Mục lục

Danh sách hình vẽ

Danh sách bảng

Lời nói đầu

Trước khi viết một chương trình máy tính, một trong những việc quan trọng đầu tiên chúng ta cần làm là suy nghĩ về cách thức tổ chức lưu trữ, quản lý các dữ liệu của chương trình và lựa chọn các giải thuật phù hợp với các yêu cầu cụ thể của bài toán. Trong tổng thể kiến thức về tin học, cấu trúc dữ liệu và thuật toán đóng vai trò quan trọng trong quá trình đào tạo kỹ sư ngành Công nghệ thông tin và Điện tử viễn thông. Đáp ứng nhu cầu của các bạn sinh viên nói chung và sinh viên Trường Công Nghệ Thông Tin và Truyền thông, Đại học Thái nguyên nói riêng, chúng tôi đã tiến hành biên soạn cuốn sách "Cấu trúc dữ liệu và Thuật toán" với mục đích giúp các bạn sinh viên chuyên ngành có một tài liệu hữu ích trong học tập, nâng cao trình độ. Ngoài ra giáo trình còn là một tài liệu hữu ích cho các bạn đọc có nhu cầu tham khảo phục vụ cho công việc của mình.

Trong giáo trình này chúng tôi sử dụng ngôn ngữ lập trình C để biểu diễn các cấu trúc dữ liệu và thuật toán, tuy nhiên bạn đọc có thể sử dụng ngôn ngữ lập trình yêu thích của mình để cài đặt, ví dụ: Pascal, Java, Python, ...

Cuốn sách này được biên soạn dựa trên kinh nghiệm giảng dạy và các kiến thức tích lũy từ nhiều năm của nhóm các tác giả tại trường Đại học Công nghệ thông tin và Truyền thông, Đại học Thái Nguyên. Bạn đọc sẽ tìm thấy trong sách một hệ thống kiến thức về

các cấu trúc dữ liệu và thuật toán khá đầy đủ và chuyên sâu. Trong cách trình bày, chúng tôi luôn cố gắng cung cấp nhiều ví dụ minh họa, các đoạn mã nguồn và thậm chí là những chương trình hoàn thiện cho những diễn giải phần kiến thức trong sách, điều này là rất cần thiết cho các bạn sinh viên trong giai đoạn đầu mới học lập trình. Kết thúc mỗi chương, chúng tôi cũng cung cấp các câu hỏi và bài tập củng cố nhằm giúp bạn đọc nắm vững các kiến thức, nâng cao kỹ năng thực hành, khả năng tư duy, áp dụng các kiến thức trong việc giải quyết các vấn đề thực tế. Đây là điểm khác biệt lớn của cuốn sách so với những cuốn sách khác hiện đã xuất bản.

Mặc dù đã gắng nhiều trong quá trình biên soạn giáo trình nhưng chắc chắn giáo trình vẫn còn nhiều thiếu sót và hạn chế. Nhóm tác giả rất mong nhận được sự đóng góp ý kiến của bạn đọc để cuốn sách này ngày càng hoàn thiện hơn và có thể trở thành một giáo trình thực sự hữu ích đối với bạn đọc. Mọi đóng góp xin gửi qua địa chỉ: .....

Thái Nguyên, nhóm tác giả

Chương 1

Tổng quan về cấu trúc dữ liệu và thuật toán

1.1 Sơ lược các giai đoạn phát triển phần mềm

Để phát triển một phần mềm/chương trình máy tính, chúng ta thường phải trải qua một quy trình gồm bốn giai đoạn chính sau:

Giai đoạn 1 - What: Khảo sát, phân tích và hình thành bài toán tin học

Giai đoạn 2 - How: Thiết kế phần mềm

Giai đoạn 3 - Buid: Xây dựng phần mềm

Giai đoạn 4 - Use: Kiểm thử, vận hành và bảo trì phần mềm

Giáo trình này tập trung vào giai đoạn 3. Từ các mô hình dữ liệu đã được thiết kế ở giai đoạn trước, chúng ta cần xây dựng hoặc

lựa chọn cấu trúc dữ liệu cũng như thuật toán thích hợp để cài đặt chương trình. Việc xây dựng, sử dụng các cấu trúc dữ liệu và thuật toán hiệu quả để xây dựng chương trình có thể tạo ra sự khác biệt lớn giữa một chương trình chỉ chạy trong vài giây với một chương trình yêu cầu thời gian chạy một vài ngày cho cùng một bài toán cần giải quyết!

Tính hiệu quả của thuật toán thường xác định bởi hai tiêu chí cơ bản (1) dung lượng không gian nhớ cần thiết để lưu trữ các dữ liệu vào, các kết quả tính toán trung gian và các kết quả đầu ra của thuật toán (ta gọi là độ phức tạp không gian); (2) thời gian cần thiết để thực hiện thuật toán (ta gọi là độ phức tạp tính toán).

Một thuật toán được xem là hiệu quả hơn thuật toán khác nếu nó có độ phức tạp tính toán và độ phức tạp không gian ít hơn so với thuật toán khác. Chi tiết về việc xác định các đại lượng này và các ví dụ minh họa được trình bày lần lượt trong các chương, mục tương ứng của giáo trình. Sau khi lựa chọn được các cấu trúc dữ liệu và thuật toán phù hợp, chúng ta cần cài đặt chúng sử dụng ngôn ngữ lập trình cụ thể để tạo ra các chương trình máy tính/thành phần phần mềm cụ thể.

1.2 Các khái niệm cơ bản

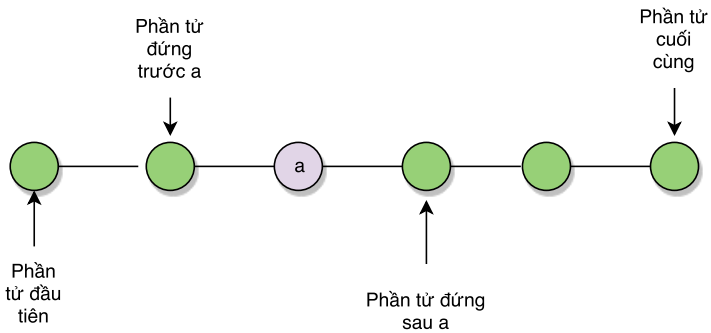
1.2.1 Mô hình dữ liệu

Mô hình dữ liệu là cấu trúc logic của dữ liệu được xử lý bởi hệ thống. Mô hình dữ liệu còn được gọi là mô hình toán học cùng với các phép toán có thể thực hiện trên các phần tử dữ liệu của mô hình.

Có thể phân loại các mô hình dữ liệu dựa trên mối quan hệ giữa các phần tử của chúng như sau:

Mô hình dữ liệu tuyến tính (Danh sách)

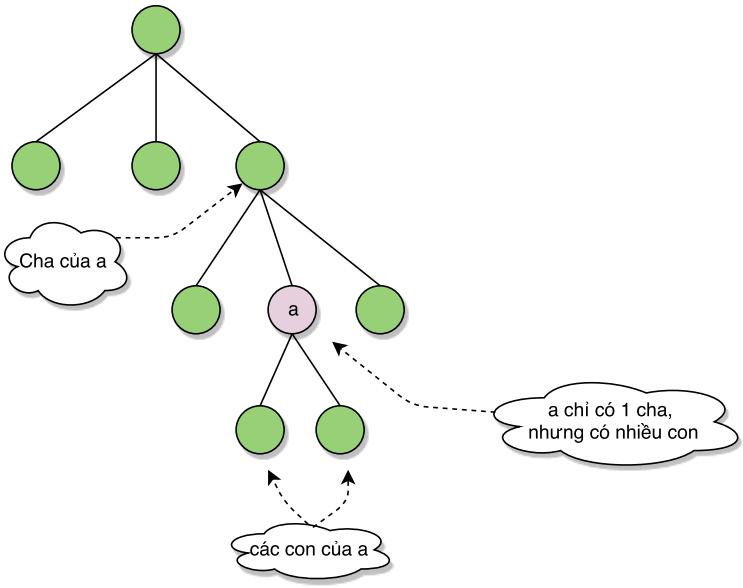
Mô hình dữ liệu tuyến tính dùng để biểu diễn các phần tử có quan hệ 1:1, tuyến tính theo thứ tự xuất hiện. Hay nói cách khác, môt mô hình dữ liệu tuyến tính sẽ có phần tử đầu tiên, phần tử cuối cùng, và nếu xét một phần tử nào đó ở giữa danh sách thì nó sẽ có một phần tử đứng ngay trước nó và một phần tử đứng ngay sau nó. Hình 1.1 biểu diễn một ví dụ về mô hình dữ liệu tuyến tính.



Hình 1.1: Mô hình danh sách.

Mô hình dữ liệu phân cấp (cây)

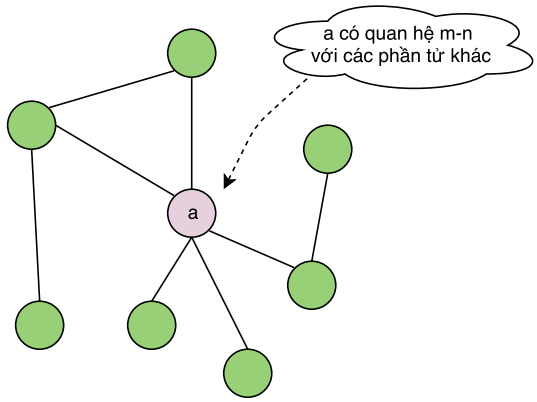
Mô hình dữ liệu cây dùng để biểu diễn các phần tử có quan hệ 1:n, quan hệ này còn được gọi là quan hệ cha-con, mỗi phần tử trong mô hình thường có nhiều con, nhưng chỉ có một cha. Hình 1.2 biểu diễn một ví dụ cụ thể về mô hình cây. Nếu ta di chuyển từ trên xuống dưới trong Hình 1.2 thì mỗi nút có thể trỏ đến nhiều nút khác – các con, nhưng nếu ta di chuyển từ dưới lên thì mỗi nút (trừ nút ở gốc) chỉ có quan hệ với 1 nút - cha. Cây là một trong những mô hình dữ liệu quan trọng và thông dụng trong khoa học máy tính.



Hình 1.2: Mô hình dữ liệu cây.

Mô hình dữ liệu đồ thị

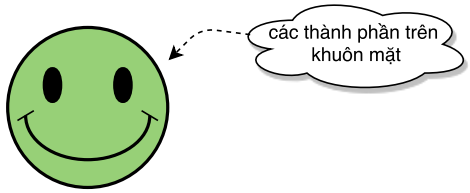
Đồ thị là mô hình dữ liệu phong phú và phức tạp nhất. Trong đồ thị, các phần tử có mối quan hệ n:m, còn được gọi là quan hệ nhiều - nhiều. Mỗi phần tử trong đồ thị thường có quan hệ với một hoặc nhiều phần tử khác. Hình 1.3 biểu diễn một ví dụ cụ thể về mô hình đồ thị.



Hình 1.3: Ví dụ về mô hình đồ thị.

Mô hình dữ liệu tập hợp

Mô hình tập hợp thường dùng để biểu diễn các phần tử nằm trong một phạm vi nhất định nào đó. Chúng có thể có quan hệ nào đó với nhau, còn gọi là quan hệ thành viên trong tập hợp. Thông thường ta không cần quan tâm tới vị trí chính xác của một phần tử nào đó trong tập hợp. Hình 1.4 biểu diễn một ví dụ cụ thể về mô hình tập hợp, ở đây trong vùng khuôn mặt người có nhiều đối tượng khác nhau như mắt, mũi, miệng, ...



Hình 1.4: Mô hình tập hợp gồm các thành phần trên khuôn mặt.

Trên đây là bốn loại mô hình dữ liệu mà chúng ta sẽ nghiên cứu trong giáo trình này. Chúng ta cũng sẽ nghiên cứu các dạng biểu diễn của các mô hình này bởi các cấu trúc dữ liệu khác nhau trong giai đoạn cài đặt chương trình. Nói chung, hầu hết các cấu trúc dữ liệu đều là biểu diễn cụ thể của bốn mô hình cơ bản này.

1.2.2 Trừu tượng hóa

Trừu tượng hóa là quá trình khái quát hóa các đối tượng cụ thể để tìm những đặc điểm chung của chúng, hay nói cách khác quá trình trừu tượng hóa giúp ta xác định được những thuộc tính, hành động chung của các đối tượng và làm mờ đi những chi tiết riêng tư, không cần thiết. Trừu tượng hóa là cơ sở cho việc hình thành nên các khái niệm.

1.2.3 Kiểu dữ liệu trừu tượng

Kiểu dữ liệu trừu tượng là một mô hình dữ liệu được xét cùng với một số phép toán xác định. Chẳng hạn, mô hình dữ liệu danh sách được xét đến các phép toán thêm vào và lấy ra, ta gọi là kiểu dữ liệu trừu tượng hàng đợi hoặc kiểu dữ liệu trừu tượng ngăn xếp; mô hình tập hợp, chỉ xét đến các phép toán thêm vào, loại bỏ và tìm kiếm ta gọi là kiểu dữ liệu trừu tượng từ điển.

1.2.4 Dữ liệu của chương trình

Thực tế dữ liệu tồn tại ở rất nhiều dạng như hình ảnh, âm thanh, ... Trong một chương trình, dữ liệu được phân làm ba loại:

1. Dữ liệu vào, đây là các đối tượng dữ liệu cần xử lý của bài toán;
2. Dữ liệu trung gian;
3. Dữ liệu đầu ra hay còn gọi là các đối tượng dữ liệu sau khi xử lý.

Ví dụ xét bài toán cho một dãy số gồm n số nguyên, tìm số lớn nhất trong dãy số. Các dữ liệu của bài toán là:

- Dữ liệu đầu vào: số nguyên n, dãy số a_1, a_2, ..., a_n
- Dữ liệu đầu ra: số lớn nhất (giả sử max)
- Dữ liệu trung gian: giả sử dùng biến điều khiển i trong quá trình duyệt từ đầu đến cuối dãy để tìm số lớn nhất

1.2.5 Biểu diễn dữ liệu trên máy tính

Trong máy tính điện tử, các dữ liệu dù tồn tại ở những hình thức khác nhau (số, văn bản, hình ảnh, đúng/sai...) đều được biểu diễn dưới dạng nhị phân khi đưa vào máy tính xử lý. Tức là mỗi dữ liệu

được biểu diễn dưới dạng một dãy các số nhị phân 0 hoặc 1. Ví dụ: Số 10 = 0000 1010 ở dạng nhị phân. Cách biểu diễn này rất không thuận tiện (dài, khó, không gợi nhớ, ...) đối với con người. Việc xuất hiện các ngôn ngữ lập trình bậc cao (ví dụ như Pascal, C, ... gần với ngôn ngữ tự nhiên) đã giải phóng con người khỏi những khó khăn khi làm việc với cách biểu diễn dữ liệu nhị phân trong máy tính.

Trong các ngôn ngữ lập trình bậc cao: Các kiểu dữ liệu là sự trừu tượng hoá các tính chất của các đối tượng dữ liệu có cùng bản chất trong thế giới thực (chỉ ra những tính chất đặc trưng cho các đối tượng thuộc phạm vi bài toán đang xét). Ví dụ, ứng với các dữ liệu dạng số, tương ứng ta có các kiểu dữ liệu số nguyên, số thực, số phức, ... trong ngôn ngữ lập trình.

Như vậy tất cả các dữ liệu mô tả trong ngôn ngữ lập trình bậc cao được máy tính xử lý đều phải được khai báo thuộc một kiểu dữ liệu xác định. Khi đó chương trình dịch của ngôn ngữ lập trình bậc cao sẽ dịch chuyển các kiểu dữ liệu này sang dạng biểu diễn nhị phân trước khi được máy tính xử lý.

1.2.6 Kiểu dữ liệu

Trong các ngôn ngữ lập trình bậc cao, các dữ liệu thường được phân loại thành các lớp dữ liệu dựa vào bản chất của chúng. Mỗi lớp dữ liệu thường được thể hiện như một kiểu dữ liệu trong ngôn ngữ lập trình. Như vậy, một kiểu T là một tập hợp các phần tử, các phần tử này được gọi là các giá trị của kiểu, hay miền giá trị của kiểu. Chẳng hạn kiểu nguyên (int) trong C là tập hợp các số nguyên, kiểu ký tự (char) là tập các ký tự, ...Trong các ngôn ngữ lập trình khác nhau có thể có các kiểu dữ liệu khác nhau, hay có một hệ kiểu riêng của mình. Hệ kiểu của một ngôn ngữ bao gồm

các kiểu dữ liệu cơ sở và các phương pháp cho phép ta định nghĩa thêm các kiểu dữ liệu mới từ chúng.

Khi nói đến một kiểu dữ liệu T, ta cần đề cập đến hai đặc trưng sau:

1. Tập hợp các giá trị thuộc kiểu (V). Khi một biến x thuộc kiểu này, các giá trị của x có thể nhận một trong các giá trị thuộc V.

2. Tập hợp các phép toán có thể thực hiện trên các dữ liệu của kiểu (O). Chẳng hạn với kiểu float trong C, các phép toán có thể thực hiện là các phép toán số học thông thường như +, -, *,/ và các phép so sánh ==, >, <, ...

Như vậy, kiểu dữ liệu T có thể được xác định bởi một bộ $< V, O >$ trong đó:

- V: tập các giá trị hợp lệ mà một đối tượng kiểu T có thể nhận;
- O: tập các thao tác xử lý có thể thi hành trên đối tượng thuộc kiểu T.

Ví dụ:

- Xét kiểu dữ liệu ký tự: $< V_c, O_c >$; trong đó: $V_c : \{'a', .., 'z', 'A', ..., 'Z'\}$, $O_c : \{getChar(), upper(c), ...\}$
- Xét kiểu dữ liệu số nguyên: $< V_i, O_i >$, trong đó: $V_i : \{-32768..32767\}$,$O_i : \{+, -, *, /, \%\}$

Khi lập trình, muốn sử dụng một kiểu dữ liệu nào đó ta cần nắm vững các thông tin sau:

- Tên kiểu: từ khóa thể hiện cho kiểu đó,
- Miền giá trị: miền các giá trị của một biến thuộc kiểu này,

- Kích thước: tổng số byte mà một biến thuộc kiểu này có thể chiếm giữ,
- Tập các toán tử: các phép toán cơ bản có thể tác động trên các biến thuộc kiểu này.

Trong các ngôn ngữ lập trình bậc cao thường định nghĩa sẵn một số kiểu dữ liệu, chúng còn được gọi là các kiểu dữ liệu tiền định, ví dụ các kiểu dữ liệu cơ bản như char, int, float, double, boolean,... và một số kiểu dữ liệu có cấu trúc thông dụng như mảng, bản ghi, file, ... Để thao tác với các loại dữ liệu phức tạp hơn, thường thì người lập trình phải tự định nghĩa ra một kiểu dữ liệu mới phù hợp với các đối tượng dữ liệu của bài toán cần giải quyết từ những kiểu dữ liệu tiền định này. Ví dụ, để quản lý một danh sách sinh viên, trong đó mỗi sinh viên cần quản lý các thông tin như sau:

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

trong đó ngày sinh lại gồm các thông tin như ngày, tháng, năm sinh. Như vậy, chúng ta phải định nghĩa thêm hai kiểu dữ liệu mới (Date và SinhVien) dựa trên cấu trúc bản ghi với các trường thông tin tương ứng như sau:

```
// dinh nghia cau truc ngay sinh:
typedef struct Date{
   int ngay;
   int thang;
   int nam;
};
```

Kiểu dữ liệu có cấu trúc thể hiện thông tin về một sinh viên:

```
// dinh nghia cau truc sinh vien
typedef struct Sinhvien{
   char maSV[15];
   char tenSV[15];
   char noiSinh[25];
   int diemThi;
   struct Date ngaySinh;
};
```

Giả sử đã có cấu trúc phù hợp để lưu trữ một sinh viên, để quản lý nhiều sinh viên ta cần xây dựng thêm kiểu dữ liệu có cấu trúc mới, ví dụ danh sách tuyến tính các sinh viên, hoặc một danh sách liên kết các sinh viên.

1.2.7 Cấu trúc dữ liệu

Cấu trúc dữ liệu được mô tả như sau:

CTDL = { Phần tử dữ liệu } (1)

Trong đó, các dữ liệu thành phần thuộc kiểu liệu đã được định nghĩa trước, chúng có một mối liên kết nào đó với nhau. Một cách khái quát, cấu trúc dữ liệu là một dạng biểu diễn của các dữ liệu, mối quan hệ và các thao tác có thể tác động lên các phần tử dữ liệu này.

Ví dụ: cấu trúc dữ liệu mảng bao gồm một dãy có thứ tự các phần tử có cùng 1 kiểu dữ liệu xác định nào đó và các thao tác cơ bản có thể thực hiện trên các phần tử của mảng; bản ghi: bao gồm một tập các phần tử có thể thuộc các kiểu dữ liệu khác nhau, có mối quan hệ với nhau đó là chúng cùng mô tả các thuộc tính của một đối tượng, ví dụ thông tin về 1 con người gồm họ tên, ngày sinh, chiều cao, cân nặng, ...và các thao tác cơ bản có thể thực hiện trên các thành phần của bản ghi.

Một số tiêu chuẩn lựa chọn cấu trúc dữ liệu liệu cho bài toán

1. **Phản ánh đúng thực tế**: Lựa chọn cấu trúc dữ liệu phản ánh trung thực dữ liệu của bài toán cần giải quyết là một tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ bài toán. Cần xem xét kỹ lưỡng cũng như dự trù các trạng thái biến đổi của dữ liệu trong chu trình sống để có thể chọn cấu trúc dữ liệu lưu trữ thể hiện chính xác đối tượng thực tế. Một số tình huống chọn cấu trúc lưu trữ sai:

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

   Trong trường trung học, mỗi lớp có thể nhận tối đa 28 học sinh. Lớp hiện có 20 học sinh, mỗi tháng mỗi học sinh đóng học phí 100.000 đ. Chọn một biến số nguyên shortint (khả năng lưu trữ 0 - 255) để lưu trữ tổng học phí của lớp học trong tháng là không phù hợp vì giá trị tổng học phí thu được > 255, vượt khỏi khả năng lưu trữ của biến đã chọn, gây ra tình trạng tràn, dẫn đến sai lệch.

2. **Phù hợp với các thao tác trên đó**: Lựa chọn cấu trúc dữ liệu phù hợp với các yêu cầu xử lý của bài toán giúp tăng tính hiệu quả của thuật toán, cụ thể là giúp cho việc phát triển các thuật toán đơn giản, tự nhiên hơn trên cấu trúc. Do đó chương trình đạt hiệu quả cao hơn về tốc độ xử lý. Một tình huống chọn cấu trúc lưu trữ không phù hợp như khi cần xây dựng một chương trình soạn thảo văn bản, các thao tác xử lý thường xảy ra là chèn, xoá sửa các ký tự trên văn bản. Trong thời gian

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

3. **Tiết kiệm tài nguyên hệ thống**: Cấu trúc dữ liệu được lựa chọn chỉ nên sử dụng tài nguyên hệ thống vừa đủ để đảm nhiệm được chức năng của nó. Thông thường có 2 loại tài nguyên cần lưu tâm nhất: CPU và bộ nhớ. Tiêu chuẩn này nên cân nhắc tùy vào tình huống cụ thể khi viết chương trình. Nếu cần một chương trình có tốc độ xử lý nhanh thì khi chọn cấu trúc dữ liệu, yếu tố tiết kiệm thời gian xử lý phải đặt nặng hơn tiêu chuẩn sử dụng tối ưu bộ nhớ, và ngược lại. Một số tình huống chọn cấu trúc lưu trữ lãng phí: Sử dụng biến integer để lưu trữ một giá trị cho biết tháng hiện hành. Biết rằng tháng chỉ có thể nhận các giá trị từ 1-12, nên chỉ cần sử dụng kiểu byte là đủ.

   Để lưu trữ danh sách học viên trong một lớp, sử dụng mảng 50 phần tử (giới hạn số học viên trong lớp tối đa là 50). Nếu số lượng học viên thật sự ít hơn 30, thì gây lãng phí. Trường hợp này cần có một cấu trúc dữ liệu linh động hơn mảng, ví dụ danh sách liên kết – ta sẽ đề cập đến trong các chương tiếp theo.

Chi phí và lợi ích của cấu trúc dữ liệu

Thông thường mỗi cấu trúc dữ liệu được kết hợp với một số chi phí và lợi ích nhất định. Một cấu trúc dữ liệu yêu cầu một lượng

không gian cụ thể cho việc lưu trữ các phần tử của nó, một lượng thời gian cụ thể để thực hiện các thao tác cơ bản trên nó.

Trong thực tế thường khó lựa chọn được cấu trúc dữ liệu tốt hơn cấu trúc dữ liệu khác trong mọi tình huống. Do đó, việc lựa chọn cấu trúc dữ liệu tốt nhất trong một số tình huống nhất định hay tốt nhất trên tập các yêu cầu của bài toán cụ thể đóng vai trò quan trọng và việc lựa chọn cấu trúc dữ liệu cho bài toán cần dựa trên các yêu cầu, nhiệm vụ này. Sau đây ta xét một ví dụ minh họa cho việc lựa chọn cấu trúc dữ liệu phù hợp nhất cho bài toán giao dịch ngân hàng.

Ví dụ: Bài toán giao dịch ngân hàng như sau:

Một ngân hàng hỗ trợ nhiều giao dịch khách hàng, tuy nhiên ở đây ta chỉ xét một mô hình đơn giản ở đó khách hàng có thể mở các tài khoản, đóng tài khoản và thực hiện các giao dịch trên tài khoản như nạp tiền, rút tiền từ các tài khoản. Ta xét bài toán với hai mức độ yêu cầu: (1) các yêu cầu về cơ sở hạ tầng vật lý và tiến trình luồng công việc mà ngân hàng sử dụng trong các giao dịch; (2) các yêu cầu về hệ cơ sở dữ liệu quản lý các tài khoản khách hàng.

Mặt khác, ta nhận thấy: một khách hàng thường mở và đóng tài khoản ít thường xuyên hơn so với việc anh ta truy cập sử dụng các dịch vụ tài khoản. Do đó, anh ta sẵn lòng đợi khoảng vài phút để tạo hoặc hủy tài khoản, tuy nhiên anh ta sẽ không sẵn lòng đợi một khoảng thời gian như thế cho mỗi giao dịch thường xuyên trên tài khoản. Nhận thức này có thể xem như các đặc tả phi hình thức cho các ràng buộc về mặt thời gian của bài toán.

Một cách tiếp cận thực tế phổ biến cho các ngân hàng là cung cấp hai kiểu dịch vụ: (1) máy trả lời tự động (ATM) - hỗ trợ khách hàng truy cập các dịch vụ thường xuyên trên tài khoản như nạp

tiền, rút tiền, ..., với yêu cầu thời gian cho mỗi giao dịch này là ít; (2) nhân viên phục vụ - hỗ trợ các dịch vụ như mở, đóng tài khoản, ở đó việc thực hiện các giao dịch này có thể mất thời gian dài hơn.

Từ khía cạnh cấu trúc dữ liệu, ta thấy: các giao dịch ATM không sửa đổi cơ sở dữ liệu một cách đáng kể, để đơn giản ta giả sử rằng: nếu tiền được thêm vào hoặc bị rút, giao dịch này chỉ đơn giản thay đổi giá trị được lưu trữ trong bản ghi tài khoản; việc thêm một tài khoản mới được phép mất thời gian khoảng vài phút, xóa tài khoản không cần cần có ràng buộc về thời gian. Do vậy, khi chọn cấu trúc dữ liệu được sử dụng trong hệ cơ sở dữ liệu tài khoản khách hàng, ta nên chọn cấu trúc dữ liệu mà ít liên quan đến chi phí xóa, nhưng hiệu quả cao cho tìm kiếm và hiệu quả vừa phải cho thao tác thêm, chèn, cần thỏa mãn các ràng buộc về yêu cầu về nguồn tài nguyên của bài toán. Các bản ghi được truy cập bởi một số tài khoản khách hàng duy nhất (đôi khi gọi là truy vấn khớp - chính xác: exact - match query).

Cấu trúc dữ liệu thỏa mãn các yêu cầu trên là bảng băm (bảng băm được mô tả chi tiết trong chương 7). Vì bảng băm cho phép tìm kiếm khớp - chính xác rất nhanh, việc sửa đổi bản ghi cũng được thực hiện một cách nhanh chóng mà không yêu cầu thêm không gian. Bảng băm cũng hỗ trợ mạnh thao tác chèn các bản ghi mới trong khi thao tác xóa cũng được hỗ trợ một cách hiệu quả. Tuy nhiên, bảng băm có thể được tổ chức lại một cách định kỳ để khôi phục lại tính hiệu quả của hệ thống. Việc tổ chức lại này có thể xảy ra ngoại tuyến để không ảnh hưởng đến các giao dịch ATM.

1.2.8 Thuật toán

Thuật toán (algorithm), đôi khi còn được gọi là giải thuật là một khái niệm cơ sở của tin học. Knuth (1973) đã định nghĩa thuật toán

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

Để hiểu đầy đủ ý nghĩa của khái niệm thuật toán, chúng ta cần nêu ra năm đặc trưng sau đây của thuật toán (D.E. Knuth, 1968):

- Input: mỗi thuật toán cần có một số dữ liệu đầu vào. Đó là các giá trị cần đưa vào khi thuật toán bắt đầu làm việc. Các dữ liệu này cần được lấy từ các tập hợp giá trị cụ thể nào đó. Chẳng hạn trong thuật toán tìm ước chung lớn nhất của hai số u, v - UCLN(u,v) - thì u, v là các dữ liệu vào lấy từ tập các số nguyên.

- Output: mỗi thuật toán cần có một hoặc nhiều dữ liệu đầu ra. Đó là các giá trị có quan hệ hoàn toàn xác định với các dữ liệu vào và là kết quả của sự thực hiện thuật toán. Trong thuật toán UCLN(u,v) thì dữ liệu ra đó là g, g là ước chung lớn nhất của u và v.

- Tính xác định: Mỗi bước của thuật toán cần phải được mô tả một cách chính xác, chỉ hiểu một cách duy nhất. Hiển nhiên đây là một đòi hỏi quan trọng. Bởi vì, nếu một bước có thể hiểu theo nhiều nghĩa khác nhau, thì cùng một dữ liệu vào, những người thực hiện thuật toán khác nhau có thể dẫn đến các kết quả khác nhau. Nếu ta mô tả thuật toán bằng ngôn ngữ thông thường, khó có thể đảm bảo người đọc hiểu đúng nghĩa của người viết thuật toán. Để đảm bảo đòi hỏi này, thuật toán cần được mô tả trong các ngôn ngữ lập trình, ở đó các mệnh đề được tạo thành theo các quy tắc cú pháp nghiêm ngặt và chỉ có một nghĩa duy nhất

- Tính khả thi: tất cả các phép toán có mặt trong các bước của thuật toán phải đủ đơn giản. Điều đó có nghĩa là, các phép toán có thể thực hiện bởi con người chỉ bằng giấy trắng và bút chì trong một khoảng thời gian hữu hạn. Chẳng hạn, trong thuật toán Euclid, ta chỉ cần thực hiện các phép chia các số nguyên, các phép gán và các phép so sánh để tìm ước chung lớn nhất.
- Tính dừng: với mọi dữ liệu vào thỏa mãn các điều kiện của dữ liệu vào, thuật toán phải dừng sau một số hữu hạn bước thực hiện.

Trong thực tế thường có nhiều cách tiếp cận khác nhau để giải một bài toán. Việc thiết kế một chương trình máy tính đạt được hai mục tiêu sau đây cũng đóng một vai trò quan trọng trong việc bảo trì để nâng cấp chương trình trong tương lai:

- Thiết kế thuật toán phải dễ hiểu, dễ mã hóa và gỡ rối
- Thiết kế thuật toán tạo sự hiệu quả cho việc sử dụng các nguồn tài nguyên của máy tính.

1.2.9 Mối quan hệ giữa cấu trúc dữ liệu và thuật toán

Trong một chương trình, thuật toán phản ánh các phép xử lý, còn đối tượng xử lý của thuật toán là dữ liệu mà chứa đựng các thông tin cần thiết để thực hiện thuật toán. Để xác định được thuật toán phù hợp cần phải biết nó tác động đến loại dữ liệu nào (ví dụ để làm nhuyễn các hạt đậu, người ta dùng cách xay chứ không băm bằng dao, vì đậu sẽ văng ra ngoài) và khi chọn lựa cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào sẽ tác động đến nó (ví dụ để biểu diễn các điểm số của sinh viên người ta dùng số thực thay vì chuỗi ký tự vì còn phải thực hiện thao tác tính trung bình từ những điểm số đó).

Trong một chương trình máy tính, thuật toán và cấu trúc dữ liệu có mối quan hệ chặt chẽ với nhau, tương hỗ lẫn nhau. Mối quan hệ này đã được Niklaus Wirth biểu diễn như tiêu đề của cuốn sách "Algorithms + Data Structures = Programs", hay:

Thuật toán + Cấu trúc dữ liệu = Chương trình (2)

Với một cấu trúc dữ liệu đã chọn, sẽ có những thuật toán tương ứng, phù hợp. Ví dụ, khi chúng ta xét cấu trúc dữ liệu danh sách, do đặc tính biến động của các phần tử trong danh sách nên các phép toán phù hợp trên danh sách là thêm, sửa, xóa, tìm kiếm, thống kê,.... trong khi với cấu trúc dữ liệu mảng, phép toán thêm, và xóa phần tử thường không phù hợp do tính cố định của các phần tử trong mảng. Một ví dụ khác về sự phù hợp giữa thuật toán và cấu trúc dữ liệu là chương trình tra từ điển. Từ điển gồm ba phép toán chính là thêm, tìm, xóa. Hơn nữa số lượng các từ trong từ điển là khá lớn, do đó cấu trúc dữ liệu phù hợp nhất với các thao tác này là bảng băm. Thông thường khi cấu trúc dữ liệu thay đổi chúng sẽ kéo theo sự thay đổi về các thuật toán được cài đặt trên cấu trúc đó. Hơn nữa, một cấu trúc dữ liệu tốt sẽ giúp thuật toán xử lý trên đó có thể phát huy tác dụng tốt hơn, vừa đáp ứng nhanh vừa tiết kiệm bộ nhớ, thuật toán cũng dễ hiểu và đơn giản hơn.

Cấu trúc lưu trữ

Cấu trúc dữ liệu (CTDL) được biểu diễn trong bộ nhớ máy tính còn được gọi là Cấu trúc lưu trữ. CTDL được lưu trữ trong bộ nhớ trong được gọi là cấu trúc lưu trữ trong, ví dụ: mảng, bản ghi, ...CTDL được lưu trữ trong bộ nhớ ngoài được gọi là cấu trúc lưu trữ ngoài, ví dụ tệp tin, bảng. Trong giáo trình này chúng tôi chủ yếu tập trung trên các cấu trúc lưu trữ trong.

1.3 Phân tích thuật toán

1.3.1 Sự cần thiết của phân tích thuật toán

Trong khi giải một bài toán chúng ta có thể có một số thuật toán khác nhau, vấn đề là cần phải đánh giá các thuật toán đó để lựa chọn một thuật toán tốt nhất. Thông thường thì ta sẽ căn cứ vào các tiêu chuẩn sau:

1. Thuật toán đúng đắn;
2. Thuật toán đơn giản;
3. Thuật toán hiệu quả.

Với yêu cầu (1), để kiểm tra tính đúng đắn của thuật toán chúng ta có thể cài đặt thuật toán đó và cho thực hiện trên máy với một số bộ dữ liệu mẫu rồi lấy kết quả thu được so sánh với kết quả đã biết. Thực ra thì cách làm này không chắc chắn bởi vì có thể thuật toán đúng với tất cả các bộ dữ liệu chúng ta đã thử nhưng lại sai với một bộ dữ liệu nào đó. Hơn nữa cách làm này chỉ phát hiện ra thuật toán sai chứ chưa chứng minh được là nó đúng. Tính đúng đắn của thuật toán cần phải được chứng minh bằng toán học. Tất nhiên điều này không đơn giản và do vậy chúng ta sẽ không đề cập đến ở đây.

Khi viết một chương trình để sử dụng một vài lần thì yêu cầu (2) là quan trọng nhất. Chúng ta cần một thuật toán dễ viết chương trình để nhanh chóng có được kết quả, thời gian thực hiện chương trình không được đề cao vì dù sao thì chương trình đó cũng chỉ sử dụng một vài lần. Khi một chương trình được sử dụng nhiều lần thì thì yêu cầu tiết kiệm thời gian thực hiện chương trình, tiết kiệm không gian lưu trữ lại rất quan trọng, đặc biệt đối với những chương trình mà khi thực hiện cần dữ liệu nhập lớn do đó yêu cầu

(3) sẽ được xem xét một cách kĩ lưỡng. Ta gọi nó là hiệu quả của thuật toán. Tính hiệu quả thể hiện qua hai mặt là thời gian và không gian.

Việc phân tích và đánh giá thời gian thực hiện và không gian bộ nhớ của một thuật toán là cần thiết, vì nó là cơ sở để chúng ta so sánh độ tốt giữa các thuật toán và lựa chọn thuật toán tối ưu và phù hợp với đặc điểm của bài toán cụ thể.

1.3.2 Độ phức tạp tính toán

Độ phức tạp tính toán của thuật toán (T) phụ thuộc vào nhiều yếu tố, trước hết phụ thuộc vào độ lớn của dữ liệu đầu vào, ngoài ra nó còn phụ thuộc vào máy tính, ngôn ngữ, kỹ sảo của người lập trình,... Tuy nhiên các yếu tố này là không đồng đều do vậy không thể dựa vào chúng khi xác lập thời gian thực hiện thuật toán.

Trong cuốn sách này, theo phương pháp lý thuyết, ta coi độ phức tạp tính toán của thuật toán như là hàm số của kích cỡ dữ liệu đầu vào (ví dụ, đối với thuật toán sắp xếp một mảng gồm n phần tử, kích cỡ của dữ liệu đầu vào là số phần tử của mảng, n; tương tự với thuật toán giải hệ gồm n phương trình tuyến tính với n ẩn, ta chọn n là cỡ, ...). Do đó, ta dữ dụng ký hiệu (n), với n là kích cỡ dữ liệu của bài toán để biểu diễn độ phức tạp tính toán của thuật toán, $T(n)$ được tính bằng số lượng các phép toán sơ cấp phải tiến hành khi thực hiện thuật toán. Các phép toán sơ cấp là các phép toán mà thời gian bị chặn trên bởi một hằng số và chỉ phụ thuộc vào cách cài đặt được sử dụng (ngôn ngữ lập trình, máy tính). Chẳng hạn các phép toán số học +, -, *, /, các phép so sánh ==, !=, <=, => là các phép toán sơ cấp. Phép toán so sánh 2 xâu ký tự không được xem là phép toán sơ cấp vì thời gian thực hiện của nó phụ thuộc vào độ dài xâu ký tự.

Ví dụ 1.1: Xét thuật toán tính trung bình cộng của một dãy gồm n số nhập vào từ bàn phím:

```
scanf("%d",&n);// so luong phan tu (1)
t = 0; // luu tong    (2)
d = 1; //bien dem     (3)
while  (d <= n){      //(4)
    k = scanf("%d",&n); //(5)
    t = t+k;          // (6)
    d++;           //    (7)
}
tbc= t/n;            // (8)
prinf("%d",tbc);  //    (9)
```

Đánh giá độ phức tạp tính toán của thuật toán:

Bảng 1.1: Đánh gía độ phức tạp tính toán của thuật toán

Các lệnh	**Số lần thực hiện**
1	1
2	1
3	1
4	n
5	n
6	n
7	n
8	1
9	1
	T(n) = 4n + 5

Độ phức tạp tính toán $T(n)$ thường không chỉ phụ thuộc vào cỡ của dữ liệu vào mà còn phụ thuộc vào các dữ liệu vào cụ thể của bài toán. Ví dụ, xét thuật toán tìm phần tử x có nằm trong danh sách gồm n phần tử (a_1, a_2, ..., a_n) hay không. Giả xử thuật toán ở đây là tìm kiếm tuần tự bằng cách lần lượt so sánh x với các phần tử a_i, $i = 1, ..., n$ cho đến khi tìm thấy thì dừng hoặc cho đến khi duyệt hết danh sách mà vẫn không tìm thấy thì dừng. Trong ví dụ này, độ phức tạp tính toán T(n) = 1 nếu danh sách nhập vào chứa x và x đứng ở vị trí đầu tiên, tuy nhiên T(n) = n/2 nếu x đứng ở giữa danh sách nhập vào, hoặc T(n) = n trong trường hợp danh sách nhập vào không chứa x hoặc x nằm ở cuối danh sách nhập

vào. Do đó, rõ ràng T(n) không chỉ phụ thuộc vào kích cỡ của bài toán mà nó còn phụ thuộc vào các dữ liệu vào cụ thể của bài toán.

Khi đánh giá độ phức tạp tính toán của thuật toán bằng phương pháp toán học, chúng ta sẽ bỏ qua nhân tố phụ thuộc vào cách cài đặt, chỉ tập trung vào xác định độ lớn $T(n)$. Khi đó ta sử dụng ký hiệu toán học O lớn để mô tả độ lớn của $T(n)$.

Định nghĩa 1.3.1. *Hàm $f(n)$ được gọi là vô cùng lớn (VCL) nếu* $\lim_{n\longrightarrow\infty} f(n) = \infty$.

Định nghĩa 1.3.2. *Giả sử $f(n), g(n)$ là VCL; nếu* $\lim_{n\longrightarrow\infty} \frac{f(n)}{g(n)} = k$ *($k \neq 0$, $k \neq +\infty$) thì $f(n), g(n)$ là cùng bậc; hay $f(n) = O(g(n))$; $g(n) = O(f(n))$.*

Ví dụ:
Hàm $2n^2 + 4n + 1$ cùng bậc với n^2 vì $\lim_{n\longrightarrow\infty} \frac{2n^2+4n+1}{n^2} = 2$

Một số quy tắc xác định xác định độ phức tạp tính toán

- **Qui tắc cộng**: $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của 2 đoạn chương trình P_1 và P_2 và $T_1(n) = O(f(n))$; $T_2(n) = O(g(n))$ thì độ phức tạp tính toán 2 đoạn chương trình nối tiếp nhau là $T(n) = T_1(n) + T_2(n)$ hay $T(n) = O(max(f(n), g(n)))$
- **Qui tắc nhân**: độ phức tạp tính toán của hai đoạn chương trình P_1, P_2 lồng nhau là: $T(n) = O(f(n).g(n))$

Ví dụ 1.2: Xét thuật toán sắp xếp dãy số tăng dần bằng phương pháp lựa chọn:

```
void selectionSort(int* a[], int n){
    for(i=0;i<n;i++){
        k=i;
        for(j=i+1;j < n; j++){
            if(a[j]<a[k]{ // (*)
```

```
                k=j
            }
        }
        temp=a[k];
        a[k]=a[i];
        a[i]=temp;
    }
}
```

Trong thuật toán này, phép toán so sánh a[j]<a[k] (*) là phép tích cực. Ta thấy:

$i = 0$, phép toán (*) thực hiện n-1 lần;

$i = 1$, phép toán (*) thực hiện n-2 lần;

$i = 2$, phép toán (*) thực hiện n-3 lần;

...

$i = n - 1$, phép toán (*) thực hiện 1 lần;

Vậy tổng số lần thực hiện (*) là:

$1 + 2 + 3 + 4 + ... + (n - 1) = \frac{(n-1)n}{2} = \frac{n^2}{2} - \frac{n}{2}$

Do đó suy ra: $T(n) = O(n^2)$.

Bảng 1.2 sau đây cho ta các cấp độ phức tạp tính toán được sử dụng rộng rãi nhất và tên gọi thông thường của chúng:

Bảng 1.2: Các cấp độ của độ phức tạp tính toán

Ký hiệu ô lớn	Tên gọi thông thường
O(1)	Hằng
O(logn)	Logarit
O(n)	Tuyến tính
O(nlogn)	nlogn
O(n^2)	Bình phương
O(n^3)	Lập phương
O(2^n)	Mũ

Danh sách trên được sắp xếp theo thứ tự tăng dần của cấp thời gian thực hiện.

Để thấy rõ sự khác nhau của các cấp của độ phức tạp tính toán, ta xét ví dụ sau. Giả xử để giải quyết một bài toán nào đó, ta có

hai thuật toán A, B. Thuật toán A có độ phức tạp tính toán cấp $O(n^2)$, thuật toán B có độ phức tạp tính toán cấp $O(nlogn)$. Với n=1024, thuật toán A đòi hỏi khoảng 1048.576 phép toán sơ cấp trong khi thuật toán B chỉ đòi hỏi 10.240 phép toán sơ cấp. Nếu cần một micro-giây cho một phép toán sơ cấp thì A cần khoảng 1.05 giây trong khi thuật toán B cần 0.01 giây. Nếu n=1024*2 thì A cần 4.2 giây trong khi B cần khoảng 0.02 giây. Với n càng lớn thì thời gian thực hiện B càng ít hơn so với A. Vì vậy, nếu một bài toán nào đó đã có thuật toán với thời gian thực hiện cấp n^2, chúng ta tìm ra được một thuật toán mới với thời gian thực hiện cấp $nlogn$ thì đó là một kết quả rất có ý nghĩa, đặc biệt khi n lớn.

Bảng 1.3 sau đây cho thấy sự khác biệt lớn giữa độ phức tạp tính toán giữa các cấp khi n lớn dần.

Bảng 1.3: độ phức tạp tính toán tương ứng với các cấp độ.

log_2n	n	$nlogn$	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 mũ. Một thuật toán mà độ phức tạp tính toán của nó có cấp là các hàm loại mũ thì tốc độ rất chậm. Các hàm như $n^3, n^2, nlogn, n, logn$ được gọi là cấp đa thức. Thuật toán với độ phức tạp tính toán có cấp đa thức thì chấp nhận được.

Tóm lại: Qui tắc tổng quát để phân tích độ phức tạp tính toán của thuật toán là như sau:

- Độ phức tạp tính toán của mỗi lệnh gán, scanf, printf là O(1).
- Độ phức tạp tính toán của một chuỗi tuần tự các lệnh được

xác định bằng qui tắc cộng, hay là thời gian thi hành một lệnh nào đó lâu nhất trong chuỗi lệnh.

- Độ phức tạp tính toán cấu trúc if là thời gian lớn nhất thực hiện các lệnh trong vòng if, hoặc trong vòng else và thời gian kiểm tra điều kiện. Thường thời gian kiểm tra điều kiện là O(1).
- Độ phức tạp tính toán vòng lặp là tổng thời gian thực hiện thân vòng lặp (trên tất cả các lần lặp). Nếu thời gian thực hiện thân vòng lặp không thay đổi thì thời gian thực hiện vòng lặp là tích của số lần lặp và thời gian thực hiện 1 lần thân vòng lặp.

1.3.3 Độ phức tạp không gian

Không gian của thuật toán, ký hiệu $L(gt)$ được tính bằng số ô nhớ cơ bản được sử dụng trong thuật toán đó.

Ví dụ: Xét các bước của thuật toán tính trung bình cộng của một dãy gồm n số nhập vào từ bàn phím như trên (gt1). Khi đó $L(gt1) = 5$ (gồm 5 biến cơ bản n, biến d, biến T, biến tbc và biến k).

1.4 Ngôn ngữ diễn đạt thuật toán

Mặc dù vấn đề ngôn ngữ lập trình không được đặt ra trong cuốn sách này, nhưng vấn đề diễn đạt các thuật toán và thuận lợi cho việc cài đặt chúng ta cần phải lựa chọn một ngôn ngữ lập trình cụ thể, trong cuốn giáo trình này chúng tôi lựa chọn ngôn ngữ lập trình C. Tuy nhiên, về bản chất các cấu trúc dữ liệu và thuật toán là độc lập với ngôn ngữ lập trình, chúng ta có thể diễn đạt thuật toán và cấu trúc dữ liệu bằng ngôn ngữ lập trình mà chúng ta ưa thích (ví dụ: Pascal, C++, C#, Java, Python, ...).

Ngôn ngữ diễn đạt thuật toán là công cụ trung gian giúp giao tiếp giữa người và máy tính điện tử. Mỗi ngôn ngữ lập trình có một hệ kiểu, trong đó có một số là kiểu dữ liệu đơn hay nguyên tử, một số là các cấu trúc dữ liệu chứa các kiểu đơn/cấu trúc khác. Ngôn ngữ diễn đạt thuật toán bao gồm một tập hợp các câu lệnh tuân theo một cú pháp nhất định. Thông qua các câu lệnh mà máy tính có thể hiểu và thực hiện những công việc mà người dùng muốn máy tính làm.

1.5 Đệ quy và thuật toán đệ quy

1.5.1 Đệ quy

Đệ quy là một kĩ thuật dùng để định nghĩa một khái niệm trực tiếp hoặc gián tiếp theo chính nó. Một đối tượng được gọi 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: cho n là số nguyên dương thì giai thừa của n chính là một đối tượng đệ quy, $n! = (n-1)! * n$.

1.5.2 Thuật toán đệ quy

Thuật toán được gọi là đệ quy nếu nó chứa lời gọi đến chính nó một cách trực tiếp hoặc gián tiếp.

Ví dụ 1.3: thuật toán tìm số Fibonacci thứ n là một thuật toán đệ quy. Dãy số Fibonacci được định nghĩa như sau:

$$fib(n) = \begin{cases} 1 & ; n = 1, 2 \\ fib(n-1) + fib(n-2) & ; n > 2 \end{cases}$$

Thuật toán tìm số Fibonacci thứ n:

```
int fib(int n)
{
    if((n==1)||(n==2)){     // (1)
       return 1;            // (2)
```

```
    } else{
       return (fib(n-1)+fib(n-2));  // (3)
    }
}
```

1.5.3 Đặc điểm của thuật toán đệ quy

1. Trong thuật toán đệ quy bao giờ cũng có lời gọi đến chính tên thuật toán.

2. Mỗi lần có lời gọi thì kích thước của bài toán thu nhỏ hơn trước.

3. Có một trường hợp đặc biệt, trường hợp suy biến: Bài toán sẽ được giải quyết theo một cách khác hẳn và lời gọi đệ quy cũng kết thúc.

Trong ví dụ 1.3, các đặc điểm này được thể hiện như sau:

1. Lời gọi đến chính nó (tên của thuật toán): câu lệnh (3).

2. Mỗi lần có lời gọi thì kích thước của bài toán thu nhỏ hơn trước: Tại câu lệnh (3), kích thước bài toán được truyền vào lời gọi nhỏ hơn trước 1 và 2 đơn vị.

3. Trường hợp suy biến: Lệnh (1).

Có 2 loại đệ quy, đệ quy trực tiếp (thủ tục chứa lời gọi đến chính nó) và đệ quy gián tiếp (thủ tục chứa lời gọi đến thủ tục khác mà thủ tục này lại chứa lời gọi đến chính nó). Phương pháp đệ quy có ưu điểm là rõ ràng, chặt chẽ, thiết kế thuật toán đơn giản. Tuy nhiên nó cũng có những hạn chế là lời gọi đệ quy tốn rất nhiều thời gian, dễ phát sinh chạy vô hạn.

1.5.4 Đánh giá độ phức tạp tính toán của thuật toán đệ quy

Đánh giá độ phức tạp tính toán của thuật toán đệ quy là khá phức tạp. Để đơn giản, ta giả thiết rằng các thuật toán là thuật toán đệ quy trực tiếp, khi đó ta có thể sử dụng phương pháp tổng quát sau đây để đánh giá:

Giả sử độ phức tạp tính toán của thuật toán là $T(n)$, với n là cỡ của dữ liệu đầu vào ban đầu của bài toán; độ phức tạp tính toán của lời gọi đệ quy của nó là $T(m)$, với m là cỡ dữ liệu của bài toán sau lời gọi đệ $m < n$. Khi đó $T(n)$ được tính theo công thức quan hệ đệ quy sau đây: $T(n) = F(T(m_1, T(m_2, ...T(m_k)...)))$, trong đó $m_1, m_2, ..., m_k \leq n$. Giải phương trình đệ quy này, ta sẽ nhận được sự đánh giá của $T(n)$. Tuy nhiên, việc giải phương trình đệ quy trong nhiều trường hợp là rất khó khăn.

Ví dụ 1.4: Xét hàm đệ quy tính $n!$

```
int fact(int n)
{
    if(n <=1){      // (1)
       return 1;            // (2)
    } else{
       return n*fact(n-1);  // (3)
    }
}
```

Trong hàm này, cỡ của dữ liệu vào là n, khi $n = 1$ ta chỉ cần thực hiện lệnh return 1, do đó $T(1) = O(1)$. Với $n > 1$, cần thực hiện lệnh return n*fact(n-1) do đó thời gian T(n) là O(1) (để thực hiện phép nhân và phép gán return) cộng với T(n-1) (thực hiện lời gọi đệ quy fact(n-1)). Tóm lại, ta có quan hệ đệ quy sau: $T(1) = O(1)$;

$T(n) = O(1) + T(n-1)$;

Thay O(1) bởi hằng nào đó, ta nhận được quan hệ đệ quy sau:

$T(1) = C_1$;

$T(n) = C_2 + T(n-1)$

Để giải phương trình đệ quy, tìm $T(n)$, chúng ta áp dụng phương pháp thế lặp. Ta có phương trình đệ quy:

$T(2) = C_2 + T(1)$

$T(3) = C_2 + T(2)$

....

$T(n-1) = C_2 + T(n-2)$

$T(n) = C_2 + T(n-1)$

Bằng phép thế liên tiếp ta nhận được $T(n) = (n-1)C2 + T(1)$, hay $T(n) = (n-1)C_2 + C_1$, trong đó C_1, C_2 là các hằng số nào đó. Do đó, $T(n) = O(n)$.

Tương tự với thuật toán tìm số fibonacci thứ n (ở trên), quan hệ đệ quy là rất đơn giản, tuy nhiên việc đánh giá thời gian thực hiện của nó là không đơn giản. Bằng cách giải phương trình đặc trưng của nó (chúng ta không trình bày ở đây), chúng ta có thể tìm thấy lời giải thực hiện $T(n)$ của nó là $O(g^n)$, với $g = (1 + \sqrt{(}5))/2$, tức thuật toán này có thời gian thực hiện hàm mũ. Do đó, với n=50, thuật toán này cần khoảng 20 ngày thực thi trên máy tính lớn. Trong khi thuật toán Fibo2 dùng vòng lặp để xác định số Fibinacci thứ n chỉ cần khoảng 1 micro giây. Với n=200, thuật toán đệ quy mất khoảng 10^9 năm còn thuật toán Fibo2 chỉ cần 1.5 micro giây. Do đó, nếu có thể chúng ta nên viết các thuật toán sử dụng vòng lặp thay vì dùng phương pháp đệ quy.

Tính số fibonacci thứ n sử dụng vòng lặp:

```
int Fibo2(int n)
{   int i,j,k;
    i=1;    // (1)
    j=0;    // (2)
    for(k=1;k<n;k++){ // (3)
        j=i+j;
        i=j-i;
    }
    return j;  // (4)
}
```

Ta thấy, các lệnh gán (1), (2), và (4) có thời gian thực hiện O(1). Thân của vòng for có 3 lệnh với thời gian thực hiện là O(1), số lần lặp là n. Do đó lệnh for (3) có thời gian thực hiện là O(n). Kết hợp lại ta có thời gian thực hiện thuật toán Fibo2 là O(n).

BÀI TẬP

1. Nêu vị trí, tầm quan trọng của cấu trúc dữ liệu và thuật toán trong quy trình phát triển một ứng dụng tin học.
2. Hãy liệt kê một số cấu trúc dữ liệu tiền định trong ngôn ngữ lập trình mà Anh (Chị) lựa chọn. Sự khác biệt giữa cấu trúc dữ liệu tiền định và cấu trúc dữ liệu do người lập trình tự định nghĩa là gì?
3. Hãy phân tích và viết một số thuật toán có độ phức tạp về thời gian là $O(n)$, $O(n^2)$ và $O(n^3)$.
4. Hãy lập bảng so sánh giá trị 2 hàm $f(n) = n^2$ và $g(n) = 2^n/4$ tương ứng với các giá trị của n tăng dần. Xác định xem từ giá trị nào của n thì $g(n) \geq f(n)$.
5. Hãy đánh giá thời gian thực hiện của các đoạn chương trình sau dùng ký pháp O:

```
for(i=1;i<n;i++)
    for(j=1;j<n;j++)
        a[i][j]=b[i,j]+c[i,j];
```

```
s=0;
for(i=1;i<n;i++){
    scanf("%d",&x);
    s=s+x;
}
```

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

```
            c[i,j]=0;
            for(k=j;k<n;k++)
                c[i,j]++;
        }
```

6. Thuật toán tính ước số chung lớn nhất (UCLN) của 2 số p, q (giả sử p>q) được mô tả như sau (thuật toán Euclide):

   Gọi r là số dư của phép chia p cho q. Nếu r=0 thì q là UCLN; nếu r khác 0 thì gán p cho q, gán q cho r rồi lặp lại quá trình này. Hãy:

   - Lập bảng ghi nhận các giá trị của p,q,r khi tìm UCLN của 2 số 124 và 846.
   - Viết thuật toán đệ quy tính UCLN(p,q) và nêu các đặc điểm của thuật toán đệ quy này.
   - Phân tích xác định thời gian tính toán của thuật toán theo ký pháp O.

Chương 2

Các thuật toán sắp xếp và tìm kiếm

Nội dung của chương này là giới thiệu về các thuật toán sắp xếp cơ bản như: Selection Sort, Heap Sort, Quick Sort, Merge Sort, Radix Sort và các thuật toán tìm kiếm như tìm kiếm tuyến tính, tìm kiếm nhị phân, tìm kiếm nội suy.

2.1 Giới thiệu về các phương pháp tìm kiếm và sắp xếp

Trong các hệ lưu trữ và quản lý dữ liệu hay các phần mềm mà chúng ta sử dụng trong thực tế thì thao tác tìm kiếm được sử dụng nhiều nhất. Do dữ liệu trong các hệ thống thường lớn nên việc tìm ra giải thuật tìm kiếm nhanh chóng là mối quan tâm hàng đầu của chúng ta. Để đạt được điều này dữ liệu phải được tổ chức theo một thứ tự nào đó thì việc tìm kiếm sẽ nhanh chóng và hiệu quả hơn, vì vậy nhu cầu sắp xếp dữ liệu cũng được quan tâm hàng đầu.

Do đó, bên cạnh những giải thuật tìm kiếm thì các giải thuật sắp xếp dữ liệu không thể thiếu trong hệ quản lý thông tin trên máy tính.

Bài toán sắp xếp

Sắp xếp là quá trình xử lý một danh sách các phần tử (hoặc các mẫu tin) để đặt chúng theo một thứ tự thỏa mãn một tiêu chuẩn nào đó dựa trên nội dung thông tin lưu giữ tại mỗi phần tử.

Bài toán sắp xếp là một bài toán quan trọng, phổ biến đặc biệt là trong khoa học máy tính. Để tạo điều kiện cho việc tìm kiếm thông tin được nhanh và hiệu quả, bước then chốt, quan trọng đầu tiên trong nhiều thuật toán tìm kiếm đòi hỏi dữ liệu phải được sắp xếp theo một trình tự nhất định nào đó.

Hiện nay có nhiều giải thuật sắp xếp như: Selection sort, Insertion sort, Bubble sort, Binary Insertion sort, Shell sort, Heap sort, Quick sort, Merge sort, Radix sort...

Bài toán tìm kiếm

Bài toán tìm kiếm tổng quát có thể được phát biểu như sau: Cho mảng A có n phần tử và giá trị x cần tìm. Yêu cầu tìm xem phần tử x có trong mảng A hay không, nếu có thì trả về vị trí của phần tử x hoặc thông báo x không có trong mảng A.

Có rất nhiều các thuật toán tìm kiếm khác nhau, tuy nhiên một số thuật toán tìm kiếm cơ bản mà chúng ta hay sử dụng đó là:

Tìm kiếm tuần tự (Sequential/ Linear Search)

Tìm kiếm nhị phân (Binary Search)

Tìm kiếm nội suy

2.2 Các thuật toán sắp xếp

2.2.1 Thuật toán sắp xếp chọn

Sắp xếp chọn là một thuật toán sắp xếp tương đối dễ hiểu và đơn giản.

a) Ý tưởng

Cho 1 dãy gồm n phần tử bất kỳ từ $a[1]$ đến $a[n]$. Ta tiến hành xét từ phần tử đầu tiên $a[1]$ của dãy, chọn phần tử có khóa nhỏ nhất trong n phần tử từ $a[1]$ đến $a[n]$ và hoán vị nó với phần tử $a[1]$ ta được phần tử đầu tiên có vị trí đúng. Bỏ qua phần tử vừa được xét, tiếp tục chọn phần tử có khóa nhỏ nhất trong $n-1$ phần tử từ $a[2]$ đến $a[n]$ rồi hoán vị nó với $a[2]$. Lặp lại quá trình trên cho dãy hiện hành đến khi dãy hiện hành chỉ còn lại 1 phần tử.

Ở bước thứ i, chọn phần tử có khóa nhỏ nhất trong $n-i+1$ phần tử từ $a[i]$ đến $a[n]$ rồi hoán vị nó với $a[i]$. Cứ như vậy sau $n-1$ bước thì mảng đã được sắp xếp theo thứ tự.

Các bước sắp xếp

Bước 1: $i = 0$ // lần xử lý đầu tiên

Bước 2: Tìm phần tử nhỏ nhất $a[min]$ trong dãy hiện hành từ $a[i]$ đến $a[N-1]$

Bước 3: Hoán vị $a[min]$ và $a[i]$

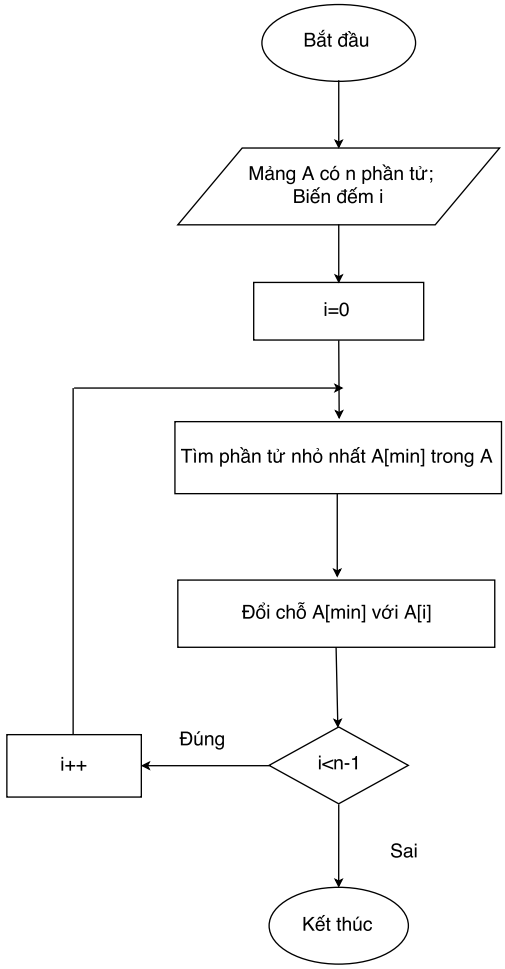
Bước 4: Nếu $i < N-1$ thì $i = i+1$; Lặp lại Bước 2

Ngược lại: Dừng

b) Lưu đồ thuật toán sắp xếp chọn (Hình 2.1)

c) Giải thuật

```
void selection_Sort(int a[], int n)
{
    int i, j, vtmin;
    for(i=0; i<n-1;i++)
    {
            vtmin = i;
            for (j=i+1, j<n, j++)
```



Hình 2.1: Lưu đồ thuật toán sắp xếp chọn

```
            if (a[j]< a[vtmin])
               vtmin = j;
         swap(a[vtmin], a[i]);
    }
}
```

Ví dụ 2.1: Cho dãy số ban đầu gồm các số: 5 6 2 2 10 12 9 10 9 3

Các bước thực hiện sắp xếp như sau:

Bước 1: 2 | 6 5 2 10 12 9 10 9 3
Bước 2: 2 2 | 5 6 10 12 9 10 9 3
Bước 3: 2 2 3 | 6 10 12 9 10 9 5
Bước 4: 2 2 3 5 | 10 12 9 10 9 6
Bước 5: 2 2 3 5 6 | 12 9 10 9 10
Bước 6: 2 2 3 5 6 9 | 12 10 9 10
Bước 7: 2 2 3 5 6 9 9 | 10 12 10
Bước 8: 2 2 3 5 6 9 9 10 | 12 10
Kết quả: Bước 9: 2 2 3 5 6 9 9 10 10 | 12

d) Đánh giá:

Các lệnh gán lấy $O(1)$ thời gian, Swap = $O(1)$ thời gian, vòng lặp for thực hiện $n-i$ lần (j chạy từ $i+1$ tới n) mỗi lần lấy $O(1)$. Vậy lấy $O(n-i)$ thời gian

Vậy thời gian tính toán là: $T(n)=\sum_{i=1}^{n-1}(n-i)=n(n-1)/2$ tương đương $O(n^2)$

2.2.2 Thuật toán sắp xếp chèn

Một trong những thuật toán sắp xếp đơn giản nhất là thuật toán sắp xếp chèn. Ví dụ về một loại sắp xếp chèn được ứng dụng trong cuộc sống hàng ngày là trong khi chơi bài. Để sắp xếp các thẻ trong tay, bạn rút thẻ, thay đổi các thẻ còn lại, và sau đó chèn thẻ rút ra vào đúng chỗ. Quá trình này được lặp lại cho đến khi tất cả các thẻ nằm trong dãy đã được sắp xếp theo đúng thứ tự.

a) Ý tưởng

Khởi đầu dãy đích (dãy đã sắp) là rỗng. Lấy dần từng phần tử từ dãy nguồn (dãy chưa sắp), chèn vào vị trí thích hợp trong dãy đã sắp. Công việc này được lặp lại cho đến khi dãy ban đầu trở thành rỗng. Ta thu được dãy đích là dãy kết quả của bài toán.

Giả sử xem phần tử $a[0]$ là một dãy đã có thứ tự (đã sắp).

Bước 1: Chèn phần tử $a[1]$ vào danh sách đã có thứ tự $a[0]$ sao cho $a[0], a[1]$ là một danh sách có thứ tự.

Bước 2: Chèn phần tử $a[2]$ vào danh sách đã có thứ tự $a[0], a[1]$ sao cho $a[0], a[1], a[2]$ là một danh sách có thứ tự ,...

Bước i: Chèn phần tử $a[i]$ vào danh sách đã có thứ tự $a[0]$, $a[1]$,..,$a[i-1]$ sao cho $a[0], a[1], .., a[i]$ là một danh sách có thứ tự.

Phần tử đang xét $a[j]$ sẽ được chèn vào vị trí thích hợp trong danh sách các phần tử đã được sắp trước đó $a[0]$, $a[1]$, ... $a[j-1]$ bằng cách so sánh $a[j]$ với $a[j-1]$ đứng ngay trước nó. Nếu $a[j] < a[j-1]$ thì đổi chỗ $a[j]$ với $a[j-1]$ và tiếp tục so sánh $a[j-1]$ với $a[j-2]$ ... Lặp cho đến khi hết dãy ($i = n-1$) như vậy mảng được sắp xếp xong.

b) Lưu đồ thuật toán (Hình 2.2)

c) Giải thuật

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

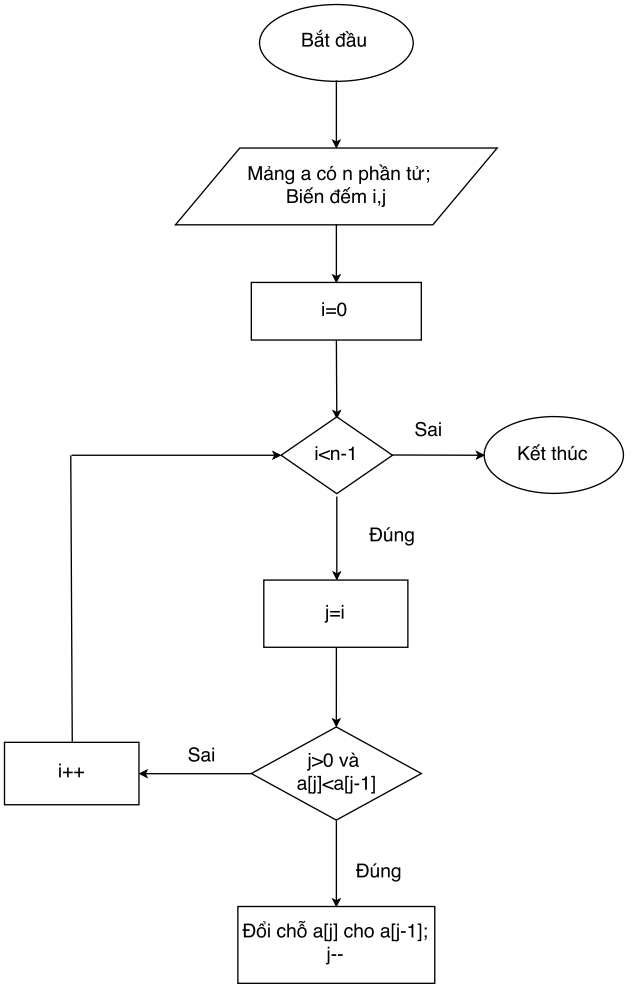
Ví dụ 2.2: Cho dãy số ban đầu gồm các số: 5 6 2 2 10 12 9 10 9 3

Bước 1 : 5 6

Bước 2: 2 5 6

Bước 3: 2 2 5 6

Bước 4: 2 2 5 6 10



Hình 2.2: Lưu đồ thuật toán sắp xếp chèn

Bước 5: 2 2 5 6 10 12

Bước 6: 2 2 5 6 9 10 12

Bước 7: 2 2 5 6 9 10 10 12

Bước 8: 2 2 5 6 9 9 10 10 12

Bước 9: 2 2 3 5 6 9 9 10 10 12 (Kết quả)

d) Đánh giá giải thuật

Độ phức tạp giải thuật phụ thuộc vào số lần so sánh và gán các

phần tử. Ở bước thứ i, tối đa cần i lần so sánh để tìm được vị trí chèn thích hợp. Do vậy số lần so sánh tối đa là:

$\sum_{i=1}^{n-i} = n(n-1)/2$.

Số phép gán giá trị phần tử: $n(n-1) + 2n$

Số phép gán chỉ số: $n(n-1)$

Vì vậy độ phức tạp là tương đương $O(n^2)$

2.2.3 Thuật toán sắp xếp nổi bọt

a)Ý tưởng

Xuất phát từ cuối dãy, đổi chỗ các cặp phần tử cạnh nhau để đưa phần tử nhỏ hơn trong cặp phần tử đó về vị trí đúng đầu dãy hiện hành, sau đó sẽ không xét đến nó ở bước tiếp theo, do vậy ở lần xử lý thứ i sẽ có vị trí đầu dãy là i. Lặp lại quá trình trên cho đến khi không còn cặp phần tử nào để xét.

Cụ thể:

Bước 1: $i = 0$ // lần xử lý đầu tiên

Bước 2: $j = N - 1$ // duyệt từ cuối dãy

Trong khi $j > i$ thực hiện:

Nếu $a[j] < a[j-1]$: hoán vị $a[j]$ và $a[j-1]$

$j = j - 1$

Bước 3: $i = i + 1$ // lần xử lý tiếp theo

Nếu $i > N - 1$ thì dừng

Ngược lại, lặp lại Bước 2

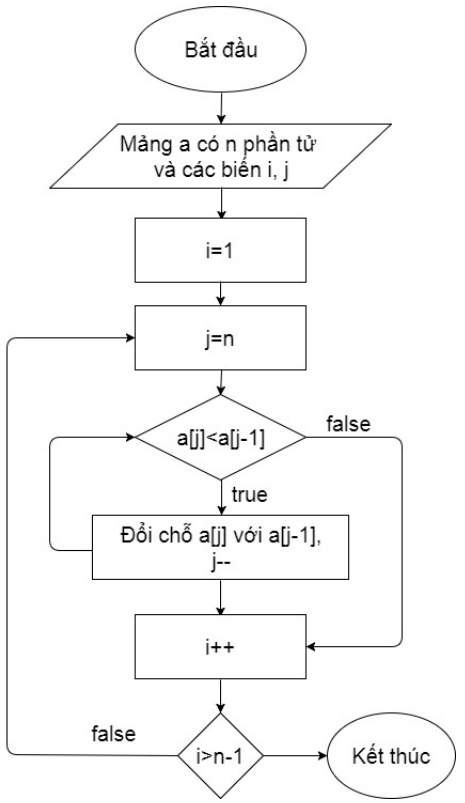
Sau n bước ta thu được mảng đã được sắp xếp theo đúng thứ tự

b) Lưu đồ thuật toán (Xem Hình 2.3)

Ví dụ 2.3 (bảng 2.1):

c) Giải thuật

```
void bubble_Sort(int a[], int n)
{
```



Hình 2.3: Lưu đồ thuật toán sắp xếp nổi bọt

Bảng 2.1: Ví dụ minh họa các bước trong sắp xếp nổi bọt

dãy ban đầu	Bước 1	Bước 2	Bước 3	Bước 4
5	2	2	2	2
6	5	2	2	2
2	6	5	3	3
2	2	6	5	5
10	3	3	6	6
12	10	9	9	9
9	12	10	9	9
10	9	12	10	10
9	10	9	12	10
3	9	10	10	12

```
    int i, j;
    for(i=0;i<=n-1;i++)
        for(j=n;j>=i;j--)
        if(a[j-1]>a[j])
            swap(a[j-1],a[j]);
}
```

d) Đánh giá giải thuật:

Ở lượt thứ i, bao giờ cũng cần $(n - i + 1)$ lần so sánh để xác định phần tử nhỏ nhất hiện hành. Do vậy số lần so sánh là:

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

Vậy độ phức tạp của thuật toán là $O(n^2)$

2.2.4 Thuật toán sắp xếp nhanh

a) Ý tưởng

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

b) Giải thuật

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

```
void quick(int left, int right)
{
    i=left; j=right; x= a[left];
```

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

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

void QuickSort()

{

quick(0, n-1);

}

c) Ví dụ 2.4:

Trong ví dụ dưới đây ở tất cả các bước ta luôn chọn phần tử chốt là phần tử đứng giữa danh sách với chỉ số của phần tử chốt được chọn là $k = int((k1 + k2)/2)$. Trong đó k1 là chỉ số đầu của mảng và k2 là chỉ số cuối của mảng.

Cho dãy a = 2 6 3 7 4 5 1

Ta có phần tử chốt là $a[4] = 7$ là phần tử lớn nhất trong dãy, ta tìm từ trái sang phải danh sách không có phần tử nào lớn hơn phần tử chốt, do đó ta đổi phần tử chốt với phần tử cuối cùng, lúc này danh sách được chia thành hai danh sách con $a[1..6]$ và $a[7..7]$

2 6 3 1 4 5 −− 7

Việc phân chia lại được tiếp tục với danh sách con $a[1..6]$. Phần tử chốt được chọn là $a[4] = 1$. Từ trái sang phải danh sách ta tìm

được phần tử đầu tiên lớn hơn $a[4]$ là $a[1] = 2$, từ phải sang phần tử đầu tiên <= 1 là chính $a[4]$. Đổi chố hai phần tử này ta được dãy

1 6 3 2 4 5 −− 7

Đi tiếp sang bên phải ta được $a[2] > 1$, ở phía ngược lại đi tiếp sang trái tìm được phần tử nhỏ hơn hoặc bằng chốt là chính $a[1] = 1$ nhưng lúc này hai đường đã chạm nhau nên ta không đổi nữa. Do vậy $a[1..6]$ được phân chia thành hai danh sách con là $a[1..1]$ và $a[2..6]$

1 −− 6 3 2 4 5 −− 7

Tiếp tục phân chia $a[2..6]$ với phần tử chốt $a[4] = 2$ ta được dãy:

1 −− 2 −− 3 6 4 5 −− 7

Tiếp tục phân chia $a[3..6]$ với phần tử chốt $a[5] = 4$ ta được dãy

1 −− 2 −− 3 4 −− 6 5 −− 7

Tiếp tục phân chia $a[3..4]$ với phần tử chốt $a[4] = 4$ và $a[5..6]$ với phần tử chốt $a[6] = 5$ ta được dãy sắp xếp theo thứ tự như sau:

1 −− 2 −− 3 −− 4 −− 5 −− 6 −− 7

d) Đánh giá

Ta nhận thấy hiệu quả của thuật toán phụ thuộc vào việc chọn giá trị khóa (hay phần tử chốt).

– **Trường hợp tốt nhất:** mỗi lần phân hoạch ta đều chọn được phần tử median (phần tử lớn hơn hay bằng nửa số phần tử và nhỏ hơn hay bằng nửa số phần tử còn lại) làm mốc. Khi đó dãy được phân hoạch thành hai phần bằng nhau, và ta cần $log(n)$ lần phân hoạch thì sắp xếp xong. Ta cũng dễ nhận thấy trong mỗi lần phân hoạch ta cần duyệt qua n phần tử. Vậy độ phức tạp trong trường hợp tốt nhất thuộc $O(nlog(n))$.

– **Trường hợp xấu nhất**: mỗi lần phần hoạch ta chọn phải phần tử có giá trị cực đại hoặc cực tiểu làm mốc. Khi đó dãy bị

phân hoạch thành hai phần không đều: một phần chỉ có một phần tử, phần còn lại có $n-1$ phần tử. Do đó, ta cần tới n lần phân hoạch mới sắp xếp xong. Vậy độ phức tạp trong trường hợp xấu nhất thuộc $O(n^2)$.

Vậy ta có độ phức tạp của Quick Sort như sau:

– Trường hợp tốt nhất: $O(nlog(n))$

– Trường hợp xấu nhất: $O(n^2)$

– Trường hợp trung bình: $O(nlog(n))$

2.2.5 Thuật toán sắp xếp trộn

a) Tư tưởng

Thuật toán sắp xếp trộn (Merge Sort) là một trong các thuật toán sắp xếp hay được sử dụng. Thuật toán này gần giống với thuật toán sắp xếp nhanh (Quick Sort) về cách sắp xếp bằng việc tách mảng cần sắp xếp thành 2 nửa, nhưng với Quick Sort thì dùng phần tử giữa làm mốc so sánh còn Merge Sort thì lại chia hẳn ra, sắp xếp từng mảng bằng cách đệ quy rồi mới "trộn" 2 mảng đã sắp xếp lại với nhau thành một mảng.

Giải thuật sắp xếp trộn tiếp tục tiến trình chia danh sách thành hai nửa cho tới khi không thể chia được nữa. Theo định nghĩa, một danh sách mà chỉ có một phần tử thì danh sách này coi như là đã được sắp xếp. Sau đó, giải thuật sắp xếp trộn kết hợp các danh sách đã xếp lại với nhau để tạo thành một danh sách mới đã được sắp xếp.

Các bước thực hiện:

1. Bước 1: Nếu chỉ có một phần tử trong danh sách thì danh sách này được xem như là đã được sắp xếp.

2. Bước 2: Chia danh sách một cách đệ qui thành hai nửa cho tới

khi không thể chia được nữa.

3. Bước 3: Kết hợp các danh sách nhỏ hơn (đã qua sắp xếp) thành danh sách mới (cũng đã được sắp xếp)

b) Giải thuật

```
mergeSort(int *A, int left, int right)
{
   if  (left >= right)
   return;
   int  mid = (left + right)/2;
   mergeSort(A, left, mid);
   mergeSort(A, mid+1, right);
   merge(a, left, mid, right);
}
```

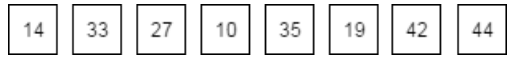
Trong đó giải thuật trộn merge() được thực hiện như sau:

```
void merge(int a[], int left, int mid, int right){
    int i, length;
    int left_end = mid - 1;
    i = left;
    length = right - left;

    int temp[length];

    /* tron cac phan tu cua 2 mang con cua a*/
    while(left <= left_end && mid <= right) {
        if (a[left] <= a[mid]) {
            temp[i++] = a[left++];
        } else {
            temp[i++] = a[mid++];
        }
    }

    /* Neu mang dau tien chua het */
    while(left <= left_end) {
        temp[i++] = a[left++];
    }

    /* Neu mang thu 2 chua het*/
    while(mid <= right) {
        temp[i++] = a[mid++];
    }

    for (i = 0; i <= length; i++, right--) {
        a[right] = temp[right];
    }
}
```

Để sắp một mảng a có n phần tử ta gọi hàm như sau:

mergeSort(a, 0 , n-1);

c) Ví dụ 2.5: Giả sử chúng ta có mảng gồm các phần tử sau:



Hình 2.4: Mảng các phần tử

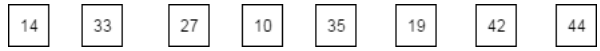
Đầu tiên, giải thuật sắp xếp trộn chia toàn bộ mảng thành hai nửa. Tiến trình chia này tiếp tục diễn ra cho đến khi không còn chia được nữa và chúng ta thu được các giá trị tương ứng biểu diễn các phần tử trong mảng. Trong hình dưới, đầu tiên chúng ta chia mảng 8 phần tử thành hai mảng 4 phần tử.

Hình 2.5: Mảng sau khi tách thành 2 dãy

Tiến trình chia này không làm thay đổi thứ tự các phần tử trong mảng ban đầu. Tương tự như trên chúng ta lại tiếp tục chia các mảng này thành 2 nửa.

Hình 2.6: Mảng sau khi tiếp tục chia

Tiến hành chia tiếp cho tới khi không còn chia được nữa.



Hình 2.7: Mảng sau khi chia

Bây giờ chúng ta tiến hành tổ hợp các phần tử lại với nhau theo như đúng cách thức mà chúng được chia ra. Đầu tiên chúng ta so sánh hai phần tử trong mỗi danh sách và sau đó tổ hợp chúng vào trong một danh sách khác theo cách thức đã được sắp xếp. Ví dụ, 14 và 33 thì hai phần tử này đã được sắp xếp theo đúng vị trí.

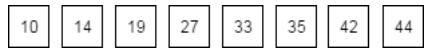
Chúng ta so sánh 27 và 10 ta thấy 27 lớn hơn 10 nên chúng ta sắp xếp lại đặt 10 ở đầu và sau đó là 27. Tương tự, chúng ta thay đổi vị trí của 19 và 35; 42 và 44 được đặt tương ứng theo đúng thứ tự từ nhỏ đến lớn.

Hình 2.8: Mảng sau khi tổ hợp

Tiếp theo chúng ta tiến hành kết hợp từng cặp danh sách chứa 2 phần tử ở trên. Chúng ta so sánh các giá trị và sau đó hợp nhất chúng lại vào trong một danh sách chứa 4 phần tử, và 4 phần tử này đều đã được sắp thứ tự.

Hình 2.9: Mảng sau khi tổ hợp ở bước tiếp theo

Sau bước kết hợp cuối cùng, ta được danh sách đã được sắp như sau:



Hình 2.10: Mảng sau khi được sắp xếp

2.2.6 Thuật toán sắp xếp vun đống

a) Giới thiệu thuật toán

Sắp xếp vun đống (*heapsort*) là một trong các phương pháp sắp xếp chọn. Ở mỗi bước của giải thuật sắp xếp chọn ta chọn phần tử lớn nhất (hoặc nhỏ nhất) đặt vào cuối (hoặc đầu) danh sách, sau đó tiếp tục với phần còn lại của danh sách. Thông thường sắp xếp chọn chạy trong thời gian $O(n^2)$. Nhưng heapsort đã giảm được độ phức tạp này bằng cách sử dụng một cấu trúc dữ liệu đặc biệt được

gọi là đống (*heap*). Đống là cây nhị phân mà trọng số ở mỗi đỉnh cha lớn hơn hoặc bằng trọng số các đỉnh con của nó. Khi danh sách dữ liệu đã được vun thành đống, gốc của nó là phần tử lớn nhất, thuật toán sẽ giải phóng nó khỏi đống để đặt vào cuối danh sách. Sắp xếp vun đống chạy trong thời gian $O(nlog(n))$.

Cụ thể, sắp xếp vun đống ($Heapsort$) dựa trên một cấu trúc dữ liệu được gọi là đống nhị phân (*binaryheap*), gọi đơn giản là đống.

Mỗi mảng $a[1..n]$ có thể xem như một cây nhị phân (có trọng số là các giá trị của mảng), với gốc là phần tử thứ nhất, con bên trái của đỉnh $a[i]$ là $a[2*i]$ con bên phải là $a[2*i+1]$ (nếu mảng bắt đầu từ 1 còn nếu mảng bắt đầu từ 0 thì 2 con là $a[2*i+1]$ và $a[2*i+2]$) (nếu $2*i <= n$ hoặc $2*i+1 <= n$, khi đó các phần tử có chỉ số lớn hơn không có con, do đó là lá).

Một cây nhị phân, được gọi là đống cực đại nếu khóa của mọi nút không nhỏ hơn khóa các con của nó. Khi biểu diễn một mảng a[ ] bởi một cây nhị phân theo thứ tự tự nhiên điều đó nghĩa là $a[i] >= a[2*i]$ và $a[i] >= a[2*i+1]$ với mọi $i = 1..int(n/2)$. Ta cũng sẽ gọi mảng như vậy là đống. Như vậy trong đống $a[1]$ (ứng với gốc của cây) là phần tử lớn nhất. Mảng bất kỳ chỉ có một phần tử luôn luôn là một đống.

Một đống cực tiểu được định nghĩa theo các bất đẳng thức ngược lại: $a[i] <= a[2*i]$ và $a[i] <= a[2*i+1]$. Phần tử đứng ở gốc cây cực tiểu là phần tử nhỏ nhất.

Vun đống

Việc sắp xếp lại các phần tử của một mảng ban đầu sao cho nó trở thành đống được gọi là vun đống.

Vun đống tại đỉnh thứ i

Nếu hai cây con gốc $2*i$ và $2*i+1$ đã là đống thì để cây con gốc i trở thành đống chỉ việc so sánh giá trị $a[i]$ với giá trị lớn hơn

trong hai giá trị $a[2 * i]$ và $a[2 * i + 1]$, nếu $a[i]$ nhỏ hơn thì đổi chỗ chúng cho nhau. Nếu đổi chỗ cho $a[2 * i]$, tiếp tục so sánh với con lớn hơn trong hai con của nó cho đến khi gặp đỉnh lá. (Thủ tục $DownHeap$ trong giả mã dưới đây)

Vun một mảng thành đống

Để vun mảng $a[1..n]$ thành đống ta vun từ dưới lên, bắt đầu từ phần tử $a[j]$ với $j = Int(n/2)$ ngược lên tới $a[1]$. (Thủ tục $MakeHeap$ trong giả mã dưới đây)

Sắp xếp bằng vun đống gồm các bước:

1. Đổi chỗ ($Swap$): Sau khi mảng a[1..n] đã là đống, lấy phần tử $a[1]$ trên đỉnh của đống ra khỏi đống đặt vào vị trí cuối cùng n, và chuyển phần tử cuối cùng $a[n]$ lên đỉnh đống thì phần tử $a[n]$ đã được đứng đúng vị trí.

2. Vun lại: Phần còn lại của mảng $a[1..n-1]$ chỉ khác cấu trúc đống ở phần tử $a[1]$. Vun lại mảng này thành đống với $n-1$ phần tử.

3. Lặp: Tiếp tục với mảng $a[1..n-1]$. Quá trình dừng lại khi đống chỉ còn lại một phần tử.

Ví dụ 2.6:

Cho mảng $a = (2, 3, 5, 6, 4, 1, 7)$

Ở đây $n = 7$. Các phần tử từ $a[4]$ đến $a[7]$ là lá.

Vun đống

Vun cây gốc $a[3]$ ta được mảng $a = (2, 3, 7, 6, 4, 1, 5)$

Vun cây gốc $a[2]$ ta được mảng $a = (2, 6, 7, 3, 4, 1, 5)$

Vun cây gốc $a[1]$ ta được mảng $a = (7, 6, 5, 3, 4, 1, 2)$

Bây giờ $a = (7, 6, 5, 3, 4, 1, 2)$ đã là đống.

Sắp xếp

Đổi chỗ $a[1]$ với $a[7] : a = (2, 6, 5, 3, 4, 1, 7)$

và vun lại mảng $a[1..6]$ ta được mảng $a = (6, 4, 5, 3, 2, 1, 7)$
Đổi chỗ $a[1]$ với $a[6]$: $a = (1, 4, 5, 3, 2, 6, 7)$
và vun lại mảng $a[1..5]$ ta được mảng $a = (5, 4, 1, 3, 2, 6, 7)$
Đổi chỗ $a[1]$ với $a[5]$: $a = (1, 4, 2, 3, 5, 6, 7)$
và vun lại mảng $a[1..4]$ ta được mảng $a = (4, 3, 1, 2, 5, 6, 7)$
Đổi chỗ $a[1]$ với $a[4]$: $a = (1, 3, 2, 4, 5, 6, 7)$
và vun lại mảng $a[1..3]$ ta được mảng $a = (3, 2, 1, 4, 5, 6, 7)$
Đổi chỗ $a[1]$ với $a[3]$: $a = (2, 1, 3, 4, 5, 6, 7)$
và vun lại mảng $a[1..2]$ ta được mảng $a = (2, 1, 3, 4, 5, 6, 7)$
Đổi chỗ $a[1]$ với $a[2]$: $a = (1, 2, 3, 4, 5, 6, 7)$
Mảng còn lại chỉ một phần tử. Quá trình sắp xếp đã xong.

b) Giải thuật

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

const int n = 10;
typedef int keytype;
typedef float othertype;
typedef struct recordtype{
        keytype key;
        othertype otherfields;
};
// khai bao mang a co n phan tu
recordtype a[n];

void Swap(recordtype &x, recordtype &y)
{
     recordtype temp;
     temp = x;
     x = y;
     y = temp;
}

void PushDown(int first, int last)
{
    //while (r <= (last - 1) / 2)
    if (first == last || first > (last - 1) / 2) return;
    if (last == 2 * first + 1) {
       if (a[first].key > a[last].key)
           Swap(a[first], a[last]);
           //first = last;
           return;
```

```
        }else
        if ((a[first].key > a[2*first+1].key) && (a[2*first+1].key <=
            a[2*first+2].key))
        {
           Swap(a[first], a[2*first+1]);
           PushDown(2 * first + 1, last);
        }else
        if ((a[first].key > a[2*first+2].key) && (a[2*first+2].key <
            a[2*first+1].key))
        {
           Swap(a[first], a[2*first+2]);
           PushDown(2 * first + 2, last);
        }
        else return; //first = last;
}

void HeapSort(void)
{       int i;
/*1*/ for(i = (n-2) / 2; i >= 0; i--)
/*2*/ PushDown(i, n-1);
/*3*/ for(i = n-1; i>=2; i--) {
/*4*/ Swap(a[0], a[i]);
/*5*/ PushDown(0, i-1);
          }
/*6*/ Swap(a[0], a[1]);
}

void readList(recordtype a[])
{
     for (int i = 0; i < n; i++)
     {
         printf("Phan tu %d = ", i+1);
         scanf("%d",&a[i]);
     }
}
void printList(recordtype a[])
{
     for (int i = 0; i < n; i++)
     {
         printf("%d ", a[i]);
     }
}
int main()
{
    printf("Nhap %d phan tu cho danh sach.\n", n);
    readList(a);
    printf("Danh sach sau khi duoc nhap: \n");
    printList(a);
    HeapSort();
    printf("\nDanh sach sau khi duoc sap xep: \n");
    printList(a);
    getch();
```

```
    return 0;
}
```

c) Ứng dụng của heap

Ngoài giải thuật sắp xếp vun đống, cấu trúc đống (*heap*) còn được ứng dụng trong nhiều giải thuật khác, khi cần lấy ra nhanh chóng các phần tử lớn nhất (hoặc nhỏ nhất) của một dãy phần tử, chẳng hạn trong hàng đợi có ưu tiên trong đó tiêu chuẩn ưu tiên là có khóa lớn nhất (hoặc nhỏ nhất). Ví dụ giải thuật tìm bộ mã Huffman cho một bảng tần số của các kí tự.

2.2.7 Thuật toán sắp xếp cơ số

a) Giới thiệu về thuật toán sắp xếp cơ số

Thuật toán sắp xếp cơ số ($RadixSort$) là một thuật toán tiếp cận theo một hướng hoàn toàn khác so với các thuật toán sắp xếp khác. Nếu như trong các thuật toán khác, cơ sở để sắp xếp luôn là việc so sánh giá trị của 2 phần tử thì $RadixSort$ lại dựa trên nguyên tắc phân loại thư của bưu điện. Vì lý do đó Radix Sort còn có tên là $Postman's Sort$. $RadixSort$ không hề quan tâm đến việc so sánh giá trị của phần tử mà bản thân việc phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.

Ta biết rằng, để chuyển một khối lượng thư lớn đến tay người nhận ở nhiều địa phương khác nhau, bưu điện thường tổ chức hệ thống phân loại thư phân cấp. Trước tiên là nhóm các thư đến cùng một tỉnh hay một thành phố sẽ được xếp chung vào một lô để gửi đến các tỉnh thành tương ứng. Sau đó bưu điện ở các tỉnh, thành này lại thực hiện công việc tương tự là xếp các thư đến cùng một quận, huyện vào chung một lô để gửi đến các quận, huyện tương ứng. Đến các quận, huyện lại thực hiện tương tự như vậy. Cứ như vậy, các bức thư sẽ được trao đến tận tay người nhận một cách có

hệ thống mà công việc xếp thư lại không quá nặng nhọc.

b) Mô phỏng qui trình

Để sắp xếp dãy $a_1, a_2, ..., a_n$, giải thuật *RadixSort* thực hiện như sau: - Trước tiên, ta có thể giả sử mỗi phần tử a_i trong dãy $a_1, a_2, ..., a_n$ là một số nguyên có tối đa m chữ số.

- Ta phân loại các phần tử lần lượt theo các chữ số hàng đơn vị, hàng chục, hàng trăm, hàng nghìn ... tương tự việc phân loại thư theo tỉnh thành, quận huyện, phường xã, thôn xóm.

Thuật toán được thực hiện theo các bước sau:

Giả sử k là biến cho biết chữ số dùng để phân loại hiện hành

1. Bước 1: gán $k = 0$; // k là chữ số phân loại, $k = 0$ là hàng đơn vị, $k = 1$ là hàng chục...

2. Bước 2: // Tạo các lô chứa phần tử phân loại từ $B[0]...B[9]$

   Khởi tạo lô từ $B[0]...B[9]$ rỗng, chưa chứa phần tử nào, $B[i]$ sẽ chứa các phần tử có chữ số thứ k là i.

3. Bước 3: *For i=1 to n do*

   Đặt $a[i]$ vào dãy $B[j]$ với j là chữ số thứ k của $a[i]$.

   Nối $B[0], B[1], ..., B[9]$ lại theo đúng trình tự thành dãy a.

4. Bước 4:

   $k = k + 1$

   Nếu $k < m$: thì quay lại Bước 2. //m là số lượng chữ số tối đa của các số trong mảng

   Ngược lại: Dừng thuật toán.

c) Thuật toán sắp xếp radix sort

Giả sử có Mảng $a[MAX]$ chứa các phần tử của mảng cần sắp xếp tăng.

Mảng $B[10][MAX]$ chứa các dãy số được phân tạm thời theo các con số. Ví dụ $B[0]$ chứa các phần tử có con số ở hàng đơn vị là 0 ví dụ $100, 210, 320....B[1]$ chứa các phần tử có con số ở hàng đơn vị là 1. Khi đó với mỗi dòng của B thì sẽ chia các phần tử có con số ở hàng thứ i (i từ $0-9$), các giá trị cột j sẽ lần lượt chứa các phần tử có cùng con số ở hàng thứ i.

Mảng $Length[10]$ chứa số lượng các phần tử của các dòng $B[i]$. Ví dụ $B[0]$ có 3 phần tử thì $Length[0] = 3, B[5]$ có 2 phần tử thì $B[5] = 2$. Tại mỗi bước trước khi phân các phần tử vào mảng B thì các $Length[i]$ được khởi tạo là 0.

Cài đặt thuật toán:

```
void radixSort(long a[], int n){
    int i, j, d;
    int h = 10;
    long B[10][MAX];
    int  Length [10];
    // Maxdigit so phan tu toi da cua a[i]
    for(d = 0; d < Maxdigit; d++)
    {
        // init B[i] la 0
        for( i = 0; i < 10; i++)
            Length [i] = 0;
        // thuc hien phan lo cac phan tu theo con so hang thu d tinh tu //cuoi

        for(i = 0; i < n; i++) // duyet qua tat ca cac phan tu cua mang
        {
            digit = (a[i] % h) / (h/ 10); // Lay con so theo hang h
            // dua vao day B[digit] o cot Length [digit]
            B[digit][ Length [digit]++] = a[i];
        }// end for i

        for(i = 0; i < 10; i++)
        {
            // lay tung phan tu cua tung day B[i]
            for(j =0; j < Length [i]; j++)
                a[num++] = B[i][j];
        }// end for i
        h *= 10;   // qua hang ke tiep.
    }// end for d
}// end radixSort
```

d) Ví dụ 2.7: Giả sử chúng ta có dãy số như sau: 493 –

812, 715, 710, 195, 437, 582, 340, 385 Đầu tiên sắp xếp các số theo hàng đơn vị: 493, 812, 715, 710 ,195, 437, 582, 340, 385. Ta được kết quả như bảng sau:

Bảng 2.2: Sắp xếp theo hàng đơn vị

Chứ số hàng đơn vị	**Dãy con**
0	710 340
1	-
2	812 582
3	493
4	-
5	715 195 385
6	-
7	437
8	-
9	-

Lưu ý những phần tử này được đưa vào trong dãy con theo thứ tự tìm thấy, do đó chúng ta có thể thấy là các dãy con chưa được sắp xếp theo thứ tự. Lúc này chúng ta thu được một danh sách gồm các dãy con từ 0 đến 9 như sau:

710, 340, 812, 582, 493, 715, 195, 385, 437

Tiếp tục chúng ta phân loại các phần tử của dãy trên theo con số của hàng chục.

710, 340, 812, 582, 493, 715, 195, 385, 437

Xem bảng sau:

Lúc này chúng ta thu được danh sách như sau:

710, 812, 715, 437, 340, 582, 385, 493, 195

Thực hiện tiếp với phân loại các con số hàng trăm:

710, 812, 715, 437, 340, 582, 385, 493, 195

Xem bảng dưới đây:

Thu được danh sách các phần tử từ dãy con được phân loại theo hàng trăm từ 0 đến 9.

195, 340, 385, 437, 493, 582, 710, 715, 812 như vậy dãy đã được

Bảng 2.3: Sắp xếp theo chứ số hàng chục

Chứ số hàng chục	**Dãy con**
0	-
1	710 812 715
2	-
3	437
4	340
5	-
6	-
7	-
8	582 385
9	493 195

Bảng 2.4: Sắp xếp theo chữ số hàng trăm

Chứ số hàng trăm	**Dãy con**
0	-
1	195
2	-
3	340 385
4	437 493
5	582
6	-
7	710 715
8	812
9	-

sắp

Tóm lại để sắp xếp dãy $a[1], a[2], ..., a[n]$ giải thuật *RadixSort* thực hiện như sau:

Xem mỗi phần tử $a[i]$ trong dãy $a[1]...a[n]$ là một số nguyên có tối đa m chữ số Lần lượt phân loại các chữ số theo hàng đơn vị, hàng chục, hàng trăm...Tại mỗi bước phân loại ta sẽ nối các dãy con từ danh sách đã phân loại theo thứ tự 0 đến 9. Sau khi phân loại xong ở hàng thứ m cao nhất ta sẽ thu được danh sách các phần tử được sắp.

e) Đánh giá giải thuật

Với một dãy n số, mỗi số có tối đa m chữ số thì thuật toán sẽ

thực hiện m lần các thao tác phân lô và ghép lô. Trong các thao tác đó, mỗi phần tử chỉ được xét đúng 1 lần và khi ghép cũng vậy.

Sau lần phân phối thứ k các phần tử của dãy A vào các lô $B_0, B_1.., B_9$ và lấy ngược lại, nếu chỉ xét đến $k+1$ chữ số của các phần tử trong B thì ta sẽ thu được một mảng tăng dần nhờ trình tự lấy ra từ 0 đến 9.

Thuật toán có độ phức tạp tuyến tính (độ phức tạp là O(n))nên hiệu quả khi sắp dãy có rất nhiều phần tử đặc biệt là khi khóa dùng để sắp xếp không quá dài so với số lượng phần tử.

Sở dĩ nói độ phức tạp của thuật toán là O(N) là do phần tử có trị tuyệt đối lớn nhất của dãy không phụ thuộc vào N, số lần sắp xếp coi là hằng số, mỗi lần sắp xếp ta nhìn mỗi phần tử 1 lần, do đó độ phức tạp của thuật toán là O(n).

Trong thực tế khi cài đặt thuật toán với các mảng có khóa sắp xếp là các chuỗi thì sẽ thuận tiện hơn với khóa là các số như trong ví dụ đã nêu.

2.2.8 So sánh sự khác nhau giữa các thuật toán sắp xếp

Trong các phần trên chúng ta đã nắm được tư tưởng và giải thuật của các phương pháp sắp xếp cơ bản. Trong phần này chúng ta sẽ tiến hành phân tích ưu, nhược điểm và độ phức tạp của một số thuật toán sắp xếp trên.

Thuật toán sắp xếp nổi bọt - *Bubblesort*: Có ưu điểm là dễ cài đặt, dễ sử dụng, code ngắn gọn. Tuy nhiên thuật toán này lại khá chậm, độ phức tạp của thuật toán này trong trường hợp xấu nhất là $O(n^2)$ và trường hợp tốt nhất là $O(n)$.

Thuật toán sắp xếp chèn - *Insertsort*: là một thuật toán khá đơn giản, chạy nhanh khi mảng nhỏ hoặc mảng đã được sắp xếp một phần. Tuy nhiên cũng tương tự như thuật toán sắp xếp nổi

bọt, thuật toán này có hiệu suất thấp. Trong trường hợp tốt nhất (tất cả các phần tử đã được sắp xếp sẵn) thì độ phức tạp của thuật toán là $O(n)$ (vì chỉ cần duyệt n lần từ trái qua, không có chèn). Tuy nhiên, với trường hợp xấu nhất thì độ phức tạp có thể lên tới $O(n^2)$.

Thuật toán sắp xếp chọn - *Selectionsort*: thuật toán này cũng là một trong các thuật toán đơn giản, dễ cài đặt. Có ưu điểm là chạy nhanh hơn khi mảng được sắp xếp một phần. Thuật toán này mặc dù có ưu việt là ít đổi chỗ các phần tử nhất, tuy nhiên vì ta phải tìm x, nên độ phức tạp của thuật toán cũng tương đương là $O(n^2)$.

Thuật toán sắp xếp trộn - $Mergesort$: Có ưu điểm là nhanh hơn hẳn 3 thuật toán trên, độ phức tạp của thuật toán là $O(nlog(n))$. Tuy nhiên chúng ta phải hiểu rõ cơ chế của thuật toán, và việc cài đặt tương đối khó.

Thuật toán sắp xếp vun đống - $Heapsort$: Tương tự $Mergesort$, $Heapsort$ có độ phức tạp của thuật toán là $O(nlog(n))$. Vì vậy nên việc cài đặt và sử dụng $Heapsort$ tương đối khó, và cần phải hiểu về cây nhị phân mới có thể sử dụng tốt.

Thuật toán sắp xếp nhanh - $Quicksort$: Quicksort cũng tương tự như $Mergesort$, cũng có độ phức tạp thuật toán trong trường hợp tốt nhất là $O(nlog(n))$, trường hợp xấu nhất là $O(n^2)$. Tuy nhiên, việc cài đặt $Quicksort$ ngắn gọn hơn hẳn so với $Mergesort$ hay $Heapsort$.

2.2.9 Thuật toán sắp xếp ngoài

Trong các giải thuật sắp xếp trình bày ở trên, thứ tự sắp xếp được thực hiện trên dãy/mảng và các dữ liệu vào là khá nhỏ có thể chứa hết ở bộ nhớ trong nên được gọi là sắp xếp nội. Với sắp

xếp nội thì toàn bộ dữ liệu cần sắp xếp được đưa vào trong bộ nhớ trong (RAM). Tuy không gian bộ nhớ trong là giới hạn cho việc lưu trữ dữ liệu, nhưng tốc độ sắp xếp lại tương đối nhanh.

Đối với những bài toán có số lượng dữ liệu vượt quá khả năng lưu trữ của bộ nhớ trong ví dụ như nếu ta muốn xử lý phiếu điều tra dân số trong toàn quốc hay thông tin về quản lý đất đai cả nước chẳng hạn. Để giải quyết các bài toán đó chúng ta phải dùng bộ nhớ ngoài để lưu trữ và xử lý. Các thiết bị lưu trữ ngoài như băng từ, đĩa từ đều có khả năng lưu trữ lớn nhưng đặc điểm truy nhập hoàn toàn khác với bộ nhớ trong. Kiểu dữ liệu tập tin là kiểu thích hợp nhất cho việc biểu diễn dữ liệu được lưu trong bộ nhớ ngoài. Vì vậy để giải quyết các bài toán sắp xếp có số lượng dữ liệu lớn thì chúng ta phải sử dụng các giải thuật sắp xếp ngoài (sắp xếp thứ tự trên tập tin/file). Sắp xếp ngoài là kiểu sắp xếp dữ liệu được tổ chức như một tập tin hoặc tổng quát hơn, sắp xếp ngoài là việc sắp xếp dữ liệu được lưu trữ trong bộ nhớ ngoài.

Do số phần tử dữ liệu thường lớn nên một phần dữ liệu cần sắp xếp được đưa vào trong bộ nhớ trong (RAM), phần còn lại được lưu ở bộ nhớ ngoài (DISK). Do vậy, tốc độ sắp xếp dữ liệu trên tập tin tương đối chậm. Các giải thuật sắp xếp ngoài bao gồ các nhóm sau:

- Sắp xếp bằng phương pháp trộn (*Mergesort*)
- Sắp xếp theo chỉ mục (*Indexsort*)

2.3 Các thuật toán tìm kiếm

2.3.1 Thuật toán tìm kiếm tuyến tính

a) Giới thiệu thuật toán tìm kiếm tuyến tính

Tìm kiếm tuyến tính (*linearsearch*) hay tìm kiếm tuần tự

(*sequentialsearch*) là phương pháp tìm kiếm một phần tử cho trước trong một danh sách bằng cách duyệt từng phần tử của danh sách đó cho đến khi tìm thấy phần tử cần tìm hay đã duyệt qua toàn bộ danh sách. Tại mỗi bước, khoá của danh sách sẽ được so sánh với giá trị cần tìm. Quá trình tìm kiếm kết thúc khi đã tìm thấy khoá thoả mãn hoặc đã duyệt hết danh sách.

Ví dụ: cho danh sách A gồm các phần tử: $5, 18, 20, 7, 10, 15, 19$. Tìm xem phần tử $x = 7$ có trong danh sách hay không?

Ta lần lượt so sánh phần tử $x = 7$ với các phân tử của danh sách và tìm được phần tử $x = 7$ tại vị trí thứ 4 của danh sách.

Tìm phần tử $x = 25$ có trong danh sách hay không?

Cũng tương tự như trên ta lần lượt so sánh phần tử $x = 25$ với các phần tử có trong danh sách, và với ví dụ này ta duyệt đến cuối danh sách cũng ko tìm thấy phần tử nào có giá trị $x = 25$. Kết luận, phần tử $x = 25$ không có trong danh sách.

b) Giải thuật tìm kiếm tuyến tính

Các bước thực hiện giải thuật:

Đầu vào: mảng $a[]$ gồm n phần tử và giá trị x cần tìm.

Đầu ra: Trả về $True$ nếu tìm thấy, ngược lại trả về $False$

1. Bước 1:

   $i = 0$; // bắt đầu từ phần tử đầu tiên của dãy

2. Bước 2:

   So sánh $a[i]$ với x, có 2 khả năng:

   Nếu $a[i] = x$: Tìm thấy dừng

   Ngược lại, chuyển sang Bước 3.

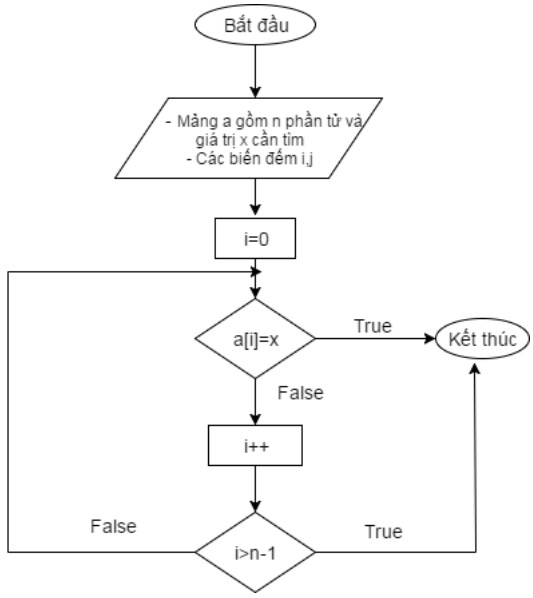
3. Bước 3:

   $i = i + 1$; // xét tiếp phần tử kế trong mảng

Nếu $i > N - 1$: Hết mảng, không tìm thấy. Dừng

Ngược lại: Lặp lại Bước 2.

Lưu đồ thuật toán (xem hình 2.11)



Hình 2.11: Sơ đồ tìm kiếm tuyến tính

Cài đặt:

```
int linearSearch(int a[], int N, int x)
{
   int i=0;
   while ((i<N) && (a[i]!=x ))
         i++;
   if(i==N)
         return -1; //tim het mang
   else
          return i; //a[i] la phan tu co khoa x
}
```

Thủ tục này tiến hành duyệt từ đầu mảng. Nếu tại vị trí nào đó, giá trị của phần tử bằng với giá trị cần tìm thì hàm trả về chỉ số tương ứng của phần tử trong mảng. Nếu không tìm thấy giá trị trong toàn bộ mảng thì hàm trả về giá trị là -1.

Thuật toán tìm kiếm tuần tự có thời gian thực hiện là $O(n)$.

Trong trường hợp xấu nhất, thuật toán mất n lần thực hiện so sánh và mất khoảng $n/2$ lần so sánh trong trường hợp trung bình.

2.3.2 Thuật toán tìm kiếm nhị phân

Thuật toán tìm kiếm nhị phân là một thuật toán dùng để tìm kiếm phần tử trong một danh sách đã được sắp xếp (tìm kiếm kiểu "tra từ điển"). Thuật toán hoạt động như sau: trong mỗi bước, so sánh phần tử cần tìm với phần tử nằm ở chính giữa danh sách. Nếu hai phần tử bằng nhau thì phép tìm kiếm thành công và thuật toán kết thúc. Nếu chúng không bằng nhau thì kiểm tra xem phần tử cần tìm nhỏ hơn hay lớn hơn phần tử ở giữa danh sách. Nếu nhỏ hơn thì thuật toán lặp lại bước so sánh trên với nửa đầu của danh sách, còn nếu phần tử cần tìm lớn hơn phần tử giữa thì tiến hành lặp lại bước so sánh với nửa sau của danh sách. Sau mỗi bước số lượng phần tử trong danh sách cần xem xét giảm đi một nửa, nên thời gian thực thi của thuật toán là $O(logn)$.

Các bước thực hiện như sau:

- Bước 1:

  $left = 0; right = N - 1;$ // tìm kiếm trên tất cả các phần tử

- Bước 2:

  $mid = (left + right)/2;$//lấy mốc so sánh

  So sánh $a[mid]$ với x, có 3 khả năng:

  + $a[mid] = x$: Tìm thấy. Dừng lại.

  + $a[mid] > x$: // tìm tiếp x trong dãy con a(left)...a(mid-1):

  $right = mid - 1;$

  + $a[mid] < x$: //tìm tiếp x trong dãy a(mid+1)...a(right):

  $left = mid + 1;$

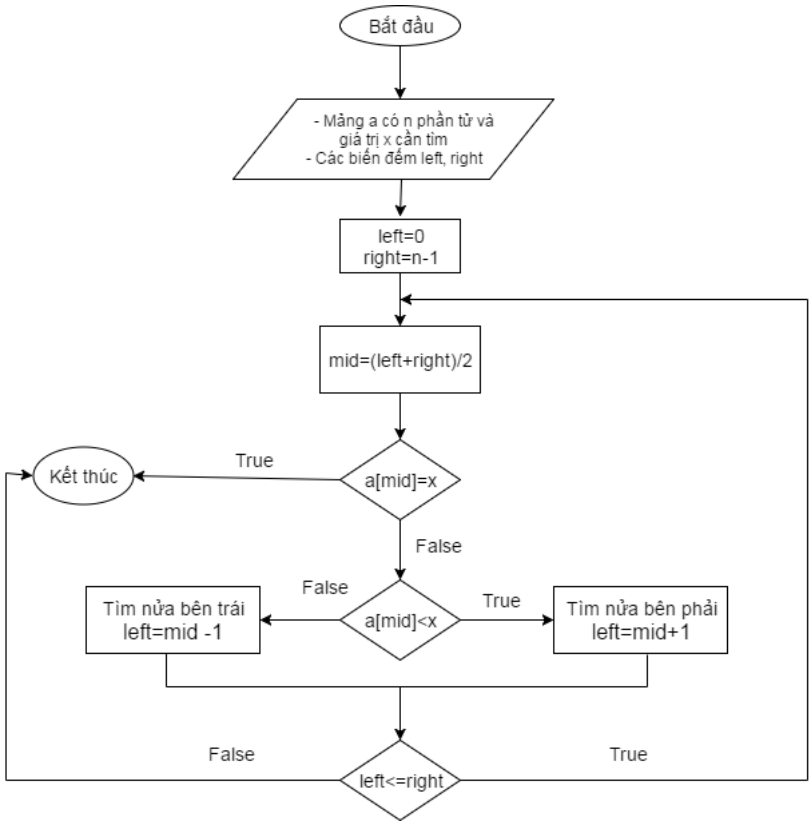
- Bước 3:

  Nếu $left <= right$ //còn phần tử chưa xét=>tìm tiếp

  Lặp lại Bước 2

  Ngược lại: Dừng;//Đã xét hết phần từ

Lưu đồ thuật toán: (xem hình 2.12)



Hình 2.12: Lưu đồ tìm kiếm nhị phân

Thuật toán:

```
int binarySearch(int a[], int N, int x)
{
    int left = 0, right = N - 1;
    int mid;
do
{
    mid = (left + right) / 2;
    if (x == a[mid])
```

```
        return mid;//Thay x tai mid
     else if (x < a[mid])
        right = mid - 1;
    else
        left = mid + 1;
}
while (left <= right);
   return -1;// tim het day ma khong co x
}
```

2.3.3 Thuật toán tìm kiếm nội suy

a) Giới thiệu thuật toán tìm kiếm nội suy

Tìm kiếm nội suy (Interpolation Search) là biến thể cải tiến của tìm kiếm nhị phân ($BinarySearch$). Để giải thuật tìm kiếm này làm việc chính xác thì tập dữ liệu phải được sắp xếp.

Có một số tình huống mà vị trí của dữ liệu cần tìm có thể đã được biết trước. Ví dụ, trong trường hợp danh bạ điện thoại, nếu chúng ta muốn tìm số điện thoại của Mai chẳng hạn. Trong trường hợp này, Linear Search và cả Binary Search có thể là chậm khi thực hiện tìm kiếm, khi mà chúng ta có thể trực tiếp nhảy tới phần không gian bộ nhớ có tên bắt đầu với M được lưu giữ.

Tìm kiếm nội suy tìm kiếm một phần tử cụ thể bằng việc tính toán vị trí dò ($ProbePosition$). Ban đầu thì vị trí dò là vị trí của phần tử nằm ở giữa nhất của tập dữ liệu. Nếu tìm thấy kết nối thì chỉ mục của phần tử được trả về.

Để chia danh sách thành hai phần, chúng ta sử dụng phương thức sau:

$mid = Lo + ((Hi - Lo)/(A[Hi] - A[Lo])) * (X - A[Lo])$

Trong đó:

A: danh sách

L_o: chỉ mục thấp nhất của danh sách

H_i: chỉ mục cao nhất của danh sách

$A[n]$: giá trị được lưu giữ tại chỉ mục n trong danh sách

Nếu phần tử cần tìm có giá trị lớn hơn phần tử ở giữa thì phần tử cần tìm sẽ ở mảng con bên phải phần tử ở giữa và chúng ta lại tiếp tục tính vị trí dò; nếu không phần tử cần tìm sẽ ở mảng con bên trái phần tử ở giữa. Tiến trình này tiến tụp diễn ra trên các mảng con cho tới khi kích cỡ của mảng con giảm về 0.

Độ phức tạp thời gian chạy của $InterpolationSearch$ là $(log(logn))$, trong khi của $BinarySearch$ là $(log(n))$

b) Giải thuật tìm kiếm nội suy

```
int find(int data) {
   int lo = 0;
   int hi = MAX - 1;
   int mid = -1;
   int comparisons = 1;
   int index = -1;
   while(lo <= hi) {
      printf("\nSo sanh lan thu %d \n" , comparisons ) ;
      printf("lo : %d, list[%d] = %d\n", lo, lo, list[lo]);
      printf("hi : %d, list[%d] = %d\n", hi, hi, list[hi]);
      comparisons++;
      // phan tu chot (probe) tai vi tri trung vi
      mid = lo + (((double)(hi - lo) / (list[hi] - list[lo])) * (data - list[lo]));
      printf("Vi tri trung vi = %d\n",mid);
      // tim thay du lieu
      if(list[mid] == data) {
         index = mid;
         break;
      }else {
         if(list[mid] < data) {
            // neu du lieu co gia tri lon hon, tim du lieu trong phan lon hon
            lo = mid + 1;
         }else {
            // neu du lieu co gia tri nho hon, tim du lieu trong phan nho hon
            hi = mid - 1;
         }
      }
   }
   printf("\nTong so phep so sanh da thuc hien: %d", --comparisons);
   return index;
      }
```

BÀI TẬP

Bài 1: Xây dựng giải thuật tìm kiếm phần tử có giá trị nhỏ nhất trong dãy số: Dùng mã tự nhiên, mã giả và lưu đồ.

Bài 2: Nhập một dãy số nguyên gồm n phần tử. Sắp xếp lại dãy sao cho:

- Số nguyên dương đầu ở đầu dãy và theo thứ tự giảm.
- Số nguyên âm tăng ở cuối dãy và theo thứ tự tăng.
- Số 0 ở giữa.

Lưu ý: Không dùng đổi chỗ trực tiếp

Bài 3: Cho dãy số nguyên A như sau:

12 2 15 3 8 51 -6 0 4 15

1. Sắp xếp dãy trên tăng dần.
2. Tìm số lớn thứ 3 trong dãy.
3. Tìm số lượng phần tử lớn nhất trong dãy
4. Sắp xếp dãy trên theo thứ tự giá trị tuyệt đối tăng dần.
5. Sắp xếp dãy trên theo quy luật sau:

Các số dương (nếu có) ở đầu mảng và có thứ tự giảm dần.

Các số âm (nếu có) ở cuối mảng và có thứ tự tăng dần.

6. Sắp xếp dãy trên theo quy luật:

Các số chẵn (nếu có) ở đầu mảng và có thứ tự tăng dần.

Các số lẻ (nếu có) ở cuối mảng và có thứ tự giảm dần.

Bài 4: Viết chương trình minh họa các giải thuật tìm kiếm và sắp xếp trên mảng có kích thước n phần tử nguyên. Chương trình được mô tả với các yêu cầu như sau:

- Cài đặt các hàm tìm kiếm

o Tìm kiếm tuyến tính (tuần tự) cho mảng bất kỳ

o Tìm kiếm nhị phân cho mảng dữ liệu được sắp xếp tăng dần

- Cài đặt các hàm sắp xếp (tăng dần) theo các phương pháp:

Bubble sort, Insertion Sort, Selection Sort, Heap Sort, Quick Sort, Merge Sort để sắp xếp một dãy các số nguyên theo thứ tự tăng dần. So sánh các thuật toán về các mặt: thời gian chạy, số phép gán, số phép so sánh.

Chương 3

Kỹ thuật thiết kế thuật toán

Một trong những bước giải quyết bài toán đó là xây dựng thuật toán giải quyết bài toán đó. Vậy làm thế nào để thiết kế thuật toán giải quyết bài toán đã cho. Chương 3 với mục đích đưa ra vài kĩ thuật giúp cho việc thiết kế thuật toán.

Kĩ thuật thiết kế thuật toán (technique, strategy, paradigm) là cách tiếp cận để giải quyết các vấn đề về thuật toán.

Trong khi các kỹ thuật thiết kế thuật toán cung cấp một tập hợp các phương pháp mạnh mẽ để giải quyết các vấn đề về thuật toán nhưng việc thiết kế một số bài toán đặc biệt hiện nay vẫn đang còn là thách thức. Đôi khi các kỹ thuật cần được kết hợp với nhau và có những thuật toán khó xác định khi áp dụng các kỹ thuật thiết kế đã biết. Thậm chí khi một kỹ thuật cụ thể được áp dụng, để đưa ra một thuật toán thường yêu cầu sự khéo léo của người thiết

kế thuật toán.

Một số phương pháp thiết kế thuật toán cơ bản: phương pháp chia để trị (Divide – and - Conquer), phương pháp quy hoạch động (dynamic programming), phương pháp tham lam (greedy method), phương pháp quay lui (backtracking), phương pháp nhánh cận (branch – and – bound), tìm kiếm địa phương (local search). Phần dưới đây sẽ trình bày 4 kỹ thuật cơ bản sau:

3.1 Phương pháp chia để trị

3.1.1 Kỹ thuật chung

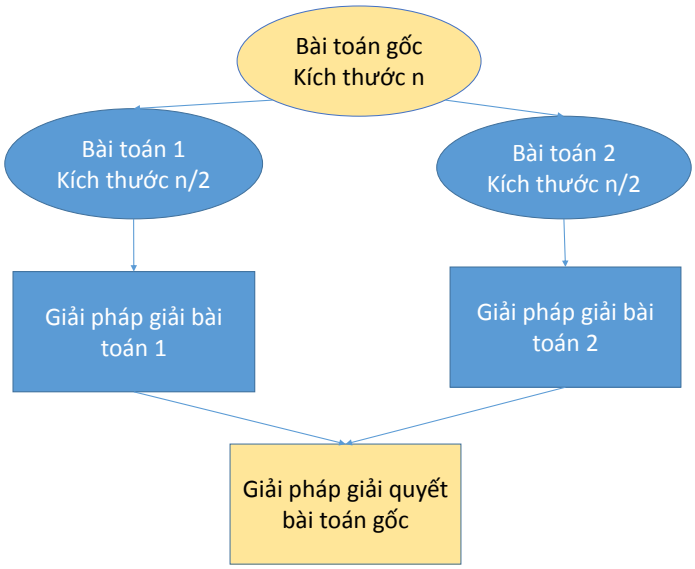
Chia để trị là một kĩ thuật mà bài toán được làm đơn giản hơn bằng cách "chia" thành các phần nhỏ hơn và sau đó giải từng phần đó.

Đây là kỹ thuật từ trên xuống (top – down), là kỹ thuật quan trọng nhất, được áp dụng rộng rãi nhất để thiết kế các giải thuật có hiệu quả (xem Hình 3.1).

Kỹ thuật chia để trị gồm 3 bước:

- Chia bài toán đã cho thành nhiều bài toán con
- Giải mỗi bài toán con bằng cách đệ quy
- Lời giải của bài toán ban đầu được trả về bằng cách kết hợp các lời giải của bài toán con.

Đối với một số bài toán, quá trình phân tích đã chứa đựng việc tổng hợp kết quả. Nếu ta giải xong các bài toán cơ sở thì bài toán ban đầu cũng đã được giải quyết. Ngược lại có những bài toán mà quá trình phân tích thì đơn giản nhưng việc tổng hợp kết quả lại rất phức tạp.



Hình 3.1: Kỹ thuật chia để trị

Kỹ thuật này sẽ cho ta một giải thuật đệ quy mà việc xác định độ phức tạp phải giải một phương trình đệ quy nào đó.

```
void DivideConquer(A, x)
{
if (A_du_nho) return Solve(A);
else
{
  for (i=1; i<= m; i++)
    return (DivideConquer(Ai, xi));
// ket hop cac nghiem xi (i = 1, 2, , m) cua bai toan con Ai
// de nhan duoc nghiem x cua bai toan A
}
}
```

3.1.2 Ví dụ áp dụng phương pháp chia để trị

1. *Giải thuật Mergesort, Quicksort và tìm kiếm nhị phân*:

   Thuật toán Mergesort và Quicksort, tìm kiếm nhị phân là cách thuật toán áp dụng ký thuật chia để trị chuẩn

   Với Mergesort, kỹ thuật "Divide and conquer" thể hiện ở quá

trình chia đôi một danh sách, quá trình này sẽ dẫn đến bài toán sắp xếp một danh sách có độ dài bằng 1 (bài toán cơ sở). Việc tổng hợp kết quả ở đây là "trộn" 2 danh sách đã sắp để được một danh sách có thứ tự.

Với Quicksort, kỹ thuật chia để trị thể hiện ở quá trình phân hoạch danh sách thành 2 danh sách con "bên trái" và "bên phải", sắp xếp "bên trái" và "bên phải" để được danh sách có thứ tự. Quá trình phân chia dẫn đến các bài toán sắp xếp một danh sách chỉ gồm một phần tử hoặc nhiều phần tử có khóa bằng nhau (bài toán cơ sở). Với Quicksort không phải tổng hợp kết quả vì việc đó đã thực hiện trong quá trình phân hoạch (các thuật toán này được trình bày ở mục **Chương 2**.

2. *Bài toán nhân các số nguyên lớn*:

   Input: Cho 2 số nguyên X, Y có n chữ số

   Output: X*Y.

   Giải thuật nhân 2 số thông thường cần n^n phép nhân và n phép cộng, thời gian $O(n^n)$.

   Sử dụng kỹ thuật chia để trị "Divide and conquer" như sau:

   - Chia mỗi số nguyên X và Y thành các số nguyên có $n/2$ chữ số

     $X = A.10^{n/2} + B$

     $Y = C.10^{n/2} + D$

     Trong đó A, B, C, D là các số nguyên có $n/2$ số. Ví dụ $X = 1234$ thì $A = 12$ và $B = 34$ vì $X = 12.10^2 + 34$). Vậy $X.Y = A.C.10^n + (AD + BC).10^{n/2} + BD(1)$

   - Với mỗi số có $n/2$ chữ số, tiếp tục phân tích theo cách trên, quá trình phân tích sẽ dẫn đến một bài toán cơ sở là nhân

các số nguyên chỉ gồm một chữ số mà ta dễ dàng thực hiện. Việc tổng hợp kết quả là việc thực hiện các phép toán theo công thức (1).

- Theo (1) phải thực hiện 4 phép nhân các số nguyên $n/2$ chữ số (AC, AD, BC, BD), tổng hợp kết quả bằng 3 phép cộng các số nguyên n chữ số và 2 phép nhân với 10^n và $10^{n/2}$. Phép cộng các số nguyên n chữ số chỉ cần $O(n)$, phép nhân với 10^n có thể thực hiện bằng cách thêm vào n chữ số 0. Thời gian là $O(n)$.

  Gọi $T(n)$ là thời gian nhân 2 số nguyên, mỗi số có n chữ số. Ta có công thức đệ quy:

  $$T(n) = \begin{cases} 1 & n = 1 \\ 4T(n/2) + Cn & n > 1 \end{cases}$$

  Giải $T(n)$ được $T(n) = O(n^2)$. Chưa cải tiến hơn so với giải thuật nhân 2 số thông thường.

  Do đó: Viết (1) thành dạng

  $XY = AC.10^n + [(A-B)(D-C) + AC + BD].10^{n/2} + BD$ (3)

  (3) chỉ cần 3 phép nhân các số nguyên $n/2$ chữ số AC, BD và $(A - B)(D - C)$, 6 phép cộng trừ và 2 phép nhân. Ta có phương trình đệ quy:

  $$T(n) = \begin{cases} 1 & n = 1 \\ 3T(n/2) + Cn & n > 1 \end{cases}$$

  Giải phương trình đệ quy được $T(n) = O(n^{log3}) = O(n^{1.59})$ Giải thuật này cải thiện hơn rất nhiều.

```
int  Mult(int x, int y, int n)
{
int m1,m2,m3,A,B,C,D;
int dau;

dau=sign(x)*sign(y);
```

```
x=ABS(x); y=ABS(y)
if (n==1) return(x*y*dau);
else
{
A=Left(x, n/ 2);  //A: so nguyen co n/2 chu so ben trai
B=Right(x, n/ 2);
C=Left(y, n/ 2);
D=Right(y, n / 2);
m1=Mult(A,C,n / 2);
m2=Mult(A-B,D-C,n / 2);
m3=Mult(B,D,n / 2);
return(dau*[(m1*10^n)+(m1+m2+m3)*10^(n/2)+m3]);
}
}
```

Với kỹ thuật "chia để trị" nói chung sẽ tốt hơn nếu ta chia bài toán cần giải thành các bài toán con có kích thước gần bằng nhau

Ví dụ: Mergesort phân chia bài toán thành 2 bài toán con có cùng kích thước $n/2$.Vậy thời gian thực hiện là $O(nlogn)$. Ngược lại trong trường hợp xấu nhất của $QuickSort$, khi kích thước 2 bài toán con lệch nhau quá lớn thì thời gian thực hiện có thể lên đến $O(n^2)$.

Nguyên tắc chung là tìm cách phân chia bài toán thành các bài toán con có kích thước xấp xỉ bằng nhau thì hiệu quả sẽ cao hơn. Cách này gọi là bài toán con cân bằng *(Balancing Subproblems)*.

3.1.3 Ứng dụng của kỹ thuật chia để trị

Kỹ thuật chia để trị có các vai trò sau:

- Giải các bài toán khó: Giải bài toán gốc ban đầu bằng cách chia bài toán lớn thành các bài toán nhỏ, sau đó kết hợp các kết quả của bài toán con đưa ra lời giải cho bài toán gốc.

- Tính toán song song: Kỹ thuật này phù hợp cho các máy tính đa xử lý (multi processor machines), đặc biệt với các hệ thống có bộ nhớ được chia sẻ (nơi mà sự trao đổi giữa dữ liệu và các bộ xử lý không được lên kế hoạch trước), bởi vì các bài toán con được tính toán trên các bộ xử lý khác nhau.
- Truy cập bộ nhớ: Làm cho việc sử dụng bộ nhớ tạm thời hiệu quả. Khi bài toán con đủ nhỏ thì nó được tính trong bộ nhớ tạm thời (Memory caches) mà không cần truy cập vào bộ nhớ chính.
- Kiểm soát vòng lặp: Trong nhiều bài toán phương pháp này cho kết quả chính xác hơn so với phương pháp lặp tương ứng.

3.2 Phương pháp quy hoạch động

3.2.1 Phương pháp chung

Kỹ thuật quy hoạch động được nhà toán học người Mỹ- Richard Bellman đưa ra vào những năm 1950. "Programming" được dung thay cho "planning" tức là lập kế hoạch và nó không liên quan tới lập trình máy tính. Kỹ thuật quy hoạch động là kỹ thuật thiết kế thuật toán rất được chú ý trong lịch sử. Nó là một phương pháp tối ưu quá trình quyết định nhiều cấp. Sau khi được chứng minh thì nó là một công cụ quan trọng trong toán học ứng dụng. Sau đó kỹ thuật quy hoạch động được phát triển trong lĩnh vực khoa học máy tính như là một kỹ thuật thiết kế thuật toán tổng quát.

Ta biết kĩ thuật chia để trị thường dẫn tới một giải thuật đệ quy. Trong các giải thuật đó, có thể có một số giải thuật thời gian mũ. Tuy nhiên, thường chỉ có một số đa thức các bài toán con, điều đó có nghĩa là ta phải giải một số bài toán con nào đó trong nhiều lần. Thuật toán nhận được sẽ kém hiệu quả. Thay vì tiếp cận bài toán

theo hướng top-down (như phương pháp chia để trị) phương pháp này tiếp cận theo hướng đi từ dưới lên (bottom – up) để tạo ra một bảng lưu tất cả các kết quả của các bài toán con và khi cần chỉ cần tham khảo tới kết quả đó được lưu trong bảng mà không phải giải lại bài toán đó. Lấp đầy bảng kết quả các bài toán con theo một quy luật nào đó để nhận được kết quả của bài toán ban đầu (cũng đã được lưu trong một ô nào đó của bảng) được gọi là quy hoạch động. Qua đó phương pháp này có thể giải quyết được vấn đề dư thừa tức là giải quyết bài toán con nhiều lần.

Đặc điểm của kỹ thuật quy hoạch động:

- Tương tự với kỹ thuật chia để trị ở chỗ cũng đi giải quyết các bài toán con, nhưng khác ở chỗ các bài toán con là không độc lập với nhau.
- Kết quả của bài toán con được tính một lần và lưu vào bảng giá trị để dùng cho các công việc sau.
- Phương pháp quy hoạch động giải quyết bài toán theo hướng bottom-up còn phương pháp chia để trị thì đi theo hướng top-down
- Kỹ thuật quy hoạch động dùng để giải quyết các bài toán tối ưu

Các bước:

- Chia các bài toán con: Ở bước này chúng ta phải định nghĩa cấu trúc của bài toán con
  - Lời giải của bài toán con nhỏ nhất giải được một cách trực tiếp
  - Các bài toán con có lời giải dễ kết hợp

- Lưu trữ lại các lời giải của các bài toán con, tránh giải quyết cùng một bài toán con.
- Kết hợp:
  - Theo chiến lược bottom up
  - Đưa ra lời giải của bài toán ban đầu từ lời giải của các bài toán con

Hạn chế của kỹ thuật quy hoạch động: Phương pháp quy hoạch động không hiệu quả trong các tình huống sau:

- Sự kết hợp lời giải của các bài toán con chưa chắc đã cho ta lời giải của các bài toán lớn hơn.
- Số lượng các bài toán con cần giải quyết và lưu trữ kết quả có thể rất lớn, không thể chấp nhận được.

Để kỹ thuật này đạt hiệu quả thì:

- Số lượng các bài toán con phải được giới hạn bởi một đa thức kích thước đầu vào.
- Lời giải của các bài toán con cũng là những lời giải tối ưu.

3.2.2 Ví dụ áp dụng

1. *Bài toán tính số tổ hợp*

   Công thức tính số tổ hợp chập k của n:

   $$C_n^k = \begin{cases} 1 & k = 0 \ or \ k = 1 \\ C_{n-1}^{k-1} + C_{n-1}^k & n > k > 0 \end{cases}$$

   Giải thuật đệ quy:

```
int Tohop(int n, int k)
{
   if (k == 0 || k == n) return 1;
   return (ToHop(n - 1, k - 1) + ToHop(n - 1, k));
}
```

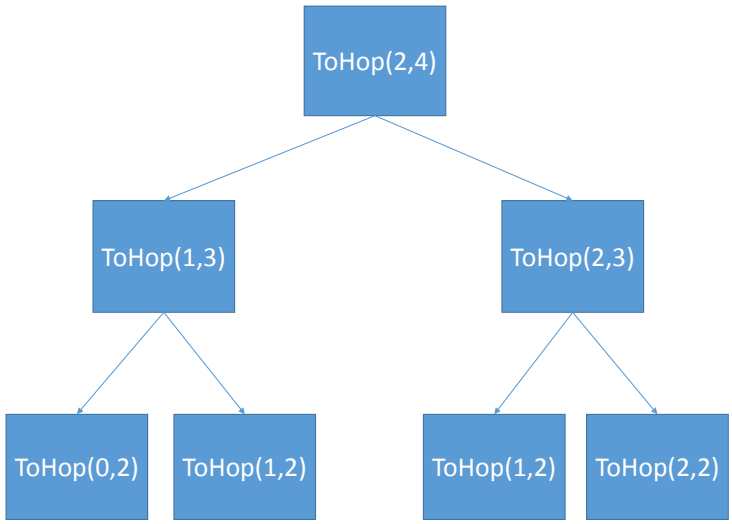
Độ phức tạp của thuật toán:

Gọi $T(n)$ là thời gian tính C_n^k. Khi đó ta có phương trình đệ quy:

$$T(n) = \begin{cases} C_1 & k = 0 \ or \ k = n \\ 2T(n-1) + C_2 & n > k > 0 \end{cases}$$

Giải phương trình đệ quy ta được: $T(n) = O(2^n)$

Như vậy giải thuật đệ quy trên có thời gian thực hiện là hàm mũ trong khi chỉ có một đa thức các bài toán con, chứng tỏ có những bài toán con được giải nhiều lần. Ví dụ (xem Hình 3.2):



Hình 3.2: Tổ hợp chập 2 của 4

Như vậy để tính $Tohop(2,4)$ phải tính $Tohop(1,2)$ 2 lần. Quy hoạch động sẽ khắc phục tình trạng trên bằng cách xây dựng

một bảng gồm $n+1$ dòng (từ $0..n$) và $n+1$ cột từ $(0..n)$ và điền giá trị ô (i, j) theo qui tắc tam giác Pascal (xem hình 3.3)

i \ j	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

Hình 3.3: Tam giác pascal

$O(0,0) = 1; O(i,0) = 1;$

$O(i,i) = 1$ với $0 < i < n$

$O(i,j) = O(i-1,j-1) + O(i-1,j)$ với $0 < j < i < n$

Giá trị ở ô $O(n,k)$ chính là $Tohop(n,k)$

```
int  Tohop(int n, int k)
{
    int C[100];
    int i,j;
    C[0,0]=1;
    for(i=1;i<=n;i++)
      {
        C[i,0]=1; C[i;i]=1;
        for(j=1;j<= i-1;j++)
            C[i,j]=C[i-1,j-1]+C[i-1,j];
     }
```

```
    return c[n,k];
}
```

Độ phức tạp: $T(n) = O(n^2)$

2. *Xâu con chung*

Bài toán: Cho hai xâu ký tự x và y, hãy tìm xâu ký tự c là xâu con chung của x và y và c có độ dài lớn nhất có thể được.

Ví dụ: $x =' ab132sc'$ và $y =' 1abdc'$ thì $c =' abc'$

Nhận xét: Nếu x và y có độ dài đủ nhỏ thì có thể giải bài toán bằng cách duyệt mọi xâu con chung và lưu lại xâu có độ dài lớn nhất. Tuy nhiên cách làm này không thể đáp ứng về mặt thời gian khi x và y có độ dài lớn. Vì vậy ta sẽ dùng phương pháp quy hoạch động để giải quyết bài toán trên như sau:

Gọi m và n lần lượt là độ dài của các xâu x và y.

- Nếu một trong hai số $m = 0$ hoặc $n = 0$ thì c là xâu rỗng.

- Xét các đoạn đầu của 2 xâu x và y có độ dài i và j tương ứng $(x_1, x_2, .., x_i)$ và $(y_1, y_2, .., y_j)$. Gọi $L(i, j)$ là độ dài lớn nhất xâu con chung của hai xâu này. Khi đó $L(m, n)$ sẽ là độ dài lớn nhất của hai xâu x và y.

Ta có cách tính $L(i, j)$ thông qua $L(s, t)$ với theo quy tắc sau:

+ Nếu $i = 0$ hoặc $j = 0$ thì $L(i, j) = 0$.

+ Nếu $i > 0$ và $j > 0$ và x_i != y_j thì $L(i, j) = MaxL(i, j-1), L(i-1, j)$.

+ Nếu $i > 0$ và $j > 0$ và $x_i = y_j$ thì $L(i, j) = 1+L(i-1, j-1)$.

- Lưu các giá trị $L(i, j)$ vào mảng $L[m][n]$. Từ quy tắc trên ta thấy nếu biết $L[i][j-1], L[i-1][j], L[i-1][j-1]$ ta tính ngay được $L[i][j]$ Có thể tính được các phần tử của mảng $L[m][n]$ từ góc trên trái lần lượt theo các đường chéo song song.

```
void Xau_con_chung()
{  Char x[100], y[100], c[100];
   int i, j;
   int L[250][250];
   for(i= 0;i<strlen(x);i++) L[0][i]=0;
        for(j= 0;j< strlen (y);j++) L[j][0]=0;
          for(i=0;i< strlen (x);i++)
            for (j=0;j< strlen(y);j++)
               if (x[i] ==y[j]) L[i][j]= L[i-1][j-1] + 1;
               else
                  if (L[i-1][j] > L[i][j-1]) L[i][j]= L[i-1][j];
                  else L[i][j]= L[i][j-1];
}
```

Truy xuất kết quả

Từ mảng L đã được làm đầy, ta xây dựng xâu con chung dài nhất có độ dài là $k = L[m][n]$. Ta xác định các thành phần của $c = (c_1, c_2, , c_k)$ lần lượt từ bên phải. Trong bảng L xuất phát từ ô $L[m][n]$, đặt $c =$ "", giả sử đang ở ô $L[i][j]$:

- Nếu $x_i = y_j$ thì gán $c = x_i + c$, và đi lên ô $L[i-1][j-1]$.

- Nếu $x_i <> y_j$ thì :

+ hoặc $L[i][j] = L[i][j-1]$ thì đi tới ô $L[i][j-1]$

+ hoặc $L[i][j] = L[i-1][j]$ thì đi tới ô $L[i-1][j]$.

Quá trình cứ tiếp tục khi ta xác định được tất cả các thành phần của c (nghĩa là khi gặp ô $L[i][j]$ nào đó có giá trị 0).

```
char* Truy_vet()
{
 int i,j;
 char* c="";
 i=strlen(x);
 j=strlen(y);
 while (L[i][j]!=0)
    if  (x[i]==y[j])
       {
           c=x[i]+c;
           i--;
           j--;
        }
    else
       if  (L[i][j] ==L[i-1][j]) j--;
```

```
      else i--;
return(c);
}
```

Ví dụ Xây dựng bảng $L(6,7)$ (xem Hình 3.4), với:

$x = abd4eb$

$y = 1ab4cde$

X / Y	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	0	1	2	2	2	2	2
3	0	1	2	2	3	3	3
4	0	1	2	2	3	3	3
5	0	1	2	3	3	3	3
6	0	1	2	3	3	4	4

Hình 3.4: L[6,7]

$C =$ ‘$ab4e$’

Chú ý:

- Trên bảng lưu, khi truy vết có thể có nhiều xâu con chung có độ dài lớn nhất.

- Sử dụng thuật toán trên kết hợp với cấu trúc dữ liệu kiểu con trỏ ta có thể mở rộng bài toán cho trường hợp tìm dãy con chung của hai dãy số nguyên có độ dài lên tới hàng ngàn đơn vị.

3.3 Phương pháp tham lam

3.3.1 Phương pháp chung

Khác với quy hoạch động, thường giải quyết các bài toán con từ dưới lên, một chiến lược tham lam thường tiến triển theo cách từ trên xuống, phương án X được xây dựng bằng cách lựa chọn từng thành phần x_i của X cho đến khi hoàn chỉnh (đủ n thành phần). Với mỗi x_i, ta sẽ chọn x_i tối ưu. Với cách này thì có thể ở bước cuối cùng ta không còn gì để chọn mà phải chấp nhận một giá trị cuối cùng còn lại.

Lược đồ của phương pháp tham lam:

```
int  Greedy_Method(A,X)
//Xay dung cac phuong an X tu tap A
{
X=rong ;
while (A !=rong )
    {
    x= Select(A); //Ham chon x tot nhat tu A
    A=A-{x};
    if (X+{x} chap nhan duoc)
        X=X+{x};
    }
return (X); //phuong an toi uu
}
```

Ta có thể dễ dàng thấy tại sao các thuật toán như thế được gọi là "tham lam". Tại mỗi bước, nó chọn "miếng ngon nhất" (được xác định bởi hàm chọn), nếu thấy có thể chấp nhận được (có thể đưa vào nghiệm) thì kết nạp vào tập kết quả, nếu không nó sẽ bỏ qua, sau này không bao giờ xem xét lại.

Cần lưu ý rằng, thuật toán tham lam trong một số bài toán, nếu xây dựng được hàm chọn thích hợp có thể cho nghiệm tối ưu hoặc xấp xỉ tối ưu.

3.3.2 Ví dụ áp dụng

1. *Bài toán đổi tiền*

   Input :

   - Có n loại tiền, mỗi loại tiền có giá trị tương ứng là $d_1, d_2, .., d_n$.

   - Có số tiền M đồng.

   Output : Đổi M đồng ra tiền lẻ sao cho số loại tiền đổi là ít nhất.

   Phân tích : Cần phải tìm 1 nghiệm $X = (x_1, x_2, ..x_m)$ với x_i là số loại tiền thứ i có giá trị d_i sao cho $M = \sum_{i=1}^{n} x_i d_i$ Tìm cách đổi sao cho tổng loại tiền cần đổi là ít nhất. Vậy ta sẽ bắt đầu đổi từ đồng có giá trị lớn nhất và cứ giảm dần cho đến khi số tiền M đã được đổi hết thì thông báo là tìm được nghiệm, ngược lại thì thông báo không đổi được.

   Áp dụng tư tưởng của thuật toán tham lam ta đi sắp xếp các tờ tiền có mệnh giá từ cao đến thấp. Sau đó lựa chọn những tờ tiền có mệnh giá cao trước rồi tới các tờ có mệnh giá nhỏ hơn.

   Cụ thể như sau:

```
void Doitien_Thamlam(int M)
{
D={d1,d2,..,dn} // mang luu gia tri cua tung loai tien
int  X, sum, i; //X la nghiem cua bai toan

Sap_xep(D);// Sap xep D theo thu tu giam dan
X=rong ; sum=0 ; i=1;
while (sum!=M) && (i <= n)) //Trong khi chua doi het tien
{
    xi= (M - sum) / di ; //so to tien
    if (sum+xi*di <= M)
```

```
    {
       X=X+{xi};
       sum=sum+xi*di; //so tien da doi duoc
    }
    i=i+1;
}
if (sum==M) Return(X) //tra lai nghiem cua bai toan
else <Thong bao khong doi duoc>;
}
```

Nhận xét:

- Tính chất tham lam thể hiện ở chỗ, tại mỗi bước luôn chọn "miếng ăn ngon nhất" mà không để ý hậu quả sau này. Cho nên, với một số trường hợp thuật toán cho ta nghiệm tối ưu, nhưng trong nhiều trường hợp nghiệm trả về chỉ là xấp xỉ tối ưu. Ví dụ: Nếu $M = 13$ và các loại tiền có mệnh giá là: $d_1 = 3, d_2 = 4, d_3 = 6$. Cần tìm cách đổi 13 đồng sao cho số tiền đổi là ít nhất. Với thuật toán trên ta cần 2 tờ 6 đồng dư 1 đồng. Như vậy số tiền này sẽ không đổi được, nhưng trong thực tế, ta có thể đổi được dễ dàng với 1 tờ 6 đồng, 1 tờ 4 đồng và 1 tờ 3 đồng.

2. *Bài toán xếp ba lô (xếp balô giá trị nguyên)*

Input:

- Một ba lô có thể tích B, n đồ vật có thể tích: $a_1, a_2, ..., a_n$
- Giá trị tương ứng của các đồ vật là: $p_1, p_2, \ldots, p_n$
- x_i là số lượng loại đồ vật i.

Output: Tìm nhóm đồ vật thoả mãn $\sum_{i=1}^{n} a_i x_i <= B$ và $\sum_{i=1}^{n} p_i x_i$ đạt max ?

Theo yêu cầu của bài toán ta cần những đồ vật có giá trị cao mà thể tích nhỏ để có thể mang được nhiều "đồ quý". Vì vậy ta quan tâm tới đơn giá của từng loại đồ vật tức là tỷ lệ giá

trị/ thể tích, đơn giá càng cao thì đồ càng quý. Khi đó phương pháp tham lam được áp dụng giải bài toán trên trong thời gian đa thức như sau:

- Tính đơn giá cho các loại đồ vật.

- Xét các đồ vật theo thứ tự đơn giá từ lớn đến nhỏ.

- Với mỗi đồ vật được xét sẽ lấy một số lượng tối đa mà thể tích còn lại của balô cho phép.

- Xác định thể tích còn lại của balô và quay lại bước 3 cho đến khi không còn chọn được đồ vật nào nữa.

- Độ phức tạp tính toán: $O(n^2)$.

Ví dụ: Cho balô có thể tích là 37 và 4 loại đồ vật có trọng lượng và giá trị tương ứng như sau (xem bảng dưới đây):

Bảng 3.1: Sắp xếp theo chữ số hàng trăm

Đồ vật	**Thể tích** a_i	**Giá trị** p_i
A	15	30
B	10	25
C	2	2
D	4	6

Bảng 3.2: Sắp xếp theo chữ số hàng trăm

Đồ vật	**Thể tích** a_i	**Giá trị** p_i	p_i/a_i
A	15	30	2
B	10	25	2.5
C	2	2	1
D	4	6	1.5

- Chọn B: 3 cái. Thể tích còn lại $37 - 3 * 10 = 7$

- Chọn A: $a_i = 15 > 7$ không được.

- Chọn D: 1 cái. Thể tích còn lại: $7 - 1 * 4 = 3$

- Chọn C: 1 cái. Thể tích còn lại: $3 - 1 * 2 = 1$

$=>$ Tổng thể tích: $3 * 10 + 1 * 4 + 1 * 2 = 36$

$=>$ Tổng giá trị: $3 * 25 + 1 * 6 + 1 * 2 = 83$.

3.4 Thuật toán vét cạn quay lui

Thuật toán quay lui dùng để giải các bài toán liệt kê thỏa mãn tính chất nào đó. Ví dụ về các bài toán liệt kê cấu hình như bài toán liệt kê dãy nhị phân có N phần tử, bài toán liệt kê tập con có k phần tử... Để giải các bài toán liệt kê còn có phương pháp sinh (Generating method) (phương pháp này không trình bày trong giáo trình này).

3.4.1 Phương pháp chung

Về bản chất, tư tưởng của phương pháp là thử từng khả năng cho đến khi tìm thấy lời giải đúng. Đó là một quá trình tìm kiếm theo độ sâu trong một tập hợp các lời giải. Trong quá trình tìm kiếm, nếu ta gặp một hướng lựa chọn không thỏa mãn, ta quay lui về điểm lựa chọn nơi có các hướng khác và thử hướng lựa chọn tiếp theo. Khi đã thử hết các lựa chọn xuất phát từ điểm lựa chọn đó, ta quay lại điểm lựa chọn trước đó và thử hướng lựa chọn tiếp theo tại đó. Quá trình tìm kiếm kết thúc khi không còn điểm lựa chọn nào nữa

Giả sử dãy cần liệt kê có dạng $x[1..n]$, khi đó thuật toán quay lui thực hiện qua các bước:

1) Xét tất cả các giá trị có thể có của $x[1]$, thử cho $x[1]$ lần lượt nhận từng giá trị này. Với mỗi giá trị gán cho $x[1]$ thực hiện bước 2):

2) Xét tất cả các giá trị $x[2]$ có thể nhận, lại thử lần lượt các giá trị $x[2]$. Với mỗi giá trị thử gán cho $x[2]$ lại xét tiếp các khả năng chọ $x[3]$.... Cứ tiếp tục như vậy cho đến bước tiếp theo:

....

n) Xét tất cả các giá trị $x[n]$ có thể nhận, thử $x[n]$ nhận lần lượt các giá trị có thể, thông báo dãy tìm được (x[1], x[2], ..., x[n]).

Thuật toán được mô tả như sau:

```
void Try(int i)
{
    for ( Moi gia tri V co the gan cho x[i])
        {
            Thu cho x[i]=V;
            if (x[i] la phan tu cuoi trong day)
                thong_bao_day_tim_duoc;
            else
            {
                Ghi nhan viec cho x[i] nhan gia tri V;
                Try(i+1);// goi de quy de goi tiep x[i+1]
                Neu can, bo ghi nhan viec thu x[i]=V de thu gia tri khac;
            }
        }
}
```

Thuật toán quay lui sẽ được bắt đầu bằng lời gọi Try(1)

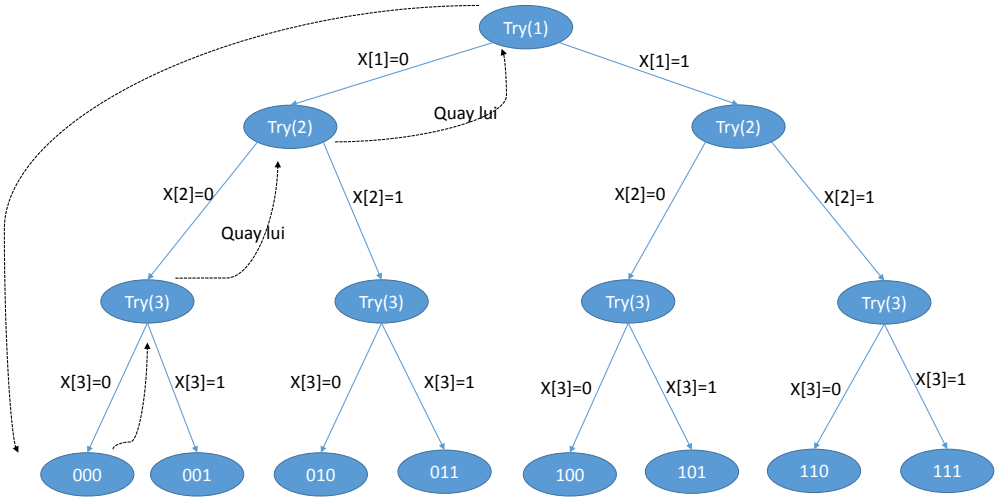
3.4.2 Ví dụ

Sau đây là ví dụ minh họa thuật toán vét cạn quay lui với bài toán liệt kê các dãy nhị phân có độ dài 3 được mô tả qua hình 3.5.

Sau đây là chương trình minh họa thuật toán quay lui của bài toán trên:

```
void Try(int i)// thu cac cach chon x[i]
{
   int j;
   for (j=0;j<=1;j++)
    {
        x[i]=j;
        if (i==n) print_Result;
        else Try(i+1);
    }
}
```

Các bài toán khác sử dụng phương pháp vét cạn quay lui như bài toán N quân hậu, bài toán liệt kê dãy con, bài toán liệt kê hoán



Hình 3.5: Thuật toán vét cạn quay lui với bài toán liệt kê dãy nhị phân có 3 phần tử

vị, ...

3.5 Phương pháp nhánh cận

3.5.1 Phương pháp chung

Với các bài toán tìm phương án tối ưu, nếu ta xét hết tất cả các phương án thì mất nhiều thời gian. Phương pháp nhánh cận là một dạng cải tiến của thuật toán quay lui được áp dụng để tìm nghiệm của bài toán tối ưu.

Với phương pháp nhánh cận ta sẽ đi xây dựng cây tìm kiếm phương án tối ưu, nhưng không xây dựng toàn bộ cây mà sử dụng giá trị cận để hạn chế bớt các nhánh.

- Cây tìm kiếm phương án có nút gốc biểu diễn cho tập tất cả các phương án có thể có, mỗi nút lá biểu diễn một phương án nào đó. Nút n có các nút con tương ứng với các khả năng có thể lựa chọn tập phương án xuất phát từ n. Kỹ thuật này gọi là phân nhánh.

- Với mỗi nút trên cây ta xác định được một giá trị cận (là giá trị gần với giá của các phương án).

+ Với bài toán tìm Min ta xác định cận dưới <= giá của phương án.

+ Với bài toán tìm Max ta xác định cận trên >= giá của phương án.

Như vậy có thể nói: Phương pháp Nhánh cận là một dạng cải tiến của phương pháp "vét cạn", ta có lược đồ của phương pháp nhánh cận như sau:

```
void xuly(int i)
{
int j;
for (i Tap gia tri de cu )
    if (<chap nhan j> )
    {
      xi=j;
      if (i==n) <ghi nhan>
      else
      if (g(x1,x2,..,xn) can duoi tam thoi) xuly (i+1)
    }
}
```

3.5.2 Ví dụ áp dụng

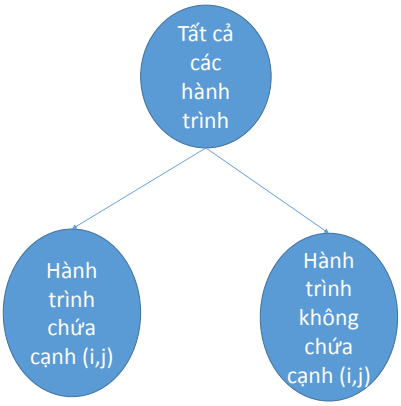
1. *Bài toán người giao hàng*

   Phân nhánh:

   Trong bài toán này khi tìm kiếm lời giải, chúng ta sẽ phân tập các hành trình ra thành hai tập con: một tập gồm những hành trình chứa một cạnh (i, j) nào đó và tập kia gồm những hành trình không chứa cạnh này (xem Hình 3.6)

   Khi đó ta có thể xây dựng cây tìm kiếm phương án là cây nhị phân, trong đó:

   - Nút gốc là nút biểu diễn cho cấu hình bao gồm tất cả các hành trình có thể có.



Hình 3.6: Phân nhánh

- Mỗi nút có hai con, con trái biểu diễn cấu hình bao gồm tất cả các hành trình chứa một cạnh nào đó, nút con phải biểu diễn cấu hình bao gồm tất cả các hành trình không chứa cạnh đó (các cạnh xét phân nhánh được xếp theo một thứ tự nào đó, chẳng hạn thứ tự từ điển).

- Mỗi nút sẽ kế thừa các thuộc tính của tổ tiên của nó và có thêm một thuộc tính mới (chứa hay không chứa một cạnh nào đó).

- Nút lá biểu diễn cho 1 cấu hình chỉ bao gồm 1 phương án.

- Trong quá trình phân nhánh phải đảm bảo các điều kiện ràng buộc của bài toán. vì vậy tại mỗi nút ta cần quy định trên nguyên tắc là mọi đỉnh trên chu trình đều có cấp 2 và không tạo ra một chu trình thiếu.

Tính cận dưới: Đây là bài toán tìm Min nên ta sử dụng cận dưới. Cận dưới tai mỗi nút là số nhỏ hơn hoặc bằng giá của tất cả các phương án biểu diễn bởi nút đó. Giá của một phương án là tổng độ dài 1 chu trình.

- Tính cận dưới cho nút gốc: mỗi đỉnh chọn hai cạnh có độ dài

nhỏ nhất. Cận dưới của nút gốc bằng tổng độ dài tất cả các cạnh được chọn chia cho 2.

- Các nút khác: Lựa chọn 2 cạnh có độ dài nhỏ nhất thỏa mãn điều kiện ràng buộc (chứa cạnh này, không chứa cạnh kia).

Kỹ thuật nhánh cận: Kết hợp hai kỹ thuật trên để xây dựng cây tìm kiếm phương án theo quy tắc sau:

- Xây dựng nút gốc, bao gồm tất cả các phương án, tính cận dưới cho nút gốc.

- Phân nhánh cho mỗi nút, tính cận dưới cho cả 2 con.

- Nếu cận dưới của một nút con lớn hơn hoặc bằng giá nhỏ nhất tạm thời của một phương án đã được tìm thấy thì không cần xây dựng các nhánh con cho nút này nữa (cắt tỉa).

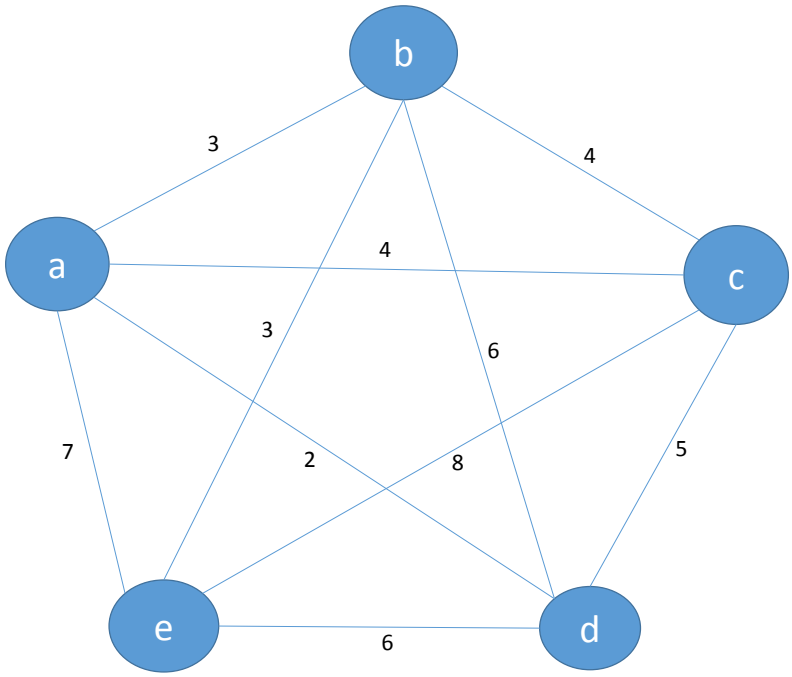
- Nếu cả hai con đều có cận dưới nhỏ hơn giá nhỏ nhất tạm thời của một phương án đã được tìm thấy thì nút nào có cận dưới nhỏ hơn sẽ được ưu tiên phân nhánh trước.

- Mỗi lần quay lui để xét nút con chưa được xét của một nút ta phải xem xét lại nút con đó để có thể cắt tỉa các nhánh của nó hay không, vì có thể một phương án có giá trị nhỏ nhất tạm thời chưa được tìm thấy.

- Sau khi tất cả các con đã được phân nhánh hoặc bị cắt tỉa thì phương án có giá nhỏ nhất trong các phương án tìm dược là phương án tối ưu.

Trong quá trình xây dựng cây có thể ta đã xây dựng được một số nút lá (một phương án). Giá nhỏ nhất trong số các giá của các phương án này được gọi là giá nhỏ nhất tạm thời.

Ví dụ : Xét bài toán người giao hàng (TSP) có 5 đỉnh (xem Hình 3.7)



Hình 3.7: Bài toán TSP có 5 đỉnh

Các cạnh theo thứ tự từ điển: $ab, ac, ad, ae, bc, bd, be, cd, ce, dc$.

- Phân nhánh: Xuất phát từ nút gốc chứa tập tất cả các hành trình, nút con trái là hành trình chứa cạnh ab, nút con phải là hành trình không chứa cạnh ab ...

- Cận dưới: Được tính như sau

Tính CD cho nút gốc:

+ đỉnh a chọn ab, ad.

+ đỉnh b chọn ba, be.

+ đỉnh c chọn ca, cb.

+ đỉnh d chọn da, dc.

+ đỉnh e chọn eb, ed.

Như vậy: Tổng là 35, cận dưới của nút gốc A là $35/2 = 17,5$.

Tính cận dưới cho nút D: Điều kiện ràng buộc chứa cả ab, ac không chứa ad, ae.

+ đỉnh a chọn ab, ac.

+ đỉnh b chọn ba, be.

+ đỉnh c chọn ca, cb.

+ đỉnh d chọn de, dc.

+ đỉnh e chọn eb, ed.

Như vậy: Tổng là 41, cận dưới của D là $20,5$.

Tương tự cho các nút còn lại ...

Sau khi phân nhánh và cắt tỉa ta tìm được hành trình tối ưu là: a, c, b, e, d, a với chi phí nhỏ nhất 19 (Phương pháp tham lam là 21).

Phương pháp cài đặt:

- Xây dựng thủ tục rút gọn ma trận chi phí để tính cận dưới.

- Xây dựng thủ tục chọn cạnh phân nhánh.

- Xây dựng thủ tục ngăn cấm tạo thành hành trình con.

- Xây dựng thủ tục đệ quy TSP- nhánh cận tìm hành trình tối ưu.

2. *Bài toán xếp ba lô*

Ta xét bài toán xếp balô giá trị nguyên đã nêu ở phần trước và phương pháp nhánh cận giải bài toán này: Tìm x thỏa mãn:

$\sum_{i=1}^{n}(x_i w_i) \leq B$

$\sum_{i=1}^{n}(x_i p_i) \longrightarrow Max$

Phương pháp Nhánh cận giải bài toán này như sau:

Ta thấy đây là một bài toán tìm Max. Danh sách các đồ vật được sắp xếp theo thứ tự giảm của đơn giá để xét phân nhánh.

1. Nút gốc biểu diễn trạng thái ban đầu của balô (chưa chọn một vật nào). Tổng giá trị $TGT = 0$; cận trên của nút gốc: $CT = B^*$(đơn giá Max).

2. Nút gốc có các nút con tương ứng với các khả năng chọn đồ vật theo đơn giá lớn nhất. Với mỗi nút con ta tính lại các thông số: $TGT = TGT(c)+$(số đồ vật được chọn)$*$(giá trị mỗi vật)

$B = B(c)-$ (số đồ vật được chọn)$*$(trọng lượng mỗi vật)

$CT = TGT + B$(mới)$*$(đơn giá vật kế tiếp)

3. Trong các nút con, ta sẽ ưu tiên phân nhánh cho nút con nào có cận trên lớn hơn trước. Các con của nút này tương ứng với các khả năng chọn đồ vật có đơn giá lớn tiếp theo. Với mỗi nút phải xác định lại các thông số TGT, B, CT.

4. Lặp lại bước 3 với chú ý: với những nút có cận trên nhỏ hơn hoặc bằng giá lớn nhất tạm thời của một phương án đã tìm thấy thì không cần phân nhánh cho nút đó nữa (cắt tỉa).

5. Nếu tất cả các nút đều đã được phân nhánh hoặc bị cắt bỏ thì phương án có giá trị lớn nhất là phương án cần tìm.

Ví dụ: Cho balô có thể tích $B = 37$ và 4 loại đồ vật có trọng lượng và giá trị tương ứng trong bảng dưới đây:

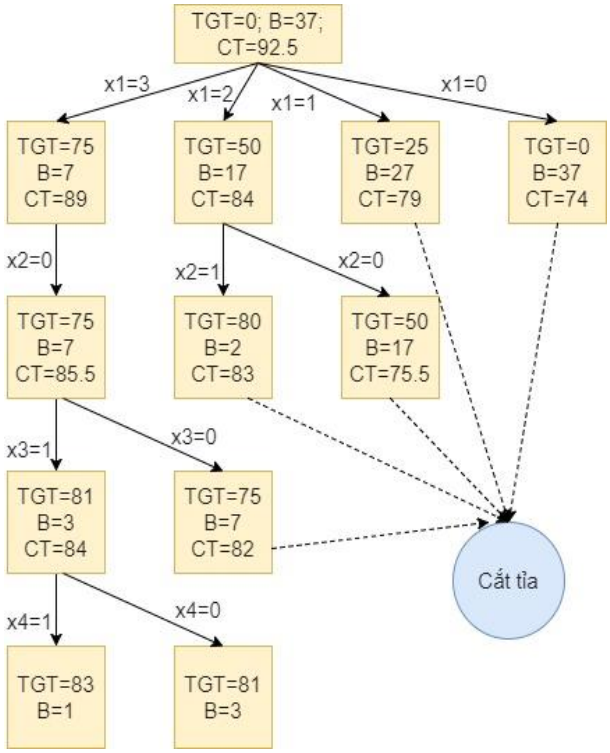
Bảng 3.3: Sắp xếp theo chữ số hàng trăm

Đồ vật	**Thể tích** a_i	**Giá trị** p_i
A	15	30
B	10	25
C	2	2
D	4	6

Hình 3.8 mô tả bài toán ba lô với kỹ thuật nhánh cận.

Bảng 3.4: Sắp xếp theo chữ số hàng trăm

Đồ vật	**Thể tích** a_i	**Giá trị** p_i	p_i/a_i
A	15	30	2
B	10	25	2.5
C	2	2	1
D	4	6	1.5



Hình 3.8: Bài toán balo với kỹ thuật nhánh cận

Kết luận: Một số kỹ thuật thiết kế thuật toán trình bày ở trên là một số kĩ thuật hay được dùng trong khi thiết kế thuật toán. Với mỗi bài toán chúng ta nên đọc kĩ bài toán, sau đó phân loại bài toán. Sau đây là một số dạng bài toán chính: Duyệt toàn bộ, chia để trị, tham lam, quy hoạch động, đồ thị, toán

học, xử lý xâu, hình học tính toán.... Tất cả những kỹ thuật này đều dùng trong giải các bài toán. Khi thiết kết thuật toán điều quan trọng là phải đánh giá được chi phí của thuật toán về thời gian và bộ nhớ, đồng thời kiểm tra chi phí đó có thỏa mãn yêu cầu cảu bài toán hay không. Nếu có nhiều cách để giải bài toán thì hãy chọn thuật toán đối với mình là cài đặt đơn giản nhất mà vẫn thỏa mãn yêu cầu, đúng và đủ nhanh trong giới hạn thời gian của bài toán. Nếu thời gian không hạn chế thì hãy tìm ra cách giải quyết bài toán với thuật toán tốt nhất.

3.6 BÀI TẬP

Bài 1: Tính x^n, trong đó x, n là số nguyên

Bài 2 (bài toán D. Pie): Vào ngày sinh nhật, có N chiếc bánh sinh nhật với các mùi vị và kích thước khác nhau. Có F người bạn tới dự sinh nhật và mỗi người sẽ ăn một mẩu bánh. Mọi người rất lo lắng nếu ai nó lấy miếng bánh to hơn và họ bắt đầu phàn nàn (mỗi người chỉ được một miếng bánh có kích thước bằng nhau, không quan tâm tới hình dạng của miếng bánh). Hỏi kích thước lớn nhất của miếng bánh mà mọi người nhận được là gì? (Chiều cao của chiếc bánh bằng nhau và bằng 1, bán kính của chiếc bánh là khác nhau)

Gợi ý: Sử dụng phương pháp tìm kiếm nhị phân.

Bài 3: Tìm dãy con có tổng lớn nhất. Ví dụ cho dãy: $-2, 11, -4, 13, -5, 2$. Dãy có tổng lớn nhất là $11, -4, 13$ có tổng là 20.

Bài 4: Một nước có n kho vàng được đặt trên một đường thẳng và được đánh số $1, 2, \ldots, n$. Mỗi kho thứ i có lượng vàng là a_i (a_i là số nguyên không âm) và được đặt ở tọa độ là i. Đất nước này

mở cuộc thi cho những thợ săn mà tìm được tập con các kho hang có tổng lượng vàng lớn nhất với điều kiện là khoảng cách giữa hai kho vàng nhỏ hơn hoặc bằng L_1 và lớn hơn hoặc bằng L_2.

Ví dụ: có 6 kho vàng có trữ lượng tương ứng là $3, 5, 9, 6, 7, 4$. Tập con có tổng lớn nhất là ở kho số $1, 3, 5$ sao cho khoảng cách giữa hai kho lớn hơn hoặc bằng 2 và nhỏ hơn hoặc bằng 3

Bài 5: Một bác nông dân mua n cây giống để trồng trong vườn. Bác nông dân trồng mỗi cây mất 1 ngày và bác nông dân biết chính xác số ngày cây đó trưởng thành. Bác muốn tổ chức một bữa tiệc sau khi tất cả n cây này đều đã trưởng thành. Hãy giúp bác nông dân xác định được số ngày sớm nhất để tổ chức bữa tiệc và xác định thứ tự trồng n cây trên. Biết t_i là số ngày trưởng thành của cây thứ i. Ví dụ có 6 cây với số ngày trưởng thành tương ứng $39, 38, 9, 35, 39, 20$ thì số ngày tổ chức sớm nhất là 42.

Chương 4

Mô hình dữ liệu danh sách

4.1 Danh sách kế tiếp

4.1.1 Định nghĩa danh sách kế tiếp

Danh sách là một tập hợp hữu hạn biến động các phần tử thuộc cùng một lớp các đối tượng nào đó (theo quan điểm lập trình: có cùng một kiểu dữ liệu).

Danh sách được cài đặt bởi mảng hay còn gọi là CTDL danh sách đặc, hoặc CTDL danh sách kế tiếp, gọi tắt là: Danh sách đặc, hoặc danh sách kế tiếp, nó thuộc loại cấu trúc dữ liệu tĩnh.

4.1.2 Biểu diễn danh sách kế tiếp

Mô tả dạng biểu diễn danh sách trên máy tính

Bảng 4.1: Danh sách kế tiếp

Chi so mang	0	1	...	size	...	N-1
Thong tin	phan tu 1	phan tu 2	...	phan tu cuoi	rong	rong

Giá sử N là số phân tử tối đa trong danh sách, với cách cài đặt này ta phải ước lượng số phần tử tối đa của danh sách để khai báo số phần tử của mảng cho thích hợp. Giả sử item là kiểu dữ liệu của các phần tử trong danh sách. Khi đó ta sẽ biểu diễn danh sách như sau:

Dùng một mảng để lưu giữ các phần tử của danh (elems), và một biến đếm để đếm số lượng phần tử hiện có trong danh sách (size). Như vậy ta có thể định nghĩa danh sách như một cấu trúc gồm 2 trường:

elems: Chứa các phần tử trong danh sách;

size: Đếm số phần tử hiện có trong danh sách (chiều dài danh sách);

Mảng chứa các phần tử trong danh sách có dạng như Bảng 4.1.

Dạng biểu diễn của danh sách:

```
#define N 100 //so phan tu toi da la 100
typedef int item;
/*kieu cac phan tu la item
ma cu the o day item la kieu int */
typedef struct{
    item elems[N]; //mang kieu item
    int size; //so phan tu toi da cua mang
}List; //kieu danh sach List
```

4.1.3 Các thao tác trên danh sách kế tiếp

1 – Khởi tạo danh sách rỗng

Danh sách rỗng là một danh sách không chứa bất kỳ một phần tử nào (hay độ dài danh sách bằng 0).Vì vậy để khởi tạo danh sách rỗng chỉ cần khai báo trường size của ta bằng 0.

Giải thuật khởi tạo danh sách rỗng

```
void init(List *L) //ham khoi tao danh sach rong
    /*Danh sach L duoc khai bao kieu con tro
    de khi ra khoi ham no co the thay doi duoc*/
{
    (*L).size = 0; //size = 0.
}
```

2 – Kiểm tra danh sách rỗng, kiểm tra danh sách đầy

Danh sách rỗng khi độ dài của nó bằng 0 và danh sách đầy khi độ dài của nó bằng N. Do vậy, để kiểm tra danh sách rỗng hay đầy ta chỉ việc xem số phần tử của danh sách có bằng 0 hay không (rỗng) và có bằng N hay không (đầy).

Giải thuật kiểm tra tính rỗng của danh sách

```
int isEmpty (List L){
    return (L.size==0);
}
```

Giải thuật kiểm tra tính đầy của danh sách

```
int isFull (List L){
    return (L.size==N);
}
```

3 – Chèn một phần tử vào danh sách

Để thêm một phần tử có giá trị x vào vị trí thứ k trong danh sách, trước hết ta cần kiểm tra một số trường hợp không thêm được sau:

- Danh sách đầy: là khi độ dài của danh sách bằng chỉ số tối đa của mảng; khi đó không còn chỗ cho phần tử mới, vì vậy việc thêm là không thể thực hiện được.
- Vị trí thêm không hợp lệ: có 2 vị trí thêm không hợp lệ, thứ nhất là k<0 – khi đó k không phải là một vị trí trong mảng hay danh sách, thứ hai là k>size+1 – khi đó việc thêm một phần tử sẽ làm cho danh sách L không còn là một danh sách

đặc nữa.

Khi các điều kiện đều hợp lệ thì ta tiến hành thêm theo các bước sau:

- Dời các phần tử từ vị trí cuối danh sách đến vị trí k sang phải 1 ô nhớ;
- Đưa phần tử mới vào vị trí thứ k;
- Tăng độ dài danh sách lên 1 đơn vị.

Trước khi chèn phần tử vào trong danh sách chúng ta nên xây dựng hàm init_x để nhập dữ liệu của phần tử cần chèn.

```
item init_x() //khoi tao gia tri x{
    int temp;
    scanf("%d",&temp);
    return temp;
}
```

Sau đó tiến hành chèn:

```
int insert_k (List *L, item x, int k){//chen x vao vi tri k
    if (isFull(*L)) //kiem tra danh sach day{
        printf("Danh sach day !");
        return 0;
    }
    if (k<1 || k>(*L).size+1){ //kiem tra dieu kien vi tri chen
        printf("Vi tri chen khong hop le !\n");
        return 0;
    }
    printf ("Nhap thong tin: ");
    x = init_x(); //gan x = ham khoi tao x
    int i;
    //di chuyen cac phan tu ve cuoi danh sach
    for (i = (*L).size; i >= k; i--)
        (*L).elems[i] = (*L).elems[i-1];
    (*L).elems[k-1]=x;//chen x vao vi tri k
    (*L).size++;//tang size len 1 don vi.
    return 1;
}
```

4 – Tìm phần tử có giá trị x trong danh sách

Ta sẽ duyệt từ đầu đến cuối danh sách, nếu tìm thấy phần tử có giá trị x thì trả về vị trí tìm thấy.

```
int search (List L, item x){
    int i;
    for (i=0; i<L.size; i++)
        if (L.elems[i] == x)
            return i+1;
    return 0;
}
```

5 – Xóa phần tử ở vị trí thứ k trong danh sách

Tương tự như phép thêm một phần tử vào danh sách, để thực hiện phép xóa một phần tử trong danh sách ta cũng cần kiểm tra một số điều kiện không xóa được:

- Danh sách rỗng: là khi danh sách không có phần tử nào, đồng nghĩa với việc không có phần tử để xóa.
- Vị trí xóa không hợp lệ: có 2 vị trí không hợp lệ, đó là k<0 và k>size – đây không phải là vị trí phần tử của danh sách.

Khi các điều kiện đều hợp lệ thì ta tiến hành xóa theo các bước sau:

- Dồn các phần tử từ vị trí k+1 đến cuối danh sách sang trái một vị trí;
- Giảm size đi một đơn vị;
- Lưu lại giá trị của phần tử xóa trước khi dồn các phần tử để lưu lại thông tin khi ta cần dùng đến.

```
int del_k (List *L, item *x, int k){
    if (isEmpty(*L)){
        printf("Danh sach rong !");
        return 0;
    }
    if (k<1 || k>(*L).size){
        printf("Vi tri xoa khong hop le !");
        return 0;
```

```
    }
    *x=(*L).elems[k-1]; //luu lai gia tri cua phan tu can xoa
    int i;
    for (i=k-1; i<(*L).size-1; i++) //don cac phan tu ve truoc
        (*L).elems[i]=(*L).elems[i+1];
    (*L).size--; //giam size
    return 1;
}
```

6 – Xóa phần tử có nội dung x trong danh sách

Để xóa phần tử có nội dung x trong danh sách ta tiến hành tìm phần tử x trước bằng hàm search sau đó giá trị trả về là vị trí của x, ta tiếp tục sử dụng hàm del_k để xóa phần tử ở vị trí mà ta tìm được.

```
int del_x (List *L, item x){
    if (isEmpty(*L)){
        printf("Danh sach rong !");
        return 0;
    }
    int i = search(*L,x);
    if (!i){
        printf("Danh sach khong co %d",x);
        return 0;
    }
    do{
        del_k(L,&x,i);
        i = search(*L,x);
    }
    while (i);
    return 1;
}
```

7 – Nhập dữ liệu cho danh sách

Để nhập dữ liệu cho danh sách, ta nhập vào số lượng phần tử của danh sách - gán cho biến size, sau đó dùng vòng lặp nhập lần lượt từng phần tử.

```
void input (List *L){
    int n;
    printf("Nhap so phan tu cua danh sach: ");
    scanf("%d",&(*L).size);
    int i;
    for (i=0; i<(*L).size; i++){
        printf("Nhap phan tu thu %d : ",i+1);
```

```
            (*L).elems[i] = init_x();
        }
    }
```

8 – Xuất dữ liệu trong danh sách

Để xuất dữ liệu trong danh sách, ta dùng vòng lặp chạy từ đầu đến cuối danh sách, in lần lượt các phần tử lên màn hình.

```
void output (List L){
    printf("Danh sach: \n");
    int i;
    for (i=0; i<L.size; i++)
        printf("%5d",L.elems[i]);
    printf("\n");
}
```

4.1.4 Ứng dụng của danh sách kế tiếp

Danh sách cài đặt bằng mảng có ưu điểm là dễ sử dụng, tốc độ truy cập cao, việc truy xuất đến một phần tử trong danh sách được thực hiện dễ dàng và nhanh chóng thông qua chỉ số của mảng.

Tuy nhiên, cài đặt bằng mảng đòi hỏi phải xác định số phần tử của mảng, do đó nếu không thể ước lượng được số phần tử trong danh sách thì khó áp dụng cách cài đặt này một cách hiệu quả vì nếu khai báo thiếu chỗ thì mảng thường xuyên bị đầy, không thể làm việc được còn nếu khai báo quá thừa thì lãng phí bộ nhớ.

Danh sách kế tiếp thích hợp với danh sách ít có sự biến động về số lượng các phần tử.

4.2 Danh sách liên kết đơn

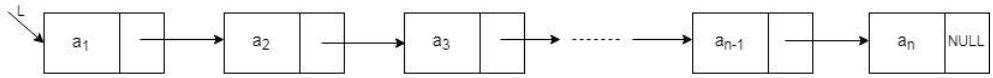
4.2.1 Định nghĩa danh sách liên kết đơn

Danh sách được cài đặt bởi con trỏ ta còn gọi là CTDL danh sách liên kết hay CTDL danh sách móc nối, gọi tắt là danh sách liên

kết/danh sách móc nối. Chúng thuộc loại cấu trúc dữ liệu động.

Mỗi phần tử trong danh sách là một ô nhớ, mỗi ô nhớ là một cấu trúc ít nhất là hai ngăn, một ngăn chứa dữ liệu của phần tử đó, một ngăn là con trỏ chứa địa chỉ của ô nhớ đứng kế sau phần tử này trong danh sách, ta có thể hình dung cơ chế này qua ví dụ sau:

Giả sử 1 nhóm có 4 bạn: Đông, Tây, Nam, Bắc có địa chỉ nhà ở lần lượt là d,t,n,b. Giả sử: Đông có địa chỉ của Nam, Tây không có địa chỉ của bạn nào, Bắc giữ địa chỉ của Đông, Nam có địa chỉ của Tây, điều này được mô tả qua hình 4.1.



Hình 4.1: Ví dụ về danh sach lien ket don.

Như vậy, nếu ta xét thứ tự các phần tử bằng cơ chế lưu địa chỉ này thì ta có một danh sách: Bắc, Đông, Nam, Tây. Hơn nữa để có thể truy cập đến các phần tử trong danh sách này thì chỉ cần giữ địa chỉ của Bắc (địa chỉ của ô nhớ chứa phần tử đầu tiên trong danh sách).

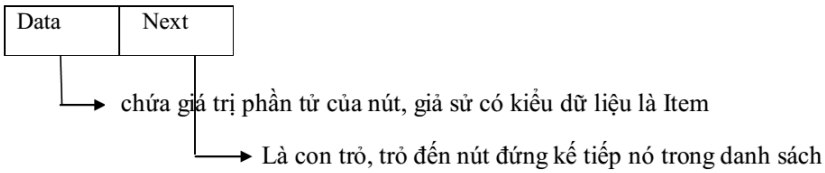
4.2.2 Biểu diễn danh sách liên kết đơn

Mô tả dạng biểu diễn danh sách trên máy tính

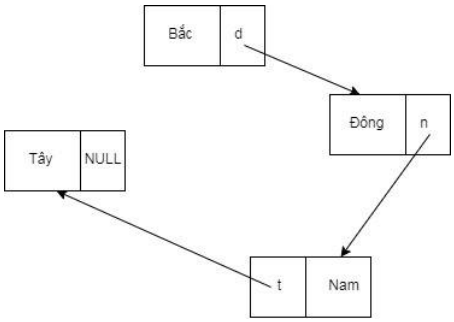
Trong cài đặt, mỗi phần tử trong danh sách được cài đặt như một nút có hai trường:

- Data: chứa giá trị của các phần tử trong danh sách;
- Next: con trỏ giữ địa chỉ của ô kế tiếp nó trong danh sách.

Cấu trúc một nút được mô tả như Hình 4.2
Hình ảnh danh sách được mô tả như Hình4.3



Hình 4.2: Cấu trúc một nút của danh sách liên kết đơn.



Hình 4.3: Hình ảnh danh sách liên kết đơn.

Nút cuối cùng trong danh sách không có nút đứng sau, nên trường Next của phần tử cuối trong danh sách, trỏ đến một giá trị đặc biệt là NULL (không trỏ tới đâu). Để truy nhập vào danh sách ta phải truy nhập tuần tự đến vị trí mong muốn, xuất phát từ phần tử đầu tiên, do đó để quản lý danh sách ta chỉ cần quản lý địa chỉ ô nhớ chứa phần tử đầu tiên của danh sách, tức là cần một con trỏ trỏ đến phần tử đầu tiên này - giả sử con trỏ L. L còn gọi là con trỏ quản lý danh sách. Danh sách L rỗng khi L=NULL.

Dạng biểu diễn của danh sách:

```
typedef struct node{
   int data;
   node* next;
}node;
typedef node* LinkList;
```

4.2.3 Các thao tác trên danh sách liên kết đơn

1- Tạo danh sách rỗng

Mục đích phép toán khởi tạo là tạo ra một danh sách rỗng, nghĩa là con trỏ quản lý đầu danh sách chưa trỏ vào đâu, nên với phép toán khởi tạo ta chỉ cần cho con trỏ L=NULL.

```
void init (LinkList &L){
    L=NULL; //Cho L tro den NULL
}
```

2- Kiểm tra một danh sách rỗng

Để kiểm tra một danh sách có rỗng hay không, ta chỉ cần kiểm tra con trỏ L có bằng NULL hay không.

```
int isEmpty (LinkList L){
    return (L==NULL);
}
```

3 - Tính độ dài của danh sách

Để tính độ dài của danh sách, ta cần con trỏ phụ trỏ vào đầu danh sách và biến đếm, ta sẽ dùng con trỏ phụ này để duyệt qua từng phần tử của danh sách, mỗi lần qua một phần tử thì tăng biến đếm lên 1 đơn vị.

```
int len (LinkList L) {
    node *P=L; //tao 1 Node P de duyet danh sach L
    int i=0; //bien dem
    while (P!=NULL){
        i++; //tang bien dem
        P=P->next; //cho P tro den Node tiep theo
    }
    return i; //tra lai so Node cua L hay do dai cua danh sach
}
```

4 - Tạo một Node trong danh sách

Để tạo một Node P chứa dữ liệu x, ta cần xin cấp phát vùng nhớ cho con trỏ P, sau đó cho trường next trỏ đến NULL và ghi dữ liệu vào trường data.

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

```
    return P;
}
```

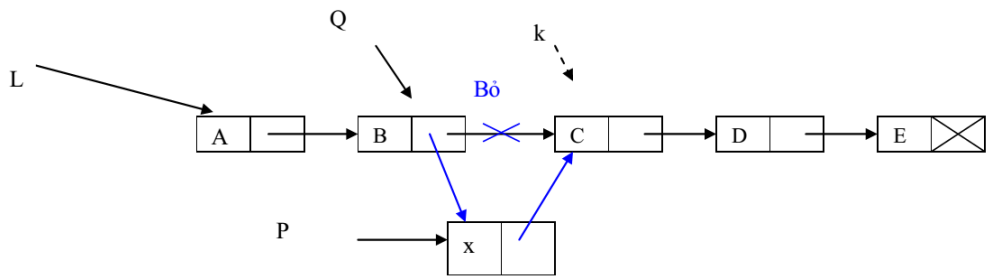
5 - Chèn một Node P vào đầu danh sách

Để chèn một phần tử vào đầu danh sách, trước hết ta gọi phép toán tạo node P chứa dữ liệu x đã viết ở trên, sau đó gắn lại các mối liên kết: cho P trỏ đến L rồi cho L trỏ về P.

```
void insert_First (LinkList &L, item x) {
    node *P;
    P = make_Node(P,x); //tao 1 Node P
    P->next = L; //Cho P tro den L
    L = P; //L tro ve P
}
```

6 - Chèn một Node P vào vị trí thứ k trong danh sách

Trước tiên ta kiểm tra vị trí chèn có hợp lệ không, nếu hợp lệ kiểm tra tiếp chèn vào vị trí 1 hay k >1. Nếu k >1 ta thực hiện duyệt bằng node Q bắt đầu từ đầu danh sách đến vị trí k-1 sau đó cho P->next trỏ đến Q->next, tiếp đến cho Q->next trỏ đến P (xem Hình 4.4.)



Hình 4.4: Chèn một phần tử vào vị trí p của danh sách liên kết đơn.

Giải thuật chèn:

```
void insert_k (LinkList &L, item x, int k){
    node *P, *Q = L;
    int i=1;
    if (k<1 || k> len(L)+1)
        printf("Vi tri chen khong hop le !"); //kiem tra dieu kien
    else{
        P = make_Node(P,x); //tao 1 Node P
```

```
        if (k == 1)
            insert_First(L,x); //chen vao vi tri dau tien
        else{ //chen vao k != 1
            while (Q != NULL && i != k-1){ //duyet den vi tri k-1
                i++;
                Q = Q->next;
            }
            P->next = Q->next;
            Q->next = P;
        }
    }
}
```

7- Tìm phần tử có giá trị x trong danh sách

Để tìm phần tử có giá trị x trong danh sách, ta dùng con trỏ phụ duyệt từ đầu đến cuối danh sách, đồng thời kiểm tra xem giá trị x có trùng với dữ liệu ở trường data của các phần tử hay không, nếu tìm thấy thì trả về vị trí.

```
int search (LinkList L, item x){
    node *P=L;
    int i=1;
    while (P != NULL && P->data != x){
        P = P->next;
        i++;
    }
    if (P != NULL)
        return i; //tra ve vi tri tim thay
    else
        return 0; //khong tim thay
}
```

8- Xóa phần tử ở vị trí đầu tiên trong danh sách

Trước tiên ta lưu giá trị của phần tử đầu tiên vào biến x, sau đó tiến hành cho L trỏ đến L->Next.

```
void del_Frist (LinkList &L, item &x){
    x = L->data; //lay gia tri ra neu can dung
    L = L->next; //Cho L tro den Node thu 2 trong danh sach
}
```

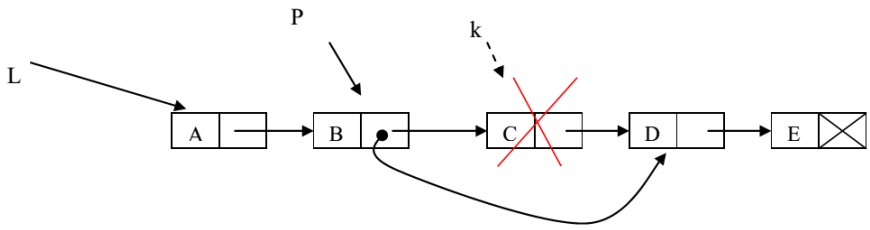
9- Xóa phần tử ở vị trí thứ k trong danh sách

Tương tự như khi thêm một phần tử vào danh sách liên kết,

muốn xóa một phần tử khỏi danh sách ta cần:

- Xác định vị trí của phần tử muốn xóa trong danh sách L, giả sử ví trí thứ k, ta di chuyển con trỏ P tới vị trí trước k (vị trí k-1);
- Cho P->next trỏ đến phần tư kế tiếp k mà bỏ qua k.

Xem Hình 4.5 để thêm chi tiết.



Hình 4.5: Xóa phần tử thứ k ra khỏi danh sách liên kết đơn.

Giải thuật xóa:

```
void del_k (LinkList &L, item &x, int k) {
    node *P=L;
    int i=1;
    if (k<1 || k>len(L))
        printf("Vi tri xoa khong hop le !"); //kiem tra dieu kien
    else{
        if (k==1)
            del_First(L,x); //xoa vi tri dau tien
        else{ //xoa vi tri k != 1
            while (P != NULL && i != k-1){ //duyet den vi tri k-1
                P=P->next;
                i++;
            }
          x=P->data;
            P->next = P->next->next;
        }
    }
}
```

10 – Nhập dữ liệu cho danh sách

Để nhập dữ liệu cho danh sách, ta có thể kết hợp dùng vòng lặp với 1 trong 3 phép toán thêm: thêm đầu, thêm vị trứ thứ k, thêm

cuối. Dưới đây mô phỏng phép toán nhập dữ liệu kết hợp với phép toán thêm một phần tử vào vị trí thứ k.

```
void input (LinkList &L){
    int i=0;
    item x;
    do {
        i++;
        printf ("Nhap phan tu thu %d : ",i);
        scanf("%d",&x);
        if (x != 0)
            insert_k(L,x,len(L)+1);
    } while(x != 0); //nhap 0 de ket thuc
}
```

11- In dữ liệu trong danh sách ra màn hình

Để xuất dữ liệu trong danh sách ra màn hình, ta dùng con trỏ phụ duyệt từ đầu đến cuối danh sách, đi qua phần tử nào thi in trường data của phần tử đó ra màn hình.

```
void output (LinkList L){
    node *P=L;
    while (P != NULL){
        printf("%5d",P->data);
        P = P->next;
    }
    printf("\n");
}
```

4.2.4 Ứng dụng của danh sách liên kết đơn

1. Khi cần xóa chèn liên tục (trong các ứng dụng real-time chẳng hạn);

2. Khi ta không biết trước số lượng phần tử của danh sách, đặc biệt là khi bộ nhớ hữu hạn không cho phép ta thực hiện các biện pháp dồn dữ liệu trên mảng;

3. Khi ta không có nhu cầu truy xuất ngẫu nhiên vào phần tử nào;

4. Khi ta muốn chèn một phần tử vào giữa danh sách (như trong cài đặt các hàng đợi ưu tiên – priority queue – chẳng hạn).

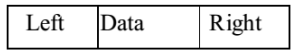
4.3 Danh sách liên kết kép

4.3.1 Định nghĩa danh sách liên kết kép

Chúng ta nhận thấy rằng với danh sách liên kết đơn chỉ có phép duyệt 1 chiều, từ phần tử trước có thể truy nhập đến phần tử đứng sau, nhưng từ phần tử đứng sau không truy cập trực tiếp đến phần tử đứng ngay trước nó được. Khắc phục hạn chế này ta có danh sách liên kết kép. Mỗi phần tử trong danh sách liên kết kép xem là một nút (cấu trúc) gồm 3 trường:

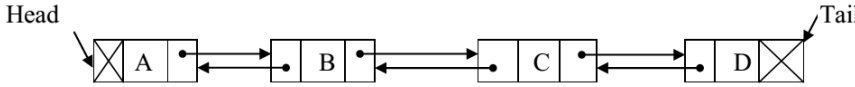
- data: chứa thông tin về đối tượng.
- left: con trỏ trỏ tới phần tử bên trái.
- right: con trỏ trỏ tới phần tử bên phải.

Cấu trúc một nút trong danh sách liên kết kép có dạng như Hình 4.6



Hình 4.6: Cấu trúc một nút trong danh sách liên kết kép.

Hình ảnh danh sách móc nối kép như chỉ ra trong Hình 4.7



Hình 4.7: Hình ảnh danh sách liên kết kép.

Việc truy nhập các phần tử của d/s phải truy cập xuất phát từ một trong hai đầu của danh sách. Do đó, để quản lý danh sách,

dùng 2 con trỏ Head, Tail lần lượt trỏ tới nút đầu tiên và cuối cùng của danh sách, hai con trỏ này còn gọi là con trỏ đầu, cuối của danh sách. Khi đó d/s rỗng nếu: Head=Tail= NULL.

4.3.2 Biểu diễn danh sách liên kết kép

```
typedef int item;
typedef struct node {//cau truc 1 node
    item data; //du lieu cua node
    node *left; //Con tro trai
    node *right; //con tro phai
};
typedef struct DList{ //cau truc Cua Danh sach
    node *head; //con tro dau
    node *tail; //con tro cuoi
};
```

4.3.3 Các thao tác trên danh sách liên kết kép

1- Khởi tạo và kiểm tra tính rỗng của danh sách

```
void init(DList &L) {
    L.head = NULL; // Con tro dau tro den NULL
    L.tail = NULL; // Con tro cuoi tro den NULL
}
int isEmpty (DList L){ // kiem tra tin rong cua ds
    return (L.head == NULL);
}
```

2- Tính độ dài danh sách

Để tính độ dài của danh sách liên kết kép hoàn toàn có thể làm giống như danh sách liên kết đơn, tức dùng con trỏ duyệt từ đầu đến cuối, nhưng trong danh sách liên kết kép ta có thể dùng 2 con trỏ ở đầu và cuối để đếm.

```
int len (DList L) {
    node *PH = L.head, *PT = L.tail; //tao node PH (con tro duyet tu dau DS) v
        PT (con tro duyet tu cuoi DS) de duyet danh sach L
    int i = 0; //bien dem
    if (PH != NULL) i = 1;
    while (PH != NULL){
        if (PH == PT)
```

```
            break;
        PH = PH->right; //cho PH tro den node tiep theo
        i++;
        if (PH == PT)
            break;
        PT = PT->left; //cho PT tro den node truoc do
        i++;
    }
    return i; //tra lai so node cua L
}
```

3- Tạo một node P chứa thông tin:

Để tạo một Node P chứa dữ liệu x, ta cần xin cấp phát vùng nhớ cho con trỏ P, ghi dữ liệu vào trường data sau đó cho trường left và right trỏ đến NULL.

```
Node *make_Node (item x) {
    node *P = (node *) malloc (sizeof (node));
    P->data = x; //Ghi du lieu vao data
    P->left = NULL;
    P->right = NULL;
    return P;
}
```

4- Chèn một phần tử vào đầu danh sách

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

```
void insert_First (DList &L, item x){
    node *P;
    P = make_Node(x); //tao 1 node P
    if (isEmpty(L)){ //Neu danh sach rong
        L.head = P;
        L.tail = P;
    } else {
        P->right = L.head;
        L.head->left = P;
        L.head = P;
    }
}
```

5- Chèn một phần tử vào cuối danh sách:

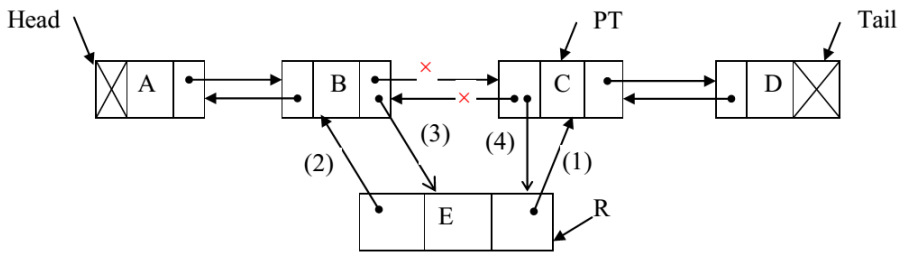
Trước khi chèn vào cuối danh sách cần kiểm tra xem danh sách

rỗng hay không. Nếu danh sách rỗng ta cho Head và Tail đều trỏ đến P. Nếu không rỗng thực hiện chèn vào sau Tail.

```
void insert_Last (DList &L, item x){
    node *P;
    P = make_Node(x); //tao 1 node P
    if (isEmpty(L)){
        L.head = P;
        L.tail = P;
    }else{
        L.tail->right = P; //ket noi voi danh sach
        P->left = L.tail; //P tro ve node truoc
     L.tail = P; //luu lai vi tri cuoi
    }
}
```

6- Chèn một phần tử vào vị trí thứ k trong danh sách

Các bước để chèn thêm một phần tử mới vào danh sách liên kết kép được thực hiện tương tự như danh sách liên kết đơn, tuy nhiên khi gắn kết nút tương ứng với phần tử mới (nút được trỏ bởi con trỏ q) vào vị trí trước phần tử được trở bởi con trỏ M được thực hiện từ bước (1) đến (4) như mô tả như Hình 4.8.



Hình 4.8: Chèn phần tử vào vị trí k của danh sách liên kết kép.

Giải thuật chèn:

```
void insert_k (DList &L, item x, int k) {
    node *PH = L.head, *PT, *R;
    int i=1, l = len(L);
    if (k<1 || k> l+1)
        printf("Vi tri chen khong hop le !"); //kiem tra dieu kien
    else{
        R = make_Node(x); //tao 1 node P
        if (k == 1)
            insert_First(L,x); //chen vao vi tri dau tien
```

```
        else
            if (k == l+1)
                insert_Last(L,x); //chen vao vi tri cuoi
            else{ //chen vao vi tri 1<k<l+1
                while (PH != NULL && i != k-1){ //duyet den vi tri k-1
                    i++;
                    PH = PH->right;
                }
                PT = PH->right; //xac dinh vi tri k
                R->right = PT; //(1)
                R->left = PH;  //(2)
                PH->right = R; //(3)
                PT->left = R;  //(4)
            }
    }
}
```

7- Xóa phần tử đầu danh sách

Để xóa phần tử ở đầu danh sách, trước tiên ta lưu lại giá trị node cần xóa, sau đó bỏ qua node đầu tiên bằng cách cho con trỏ L trỏ đến node thứ 2 trong danh sách.

```
// Lay gia tri can xoa ra, sau do bo qua 1 node dau tien
void del_First (DList &L, item &x){
    if (!isEmpty(L)){
        x = L.head->data; //lay gia tri ra neu can dung
        L.head = L.head->right; //Cho L tro den node thu 2 trong danh sach
    }
}
```

8- Xóa phần tử cuối danh sách

Để xóa phần tử ở cuối danh sách, trước tiên ta lưu lại giá trị node cần xóa, sau đó bỏ qua node cuối.

```
// Lay gia tri can xoa ra, sau do bo qua 1 node cuoi
void del_Last (DList &L, item &x) {
    if (!isEmpty(L)){
        x = L.tail->data;
        L.tail = L.tail->left;
        L.tail->right = NULL;
    }
}
```

9- Xóa phần tử ở vị trí thứ k trong danh sách

Để xóa phần tử ở vị trí thứ k, ta cần con trỏ phụ duyệt từ đầu

danh sách đến vị trí k-1, sau đó thực hiện xóa phần tử thứ k bằng cách bỏ qua phần tử sau k-1, móc nối lại mối liên kết với phần tử k+1.

```
void del_k (DList &L, item &x, int k){
    node *PH = L.head, *PT;
    int i=1, l = len(L);
    if (k<1 || k> l)
        printf("Vi tri xoa khong hop le !"); //kiem tra dieu kien
    else{
        if (k == 1)
            del_First(L,x); //xoa vi tri dau tien
        else
            if (k == l)
                del_Last(L,x); //xoa vi tri cuoi
            else{ //xoa vi tri 1<k<l+1
                while (PH != NULL && i != k-1){ //duyet den vi tri k-1
                    i++;
                    PH = PH->right;
                }
                x = PH->right->data;
                PT = PH->right->right; //xac dinh vi tri k+1
                PH->right = PT;
                PT->left = PH;
            }
    }
}
```

10- Tìm phần tử x trong danh sách

Để tìm phần tử có giá trị x trong danh sách, ta tiến hành duyệt từ đầu đến cuối danh sách, nếu tìm thấy thì trả về vị trí tìm thấy.

```
int search (DList L, item x) {
    node *P=L.head;
    int i=1;
    while (P != NULL && P->data != x){ //duyet danh sach den khi tim thay hoac
         ket thuc danh sach
        P = P->right;
        i++;
    }
    if (P != NULL)
        return i; //tra ve vi tri tim thay
    else
        return 0; //khong tim thay
}
```

4.3.4 Ứng dụng của danh sách liên kết kép

Một số ứng dụng đòi hỏi chúng ta phải duyệt danh sách theo cả hai chiều, khi đó sử dụng danh sách liên kết kép là lựa chọn phù hợp.Để thuận tiện cho việc truy cập, ta nên dùng hai con trỏ, một con trỏ chỉ đến phần tử đứng sau cùng hay con gọi là phần tử phải nhất (right), một con trỏ trỏ đến phần tử đứng đầu tiên, hay còn gọi là phần tử trái nhất (left) thay vì chỉ dùng một con trỏ như danh sách liên kết đơn.

4.4 Ngăn xếp (Stack)

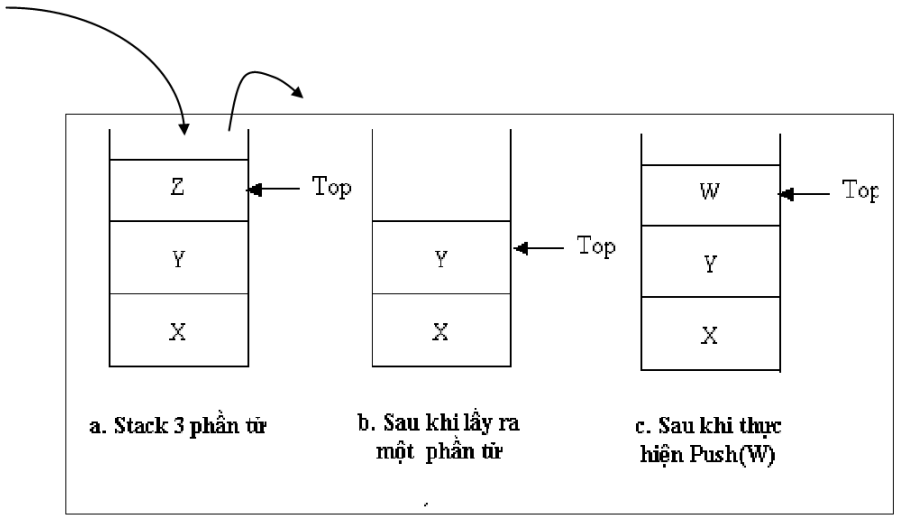
4.4.1 Định nghĩa ngăn xếp

Ngăn xếp là một danh sách đặc biệt, trong đó việc thêm vào hoặc loại bỏ một phần tử chỉ thực hiện tại một đầu của danh sách, đầu này gọi là đỉnh (TOP) của ngăn xếp.

Ví dụ, có thể xem hình ảnh trực quan của ngăn xếp bằng một chồng đĩa đặt trên bàn. Muốn thêm vào chồng đó 1 đĩa ta để đĩa mới trên đỉnh chồng, muốn lấy các đĩa ra khỏi chồng ta cũng phải lấy đĩa trên đỉnh trước. Như vậy ngăn xếp là một cấu trúc có tính chất "vào sau - ra trước" hay "vào trước – ra sau" (LIFO: Last In - First Out hay FILO: First In – Last Out). Hình ảnh của ngăn xếp có dạng như Hình 4.9.a; lấy một phần tử ra khỏi ngăn xếp (POP) mô tả như Hình 4.9.b, thêm một phần tử vào ngăn xếp được (PUSH) mô tả như Hình 4.9.c.

4.4.2 Biểu diễn ngăn xếp trên máy tính

Tương tự danh sách, ngăn xếp có thể biểu diễn bởi mảng hoặc con trỏ, ví dụ về biểu diễn ngăn xếp bởi mảng như sau:



Hình 4.9: Cấu trúc ngăn xếp và các thao tác PUSH, POP.

```
#include<stdio.h>
#define N 100
typedef int item;
typedef struct Stack{
   item elems[N];
   int top;// chi vi tri them vao va lay ra
}Stack;
```

4.4.3 Các thao tác trên ngăn xếp

1- Tạo ngăn xếp rỗng

Với phép toán khởi tạo ngăn xếp rống, ta nên để chỉ số top=-1 để khi thêm phần tử đầu tiên vào ngăn xếp, top tăng lên 1 đơn vị và chỉ vào phần tử đầu tiên.

```
// phep toan khoi tao
void init_S(Stack *S){
   S->top=-1;
}
```

2 - Kiểm tra ngăn xếp rỗng, tính đầy của ngăn xếp

Ngăn xếp rỗng khi top =-1 và đầy khi top=N-1

```
// pheps toan kiem tra ngan xep rong
int isEmpty_S(Stack S){
   if(S.top ==-1)
       return 1;
   else
       return 0;
}
// Kiem tra ngan xep day
int isFull_S(Stack S){
   if(S.top ==N-1)
       return 1;
   else
       return 0;
}
```

3 - Thêm phần tử vào ngăn xếp

Để thêm một phần tử vào đỉnh ngăn xếp, ta tăng chỉ số top lên 1 đơn vị, sau đó đẩy phần tử cần thêm vào top.

```
// Day 1 phan tu x vao ngan xep S
int push(item x, Stack *S){
    if (!isFull_S(*S)){
       S->top++;
       S->elems [S->top ]=x;
   } else
    printf("\n NGan xep day.");
    }
```

4 – Lấy 1 phần tử ra khỏi ngăn xếp

Để lấy 1 phần tử ra khỏi ngăn xếp S và lưu vào biến x, trước hết ta lưu lại giá trị tại top vào biến x, sau đó giảm top đi 1 đơn vị.

```
// Nhac 1 ptu ra khoi ngan xep
int pop(Stack *S, item *x){
   if (!isEmpty_S(*S)){
      *x=S->elems [S->top];
      S->top--;
   }
}
//-------------- Sau day la chuong trinh chinh
int main(){
   int n;
   struct Stack S;
   init_S(&S);
   printf("\n Nhap mot so nguyen:");
   scanf("%d",&n);
   while (n!=0){
      int r;
```

```
        r=n%2;
        push(r,&S);
        n=n/2;
    }
    printf("\n Dang nhi phan cua so vua nhap:");
    while(!isEmpty_S(S)){
        int x;
        pop(&S,&x);
        printf ("%d",x);
    }
    //getch();
    return 0;
}
```

4.4.4 Một số bài toán ứng dụng ngăn xếp

1- Đảo ngược xâu ký tự

Bài toán đảo ngược xâu ký tự yêu cầu hiển thị các ký tự của 1 xâu ký tự theo chiều ngược lại.

Ký tự cuối cùng của xâu sẽ được hiển thị trước, tiếp theo là ký tự sát ký tự cuối ... và ký tự đầu tiên sẽ được hiển thị cuối cùng. Để giải quyết bài toán, ta chỉ cần duyệt từ đầu đến cuối xâu, lần lượt cho các ký tự vào ngăn xếp. Khi đó, các ký tự đầu tiên sẽ vào trước, tiếp theo đến ký tự thứ 2 ... ký tự cuối cùng vào sau cùng. Sau khi đã cho toàn bộ ký tự của xâu vào ngăn xếp, lần lượt lấy các phần tử ra khỏi ngăn xếp và hiển thị lên màn hình.

2- Đổi một số từ hệ thập phân n sang hệ nhị phân

Các bước thực hiện như sau: n chia liên tiếp cho 2, số dư lưu vào ngăn xếp. Cứ thực hiện cho đến khi nào phần nguyên bằng 0 thì dừng. Để in ra dãy nhị phân chỉ cần lấy từng phần tử ra khỏi ngăn xếp và in ra màn hình.

Đoạn mã mô tả như sau:

```
init_S(S);//khoi tao ngan xep rong
while (n!=0)
{
```

```
    r=n%2;
    push(r,S);
    n=n/2;
}
while (!Is_empty_S(S))
{
    pop(S,x);
    printf(x);
}
```

3- Tính giá trị biểu thức dạng hậu tố

Bài toán: Cho một biểu thức hậu tố. Tính giá trị của biểu thức đó.

Ví dụ tính giá trị biểu thức hậu tố sau:

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

dạng trung tố tương ứng của nó như sau:

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

Các bước tính giá trị của biểu thức hậu tố:

• Duyệt biểu thức từ trái qua phải.

• Nếu gặp toán hạng thì đưa vào ngăn xếp.

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

Với biểu thức hậu tố như trên cách tính như sau:

- Khởi tạo ngăn xếp S rỗng

- Duyệt 3, push(3,S). Ngăn xếp lúc này S chứa các phần tử: 3

- Duyệt 5, push(5,S). Ngăn xếp lúc này S chứa các phần tử: 3,5

- Duyệt 2, push(2,S). Ngăn xếp lúc này S chứa các phần tử: 3,5,2

- Duyệt '-', lấy 2 phần tử ra khỏi ngăn xếp pop(S,y), pop(S,x). Thực hiện x-y=5-2=3. Đẩy kết quả vừa tính vào ngăn xếp push(x-y,S). Lúc này ngăn xếp S chứa các phần tử: 3,3

- Duyệt 7, push(7,S). Ngăn xếp S lúc này chứa các phần tử: 3,3,7

- Duyệt 1, push(1,S). Ngăn xếp S lúc này chứa các phần tử: 3,3,7,1.

- Duyệt '+', lấy 2 phần tử ra khỏi ngăn xếp: pop(S,y), pop(S,x). Thực hiện tính x+y=1+7=8. Đẩy kết quả vào ngăn xếp push(x+y,S). Ngăn xếp lúc này chứa các phần tử: 3,3,8.

- Duyệt '*', lấy 2 phần tử ra khỏi ngăn xếp: pop(S,y), pop(S,x). Thực hiện tính x*y=3*8=24. Đẩy kết quả vào ngăn xếp push(x*y,S). Ngăn xếp lúc này chứa các phần tử: 3,24.

- Duyệt 6, push(6,S). Ngăn xếp lúc này chứa các phần tử: 3,24,6.

- Duyệt '-', lấy 2 phần tử ra khỏi ngăn xếp: pop(S,y), pop(S,x). Thực hiện tính x-y=24-6=18. Đẩy kết quả vào ngăn xếp push(x-y,S). Ngăn xếp lúc này chứa các phần tử: 3,18.

- Duyệt '*', lấy 2 phần tử ra khỏi ngăn xếp: pop(S,y), pop(S,x). Thực hiện tính x*y=3*18=54. Đẩy kết quả vào ngăn xếp push(x*y,S). Ngăn xếp lúc này chứa phần tử: 54.

Duyệt hết biểu thức thì dừng. Kết quả của biểu thức là 54.

Tóm lại: cấu trúc ngăn xếp thích hợp với việc lưu trữ các loại dữ liệu mà có trình tự lưu trữ ngược với trình tự truy xuất dữ liệu. Chúng ta có thể thấy một ứng dụng của ngăn xếp trong việc tổ chức bộ nhớ trong chương trình con theo nguyên lý hoạt động của ngăn xếp (chính vì vậy bộ nhớ này còn được gọi là Stack). Cho nên để khử đệ quy/loại bỏ tính đệ qui của chương trình chúng ta sử dụng ngăn xếp để khử đệ quy.

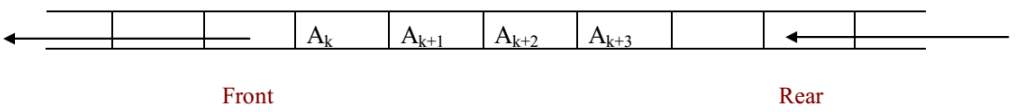
4.5 Hàng đợi (QUEUE)

4.5.1 Định nghĩa hàng đợi

Hàng đợi (hay ngắn gọn là hàng) là một danh sách đặc biệt mà phép thêm một phần tử vào hàng chỉ thực hiện tại một đầu

của hàng, gọi là cuối hàng (REAR), còn phép lấy một phần tử ra khỏi hàng thì thực hiện ở đầu còn lại của hàng, gọi là đầu hàng (FRONT).

Ví dụ, khi ta xếp hàng mua vé xem phim là một hình ảnh trực quan của khái niệm trên, người mới đến thêm vào cuối hàng còn người ở đầu hàng mua vé và ra khỏi hàng, vì vậy hàng còn được gọi là cấu trúc hoạt động theo nguyên tắc FIFO: First In - First Out, hay theo nguyên tắc "vào trước - ra trước, vào sau – ra sau". Hình ảnh của hàng đợi như mô tả trong Hình 4.10.



Hình 4.10: Cấu trúc hàng đợi.

4.5.2 Biểu diễn hàng đợi

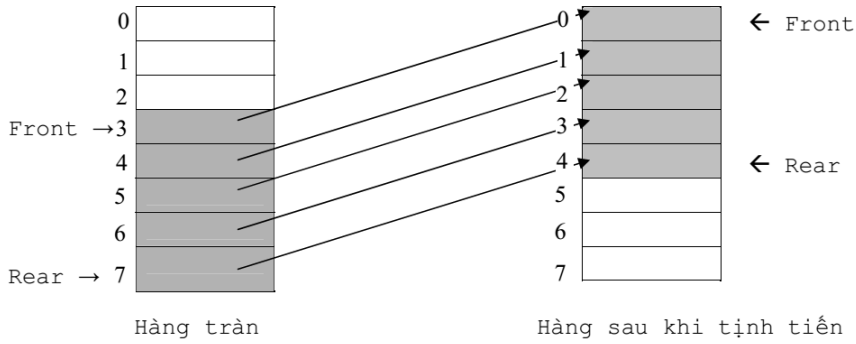
Như đã trình bày trong phần ngăn xếp, ta hoàn toàn có thể dùng danh sách để biểu diễn cho một hàng và dùng các phép toán đã được cài đặt của danh sách để cài đặt các phép toán trên hàng. Tuy nhiên làm như vậy có khi sẽ không hiệu quả, chẳng hạn dùng danh sách cài đặt bằng mảng ta thấy lời gọi INSERT_LIST(x,ENDLIST(Q),Q) tốn một hằng thời gian trong khi lời gọi DELETE_LIST(FIRST(Q),Q) để xoá phần tử đầu hàng (phần tử ở vị trí 1 của mảng) ta phải tốn thời gian tỉ lệ với số các phần tử trong hàng để thực hiện việc dời toàn bộ hàng lên một vị trí. Để cài đặt hiệu quả hơn ta phải có một suy nghĩ khác dựa trên tính chất đặc biệt của phép thêm và loại bỏ một phần tử trong hàng, đó là sử dụng hàng biểu diễn trực tiếp mảng và con trỏ.

a - Cài đặt hàng đợi bằng mảng

Sử dụng một mảng để chứa các phần tử của hàng (giả sử hàng có tối đa n phần tử, kiểu dữ liệu của các phần tử trong hàng là Item), khởi đầu, phần tử đầu tiên của hàng được đưa vào vị trí thứ 0 của mảng, phần tử thứ 2 vào vị trí thứ 1 của mảng... Giả sử hàng có n phần tử, ta có front=0 và rear=n-1. Khi xoá một phần tử chỉ số front tăng lên 1 đơn vị, khi thêm một phần tử chỉ số rear tăng lên 1 đơn vị. Như vậy hàng có khuynh hướng đi xuống, đến một lúc nào đó ta không thể thêm vào hàng được nữa (rear = n-1) dù mảng còn nhiều chỗ trống (các vị trí trước front) trường hợp này ta gọi là hàng bị tràn. Trong trường hợp toàn bộ mảng đã chứa các phần tử của hàng ta gọi là hàng bị đầy.

Các cách khắc phục hàng bị tràn:

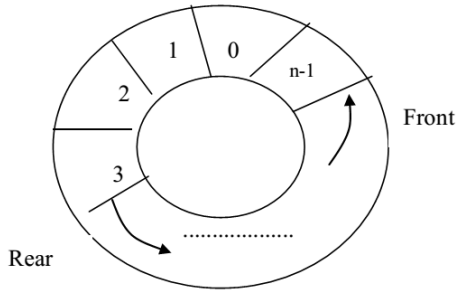
- Dời toàn bộ hàng lên front -1 vị trí: cách này gọi là di chuyển tịnh tiến. Trong trường hợp này ta luôn có front<=rear (xem Hình 4.11).



Hình 4.11: Tịnh tiến các phần tử trong hàng đợi.

- Xem mảng như là một vòng tròn: Nghĩa là khi hàng bị tràn nhưng chưa đầy ta thêm phần tử mới vào vị trí 1 của mảng, thêm một phần tử mới nữa thì thêm vào vị trí 2 (nếu có thể)...Rõ ràng cách làm này front có thể lớn hơn rear. Cách

khắc phục này gọi là dùng mảng xoay vòng. Tổ chức hàng tròn tức là ta cho phần tử thứ 1 đứng sau phần tử thứ n trong hàng ((xem Hình 4.12).



Hình 4.12: Hàng đợi cấu trúc vòng tròn.

Dạng biểu diễn hàng bằng mảng tròn

```
#include<stdio.h>
#define N 100
typedef int item;
//hang doi ghep noi vong
typedef struct Queue{
   item elem[N];// mang luu phan tu trong hang doi
   int size;// luu so luong ptu trong hang doi
   int front, rear;//chi so lay ra, them vao trong hang doi
}Queue;
```

b. Các phép toán trên hàng tròn

1 – Khởi tạo hàng rỗng

Hàng rỗng khi front và rear không trỏ đến vị trí hợp lệ nào trong mảng vậy ta có thể cho front và rear đều bằng 0, size=0. Tuy nhiên ta nên khởi tạo giá trị của rear= -1 để khi thêm một phần tử vào hàng khi hàng đang rỗng thì Q.front = Q.rear = 0: Hàng có một phần tử vào front, rear đều trỏ tới phần tử này.

```
void init_Q(Queue *Q){
   Q->size =0;
   Q->front =0;
   Q->rear =-1;
}
```

2- Kiểm tra hàng rỗng

Hàng đợi rỗng khi size = 0

```
// kiem tra hang doi rong
int isEmpty_Q(Queue Q){
   if (Q.size==0)
       return 1;
   else
       return 0;
}
```

3 - Kiểm tra hàng đầy

Hàng đợi đầy khi size=N

```
// kiem tra hang doi day
int isFull_Q(Queue Q){
   if (Q.size ==N)
       return 1;
   else
       return 0;
}
```

4 – Lấy một phần tử ra khỏi hàng

Để lấy 1 phần tử ra khỏi hàng đợi và lưu và biến x, ta lưu lại giá trị ở đầu hàng vào biến x, sau đó tăng chỉ số đầu hàng lên 1 đơn vị, giảm kích thước size xuống 1 đơn vị.

```
void deQueue(Queue*Q, item *x){
   if(!isEmpty_Q(*Q)){
      *x=Q->elem[Q->first];
      Q->size--;
      Q->first=(Q->first+1)%N;
   }
}
```

5 - Thêm một phần tử vào hàng

Để thêm một phần tử vào cuối hàng đợi, ta thực hiện tăng chỉ số cuối hàng lên 1 đơn vị, đổ dữ liệu cần thêm vào vị trí cuối hàng rồi tăng size lên 1 đơn vị.

```
//Dua 1 phan tu vao hang doi
void enQueue(item x, Queue *Q){
   if(!isFull_Q(*Q)){
```

```
        Q->size ++;
        Q->last =(Q->last +1)%N; // %N hang doi quay vong
        Q->elem [Q->last ]=x;
    }
}
```

c. Cài đặt hàng bằng con trỏ

Cách tự nhiên nhất là dùng hai con trỏ front và rear để trỏ tới phần tử đầu và cuối hàng. Hàng được cài đặt như một danh sách liên kết đơn (kép, hoặc vòng).

Giả sử ta xét dạng cài đặt hàng bằng danh sách liên kết đơn, khi đó dạng cài đặt của hàng đợi tương tự danh sách liên kết đơn, tuy nhiên để quản lý danh sách liên kết đơn ta sử dụng một con trỏ, còn để quản lý hàng đợi ta dùng 2 con trỏ, một con trỏ quản lý đầu hàng đợi và một con trỏ quản lý cuối hàng đợi.

```
typedef int item; //kieu du lieu
struct node{
    item data;
    node * next;
};
struct Queue{
    node * front, *rear; //node dau va node cuoi
    int count; //dem so phan tu
};
```

d. Các phép toán trên hàng đợi cài đặt bằng con trỏ

1 – Tạo node P chứa x

```
node *makeNode(item x) {
    node *P = (node*) malloc(sizeof(node));
    P->next = NULL;
    P->data = x;
    return P;
}
```

2 – Thêm phần tử vào cuối hàng đợi

Để thêm phần tử, ta kiểm tra xem hàng có rỗng không, nếu hàng rỗng thì cho cả front và rear cùng trỏ về node P mới tạo chứa phàn tử x cần thêm. Nếu không rỗng ta trỏ rear->next về P và rear trỏ

về P. Tăng count lên 1.

```
void push(Queue &Q, item x) {
    node *P = makeNode(x); //Neu Q rong
    if (isEmpty(Q)){
        Q.front = Q.rear = P; //dau va cuoi deu tro den P
    }else { //Khong rong
           Q.rear->next = P;
           Q.rear = P;
     }
    Q.count ++ ; //tang so phan tu len
}
```

3 – Xóa phần tử ở đầu hàng đợi

Ta kiểm tra Queue có rỗng không, Nếu không rỗng kiểm tra xem có 1 hay nhiêu hơn 1 phần tử, nếu có 1 phần tử thì ta khởi tạo lại Queue, nếu có nhiều hơn ta cho Front trỏ đến tiếp theo. Giảm count xuống 1.

```
int pop(Queue &Q){
    if (isEmpty(Q)) {
        printf("Hang doi rong !");
        return 0;
    } else{
        item x = Q.front->data;
        if (Q.count == 1) //neu co 1 phan tu
            init(Q);
        else
            Q.front = Q.front->next;
        Q.count --;
        return x; //tra ve phan tu lay ra
    }
}
```

4.5.3 Ứng dụng của hàng đợi

Hàng đợi là một cấu trúc dữ liệu được dùng khá phổ biến trong thiết kế giải thuật. Bất kỳ nơi nào ta cần quản lí dữ liệu và truy xuất chúng theo quy trình tuần tự vào trước ra trước, vào sau ra sau đều có thể sử dụng hàng đợi.

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

vô hướng cũng dùng hàng đợi để quản lí các nút đồ thị. Các giải thuật đổi biểu thức trung tố thành hậu tố, tiền tố cũng cần dùng đến cấu trúc hàng.

BÀI TẬP

Bài 1

Một cơ sở đào tạo cần quản lý n danh sách lớp sinh viên. Mỗi sinh viên cần quản lý các thông tin sau: Họ tên, giới tính, năm sinh, lớp, số báo danh, điểm tổng kết. Hãy biểu diễn dữ liệu của bài toán bởi danh sách liên kết đơn và thực hiện các thao tác sau:

1. Nhập dữ liệu cho n danh sách
2. Hiển thị sinh viên các lớp
3. Sắp xếp các danh sách theo họ tên
4. Thêm một sinh viên vào danh sách
5. Xoá một/một số sinh viên ra khỏi danh sách
6. Tìm kiếm sinh viên theo một số thông tin đã biết trước
7. Nối 2 danh sách lớp thành một lớp
8. Tách 1 danh sách thành 2 lớp
9. Huỷ danh sách: Xoá toàn bộ dữ liệu trong danh sách thứ k.

Bài 2

Cho một ngăn xếp được biểu diễn bằng mảng. Hãy viết các hàm thực hiện các công việc sau đây:

1. Lấy ra phần tử ở đáy ngăn xếp

2. Lấy ra phần tử ở vị trí thứ k tính từ đỉnh ngăn xếp, với k là số nguyên dương
3. Thêm phần tử vào đáy ngăn xếp
4. Thêm phần tử vào vị trí thứ k tính từ đỉnh ngăn xếp, với k là số nguyên dương
5. Chuyển đổi một số từ hệ thập phân sang hệ nhị phân, bát phân (octal), thập lục phân (hexa)

Bài 3

Cho một hàng đợi được cài đặt bằng con trỏ. Hãy viết các hàm thực hiện các công việc sau đây:

1. Lấy ra phần tử ở cuối hàng đợi
2. Lấy ra phần tử ở vị trí thứ k tính từ đầu hàng đợi, với k là số nguyên dương.
3. Thêm phần tử vào đầu hàng đợi
4. Thêm phần tử vào vị trí thứ k tính từ đầu hàng đợi, với k là số nguyên dương.
5. Đảo ngược thứ tự các phần tử trong hàng đợi

Chương 5

Mô hình dữ liệu cây

Khác với mô hình danh sách các phần tử trong danh sách có mối quan hệ tuyến tính thì sang mô hình cây chúng ta thấy một mô hình có mối quan hệ hoàn toàn khác đó chính là mối quan hệ phân cấp. Mô hình này sử dụng nhiều trong đời sống và nó được áp dụng vào biểu diễn dữ liệu.

5.1 Cây tổng quát - cây đa phân - gọi tắt là cây

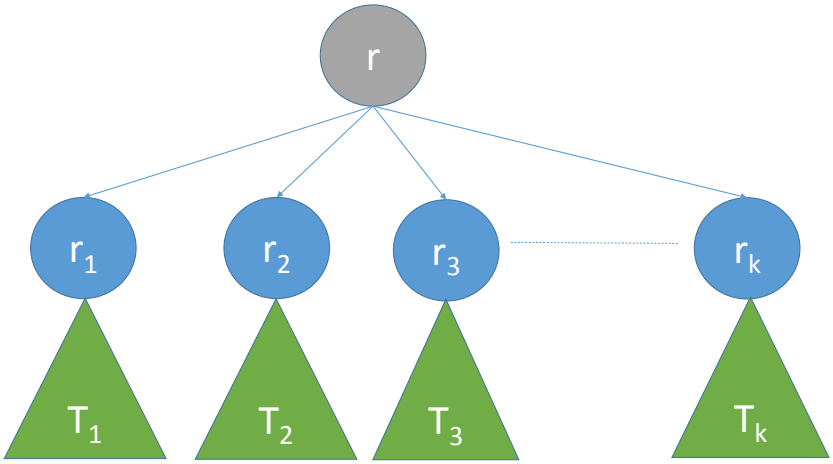
5.1.1 Định nghĩa cây và các khái niệm cơ bản trên cây

a) Định nghĩa cây

Có nhiều cách định nghĩa cây, sau đây ta định nghĩa bằng đệ quy như sau:

1- Tập có một đỉnh là một cây, cây này có gốc và lá là đỉnh duy nhất đó.

2- Giả sử $T_1, T_2, ..., T_k (k >= 1)$ là các cây có gốc tương ứng là $r_1, r_2, ..., r_k$, các cây $T_1, T_2, ..., T_k$ đôi một không giao nhau. Nếu r là một đỉnh không thuộc các cây $T_1, T_2, ..., T_k$ và có quan hệ phân cấp (quan hệ cha - con) với các đỉnh $r_1, r_2, ..., r_k$ thế thì tập T gồm đỉnh r và tất cả các đỉnh của các cây T_i làm thành một cây mới có gốc r. Ta quy ước viết các đỉnh ở mức trên là các đỉnh cha, mức dưới là con (xem Hình 5.1).



Hình 5.1: Định nghĩa cây tổng quát

Ví dụ về cấu trúc cây: Cách tổ chức bộ nhớ ngoài trong máy tính được tổ chức theo mô hình cây (gọi là cây thư mục).

Trong mô hình cây chứa tập hợp các đỉnh, giữa các đỉnh có một quan hệ phân cấp gọi là quan hệ cha - con. Ví dụ như hình 5.1: r gọi là cha của $r_1, r_2, ..., r_k$.

Nếu tập các đỉnh = rỗng => Cây là cây rỗng

Số các cây con ở mỗi đỉnh được gọi là bậc của đỉnh đó. Đỉnh có bậc bằng 0 gọi là lá (đỉnh tận cùng), đỉnh không là lá gọi là đỉnh trong.

Các đỉnh có cùng cha gọi là anh em, xét từ trái sang phải

Tập các cây con phân biệt người ta gọi là Rừng

Gốc của cây có mức 0 (level = 0). Nếu đỉnh cha có mức i thì các đỉnh con của nó sẽ có mức là $i + 1$

Chiều cao của cây (height) của cây là số mức lớn nhất của đỉnh có trên cây.

Một dãy các đỉnh $a_1, a_2, ..., a_n (n > 0)$ sao cho a_i có quan hệ với a_{i+1} gọi là một đường đi từ a_1 đến a_n với độ dài $n - 1$. Luôn tồn tại đường đi từ gốc đến một đỉnh bất kỳ trong cây.

Cây được sắp: Là cây mà các đỉnh trong cây được sắp xếp theo thứ tự nào đó. Nếu đỉnh a có các đỉnh con $b_1, b_2, ..., b_m$ theo thứ tự này thì ta nói b_1 là con trưởng và b_2 là em liền kề của nó.

5.1.2 Các phép toán cơ bản trên cây

Xét cây gốc T:

1) Hàm $parent(k, T)$ trả về nút cha của nút k trên cây T, nếu k là nút gốc thì hàm cho giá trị rỗng. Trong cài đặt cụ thể thì giá trị rỗng là một giá trị đặc biệt nào đó do ta chọn, nó phụ thuộc vào cấu trúc dữ liệu mà ta dùng để cài đặt cây.

2) Hàm $eldestChild(k, T)$: tìm con cả của nút k trên cây T, nếu k là lá thì hàm cho giá trị rỗng, ngược lại thì hàm sẽ trả về con trưởng của nút k

3) Hàm $nextSibling(k)$: tìm em liền kề của nút k trên cây T, nếu k không có em liền kề thì hàm trả về giá trị rỗng.

5.1.3 Các phép toán thăm (duyệt) cây

Duyệt cây: Là phép thăm các đỉnh trên cây, sao cho mỗi đỉnh chỉ được thăm duy nhất một lần.

Xét ba phương pháp duyệt cây cơ bản:

- Duyệt cây theo thứ tự trước – *preOrder*
- Duyệt cây theo thứ tự giữa – *inOrder*
- Duyệt cây theo thứ tự sau - *postOrder*

Giả sử xét cây gốc T, các phương pháp duyệt tương ứng là:

Duyệt theo tứ tự trước (preOrder)

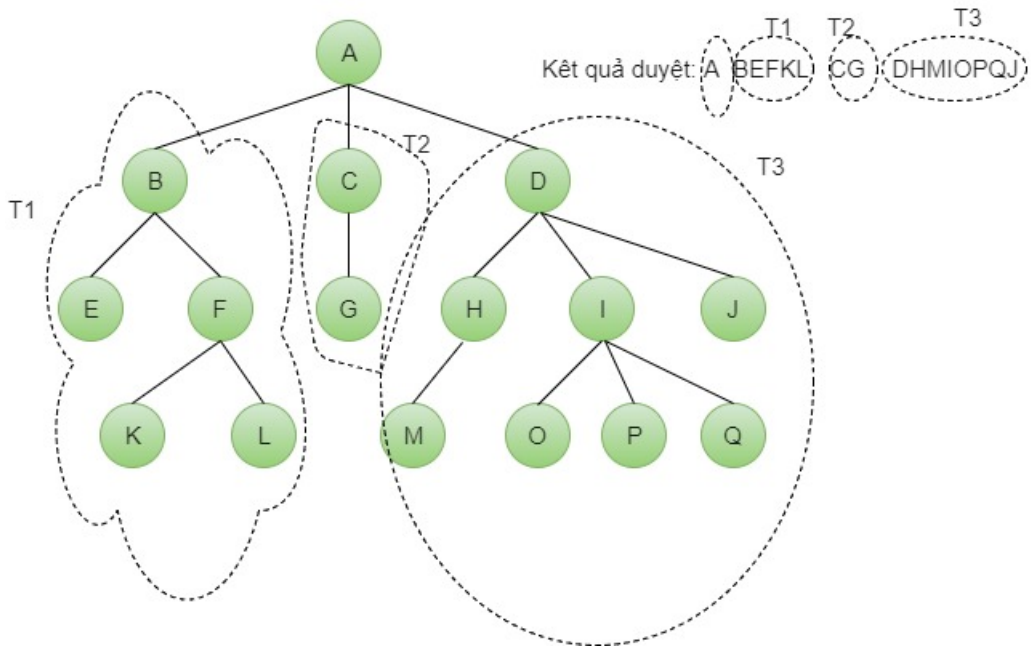
Nguyên tắc duyệt cây:

- Nếu cây T rỗng: Ta không làm gì cả
- Nếu T khác rỗng: Trình tự thăm các đỉnh trên cây như sau:

1) Thăm gốc T

2) Thăm các cây con: $T_1, T_2, ..., T_k$ của T theo thứ tự trước

Ví dụ: Xét cây (xem Hình 5.2)



Hình 5.2: Thứ tự duyệt cây theo cách duyệt trước

Chương trình có dạng như sau:

```
void PreOrder (Tree T)
```

```
{
    NodeType C;
   if (T ==rong) printf("cay rong");
   else
       {
           Visit (T);//  tham goc hay in goc
           C = eldestChild (T); // con truong
         while (C!=rong)
           {
              PreOrder (C);
              C = nextSibling (C);// em lien ke
           }
        }
 }
```

Duyệt theo thứ tự giữa (inOrder Tree)

Nguyên tắc duyệt

- Nếu cây T rỗng: Ta không làm gì cả

- Nếu T khác rỗng: Trình tự thăm các đỉnh trên cây như sau:

1) Thăm cây con $T1$ của T theo thứ tự giữa

2) Thăm gốc T

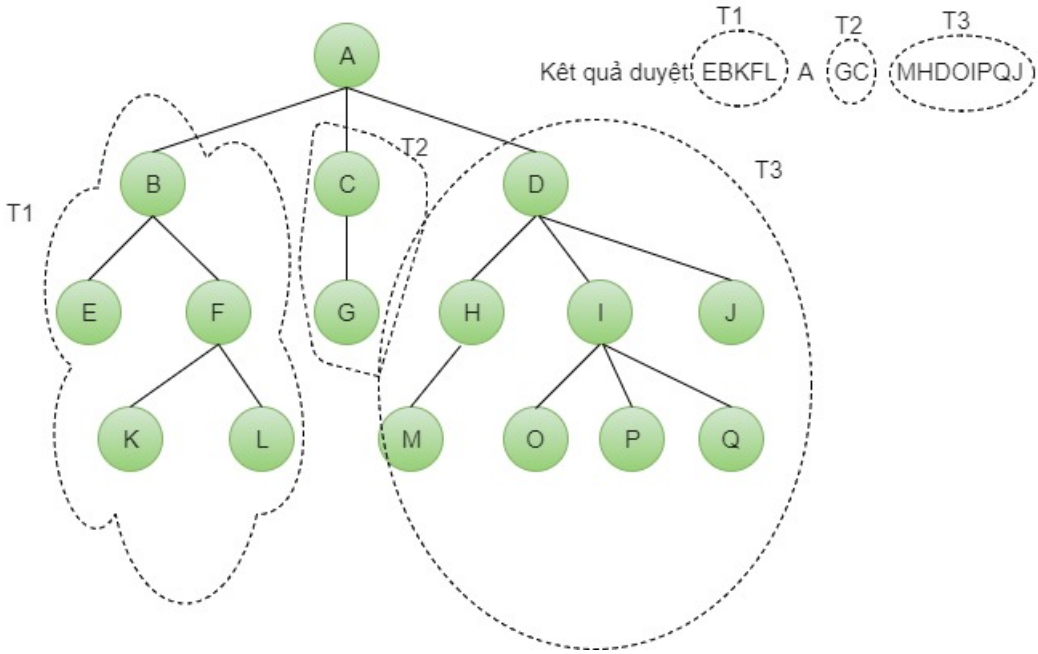
3) Thăm các cây con còn lại của T theo thứ tự giữa: $T_2, ..., T_k$

Ví dụ: Xét cây (xem Hình 5.3)

Chương trình có dạng như sau:

```
void InOrder (Tree T)
{   nodeType C;
   if (T==rong) printf ("Cay rong");
   else
        {
           C = eldestChild (T);// tim con truong
         if (C!= rong)
              {
                InOrder(C);
                C = nextSibling(C);//tim em lien ke
             }
         Visit(T);  //tham goc
         while (C!=rong)
           {
            InOrder (C);
            C = nextSibling(C);//em lien ke
           }
     }
```

```
}
```

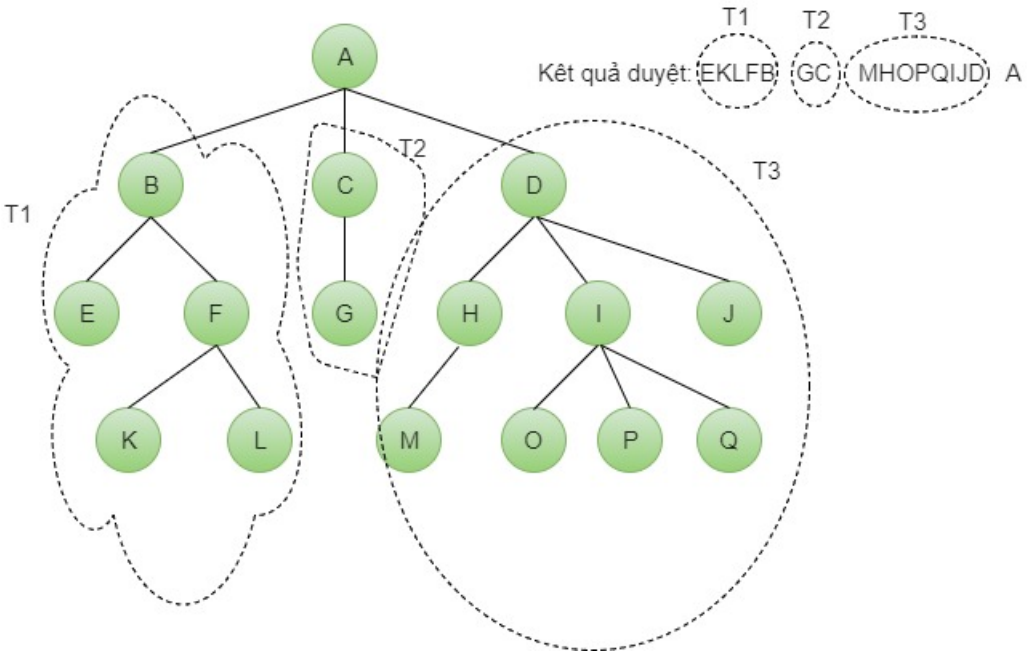


Hình 5.3: Thứ tự duyệt cây theo cách duyệt giữa

Duyệt theo thứ tự sau

Nguyên tắc duyệt cây theo thứ tự sau:
- Nếu cây T rỗng: Ta không làm gì cả
- Nếu T khác rỗng: Trình tự thăm các đỉnh trên cây như sau:
1) Thăm các cây con: $T_1, T_2, ..., T_k$ của T theo thứ tự sau
2) Thăm gốc T
Ví dụ: Xét cây (xem Hình 5.4)
Chương trình có dạng như sau:

```
void PostOrder (Tree T)
{
nodeType C;
if (T==rong) printf("cay rong");
else
    {
      C = eldestChild(T);// con truong
```



Hình 5.4: Thứ tự duyệt cây theo cách duyệt sau

```
      while (C!= rong)
            {
               PostOrder(C);
               C = nextSibling(C);// em lien ke
            }
         Visit(T);  //Tham cha hay in cha
   }
}
```

5.1.4 Biểu diễn cây trên máy tính

Để cài đặt cây trong máy tính chúng ta xét các phương pháp biểu diễn cây sau đây:

+ Biểu diễn cây bằng danh sách các con của mỗi đỉnh

+ Biểu diễn cây bằng cấu trúc móc nối các nút (hay nói cách khác là cài đặt cây bằng con trỏ)

+ Biểu diễn cây bởi con trưởng và em liền kề của mỗi đỉnh (còn gọi là phương pháp chuyển cây đa phân về cây nhị phân)

+ Biểu diễn cây bởi cha của mỗi đỉnh

Với mỗi phương pháp ta đều có thể sử dụng mảng hoặc con trỏ hoặc kết hợp giữa mảng và con trỏ để biểu diễn

5.1.4.1 Biểu diễn cây bằng danh sách các con của mỗi đỉnh

Mỗi đỉnh trong cây đi kèm với một danh sách các con của nó. Việc tổ chức lưu trữ các phần tử trong danh sách các con và lưu trữ các đỉnh của cây như thế nào là phụ thuộc vào các đặc trưng và yêu cầu của bài toán cụ thể.

Sử dụng mảng để biểu diễn cây

Phương pháp này được thực hiện như sau: - Đánh số các đỉnh trên cây. Tùy từng ngôn ngữ có thể đánh số từ 0 hay 1 cho đến hết các nút trên cây.

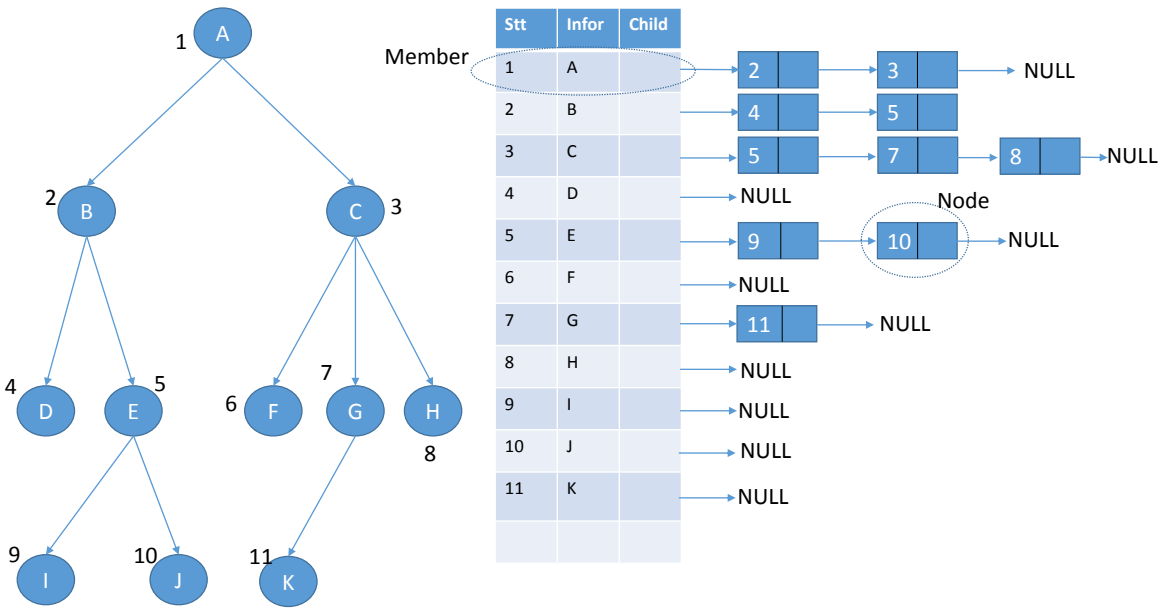
- Với mỗi nút chúng ta đi liệt kê các con của nó.

Ví dụ: Xét cây, các đỉnh trên cây được đánh số thứ tự từ trên xuống, từ trái qua phải và hình ảnh cây sau khi biểu diễn với dạng $Tree1$ (xem hình 5.5)

Trong hình 5.5 mỗi nút trên cây có dạng $Member$. $Member$ là một cấu trúc gồm hai thành phần $Infor$ và $Child$. Trong đó $Infor$ chứa dữ liệu của một nút trên cây, $Child$ là một con trỏ trỏ đến danh sách các con của nút đang xét.

Biểu diễn cây:

```
typedef struct Member
{
   int Infor;
   struct node* Child;
}Member;

typedef struct node
{
   int ID;
   node *Next;
}node;
```



Hình 5.5: Biểu diễn cây bằng danh sách con của mỗi đỉnh

```
\\cay luc nay ban chat la mang cac Member

typedef struct Tree
{
   Member elems[N];
   int size;
}Tree;
```

Cây sau khi biểu diễn trên máy tính được xem như một mảng các danh sách liên kết đơn. Thực hiện các thao tác trên cây là thực hiện các thao tác trên mảng các danh sách này

Các phép toán tác động trên cây

1 - Cho một đỉnh thứ k trên cây T, tìm con trưởng của nó

Với cách cài đặt này phép toán tìm còn trưởng được xác định rất dễ dàng thông qua thành phần **Child** của nút thứ k trên cây.

Chương trình cụ thể như sau:

```
int eldestChild(int k, Tree T)
  {
    if (T.elems[k].Child != NULL)
      return( T.elems[k].Child->ID);
    else return -1;
  }
```

2 - Cho một đỉnh thứ k trên cây T, Tìm cha của nó.

Phép toán này được mô tả như sau: Duyệt các nút trên cây. Tại mỗi nút duyệt các con của nó. Nếu con của nó có chứa nút k thì kết luận nút đang xét là cha của k. Ngược lại nếu không có con nào thì chuyển sang nút tiếp.

Ví dụ: Hình 5.5 Tìm cha của nút thứ 7 trên cây. Với cách cài đặt như trên ta thực hiện như sau: Tại nút số 1, duyệt các con của nó thấy các con của nó là 2 và 3 khác k (k=7)=> chuyển sang nút số 2. Tại nút số 2 có các con là 4 và 5 (khác k) => chuyển sang nút số 3. Tại nút k có con là 5, 7 và 8. Dừng và kết luận cha là 3.

Phép toán như sau:

```
int  Parent(int k, Tree T)
  {   int i;
    struct node* p;
    i = 0;
    while (i< T.size)
      {
           p=T.elems[i].Child;
        while (p!=NULL)
             if (p->ID== k)
               return(i);
            else p = p-> Next;
        i ++;
      }
    return(-1);
  }
```

5.1.4.2 Biểu diễn cây bằng con trỏ

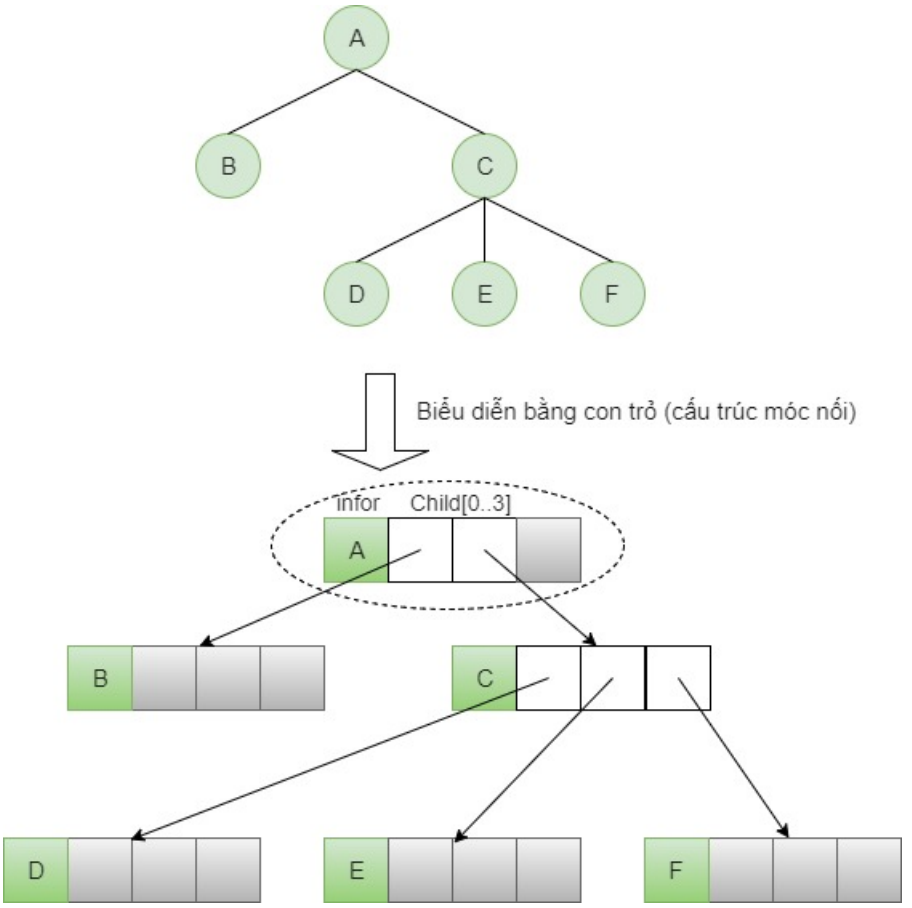
Mô tả dạng biểu diễn

- Các đỉnh của cây được lưu trữ ở những ô nhớ không kế tiếp

nhau, liên kết với nhau thông qua cơ chế lưu địa chỉ

- danh sách các con của mỗi đỉnh có thể được lưu trữ trong mảng hoặc danh sách móc nối. Nếu ta biết trước số con tối đa của mỗi đỉnh trên cây, ta có thể dùng mảng để lưu danh sách các con này.

Ví dụ: Xét cây và hình ảnh cây sau khi biểu diễn bằng con trỏ được mô tả trong hình 5.6:



Hình 5.6: Biểu diễn cây bằng móc nối các nút qua con trỏ

Trong hình 5.6 giả sử số con tối đa của mỗi đỉnh là k (có thể xem là cây k phân), mỗi đỉnh/nút là một cấu trúc gồm 2 trường:

+ Trường $Infor$: Lưu dữ liệu của đỉnh

+ Trường $Child$: Là một mảng chứa tối đa k con trỏ, mỗi con

trỏ trỏ tới một gốc cây con

Nhận xét:

- Để quản lý cây người ta dùng 1 con trỏ luôn trỏ tới gốc cây, giả sử con trỏ $ROOT$

- Cây rỗng khi $ROOT = null$ (trỏ tới rỗng/không trỏ tới đâu cả). Để truy cập đến các đỉnh trên cây chúng ta truy cập tuần tự, xuất phát từ gốc cũng giống như trong thực tế, ta leo lên cây bao giờ cũng leo từ gốc.

Dạng cài đặt:

```
#define K 100; //{K la bac cua cay}
typedef struct Node
{
     int Info;
     struct Node *childs[K] ; //mang cac con
} Node;
typedef struct Node* Tree;
```

Cài đặt các phép toán cơ bản xem như bài tập dành cho độc giả

5.1.4.3 Biểu diễn cây bằng con trưởng và em liền kề của mỗi đỉnh

Mô tả phương pháp biểu diễn:

Mỗi đỉnh trên cây cần lưu trữ 3 thông tin:

- Dữ liệu của đỉnh ($infor$)
- Con trưởng của đỉnh ($EldestChild$): Lưu ID
- Em liền kề của đỉnh ($NextSibling$): Lưu ID

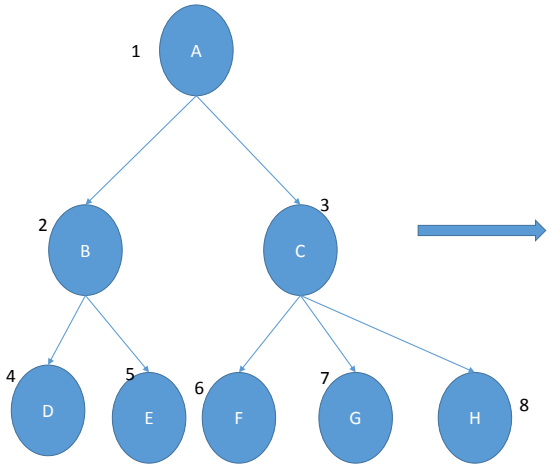
Câu hỏi: Tại sao $EldestChild$ và $NextSibling$ lại lưu ID mà không lưu $infor$?

Mỗi đỉnh, còn gọi là một nút có thể xem như một cấu trúc gồm 3 trường: $infor, EldestChild, NextSibling$. Các đỉnh trên cây được tổ chức lưu trữ kế tiếp trong 1 mảng gồm tối đa n phần tử hoặc lưu trữ dưới dạng móc nối. Ta lần lượt xét 2 dạng biểu diễn này

Biểu diễn cây bởi mảng :

- Các đỉnh trên cây được tổ chức lưu trữ kế tiếp trong 1 mảng gồm tối đa n phần tử. Mỗi đỉnh được tổ chức lưu trữ trong 1 cấu trúc gọi là nút được mô tả ở trên

Ví dụ: Xét cây và hình ảnh cây sau khi biểu diễn hình 5.7



stt	Eldest Child	Infor	NextSibling
1	2	A	-1
2	4	B	3
3	6	C	-1
4	-1	D	5
5	-1	E	-1
6	-1	F	7
7	-1	G	8
8	-1	H	-1

Node

Hình 5.7: Biểu diễn cây qua con trưởng và em liền kề sử dụng cấu trúc kế tiếp

Nhận xét

- Cây sau khi biểu diễn được xem như là một mảng các nút
- Thao tác trên cây là thao tác trên mảng các nút này (coi như bài tập)

Dạng cài đặt

```
typedef struct Node
{
   Item Info; // Infor chua du lieu co kieu Item
   int EldestChild ;// ID cua con truong
   int NextSibling ; //ID cua em lien ke
}Node;
typedef struct Tree
{
  struct Node Elems[N];
  int size;
}Tree;
```

Cài đặt các phép toán cơ bản

Với cách cài đặt này ta thấy dễ dàng tìm con trưởng và em liền kề của đỉnh thứ k trên cây T – xem như bài tập dành cho bạn đọc, ở đây ta xét phép toán tìm cha của đỉnh k trên cây T. Để tìm cha của đỉnh thứ k trên cây T, ta làm như sau:

Duyệt lần lượt các đỉnh trên cây, xuất phát từ gốc ($i = 1$), kiểm tra xem con trưởng j của i có bằng k hay không? Nếu có bằng nhau thì i chính là cha cần tìm, ngược lại, kiểm tra em liền kề của con trưởng j xem có bằng k hay không, nếu bằng nhau thì i cũng chính là cha cần tìm, nếu không kiểm tra j là em liền kề của j cũ, cứ tiếp tục như vậy cho đến khi kiểm tra hết các con của i mà không có con nào bằng k thì duyệt đỉnh i tiếp theo trên cây, ngược lại chỉ cần một con bất kì của i mà bằng k thì ta dừng việc tìm kiếm và kết luận i chính là đỉnh cha cần tìm.

Viết chương trình coi như bài tập.

Biểu diễn cây bởi con trỏ

Mô tả dạng biểu diễn: Mỗi đỉnh trên cây cần lưu trữ 3 thông tin:

• Dữ liệu của đỉnh ($infor$)

• Con trưởng của đỉnh ($EldestChild$): trỏ tới gốc cây con trưởng

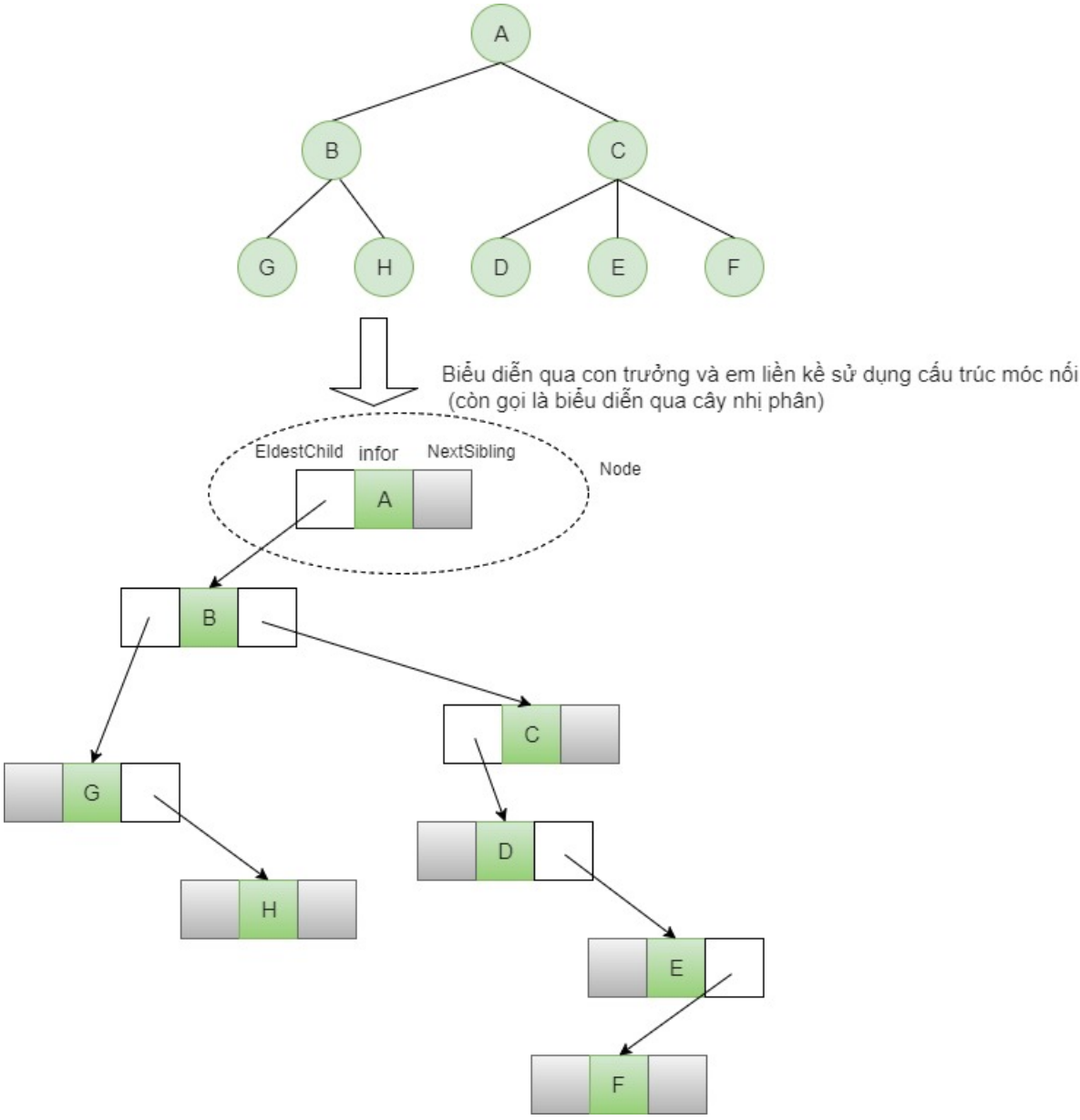
• Em liền kề của đỉnh ($NextSibling$): Trỏ tới gốc cây em kề

$=>$ Mỗi đỉnh có thể xem như một cấu trúc gồm 3 trường: $infor, eldestChild, NextSibling$.

- Các đỉnh trên cây được tổ chức lưu trữ móc nối thông qua cơ chế lưu địa chỉ.

Ví dụ: Xét cây và hình ảnh cây sau khi biểu diễn (xem hình 5.8)

Với cách cài đặt này, để quản lý cây, người ta dùng một con trỏ luôn trỏ tới gốc cây, giả sử con trỏ $Root$. Khi đó cây rỗng khi $Root = NULL$. Để truy cập đến các đỉnh trên cây ta truy cập tuần tự xuất phát từ gốc. Cách biểu diễn này là một cách chuyển



Hình 5.8: Biểu diễn cây qua con trưởng và em liền kề cài đặt bằng con trỏ

cây đa phân về cây nhị phân tương ứng.

5.1.4.4 Biểu diễn cây bởi cha của mỗi đỉnh

Cách biểu diễn:

- Đánh số các đỉnh trên cây (có thể bắt đầu từ 0 hoặc 1 tùy ngôn ngữ lập trình).

- Với mỗi đỉnh lưu 2 trường thông tin: $infor$ và $parent$. Trong đó $infor$ là dữ liệu của đỉnh, $parent$ là cha của đỉnh đó (thể hiện qua ID của cha)

- Cây lúc này là tập các đỉnh như trên.

Cài đặt như sau:

```
#define N 100 ;//phan tu toi da cua cay
typedef struct Node
   {
       Item info;
      int parent;
   }Node;
typedef struct Tree
{
    struct Node elems[N];
    int size;
}Tree;
```

Cách biểu diễn này được mô tả qua ví dụ tại Hình 5.9

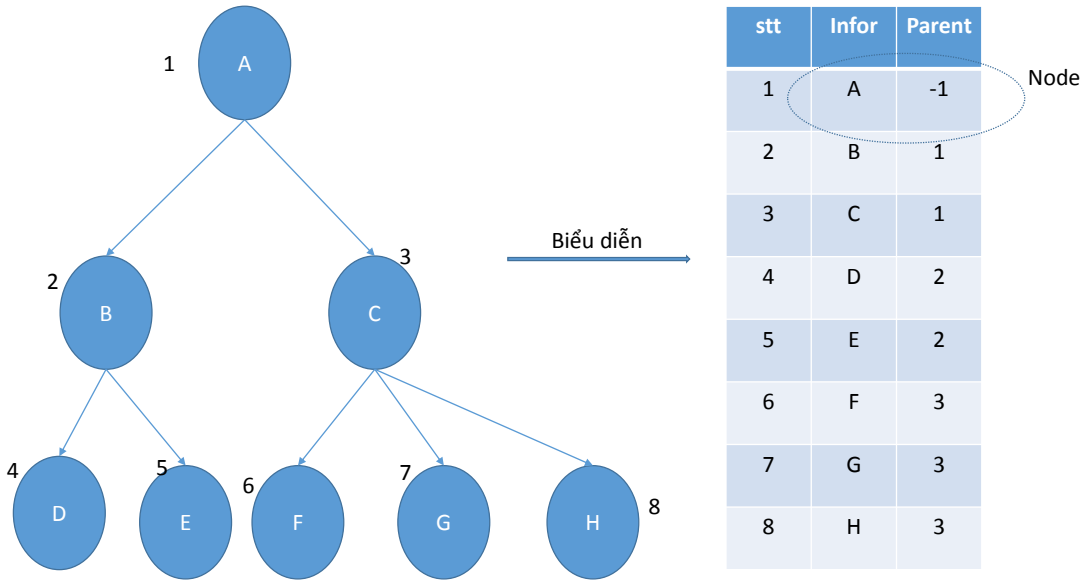
5.2 Cây nhị phân

5.2.1 Định nghĩa và phân loại cây nhị phân

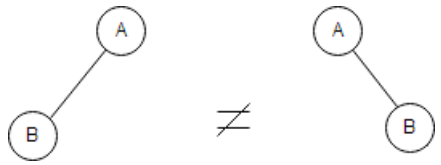
Cây nhị phân là cây mà mỗi nút có tối đa hai nút con. Trong đó các nút con của cây được phân biệt thứ tự rõ ràng, một nút con gọi là nút con trái và một nút con gọi là nút con phải.

Ví dụ 5.2.1.1: Xét 2 cây nhị phân hình 5.10. Đây là hai cây nhị phân sau khác nhau, một cây có một con và là con trái trong khi cây còn lại cũng có một con nhưng là cây con phải.

Các dạng cây nhị phân phổ biến:



Hình 5.9: Biểu diễn cây qua cha



Hình 5.10: Cây nhị phân phân biệt thứ tự

- Cây nhị phân suy biến gồm: cây lệch trái, cây lệch phải, cây zic-zắc.

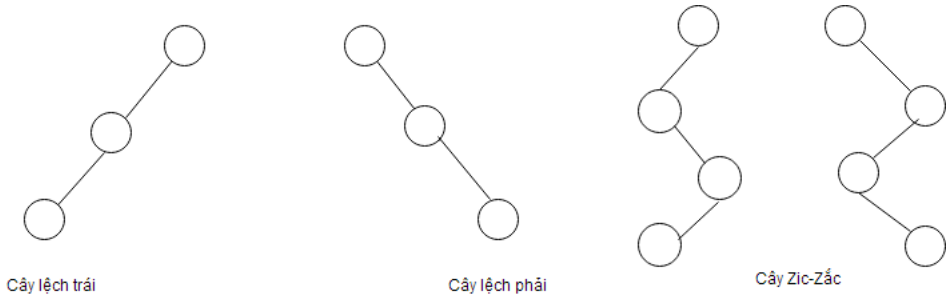
Ví dụ 5.2.1.2 cây nhị phân suy biến (hình 5.11)

- Cây nhị phân hoàn chỉnh (Complete Binary Tree): là cây nhị phân mà các nút ở tất cả các mức đều có 2 con có thể trừ mức cuối cùng.

- Cây nhị phân đầy đủ (Full Binary Tree): Là cây nhị phân trong đó tất cả các nút đều có 2 con trừ nút lá.

- Cây nhị phân hoàn hảo (Perfect Binary Tree): là cây nhị phân trong đó tất cả các nút trong đều có 2 con và tất cả các nút lá đều ở cùng một mức.

Ví dụ 5.2.1.3 cây nhị phân hoàn chỉnh, cây nhị phân đầy đủ và cây nhị phân hoàn hảo (hình 5.12)



Hình 5.11: Các cây nhị phân đặc biệt

Như vậy: Trong cây nhị phân có cùng số đỉnh thì cây nhị phân suy biến có chiều cao lớn nhất và cây nhị phân hoàn hảo có chiều cao nhỏ nhất. Với cây nhị phân hoàn hảo thì số lượng tối đa các đỉnh ở mức i là 2^i ,số lượng tối đa các đỉnh trên cây có chiều cao h là: $2^{h+1} - 1$.

5.2.2 Biểu diễn cây nhị phân trên máy tính

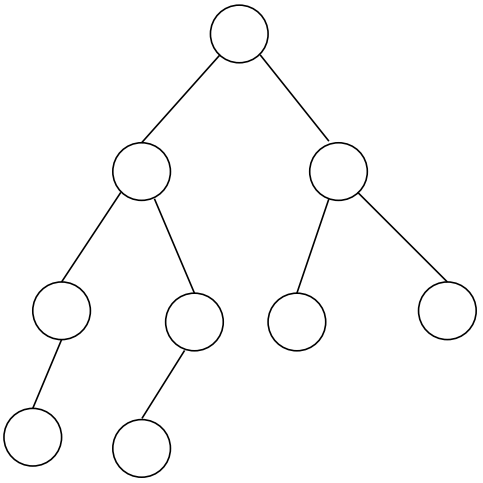
a) Biểu diễn cây nhị phân bằng mảng

Mô tả dạng biểu diễn: với cách biểu diễn bằng mảng thì cây sau khi biểu diễn được xem như một mảng các nút (đỉnh). Trong đó mỗi nút (đỉnh) là một cấu trúc gồm 3 trường: $data, left, right$ trong đó:

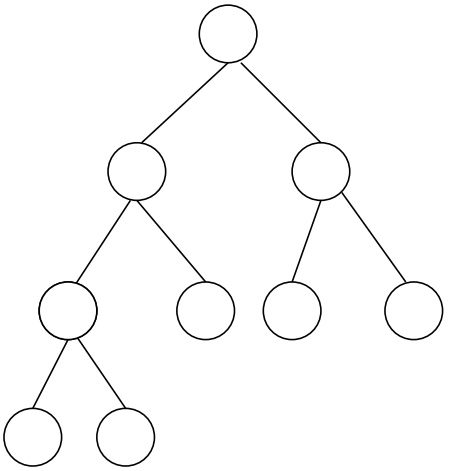
$data$: lưu giá trị của đỉnh

$left$: con trái của đỉnh (lưu ID)

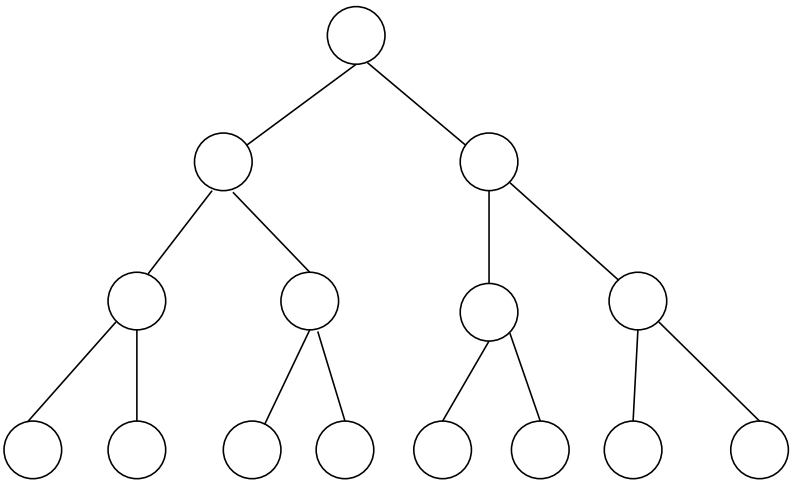
$right$: con phải của đỉnh (lưu ID)



Cây nhị phân hoàn chỉnh



Cây nhị phân đầy đủ



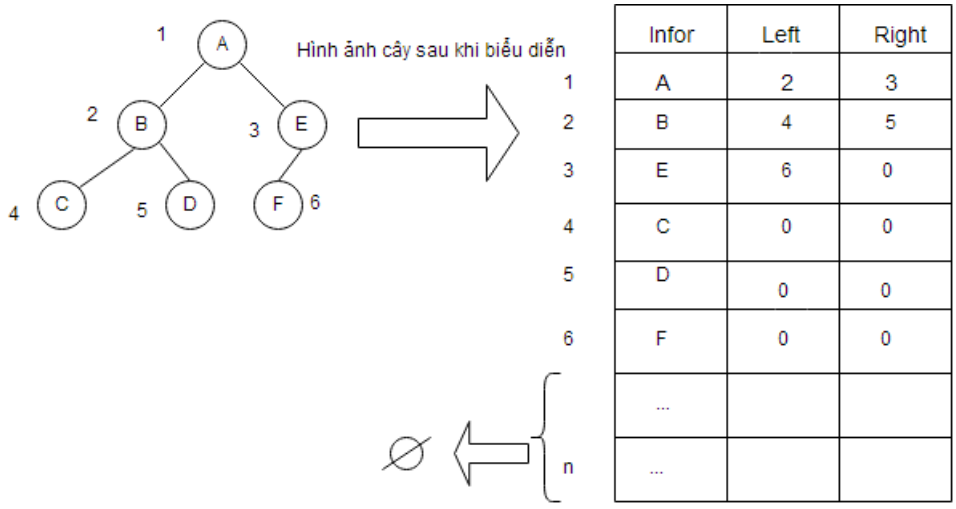
Cây nhị phân hoàn hảo

Hình 5.12: Cây nhị phân hoàn chỉnh, cây nhị phân đầy đủ và cây nhị phân hoàn hảo

Thực hiện các thao tác trên cây là thực hiện các thao tác trên mảng này.

Ví dụ 5.2.2.1 xét cây nhị phân và hình ảnh biểu diễn cây sau khi

cài đặt trên máy tính bằng mảng (hình 5.13).



Hình 5.13: Biểu diễn cây nhị phân qua chỉ số con trái và con phải

Cú pháp biểu diễn:

```
#include<stdio.h>
#define N 100
// dinh nghia mot Node tren cay
typedef struct Node
{
   item data;
   int left,right;
}Node;
// cay nhi phan ban chat la mot mang cac Node
typedef struct bTree1
{
   Node elems[N];
   int size;
}bTree1;
```

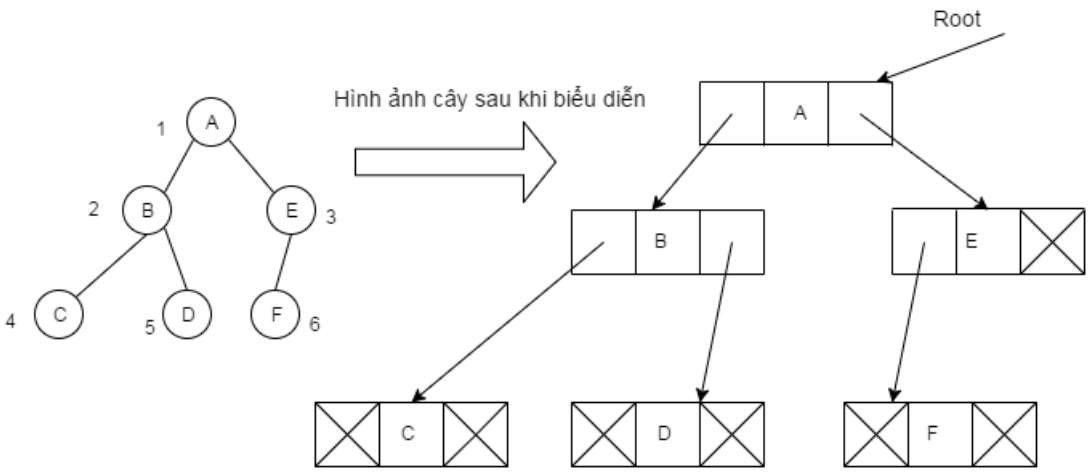
Ta thấy trong cây nhị phân thường đánh số thứ tự - *id* cho các đỉnh (chỉ danh) cho các đỉnh trên cây và chỉ danh của mỗi đỉnh là xác định duy nhất cho đỉnh đó. Chúng ta có thể đánh chỉ danh cho các đỉnh trên cây theo thứ tự nào đó do chúng ta quy định, tuy nhiên nên đánh theo thứ tự: nếu đỉnh cha có thứ tự là i, thì 2 con trái, phải của nó sẽ có chỉ danh là $2i$ và $2i+1$ để tiện cho việc truy cập đến các đỉnh trên cây, nghĩa là từ đỉnh con có thể truy cập đến

đỉnh cha và ngược lại.

b) Biểu diễn cây nhị phân bằng con trỏ

Với dạng biểu diễn này, để quản lý cây người ta sử dụng một con trỏ luôn trỏ tới gốc cây, ta có thể truy cập đến các đỉnh trên cây tuần tự xuất phát từ gốc cây, giả sử con trỏ T. Khi đó, cây rỗng khi $T = null$

Ví dụ 5.2.2.2 xét cây nhị phân và hình ảnh biểu diễn cây sau khi cài đặt trên máy tính (hình 5.14).



Hình 5.14: Biểu diễn cây nhị phân bằng cách móc nối các nút sử dụng con trỏ

Cú pháp biểu diễn:

```
#include<stdio.h>
// dinh nghia mot Node tren cay
typedef struct Node
{
   item data;
   Node *left,*right;
}Node;
Node *bTree2;
```

5.2.3 Các thao tác trên cây nhị phân

Chúng ta có thể lựa chọn một trong hai cách biểu diễn cây bởi mảng hoặc con trỏ để cài đặt các phép toán trên cây nhị phân. Giả sử cây nhị phân được cài đặt bằng con trỏ, chúng ta lần lượt cài đặt một số phép toán cơ bản trên cây nhị phân như sau:

1 - Tạo cây rỗng

Cây rỗng là một cây là không chứa một nút nào cả. Như vậy khi tạo cây rỗng ta chỉ cần cho con trỏ quản lý gốc của cây (T) trỏ tới giá trị $NULL$.

Đoạn mã chương trình

```
void makeNullTree(bTree2 *T)
{
    (*T)=NULL;
}
```

2 - Kiểm tra cây rỗng

Để kiểm tra cây có phải là cây rỗng hay không ta chỉ cần kiểm tra xem trên cây có nút nào không, nếu không có thì trả về $true$, ngược lại thì trả về $false$. Ở đây chúng ta kiểm tra xem biến T có bằng $NULL$ hay không.

Đoạn mã chương trình:

```
int emptyTree(bTree2 T)
{
    return T==NULL;
}
```

3 - Xác định con trái của một nút

Để xác định con trái của một nút bất kỳ trên cây đầu tiên ta kiểm tra xem cây có rỗng hay không, nếu cây khác rỗng thì trả về nút con trái của nút cần tìm còn nếu cây rỗng thì trả về giá trị $NULL$.

Đoạn mã chương trình:

```
bTree2 leftChild(bTree2 T)
{
    if (T!=NULL) return T->left;
    else return NULL;
}
```

4 - Xác định con phải của một nút thứ p

Tương tự như hàm $leftChild$ ở trên, để xác định con phải của một nút bất kỳ trên cây đầu tiên ta kiểm tra xem cây có rỗng hay không, nếu cây khác rỗng thì trả về nút con phải của nút cần tìm còn nếu cây rỗng thì trả về giá trị $NULL$.

Đoạn mã chương trình:

```
bTree2 rightChild(bTree2 T)
{
    if (T!=NULL) return n->right;
    else return NULL;
}
```

5 - Kiểm tra nút lá

Nếu một nút là nút lá thì nó không có con, khi đó con trái và con phải của nó cùng bằng NULL.

Đoạn mã chương trình:

```
int isLeaf(bTree2 T)
{
    if(T!=NULL)
       return(leftChild(T)==NULL)&&(rightChild(T)==NULL);
    else return NULL;
}
```

6 – Xác định số nút của cây

Để xác định số nút của cây ta thực hiện như sau: kiểm tra cây, nếu cây rỗng thì không có nút nào trên cây, nếu cây khác rỗng thì trả về số nút trên cây bằng 1 (nút gốc) cộng với số nút của cây con bên trái cộng với số nút của cây con bên phải.

Đoạn mã chương trình:

```
int numberNodes (bTree2 T)
```

```
{
    if(emptyTree(T)) return 0;
    else return 1+ numberNodes (leftChild(T))+ numberNodes (rightChild(T));
}
```

7 - Tạo cây mới từ hai cây có sẵn

Cho hai cây con l, r và x là giá trị bất kỳ. Hãy tạo một cây nhị phân có gốc lưu x và l, r là hai cây con trái, phải của gốc này

Cách giải:

- Yêu cầu máy tính cấp phát ô nhớ để làm gốc của cây
- Đổ x vào ngăn data của ô nhớ vừa cấp phát
- Gắn l, r tương ứng vào nhánh trái, phải của gốc

Thuật toán:

```
bTree2 create2(Tdata v, bTree2 l, bTree2 r)
{
    bTree2 N;
    N=(TNode*)malloc(sizeof(TNode));
    N->Data=v;
    N->left=l;
    N->right=r;
    return N;
}
```

8- Tìm nút có nội dung là x trên cây nhị phân

Để tìm nút có nội dung x trên cây nhị phân ta tiến hành kiểm tra từ nút gốc, nếu nút gốc có nội dung là x thì nút gốc chính là nút cần tìm. Nếu nút gốc bằng $NULL$ (cây rỗng) thì không có nút nào có nội dung là x trên cây. Nếu nội dung nút gốc khác x và nút gốc khác $NULL$ thì ta lần lượt thực hiện phép toán tìm kiếm trên nhánh cây con bên trái và cây con bên phải của cây đó.

Thuật toán

```
bTree2 search (bTree2 tree, int x)
{
    bTree2 p;
    if (tree->data == x)
        return tree;
    if (tree == NULL)
```

```
        return NULL;
    p = search(tree->left, x);
    if (p == NULL)
        search(tree->right, x);
}
```

9 - Các phép duyệt cây

Tương tự như cây tổng quát, cây nhị phân cũng có 3 phương pháp duyệt cây đó là duyệt cây theo thứ tự trước, duyệt cây theo thứ tự giữa và duyệt cây theo thứ tự sau. Ta có thể áp dụng các phép duyệt cây tổng quát để duyệt cây nhị phân. Tuy nhiên vì cây nhị phân là cấu trúc cây đặc biệt nên các phép duyệt cây nhị phân cũng đơn giản hơn.

Duyệt cây theo thứ tự trước (NLR):

- Nguyên tắc duyệt cây theo thứ tự trước là duyệt nút gốc sau đó duyệt cây con trái theo thứ tự trước rồi duyệt cây con phải theo thứ tự trước.

- Thuật toán:

Nếu cây rỗng thì ta không làm gì cả

Nếu cây khác rỗng: thì các bước tiến hành như sau

+ Duyệt gốc

+ Duyệt cây con bên trái theo thứ tự trước

+ Duyệt cây con bên phải theo thứ tự trước

Hàm duyệt theo thứ tự trước:

```
void preOrder(bTree2 * r)
{
    if (r != NULL)
  {
        printf("%c", r->data);
        preOrder(r->lefltChild);
        preOrder(r->rightChild);
  }
}
```

Duyệt cây theo thứ tự giữa (LNR):

- Nguyên tắc duyệt cây theo thứ tự giữa là duyệt cây con trái theo thứ tự giữa rồi duyệt nút gốc sau đó duyệt cây con phải theo thứ tự giữa.

- Thuật toán:

Nếu cây rỗng thì ta không làm gì cả

Nếu cây khác rỗng: thì các bước tiến hành như sau

+ Duyệt cây con trái theo thứ tự giữa

+ Duyệt gốc

+ Duyệt cây con phải theo thứ tự giữa

Hàm duyệt theo thứ tự giữa:

```
void inOrder(bTree2 * r)
{
if (r != NULL)
   {
        inOrder (r->leftChild);
           printf("%c", r->data);
        inOrder (r->rightChild);
   }
}
```

Duyệt cây theo thứ tự sau (LRN): - Nguyên tắc duyệt cây theo thứ tự sau là duyệt cây con trái theo thứ tự sau rồi duyệt cây con phải theo thứ tự sau cuối cùng mới duyệt nút gốc

- Thuật toán:

Nếu cây rỗng thì ta không làm gì cả

Nếu cây khác rỗng: thì các bước tiến hành như sau

+ Duyệt cây con trái theo thứ tự sau

+ Duyệt cây con phải theo thứ tự sau

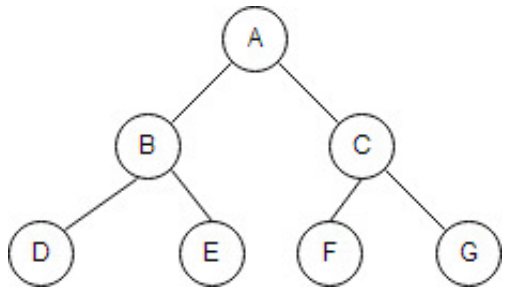
+ Duyệt gốc

Hàm duyệt theo thứ tự sau:

```
void postOrder(bTree2 * r)
{
if (r != NULL)
   {
        postOrder (r->leftChild);
```

```
        postOrder (r->rightChild);
      printf("%c", r->data);
   }
}
```

Ví dụ 5.2.3: Cho cây nhị phân:



Hình 5.15: Ví dụ cây nhị phân

Kết quả:

Duyệt theo thứ tự trước: $A\ BDE\ CFG$

Duyệt theo thứ tự giữa: $DBE\ A\ FCG$

Duyệt theo thứ tự sau: $DEB\ FGC\ A$

5.3 Cây tìm kiếm nhị phân (TKNP)

Phép toán tìm kiếm là phép toán rất phổ biến, được dùng rất nhiều. Chính vì vậy yêu cầu tìm kiếm phải nhanh. Chúng ta có thể lựa chọn mô hình danh sách để biểu diễn danh sách đặc và thực hiện tìm kiếm nhị phân. Tuy nhiên nếu lựa chọn mô hình danh sách đặc thì phép toán thêm và xóa phần tử lại không linh hoạt (thường xuyên phải dãn và dồn chỗ). Làm sao để vừa thực hiện tìm kiếm nhị phân, lại vừa thêm và xóa linh hoạt? Chúng ta xem xét mô hình cây nhị phân tìm kiếm sau:

5.3.1 Định nghĩa cây tìm kiếm nhị phân

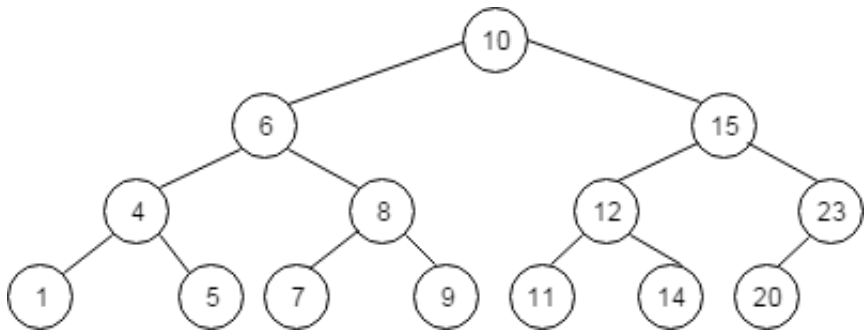
Cây tìm kiếm nhị phân là một cây nhị phân thoả mãn các điều kiện sau:

Điều kiện 1: Tất cả các khoá tại các đỉnh của cây con bên trái đều có giá trị đi trước ($<$) khoá tại đỉnh gốc

Điều kiện 2: Khoá tại gốc đi trước ($<$) tất cả các khoá ở các đỉnh của cây con bên phải

Điều kiện 3: Cây con bên trái và cây con bên phải cũng là cây tìm kiếm nhị phân

Ví dụ xét cây tìm kiếm nhị phân có khóa là các số nguyên lưu tại các đỉnh trên cây như sau (xem Hình 5.16)



Hình 5.16: Ví dụ về cây nhị phân tìm kiếm

Lưu ý:

Dữ liệu lưu trữ tại mỗi nút có thể rất phức tạp, ví dụ là một record chẳng hạn, Khi đó, khoá của nút được tính dựa trên một số trường nào đó, ta gọi là trường khoá. Trường khoá phải chứa các giá trị có thể so sánh được, tức là nó phải lấy giá trị từ một tập hợp có thứ tự.

Nhận xét:

- Trên cây $TKNP$ không có hai nút cùng khoá.
- Cây con của một cây $TKNP$ là cây $TKNP$.

- Khi duyệt cây tìm kiếm nhị phân theo tứ tự giữa, ($inOrder$), ta thu được một dãy khóa có thứ tự tăng dần.

5.3.2 Biểu diễn cây tìm kiếm nhị phân trên máy tính

Cây $TKNP$, trước hết, là một cây nhị phân. Do đó ta có thể áp dụng các cách cài đặt như đã trình bày trong phần cây nhị phân để cài đặt cây nhị phân tìm kiếm, điều lưu ý ở đây là mỗi đỉnh trên cây phải có một thành phần khóa, xác định duy nhất cho đỉnh đó.

Một cách cài đặt cây $TKNP$ thường gặp là cài đặt bằng con trỏ. Mỗi nút của cây như là một mẩu tin ($record$) có tối thiểu ba trường: một trường chứa khoá, hai trường kia là hai con trỏ trỏ đến hai nút con (nếu nút con nào vắng mặt ta gán con trỏ tương ứng trỏ tới nó bằng $null$). Xét dạng biểu diễn cây bởi con trỏ, ta gọi là BSTree như sau:

Cài đặt cây $TKNP$ bằng con trỏ:

```
#include<stdio.h>
#include<stdlib.h>
// dinh nghia cay nhi phan tim kiem banng con tro
typedef struct Node
{
   int key;
   int data;
   Node *left, *right;
} Node;
typedef Node * SearchTree;
```

5.3.3 Các phép toán cơ bản trên cây tìm kiếm nhị phân

Cây nhị phân tìm kiếm là cây nhị phân cho nên các phép toán trên cây nhị phân tìm kiếm sẽ bao gồm các phép toán trên cây nhị phân thông thường. Ngoài ra nó còn có thêm các phép toán: Tìm kiếm, thêm và xóa một nút trên cây.

1- Khởi tạo cây $TKNP$ rỗng

Ta cho con trỏ quản lý nút gốc (*Root*) của cây bằng NIL.

2 – Tìm kiếm một nút có khóa cho trước trên cây $TKNP$

Để tìm kiếm 1 nút có khoá x trên cây $TKNP$, ta bắt đầu từ nút gốc bằng cách so sánh khoá của nút gốc với khoá x.

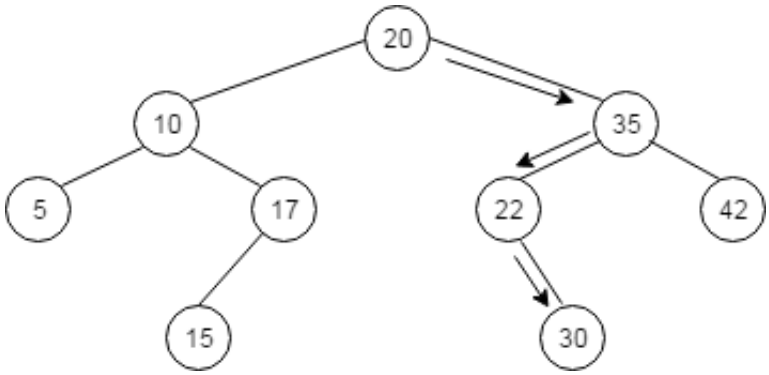
- Nếu nút gốc bằng $NULL$ thì không có khoá x trên cây.

- Ngược lại, Nếu x bằng khoá của nút gốc thì giải thuật dừng và ta đã tìm được nút chứa khoá x.

- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành việc tìm khoá x trên cây con bên phải.

- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành việc tìm khoá x trên cây con bên trái.

Ví dụ: Tìm nút có khoá 30 trong cây (xem hình 5.17)



Hình 5.17: Tìm kiếm khóa x=30 trên cây nhị phân tìm kiếm

- So sánh 30 với khoá nút gốc là 20, vì $30 > 20$ vậy ta tìm tiếp trên cây con bên phải, tức là cây có nút gốc có khoá là 35.

- So sánh 30 với khoá của nút gốc là 35, vì $30 < 35$ vậy ta tìm tiếp trên cây con bên trái, tức là cây có nút gốc có khoá là 22.

- So sánh 30 với khoá của nút gốc là 22, vì $30 > 22$ vậy ta tìm tiếp trên cây con bên phải, tức là cây có nút gốc có khoá là 30.

- So sánh 30 với khoá nút gốc là 304, $30 = 30$ vậy đến đây giải thuật dừng và ta tìm được nút chứa khoá cần tìm.

Giải thuật dưới đây trả về kết quả là con trỏ p trỏ tới nút chứa khoá x hoặc $NULL$ nếu không tìm thấy.

```
// Viet phep toan tim kiem 1 khoa x tren cay tim kiem T
//input:T searchtree; x:int;
//output: Dia chi cuar nut cos khoa la x Node;
Node * search(SearchTree T, int x)
{
   Node* V=T;
   int found=0;
   while ((V!=NULL)&& (found==0))
     if (x<V->key) V->left ;
     else if (x>V->key) V=V->right;
     else
        found=1;
   return V;
}
```

Nhận xét:

Giải thuật này sẽ rất hiệu quả về mặt thời gian nếu cây TKNP được tổ chức tốt, nghĩa là cây tương đối "cân bằng". Về cây cân bằng các bạn có thể tham khảo thêm trong các tài liệu tham khảo của môn học này.

3. Thêm một nút có khóa x vào cây $TKNP$

Xét cây $TKNP$ gốc T, hãy thêm vào T đỉnh có khóa là x nếu x chưa có trên cây. T sau khi thêm x vẫn thỏa mãn là cây $TLNP$

Theo định nghĩa cây tìm kiếm nhị phân ta thấy trên cây tìm kiếm nhị phân không có hai nút có cùng một khoá. Do đó nếu ta muốn thêm một nút có khoá x vào cây $TKNP$ trước hết ta phải tìm kiếm để xác định có nút nào chứa khoá x chưa?

+ Nếu có thì giải thuật kết thúc (không làm gì cả!).

+ Ngược lại, sẽ thêm một nút mới chứa khoá x này.

Thêm khoá x vào cây $TKNP$ đảm bảo cấu trúc cây $TKNP$ không bị phá vỡ. Có nhiều cách để thêm, tuy nhiên để tránh phức tạp, người ta thường thực hiện thêm ở mức lá. Cách giải cụ thể như sau ta bắt đầu từ nút gốc bằng cách so sánh khóa cuả nút gốc

với khoá x.

- Nếu nút gốc bằng $NULL$ thì khoá x chưa có trên cây, do đó ta thêm khoá x vào cây.

- Nếu x bằng khoá của nút gốc thì giải thuật dừng, trường hợp này ta không thêm.

- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên phải.

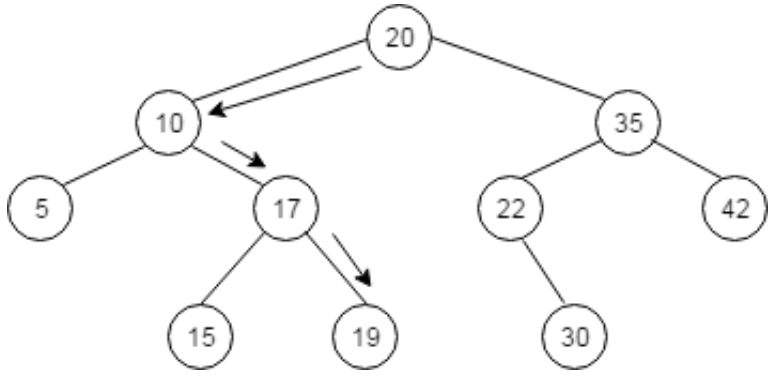
- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên trái.

Ví dụ: thêm khoá 19 vào cây (Xem hình 5.18) ở trong dưới đây, ta làm như sau:

- So sánh 19 với khoá của nút gốc là 20, vì $19 < 20$ vậy ta xét tiếp đến cây bên trái, tức là cây có nút gốc có khoá là 10.

- So sánh 19 với khoá của nút gốc là 10, vì $19 > 10$ vậy ta xét tiếp đến cây bên phải, tức là cây có nút gốc có khoá là 17.

- So sánh 19 với khoá của nút gốc là 17, vì $19 > 17$ vậy ta xét tiếp đến cây bên phải. Nút con bên phải bằng $NULL$, chứng tỏ rằng khoá 19 chưa có trên cây, ta thêm nút mới chứa khoá 19 và nút mới này là con bên phải của nút có khoá là 17, ta thu được cây như hình sau:



Hình 5.18: Thêm 1 khóa vào cây nhị phân tìm kiếm

Giải thuật:

```
void insertTree(SearchTree &T, int x)
{  Node *V,*P,*N;
   int found=0;
   // cap phat 1 nut
   N=(Node*)calloc(1,sizeof(Node));
   if (N!=NULL)
   {
   N->left=NULL;
   N->right=NULL;
   N->key=x;
   if (T==NULL) T=N;// neu cay rong nut moi them chinh la goc cua cay
   else
   {  V=T;
      while ((V!=NULL)&&(!found))
      {
         if (x<V->key )
         {
            P=V;
            V=V->left ;
         }
         else if (x>V->key )
         {
            P=V;
            V=V->right ;
         }
         else found=1;
      }
      if (found==0)
      {
      if (x<P->key) P->left =N;//them N vao ben trai cuar P
      if (x>P->key) P->right=N;
        }
   }
}
else printf(``\n Ko cap phat dc bo nho");
}
```

4. Xóa một nút có khóa cho trước ra khỏi cây $TKNP$

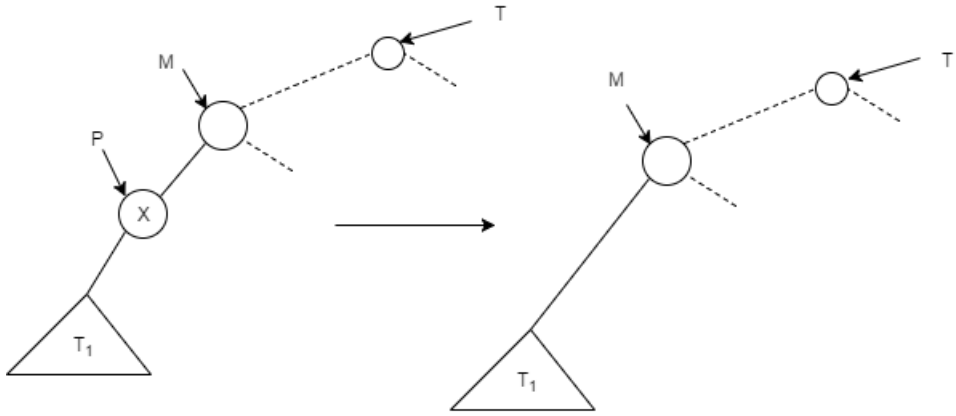
Xét cây $TKNP$ gốc T, khoá x. Nếu đỉnh có khoá x có trên T thì loại bỏ đỉnh này sao cho T sau khi loại bỏ x vẫn là cây $TKNP$

Cách giải:

- Nếu không tìm thấy nút chứa khoá x thì giải thuật kết thúc.

- Nếu tìm gặp nút được trỏ bởi p có chứa khoá x, ta có ba trường hợp sau:

Trường hợp 1: Nếu p là lá: ta thay nó bởi $NULL$. Trường hợp 2: Nếu p có một trong 2 con là rỗng: - Treo cây con khác rỗng vào vị trí của p (như hình dưới) - Giải phóng vùng nhớ được trỏ bởi p



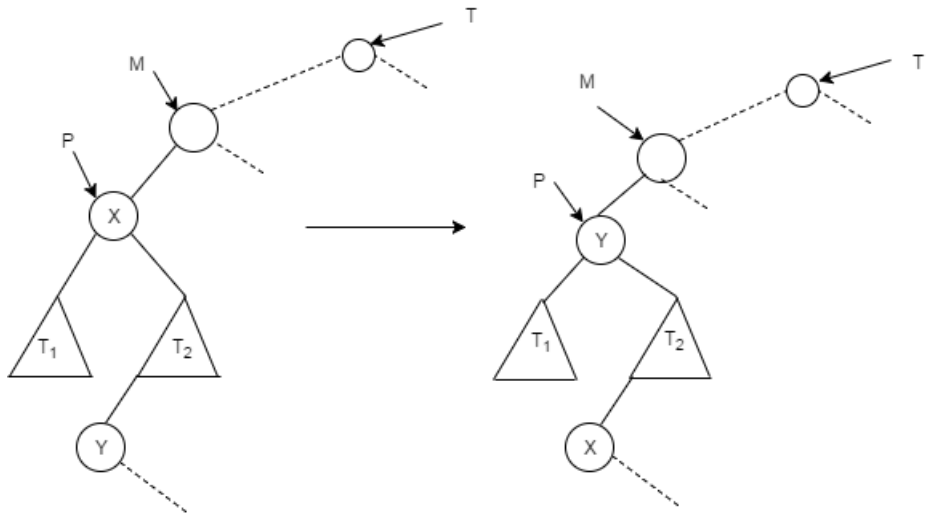
Hình 5.19: Xóa một nút trong cây NPTK với trường hợp nút cần xóa có 1 con

Trường hợp 3: Đỉnh loại bỏ được trỏ bởi P có 2 con đều khác rỗng. Thay nút được trỏ bởi p bởi nút lớn nhất trên cây con trái của nó (nút cực phải của cây con trái) hoặc là nút bé nhất trên cây con phải của nó (nút cực trái của cây con phải). Rồi xóa nút cực phải (hoặc nút cực trái), việc xoá nút này sẽ rơi vào một trong 2 trường hợp ở trên.Trong hình dưới đây, ta thay x bởi khoá của nút cực trái của cây con bên phải rồi ta xoá nút cực trái này.

Chương trình tạo cây:

```
void taoCay(SearchTree &T)
{
   int x;
   T=NULL;
   printf(‘‘\n Nhap x="); scanf(‘‘%d",&x);
   while (x!=0)
   {
      insertTree(T,x);
      printf(‘‘\n Nhap x="); scanf(‘‘%d",&x);
   }
}
```

In dữ liệu trên cây:



Hình 5.20: Xóa một nút trên cây NPTK với trương hợp nút cần xóa có đầy đủ hai con

```
//dung thuat toan duyet cay theo thu tu truoc
void inCay(SearchTree T)
{
   if (T!=NULL)
   {
      printf(‘‘%5d",T->key);
      inCay(T->left);
   // printf(‘‘%5d",T->key);
      inCay(T->right);
   }
}
```

Duyệt cây theo thứ tự giữa

```
void inOrder(SearchTree T)
{
   if (T!=NULL)
   {
      inOrder(T->left);
      printf(‘‘%5d",T->key);
      inOrder(T->right);
   }
}
```

Duyệt cây theo thứ tự trước và sau tương tự.

5.4 Cây cân bằng

5.4.1 Định nghĩa

Cây cân bằng (AVL) được gọi theo tên của hai người đề xuất ra, G.M. Adelson-Velsky và E.M. Landis, được công bố trong bài báo "An algorithm for the organization of information." ("Một thuật toán về tổ chức thông tin") [5] của họ năm 1962. Từ khi ra đời, cây AVL đã nhanh chóng được ứng dụng trong nhiều bài toán khác nhau và thu hút được nhiều nhà nghiên cứu. Cây cân bằng được định nghĩa như sau:

Cây cân bằng AVL là cây nhị phân tìm kiếm (NPTK) mà tại mỗi đỉnh của cây, độ cao của cây con trái và cây con phải khác nhau không quá 1.

Như vậy, để xét một cây có phải là cây cân bằng hay không thì trước hết phải xem có thỏa mãn là cây nhị phân tìm kiếm không, sau đó xét các nút trên cây có chỉ số cân bằng (CSCB) là một trong ba giá trị sau:

- CSCB(p) = 0 (Độ cao cây trái (p) = Độ cao cây phải (p))
- CSCB(p) = 1 (Độ cao cây trái (p) $<$ Độ cao cây phải (p))
- CSCB(p) =-1 (Độ cao cây trái (p) $>$ Độ cao cây phải (p))

Trong đó, CSCB của một nút là hiệu của chiều cao cây con phải và cây con trái của nó. Hiệu quả của cây cân bằng là các phép chèn (insertion), và xóa (deletion) trong cây luôn chỉ tốn thời gian O(log n) trong cả trường hợp trung bình và trường hợp xấu nhất.

Để tiện trong trình bày, chúng ta sẽ ký hiệu:

p $\rightarrow$ balFactor = CSCB(p);
Độ cao cây trái(p) ký hiệu là hL;
Độ cao cây phải (p) ký hiệu là hR.

Để khảo sát cây cân bằng, ta cần lưu thêm thông tin về chỉ số cân bằng tại mỗi nút. Lúc đó, cây cân bằng có thể được khai báo như sau:

```
typedef struct tagAVLNode
{
      DataType key;
      char balFactor; //Chi so can bang
      struct tagAVLNode* pLeft;
      struct tagAVLNode* pRight;
}AVLNode, *AVLTree;
// De tien trinh bay ta dinh nghia cac hang so sau
#define LH     -1  //Cay con trai cao hon cay con phai
#define EH     -0  //Hai cay con bang nhau
#define RH      1  //Cay con phai cao hon cay con trai
```

5.4.2 Các trường hợp mất cân bằng

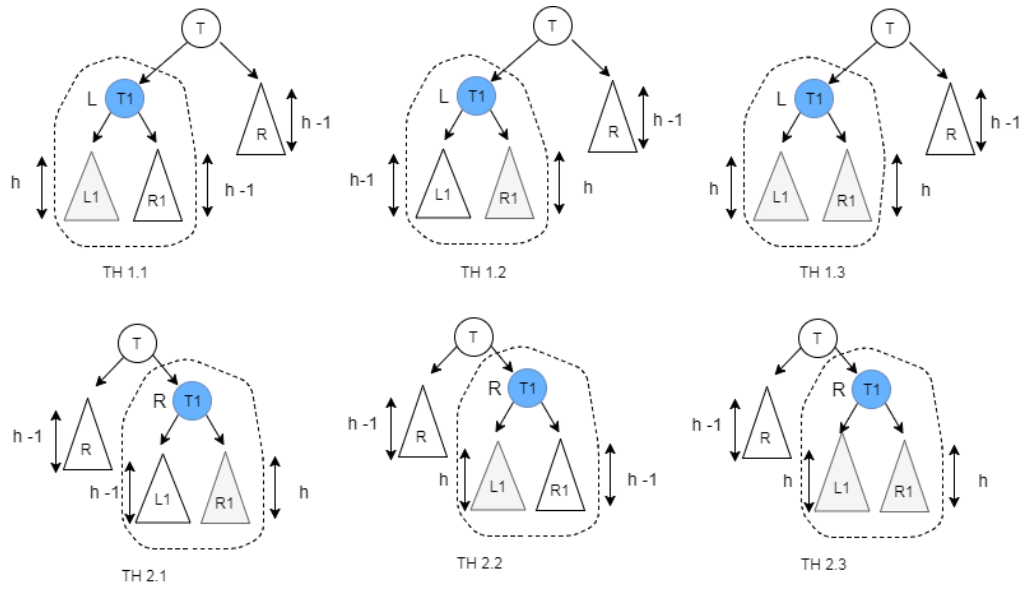
Ta sẽ không khảo sát tính cân bằng của một cây nhị phân bất kỳ mà chỉ quan tâm đến các khả năng mất cân bằng xảy ra khi thêm hoặc hủy một nút trên cây AVL. Như vậy, khi mất cân bằng, độ lệch chiều cao giữa 2 cây con sẽ là 2. Ta có 6 khả năng sau:

- **Trường hợp 1:** Cây T lệch về bên trái (trường hợp 1.1, 1.2, 1.3 trong Hình 5.21).
- **Trường hợp 2:** cây T lệch về bên phải (trường hợp 2.1, 2.2, 2.3 trong Hình 5.21).

Ta có thể thấy rằng các trường hợp lệch về bên phải hoàn toàn đối xứng với các trường hợp lệch về bên trái. Vì vậy ta chỉ cần khảo sát trường hợp lệch về bên trái.

5.4.3 Các phép quay cây

Các phép toán chèn hay xóa bỏ một chút trên cây có thể làm cây mất cân bằng. khi đó ta sử dụng một hoặc nhiều phép quay để



Hình 5.21: Các trường hợp mất cân bằng cây.

đưa cây về trạng thái cân bằng. Các phép quay cơ bản sau dùng để cân bằng lại cây.

Phép quay trái đơn trái (left - left)

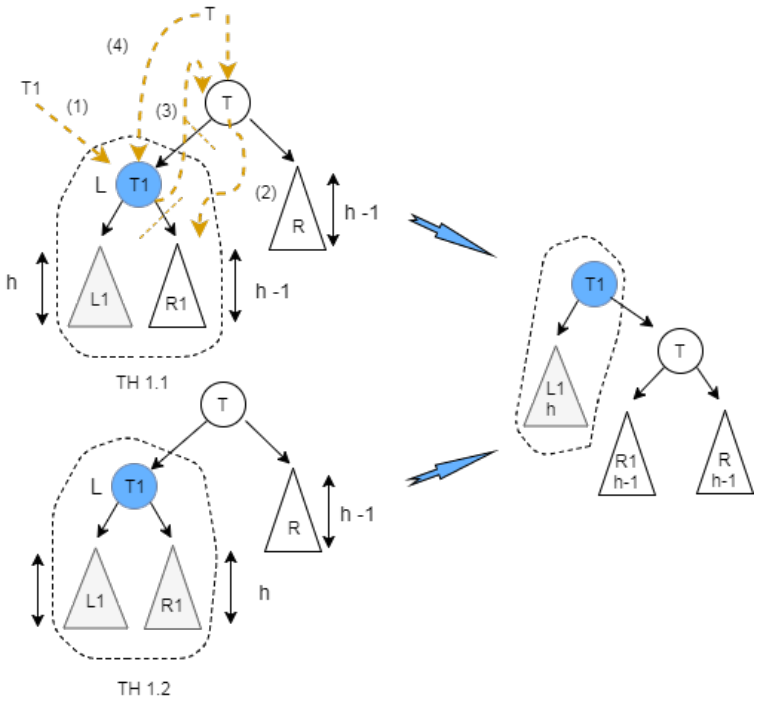
TH1.1 (cây T1 lệch trái) & TH 1.2 (cây T1 cân bằng). Ta có thể quan sát Hình 5.22, để cân bằng cây ta thực hiện phép quay trái đơn (left - left). Các bước quay như sau:

(1) Đưa T1 làm gốc

(2) Đưa cây con phải của T1 thành con trái của T

(3) Đưa cây gốc T thành cây con phải của T1.

```
//Quay don Left-Left
void rotateLL ( AVLTree &T)
{
          AVLNode * T1 = T-> pLeft ;
          T->pLeft     = T1->pRight ;
          T1-> pRight = T;
          switch( T1-> balFactor )
          {
                    case LH: T-> balFactor = EH;
                             T1-> balFactor = EH;
```



Hình 5.22: Quay trái cây mất cân bằng trong trường hợp 1.1.

```
                    break ;
            case EH: T-> balFactor = LH;
                    T1-> balFactor = RH;
                    break ;
        }
        T = T1;
}
```

Trước khi cân bằng, cây T có chiều cao h+2. Sau khi cân bằng, TH1.1, TH 1.2 cây T có chiều cao h+1.

Phép quay phải đơn Phải (right - right)

Tương tự cho trường hợp cây T lệch phải, ta có phép quay đơn phải như sau:

```
//Quay don Right-Right
void rotateRR(AVLTree &T)
{
        AVLNode * T1 = T-> pRight ;
        T->pRight = T1-> pLeft ;
```

```
            T1-> pLeft = T;
            switch ( T1-> balFactor )
            {
                        case RH: T-> balFactor = EH;
                                    T1-> balFactor = EH;
                                    break ;
                        case EH: T-> balFactor = RH;
                                    T1-> balFactor = LH;
                                    break ;
            }
            T = T1;
}
```

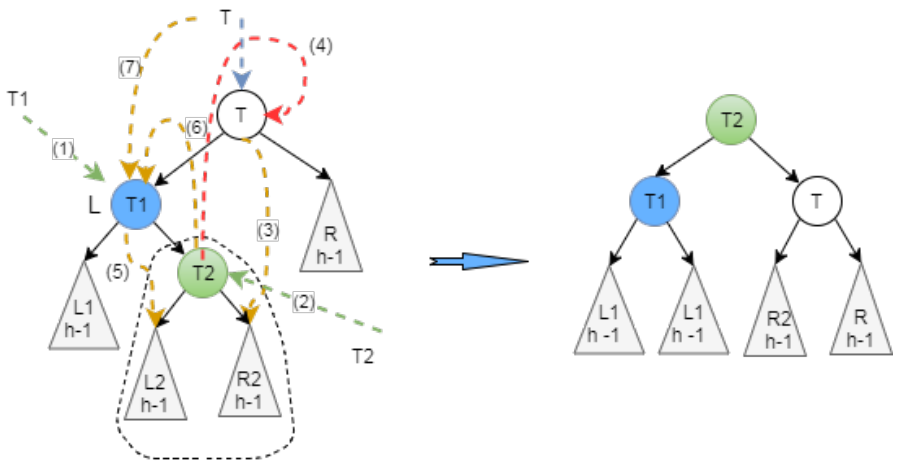
Phép quay kép left - right

TH1.3: Cây T1 lệch phải, ta thực hiện phép quay kép left - right.

```
//Quay kep Left-Right
void rotateLR(AVLTree &T)
{
            AVLNode * T1 = T-> pLeft ;
            AVLNode * T2 = T1-> pRight ;
            T-> pLeft     = T2-> pRight ;
            T2-> pRight = T;
            T1-> pRight = T2-> pLeft ;
            T2-> pLeft    = T1;
            switch (T2-> balFactor )
            {
                        case LH: T-> balFactor = RH;
                                    T1-> balFactor = EH;
                                    break ;
                        case EH: T-> balFactor = EH;
                                    T1-> balFactor = EH;
                                    break ;
                        case RH: T-> balFactor = EH;
                                    T1-> balFactor = LH;
                                    break ;
            }
             T2-> balFactor = EH;
            T = T2;
}
```

Tương tự đối với cây T lệch phải ta sử dụng phép quay kép phải - trái (Right - left).

```
//Quay kep Right-Left
void rotateRL ( AVLTree &T)
{
```



Hình 5.23: Quay kép cây mất cân bằng trong trường hợp 1.3.

```
        AVLNode * T1 = T-> pRight ;
        AVLNode * T2 = T1-> pLeft ;
        T->pRight = T2-> pLeft ;
        T2->pLeft = T;
        T1->pLeft = T2-> pRight ;
        T2-> pRight = T1;
        switch (T2-> balFactor )
        {
                    case RH: T-> balFactor = LH;
                             T1-> balFactor = EH;
                             break ;
                    case EH: T-> balFactor = EH;
                             T1-> balFactor = EH;
                             break ;
                    case LH: T-> balFactor = EH;
                             T1-> balFactor = RH;
                             break ;
        }
        T2-> balFactor = EH;
        T = T2;
}
```

5.4.4 Cân bằng cây sử dụng phép quay

Khi ta bổ sung một nút hay xóa một nút khỏi cây, cây có thể bị mất cân bằng. Đối với mỗi trường hợp mất cân bằng khác nhau ta có thể sử dụng các phép quay như trong Bảng 5.1.

Bảng 5.1: Phép quay cây.

Mất cân bằng phải	
Mất cân bằng phải-phải (R-R)	Quay trái tại node bị mất cân bằng
Mất cân bằng phải-trái (R-L)	Quay phải tại node con phải của node bị mất cân bằng Quay trái tại node bị mất cân bằng
Mất cân bằng trái	
Mất cân bằng trái-trái (L-L)	Quay phải tại node bị mất cân bằng
Mất cân bằng trái-phải (L-R)	Quay trái tại node con trái của node bị mất cân bằng Quay phải tại node bị mất cân bằng

```
//Can bang khi cay bi lech trai
int balanceLeft(AVLTree &T)
{
          switch (T-> pLeft -> balFactor )
          {
                    case LH: rotateLL (T); return 2;
                    case EH: rotateLL (T); return 1;
                    case RH: rotateLR (T); return 2;
          }
          return 0;
}
```

```
//Can bang khi cay bi lech ve ben phai
int balanceRight(AVLTree &T)
{
          switch (T-> pRight -> balFactor )
          {
                    case LH: rotateRL (T); return 2;
                    case EH: rotateRR (T); return 1;
                    case RH: rotateRR (T); return 2;
          }
          return 0;
}
```

5.4.5 Thêm một phần tử vào cây AVL

Thêm một phần tử vào cây AVL diễn ra tương tự như trên cây NPTK. Sau khi thêm, nếu chiều cao của cây thay đổi, từ vị trí thêm vào, ta lần ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng không. Nếu có, ta phải cân bằng lại ở nút này. Việc cân bằng lại chỉ cần thực hiện 1 lần tại nơi mất cân bằng.

```
int insertNode(AVLTree &T, DataType X)
{
	int res;
	if (T != NULL )
	{
		if(T->key == X)
			return 0; //da co
		else if (T->key > X)
		{
			res = insertNode(T->pLeft, X);
			if (res < 2)
				return res;
			switch (T->balFactor)
			{
				case RH: T->balFactor= EH;
					return 1;
				case EH: T->balFactor = LH;
					return 2;
				case LH: balanceLeft(T);
					return 1;
			}
		}
		else // if (T->key < X)
		{
			res = insertNode(T-> pRight, X);
			if (res < 2)
				return res;
			switch (T->balFactor)
			{
				case LH: T->balFactor = EH;
					return 1;
				case EH: T->balFactor = RH;
					return 2;
				case RH: balanceRight(T);
					return 1;
			}
		}
	}
	T = new TNode;
	if (T == NULL) return -1; //thieu bo nho

	T->key = X;
	T->balFactor = EH;
	T->pLeft = T->pRight = NULL;

	return 2;
}
```

Hàm insertNode trả về giá trị -1, 0, 1 khi không đủ bộ nhớ, gặp nốt cũ hay thành công. Nếu sau khi thêm, chiều cao cây bị tăng,

giá trị 2 sẽ được trả về:

```
int insertNode(AVLTree &T, DataType X)
{
          int res;
          if (T != NULL )
          {
                    if(T->key == X)
                              return 0;
                    else if (T->key > X)
                    {
                              res = insertNode(T->pLeft, X);
                              if (res < 2)
                                        return res;
                              switch (T->balFactor)
                              {
                                        case RH: T->balFactor= EH;
                                                  return 1;
                                        case EH: T->balFactor = LH;
                                                  return 2;
                                        case LH: balanceLeft(T);
                                                  return 1;
                              }
                    }
                    else // if (T->key < X)
                    {
                              res = insertNode(T-> pRight, X);
                              if (res < 2)
                                        return res;
                              switch (T->balFactor)
                              {
                                        case LH: T->balFactor = EH;
                                                  return 1;
                                        case EH: T->balFactor = RH;
                                                  return 2;
                                        case RH: balanceRight(T);
                                                  return 1;
                              }
                    }
          }
          T = new TNode;
          if (T == NULL) return -1; //thieu bo nho

          T->key = X;
          T->balFactor = EH;
          T->pLeft = T->pRight = NULL;

          return 2; // thanh cong, tang chieu cao
}
```

5.4.6 Xóa một phần tử

Cũng giống như thao tác thêm một nút, việc hủy một phần tử X ra khỏi cây AVL thực hiện giống như trên CNPTK. Sau khi hủy, nếu tính cân bằng của cây bị vi phạm ta sẽ thực hiện việc cân bằng lại. Tuy nhiên việc cân bằng lại trong thao tác hủy sẽ phức tạp hơn nhiều do có thể xảy ra phản ứng dây chuyền.

Hàm deleteNode trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây. Nếu sau khi huỷ, chiều cao cây bị giảm, giá trị 2 sẽ được trả về.

```
int deleteNode(AVLTree &T, DataType X)
{
          int res ;
          if (T==NULL)
                    return 0;
          if (T->key > X)
          {
                    res = deleteNode (T-> pLeft , X);
                    if ( res < 2)
                              return res ;
                    switch ( T-> balFactor )
                    {
                              case LH: T-> balFactor = EH;
                                       return 2;
                              case EH: T-> balFactor = RH;
                                          return 1;
                              case balanceRight (T);
                    }
          }
          else if (T->key < X)
          {
                    res = deleteNode (T-> pRight , X);
                    if ( res < 2) return res ;
                    switch (T-> balFactor )
                    {
                              case RH: T-> balFactor = EH;
                                          return 2;
                              case EH: T-> balFactor = LH;
                                          return 1;
                              case LH: return balanceLeft (T);
                    }
          }
          else // if (T->key == X)
          {
```

```
                    AVLNode * p = T;
                    if (T-> pLeft == NULL)
                    {
                              T = T->pRight ; res = 2;
                    }
                    else if (T-> pRight == NULL)
                    {
                              T = T-> pLeft ; res = 2;
                    }
                    else // if (T-> pLeft != NULL && T-> pRight != NULL)
                    {
                              res = searchStandFor ( p, T -> pRight );
                              if( res < 2)
                                        return res;
                               switch( T-> balFactor )
                              {
                                        case RH: T-> balFactor = EH;
                                                  return 2;
                                        case EH: T-> balFactor = LH;
                                                  return 1;
                                        case LH: return balanceLeft (T);
                              }
                    }
                    delete p;
                    return res ;
          }
}

/Tim phan tu the
int searchStandFor(AVLTree &p, AVLTree &q)
{
          int res;
          if (q->pLeft != NULLL)
          {
                    res = searchStandFor(p, q->pLeft);
                    if (res < 2)
                              return res;
                    switch (q->balFactor)
                    {
                              case LH: q->balFactor= EH;
                                        return 2;
                              case EH: q->balFactor= RH;
                                        return 1;
                              case RH: return balanceRight(T);
                    }
          } else {
                    p->key = q->key;
                    p = q;
                    q = q->pRight;
                    return 2;
          }
```

}

Với cây cân bằng trung bình 2 lần thêm vào cây thì cần một lần cân bằng lại; 5 lần hủy thì cần một lần cân bằng lại. Việc huỷ 1 nút có thể phải cân bằng dây chuyền các nút từ gốc cho đên phần tử bị huỷ trong khi thêm vào chỉ cần 1 lần cân bằng cục bộ. Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số nút trên cây là bao nhiêu.

BÀI TẬP

Bài 1. Với mỗi phương pháp biểu diễn cây tổng quát. Yêu cầu:
- Viết dạng cài đặt cây.
- Ứng với mỗi cách hãy viết các phép toán trên cây
- Nhập và In cây

Bài 2. Cho một cây nhị phân chứa các số thực
Yêu cầu:
- Viết CTDL biểu diễn cây bằng mảng
- Cài đặt các thao tác sau trên CTDL cây này:
+ Nhập dữ liệu cho cây
+ Hiển thị dữ liệu trên cây
- Duyệt cây theo thứ tự trước, thứ tự giữa, thứ tự sau

Bài 3. Cho một cây biểu thức nhị phân. Yêu cầu:
- Biểu diễn cây bởi CTDL con trỏ
- Thực hiện các thao tác sau trên cây:
+ Xoá một đỉnh lá nào đó
+ Xoá cây con có gốc tại đỉnh thứ k nào đó
+ Duyệt cây để hiển thị : Duyệt trước, duyệt giữa, duyệt sau
+ Sửa dữ liệu lưu tại các cây trên để biến cây trở thành cây thư mục và thực hiện các thao tác có thể có trên cây thư mục

+ Hiển thị hình ảnh cây sau khi nó được biểu diễn trên máy tính

+ Đánh mức cho các đỉnh trên cây

+ Tìm chiều cao của cây.

Bài 4. Cho cây nhị phân tìm kiếm T, mỗi nút chứa một số nguyên. Yêu cầu:

- Biểu diễn cây bởi CTDL con trỏ

- Nhập dữ liệu cho cây gồm n đỉnh, với n là số nguyên dương

- Thêm đỉnh có khóa x vào cây

- Tìm kiếm một đỉnh có khóa x trên cây

- Xóa một đỉnh bất kì trên cây

- Đếm xem trong cây có bao nhiêu nút có giá trị âm? bao nhiêu nút có giá trị dương?

- Tìm phần tử dương nhỏ nhất của cây

- Hủy toàn bộ dữ liệu trên cây

Chương 6

Mô hình dữ liệu đồ thị

Nhiều vấn đề trong khoa học máy tính và toán học được mô hình hóa thành các trạng thái và các chuyển tiếp giữa các trạng thái đó. Trong chương trước, chương về mô hình dữ liệu cây, chúng ta thấy rằng cây đã giúp mô hình hóa và giải quyết được các vấn đề khác nhau trong khoa học máy tính. Tuy nhiên, cây chỉ có thể biểu diễn các mối quan hệ có tính chất phân cấp, chẳng hạn như quan hệ cha con. Đồ thị tổng quát hơn cây, nó cho phép biểu diễn mối quan hệ phức tạp hơn. Chương này sẽ trình bày khái niệm cũng như kí hiệu về đồ thị, các biểu diễn khác nhau của đồ thị trong máy tính, các bài toán cơ bản trong đồ thị như tìm kiếm theo chiều sâu, tìm kiếm theo chiều rộng, tìm cây khung nhỏ nhất và tìm đường đi ngắn nhất.

6.1 Định nghĩa đồ thị và các kí hiệu

Các kí hiệu sau đây sẽ giúp trình bày các định nghĩa đồ thị trong chương này.

- Một tập V là một tập các phần tử không có thứ tự.
  Ví dụ, $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ là tập hợp gồm 13 số tự nhiên đầu tiên. Một tập con của tập V là một tập hợp, có thể có chứa bất kỳ phần tử nào trong V. Tập $U = \{5, 8, 2\}$ là tập con của V.

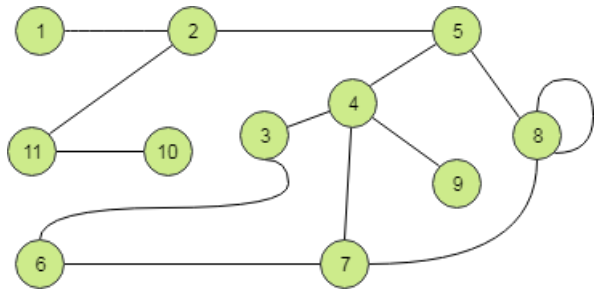
- *Số phần tử (Cardinality)* của một tập hợp là *kích thước* của tập hợp ấy, kí hiệu $|V|$.
  Ví dụ, kích thước của V là 13 và U là 3 thì kí hiệu $|V| = 13$ và $|U| = 3$.

Các định nghĩa cơ bản:

- *Đồ thị (graph):*
  Một đồ thị $G = (V, E)$ gồm hai tập hữu hạn: V là tập khác rỗng được gọi là tập đỉnh (vertices) của đồ thị, E được gọi là tâp cạnh hay cung (edge) của đồ thị (E có thể rỗng).
  Xem xét ví dụ đơn giản, đồ thị được đưa ra trong Hình 6.2. *Tập đỉnh*: $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ và *tập cạnh* E = {{0, 10}, {1, 10}, {1, 4}, {2, 3}, {2, 8}, {2, 6},{3, 9}, {5, 4}, {5, 12}, {5, 7}, {11, 12}, {11, 10}, {9, 10}} xác định đồ thị này. Mỗi cạnh là một tập hợp có kích thước bằng 2, thứ tự của các đỉnh trong mỗi tập cạnh không quan trọng. Các đỉnh còn được gọi là *nút* hoặc *điểm*; Các cạnh có thể được gọi là *cung*. Một cung là một cặp các đỉnh (u, v), u được gọi là *đầu cung* và v là *cuối cung*, kí hiệu cung như sau:

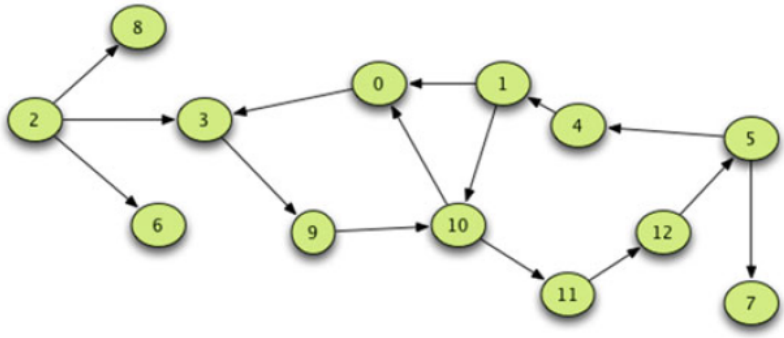
Hình 6.1: Biểu diễn cung trong đồ thị.

- G được gọi là *đơn đồ thị* nếu giữa hai đỉnh u, v của V có nhiều nhất là 1 cạnh trong E nối từ u tới v.
- G được gọi là *đa đồ thị* nếu giữa hai đỉnh u, v của V có thể có nhiều hơn 1 cạnh trong E nối từ u tới v.
- *Đồ thị vô hướng:*
  Đồ thị $G = (V, E)$, với mỗi $e \in E$ là cặp không có thứ tự (u, v), nghĩa là $(u, v) = (v, u)$ thì đồ thị G gọi là *đồ thị vô hướng* và (u, v) được gọi là cung.
  Ví dụ đồ thị vô hướng trong Hình 6.2 cạnh $\{1, 4\}$ giống cạnh $\{4, 1\}$.



Hình 6.2: Đồ thị đơn vô hướng.

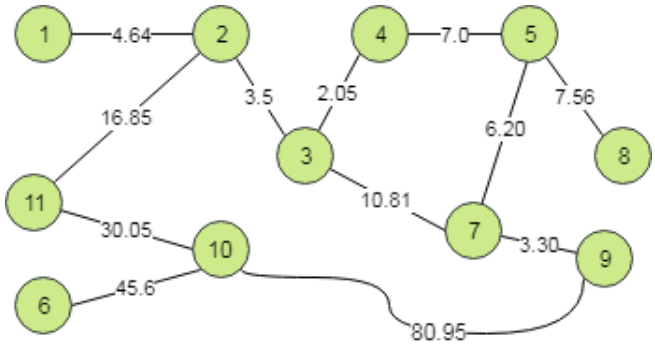
- *Đồ thị có hướng:*
  Đồ thị $G = (V, E)$ có cặp $e = (u, v), e \in E$, phân biệt với (v, u) được gọi là *đồ thị có hướng*. trong đó u gọi là đỉnh đầu của e, v gọi là đỉnh cuối của e.
  Ví dụ đồ thị có hướng trong Hình 6.3 cạnh $\{1, 4\}$ là khác cạnh $\{4, 1\}$.
- *Đồ thị có trọng số:*



Hình 6.3: Đồ thị đơn có hướng.

Đồ thị $G = (V, E)$, trên mỗi cung của đồ thị người ta gán một giá trị biểu diễn một thông tin nào đó, giá trị đó gọi là *trong số*, G được gọi là đồ thị có trọng số.

Chú ý trong giáo trình, khi không chỉ rõ là đồ thị gì thì mặc định đồ thị được nói đến là đồ thị không có trọng số.

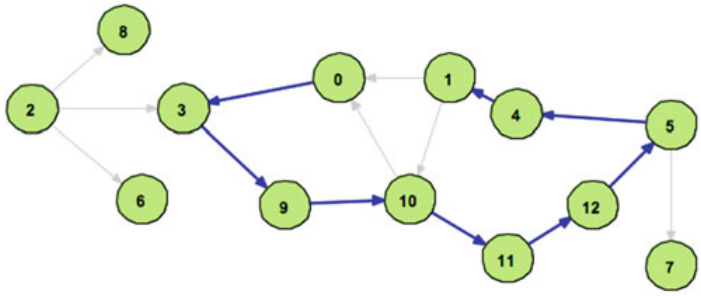


Hình 6.4: Đồ thị đơn có trọng số.

- Đối với đồ thị vô hướng $G = (V, E)$. Xét một cạnh $e \in E$, nếu $e = (u, v)$ thì ta nói hai đỉnh u và v là *kề nhau (adjacent)* và cạnh e này *liên thuộc (incident)* với đỉnh u và đỉnh v.
- Với đỉnh v trong đồ thị, ta định nghĩa *bậc (degree)* của v, ký hiệu deg(v) là số cạnh liên thuộc với v. Dễ thấy rằng trên đơn đồ thị thì số cạnh liên thuộc với v cũng là số đỉnh kề với v.

Đỉnh cô lập là đỉnh không kề với đỉnh nào; *đỉnh treo* là đỉnh chỉ kề với 1 đỉnh khác.

- Một *đường đi* trong một đồ thị G là một dãy các đỉnh $v_1, v_2, ..., v_n$, sao cho $v_1 \rightarrow v_2, v_2 \rightarrow v_3, ..., v_{n-1} \rightarrow v_n$, n là số nguyên dương thuộc E. Đường đi này đi từ *đỉnh đầu* v_1 đến đỉnh cuối v_n và đi qua tất cả các đỉnh $v_2, v_3, ..., v_{n-1}$, kết thúc tại đỉnh v_n. *Chiều dài đường đi* là số cung trên đường đi đó (đối với đồ thị không có trọng số), là tổng trọng số trên các cung của đường đi đó (đối với đồ thị có trọng số). Đỉnh u được gọi là đỉnh đầu, đỉnh v được gọi là đỉnh cuối của đường đi. Trong trường hợp đặc biệt, điểm đầu v_1 trùng với điểm cuối v_n thì đường đi được gọi là một *chu trình*(Circuit), đường đi không có cạnh nào đi qua hơn 1 lần gọi là *đường đi đơn*, tương tự ta có khái niệm *chu trình đơn*.



Hình 6.5: Một đường dẫn từ đỉnh 0 đến đỉnh 1.

Nhiều vấn đề có thể được mô hình hoá và biểu diễn bằng đồ thị. Đồ thị có ứng dụng rộng rãi trong nhiều lĩnh vực khác nhau như kỹ thuật, vật lý, khoa học sinh học, khoa học máy tính... Có một số bài toán điển hình như sau:

- ***Bài toán tô mầu đồ thị***

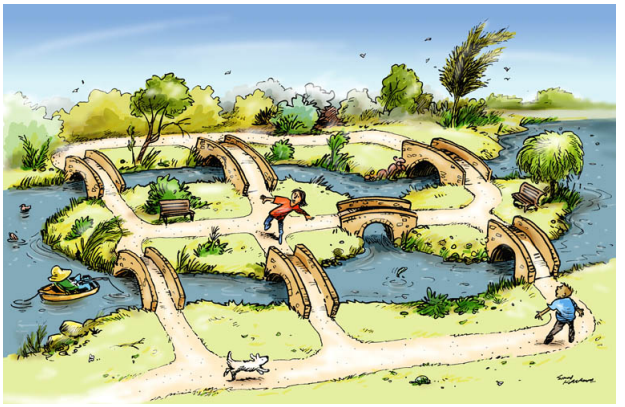
Cần bao nhiêu màu cần thiết để vẽ bản đồ thế giới sao cho không có hai quốc gia nào chung biên giới có màu giống nhau. Trong vấn đề này các đỉnh trong Hình 6.2 đại diện các quốc gia và hai quốc gia có cùng biên giới sẽ có một cạnh nối giữa 2 đỉnh đó. Bài toán được phát biểu lại là: tìm ra số lượng tối thiểu các màu để tô màu mỗi đỉnh trong đồ thị sao cho hai đỉnh kề nhau không cùng màu.

- ***Bài toán cây cầu Königsberg***

  Bài toán 7 cây cầu Königsberg có lẽ là ví dụ nổi tiếng nhất trong lý thuyết đồ thị. Đó là một vấn đề lâu dài cho đến khi được giải quyết bởi Euler vào năm 1736. Vấn đề được miêu tả trong Hình 6.6. Tại thành phố Königsberg nơi có dòng sông Pregel chảy quá, gồm hai hòn đảo lớn nối với nhau và với đất liền bởi bảy cây cầu. Bài toán đặt ra là tìm một đường đi mà đi qua mỗi cây cầu một lần và chỉ đúng một lần sau đó lại quay về điểm xuất phát. Bài toán được Euler mô hình hoá và biểu diễn bởi đồ thị. Ông loại bỏ tất cả các chi tiết ngoại trừ các vùng đất và các cây cầu, sau đó thay thế mỗi vùng đất bằng một đỉnh của đồ thị, thay mỗi cây cầu bằng một cạnh của đồ thị. Euler đã chứng minh rằng không tồn tại đường đi như thế bằng cách sử dụng các định lí trong lí thuyết đồ thị và mở đầu cho lĩnh vực lí thuyết đồ thị sau này.

- ***Bài toán người bán hàng***

  Có một người giao hàng cần đi giao hàng tại n thành phố. Anh ta xuất phát từ một thành phố nào đó, đi qua các thành phố khác để giao hàng và trở về thành phố ban đầu. Mỗi thành phố chỉ đến một lần, và khoảng cách từ một thành phố đến các thành phố khác đã được biết trước. Tìm đường đi thoả mãn



Hình 6.6: Bài toán cây cầu cầu Konigsber (Nguồn http://simonkneebone.com/2011/11/29/konigsberg-bridge-puzzle/)

yêu cầu với chi phí thấp nhất. Bài toán có thể được mô hình hoá như một đồ thị vô hướng có trọng số, trong đó mỗi thành phố là một đỉnh của đồ thị còn mỗi đường đi giữa các thành phố là một cạnh. Khoảng cách giữa hai thành phố là độ dài cạnh. Hãy tìm một chu trình sao cho tổng độ dài các cạnh là nhỏ nhất.

6.2 Biểu diễn đồ thị trên máy tính

Với một biểu đồ nhất định có một số cách biểu diễn khác nhau. Việc dễ dàng cài đặt các thuật toán trên đồ thị cũng như hiệu quả của thuật toán phụ thuộc vào sự lựa chọn cách biểu diễn đồ thị đúng đắn và hợp lí. Hai cấu trúc dữ liệu được sử dụng phổ biến nhất để biểu diễn một đồ thị (có hướng hay vô hướng) là ma trận kề và danh sách kề. Ngoài ra có cách biểu diễn bằng danh sách cạnh nhưng ít phổ biến hơn.

6.2.1 Ma trận kề

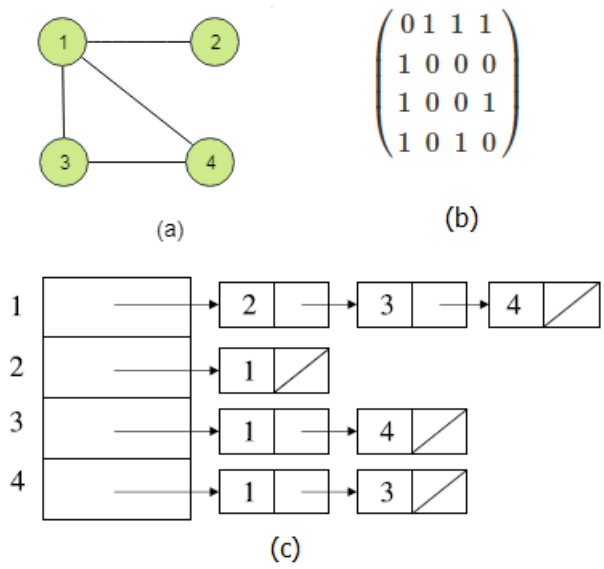
Giả sử $G = (V, E)$ là đơn đồ thị có số đỉnh (ký hiệu |V|) là n, không mất tính tổng quát có thể coi các đỉnh được đánh số 1, 2, ..., n. Khi đó, ta có thể biểu diễn đồ thị bằng một ma trận vuông $A = [a_{ij}]$ cấp n. Trong đó:

$a_{ij} = 1$ nếu $(i, j) \in E$

$a_{ij} = 0$ nếu $(i, j) \notin E$

Quy ước $a_{ii} = 0$ với $\forall i$;

Đối với đa đồ thị thì việc biểu diễn cũng tương tự trên, nhưng a_{ij} bằng số cạnh nối giữa 2 đỉnh i và j.



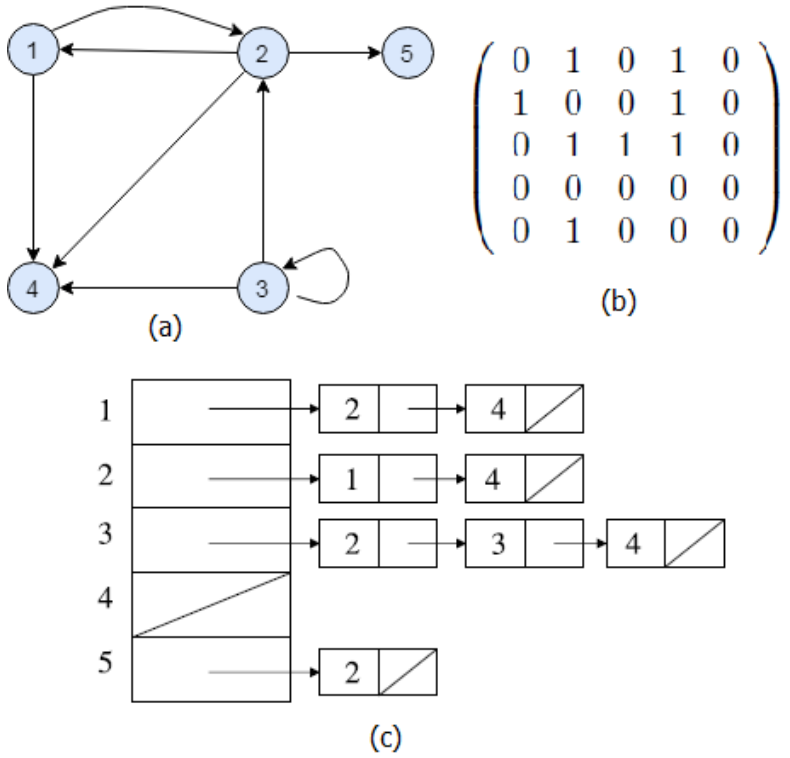
Hình 6.7: (a) Một đồ thị vô hướng; (b) Biểu diễn ma trận kề; (c) Biểu diễn danh sách kề.

Các tính chất của ma trận liền kề:

1. Với đồ thị vô hướng G, ma trận liền kề tương ứng là ma trận đối xứng (aij = aji), điều này không đúng với đồ thị có hướng.
2. Nếu G là đồ thị vô hướng và A là ma trận liền kề tương ứng

thì trên ma trận có:
Tổng các số trên hàng i = tổng các số trên cột i = Bậc của đỉnh i = deg(i)

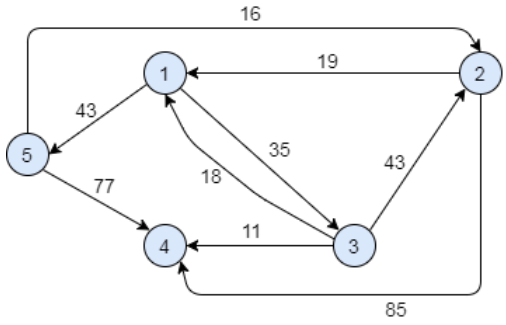
3. Nếu G là đồ thị có hướng và A là ma trận liền kề tương ứng thì trên ma trận A:
Tổng các số trên hàng i = số bậc ra của đỉnh i = deg+(i) Tổng các số trên cột i = số bậc vào của đỉnh i = deg-(i) Trong trường hợp G là đơn đồ thị, ta có thể biểu diễn ma trận liền kề A tương ứng là các phần tử logic: $a_{ij} = TRUE$ nếu $(i, j) \in E$) $a_{ij} = FALSE$ nếu $(i, j) \notin E$



Hình 6.8: (a) Một đồ thị có hướng; (b) Ma trận kề: (c) Biểu diễn danh sách kề.

Ưu điểm của ma trận liền kề:

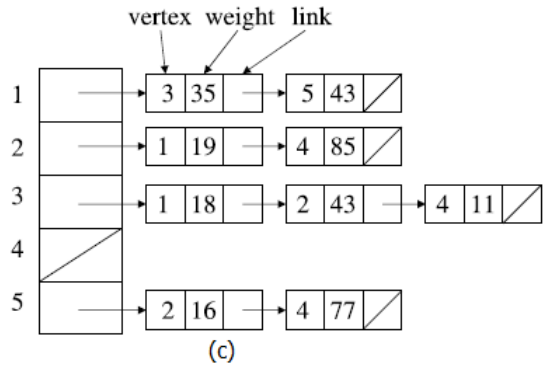
1. Đơn giản, trực quan, dễ cài đặt trên máy tính.



(a)

$$W = \begin{pmatrix} - & \infty & 35 & \infty & 43 \\ 19 & - & \infty & 85 & \infty \\ 18 & 43 & - & 11 & \infty \\ \infty & \infty & \infty & - & \infty \\ \infty & 16 & \infty & 77 & - \end{pmatrix}$$

(b)



(c)

Hình 6.9: (a) Một đồ thị trọng; (b) Biểu diễn ma trận kề: (c) Biểu diễn danh sách kề.

2. Để kiểm tra xem hai đỉnh (u, v) của đồ thị có kề nhau hay không, ta chỉ việc kiểm tra bằng một phép so sánh: $a_{uv} \neq 0$.

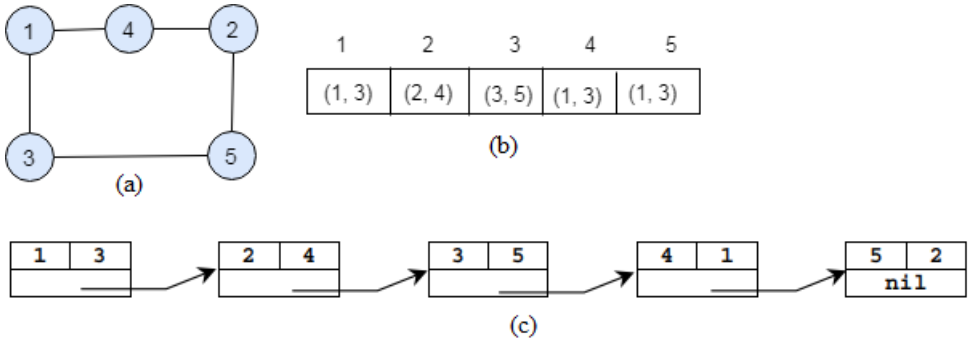
Nhược điểm của ma trận liền kề:

1. Số cạnh của đồ thị là nhiều hay ít, ma trận liền kề luôn đòi hỏi n^2 ô nhớ để lưu các phần tử ma trận gây lãng phí bộ nhớ, dẫn tới không thể biểu diễn được đồ thị với số đỉnh lớn.

2. Với đỉnh u bất kỳ của đồ thị, trong nhiều bài toán ta phải xét tất cả các đỉnh v kề với u, hoặc các cạnh liên thuộc với u. Trên ma trận liền kề, ngay cả khi đỉnh u là đỉnh cô lập (không kề với đỉnh nào) hoặc đỉnh treo (chỉ kề với 1 đỉnh) ta cũng phải xét tất cả các đỉnh và kiểm tra điều kiện trên dẫn tới lãng phí thời gian.

6.2.2 Danh sách cạnh

Trong trường hợp đồ thị có n đỉnh, m cạnh, ta có thể biểu diễn đồ thị dưới dạng danh sách cạnh, trong cách biểu diễn này, người ta liệt kê tất cả các cạnh của đồ thị trong một danh sách, mỗi phần tử của danh sách là một cặp (u, v) tương ứng với một cạnh của đồ thị. (Trong trường hợp đồ thị có hướng thì mỗi cặp (u, v) tương ứng với một cung, u là đỉnh đầu và v là đỉnh cuối của cung). Danh sách được lưu trong bộ nhớ dưới dạng mảng hoặc danh sách móc nối. Ví dụ với đồ thị dưới đây:



Hình 6.10: (a) Một đồ thị vô hướng; (b) Biểu diễn đồ thị theo danh sách cạnh cài đặt bằng ma trận; (c) Biểu diễn đồ thị theo danh sách cạnh bằng danh sách móc nối.

Ưu điểm của danh sách cạnh:

- Trong trường hợp đồ thị thưa (có số cạnh tương đối nhỏ: chẳng hạn m $<$ 6n), cách biểu diễn bằng danh sách cạnh sẽ tiết kiệm được không gian lưu trữ, bởi nó chỉ cần 2m ô nhớ để lưu danh sách cạnh.
- Trong một số trường hợp, ta phải xét tất cả các cạnh của đồ thị thì cài đặt trên danh sách cạnh làm cho việc duyệt các cạnh dễ dàng hơn (thuật toán Kruskal chẳng hạn).

Nhược điểm của danh sách cạnh:

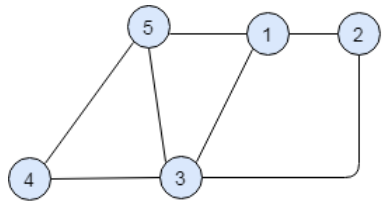
- Nhược điểm cơ bản của danh sách cạnh là khi ta cần duyệt tất cả các đỉnh kề với đỉnh v nào đó của đồ thị, thì chẳng có cách nào khác là phải duyệt tất cả các cạnh, lọc ra những cạnh có chứa đỉnh v và xét đỉnh còn lại. Điều đó khá tốn thời gian trong trường hợp đồ thị dày (nhiều cạnh).

6.2.3 Danh sách kề

Để khắc phục nhược điểm của các phương pháp ma trận kề và danh sách cạnh, người ta đề xuất phương pháp biểu diễn đồ thị bằng danh sách kề. Trong cách biểu diễn này, với mỗi đỉnh v của đồ thị, ta cho tương ứng với nó một danh sách các đỉnh kề với v. Với đồ thị $G = (V, E)$. V gồm n đỉnh và E gồm m cạnh. Có hai cách cài đặt danh sách kề phổ biến:

Cách 1: Cài đặt bằng mảng (Forward Star)

Dùng một mảng các đỉnh, mảng đó chia làm n đoạn, đoạn thứ i trong mảng lưu danh sách các đỉnh kề với đỉnh i: Ví dụ với đồ thị sau, danh sách kề sẽ là một mảng A gồm 12 phần tử: Để biết



1	2	3	4	5	6	7	8	9	10	11	12
2	**3**	**5**	**1**	**3**	**1**	**2**	**4**	**3**	**5**	**1**	**4**
Đoạn 1			Đoạn 2		Đoạn 3			Đoạn 4		Đoạn 5	

Hình 6.11: Biểu diễn đồ thị bằng danh sách kề cài đặt bằng mảng

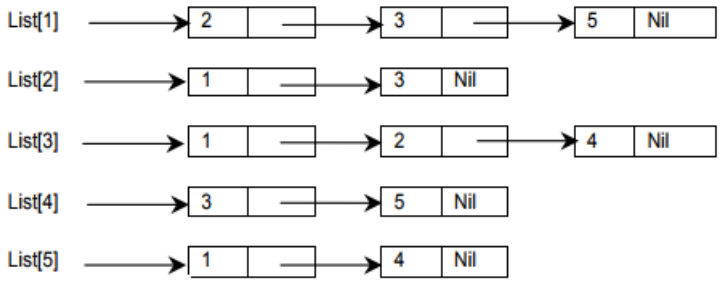
một đoạn nằm từ chỉ số nào đến chỉ số nào, ta có một mảng lưu vị trí riêng. Ta gọi mảng lưu vị trí đó là mảng Head. Head[i] sẽ bằng chỉ số đứng liền trước đoạn thứ i. Quy ước Head[n + 1] sẽ bằng m. Với đồ thị bên thì mảng VT[1..6] sẽ là: (0, 3, 5, 8, 10, 12) Như vậy đoạn từ vị trí Head[i] + 1 đến Head[i + 1] trong mảng A sẽ chứa

các đỉnh kề với đỉnh i.

Lưu ý rằng với đồ thị có hướng gồm m cung thì cấu trúc Forward Star cần phải đủ chứa m phần tử, với đồ thị vô hướng m cạnh thì cấu trúc Forward Star cần phải đủ chứa 2m phần tử.

Cách 2: Cài đặt bằng danh sách móc nối.

Dùng các danh sách móc nối: Với mỗi đỉnh i của đồ thị, ta cho tương ứng với nó một danh sách móc nối các đỉnh kề với i, có nghĩa là tương ứng với một đỉnh i, ta phải lưu lại List[i] là chốt của một danh sách móc nối. Ví dụ với đồ thị trên, danh sách móc nối sẽ là:



Hình 6.12: Biểu diễn đồ thị bằng danh sách kề cài đặt bằng danh sách móc nối.

Ưu điểm của danh sách kề:

Đối với danh sách kề, việc duyệt tất cả các đỉnh kề với một đỉnh v cho trước là hết sức dễ dàng, cái tên "danh sách kề" đã cho thấy rõ điều này. Việc duyệt tất cả các cạnh cũng đơn giản vì một cạnh thực ra là nối một đỉnh với một đỉnh khác kề nó.

Nhược điểm của danh sách kề:

Về lý thuyết, so với hai phương pháp biểu diễn trên, danh sách kề tốt hơn hẳn. Chỉ có điều, trong trường hợp cụ thể mà ma trận kề hay danh sách cạnh không thể hiện nhược điểm thì ta nên dùng ma trận kề (hay danh sách cạnh) bởi cài đặt danh sách kề có phần dài dòng hơn.

Trong nhiều trường hợp đủ không gian lưu trữ, việc chuyển đổi từ cách biểu diễn này sang cách biểu diễn khác không có gì khó khăn. Còn đối với mỗi thuật toán, với mỗi cách biểu diễn có thể gắn gọn hơn hay dài dòng hơn. Do đó, với mục đích dễ hiểu, trong sách này, từ các phần sau sẽ lựa chọn phương pháp biểu diễn sao cho việc cài đặt đơn giản nhất nhằm nêu bật được bản chất thuật toán.

6.3 Tìm kiếm trên đồ thị

6.3.1 Bài toán tìm kiếm

Một bài toán quan trọng trong lý thuyết đồ thị là bài toán duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó. Vấn đề này đưa về một bài toán liệt kê mà yêu cầu của nó là không được bỏ sót hay lặp lại bất kỳ đỉnh nào. Chính vì vậy mà ta phải xây dựng những thuật toán cho phép duyệt một cách hệ thống các đỉnh, những thuật toán như vậy gọi là những thuật toán tìm kiếm trên đồ thị và ở đây ta quan tâm đến hai thuật toán cơ bản nhất: thuật toán tìm kiếm theo chiều sâu và thuật toán tìm kiếm theo chiều rộng cùng với một số ứng dụng của chúng.

6.3.2 Tìm kiếm theo chiều sâu

Tư tưởng của thuật toán có thể trình bày như sau: Trước hết, mọi đỉnh x kề với S tất nhiên sẽ đến được từ S. Với mỗi đỉnh x kề với S đó thì tất nhiên những đỉnh y kề với x cũng đến được từ S...Điều đó gợi ý cho ta viết một thủ tục đệ quy DFS(u) mô tả việc duyệt từ đỉnh u bằng cách thông báo thăm đỉnh u và tiếp tục quá trình duyệt DFS(v) với v là một đỉnh chưa thăm kề với u.

Cài đặt đệ quy:

Để không một đỉnh nào bị liệt kê tới hai lần, ta sử dụng kỹ thuật đánh dấu, mỗi lần thăm một đỉnh, ta đánh dấu đỉnh đó lại để các bước duyệt đệ quy kế tiếp không duyệt lại đỉnh đó nữa.

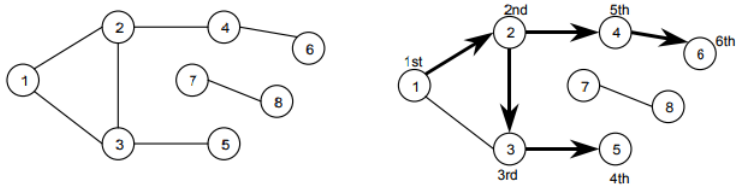
Để lưu lại đường đi từ đỉnh xuất phát S, trong thủ tục DFS(u), trước khi gọi đệ quy DFS(v) với v là một đỉnh kề với u mà chưa đánh dấu, ta lưu lại vết đường đi từ u tới v bằng cách đặt TRACE[v] := u, tức là TRACE[v] lưu lại đỉnh liền trước v trong đường đi từ S tới v. Khi quá trình tìm kiếm theo chiều sâu kết thúc, đường đi từ S tới F sẽ là: $F \leftarrow p_1 = Trace[F] \leftarrow p_2 = Trace[p_1] \leftarrow ... \leftarrow S$.

Giải thuật 1 Tìm kiếm đồ thị theo chiều sâu (sử dụng đệ quy).

Giải thuật: `DFS(u)`
Đầu vào: *G*, *u*
Đầu ra: Danh sách các đỉnh duyệt qua
1: Thông báo tới được u;
2: Đánh dấu u là đã thăm (có thể tới được từ S);
3: Xét mọi đỉnh v kề với u mà chưa thăm, với mỗi đỉnh v đó;
Trace[v] := u; *//Lưu vết đường đi, đỉnh mà từ đó tới v là u*
DFS(v); *//Gọi đệ quy duyệt tương tự đối với v*
end

Ví dụ:

Cho đồ thị G (Hình 6.13), đỉnh xuất phát S = 1, quá trình duyệt đệ quy có thể vẽ trên cây tìm kiếm DFS sau (Mũi tên $u \leftarrow v$ chỉ thao tác đệ quy: DFS(u) gọi DFS(v)).



Hình 6.13: Thứ tự duyệt đồ thị dùng DFS.

1. Đỉnh 2 và 3 kề với đỉnh 1, nhưng DFS(1) gọi đệ quy tới DFS(2) mà không gọi DFS(3). Do DFS(1) tìm thấy 2 trước và gọi DFS(2). Trong DFS(2) xét tất cả các đỉnh kề với 2 mà chưa đánh dấu nên gọi DFS(3), khi đó 3 đã bị đánh dấu nên khi kết thúc quá trình đệ quy DFS(2), lùi về DFS(1) thì đỉnh 3 đã được thăm nên DFS(1) sẽ không gọi DFS(3) nữa.

2. Nếu F = 5 thì đường đi từ 1 tới 5 trong chương trình trên sẽ là: $5 \leftarrow 3 \leftarrow 2 \leftarrow 1$. DFS(5) do DFS(3) gọi nên Trace[5] = 3. DFS(3) do DFS(2) gọi nên Trace[3] = 2. DFS(2) do DFS(1) gọi nên Trace[2] = 1.

Cài đặt chương trình (tham khảo)

```
#include<iostream>
#include<conio.h>
using namespace std;
#define MAX 100
#define TRUE 1
#define FALSE 0
int G[MAX][MAX], n, chuaxet[MAX];
void init(){
    freopen("DFS.in", "r", stdin); //Du lieu do thi luu trong file DFS.in
    cin>>n;
    cout<<"So dinh cua ma tran n = "<<n<<endl;
    //nhap ma tran ke
    for(int i=1; i<=n;i++){
        for(int j=1; j<=n;j++){
        cin>>G[i][j];
        }
    }
}
//Duyet chieu sau - thu tuc de quy
void DFS(int G[][MAX], int n, int v, int chuaxet[]){
    cout<<"Duyet dinh : "<<v<<endl;
    int u;
    chuaxet[v]=FALSE;
    for(u=1; u<=n; u++){
        if(G[v][u]==1 && chuaxet[u])
            DFS(G,n, u, chuaxet);
         }
}
void main(void){
    init();
    for(int i=1; i<=n; i++)
```

```
            chuaxet[i]=TRUE;
        for(int i=1; i<=n;i++)
            if(chuaxet[i]) DFS( G,n, i, chuaxet);
    _getch();
    }
```

Cài đặt không dùng đệ quy

Giải thuật 2 Tìm kiếm đồ thị theo chiều sâu (không dùng đệ quy).

Giải thuật: DFS(*u*)
Đầu vào: *G*, *u*
Đầu ra: Danh sách các đỉnh duyệt qua

1: Thăm S, đánh dấu S đã thăm;
2: Đẩy S vào ngăn xếp;
3: **repeat**
4: **if** (u có đỉnh kề chưa thăm) **then**
5: Chọn 1 đỉnh kề với u là chưa được thăm;
6: Thông báo thăm v;
7: Đẩy u trở lại ngăn xếp; *//Giữ lại địa chỉ quay lui*
8: Đẩy tiếp v vào ngăn xếp; *//Đường dẫn duyệt theo chiều sâu được "nối" thêm v*
9: **end if**
nếu u không có đỉnh kề chưa thăm thì ngăn xếp sẽ ngắn lại, tương ứng với quá trình lùi về của đường dẫn DFS
10: **until** (Ngăn xếp rỗng);
end

Ví dụ:

Cho đồ thị G (Hình 6.13), đỉnh bắt đầu S = 1, Quá trình thực hiện thủ tục tìm kiếm theo chiều sâu dùng ngăn xếp (với các nút từ 1 đến 6) được mô tả trong Bảng 6.1.

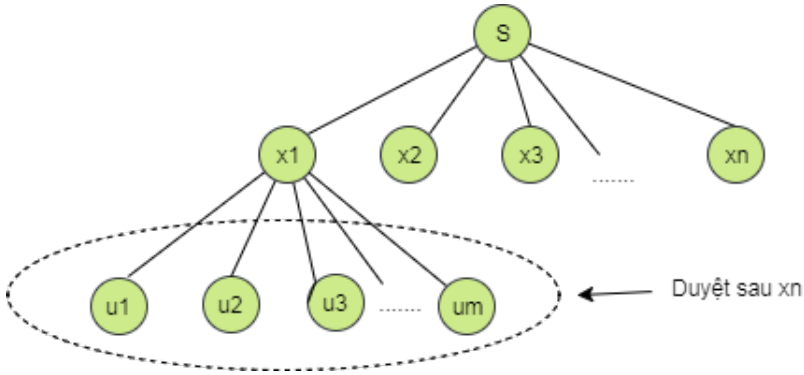
6.3.3 Tìm kiếm theo chiều rộng

Việc thăm một đỉnh sẽ lên lịch duyệt các đỉnh kề với nó nhất. Như vậy, thứ tự duyệt là ưu tiên chiều rộng. Ví dụ: Trong Hình 6.14, bắt đầu thăm đỉnh S. Việc thăm đỉnh S sẽ phát sinh thứ tự duyệt những đỉnh $(x_1, x_2, ..., x_p)$ kề với S. Chọn x_1 để thăm. Khi thăm đỉnh x_1 sẽ phát sinh yêu cầu duyệt những đỉnh $(u_1, u_2..., u_q)$

Bảng 6.1: Ví dụ về duyệt theo chiều sâu

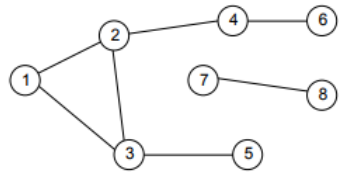
Bước lặp	***Ngăn xếp***	***u***	***v***	***Ngăn xếp sau mỗi bước***	***Giải thích***
1	(1)	1	2	(1, 2)	thăm 2
2	(1, 2)	2	3	(1, 2, 3)	thăm 3
3	(1, 2, 3)	3	5	(1, 2, 3, 5)	thăm 5
4	(1, 2, 3, 5)	5	Không có	(1, 2, 3)	Lùi lại
5	(1, 2, 3)	3	Không có	(1, 2)	Lùi lại
6	(1, 2)	2	4	(1, 2, 4)	thăm 4
7	(1, 2, 4)	4	6	(1, 2, 4, 6)	thăm 6
8	(1, 2, 4, 6)	6	Không có	(1, 2, 4)	Lùi lại
9	(1, 2, 4)	4	Không có	(1, 2)	Lùi lại
10	(1, 2)	2	Không có	(1)	Lùi lại
11	(1)	1	Không có	∅	Lùi hết, xong

kề với x_1. Duyệt theo chiều rộng nên đỉnh x_2 sẽ được thăm. Các đỉnh u chỉ được thăm khi tất cả những đỉnh x đã duyệt xong. Thứ tự duyệt đỉnh sau khi đã thăm x_1 sẽ là: $(x_2, x_3..., x_p, u_1, u_2, ..., u_q)$.



Hình 6.14: Duyệt cây theo chiều rộng BFS.

Ví dụ: Xét đồ thị dưới đây, Đỉnh xuất phát S = 1.



Hình 6.15: Cây DFS.

Giải thuật 3 Tìm kiếm đồ thị theo chiều rộng.

Giải thuật: `BFS(u)`
Đầu vào: `G`, `u`
Đầu ra: Danh sách các đỉnh duyệt qua

```
1: Khởi tạo:
   Các đỉnh ở trạng thái chưa đánh dấu, trừ đỉnh xuất phát S.
   Một hàng đợi (Queue), ban đầu chỉ có một phần tử là S.
   //Hàng đợi dùng để chứa các đỉnh sẽ được duyệt theo thứ tự ưu tiên chiều rộng.
2: Đẩy S vào ngăn xếp;
3: repeat
4:    Lấy u khỏi hàng đợi;
5:    Thông báo thăm u;  //Bắt đầu việc duyệt đỉnh u
6:    if (đỉnh v kề với u chưa được đánh dấu) then
7:       Ghi nhận vết đường đi từ u tới v; //Có thể làm chung với việc đánh dấu
8:       Đẩy v vào hàng đợi; //v sẽ chờ được duyệt tại những bước sau
9:    end if
10: until (Hàng đợi rỗng:);
11: Truy vết tìm đường đi;
end
```

Bảng 6.2: Ví dụ duyệt đồ thị theo chiều rộng

Hàng đợi Q	***u lấy từ Q***	***Q (sau khi lấy u)***	***Đỉnh v kề u (chưa lên lịch)***	***Q sau khi đẩy v vào***
(1)	1	∅	2, 3	(2, 3)
(2, 3)	2	(3)	4	(3, 4)
(3, 4)	3	(4)	5	(4, 5)
(4, 5)	4	(5)	6	(5, 6)
(5, 6)	5	(6)	Không có	(6)
(6)	6	∅	Không có	∅

6.3.4 Độ phức tạp của giải thuật DFS và BFS

Quá trình tìm kiếm trên đồ thị bắt đầu từ một đỉnh có thể thăm tất cả các đỉnh còn lại, khi đó cách biểu diễn đồ thị có ảnh hưởng lớn tới chi phí về thời gian thực hiện giải thuật:

- Trong trường hợp biểu diễn đồ thị bằng danh sách kề, thuật toán BFS và DFS đều có độ phức tạp tính toán là O(n + m) = O(max(n, m)). Đây là cách cài đặt tốt nhất.

- Nếu biểu diễn đồ thị bằng ma trận kề thì độ phức tạp tính toán

là $O(n + n^2) = O(n^2)$.

- Nếu biểu diễn đồ thị bằng danh sách cạnh, thao tác duyệt những đỉnh kề với đỉnh u sẽ dẫn tới việc phải duyệt qua toàn bộ danh sách cạnh, đây là cài đặt tồi nhất, độ phức tạp tính toán là $O(n \times m)$.

6.4 Bài toán tìm đường đi ngắn nhất

Trong các ứng dụng thực tế, chẳng hạn trong mạng lưới giao thông đường bộ, đường thuỷ hoặc đường không. Người ta không chỉ quan tâm đến việc tìm đường đi giữa hai địa điểm mà còn phải lựa chọn một hành trình tiết kiệm nhất (theo tiêu chuẩn không gian, thời gian hay chi phí). Khi đó phát sinh yêu cầu tìm đường đi ngắn nhất giữa hai đỉnh của đồ thị. Bài toán đó phát biểu dưới dạng tổng quát như sau:
Cho đồ thị có trọng số $G = (V, E)$, hãy tìm một đường đi ngắn nhất từ đỉnh xuất phát $S \in V$ đến đỉnh đích $F \in V$. Độ dài của đường đi này ta sẽ ký hiệu là $d[S, F]$ và gọi là khoảng cách từ S đến F. Nếu như không tồn tại đường đi từ S tới F thì ta sẽ đặt khoảng cách đó $= +\infty$. Trong giáo trình này, ta xét bài toán tìm đường đi ngắn nhất từ đỉnh S tới đỉnh F trên đơn đồ thị đơn có hướng $G = (V, E)$ có n đỉnh và m cung. Trong trường hợp đơn đồ thị vô hướng với trọng số không âm, bài toán tìm đường đi ngắn nhất có thể dẫn về bài toán trên đồ thị có hướng bằng cách thay mỗi cạnh của nó bằng hai cung có hướng ngược chiều nhau.

Thuật toán do Dijkstra - Đồ thị trọng số trên cung không âm

Trong trường hợp trọng số trên các cung không âm, thuật toán do Dijkstra hoạt động hiệu quả. Thuật toán Dijkstra (E.Dijkstra -

1959) có thể mô tả như sau:

Giải thuật 4 Thuật toán Dijkstra.

Giải thuật: `Dijkstra(G)`
Đầu vào: `G = (V, E), S, F`
Đầu ra: Đường đi ngắn nhất từ S tới F

1: Khởi tạo:
$d[v] = c[S, v]$ ($d[v]$ là độ dài đường đi ngắn nhất từ S tới v, $v \in V$).
(Mỗi đỉnh v có hai trạng thái: tự do (còn tối ưu được) hay cố định ($d[v]$ đã bằng độ dài đường đi ngắn nhất từ S tới v nên không thể tối ưu thêm). Dùng Free[v] để đánh dấu trạng thái của v, khởi tạo $d[v]$ đều tự do).
$Free[v] = TRUE$
2: **repeat**
3: Cố định nhãn:
Chọn trong các đỉnh có nhãn tự do, lấy ra đỉnh u là đỉnh có $d[u]$ nhỏ nhất, và cố định nhãn của đỉnh u.
4: Sửa nhãn:
Dùng đỉnh u, xét tất cả những đỉnh v và sửa lại các d[v] theo công thức:
$d[v] := min(d[v], d[u] + c[u, v])$
5: **until** (đỉnh đích F được cố định nhãn (tìm được đường đi ngắn nhất từ S tới F); hoặc tại thao tác cố định nhãn, tất cả các đỉnh tự do đều có nhãn là $+\infty$ (không tồn tại đường đi));
6: Kết hợp với việc lưu vết đường đi trên từng bước sửa nhãn, thông báo đường đi ngắn nhất tìm được hoặc cho biết không tồn tại đường đi ($d[F] = +\infty$).

end

Có thể đặt câu hỏi, ở thao tác 1, tại sao đỉnh u như vậy được cố định nhãn, giả sử d[u] còn có thể tối ưu thêm được nữa thì tất phải có một đỉnh t mang nhãn tự do sao cho $d[u] > d[t] + c[t, u]$. Do trọng số $c[t, u]$ không âm nên $d[u] > d[t]$, trái với cách chọn $d[u]$ là nhỏ nhất. Tất nhiên trong lần lặp đầu tiên thì S là đỉnh được cố định nhãn do $d[S] = 0$.

6.5 Bài toán cây khung nhỏ nhất

Cho $G = (V, E)$ là đồ thị vô hướng liên thông có trọng số, T là cây khung nhỏ nhất của G. Trọng số của T bằng tổng trọng số các cạnh trong T. Yêu cầu trong số các cây khung của G, tìm cây khung có trọng số nhỏ nhất T. Sau đây ta sẽ xét hai thuật toán

thông dụng để giải bài toán cây khung nhỏ nhất của đơn đồ thị vô hướng có trọng số.

6.5.1 Thuật toán Kruskal

Giải thuật 5 Thuật toán Kruscal.

Giải thuật: `Kruscal(G)`
Đầu vào: Đơn đồ thị vô hướng G = (V, E)
Đầu ra: Cây khung nhỏ nhất T

1: Sắp xếp các cạnh của đồ thị G theo thứ tự tăng dần của trọng số cạnh
2: Khởi tạo cây khung T = ∅
3: **repeat**
Duyệt trong danh sách các cạnh đã được sắp xếp, từ cạnh có trọng số nhỏ đến cạnh có trọng số lớn để tìm ra cạnh mà khi bổ sung nó vào T không tạo thành chu trình trong T thì kết nạp cạnh đó vào T;
4: **until** (T gồm n-1 cạnh) hoặc chưa kết nạp đủ n - 1 cạnh nhưng cứ kết nạp thêm một cạnh bất kỳ trong số các cạnh còn lại thì sẽ tạo thành chu trình đơn trong T. Trong trường hợp này đồ thị G là không liên thông).

end

6.5.2 Thuật toán Prim

Thuật toán Kruskal hoạt động chậm trong đồ thị dày (nhiều cạnh). Khi đó người ta thường sử dụng thuật toán Prim.

Xét về độ phức tạp tính toán, thuật toán Prim có độ phức tạp là $O(n^2)$. Tương tự thuật toán Dijkstra, nếu kết hợp thuật toán Prim với cấu trúc Heap sẽ được một thuật toán với độ phức tạp O((m+n)logn).

Giải thuật 6 Thuật toán Prim.

Giải thuật: `Prim(G)`
Đầu vào: Đơn đồ thị vô hướng G = (V, E)
Đầu ra: Cây khung nhỏ nhất T

1: Cài đặt G bằng ma trận trong số C;
Qui ước: $c[u, v] = +\infty$ nếu (u, v) không là cạnh; Gọi d[v] là khoảng cách từ v tới T; với 1 cây T trong G và một đỉnh v, gọi khoảng cách từ v tới T là trọng số nhỏ nhất trong số các cạnh nối v với một đỉnh nào đó trong T:
$d[v] = minc[u, v]|u \in T$
2: Khởi tạo cây khung T = 1 Free[v] = TRUE nếu như đỉnh v chưa bị kết nạp vào T;
$d[1] = 0$ $d[2] = d[3] = \doteq d[n] = +\infty$
3: **repeat**
Chọn các đỉnh trong T ra một đỉnh gần T nhất (chọn đỉnh u nào ngoài T và có d[u] nhỏ nhất), kết nạp đỉnh đó vào T đồng thời kết nạp luôn cả cạnh tạo ra khoảng cách gần nhất đó:
d[v]mới := min(d[v]cũ, c[u, v];
4: **until** (Đã kết nạp được tất cả n đỉnh thì ta có T là cây khung nhỏ nhất hoặc chưa kết nạp được hết n đỉnh nhưng mọi đỉnh ngoài T đều có khoảng cách tới T là $+\infty$).

end

BÀI TẬP

1. Viết chương trình tạo đồ thị với số đỉnh ≤ 100, trọng số các cạnh là các số được sinh ngẫu nhiên. Ghi vào file dữ liệu MINTREE.INP đúng theo khuôn dạng quy định. So sánh kết quả làm việc của thuật toán Kruskal và thuật toán Prim về tính đúng đắn và về tốc độ.

2. Hãy cài đặt thuật toán Kruscal với cấu trúc Heap chứa các đỉnh trên khung T.

3. Hãy cài đặt thuật toán Prim với cấu trúc dữ liệu Heap chứa các đỉnh ngoài cây khung T.

4. Trên một nền phẳng với hệ toạ độ Decattes vuông góc đặt n máy tính, máy tính thứ i được đặt ở toạ độ (Xi, Yi). Cho phép nối thêm các dây cáp mạng nối giữa từng cặp máy tính. Chi phí nối một dây cáp mạng tỉ lệ thuận với khoảng cách giữa

hai máy cần nối. Hãy tìm cách nối thêm các dây cáp mạng để cho các máy tính trong toàn mạng là liên thông và chi phí nối mạng là nhỏ nhất.

5. Tương tự như bài trên, nhưng ban đầu đã có sẵn một số cặp máy nối rồi, cần cho biết cách nối thêm ít chi phí nhất.

6. Hệ thống điện trong thành phố được cho bởi n trạm biến thế và các đường dây điện nối giữa các cặp trạm biến thế. Mỗi đường dây điện e có độ an toàn là p(e). ở đây $0 < p(e) \leq 1$. Độ an toàn của cả lưới điện là tích độ an toàn trên các đường dây. Ví dụ như có một đường dây nguy hiểm: p(e) = 1% thì cho dù các đường dây khác là tuyệt đối an toàn (độ an toàn = 100%) thì độ an toàn của mạng cũng rất thấp (1%). Hãy tìm cách bỏ đi một số dây điện để cho các trạm biến thế vẫn liên thông và độ an toàn của mạng là lớn nhất có thể.

Chương 7

Mô hình dữ liệu tập hợp

7.1 Khái niệm tập hợp

Trong toán học, tập hợp là một cấu trúc rời rạc, cơ bản để từ đó xây dựng lên các cấu trúc rời rạc khác như: Các tổ hợp – là những tập hợp không sắp thứ tự của các phần tử; đồ thi – là tập hợp các đỉnh và các cạnh nối các đỉnh đó, cây – là một tập hợp các đỉnh và các cạnh nối hai đỉnh có quan hệ phân cấp, ... Tập hợp được dùng để mô hình hoá hay biểu diễn một nhóm bất kỳ các đối tượng không lặp lại, không thứ tự trong thế giới thực, do đó nó đóng vai trò rất quan trọng trong mô hình hoá dữ liệu cũng như trong thiết kế thuật toán.

Một tập hợp có thể là vô hạn hoặc hữu hạn, các phương pháp mô tả tập hợp như (1) dùng biểu đồ Ven là một đường cong khép

kín, các điểm trong đường cong đó chỉ các phần tử của tập hợp; (2) liệt kê tất cả các phần tử của tập hợp A; (3) mô tả đặc trưng của các phần tử trong tập hợp, ví dụ: A là một tập các số nguyên chẵn.

7.2 Mô hình dữ liệu tập hợp

Trong thiết kế thuật toán, có thể sử dụng tập hợp như một mô hình dữ liệu. Khi đó, ngoài các phép toán hợp, giao, hiệu, chúng ta phải cần đến nhiều các phép toán khác. Sau đây chúng ta sẽ đưa ra một số phép toán cơ bản quan trọng nhất, các phép toán này sẽ được mô tả bởi các thủ tục hoặc hàm:

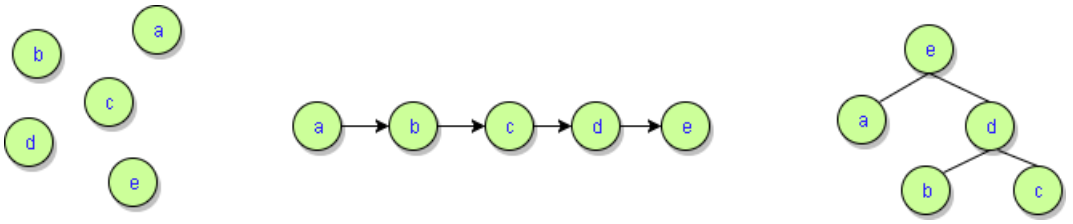
1. Thủ tục **MAKENULL(A)**: khởi tạo tập A rỗng
2. Thủ tục **UNION(A,B,C)**: phép lấy hợp của hai tập A và B và trả ra kết quả là tập hợp $C = A \cup B$.
3. Thủ tục **INTERSECTION(A,B,C)**: phép lấy giao của hai tập A và B và trả ra kết quả là tập hợp $C = A \cap B$.
4. Thủ tục **DIFFERENCE(A,B,C)**: phép lấy hợp của hai tập A và B và trả ra kết quả là tập hợp $C = A \setminus B$
5. Hàm **MEMBER(x,A)**: kiểm tra thành phần x có thuộc tậpA hay không. Nếu x thuộc A thì hàm cho kết quả là đúng, ngược lại cho kết quả sai.
6. Thủ tục **INSERTSET(x,A)**: thêm x vào tập hợp A
7. Thủ tục **DELETESET(x,A)**: xoá x khỏi tập hợp A
8. Thủ tục **ASSIGN(A,B)**: gán A cho B (tức là B:=A)
9. Hàm **MIN(A)**: tìm phần tử bé nhất trong tập A

10. Hàm **EQUAL(A,B)**: kiểm tra A=B ngược lại cho kết quả FALSE
11. Thủ tục **INPUT(A)**: nhập dữ liệu cho tập A
12. Thủ tục **OUTPUT(A)**: xuất dữ liệu tập hợp A lên màn hình

7.3 Các cấu trúc dữ liệu tập hợp

Tập hợp có thể được cài đặt sử dụng các cấu trúc dữ liệu khác . Một số cấu trúc được thiết kế để cải tiến hiệu quả thời gian thực thi của các thao tác trong khi các cấu trúc khác lại đặt trọng tâm hơn vào không gian lưu trữ, việc lựa chọn cấu trúc dữ liệu tập hợp phù hợp với bài toán phụ thuộc nhiều vào các thao tác cụ thể và các đặc tính của tập hợp (ví dụ kích cỡ) mà bài toán quan tâm.

Tập hợp có thể được biểu diễn thông qua các cấu trúc dữ liệu hàng đợi, danh sách, cây, bản đồ, bảng băm, ... Hình 7.1 mô tả hình ảnh một tập hợp gồm 5 ký tự được biểu diễn bởi hai cấu trúc dữ liệu là danh sách và cây nhị phân.



Hình 7.1: Một số cấu trúc dữ liệu tập hợp.

Một số cấu trúc dữ liệu tập hợp cơ bản:

- Biểu diễn tập hợp bởi véc tơ bit;
- Biểu diễn tập hợp bởi danh sách liên kết;
- Biểu diễn tập hợp bởi cây;

- Biểu diễn tập hợp bởi bảng băm.

Với mỗi dạng biểu diễn tập hợp hay cấu trúc dữ liệu tập hợp, các thao tác trên tập hợp được cài đặt tương ứng với cấu trúc dữ liệu đó. Ví dụ, với tập hợp được biểu diễn bởi danh sách liên kết đơn, phép toán **UNION(A,B,C)** tương như phép toán nối hai danh sách liên kết đơn A, B thành danh sách liên kết đơn C với sự loại bỏ các phần tử trùng lặp. Bạn đọc có thể tham khảo cấu trúc dữ liệu danh sách liên kết đơn trong chương 4 để cài đặt phép toán này. Phép toán **MEMBER(x,A)** sẽ tương tự như phép toán tìm kiếm phần tử x trong danh sách liên kết A, ...

7.3.1 Cài đặt tập hợp bởi vectơ bit

Giả sử, xét tập hợp A gồm các số nguyên thuộc phạm vi từ 1 đến n (hoặc được mã hóa thành các số nguyên thuộc phạm vi từ 0 -> n-1). Khi đó ta có thể dùng véc tơ bit (mảng boolean) để biểu diễn tập A : (A[0], A[1], ... ,A[n-1]), trong đó thành phần thứ i : A[i] = 1/true nếu $i \in A$, $A[i] = 0/false$ nếu $i \notin A$.

Ví dụ: Giả sử các phần tử của tập hợp được lấy trong các số nguyên từ 1 đến 10, khi đó tập hợp được biểu diễn bởi một mảng một chiều có 10 phần tử với các giá trị phần tử thuộc kiểu logic. Chẳng hạn tập hợp $A = \{1, 3, 5, 8\}$ được biểu diễn trong mảng có 10 phần tử như sau:

Bảng 7.1: Biểu diễn tập hợp A bởi véc tơ bit gồm 10 phần tử

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

- **Khai báo tập hợp:**

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

```
#define N 100 // kich co tap hop
typedef int item;// kieu phan tu cua tap hop
typedef item Set[N];
```

- **Một số thao tác cơ bản:**

  0 - Thủ tục **MAKENULL(A)**: khởi tạo tập A rỗng.

```
void MakeNull(Set A) {
   for(int i=0; i<N; i++)
      A[i]=0;
    }
```

1- Thủ tục **UNION(A,B,C)** nhận vào 3 tham số là A,B,C; Thực hiện phép toán lấy hợp của hai tập A và B và trả ra kết quả là tập hợp $C = A \cup B$.

```
void Union(Set A, Set B, Set C) {
   for (int i=0; i<N; i++)
       if ((A[i]==1)||(B[i]==1))
          C[i]=1;
       else
          C[i]=0;
}
```

2 -Thủ tục **INTERSECTION(A,B,C)** nhận vào 3 tham số là A,B,C; Thực hiện phép toán lấy giao của hai tập A và B và trả ra kết quả là tập hợp $C = A \cap B$.

```
void Intersection(Set A,Set B, Set C) {
   for (int i=0; i<N; i++)
       if ((A[i]==1)&amp;&amp;(B[i]==1))
          C[i]=1;
       else
          C[i]=0;
}
```

3 - Thủ tục **DIFFERENCE(A,B,C)** nhận vào 3 tham số là A,B,C; Thực hiện phép toán lấy hợp của hai tập A và B và trả ra kết quả là tập hợp $C = A \setminus B$

```
void Difference(Set A, Set B, Set C) {
   for (int i=0; i<N; i++)
```

```
        if ((A[i]==1)&amp;&amp;(B[i]==1))
            C[i]=0;
        else
            C[i]=A[i];
}
```

4 - Hàm **MEMBER(x,A)** cho kết quả kiểu logic (đúng/sai) tùy theo x có thuộc A hay không. Nếu $x \in A$ thì hàm cho kết quả là đúng, ngược lại cho kết quả sai.

```
item Member(item x, Set A) {
   return A[x]==1;
}
```

5 - Thủ tục **INSERTSET(x,A)** thêm x vào tập hợp A

```
void Insert(item x, Set A) {
   A[x] = 1;
}
```

6 - Thủ tục **DELETESET(x,A)** xoá x khỏi tập hợp A

```
void Delete(item x, Set A) {
   A[x] = 0;
}
```

7 - Thủ tục **ASSIGN(A,B)** gán các phần tử A cho B

```
void Assign( Set A, Set B) {
   for (int i=0; i<N; i++)
     B[i] = A[i];
}
```

8 - Hàm **MIN(A)** cho phần tử bé nhất trong tập A

```
item Min(Set A){
   int i = 0, Found = -1;
   while(i<N &amp;&amp; Found ==-1)
      if(A[i]==1)
         Found = i;
      else
         i++;
  return Found;
}
```

9 - Hàm **EQUAL(A,B)** cho kết quả TRUE nếu A=B ngược lại cho kết quả FALSE

```
item Equal(Set A, Set B) {
   int i = 0;
   Found = 0;
   while(i<N &amp;amp;&amp;amp; Found == 0)
      if(A[i] != B[i])
          Found = 1;
      else
         i++;
   return Found;
}
```

10 - Thủ tục **INPUT(A)**: nhập dữ liệu cho tập A

```
void Input(Set A) {
   int x = 0;
   printf("\nNhap den -1 thi ket thuc: ");
   do {
      scanf("%d",&amp;amp;x);
      if(x!=-1)
         A[x] = 1;
   }while(x!=-1);
}
```

11 - Thủ tục **OUTPUT(A)**: xuất dữ liệu tập hợp A lên màn hình

```
void Output(Set A) {
   for (int i=0; i<MaxLength; i++)
   if(A[i]==1)
      printf("%3d",i);
}
```

Nhận xét:

Cấu trúc dữ liệu tập hợp vecto bit thuận lợi cho việc cài đặt các thao tác, độ phức tap tính toán của các thao tác là O(n), tuy nhiên nó thường không phù hợp với các tập hợp với kích cỡ quá lớn. Ví dụ thêm x vào A chỉ việc cho A[x-1] = true , để xác định xem x có là phần tử của tập A không ta chỉ cần biết A[x] là 1/true hay 0/false.

7.3.2 Cài đặt bởi danh sách liên kết

Tập hợp cũng có thể cài đặt bằng danh sách liên kết, trong đó mỗi phần tử của danh sách là một thành viên của tập hợp. Không như biểu diễn bằng vectơ bít, sự biểu diễn này dùng kích thước bộ nhớ tỉ lệ với số phần tử của tập hợp chứ không phải là kích thước đủ lớn cho toàn thể các tập hợp đang xét. Hơn nữa, ta có thể biểu diễn một tập hợp bất kỳ. Mặc dù thứ tự của các phần tử trong tập hợp là không quan trọng nhưng nếu một danh sách liên kết có thứ tự nó có thể trợ giúp tốt cho các phép duyệt danh sách. Chẳng hạn nếu tập hợp A được biểu diễn bằng một danh sách có thứ tự tăng thì hàm MEMBER(x,A) có thể thực hiện việc so sánh x một cách tuần tự từ đầu danh sách cho đến khi gặp một phần tử $y \geq x$ chứ không cần so sánh với tất cả các phần tử trong tập hợp.

Một ví dụ khác, chẳng hạn ta muốn tìm giao của hai tập hợp A và B có n phần tử. Nếu A,B biểu diễn bằng các danh sách liên kết chưa có thứ tự thì để tìm giao của A và B ta phải tiến hành như sau: for (mỗi x thuộc A), duyệt danh sách B xem x có thuộc B không. Nếu có thì x thuộc giao của hai tập hợp A và B. Rõ ràng quá trình này có thể phải cần đến n x m phép kiểm tra (với n,m là độ dài của A và B).

Nếu A, B được biểu diễn bằng danh sách có thứ tự tăng thì đối với một phần tử $e \in A$ ta chỉ tìm kiếm trong B cho đến khi gặp phần tử $x \geq e$. Quan trọng hơn nếu f đứng ngay sau e trong A thì để tìm kiếm f trong B ta chỉ cần tìm từ phần tử x trở đi chứ không phải từ đầu danh sách lưu trữ tập hợp B.

Việc cài đặt tập hợp bởi danh sách liên kết sẽ khắc phục hạn chế về không gian khi sử dụng mảng. Tuy nhiên trong cách cài đặt này, việc thực hiện các phép toán trên tập hợp sẽ phức tạp hơn.

Khai báo cấu trúc

```
typedef int item;
typedef struct Node{
   item info;
   Node *Next;
}
typedef Node *pointer;
typedef pointer Set;
```

7.4 Từ điển (dictionary)

7.4.1 Khái niệm

Từ điển là mô hình dữ liệu tập hợp, nhưng chỉ xét đến các phép toán **Insert** (thêm một phần từ vào tập hợp), **Delete** (loại bỏ một phần tử nào đó khỏi tập hợp), **Search** (tìm xem trong tập hợp có chứa một phần tử nào đó không). Do đó, từ điển được xem là một kiểu dữ liệu trừu tượng được tổ chức sao cho việc tìm kiếm, thêm và bớt phần tử có phần hiệu quả nhất gọi là từ điển. Chúng ta cũng chấp nhận MakeNullSet như là phép khởi tạo cấu trúc từ điển.

7.4.2 Các phương pháp cài đặt từ điển

Từ điển là một tập hợp, do đó chúng ta phải sử dụng các phương pháp cài đặt tập hợp để cài đặt từ điển:

- Bằng véc tơ bit: Sử dụng phương pháp này khi từ điển là tập hợp gồm các phần tử có thể dùng làm tập chỉ số cho mảng (thuộc kiểu dữ liệu vô hướng đếm được hoặc được mã hóa thành kiểu dữ liệu vô hướng đếm được)

- Bằng danh sách (kế tiếp hoặc móc nối). Cấu trúc này thường không thuận lợi cho phép tìm kiếm(với danh sách móc nối vì phải truy cập tuần tự -> chí thích hợp khi áp dụng phương pháp tìm kiếm tuần tự), không thuận lợij cho phép toán thêm

vào và lấy ra (với danh sách kế tiếp – vì có hiện tượng co, dãn, dịch chuyển các phàn tử khác)

Giả sử từ điển là danh sách được sắp thứ tự tuyến tính -> sử dụng cây tìm kiếm nhị phân: Độ phức về mặt thời gian của các phép toán trên từ điển là lớn (tương tự như danh sách) trong trường hợp cây suy biến thành danh sách. Cũng có thể sử dụng cây cần bằng để biểu diễn tập hợp: Ưu điểm là không xảy ra trường hợp suy biến như cây TKNP, tuy nhiên các phép toán loại bỏ và xen vào trên cây cân bằng khá phức tạp, vì phải cân bằng lại cây – Về cây cân bằng, bạn đọc có thể tham khảo trong các tài liệu tham khảo của giáo trình.

Do đó, để biểu diễn từ điển thuận lợi cho các phép toán, và có thể áp dụng cho tập hợp có kích cỡ lớn (từ điểm thường có kích thước lớn), ta xét cách cài đặt khác đó là: sử dụng CTDL bảng băm để cài đặt từ điển.

7.4.3 Cấu trúc dữ liệu bảng băm

Bảng băm là một cấu trúc dữ liệu rất thích hợp cho việc biểu diễn một tập hợp có số phần tử lớn (ví dụ: từ điển), ở đó dữ liệu được lưu trữ theo định dạng mảng và mỗi giá trị có giá trị khóa duy nhất. Việc truy cập sẽ trở lên rất nhanh nếu ta biết chỉ số của phần tử cần tìm.

Bảng băm sử dụng mảng để lưu trữ các phần tử và sử dụng kỹ thuật băm để sinh khóa cho mỗi phần tử (khóa) được chèn vào bảng. Có 2 phương pháp băm khác nhau:

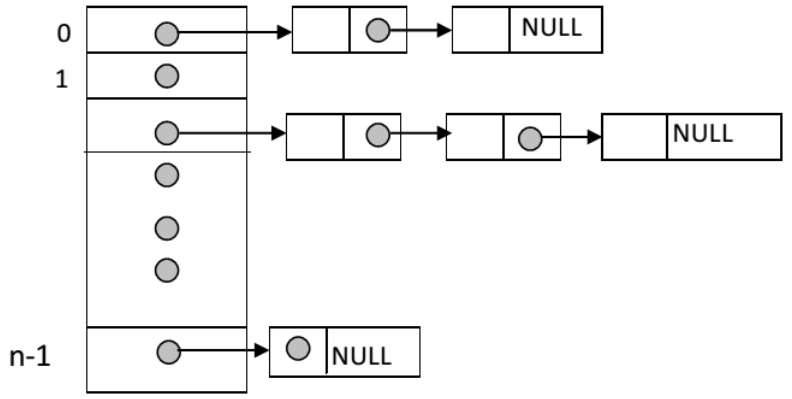
- Phương pháp Băm mở: Cho phép sử dụng một không gian không hạn chế để lưu giữ các phần tử của tập hợp.
- Phương pháp băm đóng: Sử dụng một không gian cố định và

do đó tập hợp được cài đặt phải có cỡ không vượt quá không gian cho phép

7.4.4 Cài đặt từ điển bằng bảng băm

- ***Định nghĩa bảng băm mở***

  Tư tưởng cơ bản của bảng Băm là: Phân chia một tập hợp đã cho thành một số cố định các lớp (N lớp) được đánh số 0, 1, . . . , N-1. Sử dụng mảng T với chỉ số chạy từ 0 đến N-1. Mỗi thành phần T[i] chứa một con trỏ, trỏ tới phần tử đầu tiên của danh sách chứa các phần tử của tập hợp thuộc lớp thứ i. Các phần tử của tập hợp thuộc mỗi lớp được tổ chức dưới dạng một danh sách liên kết, mỗi danh sách được gọi là con tr"rổ" , T được gọi là bảng băm (hash table, Hình 7.2).



Hình 7.2: Cấu trúc bảng băm mở.

Việc chia các phần tử của tập hợp vào các lớp được thực hiện bởi hàm băm (hashCose).

- ***Hàm băm***

  Băm là một kỹ thuật chuyển đổi một miền các giá trị khóa vào một miền gồm các chỉ số của mảng. Kỹ thuật này có thể xem

như một hàm (hàm băm) ánh xạ từ tập dữ khóa A đến các số nguyên (tập chỉ số) 0..N-1:

$$hashCost : A \longrightarrow 0..N-1; \tag{7.1}$$

Theo đó giả sử $x \in A$ thì h(x) là một số nguyên sao cho: $0 \leq hashCost(x) \leq N-1$.

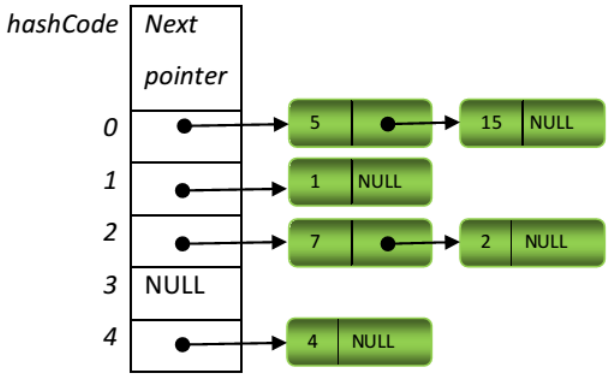
Hai tiêu chuẩn chính để lựa chọn một hàm băm là (1) hàm băm phải cho phép tính được dễ dàng và nhanh chóng giá trị Băm của mỗi khoá; (2) phải phân bố đều các khoá vào các rổ. Trong các phương pháp xây dựng hàm băm như băm gấp, băm bỏ bớt, .. thì phương pháp băm bằng cách chia lấy phần dư được xem là phổ biến.

Hàm băm lấy phần dư được thiết kế như sau:

```
int hashCode(int key){
   return
       key % SIZE;
}
```

Ví dụ: xét tập hợp $A = \{1, 4, 7, 2, 5, 15\}$, giả sử bảng băm A gồm 5 lớp, khi đó hàm băm là $hashCose(x) = x\%5$. Hình 7.3 mô tả cấu trúc của A được biểu diễn bởi bảng băm mở. Mỗi phần tử của bảng là một con trỏ (pointer) trỏ tới phần tử đầu tiên của rổ chứa các phần tử là chỉ số mảng của phần tử đó. Mỗi rổ được biểu diễn như một danh sách liên kết đơn, trong đó mỗi nút trong danh sách là một bản ghi gồm 2 trường: key (tương ứng với khóa của phần tử); next là con trỏ trỏ tới phần tử tiếp theo trong danh sách.

- ***Cài đặt từ điển bởi bảng băm mở***
  - Mỗi phần tử trong rổ có thể được mô tả theo cấu trúc sau:



Hình 7.3: Cấu trúc dữ liệu bảng băm mở biểu diễn từ điển/tập hợp A.

```
typedef struct member{
    char *key ;
    // cac truong thong tin khac (neu co)
    struct member *next ;
} NodeT;
```

Sau đó, bảng băm T là một mảng gồm N con trỏ, mỗi con trỏ trỏ tới phần tử đầu tiên của rổ tương ứng.

```
NodeT *T[N] ;
```

– Cài đặt các thao tác cơ bản của từ điển:

1- Hàm **SEARCH (key, T)**

Để kiểm tra xem phần tử có khóa x nào đó có trong từ điển T hay không, trước hết ta cần xác định giá trị băm của x, h(x). Theo cấu trúc của bảng băm thì khoá x sẽ nằm trong rổ được trỏ bởi T[h(x)]. Việc tìm kiếm phần tử có khóa x trong rổ này được thực hiện tương tự như tìm kiếm phần tử x trong danh sách liên kết đơn.

Giải thuật tìm kiếm trả về 1 nếu tìm thấy, ngược lại trả về 0:

```
struct int search(char *x, NodeT *T[N]){
```

```
    struct member *p;
    int h = hashCode(x); // gia tri bam
    p=T[h]; // bat dau tim kiem
    while(p != NULL){
        if(p->key == x)
            return 1;
        p=p->next;
    }
    return 0;
}
```

2- Thủ tục **INSERCH(key [,data], T)**

Để thêm một phần tử có khóa x vào từ điển T ta phải tính h(x) để xác định rổ sẽ chứa x, cấp phát ô nhớ chứa phần tử cần thêm, đưa phần tử cần thêm vào rổ. Vì ta không quan tâm đến thứ tự các phần tử trong mỗi rổ nên ta để đơn giản ta thêm phần tử mới ngay đầu rổ này.

```
struct void insert(char *x [,data], NoteT *T[N]){
    // cap phat o nho chua phan tu co khoa x
    struct member *p = (struct member*) malloc(sizeof(struct member));
    p->data = data; // neu co
    p->key = x;
    int h = hashCode(x);
    // them phan tu co khoa x vao ro (them vao dau ds tuong ung voi ro)
    p->next=T[h];
    T[h]=p;
}
```

3- Thủ tục **DELETE(key)**

Để xóa một phần tử có khóa x ra khỏi từ điển T, trước hết ta cần xác định mã băm của x để xác định được rổ chứa x. Sau đó ta tiến hành tìm kiếm x trong rổ đó, nếu x có trong rổ, việc xóa x được thực hiện tương tự như việc xóa một phần tử ra khỏi danh sách liên kết đơn.

```
struct void delete(char *x, NodeT *T[N]){
    struct member *p, *q;
    int h = hashCode(x);
    p=T[h]; // bat dau tim kiem
    if (p=NULL){
        printf("Ro RONG, x khong co trong T");
```

```
        } else{
            if(p->key=x){// xoa dau ds
                T[h]=p->next->next;
                free(p);
            } else{
                // tim thay x thi stop
                while(p->key != x){
                    q=p; // q tro den truoc ptu can xoa
                    p=p->next;
                }
                if(p=NULL){ // khong thay x
                    printf("x khong co trong T");
                }else{ // xoa x
                    p=p->next;
                    free(p)
                }
            }
        }
    }
```

- ***Bảng băm đóng***

  Bảng băm đóng lưu giữ các phần tử của từ điển ngay trong mảng (các phần tử của "rổ" i lưu trong chính phần tử thứ i của mảng) chứ không dùng mảng để lưu trữ các con trỏ trỏ tới đầu của các danh sách liên kết – "rổ".

  Tương tự như băm mở, trong bảng băm đóng "rổ" thứ i chứa phần tử có giá trị băm là i, nhưng vì có thể có nhiều phần tử có cùng giá trị băm nên ta sẽ gặp trường hợp sau: ta muốn đưa vào "rổ" i một phần tử x nhưng "rổ" này đã bị chiếm bởi một phần tử y nào đó => gây ra đụng độ. Như vậy khi thiết kế một bảng băm đóng ta phải có cách để giải quyết sự đụng độ này. Cách giải quyết đụng độ đó gọi là băm lại (rehash). Chi tiết về phương pháp băm đóng bạn đọc có thể tham khảo các tài liệu tham khảo đã chỉ ra trong chương này.

BÀI TẬP

1. Phân tích nêu ưu điểm và nhược điểm của các cấu trúc dữ liệu biểu diễn tập hợp.

2. Phương pháp biển diễn từ điển bởi bảng băm có ưu điểm nổi bật gì so với các phương pháp biểu diễn từ điển khác?

3. Viết chương trình cài đặt một từ điển Anh-Việt mini sử dụng cấu trúc dữ liệu bảng băm mở.

4. Giả sử ta xét các khóa là các dòng chữ cái La tinh. Người ta mã hóa chúng theo quy tắc: mỗi chữ cái được mã hóa bởi hai chữ số thập phân ứng với thứ tự của nó trong bộ chữ cái la tinh (từ 01 đến 26). Chẳng hạn từ khóa TRUNG được mã hóa bởi dải dãy số: 20 18 21 14 07. Khóa tương ứng là tổng của các số này. Giả sử địa chỉ rải (mã băm) ứng với khóa k được tính theo công thức h(k)=k mode N, N=7 là số rổ. Hãy

   - Xác định các khóa cho mỗi từ trong dòng chữ: UNIVERSITY OF INFORMATION AND COMMUNICATION TECHNOLOGY.
   - Đưa các khoa trên vào bảng băm mở và minh họa qua hình vẽ.

Tài liệu tham khảo

[1] Đinh Mạnh Tường. *Cấu trúc dữ liệu và thuật toán*, 2003.

[2] Drozdek, Adam. *Data Structures and algorithms in C++.*, Cengage Learning, 2012.

[3] Nguyễn Đức Nghĩa. *Cấu trúc dữ liệu và thuật toán*, NXB ĐH Bách Khoa - HN, 2013.

[4] Lê Minh Hoàng. *Giải thuật và lập trình*, DHSP - Hà Nội, 2002.

[5] G. M. Adel'son-Vel'skii, E. M. Landis. *An algorithm for the organization of information*, Soviet Mathematics Doklady, 3, 1259-1263, 1962.

[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. *Introduction to Algorithms*, Second Edition, MIT Press, 2001.

[7] Robert Lafore. *Data structures and Algorithms in Java*, (2nd Edition) Sams Publishing, 2002.