n_i là cha của n_{i+1} với $1 \leq i \leq k-1$, thì dãy đó gọi là *đường đi* (path) từ n_1 tới n_k . *Độ dài của đường đi* (path length) bằng số nút trên đường đi trừ đi 1. Ví dụ trên cây hình 5.4 độ dài đường đi từ A tới G là 2, từ A tới K là 3.

- * Nếu thứ tự các cây con của một nút được coi trọng thì cây đang xét là *cây có thứ tự* (ordered tree), ngược lại là *cây không có thứ tự* (unordered tree). Thường thứ tự các cây con của một nút được đặt từ trái sang phải.

Hình 5.5 cho ta hai “cây có thứ tự” khác nhau:



Hình 5.5. Hai “cây có thứ tự” khác nhau

Đối với cây, từ quan hệ cha con người ta có thể mở rộng thêm các quan hệ khác phỏng theo các quan hệ như trong gia tộc.

- * Nếu có một tập hữu hạn các cây phân biệt thì ta gọi tập đó là *rừng* (forest).

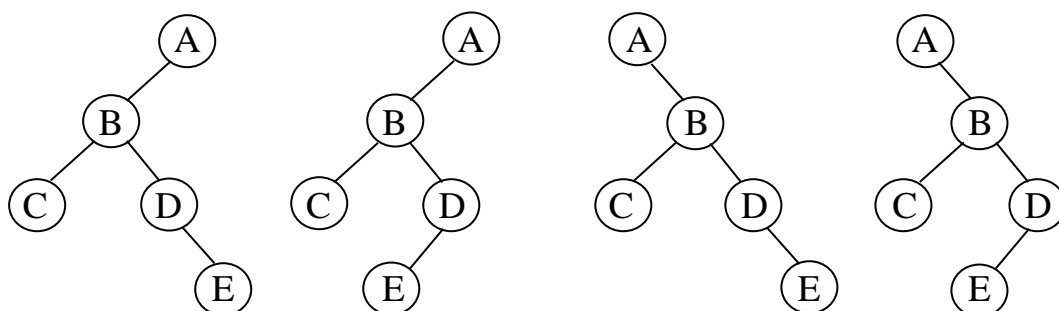
Khái niệm về rừng ở đây phải hiểu theo cách riêng vì: có một cây, nếu ta bỏ nút gốc đi ta sẽ có một rừng. Như ở hình 5.4 nếu bỏ nút gốc A đi, ta sẽ có một rừng gồm 3 cây.

5.2. Cây nhị phân (Binary tree)

5.2.1. Định nghĩa và tính chất

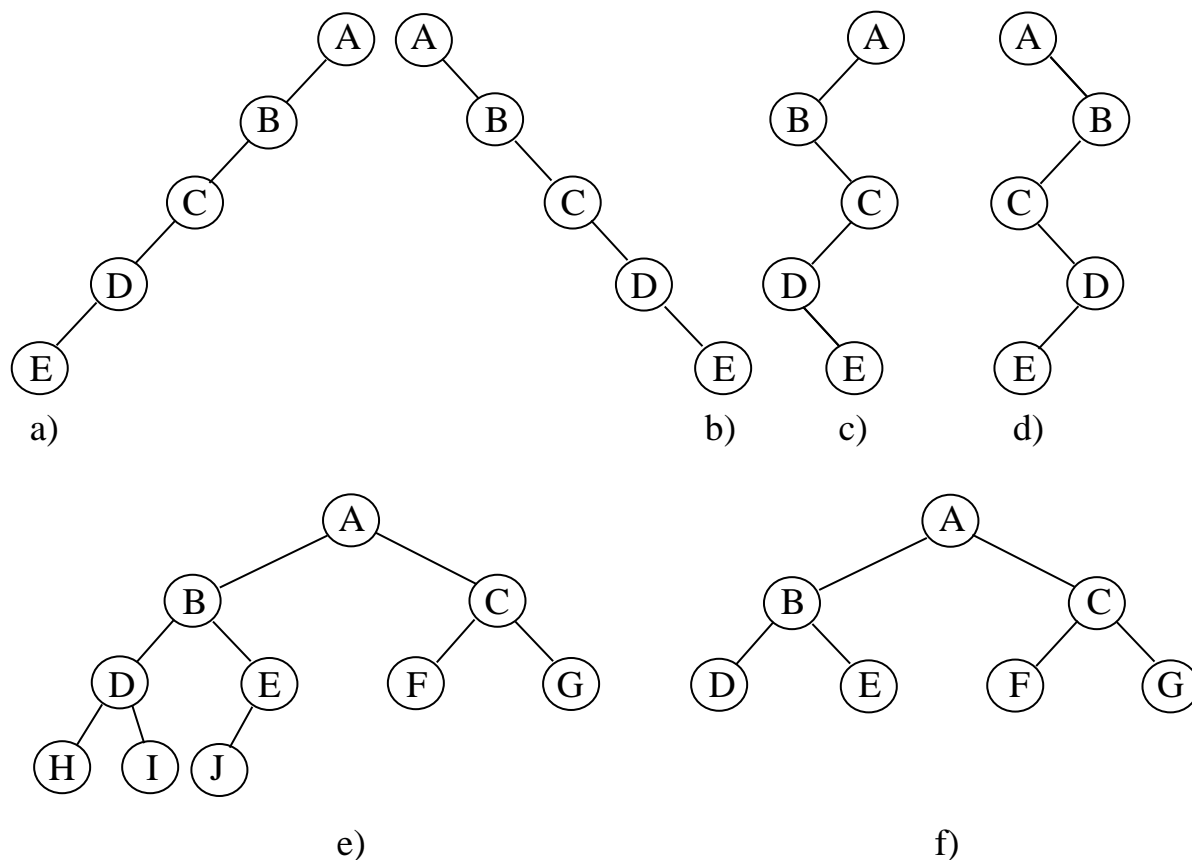
Cây nhị phân là một dạng quan trọng của cấu trúc cây. Nó có đặc điểm là: mọi nút trên cây chỉ có tối đa là hai con. Đối với cây con của một nút người ta cũng phân biệt *cây con trái* (left subtree) và *cây con phải* (right subtree). Như vậy cây nhị phân là cây có thứ tự.

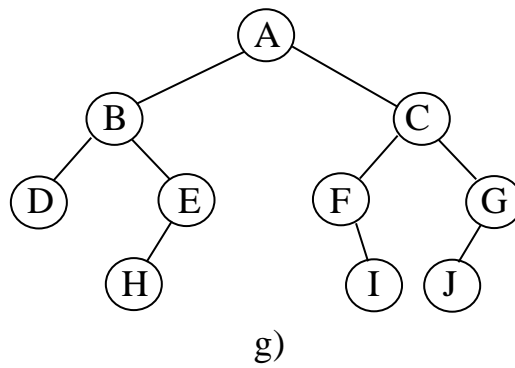
Ví dụ: Cây ở hình 5.2 là cây nhị phân với toán tử ứng với gốc, toán hạng 1 ứng với cây con trái, toán hạng 2 ứng với cây con phải. Các cây nhị phân sau đây là khác nhau, nhưng nếu coi là cây thì chúng chỉ là một (hình 5.6)



Hình 5.6. Các cây nhị phân khác nhau

Cũng cần chú ý tới một số dạng đặc biệt của cây nhị phân như hình 5.7





Hình 5.7. Các dạng đặc biệt của cây nhị phân

Các cây: a) b) c) d) được gọi là *cây nhị phân suy biến* (degenerate binary tree), vì thực chất nó có dạng của một danh sách tuyến tính.

Cây a) được gọi là cây *lệch trái*

Cây b) được gọi là cây *lệch phải*

Cây c) và d) được gọi là cây *zig – zag*

Cây e) được gọi là *cây nhị phân hoàn chỉnh* (complete binary tree), là cây có tính chất: Số các nút ứng với các mức (trừ mức cuối cùng có thể không) đều đạt tối đa, và các nút ở mức cuối cùng đều đạt về phía trái.

Cây f) được gọi là *cây nhị phân đầy đủ* (full binary tree), là cây có tính chất: Số các nút ứng với các mức đều đạt tối đa. Như vậy, cây nhị phân đầy đủ là một trường hợp đặc biệt của cây nhị phân hoàn chỉnh.

Cây g) được gọi là *cây nhị phân gần đầy* nó khác cây e) ở chỗ: các nút ở mức cuối cùng không đạt về phía trái.

Để nhận thấy: trong các cây nhị phân cùng có số lượng nút như nhau thì cây nhị phân suy biến có chiều cao lớn nhất, còn cây nhị phân hoàn chỉnh hoặc cây nhị phân gần đầy thì có chiều cao nhỏ nhất, loại cây này cũng là cây có dạng “cân đối” nhất.

Tất cả các khái niệm đã nêu ở 5.1 đều có thể áp dụng vào cây nhị phân.

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

Bổ đề

1) Số lượng tối đa các nút ở mức i trên một cây nhị phân là:

$$2^{i-1} \quad (i \geq 1)$$

2) Số lượng tối đa các nút trên một cây nhị phân có chiều cao h là:

$$2^h - 1 \quad (h \geq 1)$$

Chứng minh:

1) Sẽ được chứng minh bằng quy nạp: Ta biết:

Ở mức 1: $i = 1$, cây nhị phân có tối đa $1 = 2^0$ nút.

Ở mức 2: $i = 2$, cây nhị phân có tối đa $2 = 2^1$ nút.

Giả sử kết quả đúng với mức $i - 1$, nghĩa là mức này cây nhị phân có tối đa là 2^{i-2} nút. Mỗi nút ở mức $i - 1$ sẽ có tối đa hai con, vậy 2^{i-2} nút ở mức $i - 1$ sẽ cho:

$$2^{i-1} \times 2 = 2^i \text{ nút tối đa ở mức } i.$$

Bổ đề 1) đã được chứng minh.

2) Ta biết rằng chiều cao cây là số mức lớn nhất có trên cây. Theo 1) ta suy ra số nút tối đa có trên cây nhị phân với chiều cao h là:

$$2^0 + 2^1 + 2^2 + \dots + 2^{h-1} = 2^h - 1. \text{ Như vậy bổ đề 2) đã được chứng minh.}$$

*Từ kết quả này có thể suy ra:

Nếu cây nhị phân hoàn chỉnh có n nút thì chiều cao của nó là:

$$h = \lceil \log_2(n + 1) \rceil$$

Ta qui ước: $\lceil x \rceil$ là số nguyên ngay trên của x .

$\lfloor x \rfloor$ là số nguyên ngay dưới của x .

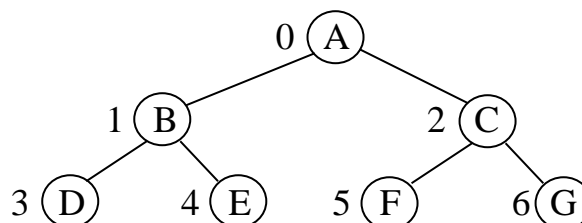
nghĩa là $\lfloor x \rfloor \leq x \leq \lceil x \rceil$

5.2.2. Cài đặt cây nhị phân

5.2.2.1. Cài đặt cây nhị phân bằng mảng

Nếu có một cây nhị phân đầy đủ, ta có thể dễ dàng đánh số cho các nút trên cây đó theo thứ tự lần lượt từ mức 0 trở lên, hết mức này đến mức khác và từ trái sang phải đối với các nút ở mỗi mức.

Ví dụ: Với cây f) ở hình 5.7, có thể đánh số như sau:



Hình 5.8. Cách đánh số cây nhị phân đầy đủ

Ta thấy ngay:

Con của nút thứ i là các nút thứ $2i+1$ và $2i+2$ hoặc

Cha của nút thứ j là $\lfloor (j-1)/2 \rfloor$

Nếu như vậy thì ta có thể lưu trữ cây bằng ma trận một chiều a , theo nguyên tắc: nút thứ i của cây được lưu trữ ở $a[i]$. Với cách lưu trữ này nếu biết địa chỉ của nút cha thì sẽ biết địa chỉ của nút con và ngược lại.

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

A	B	C	D	E	F	G
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$

Tất nhiên nếu cây nhị phân không đầy đủ thì cách lưu trữ này không thích hợp, vì sẽ gây ra lãng phí do có nhiều phần tử nhớ bỏ trống (ứng với cây con rỗng). Chẳng hạn đối với cây lệch trái ở hình 5.7a thì phải lưu trữ bằng một mảng gồm 31 phần tử mà chỉ có 5 phần tử khác rỗng. Hình ảnh miền nhớ lưu trữ nó như sau:

A	B	Ø	C	Ø	Ø	Ø	D	Ø	Ø	Ø	Ø	Ø	Ø	Ø	E	Ø	...
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	-----

(Ø: chỉ chỗ trống)

Ngoài ra nếu cây luôn biến động nghĩa là có phép bổ sung, loại bỏ các nút thường xuyên tác động, thì cách lưu trữ này tất không tránh được các nhược điểm như đã nêu ở chương 4.

Cách lưu trữ móc nối sau đây vừa khắc phục được nhược điểm này, vừa phản ánh được dạng tự nhiên của cây.

5.2.2.2. Cài đặt cây nhị phân bằng móc nối

Trong cách lưu trữ này, mỗi nút có cấu trúc như sau:

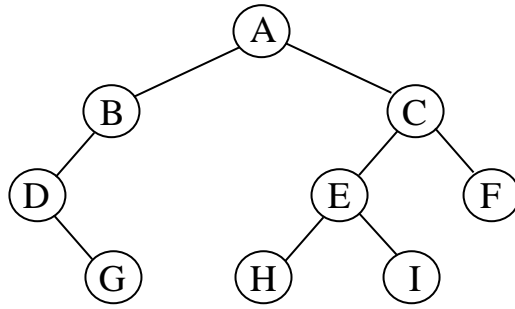
P_L	DATA	P_R
-----	------	-----

Thành phần DATA chứa dữ liệu của nút.

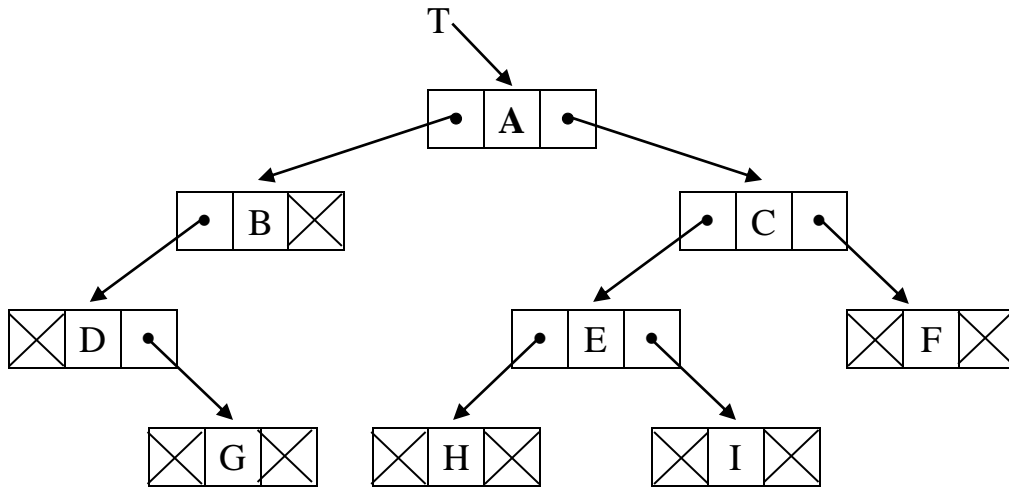
Thành phần P_L là con trỏ trái, trỏ tới cây con trái của nút đó.

Thành phần P_R là con trỏ phải, trỏ tới cây con phải của nút đó.

Ví dụ: Cây nhị phân hình 5.9 có dạng lưu trữ móc nối như ở hình 5.10.



Hình 5.9. Cây nhị phân bất kỳ



Hình 5.10. Hình ảnh cài đặt móc nối của cây nhị phân

Dấu \boxtimes chỉ mối nối không (NULL).

Để có thể truy nhập vào các nút trên cây cần có một con trỏ T, trỏ tới nút gốc của cây đó.

Người ta quy ước: nếu cây nhị phân rỗng thì $T = \text{NULL}$.

Với cách biểu diễn này từ nút cha có thể truy nhập vào nút con dễ dàng, nhưng ngược lại thì không thể làm được.

5.2.3. Phép duyệt cây nhị phân (traversing binary tree)

Thông thường ta hay phải thực hiện các phép xử lý trên mỗi nút của cây theo một thứ tự nào đó.

Ví dụ: đối với cây biểu diễn biểu thức số học nêu ở hình 5.2, để xác định giá trị của biểu thức ta sẽ phải xử lý ở mỗi nút trên cây như sau: nếu là nút biểu diễn toán hạng ta sẽ xác định giá trị của toán hạng đó, nếu là nút biểu diễn dấu phép toán ta sẽ áp đặt phép toán đó lên giá trị của các cây con của nút đó. Nhưng ta không thể thực hiện phép toán này, nếu như các cây con chưa được xử lý. Từ đó ta thấy ngay: thứ tự xử lý nút trên cây là rất quan trọng.

Phép xử lý các nút trên cây, mà ta sẽ gọi chung là phép *thăm* (visit) các nút một cách hệ thống, sao cho mỗi nút chỉ được thăm một lần, gọi là *phép duyệt cây*.

Có ba phép duyệt khác nhau đối với cây nhị phân, các phép đó được định nghĩa một cách đệ quy như sau:

a) Duyệt theo thứ tự trước (preorder traversal)

- Thăm gốc
- Duyệt cây con trái theo thứ tự trước
- Duyệt cây con phải theo thứ tự trước

b) Duyệt theo thứ tự giữa (inorder traversal)

- Duyệt cây con trái theo thứ tự giữa
- Thăm gốc
- Duyệt cây con phải theo thứ tự giữa

c) Duyệt theo thứ tự sau (postorder traversal)

- Duyệt cây con trái theo thứ tự sau
- Duyệt cây con phải theo thứ tự sau
- Thăm gốc

Chú ý rằng: Khi gặp cây rỗng thì “thăm” nghĩa là “không làm gì”.

Với cây nhị phân ở hình 5.9, thì dãy các tên ứng với nút được thăm trong các phép duyệt sẽ là:

a) Theo thứ tự trước:

A B D G C E H I F

b) Theo thứ tự giữa:

D G B A H E I C F

c) Theo thứ tự sau:

G D B H I E F C A

Với cây nhị phân ở hình 5.2, ta lại có:

a) Theo thứ tự trước:

+ + x * y - z t / u v

b) Theo thứ tự giữa:

$$x + y * z - t + u / v$$

c) Theo thứ tự sau:

$$x y z t - * + u v / +$$

Để ý ta thấy:

a) chính là dạng biểu thức tiền tố (prefix)

c) là dạng hậu tố (postfix)

còn b) là dạng trung tố (infix)

Như vậy, các phép duyệt nêu trên đối với cây nhị phân biểu diễn biểu thức số học sẽ cho ta các dạng của biểu thức số học.

Nếu viết dưới dạng hàm đệ quy thì các giải thuật của các phép duyệt cây nhị phân sẽ như sau:

TT_TRUOC(T) // Duyệt theo thứ tự trước

{ /* Ở đây T là con trỏ, trỏ tới gốc cây đã cho có dạng cài đặt móc nối. Mỗi nút trên cây T có qui cách như đã nêu ở 5.2.2.2. Phép thăm ở đây được đặc trưng bởi việc in giá trị thành phần DATA của nút */

if (T != NULL)

{

printf(T->DATA);

TT_TRUOC(T->P_L);

TT_TRUOC(T->P_R);

}

}

TT_GIUA(T) // Duyệt theo thứ tự giữa

{

if (T != NULL)

{

TT_GIUA(T->P_L);

printf(T->DATA);

```

        TT_GIUA(T->P_R);
    }
}

TT_SAU(T) // Duyệt theo thứ tự sau
{
    if (T != NULL)
    {
        TT_SAU(T->P_L);
        TT_SAU(T->P_R);
        printf(T->DATA);
    }
}

```

Ta thấy các hàm đệ quy này được viết khá dễ dàng vì nó phản ánh đúng các định nghĩa đã nêu của các phép duyệt bằng các câu lệnh tương ứng.

Còn nếu muốn thể hiện phép duyệt dưới dạng giải thuật không đệ quy thì có thể sử dụng ngăn xếp, để nạp vào đó địa chỉ các nút cần thiết khi “đi xuống” theo một nhánh nào đó và lấy các địa chỉ ra khỏi ngăn xếp, để xác định đường “đi lên”, trong khi thực hiện phép duyệt. Sau đây là giải thuật cụ thể.

```

TT_TRUOC_S(T) // Hàm không đệ quy duyệt cây theo thứ tự trước
{
    /* T là con trỏ trỏ tới gốc cây đã cho, S là một ngăn xếp (được cài đặt
       bằng mảng) với biến trỏ TOP trỏ tới đỉnh. Con trỏ P được dùng để trỏ
       tới nút hiện đang được xem xét. Trong phép duyệt này khi thăm “gốc”
       xong thì địa chỉ gốc cây con phải của nút vừa được thăm sẽ được lưu
       trữ vào ngăn xếp trước khi “đi xuống” cây con trái. Có sử dụng các
       hàm PUSH, POP đã nêu ở chương 4*/
    if (T == NULL) // 1- Khởi đầu
    {
        printf('Cây rỗng');
        return;
    }
    else

```



```

{
    TOP = -1;
    PUSH(S, TOP, T);
}
while (TOP > -1) // 2 - Thực hiện duyệt
{
    P= POP(S, TOP);
    while (P != NULL) // Thăm nút rồi xuống con trái}
    {
        printf(P->DATA); // Thăm P
        if (P->P_R!= NULL)
            PUSH(S, TOP, P->P_R); // Lưu trữ địa chỉ gốc cây con phải
        P= P->P_L; // Xuống con trái
    }
}
}

```

TT_SAU_S(T) // Hàm không đệ quy duyệt cây theo thứ tự sau

/*Trong phép duyệt này nút “gốc” chỉ được thăm sau khi đã duyệt xong hai con của nó. Như vậy chỉ khi từ con phải đi lên gặp gốc thì gốc mới được thăm, chứ không phải ở lần gặp gốc khi từ con trái đi lên. Do đó để phân biệt người ta đưa thêm dấu âm vào địa chỉ (địa chỉ âm) để đánh dấu cho lần đi lên từ con phải */

```

if (T == NULL) // 1- Khởi đầu
{
    printf('Cây rỗng');
    return;
}
else
{
    TOP = -1;
    P= T;
}

```

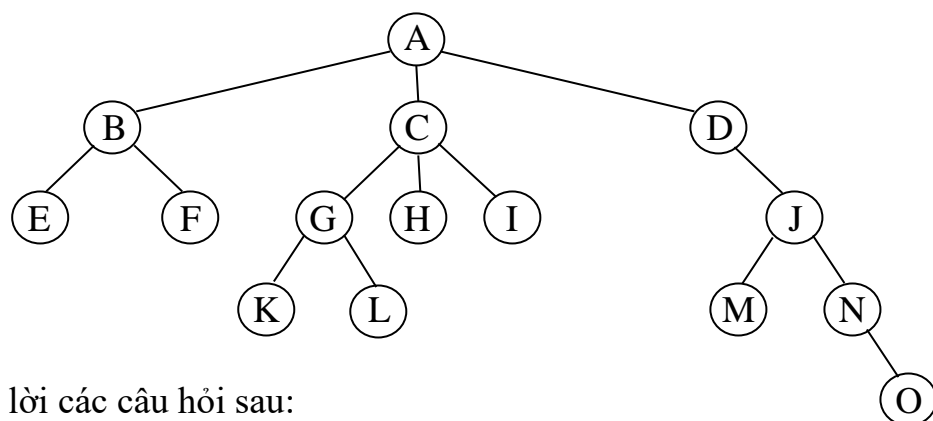
```

while (1) // 2 - Thực hiện duyệt
{
    while (P != NULL) // Lưu trữ địa chỉ “gốc” và xuống con trái
    {
        PUSH (S, TOP, P);
        P = P->P_L;
    }
    while (S[TOP] < 0) // Thăm nút mà con trái và con phải đã duyệt
    {
        P = POP(S, TOP)
        printf(P->DATA);
        if (TOP == -1)
            return
    }
    P = S[TOP]->P_R; // Xuống con phải và đánh dấu
    S[TOP] = - S[TOP];
}
}

```

Bài tập chương 5

5.1. Cho cây:



Hãy trả lời các câu hỏi sau:

- Các nút nào là nút lá?
- Các nút nào là nút nhánh?
- Cha của nút G là nút nào?
- Con của nút C là nút nào?

- e) Các nút nào là anh em của B?
- f) Mức của D, của L là bao nhiêu?
- g) Cấp của B, của D là bao nhiêu?
Cấp của cây này là bao nhiêu?
- h) Chiều cao của cây này là bao nhiêu?
- i) Độ dài đường đi từ A tới F, từ A tới O là bao nhiêu?
- j) Có bao nhiêu đường đi có độ dài 3 trên cây này?

5.2. Vẽ cây nhị phân biểu diễn các biểu thức sau đây và viết chúng dưới dạng tiền tố, hậu tố.

- a) $(a * b + c) / (d - e * f)$
- b) $A / (B + C) + D * E - A * C$
- c) $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee x_3$

Với \wedge là phép và (and)

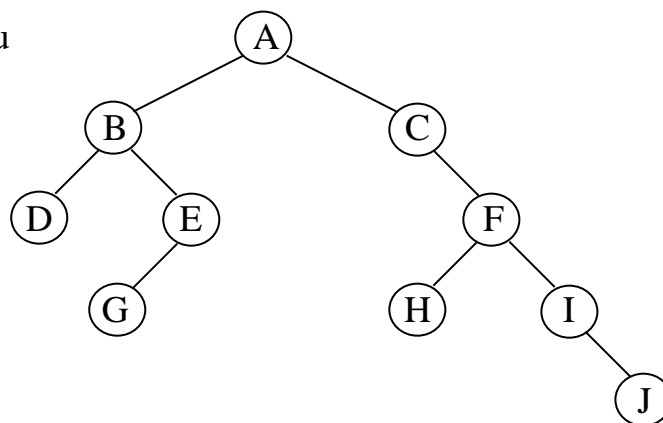
\vee là phép hoặc (or)

\neg là phép không (not)

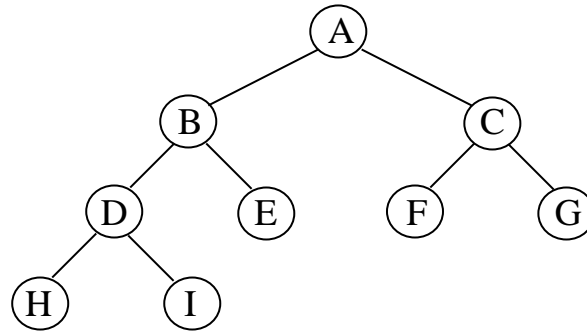
5.3. Cho cây nhị phân

Hãy viết dãy các nút được thăm khi duyệt cây này:

- a) Theo thứ tự trước
- b) Theo thứ tự giữa
- c) Theo thứ tự sau



5.4. Cho cây nhị phân hoàn chỉnh



Hãy minh hoạ mảng 1 chiều lưu trữ cây này.

5.5. Chứng tỏ rằng nếu cho biết dãy các nút được thăm của một cây nhị phân khi duyệt theo thứ tự trước và thứ tự giữa, thì có thể dựng được cây nhị phân đó.

Điều này còn đúng nữa không đối với thứ tự trước và thứ tự sau? đối với thứ tự giữa và thứ tự sau?

5.6. Tìm tất cả các cây nhị phân mà các nút sẽ xuất hiện theo một dãy giống nhau khi duyệt.

- a) theo thứ tự trước và thứ tự giữa.
- b) theo thứ tự trước và thứ tự sau.
- c) theo thứ tự giữa và thứ tự sau.

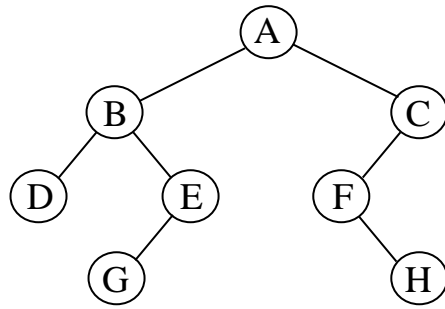
5.7. Khi thực hiện duyệt cây nhị phân theo thứ tự trước theo hàm TT_TRUOC_S nêu ở mục 5.2.3, đối với cây nhị phân có n nút thì số lượng phần tử dự trữ cho ngăn xếp S phải là bao nhiêu?

5.8. Lập hàm không đệ quy thực hiện phép duyệt cấp nhị phân trả bởi T theo thứ tự giữa.

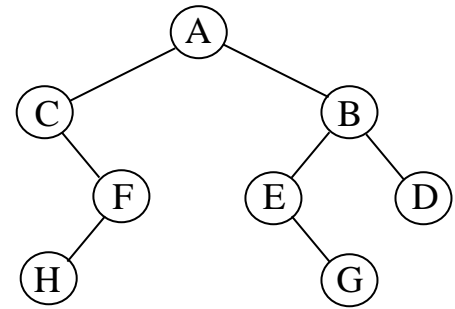
5.9. Lập giải thuật đệ quy thực hiện việc lập bản sao của một cây nhị phân trả bởi T (tạo lập một cây y hệt cây T)

5.10. Lập giả thuật đệ quy thực hiện việc đảo thứ tự trái, phải của các con của mọi nút trên cây nhị phân trả bởi T.

Ví dụ:



Cây ban đầu



Cây đã đảo thứ tự trái,
phải của các nút con

CHƯƠNG 6

SẮP XẾP

6.1. Đặt vấn đề

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

Yêu cầu về sắp xếp thường xuyên xuất hiện trong các ứng dụng về công nghệ thông tin, với những mục đích khác nhau như: sắp xếp dữ liệu lưu trữ trong máy tính để tìm kiếm cho thuận lợi, sắp xếp các kết quả xử lý để in ra trên bảng biểu để dễ theo dõi ...

Các giải thuật sắp xếp được chia làm 2 loại: *sắp xếp trong* (internal sorting), và *sắp xếp ngoài* (external sorting). Sắp xếp trong được thực hiện khi mà các đối tượng cần sắp xếp được lưu ở bộ nhớ trong của máy tính dưới dạng mảng, do đó sắp xếp trong còn được gọi là *sắp xếp mảng*. Khi các đối tượng cần sắp xếp quá lớn, phải lưu ở bộ nhớ ngoài dưới dạng file, ta cần sử dụng các phương pháp sắp xếp ngoài, hay còn gọi là *sắp xếp file*. Trong chương này ta chỉ xét tới các phương pháp sắp xếp trong (sắp xếp mảng).

Mảng cần được sắp xếp có thể là mảng các số nguyên, mảng các số thực, hoặc mảng các ký tự... . Trong trường hợp tổng quát, các đối tượng cần được sắp xếp chứa một số thành phần dữ liệu. Tuy nhiên, trên thực tế không phải toàn bộ các thành phần dữ liệu đều được xem xét đến trong quá trình sắp xếp, mà chỉ một thành phần nào đó (hoặc một vài thành phần nào đó – nhưng trường hợp này ta sẽ không đề cập đến) là được chú ý tới. Thành phần như vậy ta gọi là *khoá* (key) sắp xếp. Chẳng hạn, ta có một mảng các đối tượng sinh viên, mỗi sinh viên gồm các thành phần dữ liệu: Họ tên, ngày sinh, quê quán,..., và ta muốn sắp xếp các sinh viên theo họ tên, khi đó họ tên là khoá sắp xếp.

Do khoá có vai trò đặc biệt như vậy nên sau này, khi trình bày các phương pháp cũng như giải thuật hay trong các ví dụ minh họa, ta sẽ coi khoá như đại diện cho các đối tượng và để cho đơn giản ta chỉ nói tới giá trị khoá. Thực ra phép đổi chỗ được tác động vào các đối tượng (gồm nhiều thành phần dữ liệu), nhưng ở đây ta cũng sẽ chỉ nói tới phép đổi chỗ đối với khoá, và bài toán sắp xếp bây giờ coi như được đặt ra một cách đơn giản với một dãy các khoá K_0, K_1, \dots, K_{n-1} của mảng một chiều K có n phần tử, và $K_i \neq K_j$ nếu $i \neq j$.

Tất nhiên giá trị của khoá có thể là số, là chữ và thứ tự sắp xếp cũng được qui định tương ứng với khoá. Nhưng ở đây, vì mục đích chính là để trình bày và đánh

giá các phương pháp sắp xếp, nên trong các ví dụ minh hoạ, để cho đơn giản và có sự nhất quán, ta sẽ coi giá trị khoá là số và thứ tự sắp xếp là thứ tự tăng dần. Ta sẽ dùng dãy khoá sau đây:

45 27 75 15 65 58 94 37 99 87

để làm ví dụ chung cho các phương pháp sắp xếp.

6.2. Một số phương pháp sắp xếp đơn giản

6.2.1. Sắp xếp kiểu lựa chọn (Selection sort)

Một trong những phương pháp đơn giản nhất để thực hiện sắp xếp một mảng là dựa trên phép lựa chọn.

Nguyên tắc cơ bản của phương pháp sắp xếp này là “Ở lượt thứ j ($j = 1, 2, 3, \dots, n-1$) ta sẽ chọn trong dãy khoá $K_{j-1}, K_j, K_{j+1}, \dots, K_{n-1}$ khoá nhỏ nhất và đổi chỗ nó với K_{j-1} ”.

Như vậy thì rõ ràng là sau j lượt, j khoá nhỏ hơn đã lần lượt ở các vị trí thứ nhất, thứ hai..., thứ j theo đúng thứ tự sắp xếp. Ví dụ:

i	K_i	Lượt	1	2	3	4	.	.	.	9
0	45		15	15	15	15				15
1	27		27	27	27	27				27
2	75		75	75	37	37				37
3	15		45	45	45	45				45
4	65		65	65	65	58				58
5	58		58	58	58	65				65
6	94		94	94	94	94				75
7	37		37	37	75	75				87
8	99		99	99	99	99				94
9	87		87	87	87	87				99

Sau đây là giải thuật:

SELECT_SORT(K, n)

{

/* Cho mảng khoá K gồm n phần tử. Giải thuật này thực hiện sắp xếp các phần tử của K theo thứ tự tăng dần, dựa vào phép chọn phần tử nhỏ nhất trong mỗi lượt */

```

for (i = 0; i < n-1; i++)
{
    m = i;
    for (j = i+1; j < n; j++)
        if (K[j] < K[m])
            m = j;
    if (i < m) // nếu khoá nhỏ nhất khác K[i] thì đổi chỗ K[i] và K[m]
    {
        x = K[i];
        K[i] = K[m];
        K[m] = x;
    }
}

```

6.2.2. Sắp xếp kiểu thêm dần (Insertion sort)

Nguyên tắc sắp xếp ở đây dựa theo kinh nghiệm của những người chơi bài. Khi đã có $i-1$ lá bài đã được sắp xếp đang ở trên tay, nếu rút thêm lá bài thứ i nữa thì sắp xếp lại thế nào? Có thể so sánh lá bài mới lần lượt với lá bài thứ $(i-1)$, thứ $(i-2)$... để tìm ra “chỗ” thích hợp và “chèn” nó vào chỗ đó.

Dựa trên nguyên tắc này có thể triển khai một cách sắp xếp như sau:

Ban đầu K_0 được coi như bảng chỉ gồm có một khoá đã được sắp xếp. Xét thêm K_1 , so sánh nó với K_0 để xác định chỗ “chèn” nó vào, sau đó ta sẽ có một bảng gồm hai khoá đã được sắp xếp. Đối với K_2 lại so sánh nó với K_1 , K_0 và cứ tương tự như vậy đối với K_3 , K_4 , K_5 ... cuối cùng sau khi xét xong K_{n-1} thì bảng khoá đã được sắp xếp hoàn toàn.

Ta thấy ngay phương pháp này rất thuận tiện khi các khoá của dãy được nhập vào từ bàn phím, hoặc được đọc từ tệp, rồi được đưa dần vào mảng K (nhập xong khoá nào thì sắp xếp ngay khoá đó, khi nhập hết các khoá thì mảng K cũng đã được sắp xếp). Có thể minh hoạ qua bảng sau:

Lượt	1	2	3	4		8	9	10
Khoá đưa vào	45	27	75	15	. . .	37	99	87
0	45	27	27	15		15	15	15
1	-	45 ↓	45	27 ↓		27	27	27
2	-	-	75	45 ↓		37	37	37
3	-	-	-	75 ↓		45 ↓	45	45
4	-	-	-	-		58	58	58
5	-	-	-	-		65	65	65
6	-	-	-	-		75 ↓	75	75
7	-	-	-	-		94	94	87
8	-	-	-	-		-	99	94 ↓
9	-	-	-	-		-	-	99 ↓

(dấu - chỉ chỗ trống trong mảng K chưa dùng đến

Dấu ↓ chỉ việc phải dịch chuyển các khoá cũ để lấy chỗ chèn khoá mới vào)

Nhưng nếu các khoá chưa được sắp xếp đã có sẵn trên mảng K rồi thì sao?

Sắp xếp vẫn có thể thực hiện được ngay tại mảng K chứ không phải chuyển sang một mảng khác. Lúc đó các khoá cũng sẽ lần lượt được xét tới và việc xác định chỗ cho khoá mới vẫn làm tương tự, chỉ có khác là: do không có sẵn chỗ trống như trường hợp nói ở trên (vì khoá mới đang xét và các khoá chưa được xét đã chiếm hết các vị trí đằng sau), nên để dành chỗ cho khoá mới ta phải dịch chuyển một số khoá lùi lại sau, do đó phải đưa khoá mới này ra một biến phụ và sẽ đưa vào vị trí thực của nó sau khi đã đẩy các khoá cần thiết lùi lại một vị trí.

Sau đây là giải thuật ứng với trường hợp này.

INSERT_SORT (K, n)

```

/* Trong hàm này người ta dùng x làm biến phụ để chứa khoá mới đang
   được xét*/
for (i = 1; i < n; i++)
{

```

```

x = K[i];
j = i-1;
while ((K[j] > x) && (j>=0)) // Lùi K[j] lại một vị trí
{
    K[j+1] = K[j];
    j--;
}
K[j+1] = x;
}
}

```

Bảng ví dụ minh hoạ tương ứng với các lượt sắp xếp theo giải thuật này, tương tự như bảng đã nêu ở trên, chỉ có khác là không có chỗ nào trống trong mảng K, vì những chỗ đó đang chứa các khoá chưa được xét tới trong mỗi lượt.

6.2.3. Sắp xếp kiểu đổi chỗ (Exchange sort)

Trong các phương pháp sắp xếp nêu trên, tuy kỹ thuật đổi chỗ đã được sử dụng nhưng nó chưa trở thành một đặc điểm nổi bật. Bây giờ ta mới xét tới phương pháp mà việc đổi chỗ một cặp khoá kề cận, khi chúng ngược thứ tự, sẽ được thực hiện thường xuyên cho tới khi toàn bộ bảng các khoá đã được sắp xếp. Ý cơ bản có thể nêu như sau:

Dãy các khoá sẽ được duyệt từ đáy lên đỉnh. Dọc đường, nếu gặp hai khoá kề cận ngược thứ tự thì đổi chỗ chúng. Như vậy trong lượt đầu khoá có giá trị nhỏ nhất sẽ chuyển dần đỉnh. Đến lượt thứ hai khoá có giá trị nhỏ thứ hai sẽ được chuyển lên vị trí thứ hai Nếu hình dung dãy khoá được đặt thẳng đứng thì sau từng lượt sắp xếp các giá trị khoá nhỏ sẽ “nổi” dần lên giống như các bọt nước nổi lên trong nồi nước đang sôi. Vì vậy phương pháp này thường được gọi bằng cái tên khá đặc trưng là: sắp xếp *kiểu nổi bọt* (Bubble sort).

Ví dụ:

i	K _i	Luợt	1	2	3	4	5	.	.	.	9
0	45		→ 15	15	15	15	15				15
1	27			→ 45	27	27	27				27
2	75			→ 27	45	→ 37	37	37			37
3	15			→ 75	→ 37	45	45	45			45
4	65			→ 37	→ 75	→ 58	58	58			58
5	58			→ 65	→ 58	→ 75	→ 65	65			65
6	94			→ 58	→ 65	→ 65	→ 75	75			75
7	37			→ 94	→ 87	87	87	87			87
8	99			→ 87	→ 94	94	94	94			94
9	87			→ 99	99	99	99	99			99

Sau đây là giải thuật:

BUBLE_SORT (K, n)

```

{
    for (i = 0; i < n-1; i++)
        for (j = n-1; j > i; j--)
            if (K[j] < K[j-1])
            {
                x = K[j];
                K[j] = K[j - 1] ;
                K[j - 1] = x
            }
}

```

Giải thuật này rõ ràng còn có thể cải tiến được nhiều. Chẳng hạn xét qua ví dụ ở trên ta thấy: Sau lượt thứ ba không phải chỉ có ba khoá 15, 27, 37 vào đúng vị trí sắp xếp của nó mà là 5 khoá. Còn sau lượt thứ tư thì tất cả các khoá đã nằm đúng vào vị trí của nó rồi. Như vậy nghĩa là năm lượt cuối không có tác dụng gì thêm cả. Từ đó có thể thấy: nếu “nhớ” được vị trí của khoá được đổi chỗ cuối cùng ở

mỗi lượt thì có thể coi đó là “giới hạn” cho việc xem xét ở lượt sau. Chừng nào mà giới hạn này chính là vị trí thứ $n-1$, nghĩa là trong lượt ấy sẽ không có một phép đổi chỗ nào nữa thì quá trình sắp xếp có thể kết thúc được. Nhận xét này sẽ dẫn tới một giải thuật cải tiến hơn, chắc chắn có thể làm cho số lượt giảm đi và số lượng các phép so sánh trong mỗi lượt cũng giảm đi nữa. Việc xây dựng giải thuật theo ý cải tiến này được coi như một bài tập.

6.2.4. Phân tích và so sánh ba phương pháp

Đối với một phương pháp sắp xếp, khi xét tới hiệu lực của nó, ngoài những đánh giá về mặt không gian nhớ cần thiết, người ta thường lưu ý đặc biệt tới chi phí về thời gian. Mà thời gian thì chủ yếu phụ thuộc vào việc thực hiện các phép so sánh giá trị khoá và các phép chuyển chỗ phần tử khi sắp xếp. Vì vậy thông thường người ta lấy số lượng trung bình các phép so sánh, hoặc các phép chuyển chỗ làm đại lượng đặc trưng cho chi phí về thời gian thực hiện của từng phương pháp. Ở đây phép so sánh sẽ được coi như một “phép toán tích cực” để đánh giá thời gian thực hiện của các giải thuật.

Đối với phương pháp sắp xếp kiểu lựa chọn (giải thuật `SELECT_SORT`), ta thấy: ở lượt thứ i ($i = 1, 2, \dots, n-1$) để tìm khoá nhỏ nhất bao giờ cũng cần $C_i = (n-i)$ phép so sánh. Số lượng phép so sánh này không hề phụ thuộc gì vào tình trạng ban đầu của dãy khoá cả. Từ đó suy ra:

$$C_{\min} = C_{\max} = C_{tb} = \sum_{i=1}^{n-1} (n-i) = \frac{n(n+1)}{2}$$

Còn với phương pháp sắp xếp kiểu thêm dần (giải thuật `INSERT_SORT`) thì có hơi khác. Rõ ràng số lượng phép so sánh phụ thuộc vào dãy khoá ban đầu. Trường hợp thuận lợi nhất ứng với dãy khoá đã được sắp xếp rồi. Như vậy ở mỗi lượt chỉ cần 1 phép so sánh. Do đó:

$$C_{\min} = \sum_{i=1}^{n-1} 1 = n-1$$

Nhưng nếu dãy khoá ban đầu có thứ tự ngược với thứ tự sắp xếp thì ở lượt thứ i phải cần có: $C_i = (i-1)$ phép so sánh. Vì vậy:

$$C_{\max} = \sum_{i=2}^n (i-1) = \frac{n(n+1)}{2}$$

Nếu giả sử mọi giá trị khoá đều xuất hiện đồng khả năng thì trung bình ở lượt thứ i có thể coi như $C_i = i/2$ phép so sánh. Suy ra:

$$C_{tb} = \sum_{i=2}^n \frac{i}{2} = \frac{n^2 + n - 2}{4}$$

Với sắp xếp kiểu mới nổi bật theo giải thuật BUBBLE_SORT như trên, nghĩa là chưa hề có cải tiến gì, thì tương tự như phương pháp đầu, ta cũng có:

$$C_{\min} = C_{\max} = C_{tb} = \frac{n(n+1)}{2}$$

Nhìn vào các kết quả định giá ở trên ta thấy INSERT_SORT tỏ ra có “tốt hơn” so với hai phương pháp kia. Tuy nhiên, với n khá lớn, chi phí về thời gian thực hiện được đánh giá qua cấp độ lớn, thì cả ba phương pháp đều có cấp $O(n^2)$ và đây vẫn là một chi phí cao so với một số phương pháp mà ta sẽ xét thêm sau đây:

6.3. Sắp xếp kiểu phân đoạn (Partition Sort) hay sắp xếp nhanh (Quick Sort)

6.3.1. Giới thiệu phương pháp

Sắp xếp kiểu phân đoạn là một cải tiến của phương pháp sắp xếp kiểu đổi chỗ. Đây là một phương pháp khá tốt, do đó người sáng lập ra nó C.A.R Hoare, đã mạnh dạn đặt cho nó cái tên hấp dẫn là *Quick Sort* (sắp xếp nhanh).

Ý chủ đạo của phương pháp có thể tóm tắt như sau:

Chọn một khoá ngẫu nhiên nào đó của dãy làm “chốt” (pivot). Mọi phần tử nhỏ hơn “khoá chốt” phải được xếp vào vị trí ở trước “chốt” (đầu dãy), mọi phần tử lớn hơn khoá “chốt” phải được xếp vào vị trí ở sau “chốt” (cuối dãy). Muốn vậy, các phần tử trong dãy sẽ được so sánh với khoá chốt và sẽ đổi vị trí cho nhau, hoặc cho chốt, nếu nó lớn hơn chốt mà lại nằm trước chốt hoặc nhỏ hơn chốt mà lại nằm sau chốt. Khi việc đổi chỗ đã thực hiện xong thì dãy khoá lúc đó được phân làm hai đoạn: một đoạn gồm các khoá nhỏ hơn chốt, một đoạn gồm các khoá lớn hơn chốt, còn khoá chốt thì ở giữa hai đoạn nói trên, đó cũng chính là vị trí thực của nó trong dãy khi đã được sắp xếp, tới đây coi như kết thúc một lượt sắp xếp.

Áp dụng phương pháp như trên cho mỗi đoạn đó và tiếp tục làm như vậy cho đến khi mỗi đoạn chỉ còn 1 phần tử. Khi đó toàn bộ dãy đã được sắp.

6.3.2. Ví dụ và giải thuật

Giả sử, ta qui ước chọn khoá chốt là khoá đầu tiên của dãy. Như vậy với dãy:

45 27 75 15 65 58 94 37 99 87

thì “chốt” là 45.

Ta phải làm sao để các số nhỏ hơn 45 như: 27, 15, 37 phải chuyển về phía trước, còn các số lớn hơn 45 như: 75, 65, 58, 94, 99, 87 phải nằm ở phía sau 45. Nếu được như vậy thì ta đã tách dãy khoá đã cho thành hai dãy con (hai phân đoạn), một dãy chiếm 3 vị trí đầu, một dãy chiếm 6 vị trí sau, và khoá chốt 45 sẽ được đưa vào vị trí thứ 4, nghĩa là vị trí đúng của nó trong sắp xếp. Lúc đó ta đã thực hiện xong một lượt phân đoạn.

Giả sử việc phân đoạn như trên được thực hiện bởi hàm Partition(K, t, p, j), với qui ước:

t - là chỉ số của phần tử đầu (bên trái) của dãy khoá đang xét

p - là chỉ số của phần tử cuối (bên phải) của dãy khoá đó

K- là mảng lưu trữ dãy khoá đã cho

j - là chỉ số ứng với khoá chốt sau khi đã tách dãy khoá đang xét thành hai phân đoạn

Với hàm Partition này thì hàm đệ quy thể hiện phương pháp sắp xếp nhanh có thể viết như sau:

```
Quick_Sort(K, t, p)
{
    if (t < p)
    {
        Partition(K, t, p, j);
        QuickSort(K, t, j - 1);
        QuickSort(K, j + 1, p);
    }
}
```

Như vậy, để sắp xếp dãy khoá K với n phần tử thì phải thực hiện lời gọi:

Quick_Sort(K, 0, n-1)

Hàm phân đoạn Partition là thành phần chính của thuật toán sắp xếp nhanh. Vấn đề còn lại là xây dựng hàm phân đoạn này. Ý tưởng của hàm này như sau:

Đầu tiên ta chọn K[t] làm chốt cho phân đoạn K[t..p].

Cuối cùng ta trao đổi $K[t]$ và $K[j]$ để đặt chốt vào vị trí j . Quá trình phân đoạn đến đây kết thúc.

$$j = j-1;$$

```

    }
}
while (i <= j);
K[t] ↔ K[j]; // Đổi chỗ K[t] (chốt) và K[j]
}

```

Bảng sau đây cho kết quả của sắp xếp theo QUICK-SORT sau từng lượt, đổi với dãy khoá làm ví dụ nêu trên.

Lượt	K _i : 45	27	75	15	65	58	94	37	99	87
1	(15	27	37)	45	(65	58	94	75	99	87)
2	15	(27	37)	45	(65	58	94	75	99	87)
3	15	27	(37)	45	(65	58	94	75	99	87)
4	15	27	37	45	(58)	65	(94	75	99	87)
5	15	27	37	45	58	65	(94	75	99	87)
6	15	27	37	45	58	65	(87	75)	94	(99)
7	15	27	37	45	58	65	75	(87)	94	(99)
8	15	27	37	45	58	65	75	87	94	(99)
9	15	27	37	45	58	65	75	87	94	99

6.3.3. Nhận xét và đánh giá

Trước hết ta hãy đề ý đến một vài chi tiết có ảnh hưởng tới hiệu lực của phương pháp, đồng thời cũng thể hiện rõ đặc điểm của phương pháp này.

6.3.3.1. Vấn đề chọn “chốt”

Theo giới thiệu chung thì chốt có thể chọn tùy ý trong dãy khoá đã cho, nhưng rõ ràng khi thể hiện giải thuật ta phải định ra một cách chọn khoá chốt cụ thể. Nếu chốt ta chọn rơi vào đúng khoá nhỏ nhất (hoặc lớn nhất) của phân đoạn cần xử lý, thì sau mỗi lượt ta chỉ tách ra được một phân đoạn con có kích thước nhỏ hơn trước là 1 (vì đã bớt đi một khoá chính là “chốt”), và chính phân đoạn này sẽ được xử lý tiếp theo luôn. Như vậy là ta đã quay trở lại phương pháp sắp xếp kiểu nổi bọt đơn giản. Việc chọn chốt như thế này đã dẫn đến tình huống xấu nhất của phương pháp.

Nếu gọi “*trung vị*” (median) của một dãy khoá là khoá sẽ đứng ở giữa dãy đó sau khi dãy đã được sắp xếp, nghĩa là nó lớn hơn một nửa số khoá của dãy và nhỏ hơn số còn lại, thì tốt nhất vẫn là chọn được đúng trung vị làm “chốt”. Lúc đó sau mỗi lượt ta sẽ tách ra được hai phân đoạn con có độ dài gần như nhau và phân đoạn xử lý tiếp theo có kích thước chỉ bằng “nửa” phân đoạn đã chứa nó.

Nhưng làm sao có thể chọn được đúng trung vị? Nếu giả thiết: sự xuất hiện của các khoá trong dãy là đồng khả năng thì trung vị có thể là bất kỳ một khoá nào trong dãy. Trong giải thuật trên, ta chọn khoá đứng đầu làm chốt là dựa trên cơ sở này. Nhưng với cách chọn này, nếu dãy khoá có khuynh hướng đã theo thứ tự sắp xếp thì khả năng xấu nhất lại xuất hiện. Tuy nhiên nếu khuynh hướng này hay xuất hiện thì việc lựa chọn khoá đang được đứng ở giữa dãy lại gặp thuận lợi.

Để dung hoà với cách chọn như trên, đồng thời cũng để kết hợp với một đề nghị sau này của Hoare là: “*chọn trung vị của một dãy khoá nhỏ hơn, thuộc dãy khoá đã cho làm chốt*”, R.C Singleton đã đưa ra một cách chọn là:

Chọn k_q là “chốt” với K_q là trung vị của ba khoá K_t , $K_{[(t+p)/2]}$ và K_p , trong đó t và p là chỉ số của khoá đầu và khoá cuối của phân đoạn đang xét. Các kiểu chọn khoá chốt còn được nhiều tác giả khác nữa đưa ra và cũng có nhiều kết quả đáng chú ý.

6.3.3.2. Vấn đề phối hợp với cách sắp xếp khác

Khi kích thước của các phân đoạn đã khá nhỏ, việc tiếp tục phân đoạn nữa theo QUICK-SORT thực ra sẽ không có lợi. Lúc đó sử dụng một phương pháp sắp xếp đơn giản lại tiện hơn. Vì vậy, QUICK-SORT thường không tiến hành triệt để mà dừng lại ở lúc cần thiết để gọi tới một phương pháp sắp xếp đơn giản, giao cho nó tiếp tục thực hiện sắp xếp với các phân đoạn nhỏ còn lại. Kunth (1974) có nêu: 9 có thể coi là kích thước giới hạn của phân đoạn để sau đó QUICK-SORT gọi tới các phương pháp sắp xếp đơn giản.

Ngoài ra vấn đề chọn phân đoạn nào để xử lý tiếp theo cũng là vấn đề cần được xem xét tới. Tuy nhiên ta sẽ không đi sâu vào ở đây.

6.3.3.3. Đánh giá giải thuật

Do giải thuật là đệ quy nên ta sẽ áp dụng một cách tiếp cận khác một chút.

Gọi $T(n)$ là thời gian thực hiện giải thuật ứng với một mảng n khoá, $P(n)$ là thời gian để phân đoạn một mảng n khoá thành hai mảng con. Ta có thể viết:

$$T(n) = P(n) + T(j-t) + T(p-j)$$

Chú ý rằng $P(n) = Cn$ với C là một hằng số.

Trường hợp xấu nhất xảy ra khi mảng khoá vốn đã có thứ tự sắp xếp: sau khi phân đoạn một trong hai mảng con là rỗng ($j = t$ hoặc $j = p$)

Giả sử $j = t$, ta có

$$\begin{aligned}T_{\text{xấu}}(n) &= P(n) + T_{\text{xấu}}(0) + T_{\text{xấu}}(n-1) \\&= Cn + T_{\text{xấu}}(n-1) \\&= Cn + C(n-1) + T_{\text{xấu}}(n-2) \\&\dots \\&= \sum_{k=1}^n Ck + T_{\text{xấu}}(0) \\&= C \cdot \frac{n(n+1)}{2} = O(n^2) \quad (T_{\text{xấu}}(0)=0)\end{aligned}$$

Trường hợp này QUICK-SORT không hơn gì các phương pháp đã nêu trước đây.

Trường hợp tốt nhất xảy ra khi mảng luôn luôn được chia đôi, nghĩa là:

$$j = \frac{t+p}{2}. \text{ Lúc đó:}$$

$$\begin{aligned}T_{\text{tốt}}(n) &= P(n) + 2T_{\text{tốt}}(n/2) \\&= Cn + 2T_{\text{tốt}}(n/2) \\&= Cn + 2C(n/2) + 4T_{\text{tốt}}(n/4) = 2Cn + 2^2T_{\text{tốt}}(n/4) \\&= Cn + 2C(n/2) + 4C(n/4) + 8T_{\text{tốt}}(n/8) = 2^3Cn + 2^3T_{\text{tốt}}(n/8) \\&\dots \\&= (\log_2 n)Cn + 2^{\log_2 n}T_{\text{tốt}}(1) \\&= O(n\log_2 n)\end{aligned}$$

Việc xác định giá trị trung bình $T_{\text{tb}}(n)$ không còn đơn giản như hai trường hợp trên, nên ta sẽ không xét chi tiết. Kết quả mà ta cần ghi nhận là: người ta đã chứng minh được:

$$T_{\text{tb}}(n) = O(n\log_2 n)$$

Như vậy rõ ràng là, khi n khá lớn, QUICK-SORT đã tỏ ra có hiệu lực hơn hẳn 3 phương pháp đã nêu.

6.4. Sắp xếp kiểu vun đống (Heap sort)

Sắp xếp kiểu phân đoạn đã cho ta thời gian thực hiện trung bình khá tốt, nhưng trường hợp xấu của nó vẫn là $O(n^2)$.

Phương pháp sắp xếp mà ta sẽ xét sau đây đã đảm bảo được trong cả hai trường hợp chi phí thời gian đều cùng là $O(n \log_2 n)$.

Với phương pháp sắp xếp này, dãy khoá sẽ có cấu trúc cây nhị phân hoàn chỉnh và được cài đặt bằng mảng (xem 5.2.2.1)

6.4.1. Giới thiệu phương pháp

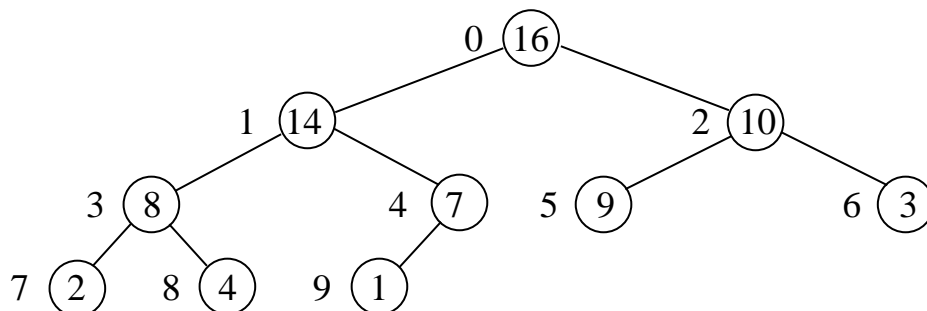
Cấu trúc cây đặc biệt được sử dụng trong phương pháp này được gọi là *đống*.

6.4.1.1. Định nghĩa “đống”

Đống là một cây nhị phân hoàn chỉnh mà mỗi nút được gán một giá trị khoá sao cho khoá ở nút cha bao giờ cũng lớn hơn khoá ở nút con của nó.

Đống được cài đặt trên máy bởi một mảng K, trong đó $K[i]$ lưu trữ giá trị khoá ở nút thứ i trên cây nhị phân hoàn chỉnh (theo cách đánh số thứ tự như đã nói ở mục 5.2.2.1).

Ví dụ: Cây ở hình 9.2 dưới đây là một đống với mảng tương ứng K.



Hình 6.1. Minh họa một cây nhị phân là đống

K	16	14	10	8	7	9	3	2	4	1
	0	1	2	3	4	5	6	7	8	9

Ta thấy ngay rằng: nếu ta có một đống thì giá trị khoá ở nút gốc (mà ta sẽ gọi là “đỉnh đống” - nút được đánh số 0) chính là giá trị khoá lớn nhất trong dãy khoá ứng với đống đó.

Từ đó có thể hiểu: việc chọn ra khoá lớn nhất trong dãy khoá đã cho sẽ thuận lợi hơn nếu ta tạo được ra một đống ứng với dãy khoá này.

Ta đã biết: một cây nhị phân hoàn chỉnh có thể được cài đặt (biểu diễn) bằng mảng.

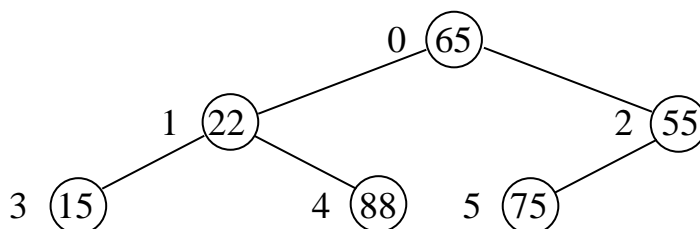
Vậy thì ngược lại: Một mảng biểu diễn một dãy giá trị khoá cũng có thể coi nó như dạng biểu diễn của một cây nhị phân hoàn chỉnh mà mỗi nút có gán một giá trị khoá tương ứng của dãy.

Dĩ nhiên cây này có thể chưa phải là đồng.

Ví dụ: Nếu có dãy khoá được lưu trữ trong mảng K như sau:

K	65	22	55	15	88	75
	0	1	2	3	4	5

thì có thể coi K là mảng biểu diễn của cây nhị phân hoàn chỉnh có dạng:



Hình 6.2. Dựng cây nhị phân hoàn chỉnh từ mảng

Rõ ràng cây này chưa phải là đồng.

6.4.1.2. Phép tạo đồng

Xét một cây nhị phân hoàn chỉnh có n nút, mà mỗi nút đã được gán một giá trị khoá. Cây này chưa phải là đồng.

Muốn tạo nó thành đồng thì làm thế nào?

Trước hết ta có nhận xét như sau:

- Nếu một cây nhị phân hoàn chỉnh đã là đồng thì các cây con của các nút (nếu có) cũng là cây nhị phân hoàn chỉnh và cũng là đồng.
- Trên cây nhị phân hoàn chỉnh có n nút thì chỉ có $\lfloor n/2 \rfloor$ nút được là “cha”.
- Một nút lá bao giờ cũng có thể coi là đồng.

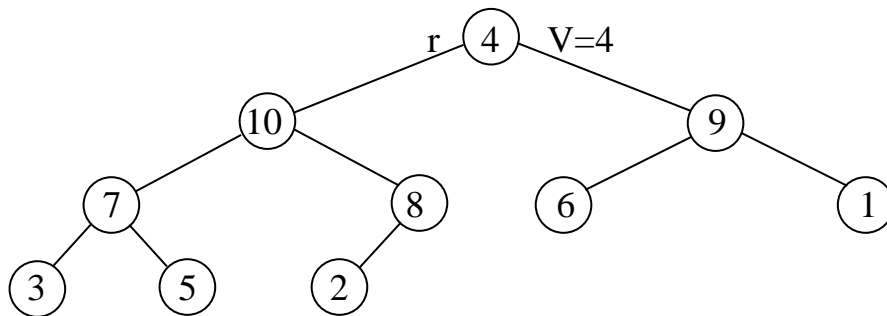
Từ đó ta thấy: có thể thực hiện phép tạo đồng theo kiểu “từ đáy lên” (bottom up), nghĩa là từ các cây con mà gốc có số thứ tự là: $\lfloor (n-2)/2 \rfloor, \lfloor (n-2)/2 \rfloor - 1, \lfloor (n-2)/2 \rfloor - 2, \dots, 0$. Ta sẽ bắt đầu từ những cây mà con của nó là lá, nghĩa là đã là đồng. Do đó việc tạo đồng này sẽ chỉ cần một phép xử lý chung, mà ta có thể phát biểu như sau:

“Hãy tạo thành đồng cho một cây nhị phân hoàn chỉnh có gốc r , và gốc có 2 cây con đã là đồng rồi”.

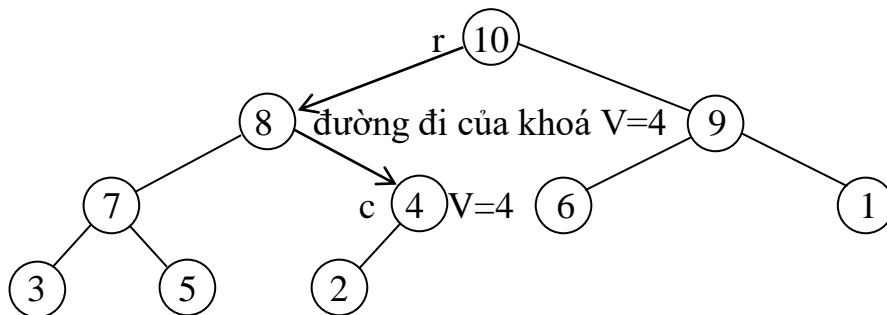
Ý tưởng thực hiện công việc này như sau:

Giải sử nút r chứa giá trị V , xuất phát từ r ta cứ đi tới nút con chứa giá trị lớn nhất trong hai 2 nút con, cho tới khi nào gặp phải một nút c mà 2 nút con của nó đều chứa giá trị $< V$ (nút lá cũng là trường hợp riêng của điều kiện này). Dọc trên đường đi từ r tới c , ta đẩy giá trị ở nút con lên nút cha, cuối cùng đặt giá trị V vào nút c .

Có thể minh hoạ ý tưởng trên bằng hình 6.3 và hình 6.4.



Hình 6.3. Cây nhị phân chưa là đồng nhưng có hai con đã là đồng



Hình 6.4. Quá trình tạo thành đồng của cây nhị phân

Ta có thể thể hiện ý tưởng trên bằng giải thuật sau:

ADJUST (root, n)

{

/*giải thuật toán này thực hiện việc chỉnh lý một cây nhị phân với gốc root để nó thoả mãn được điều kiện của đồng. Cây con trái và cây con phải của root, nghĩa là cây với gốc $2i+1$ và $2i+2$, đã thoả mãn điều kiện của đồng. Cây được lưu trữ trên mảng K có n phần tử được đánh số từ 0
/*

Key= $K[\text{root}]$; // Key nhận giá trị khoá ở nút gốc

```

while (root*2 <= n-2) // Chừng nào root chưa phải là lá
{
    c = root * 2+1; // Xét nút con trái của root
    if ((c < n-1) && (K[c] < K[c+1])) //nếu con phải lớn hơn con trái
        c = c+1; // chọn ra nút có giá trị lớn nhất
    if (K[c] < Key) // Đã tìm thấy vị trí của Key
        break; // thì dừng
    K[root] = K[c]; // Chuyển giá trị từ nút con c lên nút cha root
    root = c; // đi xuống xét nút con c
}
K[root] = Key; // Đặt giá trị Key vào nút root
}

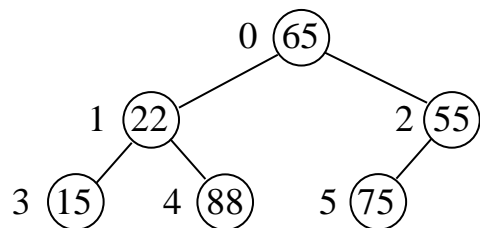
```

Với giải thuật ADJUST này, việc tạo thành đồng cho một cây nhị phân hoàn chỉnh có n nút sẽ được thực hiện bởi:

For ($i = \lfloor (n-2)/2 \rfloor$; $i \geq 0$; $i--$)

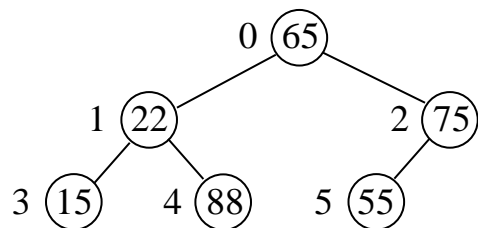
ADJUST (i, n)

Ví dụ: Hình 6.5 sau đây minh họa diễn biến của cây ở hình 6.2, trong quá trình tạo thành đồng:



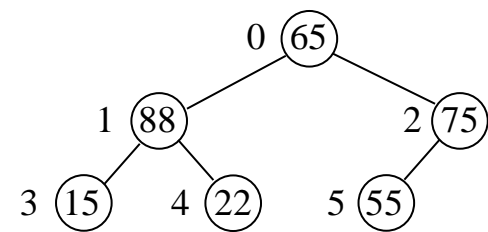
65	22	55	15	88	75
0	1	2	3	4	5

a) Cấu trúc dữ liệu và cấu trúc lưu trữ ban đầu



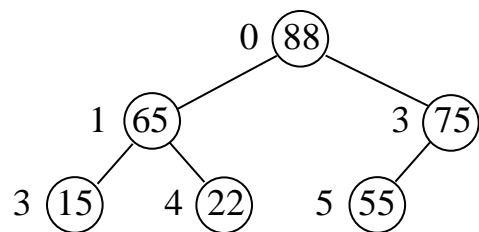
65	22	75	15	88	55
0	1	2	3	4	5

b) Sau khi thực hiện ADJUST (2, 6)



65	88	75	15	22	55
0	1	2	3	4	5

c) Sau khi thực hiện
ADJUST (1, 6)



88	65	75	15	22	55
0	1	2	3	4	5

d) Sau khi thực hiện ADJUST (0, 6)
Cây đã là đồng, khoá lớn nhất
đang ở đỉnh đồng

Hình 6.5. Minh họa quá trình tạo thành đồng của cây

6.4.1.3. Sắp xếp kiểu vun đồng

Sắp xếp kiểu vun đồng là một cải tiến của phương pháp sắp xếp kiểu lựa chọn. Tuy nhiên để chọn ra số lớn nhất, ở đây người ta đã dựa vào cấu trúc đồng và để sắp xếp theo thứ tự tăng dần của các giá trị khoá là số, như ta đã qui ước, thì khoá lớn nhất sẽ được xếp vào cuối dãy, nghĩa là nó được đổi chỗ với khoá đang ở “đáy đồng”, và sau phép đổi chỗ này một khoá trong dãy đã vào đúng vị trí của nó trong sắp xếp. Nếu không kể tới khoá này thì phần còn lại của dãy khoá ứng với một cây nhị phân hoàn chỉnh, với số lượng khoá nhỏ hơn 1, sẽ không còn là đồng nữa, ta lại gặp bài toán tạo đồng mới cho cây này (ta sẽ gọi là “vun đồng”) và lại thực hiện tiếp phép đổi chỗ giữa khoá ở đỉnh đồng và khoá ở đáy đồng tương tự như đã làm Cho tới khi cây chỉ còn là 1 nút thì các khoá đã được sắp xếp vào đúng vị trí của nó trong sắp xếp.

Như vậy sắp xếp kiểu vun đồng bao gồm 2 giai đoạn:

1- Giai đoạn tạo đồng ban đầu (như đã nêu ở mục 6.4.1.2)

2- Giai đoạn sắp xếp, bao gồm 2 bước:

- Đổi chỗ
- Vun đồng



được thực hiện (n-1) lần.

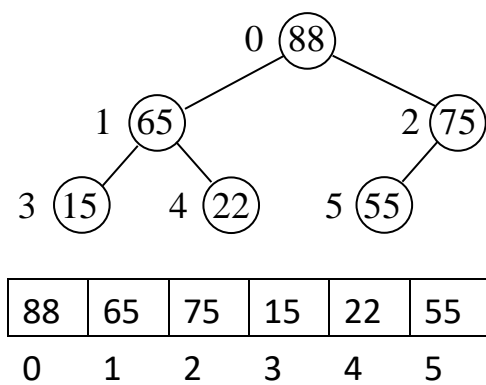
Hàm sắp xếp kiểu vun đồng được thể hiện bởi giải thuật sau:

HEAP_SORT (K, n)

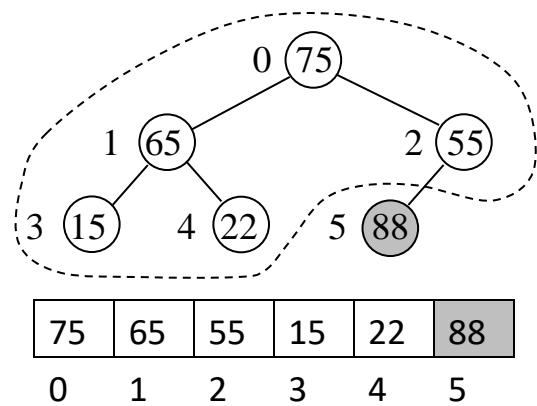
```
{
  for (i = (n-2)/2; i >= 0; i--) // 1- Tạo đống ban đầu
    ADJUST (i, n)
  for (i = n-1; i > 0; i--) // 2- Sắp xếp gồm: đổi chỗ và vun đống
    {
      x = K[0];
      K[0] = K[i];
      K[i] = x;
      ADJUST (0, i)
    }
}
```

Ví dụ: Hình 6.6 sau đây minh hoạ giai đoạn sắp xếp đối với đống đã được tạo trong ví dụ ở mục 6.4.1.2. Trong đó cần chú ý:

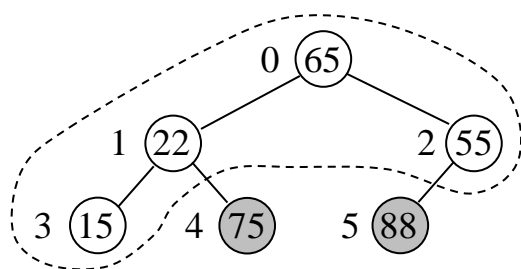
- * Hình   chỉ các khoá đã vào đúng vị trí của nó trong sắp xếp.
- * Dấu - - - bao quanh các nút ứng với cây đã được vun đống.



a) Đống được tạo ra sau khi thực hiện giai đoạn 1



b) Sau khi đổi chỗ và thực hiện ADJUST (0, 5)

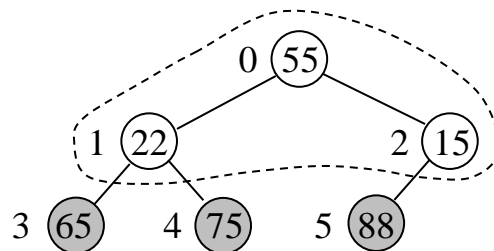


65	22	55	15	75	88
----	----	----	----	----	----

0 1 2 3 4 5

c) Sau khi đổi chỗ và thực hiện

ADJUST (0, 4)

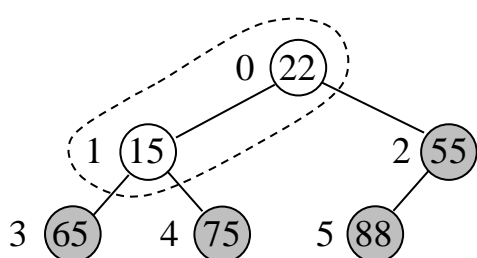


55	22	15	65	75	88
----	----	----	----	----	----

0 1 2 3 4 5

d) Sau khi đổi chỗ và thực hiện

ADJUST (0, 3)

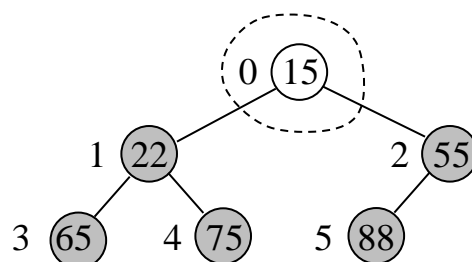


22	15	55	65	75	88
----	----	----	----	----	----

0 1 2 3 4 5

e) Sau khi đổi chỗ và thực hiện

ADJUST (0, 2)



15	22	55	65	75	88
----	----	----	----	----	----

0 1 2 3 4 5

f) Sau khi đổi chỗ và thực hiện

ADJUST (0, 1), lúc này các khoá đã được sắp xếp.

Hình 6.6. Minh hoạ quá trình sắp xếp HEAP-SORT

6.4.2. Nhận xét và đánh giá

Đối với HEAP-SORT, người ta cũng chứng minh được thời gian thực hiện trung bình $T_{tb}(n) = O(n \log_2 n)$

Ở đây ta chú ý tới việc đánh giá thời gian thực hiện trong trường hợp xấu nhất:

- Có thể thấy rằng ở giai đoạn 1 (tạo đống ban đầu) có $\lfloor n/2 \rfloor$ lần gọi thực hiện ADJUST (i, n). Còn ở giai đoạn 2 (sắp xếp) thì phải gọi thực hiện ADJUST (0, i) (n-1) lần. Như vậy có thể coi như phải gọi khoảng $\frac{3}{2}n$ lần thực hiện hàm ADJUST.

- Ta thấy cây được xét ứng với ADJUST thì nhiều nhất cũng chỉ có n nút, mà một cây nhị phân hoàn chỉnh có n nút thì chiều cao của cây đó là $\lceil \log_2(n + 1) \rceil$, nghĩa là chiều cao của cây lớn nhất cũng chỉ xấp xỉ $\log_2 n$.

- Số lượng so sánh giá trị khoá khi thực hiện hàm ADJUST, cùng lắm cũng chỉ bằng chiều cao của cây tương ứng.

- Cho nên, có thể nói rằng: cùng lắm thì số lượng phép so sánh cũng chỉ xấp xỉ:

$$\frac{3}{2}n \log_2 n$$

- Từ đó suy ra:

$$T_x(n) = O(n \log_2 n)$$

Đây chính là ưu điểm của HEAP-SORT so với QUICK-SORT

Còn về thời gian thực hiện trung bình thì, như ta đã nêu, cả hai phương pháp đều như nhau.

Có thể nhận xét thêm là: QUICK-SORT còn phải dùng thêm không gian nhớ cho ngăn xếp, để bảo lưu thông tin về các phân đoạn sẽ được xử lý tiếp theo (vì thực hiện đệ quy). Còn HEAP-SORT thì ngoài một nút nhớ phụ, để thực hiện đổi chỗ, nó không cần thêm gì nữa.

6.5. Sắp xếp theo kiểu hoà nhập (Merge-sort)

Bây giờ ta xét tới một phương pháp có vai trò khá đặc biệt do ở chỗ nó dựa trên một phép xử lý đơn giản hơn sắp xếp để thực hiện sắp xếp, đó là phép hoà nhập.

6.5.1. Phép hoà nhập hai đường

Hoà nhập hai đường là phép hợp nhất hai dãy khoá đã được sắp xếp để ghép lại thành một dãy khoá có kích thước bằng tổng kích thước của hai dãy khoá ban đầu, và dãy khoá tạo thành cũng có thứ tự sắp xếp. Nguyên tắc thực hiện nó khá đơn giản: so sánh hai khoá nhỏ nhất (hoặc lớn nhất) của hai dãy con, chọn khoá nhỏ hơn (hoặc lớn hơn) để đưa ra miền sắp xếp (một miền nhớ phụ có kích thước bằng tổng kích thước của hai dãy con) và đặt nó vào vị trí thích hợp. Khoá được chọn từ đấy bị loại ra khỏi dãy chứa nó. Quá trình như vậy cứ tiếp tục cho tới khi một trong hai dãy đã cạn. Lúc đó chỉ cần chuyển toàn bộ phần cuối của dãy còn lại ra miền sắp xếp là xong.

Ví dụ: Với hai dãy khoá đã sắp xếp: (2, 7, 20, 30) và (5, 9, 15)

Quá trình hoà nhập sẽ như bảng sau:

Dãy 1	Dãy 2	Lượt	Khoá nhỏ nhất trong hai dãy	Miền sắp xếp
(2, 7, 20, 30)	(5, 9, 15)	1	2	(2)
(7, 20, 30)	(5, 9, 15)	2	5	(2, 5)
(7, 20, 30)	(9, 15)	3	7	(2, 5, 7)
(20, 30)	(9, 15)	4	9	(2, 5, 7, 9)
(20, 30)	(15)	5	15	(2, 5, 7, 9, 15)
(20, 30)	∅	6	Dãy 2 đã ∅, đưa nốt dãy 1 vào miền sắp xếp	(2, 5, 7, 9, 15, 20, 30)

Để tiện cho việc xây dựng các giải thuật tiếp theo ta sẽ giả thiết phép hoà nhập hai đường được thực hiện đối với hai mảng con (đã được sắp xếp) là hai phân đoạn kế tiếp của một mảng lớn. Điều đó có nghĩa là hợp nhất hai mảng con (K_b, \dots, K_m) và (K_{m+1}, \dots, K_n) của một mảng lớn K thành một mảng mới X gồm các phần tử (X_b, \dots, X_n) được sắp xếp.

Sau đây là giải thuật hoà nhập hai đường.

MERGE (K, b, m, n, X)

{/* Hàm này thực hiện hoà nhập hai mảng con đã được sắp xếp có tính chất như đã nêu thành một mảng mới X được sắp xếp*/}

$i = t = b; j = m + 1;$

while ($i \leq m$) and ($j \leq n$) // Chừng nào hai mảng con vẫn còn khoá

if ($K[i] < K[j]$) // So sánh để chọn khoá nhỏ hơn

$X[t++] = K[i++];$

else

$X[t++] = K[j++];$

// Một trong hai mảng con đã hết khoá trước

if ($i > m$) // Mảng 1 hết khoá trước

for (; $j \leq n$;)

$X[t++] = K[j++];$

```

else // Mảng 2 hết khoá trước
    for (; i<=m; )
        X[t++] = K[i++];
    }

```

Chú thích:

1) Giải thuật thực hiện phép hoà nhập hai đường cho hai mảng con độc lập (X_1, \dots, X_m) và (Y_1, \dots, Y_n) để cho một mảng mới, được sắp xếp (Z_1, \dots, Z_{m+n}) cũng được thực hiện tương tự.

2) Mảng con đã có thứ tự sắp xếp thường được gọi là một *mạch* (run).

6.5.2. Sắp xếp kiểu hoà nhập hai đường trực tiếp

Từ phép hoà nhập hai đường người ta đã triển khai thành một số phương pháp sắp xếp mới. Ta hãy xuất phát từ một nhận xét như sau: mỗi khoá có thể coi là một mạch có độ dài (kích thước) bằng 1 và đã được sắp xếp. Nếu hoà nhập hai mạch như vậy ta sẽ được một mạch mới, có độ dài bằng 2. Lại hoà nhập hai mạch có độ dài 2, ta được một mạch có độ dài 4, và cứ như thế, cuối cùng ta sẽ được một mạch có độ dài n, tức là mảng đã được sắp xếp hoàn toàn.

Ví dụ:

Độ dài mạch	Kết quả hoà nhập các cặp mạch liên tiếp											
1	<u>[45] [27]</u>		<u>[75] [15]</u>		<u>[65] [58]</u>		<u>[94] [37]</u>		<u>[99] [87]</u>			
2	<u>[27 45] [15 75]</u>				<u>[58 65] [37 94]</u>				[87 99]			
4	<u>[15 27 45 75] [37 58 65 94]</u>								[87 99]			
8	<u>[15 27 37 45 58 65 75 94]</u>										[87 99]	
	[15 27 37 45 58 65 75 87 94 99]											

Sau đây là các hàm thực hiện phép sắp xếp này

MPASS (K, X, n, h)

/* Hàm này thực hiện một bước của sắp xếp kiểu hoà nhập. Nó hoà nhập từng cặp mạch kế cận nhau, có độ dài h từ mảng K sang mảng X, n là số lượng khoá trong K */

```

i = 0;
while (i + 2*h <= n) // Hoà nhập cặp mạch có độ dài h
{
    MERGE (K, i, i + h - 1, i + 2*h - 1, X);
    i = i + 2*h;
}
if (i + h < n) // Hoà nhập cặp mạch còn lại với tổng độ dài < 2*h
    MERGE (K, i, i + h - 1, n-1, X);
else // Chỉ còn một mạch
    for (; i < n; i++)
        X[i] = K[i];
}

```

Hàm MPASS trên sẽ được gọi tới trong hàm sắp xếp sau đây:

STRAIGHT_MSORT (K, n)

```

/* Để thực hiện lưu trữ các mạch mới, khi hoà nhập, ở đây phải dùng một
   mảng phụ X[n], K và X sẽ được dùng luân phiên, h ghi nhận kích thước
   của các mạch, sau mỗi lượt h được tăng gấp đôi */
h = 1;
while (h < n)
{
    MPASS (K, X, n, h); // Hoà nhập và chuyển các khoá vào X
    MPASS (X, K, n, h*2); // Hoà nhập và chuyển các khoá vào K
    h = 4*h;
}
}

```

Chú ý:

1) Phương pháp trên dựa vào phép hoà nhập hai mạch kế cận nhau, nhưng cách chọn này không phải là duy nhất, chẳng hạn có thể chọn hai mạch con theo

hướng tiến từ hai đầu vào. Từ những cách chọn khác nhau có thể đưa tới những giải thuật sắp xếp kiểu hoà nhập hai đường trực tiếp khác nhau.

2) Hoà nhập kiểu hai đường trực tiếp đã không tận dụng được khả năng “vốn đã được sắp xếp tự nhiên rồi” của các khoá trong mảng lúc ban đầu. Ở mỗi lượt nó thực hiện hoà nhập hai mạch với độ dài đã ấn định: ở lượt thứ k thì độ dài hai mạch được hoà nhập là 2^{k-1} , cho dù lúc đó mạch được hoà nhập có thể có độ dài lớn hơn cũng thế. Như ta đã biết, khi hoà nhập không nhất thiết hai mạch phải có kích thước bằng nhau cho nên nếu triệt để lợi dụng các mạch vốn đã hình thành sẵn rồi (mà ta có thể gọi là một *mạch tự nhiên*) với kích thước càng lớn, thì sắp xếp có khả năng kết thúc với số lượt ít hơn. Đây cũng là ý cơ bản của một phương pháp sắp xếp khác gọi là sắp xếp kiểu hoà nhập hai đường tự nhiên (natural two-way merge). Với dãy khoá làm ví dụ ở trên, ngay từ đầu ta đã có 6 mạch, phần lớn đã có độ dài 2. Ta có thể thực hiện hoà nhập như sau:

Lượt	Kết quả hoà nhập các cặp mạch liên tiếp									
1	[45]	[27	75]	[15	65]	[58	94]	[37	99]	[87]
2	[27	45	75]	[15	58	65	94]	[37	87	99]
3	[15	27	45	58	65	75	94]	[37	87	99]
	[15	27	37	45	58	65	75	87	94	99]

Rõ ràng số lượt ở đây đã giảm đi 1.

6.5.3. Phân tích đánh giá

Có thể thấy ngay trong phương pháp sắp xếp kiểu hoà nhập số lượng phép chuyển chỗ thường vượt số lượng các phép so sánh. Ví dụ ở hàm MERGE, trong câu lệnh while ứng với một phép so sánh thì có một phép chuyển chỗ, nhưng nếu một mạch nào đó kết thúc sớm thì cả phần đuôi của mạch còn lại được chuyển sang mảng sắp xếp mà không tương ứng với một phép so sánh nào nữa. Vì vậy đối với phương pháp này, ta lại chọn phép tích cực là phép chuyển chỗ để làm căn cứ đánh giá thời gian thực hiện giải thuật.

Ta thấy ở lượt sắp xếp nào (hàm MPASS) thì toàn bộ các khoá cũng được chuyển sang mảng mới (từ K sang X hoặc từ X sang K), như vậy chi phí thời gian cho một lượt có cấp $O(n)$. Ngoài ra nếu để ý sẽ thấy: số lượt gọi MPASS trong giải thuật STRAIGHT_MSORT chính là $\lceil \log_2 n \rceil$. Sở dĩ như vậy là vì: ở lượt 1 kích

thước của mảng con là $l = 2^i$. Ở lượt i kích thước mảng con sẽ là 2^{i-1} mà sau lượt gọi cuối cùng thì mảng con đã có kích thước bằng n .

Vậy sắp xếp kiểu hoà nhập hai đường trực tiếp có cấp thời gian là $O[n\log_2 n]$.

Một đặc điểm khác và cũng là nhược điểm của phương pháp sắp xếp kiểu hoà nhập là chi phí về không gian khá lớn. Nó đòi hỏi tới $2n$ phần tử nhớ, gấp đôi so với các phương pháp thông thường. Do đó người ta thường sử dụng phương pháp này khi sắp xếp ngoài, đối với các tệp.

6.6. Những nhận xét cuối cùng

Qua những giải thuật nêu trên ta đã thấy rõ: cùng một mục đích sắp xếp như nhau mà có rất nhiều phương pháp và kỹ thuật giải quyết khác nhau. Cấu trúc dữ liệu được lựa chọn để hình dung đối tượng của sắp xếp đã ảnh hưởng rất sát tới giải thuật xử lý.

Các phương pháp sắp xếp đơn giản đã thể hiện ba kỹ thuật cơ sở của sắp xếp (dựa vào phép so sánh giá trị khoá), cấp độ lớn của thời gian thực hiện chung là $O(n^2)$, vì vậy chỉ nên sử dụng chúng khi n nhỏ. Các giải thuật cải tiến như QUICK_SORT, HEAP_SORT đã đạt được hiệu quả cao (có cấp độ lớn của thời gian thực hiện là $O[n\log_2 n]$), nên thường được sử dụng khi n lớn. MERGE_SORT cũng không kém hiệu lực về thời gian thực hiện nhưng về không gian thì đòi hỏi của nó không thích nghi với sắp xếp trong. Nếu bảng cần sắp xếp vốn có khuynh hướng hầu như “đã được sắp sẵn” thì QUICK_SORT lại không nên dùng. Nhưng nếu ban đầu bảng có khuynh hướng ít nhiều có thứ tự ngược với thứ tự sắp xếp thì HEAP_SORT lại tỏ ra thuận lợi. Việc khẳng định một kỹ thuật sắp xếp nào vừa nói trên luôn luôn tốt hơn mọi kỹ thuật khác là một điều không nên. Việc chọn một phương pháp sắp xếp thích hợp thường tuỳ thuộc vào từng yêu cầu, từng điều kiện cụ thể.

Bài tập chương 6

6.1. Cho dãy khoá

97 15 27 45 21 59 62 83 25 65

Hãy minh hoạ quá trình thực hiện của ba phương pháp sắp xếp cơ bản đã nêu đối với dãy khoá này theo thứ tự tăng dần, giảm dần.

6.2. Một giải thuật sắp xếp được gọi là ổn định (stable) nếu thứ tự tương đối giữa các đối tượng có khoá bằng nhau vẫn không thay đổi sau khi thực hiện sắp xếp. Hãy xem các giải thuật SELECT_SORT và BUBBLE_SORT có thể sửa đổi

để áp dụng cho dãy khoá có chứa khoá bằng nhau không? Lúc đó chúng có ổn định không?

6.3. Hãy sửa lại ba giải thuật sắp xếp đơn giản để thực hiện sắp xếp theo thứ tự giảm dần.

6.4. Với dãy khoá cho ở bài 6.1, hãy minh hoạ hai phương pháp sắp xếp kiểu phân đoạn và kiểu vun đồng.

6.5. Nhận xét và so sánh về cấu trúc dữ liệu và cấu trúc lưu trữ của dữ liệu ứng với hai phương pháp sắp xếp kiểu phân đoạn và kiểu vun đồng.

6.6. Nếu áp dụng cách chọn “chốt” theo kiểu của Singleton thì đối với dãy khoá làm ví dụ nêu trong bài, thì “chốt” sẽ là khoá nào? Hãy thực hiện sắp xếp với cách chọn khoá “chốt” này với dãy đó.

6.7. Nếu thực hiện sắp xếp theo thứ tự giảm dần thì đồng sẽ được định nghĩa thế nào cho thích hợp?

6.8. Hãy giải thích tại sao khi hoà nhập hai đường với S mạch ban đầu thì số lượt sẽ là $\lceil \log_2 S \rceil$.

6.9. Có thể phát triển phép hoà nhập hai đường để thực hiện hoà nhập k đường được không ($k > 2$).

6.10. Hãy thực hiện sắp xếp kiểu hoà nhập hai đường tự nhiên với dãy khoá cho ở bài 6.1.

CHƯƠNG 7

TÌM KIẾM

7.1. Bài toán tìm kiếm

Tìm kiếm (Searching) là yêu cầu thường xuyên trong đời sống hàng ngày cũng như trong các ứng dụng về công nghệ thông tin. Ngay trong chương trước ta cũng thấy xuất hiện các yêu cầu về tìm kiếm. Tuy nhiên, lúc đó vấn đề này đã được xét và giải quyết trong mối quan hệ mật thiết với phép xử lý chính là sắp xếp. Còn trong chương này, bài toán tìm kiếm sẽ được đặt ra một cách độc lập và tổng quát, không liên quan đến mục đích xử lý cụ thể nào khác. Ta có thể phát biểu như sau:

Cho một dãy có n phần tử (có thể coi mỗi phần tử là một cấu trúc): S_0, S_1, \dots, S_{n-1} . Mỗi phần tử S_i ($0 \leq i \leq n-1$) tương ứng với một khóa K_i . Hãy tìm phần tử có giá trị khóa bằng x cho trước.

x được gọi là khóa tìm kiếm hay giá trị tìm kiếm.

Công việc tìm kiếm sẽ hoàn thành khi có một trong hai khả năng sau xảy ra:

1) Tìm được phần tử có giá trị khóa bằng x , lúc này phép tìm kiếm thành công (successfull).

2) Không có phần tử nào trong dãy có khóa bằng x , phép tìm kiếm không thành công (unsuccessfull). Sau một phép tìm kiếm không thành công. Có thể xuất hiện yêu cầu bổ sung thêm phần tử mới có khóa bằng x vào dãy. Giải thuật thể hiện cả yêu cầu này được gọi là giải thuật *tìm kiếm có bổ sung*.

Tương tự như sắp xếp, khóa của mỗi phần tử chính là đặc điểm nhận biết của phần tử trong tìm kiếm, ta sẽ coi khóa của phần tử là đại diện cho phần tử đó và trong các giải thuật, trong các ví dụ ta cũng chỉ nói tới khóa. Để cho tiện, ta cũng coi dãy khóa K_i ($0 \leq i \leq n-1$) là các số khác nhau.

Trong chương này ta chỉ xét tới các giải thuật tìm kiếm cơ bản và thông dụng, được áp dụng để tìm kiếm với dữ liệu được lưu trữ ở bộ nhớ trong, nên gọi là *tìm kiếm trong*.

7.2. Tìm kiếm tuần tự (Sequential Searching)

7.2.1. Ý tưởng và giải thuật

Tìm kiếm tuần tự là một kỹ thuật tìm kiếm rất đơn giản và cổ điển. Nội dung có thể tóm tắt như sau: Bắt đầu từ phần tử thứ nhất, lần lượt so sánh khóa tìm kiếm

với khóa tương ứng của các phần tử trong dãy, cho tới khi tìm được phần tử mong muốn hoặc đã hết dãy mà chưa tìm thấy.

Giải thuật tìm kiếm tuần tự sau đây sẽ trả về giá trị i nếu tại đó K_i bằng x , và trả lại giá trị -1 nếu x không có trong dãy.

```
Sequential_search(K, n, x)
{ // K là mảng có n khoá
  for (i=0; i<n ; i++)
    if (K[i] == x)
      return i; // Tìm thấy
  return -1; // Không tìm thấy
}
```

7.2.2. Đánh giá giải thuật

Trong giải thuật trên, phép toán tích cực là phép toán so sánh để tìm ra x . Chúng ta sẽ ước lượng độ phức tạp của giải thuật tìm kiếm thông qua số lần thực hiện phép toán này. Có các khả năng sau xảy ra:

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	1	Phần tử đầu tiên có giá trị bằng x
Xấu nhất	n	Phần tử cuối cùng có giá trị bằng x hoặc không có x trong mảng.
Trung bình	$\frac{n+1}{2}$	Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau.

Như vậy, cả trường hợp xấu nhất cũng như trung bình, cấp độ lớn của thời gian thực hiện giải thuật trên là $O(n)$.

7.3. Tìm kiếm nhị phân

Giải thuật tìm kiếm tuần tự đã xét ở trên không phụ thuộc vào các phần tử dữ liệu trong mảng có được sắp xếp hay không. Do đó, đây là phương pháp tổng quát nhất để tìm kiếm trên một dãy số bất kỳ. Tuy nhiên trong thực tế, hầu hết dữ liệu của chúng ta đã được sắp xếp theo một trình tự nào đó. Khi đó nếu sử dụng giải thuật tìm kiếm tuần tự thì sẽ không tận dụng được tính sắp xếp của dữ liệu. Trong trường hợp này để nâng cao hiệu quả tìm kiếm người ta sử dụng giải thuật tìm kiếm nhị phân.

7.3.1. Ý tưởng và giải thuật

Tìm kiếm nhị phân là một phương pháp tìm kiếm khá thông dụng. Nó tương tự như cách thức ta tìm một từ trong từ điển. Chỉ có một điều hơi khác là:

- Khi tìm kiếm từ trong từ điển, ta thường đoán từ cần tìm nằm trong khoảng nào, rồi mở một trang ở khoảng đó ra để tìm, nếu không tìm thấy từ trên trang này thì dựa vào thì dựa vào tính sắp xếp của từ điển mà ta quyết định tìm từ ở phần trước hoặc phần sau trang này theo cách tương tự cho đến khi tìm thấy.

- Còn với phép tìm kiếm nhị phân, giả sử dãy khoá ban đầu được sắp theo thứ tự tăng dần, nghĩa là: $K_0 < K_1 < \dots < K_{n-1}$, thì ta luôn chọn khoá ở “giữa” (giả sử là K_g) của dãy khoá đang xét để so sánh, có ba khả năng có thể xảy ra:

- Nếu $x = K_g$: Tìm thấy x trong dãy, dừng quá trình tìm kiếm.
- Nếu $x < K_g$: Nếu x có trong dãy thì x chỉ nằm trong nửa dãy bên trái của K_g , tức là tiếp tục tìm x trên đoạn $[K_0, K_{g-1}]$.
- Nếu $x > K_g$: Nếu x có trong dãy thì x chỉ nằm trong nửa dãy bên phải của K_g , tức là tiếp tục tìm x trên đoạn $[K_{g+1}, K_{n-1}]$.

Việc tìm kiếm x trên nửa dãy bên trái hay nửa dãy bên phải của K_g được thực hiện giống như việc tìm x trên cả dãy ban đầu.

Nếu gọi t là chỉ số của *phần tử bên trái nhất* và p là chỉ số của phần tử *bên phải nhất* của đoạn tìm kiếm thì t luôn nhỏ hơn hoặc bằng p , nếu thấy $t > p$ nghĩa là không tìm thấy x trong dãy.

Sau đây là giải thuật tìm kiếm nhị phân được viết dưới dạng hàm đệ quy. Quá trình gọi đệ quy sẽ kết thúc (trường hợp suy biến của bài toán) nếu: $t > p$ (không tìm thấy, hàm trả về -1) hoặc $x = K_g$ (tìm thấy, hàm trả về g).

Binary_Search(K, t, p, x)

```
{
  if ( $t > p$ ) return -1; // Không tìm thấy
  else
  {
     $g = (t+p) / 2$ ;
    if ( $x == K[g]$ ) return  $g$ ; // Tìm thấy
    if ( $x < K[g]$ )
      Binary_Search( $K, t, g - 1, x$ ) // Tìm tiếp ở nửa trước
```


b) Nếu $x = 71$ phép tìm kiếm không thoả mãn và các bước sẽ như sau:

Bước 1: [15 27 36 45 **58** 65 75 87 94 99]
 t g p

Bước 2:

[65	75	<u>87</u>	94	99]
t		g		p

Bước 3: [65 75]
t,g p

Bước 4: **[75]** (không tìm thấy, dùng)
t,g,p

7.3.2. Phân tích và đánh giá

Xét giải thuật đệ quy nêu trên, ta thấy:

- Trường hợp tốt nhất, phân tử giữa dãy ban đầu có giá trị bằng x, lúc này chỉ cần thực hiện 1 phép so sánh, tức là $T_{\text{tốt}}(n) = O(1)$.

- Trường hợp xấu nhất là phần tử cuối cùng hoặc đầu tiên có giá trị bằng x hoặc không có x trong dãy. Khi đó, dãy liên tiếp được chia đôi và ta phải gọi đệ quy cho tới khi dãy khoá được xét chỉ còn 1 phần tử.

Giả sử gọi $w(n)$ là hàm biểu thị số lượng các phép so sánh trong trường hợp xấu nhất. Khi đó, ta có:

$$w(n) = 1 + w(\lfloor n/2 \rfloor) \quad (\text{cần 1 phép so sánh để tách đôi dãy})$$

Với phương pháp truy hồi, ta có thể viết:

$$\begin{aligned} w(n) &= 1+1+ w(\lfloor n/2^2 \rfloor) \\ &= 1+1+1+ w(\lfloor n/2^3 \rfloor) \end{aligned}$$

Như vậy, tại bước k, ta có:

$$w(n) = 1 + \dots + 1 + w(\lfloor n/2^k \rfloor) = k + w(\lfloor n/2^k \rfloor) \quad (*)$$

Quá trình gọi đệ quy sẽ dừng khi dãy chỉ còn lại một phần tử, tức là khi $\lfloor n/2^k \rfloor = 1$. Ta có, $w(\lfloor n/2^k \rfloor) = w(1) = 1$, và khi $\lfloor n/2^k \rfloor = 1$ thì suy ra $2^k \leq n \leq 2^{k+1}$, do đó $k \leq \log_2 n \leq k+1$, nghĩa là có thể viết $k = \lfloor \log_2 n \rfloor$. Thay k vào (*) ta có:

$$w(n) = \lfloor \log_2 n \rfloor + w(1) = \lfloor \log_2 n \rfloor + 1.$$

Như vậy: $T_{\text{xấu}}(n) = O(\log_2 n)$

Người ta cũng chứng minh được: $T_{tb}(n) = O(\log_2 n)$

Rõ ràng so với giải thuật tìm kiếm tuần tự, chi phí tìm kiếm nhị phân thấp hơn rất nhiều. Tuy nhiên, trước khi sử dụng tìm kiếm nhị phân thì dãy khoá đã phải được sắp xếp, nghĩa là thời gian chi phí cho sắp xếp cũng phải kể đến. Nếu dãy khoá luôn biến động (thường xuyên được bổ sung thêm hoặc bớt đi) thì lúc đó chi phí cho sắp xếp lại trở nên đáng kể, và chính trường hợp này đã bộc lộ nhược điểm của phương pháp tìm kiếm nhị phân.

7.4. Cây nhị phân tìm kiếm

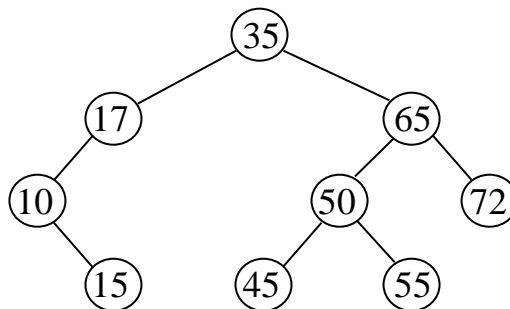
Để khắc phục nhược điểm vừa nêu trên đối với tìm kiếm nhị phân và đáp ứng yêu cầu tìm kiếm đối với dãy khoá biến động, một phương pháp mới được hình thành dựa trên cơ sở dãy khoá được tổ chức dưới dạng cây nhị phân (mỗi khoá ứng với một nút trên cây đó) mà ta gọi là *cây nhị phân tìm kiếm*.

7.4.1. Định nghĩa cây nhị phân tìm kiếm

Cây nhị phân tìm kiếm là một cây nhị phân mà mỗi nút trên cây đều thỏa mãn tính chất:

- Mọi khoá thuộc cây con trái nút đó đều nhỏ hơn khoá ứng với nút đó.
- Mọi khoá thuộc cây con phải nút đó đều lớn hơn khoá ứng với nút đó.

Ví dụ:



Hình 7.1. Ví dụ về cây nhị phân tìm kiếm

Ở đây ta sẽ cài đặt cây nhị phân tìm kiếm bằng móc nối, mỗi nút có cấu trúc như sau:

P_L	KEY	P_R
-----	-----	-----

Thành phần KEY chứa giá trị khoá của nút.

Thành phần P_L là con trỏ trái, trỏ tới cây con trái của nút đó.

Thành phần P_R là con trỏ phải, trỏ tới cây con phải của nút đó.

7.4.2. Giải thuật tìm kiếm

Đối với cây nhị phân tìm kiếm, để tìm xem giá trị khoá x nào đó có trên cây hay không ta có thể thực hiện như sau:

So sánh x với khoá ở gốc, có 4 tình huống sau có thể xảy ra:

1) Không có gốc (cây rỗng): x không có trên cây, phép tìm kiếm không thành công.

2) x trùng với khoá ở gốc: Phép tìm kiếm thành công, dừng.

3) x nhỏ hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con trái của gốc.

4) x lớn hơn khoá của gốc, phép tìm kiếm được tiếp tục trong cây con phải của gốc.

Việc tìm x trên cây con trái hay cây con phải của gốc được tiến hành giống như việc tìm x trên cây ban đầu.

Ví dụ: Với cây ở hình 7.1, nếu $x = 45$, quá trình tìm kiếm sẽ như sau:

- So sánh x với 35: $x > 35$, ta chuyển sang tìm ở cây con phải.
- So sánh x với 65: $x < 65$, ta chuyển sang tìm ở cây con trái.
- So sánh x với 50: $x < 50$, ta chuyển sang tìm ở cây con trái.
- So sánh x với 45: $x = 45$, vậy tìm kiếm đã thành công.

Nhưng nếu $x = 20$, thì phải:

- So sánh x với 35: $x < 35$, ta chuyển sang tìm ở cây con trái.
- So sánh x với 17: $x > 17$, ta chuyển sang tìm ở cây con phải, nhưng cây con phải rỗng, nên phép tìm kiếm không thành công.

Nếu sau phép tìm kiếm không thành công, ta bổ sung luôn x vào cây nhị phân tìm kiếm (như ví dụ vừa xét ta bổ sung khoá 20 vào thành con phải của nút 17), ta thấy phép bổ sung này thực hiện rất đơn giản và không làm ảnh hưởng gì tới vị trí của các khoá hiện có trên cây, tính chất của cây nhị phân tìm kiếm vẫn được đảm bảo.

Giải thuật tìm kiếm có bổ sung trên cây nhị phân tìm kiếm được viết như sau:

BTS(T, x)

```
/* Hàm này thực hiện tìm kiếm trên cây nhị phân tìm kiếm, có gốc được  
   trả bởi T, khoá tìm kiếm là x. Nếu tìm kiếm thành công thì cho con trỏ  
   P trỏ tới nút đó, nếu tìm kiếm không thành công thì bổ sung nút mới có  
   khoá là x vào T và cho con trỏ P trỏ vào nút mới đó kèm theo thông báo  
   */
```

```
Q = NULL; P = T;
```

```
while (P!=NULL) // Tìm kiếm, con trỏ Q trỏ vào nút cha của P
```

```
{  
    if ( x == P->KEY)  
        return ;  
    Q = P;  
    if (x < P->KEY)  
        P = P->P_L;  
    else  
        P = P->P_R;  
}
```

```
P = malloc(); // Khóa x chưa có trên cây, thực hiện bổ sung
```

```
P->KEY = x;
```

```
P->P_L = P->P_R = NULL;
```

```
if (T == NULL) // Cây rỗng, nút mới chính là gốc
```

```
    T = P;
```

```
else
```

```
    if ( x < Q->KEY)
```

```
        Q->P_L = P;
```

```
    else
```

```
        Q->P_R = P;
```

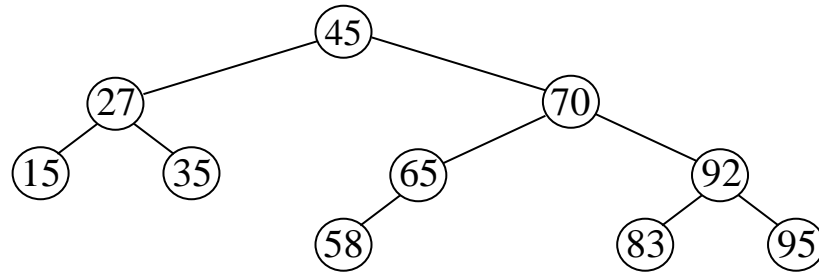
```
    printf("Không tìm thấy, đã bổ sung")
```

```
}
```

Với giải thuật trên ta nhận thấy: có thể dựng được cây nhị phân tìm kiếm ứng với một dãy khoá đưa vào bằng cách liên tục bổ sung các nút ứng với từng khoá, bắt đầu từ một cây rỗng. Ví dụ, với dãy khoá:

45 27 15 70 65 58 35 92 95 83

thì cây nhị phân tìm kiếm được dựng sẽ là:



Hình 7.2. Dựng cây nhị phân tìm kiếm từ dãy khoá đã cho

7.4.3. Phân tích đánh giá

Rõ ràng với giải thuật BST nêu trên, ta thấy dạng của cây nhị phân tìm kiếm được dựng hoàn toàn phụ thuộc vào dãy khoá đưa vào. Như vậy có nghĩa là: trong quá trình xử lý động ta không thể biết trước được cây sẽ phát triển ra sao, hình dạng của nó sẽ như thế nào. Rất có thể đó là một cây nhị phân hoàn chỉnh hoặc cây nhị phân gần đầy (mà ta sẽ gọi là *cây cân đối*), chiều cao của nó là $\lceil \log_2(n+1) \rceil$ (n là số nút trên cây), nên chi phí tìm kiếm có cấp độ lớn chỉ là $O(\log_2 n)$, đây là một trường hợp rất thuận lợi mà ta luôn mong đợi. Nhưng cũng có thể đó là một cây nhị phân suy biến (nếu dãy khoá đưa vào vốn đã được sắp xếp rồi), khi đó cây không khác gì một danh sách tuyến tính, và việc tìm kiếm trên cây đó chính là tìm kiếm tuần tự với chi phí có cấp $O(n)$. Điều này cũng dễ đưa tới một khuynh hướng lo ngại khiến ta thiếu tin tưởng vào phương pháp tìm kiếm này.

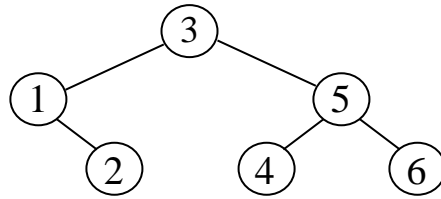
Thực ra, người ta cũng đã chứng minh được số lượng trung bình các phép so sánh trong tìm kiếm trên cây nhị phân tìm kiếm chỉ là:

$$C_{tb} \approx 1,386 \log_2 n$$

Như vậy cấp độ lớn của thời gian thực hiện trung bình của giải thuật BST cũng chỉ là $O(\log_2 n)$, còn nếu xét chi tiết ra thì chi phí tìm kiếm trung bình ở đây lớn hơn khoảng 39% so với chi phí tìm kiếm trên cây cân đối.

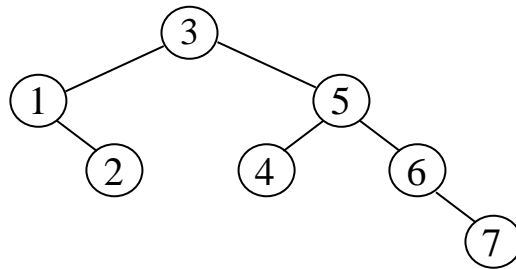
Tới đây cũng có thể xuất hiện thêm câu hỏi là: tại sao không tìm cách dựng lên một cây nhị phân tìm kiếm luôn cân đối để có thể đạt được chi phí tối thiểu? Ta có thể thấy câu trả lời qua việc xét ví dụ đơn giản sau:

Giả sử có cây nhị phân tìm kiếm như ở hình 7.3



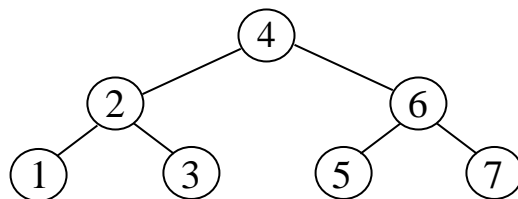
Hình 7.3. Ví dụ về cây nhị phân cân đối

Nếu bây giờ xuất hiện khoá tìm kiếm $x = 7$ thì vì x không có trên cây, nên sẽ được bổ sung vào cây như hình 7.4.



Hình 7.4. Cây sau khi bổ sung thêm giá trị khoá

Ta thấy cây sẽ không còn cân đối nữa. Khi đó ta phải “tái cân đối” lại để có cây cân đối với 7 nút như hình 7.5.



Hình 7.5. Cây sau khi tái cân đối

Nhìn vào hình trên ta thấy hầu như không còn nút nào mà mỗi nối được giữ nguyên như cũ. Như vậy, việc tái cân đối đã đòi hỏi phải sửa lại khá nhiều mối nối, nghĩa là sẽ tốn khá nhiều thời gian. Do đó khi phép bổ sung thường xuyên được thực hiện thì cách làm đó sẽ trở nên không thực tế. Đó chính là câu trả lời cho câu hỏi đặt ra ở trên.

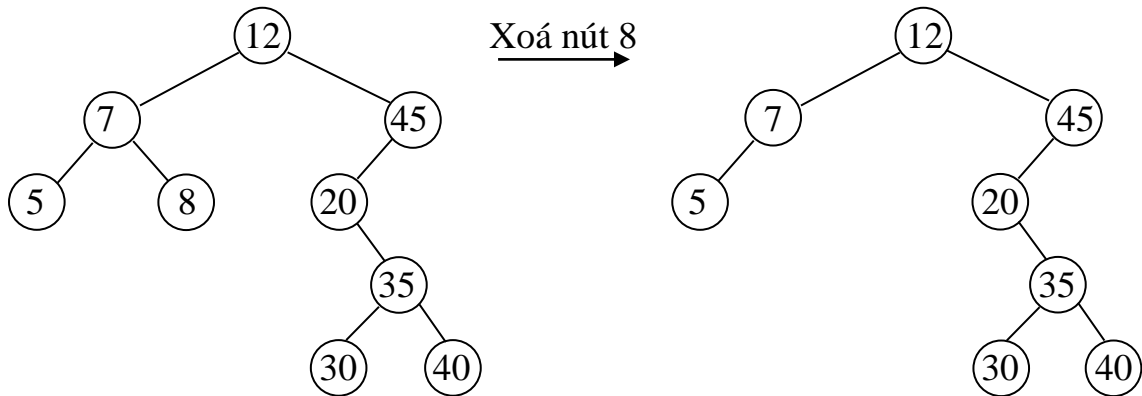
7.4.4. Phép loại bỏ trên cây nhị phân tìm kiếm

Phép loại bỏ trên cây nhị phân tìm kiếm không đơn giản như phép bổ sung, vì để đảm bảo được phần cây còn lại vẫn là một cây nhị phân tìm kiếm ta sẽ phải sửa lại cây, nghĩa là phải tìm “nút thay thế nó” rồi tiến hành điều chỉnh các mối nối sao cho số các thao tác thực hiện trên cây phải là ít nhất và cây ít biến động nhất.

Giả sử ta cần xóa một nút D trên cây nhị phân tìm kiếm, khi đó có các khả năng sau xảy ra:

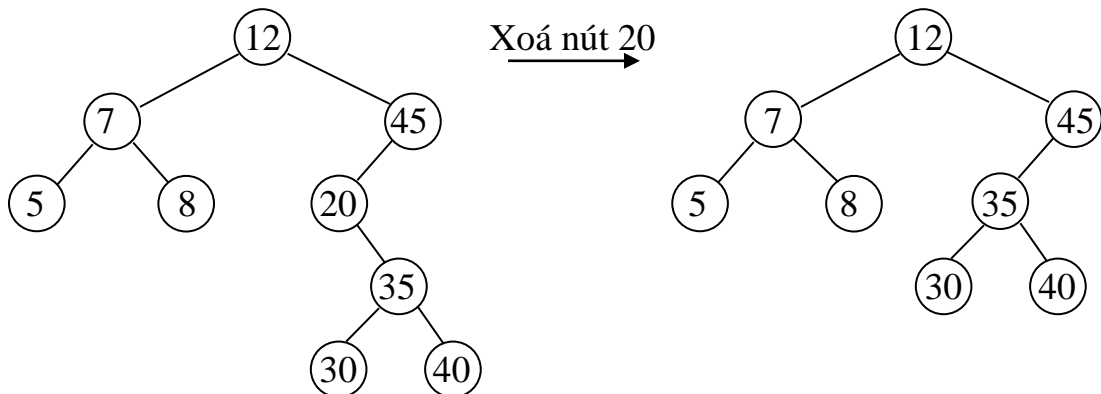
a) Nếu D là nút lá: việc xóa D tương đối đơn giản, ta không phải tìm nút thay thế, chỉ cần cho mỗi nối trỏ tới D (từ nút cha của D) trở về NULL, rồi giải phóng D.

Ví dụ:



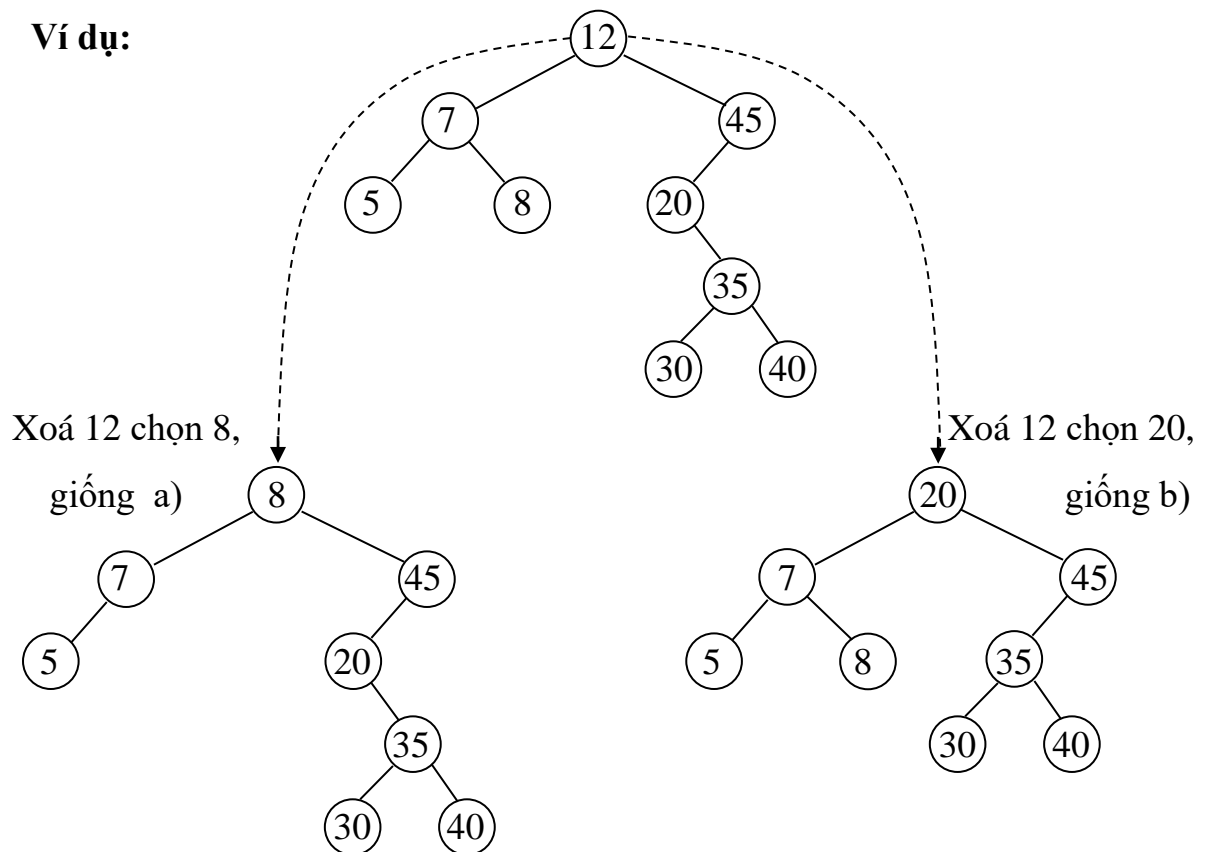
b) Nếu D là nút “nửa lá”, nghĩa là nó chỉ có một nhánh con (là cây con trái hoặc cây con phải), thì nút thay thế D chính là nút gốc của nhánh con đó. Khi đó ta điều chỉnh mỗi nối như sau: Cho mỗi nối trỏ tới D (từ nút cha của D) trở vào nút gốc nhánh con của D. Sau đó giải phóng D.

Ví dụ:



c) Nếu D là nút có đầy đủ hai nhánh con (trường hợp tổng quát), thì nút thay thế D sẽ là *nút lớn nhất thuộc cây con trái (nút cực phải của cây con trái)* của D, hoặc là *nút nhỏ nhất thuộc cây con phải (nút cực trái của cây con phải)* của D. Khi đó, thay vì xóa nút D, ta lấy giá trị của nút “thay thế” thay cho giá trị của nút D, rồi xóa nút “thay thế”. Vì đặc điểm của nút “thay thế” nên nó chỉ có thể là nút có một nhánh con hoặc là nút lá, do vậy việc xóa nó giống như hai trường hợp đầu.

Ví dụ:



Giải thuật xoá nút có giá trị khoá x trên cây nhị phân tìm kiếm được trợ bởi T có thể viết như sau:

```

BST_Delete(T, x)
{
    P = T; Q = NULL ; // Q luôn là cha của P
    while (P!=NULL) // Tìm xem có khóa x trên cây không
    {
        if (P->Key == x) // Tìm thấy, P trở vào nút cần xoá
            break ;
        Q = P;
        if ( x < P->KEY)
            P = P->P_L
        else
            P = P->P_R;
    }
    if (P == NULL) // x không có trên cây
        return;

```

```

if (P->P_L!=NULL && P->P_R!=NULL) // P có đầy đủ hai cây con
{ // sẽ tìm nút cực phải của cây con trái làm nút thay thế
  Node = P; // Ghi nhớ nút cần xóa
  // Chuyển sang nhánh trái để tìm nút “thay thế” và cho P trở vào
  Q = P; P = P->P_L;
  while (P->P_R != NULL) // Tìm đến nút cực phải
  {
    Q = P; P = P->P_R;
  }
  // Chuyển giá trị trong nút thay thế lên nút cần xóa
  Node->KEY = P->KEY;
}

/* Nút cần xóa bây giờ là nút P, nó chỉ có thể là nút lá hoặc có một nhánh
con: Nếu P có một nhánh con thì dùng con trở Child trở vào nút gốc của
nhánh con đó; Còn nếu P là nút lá thì cho Child = NULL */
if (P->P_L != NULL)
  Child = P->P_L;
else
  Child = P->P_R;
if (P == T) // Nếu nút cần xóa là nút gốc của cây
  T = Child;
else // Sửa mỗi nối cũ trở vào P thành trở vào Child
  if (Q->P_L == P)
    Q->P_L = Child;
  else
    Q->P_R = Child;
free(P); // Giải phóng P
}

```

Qua giải thuật trên ta thấy khi loại bỏ một nút ra khỏi cây nhị phân tìm kiếm, thì việc sửa lại cây chỉ cần sửa lại một mỗi nối. Như vậy chi phí thời gian cho sửa đổi này cũng không đáng kể. Nếu cho biết giá trị khoá của nút cần loại bỏ trên cây nhị phân tìm kiếm, thì trước hết ta phải tìm ra nút cần loại bỏ, rồi có thể từ vị trí

nút cần loại bỏ tìm tiếp xuống phía dưới để tìm nút cần thay thế, bởi vậy tương tự như khi tìm kiếm rồi bỏ sung, phép tìm kiếm rồi loại bỏ cũng có chi phí trung bình về thời gian ở cấp: $O(\log_2 n)$.

Bài tập chương 7

7.1. Cho dãy số nguyên sau:

3 5 7 9 10 12 15 20

- a) Mô tả các bước hoạt động của giải thuật tìm kiếm tuần tự để tìm $x=10$.
- b) Mô tả các bước hoạt động của giải thuật tìm kiếm nhị phân để tìm $x=13$.

7.2. Cho dãy các số nguyên sau:

55 60 15 25 45 40 50 20 10 30

- a) Áp dụng giải thuật tìm kiếm có bỏ sung trên cây nhị phân tìm kiếm, hãy dựng cây nhị phân tìm kiếm khi thêm tuần tự dãy số trên vào cây rỗng ban đầu.
- b) Hãy biểu diễn hình ảnh của cây từ câu a) khi ta xoá nút có khoá 50 và 15.
- c) Mô tả quá trình tìm $x=30$ trên cây đã xây dựng.

7.3. Từ ý tưởng của giải thuật tìm kiếm nhị phân, người ta đưa ra ý tưởng cho giải thuật tìm kiếm tam phân như sau: chia dãy ban đầu $[K_0, K_1, \dots, K_{n-1}]$ thành ba dãy con tại các vị trí K_{m1}, K_{m2} . Lần lượt so sánh x với hai phần tử đó, nếu x không bằng một trong hai phần tử đó, thì giới hạn tiếp vùng tìm kiếm là dãy $[K_0, K_1, \dots, K_{m1-1}]$, hay dãy $[K_{m1+1}, \dots, K_{m2-1}]$, hoặc dãy $[K_{m2+1}, \dots, K_{n-1}]$

- a) Viết giải thuật tìm kiếm tam phân.
- b) Hãy xác định độ phức tạp tính toán của giải thuật
- c) Lấy ví dụ minh họa hoạt động của giải thuật.

7.4. Viết chương trình thực hiện các yêu cầu sau:

- a) Nhập vào dãy khóa là các số nguyên khác nhau và lưu trữ dưới dạng cấu trúc cây nhị phân tìm kiếm.
- b) Viết hàm tìm kiếm một khóa x trên cây.
- c) Viết hàm xóa một nút trên cây.
- d) Viết hàm in giá trị nhỏ nhất, lớn nhất trên cây.
- e) Viết hàm in dãy khóa theo chiều tăng dần.

TÀI LIỆU THAM KHẢO

- [1] Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, Nhà xuất bản Đại học Quốc gia Hà Nội, 2010.
- [2] Phạm Văn Ất, *Giáo trình Kỹ thuật lập trình C Căn bản & Nâng cao*, Nhà xuất bản Hồng Đức, 2009.
- [3] A. V. Aho, J. E. Hopcroft , and J. D. Ullman, *Data Structures and Algorithms*. Addison - Wesley, 1983.
- [4] Niklaus Wirth, *Data Structures and Algorithms*, Prentice Hall, 2004.