MỤC LỤC

LỜI NÓI ĐẦU

Cấu trúc dữ liệu và thuật giải là kiến thức nền tảng của chương trình đào tạo ngành công nghệ thông tin. Trong hệ thống tín chỉ của chương trình đào tạo tại khoa Công nghệ thông tin trường Đại học Đà Lạt, lĩnh vực này được tổ chức thành 2 học phần: cấu trúc dữ liệu và thuật giải 1, cấu trúc dữ liệu và thuật giải 2.

Nội dung học phần cấu trúc dữ liệu và thuật giải 1 được tổ chức trong 4 chương:

- Chương 1 trình bày một số vấn đề cơ bản về cấu trúc dữ liệu và thuật giải.
  - Các bước trong lập trình để giải quyết cho một bài giải,
  - Các khái niệm kiểu dữ liệu, kiểu dữ liệu trừu tượng,
  - Tiếp cận phân tích thuật giải.
  - Một số lược đồ thuật giải.
- Chương 2 trình bày các phương pháp tìm kiếm và sắp xếp trong.
  - Phương pháp tìm kiếm tuyến tính, tìm kiếm nhị phân, nội suy.
  - Các thuật giải sắp xếp đơn giản: Chọn trực tiếp, Chèn trực tiếp, đổi chỗ trực tiếp, nổi bọt.
  - Các thuật giải sắp xếp năng cao : Heap sort, Quick sort, Merge sort, Radix sort, . . .
- Chương 3 trình bày cấu trúc dữ liệu danh sách liên kết.
  - Định nghĩa và tổ chức danh sách liên kết
  - Danh sách liên kết đơn: định nghĩa, cách tổ chức và các thao tác cơ bản
  - Các cấu trúc đặc biệt của danh sách liên kết đơn: Ngăn xếp, Hàng đợi
  - Các cấu trúc dữ liệu dạng danh sách liên kết khác như danh sách liên kết vòng, danh sách liên kết kép.
- Chương 4 trình bày cấu trúc dữ liệu cây.
  - Các thuật ngữ cơ bản trên cây
  - Cây nhị phân
  - Cây nhị phân tìm kiếm
  - Cây nhị phân tìm kiếm cân bằng

Vì trình độ người biên soạn có hạn nên tập giáo trình không tránh khỏi nhiều khiếm khuyết, Chúng tôi rất mong sự góp ý của các bạn đồng nghiệp và sinh viên.

Cuối cùng, Chúng tôi cảm ơn sự động viên, giúp đỡ của các bạn đồng nghiệp trong khoa Công nghệ thông tin để tập giáo trình này được hoàn thành.

Các tác giả
Nguyễn Thị Thanh Bình
Trần Tuấn Minh
Đinh Viết Tuấn

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

1.1. Từ bài giải đến chương trình

1.1.1 Mô hình hóa bài giải thực tế

Để giải một bài giải trong thực tế bằng máy tính ta phải bắt đầu từ việc xác định bài giải. Nhiều thời gian và công sức bỏ ra để xác định bài giải cần giải quyết, tức là phải trả lời rõ ràng câu hỏi "phải làm gì?" sau đó là "làm như thế nào?". Thông thường, khi khởi đầu, hầu hết các bài giải là không đơn giản, không rõ ràng. Để giảm bớt sự phức tạp của bài giải thực tế, ta phải hình thức hóa nó, nghĩa là phát biểu lại bài giải thực tế thành một bài giải hình thức (hay còn gọi là mô hình giải). Có thể có rất nhiều bài giải thực tế có cùng một mô hình giải.

Ví dụ 1.1: Tô màu bản đồ thế giới.

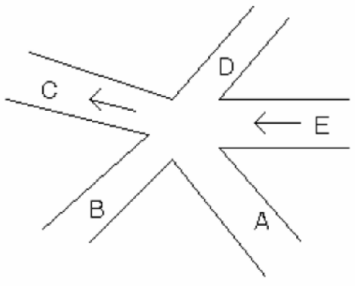
Ta cần phải tô màu cho các nước trên bản đồ thế giới. Trong đó mỗi nước đều được tô một màu và hai nước láng giềng (cùng biên giới) thì phải được tô bằng hai màu khác nhau. Hãy tìm một phương án tô màu sao cho số màu sử dụng là ít nhất.

Ta có thể xem mỗi nước trên bản đồ thế giới là một đỉnh của đồ thị, hai nước láng giềng của nhau thì hai đỉnh ứng với nó được nối với nhau bằng một cạnh. Bài giải lúc này trở thành bài giải tô màu cho đồ thị như sau: Mỗi đỉnh đều phải được tô màu, hai đỉnh có cạnh nối thì phải tô bằng hai màu khác nhau và ta cần tìm một phương án tô màu sao cho số màu được sử dụng là ít nhất.

Ví dụ 1.2: Đèn giao thông.

Cho một ngã năm như hình I.1, trong đó C và E là các đường một chiều theo chiều mũi tên, các đường khác là hai chiều. Hãy thiết kế một bảng đèn hiệu điều khiển giao thông tại ngã năm này một cách hợp lý, nghĩa là: phân chia các lối đi tại ngã năm này thành các nhóm, mỗi nhóm gồm các lối đi có thể cùng đi đồng thời nhưng không xảy ra tai nạn giao thông (các hướng đi không cắt nhau), và số lượng nhóm là ít nhất có thể được.

Ta có thể xem đầu vào (input) của bài giải là tất cả các lối đi tại ngã năm này, đầu ra (output) của bài giải là các nhóm lối đi có thể đi đồng thời mà không xảy ra tai nạn giao thông, mỗi nhóm sẽ tương ứng với một pha điều khiển của đèn hiệu, vì vậy ta phải tìm kiếm lời giải với số nhóm là ít nhất để giao thông không bị tắc nghẽn vì phải chờ đợi quá lâu.



Hình I.1

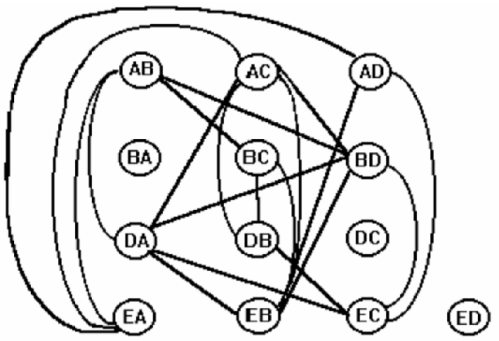
Trước hết ta nhận thấy rằng tại ngã năm này có 13 lối đi: AB, AC, AD, BA, BC, BD, DA, DB, DC, EA, EB, EC, ED. Tất nhiên, để có thể giải được bài giải ta phải tìm

một cách nào đó để thể hiện mối liên quan giữa các lối đi này. Lối nào với lối nào không thể đi đồng thời, lối nào và lối nào có thể đi đồng thời. Ví dụ cặp AB và EC có thể đi đồng thời, nhưng AD và EB thì không, vì các hướng giao thông cắt nhau. Ở đây ta sẽ dùng một sơ đồ trực quan như sau: tên của 13 lối đi được viết lên mặt phẳng, hai lối đi nào nếu đi đồng thời sẽ xảy ra đụng nhau (tức là hai hướng đi cắt qua nhau) ta nối lại bằng một đoạn thẳng, hoặc cong, hoặc ngoằn ngoèo tuỳ thích. Ta sẽ có một sơ đồ như hình I.2. Như vậy, trên sơ đồ này, hai lối đi có cạnh nối lại với nhau là hai lối đi không thể cho đi đồng thời.

Với cách biểu diễn như vậy ta đã có một đồ thị (Graph), tức là ta đã mô hình hoá bài giải giao thông ở trên theo mô hình giải là đồ thị; trong đó mỗi lối đi trở thành một đỉnh của đồ thị, hai lối đi không thể cùng đi đồng thời được nối nhau bằng một đoạn ta gọi là cạnh của đồ thị. Bây giờ, ta phải xác định các nhóm, với số nhóm ít nhất, mỗi nhóm gồm các lối đi có thể đi đồng thời, nó ứng với một pha của đèn hiệu điều khiển giao thông. Giả sử rằng, ta dùng màu để tô lên các đỉnh của đồ thị này sao cho: Các lối đi cho phép cùng đi đồng thời sẽ có cùng một màu. Dễ dàng nhận thấy rằng hai đỉnh có cạnh nối nhau sẽ không được tô cùng màu.

Số nhóm là ít nhất: ta phải tính giải sao cho số màu được dùng là ít nhất.

Tóm lại, ta phải giải quyết bài giải "Tô màu cho đồ thị" ở hình I.2 sao cho:



Hình I.2

- Hai đỉnh có cạnh nối với nhau (hai còn gọi là hai đỉnh kề nhau) không cùng màu.
- Số màu được dùng là ít nhất.

Hai bài giải thực tế "tô màu bản đồ thế giới" và "đèn giao thông" xem ra rất khác biệt nhau nhưng sau khi mô hình hóa, chúng thực chất chỉ là một, đó là bài giải "tô màu đồ thị".

Đối với một bài giải đã được hình thức hoá, chúng ta có thể tìm kiếm cách giải trong thuật ngữ của mô hình đó và xác định có hay không một chương trình có sẵn để giải. Nếu không có một chương trình như vậy thì ít nhất chúng ta cũng có thể tìm được những gì đã biết về mô hình và dùng các tính chất của mô hình để xây dựng một thuật giải tốt.

1.1.2 Thuật giải (algorithms)

Khi đã có mô hình thích hợp cho một bài giải ta cần cố gắng tìm cách giải quyết bài giải trong mô hình đó. Khởi đầu là tìm một thuật giải, đó là một chuỗi hữu hạn các chỉ thị (instruction) mà mỗi chỉ thị có một ý nghĩa rõ ràng và thực hiện được trong một lượng thời gian hữu hạn.

Knuth (1973) định nghĩa thuật giải là một chuỗi hữu hạn các thao tác để giải một bài giải nào đó. Các tính chất quan trọng của thuật giải là:

- Hữu hạn (finiteness): thuật giải phải luôn luôn kết thúc sau một số hữu hạn bước.
- Xác định (definiteness): mỗi bước của thuật giải phải được xác định rõ ràng và phải được thực hiện chính xác, nhất quán.
- Hiệu quả (effectiveness): các thao tác trong thuật giải phải được thực hiện trong một lượng thời gian hữu hạn.

Ngoài ra, một thuật giải còn phải có đầu vào (input) và đầu ra (output). Nói tóm lại, một thuật giải phải giải quyết xong công việc khi ta cho dữ liệu vào. Có nhiều cách để thể hiện thuật giải: dùng lời, dùng lưu đồ,... Và một lối dùng rất phổ biến là dùng ngôn ngữ mã giả, đó là sự kết hợp của ngôn ngữ tự nhiên và các cấu trúc của ngôn ngữ lập trình.

Ví dụ 1.3: Thiết kế thuật giải để giải bài giải " tô màu đồ thị" trên.

Bài giải tô màu cho đồ thị không có thuật giải tốt để tìm lời giải tối ưu, tức là, không có thuật giải nào khác hơn là "thử tất cả các khả năng" hay "vét cạn" tất cả các trường hợp có thể có, để xác định cách tô màu cho các đỉnh của đồ thị sao cho số màu dùng là ít nhất. Thực tế, ta chỉ có thể "vét cạn" trong trường hợp đồ thị có số đỉnh nhỏ, trong trường hợp ngược lại ta không thể "vét cạn" tất cả các khả năng trong một lượng thời gian hợp lý, do vậy ta phải suy nghĩ cách khác để giải quyết vấn đề:

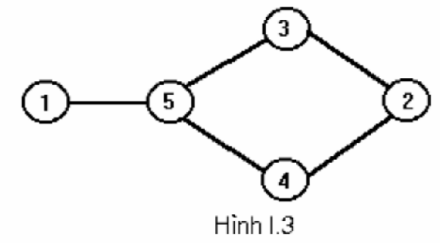
Thêm thông tin vào bài giải để đồ thị có một số tính chất đặc biệt và dùng các tính chất đặc biệt này ta có thể dễ dàng tìm lời giải, hoặc thay đổi yêu cầu bài giải một ít cho dễ giải quyết, nhưng lời giải tìm được chưa chắc là lời giải tối ưu. Một cách làm như thế đối với bài giải trên là "Cố gắng tô màu cho đồ thị bằng ít màu nhất một cách nhanh chóng". Ít màu nhất ở đây có nghĩa là số màu mà ta tìm được không phải luôn luôn là số

màu của lời giải tối ưu (ít nhất) nhưng trong đa số trường hợp thì nó sẽ trùng với đáp số của lời giải tối ưu và nếu có chênh lệch thì nó "không chênh lệch nhiều" so với lời giải tối ưu, bù lại ta không phải "vét cạn" mọi khả năng có thể! Nói khác đi, ta không dùng thuật giải "vét cạn" mọi khả năng để tìm lời giải tối ưu mà tìm một giải pháp để đưa ra lời giải hợp lý một cách khả thi về thời gian. Một giải pháp như thế gọi là một HEURISTIC. HEURISTIC cho bài giải tô màu đồ thị, thường gọi là thuật giải "háu ăn" (GREEDY) là:

- Chọn một đỉnh chưa tô màu và tô nó bằng một màu mới C nào đó.
- Duyệt danh sách các đỉnh chưa tô màu. Đối với một đỉnh chưa tô màu, xác định xem nó có kề với một đỉnh nào được tô bằng màu C đó không. Nếu không có, tô nó bằng màu C đó.

Ý tưởng của Heuristic này là hết sức đơn giản: dùng một màu để tô cho nhiều đỉnh nhất có thể được (các đỉnh được xét theo một thứ tự nào đó), khi không thể tô được nữa với màu đang dùng thì dùng một màu khác. Như vậy, ta có thể "hi vọng" là số màu cần dùng sẽ ít nhất.

Ví dụ 1.4: Đồ thị hình I.3 và cách tô màu cho nó



Hình I.3

Tô theo GREEDY (xét lần lượt theo số thứ tự các đỉnh)	Tối ưu (thử tất cả các khả năng)
1: đỏ; 2: đỏ 3: xanh;4: xanh 5: vàng	1,3,4 : đỏ 2,5 : xanh

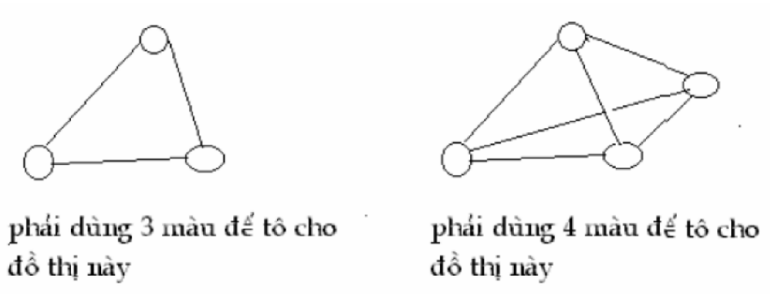
Rõ ràng cách tô màu trong thuật giải "háu ăn" không luôn luôn cho lời giải tối ưu nhưng nó được thực hiện một cách nhanh chóng.

Trở lại bài giải giao thông ở trên và áp dụng HEURISTIC Greedy cho đồ thị trong hình I.2 (theo thứ tự các đỉnh đã liệt kê ở trên), ta có kết quả:

- Tô màu xanh cho các đỉnh: AB,AC,AD,BA,DC,ED
- Tô màu đỏ cho các đỉnh: BC,BD,EA
- Tô màu tím cho các đỉnh: DA,DB
- Tô màu vàng cho các đỉnh: EB,EC

Như vậy, ta đã tìm ra một lời giải là dùng 4 màu để tô cho đồ thị hình I.2. Như đã nói, lời giải này không chắc là lời giải tối ưu. Vậy liệu có thể dùng 3 màu hoặc ít hơn 3 màu không? Ta có thể trở lại mô hình của bài giải và dùng tính chất của đồ thị để kiểm tra kết quả. Nhận xét rằng:

- Một đồ thị có k đỉnh và mỗi cặp đỉnh bất kỳ đều được nối nhau thì phải dùng k màu để tô. Hình I.4 chỉ ra hai ví dụ với k=3 và k=4.



Hình I.4

- Một đồ thị trong đó có k đỉnh mà mỗi cặp đỉnh bất kỳ trong k đỉnh này đều được nối nhau thì không thể dùng ít hơn k màu để tô cho đồ thị.

Đồ thị trong hình I.2 có 4 đỉnh: AC,DA,BD,EB mà mỗi cặp đỉnh bất kỳ đều được nối nhau vậy đồ thị hình I.2 không thể tô với ít hơn 4 màu. Điều này khẳng định rằng lời giải vừa tìm được ở trên trùng với lời giải tối ưu.

Như vậy, ta đã giải được bài giải giao thông đã cho. Lời giải cho bài giải là 4 nhóm, mỗi nhóm gồm các lối có thể đi đồng thời, nó ứng với một pha điều khiển của đèn hiệu. Ở đây cần nhấn mạnh rằng, sở dĩ ta có lời giải một cách rõ ràng chặt chẽ như vậy là vì chúng ta đã giải bài giải thực tế này bằng cách mô hình hoá nó theo một mô hình thích hợp (mô hình đồ thị) và nhờ các kiến thức trên mô hình này (bài giải tô màu và heuristic

để giải) ta đã giải quyết được bài giải. Điều này khẳng định vai trò của việc mô hình hoá bài giải.

Từ những thảo luận trên, chúng ta có thể tóm tắt các bước tiếp cận với một bài giải bao gồm:

1. Mô hình hoá bài giải bằng một mô hình giải học thích hợp.

2. Tìm thuật giải trên mô hình này. Thuật giải có thể mô tả một cách không hình thức, tức là nó chỉ nêu phương hướng giải hoặc các bước giải một cách tổng quát.

3. Phải hình thức hoá thuật giải bằng cách viết một thủ tục bằng ngôn ngữ giả, rồi chi tiết hoá dần ("mịn hoá") các bước giải tổng quát ở trên, kết hợp với việc dùng các kiểu dữ liệu trừu tượng và các cấu trúc điều khiển trong ngôn ngữ lập trình để mô tả thuật giải. Ở bước này, nói chung, ta có một thuật giải tương đối rõ ràng, nó gần giống như một chương trình được viết trong ngôn ngữ lập trình, nhưng nó không phải là một chương trình chạy được vì trong khi viết thuật giải ta không chú trọng nặng đến cú pháp của ngôn ngữ và các kiểu dữ liệu còn ở mức trừu tượng chứ không phải là các khai báo cài đặt kiểu trong ngôn ngữ lập trình.

4. Cài đặt thuật giải trong một ngôn ngữ lập trình cụ thể (Pascal,C,...). Ở bước này, ta dùng các cấu trúc dữ liệu được cung cấp trong ngôn ngữ, ví dụ Array, Record,... để thể hiện các kiểu dữ liệu trừu tượng, các bước của thuật giải được thể hiện bằng các lệnh và các cấu trúc điều khiển trong ngôn ngữ lập trình được dùng để cài đặt thuật giải.

1.2. Kiểu dữ liệu trừu tượng (Abstract Data Type - ADT)

1.2.1 Khái niệm trừu tượng hóa

Trong tin học, trừu tượng hóa nghĩa là đơn giản hóa, làm cho nó sáng sủa hơn và dễ hiểu hơn. Cụ thể trừu tượng hóa là che đi những chi tiết, làm nổi bật cái tổng thể. Trừu tượng hóa có thể thực hiện trên hai khía cạnh là trừu tượng hóa dữ liệu và trừu tượng hóa chương trình.

1.2.2 Trừu tượng hóa chương trình

Trừu tượng hóa chương trình là sự định nghĩa các chương trình con để tạo ra các phép giải trừu tượng (sự tổng quát hóa của các phép giải nguyên thủy). Chẳng hạn, ta có thể tạo ra một chương trình con Matrix_Mult để thực hiện phép giải nhân hai ma trận. Sau khi Matrix_mult đã được tạo ra, ta có thể dùng nó như một phép giải nguyên thủy (chẳng hạn phép cộng hai số).

Trừu tượng hóa chương trình cho phép phân chia chương trình thành các chương trình con. Sự phân chia này sẽ che dấu tất cả các lệnh cài đặt chi tiết trong các chương trình con. Ở cấp độ chương trình chính, ta chỉ thấy lời gọi các chương trình con và điều này được gọi là sự bao gói.

Ví dụ như một chương trình quản lý sinh viên được viết bằng trừu tượng hóa có thể:

```
void Main()
{
        Nhap( Lop);
        Xu_ly (Lop);
        Xuat (Lop);
}
```

Trong chương trình trên, Nhap, Xu_ly, Xuat là các phép giải trừu tượng. Chúng che dấu bên trong rất nhiều lệnh phức tạp mà ở cấp độ chương trình chính ta không nhìn thấy được. Còn Lop là một biến thuộc kiểu dữ liệu trừu tượng mà ta sẽ xét sau.

1.2.3 Trừu tượng hóa dữ liệu

Trừu tượng hóa dữ liệu là định nghĩa các kiểu dữ liệu trừu tượng.

Một kiểu dữ liệu trừu tượng (Abstract Data Type - ADT) là một mô hình giải học cùng với một tập hợp các phép giải (operator) trừu tượng được định nghĩa trên mô hình đó. Ví dụ, tập hợp số nguyên cùng với các phép giải hợp, giao, hiệu là một kiểu dữ liệu trừu tượng.

Trong một ADT các phép giải có thể thực hiện trên các đối tượng (giải hạng) không chỉ thuộc ADT đó, cũng như kết quả không nhất thiết phải thuộc ADT. Tuy nhiên, phải có ít nhất một giải hạng hoặc kết quả phải thuộc ADT đang xét.

ADT là sự tổng quát hoá của các kiểu dữ liệu nguyên thuỷ.

Ví dụ 1.5: một danh sách (LIST) các số nguyên và các phép giải trên danh sách là:

- Tạo một danh sách rỗng.
- Lấy phần tử đầu tiên trong danh sách và trả về giá trị null nếu danh sách rỗng.
- Lấy phần tử kế tiếp trong danh sách và trả về giá trị null nếu không còn phần tử kế tiếp.
- Thêm một số nguyên vào danh sách.

Điều này cho thấy sự thuận lợi của ADT, đó là ta có thể định nghĩa một kiểu dữ liệu tuỳ ý cùng với các phép giải cần thiết trên nó rồi chúng ta dùng như là các đối tượng nguyên thuỷ. Hơn nữa chúng ta có thể cài đặt một ADT bằng bất kỳ cách nào, chương trình dùng chúng cũng không thay đổi.

Cài đặt ADT là sự thể hiện các phép giải mong muốn (các phép giải trừu tượng) thành các câu lệnh của ngôn ngữ lập trình, bao gồm các khai báo thích hợp và các thủ tục thực hiện các phép giải trừu tượng. Để cài đặt ta chọn một cấu trúc dữ liệu thích hợp có trong ngôn ngữ lập trình hoặc là một cấu trúc dữ liệu phức hợp được xây dựng lên từ các kiểu dữ liệu cơ bản của ngôn ngữ lập trình.

1.2.4 Kiểu dữ liệu, cấu trúc dữ liệu và kiểu dữ liệu trừu tượng

(Data Types, Data Structures, Abstract Data Types)

Mặc dù các thuật ngữ kiểu dữ liệu (hay kiểu - data type), cấu trúc dữ liệu (data structure), kiểu dữ liệu trừu tượng (abstract data type) nghe như nhau, nhưng chúng có ý nghĩa rất khác nhau.

Kiểu dữ liệu là một tập hợp các giá trị và một tập hợp các phép giải trên các giá trị đó. Ví dụ kiểu Boolean là một tập hợp có 2 giá trị TRUE, FALSE và các phép giải trên nó như OR, AND, NOT,… Kiểu Integer là tập hợp các số nguyên có giá trị từ -32768 đến 32767 cùng các phép giải cộng, trừ, nhân, chia, Div, Mod…

Kiểu dữ liệu có hai loại là kiểu dữ liệu sơ cấp và kiểu dữ liệu có cấu trúc hay còn gọi là cấu trúc dữ liệu.

Kiểu dữ liệu sơ cấp là kiểu dữ liệu mà giá trị dữ liệu của nó là đơn nhất. Ví dụ: kiểu Boolean, Integer….

Kiểu dữ liệu có cấu trúc hay còn gọi là cấu trúc dữ liệu là kiểu dữ liệu mà giá trị dữ liệu của nó là sự kết hợp của các giá trị khác. Ví dụ: ARRAY là một cấu trúc dữ liệu.

Một kiểu dữ liệu trừu tượng là một mô hình giải học cùng với một tập hợp các phép giải trên nó. Có thể nói kiểu dữ liệu trừu tượng là một kiểu dữ liệu do chúng ta định nghĩa ở mức khái niệm (conceptual), nó chưa được cài đặt cụ thể bằng một ngôn ngữ lập trình.

Khi cài đặt một kiểu dữ liệu trừu tượng trên một ngôn gnữ lập trình cụ thể, chúng ta phải thực hiện hai nhiệm vụ:

1. Biểu diễn kiểu dữ liệu trừu tượng bằng một cấu trúc dữ liệu hoặc một kiểu dữ liệu trừu tượng khác đã được cài đặt.
2. Viết các chương trình con thực hiện các phép giải trên kiểu dữ liệu trừu tượng mà ta thường gọi là cài đặt các phép giải.

1.2.5 Các kiểu dữ liệu trong C/C++:

a. Các kiểu dữ liệu cơ sở:

- Ký tự: char
- Số nguyên : int, unsigned, long
- Số thực: float, double

b. Các cấu trúc dữ liệu cơ bản

- Mảng 1 chiều:

```
type ten_mang[MAX];
```

- Mảng 2 chiều:

```
type ten_mang[MAX][MAX]
```

- Xâu ký tự (chuỗi):

```
char ten_chuoi[MAX];
```

-Kiểu Cấu trúc:

```
struct  Ten_Kieu
{
        //các trường dữ liệu
};
```

...

1.3. Phân tích thuật giải

Với một vấn đề đặt ra có thể có nhiều thuật giải giải, chẳng hạn người ta đã tìm ra rất nhiều thuật giải sắp xếp một mảng dữ liệu. Trong các trường hợp như thế, khi cần sử dụng thuật giải người ta thường chọn thuật giải có thời gian thực hiện ít hơn các thuật giải khác. Mặt khác, khi đưa ra một thuật giải để giải quyết một vấn đề thì một câu hỏi đặt ra là thuật giải đó có ý nghĩa thực tế không? Nếu thuật giải đó có thời gian thực hiện quá lớn chẳng hạn hàng năm, hàng thế kỷ thì đương nhiên không thể áp dụng thuật giải này trong thực tế. Như vậy chúng ta cần đánh giá thời gian thực hiện thuật giải. Phân tích thuật giải, đánh giá thời gian chạy của thuật giải là một lĩnh vực nghiên cứu quan trọng của khoa học máy tính.

1.3.1 Thuật giải và các vấn đề liên quan

Thuật giải được hiểu là sự đặc tả chính xác một dãy các bước thực hiện giải quyết một vấn đề.

Cần nhấn mạnh rằng, mỗi thuật giải có một dữ liệu vào (Input) và một dữ liệu ra (Output); khi thực hiện thuật giải (thực hiện các bước đã mô tả), thuật giải cần cho ra các dữ liệu ra tương ứng với các dữ liệu vào.

- Biểu diễn thuật giải.

Để đảm bảo tính chính xác, chỉ có thể hiểu một cách duy nhất, thuật giải cần được mô tả trong một ngôn ngữ lập trình thành một chương trình (hoặc một hàm, một thủ tục), tức là thuật giải cần được mô tả dưới dạng mã (code). Tuy nhiên, khi trình bày một thuật giải để cho ngắn gọn nhưng vẫn đảm bảo đủ chính xác, người ta thường biểu diễn thuật giải dưới dạng giả mã (pseudo code). Trong cách biểu diễn này, người ta sử dụng các câu lệnh trong một ngôn ngữ lập trình (pascal hoặc C++) và cả các ký hiệu giải học, các mệnh đề trong ngôn ngữ tự nhiên (tiếng Anh hoặc tiếng Việt chẳng hạn). Trong một số trường hợp, để người đọc hiểu được ý tưởng khái quát của thuật giải, người ta có thể biểu diễn thuật giải dưới dạng sơ đồ (thường được gọi là sơ đồ khối, lược đồ).

- **Tính đúng đắn (correctness)** của thuật giải. Đòi hỏi trước hết đối với thuật giải là nó phải đúng đắn, tức là khi thực hiện nó phải cho ra các dữ liệu mà ta mong muốn tương ứng với các dữ liệu vào. Chẳng hạn nếu thuật giải được thiết kế để tìm ước chung lớn nhất của 2 số nguyên dương, thì khi đưa vào 2 số nguyên dương (dữ liệu

vào) và thực hiện thuật giải phải cho ra một số nguyên dương (dữ liệu ra) là ước chung lớn nhất của 2 số nguyên đó.

Chứng minh một cách chặt chẽ (bằng giải học) tính đúng đắn của thuật giải là một công việc rất khó khăn.

- **Tính hiệu quả (efficiency)** là một tính chất quan trong khác của thuật giải, chúng ta sẽ thảo luận về tính hiệu quả của thuật giải trong mục tiếp theo.

Đến đây chúng ta có thể đặt câu hỏi: có phải đối với bất kỳ vấn đề nào cũng có thuật giải giải (có thể tìm ra lời giải bằng thuật giải)? câu trả lời là không. Người ta đã phát hiện ra một số vấn đề không thể đưa ra thuật giải để giải quyết nó. Các vấn đề đó được gọi là các vấn đề không giải được bằng thuật giải.

1.3.2 Tính hiệu quả của thuật giải

Người ta thường xem xét thuật giải, lựa chọn thuật giải để áp dụng dựa vào các tiêu chí sau:

- Thuật giải đơn giản, dễ hiểu.
- Thuật giải dễ cài đặt (dễ viết chương trình)
- Thuật giải cần ít bộ nhớ
- Thuật giải chạy nhanh

Khi cài đặt thuật giải chỉ để sử dụng một số ít lần, người ta thường lựa chọn thuật giải theo tiêu chí 1 và 2. Tuy nhiên, có những thuật giải được sử dụng rất nhiều lần, trong nhiều chương trình, chẳng hạn các thuật giải sắp xếp, các thuật giải tìm kiếm, các thuật giải đồ thị… Trong các trường hợp như thế người ta lựa chọn thuật giải để sử dụng theo tiêu chí 3 và 4. Hai tiêu chí này được nói tới như là tính hiệu quả của thuật giải.

Tính hiệu quả của thuật giải gồm hai yếu tố: dung lượng bộ nhớ mà thuật giải đòi hỏi và thời gian thực hiện thuật giải. Dung lượng bộ nhớ gồm bộ nhớ dùng để lưu dữ liệu vào, dữ liệu ra, và các kết quả trung gian khi thực hiện thuật giải; dung lượng bộ nhớ mà thuật giải đòi hỏi còn được gọi là độ phức tạp không gian của thuật giải. Thời gian thực hiện thuật giải được nói tới như là thời gian chạy (running time) hoặc độ phức tạp thời gian của thuật giải.

Sau này chúng ta chỉ quan tâm tới đánh giá thời gian chạy của thuật giải. Đánh giá thời gian chạy của thuật giải bằng cách nào? Với cách tiếp cận thực nghiệm chúng ta có

thể cài đặt thuật giải và cho chạy chương trình trên một máy tính nào đó với một số dữ liệu vào. Thời gian chạy mà ta thu được sẽ phụ thuộc vào nhiều nhân tố:

- Kỹ năng của người lập trình
- Chương trình dịch
- Tốc độ thực hiện các phép giải của máy tính
- Dữ liệu vào

Vì vậy, trong cách tiếp cận thực nghiệm, ta không thể nói thời gian chạy của thuật giải là bao nhiêu đơn vị thời gian. Chẳng hạn câu nói "thời gian chạy của thuật giải là 30 giây" là không thể chấp nhận được. Nếu có hai thuật giải A và B giải quyết cùng một vấn đề, ta cũng không thể dùng phương pháp thực nghiệm để kết luận thuật giải nào chạy nhanh hơn, bởi vì ta mới chỉ chạy chương trình với một số dữ liệu vào.

Một cách tiếp cận khác để đánh giá thời gian chạy của thuật giải là phương pháp phân tích sử dụng các công cụ giải học. Chúng ta mong muốn có kết luận về thời gian chạy của một thuật giải mà nó không phụ thuộc vào sự cài đặt của thuật giải, không phụ thuộc vào máy tính mà trên đó thuật giải được thực hiện.

Để phân tích thuật giải chúng ta cần sử dụng khái niệm cỡ (size) của dữ liệu vào. Cỡ của dữ liệu vào được xác định phụ thuộc vào từng thuật giải. Ví dụ, trong thuật giải tính định thức của ma trận vuông cấp n, ta có thể chọn cỡ của dữ liệu vào là cấp n của ma trận; còn đối với thuật giải sắp xếp mảng cỡ n thì cỡ của dữ liệu vào chính là cỡ n của mảng. Đương nhiên là có vô số dữ liệu vào cùng một cỡ. Nói chung trong phần lớn các thuật giải, cỡ của dữ liệu vào là một số nguyên dương n. Thời gian chạy của thuật giải phụ thuộc vào cỡ của dữ liệu vào; chẳng hạn tính định thức của ma trận cấp 20 đòi hỏi thời gian chạy nhiều hơn tính định thức của ma trận cấp 10.

Nói chung, cỡ của dữ liệu càng lớn thì thời gian thực hiện thuật giải càng lớn. Nhưng thời gian thực hiện thuật giải không chỉ phụ thuộc vào cỡ của dữ liệu vào mà còn phụ thuộc vào chính dữ liệu vào. Trong số các dữ liệu vào cùng một cỡ, thời gian chạy của thuật giải cũng thay đổi. Chẳng hạn, xét bài giải tìm xem đối tượng a có mặt trong danh sách $(a_1,\ldots,a_i,\ldots,a_n)$ hay không. Thuật giải được sử dụng là thuật giải tìm kiếm tuần tự: Xem xét lần lượt từng phần tử của danh sách cho tới khi phát hiện ra đối tượng cần tìm thì dừng lại, hoặc đi hết danh sách mà không gặp phần tử nào bằng a. Ở đây cỡ của

dữ liệu vào là n, nếu một danh sách với a là phần tử đầu tiên, ta chỉ cần một lần so sánh và đây là trường hợp tốt nhất, nhưng nếu một danh sách mà a xuất hiện ở vị trí cuối cùng hoặc a không có trong danh sách, ta cần n lần so sánh a với từng ai (i=1,2,…,n), trường hợp này là trường hợp xấu nhất. Vì vậy, chúng ta cần đưa vào khái niệm thời gian chạy trong trường hợp xấu nhất và thời gian chạy trung bình.

Thời gian chạy trong trường hợp xấu nhất (worst-case running time) của một thuật giải là thời gian chạy lớn nhất của thuật giải đó trên tất cả các dữ liệu vào cùng cỡ. Chúng ta sẽ ký hiệu thời gian chạy trong trường hợp xấu nhất là T(n), trong đó n là cỡ của dữ liệu vào. Sau này khi nói tới thời gian chạy của thuật giải chúng ta cần hiểu đó là thời gian chạy trong trường hợp xấu nhất. Sử dụng thời gian chạy trong trường hợp xấu nhất để biểu thị thời gian chạy của thuật giải có nhiều ưu điểm. Trước hết, nó đảm bảo rằng, thuật giải không khi nào tiêu tốn nhiều thời gian hơn thời gian chạy đó. Hơn nữa, trong các áp dụng, trường hợp xấu nhất cũng thường xuyên xảy ra.

Chúng ta xác định **thời gian chạy trung bình** (average running time) của thuật giải là số trung bình cộng của thời gian chạy của thuật giải đó trên tất cả các dữ liệu vào cùng cỡ n. Thời gian chạy trung bình của thuật giải sẽ được ký hiệu là $T_{tb}(n)$. Đánh giá thời gian chạy trung bình của thuật giải là công việc rất khó khăn, cần phải sử dụng các công cụ của xác suất, thống kê và cần phải biết được phân phối xác suất của các dữ liệu vào. Rất khó biết được phân phối xác suất của các dữ liệu vào. Các phân tích thường phải dựa trên giả thiết các dữ liệu vào có phân phối xác suất đều. Do đó, sau này ít khi ta đánh giá thời gian chạy trung bình.

Để có thể phân tích đưa ra kết luận về thời gian chạy của thuật giải độc lập với sự cài đặt thuật giải trong một ngôn ngữ lập trình, độc lập với máy tính được sử dụng để thực hiện thuật giải, chúng ta đo **thời gian chạy của thuật giải bởi số phép giải sơ cấp cần phải thực hiện** khi ta thực hiện thuật giải. Cần chú ý rằng, các phép giải sơ cấp là các phép giải số học, các phép giải logic, các phép giải so sánh,…, nói chung, các phép giải sơ cấp cần được hiểu là các phép giải mà khi thực hiện chỉ đòi hỏi một thời gian cố định nào đó (thời gian này nhiều hay ít là phụ thuộc vào tốc độ của máy tính). Như vậy chúng ta xác định thời gian chạy T(n) là số phép giải sơ cấp mà thuật giải đòi hỏi, khi thực hiện thuật giải trên dữ liệu vào cỡ n.

Tính ra biểu thức mô tả hàm T(n) được xác định như trên là không đơn giản, và biểu thức thu được có thể rất phức tạp. Do đó, chúng ta sẽ chỉ quan tâm tới **tốc độ tăng (rate of growth)** của hàm T(n), tức là tốc độ tăng của thời gian chạy khi cỡ dữ liệu vào tăng. Ví dụ, giả sử thời gian chạy của thuật giải là $T(n) = 3n^2 + 7n + 5$ (phép giải sơ cấp). Khi cỡ n tăng, hạng thức $3n^2$ quyết định tốc độ tăng của hàm T(n), nên ta có thể bỏ qua các hạng thức khác và có thể nói rằng thời gian chạy của thuật giải tỉ lệ với bình phương của cỡ dữ liệu vào. Trong mục tiếp theo chúng ta sẽ định nghĩa ký hiệu ô lớn và sử dụng ký hiệu ô lớn để biểu diễn thời gian chạy của thuật giải.

1.3.3 Ký hiệu O và biểu diễn thời gian chạy bởi ký hiệu O

1. Định nghĩa ký hiệu O

Định nghĩa. Giả sử f(n) và g(n) là các hàm thực không âm của đối số nguyên không âm n. Ta nói "f(n) là ô lớn của g(n)" và viết là f(n) = O(g(n)) nếu tồn tại các hằng số dương c và n_0 sao cho f(n) <= cg(n) với mọi n >= n_0.

Như vậy, f(n) = O(g(n)) có nghĩa là hàm f(n) bị chặn trên bởi hàm g(n) với một nhân tử hằng nào đó khi n đủ lớn. Muốn chứng minh được f(n)= O(g(n)), chúng ta cần chỉ ra nhân tử hằng c, số nguyên dương n_0 và chứng minh được f(n) <= cg(n) với mọi n >= n_0.

Ví dụ 1.6: Giả sử $f(n) = 5n^3+ 2n^2+ 13n + 6$,

ta có: $f(n) = 5n^3+ 2n^2+ 13n + 6 <= 5n^3+ 2n^3+ 13n^3+ 6n^3= 26n^3$

Bất đẳng thức trên đúng với mọi n >= 1, và ta có n_0= 1, c = 26.

Do đó, ta có thể nói $f(n) = O(n^3)$.

Tổng quát nếu f(n) là một đa thức bậc k của n:

$f(n) = a_{kn}{}^k+ a_{k-1n}{}^{k-1}+... + a_{1n} + a_0$ thì $f(n) = O(n^k)$

Sau đây chúng ta đưa ra một số hệ quả từ định nghĩa ký hiệu ô lớn, nó giúp chúng ta hiểu rõ bản chất ký hiệu ô lớn. (Lưu ý, các hàm mà ta nói tới đều là các hàm thực không âm của đối số nguyên dương)

- Nếu $f(n) = g(n) + g_1(n) + ... + g_k(n)$, trong đó các hàm $g_i(n)$ (i=1,...,k) tăng chậm hơn hàm g(n) (tức là $g_i(n)/g(n) \rightarrow 0$, khi $n \rightarrow 0$) thì f(n) = O(g(n))
- Nếu f(n) = O(g(n)) thì f(n) = O(d.g(n)), trong đó d là hằng số dương bất kỳ

- Nếu f(n) = O(g(n)) và g(n) = O(h(n)) thì f(n) = O(h(n)) (tính bắc cầu)

Các kết luận trên dễ dàng được chứng minh dựa vào định nghĩa của ký hiệu ô lớn. Đến đây, ta thấy rằng, chẳng hạn nếu f(n) = $O(n^2)$ thì f(n) =$O(75n^2)$, f(n) = $O(0,01n^2)$, f(n) = $O(n^2+ 7n + \log n)$, f(n) = $O(n^3)$,..., tức là có vô số hàm là cận trên (với một nhân tử hằng nào đó) của hàm f(n).

Một nhận xét quan trọng nữa là, ký hiệu O(g(n)) xác định một tập hợp vô hạn các hàm bị chặn trên bởi hàm g(n), cho nên ta viết f(n) = O(g(n)) chỉ có nghĩa f(n) là một trong các hàm đó.

2. Biểu diễn thời gian chạy của thuật giải

Thời gian chạy của thuật giải là một hàm của cỡ dữ liệu vào: hàm T(n). Chúng ta sẽ biểu diễn thời gian chạy của thuật giải bởi ký hiệu ô lớn:

T(n) = O(f(n)), biểu diễn này có nghĩa là thời gian chạy T(n) bị chặn trên bởi hàm f(n). Thế nhưng như ta đã nhận xét, một hàm có vô số cận trên. Trong số các cận trên của thời gian chạy, chúng ta sẽ lấy **cận trên chặt** (tight bound) để biểu diễn thời gian chạy của thuật giải.

Định nghĩa. Ta nói f(n) là cận trên chặt của T(n) nếu

- T(n) = O(f(n)), và
- Nếu T(n) = O(g(n)) thì f(n) = O(g(n)).

Nói một cách khác, f(n) là cận trên chặt của T(n) nếu nó là cận trên của T(n) và ta không thể tìm được một hàm g(n) là cận trên của T(n) mà lại tăng chậm hơn hàm f(n).

Sau này khi nói thời gian chạy của thuật giải là O(f(n)), chúng ta cần hiểu f(n) là cận trên chặt của thời gian chạy.

Nếu T(n) = O(1) thì điều này có nghĩa là thời gian chạy của thuật giải bị chặn trên bởi một hằng số nào đó, và ta thường nói thuật giải có thời gian chạy hằng. Nếu T(n) = O(n), thì thời gian chạy của thuật giải bị chặn trên bởi hàm tuyến tính, và do đó ta nói thời gian chạy của thuật giải là tuyến tính. Các cấp độ thời gian chạy của thuật giải và tên gọi của chúng được liệt kê trong bảng sau:

Kí hiệu	**Tên gọi**
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

Đối với một thuật giải, chúng ta sẽ đánh giá thời gian chạy của nó thuộc cấp độ nào trong các cấp độ đã liệt kê trên. Trong bảng trên, chúng ta đã sắp xếp các cấp độ thời gian chạy theo thứ tự tăng dần, chẳng hạn thuật giải có thời gian chạy là O(logn) chạy nhanh hơn thuật giải có thời gian chạy là O(n),... Các thuật giải có thời gian chạy là $O(n^k)$, với k = 1,2,3,..., được gọi là các thuật giải **thời gian chạy đa thức** (polynimial-time algorithm).

Để so sánh thời gian chạy của các thuật giải thời gian đa thức và các thuật giải thời gian mũ, chúng ta hãy xem xét bảng sau:

Thời gian chạy	Cỡ dữ liệu vào					
	10	20	30	40	50	60
N	0,00001 giây	0,00002 giây	0,00003 giây	0,00004 giây	0,00005 giây	0,00006 giây
N^2	0,0001 giây	0,0004 giây	0,0009 giây	0,0016 giây	0,0025 giây	0,0036 giây
N^3	0,001 giây	0,008 giây	0,027 giây	0,064 giây	0,125 giây	0,216 giây
N^5	0,1 giây	3,2 giây	24,3 giây	1,7 phút	5,2 phút	13 phút
2^n	0,001 giây	1,0 giây	17,9 phút	12,7 ngày	35,7 năm	366 thế kỷ
3^n	0,059	58 phút	6,5 năm	3855 thế	2.10^8 thế	$1,3.10^{13}$

	giây			kỷ	kỷ	thế kỷ

Trong bảng trên, ta giả thiết rằng mỗi phép giải sơ cấp cần 1 micro giây để thực hiện. Thuật giải có thời gian chạy n^2, với cỡ dữ liệu vào n = 20, nó đòi hỏi thời gian chạy là $20^2 x 10^{-6} = 0{,}004$ giây. Đối với các thuật giải thời gian mũ, ta thấy rằng thời gian chạy của thuật giải là chấp nhận được chỉ với các dữ liệu vào có cỡ rất khiêm tốn, n < 30; khi cỡ dữ liệu vào tăng, thời gian chạy của thuật giải tăng lên rất nhanh và trở thành con số khổng lồ.

Chẳng hạn, thuật giải với thời gian chạy 3^n, để tính ra kết quả với dữ liệu vào cỡ 60, nó đòi hỏi thời gian là $1{,}3 x 10^{13}$ thế kỷ! Để thấy con số này khổng lồ đến mức nào, ta hãy liên tưởng tới vụ nổ "big-bang", "big-bang" được ước tính là xảy ra cách đây $1{,}5 x 10^8$ thế kỷ. Chúng ta không hy vọng có thể áp dụng các thuật giải có thời gian chạy mũ trong tương lai nhờ tăng tốc độ máy tính, bởi vì không thể tăng tốc độ máy tính lên mãi được, do sự hạn chế của các quy luật vật lý. Vì vậy nghiên cứu tìm ra các thuật giải hiệu quả (chạy nhanh) cho các vấn đề có nhiều ứng dụng trong thực tiễn luôn luôn là sự mong muốn của các nhà tin học.

1.3.4 Đánh giá thời gian chạy của thuật giải

Mục này trình bày các kỹ thuật để đánh giá thời gian chạy của thuật giải bởi ký hiệu ô lớn. Cần lưu ý rằng, đánh giá thời gian chạy của thuật giải là công việc rất khó khăn, đặc biệt là đối với các thuật giải đệ quy. Tuy nhiên các kỹ thuật đưa ra trong mục này cho phép đanh giá được thời gian chạy của hầu hết các thuật giải mà ta gặp trong thực tế. Trước hết chúng ta cần biết cách thao tác trên các ký hiệu ô lớn. Quy tắc "cộng các ký hiệu ô lớn" sau đây được sử dụng thường xuyên nhất.

1. Luật tổng

Giả sử thuật giải gồm hai phần (hoặc nhiều phần), thời gian chạy của phần đầu là $T_1(n)$, phần sau là $T_2(n)$. Khi đó thời gian chạy của thuật giải là $T_1(n) + T_2(n)$ sẽ được suy ra từ sự đánh giá của $T_1(n)$ và $T_2(n)$ theo luật sau:

Giả sử $T_1(n) = O(f(n))$ và $T_2(n) = O(g(n))$. Nếu hàm f(n) tăng nhanh hơn hàm g(n), tức là $g(n) = O(f(n))$, thì $T_1(n) + T_2(n) = O(f(n))$.

Luật này được chứng minh như sau. Theo định nghĩa ký hiệu ô lớn, ta tìm được các hằng số c1, c2, c3 và n1, n2, n3 sao cho:

$T_1(n) <= c_1f(n)$ với $n >= n_1$

$T_2(n) <= c_2\ g(n)$ với $n >= n_2$

$g(n) <= c_3f(n)$ với $n >= n_3$

Đặt n_0= max(n_1, n_2, n_3). Khi đó với mọi $n >= n_0$, ta có:

$$T_1(n) + T_2(n) <= c_1f(n) + c_2g(n)$$
$$<= c_1f(n) + c_2c_3f(n) = (c_1+c_2c_3)f(n)$$

Như vậy với $c = c_1+ c_2c_3$ thì $T_1(n) + T_2(n) <= cf(n)$ với mọi $n >= n_0$

Ví dụ 1.7: Giả sử thuật giải gồm ba phần, thời gian chạy của từng phần được đánh giá là $T_1(n) = O(n\log n)$, $T_2(n) = O(n^2)$ và $T_3(n) = O(n)$. Khi đó thời gian chạy của toàn bộ thuật giải là $T(n) = T_1(n) + T_2(n) + T_3(n) = O(n^2)$, vì hàm n^2 tăng nhanh hơn các hàm nlogn và n.

2. Thời gian chạy của các lệnh

Thời gian thực hiện các phép giải sơ cấp là O(1).

- **Lệnh gán**

Lệnh gán có dạng

X = <biểu thức>

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

- **Lệnh lựa chọn**

Lệnh lựa chọn **if-else** có dạng

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

Trong đó, điều kiện là một biểu thức cần được đánh giá, nếu điều kiện đúng thì lệnh 1 được thực hiện, nếu không thì lệnh 2 được thực hiện. Giả sử thời gian đánh giá điều kiện là $T_0(n)$, thời gian thực hiện lệnh 1 là $T_1(n)$, thời gian thực hiện lệnh 2 là $T_2(n)$.

Thời gian thực hiện lệnh lựa chọn if-else sẽ là thời gian lớn nhất trong các thời gian $T_0(n) + T_1(n)$ và $T_0(n) + T_1(n)$.

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

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

- **Các lệnh lặp**

Các lệnh lặp: for, while, do-while

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

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

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

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

Ví dụ 1.8: Giả sử ta có mảng A các số thực, cỡ n và ta cần tìm xem mảng có chứa số thực x không. Điều đó có thể thực hiện bởi thuật giải tìm kiếm tuần tự như sau:

(1) i = 0;

```
(2) while (i < n && x != A[i])
(3)     i++;
```

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

Ví dụ 1.9: Thuật giải tạo ra ma trận đơn vị A cấp n;

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

Thuật giải gồm hai lệnh lặp for. Lệnh lặp for đầu tiên (các dòng (1)-(3)) có thân lại là một lệnh lặp for ((2)-(3)). Số lần lặp của lệnh for ((2)-(3)) là n, thân của nó là lệnh (3) có thời gian chạy là O(1), do đó thời gian chạy của lệnh lặp for này là O(n). Lệnh lặp for ((1)-(3)) cũng có số lần lặp là n, thân của nó có thời gian đã đánh giá là O(n), nên thời gian của lệnh lặp for ((1)-(3)) là $O(n^2)$. Tương tự lệnh for ((4)-(5)) có thời gian chạy là O(n). Sử dụng luật tổng, ta suy ra thời gian chạy của thuật giải là $O(n^2)$.

- **Phân tích các hàm đệ quy**

Các hàm đệ quy là các hàm có chứa lời gọi hàm đến chính nó. Trong mục này, chúng ta sẽ trình bày phương pháp chung để phân tích các hàm đệ quy, sau đó sẽ đưa ra một số kỹ thuật phân tích một số lớp hàm đệ quy hay gặp.

Giả sử ta có hàm đệ quy F, thời gian chạy của hàm này là T(n), với n là cỡ dữ liệu vào. Khi đó thời gian chạy của các lời gọi hàm ở trong hàm F sẽ là T(m) với m<n. Trước hết ta cần đánh giá thời gian chạy của hàm F trên dữ liệu cỡ nhỏ nhất n = 1, giả sử T(1) = a với a là một hằng số nào đó.

Sau đó bằng cách đánh giá thời gian chạy của các câu lệnh trong thân của hàm F, chúng ta sẽ tìm ra quan hệ đệ quy biểu diễn thời gian chạy của hàm F thông qua lời gọi hàm, tức là biểu diễn T(n) thông qua các T(m), với m < n.

Chẳng hạn, giả sử hàm đệ quy F chứa hai lời gọi hàm với thời gian chạy tương ứng là $T(m_1)$ và $T(m_2)$, trong đó m_1, $m_2<n$, khi đó ta thu được quan hệ đệ quy có dạng như sau:

$T(1) = 1$

$T(n) = f(T(m_1),T(m_2))$

Trong đó, f là một biểu thức nào đó của $T(m_1)$ và $T(m_2)$. Giải quan hệ đệ quy trên, chúng ta sẽ đánh giá được thời gian chạy T(n). Nhưng cần lưu ý rằng, giải các quan hệ đệ quy là rất khó khăn, chúng ta sẽ đưa ra kỹ thuật giải cho một số trường hợp đặc biệt.

Ví dụ 1.10: (Hàm tính giai thừa của số nguyên dương n).

```
int Fact(int n)
{
        if (n = = 1)
                return 1;
        else return n * Fact(n-1);
}
```

Giả sử thời gian chạy của hàm là T(n), với n = 1 ta có T(1) = O(1). Với n > 1, ta cần kiểm tra điều kiện của lệnh if-else và thực hiện phép nhân n với kết quả của lời gọi hàm, do đó T(n) = T(n-1) + O(1). Như vậy ta có quan hệ đệ quy sau:

$T(1) = O(1)$

$T(n) = T(n-1) + O(1)$ với $n >1$

Thay các ký hiệu O(1) bởi các hằng số dương a và b tương ứng, ta có

$T(1) = a$

$T(n) = T(n-1) + b$ với $n > 1$

Sử dụng các phép thế T(n-1) = T(n-2) + b, T(n-2) = T(n-3) + b,..., ta có

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

$= T(n-2) + 2b$

$= T(n-3) + 3b$

...

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

$= a + (n-1)b$

Từ đó, ta suy ra $T(n) = O(n)$.

Kỹ thuật thế lặp còn có thể được sử dụng để giải một số dạng quan hệ đệ quy khác, chẳng hạn quan hệ đệ quy sau

$T(1) = a$

$T(n) = 2\ T(n/2) + g(n)$

Quan hệ đệ quy này được dẫn ra từ các thuật giải đệ quy được thiết kế theo ý tưởng: giải quyết bài giải cỡ n được quy về giải quyết hai bài giải con cỡ n/2. Ở đây g(n) là các tính giải để chuyển bài giải về hai bài giải con và các tính giải cần thiết khác để kết hợp nghiệm của hai bài giải con thành nghiệm của bài giải đã cho. Một ví dụ điển hình của các thuật giải được thiết kế theo cách này là thuật giải sắp xếp hoà nhập (MergeSort).

Chúng ta đã xem xét một vài dạng quan hệ đệ quy đơn giản. Thực tế, các hàm đệ quy có thể dẫn tới các quan hệ đệ quy phức tạp hơn nhiều; và có những quan hệ đệ quy rất đơn giản nhưng tìm ra nghiệm của nó cũng rất khó khăn. Chúng ta không đi sâu vào vấn đề này.

1.4 Một số lược đồ thuật giải

1.4.1 Thuật giải vét cạn:

Ý tưởng thuật giải vét cạn, còn gọi là thuật giải Brute-force, là xem xét tất cả các ứng viên để phát hiện đối tượng mong muốn. Thuật giải vét cạn sinh ra tất cả các khả năng có thể có và kiểm tra mỗi khả năng có thỏa yêu cầu bài giải không ?
Thuật giải vét cạn thường dùng để giải các bài giải liệt kê tổ hợp, hoán vị.
Thuật giải vét cạn kém hiệu quả vì độ phức tạp tính giải lớn, nhưng cũng có nhiều vấn đề không có các giải quyết nào khác bằng vét cạn.

Một số thuật giải quan trọng liên quan đến vét cạn như : thuật giải sinh, quay lui, nhánh cận, . . .

1.4.2 Thuật giải chia để trị (Divide - and - conquer)

1. Ý tưởng :

Có lẽ quan trọng và áp dụng rộng rãi nhất là kỹ thuật thiết kế “Chia để trị” . Nó phân rã bài giải kích thước n thành các bài giải con nhỏ hơn mà việc tìm lời giải của chúng là cùng một cách. Lời giải của bài giải đã cho được xây dựng từ lời giải của các bài giải con này .

Ta có thể nói vắn tắt ý tưởng chính của phương pháp này là : chia dữ liệu thành từng miền đủ nhỏ, giải bài giải trên các miền đã chia rồi tổng hợp kết quả lại.

2. Lược đồ :

Nếu gọi D&C($\Re$) - Với $\Re$ là miền dữ liệu - là hàm thể hiện cách giải bài giải theo phương pháp chia để trị thì ta có thể viết :

```
void D&C(ℜ)
{
    If (ℜ đủ nhỏ)
        giải bài giải;
    Else
    {
        Chia ℜ thành ℜ1 , …,ℜm ;
        for (i = 1; i <=m; i++)
            D&C(ℜi);
        Tổng hợp kết quả;
    }
}
```

Ví dụ 1.11: Bài giải chia thưởng

Chia m phần thưởng cho n học sinh có thứ tự.

- Thiết kế :

Gọi Part (m ,n) là số cách chia, m phần thưởng cho n học sinh có thứ tự.

Các trường hợp đặc biệt :

- m = 0 :
  Khi đó chỉ có 1 cách chia : mọi học sinh đều nhận được 0 phần thưởng .
  Vậy : Part(0 , n) = 1 với mọi n
- n = 0 (không có học sinh nào), $m \neq 0$:
  Khi đó không có cách chia nào.
  Vậy : Part(m , 0) = 0, $\forall$ $m \neq 0$.
- n = 1 (có 1 học sinh), $m \neq 0$:
  Khi đó chỉ có 1 cách chia.
  Vậy : Part(m , 1) = 1, $\forall$ $m \neq 0$.

Trường hợp tổng quát :

Phân rã bài giải thành các bài giải con.

- m < n :
  Khi số phần thương m nhỏ hơn số học sinh n thì n - m học sinh xếp cuối sẽ luôn không nhận được gì cả trong mọi cách chia .

Vậy :

khi n > m thì Part(m , n) = Part(m , m) .

- m >= n : (Số phần thưởng lớn hơn hoặc bằng số học sinh)
  Ta phân các cách chia làm 2 nhóm :
  - Nhóm 1: không dành cho học sinh xếp cuối cùng phần thưởng nào cả

Số cách chia này sẽ bằng số cách chia m phần thương cho n -1 học sinh .

Tức là : Số cách chia trong nhóm 1 là : Part(m , n -1) .

- Nhóm 2: có phần cho người cuối cùng. Dễ thấy rằng số cách chia của nhóm này bằng số cách chia m - n phần thương cho n học sinh.

Tức là : Số cách chia trong nhóm 2 là: PART(m - n , n) .

Vậy :

Với $m \geq n$ Part(m , n) = Part(m , n -1) + Part(m - n , n)

- Mô tả thuật giải :

Input : m, số phần thưởng
n, số học sinh

Output :s, Số các chia m phần thưởng cho n học sinh

Mô tả :

```
PART(m , n ) ≡
{
        Nếu (m == 0)
                s = 1;
        Ngược lại //m!=0
                Nếu ( n == 0)
                        s = 0;
                Ngược lại
                        Nếu ( n == 1)
                                s = 1;
                        Ngược lại //n != 0,1
                                Nếu (m < n)
                                        s = Part(m , m);
                                Ngược lại
                                        s = Part(m , n -1) + Part(m - n , n );
        return s;
}
```

- Cài đặt :

```
int Part(int m, int n)
{
        int s;

        if (m == 0)
                s = 1;
        else
                if (n == 0)
                        s = 0;
                else
                        if (n == 1)
                                s = 1;
                        else
                        if (m < n)
                                s = Part(m, m);
                        else
                                s = Part(m, n - 1) + Part(m - n, n);
        return s;
}
```

1.4.3 Thuật giải quay lui

1. Ý tưởng :

Nét đặc trưng của phương pháp quay lui là các bước hướng tới lời giải cuối cùng của bài giải hoàn toàn được làm thử.

Tại mỗi bước, nếu có một lựa chọn được chấp nhận thì ghi nhận lại lựa chọn này và tiến hành các bước thử tiếp theo. Còn ngược lại không có lựa chọn nào thích hợp thì làm lại bước trước, xoá bỏ sự ghi nhận và quay về chu trình thử các lựa chọn còn lại. Hành động này được gọi là quay lui, thuật giải thể hiện phương pháp này gọi là quay lui.

Điểm quan trọng của thuật giải là phải ghi nhớ tại mỗi bước đi qua để tránh trùng lặp khi quay lui.

Dễ thấy là các thông tin này cần được lưu trử vào một ngăn xếp, nên thuật giải thể hiện ý thiết kế một cách đệ quy.

2. Thiết kế :

Lời giải của bài giải thường biểu diễn bằng một vec tơ gồm n thành phần $x = (x_1,.., x_n)$ phải thỏa mãn các điều kiện nào đó. Để chỉ ra lời giải x, ta phải xây dựng dần các thành phần lời giải x_i .

Tại mỗi bước i :

- Đã xây dựng xong các thành phần $x_1,.., x_{i-1}$.
- Xây dựng thành phần x_i bằng cách lần lượt thử tất cả các khả năng mà x_i có thể chọn :
  - Nếu một khả năng j nào đó phù hợp cho x_i thì xác định x_i theo khả năng j. Thường phải có thêm thao tác ghi nhận trạng thái mới của bài giải để hổ trợ cho bước quay lui. Nếu i = n thì ta có được một lời giải, ngược lại thì tiến hành bước i+1 để xác định x_{i+1} .
  - Nếu không có một khả năng nào chấp nhận được cho x_i thì ta lùi lại bước trước (bước 1-1) để xác định lại thành phần x_{i-1}.

Để đơn giản, ta giả định các khả năng chọn lựa cho các x_i tại mỗi bước là như nhau, do đó ta phải có thêm một thao tác kiểm tra khả năng j nào là chấp nhận được cho x_i .

3. Lược đồ:

Lược đồ phương pháp quay lui có thể viết bằng thủ tục sau, với n là số bước cần phải thực hiện, k là số khả năng mà xi có thể chọn lựa.

```
Try(i) ≡
    for ( j = 1 → k)
        If ( xi chấp nhận được khả năng j)
        {
            Xác định xi theo khả năng j;
            Ghi nhận trạng thái mới;
            if( i < n)
                Try(i+1);
            else
                Ghi nhận nghiệm;
            Trả lại trạng thái cũ cho bài giải;
        }
```

Ghi chú :

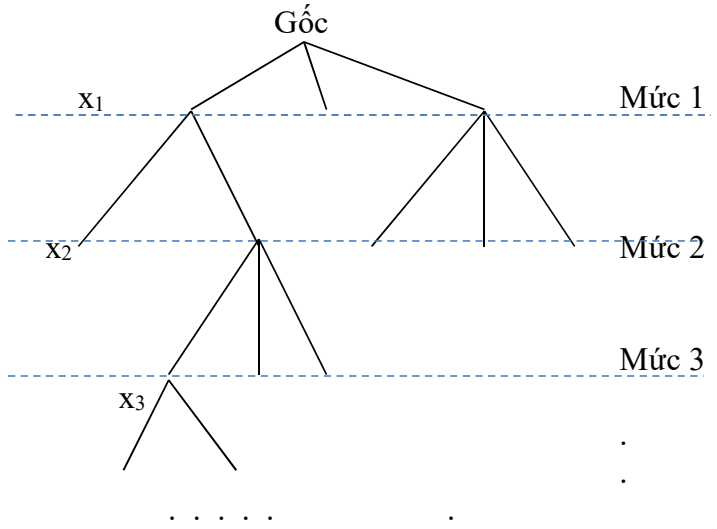
Tìm nghiệm bằng phương pháp quay lui có thể chuyển về tìm kiếm trên cây không gian các trạng thái, với cây được xây dựng từng mưc như sau :

Các nút con của gốc (thuộc mức 1) là các khả năng có thể chọn cho x_1.

Giả sử x_{i-1} làmột nút ở mức thứ i-1, khi đó các nút con của x_{i-1} là các khả năng mà x_i có thể chọn, một khi đã tìm được các thành phần $x_1,.., x_{i-1}$.

Như vậy, mỗi nút x_i của cây biểu diễn một lời giải bộ phận, đó là các nút nằm trên đường đi từ gốc đến nút đó.

Ta có thể nói việc tìm kiếm nghiệm bằng phương pháp quay lui chính là tìm kiếm theo chiều sâu trên cây không gian các trạng thái.



Ví dụ 1.12: Bài giải Ngựa đi tuần.

Cho bàn cờ có n x n ô . Một con ngựa được phép đi theo luật cờ vua, đầu tiên được đặt ở ô có tọa độ x_0 , y_0 .

Vấn đề là hãy chỉ ra các hành trình (nếu có) của ngựa – Đó là ngựa đi qua tất cả các ô của bàn cờ, mỗi ô đi qua đúng một lần .

-Thiết kế thuật giải :

Cách giải quyết rõ ràng là xét xem có thể thực hiện một nước đi kế nữa hay không. Sơ đồ đầu tiên có thể phát thảo như sau :

```
Try(i) ≡
        for ( j = 1 → k)
                If ( xi chấp nhận được khả năng k)
                {
                        Xác định xi theo khả năng k;
                        Ghi nhận trạng thái mới;
                        if( i < n2 )
                                Try(i+1);
                        else
                                Ghi nhận nghiệm;
                        Trả lại trạng thái cũ cho bài giải;
}
```

Để mô tả chi tiết thuật giải, ta phải qui định cách mô tả dữ liệu và các thao tác, đó là :

- Biểu diễn bàn cờ .
- Các khả năng chọ lựa cho x_i ?
- Cách thức xác định x_i theo j.
- Cách thức gi nhận trạng thái mới, trả về trạng thái cũ.

- Ghi nhận nghiệm.
- . . .

* Ta sẽ biểu diễn bàn cờ bằng 1 ma trận vuông cấp n : int h[n][n];

Sở dĩ thể hiện mỗi ô cờ bằng 1 số nguyên thay cho giá trị boole (để đánh dấu ô đã được đi qua chưa) là vì ta muốn lần dò theo quá trình di chuyển của con ngựa.

Ta qui ước như sau :

h[x][y] = 0 ≡ Ô <x,y> ngựa chưa đi qua;

h[x][y] = i ≡ Ô <x,y> ngựa đã đi qua ở bước thứ i ($1 \le i \le n^2$).

* Các khả năng chon lựa cho x_i? Đó chính là các nước đi của ngựa mà x_i có thể chấp nhận.

* Với cặp tọa độ bắt đầu <x,y> như trong hình vẽ, có tất cả 8 ô <u,v> mà con ngựa có thể đi đến. Giả sử chúng được đánh số từ 0 đến 7 như hình sau :

Tọa độ (a,b)		1	2	3	4	5		Tọa độ (a,b)
(-2,-1)			4		3		1	(-2,1)
(-1,-2)		5				2	2	(-1,2)
	hàng x →						3	
(1,-2)		6				1	4	(1,2)
(2,-1)			7		0		5	(2,1)

cột y ↑

(8 bước đi có thể có của con ngựa)

Một phương pháp đơn giản để có được u, v từ x, y là ta dùng 2 mảng a và b lưu trử các sai biệt về tọa độ .Nếu ta dùng một chỉ số k để đánh số "bước đi kế " thì chi tiết đó được thể hiện bởi : u = x +a[k]; v = y + b[k]; $k = \overline{0,7}$.

Điều kiện "chấp nhận được" có thể được biểu diễn như kết hợp của các điều kiện :

Ô mới phải thuộc bàn cờ ($1 \le u \le n$ và $1 \le v \le n$) và chưa đi qua ô đó, nghĩa là h[u,v] = 0;

* Để ghi nhận nước đi hợp lệ ở bước i, ta gán h[u][v] = i; và để hủy một nước đi thì ta gán h[u][v] = 0.

* Ma trận h ghi nhận kết quả nghiệm. Nếu có <x,y> sao cho h<x,y> = 0 thì đó không phải là lời giải của bài giải , còn ngược là h chứa đường đi của ngựa.

Vậy thuật giải có thể mô tả như sau :

Input n, //Kích thước bàn cờ
x, y;//Toạ độ xuất phát ở bước i

Output h;

Mô ta :

```
Try(i, x, y) ≡
    for(k = 0; k <= 7; k++)
    {
```

```
            u  = x + a[k];
            v  = y + b[k];
            if (1 <= u ,v <= n &&h[u][v] == 0)
            {
                    h[u][v] = i;
                    if (i < n*n)
                            Try(i+1,u,v);
                    else
                            xuat_h(); // In ma trận h
            }
            h[u][v] = 0;
    }
```

Thủ tục này xuất tất cả các lời giải, nếu có.

Thủ tục đệ qui được khởi động bằng một lệnh gọi các tọa độ đầu x_0, y_0 là tham số. Ô xuất phát có trị 1, còn các ô khác được đánh dấu còn trống.

```
            H[x0 ][y0 ]  = 1;
            Try(2,x , y );
```

Các mảng a và b có thể khởi đầu như sau :

```
int a[8]= {2,1,-1,-2,-2,-1,1,2};
int b[8]= {1,2,2,1,-1,-2,-2,-1};
```

* Các lời giải sau là một số kết quả cho từ thuật giải trên :

	n=5	x=1	y=1	
1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

	n=6	x=2	y=3		
36	17	6	29	8	11
19	30	1	10	5	28
16	35	18	7	12	9
23	20	31	2	27	4
34	15	22	25	32	13
21	24	33	14	3	26

Ví dụ 1.13: Bài giải liệt kê các dãy nhị phân độ dài n.

Liệt kê các dãy có chiều dài n dưới dạng $x_1x_2...x_n$, trong đó $x_i \in \{ 0,1 \}$.

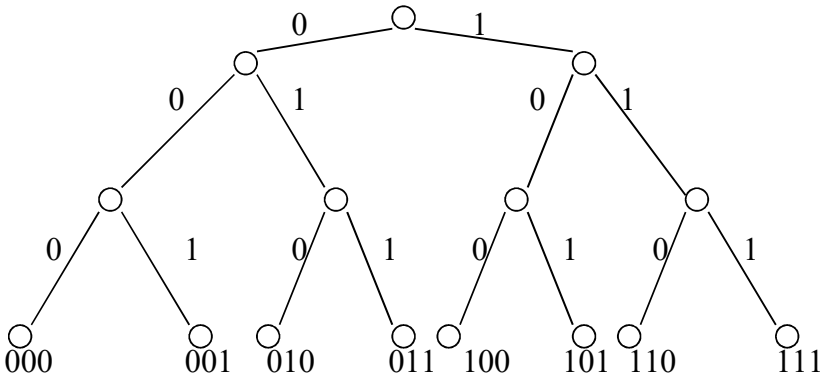
- Thiết kế thuật giải :

Ta có thể sử dụng sơ đồ tìm tất cả các lời giải của bài giải.Hàm Try(i) xác định x_i, trong đó x_i chỉ có 1 trong 2 giá trị là 0 hay 1. Các giá trị này mặc nhiên được chấp nhận mà không cần phải thoả mãn điều kiện gì. Nên Hàm try(i) có thể viết như sau :

```
Try ( i) ≡
    for (j = 0; j <= 1; j++)
    {
            x[i]   = j;
            if (i < n )
                    Try (i+1);
            else
                    Xuất(x);
```

```
}
```

Cây không gian các trạng thái của bài giải có thể mô tả bởi :



Bài tập

Bài 1: Tìm 2 ví dụ minh hoạ mối quan hệ giữa cấu trúc dữ liệu và thuật giải.

Bài 2: Tổ chức chương trình menu thực hiện trên các chuỗi với các chức năng:

- Tính chiều dài chuỗi
- Đảo ngược chuỗi
- copy chuỗi s vào chuỗi t.
- So sánh 2 chuỗi s và t
- Nối thêm chuỗi s vào sau chuỗi t

Yêu cầu:

- Hình thức 1: Tổ chức chương trình gồm 1 tập tin *.cpp, cài đặt hệ thống điều khiển menu, các hàm chức năng sử dụng kết quả cài đặt sẵn trong thư viện <string.h>.
- Hình thức 2: Tổ chức chương trình gồm 2 tập tin:
  - tập tin thư viện *.h (trong header files) chưa các định nghĩa hằng, dữ liệu, và cài đặt các hàm chức năng.
  - tập tin *.cpp (trong source files) cài đặt hàm main(), hệ thống điều khiển menu, nhập xuất dữ liệu.

  (xem lại cách tạo đề án bằng win32 console)

Bài 3:

Giả sử có một bảng giờ tàu cho biết thông tin về các chuyến tàu khác nhau của mạng đường sắt. Hãy biểu diễn các dữ liệu này bằng một cấu trúc dữ liệu thích hợp sao cho dễ dàng truy xuất giờ khởi hành, giờ đến của một chuyến tàu bất kỳ tại một nhà ga bất kỳ (Giả sử mỗi ngày đều có các chuyến tàu như thế)

Hướng dẫn:

- Mỗi chuyến tàu lưu trữ trong 1 cấu trúc chứa các thông tin:
  - Tên ga xuất phát
  - Giờ xuất phát
  - Tên ga đến
  - Khoảng cách (tính theo giờ)
  - Giá vé
  - …

```
struct CHUYENTAU
{
        char Gadi[20];
        char Gaden[20];
        GIO_XP Xp;
        int Tg;
        int Gv;
};
struct GIO_XP
{
        int Gio;
        int Phut;
};
```

- Bảng giờ tàu có thể lưu trữ trong mảng 1 chiều các cấu trúc trên

CHUYENTAU BangTau[MAX];

- Để khỏi phải nhập dữ liệu cho mỗi lần chạy chương trình, có thể khởi đầu mảng 1 chiều các cấu trúc, hoặc có thể tổ chức các hàm nhập dữ liệu cho mảng cấu trúc như sau:

```
void Setup()
{
        Chen_Ct("Saigon", "HaNoi",18,30,26,1200000);
        //...
}
void Chen_Ct(char gdi[20], char gden[20], int gio, int ph, int Tg, int Gv)
{
        if (Sct < MAX)
        {
                strcpy(BangTau[Sct].Gadi, gdi);
                strcpy(BangTau[Sct].Gaden, gden);
                //...
        }
}
```

Bài 4 :

Các giải vô địch bóng đá của các nước Anh (Premier League), Ý (Seria A), Tây Ban Nha (Laliga Santander) , mỗi giải có n (20) câu lạc bộ bóng đá chuyên nghiệp đá vòng tròn 2 lượt đi về xếp hạng với thể thức giải đấu như sau :

- *Tính điểm cho một trận đấu* : đội thắng – 3 điểm; hòa – 1 điểm ; thua – 0 điểm
- *Thực hiện xếp hạng dựa vào kết quả thi đấu của các đội, các tiêu chí xếp loại chung cuộc (ưu tiên theo thứ tự) sau :*
  - *Tổng điểm*
  - *Nếu có nhiều đội bằng điểm, xét tiếp các tiêu chí theo thứ tụ ưu tiên sau :*
    - Hiệu số bàn thắng thua
    - Số bàn thắng
- *Đội có kết quả cao hơn được xếp trên, các đội có các kết quả trùng nhau sẽ được xếp cùng hạng.* Nếu việc xếp cùng hạng đó ảnh hưởng đến tới chức vô địch, xuống hạng hay giành quyền tham dự một giải đấu khác, một trận play-off sẽ được diễn ra trên sân trung lập để xác định thứ hạng.
- Kết quả xếp hạng chung cuộc :
  - Chọn được 3 đội có kết quả cao nhất để trao giải : Vô địch (HC vàng), Nhì (HC bạc), ba (HC đồng)
  - 3 đội có điểm thấp nhất sẽ phải xuống hạng (chơi ở giải kế dưới vào năm sau).

Viết chương trình tùy chọn thực hiện các chức năng sau :

- Thoát khỏi chương trình
- Xem lịch thi đấu vòng j
- Xem lịch thi đấu toàn giải
- Xem kết quả các trận đấu vòng j
- Xem kết quả các trận đấu đến vòng j (từ vòng 1 đến vòng j)
- Xem kết quả xếp hạng đến vòng j
- Xem kết quả xếp hạng lượt đi

- Xem kết quả chung cuộc
- . . .

Yêu cầu : Mỗi giải sử dụng kết quả dữ liệu 2 năm : 2017-2018 (đã hoàn thành), 2018 – 2019 (đang diễn ra).

Bài 5:

Cho biết có bao nhiêu phép so sánh các dữ liệu trong mảng trong lệnh lặp sau:

```
for (g = 1; j < = n-1; j + +)
{
        a = j + 1;
        do
        {
                if (A[i] < A[j])
                        swap (A[i], A[j]);
                i + +;
        } while (i <= n)
} ;
```

Bài 6:

Tính số lần lặp các lệnh trong {…} trong lệnh sau:

```
for ( i = 0; i < n; i + +)
        for ( j = i + 1; i < = n; j + +)
                for ( k = 1; k < 10; k + +)
                { các lệnh };
```

Bài 8:

Đánh giá thời gian chạy của các đoạn chương trình sau:

a.

```
sum = 0;
for ( int i = 0; i < n; i + +)
        for ( int j = 0; j < n; j + +)
                sum + +;
```

b.

```
sum = 0;
```

```
for ( int i = 0; i < n; i + +)
        for ( int j = 0; j < n*n; j + +)
                for ( int k = 0; k < j; k + +)
                        sum + + ;
```

Bài 9: Đánh giá thời gian chạy của hàm đệ quy sau:

```
int Bart(int n)// n nguyên dương
{
        if ( n = = 1)
                return 1;
        else
        {
                result = 0;
                for ( int i = 2; i < = n; i + +)
                result + = Bart(i – 1);
                return result ;
        }
}
```

Bài 10: Tìm MinMax.

Tìm giá trị Min, Max trong đoạn a[l..r] của mảng a[0...n-1].

Bài 11 : Bài toán hoán đổi 2 phần trong 1 dãy.

a[0..n-1] là một mảng gồm n phần tử. Ta cần chuyển m phần tử đầu tiên của mảng với phần còn lại của mảng (n-1-m phân tử) mà không dùng một mảng phụ .

Chẳng hạn, với n = 8, a[] = (0,1, 2, 3, 4, 5, 6, 7)

Nếu m = 3, thì kết quả là : (3, 4, 5, 6, 7, 0, 1, 2)

Nếu m = 5, thì kết quả là : (5, 6, 7, 0, 1, 2, 3, 4)

Nếu m = 4, thì kết quả là : (4, 5, 6, 7, 0, 1, 2, 3)

Bài 12: Bài giải liệt kê các hoán vị.

Liệt kê các hoán vị của n số nguyên dương đầu tiên.

Bài 13: Bài giải liệt kê các tổ hợp.

Liệt kê các tổ hợp chặp k trong n phần tử.

Chương 2. Tìm kiếm và sắp xếp trong

2.1. Các phương pháp tìm kiếm trong

Phương pháp tìm kiếm trong thường xuyên sử dụng trong đời sống hàng ngày cũng như trong xử lý tin học.

Cho một dãy X gồm n phần tử x0,x1,…,xN-1 và một phần tử item có cùng kiểu dữ liệu T với dãy. Bài giải đặt ra là hãy tìm trong dãy X có chứa item hay không?

Việc tìm kiếm sẽ xảy ra một trong hai trường hợp sau:

(1) Có phần tử trong dãy mà giá trị tương ứng bằng item cần tìm: phép tìm kiếm được thỏa

(2) Không tìm được phần tử nào có giá trị tương ứng bằng giá trị item cần tìm: phép tìm kiếm không thỏa

Bài giải có thể mô tả như sau:

Input: $X = \{ x_0, x_1, \ldots, x_{N-1} \}$

Item; // dữ liệu cần tìm

Output: 0; nếu không tìm thấy

1, nếu tìm thấy.

2.1.1. Phương pháp tìm kiếm tuyến tính (Linear Search)

a. Ý tưởng Thuật giải

Phương pháp tìm kiếm tuyến tính là tìm tuần tự từ đầu đến cuối dãy.

b. Mô tả thuật giải

```
TìmTuyếnTính(X, N, item)
{
    Bước 1: chiso = 0;
    Bước 2:
            nếu (chiso<N) và (x[chiso] != item)
                    chiso=chiso+1
                    Quay lai bước 2
            Ngược lại chuyển sang bước 3
    Bước 3:
            nếu chiso = N
                    return 0;
    return 1;
}
```

c. Cài đặt:

```
int TimTuyenTinh(mang X, int N, DL item)
{
    int cs = 0;
    while(cs<N && X[cs] !=item)
            Cs++;
    if(cs==N)
            return  0;
    return 1;
}
```

d. Độ phức tạp

Ta thấy với thuật giải trên, trong trường hợp tốt nhất chỉ cần một phép so sánh, tức là phần tử cần tìm nằm ngay đầu dãy, còn trường hợp xấu nhất cần N+1 phép so sánh. Giả sử xác suất các phần tử trong mảng nhận giá trị x là như nhau, trường hợp trung bình là $\frac{N+1}{2}$. Vậy thuật giải tìm kiếm tuyến tính có độ phức tạp tính giải cấp N: T(N)=O(N).

Cải tiến:

Để giảm bớt số phép so sánh chỉ số trong biểu thức điều kiện của lệnh if và lệnh while, ta dùng một biến phụ đóng vai trò là lính canh bên phải (hay trái) x_n = item cần tìm (x_0 trong trường hợp dãy đánh số từ 1)

Mô tả thuật giải:

```
TimTuyenTinh_LinhCanh(X, N, item)
{
    Bước 1:
            chiso=0;
            x[N] = item
    Bước 2:
            Nếu(x[chiso] != item)
                    chiso=chiso+1
                    Quay lai bước 2
            Ngược lại chuyển sang bước 3
    Bước 3:
            Nếu chiso = N
                    return 0; //không thấy
    return 1; //tìm thấy
}
```

2.1.2 Tìm kiếm nhị phân (Binary Search)

Thuật giải tìm kiếm nhị phân chỉ dùng cho các dãy số có thứ tự (tăng hay giảm). Không mất tính tổng quát ta xét các dãy có thứ tự tăng.

a. Ý tưởng Thuật giải

Trước tiên so sánh phần tử giữa trong dãy, nếu item bằng phần tử giữa thì kết luận tìm thấy và dừng thuật giải tìm kiếm, ngược lại, nếu item cần tìm có giá trị nhỏ hơn phần tử giữa thì ta chỉ tìm item trong nửa dãy có giá trị nhỏ hơn phần tử giữa, ngược lại tìm item trong nửa dãy có giá trị lớn hơn phần tử giữa.

b. Mô tả thuật giải

Thuật giải được mô tả như sau:

```
TimNhiPhan(X, N, item)
{
    Bước 1:
        csdau = 0; //chỉ số đầu
        cscuoi = N-1; //chỉ số cuối
    Bước 2:
        Nếu (csdau <= cscuoi)
            csgiua = (csdau + cscuoi)/2
            Nếu (X[csgiua] = item)
                return 1;
        Ngược lại
            Nếu (item < X[csgiua])
                cscuoi = csgiua – 1
            Ngược lại
                csdau = csgiua + 1;
            Quay lại bước 2
    return 0; //không tìm thấy
}
```

c. Cài đặt

```
//X: dãy tăng, N: số phần tử, item: dữ liệu cần tìm
int TimNhiPhan(mang X, int N, DL item)
{
    int csdau = 0, cscuoi=N-1, csgiua;
    while(csdau<=cscuoi)
    {
        csgiua = (csdau + cscuoi)/2;
        if(X[csgiua]==item) return 1;
        else
            if (X[csgiua] > item)
                cscuoi = csgiua-1
            else
                csdau = csgiua+1
    }
    return 0;// không tìm thấy
}
```

d. Độ phức tạp

Trường hợp	Số lần so sánh	Giải thích
Tốt nhất	**1**	**Phần tử giữa mảng có giá trị item**
Xấu nhất	**$\log_2 N$**	**Không có item trong mảng**
Trung bình	**$\log_2(N/2)$**	**Giả sử xác suất các phần tử trong mảng có giá trị item như nhau**

Thuật giải tìm nhị phân có độ phức tạp tính giải:

$T(N) = O(\log_2 N)$

2.1.3 Phương pháp tìm kiếm nội suy (Interpolation Search):

- Ý tưởng:

Tìm kiếm nội suy là phương pháp cải tiến tìm kiếm nhị phân, nên tập dữ liệu đầu vào cho phương pháp này phải có thứ tự. Giả sử dữ liệu là một dãy tăng.

Phương pháp tìm kiếm nội suy hoạt động như thuật giải tìm kiếm nhị phân, chỉ khác ở điểm xác định phần tử để chia mảng thành 2 phần.

Nếu trong thuật giải tìm kiếm nhị phân, phần tử trong mảng dùng để so khớp giá trị cần tìm là phần tử giữa mảng xác định bởi chỉ số giữa trong mỗi bước lặp tìm kiếm:

csgiua = (csdau + cscuoi)/2;

Thực chất của biểu thức trên là:

$$csgiua = csdau + \frac{1}{2} * (\text{csc}uoi - csdau)$$

với ý nghĩa thêm vào biên trái csdau một giá trị bằng nửa (1/2) đoạn đang xét ta được csgiua, thì trong tìm kiếm nội suy 1/2 sẽ được thay thế bằng giá trị:

$$\frac{item - x[csdau]}{x[\text{csc}uoi] - x[csdau]}$$

cho nên phần tử trong mảng dùng để so khớp giá trị cần tìm là phần tử xác định bởi chỉ số sau đây trong mỗi bước lặp tìm kiếm:

$$csgiua = csdau + \frac{item - x[csdau]}{x[\text{csc}uoi] - x[csdau]} * (\text{csc}uoi - csdau);$$

- Cài đặt thuật giải:

```
int TimNoiSuy(int a[MAX], int n, int item)
{
	int csdau = 0, cscuoi = n - 1, csgiua;
	while (csdau <= cscuoi)
	{
		csgiua=csdau+(cscuoi-csdau)*(item-x[csdau]) / (x[cscuoi] -  x[csdau]);

		if (a[csgiua] == item)
			return 1; // tìm thấy
		else
			if (a[csgiua] > item)
				cscuoi = csgiua - 1;
```

```
                else
                        csdau = csgiua + 1;
        }
        return 0;// không tìm thấy
}
```

Người ta chứng minh được rằng độ phức tạp tính giải của thuật giải nội suy là lglgN + 1, hiệu quả hơn nhiều thuật giải nhị phân.

2.2. Các phương pháp sắp xếp trong

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.

Cho trước một dãy số a_0, a_1,..., a_{N-1}**,** sắp xếp dãy số a_0, a_1,...,a_{N-1} là thực hiện việc bố trí lại các phần tử sao cho hình thành được dãy mới a_{k0}, a_{k1},...,a_{kN-1} có thứ tự (giả sử xét thứ tự tăng) nghĩa là $a_{ki} > a_{ki-1}$.

Mà để quyết định được những tình huống cần thay đổi vị trí các phần tử trong dãy, cần dựa vào kết quả của một loạt phép so sánh. Chính vì vậy, hai thao tác so sánh và gán là các thao tác cơ bản của hầu hết các thuật giải sắp xếp.

Khi xây dựng một thuật giải sắp xếp cần chú ý tìm cách giảm thiểu những phép so sánh và đổi chỗ không cần thiết để tăng hiệu quả của thuật giải.

Đối với các dãy số được lưu trữ trong bộ nhớ chính, nhu cầu tiết kiệm bộ nhớ được đặt nặng, do vậy những thuật giải sắp xếp đòi hỏi cấp phát thêm vùng nhớ để lưu trữ dãy kết quả ngoài vùng nhớ lưu trữ dãy số ban đầu thường ít được quan tâm.

Phần này giới thiệu một số thuật giải sắp xếp từ đơn giản đến phức tạp có thể áp dụng thích hợp cho việc sắp xếp trong.

2.2.1 Thuật giải sắp xếp chọn (Selection Sort)

a. Ý tưởng

Ta thấy rằng, nếu mảng có thứ tự, giả sử xét thứ tự tăng, phần tử a_i luôn là min(a_i, a_{i+1}, .., a_{N-1}). Ý tưởng của thuật giải chọn trực tiếp mô phỏng một trong những cách sắp xếp tự nhiên nhất trong thực tế: chọn phần tử nhỏ nhất trong N phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu dãy hiện hành; sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn N-1 phần tử của dãy ban đầu, bắt đầu từ vị trí thứ 2; lặp lại

quá trình trên cho dãy hiện hành... đến khi dãy hiện hành chỉ còn 1 phần tử. Dãy ban đầu có N phần tử, vậy tóm tắt ý tưởng thuật giải là thực hiện N-1 lượt việc đưa phần tử nhỏ nhất trong dãy hiện hành về vị trí đúng ở đầu dãy.

Các bước tiến hành như sau:

b. Mô tả thuật giải:

- Bước 1: i = 0;
- Bước 2: Tìm phần tử a[min] nhỏ nhất 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. //N-1 phần tử đã nằm đúng vị trí.

Ví dụ 2.1:

Cho mảng a như sau:

i	0	1	2	3	4	5	6	7
a[i]	25	7	15	8	18	6	4	0

Các bước thuật giải chạy để sắp xếp mảng a có thứ tự tăng như sau:

i = 0:	0	7	15	8	18	6	4	25
i = 1:	0	4	15	8	18	6	4	25
i = 2:	0	4	6	8	18	15	7	25
i = 3:	0	4	6	7	18	15	8	25
i = 4:	0	4	6	7	8	15	18	25
i = 5:	0	4	6	7	8	15	18	25
i = 6:	0	4	6	7	8	15	18	25
i = 7:	Dừng							

c. Cài đặt

Cài đặt thuật giải sắp xếp chọn trực tiếp thành hàm SelectionSort

```
void SelectionSort(int a[],int N )
{
    int i, j, Cs_min; // chỉ số phần tử nhỏ nhất trong dãy hiện hành
    for (i=0; i<N-1 ; i++)
```

```
    {
        Cs_min = i;
        for( j = i+1; j <N ; j++)
            if (a[j ] < a[Cs_min])
                    Cs_min = j; // ghi nhận vị trí phần tử hiện nhỏ nhất
        Hoanvi(a[Cs_min], a[i]);
    }
}
```

d. Đánh giá thuật giải

Đối với thuật giải chọn trực tiếp, có thể thấy rằng ở lượt thứ i, bao giờ cũng cần (N-i) lần so sánh để xác định phần tử nhỏ nhất hiện hành. Số lượng phép so sánh này không phụ thuộc vào tình trạng của dãy số ban đầu, do vậy trong mọi trường hợp có thể kết luận:

Số lần so sánh là

$$\sum_{i=1}^{N-1}(N-i) = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2}$$

Số lần hoán vị (một phép hoán vị cần ba phép gán) phụ thuộc vào tình trạng ban đầu của dãy, ta có thể ước lượng trong từng trường hợp như sau:

Trường hợp	Số lần so sánh	Số phép gán
Tốt nhất	N(N-1)/2	0
Xấu nhất	N(N-1)/2	3N(N-1)/2

2.2.2 Thuật giải sắp xếp chèn (Insertion Sort)

a. Ý tưởng

Giả sử có một dãy $a_0, a_1,...,a_{N-1}$ trong đó i phần tử đầu tiên $a_0, a_1,...,a_{i-1}$ đã có thứ tự. Ý tưởng chính của phương pháp chèn trực tiếp là tìm cách chèn phần tử a_i vào vị trí thích hợp của đoạn đã được sắp để có dãy mới $a_0, a_1,...,a_i$ có thứ tự.

Vị trí này có thể là:

- Trước a_0
- Sau a_{i-1}
- Giữa hai phần tử $\mathbf{a_{k-1}}$ và $\mathbf{a_k}$ thỏa $\mathbf{a_{k-1} \le a_i < a_k}$ $(1 \le k \le i-1)$.

Cho dãy ban đầu $a_0, a_1,...,a_{N-1}$, ta có thể xem như đã có đoạn gồm một phần tử a_0 đã được sắp, sau đó thêm a_1 vào đoạn a_0 sẽ có đoạn a_0 a_1 được sắp; tiếp tục thêm a_2 vào

đoạn a_0 a_1 để có đoạn a_0 a_1 a_2 được sắp; tiếp tục cho đến khi thêm xong a_{N-1} vào đoạn a_0 a_1 ... a_{N-2} sẽ có dãy a_0 a_1.... a_{N-1} được sắp. Các bước tiến hành như sau:

b. Mô tả thuật giải:

Bước 1: i = 1; *// đoạn có 1 phần tử* a[0] *đã được sắp*

Bước 2: x = a[i]; Tìm vị trí pos thích hợp trong đoạn a[0] đến a[i-1] để chèn x vào.

Bước 3: Dời các phần tử từ a[pos] đến a[i-1] sang phải 1 vị trí để dành chỗ cho a[i].

Bước 4: a[pos] = x; *// có đoạn* a[0]..a[i] *đã được sắp*

Bước 5: i = i+1;

Nếu i < N-1 : Lặp lại Bước 2.

Ngược lại : Dừng.

Ví dụ 2.2:

Xét dãy a trong ví dụ trên. Chạy thuật giải sắp xếp chèn

i = 1:	7	25	15	8	18	6	4	0
i = 2:	7	15	25	8	18	6	4	0
i = 3:	7	8	15	25	18	6	4	0
i = 4:	7	8	15	18	25	6	4	0
i = 5:	6	7	8	15	18	25	4	0
i = 6:	4	6	7	8	15	18	25	0
i = 7:	0	4	6	7	8	15	18	25

c. Cài đặt

Cài đặt Thuật giải sắp xếp chèn trực tiếp thành hàm InsertionSort

```
void InsertionSort(int a[], int N )
{
    int pos, i, x;
    for( i=1 ; i<N ; i++) //đoạn a[0], ... a[i-1] đã sắp
    {
        x = a[i]; //lưu giá trị a[i] tránh bị ghi đè khi dời chỗ các phần tử.
        pos = i-1;
        // tiến về trái tìm vị trí chèn x
        while((pos >= 0)&&(a[pos] > x))
        {// dời chỗ các phần tử sẽ đứng sau x
            a[pos+1] = a[pos];
            pos--;
        }
        a[pos+1] = x;// chèn x vào dãy
    }
}
```

Nhận xét

Khi tìm vị trí thích hợp để chèn a[i] vào đoạn a[0] đến a[i-1], do đoạn đã được sắp, nên có thể sử dụng thuật giải tìm nhị phân để thực hiện việc tìm vị trí pos, khi đó có thuật giải sắp xếp chèn nhị phân:

```
void  BInsertionSort(int a[], int N )
{
    int  l,r,m;
    int  i,j;
    int x;
    for(i=1 ; i<N ; i++)
    {
            x = a[i]; l = 0; r = i-1;
            while(l<=r)
            {
                    m = (l+r)/2;
                    if(x < a[m])
                            r = m-1;
                    else
                            l = m+1;
            }
            for( j = i-1 ; j >=l ; j--) // dời các phần tử sẽ đứng sau x
                    a[j+1] = a[j];
            a[l] = x; // chèn x vào dãy
    }
}
```

d. Đánh giá thuật giải

Đối với thuật giải chèn trực tiếp, các phép so sánh xảy ra trong mỗi vòng lặp while tìm vị trí thích hợp pos, và mỗi lần xác định vị trí đang xét không thích hợp, sẽ dời chỗ phần tử a[pos] tương ứng. Thuật giải thực hiện tất cả N-1 vòng lặp while, do số lượng phép so sánh và dời chỗ này phụ thuộc vào tình trạng của dãy số ban đầu, nên chỉ có thể ước lược trong từng trường hợp như sau:

Trường hợp	Số lần so sánh	Số phép gán
Tốt nhất	$\sum_{i=1}^{N-1} 1 = N-1$	$\sum_{i=1}^{N-1} 1 = 2(N-1)$
Xấu nhất	$\sum_{i=1}^{N-1} i = \frac{N(N-1)}{2}$	$\sum_{i=1}^{N-1} (i+1) = \frac{N(N+1)}{2} - 1$

2.2.3 Thuật giải sắp xếp đổi chỗ trực tiếp (Interchange Sort)

Để sắp xếp một dãy số, ta có thể xét các nghịch thế (tức là các cặp phần tử a[j], a[i] với a[j]<a[i] và j>i) có trong dãy và làm triệt tiêu dần chúng đi.

a. Ý tưởng

Xuất phát từ đầu dãy, tìm tất cả các nghịch thế chứa phần tử này, triệt tiêu chúng bằng cách đổi chỗ phần tử này với phần tử tương ứng trong cặp nghịch thế. Lặp lại việc xử lý trên với các phần tử tiếp theo trong dãy.

b. Mô tả thuật giải:

Bước 1: i=0; // bắt đầu từ đầu dãy
Bước 2: j=i+1;//tìm các phần tử a[j]<a[i], j>i

Bước 3:

Trong khi j<N thực hiện

Nếu a[j]<a[i]: a[i]↔a[j]; //xét cặp phần tử a[i], a[j]

j = j+1;

Bước 4: i=i+1;

Nếu i<N-1: lặp lại bước 2

Ngược lại: dừng

Ví dụ 2.3:

Sắp xếp tăng dãy a (trong ví dụ trên) theo thuật giải đổi chỗ trực tiếp, kết quả các bước như sau:

i = 0:	0	25	15	8	18	7	6	4
i = 1:	0	4	25	15	18	8	7	6
i = 2:	0	4	6	25	18	15	8	7
i = 3:	0	4	6	7	25	18	15	8
i = 4:	0	4	6	7	8	25	18	15
i = 5:	0	4	6	7	8	15	25	18
i = 6:	0	4	6	7	8	15	18	25

c. Cài đặt

```
void InterchangeSort(int a[], int N )
{
        int  i, j;
        for (i = 0 ; i<N-1 ; i++)
                for (j =i+1; j < N ; j++)
                  if(a[j ]< a[i]);
```

```
            Hoanvi(a[i],a[j]);
}
```

d. Đánh giá thuật giải

Số lượng các phép so sánh xảy ra không phụ thuộc vào tình trạng của dãy ban đầu. Số lượng các phép hoán vị tùy thuộc vào kết quả của phép so sánh. Ta chỉ có thể ước lược trong từng trường hợp như sau:

Trường hợp	Số lần so sánh	Số phép gán
Tốt nhất	$\sum_{i=1}^{N-1}(N-i)=\frac{N(N-1)}{2}$	0
Xấu nhất	$\frac{N(N-1)}{2}$	$\frac{N(N-1)}{2}$

2.2.4 Thuật giải sắp xếp nổi bọt (Bubble Sort)

a. Ý tưởng

Ý tưởng chính của thuật giải là xuất phát từ cuối (đầu) dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ (lớn) hơn trong cặp phần tử đó về vị trí đúng đầu (cuối) 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 xử lý trên cho đến khi không còn cặp phần tử nào để xét.

b. Mô tả thuật giải:

- Bước 1: i = 0; // lần xử lý đầu tiên
- Bước 2: j = N-1; //Duyệt từ cuối dãy ngược về vị trí i

  Trong khi (j < i) thực hiện:

  Nếu a[j]<a[j-1]: a[j]↔a[j-1];//xét cặp phần tử kế cận

  j = j-1;
- Bước 3: i = i+1; // lần xử lý kế tiếp

  Nếu i = N-1: Hết dãy. Dừng

  Ngược lại: lặp lại bước 2

Ví dụ 2.4:

Kết quả của thuật giải khi thực hiện trên dãy số trong các ví dụ trên:

```
i = 0:    0   25    7   15    8   18    6    4
i = 1:    0    4   25    7   15    8   18    6
```

i = 2:	0	4	6	25	7	15	8	18
i = 3:	0	4	6	7	25	8	15	18
i = 4:	0	4	6	7	8	15	25	18
i = 5:	0	4	6	7	8	15	18	25
i = 6:	0	4	6	7	8	15	18	25

c. Cài đặt

```
void BubbleSort(int a[], int N )
{   int i, j;
    for (i = 0 ; i<N-1 ; i++)
        for (j =N-1; j >i ; j --)
          if(a[j]< a[j-1])
                Hoanvi(a[j],a[j-1]);
}
```

d. Đánh giá thuật giải

Đối với thuật giải nổi bọt, số lượng các phép so sánh xảy ra không phụ thuộc vào tình trạng của dãy số ban đầu, nhưng số lượng phép hoán vị thực hiện tùy thuộc vào kết qủa so sánh, có thể ước lược trong từng trường hợp như sau:

Trường hợp	Số lần so sánh	Số phép gán
Tốt nhất	$\sum_{i=1}^{N-1}(N-i)=\frac{N(N-1)}{2}$	0
Xấu nhất	$\frac{N(N-1)}{2}$	$\frac{N(N-1)}{2}$

Nhận xét

BubbleSort có các khuyết điểm sau: không nhận diện được tình trạng dãy đã có thứ tự hay có thứ tự từng phần. Các phần tử nhỏ được đưa về vị trí đúng rất nhanh, trong khi các phần tử lớn lại được đưa về vị trí đúng rất chậm.

2.2.5 Thuật giải vun đống (Heap Sort)

a. Ý tưởng

Cải tiến thuật giải Chọn trực tiếp.

Khi tìm phần tử nhỏ nhất ở bước i, phương pháp chọn trực tiếp không tận dụng được các thông tin đã có trong các phép so sánh ở bước i-1. Thuật giải Heap Sort khắc phục nhược điểm này.

Khái niệm heap và thuật giải Heapsort do J.Williams đề xuất.

b. Định nghĩa Heap:

Giả sử xét trường hợp sắp xếp tăng dần, khi đó Heap được định nghĩa là một dãy các phần tử a_l, a_2,..., a_r thoả các quan hệ sau với mọi i $\in$ [l, r]:

- $a_i >= a_{2i+1}$
- $a_i >= a_{2i+2}$ {(a_i, a_{2i+1}), (a_i,a_{2i+2}) là các cặp phần tử liên đới }

Heap có các tính chất sau:

- Tính chất 1: Nếu a_l, a_2,..., a_r là một heap thì khi cắt bỏ một số phần tử ở hai đầu của heap, dãy con còn lại vẫn là một heap.
- Tính chất 2: Nếu a_0, a_2,..., a_{N-1} là một heap thì phần tử a_0 (đầu heap) luôn là phần tử lớn nhất trong heap.
- Tính chất 3: Mọi dãy a_l, a_2,..., a_r với 2l > r là một heap.
- Tính chất 4: Nếu dãy a_0,…,a_{N-1} là một heap thì ta có thể mô tả"liên tiêp" những phần tử của dãy này trên một cây nhị phân có tính chất:
  - Con trái (nếu có) của a_i là a_{2i+1}<= a_i
  - Và con phải (nếu có) của a_i là a_{2i+2}<=a_i

c. Thuật giải HeapSort

Thuật giải Heapsort trải qua 2 giai đoạn:

- Giai đoạn 1:Hiệu chỉnh dãy số ban đầu thành heap;
- Giai đoạn 2: Sắp xếp dãy số dựa trên heap:
  - Bước 1:

    Đưa phần tử nhỏ nhất về vị trí đúng ở cuối dãy: r = N-1;
    Hoánvị (a_0, a_r);
  - Bước 2:

    Loại bỏ phần tử nhỏ nhất ra khỏi heap: r = r-1;
    Hiệu chỉnh phần còn lại của dãy từ a_0, a_2, ..., a_r thành một heap.

- Bước 3:

  Nếu r>1 (heap còn phần tử): Lặp lại Bước 2

  Ngược lại: Dừng

Vậy thuật giải Heapsort cần các thao tác:

- Tạo heap đầy đủ ban đầu từ một heap ban đầu.
- Bổ sung một phần tử vào bên trái của một heap để tạo ra một heap dài hơn một phần tử.

d. Cài đặt

Để cài đặt thuật giải Heapsort cần xây dựng các thủ tục phụ trợ:

- Thủ tục hiệu chỉnh dãy a_l, a_{l+1}, …, a_r thành heap:

  void Shift (int a[], int l, int r)

- Thủ tục hiệu chỉnh dãy a_0, a_2, …, a_{N-1} thành heap:

  void CreateHeap(int a[], int N)

d1. Thủ tục hiệu chỉnh dãy a_l, a_{l+1}, …, a_r thành heap:

- Giả sử có dãy $\mathbf{a_l}$, $\mathbf{a_{l+1}}$ **...**$\mathbf{a_r}$, trong đó đoạn $\mathbf{a_{l+1}}$ **...**$\mathbf{a_r}$, đã là một heap, ta cần xây dựng hàm hiệu chỉnh $\mathbf{a_l}$, $\mathbf{a_{l+1}}$ **...**$\mathbf{a_r}$ thành heap.
- Để làm điều này, ta lần lượt xét quan hệ của một phần tử $\mathbf{a_i}$ nào đó với các phần tử liên đới của nó trong dãy là $\mathbf{a_{2i+1}}$ và $\mathbf{a_{2i+2}}$, nếu vi phạm điều kiện quan hệ của heap, thì đổi chỗ $\mathbf{a_i}$ với phần tử liên đới thích hợp của nó.
- Lưu ý việc đổi chỗ này có thể gây phản ứng dây chuyền:

```
void Shift (int a[ ], int l, int r )
{
        int     x,i,j;
        i = l; j =2*i+1;//(ai, aj ),(ai, aj+1)là các phần tử liên đới
        x = a[i];
        while (j<=r)
        {
                if (j<r)  //nếu có hai phần tử liên đới
                if (a[j]<a[j+1])// xác định phần tử liên đới lớn nhất
                        j = j+1;
                if (a[j]<=x)
                        return;//thỏa quan hệ liên đới, dừng
                else
```

```
            {
                    a[i] = a[j];
            i = j;  //xét tiếp khả năng hiệu chỉnh lan truyền
            j = 2*i+1;
            a[i] = x;
            }
    }
```

d2. Hiệu chỉnh dãy $a_0, a_2 ... a_{N-1}$ thành heap:

Cho một dãy bất kỳ a_0, a_1, ..., a_{N-1}, theo tính chất 3, ta có dãy $a_{(N-1)/2 +1}$, ... a_{N-1} đã là một heap. Ghép thêm phần tử $a_{(N-1)/2}$ vào bên trái heap hiện hành và hiệu chỉnh lại dãy $a_{(N-1)/2}$, $a_{(N-1)/2+1}$, ...,a_{N-1} thành heap.

```
void CreateHeap(int a[], int N )
{
    int  l;
    l = (N-1)/2;    // a[l] là phần tử ghép thêm
    while (l >= 0)
    {
        Shift(a,l,N-1);
        l = l -1;
    }
}
```

<u>Thủ tục sắp xếp HeapSort:</u>

```
void HeapSort (int a[], int N)
{
    int   r;
    CreateHeap(a,N);
    r = N-1; // r là vị trí đúng cho phần tử nhỏ nhất
    while(r > 0)
    {
        Hoanvi(a[0],a[r]);
        r = r -1;
        Shift(a,0,r);
    }
}
```

Đánh giá thuật giải

Việc đánh giá thuật giải Heapsort rất phức tạp, nhưng đã chứng minh được trong trường hợp xấu nhất độ phức tạp là $O(n\log_2 n)$.

2.2.6 Thuật giải sắp xếp nhanh (Quick Sort)

a. Ý tưởng

Để sắp xếp dãy a_0, a_1, ...,a_{N-1} thuật giải QuickSort dựa trên việc phân hoạch dãy ban đầu thành hai phần:

- Dãy con 1: Gồm các phần tử a_0.. a_j có giá trị không lớn hơn x
- Dãy con 2: Gồm các phần tử a_i.. a_{N-1} có giá trị không nhỏ hơn x

với x là giá trị của một phần tử tùy ý trong dãy ban đầu.

Sau khi thực hiện phân hoạch, dãy ban đầu được phân thành 3 phần:

- $a_k < x$, với k = 0..j
- $a_k = x$, với k = j+1...i-1
- $a_k > x$, với k = i...N-1

trong đó dãy con thứ 2 đã có thứ tự, nếu các dãy con 1 và 3 chỉ có 1 phần tử thì chúng cũng đã có thứ tự, khi đó dãy ban đầu đã được sắp. Ngược lại, nếu các dãy con 1 và 3 có nhiều hơn 1 phần tử thì dãy ban đầu chỉ có thứ tự khi các dãy con 1, 3 được sắp. Để sắp xếp dãy con 1 và 3, ta lần lượt tiến hành việc phân hoạch từng dãy con theo cùng phương pháp phân hoạch dãy ban đầu vừa trình bày.

b. Thuật giải phân hoạch dãy a_l, a_{l+1}, ..,a_r thành 2 dãy con:

Bước 1:

Chọn tùy ý một phần tử a[k] trong dãy là giá trị mốc, $l \le k \le r$:

x = a[k]; i = l; j = r;

Bước 2: Phát hiện và hiệu chỉnh cặp phần tử a[i], a[j] nằm sai chỗ:

Bước 2a: Trong khi (a[i]<x) i++;

Bước 2b: Trong khi (a[j]>x) j--;

Bước 2c:

Nếu i< j // $a[i] \ge x \ge a[j]$ mà a[j] đứng sau a[i]

Hoán vị (a[i],a[j]);

Bước 3:

Nếu i < j: Lặp lại Bước 2.//chưa xét hết mảng

Nếu $i \ge j$: Dừng

Nhận xét

- Về nguyên tắc, có thể chọn giá trị mốc x là một phần tử tùy ý trong dãy, nhưng để đơn giản, dễ diễn đạt thuật giải, phần tử có vị trí giữa thường được chọn, khi đó k = (l+r)/ 2.

- Giá trị mốc x được chọn sẽ có tác động đến hiệu quả thực hiện thuật giải vì nó quyết định số lần phân hoạch. Số lần phân hoạch sẽ ít nhất nếu ta chon được x là phần tử median của dãy. Tuy nhiên do chi phí xác định phần tử median quá cao nên trong thực tế người ta không chọn phần tử này mà chọn phần tử nằm chính giữa dãy làm mốc với hy vọng nó có thể gần với giá trị median

c. Thuật giải sắp xếp phân hoạch

Sắp xếp dãy a_l, a_{l+1}, ..,a_r

Có thể phát biểu thuật giải sắp xếp QuickSort một cách đệ qui như sau:

- Bước 1: Phân hoạch dãy a_l…a_r thành các dãy con:
  - Dãy con 1: a_0 .. $a_j \le x$
  - Dãy con 2: a_{j+1} .. $a_{i-1} = x$
  - Dãy con 3: a_i .. $a_r \ge x$
- Bước 2:

```
Nếu ( l < j ) // dãy con 1 có nhiều hơn 1 phần tử
        Phân hoạch dãy a0 .. aj
Nếu ( i < r ) // dãy con 3 có nhiều hơn 1 phần tử
        Phân hoạch dãy ai .. ar
```

Ví dụ 2.6:

	i	0	1	2	3	4	5	6	7
	a[i]	25	7	15	8	18	6	4	0

Phân hoạch đoạn [l..r], l = 0; r = 7, x = a[(l+r) /2] = a[3] = 8

	0	7	4	6	18	8	15	25

Phân hoạch đoạn [l..r], l = 0; r = 3, x = a[(l+r) /2] = a[1] = 7

0	6	4	7			

Phân hoạch đoạn [l..r], l = 0; r = 2, x = a[(l+r) /2] = a[1] = 6

0	4	6	

Phân hoạch đoạn [l..r], l = 0; r = 1, x = a[(l+r) /2] = a[0] = 0

0	4	

Phân hoạch đoạn [l..r], l = 4; r = 7, x = a[(l+r) /2] = a[5] = 8

8	18	15	25

Phân hoạch đoạn [l..r], l = 5; r = 7, x = a[(l+r) /2] = a[6] = 18

15	18	25

Phân hoạch đoạn [l..r], l = 6; r = 7, x = a[(l+r) /2] = a[6] = 18

18	25

Kết quả:

0	4	6	7	8	15	18	25

d. Cài đặt

```
void QuickSort(int a[], int l, int r)
{
        int  i,j,x;
        x = a[(l+r)/2];          //chọn phần tử giữa làm mốc
        i =l; j = r;
        do
        {
                while(a[i] < x) i++;
                while(a[j] > x) j--;
                if(i <= j)
                {
                        Hoanvi(a[i],a[j]);
                        i++ ; j--;
                }
        }
        while(i <= j);
        if(l < j) QuickSort(a,l,j);
        if(i < r) QuickSort(a,i,r);
    }
```

e. Đánh giá Thuật giải

Hiệu quả thực hiện của thuật giải QuickSort phụ thuộc vào việc chọn giá trị mốc. Trường hợp tốt nhất xảy ra nếu mỗi lần phân hoạch 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 chia thành 2 phần bằng nhau và cần $\log_2(N)$ lần phân hoạch thì sắp xếp xong. Nhưng nếu mỗi lần phân hoạch lại chọn nhằm phần tử có giá trị cực đại (hay cực tiểu) là mốc, dãy sẽ bị phân chia thành 2 phần không đều: một phần chỉ có 1 phần tử, phần còn lại gồm (N-1) phần tử, do vậy cần phân hoạch n lần mới sắp xếp xong.

Ta có bảng tổng kết

Trường hợp	Số lần so sánh
Tốt nhất	NlogN
Trung bình	NlogN
Xấu nhất	N^2

2.2.7 Thuật giải sắp xếp trộn (Merge Sort)

a. Ý tưởng

Để sắp xếp dãy $a_0, a_1, ..., a_{N-1}$, thuật giải Merge Sort dựa trên nhận xét sau:

Mỗi dãy $a_0, a_1, ..., a_{N-1}$ bất kỳ đều có thể coi như là một tập hợp các dãy con liên tiếp mà mỗi dãy con đều đã có thứ tự. Ví dụ dãy 12, 2, 8, 5, 1, 6, 4, 15 có thể coi như gồm 5 dãy con không giảm (12); (2, 8); (5); (1, 6); (4, 15).

Dãy đã có thứ tự coi như có 1 dãy con.

Như vậy, một cách tiếp cận để sắp xếp dãy là tìm cách làm giảm số dãy con không giảm của nó. Đây chính là hướng tiếp cận của thuật giải sắp xếp theo phương pháp trộn. Trong phương pháp Merge sort, mấu chốt của vấn đề là cách phân hoạch dãy ban đầu thành các dãy con. Sau khi phân hoạch xong, dãy ban đầu sẽ được tách ra thành 2 dãy phụ theo nguyên tắc phân phối đều luân phiên. Trộn từng cặp dãy con của hai dãy phụ thành một dãy con của dãy ban đầu, ta sẽ nhân lại dãy ban đầu nhưng với số lượng dãy con ít nhất giảm đi một nửa. Lặp lại qui trình trên sau một số bước, ta sẽ nhận được 1 dãy chỉ gồm 1 dãy con không giảm. Nghĩa là dãy ban đầu đã được sắp xếp.

Thuật giải trộn trực tiếp là phương pháp trộn đơn giản nhất. Việc phân hoạch thành các dãy con đơn giản chỉ là tách dãy gồm N phần tử thành N dãy con. Đòi hỏi của Thuật giải

về tính có thứ tự của các dãy con luôn được thỏa trong cách phân hoạch này vì dãy gồm một phân tử luôn có thứ tự. Cứ mỗi lần tách rồi trộn, chiều dài của các dãy con sẽ được nhân đôi.

b. Thuật giải

- Bước 1: // Chuẩn bị

  k = 1; // k là chiều dài của dãy con trong bước hiện hành

- Bước 2: Tách dãy $a_0, a_1, ., a_{N-1}$ thành 2 dãy b, c theo nguyên tắc luân phiên từng nhóm k phần tử:

  $b = a_0, .., a_{k-1}, a_{2k}, .., a_{3k-1}, \ldots$

  $c = a_k, .., a_{2k-1}, a_{3k}, .., a_{4k-1}, ...$

- Bước 3: Trộn từng cặp dãy con gồm k phần tử của 2 dãy b, c vào a.

- Bước 4:

  k = k*2;

  Nếu k < n thì trở lại bước 2.

  Ngược lại: Dừng

c. Cài đặt

```
void MergeSort(int a[], int N)
{
    int  p, pb, pc; //các chỉ số trên các mảng a, b, c
    int  i, k = 1;   //độ dài của dãy con khi phân hoạch
    do // tách a thành b và c
    {
        p = pb = pc = 0;
        while(p < N)
        {
            for(i = 0; (p < n)&&(i < k); i++)
                b[pb++] = a[p++];
            for(i = 0; (p < n)&&(i < k); i++)
           c[pc++] = a[p++];
    }
    Merge(a, pb, pc, k); //trộn b và c thành a
    k *= 2;
  }while(k < N);
}
```

Trong đó hàm Merge có thể cài đặt như sau:

```
void Merge(int a[], int nb, int nc, int k)
{
    int p, pb, pc, ib, ic, kb, kc;
    p = pb = pc = 0; ib = ic = 0;
    while((0 < nb)&&(0 < nc))
    {
        kb = min(k, nb); kc = min(k, nc);
        if(b[pb+ib] <= c[pc+ic])
        {
            a[p++] = b[pb+ib]; ib++;
            if(ib == kb)
            {
                for(; ic<kc; ic++)
                    a[p++] = c[pc+ic];
                pb += kb; pc += kc; ib = ic = 0;
                nb -= kb; nc -= kc;
            }
        }
        else    //(ib != kb)
        {
            a[p++] = c[pc+ic]; ic++;
            if(ic == kc)
            {
                for(; ib<kb; ib++)
                    a[p++] = b[pb+ib];
                pb += kb;
                pc += kc; ib = ic = 0;
                nb -= kb; nc -= kc;
            }
        }
    }
}
```

d. Đánh giá Thuật giải

Ta thấy rằng số lần lặp của bước 2 và bước 3 trong thuật giải MergeSort bằng $\log_2 n$ do sau mỗi lần lặp giá trị của k tăng lên gấp đôi. Dễ thấy, chi phí thực hiện bước 2 và bước 3 tỉ lệ thuận bởi N. Như vậy, chi phí thực hiện của thuật giải MergeSort sẽ là $O(N\log_2 N)$. Do không sử dụng thông tin nào về đặc tính của dãy cần sắp xếp, nên trong mọi trường hợp của Thuật giải chi phí là không đổi.

Nhược điểm lớn của Thuật giải là không tận dụng những thông tin đặc tính của dãy cần sắp xếp. Ví dụ trong trường hợp dãy đã có thứ tự sẵn.

Thực tế người ta ít chọn phương pháp trộn trực tiếp mà cải tiến nó gọi là phương pháp trộn tự nhiên.

Thuật giải trộn tự nhiên:

Một đường chạy của dãy số a là một dãy con không giảm của cực đại của a. Nghĩa là, đường chạy r = (a_i, a_{i+1}, .., a_j) phải thỏa điều kiện:

$$\begin{cases} a_k \le a_{k+1} & \forall k \in [i, j) \\ a_i < a_{i-1} & \\ a_j > a_{j+1} & \end{cases}$$

Ví dụ dãy 12, 2, 8, 5, 1, 6, 4, 15 có thể coi như gồm 5 đường chạy: (12), (2, 8), (5), (1,6), (4,15).

Thuật giải trộn tự nhiên khác thuật giải trộn trực tiếp ở chỗ thay vì luôn cứng nhắc phân hoạch theo dãy con có chiều dài k, việc phân hoạch sẽ theo đơn vị là đường chạy. Ta chỉ cần biết số đường chạy của a sau lần phân hoạch cuối cùng là có thể biết thời điểm dừng của thuật giải vì dãy đã có thứ tự là dãy chỉ có một đường chạy.

Mô tả Thuật giải:

Bước 1: //chuẩn bị

r=0; //r dùng để đếm số đường chạy

Bước 2: tách dãy a1, a2,.., an thành 2 dãy b, c theo nguyên tắc luân phiên từng đường chạy:

Bước 2.1:

Phân phối cho b một đường chạy; r = r + 1;

Nếu a còn phần tử chưa phân phối

Phân phối cho c một đường chạy; r = r+1;

Bước 2.2: nếu a còn phần tử: quay lại bước 2.1

Bước 3: trộn từng cặp đường chạy của 2 dãy b, c vào a

Bước 4:

Nếu r<=2 thì trở lại bước 2

Ngược lại: dừng

Một nhược điểm lớn nữa của thuật giải trộn là khi cài đặt thuật giải đòi hỏi thêm không gian bộ nhớ để lưu trữ các dãy phụ b, c. Hạn chế này khó chấp nhận trong thực tế vì các dãy cần sắp xếp có kích thước lớn. Vì vậy thuật giải trộn thường được dùng để sắp xếp các cấu trúc dữ liệu khác phù hợp hơn như danh sách liên kết hoặc file.

2.2.8 Phương pháp sắp xếp theo cơ số (Radix Sort)

a. Ý tưởng

Radix Sort là một thuật giải tiếp cận theo một hướng hoàn toàn khác. Nếu như trong các thuật giải khác, cơ sở để sắp xếp luôn là việc so sánh giá trị của 2 phần tử thì Radix Sort lại dựa trên nguyên tắc phân loại thư của bưu điện. Vì lý do đó nó còn có tên khác là Postman's sort

Nó không hề quan tâm đến việc so sánh giá trị của phần tử và bản thân việc phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.

Để chuyển một khối lượng thư lớn đến tay người nhận ở nhiều địa phương khác nhau, bưu điện thường tổ chức một hệ thống phân loại thư phân cấp. Trước tiên, các thư đến cùng một tỉnh, thành phố sẽ được sắp chung vào một lô để gửi đến tỉnh, thành phố tương ứng. Bưu điện các tỉnh thành này lại thực hiện công việc tượng tự. Các thư đến cùng một quận, huyện sẽ được xếp vào chung một lô và gởi đến quận, huyện tương ứng.

Cứ như vậy, các bức thư sẽ được trao đến tay người nhận một cách có hệ thống mà công việc sắp xếp thư không quá năng nhọc.

Mô phỏng lại qui trình trên, để sắp xếp dãy a_0, a_1,.., a_{N-1}, thuật giải Radix Sort thực hiện như sau:

- Trước tiên, ta có thể giả sử mỗi phần tử a_i trong dãy a_0, a_1,.., a_{N-1} 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,.. tương tự việc phân loại thư theo tỉnh, thành, huyện, phường xã,…

b. Mô tả thuật giải:

Bước 1://k cho biết chữ số dùng để phân loại hiện hành

k = 0; //k = 0:hàng đơn vị; k = 1: hàng chục;…

Bước 2:// tạo các lô chứa các loại phần tử khác nhau

Khởi tạo 10 lô B_0, B_1, …, B_9 rỗng.

Bước 3:

For(i=0; i < N; i++)

Đặt a_i vào lô B_t với t = chữ số thứ k của a_i.

Bước 4: Nối B_0, B_1, …, B_9 lại theo đúng trình tự thành a.

Bước 5:

k = k+1;

Nếu k<m trở lại bước 2

Ngược lại: dừng.

Ví dụ 2.7:

i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	7013	8425	1239	0428	1424	7009	4518	3252	9170	0999	1725	0701

Lần 1:

12	0701										
11	1725										
10	0999										
9	9170										
8	3252										
7	4518										
6	7009										
5	1424										
4	0428										
3	1239										0999
2	8425						1725			4518	7009
1	7013	9170	0701	3252	7013	1424	8425			0428	1239
CS	A	0	1	2	3	4	5	6	7	8	9

Lần 2:

12	0999										
11	7009										
10	1239										
9	4518										
8	0428										
7	1725										
6	8425										
5	1424										
4	7013			0428							
3	3252			1725							
2	0701	7009	4518	8425							
1	9170	0701	7013	1424	1239		3252		9170		0999
CS	A	0	1	2	3	4	5	6	7	8	9

Lần 3:

12	0999										
11	9170										
10	3252										
9	1239										
8	0428										
7	1725										
6	8425										
5	1424										
4	4518										
3	7013					0428					
2	7009	7013		3252		8425			1725		
1	0701	7009	9170	1239		1424	4518		0701		0999
CS	A	0	1	2	3	4	5	6	7	8	9

Lần 4:

12	0999										
11	1725										
10	0701										
9	4518										
8	0428										
7	8425										
6	1424										
5	3252										
4	1239										
3	9170	0999	1725								
2	7013	0701	1424						7013		
1	7009	0428	1239		3252	4518			7009	8425	9170
CS	A	0	1	2	3	4	5	6	7	8	9

Kết quả:

12	9170										
11	8425										
10	7013										
9	7009										
8	4518										
7	3252										
6	1725										
5	1424										
4	1239										
3	0999										
2	0701										
1	0428										
CS	A	0	1	2	3	4	5	6	7	8	9

c. Đánh giá thuật giải

Với dãy N số, mỗi con số có tối đa m chữ số, thuật giải thực hiện m lần các thao tác phân lô và ghép lô.

Trong thao tác phân lô, mỗi phần tử chỉ được xét đúng một lần, khi ghép cũng vậy. Như vậy, chi phí cho việc thực hiện thuật giải hiển nhiên là O(2mn) = O(n).

Nhận xét

- Sau lần phân phối thứ k các phần tử của A vào các lô B_0, B_1, …, B_9, và lấy ngược trở ra, nếu chỉ xét đến k+1 chữ số của các phần tử trong A, ta sẽ có một mảng tăng dần nhờ trình tự lấy ra từ 0 →9. Nhận xét này đảm bảo tính đúng đắn của thuật giải.

- Thuật giải có độ phức tạp tuyến tính nên rất hiệu quả khi sắp dãy có nhiều phần tử, nhất là khi khóa sắp xếp không quá dài so với số lượng phần tử (điều này thường gặp trong thực tế).

- Số lượng lô lớn (10 khi dùng số thập phân, 26 khi dùng chuỗi kí tự tiếng Anh…) nhưng tổng kích thước của tất cả các lô chỉ bằng dãy ban đầu nên ta không thể dùng mảng để biểu diễn B. Như vậy phải dùng cấu trúc dữ liệu động để biểu diễn B → Radix Sort rất thích hợp cho sắp xếp trên danh sách liên kết.

- Người ta dùng phương pháp phân lô theo biểu diễn nhị phân của khóa sắp xếp. Khi đó ta có thể dùng hoàn toàn cấu trúc dữ liệu mảng để biểu diễn B vì chỉ cần dùng hai lô B_0 và B_1. Tuy nhiên, khi đó chiều dài khóa sẽ lớn. Khi sắp xếp dãy không nhiều phần tử, thuật giải Radix Sort sẽ mất ưu thế so với các thuật giải khác.

Bài tập

A. Các thuật giải tìm kiếm trong

Bài 1:

Tìm kiếm số nguyên x trên dãy n số nguyên a[0..n-1]:

- Nếu không có, trả về -1;
- Nếu có, trả về các trường hợp sau:
  - Chỉ số đầu tiên
  - Chỉ số cuối cùng
  - Các chỉ số tại các phần tử trong dãy trùng x.

Sử dụng thuật giải tìm kiếm tuyến tính.

Ta sẽ tạo một Project gồm 2 tập tin:

- Tập tin thư viện *h: Chứa các hàm chức năng của chương trình
- Tập tin chương trình *cpp: Chứa hàm main(), các hàm tổ chức menu, nhập xuất dữ liệu.

Bài 2:

Giả sử có một danh sách sinh viên, mỗi một sinh viên được lưu trữ các thông tin:

- Mã số,
- họ tên,
- lớp,
- điểm trung bình,
- tổng số tín chỉ đã tích lũy được.

Thực hiện các thao tác tìm kiếm trên danh sách sinh viên:

- Tìm kiếm theo mã số
- Tìm kiếm theo họ tên: Xuất tất cả các sinh viên nếu họ tên trùng với họ tên cho trước.
- Tìm kiếm theo điểm trung bình: Xuất tất cả sinh viên có điểm >= x.
- Tìm kiếm theo lớp: Xuất sinh viên thuộc lớp cho trước.

Ta sẽ định nghĩa kiểu cấu trúc chứa các thông tin sinh viên tên là SINHVIEN. Danh sách sinh viên sẽ được tổ chức bằng mảng 1 chiều các cấu trúc kiểu SINHVIEN. Việc thực hiện nhập, xuất, tìm kiếm sẽ thực hiện trên mảng các cấu trúc kiểu SINHVIEN

Ta sẽ tạo một Project gồm 2 tập tin:

- Tập tin thư viện *h: Chứa các hàm chức năng của chương trình
- Tập tin chương trình *cpp: Chứa hàm main(), các hàm tổ chức menu, nhập xuất dữ liệu.

Bài 3:

Giả sử có một danh sách nhân viên, mỗi một nhân viên được lưu trữ các thông tin:

- Mã nhân viên,
- Họ
- Chữ lót
- Tên
- Năm sinh
- Lương

Thực hiện thao tác xuất tất các các nhân viên:

- Có Tên cho trước.
- Có tiền lương cho trước

- Có năm sinh cho trước

B. Các thuật giải sắp xếp trong

Bài 1:

Cho mảng a gồm n phần tử: a[0], a[1], …, a[n-1]. Sắp xếp mảng a tăng dần bằng các thuật giải sắp xếp trong:

1. Phương pháp chọn trực tiếp.
2. Phương pháp heap sort
3. Phương pháp chèn trực tiếp.
4. Phương pháp đổi chỗ trực tiếp
5. Phương pháp nổi bọt.
6. Phương pháp sắp xếp dựa trên phân hoạch (Quich sort).
7. Phương pháp sắp xếp trộn trực tiếp. (Merge sort)
8. Phương pháp sắp xếp theo cơ số. (Radix sort).

Yêu cầu:

Tạo một Project gồm 2 tập tin:

- *Tập tin thư viện *.h: Chứa các hàm chức năng của chương trình*
- *Tập tin chương trình *.cpp: Chứa hàm main(), các hàm tổ chức menu, nhập xuất dữ liệu.*

Bài 2:

Giả sử có một danh sách nhân viên, mỗi một nhân viên được lưu trữ các thông tin:

- Mã nhân viên,
- Họ
- Chữ lót
- Tên
- Năm sinh
- Lương

Sắp tăng dần danh sách theo khóa:

- Mã nhân viên.
- Tên nhân viên
- Năm sinh
- Lương

(lần lượt sử dụng các thuật giải sắp xếp trong)

Bài 3:

Thuật giải chọn trực tiếp: Sắp tăng dãy a[0..N-1]

Mô tả:

∀i=0,..., N-2:

- Chọn $a_k = Max(a_0,...,a_{N-1-i})$
- Hoán vị a_{N-1-i}, a_k cho nhau.

Cài đặt thuật giải

Bài 4:

- Thuật giải chọn trực tiếp: Sắp tăng dãy a[0..N-1]
- Mô tả:
  ∀i=0,..., N/2:

- Chọn:
  - ✓ a_k = Max(a_i,...,a_{N-1-i})
  - ✓ a_j = Min(a_i,...,a_{N-1-i})
- Với điều kiện thích hợp (?):
  - ✓ Hoán vị a_i, a_j cho nhau.
  - ✓ Hoán vị a_k, a_{N-1-i} cho nhau.

Cài đặt thuật giải.

Bài 5:

Thuật giải chèn nhị phân:

Khi sử dụng thuật giải chèn trực tiếp, ta dùng phương pháp tìm kiếm nhị phân để xác định vị trí cần chèn của a_i trong dãy con đã có thứ tự a_1,…, a_{i-1}.

Cài đặt thuật giải.

Chương 3. Danh sách liên kết

3.1. Giới thiệu đối tượng dữ liệu con trỏ

3.1.1 Cấu trúc dữ liệu tĩnh và cấu trúc dữ liệu động

Với kiểu dữ liệu tĩnh, đối tượng dữ liệu được định nghĩa đệ quy, và tổng kích thước vùng nhớ dành cho tất cả các biến dữ liệu tĩnh chỉ là 64Kb (1 segment bộ nhớ). Vì lý do đó, khi có nhu cầu dùng nhiều bộ nhớ hơn ta phải sử dụng các cấu trúc dữ liệu động.

Nhằm đáp ứng nhu cầu thể hiện sát thực bản chất của dữ liệu cũng như xây dựng các thao tác hiệu quả trên dữ liệu, ta cần phải tìm cách tổ chức kết hợp dữ liệu với những hình thức linh động hơn, có thể thay đổi kích thước, cấu trúc trong suốt thời gian sống. Các hình thức tổ chức dữ liệu như vậy được gọi là cấu trúc dữ liệu động. Cấu trúc dữ liệu động cơ bản nhất là danh sách liên kết.

3.1.2 Kiểu con trỏ

1. Biến không động

Biến không động (biến tĩnh) là những biến thỏa các tính chất sau:

- Được khai báo tường minh.
- Tồn tại khi vào phạm vi khai báo và chỉ mất khi ra khỏi phạm vi này.
- Được cấp phát vùng nhớ trong vùng dữ liệu (Data segment) hoặc là Stack (đối với các biến nửa tĩnh, các biến cục bộ).
- Kích thước không thay đổi trong suốt quá trình sống.

2. Kiểu dữ liệu con trỏ

Khi nói đến kiểu dữ liệu T, ta thường chú ý đến hai đặc trưng quan trọng và liên hệ mật thiết với nhau:

- Tập V các giá trị thuộc kiểu: đó là tập các giá trị hợp lệ mà đối tượng kiểu T có thể nhận được và lưu trữ.
- Tập O các phép giải (hay thao tác xử lý) xác định có thể thực hiện trên các đối tượng dữ liệu kiểu đó.

Kí hiệu: T = <V, O>

3. Định nghĩa kiểu dữ liệu con trỏ

Cho trước kiểu T=<V,O>

Kiểu con trỏ - kí hiệu "Tp" – chỉ đến các phần tử có kiểu "T" được định nghĩa: Tp=<Vp, Op>, trong đó:

- Vp={{các địa chỉ có thể lưu trữ những đối tượng có kiểu T}, NULL} (với NULL là một giá trị đặc biệt tượng trưng cho một giá trị không biết hoặc không quan tâm).
- Op = {các thao tác định địa chỉ của một đối tượng thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó} (thường gồm các thao tác tạo một con trỏ chỉ đến một đối tượng thuộc kiểu T; hủy một đối tượng thuộc kiểu T khi biết con trỏ chỉ đến đối tượng đó)

Kiểu con trỏ là kiểu cơ sở dùng lưu địa chỉ của một đối tượng dữ liệu khác.

Biến thuộc kiểu con trỏ Tp là biến mà giá trị của nó là địa chỉ của một vùng nhớ ứng với một biến kiểu T, hoặc là giá trị NULL.

Kích thước của biến con trỏ tùy thuộc vào quy ước số byte địa chỉ trong từng mô hình bộ nhớ của từng ngôn ngữ lập trình cụ thể. Chẳng hạn biến con trỏ trong C++ trên môi trường Windows có kích thước 4 bytes.

Cú pháp định nghĩa kiểu con trỏ trong ngôn ngữ C, C++:

typedef <kiểu cơ sở> *<kiểu con trỏ>;

Ví dụ 3.1:

```
typedef  int *intpointer; //Kiểu con trỏ
intpointer  p; //Biến con trỏ
```

Cú pháp định nghĩa trực tiếp một biến con trỏ trong ngôn ngữ C, C++:

<kiểu cơ sở> *<tên biến>;

Ví dụ 3.2:

```
int *p;
```

Các thao tác cơ bản trên kiểu con trỏ (minh họa bằng C++)

Khi một biến con trỏ p lưu địa chỉ của đối tượng x, ta nói "p trỏ đến x".

- Gán địa chỉ của một vùng nhớ con trỏ p:

  p = <địa chỉ>;

  p = <địa chỉ>+<giá trị nguyên>;
- Truy xuất nội dung của đối tượng do p trỏ đến: *p

4. Biến động

Trong nhiều trường hợp, tại thời điểm biên dịch không thể xác định trước kích thước chính xác của một số đối tượng dữ liệu do sự tồn tại và tăng trưởng của chúng phụ thuộc vào ngữ cảnh của việc thực hiện chương trình.

Các đối tượng có đặc điểm như vậy được khai báo như biến động.

Đặc trưng của biến động

- Biến không được khai báo tường minh.
- Có thể được cấp phát hoặc giải phóng bộ nhớ khi người sử dụng yêu cầu.
- Các biến này không theo qui tắc phạm vi (tĩnh).
- Vùng nhớ của biến được cấp phát trong Heap.
- Kích thước có thể thay đổi trong quá trình sống.

Do không được khai báo tường minh nên các biến động không có một định danh được kết buộc với địa chỉ vùng nhớ cấp phát cho nó, do đó gặp khó khăn khi truy xuất đến một biến động.

Để giải quyết vấn đề này, phải dùng một con trỏ (là biến không động) để trỏ đến biến động. Khi tạo ra một biến động, phải dùng một con trỏ để lưu địa chỉ của biến này thông qua biến con trỏ đã biết định danh.

Hai thao tác cơ bản trên biến động là tạo và hủy một biến động do biến con trỏ "p" trỏ đến:

- Tạo ra một biến động và cho con trỏ "p" chỉ đến nó (minh họa bằng C++: dùng hàm new để cấp phát bộ nhớ):

```
p = new KieuCoSo;
```

- Hủy vùng nhớ cấp phát bởi hàm new do p trỏ tới (dùng hàm delete)

```
delete(p);
```

Cấp phát bộ nhớ cho mảng động

```
KieuCoSo *p;
p = new KieuCoSo[Max];/Max là già trị nguyên dương
```

Thu hồi vùng nhớ của mảng động

```
delete([]p);
```

3.2. Danh sách liên kết

4.2.1 Định nghĩa

Cho T là một kiểu được định nghĩa trước, kiểu danh sách Tx gồm các phần tử thuộc kiểu T được định nghĩa là:

Tx = <Vx, Ox>

Trong đó:

- Vx = {tập hợp có thứ tự các phần tử kiểu T được mócnối với nhau theo trình tự tuyến tính};
- Ox = {các thao tác trên danh sách liên kết như: tạo danh sách; tìm một phần tử trong danh sách; chèn một phần tử vào danh sách; hủy một phần tử khỏi danh sách; sắp xếp danh sách…}

Ví dụ 3.3:

Hồ sơ các học sinh cụa một trường được tổ chức thành danh sách gồm nhiều hồ sơ của từng học sinh, số lượng học sinh trong trường có thể thay đổi do vậy cần có các thao tác thêm, hủy một hồ sơ. Để phục vụ cho công tác giáo vụ cần thực hiện các thao tác tìm hồ sơ của một học sinh, in danh sách hồ sơ…

3.2.2 Tổ chức danh sách liên kết

Mối liên hệ giữa các phần tử được thể hiện ngầm

- Mỗi phần tử trong danh sách được đặc trưng bởi một chỉ số, cặp phần tử x_i, x_{i+1} được xác định là kế cận trong danh sách nhờ vào quan hệ của cặp chỉ số I và i+1. Các phần tử trong danh sách buộc phải lưu trữ liên tiếp trong bộ nhớ để có thể xây dựng công thức xác định địa chỉ của phần tử thứ i: address(i) = address(1) + (i-1)*sizeof(T).
- Có thể xem mảng và tập tin là những danh sách đặc biệt được tổ chức theo hình thức liên kết "ngầm" giữa các phần tử.
- Với cách tổ chức này có ưu điểm là cho phép truy xuất ngẫu nhiên, đơn giản và nhanh chóng đến một phần tử bất kỳ trong danh sách. Tuy nhiên hạn chế của nó là hạn chề về mặt sử dụng bộ nhớ.
- Đối với mảng, số phần tử được xác định trong thời gian biên địch và cần cấp phát bộ nhớ liên tục. Trong trường hợp kích thước bộ nhớ trống còn đủ để chứa toàn bộ mảng nhưng các ô nhớ trống lại không nằm kế cận nhau thì cũng không cấp phát vùng nhớ cho mảng được. Ngoài ra do kích thước mảng cố định mà số phần tử của

danh sách lại khó dự trù chính xác nên có thể gây ra tình trạng thiếu hụt hay lãng phí bộ nhớ. Các thao tác thêm hủy các phân tử thục hiện không tự nhiên.

Mối liên hệ giữa các phần tử được thể hiện tường minh

- Mỗi phần tử ngoài thông tin về bản thân còn chứa một liên kết (địa chỉ) đến phần tử kế trong danh sách nên còn được gọi là danh sách móc nối (liên kết).
- Do liên kết tường minh nên các phần tử trong danh sách không cần phải lưu trữ kế cận trong bộ nhớ. Nhờ đó khắc phục được các khuyết điểm của hình thức tổ chức mảng. Tuy nhiên việc truy xuất đòi hỏi phải thực hiện truy xuất thông qua một số phần tử khác.
- Có nhiều kiểu tổ chức liên kết giữa các phần tử trong danh sách như: danh sách liên kết đơn, danh sách liên kết kép, vòng,…

3.2.3 Danh sách liên kết đơn

Danh sách liên kết đơn tổ chức theo cách cấp phát liên kết. Các thao tác cơ bản trên danh sách đơn gồm có tạo phần tử, chèn phần tử vào danh sách, hủy một phần tử trong danh sách, tìm kiếm phần tử trong danh sách và sắp xếp danh sách.

4.2.4 Tổ chức danh sách theo cách cấp phát liên kết.

Một danh sách liên kết bao gồm tập các phần tử (nút), mỗi nút là một cấu trúc chứa hai thông tin:

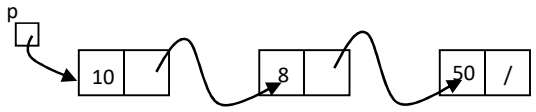
- Thành phần dữ liệu: lưu trữ các thông tin về bản thân phần tử.
- Thành phần mối liên kết: lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử cuối danh sách.

Mỗi nút như trên có thể được cài đặt như sau:

```
struct  tagNode
{
        Data        Info;        //thành phần dữ liệu
        tagNode*  pNext;     //thành phần mối liên kết (tự trỏ)
};
//Đổi lại tên kiểu phần tử trong danh sách
typedef  tagNode  NODE;
```

Mỗi phần tử trong danh sách đơn là một biến động, sẽ được yêu cầu cấp phát bộ nhớ khi cần.

Danh sách liên kết đơn chính là sự liên kết các biến động này với nhau, do vậy đạt được sự linh động khi thay đổi số lượng các phần tử.



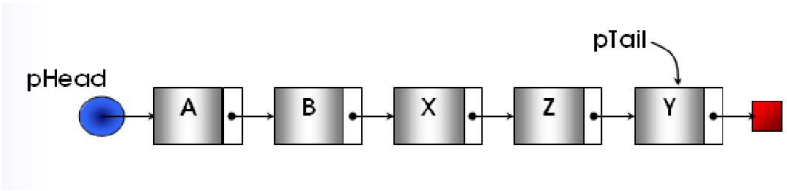
Để quản lý một danh sách liên kết đơn chỉ cần biết địa chỉ phần tử đầu danh sách, từ phần tử đầu danh sách ta có thể đi đến các nút tiếp theo trong danh sách liên kết nhờ vào thành phần địa chỉ của nút.

Con trỏ pHead được dùng để lưu trữ địa chỉ của phần tử ở đầu danh sách, ta gọi Head là đầu của danh sách. Ta có khai báo:

```
NODE*       pHead;
```

Để tiện lợi, ta có thể sử dụng thêm một con trỏ pTail để giữ địa chỉ phần tử cuối danh sách. Khai báo pTail như sau:

```
NODE*       pTail;
```



4.2.5 Định nghĩa cấu trúc danh sách liên kết

Giả sử ta có định nghĩa nút trong danh sách như sau:

```
struct  tagNode
{
    Data        Info;
    tagNode*    pNext;
};
typedef tagNode NODE;
```

Định nghĩa kiểu danh sách liên kết: LIST

```
struct  LIST
{
```

```
    NODE* pHead;      //con trỏ lưu trữ địa chỉ đầu DSLK
    NODE* pTail;      //con trỏ lưu trữ địa chỉ cuối DSLK
};
```

Ví dụ 3.4:

(định nghĩa danh sách đơn lưu trữ hồ sơ sinh viên)

```
//Kiểu thành phần dữ liệu Data: SV
struct  SV //Data
{   char    Ten[30];
    int     MaSV;
};
//Kiểu phần tử trong DS: SinhvienNode
struct  SinhvienNode
{   SV      Info;
    SinhvienNode* pNext;
};
//Đổi lại tên kiểu phần tử trong DS: SVNode
typedef   SinhvienNode   SVNode;
//Định nghĩa danh sách liên kết: LIST
struct  LIST
{
     SVNode * pHead;
     SVNode * pTail;
};
```

3.2.6 Các thao tác cơ bản trên danh sách liên kết đơn

- Tạo một phần tử cho danh sách liên kết với thông tin x.
- Khởi tạo danh sách rỗng.
- Kiểm tra danh sách rỗng.
- Duyệt danh sách.
- Tìm một phần tử trong danh sách.
- Chèn một phần tử vào danh sách
- Hủy một phần tử khỏi danh sách
- Sắp xếp danh sách

Giả sử với định nghĩa danh sách liên kết như trên, ta có cài đặt các thao tác như sau:

1. Tạo phần tử

Hàm CreateNode(x): tạo một nút trong DSLK vời thành phần dữ liệu x, hàm trả về con trỏ lưu trữ địa chỉ của phần tử vừa tạo (nếu thành công).

```
NODE*  GetNode(Data x)
{
    NODE *p;
    // cấp phát vùng nhớ cho phần tử
    p = new NODE;
    if ( p==NULL)
    {
            cout<<“Loi cap phat";
            return NULL;  //exit(1);
    }
    p ->Info = x; // gán thông tin cho phần tử p
    p->pNext = NULL;
    return p;
}
```

Sử dụng hàm CreateNode: gán giá trị của hàm (là con trỏ) cho 1 biến con trỏ kiểu NODE, phần tử này đặt tên là new-ele, giữ địa chỉ của phần tử đã tạo:

```
    NODE *new_ele = CreateNode(x);
```

2. Khởi tạo danh sách rỗng l

```
void CreatList(LIST &l)
{
    l.pHead = l.pTail = NULL;
}
```

3. Kiểm tra danh sách rỗng

Hàm IsEmpty(l) = 1 nếu danh sách rỗng, bằng 0 nếu danh sách l không rỗng

```
int  IsEmpty(LIST l)
{
        if (l.pHead == NULL) // DS rỗng
                return 1;
```

```
        return 0;
}
```

4. Duyệt danh sách

Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách như:

- Đếm các phần tử của danh sách.
- Xuất dữ liệu các phần tử trong DS ra màn hình
- Tìm tất cả các phần tử thỏa điều kiện
- Hủy toàn bộ danh sách (và giải phóng bộ nhớ)

Thuật giải có thể mô tả như sau:

Bước 1: p = Head; *//Cho p trỏ đến phần tử đầu danh sách*

Bước 2: Trong khi (Danh sách chưa hết) thực hiện

B2.1: Xử lý phần tử p;

B2.2: p:=p->pNext; *// Cho p trỏ tới phần từ kế tiếp*

Cài đặt:

```
void ProcessList (LIST l)
{
    NODE        *p;
    p = l.pHead;
    while (p!= NULL)
    {
        ProcessNode(p); // xử lý cụ thể từng ứng dụng
            p = p->pNext;
    }
}
```

Việc xuất dữ liệu ra màn hình cũng chính là hàm duyệt danh sách, trong đó với mỗi nút hàm duyệt qua, ta xuất thông tin của nút đó ra màn hình (tức là thủ tục *ProcessNode(p)* chính là thủ tục xuất thông tin nút p ra màn hình).

```
void XuatDS(LIST l)
{
    NODE  *p;
```

```
if(IsEmpty(l))
{       cout<<"\nDS rong!\n";  return;   }
        cout<<"\nDu lieu cua Danh sach:\n";
        p = l.pHead;
        while (p!= NULL)
        {
                cout<<(p->Info)<<'\t';  // ProcessNode(p)
                p = p->pNext;
        }
    }
}
```

5. Tìm phần tử có thành phần dữ liệu x trong danh sách

Mô tả

Bước 1: p = Head; //Cho p trỏ đến phần tử đầu danh sách

Bước 2: Trong khi (p != NULL) và (p->Info != x) thực hiện:

p = p->Next;// Cho p trỏ tới phần tử kế tiếp

Bước 3:

Nếu p != NULL thì p trỏ tới phần tử cần tìm

Ngược lại: không có phần tử cần tìm.

Cài đặt:

```
/*
Search(l,x) = p;  //p là con trỏ chỉ đến phần tử có chứa thông tin x cần tìm
            = NULL; //ngược lại
*/
NODE *Search(LIST l, Data  x)
{
        NODE        *p;
        p = l.pHead;
        while((p!= NULL)&&(p->Info != x))
                p = p->pNext;
        return p;
}
```

6. Thao tác chèn một phần tử vào danh sách

Có ba vị trí để có thể chèn một phần tử new_ele vào danh sách:

- Chèn phần tử vào đầu danh sách
- Chèn phần tử vào cuối danh sách
- Chèn phần tử vào danh sách sau một phần tử q.

a. Chèn phần tử vào đầu danh sách

Mô tả

Kiểm tra nếu danh sách rỗng //chèn nút vào danh sách rỗng

Head = new_ele;

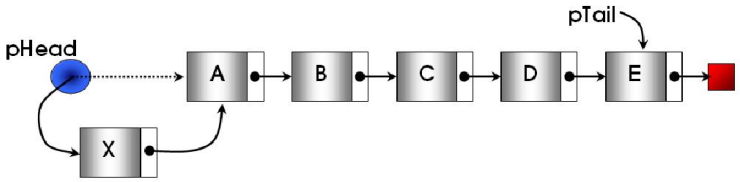
Tail = Head;

Ngược lại

New_ele ->pNext = Head;

Head = new_ele;

Minh họa



Cài đặt

```
/*AddFirst(l,new_ele): chèn một phần tử new_ele vào đầu danh sách*/
void AddFirst(LIST &l, NODE* new_ele)
{
    if (l.pHead==NULL) //danh sách rỗng:
        if(IsEmpty(l))
    {
        l.pHead = new_ele;
        l.pTail = l.pHead;
    }
    else
    {
        Newele->pNext = l.pHead;
        l.pHead = new_ele;
    }
}
```

Tạo nút có dữ liệu x, sau đó chèn nút vừa tạo vào đầu danh sách, hàm trả về con trỏ lưu vị trí nút vừa tạo.

```
NODE* InsertHead(LIST &l, Data x)
{
    NODE* new_ele = CreateNode(x);
    if (new_ele ==NULL) return NULL;
    if (l.pHead==NULL)
    {       l.pHead = new_ele; l.pTail = l.pHead;
    }
    else
    {       new_ele->pNext = l.pHead;
            l.pHead = new_ele;
    }
    return new_ele;
}
```

Hoặc

```
NODE* InsertHead(LIST &l, Data x)
{
    NODE* new_ele = CreateNode(x);
    if (new_ele ==NULL) return NULL;
    AddHead(l, new_ele);
    return new_ele;
}
```

b. Chèn một phần tử vào cuối danh sách

Mô tả:

Nếu danh sách rỗng thì

Head = new_ele;

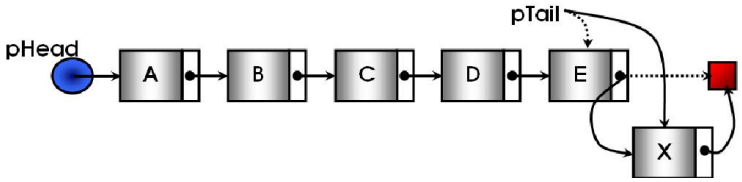
Tail = Head;

Ngược lại

Tail ->Next = new_ele;

Tail = new_ele ;

Minh họa



Cài đặt

```
void AddTail(LIST &l, NODE *new_ele)
{
        if (l.pHead==NULL)
        {
                l.pHead = new_ele; l.pTail = l.pHead;
        }
        else
        {
                l.pTail->Next = new_ele;
                l.pTail = new_ele;
        }
}
```

Tạo nút có dữ liệu x, sau đó chèn nút vừa tạo vào cuối danh sách, hàm trả về con trỏ lưu vị trí nút vừa tạo

```
NODE* InsertTail(LIST &l, Data x)
    {
        NODE* new_ele = GetNode(x);
        if (new_ele ==NULL)
                return NULL;
        if (l.pHead==NULL)
        {       l.pHead = new_ele; l.pTail = l.pHead;
        }
        else
        {       l.pTail->Next = new_ele;
                l.pTail = new_ele;
        }
```

```
        return new_ele;
    }
```

Hoặc

```
NODE* InsertTail(LIST &l, Data x)
    {
        NODE* new_ele = CreateNode(x);
        if (new_ele ==NULL)
                return NULL;
        AddTail(l,new_ele);
        return new_ele;
    }
```

c. Chèn một phần tử x vào danh sách sau một phần tử q

Mô tả

Nếu (q != NULL) thì

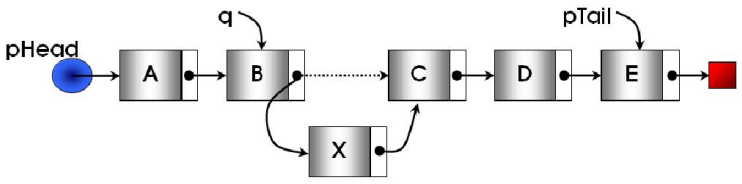
```
        new_ele -> pNext = q->pNext;
        q->pNext = new_ele;
```

Nếu q = l.Tail thì l.Tail = new_ele

Ngược lại

Chèn phần tử x vào đầu danh sách

Minh họa



Cài đặt

```
void AddAfter(LIST &l,NODE *q, NODE* new_ele)
{
    if ( q!=NULL)
    { new_ele->pNext  =  q->pNext;
      q->pNext = new_ele;
      if(q == l.pTail)
       l.pTail = new_ele;
```

```
    }
    else
            AddFirst(l, new_ele);
}
```

Tạo nút có dữ liệu x, sau đó chèn nút vừa tạo sau nút q của danh sách, hàm trả về con trỏ lưu vị trí nút vừa tạo

```
NODE* InsertAfter(LIST &l,NODE *q, Data x)
{
    NODE* new_ele = CreateNode(x);
    if (new_ele ==NULL) reuturn NULL;
    AddAfter(l, q, new_ele);
}
```

Đến đây, ta có thể cài đặt thủ tục nhập danh sách liên kết như sau:

```
//Nhap Du lieu cho DSLKD: chèn cuối
void NhapDS(LIST &l)
{
    Data x;
    NODE *new_ele;
    CreatList(l);
    for(;;)
    {       cout<<"\nNhap x: ";  cin>>x;
            if(x == Thoat)
            {
                    cout<<"\nDa nhap xong du lieu\n";   break;
            }
            new_ele = new NODE;
            new_ele = CreateNode(x);
            AddTail(l, new_ele); //AddFirst(l,new_ele)
    }
}
```

7. Hủy một phần tử khỏi danh sách liên kết

Có ba thao tác thông dụng khi hủy một phần tử ra khỏi danh sách:

- Hủy phần tử đầu danh sách
- Hủy một phần tử đứng sau phần tử q trong danh sách
- Hủy một phần tử có dữ liệu x.

a. Hủy phần tử đầu danh sách

Mô tả

Nếu danh sách khác rỗng

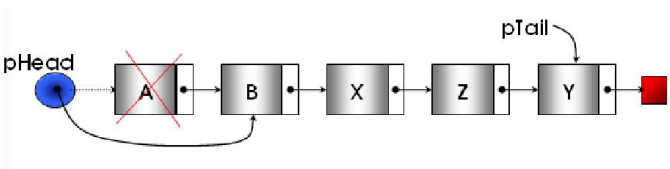
p = Head; *// p là phần tử cần hủy*

Head = Head->pNext; *// tách p ra khỏi xâu*

free(p); *// hủy biến động do p trỏ đến*

Nếu Head=NULL thì Tail = NULL; *//Xâu rỗng*

Minh họa



Hình III.6: Xóa phần tử đầu danh sách

Cài đặt

```
/*RemoveHead(l): hủy phần tử đầu danh sách, hàm trả về dữ liệu của phần tử đầu đã bị hủy*/
Data RemoveHead(LIST &l)
{
    NODE        *p;
    Data   x;
    if ( l.pHead != NULL)
    {
            p = l.pHead;
        x = p->Info;
        l.pHead = l.pHead->pNext;
        delete p;
        if(l.pHead==NULL)
            l.pTail =  NULL;
    }
```

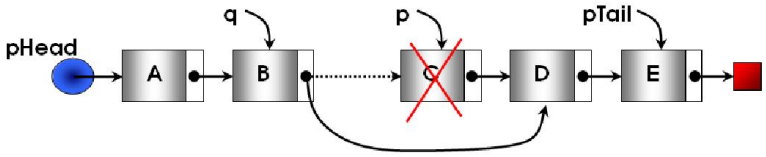
```
    return x;
}
```

b. Hủy một phần tử đứng sau phần tử q trong danh sách

Mô tả

```
    Nếu (q!= NULL)  thì
            p = q->Next;       // p là phần tử cần hủy
            Nếu (p != NULL)  thì // q không phải là cuối xâu
                    q->Next = p->Next; // tách p ra khỏi xâu
                    free(p);          // Hủy biến động do p trỏ đến
```

Minh họa



Hình III.7: Xóa phần tử sau phần tử q

Cài đặt

```
void RemoveAfter (LIST &l, NODE *q)
{
    NODE        *p;
    if ( q != NULL)
    {
            p = q ->pNext ;
            if ( p != NULL)
            {
                    if (p == l.pTail)
                                    l.pTail = q;
                    q->pNext = p->pNext;
                    delete p;
            }
    }
    else RemoveHead(l);
}
```

c. Hủy một phần tử có dữ liệu x

Mô tả

Bước 1: tìm phần tử p có dữ liệu x và phần tử q đứng trước nó

Bước 2:

Nếu (p!=NULL) thì //tìm thấy x

Hủy p ra khỏi xâu như hủy phần tử sau q;

Ngược lại

Báo không có phần tử có dữ liệu x.

Cài đặt

```
int RemoveNode(LIST &l, Data x)
{
    NODE        *p = l.pHead;
    NODE        *q = NULL;//giữ lại địa chỉ nút trước nút cần xóa
//Bước1: Tìm phần tử p có dữ liệu x và phần tử q đứng trước p
    while( p != NULL)
    {
        if(p->Info == x) break;
        q = p;
        p = p->pNext;
    }
//Bước 2: Hủy phần tử có dữ liệu x nếu tìm thấy
    if(p == NULL) return 0; //Không tìm thấy x
    if(q != NULL)
    {
        if(p == l.pTail)
            l.pTail = q;
        q->pNext = p->pNext;
        delete p;
    }
    else //p là phần tử đầu danh sách
    {
        l.pHead = p->pNext;
```

```
            if(l.pHead == NULL) l.pTail = NULL;
    }
    return 1;
}
```

Ta có thể cài đặt thủ tục hủy toàn bộ danh sách như sau:

Mô tả

Để hủy toàn bộ danh sách, thao tác xử lý bao gồm hành động giải phóng một phần tử, do vậy phải cập nhật các liên kết liên quan.

Bước 1: trong khi (danh sách chưa hết) thực hiện

p = Head;

Head:=Head->pNext; *// Cho p trỏ tới phần tử kế tiếp*

Hủy p;

Bước 2: Tail = NULL; *//Bảo giải tính nhất quán khi danh sách rỗng*

Cài đặt

```
void RemoveList(LIST &l)
{
    NODE        *p;
    while (l.pHead!= NULL)
    {
            p = l.pHead;
            l.pHead = p->pNext;
            delete p;
    }
    l.pTail = NULL;
}
```

8. Sắp xếp danh sách

Có hai cách tiếp cận sắp xếp trên danh sách liên kết:

- Phương án 1: hoán vị nội dung các phần tử trong danh sách (thao tác trên vùng info)
- Phương án 2: thay đổi các mối liên kết (thao tác trên trường Next)

Phương án 1: hoán vị nội dung các phần tử trong danh sách liên kết

Ta có thể áp dụng một thuật giải sắp xếp đã biết trên mảng, chẳng hạn thuật giải chọn trực tiếp. Điểm khác biệt duy nhất là cách thức truy xuất đến các phần tử trên xâu thông qua liên kết thay vì chỉ số như trên mảng.

Do thực hiện hoán vị nội dung của các phần tử nên đòi hỏi sử dụng thêm vùng nhớ trung gian → chỉ thích hợp với những danh sách có các phần tử có thành phần info kích thước nhỏ. Nếu kích thước của trường info lớn, việc hoán vị giá trị của hai phần tử sẽ chiếm chi phí đáng kể.

Hoán vị nội dung như trên không tận dụng được các ưu điểm của danh sách liên kết.

Sắp xếp theo phương pháp chọn trên danh sách liên kết, hoán vị nội dung có thể được cài đặt như sau:

```
void        ListSelectionSort (LIST &l)
{
    NODE        *min; //chỉ đến phần tử có giá trị nhỏ nhất trong DSLK
    NODE        *p,*q;
    p = l.pHead;
    while(p != l.pTail)
    {
            min = p;
            q = p->pNext;
            while(q != NULL)
            {
                    if(q->Info  <   min->Info )
                            min = q; //ghi nhận vị trí phần tử min hiện hành
                    q = q->pNext;
              }
            // Hoán vị nội dung hai phần tử
            Hoanvi(min->Info, p->Info);
            p = p->pNext;
    }
}
```

Phương án 2: thay đổi mối liên kết

Thay vì hoán đổi giá trị, ta sẽ tìm cách thay đổi trình tự móc nối của các phần tử sao cho tạo lập nên được thứ tự mong muốn, do đó chỉ thao tác trên các móc nối (pNext).

Kích thước của trường pNext không phụ thuộc vào bản chất dữ liệu lưu trong DSLK và bằng kích thước của một con trỏ (2 hoặc 4 byte trong môi trườnh 16 bit, 4 hoặc 8 byte trong môi trường 32 bit..)

Tuy nhiên, thao tác trên các móc nối thường phức tạp hơn thao tác trực tiếp trên dữ liệu, do đó cần cân nhắc khi chọn cách tiếp cận. Nếu dữ liệu không quá lớn thì nên chọn phương án 1.

Một trong những cách thay đổi móc nối đơn giản nhất là tạo một danh sách mới là danh sách có thứ tự từ danh sách cũ, đồng thời hủy danh sách cũ. Giả sử danh sách mới được quản lý bởi con trỏ đầu xâu Result, ta có phương án 2 của thuật giải chọn trực tiếp như sau:

- Bước 1: Khởi tạo danh sách mới Result là rỗng;
- Bước 2: trìm trong danh sách cũ l phần tử nhỏ nhất;
- Bước 3: tách min khỏi danh sách l;
- Bước 4: chèn min vào cuối danh sách Result;
- Bước 5: Lặp lại bước 2 khi chưa hết danh sách l.

Cài đặt

```
void        ListSelectionSort2 (LIST &l)
{
    LIST   lRes; // DS moi
    NODE          *min;//chỉ đến phần tử có giá trị nhỏ nhất trong DSLK
    NODE          *p,*q, *minprev;
    CreatList(lRes); // khởi tạo lRes
    while(l.pHead != NULL)
    {
            p = l.pHead;
            q = p->pNext;
            min = p;  minprev = NULL;
            while(q != NULL)
            {
                    if(q->Info < min->Info)
```

```
                {
                        min = q;   minprev = p;
                }
                p = q;   q = q->pNext;
        }
        if(minprev != NULL)
                minprev->pNext = min->pNext;
        else
                l.pHead       = min->pNext;
        min->pNext = NULL;
        AddTail(lRes, min);
    }
    l = lRes;
}
```

3.3. Cài đặt tập hợp bằng danh sách liên kết đơn

a. Ý tưởng:

- Tập hợp là một danh sách liên kết đơn,
- Mỗi phần tử của tập hợp là một nút,
- Mỗi nút này có 2 thành phần, một thành phần chứa dữ liệu của nút, thành phần thứ 2 là con trỏ, trỏ đến nút kế tiếp.

b. Cài đặt kiểu tập hợp:

- Kiểu của thành phần dữ liệu của các nút trong tập hợp (Giả sử là kiểu int):

```
typedef  int  Data;
```

- Kiểu Phần tử (Nút) của tập hợp (DSLK):

```
//Định nghĩa ban đầu
struct  tagNode
{
        Data            Info;
        tagNode*        pNext;
};
//Đổi lại tên:
typedef  tagNode  NODE;
```

- Kiểu tập hợp SET (DSLK đơn):

```
struct  SET
{
    NODE* pHead;
    NODE* pTail;
```

};

c. Các phép giải trên tập hợp:

Có thể thực hiện các phép giải:

- Kiểm tra một phần tử có thuộc vào tập hợp
- Kiểm tra quan hệ bao hàm tập hợp
- Hợp 2 tâp hợp
- Giao 2 tập hợp
- ...

1. Kiểm tra 1 phần tử thuộc tập hợp:

Input: S,
x;
Output: 1; x ∈ S
0; Ngược lại
Mô tả:

```
Thuoc(SET S, Data x) ≡
        NODE *p = S.pHead;
        Trong khi (p !=NULL)
        {
                if (p->Info) == x )
                        return 1;
                p = p -> pNext;
        }
        return 0;
```

Cài đặt:

```
int Thuoc(SET S,Data x)
{
        NODE *p;
        p =  S.pHead;
        while(p != NULL)
        {
                if (p -> Info == x)
                        return 1;
                p = p -> pNext;
        }
        return 0;
}
```

2. Giao hai tập hợp

Input: A,B

Output: A∩B
Mô tả:

```
Giao(SET A, SET B, SET &C) ≡
        Tạo tập C  rỗng;
        NODE *p = B.pHead;
        Trong khi (p !=NULL)
        {
                if (Thuoc(A,p->Info))

                        Chèn p->Info vào cuối C;//Chèn x vào DSLK

                p = p -> pNext;

        }
```

3. Hợp hai tập hợp

Input: A,B

Output: A∪B=C
Mô tả:

```
Hop(SET A, SET B, SET &C) ≡
        Tạo tập C  rỗng;
        Copy  A sang C;
        NODE *p = B.pHead;
        Trong khi (p !=NULL)
        {
                if (!Thuoc(A,p->Info)) //Không thuộc

                        Chèn p->Info vào cuối C;

                p = p -> pNext;

        }
```

Tương tự cho các phép giải tập hợp khác...

3.4. Cài đặt đa thức rời rạc bằng danh sách liên kết đơn.

a. Đa thức rời rạc:

Đa thức A một biến theo x, hệ số thực, bậc n có dạng:

$$A(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + \cdots + a_n x^n; a_n \neq 0; a_i \in R, \forall i = 0 \cdots n$$

Ta xét các đa thức trong đó chỉ có một số các hệ số khác 0, các đa thức như vậy ta gọi là đa thức rời rạc. Khi lưu trữ các đa thức rời rạc, ta chỉ lưu trữ các hệ số khác 0.

b. Ý tưởng:

- Đa thức rời rạc là danh sách liên kết đơn,
- Mỗi phần tử của đa thức là một nút,

- Mỗi nút có 2 thành phần, một thành phần chứa dữ liệu của nút được đặt tên là Info có kiểu Data, thành phần thứ 2 là con trỏ, trỏ đến nút kế tiếp.
- Data là kiểu cấu trúc có 2 trường dữ liệu mô tả đặc trưng của một đa thức, bao gồm:
  - ✓ Hệ số của đa thức
  - ✓ Số mũ

c. Cài đặt:

- Kiểu của thành phần dữ liệu của các nút:

```
struct  POLY
{
        double Coef;
        int Expo;
}
//Đổi tên
typedef  POLY  Data;
```

- Kiểu Phần tử (Nút) của Đa thức

```
struct  tagNode
{
        Data            Info;
        tagNode*        pNext;
};
// Đổi tên:
typedef  tagNode  NODE;
```

- Kiểu Đa thức (DSLK đơn):

```
struct  POLYNOME
{
        NODE* pHead;
        NODE* pTail;
};
```

c. Các phép giải:

Có thể thực hiện các phép giải trên đa thức:

- Tổng 2 đa thức,
- Hiệu 2 đa thức,
- Tích 2 đa thức
- Chia đa thức
- Tính giá trị đa thức
- ...

Tổng của 2 đa thức

- Input: A, B
- Output: C = A+B

- Nguyên mẫu: void Add(POLYNOME A, POLYNOME B, POLYNOME &C);
- Mô tả:

```
Add(A,B,C) ≡
        Khởi tạo C rỗng;
        pA = A.pHead; pB = B.pHead;
        Trong khi (pA != NULL && pB != NULL)
        {
                Nếu (pA->Info.Expo <  pB->Info.Expo)
                {       Chèn  pA->Info vào cuối C;
                        pA = pA->pNext ;
                }
                Nếu (pA->Info.Expo  >  pB->Info.Expo)
                {
                        Chèn  pB->Info  vào cuối C;
                        pB = pB->pNext ;
                }
                Nếu (pA->Info.Expo  ==  pB->Info.Expo)
                {
                        P.Coef = pA->Info.Coef + pB->Info.Coef ;
                        P.Expo = pA->Info.Expo ;
                        Chèn  P  vào cuối C;
                        pA = pA->pNext ;
                        pB = pB->pNext ;
                }
        }
        Trong khi (pA != NULL)
        {
                Chèn  pA->Info vào cuối C;
                pA = pA->pNext ;
        }
        Trong khi (pB != NULL)
        {
                Chèn  pB->Info vào cuối C;
                pB = pB->pNext ;
        }
```

Cài đặt:

```
void Add(POLYNOME A, POLYNOME B, POLYNOME &C)
{
    NODE *pA, *pB;
    POLY P;
    CreatP(C);
    pA = A.pHead;
    pB = B.pHead;
```

```
while (pA != NULL && pB != NULL)
{
	if(pA->Info.Expo < pB->Info.Expo)
	{
		InsertTail(C, pA->Info);
		pA = pA->pNext;
	}
	else
		if(pA->Info.Expo > pB->Info.Expo)
		{
			InsertTail(C, pB->Info);
			pB = pB->pNext;
		}
		else
		{
			P.Coef = pA->Info.Coef + pB->Info.Coef;
			P.Expo = pA->Info.Expo;
			InsertTail(C, P);
			pB = pB->pNext;
					pA = pA->pNext;
		}
}
while (pA != NULL)
{
	InsertTail(C, pA->Info);
	pA = pA->pNext;
}
while (pB != NULL)
{
	InsertTail(C, pB->Info);
	pB = pB->pNext;
}
```

}

//void InsertTail(POLYNOME &a, Data x): hàm chèn x vào cuối a.

Tương tự cho các phép giải khác.

3.5. Một số cấu trúc đặc biệt của danh sách liên kết đơn: Ngăn xếp, hàng đợi

3.5.1 Ngăn xếp (Stack)

1. Định nghĩa

Stack là một vật chứa (container) các đối tượng làm việc theo cơ chế LIFO (Last In First Out), do đó việc thêm một đối tượng vào stack hoặc lấy một đối tượng ra khỏi stack được thực hiện trên cùng một đầu theo cơ chế "Vao sau ra trước".

Stack có nhiều ứng dụng: khử đệ quy, tổ chức lưu vết các quá trình tìm kiếm theo chiều sâu và quay lui, vét cạn, ứng dụng trong các bài giải tính giải biểu thức, ...

Stack là một cấu trúc dữ liệu trừu tượng hỗ trợ hai thao tác chính:

- Push(x): thêm đối tượng x vào đầu danh sách
- Pop(): lấy đối tượng ở đầu stack ra khỏi stack và trả về giá trị của nó.

Stack cũng hỗ trợ một số thao tác khác:

- isEmpty(): kiểm tra xem stack có rỗng hay không.
- Top(): trả về giá trị của phần tử nằm ở đầu stack mà không hủy nó khỏi stack. Nếu stack rỗng thì lỗi xảy ra.

Có thể dủng mảng hay danh sách liên kết để biểu diễn, cài đặt stack.

2. Dùng mảng:

a. Biểu diễn stack

Có thể tạo một stack bằng cách khai báo một mảng 1chiều với kích thước tối đa là N (ví dụ: N = 1000). Khi đó stack có thể chứa tối đa N phần tử đánh số từ 0 đến N-1. Phần tử nằm ở đầu stack sẽ có chỉ số t (lúc đó trong stack đang chứa t+1 phần tử).

Để khai báo một stack, ta cần một mảng một chiều S, biến nguyên t cho biết chỉ số

của đầu stack và hằng số N cho biết kích thước tối đa của stack.

```
Data S[N];
int  t;
```

b. Cài đặt stack:

Lệnh t = 0 sẽ tạo ra một stack S rỗng.

Giá trị của t cho biết số phần tử hiện hành có trong stack.

Khi cài đặt bằng mảng một chiều, stack có kích thước tối đa nên cần xây dựng thêm một thao tác phụ cho stack IsFull() để kiểm tra xem stack đã đầy hay chưa. Khi stack đầy, việc gọi hàm push để đẩy thêm một phần tử vào stack sẽ phát sinh ra lỗi.

- Hàm kiểm tra stack có rỗng hay không

```
int IsEmpty()
{   if(t == 0) // stack rỗng
      return 1;
 return 0;
}
```

- Hàm kiểm tra stack đầy hay không

```
int IsFull()
{   if(t >= N) // stack đầy
            return 1;
    return 0;
}
```

- Thêm một phần tử x vào stack

```
void Push(Data x)
{   if(t < N) // stack chưa đầy
    {       S[t] = x; t++; }
    else cout<<"Stack day";
}
```

- Lấy thông tin và hủy phần tử ở đỉnh của stack

```
Data Pop()
{   Data   x;
    if(t > 0) // stack khác rỗng
    { t--; x = S[t]; return x;}
    else cout<<" Stack rong";
}
```

- Xem thông tin của phần tử ở đỉnh của stack

```
Data Top()
{   Data   x;
    if(t > 0) // stack khác rỗng
    {         x = S[t-1];
     return   x;
    }
    else  puts("Stack rong")
}
```

Nhận xét

- Các thao tác trên đều làm việc với chi phí O(1).
- Việc cài đặt stack thông qua mảng một chiều đơn giản và khá hiệu quả. Tuy nhiên hạn chế lớn nhất của phương án cài đặt này là giới hạn về kích thước của stack. Giá trị N có thể quá nhỏ so với như cầu thực tế hoặc quá lớn sẽ gây lãng phí bộ nhớ.

3. Dùng DSLK:

a. Biểu diễn stack

Có thể tạo một stack bằng cách sử dụng một danh sách liên kết đơn. DSLK có những đặc tính rất phù hợp để dùng làm stack vì mọi thao tác trên stack đều diễn ra ở đầu stack.

b. Cài đặt stack:

Giả sử ta có các định nghĩa:

```
typedef int Data;
struct  tagNode
{
    Data            Info;
    tagNode*        pNext;
};
    // kiểu của một phần tử trong DSLK
        typedef tagNode NODE;
```

Định nghĩa stack

```
struct  STACK
{
```

```
    NODE* pHead;
    NODE* pTail;
};
```

- Tạo stack rỗng

```
void CreatStack(STACK &S)
{
        S.pHead=S.pTail= NULL;
}
```

- Kiểm tra stack rỗng

```
int  IsEmpty(STACK S)
{ if (S.pHead == NULL) // stack rỗng
        return 1;
  return 0;
}
```

Thêm một phần tử x vào stack

```
void  Push(STACK &S, Data x)
{
        NODE* new_ele = CreateNode(x);
        if (new_ele ==NULL)  exit(1); //return;
        if (l.pHead==NULL) //DS rỗng
        {
                l.pHead = new_ele;
                l.pTail = l.pHead;
        }
        else
        {
                new_ele->pNext = l.pHead;
                l.pHead = new_ele;
        }
}
```

- Lấy thông tin và hủy phần tử ở đỉnh stack

```
Data  Pop(STACK &S)
{
    NODE *p;
    Data  x;
    if (isEmpty(S))
            return NULLDATA;
    if ( l.pHead != NULL)
    {
            p = l.pHead;
            x = p->Info;
            l.pHead = l.pHead->pNext;
            delete p;
            if(l.pHead==NULL)
                    l.pTail =  NULL;
    }
    return x;
}
```

- Lấy thông tin phần tử ở đỉnh của stack

```
Data Top(STACK &S)
{
            if(isEmpty(S))
                    return  NULLDATA;
            return  l.pHead->Info;
}
```

Nhận xét

Stack thích hợp lưu trữ các loại dữ liệu mà trình tự truy xuất ngược với trình tự lưu trữ.

c. Một số ứng dụng của stack

- Trong trình biên dịch (thông dịch), khi thực hiện các thủ tục, stack được sử dụng để lưu môi trường của các thủ tục.
- Đánh giá biểu thức

- Lưu dữ liệu khi giải một số bài giải của lý thuyết đồ thị (như tìm đường đi).
- Khử đệ qui

Ví dụ 1:

Thuật giải POLISH chuyển biểu thức trung tố sang hậu tố RPN

Input: Sin (Chuỗi chứa biểu thức trung tố)

Output: Sout (Chuỗi chứa biểu thức hậu tố)

Mô tả thuật giải:

B1. Khởi tạo STACK (dùng để chứa các giải tử) S rỗng;

B2. Lặp lại các việc sau cho đến khi dấu kết thúc biểu thức được đọc:

. Đọc phần tử tiếp theo (hằng, biến, giải tử, '(', ')') trong biểu thức trung tố.

. Nếu phần tử là:

- Dấu '(': đẩy nó vào S;
- Dấu ')': hiển thị các phần tử của S cho đến khi dấu '(' (không hiển thị) được đọc;
- Giải tử:

Nếu S rỗng: đẩy giải tử vào S; // (1)

Ngược lại:

Nếu giải tử đó có độ ưu tiên cao hơn giải tử ở đỉnh S thì:

Đẩy giải tử đó vào S;

Ngược lại: lấy ra và hiển thị giải tử ở đỉnh S ;

Quay lại (1);

- Giải hạng (hằng hoặc biến): Hiển thị nó;

B3. Khi đạt đến dấu kết thúc biểu thức thì lấy ra và hiển thị các giải tử của S cho đến khi S rỗng;

(trong đó, ta xem dấu '(' có độ ưu tiên thấp hơn độ ưu tiên các giải tử +, -, *, /, %)

Minh họa:

Input: 2*3+(6-4) //Biểu thức trung tố

Output: 2 3 * 6 4 - + //Biểu thức hậu tố RPN

	Sin[i]	Stack S	Sout
			""
1	2		"2"
2	*	*	
3	3	*	"23"

4	+	+	“23*”
5	(	+(	“AB*”
6	6	+(	“23*6”
7	-	+(-	“23*6”
8	4	+(-	“23*64”
9	)	+	“23*64-”

Vây: Sout = “23*64-+”

Ví dụ 2:

Thuật giải đánh giá biểu thức hậu tố RPN

Input: Biểu thức hậu tố S

Output: Giá trị của biểu thức trung tố tương ứng

Mô tả thuật giải:

B1. Khởi tạo STACK *S* rỗng;

B2. Lặp lại các việc sau cho đến khi dấu kết thúc biểu thức được đọc:

. Đọc phần tử (giải hạng, giải tử) tiếp theo trong biểu thức;
Nếu phần tử là giải hạng: đẩy nó vào *S*;
Ngược lại: // phần tử là giải tử
- Lấy từ đỉnh *S* hai giải hạng;
- Áp dụng giải tử đó vào 2 giải hạng (theo thứ tự ngược);
- Đẩy kết quả vừa tính trở lại *S*;

B3. Khi gặp dấu kết thúc biểu thức, giá trị của biểu thức chính là giá trị ở đỉnh S.

Minh họa:

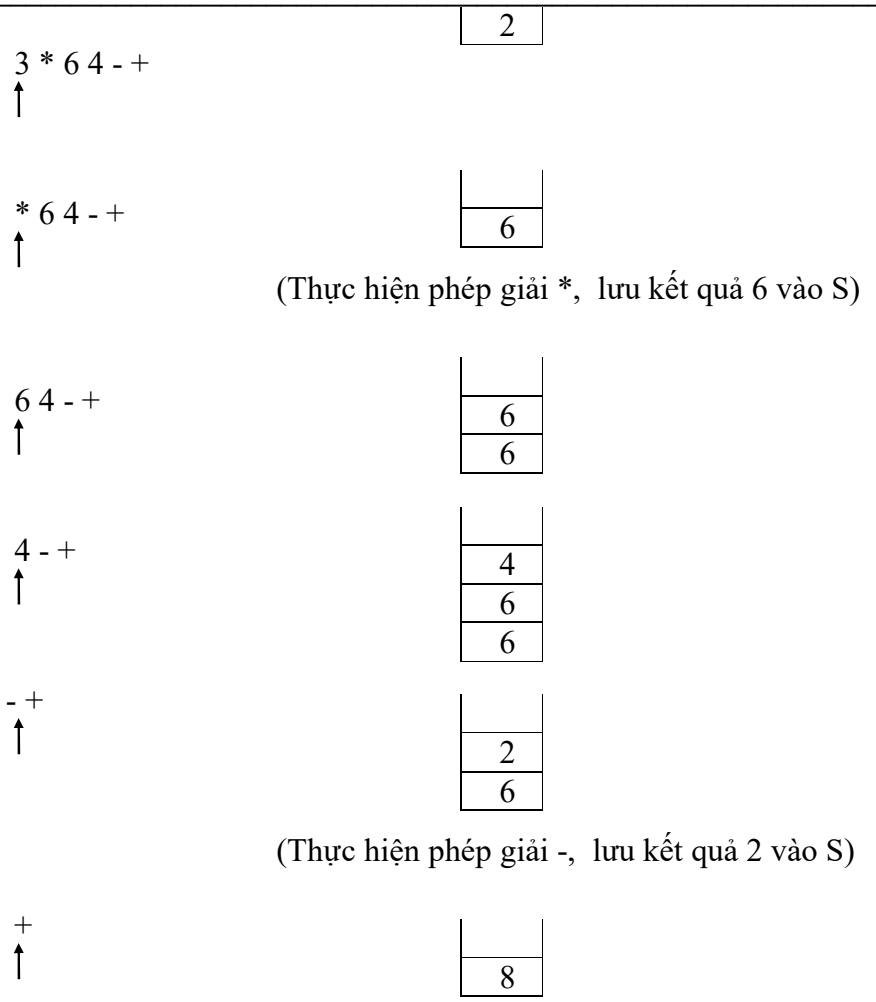
Input: 2 3 * 6 4 - + //Biểu thức hậu tố RPN

Output: Giá trị biểu thức trung tố 2*3+(6-4)

Biểu thức RPN

2 3 * 6 4 - +
↑

Stack *S*



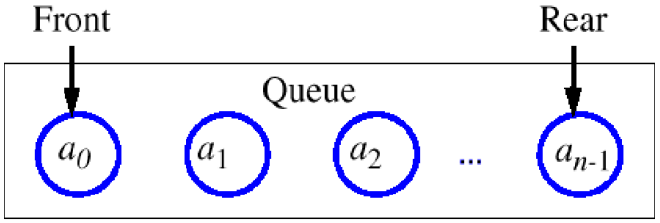
Kết thúc biểu thức: 8

3.5.2 Hàng đợi (Queue)

1. Định nghĩa

Hàng đợi là một vật chứa (container) các đối tượng làm việc theo cơ chế FIFO (First In First Out), do đó việc thêm một đối tượng vào hảng đợi hoặc lấy một đối tượng ra khỏi hàng đợi được thực hiện theo cơ chế "Vào trước ra trước".

Việc thêm một đối tượng vào hàng đợi luôn diễn ra ở cuối hàng đợi và một phần tử luôn được lấy ra từ đầu hàng đợi.



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

Hàng đợi hỗ trợ các thao tác:

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

2. Dùng mảng:

a. Biểu diễn hàng đợi:

Có thể tạo một hàng đợi bằng cách sử dụng một mảng một chiều với kích thước tối đa là N (ví dụ: N=1000) theo kiểu xoay vòng (coi phần tử a_{N-1} kề với phần tử a_0). Do đó hàng đợi chứa tối đa N phần tử.

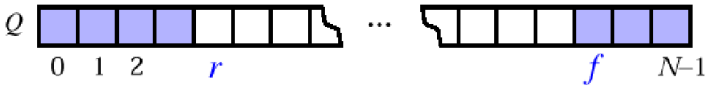
Phần tử ở đầu hàng đợi (front element) sẽ có chỉ số f, phần tử ở cuối hàng đợi (rear element) sẽ có chỉ số là r.

Để khai báo hàng đợi ta cần một mảng một chiều Q, hai biến nguyên f và r cho biết chỉ số của đầu và cuối hàng đợi, hằng số N cho biết kích thước tối đa của hàng đợi.

Ngoài ra khi dùng mảng biểu diễn hàng đợi, ta cần dùng một giá trị đặc biệt, kí hiệu NULLDATA để gán cho những ô còn trống trên hàng đợi. Giá trị này là một giá trị nằm ngoài miền xác định của dữ liệu trong hàng đợi.

Trạng thái hàng đợi lúc bình thường

Trạng thái hàng đợi lúc xoay vòng



b. Cài đặt hàng đợi:

Hàng đợi có thể được khai báo cụ thể như sau:

```
Data  Q[N];
```

```
int      f, r;
```

Do khi cài đặt bằng mảng một chiều, hàng đợi có kích thước tối đa nên cần xây dựng thêm một thao tác phụ cho hàng đợiIsFull() để kiểm tra xem hàng đợi có đầy hay chưa.

- Tạo hàng đợi rỗng

```
void   InitQueue()
{
    f = r = 0;
    for(int i = 0; i < N; i++)
            Q[i] = NULLDATA;
}
```

Kiểm tra hàng đợi rỗng hay không

```
int  IsEmpty()
{
    return (Q[f] == NULLDATA);
}
```

- Kiểm tra hàng đợi đầy hay chưa

```
int IsFull()
{
 return (Q[r] != NULLDATA);
}
```

- Thêm một phần tử x vào cuối hàng đợi Q

```
int EnQueue(Data x)
{
    if(IsFull()) return -1; //Queue  đầy
    Q[r++] = x;
    if(r == N)      // xoay vòng
            r = 0;
}
```

- Lấy thông tin và hủy phần từ ở đầu hàng đợi Q

```
Data        DeQueue()
{   Data x;
    if(IsEmpty()) return NULLDATA; //Queue  rỗng
    x = Q[f]; Q[f++] = NULLDATA;
    if(f == N)      f = 0; // xoay vòng
    return x;
}
```

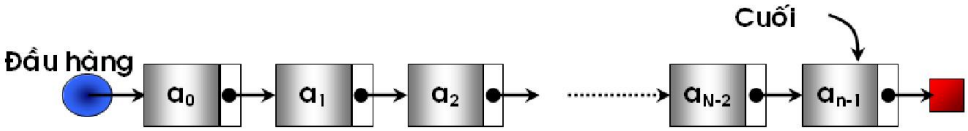
- Lấy thông tin của phần tử ở đầu hàng đợi Q

```
Data        Front()
{
    if(IsEmpty()) return NULLDATA; //Queue  rỗng
    return Q[f];
}
```

3. Dùng danh sách liên kết

a. Biểu diễn hàng đợi:

Ta có thể tạo một hàng đợi sử dụng một DSLK đơn, phần tử đầu DSLK (head) sẽ là phần tử đầu hàng đợi, phần tử cuối DSLK (tail) sẽ là phần tử cuối hàng đợi.



b. Cài đặt hàng đợi

Cài đặt các thao tác trên danh sách liên kết

- Tạo hàng đợi rỗng

```
void CreatQ(QUEUE &Q)
{
        Q.pHead=Q.pTail= NULL;
}
```

Kiểm tra hàng đợi rỗng

```
int  IsEmpty(QUEUE Q)
{ if (Q.pHead == NULL) // hàng đợi rỗng
        return 1;
```

```
    return 0;
}
Thêm một phần tử p vào cuối hàng đợi
void EnQueue(QUEUE &Q, Data x)
{   InsertTail(Q, x);}
```

- Lấy, hủy phần tử ở đầu hàng đợi

```
Data        DeQueue(QUEUE &Q)
{   Data    x;
    if (IsEmpty(Q))
                    return NULLDATA;
    x = RemoveHead(Q);
    return x;
}
```

- Xem thông tin của phần tử ở đầu hàng đợi

```
Data        Front(QUEUE Q)
{   if (IsEmpty(Q))
                    return NULLDATA;
    return Q.pHead->Info;
}
```

Nhận xét

- Các thao tác trên hàng đợi biểu diễn bằng danh sách liên kết làm việc với chi phí O(1).
- Nếu không quản lý phần tử cuối xâu, thao tác Dequeue sẽ có độ phức tạp O(n).

c. Ứng dụng của hàng đợi trong một số bài toán

- Bài giải "sản xuất và tiêu thụ" (ứng dụng trong các hệ điều hành song song).
- Bộ đệm (ví dụ: nhấn phím → bộ đệm → CPU xử lý).

Xử lý các lệnh trong máy tính (ứng dụng trong HDH, trình biên dịch), hàng đợi các tiến trình chờ được xử lý, …

3.6. Một số cấu trúc dữ liệu dạng danh sách liên kết khác

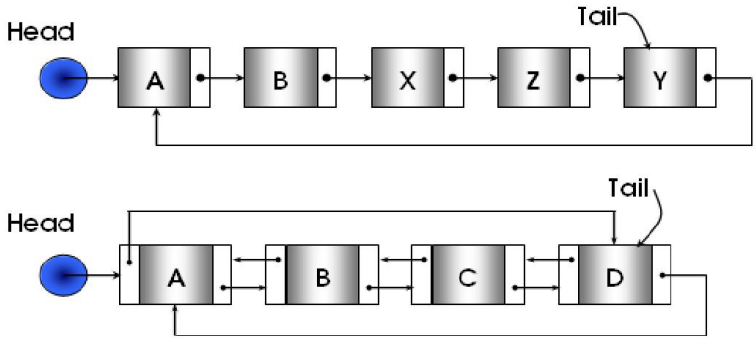
3.6.1 Danh sách liên kết vòng

1. Định nghĩa

Danh sách liên kết vòng (xâu vòng) là một danh sách đơn (hoặc kép) mà phần tử cuối danh sách thay vì mang giá trị NULL thì nó trỏ tới phần tử đầu danh sách.

Đối với danh sách vòng, ta có thể xuất phát từ một phần từ bất kì để duyệt toàn bộ danh sách.

Biểu diễn danh sách liên kết vòng



2. Các thao tác trên danh sách liên kết vòng (biểu diễn bằng DSLK đơn)
   - Duyệt danh sách liên kết vòng:

Danh sách vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kì trên danh sách xem như phần tử đầu danh sách để kiểm tra việc duyệt đã hết phần tử của danh sách hay chưa.

```
NODE*    Search(LIST l, Data  x)
{   NODE          *p;
    p = l.pHead;
    do
    {  if ( p->Info == x) return p;
       p = p->pNext;
    } while  (p != l.pHead); // chưa đi hết vòng
    return NULL;//Khong co
}
```

   - Thêm phần tử vào đầu danh sách

```
void        AddHead(LIST &l, NODE *new_ele)
{
    if(l.pHead == NULL)  //danh sách rỗng
    {
```

```
            l.pHead = l.pTail = new_ele;
            l.pTail->pNext = l.pHead;
    }
    else
    {
            new_ele->pNext = l.pHead;
            l.pTail->pNext = new_ele;
            l.pHead = new_ele;
    }
}
```

- Thêm phần tử vào cuối danh sách

```
void        AddTail(LIST &l, NODE *new_ele)
{
    if(l.pHead == NULL)  //danh sách rỗng
    {
            l.pHead = l.pTail = new_ele;
            l.pTail->pNext = l.pHead;
    }
    else
    {
            new_ele->pNext = l.pHead;
            l.pTail->pNext = new_ele;
            l.pTail = new_ele;
    }
}
```

- Thêm phần tử sau nút p

```
void        AddAfter(LIST &l, NODE *q, NODE *new_ele)
{
    if(l.pHead == NULL)  //danh sách rỗng
    {
            l.pHead = l.pTail = new_ele;
```

```
            l.pTail->pNext = l.pHead;
    }
    else
    {
            new_ele->pNext = q->pNext;
            q->pNext = new_ele;
            if(q == l.pTail)
              l.pTail = new_ele;
    }
}
```

- Hủy phần tử đầu xâu

```
void RemoveHead(LIST &l)
{
    NODE        *p = l.pHead;
    if(p == NULL) return;
    if (l.pHead == l.pTail)
     l.pHead = l.pTail = NULL;
    else
    {
            l.pHead = p->Next;
            if(p == l.pTail)
              l.pTail->pNext = l.pHead;
    }
    delete p;
}
```

- Hủy phần tử đứng sau nút p

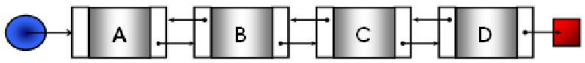
```
void RemoveAfter(LIST &l, NODE *q)
{   NODE        *p;
    if(q != NULL)
    {
            p = q ->Next ;
```

```
            if ( p == q) //chi co Mot nut
                    l.pHead = l.pTail = NULL;
            else
            {
              q->Next = p->Next;
              if(p == l.pTail)
                    l.pTail = q;
            }
            delete p;
        }
    }
```

3.6.2 Danh sách liên kết kép

1. Định nghĩa

Danh sách liên kết kép là danh sách mà mỗi phần tử trong danh sách có kết nối với 1 phần tử đứng trước và 1 phần tử đứng sau nó.



2. Cài đặt:

pPre liên kết với phần từ dứng trước.

pNext liên kết với phần từ dứng sau.

```
struct  tagDNode
{
    Data    Info;
    tagDNode* pPre;
    tagDNode* pNext;
};
typedef tagDNode DNODE;
struct DLIST
{
    DNODE* pHead;    // trỏ đến phần tử đầu danh sách
    DNODE* pTail;    // trỏ đến phần tử cuối danh sách
};
```

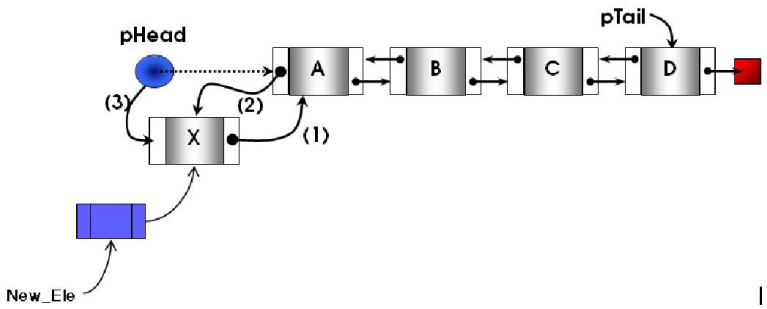
- Thủ tục tạo nút cho danh sách liên kết kép với trường Info là x

```
DNODE* CreateNode(Data x)
{   DNODE *p;
    // Cấp phát vùng nhớ cho phần tử
    p = new DNODE;
    if ( p==NULL)
    {   cout<<"Khong du bo nho";
            return NULL; // exit(1);
    }
    // Gán thông tin cho phần tử p
    p->Info = x;
    p->pPrev = p->pNext = NULL;
    return p;
}
```

- Chèn một phần tử vào DSLK kép

Có 4 cách chèn một nút new_ele vào danh sách kép:

- ✓ Chèn đầu danh sách
- ✓ Chèn cuối danh sách
- ✓ Chèn nút vào sau một phần tử p
- ✓ Chèn nút vào trước phần tử p
- ✓ Chèn đầu danh sách

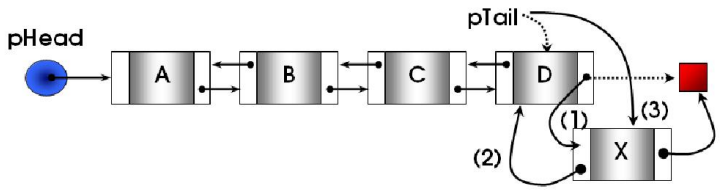


✓ Chèn đầu danh sách

```
void AddFirst(DLIST &l, DNODE* new_ele)
{
    if (l.pHead==NULL)  //DS rỗng
```

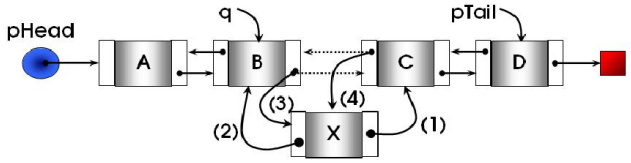
```
    {
        l.pHead = new_ele;
        l.pTail = l.pHead;
    }
    else
    {
        new_ele->pNext = l.pHead;          // (1)
        l.pHead->pPrev = new_ele;// (2)
        l.pHead = new_ele;  // (3)
    }
}
```

✓ Chèn phần tử vào cuối danh sách liên kết



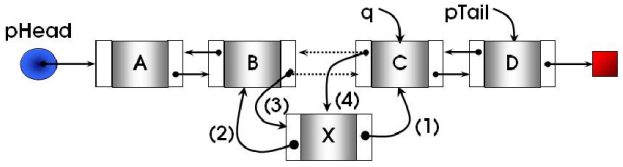

```
void AddTail(DLIST &l, DNODE *new_ele)
{
    if (l.pHead==NULL)
    {
        l.pHead = new_ele;
        l.pTail = l.pHead;
    }
    else
    {
        l.pTail->Next = new_ele;    // (1)
        new_ele->pPrev = l.pTail;  // (2)
        l.pTail = new_ele;    // (3)
    }
}
```

✓ Chèn một phần tử sau một phần tử q trong danh sách




```
void AddAfter(DLIST &l, DNODE* q,DNODE* new_ele)
{
    DNODE* p = q->pNext;
    if ( q!=NULL)
    {
            new_ele->pNext = p;                 //(1)
            new_ele->pPrev = q;                 //(2)
            q->pNext = new_ele;                 //(3)
            if(p != NULL)   p->pPrev = new_ele; //(4)
            if(q == l.pTail) l.pTail = new_ele;
    }
    else  //chèn đầu danh sách
            AddFirst(l, new_ele);
}
```

✓ Chèn một phần tử vào trước phần tử q trong danh sách



Hình III.10: chèn phần tử x trước phần tử q

```
void AddBefore(DLIST &l, DNODE q, DNODE* new_ele)
{
    DNODE* p = q->pPrev;
    if ( q!=NULL)
    {
            new_ele->pNext = q;                 //(1)
            new_ele->pPrev = p;                 //(2)
            q->pPrev = new_ele;                 //(3)
```

```
            if(p != NULL)   p->pNext = new_ele;      //(4)
            if(q == l.pHead) l.pHead = new_ele;
    }
    else  //chèn vào cuối danh sách
            AddTail(l, new_ele);
}
```

- Hủy một phần tử ra khỏi danh sách

Có 5 thao tác thông dụng để hủy một phần tử ra khỏi danh sách liên kết kép

- ✓ Hủy phần tử đầu danh sách
- ✓ Hủy phần tử đầu danh sách
- ✓ Hủy phần tử sau phần tử q
- ✓ Hủy phần tử trước phần tử q
- ✓ Hủy phần tử có khóa k
- ✓ Hủy phần tử đầu xâu

- Hủy phần tử đầu danh sách

```
Data RemoveHead(DLIST &l)
{
    DNODE       *p;
    Data    x = NULLDATA;
    if ( l.pHead != NULL)
    {
            p = l.pHead; x = p->Info;
            l.pHead = l.pHead->pNext;
            l.pHead->pPrev = NULL;
            delete p;
            if(l.pHead == NULL) l.pTail = NULL;
            else l.pHead->pPrev = NULL;
    }
    return x;
}
```

- Hủy phần tử cuối xâu

```
Data RemoveTail(DLIST &l)
{
    DNODE       *p;
    Data    x = NULLDATA;
    if ( l.pTail != NULL)
    {
            p = l.pTail; x = p->Info;
            l.pTail = l.pTail->pPrev;
            l.pTail->pNext = NULL;
            delete p;
            if(l.pHead == NULL) l.pTail = NULL;
            else l.pHead->pPrev = NULL;
    }
    return x;
}
```

- Hủy một phần tử đứng sâu phần tử q

```
Data RemoveTail(DLIST &l)
{
    DNODE       *p;
    Data    x = NULLDATA;
    if ( l.pTail != NULL)
    {
            p = l.pTail; x = p->Info;
            l.pTail = l.pTail->pPrev;
            l.pTail->pNext = NULL;
            delete p;
            if(l.pHead == NULL) l.pTail = NULL;
            else l.pHead->pPrev = NULL;
    }
    return x;
```

```
}
```

- Hủy một phần tử đứng trước phần tử q

```
void RemoveBefore (DLIST &l, DNODE *q)
{
    DNODE       *p;
    if ( q != NULL)
    {
            p = q ->pPrev;
            if ( p != NULL)
            {
                    q->pPrev = p->pPrev;
                    if(p == l.pHead) l.pHead = q;
                    else p->pPrev->pNext = q;
                    delete p;
            }
    }
    else
            RemoveTail(l);
}
```

- Hủy một phần tử có khóa k

```
int RemoveNode(DLIST &l, Data k)
{
    DNODE       *p = l.pHead;
    NODE        *q;
    while( p != NULL)
    {
            if(p->Info == k) break;
            p = p->pNext;
    }
    if(p == NULL) return 0; //Không tìm k
    q = p->pPrev;
```

```
        if ( q != NULL)
        {
                p = q ->pNext ;
                if ( p != NULL)
                {
                        q->pNext = p->pNext;
                        if(p == l.pTail) l.pTail = q;
                        else p->pNext->pPrev = q;
                 }
        }
        else //p là phần tử đầu danh sách
        {
                l.pHead = p->pNext;
                if(l.pHead == NULL) l.pTail = NULL;
                else l.pHead->pPrev = NULL;
        }
        delete p;
        return 1;
    }
```

- Sắp xếp trên danh sách liên kết kép

Việc sắp xếp trên danh sách liên kết kép về cơ bản không khác gì trên danh sách liên kết đơn. Ta chỉ cần lưu ý một điều duy nhất là cần bảo toàn các mối liên kết hai chiều trong khi sắp xếp.

Ví dụ 3.4: cài đặt thuật giải sắp xếp QuickSort

```
void   DListQSort(DLIST & l)
{
    DNODE       *p, *X;         // X chỉ đến phần tử cầm canh
    DLIST       l1, l2;
    if(l.pHead == l.pTail) return;//đã có thứ tự
    l1.pHead == l1.pTail = NULL; //khởi tạo
    l2.pHead == l2.pTail = NULL;
```

```
        X = l.pHead; l.pHead = X->pNext;
        while(l.pHead != NULL) //Tách l thành l1, l2;
        {
                p = l.pHead;
                l.pHead = p->pNext; p->pNext = NULL;
                if (p->Info <= X->Info) AddTail(l1, p);
                else AddTail(l2, p);
        }
        DListQSort(l1);       //Gọi đệ qui để sắp l1
        DListQSort(l2);       //Gọi đệ qui để sắp l2
        //Nói l1, X, l2 thành l đã sắp.
        if(l1.pHead != NULL)
        {
                l.pHead = l1.pHead; l1.pTail->pNext = X;
                X->pPrev = l1.pTail;
        }
        else l.pHead = X;
        X->pNext = l2;
        if(l2.pHead != NULL)
        {
                l.pTail = l2.pTail;
                l2->pHead->pPrev = X;
        }
        else l.pTail = X;
    }
```

Nhận xét

Xâu kép về co bản có tính chất giống như xâu đơn, tuy nhiên nó cómột số tính chất khác như sau:

- Xâu kép có mối liên kết hai chiều nên từ một phần tử bất kì có thể truy xuất một phần tử bất kì khác. Trong khi trên xâu đơn ta chỉ có thể truy xuất đến các phần tử đứng sau phần tử cho trước. Điều này dẫn đến việc ta có thể dễ dàng hủy phần tử cuối xâu kép, còn trên xâu đơn thao tác này tốnchi phí O(n).

- Bù lại, xâu kép tốn chi phí gấp đôi so với xâu đơn cho việc lưu trữ các mối liên kết. Điều này khiến việc cập nhật cũng nặng nề hơn trong một số trường hợp. Như vậy ta cần cân nhắc lựa chọn CTDL hợp lý khi cài đặtcho một ứng dụng cụ thể.

Bài tập

Bài 1:

Kiểu Data định nghĩa là kiểu số nguyên

Viết chương trình thực hiện các thao tác trên danh sách liên kết đơn. Yêu cầu của chương trình là:

- In ra màn hình menu có các chức năng sau:

1. Xuất dữ liệu của danh sách ra màn hình
2. Tìm một phần tử trong danh sách có dữ liệu x? nếu không có trả về 0, nếu có trả về vị trí nút đầu tiên (lưu ý nút đầu của danh sách đánh dấu là vị trí 1)
3. Tìm một phần tử trong danh sách có dữ liệu x? nếu không có trả về 0, nếu có trả về vị trí nút cuối cùng (lưu ý nút đầu của danh sách đánh dấu là vị trí 1)
4. Đảo ngược danh sách đã cho
5. Hủy tất cả các nút có dữ liệu là x.
6. Hủy nút cuối danh sách.
7. Đếm số nút của danh sách
8. Chép danh sách đã cho sang một danh sách khác.
9. Tách lần lượt từng nút của danh sách l sang các danh sách đơn l1, l2.
10. Tách danh sách l sang các danh sách đơn l1, l2 với l1 chứa nửa số nút đầu của l, l2 chứa các nút còn lại
11. Thoát

- Nhập dữ liệu cho danh sách, muốn thực hiện thao tác nào thì chọn chức năng tương ứng của menu.

Bài 2:

Kiểu Data định nghĩa như sau:

```
struct Data
{
        char  HoTen[30];
        double Luong;
        char CMND[15];
}
```

Viết chương trình thực hiện các thao tác trên danh sách liên kết đơn. Yêu cầu của chương trình là:

- In ra màn hình menu có các chức năng sau:

1. Xuất dữ liệu của danh sách ra màn hình
2. Sắp xếp tăng dần danh sách liên kết theo khóa lương.
3. Xuất dữ liệu các nút có tên trùng với Họ tên cho trước.
4. Thoát

- Nhập dữ liệu cho danh sách, muốn thực hiện thao tác nào thì chọn chức năng tương ứng của menu.

Bài 3:

Viết chương trình thực hiện các phép giải trên tập hợp (tập hợp được cài đặt bằng danh sách liên kết đơn). Yêu cầu của chương trình là:

- In ra màn hình menu có các chức năng sau:

1. Xem tập hợp.
2. Tính hợp 2 tập hợp
3. Tính giao 2 tập hợp.
4. Kiểm tra 1 phần tử có thuộc vào tập hợp.
5. Tính A\B.
6. Tính hiệu đối xứng 2 tập hợp: (A\B)U(B\A).
7. Tính lực lượng tập hợp
8. Thoát

- Nhập dữ liệu cho tập hợp, muốn thực hiện thao tác nào thì chọn chức năng tương ứng của menu.

Bài 4:

Viết chương trình thực hiện các phép giải trên Đa thức (dùng danh sách liên kết đơn lưu trữ các hệ số khác 0 và số mũ tương ứng). Yêu cầu của chương trình là:

- In ra màn hình menu có các chức năng sau:

1. Xuất đa thức ra màn hình.
2. Cộng 2 đa thức
3. Trừ 2 đa thức.
4. Nhân 2 đa thức.
5. Tính giá trị của đa thức tại x.
6. Thoát

- Nhập dữ liệu cho tập hợp, muốn thực hiện thao tác nào thì chọn chức năng tương ứng của menu.

Bài 5:

Viết chương trình thực hiện các thao tác trên Stack. Yêu cầu của chương trình là:

- In ra màn hình menu có các chức năng sau:

1. Xuất dữ liệu của Stack ra màn hình
2. Kiem tra Stack rỗng hay không?
3. 3, Xem đỉnh của Stack
4. Tìm một phần tử trong Stack.
5. Thêm một phần tử vào đầu Stack.
6. Hủy phần tử đầu khỏi Stack.

7. Đếm số nút của Stack
8. Copy Stack đã cho sang một Stack khác.
9. Thoát

- Nhập dữ liệu cho Stack, muốn thực hiện thao tác nào thì chọn chức năng tương ứng của menu.

Bài 6:

Cài đặt thuật giải chuyển đổi biểu thức trung tố sang biểu thức hậu tố

Bài 7:

Cài đặt thuật giải đánh giá biểu thức hậu tố

Bài 8:

Viết chương trình thực hiện các thao tác trên Queue (cài đặt bằng danh sách liên kết đơn).Yêu cầu của chương trình là:

- In ra màn hình menu có các chức năng sau:

1. Xuất dữ liệu của Queue ra màn hình
2. Kiem tra Queue rỗng hay không?
3. Tìm một phần tử trong Queue.
4. Thêm một phần tử vào cuối Queue.
5. Hủy phần tử đầu khỏi Queue.
6. Đếm số nút của Queue
7. Thoát.

- Nhập dữ liệu cho Queue, muốn thực hiện thao tác nào thì chọn chức năng tương ứng của menu.

Bài 9:

Viết chương trình thực hiện các thao tác trên danh sách liên kết vòng (đơn). Yêu cầu của chương trình là:

- In ra màn hình menu có các chức năng sau:

1. Xuất dữ liệu của danh sách ra màn hình
2. Kiem tra danh sách rỗng hay không?
3. Tìm một phần tử trong danh sách.
4. Chèn một phần tử vào danh sách.
5. Hủy một phần tử khỏi danh sách.
6. Thoát

- Nhập dữ liệu cho danh sách, muốn thực hiện thao tác nào thì chọn chức năng tương ứng của menu.

Bài 10 *(Bài giải Josephus):*

Một nhóm binh sĩ bị kẻ thù bao vây và một binh sĩ được chọn để đi cầu cứu.

Việc chọn được thực hiện theo cách sau đây:

“ Một số nguyên n và một binh sĩ được chọn ngẫu nhiên. Các binh sĩ được sắp theo vòng tròn và họ đếm từ binh sĩ được chọn ngẫu nhiên. Khi đạt đến n, binh sĩ tương

ứng được lấy ra khỏi vòng và việc đếm lại được bắt đầu từ binh sĩ tiếp theo. Quá trình này tiếp tục cho đến khi chỉ còn lại một binh sĩ được chọn để đi cầu cứu".

Hãy viết một thuật giải cài đặt cách chọn này, dùng danh sách liên kết vòng để lưu trữ các tên của binh sĩ.

Bài 11:

Giả sử số nguyên lưu trữ trong danh sách liên kết đơn. Cài đặt các phép giải số học tương ứng

Bài 12:

Giả sử số nguyên lưu trữ trong danh sách liên kết kép. Cài đặt các phép giải số học tương ứng.

CHƯƠNG 4. CÂY

4.1 Cấu trúc cây

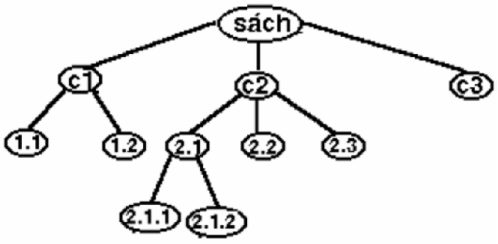
Cây là một tập hợp các phần tử gọi là nút (nodes) trong đó có một nút được phân biệt gọi là nút gốc (root). Trên tập hợp các nút này có một quan hệ, gọi là mối quan hệ cha - con (parenthood), để xác định hệ thống cấu trúc trên các nút. Mỗi nút, trừ nút gốc, có duy nhất một nút cha. Một nút có thể có nhiều nút con hoặc không có nút con nào. Mỗi nút biểu diễn một phần tử trong tập hợp đang xét và nó có thể có một kiểu nào đó bất kỳ, thường ta biểu diễn nút bằng một kí tự, một chuỗi hoặc một số ghi trong vòng tròn. Mối quan hệ cha con được biểu diễn theo qui ước nút cha ở dòng trên nút con ở dòng dưới và được nối bởi một đoạn thẳng.

Một cách hình thức ta có thể định nghĩa cây một cách đệ qui như sau:

4.1.1 Định nghĩa :

- Một nút đơn độc là một cây. Nút này cũng chính là nút gốc của cây.
- Giả sử ta có n là một nút đơn độc và k cây T_1,.., T_k với các nút gốc tương ứng là n_1,.., n_k thì có thể xây dựng một cây mới bằng cách cho nút n là cha của các nút n_1,.., n_k. Cây mới này có nút gốc là nút n và các cây T_1,.., T_k được gọi là các cây con. Tập rỗng cũng được coi là một cây và gọi là cây rỗng.

Ví dụ: xét mục lục của một quyển s. Mục lục này có thể xem là một cây (hình 4.1).



Hình 4.1: Cây mục lục sách

Nút gốc là sách, nó có ba cây con có gốc là C1, C2, C3. Cây con thứ 3 có gốc C3 là một nút đơn độc trong khi đó hai cây con kia (gốc C1 và C2) có các nút con.

Nếu n^1,.., n^k là một chuỗi các nút trên cây sao cho n^i là nút cha của nút n^{i+1}, với i=1..k-1, thì chuỗi này gọi là một đường đi trên cây (hay ngắn gọn là đường đi) từ n^1 đến n^k. Độ dài đường đi được định nghĩa bằng số nút trên đường đi trừ 1. Như vậy độ dài đường đi từ một nút đến chính nó bằng không.

Nếu có đường đi từ nút a đến nút b thì ta nói a là tiền bối (ancestor) của b, còn b gọi là hậu duệ (descendant) của nút a. Rõ ràng một nút vừa là tiền bối vừa là hậu duệ của chính nó. Tiền bối hoặc hậu duệ của một nút khác với chính nó gọi là tiền bối hoặc hậu duệ thực sự. Trên cây nút gốc không có tiền bối thực sự. Một nút không có hậu duệ thực

sự gọi là nút lá (leaf). Nút không phải là lá ta còn gọi là nút trung gian (interior). Cây con của một cây là một nút cùng với tất cả các hậu duệ của nó.

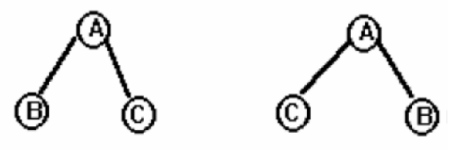
Chiều cao của một nút là độ dài đường đi lớn nhất từ nút đó tới lá. Chiều cao của cây là chiều cao của nút gốc. Độ sâu của một nút là độ dài đường đi từ nút gốc đến nút đó. Các nút có cùng một độ sâu i ta gọi là các nút có cùng một mức i. Theo định nghĩa này thì nút gốc ở mức 0, các nút con của nút gốc ở mức 1.

Ví dụ:

Đối với cây trong hình 5.1 ta có nút C2 có chiều cao 2. Cây có chiều cao 3. nút C3 có chiều cao 0. Nút 2.1 có độ sâu 2. Các nút C1,C2,C3 cùng mức 1.

4.1.2 Thứ tự các nút trong cây

Nếu ta phân biệt thứ tự các nút con của cùng một nút thì cây gọi là cây có thứ tự, thứ tự qui ước từ trái sang phải. Như vậy, nếu kể thứ tự thì hai cây ở hình 5.2 là hai cây khác nhau:



Hình 4.2: Cây có thứ tự khác nhau

Trong trường hợp ta không phân biệt rõ ràng thứ tự các nút thì ta gọi là cây không có thứ tự. Các nút con cùng một nút cha gọi là các nút anh em ruột (siblings). Quan hệ "trái sang phải" của các anh em ruột có thể mở rộng cho hai nút bất kỳ theo qui tắc: nếu A, B là hai anh em ruột và A bên trái B thì các hậu duệ của A là "bên trái" mọi hậu duệ của B.

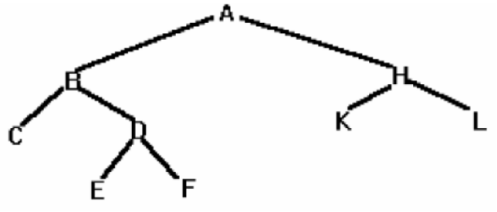
4.1.3 Các thứ tự duyệt cây quan trọng

Duyệt cây là một qui tắc cho phép đi qua lần lượt tất cả các nút của cây mỗi nút đúng một lần, danh sách liệt kê các nút (tên nút hoặc giá trị chứa bên trong nút) theo thứ tự đi qua gọi là danh sách duyệt cây. Có ba cách duyệt cây quan trọng: Duyệt tiền tự (preorder), duyệt trung tự (inorder), duyệt hậu tự (posorder).

- Cây rỗng thì danh sách duyệt cây là rỗng và nó được coi là biểu thức duyệt tiền tự, trung tự, hậu tự của cây.
- Cây chỉ có một nút thì danh sách duyệt cây gồm chỉ một nút đó và nó được coi là biểu thức duyệt tiền tự, trung tự, hậu tự của cây.
- Ngược lại: giả sử cây T có nút gốc là n và có các cây con là T1,..,Tn thì:
    - Biểu thức duyệt tiền tự của cây T là liệt kê nút n kế tiếp là biểu thức duyệt tiền tự của các cây T1, T2, .., Tn theo thứ tự đó.
    - Biểu thức duyệt trung tự của cây T là biểu thức duyệt trung tự của cây T1 kế tiếp là nút n rồi đến biểu thức duyệt trung tự của các cây T2,.., Tn theo thứ tự đó.

- Biểu thức duyệt hậu tự của cây T là biểu thức duyệt hậu tự của các cây T1, T2,.., Tn theo thứ tự đó rồi đến nút n.

Ví dụ: Cho cây như hình 4.3



Hình 4.3: cây nhị phân

Biểu thức duyệt tiền tự: A B C D E F H K L

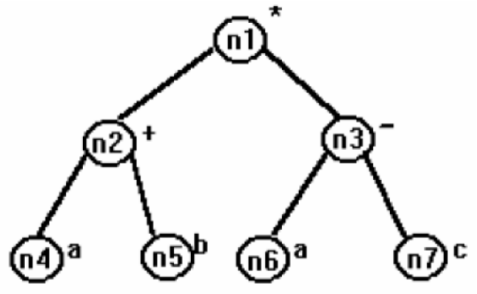
trung tự: C B E D F A K H L

hậu tự: C E F D B K L H A

4.1.4 Cây có nhãn và cây biểu thức

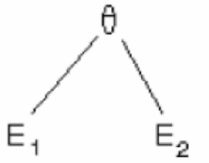
Ta thường lưu trữ kết hợp một nhãn (label) hoặc còn gọi là một giá trị (value) với một nút của cây. Như vậy nhãn của một nút không phải là tên nút mà là giá trị được lưu giữ tại nút đó. Nhãn của một nút đôi khi còn được gọi là khóa của nút, tuy nhiên hai khái niệm này là không đồng nhất. Nhãn là giá trị hay nội dung lưu trữ tại nút, còn khoá của nút có thể chỉ là một phần của nội dung lưu trữ này. Chẳng hạn, mỗi nút cây chứa một record về thông tin của sinh viên (mã SV, họ tên, ngày sinh, địa chỉ,...) thì khoá có thể là mã SV hoặc họ tên hoặc ngày sinh tuỳ theo giá trị nào ta đang quan tâm đến trong thuật giải.

Ví dụ: Cây biểu diễn biểu thức (a+b)*(a-c) như trong hình 5.4.



Hình 4.4: Cây biểu diễn thứ tự (a+b)*(a-c)

- Ở đây n_1, n_2,.., n_7 là các tên nút và *,+,-,a,b,c là các nhãn.
- Qui tắc biểu diễn một biểu thức giải học trên cây như sau:
  - Mỗi nút lá có nhãn biểu diễn cho một giải hạng.
  - Mỗi nút trung gian biểu diễn một giải tử.



Hình 4.5: Cây biểu diễn biểu thức E_1 θ E_2

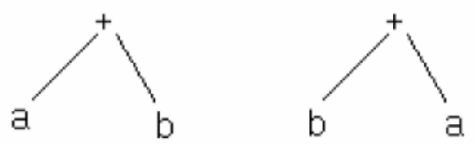
- Giả sử nút n biểu diễn cho một giải tử hai ngôi θ (chẳng hạn + hoặc *), nút con bên trái biểu diễn cho biểu thức E1, nút con bên phải biểu diễn cho biểu thức E2 thì nút n biểu diễn biểu thức E1 θ E2, xem hình 4.5. Nếu θ là phép giải một ngôi thì nút chứa phép giải θ chỉ có một nút con, nút con này biểu diễn cho giải hạng của θ.
- Khi chúng ta duyệt một cây biểu diễn một biểu thức giải học và liệt kê nhãn của các nút theo thứ tự duyệt thì ta có:
  - Biểu thức dạng tiền tố (prefix) tương ứng với phép duyệt tiền tự của cây.
  - Biểu thức dạng trung tố (infix) tương ứng với phép duyệt trung tự của cây.
  - Biểu thức dạng hậu tố (posfix) tương ứng với phép duyệt hậu tự của cây.

Ví dụ: Đối với cây trong hình 4.4 ta có:

- Biểu thức tiền tố: *+ab-ac
- Biểu thức trung tố: a+b*a-c
- Biểu thức hậu tố: ab+ac-*

Chú ý

- Các biểu thức này không có dấu ngoặc.
- Các phép giải trong biểu thức giải học có thể có tính giao hoán nhưng khi ta biểu diễn biểu thức trên cây thì phải tuân thủ theo biểu thức đã cho. Ví dụ biểu thức a+b, với a,b là hai số nguyên thì rõ ràng a+b=b+a nhưng hai cây biểu diễn cho hai biểu thức này là khác nhau (vì cây có thứ tự).



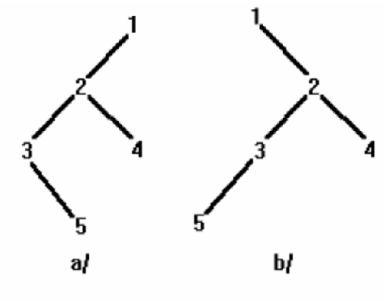
Hình 4.6: Cây biểu diễn biểu thức a+b và b+a

- Chỉ có cây ở phía bên trái của hình 4.6 mới đúng là cây biểu diễn cho biểu thức a+b theo qui tắc trên.
- Nếu ta gặp một dãy các phép giải có cùng độ ưu tiên thì ta sẽ kết hợp từ trái sang phải. Ví dụ a+b+c-d = ((a+b)+c)-d.

4.2 Cây nhị phân (Binary Trees)

4.2.1 Định nghĩa

Cây nhị phân là cây rỗng hoặc là cây mà mỗi nút có tối đa hai nút con. Hơn nữa 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. Ta qui ước vẽ nút con trái bên trái nút cha và nút con phải bên phải nút cha, mỗi nút con được nối với nút cha của nó bởi một đoạn thẳng. Ví dụ các cây trong hình 4.7.



Hình 4.7: Hai cây có thứ tự giống nhau nhưng là hai cây nhị phân khác nhau

Chú ý rằng, trong cây nhị phân, một nút con chỉ có thể là nút con trái hoặc nút con phải, nên có những cây có thứ tự giống nhau nhưng là hai cây nhị phân khác nhau. Ví dụ hình 4.7 cho thấy hai cây có thứ tự giống nhau nhưng là hai cây nhị phân khác nhau. Nút 2 là nút con trái của cây a/ nhưng nó là con phải trong cây b/. Tương tự nút 5 là con phải trong cây a/ nhưng nó là con trái trong cây b/.

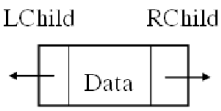
4.2.2 Một số tính chất của cây nhị phân

Gọi h và n lần lượt là chiều cao và số phần tử của cây nhị phân. Ta có các tính chất sau:

- Số nút ở mức **i** $\leq$ $\mathbf{2^{i+1}}$. Do đó số nút tối đa của nó là $\mathbf{2^{h-1}}$
- Số nút tối đa trong cây nhị phân là 2^h-1, hay $\mathbf{n \leq 2^h-1}$. Do đó chiều cao của nó: $\mathbf{n \geq h \geq \log_2(n+1)}$

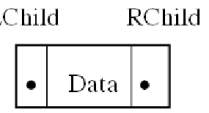
4.2.3 Biểu diễn cây nhị phân

Ta chọn cấu trúc động để biểu diễn cây nhị phân:



Trong đó: Lchild, Rchild lần lượt là các con trỏ chỉ đến nút con bên trái và nút con bên phải. Nó sẽ bằng rỗng nếu không có nút con.

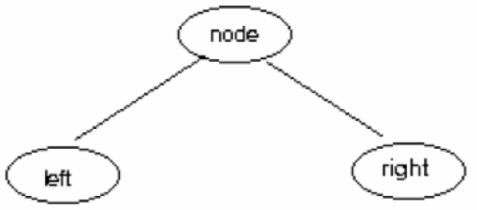
Nút lá có dạng



4.2.4 Duyệt cây nhị phân

Ta có thể áp dụng các phép duyệt cây tổng quát để duyệt cây nhị phân như duyệt theo chiều sâu, duyệt theo chiều rộng (theo mức). Tuy nhiên vì cây nhị phân là cấu trúc cây đặc biệt nên các phép duyệt cây nhị phân cũng đơn giản hơn. Có ba cách duyệt cây nhị phân thường dùng (xem kết hợp với hình 4.8):

- Duyệt tiền tự (Node-Left-Right): duyệt nút gốc, duyệt tiền tự con trái rồi duyệt tiền tự con phải.
- Duyệt trung tự (Left-Node-Right): duyệt trung tự con trái rồi đến nút gốc sau đó là duyệt trung tự con phải.
- Duyệt hậu tự (Left-Right-Node): duyệt hậu tự con trái rồi duyệt hậu tự con phải sau đó là nút gốc.



Hình 4.8

4.2.5 Cài đặt cây nhị phân

I. Cài đặt cấu trúc dữ liệu cây nhị phân

Tương tự cây tổng quát, ta cũng có thể cài đặt cây nhị phân bằng con trỏ bằng cách thiết kế mỗi nút có hai con trỏ, một con trỏ trỏ nút con trái, một con trỏ trỏ nút con phải, trường Data sẽ chứa nhãn của nút.

```
typedef  < kiểu dữ liệu của khoá>    TData;
struct TNode
{
        TData data;
        TNode *left;
        TNode * right;
};
typedef  TNode * Tree;
```

II. Các thao tác cơ bản trên cây nhị phân:

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 cây trỏ tới giá trị NULL.

```
void CreateTree(Tree *T)
{
        (*T)=NULL;
}
```

2. Kiểm tra cây rỗng

```
int EmptyTree(Tree T)
{
    if (T==NULL)
        return 1;
    return 0;
}
```

3. Xác định con trái của một nút

```
//Hàm trả về con trái của nút
Tree LeftChild(Tree n)
{
    if (n != NULL)
        return n->left;
    return NULL;
}
```

4. Xác định con phải của một nút

```
//Hàm trả về con phải của nút
Tree RightChild(Tree n)
{
    if (n!=NULL)
        return n->right;
    return NULL;
}
```

5. Kiểm tra nút lá:

Nếu nút là nút lá thì nó không có bất kỳ một con nào cả nên khi đó con trái và con phải của nó cùng bằng NULL

```
int IsLeaf(Tree n)
{
    if(n!=NULL)
        return(LeftChild(n)==NULL)&&(RightChild(n)==NULL);
    return 0;
}
```

6. Xác định số nút của cây

```
int CN_Nodes(Tree T)
{
    if(EmptyTree(T))
```

```
        return 0;
    return 1+CN_Nodes(LeftChild(T)) + CN_Nodes(RightChild(T));
}
```

III. Các phép duyệt cây:

1. Phép duyệt tiền tự (PreOrder) : N-L-R

```
void PreOrder(Tree T)
{
        if( T != NULL)
        {
            <Xử lý nút>
            PreOrder(T->left);
            PreOrder(T->right);
        }
}
```

2. Phép duyệt trung tự (InOrder): L-N-R

```
void InOrder(Tree T)
{
        if( T != NULL)
        {
            InOrder(T->left);
            <Xử lý nút>
            InOrder(T->right);
        }
}
```

3. Phép duyệt hậu tự (PosOrder):

```
void  PosOrder(Tree T)
{
        if (T!=NULL)
        {
            PosOrder(T->left);
            PosOrder(T->right);
            <Xử lý nút>
        }
}
```

4.3 Cây nhị phân tìm kiếm (BST : BINARY SEARCH TREES)

4.3.1 Định nghĩa cây nhị phân tìm kiếm

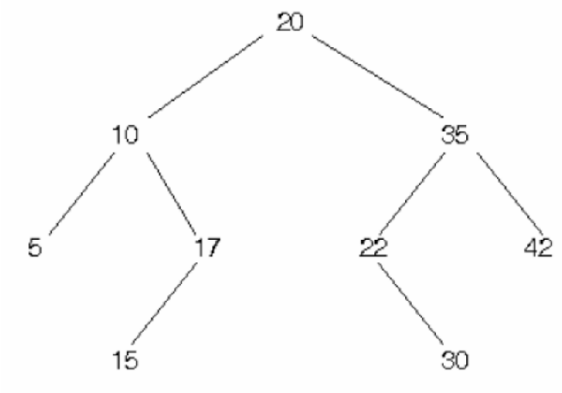
Cây nhị phân tìm kiếm (TKNP) là cây nhị phân mà khoá tại mỗi nút cây lớn hơn khoá của tất cả các nút thuộc cây con bên trái và nhỏ hơn khoá của tất cả các nút thuộc cây con bên phải.

- Lưu ý:

Dữ liệu lưu trữ tại mỗi nút có thể rất phức tạp như là một cấu trúc chẳng hạn, trong trường hợp này khoá của nút được tính dựa trên một 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.

Ví dụ:

Hình 4.9 minh hoạ một cây TKNP có khoá là số nguyên (với quan hệ thứ tự trong tập số nguyên).



Hình 4.9: Ví dụ Cây nhị phân tìm kiếm

Qui ước: Cũng như tất cả các cấu trúc khác, ta coi cây rỗng là cây TKNP

Nhận xét:

- Trên cây TKNP không có hai nút cùng khoá.
- Cây con của một cây TKNP là cây TKNP.
- Khi duyệt trung tự (InOrder) cây TKNP ta được một dãy có thứ tự tăng. Chẳng hạn duyệt trung tự cây trên ta có dãy: 5, 10, 15, 17, 20, 22, 30, 35, 42.

4.3.2 Cài đặt cây nhị phân tìm kiếm

I. Cấu trúc dữ liệu cây nhị phân tìm kiếm:

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. Sẽ không có sự khác biệt nào trong việc cài đặt cấu trúc dữ liệu cho cây TKNP so với cây nhị phân, nhưng tất nhiên, sẽ có sự khác biệt trong các thuật giải thao tác trên cây TKNP như tìm kiếm, thêm hoặc xoá một nút trên cây TKNP để luôn đảm bảo tính chất cuả cây TKNP.

Một cách cài đặt cây TKNP thường gặp là cài đặt bằng con trỏ. Mỗi nút của cây như là một cấu trúc có ba trường: một trường chứa khoá, hai trường kia là hai con trỏ trỏ đến hai nút con (nếu nút con vắng mặt ta gán con trỏ bằng NULL)

Khai báo như sau :

```
typedef  <kiểu dữ liệu của khoá>  KeyType;

struct BSNode
{
        KeyType key;
        BSNode *left;
        BSNode *right;
}

typedef   BSNode  *BSTree;
```

II. Các thao tác cơ bản trên cây nhị phân tìm kiếm

1. Tạo nút mới

```
//Input x
//Output : NULL ; khong thanh cong
//           p; tro den nut vua tao, neu thanh cong
BSNode  *CreateNode(KeyType x)
{
        BSNode *p = new BSNode;
        if (p != NULL)
        {
                p->key = x;
                p->left = NULL;
                p->right = NULL;
        }
        return p;
}
```

2. Khởi tạo cây TKNP rỗng

Ta cho con trỏ quản lý nút gốc (Root) của cây bằng NULL.

```
void CreateBST(BSTree &root)
{
        root = NULL;
}
```

3. Tìm kiếm một nút có khóa cho trước trên cây TKNP

Để tìm kiếm 1 nút có khoá x trên cây TKNP, ta tiến hành 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.
- Nếu x bằng khoá của nút gốc thì thuật giải dừng và ta đã tìm được nút chứa khoá x.
- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) việc tìm khoá x trên cây con bên phải.

- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) việc tìm khoá x trên cây con bên trái.

Ví dụ:

Tìm nút có khoá 30 trong cây ở trong hình 4.9 :

- 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 30, 30 = 30 vậy đến đây thuật giải dừng và ta tìm được nút chứa khoá cần tìm.

a. Cài đặt thuật giải tìm kiếm (đệ quy)

```
//Tim kien nut co khoa x cho truoc
//Input : x, root
//Output : p, con tro tro den nut chua x neu co
//                 : NULL, neu khong co

BSTree Search(KeyType x, BSTree root)
{
        if (root != NULL)
        {
                if (root->key == x) //Tim thay x
                        return root;
                else
                if (root->key < x)
                        return Search(x, root->right); //tim x trong cay con phai
                else
                        return Search(x, root->left); //tim x trong cay con trai
        }
        return NULL;
}
```

b. Cài đặt thuật giải tìm kiếm (dạng lặp)

Thuật giải tìm kiếm dạng lặp, trả về con trỏ chứa dữ liệu cần tìm và đồng thời giữ lại nút cha của nó nếu tìm thấy, ngược lại trả về rỗng.

```
BSTree SearchLap(BSTree root, KeyType  x, BSTree &parent)
{
        BSTree  locPtr = root;
        parent = NULL;
        while (locPtr != NULL)
```

```
        {
                if (x == locPtr -> key)
                        return  locPtr;
                else
                {
                        parent = locPtr;
                        if (x > locPtr->key)
                                locPtr = locPtr->rChild;
                         else
                                locPtr = locPtr->lChild;
                }
        }
        return NULL;
}
```

Nhận xét:

Thuật giải này sẽ rất hiệu quả về mặt thời gian nếu cây TKNP được tổ chức tốt, nghĩa là cây tương đối "cân bằng".

4. Thêm một nút có khóa cho trước vào cây TKNP

Theo định nghĩa Cây nhị phân tìm kiếm ta thấy trên Cây nhị phân tìm kiếm không có hai nút có cùng một khoá. Do đó nếu ta muốn thêm một nút có khoá x vào cây TKNP thì trước hết ta phải tìm kiếm để xác định có nút nào chứa khoá x chưa. Nếu có thì thuật giải kết thúc (không làm gì cả!). Ngược lại, sẽ thêm một nút mới chứa khoá x này. Việc thêm một khoá vào cây TKNP là việc tìm kiếm và thêm một nút, tất nhiên, phải đảm bảo cấu trúc cây TKNP không bị phá vỡ. Thuật giải cụ thể như sau:

Ta tiến hành từ nút gốc bằng cách so sánh khóa cuả nút gốc với khoá x.

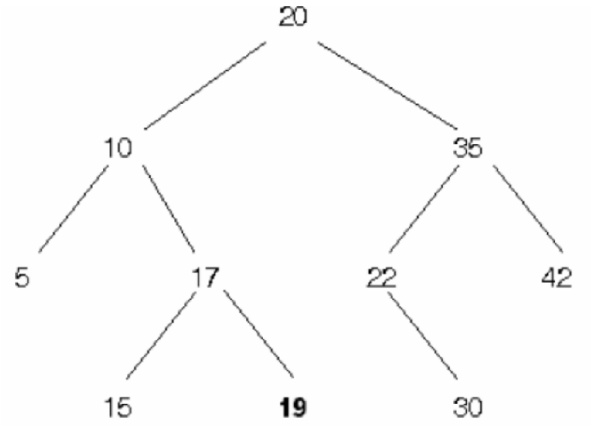
- Nếu nút gốc bằng NULL thì khoá x chưa có trên cây, do đó ta thêm một nút mới chứa khoá x.
- Nếu x bằng khoá của nút gốc thì thuật giải dừng, trường hợp này ta không thêm nút.
- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) thuật giải này trên cây con bên phải.
- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) thuật giải này trên cây con bên trái.

Ví dụ:

Thêm khoá 19 vào cây ở trong hình 4.10

- 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 là10.

- 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, xem hình 4.10



Hình 4.10: Thêm khoá 19 vào cây hình 5.10

a. Cài đặt thuật giải thêm khóa x vào cây BST (đệ quy)

```
//Input : x, root;
//Output : -1; khong thanh cong - khong du bo nho
//          0; khong thanh cong - x da co san trong cay
//          1; Thanh cong; cay BST moi co them nut chua x

int InsertNode(BSTree &root, KeyType x)
{
        //Cay khac rong
        if (root != NULL)
        {
                if (root->key == x)
                        return 0; // x da co san
                if (root->key >  x)
                        return  insertNode(root->left, x);
                else
                        return insertNode(root->right, x);
        }//root == NULL

        root = CreateNode(x);
        if (root == NULL)
                return -1;
        return 1;
}
```

b. Cài đặt thuật giải thêm khóa x vào cây BST (lặp)

```
int InsertNodeLap(BSTree &root, KeyType x)
{
	BSTree locPtr, parent;
	if (SearchLap(root, x, parent))
	{
		cout << "\nda co ptu  "<<Item<<" trong cây !" ;
		return -1;
	}
	else
	{
		if (locPtr=GetNode(x))==NULL)
			return 0;
		LocPtr->Key = Item;
		LocPtr->LChild = NULL;
		LocPtr->RChild = NULL;
		if  (Parent == NULL)
			Root = LocPtr;   // cây rỗng
		 else
			if (Item < Parent->Data)
				Parent->LChild =  LocPtr;
			else
				Parent->RChild = LocPtr;
		return 1;
	 }
}
```

5. Nhập dữ liệu cho BST : từ tập tin, từ bàn phím

Để nhập dữ liệu cho BST, ta lặp thêm x vào BST (xuất phát từ cây rỗng). Hảm sau đây chuyển dữ liệu tập tin filename vào cây BST root .
- trường hợp kiểu KeyType là kiểu só nguyên int
- Filename là tập tin chứa các số nguyên.

```
//Input  : filename
//Output : 0; khong thanh cong
//           1; thanh cong
//           root;

int File_BST(BSTree &root, char *filename)
{
	ifstream in(filename);
	if (!in)
		return 0;

	KeyType x;
```

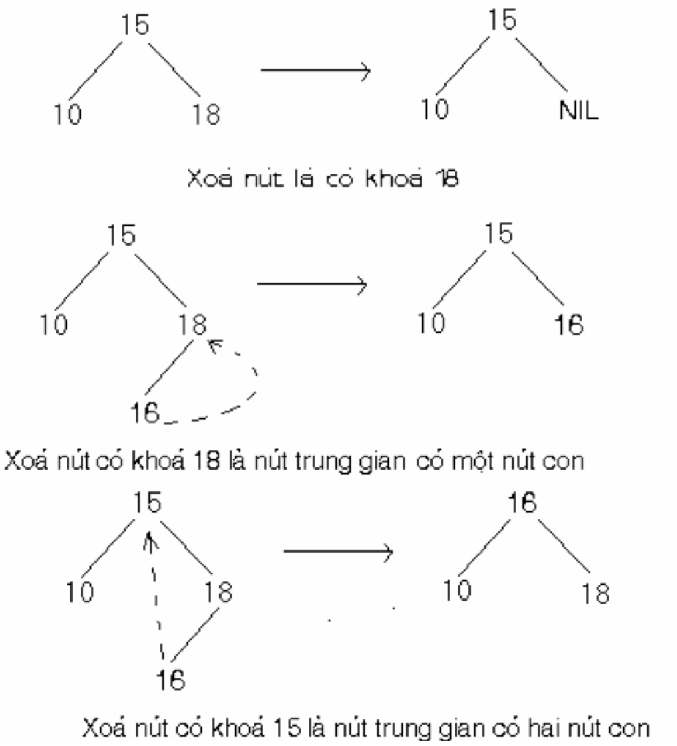
```
        int kq;
        Create_BST(root);
        in >> x;
        kq = insertNode(root, x);
        if (kq == 0 || kq == -1)
                return 0;
        while (!in.eof())
        {
                in >> x;
                kq = insertNode(root, x);
                if (kq == 0 || kq == -1)
                        return 0;
        }
        in.close();
        return 1;
}
```

6. Xuất cây BST ra màn hình :

Có thể sử dụng các phép duyệt cây : tiền tự, trung tự, hậu tự, và sử dụng thao tác <xuất dữ liệu của nút ra màn hình > thay thể < xử lý nút>.

7. Xóa một nút có khóa cho trước ra khỏi cây TKNP.

Giả sử ta muốn xoá một nút có khoá x, trước hết ta phải tìm kiếm nút chứa khoá x trên cây (hình 4.11)

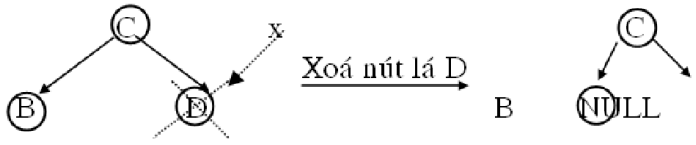


Hình 4.11

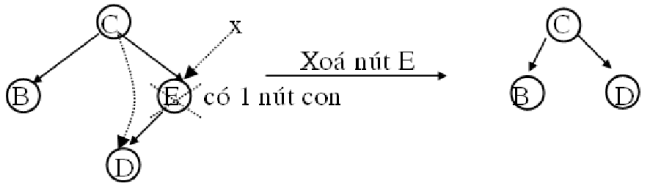
Việc xoá một nút như vậy, tất nhiên, ta phải bảo đảm cấu trúc cây TKNP không bị phá vỡ.

- Nếu không tìm thấy nút chứa khoá x thì thuật giải kết thúc.
- Nếu tìm gặp nút N có chứa khoá x, ta có ba trường hợp sau

  - Nếu N là lá ta thay nó bởi NULL.



  - N chỉ có một nút con ta thay nó bởi nút con của nó.

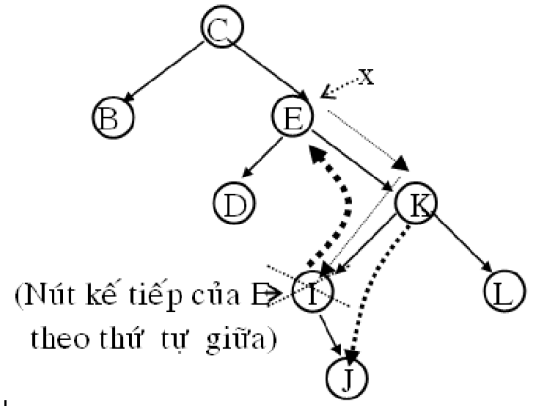


- N có hai nút con :

- ta thay nó 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).

- Trong thuật giải sau, ta thay x bởi khoá của nút cực trái của cây con bên phải rồi ta xoá nút cực trái này như hình 4.12.

  Việc xoá nút cực trái của cây con bên phải sẽ rơi vào một trong hai trường hợp trên.



Hình 4.12

7.1 Xoa nut cuc trai cua cay con trai của root

```
//Ham Huy nut co gia tri nho nhat cua cay con trai cua root
//Input root
//Output : root (da xoa duoc nut co gia tri nho nhat cua cay con trai root
//          x; khoa cua nut bi xoa

KeyType DeleteMin(BSTree &root)
{
        KeyType k;
        if (root->left == NULL)
        {
                k = root->key;
                root = root->right;
                return k;
        }
        else
                return DeleteMin(root->left);
}
```

7.2 Xóa nút có khoá cho trước trên cây TKNP

a. Dạng đệ quy
//Input : x, root

```
//Output :  cay BST ket qua
int DeleteNode(KeyType  x, BSTree &root)
{
        if (root != NULL)
        {
                if (x < root->key)
                        DeleteNode(x, root->left);
                else
                if (x  > root->key)
                        DeleteNode(x, root->right);
                else //x  == root->key
                if ((root->left == NULL) && (root->right == NULL))
                                //khong co con trai, khong co con phai
                        root = NULL;
                else
                if (root->left == NULL) //co 1 con : con phai, khong co con trai
                        root = root->right;
                else
                if (root->right == NULL) //co 1 con : con trai, khong co con phai
                        root = root->left;
                else //co ca 2 con trai, phai
                        root->key = DeleteMin(root->right);
                return 1;
        }
        return 0;
}
```

b. Dạng lặp :

```
int DeleteNode (BSTree  &root, KeyType x)
{
        BSTree x, parent, xSucc, subTree;
        if((x=SearchLap(root,x,parent)) == NULL)
                return 0;   //không thấy x
        else
        {
                if((x->left!=NULL)&&(x->right != NULL))
                // nút có hai con
                {
                        xSucc =  x->right;
                        parent = x;
                        while (xSucc->left != NULL)
                        {
                                parent = xSucc;
                                xSucc =  xSucc->left;
                        }
                        x->key = xSucc->Key;
```

```
                x = xSucc;
            }
            //đã đưa nút 2 con về nút có tối đa 1 con
            subTree =  x->left;
            if  (subTree ==  NULL)
                subTree =  x->right;
            if  (parent == NULL)
                root =  subTree; // xóa nút gốc
            else
                if (parent->left == x)
                    parent->left = subTree;
            else
                parent->right = bubTree;
            delete  x;
            return 1;
        }
}
```

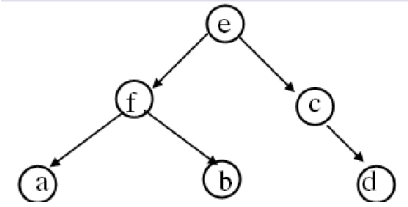
4.4. Cây nhị phân tìm kiếm cân bằng (Cây AVL)

4.4.1 Cây nhị phân cân bằng hoàn toàn

1. Định nghĩa

Cây nhị phân cân bằng hoàn toàn (CBHT) là cây nhị phân mà đối với mỗi nút của nó, số nút của cây con trái chênh lệch không quá 1 so với số nút của cây con phải như hình 4.13.

Ví dụ:



Hình 4.13

2. Xây dựng cây nhị phân cân bằng hoàn toàn

```
Tree  CreateTreeCBHT(int  n)
{
        Tree Root;
        int nl, nr;
        KeyType x;
        if (n<=0)
                return NULL;
```

```
        nl = n/2; nr = n-nl-1;
        Input(x);//nhập phần tử x
        if ((Root =CreateNode()) == NULL)
                return NULL;
        Root->Key = x;
        Root->left = CreateTreeCBHT(nl);
        Root->right = CreateTreeCBHT(nr);
        return Root;
}
```

Chú ý:

- Nếu cây cân đối thì việc tìm kiếm sẽ nhanh. Đối với cây cân bằng hoàn toàn, trong trường hợp xấu nhất ta chỉ phải tìm qua $\log_2 n$ phần tử (n là số nút trên cây).
- Một cây rất khó đạt được trạng thái cân bằng hoàn toàn và cũng rất dễ mất cân bằng khi thêm hay hủy các nút trên cây.
- Cân bằng lại cây!!! Chi phí lớn vì phải thao tác trên toàn bộ cây.

4.4.2 Định nghĩa cây nhị phân tìm kiếm cân bằng

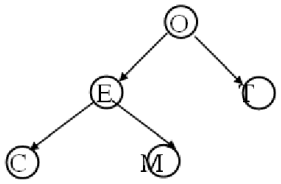
Trên cây nhị phân tìm kiếm BST có n phần tử mà là cây CBHT, phép tìm kiếm một phần tử trên nó sẽ thực hiện rất nhanh: trong trường hợp xấu nhất, ta chỉ cần thực hiên $\log_2 n$ phép so sánh. Nhưng cây CBHT có cấu trúc kém ổn định trong các thao tác cập nhật cây, nên nó ít được sử dụng trong thực tế. Vì thế, người ta tận dụng ý tưởng cây cân bằng hoàn toàn để xây dựng một cây nhị phân tìm kiếm có trạng thái cân bằng yếu hơn, nhưng việc cân bằng lại chỉ xảy ra ở phạm vi cục bộ đồng thời chi phí cho việc tìm kiếm vẫn đạt ở mức $O(\log_2 n)$. Đó là cây tìm kiếm cân bằng.

- Định nghĩa:

Cây nhị phân tìm kiếm gọi là cây nhị phân tìm kiếm cân bằng (gọi tắt là cây AVL) nếu tại mỗi nút của nó, độ cao của cây con trái và độ cao của cây con phải chênh lệch nhau không quá 1.

Rõ ràng một cây nhị phân tìm kiếm cân bằng hoàn toàn là cây cân bằng, nhưng điều ngược lại là không đúng. Chẳng hạn cây nhị phân tìm kiếm trong ví dụ sau là cân bằng nhưng không phải là cân bằng hoàn toàn.

Ví dụ: Cây AVL như hình 4.14



Hình 4.14

Cây cân bằng AVL vẫn thực hiện việc tìm kiếm nhanh tương đương cây cân bằng hoàn toàn và vẫn có cấu trúc ổn định hơn hẳn cây cân bằng.

4.4.3 Chỉ số cân bằng :

- Định nghĩa:

Chỉ số cân bằng (CSCB) của một nút p là hiệu của chiều cao cây con phải và cây con trái của nó.

- Kí hiệu:

- $h_l(p)$ hay h_l là chiều cao của cây con trái của p.
- $h_r(p)$ hay h_r là chiều cao của cây con phải của p.
- EH = 0, RH = 1, LH = -1

CSCB(p) = EH	⇔	$h_R(p) = h_L(p)$:2 cây con cao bằng nhau
CSCB(p) = RH	⇔	$h_R(p) > h_L(p)$: cây lệch phải
CSCB(p) = LH	⇔	$h_R(p) < h_L(p)$: cây lệch trái

Với mỗi nút của cây AVL, ngoài các thuộc tính thông thường như cây nhị phân, ta cần lưu ý thêm thông tin về chỉ số cân bằng trong cấu trúc của một nút.

4.4.4 Cài đặt cây AVL

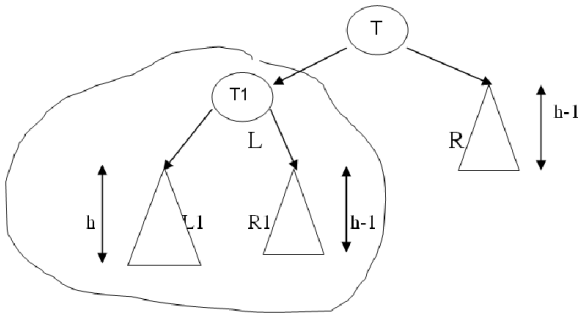
```
typedef    .....   ElementType;  /* Kiểu dữ liệu của nút */

struct  AVLTN
{
        ElementType data;
        int  balfactor; //Chỉ số cân bằng
        AVLTN * lChild, *rChild;
}
typedef  AVLTN  AVLTreeNode;

typedef   AVLTreeNode  *AVLTree;
```

4.4.5 Các trường hợp mất cân bằng

Việc thêm hay hủy một nút trên cây AVL có thể làm cây tăng hay giảm chiều cao, khi đó ta cần phải cân bằng lại cây. Để giảm tối đa chi phí cân bằng lại cây, ta chỉ cân bằng lại cây AVL ở phạm vi cục bộ.

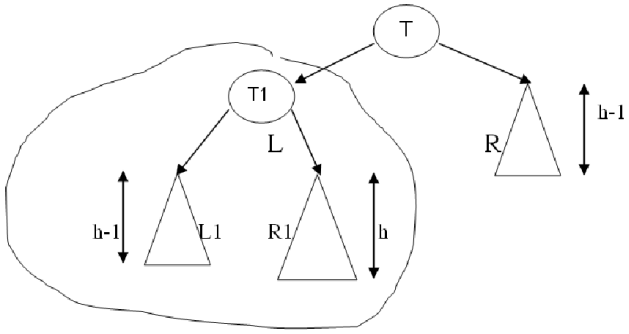
Ngoài các thao tác thêm và hủy đối với cây cân bằng, ta còn có thêm thao tác cơ bản là cân bằng lại cây AVL trong trường hợp thêm hoặc hủy một nút của nó. Khi đó độ lệch giữa chiều cao cây con phải và trái sẽ là 2. Do đó trường hợp cây lệch trái và phải tương ứng là đối xứng nhau, nên ta chỉ xét trường hợp cây AVL lệch trái.

Trường hợp a: cây con T1 lệch trái (hình 4.15)



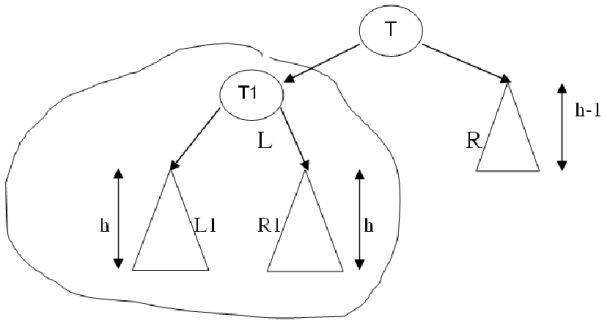
Hình 4.15

Trường hợp b: cây con T1 lệch phải (hình 4.16)



Hình 4.16

Trường hợp c: cây con T1 không lệch (hình 5.17)



Hình 4.17

4.4.6 Phép quay:

Để thực hiện cân bằng cây, ta cần một thao tác là quay. Có thể quay 1 lần (quay đơn), nhiều khi phải quay 2 lần (quay kép)

- Định nghĩa:

Phép quay là cách tái sắp xếp các nút, chúng được thiết kế làm các công việc sau:

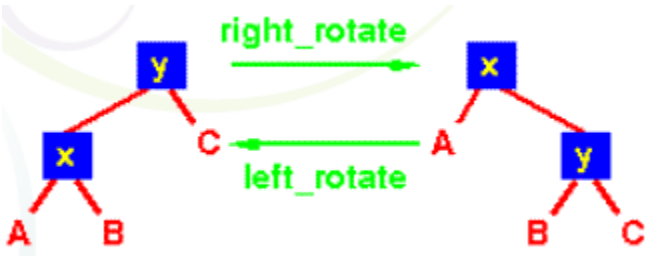
- ☐ Nâng một số nút lên và hạ một số nút khác xuống để giúp cân bằng cây.
- ☐ Bảo đảm những tính chất của cây tìm kiếm nhị phân không bị vi phạm.

Thuật ngữ quay có thể bị hiểu nhầm. Thực ra quay không có nghĩa là các nút bị quay mà là chỉ sự thay đổi quan hệ giữa chúng.

- Hoạt đông phép quay:

- ■ Một nút (Y) được chọn làm "đỉnh" của phép quay. Nếu ta thực hiện một phép quay qua phải từ nút đỉnh này (Y), phép quay phải hoạt động như sau:
  - ☐ Nút "đỉnh" (Y) này sẽ di chuyển xuống dưới và về bên phải, vào vị trí của nút con bên phải của nó (C).
  - ☐ Nút con bên trái của nó (X) sẽ đi lên để chiếm lấy vị trí cũ của nó (X trở thành đỉnh – gốc) – (Y trở thành con phải của X)
  - ☐ Con trái cũ của X (là A) tiếp tục là con trái của X.
  - ☐ Con phải cũ của của X là B (>X và <Y) nên làm con trái của Y.
  - ☐ Con phải cũ của của Y là C trở thành con phải của Y

(dùng tính chất BST)



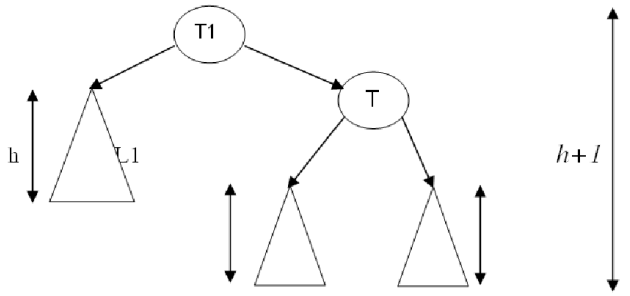
- ■ Kết quả của 2 phép quay thứ tự duyệt cây trong phép duyệt không thay đổi: A x B y C
- ■ Ta phải đảm bảo là nếu làm phép quay phải, nút đỉnh phải có nút con trái. Nếu không chẳng có gì để quay vào điểm đỉnh.

Tương tự, nếu làm phép quay trái, nút đỉnh trái có nút con phải.

4.4.7 Cân bằng lại cây

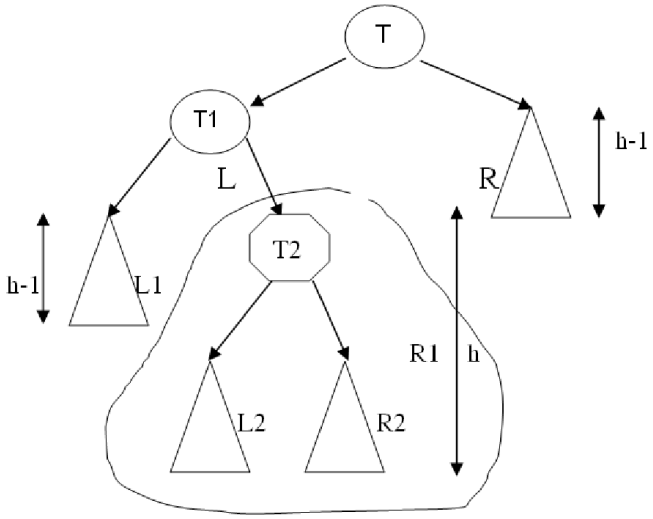
- Cân bằng lại trường hợp a (hình 4.15):

  Ta cân bằng lại bằng phép quay đơn left-left ta được:



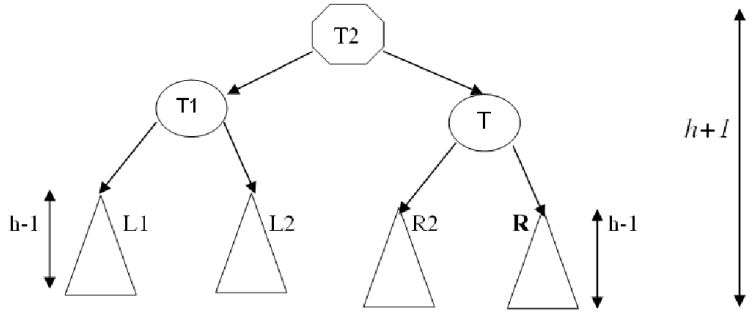
Hình 4.18

- Cân bằng lại trường hợp b (hình 4.16):



Hình 4.19

Cân bằng lại bằng phép quay kép left-right, ta có kết quả như sau (hình 5.21):



Hình 4.20

- Cài đặt

```
        //Phép quay đơn Left – Left
void RotateLL(AVLTree &T)
{
        AVLTree T1 = T->Lchild;
        T->Lchild = T1->Rchild;
        T1->Rchild = T;
        switch (T1->Balfactor)
        {
                case LH: T->Balfactor = EH;
                     T1->Balfactor = EH; break;
                case EH: T->Balfactor = LH;
                     T1->Balfactor = RH; break;
        }
        T = T1;
        return ;
}

        // Phép quay đơn Right – Right
void RotateRR (AVLTree &T)
{
        AVLTree T1 = T->Rchild;
        T->Rchild = T1->Lchild;
        T1->Lchild = T;
        switch (T1->Balfactor)
        {
                case RH: T->Balfactor = EH;
                     T1->Balfactor = EH; break;
                case EH: T->Balfactor = RH;
                     T1->Balfactor = LH; break;
        }
        T = T1;
        return ;
}
        //Phép quay kép Left – Right
void RotateLR(AVLTree &T)
{
        AVLTree T1 = T->Lchild, T2 = T1->Rchild;
        T->Lchild = T2->Rchild; T2->Rchild = T;
        T1->Rchild = T2->Lchild; T2->Lchild = 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;
        return ;
}
        //Phép quay kép Right-Left
void RotateRL(AVLTree &T)
{
        AVLTree T1 = T->RLchild, T2 = T1->Lchild;
        T->Rchild = T2->Lchild; T2->Lchild = T;
        T1->Lchild = T2->Rchild; T2->Rchild = T1;
        switch (T2->Balfactor)
        {
            case LH: T->Balfactor = EH;
                T1->Balfactor = RH; break;
            case EH: T->Balfactor = EH;
                T1->Balfactor = EH; break;
            case RH: T->Balfactor = LH;
                T1->Balfactor = EH; break;
        }
        T2->Balfactor = EH;
        T = T2;
        return ;
}
```

Cài đặt các thao tác cân bằng lại

```
        //Cân bằng lại khi cây bị lệch trái
int LeftBalance(AVLTree &T)
{
        AVLTree T1 = T->Lchild;
        switch (T1->Balfactor)
        {
            case LH: RotateLL(T);
                    return 2; //cây T không bị lệch
            case EH: RotateLL(T);
                    return 1;//cây T bị lệch phải
        case RH: RotateLR(T); return 2;
        }
        return  0;
}
        //Cân bằng lại khi cây bị lệch phải
int RightBalance(AVLTree &T)
```

```
{
        AVLTree T1 = T->Rchild;
        switch (T1->Balfactor)
        {
                case LH: RotateRL(T);
                        return 2; //cây T không lệch
                case EH: RotateRR(T);
                        return 1; //cây T lệch trái
        case RH: RotateRR(T); return 2;
        }
        return  0;
}
```

4.4.8 Chèn 1 phần tử vào cây AVL

Việc chèn một phần tử vào cây AVL xảy ra tương tự như trên cây nhị phân tìm kiếm. Tuy nhiên sau khi chèn xong, nếu chiều cao của cây thay đổi tại vị trí thêm vào, ta cần phải ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng hay không. Nếu có, ta chỉ cần phải cân bằng lại ở nút này.

```
AVLTree CreateAVL()
{
        AVLTree Tam= new AVLTreeNode;
        if (Tam == NULL)
                cout << "\nLỗi !";
        return Tam;
}

int  InsertNodeAVL( AVLTree &T, ElementType x)
{
        int  Kqua;
        if (T)
        {
                if(T->Data==x)
                        return 0; //Đã có nút trên cây
                if (T-> Data >  x)
                {
                        //chèn nút vào cây con trái
                        Kqua = InsertNodeAVL(T->Lchild,x);
                        if (Kqua < 2) return  Kqua;
                        switch  (T->Balfactor)
                        {
                                case LH: LeftBalance(T);
                                        return 1;//T lệch trái
                                case EH: T->Balfactor=LH;
                                        return 2;//T không lệch
```

```
                        caseRH:T->Balfactor=EH;
                                return 1;//T lệch phải
                }
            }
            else   // T-> Data <  x
            {
                Kqua= InsertNodeAVL(T->Rchild,x);
                if (Kqua < 2) return  Kqua;
                switch  (T->Balfactor)
                {
                        case LH: T->Balfactor = EH;
                                return 1;
                        case EH:T->Balfactor=RH;
                                return 2;
                        case RH: RightBalance(T);
                                return 1;
                }
            }
            else //T==NULL
            {
                if ((T = CreateAVL()) == NULL)
                        return –1;
                T->Data = x;
                 T->Balfactor = EH;
                T->Lchild  = T->Rchild = NULL;
                return 2;
            }
        }
```

4.4.9 Xóa một phần tử ra khỏi cây AVL

Việc xóa một phần tử ra khỏi cây AVL diễn ra tương tự như đối với cây nhị phân tìm kiếm, chỉ khác là sau khi hủy, nếu cây AVL bị mất cân bằng, ta phải cân bằng lại cây. Việc cân bằng lại cây có thể xảy ra phản ứng dây chuyền.

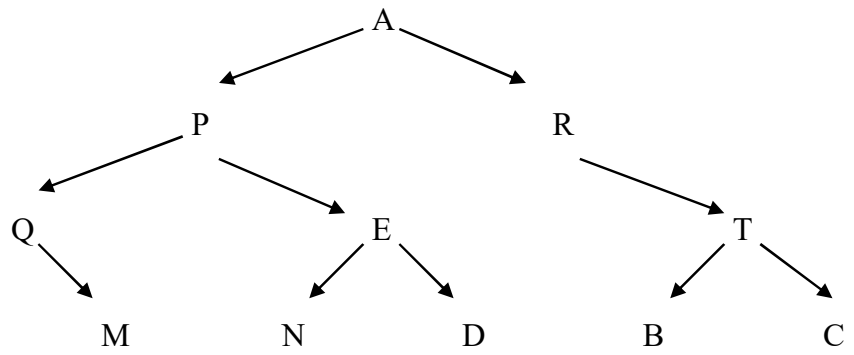
```
int DeleteAVL(AVLTree &T, ElementType x)
{
        int Kqua;
        if (T== NULL) return 0; // không có x trên cây
        if  (T-> Data >  x)
        {
                Kqua = DeleteAVL(T->Lchild,x);// tìm và xóa x trên cây con trái của T
                if (Kqua < 2) return  Kqua;
                switch  (T->Balfactor)
                {
                        case LH: T->Balfactor = EH;
                                return 2; //trước khi xóa T lệch trái
```

```
                case EH: T->Balfactor = RH;
                        return 1;//trước khi xóa T không lệch
        case RH: return RightBalance(T);
                // trước khi xóa T lệch phải
        }
}
else if (T-> Data <  x)
{
        Kqua = DeleteAVL(T->Rchild,x);// tìm và xóa x trên cây con trái của T
        if (Kqua < 2) return  Kqua;
        switch  (T->Balfactor)
        {
                case LH: return LeftBalance(T);//trước khi xóa T lệch trái
                case EH: T->Balfactor = LH;
                        return 1;       //trước khi xóa T không lệch
                 case RH: T->Balfactor = EH;
                return 2; //trước khi xóa T lệch phải
        }
}
else //T->Data== x
{
        AVLTree p = T;
        if (T->Lchild == NULL)
        {
                T = T->Rchild; Kqua = 2;
        }
        else    if  (T->Rchild == NULL)
        {
                T = T->Lchild; Kqua = 2;
        }
        else // T có hai con
        {
                Kqua = TimPhanTuThayThe(p,T->Rchild);
                //Tìm phần tử thay thế P để xóa trên nhánh phải cuả T
                if (Kqua < 2) return Kqua;
                switch (T->Balfactor)
                {
                        case  LH: return LeftBalnce(T);
                        case  EH: T->Balfactor = LH;
                                        return 2;
                        case  RH: T->Balfactor = EH;
                                        return 2;
                }
        }
        delete p;
        return Kqua;
}
```

```
}
        // Tìm phần tử thay thế
int TimPhanTuThayThe(AVLTree  &p, AVLTree  &q)
{
        int Kqua;
        if (q->Lchild)
        {
                Kqua = TimPhanTuThayThe(p, q->Lchild);
                if (Kqua < 2) return Kqua;
                switch (q->Balfactor)
                {
                        case  LH: q->Balfactor = EH;
                                  return 2;
                        case  EH: q->Balfactor = RH;
                                  return 1;
                        case  RH: return RightBalance(q);
                }
        }
        Else
        {
                p->Data = q->Data;
                p = q;
                q = q->Rchild;
                return 2;
        }
}
```

Bài tập

Xuất ra theo thứ tự: giữa, đầu, cuối các phần tử trên cây nhị phân sau:



Tìm cây nhị phân thỏa đồng thời hai điều kiện kết xuất sau:

- Theo thứ tự đầu NLR của nó là dãy ký tự sau:

  A, B, C, D, E, Z, U, T, Y
- Theo thứ tự giữa LNR của nó là dãy ký tự sau:

  D, C, E, B, A, U, Z, T, Y

Biểu diễn mỗi biểu thứ số học dưới đây trên cây nhị phân, từ đó rút ra dạng biểu thức hậu tố của chúng:

- a/(b*c)
- $a^5 + 4a^3 – 3a^2 + 7$
- (a + b) * (c - d)
- S^{a+b}

Viết thuật giải vàchương trình:

- Chuyển một biểu thức số học ký hiệu lên cây nhị phân (có kiểm tra biểu thức đã cho có hợp cú pháp không ?)
- Xuất ra biểu thức số học đó dưới dạng: trung tố, hậu tố, tiền tố

Xây dựng Cây nhị phân tìm kiếm BST từ mỗi bộ mục dữ liệu đầu vào như sau:

- 1,2,3,4,5
- 5,4,3,2,1
- fe, cx, jk, ha, ap, aa, by, my, da
- 8,9,11,15,19,20,21,7,3,2,1,5,6,4,13,10,12,17,16,18. Sau đó xóa lần lượt các nút sau: 2,10,19,8,20

Viết chương trình với các chức năng sau:

- Nhập từ bàn phím các số nguyên vào một cây nhị phân tìm kiếm (BST) mà nút gốc được trỏ tới bởi con trỏ Root.

- Xuất các phần tử trên cây BST trên theo thứ tự: đầu, giữa, cuối.
- Tìm và xóa (nếu có thể) phần tử trên cây Root có dữ liệu trùng với một mục dữ liệu Item cho trước được nhập từ bàn phím.
- Sắp xếp n mục dữ liệu (được cài đặt bằng DSLK) bằng phương pháp cây nhị phân tìm kiếm BSTSort.

Yêu cầu: viết các thao tác trên bằng 2 phương pháp đệ quy và lặp

Tương tự bài 5 nhưng trong mỗi nút có thêm trường parent để trỏ tới cha của nó.

Cho cây nhị phân T. Viết chương trình chứa các hàm có tác dụng xác định:

- Tổng số nút của cây.
- Số nút của cây ở mức k.
- Số nút lá.
- Chiều cao của cây.
- Kiểm tra xem cây T có phải cây cân bằng hoàn toàn hay không?
- Số nút có đúng hai con khác rỗng
- Số nút có đúng một con khác rỗng
- Số nút có khóa nhỏ hơn x trên cây nhị phân hoặc cây BST
- Số nút có khóa lớn hơn x trên cây nhị phân hoặc cây BST
- Duyệt theo chiều rộng
- Duyệt theo chiều sâu
- Đảo nhánh trái và phải của một cây nhị phân.

Viết chương trình thực hiện các thao tác cơ bản trên cây AVL: chèn một nút, xóa một nút, tạo cây AVL, hủy cây AVL.

Viết chương trình cho phép tạo, thêm, bớt, tra cứa, sửa chữa từ điển.

TÀI LIỆU THAM KHẢO

[1] ALFRED V. AHO & JOHN E.HOPCROFT & JOHN D. ULMANN (1983), "Data structures and algorithms", Addison Wesley.

[2] DONALD E. KNUTH (1973), "The art of Computer programming", volume 3, "Sorting and Searching", Addison Wesley.

[3] LARRY NYHOFF & SANFORD LESSTMA, "Lập trình nâng cao bằng Passcal với các cấu trúc dữ liệu"
(bản dịch: Lê Minh Trung (1997))

[4] NIKLAUS WIRTH (1985), "Algorithms + data structures = Programs", Prentice-Hall INC.
(Bản dịch của Nguyễn Quốc Cường (1993))

[5] S.E.GOODMAN & S.T. HEDETNIEMI (1977), "Introduction to the design and analysis of algorithms", Mcgraw-Hill.

[6] NGUYỄN THỊ THANH BÌNH & TRẦN TUẤN MINH, giáo trình "Cấu trúc dữ liệu và thuật giải 1", 2010, Khoa Công nghệ thông tin, Đại học Đà Lạt

[7] NGUYỄN THỊ THANH BÌNH & NGUYỄN VĂN PHÚC, giáo trình "Cấu trúc dữ liệu và thuật giải 2", 2010, Khoa Công nghệ thông tin, Đại học Đà Lạt