MỤC LỤC

Lời nói đầu

Cấu trúc dữ liệu và Giải thuật là các lĩnh vực nghiên cứu gắn liền với nhau và là một trong những lĩnh vực nghiên cứu lâu đời của khoa học máy tính. Hầu hết các chương trình được viết ra, chạy trên máy tính, dù lớn hay nhỏ, dù đơn giản hay phức tạp, đều phải sử dụng các cấu trúc dữ liệu tuân theo các trình tự, cách thức làm việc nào đó, chính là các giải thuật. Việc hiểu biết về các cấu trúc dữ liệu và thuật toán cho phép các lập trình viên, các nhà khoa học máy tính có nền tảng lý thuyết vững chắc, có nhiều lựa chọn hơn trong việc đưa ra các giải pháp cho các bài toán thực tế. Vì vậy việc học tập môn học Cấu trúc dữ liệu và Giải thuật là một điều quan trọng.

Tài liệu này dựa trên những kinh nghiệm và nghiên cứu mà tác giả đã đúc rút, thu thập trong quá trình giảng dạy môn học Cấu trúc dữ liệu và giải thuật tại khoa Công nghệ Thông tin, Đại học Hàng hải Việt nam, cùng với sự tham khảo của các tài liệu của các đồng nghiệp, các tác giả trong và ngoài nước, từ điển trực tuyến Wikipedia. Với năm chương được chia thành các chủ đề khác nhau từ các khái niệm cơ bản cho tới thuật toán sắp xếp, tìm kiếm, cấu trúc dữ liệu cơ bản như ngăn xếp, hàng đợi, danh sách liên kết, cây cân bằng … hy vọng sẽ cung cấp cho các em sinh viên, các bạn độc giả một tài liệu bổ ích. Mặc dù đã rất cố gắng song vẫn không tránh khỏi một số thiếu sót, hy vọng sẽ được các bạn bè đồng nghiệp, các em sinh viên, các bạn độc giả góp ý chân thành để tôi có thể hoàn thiện hơn nữa tài liệu này.

Xin gửi lời cảm ơn chân thành tới các bạn bè đồng nghiệp và Ban chủ nhiệm khoa Công nghệ Thông tin đã tạo điều kiện giúp đỡ để tài liệu này có thể hoàn thành.

Hải phòng, tháng 04 năm 2007

Tác giả

Nguyễn Hữu Tuân

Chương 1: Thuật toán và cấu trúc dữ liệu

1. Thuật toán (giải thuật) - Algorithm

1.1. Định nghĩa thuật toán

Có rất nhiều các định nghĩa cũng như cách phát biểu khác nhau về định nghĩa của thuật toán. Theo như cuốn sách giáo khoa nổi tiếng viết về thuật toán là "**Introduction to Algorithms**" (Second Edition của Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest và Clifford Stein) thì thuật toán được định nghĩa như sau: "một thuật toán là một thủ tục tính toán xác định (well-defined) nhận các giá trị hoặc một tập các giá trị gọi là input và sinh ra ra một vài giá trị hoặc một tập giá trị được gọi là output".

Nói một cách khác các thuật toán giống như là các cách thức, qui trình để hoàn thành một công việc cụ thể xác định (well-defined) nào đó. Vì thế một đoạn mã chương trình tính các phần tử của dãy số Fibonaci là một cài đặt của một thuật toán cụ thể. Thậm chí một hàm đơn giản để cộng hai số cũng là một thuật toán hoàn chỉnh, mặc dù đó là một thuật toán đơn giản.

1.2. Đặc trưng của thuật toán

Tính đúng đắn: Thuật toán cần phải đảm bảo cho một kết quả đúng sau khi thực hiện đối với các bộ dữ liệu đầu vào. Đây có thể nói là đặc trưng quan trọng nhất đối với một thuật toán.

Tính dừng: thuật toán cần phải đảm bảo sẽ dừng sau một số hữu hạn bước.

Tính xác định: Các bước của thuật toán phải được phát biểu rõ ràng, cụ thể, tránh gây nhập nhằng hoặc nhầm lẫn đối với người đọc và hiểu, cài đặt thuật toán.

Tính hiệu quả: thuật toán được xem là hiệu quả nếu như nó có khả năng giải quyết hiệu quả bài toán đặt ra trong thời gian hoặc các điều kiện cho phép trên thực tế đáp ứng được yêu cầu của người dùng.

Tính phổ quát: thuật toán được gọi là có tính phổ quát (phổ biến) nếu nó có thể giải quyết được một lớp các bài toán tương tự.

Ngoài ra mỗi thuật toán theo định nghĩa đều nhận các giá trị đầu vào được gọi chung là các giá trị dữ liệu Input. Kết quả của thuật toán (thường là một kết quả cụ thể nào đó tùy theo các bài toán và thuật toán cụ thể) được gọi là Output.

2. Biểu diễn thuật toán

Thường có hai cách biểu diễn một thuật toán, cách thứ nhất là mô tả các bước thực hiện của thuật toán, cách thứ hai là sử dụng sơ đồ giải thuật.

2.1. Mô tả các bước thực hiện

Để biểu diễn thuật toán người ta mô tả chính xác các bước thực hiện của thuật toán, ngôn ngữ dùng để mô tả thuật toán có thể là ngôn ngữ tự nhiên hoặc một ngôn ngữ lai ghép giữa ngôn ngữ tự nhiên với một ngôn ngữ lập trình nào đó gọi là các đoạn giả mã lệnh.

Ví dụ: mô tả thuật toán tìm ước số chung lớn nhất của hai số nguyên.

Input: Hai số nguyên a, b.

Output: Ước số chung lớn nhất của a, b.

Thuật toán:

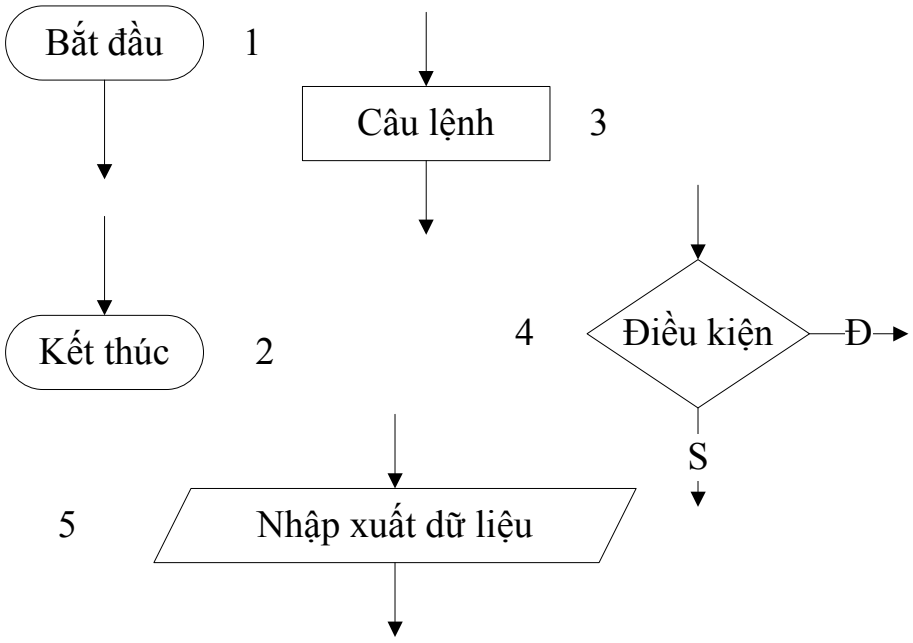
Bước 1: Nếu a=b thì USCLN(a, b)=a.

Bước 2: Nếu a > b thì tìm USCLN của a-b và b, quay lại bước 1;

Bước 3: Nếu a < b thì tìm USCLN của a và b-a, quay lại bước 1;

2.2. Sử dụng sơ đồ (lưu đồ) giải thuật (flowchart)

Sử dụng các ký hiệu hình khối cơ bản để tạo thành một mô tả mang tính hình thức (cách này rõ ràng hơn so với việc mô tả các bước thực hiện thuật toán).



Khối 1: Khối bắt đầu thuật toán, chỉ có duy nhất một đường ra;

Khối 2: Khối kết thúc thuật toán, có thể có nhiều đường vào;

Khối 3: Thực hiện câu lệnh (có thể là một hoặc nhiều câu lệnh); gồm một đường vào và một đường ra;

Khối 4: Rẽ nhánh, kiểm tra biểu thức điều kiện (biểu thức Boolean), nếu biểu thức đúng thuật toán sẽ đi theo nhánh Đúng (True), nếu biểu thức sai thuật toán sẽ đi theo nhánh Sai (False).

Khối 5: Các câu lệnh nhập và xuất dữ liệu.

3. Độ phức tạp thuật toán – Algorithm Complexity

3.1. Các tiêu chí đánh giá thuật toán

Thông thường để đánh giá mức độ tốt, xấu và so sánh các thuật toán cùng loại, có thể dựa trên hai tiêu chuẩn:

- Thuật toán đơn giản, dễ hiểu, dễ cài đặt.

- Dựa vào thời gian thực hiện và tài nguyên mà thuật toán sử dụng để thực hiện trên các bộ dữ liệu.

Trên thực tế các thuật toán hiệu quả thì không dễ hiểu, các cài đặt hiệu quả cũng không dễ dàng thực hiện và hiểu được một cách nhanh chóng. Và một điều có vẻ nghịch lý là các thuật toán càng hiệu quả thì càng khó hiểu, cài đặt càng phức tạp lại càng hiệu quả (không phải lúc nào cũng đúng). Vì thế để đánh giá và so sánh các thuật toán người ta thường dựa trên độ phức tạp về thời gian thực hiện của thuật toán, gọi là độ phức tạp thuật toán (**algorithm complexity**). Về bản chất độ phức tạp thuật toán là một hàm ước lượng (có thể không chính xác) số phép tính mà thuật toán cần thực hiện (từ đó dễ dàng suy ra thời gian thực hiện của thuật toán) đối với một bộ dữ liệu input có kích thước N. N có thể là số phần tử của mảng trong trường hợp bài toán sắp xếp hoặc tìm kiếm, hoặc có thể là độ lớn của số trong bài toán kiểm tra số nguyên tố chẳng hạn.

3.2. Đánh giá thời gian thực hiện thuật toán

Để minh họa việc đánh giá độ phức tạp thuật toán ta xem xét ví dụ về thuật toán sắp xếp chọn (selection sort) và sắp xếp đổi chỗ trực tiếp (exchange sort) như sau:

Cài đặt của thuật toán sắp xếp chọn:

```
for(i=0;i<n;i++)
{
        min_idx = i;
        for(j=i+1;j<n;j++)
                if(a[j]<a[min_idx])
                        min_idx = j;
        if(min_idx!=i)
        {
                temp = a[i];
                a[i] = a[min_idx];
                a[min_idx] = temp;
        }
}
```

Số phép tính thuật toán cần thực hiện được tính như sau:

(N-1) + (N-2) + … + 2 + 1 = N*(N-1)/2.

Phân tích chi tiết hơn thì N*(N-1)/2 là số phép toán so sánh cần thực hiện, còn số lần thực hiện đổi chỗ hai phần tử (số nguyên) tối đa của thuật toán là N.

Cài đặt của thuật toán sắp xếp đổi chỗ trực tiếp:

```
for(i=0;i<n;i++)
        for(j=i+1;j<n;j++)
        if(a[j] < a[i])
```

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

Tương tự đối với thuật toán sắp xếp chọn ta có số phép toán thực hiện là: (N-1) + (N-2) + …+2 +1 = N*(N-1)/2. Chi tiết hơn, N*(N-1)/2 là số lần so sánh thuật toán thực hiện, và cũng là số lần đổi chỗ hai phần tử (hai số nguyên) tối đa của thuật toán.

Trong trường hợp trung bình, thuật toán sắp xếp chọn có xu hướng tốt hơn so với sắp xếp đổi chỗ trực tiếp vì số thao tác đổi chỗ ít hơn, còn trong trường hợp tốt nhất thì như nhau, trường hợp tồi nhất thì chắc chắn thuật toán sắp xếp chọn tốt hơn, do đó có thể kết luận thuật toán sắp xếp chọn nhanh hơn so với thuật toán sắp xếp đổi chỗ trực tiếp.

3.3. Các định nghĩa hình thức về độ phức tạp thuật toán

Gọi f, g là các hàm không giảm định nghĩa trên tập các số nguyên dương. (chú ý là tất cả các hàm thời gian đều thỏa mãn các điều kiện này). Chúng ta nói rằng hàm f(N) là ***O***(g(N)) (đọc là: f là O lớn của g) nếu như tồn tại một hằng số c và N_0:

$$\forall N > N_0; f(N) < c.g(N)$$

Phát biểu thành lời như sau: f(N) là ***O***(g(N)) nếu tồn tại c sao cho hầu hết phần đồ thị của hàm f nằm dưới phần đồ thị của hàm c*g. Chú ý là hàm f tăng nhiều nhất là nhanh bằng hàm c*g.

Thay vì nói f(N) là ***O***(g(N)) chúng ta thường viết là f(N) = ***O***(g(N)). Chú ý rằng đẳng thức này không có tính đối xứng có nghĩa là chúng ta có thể viết ngược lại ***O***(g(N)) = f(N) nhưng không thể suy ra g(N) = ***O***(f(N)).

Định nghĩa trên được gọi là ký hiệu O lớn (big-O notation) và thường được sử dụng để chỉ định các chặn trên của hàm tăng.

Chẳng hạn đối với ví dụ về sắp xếp bằng chọn ta có f(N) = N*(N-1)/2 = $0.5N^2 – 0.5N$ chúng ta có thể viết là f(N) = ***O***(N^2). Có nghĩa là hàm f không tăng nhanh hơn hàm N^2.

Chú ý rằng thậm chí hàm f chính xác có công thức như thế nào không cho chúng ta câu trả lời chính xác của câu hỏi “Chương trình có thời gian thực hiện là bao lâu trên máy tính của tôi?”. Nhưng điều quan trọng là qua đó chúng ta biết được hàm thời gian thực hiện của thuật toán là hàm bậc hai. Nếu chúng ta tăng kích thước input lên 2 lần, thời gian thực hiện của chương trình sẽ tăng lên xấp xỉ 4 lần không phụ thuộc vào tốc độ của máy.

Chặn trên f(N) = ***O***(N^2) cho chúng ta kết quả gần như thế – nó đảm bảo rằng độ tăng của hàm thời gian nhiều nhất là bậc hai.

Do đó chúng ta sẽ sử dụng ký pháp O lớn để mô tả thời gian thực hiện của thuật toán (và đôi khi cả bộ nhớ mà thuật toán sử dụng). Đối với thuật toán trong ví dụ 2 chúng ta có thể nói “độ phức tạp thời gian của thuật toán là ***O***(N^2) hoặc ngắn gọn là “thuật toán là ***O***(N^2)”.

Tương tự chúng ta có các định nghĩa Ω (**omega**)và Θ (**theta**):

Chúng ta nói rằng hàm f(N) là Ω(g(N)) nếu như g(N) = ***O***(f(N)), hay nói cách khác hàm f tăng ít nhất là nhanh bằng hàm g.

Và nói rằng hàm f(N) là Θ(g(N)) nếu như f(N) = ***O***(g(N)) và g(N) = ***O***(f(N)), hay nói cách khác cả hai hàm xấp xỉ như nhau về độ tăng.

Hiển nhiên là cách viết Ω là để chỉ ra chặn dưới và Θ là để chỉ ra một giới hạn chặt chẽ của một hàm. Còn có nhiều giới hạn khác nữa nhưng đây là các giới hạn mà chúng ta hay gặp nhất.

Một vài ví dụ:

- $0.5N^2 – 0.5N$ = ***O***(N2)
- 47 N*log(N) = ***O***(N*log(N))
- N*log(N) + 1000047N = Θ(N*log(N))
- Tất cả các hàm đa thức bậc k đều là ***O***(N^k)
- Độ phức tạp thời gian của thuật toán sắp xếp chọn và sắp xếp đổi chỗ trực tiếp là $\Theta(N^2)$
- Nếu một thuật toán là ***O***(N^2) thì nó cũng là ***O***(N^5)
- Mỗi thuật toán sắp xếp dựa trên so sánh có độ phức tạp tối ưu là Ω(N*log(N))
- Thuật toán sắp xếp MergeSort có số thao tác so sánh là N*log(N). Do đó độ phức tạp thời gian của MergeSort là Θ(N*log(N)) có nghĩa là MergeSort là tiệm cận với thuật toán sắp xếp tối ưu.

Khi xem xét so sánh các thuật toán cùng loại người ta thường xét độ phức tạp của thuật toán trong các trường hợp: trung bình (average case), trường hợp xấu nhất (the worst case) và trường hợp tốt nhất (the best case).

3.4. Các lớp thuật toán

Khi chúng ta nói về độ phức tạp thời gian/ không gian nhớ của một thuật toán thay vì sử dụng các ký hiệu hình thức Θ(f(n)) chúng ta có thể đơn giản đề cập tới lớp của hàm f. Ví dụ f(N) = Θ(N) chúng ta sẽ nói thuật toán là tuyến tính (linear). Có thể tham khảo thêm:

f(N) = 1: hằng số (constant)

f(N) = Θ(log(N)): logarit

f(N) = Θ(N): tuyến tính (linear)

f(N) = Θ(N*log(N)): N log N

f(N) = $\Theta(N^2)$: bậc hai (quadratic)

f(N) = $\Theta(N^3)$: bậc 3 (cubic)

f(N) = ***O***(N^k) với k là một số nguyên dương: đa thức (polynomial)

f(N) = $\Omega(b^N)$: hàm mũ (exponential)

Đối với các bài toán đồ thị độ phức tạp Θ(V+E) có nghĩa là "tuyến tính đối với kích thước của đồ thị".

Xác định thời gian thực hiện từ một giới hạn tiệm cận

Đối với hầu hết các thuật toán chúng ta có thể gặp các hằng số bị che đi bởi cách viết ***O*** hoặc **b** thường là khá nhỏ. Chăng hạn nếu độ phức tạp thuật toán là $\Theta(N^2)$ thì chúng ta sẽ mong muốn chính xác độ phức tạp thời gian là $10N^2$ chứ không phải là 10^7N^2.

Có nghĩa là nếu hằng số là lớn thì thường là theo một cách nào đó liên quan tới một vài hằng số của bài toán. Trong trường hợp này tốt nhất là gán cho hằng đó một cái tên và đưa nó vào ký hiệu tiệm cận của hằng số đó.

Ví dụ: bài toán đếm số lần xuất hiện của mỗi ký tự trong một xâu có N ký tự. Một thuật toán cơ bản là duyệt qua toàn bộ xâu đối với mỗi ký tự để thực hiện đếm xem ký tự đó xuất hiện bao nhiêu lần. Kích thước của bảng chữ cái là cố định (nhiều nhất là 255 đối với ngôn ngữ lập trình C) do đó thuật toán là tuyến tính đối với N. Nhưng sẽ là tốt hơn nếu viết là độ phức tạp của thuật toán là $\Theta(S*N)$ trong đó S là số phần tử của bảng chữ cái sử dụng. (Chú ý là có một thuật toán tốt hơn để giải bài toán này với độ phức tạp là $\Theta(S + N)$.

Trong các cuộc thi lập trình một thuật toán thực hiện 1000000000 phép nhân có thể không thỏa mãn ràng buộc thời gian. Chúng ta có thể tham khảo bảng sau để biết thêm:

Độ phức tạp	Giá trị N lớn nhất
$\Theta(N)$	100 000 000
$\Theta(N \log N)$	40 000 000
$\Theta(N2)$	10 000
$\Theta(N3)$	500
$\Theta(N4)$	90
$\Theta(2N)$	20
$\Theta(N!)$	11

Thời gian thực hiện của các thuật toán có độ phức tạp khác nhau	
O(Log(N))	10^{-7} giây
O(N)	10^{-6} giây
O(N*Log(N))	10^{-5} giây
O(N^2)	10^{-4} giây
O(N^6)	3 phút
O(2^N)	10^{14} năm
O(N!)	10^{142} năm

Chú ý về phân tích thuật toán.

Thông thường khi chúng ta trình bày một thuật toán cách tốt nhất để nói về độ phức tạp thời gian của nó là sử dụng các chặn Θ. Tuy nhiên trên thực tế chúng ta hay dùng ký pháp big-O – các kiểu khác không có nhiều giá trị lắm, vì cách này rất dễ gõ và cũng được

nhiều người biết đến và hiểu rõ hơn. Nhưng đừng quên là big-O là chặn trên và thường thì chúng ta sẽ tìm môt chặn trên càng nhỏ càng tốt.

Ví dụ: Cho một mảng đã được sắp A. Hãy xác định xem trong mảng A có hai phần tử nào mà hiệu của chúng bằng D hay không. Hãy xem đoạn mã chương trình sau:

```
int j=0;
for (int i=0; i<N; i++)
{
        while ( (j<N-1) && (A[i]-A[j] > D) )
                j++;
        if (A[i]-A[j] == D)
                return 1;
}
```

Rất dễ để nói rằng thuật toán trên là $\boldsymbol{O}(N^2)$: vòng lặp while bên trong được gọi đến N lần, mỗi lần tăng j lên tối đa N lần. Nhưng một phân tích tốt hơn sẽ cho chúng ta thấy rằng thuật toán là $\boldsymbol{O}(N)$ vì trong cả thời gian thực hiện của thuật toán lệnh tăng j không chạy nhiều hơn N lần.

Nếu chúng ta nói rằng thuật toán là $\boldsymbol{O}(N^2)$ chúng ta vẫn đúng nhưng nếu nói là thuật toán là $\boldsymbol{O}(N)$ thì chúng ta đã đưa ra được thông tin chính xác hơn về thuật toán.

4. Cấu trúc dữ liệu – Data structure

4.1. Mối liên hệ giữa cấu trúc dữ liệu và giải thuật

4.2. Các tiêu chuẩn đánh giá cấu trúc dữ liệu

4.3. Các kiểu dữ liệu cơ bản của ngôn ngữ C

4.4. Các kiểu dữ liệu có cấu trúc

4.5. Một số kiểu dữ liệu có cấu trúc cơ bản

5. Các chiến lược thiết kế thuật toán.

Không có một phương pháp nào có thể giúp chúng ta xây dựng (thiết kế) nên các thuật toán cho tất cả các loại bài toán. Các nhà khoa học máy tính đã nghiên cứu và đưa ra các chiến lược thiết kế các giải thuật chung nhất áp dụng cho các loại bài toán khác nhau.

5.1. Chiến lược vét cạn (Brute force)

Đây là chiến lược đơn giản nhất nhưng cũng là không hiệu quả nhất. Chiến lược vét cạn đơn giản thử tất cả các khả năng xem khả năng nào là nghiệm đúng của bài toán cần giải quyết.

Ví dụ thuật toán duyệt qua mảng để tìm phần tử có giá trị lớn nhất chính là áp dụng chiến lược vét cạn. Hoặc bài toán kiểm tra và in ra tất cả các số nguyên tố có 4 chữ số abcd sao cho ab = cd (các số có 2 chữ số) được thực hiện bằng thuật toán vét cạn như sau:

```
for(a=1;a<=9;a++)
for(b=0;b<=9;b++)
        for(c=0;c<=9;c++)
        for(d=0;d<=9;d++)
        if(ktnguyento(a*1000+b*100+c*10+d) && (10*a+b==10*c+d))
                printf("%d%d%d%d", a, b, c, d);
```

Hàm ktnguyento() kiểm tra xem một số nguyên có phải là số nguyên tố hay không.

Các thuật toán áp dụng chiến lược vét cạn thuộc loại: tìm tất cả các nghiệm có thể có. Về mặt lý thuyết, chiến lược này có thể áp dụng cho mọi loại bài toán, nhưng có một hạn chế khiến nó không phải là chìa khóa vạn năng về mặt thực tế: do cần phải thử tất cả các khả năng nên số trường hợp cần phải thử của bài toán thường lên tới con số rất lớn và thường quá lâu so với yêu cầu của bài toán đặt ra.

5.2. Chiến lược quay lui (Back tracking / try and error)

Đây là một trong những chiến lược quan trọng nhất của việc thiết kế thuật toán. Tương tự như chiến lược vét cạn song chiến lược quay lui có một điểm khác: nó lưu giữ các trạng thái trên con đường đi tìm nghiệm của bài toán. Nếu tới một bước nào đó, không thể tiến hành tiếp, thuật toán sẽ thực hiện thao tác quay lui (back tracking) về trạng thái trước đó và lựa chọn các khả năng khác. Bài toán mà loại thuật toán này thường áp dụng là tìm một nghiệm có thể có của bài toán hoặc tìm tất cả các nghiệm sau đó chọn lấy một nghiệm thỏa mãn một điều kiện cụ thể nào đó (chẳng hạn như tối ưu nhất theo một tiêu chí nào đó), hoặc cũng có thể là tìm tất cả các nghiệm của bài toán. Và cũng như chiến lược vét cạn, chiến lược quay lui chỉ có thể áp dụng cho các bài toán kích thước input nhỏ.

Vecto nghiệm

Một trong các dạng bài toán mà chiến lược quay lui thường áp dụng là các bài toán mà nghiệm của chúng là các cấu hình tổ hợp. Tư tưởng chính của giải thuật là xây dựng dần các thành phần của cấu hình bằng cách thử lần lượt tất cả các khả năng có thể có. Nếu tồn tại một khả năng chấp nhận được thì tiến hành bước kế tiếp, trái lại cần lùi lại một bước để thử lại các khả năng chưa được thử. Thông thường giải thuật này thường được gắn liền với cách diễn đạt qui nạp và có thể mô tả chi tiết như sau:

Trước hết ta cần hình thức hóa việc biểu diễn một cấu hình. Thông thường ta có thể trình bày một cấu hình cần xây dựng như là một bộ có thứ tự (vecto) gồm N thành phần:

$X = (x_1, x_2, \ldots, x_N)$

thoả mãn một số điều kiện nào đó.

Giả thiết ta đã xây dựng xong i-1 thành phần x_1, x_2, …, x_{i-1}, bây giờ là bước xây dựng thành phần x_i. Ta lần lượt thử các khả năng có thể có cho x_i. Xảy ra các trường hợp:

Tồn tại một khả năng j chấp nhận được. Khi đó x_i sẽ được xác định theo khả năng này. Nếu x_i là thành phần cuối (i=n) thì đây là một nghiệm, trái lại (i<n) thì tiến hành các bước tiếp theo qui nạp.

Tất cả các khả năng đề cử cho x_i đều không chấp nhận được. Khi đó cần lùi lại bước trước để xác định lại x_{i-1}.

Để đảm bảo cho việc vét cạn (exhausted) tất cả các khả năng có thể có, các giá trị đề cử không được bỏ sót. Mặt khác để đảm bảo việc không trùng lặp, khi quay lui để xác định lại giá trị x_{i-1} cần không được thử lại những giá trị đã thử rồi (cần một kỹ thuật đánh dấu các giá trị đã được thử ở các bước trước).

Trong phần lớn các bài toán, điều kiện chấp nhận j không những chỉ phụ thuộc vào j mà còn phụ thuộc vào việc xác định i-1 thành phần trước, do đó cần tổ chức một số biến trạng thái để cất giữ trạng thái của bài toán sau khi đã xây dựng xong một thành phần để chuẩn bị cho bước xây dựng tiếp. Trường hợp này cần phải hoàn nguyên lại trạng thái cũ khi quay lui để thử tiếp các khả năng trong bước trước.

Thủ tục đệ qui

Thủ tục cho thuật toán quay lui được thiết kế khá đơn giản theo cơ cấu đệ qui của thủ tục try dưới đây (theo cú pháp ngôn ngữ C).

```
void try(i: integer)
{
        //  xác định thành phần xi bằng đệ qui
        int j;
        for j ∈ < tập các khả năng đề cử > do
                if(chấp nhận  j)
                {
                        <xác định xi theo khả năng j >
                        < ghi nhận trạng thái mới >
                        If(i=n)
                                < ghi nhận một nghiệm>
                        else
                                try(i+1);
                        < trả lại trạng thái cũ >
                }
}
```

Trong chương trình chính chỉ cần gọi tới try(1) để khởi động cơ cấu đệ qui hoạt động. Tất nhiên, trước đấy cần khởi tạo các giá trị ban đầu cho các biến. Thông thường việc này được thực hiện qua một thủ tục nào đó mà ta gọi là init (khởi tạo).

Hai điểm mấu chốt quyết định độ phức tạp của thuật toán này trong các trường hợp cụ thể là việc xác định các giá trị đề cử tại mỗi bước dành cho xi và xác định điều kiện chấp nhận được cho các giá trị này.

Các giá trị đề cử

Các giá trị đề cử thông thường lớn hơn nhiều so với số các trường hợp có thể chấp nhận được. Sự chênh lệch này càng lớn thì thời gian phải thử càng nhiều, vì thế càng thu hẹp được điều kiện đề cử càng nhiều càng tốt (nhưng không được bỏ sót). Việc này phụ thuộc vào việc phân tích các điều kiện ràng buộc của cấu hình để phát hiện những điều kiện cần của cấu hình đang xây dựng. Lý tưởng nhất là các giá trị đề cử được mặc nhiên chấp nhận. Trong trường hợp này mệnh đề < chấp nhận j > được bỏ qua (vì thế cũng không cần các biến trạng thái).

Ví dụ 1: Sinh các dãy nhị phân độ dài N (N ≤ 20)

Ví dụ dưới đây trình bày chương trình sinh các dãy nhị phân độ dài N, mỗi dãy nhị phân được tổ chức như một màng n thành phần:

x[0], x[1], …, x[n-1]

trong đó mỗi x[i] có thể lấy một trong các giá trị từ 0 tới 1, có nghĩa là mỗi phần tử x[i] của vecto nghiệm có 2 giá trị đề cử, và vì cần sinh tất cả các xâu nhị phân nên các giá trị đề cử này đều được chấp nhận. Thủ tục chính của chương trình đơn giản như sau:

```
void try(int k)
{
        int j;
        if(k==n)
                in_nghiem();
        else
                for(j=0;j<=1;j++)
                {
                        x[k] = j;
                        try(k+1);
                }
}
```

Trong đó in_nghiem() là hàm in nghiệm tìm được ra màn hình. Dưới đây là toàn bộ chương trình. Trong chương trình có khai báo thêm biến count để đếm các chỉnh hợp được tạo.

5.3. Chia để trị (Divide and Conquer)

Chiến lược chia để trị là một chiến lược quan trọng trong việc thiết kế các giải thuật. Ý tưởng của chiến lược này nghe rất đơn giản và dễ nhận thấy, đó là: khi cần giải quyết một bài toán, ta sẽ tiến hành chia bài toán đó thành các bài toán nhỏ hơn, giải các bài toán nhỏ hơn đó, sau đó kết hợp nghiệm của các bài toán nhỏ hơn đó lại thành nghiệm của bài toán ban đầu.

Tuy nhiên vấn đề khó khăn ở đây nằm ở hai yếu tố: làm thế nào để chia tách bài toán một cách hợp lý thành các bài toán con, vì nếu các bài toán con lại được giải quyết bằng các thuật toán khác nhau thì sẽ rất phức tạp, yếu tố thứ hai là việc kết hợp lời giải của các bài toán con sẽ được thực hiện như thế nào?.

Các thuật toán sắp xếp trộn (merge sort), sắp xếp nhanh (quick sort) đều thuộc loại thuật toán chia để trị (các thuật toán này được trình bày ở chương 3).

Ví dụ[6, trang 57]: Trong ví dụ này chúng ta sẽ xem xét thuật toán tính a^N.

Để tính a^N ta để ý công thức sau:

$$a^N = \begin{cases} 1 \text{ nếu } N = 0 \\ (a^{N/2})^2 \text{ nếu N chẵn} \\ a*(a^{N/2})^2 \text{ nếu N lẻ} \end{cases}$$

Từ công thức trên ta suy ra cài đặt của thuật toán như sau:

```
int power(int a, int n)
{
        if(n==0)
                return 1;
        else{
                int t = power(a, n/2);
                if(n%2==0)
                        return t*t;
                else
                        return a*t*t;
        }
}
```

5.4. Chiến lược tham lam (Greedy)

Chú ý: Trong một số bài toán nếu xây dựng được hàm chọn thích hợp có thể cho nghiệm tối ưu. Trong nhiều bài toán, thuật toán tham ăn chỉ cho nghiệm gần đúng với nghiệm tối ưu.

5.5. Qui hoạch động (Dynamic Programming)

6. Bài tập

Bài tập 1: Xây dựng sơ đồ giải thuật cho bài toán tính số Fibonaci thứ N, biết rằng dãy số Fibonaci được định nghĩa như sau:

F[0] = F[1] = 1, F[N] = F[N-1] + F[N-2] với N ≥ 2.

Bài tập 2: Xây dựng sơ đồ giải thuật cho bài toán tính biểu thức:

$\sqrt{x+\sqrt{x+...+\sqrt{x}}}$, với N số x thực nằm trong các dấu căn bậc hai, N và x nhập từ bàn phím.

Bài tập 3: Trong một ma trận hai chiều cấp MxN, một phần tử a[i][j] được gọi là điểm yên ngựa của ma trận (saddle point) nếu như nó là phần tử nhỏ nhất trên hàng i và phần tử lớn nhất trên cột j của ma trận. Chẳng hạn a[2][0] = 7 là một phần tử yên ngựa trong ma trận sau:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Hãy viết chương trình tìm tất cả các điểm yên ngựa của một ma trận nhập vào từ bàn phím và đưa ra độ phức tạp của thuật toán.

Bài tập 4: Cho một ma trận kích thước MxN gồm các số nguyên (có cả số âm và dương). Hãy viết chương trình tìm ma trận con của ma trận đã cho sao cho tổng các phần tử trong ma trận con đó lớn nhất có thể được (bài toán maximum sum plateau). Hãy đưa ra đánh giá về độ phức tạp của thuật toán sử dụng.

Bài tập 5: Viết chương trình nhập vào các hệ số của một đa thức (giả sử các hệ số là nguyên và đa thức có biến x là một số nguyên) và một giá trị x0. Hãy tính giá trị của đa thức theo công thức Horner sau:

Nếu $f(x) = a_n*x^n + a_{n-1}*x^{n-1} + .. + a_1*x + a_0$ thì

$f(x) = a_0 + x*(a_1+x*(a_2+x*(….+x(a_{n-1}+a_n*x)…)$ (Công thức Horner).

Bài tập 6: Cho 4 hình hộp kích thước bằng nhau, mỗi mặt của hình hộp được tô bằng 1 trong 4 màu xanh, đỏ, tím, vàng. Hãy đưa ra tất cả các cách xếp các hình hộp thành 1 dãy sao cho khi nhìn theo các phía trên xuống, đằng trước và đằng sau của dãy đều có đủ cả 4 màu xanh, đỏ, tím vàng.

Bài tập 7: Hãy viết chương trình nhanh nhất có thể được để in ra tất cả các số nguyên số có hai chữ số.

Bài tập 8: Áp dụng thuật toán sàng để in ra tất cả các số nguyên tố nhỏ hơn N.

Chương 2: Tìm kiếm (Searching)

1. Bài toán tìm kiếm

Tìm kiếm là một trong những vấn đề thuộc lĩnh vực nghiên cứu của ngành khoa học máy tính và được ứng dụng rất rộng rãi trên thực tế. Bản thân mỗi con người chúng ta đã có những tri thức, những phương pháp mang tính thực tế, thực hành về vấn đề tìm kiếm. Trong các công việc hàng ngày chúng ta thường xuyên phải tiến hành tìm kiếm: tìm kiếm một cuốn sách để đọc về một vấn đề cần quan tâm, tìm một tài liệu lưu trữ đâu đó trên máy tính hoặc trên mạng, tìm một từ trong từ điển, tìm một bản ghi thỏa mãn các điều kiện nào đó trong một cơ sở dữ liệu, tìm kiếm trên mạng Internet.

Trong môn học này chúng ta quan tâm tới bài toán tìm kiếm trên một mảng, hoặc một danh sách các phần tử cùng kiểu. Thông thường các phần tử này là một bản ghi được phân chia thành hai trường riêng biệt: trường lưu trữ các dữ liệu và một trường để phân biệt các phần tử với nhau (các thông tin trong trường dữ liệu có thể giống nhau hoàn toàn) gọi là trường khóa, tập các phần tử này được gọi là không gian tìm kiếm của bài toán tìm kiếm, không gian tìm kiếm được lưu hoàn toàn trên bộ nhớ của máy tính khi tiến hành tìm kiếm.

Kết quả tìm kiếm là **vị trí của phần tử thỏa mãn điều kiện tìm kiếm**: có trường khóa bằng với một giá trị khóa cho trước (khóa tìm kiếm). Từ vị trí tìm thấy này chúng ta có thể truy cập tới các thông tin khác được chứa trong trường dữ liệu của phần tử tìm thấy. Nếu kết quả là không tìm thấy (trong trường hợp này thuật toán vẫn kết thúc thành công) thì giá trị trả về sẽ được gán cho một giá trị đặc biệt nào đó tương đương với việc không tồn tại phần tử nào có ví trí đó: chẳng hạn như -1 đối với mảng và NULL đối với danh sách liên kết.

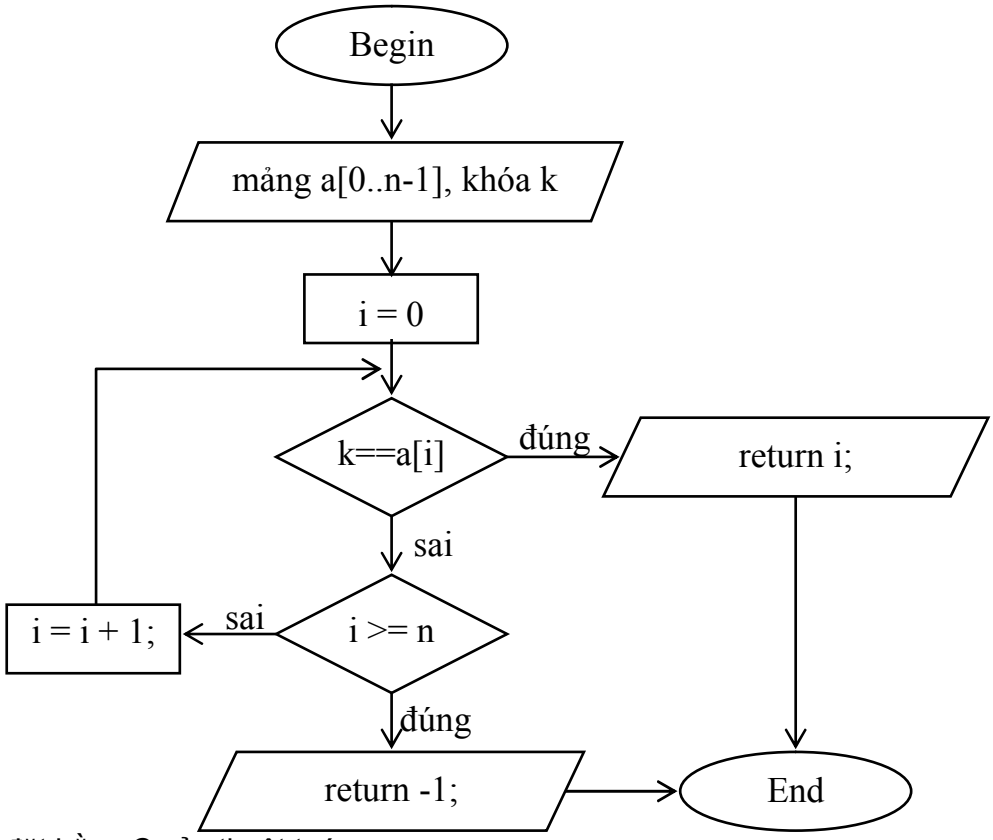
Các thuật toán tìm kiếm cũng có rất nhiều: từ các thuật toán tìm kiếm vét cạn, tìm kiếm tuần tự, tìm kiếm nhị phân, cho tới những thuật toán tìm kiếm dựa trên các cấu trúc dữ liệu đặc biệt như các từ điển, các loại cây như cây tìm kiếm nhị phân, cây cân bằng, cây đỏ đen … Tuy nhiên ở phần này chúng ta sẽ xem xét hai phương pháp tìm kiếm được áp dụng với cấu trúc dữ liệu mảng (dữ liệu tìm kiếm được chứa hoàn toàn trong bộ nhớ của máy tính).

Điều đầu tiên mà chúng ta cần lưu ý là đối với cấu trúc mảng này, việc truy cập tới các phần tử ở các vị trí khác nhau là như nhau và dựa vào chỉ số, tiếp đến chúng ta sẽ tập trung vào thuật toán nên có thể coi như mỗi phần tử chỉ có các trường khóa là các số nguyên.

2. Tìm kiếm tuần tự (Sequential search)

Ý tưởng của thuật toán tìm kiếm tuần tự rất đơn giản: duyệt qua tất cả các phần tử của mảng, trong quá trình duyệt nếu tìm thấy phần tử có khóa bằng với khóa tìm kiếm thì trả về vị trí của phần tử đó. Còn nếu duyệt tới hết mảng mà vẫn không có phần tử nào có khóa bằng với khóa tìm kiếm thì trả về -1 (không tìm thấy).

Thuật toán có sơ đồ giải thuật như sau:



Cài đặt bằng C của thuật toán:

```
int sequential_search(int a[], int n, int k)
{
        int i;
        for(i=0;i<n;i++)
                if(a[i]==k)
                        return i;
        return -1;
}
```

Dễ dàng nhận ra thuật toán sẽ trả về kết quả là vị trí của phần tử đầu tiên thỏa mãn điều kiện tìm kiếm nếu tồn tại phần tử đó.

Độ phức tạp thuật toán trong trường hợp trung bình và tồi nhất: ***O***(n).

Trong trường hợp tốt nhất thuật toán có độ phức tạp ***O***(1).

Các bài toán tìm phần tử lớn nhất và tìm phần tử nhỏ nhất của một mảng, danh sách cũng là thuật toán tìm kiếm tuần tự. Một điều dễ nhận thấy là khi số phần tử của mảng nhỏ (cỡ 10000000) thì thuật toán làm việc ở tốc độ chấp nhận được, nhưng khi số phần tử của mảng lên đến hàng tỷ, chẳng hạn như tìm tên một người trong số tên người của cả thế giới thì thuật toán tỏ ra không hiệu quả.

3. Tìm kiếm nhị phân (binary search)

Thuật toán tìm kiếm nhị phân là một thuật toán rất hiệu quả, nhưng điều kiện để áp dụng được thuật toán này là không gian tìm kiếm cần phải được sắp xếp trước theo khóa tìm kiếm.

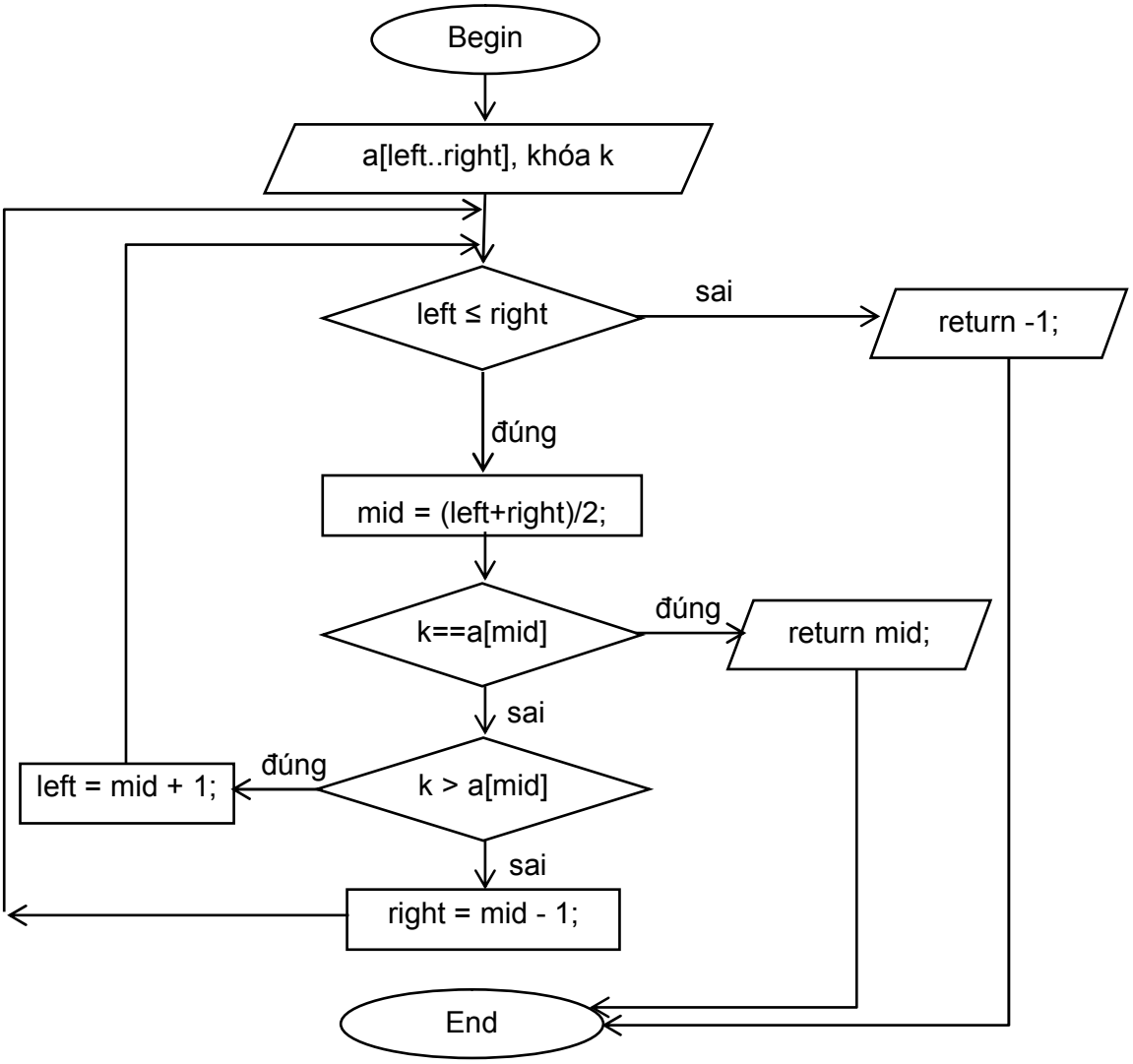
Mô tả thuật toán:

Input: mảng a[left..right] đã được sắp theo khóa tăng dần, khóa tìm kiếm k.

Output: vị trí của phần tử có khóa bằng k.

Thuật toán này thuộc loại thuật toán chia để trị, do mảng đã được sắp xếp, nên tại mỗi bước thay vì duyệt qua các phần tử như thuật toán tìm kiếm tuần tự, thuật toán tìm kiếm nhị phân xét phần tử ở vị trí giữa mảng tìm kiếm a[(left+right)/2], nếu đó là phần tử có khóa bằng với khóa tìm kiếm k thì trả về vị trí đó và kết thúc quá trình tìm kiếm. Nếu không sẽ có hai khả năng xảy ra, một là phần tử đó lớn hơn khóa tìm kiếm k, khi đó do mảng đã đước sắp nên nếu trong mảng có phần tử có trường khóa bằng k thì vị trí của phần tử đó sẽ ở phần trước a[(left+right)/2], có nghĩa là ta sẽ điều chỉnh right = (left+right)/2 - 1. Còn nếu a[(left+right)/2] < k thì theo lý luận tương tự ta sẽ điều chỉnh left = (left+right)/2 + 1. Trong bất cứ trường hợp nào thì không gian tìm kiếm đều sẽ giảm đi một nửa số phần tử sau mỗi bước tìm kiếm.

Sơ đồ thuật toán:



Cài đặt bằng C của thuật toán tìm kiếm nhị phân:

```
int binary_search(int a[], int left, int right, int key)
{
	// cài đặt không đệ qui
	int mid;
	while(left<=right)
	{
		mid = (left + right)/2;
		if(a[mid] == key)
			return mid;
		if(a[mid] < key)
			left = mid + 1;
		else
			right = mid – 1;
	}
	return -1;
}
```

Cài đặt đệ qui:

```
int recursive_bsearch(int a[], int left, int right, int key)
{
	// cài đặt đệ qui
	int mid;
	mid = (left + right)/2;
	if(left>right)
		return -1;
	if(a[mid] == key)
		return mid;
	else
		if(a[mid] < key)
			return recursive_bsearch(a, mid+1, right, key);
		else
			return recursive_bsearch(a, left, mid-1, key);
}
```

Để gọi hàm cài đặt với mảng a có n phần tử ta gọi như sau:

int kq = binary_search(a, 0, n – 1, k);

hoặc:

int kq = recursive_bsearch(a, 0, n – 1, k);

Thuật toán có độ phức tạp là hàm logarit ***O***(log(N)). Với n = 6.000.000.000 thì số thao tác cần thực hiện để tìm ra kết quả là log(n) = 31 thao tác, có nghĩa là chúng ta chỉ cần 31 bước thực hiện để tìm ra tên một người trong số tất cả dân số trên thế giới, thuật toán tìm kiếm nhị phân thực sự là một thuật toán hiệu quả. Trên thực tế việc sắp các từ của từ điển là một áp dụng của thuật toán tìm kiếm nhị phân.

4. Bài tập

Bài tập 1: Viết chương trình nhập vào 1 mảng số nguyên và một số nguyên k, hãy đếm xem có bao nhiêu số bằng k. Nhập tiếp 2 số x < y và đếm xem có bao nhiêu số lớn hơn x và nhỏ hơn y.

Bài tập 2: Cài đặt thuật toán tìm kiếm tuyến tính theo kiểu đệ qui.

Bài tập 3: Viết chương trình nhập một mảng các số nguyên từ bàn phím, nhập 1 số nguyên S, hãy đếm xem có bao nhiêu cặp số của mảng ban đầu có tổng bằng S, có hiệu bằng S.

Chương 3: Sắp xếp (Sorting)

1. Bài toán sắp xếp

Sắp xếp được xem là một trong những lĩnh vực nghiên cứu cổ điển của khoa học máy tính. Trước khi đi vào các thuật toán chi tiết chúng ta cần nắm vững một số khái niệm cơ bản sau:

- Một trường (field) là một đơn vị dữ liệu nào đó chẳng hạn như tên, tuổi, số điện thoại của một người ...
- Một bản ghi (record) là một tập hợp các trường
- Một file là một tập hợp các bản ghi

Sắp xếp (sorting) là một quá trình xếp đặt các bản ghi của một file theo một thứ tự nào đó. Việc xếp đặt này được thực hiện dựa trên một hay nhiều trường nào đó, và các thông tin này được gọi là khóa xắp xếp (key). Thứ tự của các bản ghi được xác định dựa trên các khóa khác nhau và việc sắp xếp đối được thực hiện đối với mỗi khóa theo các thứ tự khác nhau. Chúng ta sẽ tập trung vào các thuật toán xắp xếp và giả sử khóa chỉ gồm 1 trường duy nhất. Hầu hết các thuật toán xắp xếp được gọi là các thuật toán xắp xếp so sánh: chúng sử dụng hai thao tác cơ bản là so sánh và đổi chỗ (swap) các phần tử cần sắp xếp.

Các bài toán sắp xếp đơn giản được chia làm hai dạng.

Sắp xếp trong (internal sorting): Dữ liệu cần sắp xếp được lưu đầy đủ trong bộ nhớ trong để thực hiện thuật toán sắp xếp.

Sắp xếp ngoài (external sorting): Dữ liệu cần sắp xếp có kích thước quá lớn và không thể lưu vào bộ nhớ trong để sắp xếp, các thao tác truy cập dữ liệu cũng mất nhiều thời gian hơn.

Trong phạm vi của môn học này chúng ta chỉ xem xét các thuật toán sắp xếp trong. Cụ thể dữ liệu sắp xếp sẽ là một mảng các bản ghi (gồm hai trường chính là trường dữ liệu và trường khóa), và để tập trung vào các thuật toán chúng ta chỉ xem xét các trường khóa của các bản ghi này, các ví dụ minh họa và cài đặt đều được thực hiện trên các mảng số nguyên, coi như là trường khóa của các bản ghi.

2. Sắp xếp gián tiếp

Khi các bản ghi có kích thước lớn việc hoán đổi các bản ghi là rất tốn kém, khi đó để giảm chi phí người ta có thể sử dụng các phương pháp sắp xếp gián tiếp. Việc này có thể được thực hiện theo nhiều cách khác nhau và môt trong những phương pháp đó là tạo ra một file mới chứa các trường khóa của file ban đầu, hoặc con trỏ tới hoặc là chỉ số của các bản ghi ban đầu. Chúng ta sẽ sắp xếp trên file mới này với các bản ghi có kích thước nhỏ và sau đó truy cập vào các bản ghi trong file ban đầu thông qua các con trỏ hoặc chỉ số (đây là cách làm thường thấy đối với các hệ quản trị cơ sở dữ liệu).

Ví dụ: chúng ta muốn sắp xếp các bản ghi của file sau đây:

Index	Dept	Last	First	Age	ID number
1	123	Smith	Jon	3	234-45-4586

2	23	Wilson	Pete	4	345-65-0697
3	2	Charles	Philip	9	508-45-6859
4	45	Shilst	Greg	8	234-45-5784

Index	Key
1	123
2	23
3	2
4	45

$\xrightarrow{Sort}$

Index	Key
3	2
2	23
4	45
1	123

Sau khi sắp xếp xong để truy cập vào các bản ghi theo thứ tự đã sắp xếp chúng ta sử dụng thứ tự được cung cấp bởi cột index (chỉ số). Trong trường hợp này là 3, 2, 4, 1. (chúng ta không nhất thiết phải hoán đổi các bản ghi ban đầu).

3. Các tiêu chuẩn đánh giá một thuật toán sắp xếp

Các thuật toán sắp xếp có thể được so sánh với nhau dựa trên các yếu tố sau đây:

- Thời gian thực hiện (run-time): số các thao tác thực hiện (thường là số các phép so sánh và hoán đổi các bản ghi).
- Bộ nhớ sử dụng (Memory): là dung lượng bộ nhớ cần thiết để thực hiện thuật toán ngoài dung lượng bộ nhớ sử dụng để chứa dữ liệu cần sắp xếp.
    - Một vài thuật toán thuộc loại “in place” và không cần (hoặc cần một số cố định) thêm bộ nhớ cho việc thực hiện thuật toán.
    - Các thuật toán khác thường sử dụng thêm bộ nhớ tỉ lệ thuận theo hàm tuyến tính hoặc hàm mũ với kích thước file sắp xếp.
    - Tất nhiên là bộ nhớ sử dụng càng nhỏ càng tốt mặc dù việc cân đối giữa thời gian và bộ nhớ cần thiết có thể là có lợi.
- Sự ổn định (Stability):Một thuật toán được gọi là ổn định nếu như nó có thể giữ được quan hệ thứ tự của các khóa bằng nhau (không làm thay đổi thứ tự của các khóa bằng nhau).

Chúng ta thường lo lắng nhiều nhất là về thời gian thực hiện của thuật toán vì các thuật toán mà chúng ta bàn về thường sử dụng kích thước bộ nhớ tương đương nhau.

Ví dụ về sắp xếp ổn định: Chúng ta muốn sắp xếp file sau đây dự trên ký tự đầu của các bản ghi và dưới đây là kết quả sắp xếp của các thuật toán ổn định và không ổn định:

File	Stable sorting	Unstable sorting
A Mary	A Mary	A Michel
B Cindy	A Michel	A Peter
A Michel	A Peter	A Mary
B Diana	B Cindy	B Tony
A Peter	B Diana	B Diana
B Tony	B Tony	B Cindy

Chúng ta sẽ xem xét tại sao tính ổn định trong các thuật toán sắp xếp lại được đánh giá quan trọng như vậy.

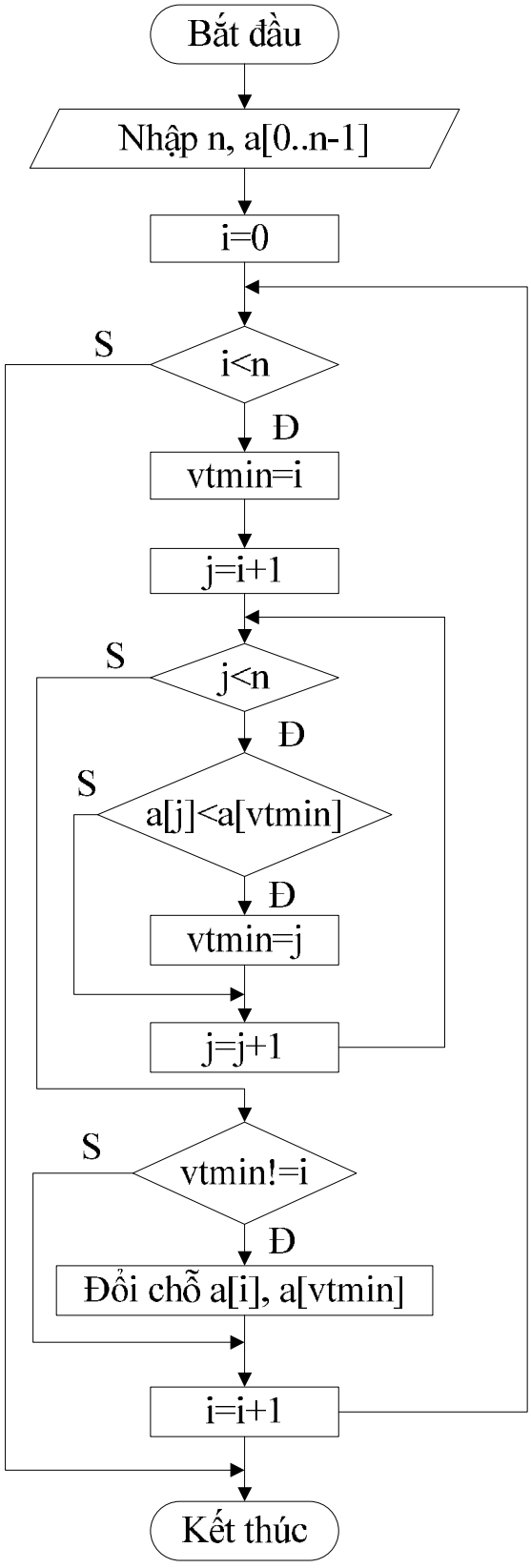
4. Các phương pháp sắp xếp cơ bản

4.1. Sắp xếp chọn (Selection sort)

Mô tả thuật toán:

Tìm phần tử có khóa lớn nhất (nhỏ nhất), đặt nó vào đúng vị trí và sau đó sắp xếp phần còn lại của mảng.

Sơ đồ thuật toán:

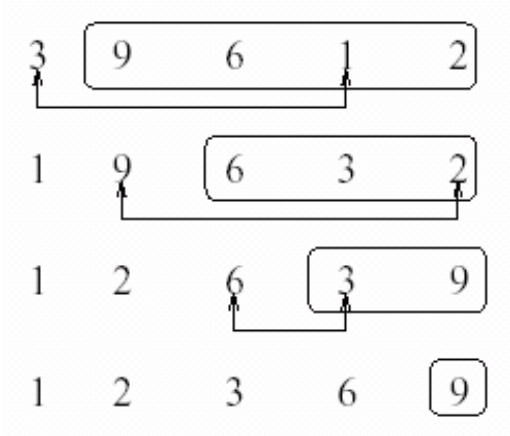


Đoạn mã sau minh họa cho thuật toán:

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

```
        {
            vtmin = i; //biến vtmin lưu vị trí phần tử nhỏ nhất a[i..n-1]
            for(j=i+1;j<n;j++)
                if(a[j] < a[vtmin])
                    vtmin = j;
            swap(a[vtmin], a[i]); // hàm đổi chỗ a[vtmin], a[i]
        }
    }
```

Ví dụ:



Với mỗi giá trị của i thuật toán thực hiện (n – i – 1) phép so sánh và vì i chạy từ 0 cho tới (n–2), thuật toán sẽ cần (n-1) + (n-2) + … + 1 = n(n-1)/2 tức là $\boldsymbol{O}(n^2)$ phép so sánh. Trong mọi trường hợp số lần so sánh của thuật toán là không đổi. Mỗi lần chạy của vòng lặp đối với biến i, có thể có nhiều nhất một lần đổi chỗ hai phần tử nên số lần đổi chỗ nhiều nhất của thuật toán là n. Như vậy trong trường hợp tốt nhất, thuật toán cần 0 lần đổi chỗ, trung bình cần n/2 lần đổi chỗ và tồi nhất cần n lần đổi chỗ.

4.2. Sắp xếp đổi chỗ trực tiếp (Exchange sort)

Tương tự như thuật toán sắp xếp bằng chọn và rất dễ cài đặt (thường bị nhầm với thuật toán sắp xếp chèn) là thuật toán sắp xếp bằng đổi chỗ trực tiếp (một số tài liệu còn gọi là thuật toán Interchange sort hay Straight Selection Sort).

Mô tả: Bắt đầu xét từ phần tử đầu tiên a[i] với i = 0, ta xét tất cả các phần tử đứng sau a[i], gọi là a[j] với j chạy từ i+1 tới n-1 (vị trí cuối cùng). Với mỗi cặp a[i], a[j] đó, để ý là a[j] là phần tử đứng sau a[i], nếu a[j] < a[i], tức là xảy ra sai khác về vị trí thì ta sẽ đổi chỗ a[i], a[j].

Ví dụ minh họa: Giả sử mảng ban đầu là int a[] = {2, 6, 1, 19, 3, 12}. Các bước của thuật toán sẽ được thực hiện như sau:

i=0, j=2: 1, 6, 2, 19, 3, 12

i=1, j=2: 1, 2, 6, 19, 3, 12

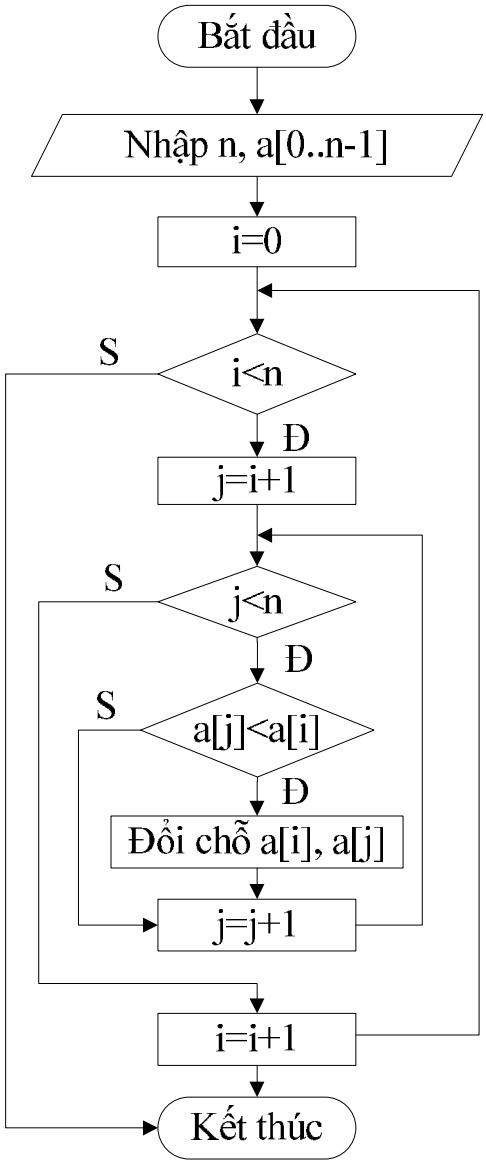
i=2, j=4: 1, 2, 3, 19, 6, 12

i=3, j=4: 1, 2, 3, 6, 19, 12

i=4, j=5: 1, 2, 3, 6, 12, 19

Kết quả cuối cùng: 1, 2, 3, 6, 12, 19.

Sơ đồ thuật toán:



Cài đặt của thuật toán:

```
void exchange_sort(int a[], int n)
{
        int i, j;
        int tam;
        for(i=0; i<n-1;i++)
                for(j=i+1;j<n;j++)
                if(a[j] < a[i])
                {
                        // đổi chỗ a[i], a[j]
```

```
                    tam = a[i];
                    a[i] = a[j];
                    a[j] = tam;
                }
        }
```

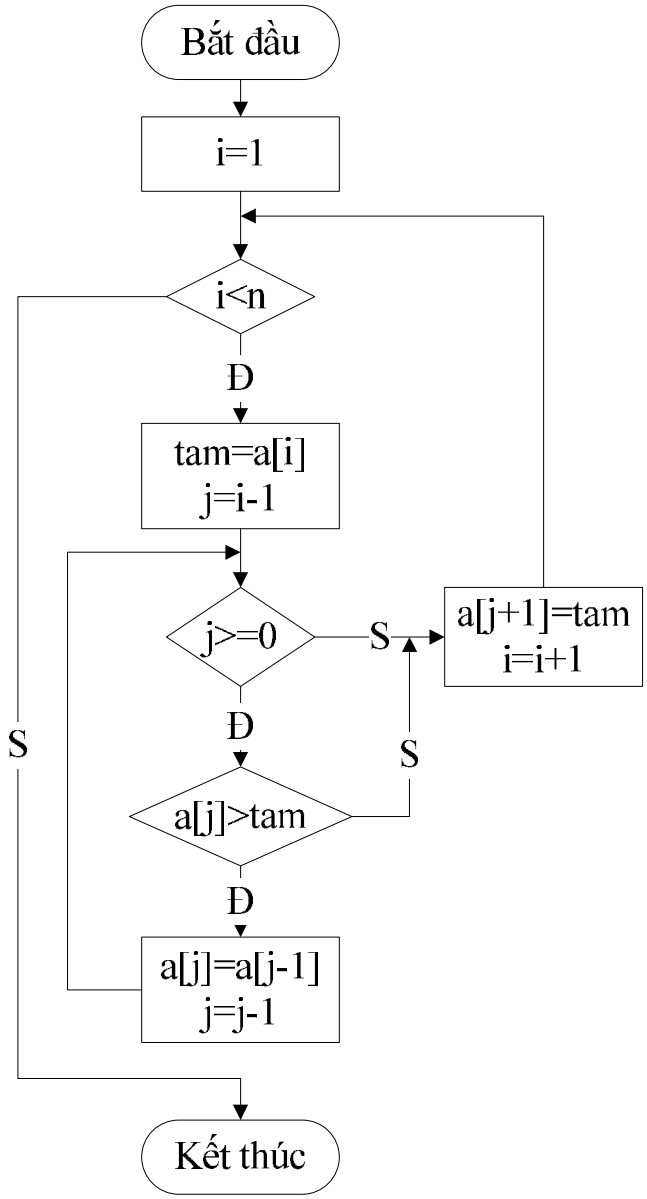
Độ phức tạp của thuật toán: Có thể thấy rằng so với thuật toán sắp xếp chọn, thuật toán sắp xếp bằng đổi chỗ trực tiếp cần số bước so sánh tương đương: tức là n*(n-1)/2 lần so sánh. Nhưng số bước đổi chỗ hai phần tử cũng bằng với số lần so sánh: n*(n-1)/2. Trong trường hợp xấu nhất số bước đổi chỗ của thuật toán bằng với số lần so sánh, trong trường hợp trung bình số bước đổi chỗ là n*(n-1)/4. Còn trong trường hợp tốt nhất, số bước đổi chỗ bằng 0. Như vậy thuật toán sắp xếp đổi chỗ trực tiếp nói chung là chậm hơn nhiều so với thuật toán sắp xếp chọn do số lần đổi chỗ nhiều hơn.

4.3. Sắp xếp chèn (Insertion sort)

Mô tả thuật toán:

Thuật toán dựa vào thao tác chính là chèn mỗi khóa vào một dãy con đã được sắp xếp của dãy cần sắp. Phương pháp này thường được sử dụng trong việc sắp xếp các cây bài trong quá trình chơi bài.

Sơ đồ giải thuật của thuật toán như sau:



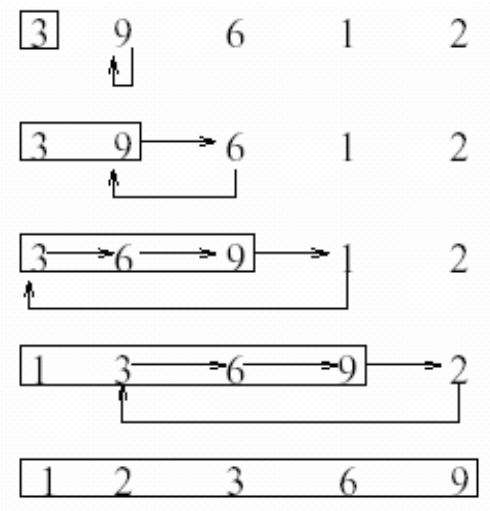
Có thể mô tả thuật toán bằng lời như sau: ban đầu ta coi như mảng a[0..i-1] (gồm i phần tử, trong trường hợp đầu tiên i = 1) là đã được sắp, tại bước thứ i của thuật toán, ta sẽ tiến hành chèn a[i] vào mảng a[0..i-1] sao cho sau khi chèn, các phần tử vẫn tuân theo thứ tự tăng dần. Bước tiếp theo sẽ chèn a[i+1] vào mảng a[0..i] một cách tương tự. Thuật toán cứ thế tiến hành cho tới khi hết mảng (chèn a[n-1] vào mảng a[0..n-2]). Để tiến hành chèn a[i] vào mảng a[0..i-1], ta dùng một biến tạm lưu a[i], sau đó dùng một biến chỉ số j = i-1, dò từ vị trí j cho tới đầu mảng, nếu a[j] > tam thì sẽ copy a[j] vào a[j+1], có nghĩa là lùi mảng lại một vị trí để chèn tam vào mảng. Vòng lặp sẽ kết thúc nếu a[j] < tam hoặc j = -1, khi đó ta gán a[j+1] = tam.

Đoạn mã chương trình như sau:

```
void insertion_sort(int a[], int n)
{
        int i, j, temp;
        for(i=1;i<n;i++)
        {
```

```
                int j = i;
                temp = a[i];
                while(j>0 && temp < a[j-1])
                {
                        a[j] = a[j-1];
                        j = j-1;
                }
                a[j] = temp;
        }
}
```

Ví dụ:



Thuật toán sắp xếp chèn là một thuật toán sắp xếp ổn định (stable) và là thuật toán nhanh nhất trong số các thuật toán sắp xếp cơ bản.

Với mỗi i chúng ta cần thực hiện so sánh khóa hiên tại (a[i]) với nhiều nhất là i khóa và vì i chạy từ 1 tới n-1 nên chúng ta phải thực hiện nhiều nhất: 1 + 2 + … + n-1 = n(n-1)/2 tức là $\boldsymbol{O}(n^2)$ phép so sánh tương tự như thuật toán sắp xếp chọn. Tuy nhiên vòng lặp while không phải lúc nào cũng được thực hiện và nếu thực hiện thì cũng không nhất định là lặp i lần nên trên thực tế thuật toán sắp xếp chèn nhanh hơn so với thuật toán sắp xếp chọn. Trong trường hợp tốt nhất, thuật toán chỉ cần sử dụng đúng n lần so sánh và 0 lần đổi chỗ. Trên thực tế một mảng bất kỳ gồm nhiều mảng con đã được sắp nên thuật toán chèn hoạt động khá hiệu quả. Thuật toán sắp xếp chèn là thuật toán nhanh nhất trong các thuật toán sắp xếp cơ bản (đều có độ phức tạp $\boldsymbol{O}(n^2)$).

4.4. Sắp xếp nổi bọt (Bubble sort)

Mô tả thuật toán:

Thuật toán sắp xếp nổi bọt dựa trên việc so sánh và đổi chỗ hai phần tử ở kề nhau:

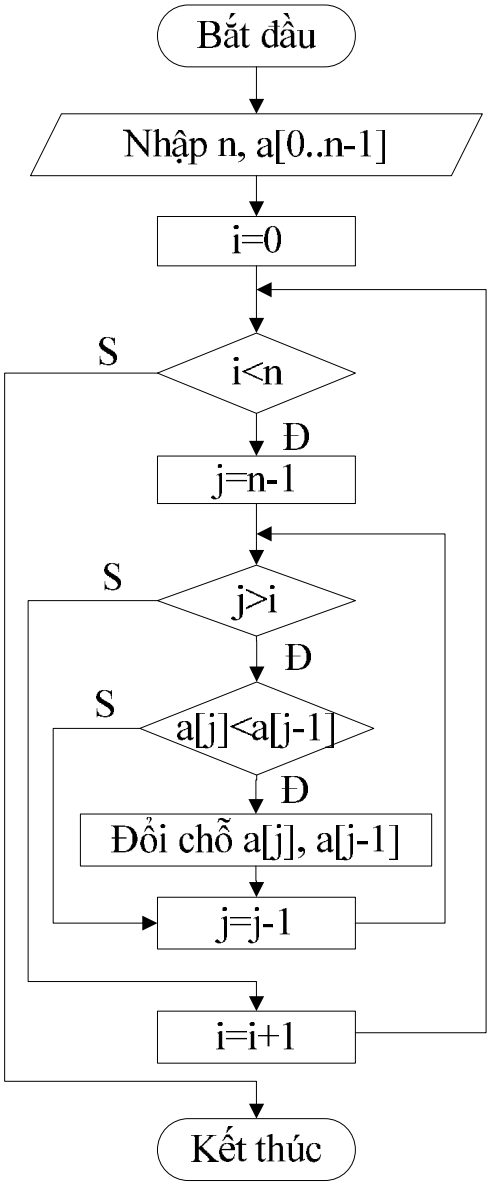
- Duyệt qua danh sách các bản ghi cần sắp theo thứ tự, đổi chỗ hai phần tử ở kề nhau nếu chúng không theo thứ tự.

- Lặp lại điều này cho tới khi không có hai bản ghi nào sai thứ tự.

Không khó để thấy rằng n pha thực hiện là đủ cho việc thực hiện xong thuật toán.

Thuật toán này cũng tương tự như thuật toán sắp xếp chọn ngoại trừ việc có thêm nhiều thao tác đổi chỗ hai phần tử.

Sơ đồ thuật toán:



Cài đặt thuật toán:

```
void bubble_sort1(int a[], int n)
{
        int i, j;
        for(i=n-1;i>=0;i--)
                for(j=1;j<=i;j++)
                if(a[j-1]>a[j])
                        swap(a[j-1],a[j]);
```

```
}
void bubble_sort2(int a[], int n)
{
        int i, j;
        for(i=0;i<n;i++)
                for(j=n-1;j>i;j--)
                if(a[j-1]>a[j])
                        swap(a[j-1],a[j]);
}
```

Thuật toán có độ phức tạp là ***O***(N*(N-1)/2) = ***O***(N^2), bằng số lần so sánh và số lần đổi chỗ nhiều nhất của thuật toán (trong trường hợp tồi nhất). Thuật toán sắp xếp nổi bọt là thuật toán chậm nhất trong số các thuật toán sắp xếp cơ bản, nó còn chậm hơn thuật toán sắp xếp đổi chỗ trực tiếp mặc dù có số lần so sánh bằng nhau, nhưng do đổi chỗ hai phần tử kề nhau nên số lần đổi chỗ nhiều hơn.

4.5. So sánh các thuật toán sắp xếp cơ bản

Sắp xếp chọn:

- Trung bình đòi hỏi $n^2/2$ phép so sánh, n bước đổi chỗ.
- Trường hợp xấu nhất tương tự.

Sắp xếp chèn:

- Trung bình cần n2/4 phép so sánh, n2/8 bước đổi chỗ.
- Xấu nhất cần gấp đôi các bước so với trường hợp trung bình.
- Thời gian là tuyến tính đối với các file hầu như đã sắp và là thuật toán nhanh nhất trong số các thuật toán sắp xếp cơ bản.

Sắp xếp nổi bọt:

- Trung bình cần n2/2 phép so sánh, n2/2 thao tác đổi chỗ.
- Xấu nhất cũng tương tự.

5. Các phương pháp sắp xếp nâng cao

Các thuật toán sắp xếp tốt nhất đều là các thuật toán đệ qui. Chúng đều tuân theo chiến lược chung sau đây:

Cho một danh sách các bản ghi L.

- Nếu L có không nhiều hơn 1 phần tử thì có nghĩa là nó đã được sắp
- Ngược lại
    - Chia L thành hai dãy nhỏ hơn là L1, L2
    - Sắp xếp L1, L2 (đệ qui – gọi tới thủ tục này)
    - Kết hợp L1 và L2 để nhận được L đã sắp

Các thuật toán sắp xếp trộn và sắp xếp nhanh đều sử dụng kỹ thuật này.

5.1. Sắp xếp nhanh (Quick sort)

Quick sort là thuật toán sắp xếp được C. A. R. Hoare đưa ra năm 1962.

Quick sort là một thuật toán sắp xếp dạng chia để trị với các bước thực hiện như sau:

- Selection: chọn một phần tử gọi là phần tử quay (pivot)
- Partition (phân hoạch): đặt tất cả các phần tử của mảng nhỏ hơn phần tử quay sang bên trái phần tử quay và tất cả các phần tử lớn hơn phần tử quay sang bên phải phần tử quay. Phần tử quay trở thành phần tử có vị trí đúng trong mảng.
- Đệ qui: gọi tới chính thủ tục sắp xếp nhanh đối với hai nửa mảng nằm 2 bên phần tử quay

Thuật toán:

```
void quicksort(int *A, int l, int r)
{
        if(r>l)
        {
                int  p =partition(A, l, r);
                quicksort(A, l, p -1);
                quicksort(A, p+1, r);
        }
}
```

Hàm phân hoạch partition:

- Lấy một số k: $l \le k \le r$.
- Đặt x = A[k] vào vị trí đúng của nó là p
- Giả sử $A[j] \le A[p]$ nếu $j < p$
- $A[j] \ge A[p]$ nếu $j > p$

Đây không phải là cách duy nhất để định nghĩa Quicksort. Một vài phiên bản của thuật toán quick sort không sử dụng phần tử quay thay vào đó định nghĩa các mảng con trái và mảng con phải, và giả sử các phần tử của mảng con trái nhỏ hơn các phần tử của mảng con phải.

Chọn lựa phần tử quay

Có rất nhiều cách khác nhau để lựa chọn phần tử quay:

- Sử dụng phần tử trái nhất để làm phần tử quay
- Sử dụng phương thức trung bình của 3 để lấy phần tử quay
- Sử dụng một phần tử ngẫu nhiên làm phần tử quay.

Sau khi chọn phần tử quay làm thế nào để đặt nó vào đúng vị trí và bảo đảm các tính chất của phân hoạch? Có một vài cách để thực hiện điều này và chúng ta sử dụng phương thức chọn phần tử quay là phần tử trái nhất của mảng. Các phương thức khác cũng có thể cài đặt bằng cách sử đổi đôi chút phương thức này.

Hàm phân hoạch:

```
int  partition(int *A, int l, int r)
{
        int p = A[l];
        int i = l+1;
        int j = r;
        while(1){
                while(A[i] ≤ p && i<r)
                        ++i;
                while(A[j] ≥ p && j>l)
                        --j;
                if(i>=j)
                {
                        swap(A[j], A[l]);
                        return j;
                }else
                        swap(A[i], A[j]);
        }
}
```

Để gọi tới hàm trên sắp xếp cho mảng a có n phần tử ta gọi hàm như sau:

quicksort(a, 0, n-1);

Trong thủ tục trên chúng ta chọn phần tử trái nhất của mảng làm phần tử quay, chúng ta duyệt từ hai đầu vào giữa mảng và thực hiện đổi chỗ các phần tử sai vị trí (so với phần tử quay).

Các phương pháp lựa chọn phần tử quay khác:

Phương pháp ngẫu nhiên:

Chúng ta chọn một phần tử ngẫu nhiên làm phần tử quay

Độ phức tạp của thuật toán khi đó không phụ thuộc vào sự phân phối của các phần tử input

Phương pháp 3-trung bình:

Phần tử quay là phần tử được chọn trong số 3 phần tử a[l], a[(l+r)/2] hoặc a[r] gần với trung bình cộng của 3 số nhất.

Hãy suy nghĩ về các vấn đề sau:

Sửa đổi cài đặt của thủ tục phân hoạch lựa chọn phần tử trái nhất để nhận được cài đặt của 2 phương pháp trên

Có cách cài đặt nào tốt hơn không?

Có cách nào tốt hơn để chọn phần tử phân hoạch?

Các vấn đề khác:

Tính đúng đắn của thuật toán, để xem xét tính đúng đắn của thuật toán chúng ta cần xem xét 2 yếu tố: thứ nhất do thuật toán là đệ qui vậy cần xét xem nó có dừng không, thứ hai là khi dừng thì mảng có thực sự đã được sắp hay chưa.

Tính tối ưu của thuật toán. Điều gì sẽ xảy ra nếu như chúng ta sắp xếp các mảng con nhỏ bằng một thuật toán khác? Nếu chúng ta bỏ qua các mảng con nhỏ? Có nghĩa là chúng ta chỉ sử dụng quicksort đối với các mảng con lớn hơn một ngưỡng nào đó và sau đó có thể kết thúc việc sắp xếp bằng một thuật toán khác để tăng tính hiệu quả?

Độ phức tạp của thuật toán:

Thuật toán phân hoạch có thể được thực hiện trong ***O***(n). Chi phí cho các lời gọi tới thủ tục phân hoạch tại bất cứ độ sâu nào theo đệ qui đều có độ phức tạp là ***O***(n). Do đó độ phức tạp của quicksort là độ phức tạp của thời gian phân hoạch độ sau của lời gọi đệ qui xa nhất.

Kết quả chứng minh chặt chẽ về mặt toán học cho thấy Quick sort có độ phức tạp là ***O***(n*log(n)), và trong hầu hết các trường hợp Quick sort là thuật toán sắp xếp nhanh nhất, ngoại trừ trường hợp tồi nhất, khi đó Quick sort còn chậm hơn so với Bubble sort.

5.2. Sắp xếp trộn (merge sort)

Về mặt ý tưởng thuật toán merge sort gồm các bước thực hiện như sau:

- Chia mảng cần sắp xếp thành 2 nửa
- Sắp xếp hai nửa đó một cách đệ qui bằng cách gọi tới thủ tục thực hiện chính mergesort
- Trộn hai nửa đã được sắp để nhận được mảng được sắp.

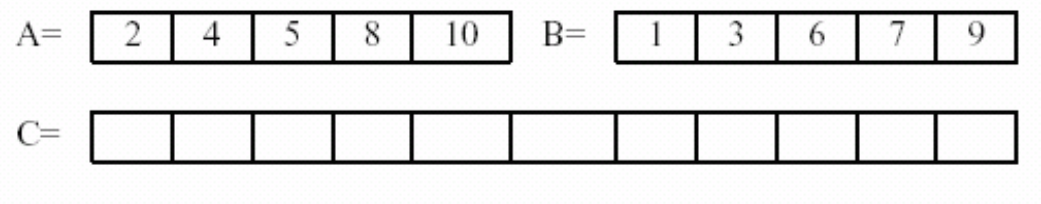
Đoạn mã C thực hiện thuật toán Merge sort:

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

Để sắp một mảng a có n phần tử ta gọi hàm như sau: merge_sort(a, 0, n-1);

Nhưng thuật toán trộn làm việc như thế nào?

Ví dụ với 2 mảng sau:



Thuật toán 1:

Thuật toán trộn nhận 2 mảng con đã được sắp và tạo thành một mảng được sắp. Thuật toán 1 làm việc như sau:

- Đối với mỗi mảng con chúng ta có một con trỏ trỏ tới phần tử đầu tiên
- Đặt phần tử nhỏ hơn của các phần tử đang xét ở hai mảng vào mảng mới
- Di chuyển con trỏ của các mảng tới vị trí thích hợp
- Lặp lại các bước thực hiện trên cho tới khi mảng mới chứa hết các phần tử của hai mảng con

Đoạn mã C++ thực hiện thuật toán trộn hai mảng A, B thành mảng C:

```
int p1 = 0, p2 = 0, index = 0;
int n = sizeA + sizeB;
while(index<n)
{
	if(A[p1] < B[p2]){
		C[index] = A[p1];
		p1++;
		index++;
	}else{
		C[index] = A[p2];
		p2++;
		index++;
	}
}
```

Thuật toán 2:

Thuật toán 1 giả sử chúng ta có 3 mảng phân biệt nhưng trên thực tế chúng ta chỉ có 2 mảng hay chính xác là 2 phần của một mảng lớn sau khi trộn lại thành mảng đã sắp thì đó cũng chính là mảng ban đầu.

Chúng ta sẽ sử dụng thêm một mảng phụ:

```
void merge(int *A, int l, int m, int r)
{
	int *B1 = new int[m-l+1];
	int *B2 = new int[r-m];
```

```
        for(int i=0;i<m-l+1;i++)
                B1[i] = a[i+l];
        for(int i=0;i<r-m;i++)
                B2[i] = a[i+m+l];
        int b1=0,b2=0;
        for(int k=l;k<=r;k++)
                if(B1[b1]<B2[b2])
                        a[k] = B1[b1++];
                else
                        a[k] = B2[b2++];
}
```

Thuật toán 3:

Thuật toán 2 có vẻ khá phù hợp so với mục đích của chúng ta, nhưng cả hai phiên bản của thuật toán trộn trên đều có một nhược điểm chính đó là không kiểm tra các biên của mảng. Có 3 giải pháp chúng ta có thể nghĩ tới:

- Thực hiện kiểm tra biên một cách cụ thể
- Thêm 1 phần tử lính canh vào đầu của mỗi mảng input.
- Làm gì đó thông minh hơn

Và đây là một cách để thực hiện điều đó:

```
void merge(int *A,int l,int m,int r)
{
        int *B=new int[r-l+1];
        for (i=m; i>=l; i--)
                B[i-l] = A[i];
        for (j=m+1; j<=r; j++)
                B[j-l] = A[j];
        for (k=l;k<=r;k++)
                if(((B[i] < B[j])&&(i<m))||(j==r+1))
                        A[k]=B[i++];
                else
                        A[k]=B[j--];
}
```

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

```
merge_sort(a, 0, n-1);
```

Độ phức tạp của thuật toán sắp xếp trộn:

Gọi T(n) là độ phức tạp của thuật toán sắp xếp trộn. Thuật toán luôn chia mảng thành 2 nửa bằng nhau nên độ sâu đệ qui của nó luôn là ***O***(log n). Tại mỗi bước công việc thực hiện có độ phức tạp là ***O***(n) do đó:

T(n) = ***O***(n*log(n)).

Bộ nhớ cần dùng thêm là ***O***(n), đây là một con số chấp nhận được, một trong các đặc điểm nổi bật của thuật toán là tính ổn định của nó, ngoài ra thuật toán này là phù hợp cho các ứng dụng đòi hỏi sắp xếp ngoài.

Chương trình hoàn chỉnh:

```
void merge(int *a,int l,int m,int r)
{
        int *b=new int[r-l+1];
        int i,j,k;
        i = l;
        j = m+1;
        for (k=l;k<=r;k++)
                if((a[i] < a[j])||(j>r))
                        b[k]=a[i++];
                else
                        b[k]=a[j++];
        for(k=l;k<=r;k++)
                a[k] = b[k];
}
void mergesort(int *a, int l, int r)
{
        int mid;
        if(l>=r)
                return;
        mid = (l+r)/2;
        mergesort(a, l, mid);
        mergesort(a, mid+1, r);
        merge(a, l, mid, r);
}
```

Các chứng minh chặt chẽ về mặt toán học cho kết quả là Merge sort có độ phức tạp là ***O***(n*log(n)). Đây là thuật toán ổn định nhất trong số các thuật toán sắp xếp dựa trên so sánh và đổi chỗ các phần tử, nó cũng rất thích hợp cho việc thiết kế các giải thuật sắp xếp

ngoài. So với các thuật toán khác, Merge sort đòi hỏi sử dụng thêm một vùng bộ nhớ bằng với mảng cần sắp xếp.

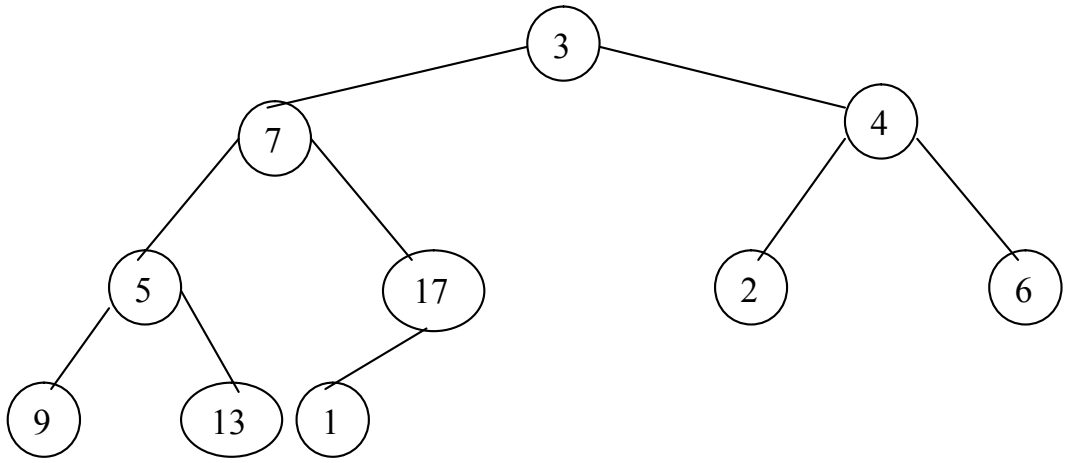
5.3. Cấu trúc dữ liệu Heap, sắp xếp vun đống (Heap sort).

5.3.1. Cấu trúc Heap

Trước khi tìm hiểu về thuật toán heap sort chúng ta sẽ tìm hiểu về một cấu trúc đặc biệt gọi là cấu trúc Heap (heap data structure, hay còn gọi là đống).

Heap là một cây nhị phân đầy đủ và tại mỗi nút ta có key(child) ≤ key(parent). Hãy nhớ lại một cây nhị phân đầy đủ là một cây nhị phân đầy ở tất cả các tầng của cây trừ tầng cuối cùng (có thể chỉ đầy về phía trái của cây). Cũng có thể mô tả kỹ hơn là một cây nhị phân mà các nút có đặc điểm sau: nếu đó là một nút trong của cây và không ở mức cuối cùng thì nó sẽ có 2 con, còn nếu đó là một nút ở mức cuối cùng thì nó sẽ không có con nào nếu nút anh em bên trái của nó không có con hoặc chỉ có 1 con và sẽ có thể có con (1 hoặc 2) nếu như nút anh em bên trái của nó có đủ 2 con, nói tóm lại là ở mức cuối cùng một nút nếu có con sẽ có số con ít hơn số con của nút anh em bên trái của nó.

Ví dụ:



Chiều cao của một heap:

Một heap có n nút sẽ có chiều cao là O(log n).

Chứng minh:

Giả sử n là số nút của một heap có chiều cao là h.

Vì một cây nhị phân chiều cho h có số nút tối đa là 2^h -1 nên suy ra:

$$2^{h-1} \le n \le 2^h - 1$$

Lấy logarit hai vế của bất đẳng thức thứ nhất ta được:

$$h - 1 \le \log n$$

Thêm 1 vào 2 vế của bất đẳng thức còn lại và lấy logarit hai vế ta lại được:

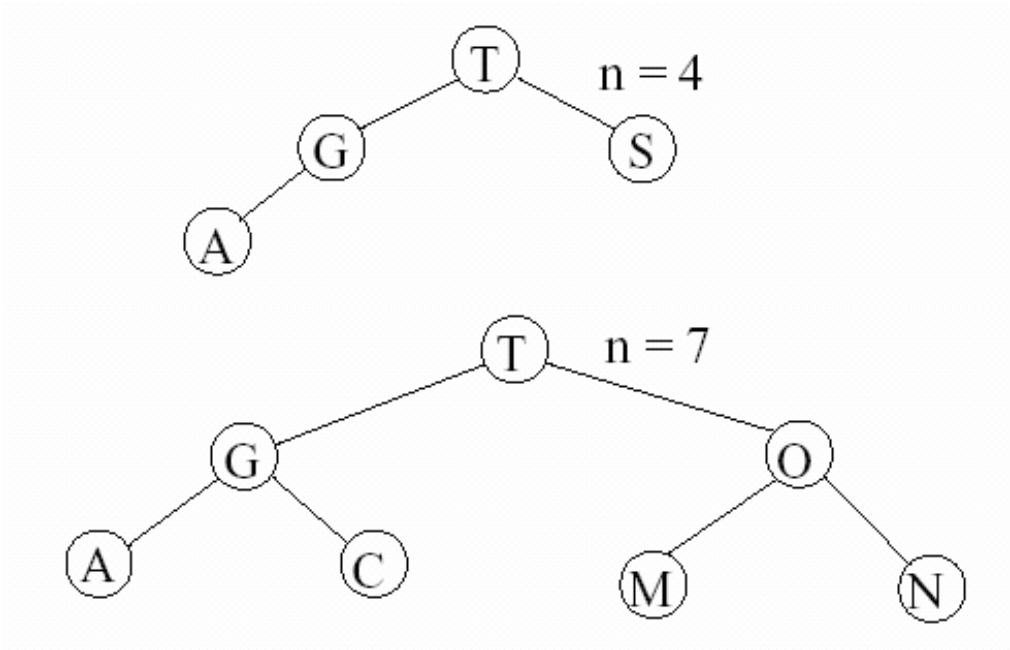
$$\log(n + 1) \le h$$

Từ 2 điều trên suy ra:

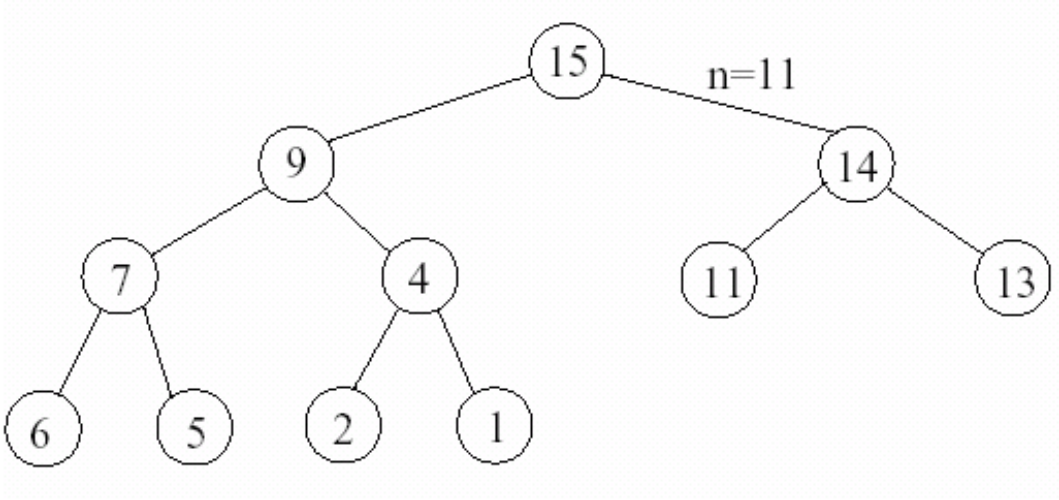
$$\log(n + 1) \le h \le \log(n) + 1$$

Các ví dụ về cấu trúc Heap:

Heap với chiều cao h = 3:



heap với chiều cao h = 4



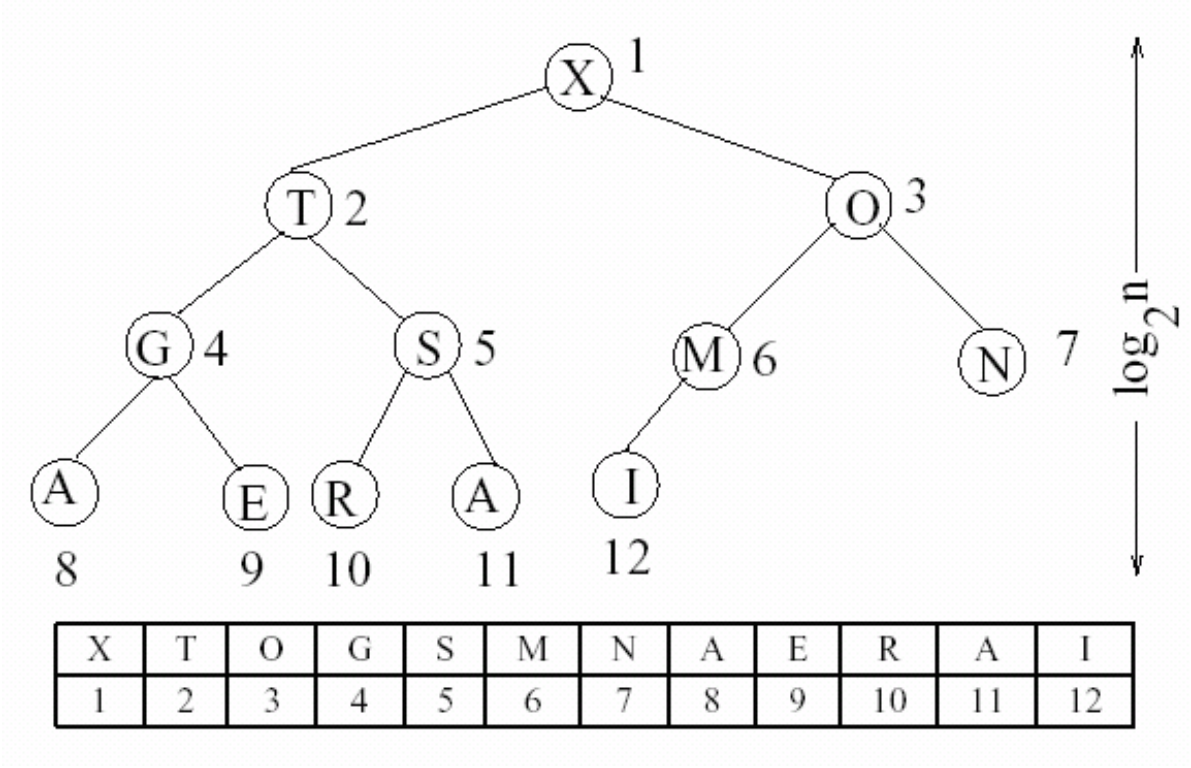
Biểu diễn Heap

Chúng ta đã biết các biểu diễn bằng một cây nhị phân nên việc biểu diễn một heap cũng không quá khó, cũng tương tự giống như biểu diễn một cây nhị phân bằng một mảng.

Đối với một heap lưu trong một mảng chúng ta có quan hệ sau (giả sử chúng ta bắt đầu bằng 0):

- Left(i) = 2*i + 1
- Right(i) = 2*i + 2
- Parent(i) = (i-1)/2

Ví dụ:



X	T	O	G	S	M	N	A	E	R	A	I
1	2	3	4	5	6	7	8	9	10	11	12

Thủ tục heaprify

Đây là thủ tục cơ bản cho tất cả các thủ tục khác thao tác trên các heap

Input:

- Một mảng A và một chỉ số i trong mảng
- Giả sử hai cây con Left(i) và Right(i) đều là các heap
- A[i] có thể phá vỡ cấu trúc Heap khi tạo thành cây với Left(i) và Right(i).

Output:

- Mảng A trong đó cây có gốc là tại vị trí i là một Heap

Không quá khó để nhận ra rằng thuật toán này có độ phức tạp là ***O***(log n).

Chúng ta sẽ thấy đây là một thủ tục rất hữu ích, tạm thời hãy tưởng tượng là nếu chúng ta thay đổi giá trị của một vài khóa trong heap cấu trúc của heap sẽ bị phá vỡ và điều này đòi hỏi phải có sự sửa đổi.

Sau đây là cài đặt bằng C của thủ tục:

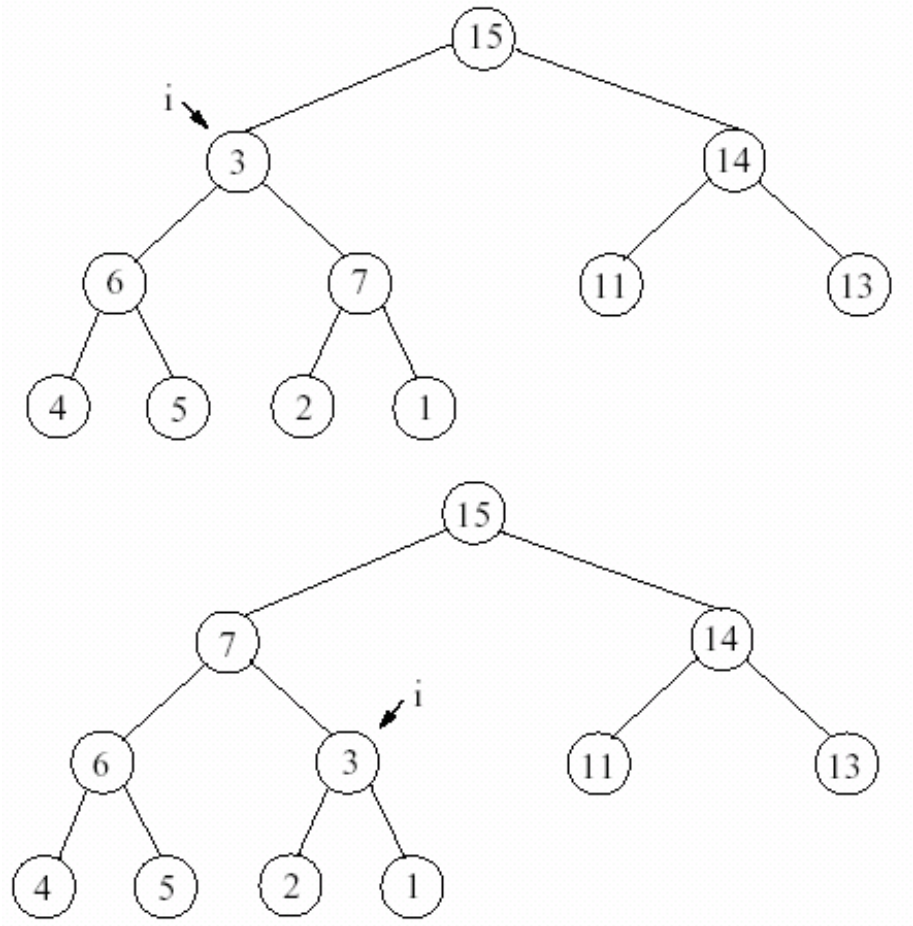
```
void heaprify(int *A, int i, int n)
{
        l = Left(i); /* l = 2*i +1*/
        r = Right(i); /* r = 2*i + 2 */
        if(l < n && A[l] > A[i])
                largest = l;
        else
                largest = i;
```

```
        if(r < n && A[r] > A[largest])
                largest = r;
        if(lagest !=i)
        {
                swap(A[i], A[largest]);
                heaprify(A, largest, n);
        }
}
```

Về cơ bản đây là một thủ tục đơn giản hơn nhiều so với những gì chúng ta cảm nhận về nó. Thủ tục này đơn giản với các bước như sau:

- Xác định phần tử lớn nhất trong 3 phần tử A[i], A[Left(i)], A[Right(i)].
- Nếu A[i] không phải là phần tử lớn nhất trong 3 phần tử trên thì đổi chỗ A[i] với A[largest] trong đó A[largest] sẽ là A[Left(i)] hoặc A[Right(i)].
- Gọi thủ tục với nút largest (vì việc đổi chỗ có thể làm thay đổi tính chất của heap có đỉnh là A[largest]).

Ví dụ:



Thủ tục buildheap

Thủ tục buildheap sẽ chuyển một mảng bất kỳ thành một heap. Về cơ bản thủ tục này thực hiện gọi tới thủ tục heaprify trên các nút theo thứ tự ngược lại. Và vì chạy theo thứ

tự ngược lại nên chúng ta biết rằng các cây con có gốc tại các đỉnh con là các heap. Nửa cuối của mảng tương ứng với các nút lá nên chúng ta không cần phải thực hiện thủ tục tạo heap đối với chúng.

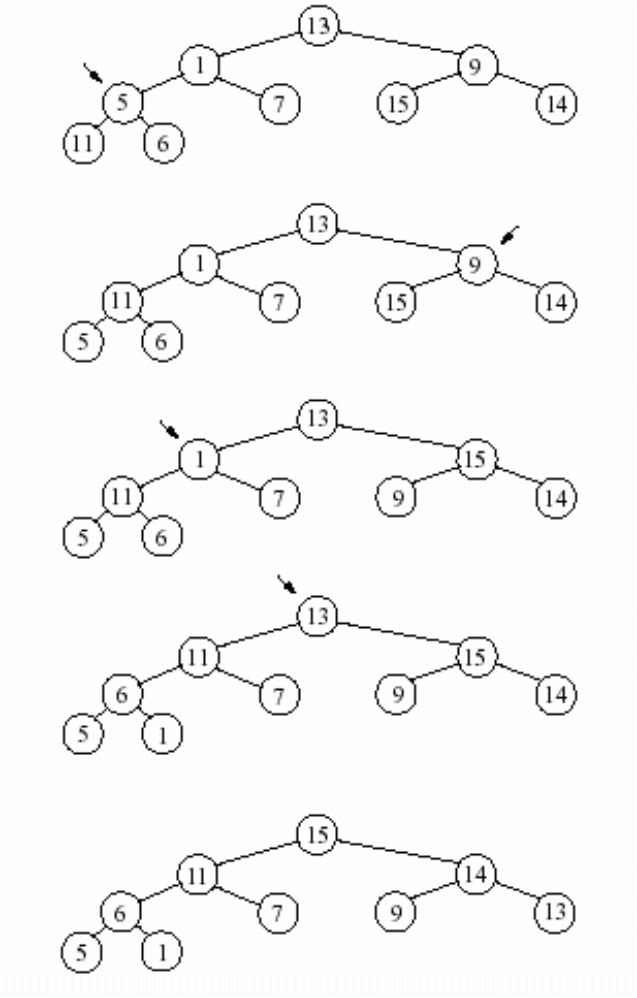
Đoạn mã C thực hiện buildheap:

```
void buildheap(int *a, int n)
{
        int i;
        for(i=n/2; i>=0; i--)
                heaprify(a, i, n);
}
```

Dựa và thuật toán này dễ thấy thuật toán tạo heap từ mảng có độ phức tạp là ***O***(n*log n). Trên thực tế thuật toán tạo heap có độ phức tạp là ***O***(n). Thời gian thực hiện của thuật toán heaprify trên một cây con có kích thước n tại một nút cụ thể i nào đó để chỉnh lại mối quan hệ giữa các phần tử tại a[i], a[Left(i)] và a[Right(i)] là ***O***(1). Cộng thêm với thời gian thủ tục này thực hiện trên một cây con có gốc tại một trong các nút là con của nút i. Số cây con của các con của nút i (i có thể là gốc) nhiều nhất là 2n/3. Suy ra ta có công thức tính độ phức tạp của thuật toán là: T(n) = T(2n/3) + ***O***(1) do đó T(n) = ***O***(log n), từ đây cũng suy ra độ phức tạp của thuật toán buildheap là n*log(n). Cũng có thể lý luận khác như sau: Kích thước của các cấp của cây là: n/4, n/8, n/16, …, 1 trong đó n là số nút của cây. Thời gian để tạo thực hiện thuật toán heaprify đối với các kích thước này nhiều nhất là 1, 2, 3, …, log(n) – 1, vì thế thời gian tổng sẽ xấp xỉ là:

1*n/4 + 2 * n/8 + 3 * n/16 + … + (log(n)-1) * 1 < n/4(1 + 2* ½ + 3 * ¼ + 4 * 1/8 + ...) = ***O***(n).

Ví dụ:



Các thao tác trên heap khác.

Ngoài việc tạo heap các thao tác sau đây cũng thường thực hiện đối với một heap:

- Insert()
- Extract_Max()

Chúng ta không bàn về các thao tác này ở đây nhưng các thao tác này đều không khó thực hiện với việc sử dụng thủ tục heaprify mà chúng ta đã cài đặt ở trên. Với các thao tác này chúng ta có thể sử dụng một heap để cài đặt một hàng đợi ưu tiên. Một hàng đợi ưu tiên là một cấu trúc dữ liệu với các thao tác cơ bản là insert, maximum và extractmaximum và chúng ta sẽ bàn về chúng trong các phần sau của khóa học.

5.3.2. Sắp xếp vun đống (Heap sort)

Thuật toán Heap sort về ý tưởng rất đơn giản:

- Thực hiện thủ tục buildheap để biến mảng A thành một heap
- Vì A là một heap nên phần tử lớn nhất sẽ là A[1].
- Đổi chỗ A[0] và A[n-1], A[n-1] đã nằm đúng vị trí của nó và vì thế chúng ta có thể bỏ qua nó và coi như mảng bây giờ có kích thước là n-1 và quay trở lại xem xét phần đầu của mảng đã không là một heap nữa.
- Vì A[0] có thể lỗi vị trí nên ta sẽ gọi thủ tục heaprify đối với nó để chỉnh lại mảng trở thành một heap.

- Lặp lại các thao tác trên cho tới khi chỉ còn một phần tử trong heap khi đó mảng đã được sắp.

Cài đặt bằng C của thuật toán:

```
void heapsort(int *A, int n)
{
        int i;
        buildheap(A, n);
        for(i=n-1;i>0;i--)
        {
                swap(A[0], A[i]);
                heaprify(A, 0, i-1);
        }
}
```

Chú ý: Để gọi thuật toán sắp xếp trên mảng a có **n** phần tử gọi hàm heapsort() như sau: heapsort(a, n);

Độ phức tạp của thuật toán heapsort:

Thủ tục buildheap có độ phức tạp là O(n).

Thủ tục heaprify có độ phức tạp là O(log n).

Heapsort gọi tới buildheap 1 lần và n-1 lần gọi tới heaprify suy ra độ phức tạp của nó là O(n + (n-1)logn) = O(n*log n).

Trên thực tế heapsort không nhanh hơn quicksort.

6. Các vấn đề khác

Ngoài các thuật toán đã được trình bày ở trên vẫn còn có một số thuật toán khác mà chúng ta có thể tham khảo: chẳng hạn như thuật toán sắp xếp Shell sort, sắp xếp bằng đếm Counting sort hoặc sắp xếp cơ số Radix sort. Các thuật toán này được xem như phần tự tìm hiểu của sinh viên.

7. Bài tập

Bài tập 1: Cài đặt các thuật toán sắp xếp cơ bản bằng ngôn ngữ lập trình C trên 1 mảng các số nguyên, dữ liệu của chương trình được nhập vào từ file text được sinh ngẫu nhiên (số phần tử khoảng 10000) và so sánh thời gian thực hiện thực tế của các thuật toán.

Bài tập 2: Cài đặt các thuật toán sắp xếp nâng cao bằng ngôn ngữ C với một mảng các cấu trúc sinh viên (tên: xâu ký tự có độ dài tối đa là 50, tuổi: số nguyên, điểm trung bình: số thức), khóa sắp xếp là trường tên. So sánh thời gian thực hiện của các thuật toán, so sánh với hàm qsort() có sẵn của C.

Bài tập 3[6, trang 52]: Cài đặt của các thuật toán sắp xếp có thể thực hiện theo nhiều cách khác nhau. Hãy viết hàm nhận input là mảng a[0..i] trong đó các phần tử ở chỉ số 0 tới chỉ số i-1 đã được sắp xếp tăng dần, a[i] không chứa phần tử nào, và một số x, chèn x

vào mảng a[0..i-1] sao cho sau khi chèn kết quả nhận được là a[0..i] là một mảng được sắp xếp. Sử dụng hàm vừa xây dựng để cài đặt thuật toán sắp xếp chèn.

Gợi ý: Có thể cài đặt thuật toán chèn phần tử vào mảng như phần cài đặt của thuật toán sắp xếp chèn đã được trình bày hoặc sử dụng phương pháp đệ qui.

Chương 4: Các cấu trúc dữ liệu cơ bản

1. Ngăn xếp - Stack

1.1. Khái niệm

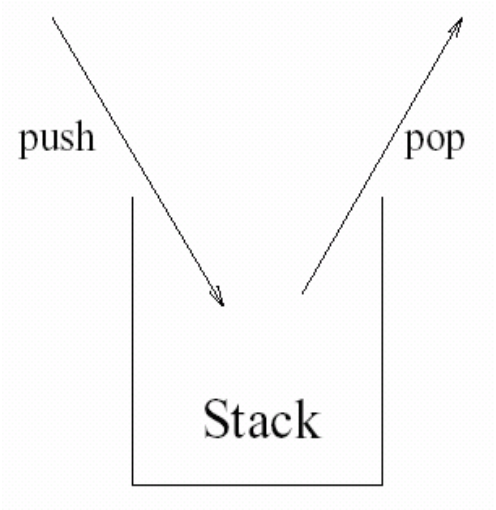
Khái niệm: Ngăn xếp (stack) là một tập hợp các phần tử (items) cùng kiểu được tổ chức một cách tuần tự (chính vì thế một số tài liệu còn định nghĩa ngăn xếp là một danh sách tuyến tính các phần tử với các thao tác truy cập hạn chế tới các phần tử của danh sách đó) trong đó phần tử được thêm vào cuối cùng của tập hợp sẽ là phần tử bị loại bỏ đầu tiên khỏi tập hợp. Các ngăn xếp thường được gọi là các cấu trúc LIFO (Last In First Out).

Ví dụ về ngăn xếp: Chồng tài liệu của một công chức văn phòng, chồng đĩa … là các ví dụ về ngăn xếp.

Chú ý: Phần tử duy nhất có thể truy cập tới của một ngăn xếp là phần tử mới được thêm vào gần đây nhất (theo thời gian) của ngăn xếp.

1.2. Các thao tác của ngăn xếp

Đối với một ngăn xếp chỉ có 2 thao tác cơ bản, thao tác thứ nhất thực hiện thêm một phần tử vào stack gọi là push, thao tác thứ hai là đọc giá trị của một phần tử và loại bỏ nó khỏi stack gọi là pop.



Để nhất quán với các thư viện cài đặt cấu trúc stack chuẩn STL (và một số tài liệu cũng phân chia như vậy), ta xác định các thao tác đối với một stack gồm có:

1. Thao tác push(d) sẽ đặt phần tử d lên đỉnh của stack.
2. Thao tác pop() loại bỏ phần tử ở đỉnh stack.
3. Thao tác top() sẽ trả về giá trị phần tử ở đỉnh stack.
4. Thao tác size() cho biết số phần tử hiện tại đang lưu trong stack

Ngoài hai thao tác cơ bản trên chúng ta cần có một số thao tác phụ trợ khác: chẳng hạn làm thế nào để biết là một stack không có phần tử nào – tức là rỗng (empty) hay là đầy (full) tức là không thể thêm vào bất cứ một phần tử nào khác nữa. Để thực hiện điều này người ta thường thêm hai thao tác tiến hành kiểm tra là empty() và full().

Để đảm bảo không xảy ra tình trạng gọi là stack overflow (tràn stack – không thể thêm vào stack bất cứ phần tử nào) chúng ta có thể cho hàm push trả về chẳng hạn 1 trong trường hợp thực hiện thành công và 0 nếu không thành công.

1.3. Ví dụ về hoạt động của một stack

Giả sử chúng ta có một stack kích thước bằng 3 (có thể chứa được tối đa 3 phần tử) và các phần tử của stack là các số nguyên trong khoảng từ -100 đến 100. Sau đây là minh họa các thao tác đối với stack và kết quả thực hiện của các thao tác đó.

Thao tác	Nội dung stack	Kết quả
Khởi tạo	()	
push(55)	(55)	1
push(-7)	(-7, 55)	1
push(16)	(16, -7, 55)	1
pop	(-7, 55)	16
push(-8)	(-8, -7, 55)	1
push(23)	(-8, -7, 55)	0
pop	(-7, 55)	-8
pop	(55)	-7
pop	()	55
pop	()	101

1.4. Cài đặt stack bằng mảng

Cấu trúc dữ liệu stack có thể cài đặt bằng cách sử dụng một mảng và một số nguyên top_idx để chứa chỉ số của phần tử ở đỉnh stack.

Ngăn xếp rỗng khi top_idx = -1 và đầy khi top_idx = n-1 trong đó n là kích thước của mảng.

Khi thực hiện thao tác push chúng ta tăng top_idx lên 1 và ghi dữ liệu vào vị trí tương ứng của mảng.

Khi thực hiện thao tác pop chúng ta chỉ việc giảm chỉ số top_idx đi 1.

Ví dụ về ngăn xếp cài đặt bằng mảng:

Giả sử chúng ta sử dụng mảng E[0..4] để chứa các phần tử của stack và biến top_idx để lưu chỉ số của phần tử ở đỉnh stack. Trong bảng sau cột cuối cùng kết quả là giá trị trả về của việc gọi hàm.

Thao tác	top_idx	E[0]	E[1]	E[2]	E[3]	E[4]	Kết quả
khởi tạo	-1	?	?	?	?	?	
push(55)	0	55	?	?	?	?	1
push(-7)	1	55	-7	?	?	?	1

push(16)	2	55	-7	16	?	?	1
pop	1	55	-7	16	?	?	16
push(-8)	2	55	-7	-8	?	?	1
pop	1	55	-7	-8	?	?	-8
pop	0	55	-7	-8	?	?	-7

Chú ý rằng trong minh họa này chúng ta thấy rằng một số giá trị vẫn còn trong mảng nhưng chúng được xem như không có trong stack vì không có thao tác nào truy cập tới chúng. Nói chúng thì phần tử E[i] được xem là rác nếu như i>top_idx. Tại sao chúng ta không xóa bỏ các phần tử này (chẳng hạn như đặt chúng bằng các giá trị mặc định nào đó?).

Cài đặt của stack bằng ngôn ngữ C như sau (áp dụng stack cho bài toán chuyển số từ cơ số 10 sang cơ số 2):

```
#include <stdio.h>
#include <stdlib.h>
const int MAX_ELEMENT = 100; // so phan tu toi da cua stack la 100
// khai bao stack chua cac so nguyen
typedef struct
{
	int * data; // khai bao mang dong
	int top_idx;
} stack;
// ham khoi tao stack rong
void init(stack *s);
void push(stack * s, int d);
void pop(stack *s);
int top(const stack *s);
int size(const stack *s);
int empty(const stack *s);
int full(const stack *s);
// ham giai phong bo nho danh cho stack
void clear(stack *s);

int main()
{
	int n;
```

```
        int bit;
        stack s;
        init(&s);
        printf("Nhap so nguyen n = ");
        scanf("%d", &n);
        while(n)
        {
                push(&s, n%2);
                n /= 2;
        }
        while(!empty(&s))
        {
                bit = top(&s);
                pop(&s);
                printf("%d", bit);
        }
        clear(&s);
        return 0;
}
void init(stack *s)
{
        s->data = (int*)malloc(MAX_ELEMENT * sizeof(int));
        s->top_idx = -1;
}
void clear(stack *s)
{
        if(s->data != NULL)
                free(s->data);
        s->top_idx = -1;
}
void push(stack *s, int d)
{
        s->data[++s->top_idx] = d;
}
```

```
void pop(stack *s)
{
        s->top_idx --;
}
int top(const stack *s)
{
        return s->data[s->top_idx];
}
int size(const stack *s)
{
        return s->top_idx+1;
}
int empty(const stack * s)
{
        return (s->top_idx==-1)?(1):(0);
}
int full(const stack * s)
{
        return (s->top_idx==MAX_ELEMENT-1)?(1):(0);
}
```

Cấu trúc stack có thể cài đặt bằng mảng theo các cách khác, hoặc cài đặt bằng con trỏ (chúng ta sẽ học về phần cài đặt này sau phần danh sách liên kết).

Stack có thể được cài đặt bằng cả mảng và danh sách liên kết vậy khi nào chúng ta sử dụng mảng và khi nào dùng danh sách liên kết?

Danh sách liên kết	Mảng
Cứ mỗi phần tử của stack cần có thêm 2 byte (1 con trỏ). Nếu kích thước của một phần tử lớn thì không đáng kể nhưng nếu là kiểu int thì kích thước sẽ tăng gấp đôi.	Xin cấp phát một vùng nhớ có kích thước cố định và có thể một phần trong số đó không bao giờ được dùng đến và nếu như kích thước của một phần tử lớn thì vùng nhớ lãng phí này cũng rất lớn.
Không có giới hạn về số phần tử của stack.	Kích thước tối đa của stack được xác định ngay khi nó được tạo ra.

1.5. Ứng dụng của stack

Ví dụ 1

Stack có thể dùng để kiểm tra các cặp ký hiệu cân bằng trong một chương trình (chẳng hạn {}, (), []).

Ví dụ {()} và {()({})} là các biểu thức đúng còn {(({} và {(}) không phải là các biểu thức đúng.

Chúng ta thấy rằng nếu như một biểu thức là đúng thì trong quá trình đọc các ký hiệu của biểu thức đó nếu chúng ta thấy một ký hiệu đóng (chẳng hạn), } hay]) thì ký hiệu này phải khớp với ký hiệu mở được đọc thấy gần nhất (theo thời gian), và khi đó việc sử dụng stack cho các bài toán như thế này là hoàn toàn hợp lý.

Chúng ta có thể sử dụng thuật toán sau đây:

```
while not end of Input
        S = next sumbol
        if(s is opening symbol)
                push(s)
        else    // s là dấu đóng ngoặc
                if(stack.empty)
                        Báo lỗi
                else
                        R = stack.top()
                        stack.pop()
                        if(!match(s,r))
                                Báo lỗi
If(!stack.empty())
        Báo lỗi
```

Ví dụ:

1. Input: {()}

s = {, push{,

s = (, push (,

s =), r = pop = (, r,s match

s = }, r = pop = {, r,s match

End of Input, stack rỗng => biểu thức là đúng.

Ví dụ: Input = { ( ) ({) } } (sai)

Input = { ({ }) { } ( ) }

Ví dụ 2

Sử dụng Stack để chuyển đổi các dạng biểu thức đại số. Trong ví dụ này chúng ta sẽ xem xét các thuật toán sử dụng stack để chuyển đổi từ biểu đại số ở dạng trung tố (dạng thông thường, hay còn gọi là infix notation) thành các biểu thức ở dạng tiền tố (prefix notation, hay còn gọi là biểu thức Balan – Polish notation) và biểu thức hậu tố (postfix notation, hay biểu thức Balan ngược).

Biểu thức đại số là một sự kết hợp đúng đắn giữa các toán hạng (operand) và các toán tử (operator). Toán hạng là các số liệu có thể thực hiện được các thao tác tính toán toán học. Toán hạng cũng có thể là các biến số x, y, z hay các hằng số. Toán tử là một ký hiệu chỉ ra thao tác tính toán toán học hay logic giữa các toán hạng, chẳng hạn như các toán hạng +, -, *, /, ^ (toán tử mũ hóa). Việc định nghĩa các biểu thức đại số một cách chặt chẽ về lý thuyết là như sau:

- Một toán hạng là một biểu thức hợp lệ
- Nếu expression1 và expression2 là hai biểu thức hợp lệ và op là một toán tử thì một kết hợp hợp lệ giữa biểu thức expression1 với biểu thức expression2 sử dụng toán tử op sẽ cho ta một biểu thức đại số hợp lệ

Theo định nghĩa trên ta có x + y*z là một biểu thức đại số hợp lệ nhưng *x y z+ không phải là một biểu thức hợp lệ. Trong các biểu thức đại số người ta có thể sử dụng các dấu đóng và mở ngoặc.

Một biểu thức đại số có thể được biểu diễn bằng 3 dạng khác nhau.

Biểu thức trung tố (infix notation): đây là dạng biểu diễn phổ biến nhất của các biểu thức đại số, trong các biểu thức trung tố, toán tử nằm giữa các toán hạng. Ví dụ như 2 + 3 * (7 – 3)

Biểu thức tiền tố (prefix notation): dạng biểu diễn này do nhà toán học người Balan Jan Lukasiewicz đưa ra vào những năm 1920. Trong dạng biểu diễn này, toán tử đứng trước các toán hạng. Ví dụ như + * 2 3 7

Biểu thức hậu tố (postfix notation): hay còn gọi là biểu thức Balan ngược, toán tử đứng sau các toán hạng. Ví dụ như 2 3 – 7 *.

Câu hỏi mà chúng ta có thể đặt ra ngay lập tức ở đây là: tại sao lại cần sử dụng tới các dạng biểu diễn tiền tố và hậu tố trong khi chúng ta vẫn quen và vẫn sử dụng được các biểu thức ở dạng trung tố.

Lý do là các biểu thức trung tố không đơn giản và dễ dàng khi tính giá trị của chúng như chúng ta vẫn tưởng. Để tính giá trị của một biểu thức trung tố chúng ta cần tính tới độ ưu tiên của các toán tử của biểu thức và các qui tắc kết hợp. Độ ưu tiên của các toán tử và các qui tắc kết hợp sẽ quyết định tới quá trình tính toán giá trị của một biểu thức trung tố.

Chúng ta có bảng độ ưu tiên của các toán tử thường gặp như sau:

Toán tử	Độ ưu tiên
()	
+ (một ngôi), - (một ngôi), !	

+ (cộng), - (trừ)	
<, <=, >, >=	
==, !=	
&&	
||	

Khi đã biết độ ưu tiên toán tử chúng ta có thể tính toán các biểu thức chẳng hạn 2 + 4 * 5 sẽ bằng 22 vì trước hết cần lấy 4 nhân với 5, sau đó kết quả nhận được đem cộng với 2 vì phép nhân có độ ưu tiên cao hơn phép cộng. Nhưng với biểu thức 2*7/3 thì ta không thể tính được vì phép nhân và phép chia có độ ựu tiên bằng nhau, khi đó cần sử dụng tới các qui tắc kết hợp các toán tử. Qui tắc kết hợp sẽ cho chúng ta biết thứ tự thực hiện các toán tửu có cùng độ ưu tiên. Chẳng hạn chúng ta có qui tắc kết hợp trái, nghĩa là các toán tử cùng độ ưu tiên sẽ được thực hiện từ trái qua phải, hay qui tắc kết hợp phải. Nếu theo qui tắc kết hợp trái thì phép toán trên sẽ có kết quả là 4 (lấy kết quả nguyên).

Vì những vấn đề liên quan tới độ ưu tiên toán tử và các qui luật kết hợp nên chúng ta thường sử dụng các dạng biểu diễn tiền tố và hậu tố trong việc tính toán các biểu thức đại số.

Cả biểu thức hậu tố và tiền tố đều có một ưu điểm hơn so với cách biểu diễn trung tố: đó là khi tính toán các biểu thức ở dạng tiền tố và hậu tố chúng ta không cần phải để ý tới độ ưu tiên toán tử và các luật kết hợp. Tuy nhiên so với biểu thức trung tố, các biểu thức tiền tố và hậu tố khó hiểu hơn và vì thế nên khi biểu diễn chúng ta vẫn sử dụng dạng biểu thức trung tố, nhưng khi tính toán sẽ sử dụng dạng tiền tố hoặc hậu tố, điều này yêu cầu cần có các thuật toán chuyển đổi từ dạng trung tố sang dạng tiền tố hoặc hậu tố.

Việc chuyển đổi của chúng ta có thể thực hiện bằng cách sử dụng cấu trúc stack hoặ cây biểu thức (chương 5), phần này chúng ta sẽ chỉ xem xét các thuật toán sử dụng stack vì thuật toán sử dụng cây biểu thức khá phức tạp.

Thuật toán chuyển đổi biểu thức dạng trung tố thành dạng hậu tố sử dụng stack.

Ví dụ 3

Phân tích độ ưu tiên toán tử

Chúng ta có thể sử dụng cấu trúc stack để phân tích và lượng giá các biểu thức toán học kiểu như:

5 * (((9+8) * (4 * 6)) + 7)

Trước hết chúng ta chuyển chúng thành dạng hậu tố (postfix):

5 8 9 + 4 6 * * 7 + *

Sau đó sử dụng một stack để thực hiện việc tính toán giá trị của biểu thức hậu tố nhận được.

Ngoài ra các stack còn có thể dùng để cài đặt các thuật toán đệ qui và khử đệ qui các cài đặt thuật toán.

2. Hàng đợi - Queue

2.1. Khái niệm

Hàng đợi là một tập hợp các phần tử cùng kiểu được tổ chức một cách tuần tự (tuyến tính) trong đó phần tử được thêm vào đầu tiên sẽ là phần tử được loại bỏ đầu tiên khỏi hàng đợi. Các hàng đợi thường được gọi là các cấu trúc FIFO (First In First Out).

Các ví dụ thực tế về hàng đợi mà chúng ta có thể thấy trong cuộc sống hàng ngày đó là đoàn người xếp hàng chờ mua vé tầu, danh sách các cuộc hẹn của một giám đốc, danh sách các công việc cần làm của một người …

Cũng có thể định nghĩa hàng đợi là một danh sách tuyến tính các phần tử giống nhau với một số thao tác hạn chế tới các phần tử trên danh sách đó.

2.2. Các thao tác cơ bản của một hàng đợi

Tương tự như cấu trúc ngăn xếp, chúng ta định nghĩa các thao tác trên hàng đợi tuân theo cài đặt chuẩn của hàng đợi trong thư viện STL và các tài liệu khác, gồm có:

1. push(d): thêm phần tử d vào vị trí ở cuối hàng đợi.
2. pop(): loại bỏ phần tử ở đầu hàng đợi.
3. front(): trả về giá trị phần tử ở đầu hàng đợi.
4. back(): trả về giá trị phần tử ở cuối hàng đợi.
5. size(): trả về số phần tử đang ở trong hàng đợi.
6. empty(): kiểm tra hàng đợi có rỗng hay không.
7. full(): kiểm tra hàng đợi đầy (chỉ cần khi cài đặt hàng đợi bằng mảng).

Ví dụ:

Thao tác	Nội dung	Giá trị trả về
Khởi tạo	( )	
push(7)	(7)	
push(8)	(7, 8)	
push(5)	(7, 8, 5)	
pop()	(8, 5)	7
pop()	(5)	8

2.3. Cài đặt hàng đợi sử dụng mảng

Để cài đặt cấu trúc hàng đợi chúng ta có thể sử dụng mảng hoặc sử dụng con trỏ (phần này sẽ học sau phần danh sách liên kết):

- Ta lưu các phần tử của hàng đợi trong một mảng data. Đầu của hàng đợi là phần tử đầu tiên, và đuôi được chỉ ra bằng cách sử dụng một biến tail.
- push(d) được thực hiện một cách dễ dàng: tăng tail lên 1 và chèn phần tử vào vị trí đó

- pop() được thực hiện không hiệu quả: tất cả các phần tử đều sẽ bị dồn về đầu mảng do đó độ phức tạp là ***O***(n).

Làm thế nào chúng ta có thể cải thiện tình hình này?

Thay vì chỉ sử dụng một biến chỉ số tail chúng ta sử dụng hai biến tail và head, khi cần loại bỏ (pop) một phần tử khỏi hàng đợi chúng ta sẽ tăng biến head lên 1:

Tuy vậy vẫn còn có vấn đề, đó là sau n lần push() (n là kích thước mảng) mảng sẽ đầy kể cả trong trường hợp nó gần như rỗng về mặt logic. Để giải quyết vấn đề này chúng ta sẽ sử dụng lại các phần tử ở đầu mảng. Khi push() một phần tử mới tail sẽ được tăng lên 1 nhưng nếu như nó ở cuối mảng thì sẽ đặt nó bằng 0.

Vấn đề mới nảy sinh ở đây là làm thế nào chúng ta có thể xác định được khi nào hàng đợi rỗng hoặc đầy?

Cách giải quyết đơn giản là ta sẽ dùng một biến lưu số phần tử thực sự của hàng đợi để giải quyết cho tất cả các thao tác kiểm tra hàng đợi rỗng, đầy hoặc lấy số phần tử của hàng đợi.

```
#include <stdio.h>
#include <stdlib.h>
const int MAX_ELEMENT = 100; // so phan tu toi da cua queue la 100
// khai bao queue chua cac so nguyen
typedef struct
{
        int * data; // khai bao mang dong
        int head;
        int tail;
        int cap; // luu so phan tu cua hang doi
} queue;
// ham khoi tao queue rong
void init(queue *q);
void push(queue * s, int d);
void pop(queue *q);
int front(const queue *q);
int back(const queue *q);
int size(const queue *q);
int empty(const queue *q);
int full(const queue *q);
// ham giai phong bo nho danh cho queue
void clear(queue *q);
```

```
int main()
{
	int a[] = {3, 5, 1, 8};
	int n = 4;
	int i;
	int d;
	queue q;
	init(&q);
	for(i=0;i<n;i++)
		push(&q, a[i]);
	while(!empty(&q))
	{
		d = front(&q);
		printf("%d ", d);
		pop(&q);
	}
	clear(&q);
	return 0;
}
void init(queue *q)
{
	q->data = (int*)malloc(MAX_ELEMENT * sizeof(int));
	q->head = q->tail = -1;
	q->cap = 0;
}
void clear(queue *q)
{
	if(q->data != NULL)
		free(q->data);
	q->head = q->tail = -1;
	q->cap = 0;
}
void push(queue *q, int d)
{
```

```
        q->tail = (q->tail + 1) % MAX_ELEMENT;
        q->data[q->tail] = d;
        if(q->cap==0)
                // neu hang doi rong thi sau khi push
                // ca head va tail deu chi vao 1 phan tu
                q->head = q->tail;
        q->cap++;
}
void pop(queue *q)
{
        q->head = (q->head + 1)%MAX_ELEMENT;
        q->cap--;
        if(q->cap==0)
                q->head = q->tail = -1;
}
int front(const queue *q)
{
        return q->data[q->head];
}
int back(const queue *q)
{
        return q->data[q->tail];
}
int size(const queue *q)
{
        return q->cap;
}
int empty(const queue *q)
{
        return (q->cap==0)?(1):(0);
}
int full(const queue *q)
{
        return (q->cap==MAX_ELEMENT-1)?(1):(0);
```

}

2.4. Ví dụ về hoạt động của hàng đợi với cài đặt bằng mảng vòng tròn

Ta giả sử mảng lưu các phần tử của hàng đợi là E[0..3], các biến head, tail lưu vị trí của phần tử ở đầu và cuối hàng đợi, cột R là cột kết quả thực hiện các thao tác trên hàng đợi, các dấu ? tương ứng với giá trị bất kỳ.

Thao tác	head	tail	E[0]	E[1]	E[2]	E[3]	R
Khởi tạo	-1	-1	?	?	?	?	
push(55)	0	0	55	?	?	?	
push(-7)	0	1	55	-7	?	?	
push(16)	0	2	55	-7	16	?	
pop()	1	2	55	-7	16	?	55
push(-8)	1	3	55	-7	16	-8	
pop()	2	3	55	-7	16	-8	-7
pop()	3	3	55	-7	16	-8	16
push(11)	3	4	11	-7	16	-8	

2.5. Ứng dụng của hàng đợi

Trong các hệ điều hành:

- Hàng đợi các công việc hoặc các tiến trình đang đợi để được thực hiện
- Hàng đợi các tiến trình chờ các tín hiệu từ các thiết bị IO
- Các file được gửi tới máy in

Mô phỏng các hệ thống hàng đợi thời trong thực tế.

- Các khách hàng trong các cửa hàng tạp hóa, trong các hệ thống ngân hàng
- Các đơn đặt hàng của một công ty
- Phòng cấp cứu
- Các cuộc gọi điện thoại hoặc các đặt hàng vé máy bay, các đặt hàng của khách hàng …

Các ứng dụng khác:

Thứ tự topo: với một tập các sự kiện, và các cặp (a, b) trong đó sự kiện a có độ ưu tiên cao hơn so với sự kiện b (bài toán lập lịch), duyệt đồ thị theo chiều rộng (Breadth First Search).

Bài tập: Hãy viết chương trình chuyển đổi một biểu thức dạng infix (dạng thông thường) đơn giản (không chứa các dấu ()) thành một biểu thức dạng tiền tố (prefix). Ví dụ này xem như một bài tập để sinh viên tự làm.

3. Hàng đợi hai đầu – Double Ended Queue (dequeu)

4. Hàng đợi ưu tiên – Priority Queue (pqueue)

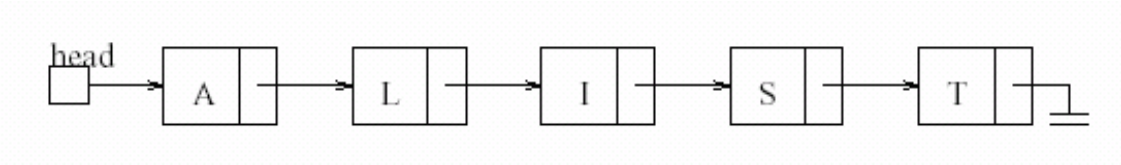
Hàng đợi ưu tiên là một hàng đợi trong đó việc thêm và loại bỏ các phần tử không tuân theo thứ tự thông thường mà dựa trên một thuộc tính của các phần tử gọi là độ ưu tiên (priority).

5. Danh sách liên kết – Linked list

5.1. Định nghĩa

Danh sách liên kết (linked list) là một tập hợp tuyến tính các phần tử cùng kiểu gọi là các nút (node), mỗi nút có các đặc điểm sau đây:

- Mỗi nút có ít nhất hai trường (field) một trường gọi là trường dữ liệu (data) và trường còn lại là trường liên kết (link) trỏ tới (point to) (thường gọi là next).
- Trường liên kết của phần tử thứ i của danh sách sẽ trỏ tới phần tử thứ (i+1) của danh sách
- Phần tử đầu tiên của danh sách liên kết được gọi là head và phần tử cuối cùng được gọi là tail. Head không chứa dữ liệu và trường next của tail sẽ chỉ vào NULL.
- Trường data là trường chứa dữ liệu mà chúng ta thực sự lưu trong danh sách liên kết
- Giá trị NULL và việc thực hiện trỏ tới (point to) của mỗi liên kết thực sự diễn ra như thế nào phụ thuộc nhiều vào việc cài đặt cụ thể danh sách liên kết.



Có nhiều loại danh sách liên kết khác nhau tùy thuộc vào cấu trúc của mỗi phần tử trong danh sách (số trường liên kết với các phần tử khác trong danh sách) nhưng cơ bản nhất là danh sách liên kết đơn (single linked list), mỗi phần tử có một trường liên kết như trên hình vẽ minh họa, và khi chúng ta nói đến danh sách liên kết, nếu không có các chú giải đi kèm thì ngầm hiểu đó là danh sách liên kết đơn.

5.2. Các thao tác trên danh sách liên kết.

Tương tự như các cấu trúc cơ bản stack và queue, chúng ta định nghĩa các thao tác của danh sách liên kết dựa trên cài đặt chuẩn của cấu trúc danh sách liên kết trong thư viện STL:

1. push_front(d): thêm một phần tử vào đầu danh sách.
2. push_back(d): thêm một phần tử vào cuối danh sách.
3. pop_front(): loại bỏ phần tử ở đầu danh sách.
4. pop_back(): loại bỏ phần tử cuối danh sách.
5. erase(): xỏa bỏ một phần tử khỏi danh sách.
6. insert(): chèn một phần tử mới vào một vị trí cụ thể của danh sách.
7. size(): cho biết số phần tử trong danh sách.

8. empty(): kiểm tra danh sách rỗng.
9. begin(): trả về phần tử ở đầu danh sách.
10. end(): trả về phần tử ở cuối danh sách.
11. sort(): sắp xếp danh sách theo trường khóa (là một trường con của trường dữ liệu).
12. merge(): trộn danh sách với một danh sách khác.
13. clear(): xóa bỏ toàn bộ các phần tử của danh sách.
14. find(): tìm kiếm một phần tử trong danh sách theo khóa tìm kiếm.

Các thao tác khác cũng có thể được cài đặt với một danh sách liên kết để làm cho công việc của các lập trình viên trở nên dễ dàng hơn:

- Di chuyển một phần tử trong danh sách
- Đổi hai phần tử cho nhau

5.3. Cài đặt danh sách liên kết sử dụng con trỏ

```
typedef struct Node
{
        // truong du lieu
        int data;
        struct Node * next;
} NodeType;
```

Khởi tạo danh sách:

```
NodeType * head, * tail;
head = new node;
head→next = NULL;
```

Các thao tác trên sẽ tạo ra một danh sách liên kết rỗng (empty – không chứa phần tử nào)

Trong cài đặt này chúng ta cho tail chỉ vào NULL, thường được định nghĩa là 0. Do đó trường next của một phần tử sẽ là 0, và khi đó chúng ta biết là chúng ta đang ở phần tử tail của danh sách.

Ở đây trỏ tới (point to) có nghĩa là chúng ta thực sự sử dụng các con trỏ. Chúng ta sẽ sớm thấy rằng chúng ta không cần thiết phải sử dụng các con trỏ thực sự để cài đặt một danh sách liên kết.

Chèn một nút (node) vào danh sách liên kết

Dễ dàng nhận thấy rằng cách đơn giản nhất để chèn một nút mới vào một danh sách liên kết là đặt nút đó ở đầu (hoặc cuối) của danh sách. Hoặc cũng có thể chúng ta muốn chèn các phần tử vào giữa danh sách.

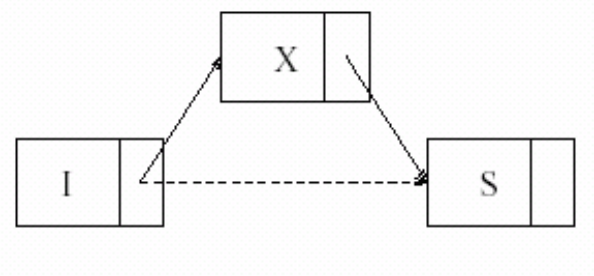
Chèn X vào giữa I và S:

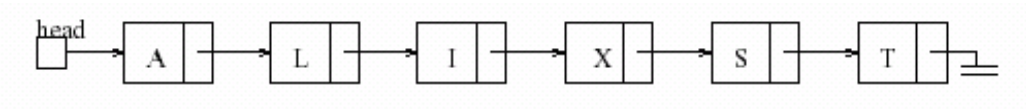
```
struct node * A;
A = new node;
```

A→key = X;

A→next = I→next;

I→next = A;



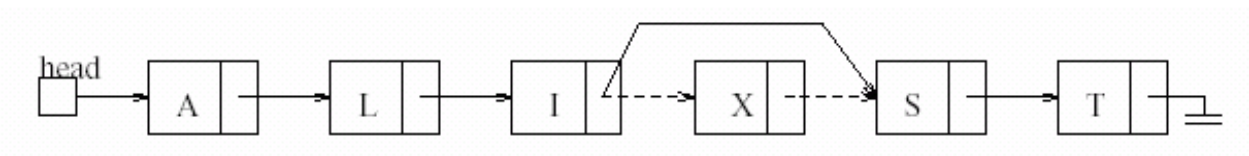


Để thực hiện điều này chúng ta cần 2 tham chiếu tới hai nút trong danh sách và không cần quan tâm tới độ dài (số phần tử) của danh sách. Tuy nhiên thực hiện việc này với các mảng chắc chắn sẽ khác nhiều.

Xóa một nút (node) khỏi danh sách liên kết

Xóa một nút khỏi danh sách liên kết rất đơn giản chúng ta chỉ cần thay đổi một con trỏ, tuy nhiên vấn đề là chúng ta cần biết nút nào trỏ tới nút mà chúng ta định xóa. Giả sử chúng ta biết nút i trỏ tới nút x và chúng ta muốn xóa bỏ x:

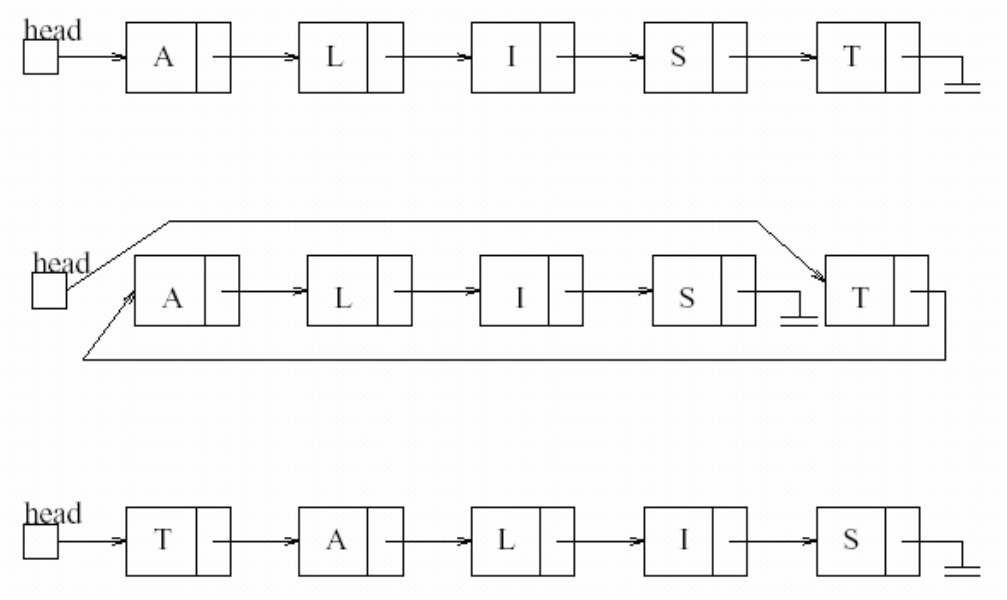
i→next = x→next;



Chỉ có một tham chiếu bị thay đổi không phụ thuộc vào độ dài của danh sách (so sánh với cài đặt bằng mảng, tuy vậy vẫn có vấn đề với cài đặt trên).

Di chuyển (move) một nút trong danh sách liên kết

Di chuyển một nút trong danh sách liên kết bao gồm hai thao tác: xóa bỏ một nút sau đó chèn vào một nút. Ví dụ chúng ta muốn di chuyển nút T từ cuối danh sách lên đầu danh sách:



Một lần nữa chúng ta thấy rằng thao tác di chuyển này chỉ đòi hỏi thay đổi 3 tham chiếu và không phụ thuộc vào độ dài của danh sách (so sánh điều này với cài đặt bằng mảng).

Cài đặt minh họa đầy đủ của danh sách liên kết đơn:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
// khai bao cau truc cho mot nut cua danh sach
typedef struct Node
{
        // truong du lieu
        int data;
        struct Node * next;
} NodeType;
// khai bao kieu danh sach
typedef struct
{
        NodeType * head;
        NodeType * tail;
        // so phan tu cua danh sach
        int spt;
}LList;
// ham khoi tao danh sach
void init(LList * list);
```

```
// ham them 1 phan tu vao dau danh sach
void push_front(LList *list, int d);
// ham them mot phan tu vao cuoi danh sach
void push_back(LList *list, int d);
// ham xoa phan tu o cuoi danh sach
int pop_back(LList * list);
// ham xoa phan tu o dau danh sach
int pop_front(LList * list);
// ham tra ve phan tu dau tien
int begin(const LList * list);
// ham tra ve phan tu cuoi cung
int end(const LList * list);
void insertAfter(LList * list, NodeType * p);
void insertBefore(LList * list, NodeType * p);
void eraseAfter(LList * list, NodeType * p);
void eraseBefore(LList * list, NodeType * p);
// ham in danh sach
void printList(LList list);
// ham sap xep danh sach
void sort(LList *list);
// ham tim kiem trong danh sach
NodeType * find(LList *, int d);
// giai phong toan bo danh sach
void clear(LList * list);
// ham tron hai danh sach, ket qua luu trong danh sach thu nhat
void merge(LList *list1, const LList *list2);
// ham kiem tra danh sach lien ket co rong khong
int empty(const LList *list);
int main()
{
        LList myList;
        LList list2;
        init(&myList);
        init(&list2);
```

```
        push_front(&myList, 10);
        push_front(&myList, 1);
        push_front(&myList, 12);
        push_back(&myList, 20);
        push_back(&myList, 23);
        push_back(&myList, 25);
        sort(&myList);
        printList(myList);
        push_front(&list2, 14);
        push_front(&list2, 9);
        merge(&myList, &list2);
        printList(myList);
        printList(list2);
        clear(&myList);
        printList(myList);
        return 0;
}
void init(LList * list)
{
        list->head = list->tail = NULL;
        list->spt = 0;
}
void printList(LList list)
{
        NodeType * tmp;
        tmp = list.head;
        while(tmp!=NULL)
        {
                printf("%d ", tmp->data);
                tmp = tmp->next;
        }
        printf("\n");
}
void push_front(LList * list, int d)
```

```
{
        NodeType * tmp = (NodeType *)malloc(sizeof(NodeType));
        tmp->data = d;
        if(list->spt==0)
        {
                tmp->next = NULL;
                list->head = list->tail = tmp;
        }
        else
        {
                tmp->next = list->head;
                list->head = tmp;
        }
        list->spt = list->spt+1;
}
void push_back(LList * list, int d)
{
        NodeType * tmp = (NodeType *)malloc(sizeof(NodeType));
        tmp->data = d;
        tmp->next = NULL;
        if(list->spt==0)
                list->head = list->tail = tmp;
        else
        {
                list->tail->next = tmp;
                list->tail = tmp;
        }
        list->spt = list->spt+1;
}
NodeType * find(LList * list, int d)
{
        NodeType * tmp = list->head;
        while(tmp!=NULL)
        {
```

```
                if (tmp->data==d)
                        break;
                tmp = tmp->next;
        }
        return tmp;
}
int pop_back(LList * list)
{
        NodeType * p, * q;
        int ret = -1;

        if(list->spt>0)
        {
                p = list->head;
                q = NULL;
                while(p->next!=NULL)
                {
                        q = p;
                        p = p->next;
                }
                if(q!=NULL)
                {
                        // danh sach chi co 1 phan tu
                        q->next = NULL;
                        list->tail = q;
                }
                else
                        list->head = list->tail = NULL;
                ret = p->data;
                free(p);
                list->spt = list->spt-1;
        }
        return ret;
}
```

```
int pop_front(LList * list)
{
	int ret=-1;
	NodeType * tmp;
	if(list->spt>0)
	{
		tmp = list->head;
		if(list->spt==1)
		{
			// danh sach chi co 1 phan tu
			ret = list->head->data;
			list->head = list->tail = NULL;
		}else
			list->head = list->head->next;
		free(tmp);
		list->spt = list->spt - 1;
	}
	return ret;
}
// sap xep dung thuat toan doi cho truc tiep (interchange sort)
void sort(LList * list)
{
	// sử dụng thuật toán sắp xếp nổi bọt Bubble sort
	NodeType * p, * q;
	int tmp;

	p = list->head;
	while(p!=NULL)
	{
		q = p->next;
		while(q!=NULL)
		{
			if(q->data < p->data)
			{
```

```
                                tmp = q->data;
                                q->data = p->data;
                                p->data = tmp;
                        }
                        q = q->next;
                }
                p = p->next;
        }
}
void clear(LList * list)
{
        NodeType * p, * q;
        if(list->spt>0)
        {
                p = list->head;
                list->head = list->tail = NULL;
                list->spt = 0;
                while(p)
                {
                        q = p->next;
                        free(p);
                        p = q;
                }
        }
}
void merge(LList *list1, const LList *list2)
{
        NodeType * tmp;
        tmp = list2->head;
        while(tmp)
        {
                push_back(list1, tmp->data);
                tmp = tmp->next;
        }
```

```
}
int empty(const LList *list)
{
        return (list->spt==0)?(1):(0);
}
int begin(const LList * list)
{
        return list->head->data;
}
int end(const LList * list)
{
        return list->tail->data;
}
```

So sánh giữa danh sách liên kết và mảng

Danh sách liên kết có số phần tử có thể thay đổi và không cần chỉ rõ kích thước tối đa của danh trước. Ngược lại mảng là các cấu trúc có kích thước cố định

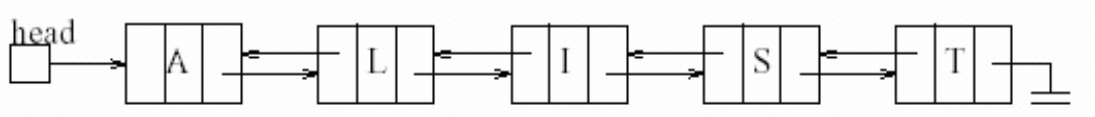
Chúng ta có thể sắp xếp lại, thêm và xóa các phần tử khỏi danh sách liên kết chỉ với một số cố định các thao tác. Với mảng các thao tác này thường tương đương với kích thước mảng.

Để tìm đến phần tử thứ i trong một danh sách liên kết chúng ta cần phải dò qua i-1 phần tử đứng trước nó trong danh sách (i-1 thao tác) trong khi với mảng để làm điều này chỉ mất 1 thao tác.

Tương tự kích thước của một danh sách không phải hiển nhiên mà biết được trong khi chúng ta luôn biết rõ kích thước của một mảng trong chương trình (tuy nhiên có một cách đơn giản để khắc phục điều này).

5.4. Các kiểu danh sách liên kết khác

Danh sách liên kết đôi (double linked list) giống như một danh sách liên kết đơn ngoại trừ việc mỗi nút có thêm một trường previous trỏ vào nút đứng trước nó.



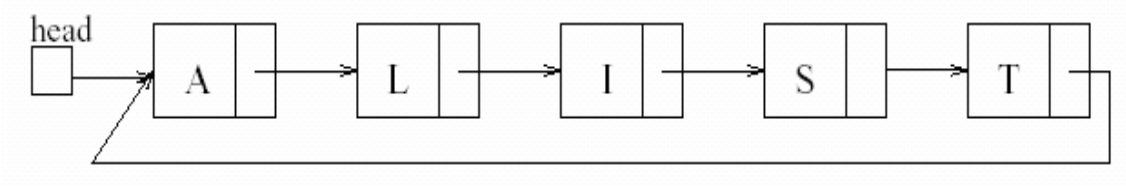
Với danh sách liên kết đôi các thao tác như tìm kiếm, xóa bỏ một nút khỏi danh sách thực hiện dễ dàng hơn nhưng đồng thời cũng mất nhiều bộ nhớ hơn và số lượng các lệnh để thực hiện một thao tác trên danh sách liên kết đôi chắc chắn cũng xấp xỉ gấp đôi so với danh sách liên kết đôi.

Ngoài ra còn có một loại danh sách liên kết khác được gọi là danh sách liên kết vòng (circular-linked list).

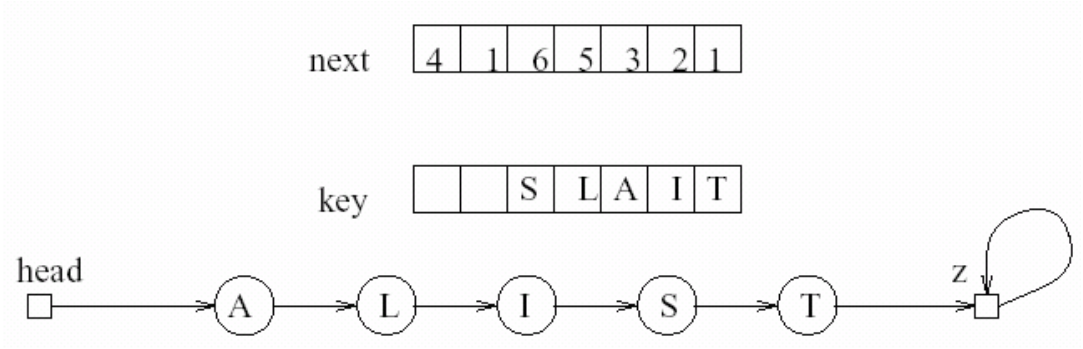
Nút cuối cùng trỏ tới nút đầu tiên

Danh sách liên kết vòng có thể là danh sách liên kết đơn hoặc danh sách liên kết đôi

Nó có thể được cài đặt với một đầu (head) cố định hoặc thay đổi



Việc cài đặt danh sách liên kết có thể không cần thiết sử dụng tới các con trỏ. Thay vào đó chúng ta có thể sử dụng các mảng để cài đặt các danh sách liên kết, ở đây chúng ta không đi sâu vào xem xét cụ thể cài đặt một danh sách liên kết bằng mảng như thế nào nhưng cũng không quá khó để hình dung cách thức hoạt động của các danh sách kiểu như thế.



Kết luận:

Danh sách liên kết là các cấu trúc dữ liệu rất giống với các mảng

Các thao tác chính thường được sử dụng đối với một danh sách liên kết là thêm, xóa và tìm kiếm trên danh sách

Thao tác chèn và xóa có thể thực hiện với thời gian hằng số

Việc tìm một phần tử trong danh sách liên kết thường mất thời gian tuyến tính (xấp xỉ độ dài danh sách) và trường hợp xấu nhất là đúng bằng độ dài của danh sách. Đây cũng chính là một trong những nhược điểm lớn nhất của danh sách liên kết.

5.5. Một số ví dụ sử dụng cấu trúc danh sách liên kết

Các bài toán mà danh sách liên kết thường được sử dụng là các bài toán trong đó việc sử dụng mảng sẽ là không thuận lợi, chẳng hạn một bài toán yêu cầu các thao tác thêm, xóa bỏ xảy ra thường xuyên thì lựa chọn thông minh sẽ là sử dụng danh sách liên kết. Một ví dụ nữa là khi ta làm việc với các đồ thị thưa (các cạnh ít) lớn (nhưng số đỉnh nhiều), thay vì dùng một mảng hai chiều, ta sẽ dùng một mảng các danh sách liên kết, mỗi danh sách liên kết chứa các đỉnh liền kề của một đỉnh của đồ thị.

5.6. Cài đặt stack và queue bằng con trỏ

Về bản chất, các cấu trúc dữ liệu stack và queue là các cấu trúc danh sách liên kết hạn chế, các thao tác được giới hạn so với cấu trúc danh sách liên kết. Vì thế có thể coi một stack hay queue là một danh sách liên kết, và có thể lợi dụng cài đặt bằng con trỏ của danh

sách liên kết để cài đặt các cấu trúc stack và queue (sử dụng con trỏ). Phần này được để lại xem như một bài tập của sinh viên.

5.7. Bài tập

Làm bài thực hành số 3 tương ứng của môn học.

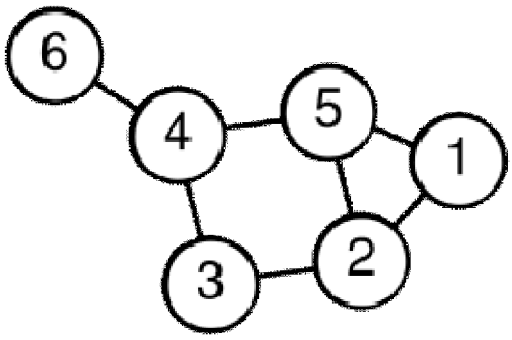
Chương 5: Cây (Tree)

1. Định nghĩa

1.1. Đồ thị (Graph)

Trước khi xem xét khái niệm thế nào là một cây (tree) chúng ta nhắc lại khái niệm đồ thị (graph) đã được học trong học phần Toán rời rạc: Đồ thị G bao gồm hai thành phần chính: tập các đỉnh V (Vertices) và tập các cung E(hay cạnh Edges), thường viết ở dạng G = <V, E>. Trong đó tập các đỉnh V là tập các đối tượng cùng loại, độc lập, chẳng hạn như các điểm trên mặt phẳng tọa độ, hoặc tập các thành phố, tập các trạng thái của một trò chơi, một đối tượng thực như con người, … tất cả đều có thể là các đỉnh của một đồ thị nào đó. Tập các cung E là tập các mối quan hệ hai ngôi giữa các đỉnh của đồ thị, đối với đỉnh là các điểm thì đây có thể là quan hệ về khoảng cách, tập đỉnh là các thành phố thì đây có thể là quan hệ về đường đi (có tồn tại đường đi trực tiếp nào giữa các thành phố hay không), hoặc nếu đỉnh là các trạng thái của một trò chơi thì cạnh có thể là cách biến đổi (transform) để đi từ trạng thái này sang một trạng thái khác, quá trình chơi chính là biến đổi từ trạng thái ban đầu tới trạng thái đích (có nghĩa là đi tìm một đường đi).

Ví dụ về đồ thị:



Hình 5.1. Đồ thị có 6 đỉnh và 7 cạnh, tham khảo từ wikipedia.

Có rất nhiều vấn đề liên quan tới đồ thị, ở phần này chúng ta chỉ nhắc lại một số khái niệm liên quan.

Một đồ thị được gọi là đơn đồ thị (simple graph) nếu như không có đường đi giữa hai đỉnh bất kỳ của đồ thị bị lặp lại, ngược lại nếu như có đường đi nào đó bị lặp lại hoặc tồn tại khuyên (self-loop), một dạng cung đi từ 1 đỉnh đến chính đỉnh đó, thì đồ thị được gọi là đa đồ thị (multigraph).

Giữa hai đỉnh u, v trong đồ thị có đường đi trực tiếp thì u, v được gọi là liền kề với nhau, cạnh (u, v) được gọi là liên thuộc với hai đỉnh u, v.

Đồ thì được gọi là đồ thị có hướng (directed graph) nếu như các đường đi giữa hai đỉnh bất kỳ trong đồ thị phân biệt hướng với nhau, khi đó các quan hệ giữa các đỉnh được gọi chính xác là các cung, ngược lại đồ nếu không phân biệt hướng giữa các đỉnh trong các cạnh nối giữa hai đỉnh thì đồ thị được gọi là đồ thị vô hướng (undirected graph), khi đó ta nói tập E là tập các cạnh của đồ thị.

Các cung hay các cạnh của đồ thị có thể được gán các giá trị gọi là các trọng số (weight), một đồ thị có thể là đồ thị có trọng số hoặc không có trọng số. Ví dụ như đối với đồ thị mà các đỉnh là các thành phố ta có thể gán trọng số của các cung là độ dài đường đi nối giữa các thành phố hoặc chi phí đi trên con đường đó …

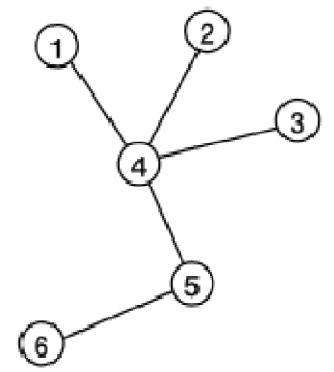
Một đường đi (path) trong đồ thị là một dãy các đỉnh v_1, v_2, …, v_k, trong đó các đỉnh v_i, v_{i+1} là liền kề với nhau. Đường đi có đỉnh đầu trùng với đỉnh cuối được gọi là chu trình (cycle).

Giữa hai đỉnh của đồ thị có thể có các đường đi trực tiếp nếu chúng liền kề với nhau, hoặc nếu có một đường đi giữa chúng (gián tiếp) thì hai đỉnh đó được gọi là liên thông (connected) với nhau. Một đồ thị được gọi là liên thông nếu như hai đỉnh bất kỳ của nó đều liên thông với nhau. Nếu đồ thị không liên thông thì luôn có thể chia nó thành các thành phần liên thông nhỏ hơn.

1.2. Cây (tree)

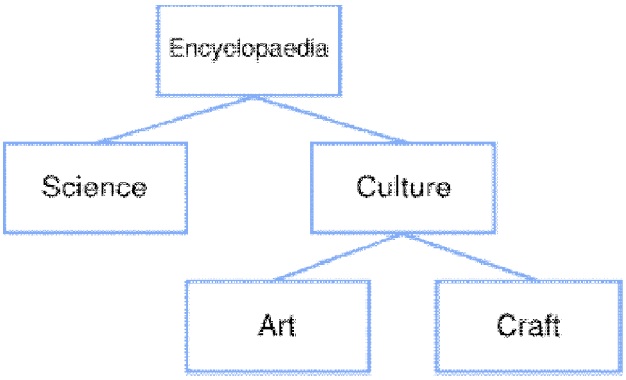
Có nhiều cách định nghĩa cây khác nhau nhưng ở đây chúng ta sẽ định nghĩa khái niệm cây theo lý thuyết đồ thị (graph theory).

Cây là một đồ thị vô hướng, không có trọng số, liên thông và không có chu trình. Ví dụ hình vẽ sau là một cây:



Hình 5.2. Cây, tham khảo từ wikipedia

Cấu trúc cây là một cấu trúc được sử dụng rất rộng rãi trong cuộc sống hàng ngày và trên máy tính, chẳng hạn cấu trúc tổ chức của một công ty là một cây phân cấp, cấu trúc của một web site cũng tương tự:



Hình 5.3. Cấu trúc web site wikipedia, tham khảo từ wikipedia.

Cấu trúc tổ chức thư mục của hệ điều hành là một cây …

Trong cây luôn có một nút đặc biệt gọi là gốc của cây (root), các đỉnh trong cây được gọi là các nút (nodes). Từ gốc của cây đi xuống tất cả các đỉnh liền kề với nó, các đỉnh này gọi là con của gốc, đến lượt các con của gốc lại có các nút con (child nodes) khác, như vậy quan hệ giữa hai nút liền kề nhau trong cây là quan hệ cha con, một nút là cha (parent), một nút là con (child), nút cha của cha của một nút được gọi là tổ tiên (ancestor) của nút đó.

Các nút trong cây được phân biệt làm nhiều loại: các nút có ít nhất 1 nút con được gọi là các nút trong (internal nodes hay inner nodes), các nút không có nút con được gọi là các nút lá (leaf nodes). Các nút lá không có các nút con nhưng để thuận tiện trong quá trình cài đặt người ta vẫn coi các nút lá có hai nút con giả, rỗng (NULL) đóng vai trò lính canh, gọi là các nút ngoài (external nodes).

Các nút trong cây được phân chia thành các tầng (level), nút gốc thuộc tầng 0 (level 0), sau đó các tầng tiếp theo sẽ được tăng lên 1 đơn vị so với tầng phía trên nó cho đến tầng cuối cùng. Độ cao (height) của cây được tính bằng số tầng của cây, độ cao của cây sẽ quyết định độ phức tạp (số thao tác) khi thực hiện các thao tác trên cây.

Mỗi nút trong của cây tổng quát có thể có nhiều nút con, tuy nhiên các nghiên cứu của ngành khoa học máy tính đã cho thấy cấu trúc cây quan trọng nhất cần nghiên cứu chính là các cây nhị phân (binary tree), là các cây là mỗi nút chỉ có nhiều nhất hai nút con. Một cây tổng quát luôn có thể phân chia thành các cây nhị phân.

Các nút con của một nút trong cây nhị phân được gọi là nút con trái (**left child**) và nút con phải (**right child**).

Trong chương này chúng ta sẽ nghiên cứu một số loại cây nhị phân cơ bản và được ứng dụng rộng rãi nhất, đó là cây tìm kiếm nhị phân BST (**Binary Search Tree**), cây biểu thức (**expression tree** hay **syntax tree**) và cây cân bằng (**balanced tree**) AVL.

2. Cây tìm kiếm nhị phân

2.1. Định nghĩa

Mỗi nút trong cây bất kỳ đều chứa các trường thông tin, trên một cây tìm kiếm nhị phân mỗi nút là một struct (bản ghi – record) gồm các trường: trường dữ liệu data, trường khóa key để so sánh với các nút khác, các liên kết tới các nút con của nút left và right.

Để tập trung vào các vấn đề thuật toán ta bỏ qua trường dữ liệu, chỉ xem như mỗi nút trên cây tìm kiếm nhị phân gồm có một trường khóa **key** và hai trường liên kết **left** và **right**.

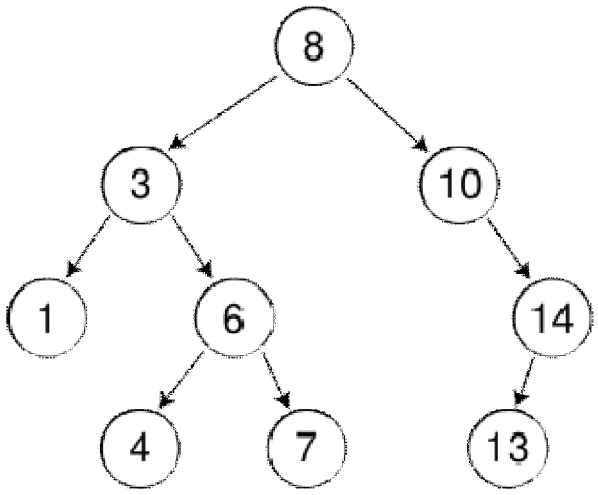
Với các giả thiết trên ta định nghĩa cây tìm kiếm nhị phân như sau:

Cây tìm kiếm nhị phân là một cây nhị phân (binary tree) mà mỗi nút x trong cây thỏa mãn bất đẳng thức kép sau:

$$key(left\_child(x)) < key(x) < key(right\_child(x))$$

Trong đó left_child(x), right_child(x) là các nút con trái và phải của nút x, key() là hàm trả về giá trị khóa ở nút tương ứng.

Ví dụ:



Hình 5.4. Cây tìm kiếm nhị phân BST, tham khảo từ wikipedia.

Ưu điểm chính của cây tìm kiếm nhị phân là: nó cung cấp thuật toán sắp xếp và tìm kiếm dựa trên kiểu duyệt thứ tự giữa (in-order) một cách rất hiệu quả, và là cấu trúc dữ liệu cơ bản cho các cấu trúc dữ liệu cao cấp hơn (trừu tượng hơn) như tập hợp (set), các mảng liên kết (associative array), các ánh xạ **map**, và các cây cân bằng tối ưu như AVL, cây đỏ đen. Chúng ta sẽ xem xét tại sao cây tìm kiếm nhị phân lại hiệu quả như vậy.

2.2. Khởi tạo cây rỗng

Thao tác đầu tiên là khai báo cấu trúc cây và khởi tạo một cây rỗng để bắt đầu thực hiện các thao tác khác.

Ở đây ta giả sử cây tìm kiếm nhị phân chỉ chứa các khóa là các số nguyên dương.

Khai báo cây tìm kiếm nhị phân trong ngôn ngữ C như sau:

```
// khai bao cau truc cay tim kiem nhi phan
typedef struct  tree
{
        int       key;
        struct  tree  *left,*right;
}BSTree;
```

Để khởi tạo một cây rỗng ta khai báo gốc của cây và gán cho gốc đó bằng NULL:

```
// cay
BSTree  **root;
*root = NULL;
```

2.3. Chèn thêm một nút mới vào cây

Để chèn một nút mới vào cây ta xuất phát từ gốc của cây, ta gọi đó là nút đang xét. Nếu như nút đang xét có khóa bằng với khóa cần chèn vào cây thì xảy ra hiện tượng trùng khóa, thuật toán kết thúc với thông báo trùng khóa. Nếu như nút đang xét là một nút ngoài (external nodes) thì ta tạo một nút mới và gán các trường thông tin tương ứng cho nút đó, gán các con của nút đó bằng NULL.

```
// them mot nut moi vao cay, gia tri khoa cua nut moi luu trong bien toan cuc newkey
void insert(BSTree **root)
{
	if(*root==NULL)
	{
		*root=calloc(1,sizeof(BSTree));
		(*root)->key = newkey;
		(*root)->left=NULL;
		(*root)->right=NULL;
	}else{
		if((*root)->key>newkey)
			insert(&((*root)->left));
		else
			if((*root)->key<newkey)
				insert(&((*root)->right));
			else
				printf("\nError: Duplicate key");
	}
}
```

Thuật toán trên sử dụng bộ nhớ Θ(log n) trong trường hợp trung bình và Ω(n) trong trường hợp tồi nhất. Độ phức tạp thuật toán bằng với độ cao của cây, tức là O(log n) trong trường hợp trung bình đối với hầu hết các cây, nhưng sẽ là Ω(n) trong trường hợp xấu nhất.

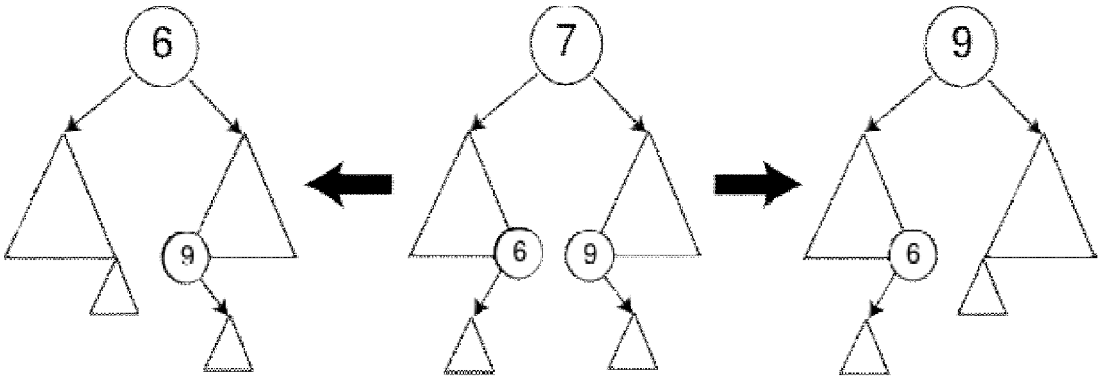
Cũng nên chú ý là các nút mới luôn được chèn vào các nút ngoài của cây tìm kiếm nhị phân, gốc của cây không thay đổi trong quá trình chèn thêm nút vào cây.

2.4. Xóa bỏ khỏi cây một nút

Khi xóa bỏ một nút X khỏi cây (dựa trên giá trị khóa), chúng ta chia ra một số trường hợp sau:

- X là một nút lá: khi đó việc xóa nút không làm ảnh hưởng tới các nút khác, ta chỉ việc xóa bỏ nút đó khỏi cây.
- X chỉ có một nút con (trái hoặc phải): khi đó ta đưa nút con duy nhất của X lên thay cho nút X và xóa bỏ X.
- Còn nếu X là một nút trong và có hai con, ta sẽ có hai lựa chọn, một là tìm nút hậu duệ nhỏ nhất bên nhánh phải của X (gọi là Y), thay khóa của Y lên X và xóa bỏ Y. Cách thứ hai là tìm nút hậu duệ lớn nhất bên nhánh trái của X (gọi là Z), thay khóa của Z lên X và xóa bỏ Z. Các thao tác với Y hoặc Z được lặp lại tương tự như đối với X.

Hình minh họa:



Hình 5.5. Xóa nút trên cây BST, tham khảo từ wikipedia

Do các nút thực sự bị xóa trong trường hợp thứ ba sẽ có thể rơi vào trường hợp 1 hoặc 2 (là các nút lá hoặc các nút chỉ có 1 con), đồng thời nút bị xóa sẽ có khóa nhỏ hơn hai con của X nên trong cài đặt ta nên tránh chỉ sử dụng một phương pháp, vì có thể dẫn tới tình huống mất tính cân bằng của cây.

Việc cài đặt thuật toán xóa một nút trên cây tìm kiếm nhị phân không đơn giản như việc mô tả thuật toán xóa ở trên. Trước hết ta sẽ xuất phát từ gốc của cây để đi tìm nút chứa khóa cần xóa trên cây. Trong quá trình này điều quan trọng là ta xác định rõ nút cần xóa (biến p trong đoạn mã chương trình bên dưới) là một nút lá, hay là một nút chỉ có một con, hay là nút có đầy đủ cả hai con. Dù trong trường hợp nào thì chúng ta cũng cần xác định nút cha của nút p (nút q), và p là con trái hay con phải của q. Để xác định các trường hợp trên ta sử dụng một biến cờ f, f bằng 0 tương ứng với việc nút cần xóa là gốc của cây, f bằng 1 tương ứng với p là con phải của q, và f bằng 2 tương ứng với p là con trái của q.

Cài đặt bằng C của thao tác xóa một nút khỏi cây BST:

```
// xoa bo mot khoa khoi cay
void del(BSTree ** root, int key)
{
  BSTree *p, *q, *r;
  int f=0;
  p = *root;
```

```
q = NULL;
while(p!=NULL&&p->key!=key)
{
        q = p;
        if(p->key<key)
        {
                f = 1;
                p = p->right;
        }
        else
        {
                f = 2;
                p = p->left;
        }
}
if(p!=NULL)
{
        if(p->right==NULL)
        {
                if(f==1)
                {
                        q->right=p->left;
                        free(p);
                }
                else if(f==2)
                {
                        q->left=p->left;
                        free(p);
                }else
                {
                        *root = p->left;
                        free(p);
```

```
                }
        }else
        {
                q = p->right;
                r = NULL;
                while(q->left)
                {
                        r = q;
                        q = q->left;
                }
                p->key = q->key;
                if(r==NULL)
                        p->right = q->right;
                else
                        r->left = q->right;
                free(q);
        }
    }
  }
```

Mặc dù việc xóa cây không phải luôn đòi hỏi phải duyệt từ gốc xuống thực hiện ở một nút lá nhưng tình huống này luôn có thể xảy ra (duyệt qua từng nút tới một nút lá), khi đó độ phức tạp của thuật toán xóa cây tương đương với độ cao của cây (tình huống tồi nhất).

2.5. Tìm kiếm trên cây

Việc tìm kiếm trên cây nhị phân tìm kiếm giống như khi ta thêm một nút mới vào cây. Dựa trên khóa tìm kiếm key ta xuất phát từ gốc, gọi nút đang xét là X. Nếu khóa của X bằng với key, thì kết thúc và trả về X. Nếu X là một nút lá thì kết quả trả về NULL (cũng chính là X). Nếu khóa của X nhỏ hơn key thì ta lặp lại thao tác tìm kiếm với nút con phải của X, ngược lại thì tiến hành tìm kiếm với nút con trái của X.

Độ phức tạp của thuật toán này bằng với độ phức tạp của thuật toán chèn một nút mới vào cây.

Cài đặt của thuật toán được để lại như một bài tập dành cho các bạn độc giả.

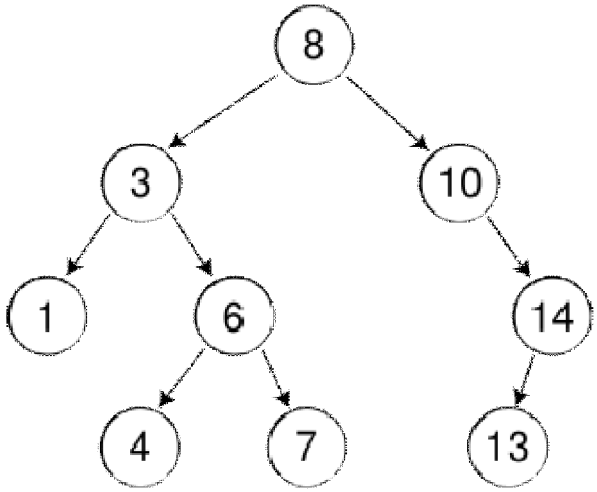
2.6. Duyệt cây

Duyệt cây (tree travel) là thao tác duyệt qua (đến thăm) tất cả các nút trên cây.

Có nhiều cách để duyệt một cây, chẳng hạn như duyệt theo chiều sâu (DFS), duyệt theo chiều rộng (BFS), nhưng ở đây ta phân chia các cách duyệt một cây BST dựa trên thứ tự đến thăm nút gốc, nút con trái, và nút con phải của gốc.

Cụ thể có ba cách duyệt một cây BST: duyệt thứ tự trước, thứ tự giữa, thứ tự sau.

Để minh họa kết quả của các cách duyệt cây ta xét cây ví dụ sau:



Hình 5.6. Cây tìm kiếm nhị phân, tham khảo từ wikipedia

Duyệt thứ tự trước (pre-order traversal):

- Thăm gốc (visit root).
- Duyệt cây con trái theo thứ tự trước
- Duyệt cây con phải theo thứ tự trước.

Cụ thể thuật toán được cài đặt như sau:

```
// duyet theo thu tu truoc
void pre_order(BSTree *node)
{
	if(node!=NULL)
	{
		visit(node); // ham tham mot nut, don gian la in gia tri khoa
		pre_order(node->left);
		pre_order(node->right);
	}
}
```

Kết quả duyệt cây theo thứ tự trước: 8, 3, 1, 6, 4, 7, 10, 14, 13.

Trong cách duyệt theo thứ tự trước, gốc của cây luôn được thăm đầu tiên.

Duyệt thứ tự giữa (in-order traversal):

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

Kết quả duyệt cây theo thứ tự trước: 1, 3, 4, 6, 7, 8, 10, 13, 14.

Một điều dễ nhận thấy là các khóa của cây khi duyệt theo thứ tự giữa xuất hiện theo thứ tự tăng dần.

Duyệt thứ tự sau (post-order traversal):

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

Kết quả duyệt cây theo thứ tự sau: 1, 4, 7, 6, 3, 13, 14, 10, 8.

Trong cách duyệt này, gốc được thăm sau cùng.

Cài đặt bằng C của hai cách duyệt sau được dành cho các bạn độc giả như một bài tập.

2.7. Cài đặt cây BST

Phần cài đặt của cây BST được để lại như một bài tập cho các bạn độc giả.

3. Cây biểu thức (syntax tree)

3.1. Định nghĩa

3.2. Chuyển đổi biểu thức dạng trung tố thành cây biểu thức

3.3. Tính toán giá trị của biểu thức trung tố

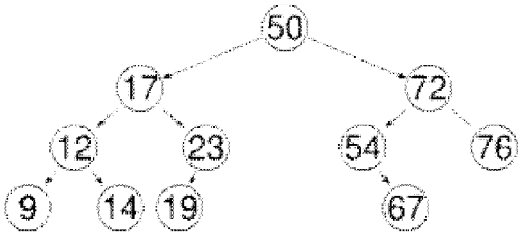
4. Cây cân bằng AVL

4.1. Định nghĩa

Cây AVL là một cây tìm kiếm nhị phân tự cân bằng (self-balancing binary search tree), và là cấu trúc cây cân bằng đầu tiên được phát minh ra. Trong một cây AVL, độ cao của hai nút con của một nút bất kỳ trong cây sai khác nhau nhiều nhất là 1, và do đó thuộc tính này của cây được gọi là thuộc tính cân bằng độ cao (height – balanced). Các thao tác tìm kiếm, chèn một phần tử mới, và xóa bỏ một phần tử khỏi cây đều có độ phức tạp là O(log N) trong cả trường hợp trung bình và trường hợp tồi nhất. Các thao tác làm ảnh hưởng tới tính chất cân bằng của cây là thao tác chèn và xóa bỏ khỏi cây một nút, sau các thao tác này để điều chỉnh lại các nút trong cây nhằm giữ cho cây vẫn được cân bằng, chúng ta cần thực hiện các thao tác quay (xoay – rotation) cây.

Thuật ngữ cây AVL được đặt theo tên của hai tác giả người Nga phát minh ra loại cấu trúc cây này là G.M. Adelson – Velsky và E.M. Landis, trong bài báo của hai người công bố vào năm 1962: “An algorithm for the organization of information”.

Nhân tố cân bằng của một nút bất kỳ trong cây chính là độ cao cây con phải trừ đi độ cao cây con trái của nó. Một nút có nhân tố cân bằng bằng 1, 0 hoặc -1 được coi là đã cân bằng. Một nút với nhân tố cân bằng nhận các giá trị khác sẽ được xem là không cân bằng và cần phải cân bằng lại. Nhân tố cân bằng có thể được lưu tại các nút trong cây hoặc tính dựa trên độ cao các cây con của nó.



Hình 5.1. Cây AVL

Cây AVL tương đối giống với một loại cây cân bằng khác, đó là cây đỏ đen (Red Black Tree) về các thao tác và số thao tác cần thực hiện khi tìm kiếm, thêm, xóa phần tử khỏi cây. Mặc dù cây đỏ đen có tính chất cân bằng không chặt chẽ bằng cây AVL nhưng về hiệu năng thì tương đương nhau, trong một số ứng dụng đòi hỏi thực hiện các thao tác tìm kiếm trên các tập dữ liệu rất lớn thì cây AVL tỏ ra chiếm ưu thế hơn.

4.2. Các thao tác trên cây AVL

4.3. Xoay trên cây AVL

4.4. Cài đặt cây AVL

Tài liệu tham khảo

1. Wikipedia, “Từ điển bách khoa toàn thư trực tuyến tiếng Việt”, http://vi.wikipedia.org/wiki/.
2. Wikipedia, “Từ điển bách khoa toàn thư trực tuyến tiếng Anh”, http://en.wikipedia.org/wiki/Main_Page.
3. Các tài liệu và bài giảng tại website: http://csce.unl.edu/~cusack/Teaching/?page=notes.
4. Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest and Clifford Stein, “Introduction to Algorithms, Second Edition”, The MIT Press, 2001, 1180 pages.
5. Jeff Cogswell, Christopher Diggins, Ryan Stephens, Jonathan Turkanis, “C++ Cookbook”, O’Reilly, November 2005, 592 pages.
6. 158488360X.Chapman & Hall.CRC.Computer Science Handbook, Second Edition.pdf