ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC SƯ PHẠM

TS. PHẠM ANH PHƯƠNG (Chủ biên)
TS. TRẦN VĂN HƯNG, TS. NGUYỄN ĐÌNH LẦU
TS. QUÁCH HẢI THỌ

Giáo trình

Cấu trúc dữ liệu và giải thuật

NHÀ XUẤT BẢN
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC SƯ PHẠM

Phạm Anh Phương (Chủ biên)
Nguyễn Đình Lâu, Trần Văn Hưng, Quách Hải Thọ

GIÁO TRÌNH

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

LỜI MỞ ĐẦU

Theo khung chương trình đào tạo ngành Công nghệ Thông tin ở các hệ Đại học và Cao đẳng, **Cấu trúc dữ liệu và giải thuật** là khối kiến thức cơ bản của ngành Khoa học máy tính, là nền tảng để rèn luyện kỹ năng lập trình đối với các lập trình viên Công nghệ Thông tin.

Giáo trình này trang bị cho người học các nguyên lý thiết kế các cấu trúc dữ liệu cơ bản cùng với các phép toán (thao tác) trên các cấu trúc dữ liệu đó: danh sách đặc (các thuật toán tìm kiếm, sắp xếp,…); danh sách liên kết (với các thao tác: khởi tạo, bổ sung, xóa, duyệt,…); danh sách hạn chế: ngăn xếp, hàng đợi (với các thao tác: khởi tạo, push, pop,...); cây nhị phân, cây tìm kiếm nhị phân (với các thao tác khởi tạo, duyệt, bổ sung, xóa nút…); cấu trúc bảng băm, hàm băm.

Qua nhiều năm nghiên cứu và giảng dạy ở các Trường Đại học Sư phạm - Đại học Đà Nẵng, Đại học Khoa học - Đại học Huế, Đại học Duy Tân, Đại học Công nghệ Thông tin và Truyền thông Việt-Hàn và một số trường đại học khác ở miền Trung và Tây Nguyên, chúng tôi đã cố gắng đúc kết để biên soạn cuốn sách **Giáo trình Cấu trúc dữ liệu và giải thuật** nhằm đáp ứng nhu cầu học tập và nghiên cứu của học sinh, sinh viên và những bạn đọc quan tâm đến lĩnh vực lập trình, giúp bạn đọc có một tài liệu tham khảo tốt khi tìm hiểu sâu về lĩnh vực lập trình.

Nội dung của giáo trình này được chia thành 7 chương, đầu mỗi chương đều có tóm tắt chương để bạn đọc nắm khái quát về nội dung chương, sau đó là nội dung chương được trình bày ngắn gọn từ các kiến thức cơ bản về cách xây dựng cấu trúc dữ liệu đến xây dựng thuật toán và cài đặt mã lệnh. Cuối mỗi chương đều có hệ thống bài tập thực hành từ dễ đến khó, đối với các bài tập khó đều có gợi ý về cách giải. Tất cả các mã nguồn trong giáo trình đều được viết theo phong cách hướng đối tượng, tương thích với trình biên dịch Dev C++ 5.X, đây cũng là một trong những công cụ hỗ trợ lập trình gọn nhẹ, có thể biên dịch được trên cả hai hệ điều hành Windows lẫn Linux và được sử dụng khá phổ biến trong việc học tập và giảng dạy tại các trường học cũng như trong các kỳ thi Olympic Tin học sinh viên và ACM/IPCP Quốc tế.

Chân thành cảm ơn các đồng nghiệp ở các Trường Đại học Sư phạm - Đại học Đà Nẵng, Đại học Bách Khoa - Đại học Đà Nẵng, Đại học Công nghệ Thông tin và Truyền thông Việt-Hàn, Đại học Duy Tân, Đại học Khoa học - Đại học Huế đã giúp đỡ, đóng góp nhiều ý kiến quý báu để chúng tôi hoàn thiện nội dung giáo trình này.

Chúng tôi cũng hy vọng sớm nhận được các ý kiến đóng góp, phê bình của bạn đọc về nội dung, chất lượng và hình thức trình bày để giáo trình ngày một hoàn thiện hơn.

Đà Nẵng, tháng 3 năm 2024

Thay mặt nhóm tác giả

Phạm Anh Phương

Mục lục

Chương 1
TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Tóm tắt chương

- Khái niệm về giải thuật và các tính chất của giải thuật
- Mối liên hệ giữa cấu trúc dữ liệu và giải thuật
- Ngôn ngữ biểu diễn giải thuật
- Độ phức tạp tính toán của giải thuật

1.1. CÁC KHÁI NIỆM

1.1.1. Khái niệm dữ liệu

Trong máy tính, dữ liệu là thông tin đã được mã hóa sang một định dạng khác thuận tiện hơn để có thể xử lý trên máy tính. Đối với khoa học máy tính ngày nay và phương tiện truyền thông, dữ liệu là thông tin đã chuyển đổi thành dạng số nhị phân.

1.1.2. Khái niệm về cấu trúc lưu trữ

Cách biểu diễn một cấu trúc dữ liệu trong bộ nhớ máy tính được gọi là cấu trúc lưu trữ. Có thể có nhiều cấu trúc lưu trữ khác nhau cho một cấu trúc dữ liệu. Chẳng hạn một cấu trúc dữ liệu kiểu danh sách có thể lưu trữ dữ liệu bằng các ô nhớ liên tiếp nhau ở bộ nhớ trong (mảng) hoặc có thể lưu trữ ở các vùng nhớ rời nhau trong bộ nhớ (danh sách liên kết).

Có thể có nhiều cấu trúc dữ liệu khác nhau được cài đặt ở bộ nhớ trong bằng một cấu trúc lưu trữ. Chẳng hạn cấu trúc chuỗi ký tự, cấu trúc mảng đều có thể cài đặt ở bộ nhớ trong bằng các ô nhớ liên tiếp nhau.

1.1.3. Lựa chọn cấu trúc dữ liệu cho bài toán

Lựa chọn cấu trúc dữ liệu thích hợp để tổ chức dữ liệu vào ra và trên cơ sở đó xác lập một giải thuật để đưa ra kết quả mong muốn là một khâu quan trọng.

Việc chọn một cấu trúc dữ liệu phải xét tới các phép toán tác động lên cấu trúc dữ liệu đó. Ngược lại khi xét đến phép toán, cần phải chú ý đến phép toán đó tác động trên cấu trúc dữ liệu nào, bởi vì có phép toán hữu hiệu đối với cấu trúc dữ liệu này nhưng không hữu hiệu với cấu trúc dữ liệu kia.

1.2. GIẢI THUẬT

1.2.1. Khái niệm về giải thuật

Khái niệm giải thuật hay thuật toán dùng để chỉ phương pháp hay cách thức để giải quyết vấn đề.

Giải thuật (algorithrm) là một dãy các câu lệnh chặt chẽ và rõ ràng, xác định trình tự các thao tác trên các đối tượng nào đó (input) sao cho sau một số hữu hạn các bước thực hiện sẽ đạt được kết quả mong muốn (output).

Theo Donald Knuth viết trong cuốn **The Art of Computer Programming,** "*Giải thuật là một thủ tục hữu hạn, xác định và hiệu quả với một số đầu vào (input) và đầu ra (output)*".

Trong cuộc sống hàng ngày chúng ta gặp rất nhiều giải thuật khác nhau cho một bài toán. Ví dụ, tính tổng S = 1 + 2 + … + n.

Cách 1: sử dụng kỹ thuật cộng dồn

- Gán S = 0;
- Cho biến i chạy từ 1 đến n: S = S + i;

Cách 2: Sử dụng công thức của cấp số cộng công bội 1: S = n x (n + 1)/2.

1.2.2. Các tính chất của giải thuật

Các giải thuật đều có một số tính chất chung, biết được các tính chất sẽ rất bổ ích khi mô tả giải thuật.

Dữ liệu vào (input)

Mỗi bài toán đều có giả thiết với một vài đại lượng đầu vào xác định mà ta thường gọi là dữ liệu vào.

Dữ liệu ra (output)

Thuật toán xử lý dữ liệu đầu vào và sẽ thu được một số đại lượng đầu ra xác định. Các đại lượng đầu ra cũng chính là nghiệm hay kết quả của bài toán.

Tính đúng đắn

Yêu cầu bắt buộc của giải thuật là tính đúng đắn, với mỗi bộ dữ liệu đầu vào cho trước, sau một số hữu hạn bước thực hiện sẽ dừng và cho kết quả đúng của bài toán.

Tính xác định

Tính xác định đòi hỏi ở mỗi bước của thuật toán, các thao tác đều phải rõ ràng, không gây ra sự nhập nhằng, lẫn lộn. Nói khác đi là trong cùng một điều kiện, hai bộ xử lý (người hoặc máy) thực hiện cùng một bước của thuật toán phải cho cùng một kết quả. Hơn thế nữa, các bộ xử lý thuật toán không cần phải hiểu được ý nghĩa của các bước ở thao tác này.

Tính hữu hạn (tính dừng)

Với dữ liệu đầu vào xác định, thuật toán bao giờ cũng phải dừng sau một số hữu hạn bước thực hiện và cho kết quả đầu ra.

Tính phổ dụng

Thuật toán thường được xây dựng không chỉ để giải quyết một bài toán riêng lẻ mà phải giải một lớp các bài toán có cùng cấu trúc với dữ liệu cụ thể khác nhau và luôn luôn dẫn đến kết quả mong muốn.

Tính hiệu quả

Tính hiệu quả được đánh giá dựa trên một số tiêu chuẩn nhất định như khối lượng tính toán, thời gian và không gian thực hiện giải thuật.

1.2.3. Mối quan hệ giữa cấu trúc dữ liệu và giải thuật

Khi giải một bài toán trên máy tính, ta thường quan tâm đến việc thiết kế giải thuật. Giải thuật là đặc trưng cho cách xử lý, thường liên quan đến đối tượng để xử lý, tức là dữ liệu của đối tượng đó. Cách thể hiện dữ liệu theo một khuôn dạng nào đó để lưu trữ và xử lý hiệu quả trong máy tính gọi là cấu trúc dữ liệu.

Theo cách tiếp cận của lập trình có cấu trúc, Niklaus Wirth đưa ra công thức thể hiện mối liên hệ giữa cấu trúc dữ liệu và giải thuật như sau:

THUẬT TOÁN + CẤU TRÚC DỮ LIỆU = CHƯƠNG TRÌNH

(Algorithms + Data Structures = Programs)

Khi cấu trúc dữ liệu của bài toán thay đổi, giải thuật cũng phải thay đổi theo cho phù hợp với cách thức tổ chức dữ liệu mới. Ngược lại trong quá trình xây dựng, hoàn thiện giải thuật cũng gợi mở cho người lập trình cách tổ chức dữ liệu cho phù hợp với giải thuật và tiết kiệm tài nguyên hệ thống.

Quá trình giải một bài toán trên máy tính phải chú ý đến mối liên hệ mật thiết giữa giải thuật và cấu trúc dữ liệu. Vì thế khi tiến hành nghiên cứu về cấu trúc dữ liệu cho bài toán đồng thời phải xác lập các giải thuật tương ứng cho cấu trúc dữ liệu đó.

1.3. NGÔN NGỮ BIỂU DIỄN GIẢI THUẬT

Khi thiết kế một giải thuật, chúng ta cần phải trình bày giải thuật đó để kiểm tra giải thuật đã đáp ứng được các yêu cầu chưa (tính đúng đắn, tính phổ dụng, tính hữu hạn...). Qua đó, người đọc có thể hiểu được giải thuật của chúng ta trình bày.

Có nhiều cách thức khác nhau để biểu diễn giải thuật, cụ thể:

- Ngôn ngữ tự nhiên (Natural language)
- Mã giả (Pseudo-code)
- Lưu đồ (Flowchart)

1.3.1. Ngôn ngữ tự nhiên

Để biểu diễn giải thuật theo ngôn ngữ tự nhiên, có thể sử dụng ngôn ngữ đời thường để liệt kê các bước của thuật toán.

Ví dụ 1.1: Tìm số lớn nhất trong ba số a, b, c.

```
max(a,b,c)
Bước 1: Gán m = a;
Bước 2: Nếu b > m, gán m = b;
Bước 3: Nếu c > m, gán m = c;
Bước 4: Kết quả: max(a,b,c) = m;
```

1.3.2. Mã giả

Khi mô tả giải thuật bằng mã giả, ta vay mượn cú pháp của một ngôn ngữ lập trình nào đó để thể hiện giải thuật. Dùng mã giả vừa tận dụng được các khái niệm trong ngôn ngữ lập trình, vừa giúp người sử dụng dễ dàng nắm bắt nội dung của giải thuật. Tất nhiên, trong mã giả vẫn dùng một phần của ngôn ngữ tự nhiên, một khi đã vay mượn cú pháp và khái niệm của ngôn ngữ lập trình thì chắc chắn mã giả sẽ phụ thuộc vào ngôn ngữ lập trình đó.

Ví dụ 1.2: Tìm số lớn nhất trong ba số a, b, c.

```
function max(a,b,c)
begin
    m = a;
    if b > m then m = b;
    if c > m then m = c;
    return m;
end;
```

1.3.3. Lưu đồ

Lưu đồ hay sơ đồ khối là công cụ trực quan để diễn đạt giải thuật. Nếu biết sử dụng khéo léo ngôn ngữ này, ta có thể tránh được những

đoạn giải thích bằng lời có thể dẫn đến sự nhập nhằng về ngữ nghĩa, đồng thời biểu diễn bằng lưu đồ sẽ giúp có được cái nhìn tổng quan hơn về toàn cảnh của quá trình xử lý của một giải thuật cho trước.

Lưu đồ là hệ thống các nút có hình dạng khác nhau, thể hiện các chức năng khác nhau, được nối với nhau bởi các cung. Cụ thể, lưu đồ được tạo bởi 5 thành phần chủ yếu sau đây:

1.3.3.1. Nút giới hạn

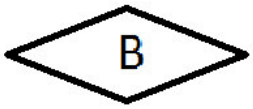
Được biểu diễn bởi hình ôvan, trong đó có ghi chữ: **Begin** hoặc **End**. Chúng còn được gọi là các nút đầu và nút cuối của lưu đồ.

1.3.3.2. Nút thao tác

Là hình chữ nhật trong đó có ghi các lệnh cần thực hiện.

1.3.3.3. Nút điều kiện

Được biểu diễn dưới dạng một hình thoi, bên trong ghi điều kiện cần kiểm tra.



1.3.3.4. Nút xuất/nhập dữ liệu

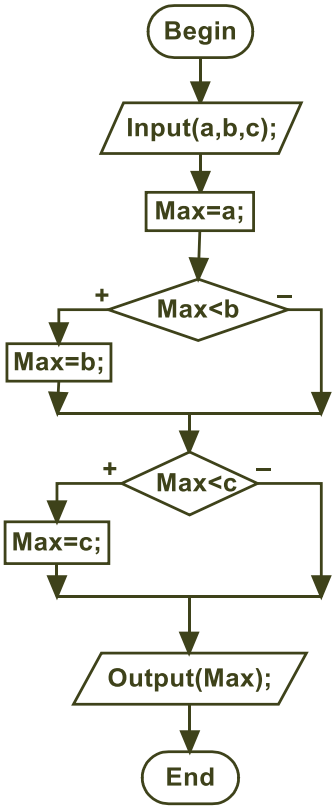
Được biểu diễn dưới dạng một hình bình hành, bên trong ghi các lệnh xuất/nhập: Input(…)/Output(…).

1.3.3.5. Đường đi của thuật toán

Là những đường có hướng nối từ nút này đến nút khác của lưu đồ.

Hoạt động của thuật toán dưới dạng lưu đồ được bắt đầu từ nút đầu tiên. Sau khi thực hiện các thao tác hoặc kiểm tra điều kiện ở mỗi nút, bộ xử lý sẽ theo đường đi của một cung để đến nút khác cho đến khi gặp nút kết thúc thì dừng thuật toán.

Ví dụ 1.3: Tìm số lớn nhất trong ba số a, b, c.



1.4. PHÂN TÍCH VÀ ĐÁNH GIÁ GIẢI THUẬT

1.4.1. Đặt vấn đề

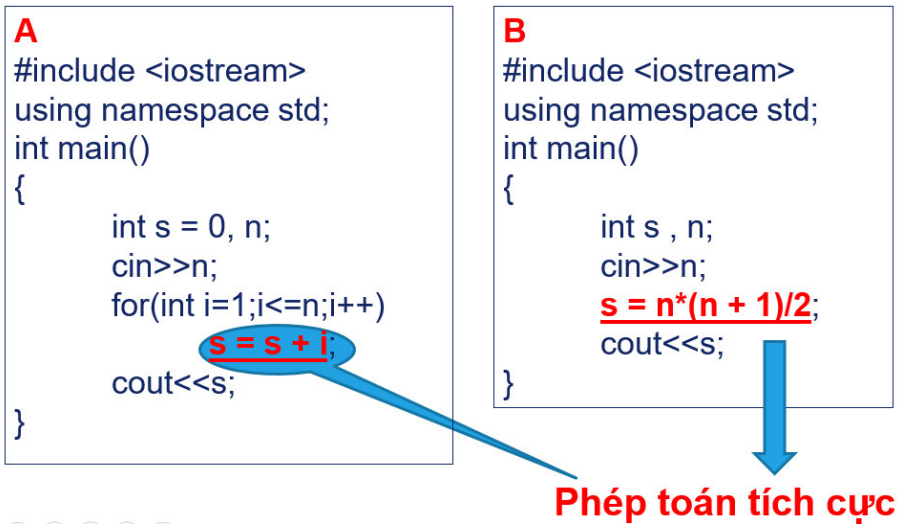
Với một giải thuật đã được thiết kế để giải quyết một bài toán, có nhiều góc độ để đánh giá giải thuật đó. Chẳng hạn:

- Đánh giá tính đúng đắn của giải thuật, liệu giải thuật có cho kết quả đúng với mọi bộ dữ liệu đầu vào hay không?
- Giải thuật có dễ hiểu, dễ cài đặt và dễ chỉnh sửa hay không?
- Giải thuật phải sử dụng bao nhiêu bộ nhớ khi dữ liệu đầu vào có kích thước tương đối lớn?
- Khi thực hiện giải thuật với các bộ dữ liệu tương đối lớn, thời gian thực hiện nhanh hay chậm? Việc đánh giá này được gọi là đánh giá ***thời gian thực hiện giải thuật***.

Trong phạm vi giáo trình chỉ quan tâm đến đánh giá thời gian thực hiện giải thuật vì đây là một tiêu chuẩn quan trọng để đánh giá hiệu quả của giải thuật.

Độ phức tạp thời gian của giải thuật có thể được đánh giá thông qua số ***phép toán tích cực (phép toán được dùng không ít hơn các phép toán khác)*** khi các giá trị đầu vào có kích thước xác định.

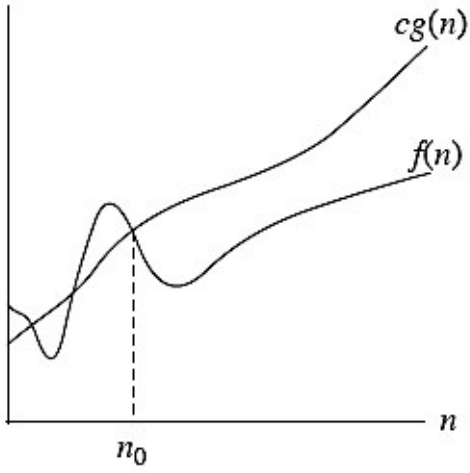
Ví dụ 1.4: Tính tổng S = 1 + 2 + … + n.



Phép toán tích cực của chương trình A thực hiện n lần, trong khi chương trình B chỉ thực hiện 1 lần. Kết luận: chương trình B chạy nhanh hơn chương trình A.

1.4.2. Độ phức tạp tính toán của giải thuật

Cho một giải thuật với kích thước dữ liệu vào là n, thời gian thực hiện giải thuật là một hàm không âm theo n, gọi là *hàm thời gian*, được ký hiệu là f(n), sao cho: $f(n) \geq 0, \forall n \geq 0$.



Hình 1.1: *Hàm f(n) bị chặn trên bởi hàm g(n) kể từ thời điểm* n_0 [7].

Định nghĩa 1.1: Hàm f(n) có cấp bé hơn hoặc bằng hàm g(n) nếu $\exists C>0$ và số tự nhiên n_0 sao cho:

$$|f(n)| \leq C|g(n)| \text{ với mọi } n \geq n_0$$

Ký hiệu: f(n) = O(g(n)) và gọi f(n) thỏa mãn ***quan hệ big-O*** đối với g(n).

Ví dụ 1.5: Hàm $f(n) = \frac{n(n+3)}{2}$ là hàm bậc hai và hàm bậc hai đơn giản nhất là n^2. Ta có:

$$f(n) = \frac{n(n+3)}{2} - O(n^2)$$

vì $\frac{n(n+3)}{2} \leq n^2 \ \forall n \geq 3$ (C=1, n_0=3)

Định nghĩa 1.2: Nếu thuật toán có độ phức tạp là f(n) với f(n) = O(g(n)), ta nói thuật toán có độ phức tạp **O(g(n))**.

Mệnh đề 1.1: Cho $f_1(n)=O(g_1(n))$ và $f_2(n)=O(g_2(n))$. Khi đó:

- $(f_1 + f_2)(n) = O(max(|g_1(n)|,|g_2(n)|)$ (Quy tắc tổng)
- $(f_1f_2)(n) = O(g_1(n)g_2(n))$ (Quy tắc nhân)

Ví dụ 1.6: Cho đoạn chương trình sau (k<m<n).

```
int d=0;
for (int i=1;i<=k;i++) d++;
for (int j=1;j<=m;j++) d++;
for (int h=1;h<=n;h++) d++;
```

Với các vòng lặp rời nhau, theo quy tắc tổng, độ phức tạp của đoạn chương trình trên sẽ là: O(max(k,m,n)) = O(n)

Ví dụ 1.7: Cho đoạn chương trình sau

```
int d=0;
  for (int i=1;i<=n;i++)
    for (int j=1;j<=n;j++)
      for (int h=1;h<=n;h++) d++;
```

Với các vòng lặp lồng nhau, theo quy tắc nhân, độ phức tạp của đoạn chương trình trên sẽ là: $O(n\times n\times n) = O(n^3)$.

Ví dụ 1.8: Đánh giá độ phức tạp của thuật toán sắp xếp sau:

```
void Sort(int n,int A[])
{
  for(int i=1;i<n;i++)
    for(int j=i+1;j<=n;j++)
      if(A[i]>A[j])
        swap(A[i],A[j]);
}
```

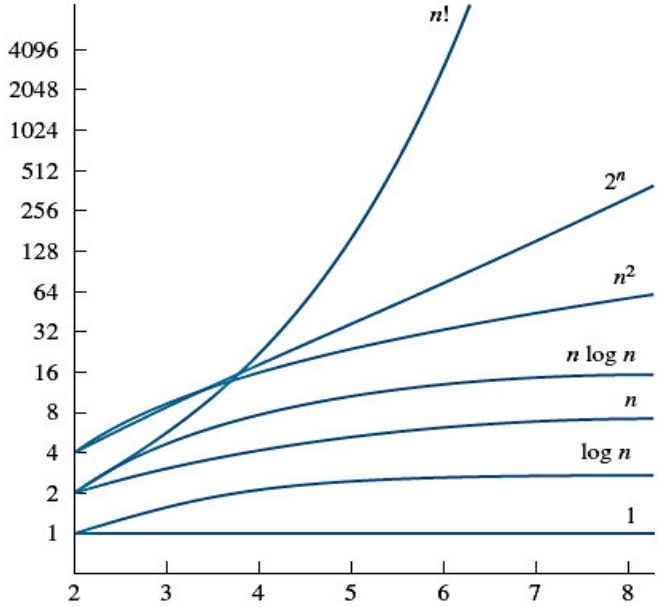
Nhận xét:

Với i=1 ➔ j = n-1
i=2 ➔ j = n-2
…
i=n-2 ➔ j = 2
i=n-1 ➔ j = 1
f(n) = (n-1).n/2
➔g(n) = n^2

Vậy, độ phức tạp của thuật toán sắp xếp: O(n^2).

1.4.3. Độ phức tạp của một số thuật toán thông dụng

- Thuật toán tìm kiếm tuyến tính: O(n)
- Thuật toán tìm kiếm nhị phân: O($\log_2 n$)
- Thuật toán sắp xếp chọn, chèn, nổi bọt: O(n^2)
- Thuật toán Quick sort: O($n\log_2 n$)
- Liệt kê dãy nhị phân độ dài n: O(2^n)
- Liệt kê các hoán vị của n phần tử: O(n!)



Hình 1.2: *Biểu đồ tăng dần về độ phức tạp của giải thuật* [7]

BÀI TẬP CHƯƠNG 1

Bài 1.1: Xây dựng giải thuật để tìm số lớn nhất trong một dãy số nguyên có n phần tử: a_1, a_2,…, a_n.

Bài 1.2: Xây dựng giải thuật để tính x^n, trong đó x là số thực, n là số nguyên không âm. Xác định độ phức tạp của giải thuật.

Bài 1.3: Cho dãy số nguyên có n phần tử: a_1, a_2,…, a_n. Xây dựng giải thuật để tìm kiếm giá trị X có trong dãy hay không. Xác định độ phức tạp của giải thuật.

Bài 1.4: Cho dãy số nguyên có n phần tử: a_1, a_2,…, a_n đã xếp thứ tự tăng dần. Xây dựng giải thuật để tìm kiếm giá trị X có trong dãy hay không. Xác định độ phức tạp của giải thuật.

Bài 1.5: Xây dựng giải thuật để đếm số lần xuất hiện của một chuỗi con trong một chuỗi cho trước.

Bài 1.6: Xác định độ phức tạp tính toán của các giải thuật với hàm thời gian của chúng được cho như sau:

a) $f(n) = (2+n).(3+\log n)$

b) $f(n) = (n^3+2n)/(2n+1)$

c) $f(n) = n.(2+n) - 7.n$

d) $f(n) = \log n^2 + n$

Bài 1.7: Xác định độ phức tạp tính toán của đoạn chương trình sau:

```
for(int i=1;i<=n;i++)
  for(int j=1;j<=n;j++)
  {
    C[i][j]=0;
    for(int k=1;k<=n;k++)
      C[i][j]= C[i][j] + A[i][k]*B[k][j];
  }
```

Chương 2
GIẢI THUẬT ĐỆ QUY

Tóm tắt chương

- Định nghĩa đệ quy
- Cấu trúc của giải thuật đệ quy
- Phương pháp xây dựng giải thuật đệ quy
- Phân loại đệ quy
- Giải thuật quay lui

2.1. GIỚI THIỆU

Cách để mô tả sự lặp lại trong chương trình máy tính là sử dụng các vòng lặp, chẳng hạn như cấu trúc lặp ***while*** và ***for*** của C/Java.

Ngoài ra, còn có cách khác để mô tả sự lặp lại trong chương trình máy tính thông qua một quá trình được gọi là **đệ quy**.

Đệ quy là kỹ thuật mà một hàm thực hiện một hoặc nhiều ***lệnh gọi đến chính nó*** trong quá trình thực thi, theo đó dữ liệu dựa trên các thể hiện với quy mô nhỏ hơn của cùng một kiểu cấu trúc trong biểu diễn của nó.

Đệ quy là một công cụ thường dùng trong khoa học máy tính, nó có một ý nghĩa đặc biệt trong định nghĩa quy nạp toán học.

Trong lập trình, đệ quy cung cấp một giải pháp thay thế tinh tế và mạnh mẽ để thực hiện các tác vụ lặp đi lặp lại. Hầu hết các ngôn ngữ lập trình hiện đại đều hỗ trợ lập trình đệ quy.

Đệ quy là một kỹ thuật quan trọng trong nghiên cứu cấu trúc dữ liệu và thuật toán.

2.2. ĐỊNH NGHĨA ĐỆ QUY

Một đối tượng được gọi là đệ quy nếu nó hoặc một phần của nó được định nghĩa thông qua khái niệm về chính nó.

Một hàm được gọi là đệ quy nếu trong hàm đó có lời gọi đến chính nó.

Tổng quát: *Nếu một lời giải của bài toán T được thực hiện bằng lời giải của bài toán T', có dạng giống như T, đó là lời giải đệ quy. Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ quy. Nếu giải thuật ấy được viết dưới dạng một hàm, hàm ấy được gọi là hàm đệ quy.*

2.3. CẤU TRÚC CỦA GIẢI THUẬT ĐỆ QUY

Một giải thuật đệ quy bao giờ cũng gồm có hai thành phần:

Thành phần dừng (phần neo/suy biến): Không chứa khái niệm đang định nghĩa. Phần này xác định điểm dừng của giải thuật đệ quy. Trường hợp này còn được gọi là trường hợp suy biến. Nếu giải thuật đệ quy không có trường hợp suy biến, sẽ dẫn đến lặp vô hạn và sinh lỗi khi thực thi chương trình.

Thành phần đệ quy: Có chứa khái niệm đang định nghĩa. Trường hợp này là phân tích và xây dựng trường hợp chung của bài toán (nghĩa là đưa bài toán về cùng loại nhưng với dữ liệu có kích thước "nhỏ" hơn và mục đích là đưa bài toán tiến dần về phần neo).

2.4. PHƯƠNG PHÁP XÂY DỰNG GIẢI THUẬT ĐỆ QUY

Khi xây dựng giải thuật đệ quy ta tiến hành những bước sau:

Bước 1: Tham số hóa bài toán.

Bước 2: Xác định trường hợp suy biến.

Bước 3: Phân tích và xây dựng trường hợp chung của bài toán (thành phần đệ quy) có nghĩa là đưa bài toán về dạng bài toán cùng loại nhưng với kích thước dữ liệu nhỏ hơn.

Ví dụ 2.1: Viết hàm đệ quy để tính n! = 1.2...n.

- Tham số hóa: n! = Factorial(n);
- Factorial(0) = 1 (trường hợp suy biến)
- Factorial(n) = n*Factorial(n-1) (trường hợp chung)

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

Ví dụ 2.2: Viết hàm in ra biểu diễn nhị phân của một số nguyên.

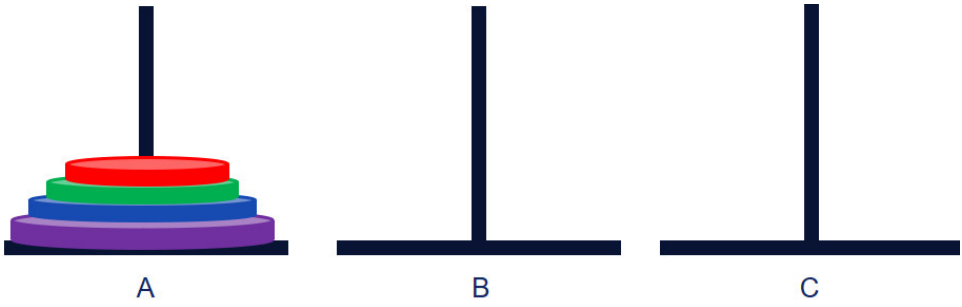
- Tham số hóa: Bin(n);
- n=0: kết thúc (trường hợp suy biến)
- n>0: lưu lại **cout<<n%2;**
  và thực hiện tiếp **Bin(n/2)** (trường hợp chung)

```
void Bin(int n)
{
  if (n> 0)
  {
    Bin(n/2)
    cout<<n%2;
  }
}
```

Ví dụ 2.3: Bài toán tháp Hà Nội (**Lucas' Tower** 1883)

Có n đĩa với kích thước nhỏ dần xếp chồng lên nhau. Đĩa to nằm dưới, đĩa nhỏ nằm trên. Yêu cầu bài toán: Chuyển chồng đĩa từ cọc A sang cọc C với điều kiện sau:

- Mỗi lần chỉ được chuyển một đĩa.
- Được phép dùng cọc B làm cọc trung gian để trung chuyển.
- Không được đặt đĩa to nằm ở trên đĩa nhỏ.



Hình 2.1: *Bài toán tháp Hà Nội*

Bước 1: Tham số hóa bài toán.

Gọi hàm ***ThapHN(n, A, B, C)*** là hàm chuyển n đĩa từ cọc A sang cọc C (lấy cọc B làm trung gian).

Bước 2: Trường hợp suy biến

Với n = 1: Chuyển đĩa từ cột A sang cột C là xong.

Bước 3: Phân tích trường hợp tổng quát.

* **Xét trường hợp n=2**: thực hiện 3 phép chuyển:

- Chuyển đĩa thứ nhất từ cọc A sang cọc B: ThapHN(1,A,C,B);
- Chuyển đĩa thứ hai từ cọc A sang cọc C: ThapHN(1,A,B,C);
- Chuyển đĩa thứ nhất từ cọc B sang cọc C: ThapHN(1,B,A,C);

* **Tổng quát hóa với n ≥ 2**: Xem (n-1) đĩa nằm ở trên đóng vai trò như 1 đĩa, có thể hình dung đang có 2 đĩa trên cọc A. Nếu mô phỏng như trường hợp n = 2, giải thuật chuyển như sau:

- Chuyển (n-1) đĩa từ cọc A sang cọc B: ThapHN(n-1,A,C,B);
- Chuyển 1 đĩa từ cọc A sang cọc C: ThapHN(1,A,B,C);
- Chuyển (n-1) đĩa từ cọc B sang cọc C: ThapHN(n-1,B,A,C);

Như vậy, bài toán tháp Hà Nội có thể cài đặt như sau:

```
void ThapHN(int n,char A,char B,char C)
{
  if(n==1) cout<<endl<<   "Chuyen dia tu
                          "<<A<<" qua "<<C;
  else
  {
    ThapHN(n-1,A,C,B);
    ThapHN(1,A,B,C);
    ThapHN(n-1,B,A,C);
   }
}
```

2.5. ƯU VÀ NHƯỢC ĐIỂM CỦA ĐỆ QUY

2.5.1. Ưu điểm

- Cung cấp cơ chế giải quyết bài toán phức tạp một cách đơn giản.
- Chương trình trong sáng, ngắn gọn.
- Dễ dàng chuyển thành mã lệnh trên các ngôn ngữ lập trình.

2.5.2. Nhược điểm

- Tốn bộ nhớ
- Xử lý chồng chéo nên mất nhiều thời gian dẫn đến giảm tốc độ chạy chương trình.
- Không thể áp dụng cho mọi ngôn ngữ, ví dụ như Fortran.

2.6. PHÂN LOẠI ĐỆ QUY

2.6.1. Đơn đệ quy

Trong định nghĩa đơn đệ quy (simple recursion) chỉ có duy nhất một lần gọi đệ quy.

Ví dụ 2.4: Định nghĩa hàm tính x^n.

Hàm được định nghĩa đệ quy: **pow**(x,n)

$$\textbf{pow}(x,n) = \begin{cases} 1, & n = 0 \\ x.\boldsymbol{pow}(x, n-1), & n > 0 \end{cases}$$

Thuật toán cài đặt:

```
function pow(x,n)
begin
    if(n==0) return 1;
    else return x*pow(x,n-1);
end
```

2.6.2. Đa đệ quy

Trong định nghĩa đa đệ quy (multiple recursion) có nhiều hơn một lần gọi đệ quy.

Ví dụ 2.5: Định nghĩa dãy số Fibonacy.

Hàm được định nghĩa đệ quy: **Fib**(n)

$$\textbf{Fib}(n) = \begin{cases} 1, & n = 0 \; or \; n = 1 \\ \boldsymbol{Fib}(n-1) + \boldsymbol{Fib}(n-2), & n > 1 \end{cases}$$

Thuật toán cài đặt:

```
function Fib(n)
begin
    if(n<2) return 1;
    else return Fib(n-1) + Fib(n-2);
end
```

2.6.3. Đệ quy chéo

Trong đệ quy chéo (mutual recursion), các định nghĩa của chúng phụ thuộc lẫn nhau.

Ví dụ 2.6: Định nghĩa số chẵn/lẻ.

Định nghĩa đệ quy: **even**(n), **odd**(n)

$$\textbf{even}(n) = \begin{cases} true, & n = 0 \\ \boldsymbol{odd}(n-1), & n > 0 \end{cases}$$

$$\textbf{odd}(n) = \begin{cases} false, & n = 0 \\ \boldsymbol{even}(n-1), & n > 0 \end{cases}$$

Thuật toán cài đặt:

```
bool even(n)
{
  if(n==0) return true;
  else return odd(n-1);
}

bool odd(n)
{
  if(n==0) return false;
  else return even(n-1);
}
```

2.6.4. Đệ quy chồng

Trong đệ quy chồng (implicated recursion), các định nghĩa của chúng được gọi lồng nhau.

Ví dụ 2.7: Định nghĩa hàm Ackermann

Định nghĩa đệ quy: **A**(m,n)

$$\mathbf{A}(m,n) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & n = 0 \\ A\big(m-1, A(m,n-1)\big), & m,n > 0 \end{cases}$$

Thuật toán cài đặt:

```
function A(m,n)
begin
  if(m=0) return n+1;
  else if(n=0) return A(m-1,1);
    else return A(m-1,A(m,n-1));
end
```

2.6.5. Đệ quy đuôi

Trong định nghĩa đệ quy đuôi (tail recursion) chỉ có duy nhất một lần gọi đệ quy ở cuối hàm cài đặt.

Ví dụ 2.8: Hàm **Tail** là đệ quy đuôi, còn hàm **nonTail** không phải là đệ quy đuôi.

```
void Tail(int n)
{
  if(n >0)
  {
    cout<<n<<" ";
    Tail(n-1);
  }
}

void nonTail(int n)
{
  if(n >0)
  {
    nonTail(n-1);
    cout<<n<<" ";
    nonTail(n-1);
  }
}
```

2.7. GIẢI THUẬT QUAY LUI

*** Bài toán**: Xây dựng các bộ giá trị gồm n phần tử (x_1,...,x_n) từ một tập hữu hạn cho trước sao cho các bộ đó thỏa mãn một yêu cầu B nào đó.

*** Phương pháp**

Giả sử đã xác định được k-1 phần tử đầu tiên của bộ giá trị là x_1,...,x_{k-1}. Cần xác định phần tử thứ k tiếp theo, phần tử này được xác định theo cách sau:

- Giả sử T_k là tập tất cả các giá trị mà x_k có thể nhận. Vì T_k hữu hạn nên có thể đặt n_k là số phần tử của T_k theo một thứ tự nào đó, nghĩa là có thể thành lập một ánh xạ 1-1 từ tập T_k lên tập $\{1,2,...,n_k\}$
- Xét $j \in \{1,2,...,n_k\}$, ta nói rằng "**j chấp nhận được**" nếu có thể bổ sung phần tử thứ j trong T_k với tư cách là phần tử x_k vào dãy **x_1,...,x_{k-1}** để được dãy **x_1,...,x_k**.
- Nếu k = n: bộ (x_1,...,x_k) thỏa mãn yêu cầu B $\Rightarrow$ bộ này sẽ được liệt kê.
- Nếu k < n: cần lặp lại quá trình trên, tức là phải bổ sung tiếp phần tử x_{k+1} vào dãy x_1,...,x_k.

✍ NHẬN XÉT

Nét đặc trưng của giải thuật quay lui là muốn có được lời giải, phải đi từng bước bằng phép thử. Khi một bước lựa chọn thỏa mãn, cần ghi nhận kết quả và tiến hành các bước tiếp theo. Ngược lại, khi không có lựa chọn nào thỏa mãn, phải ***quay lui bước trước đó***, xóa bớt các trường hợp đã đi qua và thử với các lựa chọn còn lại.

Sau đây là hàm đệ quy mô tả giải thuật quay lui:

```
void  Thu(int k)
{
  int j;
  for(j=1;j<=nk;j++)   //Duyệt qua các giá
                       //trị của Tk
```

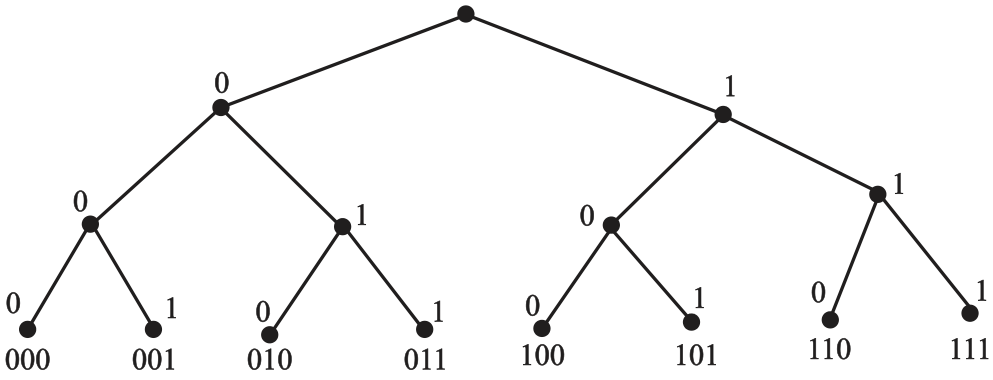
```
    {
      <Xác định xk theo j>;
      if(k==n) <Ghi nhận 1 bộ giá trị>;
      else Thu(k+1); //Quay lui
    }
  }
```

Ví dụ 2.9: Viết chương trình liệt kê các dãy nhị phân có độ dài n.

```
#include <iostream>
using namespace std;
class BinSeq
{
  int n,x[100];
  public:
    void Input()
    {
      cout<<endl<<"Nhap do dai xau nhi phan: ";
      cin>>n;
    }
    void InKetQua()
    {
        for(int i=1;i<=n;i++) cout<<x[i]<<" ";
        cout<<endl;
    }
    void  Thu(int k)  //Tìm phần tử xk
    {
      for(int j=0;j<=1;j++)
      {
        x[k]=j;              //Xác định xk theo j
        if(k==n) InKetQua();
        else Thu(k+1); //Quay lui: tìm xk+1
```

```
        }
    }
};
int main()
{
    BinSeq t;
    t.Input();
  t.Thu(1);
return 0;
}
```

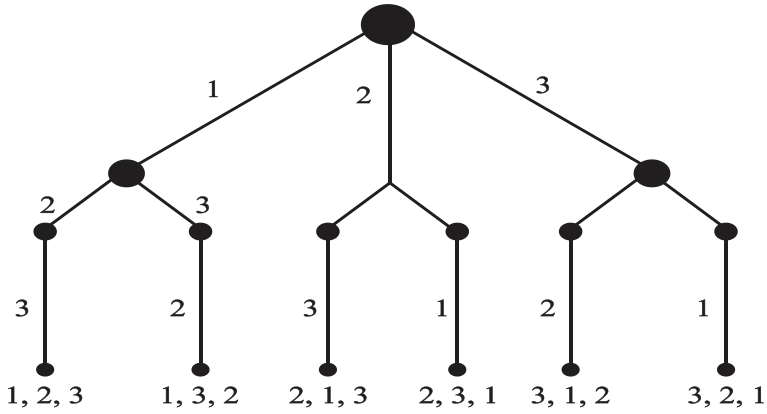
Với n = 3, kết quả nhận được như sau:



Hình 2.2: *Kết quả liệt kê dãy nhị phân với n = 3*

Ví dụ 2.10: Viết chương trình liệt kê các hoán vị của {1,2,...,n}.

- Biểu diễn các hoán vị dưới dạng $x_1,x_2,...,x_n$, trong đó $x_i \in [1,n]$ và $x_i \neq x_j$ $(i \neq j)$.
- Các giá trị từ 1 tới n sẽ lần lược đề cử cho p_i, trong đó $j \in [1,n]$ được chấp nhận ***nếu nó chưa được dùng đến***. Vì vậy cần tạo ra một dãy biến logic b_j để xét xem j đã được dùng hay chưa.
- Gán b_j = TRUE nếu j chưa được dùng, ngược lại b_j = FALSE.
- Ta có thể hình dung bài toán như hình vẽ sau: Với n=3, bài toán trở thành liệt kê các hoán vị của các phần tử 1, 2, 3. Các hoán vị được liệt kê theo thứ tự từ điển tăng dần như hình vẽ sau:



Hình 2.3: *Kết quả liệt kê các hoán vị với n = 3*

```
#include <iostream>
using namespace std;
class Permutation
{
      int n, x[100];
      bool b[100];
    public:
      void InKQ()
      {
        for(int i=1;i<=n;i++) cout<<x[i]<<" ";
        cout<<endl;
      }
      void Init()
      {
        cout<<"Nhap n = "; cin>>n;
        for(int i=1;i<=n;i++) b[i] = true; //chưa
                                           //dùng
      }
      void Thu(int k) //Tìm phần tử xk
      {
        for(int j=1;j<=n;j++)
```

```
        {
            if(b[j])  //nếu j chưa sử dụng
            {
              x[k] =j; //xác định x_k theo j
              b[j] = false;  //j đã sử dụng
              if(k==n) InKQ();
              else Thu(k+1);
              b[j] = true;//trả lại trạng thái cũ
            }
        }
      }
};
int main()
{
    Permutation t;
    t.Init();
    t.Thu(1);
    return 0;
}
```

BÀI TẬP CHƯƠNG 2

Bài 2.1: Viết hàm tìm ước chung lớn nhất của hai số nguyên dương a, b theo 2 cách: đệ quy và không đệ quy.

Bài 2.2: Viết hàm đếm số chữ số của một số nguyên dương n theo hai cách: đệ quy và không đệ quy.

Bài 2.3: Dãy số Fibonacci được định nghĩa như sau:

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

Viết hàm để tính F(n) theo hai cách: đệ quy và không đệ quy.

Bài 2.4: Xét định nghĩa đệ quy

$$A(m,n) = \begin{cases} n+1 & , \quad m = 0 \\ A(m-1,1) & , \quad n = 0 \\ A(m-1, A(m,n-1)) & , \quad m > 0 \wedge n > 0 \end{cases}$$

Hàm này được gọi là hàm Ackermann.

1. Tính A(1,2)?
2. Viết hàm đệ quy để tính A(m,n).

Bài 2.5: Một số nguyên dương được gọi là đối xứng nếu chữ số thứ nhất bằng chữ số cuối, chữ số thứ hai bằng chữ số gần cuối,... Viết hàm đệ quy để kiểm tra số nguyên dương n có phải là số đối xứng hay không.

Bài 2.6: Viết các hàm đệ quy và không đệ quy để tính:

$S_1 = 1+2+3+.....+n$;

$S_2 = 1+1/2 + ...+ 1/n$;

$S_3 = 1-1/2 +.....+ (-1)^{n+1}\ 1/n$

$S_4 = 1 + \sin(x) + \sin^2(x) + ...+ \sin^n(x)$

Bài 2.7: Viết hàm đệ quy để tính C^k_n biết :

$C^n_n = 1$, $C^0_n = 1$, $C^k_n = C^{k-1}_{n-1} + C^k_{n-1}$.

Bài 2.8: Viết hàm để in ra màn hình số đảo ngược của một số nguyên cho trước theo hai cách: đệ quy và không đệ quy.

Bài 2.9: Cho mảng số nguyên có n phần tử: $a_1, a_2,..., a_n$. Viết các hàm đệ quy:

1. Tính tổng các phần tử của mảng.
2. Tìm giá trị lớn nhất trong mảng.
3. Kiểm tra phần tử x có trong mảng hay không?
4. Kiểm tra mảng đã được sắp xếp theo thứ tự không giảm chưa?

Bài 2.10*: Viết chương trình cài đặt bài toán 8 quân hậu: đặt 8 quân hậu lên bàn cờ 8 x 8 sao cho các quân hậu không thể ăn lẫn nhau.

Chương 3
DANH SÁCH ĐẶC

Tóm tắt chương

- Khái niệm danh sách
- Các thao tác cơ bản: duyệt danh sách, chèn, xóa một phần tử
- Các thuật toán sắp xếp
- Tìm kiếm

3.1. KHÁI NIỆM DANH SÁCH

Danh sách là tập hợp gồm nhiều phần tử (element) $a_1, a_2, \ldots, a_n$ với tính chất cấu trúc của nó là mối liên hệ tương đối giữa các phần tử với nhau: nếu biết được phần tử a_i, sẽ biết được vị trí của phần tử a_{i+1}.

Số phần tử của danh sách được gọi là chiều dài của danh sách. Một danh sách có chiều dài bằng 0 là danh sách rỗng.

Một tính chất quan trọng của danh sách là các phần tử có thể được sắp xếp tuyến tính theo vị trí của chúng trong danh sách.

Danh sách đặc (condensed list) là danh sách mà các phần tử được sắp xếp kế tiếp nhau trong bộ nhớ, đứng ngay sau vị trí phần tử a_i là vị trí phần tử a_{i+1}.

3.2. CÁC THAO TÁC TRÊN DANH SÁCH ĐẶC

Thông thường, ta sử dụng mảng một chiều để lưu trữ danh sách đặc. Giả sử danh sách được lưu trong mảng A[ ] có n phần tử.

3.2.1. Duyệt danh sách

Duyệt danh sách là công việc thăm tất cả các phần tử của danh sách mà không được bỏ sót phần tử nào. Thông thường ta dùng vòng lặp để duyệt qua tất cả các phần tử của danh sách.

Giải thuật duyệt mảng có thể được thực hiện như sau:

```
for (int i=1; i<=n; i++) <Thăm A[i]>;
```

3.2.2. Chèn một phần tử vào danh sách

Đầu vào: Mảng A có n phần tử, vị trí chèn k, giá trị cần chèn X.

Đầu ra: Mảng A có n+1 phần tử.

Giải thuật:

Bước 1: Dời các phần tử của mảng A từ vị trí thứ k sang phải một vị trí

for(i=n; i>=k; i--) A[i+1] = A[i];

Bước 2: Chèn giá trị X vào vị trí k

A[k] = X;

Bước 3: Tăng kích thước của mảng lên 1 đơn vị

n++;

3.2.3. Xóa một phần tử ra khỏi danh sách

Muốn xóa phần tử tại vị trí k trong mảng A có n phần tử ($1 \le k \le n$), ta thực hiện giải thuật như sau:

Bước 1: Dời các phần tử của mảng A từ vị trí thứ k+1 qua trái một vị trí.

for (i=k; i<n; i++) A[i] =A[i+1];

Bước 2: Số phần tử của mảng sẽ giảm bớt 1.

n—

3.2.4. Ưu và nhược điểm khi dùng mảng

3.2.4.1. Ưu điểm

- Tốc độ truy cập nhanh.
- Tổ chức và cài đặt đơn giản.

3.2.4.2. Nhược điểm

- Số phần tử của mảng phải được xác định trước nên gây lãng phí không gian lưu trữ nếu không sử dụng hết số lượng đã khai báo.
- Nếu việc chèn và xóa các phần tử diễn ra liên tục, tốc độ xử lý sẽ rất chậm.

Ví dụ 3.1: Sử dụng danh sách đặc để lưu trữ và tính giá trị của đa thức

$$P(x) = a_0 + a_1.x + a_2.x^2 + \ldots + a_n.x^n$$

Ví dụ, với đa thức $P(x) = 3 + 2x^3 – 6x^4$, ta sử dụng mảng một chiều để lưu trữ các hệ số của đa thức như sau: $a_0 = 3$, $a_1 = 0$, $a_2 = 0$, $a_3 = 2$ và $a_4 = -6$.

	0	1	2	3	4
a	3	0	0	2	-6

Sau đây là chương trình nhập vào đa thức bậc n với các hệ số của đa thức lưu trong mảng a[ ] và nhập số thực x. Sau đó in đa thức đã nhập ra màn hình và tính giá trị của đa thức.

```
#include <iostream>
#include <math.h>
using namespace std;
class Polynomial
{
int n;
float a[1000];
    public:
      float x;
void Input() //Nhập các hệ số của đa thức
{
        cout<<"Bac da thuc: n = "; cin>>n;
        for(int i=0;i<=n;i++)
        {
```

```
            cout<<"A["<<i<<"]="; cin>>a[i];
        }
        cout<<endl<<"x = "; cin>>x;
}
void Display() //Hiển thị đa thức
{
        cout<<endl;
        cout<<"f(x)=";
        if(a[0]) cout<<a[0];
        if(a[1]<0)cout<<a[1]<<"x";
        else if(a[1]>0) cout<<"+"<<a[1]<<"x";
        for(int i=2;i<=n;i++)
          if(a[i]<0)cout<<a[i]<<"x^"<<i;
          else if(a[i]>0)
          cout<<"+"<<a[i]<<"x^"<<i;
}
float f(float x) //Tính giá trị đa thức
{
        float s=a[0];
        for(int i=1;i<=n;i++) s+=a[i]*pow(x,i);
        return s;
}
};
int main()
{
Polynomial t;
    t.Input();
    t.Display();
    cout<<endl<<"f("<<t.x<<")="<<t.f(t.x);
    return 0;
}
```

3.3. CÁC THUẬT TOÁN SẮP XẾP

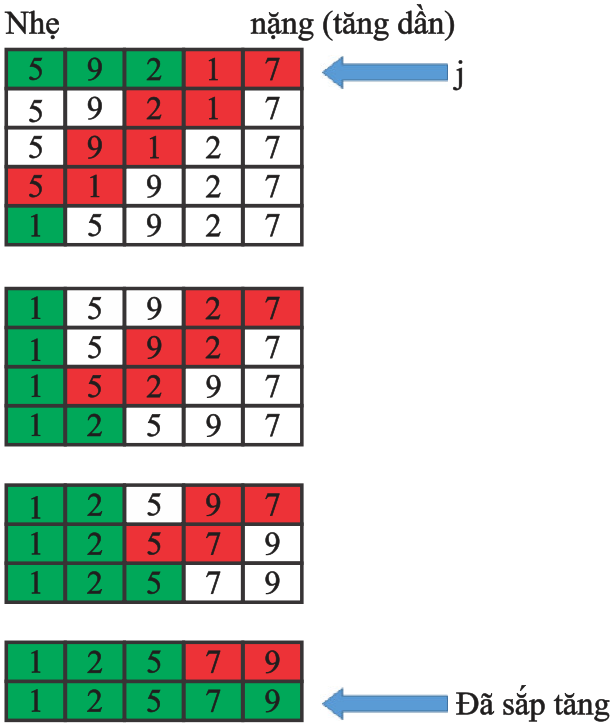
3.3.1. Sắp xếp nổi bọt (Bubble sort)

Ý tưởng

Đi từ cuối mảng về đầu mảng, trong quá trình đi nếu phần tử ở dưới (phía sau) nhỏ hơn phần tử đứng ngay trên (trước) nó, theo nguyên tắc của bọt khí "phần tử nhẹ" sẽ "trồi" lên phía trên "phần tử nặng" (hai phần tử này sẽ được đổi chỗ cho nhau). Kết quả là phần tử nhỏ nhất (nhẹ nhất) sẽ được đưa lên (trồi lên) trên bề mặt (đầu mảng).

Sau mỗi lần đi chúng ta sẽ được một phần tử trồi lên đúng chỗ. Như vậy sau n-1 lần đi, tất cả các phần tử trong mảng được sắp xếp theo thứ tự tăng dần.

Ví dụ 3.2: Sắp xếp dãy số 5, 9, 2, 1, 7 theo thứ tự không giảm.



Hình 3.1: *Sắp xếp nổi bọt*

Cài đặt thuật toán

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

Thuật toán Bubble Sort có độ phức tạp là **O(n^2)**.

3.3.2. Sắp xếp chọn (Selection sort)

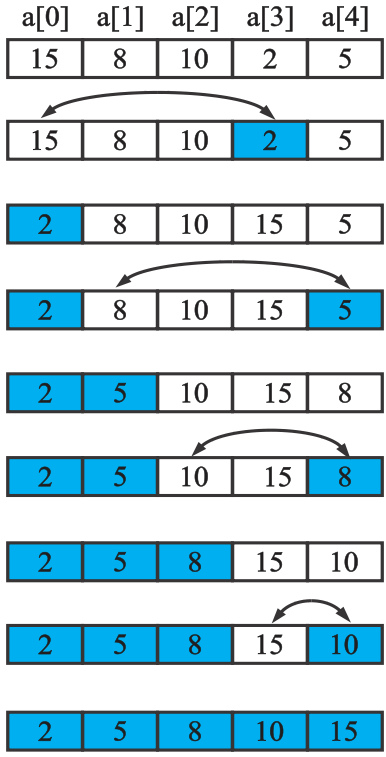
Ý tưởng

Giả sử dãy a có n phần tử chưa có thứ tự. Chọn phần tử có giá trị nhỏ nhất trong n phần tử chưa có thứ tự này để đưa lên đầu nhóm có n phần tử.

Tiếp tục chọn phần tử có giá trị nhỏ nhất trong n-1 phần tử chưa có thứ tự này để đưa lên đầu nhóm của n-1 phần tử,…

Sau n-1 lần lựa chọn phần tử nhỏ nhất để đưa lên đầu nhóm, tất cả các phần tử trong dãy a sẽ có thứ tự tăng dần.

Ví dụ 3.3:



Hình 3.2: *Sắp xếp chọn*

Cài đặt thuật toán

```
void SelectionSort(int a[], int n)
{
for (int i=1; i<n; i++)
{
  //tìm vị trí phần tử nhỏ nhất: min_pos
  int min = a[i], min_pos = i;
  for(int j=i+1; j<n; j++)
    if (a[j]<min)
    {
      min =a[j];
      min_pos = j;
    }
```

```
    //Hoán đổi a[i] với a[min_pos]
    swap(a[i],a[min_pos]);
    }
}
```

Thuật toán Selection Sort có độ phức tạp là $\mathbf{O(n^2)}$.

3.3.3. Sắp xếp chèn (Insertion Sort)

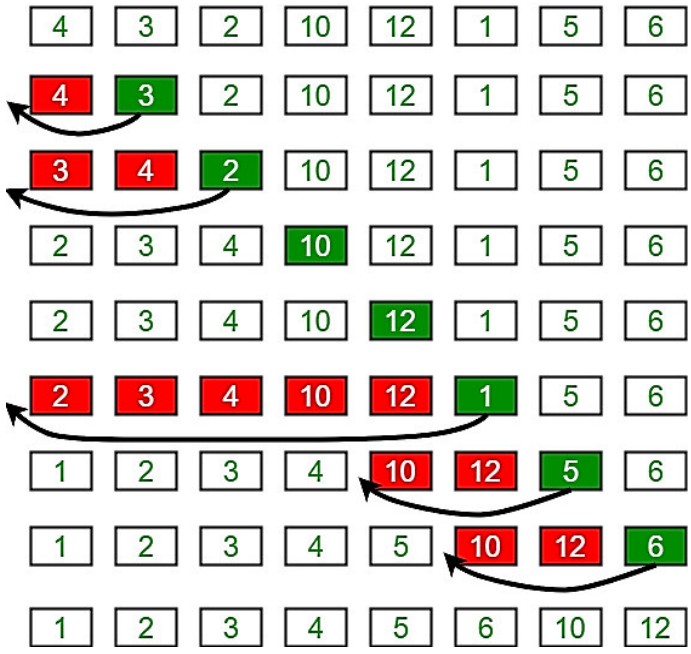
Ý tưởng

Giả sử dãy a có i-1 phần tử đã có thứ tự. Cần chèn phần tử a[i] vào dãy sao cho vẫn đảm bảo thứ tự trong dãy.

Bắt đầu bằng cách xét phần tử đầu tiên của mảng, đó là phần tử a[0].

Tiếp theo, chèn các phần tử a[i] vào mảng con đã sắp xếp trước i, với i = 1,2,…, n-1.

Ví dụ 3.4:



Hình 3.3: *Sắp xếp chèn* [8]

Cài đặt thuật toán:

```
void InsertSort(int a[], int n)
{
  for(int i=1;i<n;i++)
  {
    int x=a[i];
    int j=i; //Tìm j: vị trí cần chèn
    while(j>0 && x<a[j-1])
    {
      a[j]=a[j-1];
      j--;
    }
    a[j]=x; //chèn x vào vị trí j
  }
}
```

Thuật toán Insertion Sort có độ phức tạp là $\mathbf{O(n^2)}$.

3.3.4. Sắp xếp nhanh (Quick sort)

Thuật toán Quicksort còn gọi là sắp xếp theo kiểu phân đoạn (partition sort).

Ý tưởng:

Chọn một phần tử bất kỳ làm chốt (pivot).

Đưa các phần tử bé hơn hoặc bằng pivot về bên trái của pivot, đưa các phần tở lớn hơn pivot về bên phải của pivot.

Các phần tử bé hơn hoặc bằng pivot sẽ tạo thành một mảng con thứ nhất, các phần tử lớn hơn pivot sẽ tạo thành mảng con thứ hai.

Thực hiện tương tự cho hai dãy con vừa tạo ra cho đến khi dãy được sắp xếp hoàn toàn.

Như vậy, có thể mô tả thuật toán Quicksort như sau:

Để sắp xếp mảng a[left]..a[right], tiến hành các bước:

- Xác định chốt (pivot).
- Phân hoạch mảng đã cho thành hai mảng con a[left]..a[k-1] và a[k]..a[right].
- Sắp xếp mảng a[left]..a[k-1] (đệ quy).
- Sắp xếp mảng a[k]..a[right] (đệ quy).

Quá trình đệ quy sẽ dừng khi không còn tìm thấy chốt.

Cài đặt thuật toán

```
void QuickSort(int left,int right)
{
  int pivot = A[(left+right)/2];
  int i = left, j = right;
  while(i<j)
  {
    while(A[i] < pivot) i++;
    while(A[j] > pivot) j--;
    if(i<=j)
    {
      swap(A[i],A[j]);
      i++; j--;
    }
  }
  if(left<j) QuickSort(left,j,A);
  if(right>i) QuickSort(i,right,A);
}
```

Thuật toán QuickSort có độ phức tạp là O(nlogn).

3.3.5. Sắp xếp trộn (Mergesort)

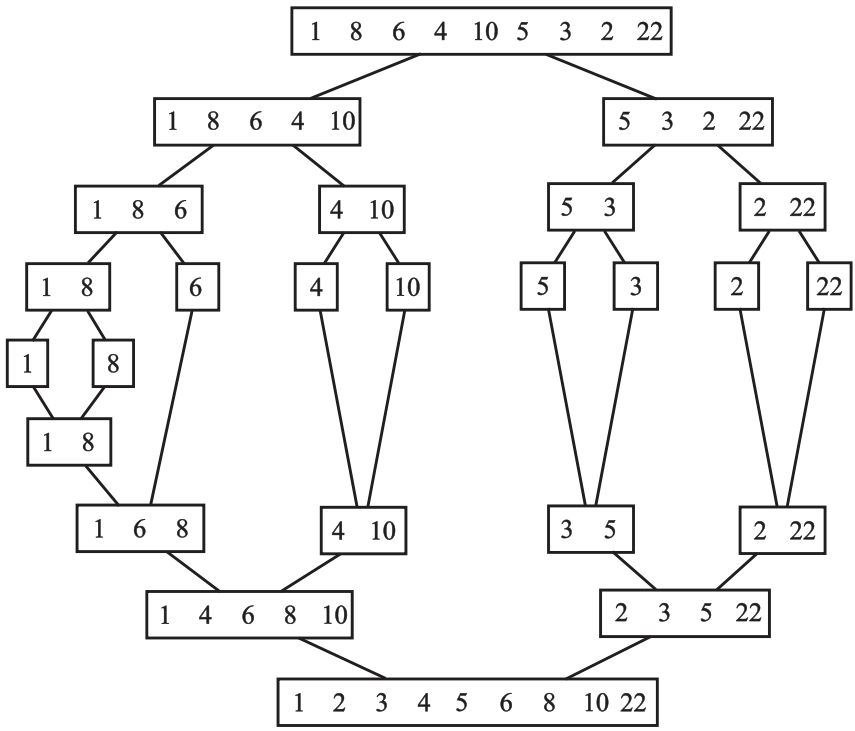
Ý tưởng:

Thuật toán Mergesort tạo ra các phân đoạn đơn giản nhất có thể và tập trung vào việc ***trộn các nửa đoạn*** đã được sắp xếp thành một mảng đã được sắp xếp.

Đây là một trong những thuật toán sắp xếp đầu tiên được sử dụng trên máy tính được phát triển bởi John von Neumann.

```
mergesort(data[ ])
    if (data[ ] có ít nhất 2 phần tử)
        mergesort(nửa trái của data[ ]);
        mergesort(nửa phải của data[ ]);
        merge(trộn hai nửa thành mảng đã sắp xếp);
```

Ví dụ 3.5: Sắp xếp mảng [1 8 6 4 10 5 3 2 22] tăng dần.



Hình 3.4: *Sắp xếp trộn*

Cài đặt thuật toán:

```
//Gộp hai mảng con A[left...m] và A[m+1..right]
void merge(int left, int m, int right)
{
  int i, j, k;
  int n1 = m - left + 1;
  int n2 =  right - m;
  //Tạo các mảng tạm
  int L[] = new int[n1];
  int R[] = new int[n2];
  //Copy dữ liệu sang các mảng tạm
  for (i = 0; i < n1; i++)
    L[i] = A[left + i];
  for (j = 0; j < n2; j++)
    R[j] = A[m + 1 + j];
  //Gộp hai mảng tạm vừa rồi vào mảng A
  i = 0; //chỉ số bắt đầu của mảng con đầu tiên
  j = 0; //chỉ số bắt đầu của mảng con thứ hai
  k = left;//chỉ số bắt đầu của mảng lưu kết quả
  while (i < n1 && j < n2)
  {
    if (L[i] <= R[j])
      A[k] = L[i++];
    else
      A[k] = R[j++];
      k++;
   }
    //Copy phần còn lại của mảng L vào A
    while (i < n1)
      A[k++] = L[i++];
    //Copy phần còn lại của mảng R  vào A
    while (j < n2)
```

```
    A[k++] = R[j++];
}
void mergeSort(int left, int right)
{
  if (left < right)
  {
    int mid = (left+right)/2;
    mergeSort(left,mid);
    mergeSort(mid+1,right);
    merge(left, mid, right);
   }
}
```

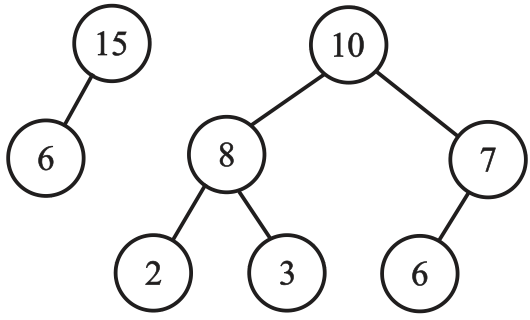
Thuật toán Merge Sort có độ phức tạp là O(nlogn).

3.3.6. Sắp xếp vun đống (Heap sort)

Ý tưởng

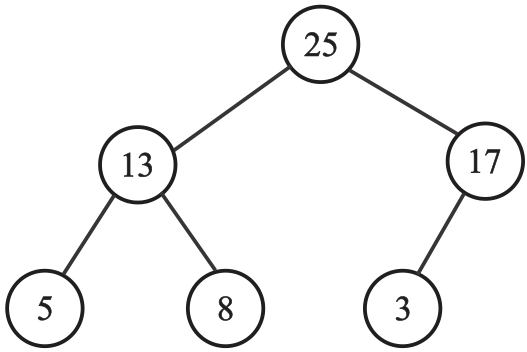
Dựa vào cấu trúc dữ liệu Heap. Heap là một cây nhị phân đặc biệt, có hai thuộc tính:

- Giá trị của mỗi nút lớn hơn hoặc bằng các giá trị được lưu trữ trong mỗi nút con của nó.
- Nếu chiều cao của cây nhị phân là ***h***, cây nhị phân luôn hoàn chỉnh ở cấp ***d***, với d = 1, 2, ..., h-1 và các nút lá ở cấp cuối cùng đều ở vị trí nút con bên trái (có thể không có nút lá bên phải).



Hình 3.5: *Cấu trúc cây nhị phân Heap*

- Ta có thể biểu diễn Heap bằng mảng theo thứ tự mức, đi từ trái sang phải.



Hình 3.6: *Mảng tương ứng với Heap ở trên là [25, 13, 17, 5, 8, 3]*

- Giả sử nút gốc của cây là a[0] và ***i*** là chỉ số của nút hiện tại, các chỉ số của nút cha, nút con trái và nút con phải của nó có thể được tính:

```
PARENT (i)
    return floor((i-1)/2);
LEFT (i)
    return 2*i+1;
RIGHT(i)
    return 2*i + 2;
```

Thuật toán Heap sort bao gồm 2 bước:

Bước 1: Chuyển đổi mảng thành heap

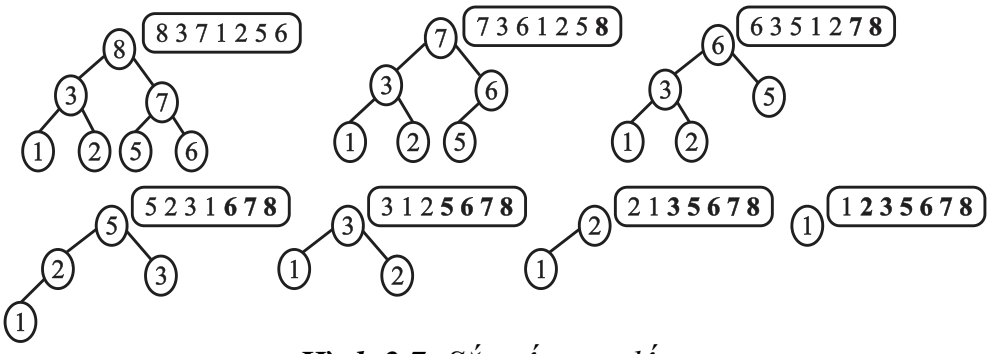
- Giả sử mảng a cần sắp xếp có n phần tử: a [0], a [1], ..., a [n-1].
- Đầu tiên, heap chỉ có một phần tử: a [0].
- Sau đó, ta chèn các phần tử a[i] vào heap sao cho thuộc tính heap vẫn được duy trì, với i ∈ [1, n-1].

Bước 2: Chuyển đổi heap thành mảng đã được sắp xếp

for k = 1 to n-1 do

- Xóa và đặt gốc (a[0]) đến vị trí **n-k**;
- Cập nhật lại các phần tử để duy trì thuộc tính heap;

Ví dụ 3.6: Sắp xếp dãy 2 3 7 1 8 5 6 theo thuật toán Heap sort.



Hình 3.7: *Sắp xếp vun đống*

Cài đặt thuật toán:

```
// Vun đống cây con với nút gốc có chỉ số i trong
// mảng A[]. n là kích thước của heap
void heapify(int n, int i)
{
  int largest = i;//khởi tạo largest như nút gốc
  int l = 2*i + 1;
  int r = 2*i + 2;
  //Nếu con trái lớn hơn gốc
  if (l < n && A[l] > A[largest])
    largest = l;
  //Nếu con phải lớn hơn gốc
  if (r < n && A[r] > A[largest])
    largest = r;
```

```
  //Nếu largest không phải nút gốc
  if (largest != i)
  {
    int temp = A[i];
    A[i] = A[largest];
    A[largest] = temp;
    //vun đống đệ quy
    heapify(n, largest);
  }
}
void heapSort(int n)
{
  //Tạo heap
  for (int i = n / 2 - 1; i >= 0; i--)
    heapify(n, i);
  //Lần lượt trích xuất một phần tử từ heap
  for (int i=n-1; i>0; i--)
  {
    swap(A[0],A[i]);
    heapify(i, 0);
      }
  }
```

Thuật toán Heap Sort có độ phức tạp là **O(nlogn)**.

3.4. THUẬT TOÁN TÌM KIẾM

3.4.1. Tìm kiếm tuần tự

Đây là một kỹ thuật tìm kiếm cổ điển. Thuật toán tiến hành so sánh x lần lượt với phần tử thứ nhất, thứ hai,... của mảng a cho đến khi gặp được phần tử có khóa cần tìm, hoặc duyệt tìm hết mảng mà không thấy x.

```
int Search(int x, int n, int a[])
{
  for(int i=0;i<n;i++)
    if(a[i] == x) return i;//vị trí tìm thấy
  return -1; //không tìm thấy
}
```

3.4.2. Tìm kiếm nhị phân

Áp dụng đối với mảng đã được sắp xếp theo thứ tự tăng hoặc giảm dần. Giả sử ***left, right*** là chỉ số đầu và cuối của mảng a[ ].

```
int BinSearch(int x,int left,int right)
{
if(left>right)
  return -1; //không tìm thấy
else {
      int mid = (left + right)/2;
      if(x==a[mid]) return mid; //tìm thấy
      else if(x<a[mid])
          return BinSearch(x,left,mid-1);
        else
          return BinSearch(x,mid+1,right);
      }
  }
```

BÀI TẬP CHƯƠNG 3

Bài 3.1: (ARRMAX) Viết chương trình tìm giá trị lớn nhất của một mảng số nguyên gồm N phần tử.

Input:

- Dòng đầu chứa số nguyên: N
- Dòng tiếp theo chứa các phần tử a1,…,an

Output: M là số nguyên lớn nhất trong mảng

Ví dụ:

INPUT	OUTPUT
5 1 2 30 4 5	30

Bài 3.2: (MERGEARR) Cho 2 mảng số nguyên đã được sắp xếp theo thứ tự tăng dần. Hãy trộn 2 mảng đó lại thành mảng C sao cho mảng C vẫn có thứ tự tăng dần.

Input:

- Dòng đầu chứa 2 số nguyên: M, N ≤ 500,000
- Dòng tiếp theo chứa các phần tử $a_1,\ldots,a_M$ tăng dần
- Dòng tiếp theo chứa các phần tử $b_1,\ldots,b_N$ tăng dần ($a_i, b_i \leq 10^6$)

Output:

- Dòng đầu ghi: M+N
- Dòng sau ghi: $C_1, C_2,\ldots, C_{M+N}$ tăng dần

Ví dụ:

INPUT	OUTPUT
5 3 1 3 5 7 9 2 4 6	8 1 2 3 4 5 6 7 9

Gợi ý:

- Dùng 2 chỉ số i,j để duyệt qua các phần tử của 2 mảng A, B và k là chỉ số cho mảng C.
- Trong khi (i<m) và (j<n) thì:

  */*Tức trong khi cả 2 dãy A, B đều chưa duyệt hết */*

+ Nếu A[i]>B[j] thì: C[k]=A[i]; i=i+1;
+ Ngược lại: C[k]=B[j]; j=j+1;

- Nếu dãy nào hết trước, đem phần còn lại của dãy kia bổ sung vào cuối dãy C.

Bài 3.3: (DAYTANG) Viết chương trình nhập vào một dãy số nguyên a_1, a_2, ..., a_n. Tìm độ dài dãy con tăng dần dài nhất. (Dãy con là dãy các phần tử liên tiếp nhau trong mảng).

Input:

- Dòng đầu chứa số nguyên: N ($N \leq 10^6$)
- Dòng tiếp theo chứa các phần tử $a_1,\ldots,a_N$ ($a_i \leq 10^{12}$)

Output: S là độ dài dãy con tăng dần dài nhất

Ví dụ:

INPUT	OUTPUT
10 9 12 **2 2 3 7 8** 5 10 15	5

Bài 3.4: (SETNUM) Cho *n* số nguyên dương $a_1, a_2,..., a_n$ ($1 \leq n$, $a_i \leq 10^6$). Hãy xác định số lượng các phần tử khác nhau trong dãy số đã cho.

Input:

- Dòng đầu chứa số nguyên dương n.
- Dòng sau chứa n số nguyên a_1, a_2,..., a_n.

Output: một số nguyên S là số các phần tử khác nhau trong dãy số đã cho.

Ví dụ:

INPUT	OUTPUT
5 4 1 4 2 1	3

Bài 3.5: (SUPASCEN) Dãy số nguyên dương được gọi là dãy ***siêu tăng*** nếu kể từ phần tử thứ hai trở đi, mỗi phần tử không nhỏ hơn tổng của các phần tử đứng trước đó.

Ví dụ: a = {1, 5, 6, 15, 30} là dãy siêu tăng; b = {1, 5, 9, 10, 21} không phải là dãy siêu tăng.

Cho dãy số nguyên dương {a} gồm n phần tử ($n \leq 50$, $a < 10^{18}$). Hãy kiểm tra tính siêu tăng của dãy a.

Input:

- Dòng đầu chứa số nguyên dương n.
- Dòng sau chứa n số nguyên a_1, a_2,..., a_n.

Output: TRUE/FALSE ứng với dãy siêu tăng.

Ví dụ:

INPUT	**OUTPUT**
5 1 5 6 15 30	TRUE

INPUT	**OUTPUT**
5 1 5 9 10 21	FALSE

Bài 3.6: (SYMABILITY) Dãy số nguyên được gọi là dãy ***khả đối xứng*** nếu có thể sắp xếp lại thành một dãy đối xứng. Cho dãy số nguyên gồm n phần tử (n>0). Hãy kiểm tra dãy có phải là khả đối xứng?

Input:

- Dòng đầu: n ($n \leq 10^6$)
- Dòng sau ghi n số nguyên a_1, a_2,...,a_n ($a_i \leq 10^{18}$).

Output: TRUE/FALSE tương ứng với khả đối xứng.

Ví dụ:

INPUT	OUTPUT
5 1 5 6 5 6	TRUE

Bài 3.7: (PASCAL) Viết chương trình in ra màn hình tam giác Pascal.

Input: N ≤ 50

Output: Ma trận tam giác Pascal

Ví dụ:

INPUT	OUTPUT
4	1 1 1 1 2 1 1 3 3 1 1 4 6 4 1

Ý tưởng:

Tam giác Pascal được tạo ra theo quy luật sau:

+ Mỗi dòng đều bắt đầu và kết thúc bởi số 1.
+ Phần tử thứ j ở dòng i nhận được bằng cách cộng 2 phần tử thứ j-1 và j ở dòng thứ i-1: **a(i,j) = a(i-1,j-1) + a(i-1,j)**.

Bài 3.8: (FSEQ) (Olympic Tin học 2013, khối Cao đẳng)

Một dãy số gồm n số nguyên $f_1, f_2, \ldots, f_n$ được gọi là dãy có tính chất của dãy số Fibonacci nếu $n \geq 3$ và với mọi số f_i ($i \geq 3$) thỏa mãn điều kiện $f_i = f_{i-1} + f_{i-2}$.

Ví dụ: dãy 1, 1, 2, 3, 5, 8 là dãy số có tính chất của dãy số Fibonacci; còn dãy 3, 3, 6, 9, 14, 23 không phải là dãy số có tính chất của dãy số Fibonacci.

Yêu cầu: Cho dãy số nguyên a_1, a_2,…, a_n. Hãy tìm một dãy con liên tiếp gồm nhiều phần tử nhất của dãy số đã cho có tính chất của dãy số Fibonacci.

Input*:*

- Dòng đầu chứa số nguyên n ($3 \le n \le 30000$).
- Dòng thứ hai chứa n số nguyên a_1, a_2,…,a_n ($|a_i| \le 10^9$).

Output*:* Một số nguyên là số lượng phần tử của dãy con tìm được, ghi -1 nếu không tồn tại dãy con liên tiếp nào của dãy có tính chất của dãy số Fibonacci.

Ví dụ:

INPUT	OUTPUT
7 1 **3 3 6 9** 14 23	4

Bài tập 3.9: (DEL) (Olympic Tin học 2012, khối Không Chuyên)

Cho dãy số nguyên không âm $a_1, a_2, .., a_n$. Người ta tiến hành chọn ra 2 chỉ số i, j sao cho $1 \le i < j \le n$ và xóa khỏi dãy 2 số a_i, a_j để tổng giá trị các số còn lại trong dãy là số chẵn.

Yêu cầu: Cho dãy số $a_1, a_2, .., a_n$. Hãy đếm số lượng cách chọn 2 chỉ số i, j thỏa mãn. Hai cách chọn khác nhau nếu tồn tại một chỉ số khác nhau.

Input*:*

- Dòng 1: chứa số nguyên n ($n \le 10^6$)
- Dòng 2: chứa n số nguyên không âm $a_1, a_2, .., a_n$ ($a_i \le 10^9$)

Output*:* Một số nguyên là số cách chọn 2 chỉ số thỏa mãn.

Ví dụ:

INPUT	OUTPUT
5 1 2 3 4 5	6

Gợi ý:

Gọi S là số cách chọn.

- Nếu tất cả đều là số chẵn: $S = C^2_n$
- Nếu tất cả đều là lẻ:
  - Nếu n lẻ: S = 0
  - Nếu n chẵn: $S = C^2_n$
- Gọi A: số lượng số chẵn, B: số lượng số lẻ
  - Nếu B lẻ: lấy 1 chẵn × 1 lẻ ⇨ S = A × B
  - Nếu B chẵn: $S = C^2_A \times C^2_B$

Bài 3.10: Viết chương trình tính tổng của 2 đa thức h(x) = f(x) + g(x). Trong đó, mỗi đa thức có dạng: $a_0 + a_1x + a_2x^2 + ... + a_nx^n$.

Gợi ý:

Dùng các mảng A, B, C để lưu trữ các hệ số a_i của các đa thức f(x), g(x) và h(x).

Chương 4
DANH SÁCH LIÊN KẾT

Tóm tắt chương

- Danh sách liên kết đơn
- Danh sách liên kết kép
- Danh sách liên kết nối vòng

4.1. GIỚI THIỆU

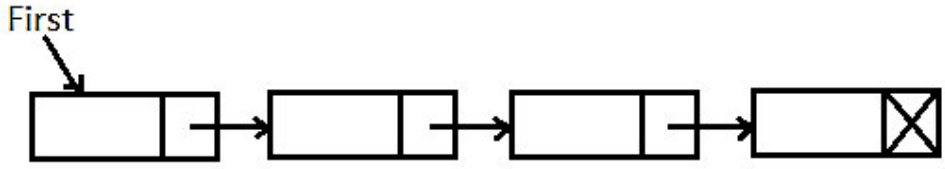
Danh sách liên kết (linked list) là danh sách có các phần tử được nối kết với nhau nhờ vào vùng liên kết của chúng (quản lý theo địa chỉ).

Danh sách liên kết phù hợp với các phép bổ sung, loại bỏ hoặc ghép nhiều danh sách, các thao tác này không phù hợp với danh sách đặc.

Quản lý theo danh sách liên kết phù hợp với các loại danh sách có số phần tử luôn biến động (các thao tác bổ sung và loại bỏ được thực hiện liên tục).

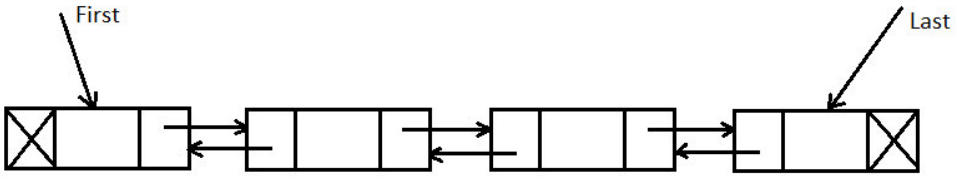
Các phần tử trong danh sách liên kết có thể lưu trữ ở những vùng không liền kề nhau trong bộ nhớ, chúng kết nối với nhau nhờ vào trường liên kết.

Danh sách liên kết đơn: mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách:



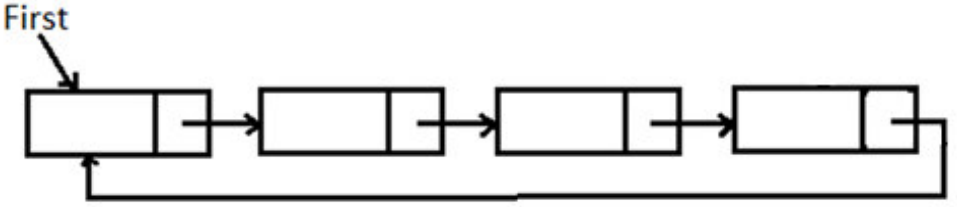
Hình 4.1: *Danh sách liên kết đơn*

Danh sách liên kết kép: mỗi phần tử liên kết với các phần tử đứng trước và sau nó trong danh sách:

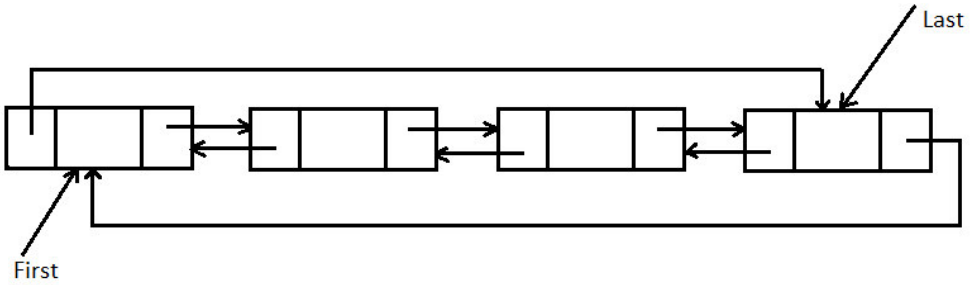


Hình 4.2: *Danh sách liên kết kép*

Danh sách liên kết vòng: phần tử cuối danh sách liên kết với phần tử đầu danh sách:



Hình 4.3: *Danh sách liên kết đơn nối vòng*



Hình 4.4: *Danh sách liên kết kép nối vòng*

Hình thức liên kết này cho phép các thao tác chèn, xóa trên danh sách được thực hiện dễ dàng, phản ánh được bản chất linh động của danh sách.

4.2. DANH SÁCH LIÊN KẾT ĐƠN

4.2.1. Định nghĩa và khai báo

Danh sách liên kết đơn là danh sách liên kết mà mỗi phần tử (Node) của nó chỉ có một trường liên kết chứa địa chỉ của phần tử (Node) đứng ngay sau nó.

Mỗi phần tử của danh sách liên kết đơn là một cấu trúc chứa 2 thông tin (Hình 4.5):

- Data: lưu trữ các thông tin của phần tử đó.
- Link: lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu nó là phần tử cuối danh sách.

Hình 4.5: *Cấu trúc của một nút*

Để định nghĩa một danh sách liên kết trước hết phải định nghĩa kiểu của mỗi **nút** trong danh sách.

```
struct < NODE>
{
<Type> <Data>;  // chứa dữ liệu
NODE *Next;     // chứa địa chỉ nút tiếp theo
};
```

Sau đó có thể khai báo một danh sách liên kết như sau:

< NODE> *First;

First là con trỏ trỏ đến nút đầu tiên trong danh sách, dựa vào trường **Next** của nút này để biết được địa chỉ của các nút tiếp theo trong danh sách.

Ví dụ 4.1: Tổ chức một danh sách liên kết đơn chứa các số nguyên:

```
struct NODE
```

```
{
int x;              //chứa số nguyên
NODE *Next;
};
NODE *First;        //Danh sách được trỏ bởi
                    //con trỏ đầu First
```

4.2.2. Các thao tác trên danh sách liên kết đơn

4.2.2.1. Khởi tạo danh sách

```
class SimpleLinkedList
{
  struct NODE
  {
    int Data;
    NODE *Next;
  };
  NODE *First;
public:
  SimpleLinkedList() //Khoi tao danh sach
  {
    First = NULL;
  }
};
```

4.2.2.2. Bổ sung một phần tử vào danh sách

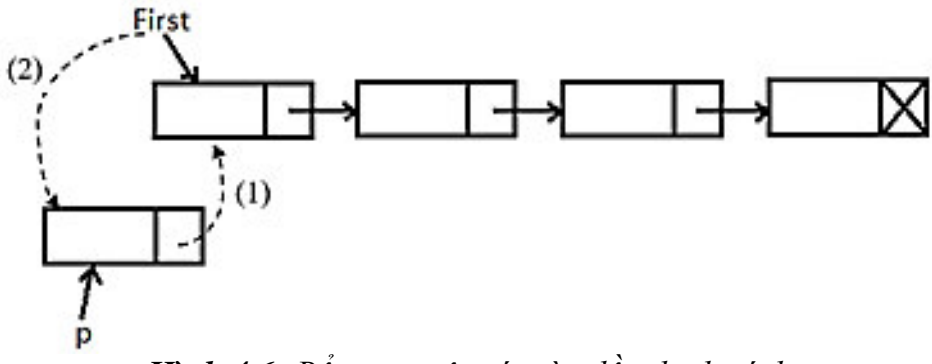
Trường hợp 1: Bổ sung vào đầu danh sách

Bước 1: Tạo ra nút mới

```
<Trỏ_Nút> p=new(NODE);
p->Data = <Giá trị>;
```

Bước 2: Bổ sung vào đầu danh sách

```
p->Next=First;        (1)
First=p;              (2)
```



Hình 4.6: *Bổ sung một nút vào đầu danh sách*

Hàm cài đặt:

```
void SimpleLinkedList::FirstInsert(int x)
{
  //B1: tạo nút mới p chứa x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chèn p vào đầu danh sách
  p->Next = First;     //1
  First = p;           //2
}
```

Trường hợp 2: Bổ sung vào cuối danh sách

Bước 1: Tạo ra nút mới

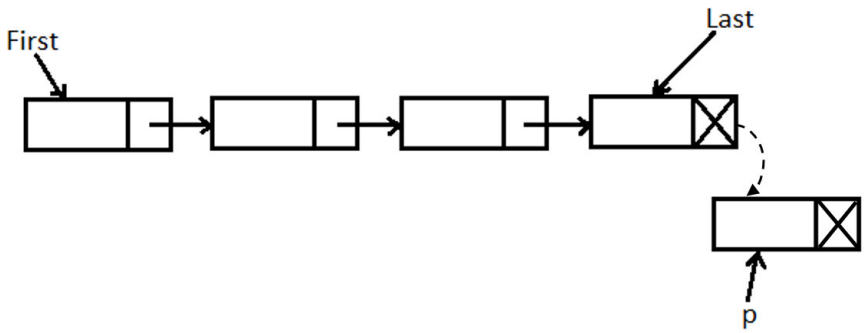
```
<Trỏ_Nút> p=new(NODE);
p->Data = <Giá trị>;
p->Next = NULL;
```

Bước 2: Tìm đến nút cuối danh sách Last

```
Last=First;
while(Last->Next!=NULL) Last=Last->Next;
```

Bước 3: Nối p vào sau Last

```
Last->Next = p;
```



Hình 4.7: *Bổ sung một nút vào cuối danh sách*

Hàm cài đặt:

```
void SimpleLinkedList::LastInsert(int x)
{
  //B1: tạo nút mới p chứa x
  NODE *p = new(NODE);
  p->Data = x;
  p->Next = NULL;
  if(First==NULL) //Nếu danh sách rỗng
    First = p;
  else {
      //B2: Tìm đến nút cuối danh sách: Last
      NODE *Last = First;
      while(Last->Next) Last = Last->Next;
      //B3: chèn p vào sau Last
      Last->Next = p;
  }
}
```

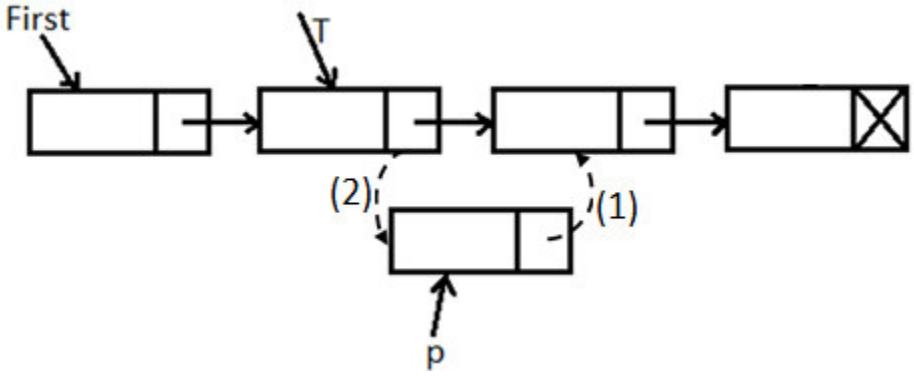
Trường hợp 3: Bổ sung vào sau nút được trỏ bởi T

Bước 1: Tạo ra nút mới

```
<Trỏ_Nút> p=new(NODE);
p->Data = <Giá trị>;
```

Bước 2: Chèn p vào sau nút T

```
p->Next = T->Next;          (1)
T->Next = p;                (2)
```



Hình 4.8: *Bổ sung nút p vào sau nút T*

Hàm cài đặt:

```
void SimpleLinkedList::Insert(int x,NODE *T)
{
  //B1: tạo nút mới p chứa x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chèn p vào sau T
  p->Next = T->Next;   //1
  T->Next = p;         //2
}
```

4.2.2.3. Xóa một nút khỏi danh sách

Lưu ý rằng khi cấp phát bộ nhớ, chúng ta đã dùng phương thức ***new***. Vì vậy khi giải phóng bộ nhớ ta phải dùng phương thức ***delete***.

Trường hợp 1: Xóa nút đầu tiên của danh sách

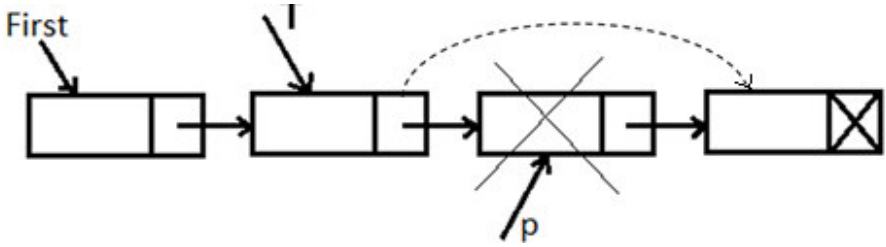
```
void SimpleLinkedList::FirstDel()
{
  //Định vị đến nút cần xóa
  NODE *p = First;
  //Dịch chuyển First qua nút kế tiếp
  First = p->Next;
  //Xóa nút p
  delete p;
}
```

Trường hợp 2: Xóa nút cuối của danh sách

```
void SimpleLinkedList::LastDel()
{
  NODE *p = First, *T;
  if(First->Next==NULL)  //Danh sách chỉ
                         //có 1 nút
  First = NULL;
  else {
    //B1: định vị p đến nút cuối danh sách
    while(p->Next)
    {
      T = p; //T trỏ đến nút đứng trước p
      p = p->Next;
    }
    //B2: Chuẩn bị xóa
```

```
    T->Next = NULL;
  }
  //B3: xóa p
  delete p;
}
```

Trường hợp 3: Xóa nút đứng sau nút được trỏ bởi T



Hình 4.9: *Xóa nút đứng sau nút được trỏ bởi T*

```
void SimpleLinkedList::Delete(NODE *T)
{
  //B1: định vị đến nút cần xóa
  NODE *p = T->Next;
  //B2: chuẩn bị xóa
  T->Next = p->Next;
  //B3: xóa p
  delete p;
}
```

4.2.2.4. Duyệt danh sách

Duyệt qua và xử lý từng nút trong danh sách.

```
void SimpleLinkedList::View()
{
  NODE *p = First;
  while(p)
```

```
  {
    cout<<p->Data<<" ";
    p = p->Next;
  }
}
```

4.2.2.5. Ví dụ ứng dụng

Dùng danh sách liên kết đơn để biểu diễn đa thức $\mathbf{P_n(x) = a_n x^n + a_{n-1}x^{n-1} +...+ a_0}$. Trong đó, mỗi số hạng của đa thức được xác định bởi 2 thành phần: hệ số $\mathbf{a_i}$ và số mũ **i**. Ta có thể xây dựng cấu trúc dữ liệu để lưu trữ đa thức như sau:

```
struct Node
{
  float HeSo;
  int SoMu;
  Node *Next;
};
```

Viết các hàm thực hiện các công việc sau:

1. Nhập vào một đa thức.
2. Cộng hai đa thức P1, P2 thành đa thức P3.
3. Tính giá trị của đa thức với giá trị X cho trước.

```
#include <iostream>
#include <math.h>
using namespace std;
class Polynomial
{
  struct Node
  {
    int Somu;
    float Heso;
```

```
  Node *Next;
};
Node *First;
public:
Polynomial()
{
  First = NULL;
}
void First_Insert(float hs,int sm)
{
  Node *p = new(Node);
  p->Somu = sm;
  p->Heso = hs;
  p->Next = First;
  First = p;
}
void Last_Insert(float hs,int sm)
{
  Node *p = new(Node);
  p->Somu = sm;
  p->Heso = hs;
  p->Next = NULL;
  if(First==NULL) First = p;
  else
  {
    Node *q = First;
    while(q->Next) q = q->Next;
    q->Next = p;
  }
}
void Input()
{
  int n;
```

```
  cout<<"Bac da thuc: "; cin>>n;
  for(int i=0;i<=n;i++)
  {
    int sm;
    float hs;
    cout<<endl<<"He so a"<<i<<"= ";
    cin>>hs;
    First_Insert(hs,i);
  }
}
void View()
{
  Node *p = First;
  cout<<endl<<"P(x) = ";
  while(p)
  {
    if(p->Heso!=0)
    {
      if(p->Somu==0) cout<<p->Heso;
      else if(p->Somu==1)
          cout<<p->Heso<<"x + ";
             else if(p->Next==NULL)
          cout<<p->Heso<<"x^"<<p->Somu;
              else
    cout<<p->Heso<<".x^"<<p->Somu<<" + ";
    }
    p = p->Next;
  }
}
float Value(float x)
{
  float s = 0;
  Node *p = First;
  while(p)
```

```
  {
    s += p->Heso*pow(x,p->Somu);
    p = p->Next;
  }
  return s;
}
friend Polynomial operator +(Polynomial
P1, Polynomial P2)
{
  Polynomial P3;
  float hs;
  int sm;
  Node *p = P1.First;
  Node *q = P2.First;
  while(p && q)//Cả 2 đa thức đều khác rỗng
  {
    if(p->Somu==q->Somu)
    {
      hs = p->Heso + q->Heso;
      sm = p->Somu;
      p = p->Next;
      q = q->Next;
    }
    else if(p->Somu>q->Somu)
      {
       hs = p->Heso;
       sm = p->Somu;
       p = p->Next;
      }
      else
      {
         hs = q->Heso;
         sm = q->Somu;
         q = q->Next;
```

```
            }
          P3.Last_Insert(hs,sm);
        }
        if(p==NULL) //P1 hết trước
        {
          while(q)
          {
            hs = q->Heso;
            sm = q->Somu;
            q = q->Next;
            P3.Last_Insert(hs,sm);
          }
        }
        else //P2 hết trước
        {
          while(p)
          {
            hs = p->Heso;
            sm = p->Somu;
            p = p->Next;
            P3.Last_Insert(hs,sm);
          }
        }
        return P3;
      }
};
int main()
{
    Polynomial P1,P2,P3;
    cout<<"Nhap da thuc P1:"<<endl;
    P1.Input();
    P1.View();
    cout<<endl<<"Nhap da thuc P2:"<<endl;
    P2.Input();
```

```
    P2.View();
    P3 = P1 + P2;
    P3.View();
    float x;
    cout<<endl<<"Nhap x = "; cin>>x;
    cout<<P3.Value(x);
    return 0;
}
```

4.2.3. Danh sách liên kết đơn nối vòng

4.2.3.1. Định nghĩa

Danh sách liên kết đơn nối vòng là danh sách liên kết đơn có phần tử cuối cùng của danh sách trỏ vào (liên kết với) phần tử đầu tiên của danh sách (Hình 4.3).

4.2.3.2. Các thao tác trên danh sách liên kết đơn nối vòng

a) Khởi tạo danh sách liên kết đơn nối vòng

Tương tự như khởi tạo danh sách liên kết đơn.

```
class CircleLinkedList
{
struct NODE
{
  int Data;
  NODE *Next;
};
NODE *First;
public:
CircleLinkedList() //Khởi tạo danh sách
{
  First = NULL;
  }
};
```

b) Duyệt danh sách

Danh sách chỉ được duyệt khi khác rỗng và việc duyệt được thực hiện tuần tự từ nút đầu đến nút cuối cùng.

```
void CircleLinkedList::View()
{
if(First)
  {
  NODE *p = First;
  do
  {
    cout<<p->Data<<" ";
    p = p->Next;
  }
  while(p!=First);
  }
}
```

c) Tìm một phần tử trên danh sách liên kết đơn nối vòng

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

```
NODE* CircleLinkedList::Search(int x)
{
  if(First)
  {
    NODE *p = First;
    do
    {
      if(p->Data!=x);
```

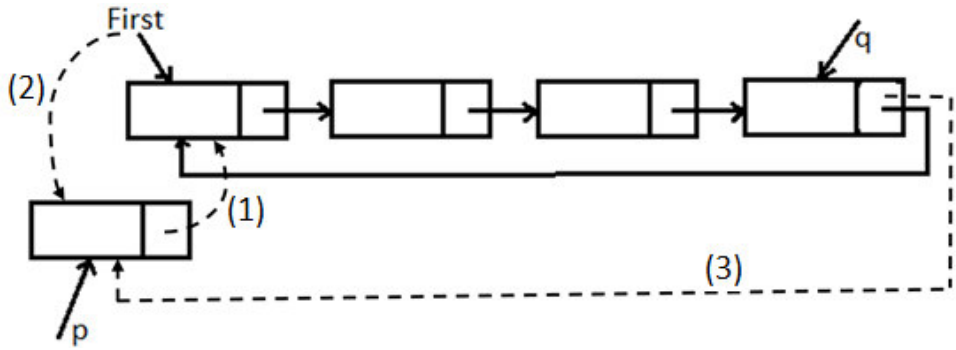
```
        p = p->Next;
      }
      while(p->Data!=x && p!=First);
      if(p!=First) return p;
    }
    return NULL;
}
```

d) Thêm một phần tử vào danh sách

Trường hợp 1: Thêm nút mới vào đầu danh sách

Danh sách liên kết đơn nối vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kỳ trên danh sách xem như phần tử đầu danh sách để thực hiện thêm phần tử vào đầu danh sách.

Thuật toán:



Hình 4.10: *Bổ sung nút mới vào đầu danh sách liên kết đơn nối vòng*

Bước 1: Tạo nút mới được trỏ bởi p

Bước 2: Nếu danh sách rỗng, thực hiện

```
First=p;
First->Next = First;
```

Ngược lại: Chuyển qua bước 3.

Bước 3: Tìm đến nút cuối cùng của danh sách: q

```
NODE *q = First;
While (q->Next !=First) q = q->Next;
```

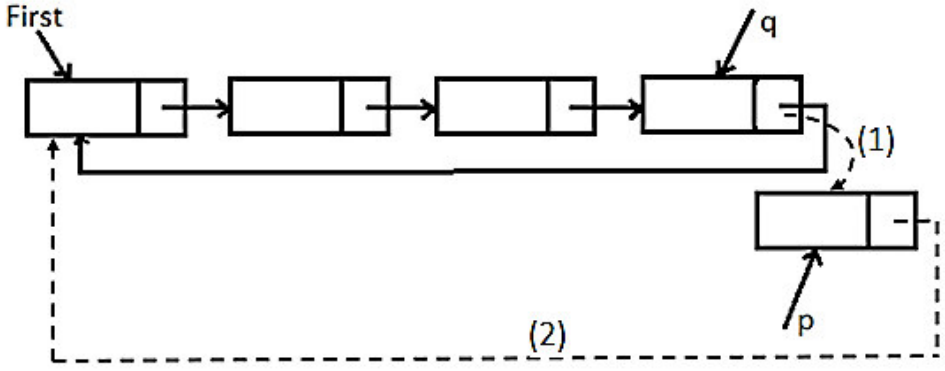
Bước 4: Chèn p vào đầu danh sách

```
p->Next = First;      (1)
First = p;            (2)
q->Next = First;      (3)
```

Hàm cài đặt:

```
void CircleLinkedList::FirstInsert(int x)
{
  //B1: tạo nút mới p chứa x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chèn
  if(First==NULL)
  {
    First = p;              //1
    First->Next = First;    //2
  }
  else
  {
  //B3: Tìm đến nút cuối danh sách q
  NODE *q = First;
  while (q->Next != First) q = q->Next;
  //B4: Chen p vao dau danh sach
  p->Next = First;
  First = p;
  q->Next = First; //nối vòng
  }
}
```

Trường hợp 2: Thêm nút mới vào cuối danh sách



Hình 4.11: *Bổ sung nút mới vào cuối danh sách liên kết đơn nối vòng*

Thuật toán:

Bước 1: Tạo nút mới được trỏ bởi p

Bước 2: Nếu danh sách rỗng, thực hiện

```
First = p;
First->Next = First;
```

Ngược lại: Chuyển qua bước 3.

Bước 3: Tìm đến nút cuối cùng của danh sách q

```
Tro q = First;
While (q->Next !=First) q=q->Next;
```

Bước 4: Chèn p vào cuối danh sách

```
q->Next=p;
p->Next = First;
```

Hàm cài đặt:

```
void CircleLinkedList::LastInsert(int x)
{
  //B1: tạo nút mới p chứa x
```

```
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chèn
  if(First==NULL)
  {
    First = p;              //1
    First->Next = First;    //2
  }
  else
  {
  //B3: Tìm đến nút cuối danh sách q
  NODE *q = First;
  while (q->Next != First) q = q->Next;
  //B4: Chèn p vào cuối danh sách
  q->Next = p;
  p->Next = First; //nối vòng
  }
}
```

Trường hợp 3: Thêm một nút vào sau nút được trỏ bởi q (tương tự trường hợp danh sách liên kết đơn bình thường).

e) Xóa một phần tử trong danh sách

Trường hợp 1: Xóa nút đầu danh sách

- Nếu danh sách chỉ có 1 nút: xóa xong, gán First = NULL;
- Ngược lại: sau khi xóa xong, nút đứng sau First sẽ là nút đầu danh sách.

```
void CircleLinkedList::FirstDel()
{
  NODE *p = First;
  if(First->Next==First)  //danh sách chỉ có
                          //1 nút
```

```
  First = NULL;
  else
  {
  //Tìm tới nút cuối danh sách
  NODE *q = First;
  while(q->Next!=First) q = q->Next;
  //cắt nút p ra khỏi danh sách
  First = p->Next; //1
  q->Next = First; //2
  }
  delete p;
}
```

Trường hợp 2: Xóa nút đứng sau nút q

```
void CircleLinkedList::Delete(NODE *q)
{
  //Định vị tới nút cần xóa
  NODE *p = q->Next;
  //tách p ra khỏi danh sách
  q->Next = p->Next;
  //Xóa nút p
  delete p;
}
```

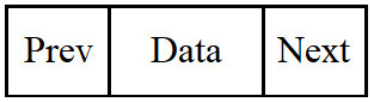
4.3. DANH SÁCH LIÊN KẾT KÉP

4.3.1. Định nghĩa

Danh sách liên kết kép là danh sách liên kết mà mỗi phần tử có hai vùng liên kết: một vùng liên kết trỏ đến phần tử đứng ngay trước nó, gọi là liên kết ngược (Previous) và một vùng liên kết trỏ đến phần tử đứng ngay sau nó gọi là liên kết thuận (Next). Hình 4.4 minh họa cho danh sách liên kết kép.

Một phần tử của danh sách liên kết kép có dạng (Hình 4.12):

- Data: chứa dữ liệu của nút
- Prev, Next: chứa địa chỉ của nút đứng trước và nút đứng sau.



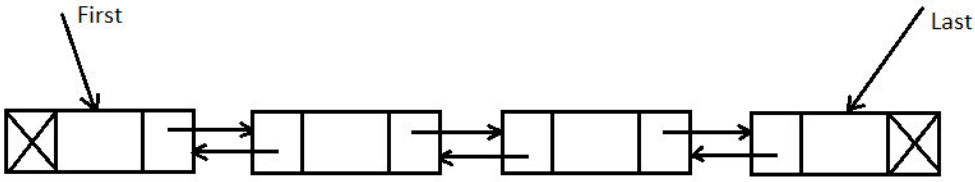
Hình 4.12: *Một nút của danh sách liên kết kép*

Khai báo cấu trúc dữ liệu:

```
struct NODE
{
  <Type> Data;
  NODE *Prev;      //trỏ đến nút đứng trước
  Node *Next;      //trỏ đến nút đứng sau
};
```

4.3.2. Các thao tác trên danh sách liên kết kép

Để quản lý danh sách liên kết kép, ta dùng hai con trỏ First và Last trỏ đến đầu và cuối danh sách (Hình 4.13):



Hình 4.13: *Danh sách liên kết kép*

4.3.2.1. Khởi tạo một danh sách liên kết kép

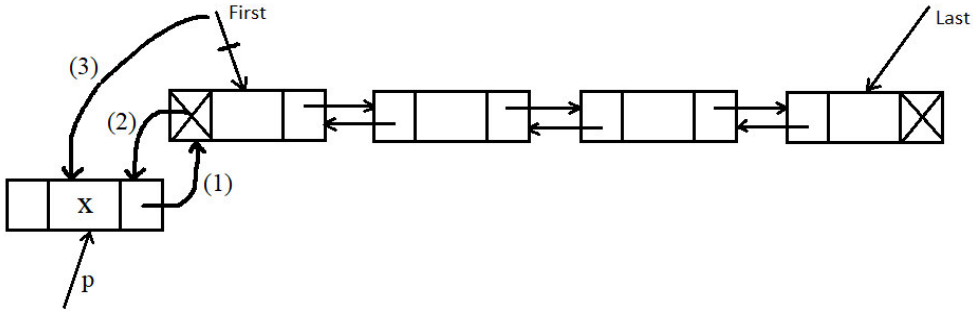
Khi khởi tạo danh sách liên kết kép ta khai báo hai biến con trỏ First và Last trỏ đến nút đầu và cuối danh sách:

First = Last = NULL;

```
class DoubleLink
{
  struct NODE
  {
    int Data;
    NODE *Prev, *Next;
  };
  NODE *First, *Last;
  public:
  DoubleLink()//khởi tạo danh sách
  {
    First = Last = NULL;
  }
};
```

4.3.2.2. Bổ sung một phần tử vào danh sách

Trường hợp 1: Bổ sung vào đầu danh sách

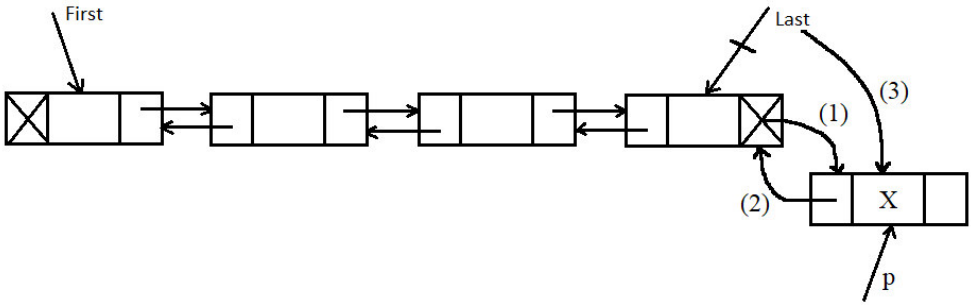


Hình 4.14: *Bổ sung nút mới vào đầu danh sách liên kết kép*

```
void DoubleLink::FirstInsert(int x)
{
  //B1: tạo nút mới p chứa x
  NODE *p = new(NODE);
```

```
  p->Data = x;
  p->Next = p->Prev = NULL;
  //B2: Chèn p vào đầu danh sách
  if(First==NULL) //danh sách rỗng
    First = Last = p;
  else
  {
    p->Next = First;       //1
    First->Prev = p;       //2
    First = p;             //3
  }
}
```

Trường hợp 2: Bổ sung vào cuối danh sách

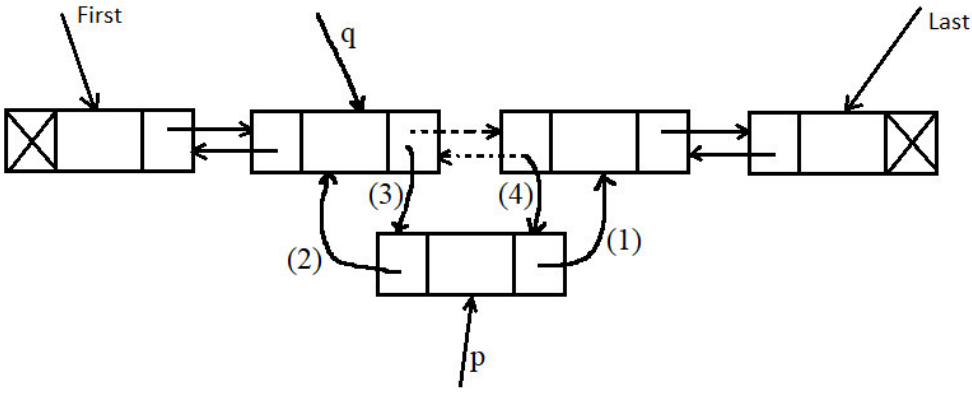


Hình 4.15: *Bổ sung nút mới vào cuối danh sách liên kết kép*

```
void DoubleLink::LastInsert(int x)
{
  //B1: tạo nút mới p chứa x
  NODE *p = new(NODE);
  p->Data = x;
  p->Next = p->Prev = NULL;
  //B2: Chèn p vào đầu danh sách
  if(First==NULL) //danh sách rỗng
```

```
    First = Last = p;
  else
  {
    Last->Next = p;          //1
    p->Prev = Last;          //2
    Last = p;                //3
  }
}
```

Trường hợp 3: Chèn vào sau nút được trỏ bởi q



Hình 4.16: *Bổ sung nút mới vào sau nút được trỏ bởi q*

```
void DoubleLink::Insert(int x,NODE *q)
{
  //B1: tạo nút mới p chứa x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chèn p vào sau q
  p->Next = q->Next;         //1
  p->Prev = q;               //2
  q->Next = p;               //3
  p->Next->Prev = p;         //4
}
```

4.3.2.3. Loại bỏ một phần tử ra khỏi danh sách

Trường hợp 1: Loại bỏ nút đầu danh sách

```
void DoubleLink::FirstDel()
{
  NODE *p = First;
  if(First==Last) //danh sách chỉ có 1 nút
    First = Last = NULL;
  else
  {
    First = p->Next;
    First->Prev = NULL;
  }
  delete p;
}
```

Trường hợp 2: Loại bỏ nút cuối danh sách

```
void DoubleLink::LastDel()
{
  NODE *p = Last;
  if(First==Last)   //danh sách chỉ có 1 nút
    First = Last = NULL;
  else
  {
    Last = p->Prev;
    Last->Next = NULL;
  }
  delete p;
}
```

Trường hợp 3: Loại bỏ nút đứng sau nút được trỏ bởi T

```
void DoubleLink::Delete(NODE *T)
{
  NODE *p = T->Next;
  T->Next = p->Next;
  p->Next->Prev = T;
  delete p;
}
```

BÀI TẬP CHƯƠNG 4

Bài 4.1: Tổ chức danh sách liên kết đơn chứa các số nguyên với nút đầu tiên trỏ bởi con trỏ First. Viết các hàm thực hiện các chức năng sau:

1. **void NHAP()** để nhập vào một danh sách các số nguyên có nút đầu tiên được trỏ bởi con trỏ First.
2. **void LIETKE()** để in ra màn hình giá trị của tất cả các nút trong danh sách được trỏ bởi con trỏ First.
3. **int MAX()** để tìm giá trị lớn nhất của các nút trong danh sách được trỏ bởi con trỏ First.
4. **int DEM()** để đếm xem trong danh sách có bao nhiêu nút.
5. **float TBINH()** để tính giá trị trung bình của các phần tử trong danh sách.
6. **void SAPXEP()** để sắp xếp lại danh sách được trỏ bởi con trỏ First theo thứ tự tăng dần.
7. **bool TANG()** để xác định xem danh sách được trỏ bởi First đã có thứ tự tăng dần hay chưa theo 2 cách: ***Không đệ quy*** và ***đệ quy***.

Bài 4.2: Cho 2 danh sách liên kết đơn biểu diễn cho 2 tập hợp các số nguyên được trỏ bởi L1 và L2.

1. Viết hàm **GIAO(L1,L2);** để hiển thị phần giao của 2 tập hợp L1 $\cap$ L2.

2. Viết hàm **HOP(L1,L2);** để hiển thị phần hợp của 2 tập hợp L1 ∪ L2.
3. Viết hàm **HIEU(L1,L2);** để hiển thị phần hiệu của 2 tập hợp L1 \ L2.

Bài 4.3: Cho 2 danh sách liên kết đơn được trỏ bởi L1 và L2. Viết hàm để nối 2 danh sách đó lại thành một danh sách được trỏ bởi L1.

Bài 4.4: Cho danh sách liên kết chứa các số nguyên. Viết hàm để xóa tất cả các nút chứa các giá trị âm trong danh sách.

Bài 4.5: Tổ chức danh sách liên kết đơn chứa các số nguyên. Viết các hàm:

1. **void Input()** để nhập vào danh sách các số nguyên được trỏ bởi con trỏ First.
2. **int Palin()** để kiểm tra các phần tử trong danh dách được trỏ bởi con trỏ First có đối xứng hay không?
3. **bool isPalin()** để kiểm tra xem với danh sách được trỏ bởi con trỏ First có tồn tại một cách sắp xếp lại các phần tử để cuối cùng nhận được một danh sách đối xứng hay không?
4. **void SAPLAI()** để đưa ra một cách sắp xếp lại các phần tử trong danh sách được trỏ bởi con trỏ First để có được một danh sách đối xứng (giả thiết điều này có thể làm được nhờ hàm **isPalin**).

Bài 4.6: Quản lý khách hàng

Để quản lý khách hàng sử dụng điện thoại SmartPhone, người ta tổ chức lưu trữ danh sách khách hàng dưới dạng danh sách liên kết đơn với 4 trường sau: ***Name*** chứa họ tên khách hàng, 2 con trỏ ***First*** và ***Last*** lần lượt trỏ tới phần tử đầu tiên và cuối cùng của một danh sách liên kết đơn khác dùng để ghi nhận danh sách điện thoại của khách hàng đó đang sở hữu, cuối cùng là trường ***Next*** trỏ đến khách hàng tiếp theo. Mỗi phần tử của danh sách liên kết chứa điện thoại là một cấu trúc gồm 3 trường: ***PhoneName*** (tên điện thoại), ***Num*** (số lượng) và ***Link*** (chứa địa chỉ nút tiếp theo).

CTDL được khai báo như sau:

```
struct Smartphone
{
  string PhoneName;
  int Num;
  Smartphone *Link;
};
typedef Smartphone* TroPhone;

struct KhachHang
{
  string Name;
  TroPhone First,Last;
  KhachHang *Next;
};
typedef KhachHang* TroKH;
```

1. Viết hàm **TroKH KTraName(string Name,TroKH Dau)** để kiểm tra khách hàng có tên Name có trong danh sách Dau hay không? Nếu có trả về địa chỉ nút tìm được, ngược lại trả về NULL.
2. Viết hàm **TroPhone KTraPhoneName(string PhoneName,TroPhone First)** để kiểm tra điện thoại có tên PhoneName có trong danh sách First hay không? Nếu có trả về địa chỉ nút tìm được, ngược lại trả về NULL.
3. Viết hàm **void Insert(string PhoneName,string Name,TroKH &Dau)** để bổ sung khách hàng có tên Name sử dụng điện thoại có nhãn hiệu là PhoneName vào danh sách Dau.
4. Viết hàm **int Dem(TroKH Dau)** để đếm xem có bao nhiêu khách hàng sử dụng từ hai loại điện thoại trở lên.

Bài 4.7: Xử lý văn bản

Người ta tổ chức lưu trữ các dòng của một văn bản trong bộ nhớ dưới dạng một danh sách liên kết kép mà mỗi nút của nó tương ứng với một dòng. Ta dùng 2 con trỏ *First* và *Last* lần lượt trỏ vào dòng đầu tiên và cuối cùng của văn bản.

CTDL được khai báo như sau:

```
struct TextLine
{
  string Line;
  TextLine *Prev, *Next;
};
typedef TextLine* Text;
```

Giả sử văn bản khác rỗng.

1. Viết hàm **void PrevInsert(string st, Text pos, Text &First)** cho phép chèn một dòng có nội dung là *st* vào trước dòng có địa chỉ là *pos* trong văn bản có nút đầu được trỏ bởi con trỏ *First*.
2. Viết hàm **void NextInsert(string st, Text pos, Text &Last)** cho phép chèn một dòng có nội dung là *st* vào sau dòng có địa chỉ là *pos* trong văn bản có nút cuối được trỏ bởi con trỏ *Last*.
3. Viết hàm **void View(Text First)** để hiển thị văn bản được trỏ bởi First ra màn hình.
4. Viết hàm **void DelLine(Text pos, Text &First, Text &Last)** cho phép xóa dòng tại địa chỉ *pos* trong danh sách được trỏ bởi *First và Last*.
5. Ta định nghĩa *khối* là một dãy liên tiếp các dòng trong văn bản. Ký hiệu (fb,lb) là khối được xác định vị trí bởi địa chỉ dòng đầu *fb* và dòng cuối *lb*. Viết hàm **void CopyBlock(Text fb, Text lb,Text dest, Text &First, Text &Last)** cho phép sao chép khối (fb,lb) tới trước dòng được trỏ bởi *dest* trong danh sách được trỏ bởi *First và Last*. Giả sử *dest* không nằm ở trong khối (fb,lb).

6. Viết hàm **void RemoveBlock(Text fb, Text lb, Text &First, Text &Last)** để xóa khối (fb,lb) trong danh sách được trỏ bởi *First và Last*.
7. Viết hàm **void MoveBlock(Text fb, Text lb,Text dest, Text &First, Text &Last)** để di chuyển khối (fb,lb) tới trước dòng được trỏ bởi *dest* trong danh sách được trỏ bởi *First và Last*. Giả sử *dest* không nằm ở trong khối (fb,lb).

Chương 5
DANH SÁCH HẠN CHẾ

Tóm tắt chương

- Giới thiệu hai dạng danh sách đặc biệt: ngăn xếp (stack) và hàng đợi (queue).
- Các thao tác cơ bản trên stack và queue.
- Ứng dụng stack và queue trong xử lý các vấn đề về khoa học máy tính.

5.1. ĐẶT VẤN ĐỀ

Trong các thao tác trên danh sách không phải lúc nào cũng có thể thực hiện hết tất cả các thao tác của danh sách mà đôi khi các thao tác này bị hạn chế trong một số ngữ cảnh nào đó nên các danh sách này được gọi là ***danh sách hạn chế***.

Chương này sẽ xem xét hai loại danh sách hạn chế là **ngăn xếp (stack)** và **hàng đợi (queue)**. Các danh sách hạn chế này có thể được tổ chức theo danh sách đặc (mảng) hoặc danh sách liên kết.

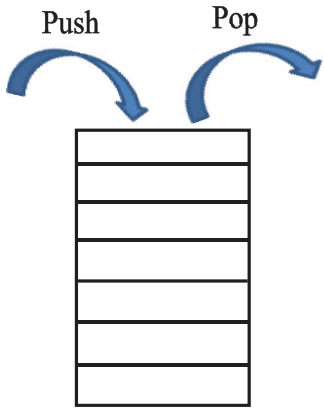
5.2. NGĂN XẾP

5.2.1. Định nghĩa ngăn xếp

Ngăn xếp là danh sách có các thao tác thêm một phần tử vào danh sách và lấy một phần tử ra khỏi danh sách đều được thực hiện ở cùng một đầu của danh sách.

Ta nói ngăn xếp là một cấu trúc hoạt động theo cơ chế "**vào sau – ra trước**" hay còn gọi là **LIFO** (**L**ast **I**n **F**irst **O**ut).

Ngăn xếp có thể được tổ chức theo danh sách đặc hoặc danh sách liên kết.



Hình 5.1: *Stack*

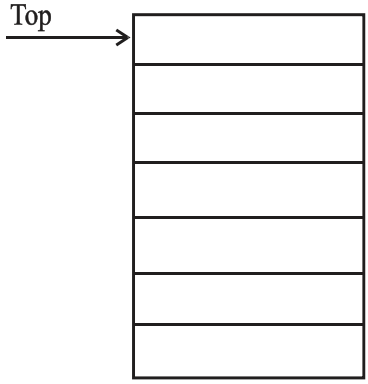
5.2.2. Các thao tác trên ngăn xếp

- Khởi tạo ngăn xếp
- Bổ sung một phần tử vào ngăn xếp (Push)
- Lấy một phần tử ra khỏi ngăn xếp (Pop)
- Kiểm tra ngăn xếp có rỗng (isEmpty)?

5.2.3. Cài đặt ngăn xếp

5.2.3.1. Cài đặt ngăn xếp bằng mảng

Dùng mảng một chiều để lưu trữ ngăn xếp, sử dụng một biến ***Top*** để trỏ đến phần tử ở đỉnh ngăn xếp (Hình 5.2).



Hình 5.2: *Dùng mảng để tổ chức ngăn xếp*

- **Khởi tạo ngăn xếp**

```
template <class T>
class Stack
{
  int top;
  T a[1000];
public:
  Stack()
  {
    top = 0;
  }
};
```

- **Thêm một phần tử vào ngăn xếp (Push)**

```
void Stack::Push(T x)
{
  a[++top] = x;
}
```

- **Lấy một phần tử ra khỏi ngăn xếp (Pop)**

```
T Stack::Pop()
{
  return a[top--];
}
```

- **Kiểm tra ngăn xếp rỗng?**

```
bool Stack::isEmpty()
{
  return top == 0;
}
```

5.2.3.2. Cài đặt ngăn xếp bằng danh sách liên kết đơn

- **Khởi tạo ngăn xếp**

```
class Stack
{
    struct NODE
    {
      int Data;
      NODE *Next;
    };
    NODE *top;
  public:
    Stack()
    {
      top = NULL;
    }
};
```

- **Thêm một phần tử vào ngăn xếp (Push)**

```
void Stack::Push(int x)
{
  //B1: Tao nut moi p chua x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chen p vao dau danh sach
  p->Next = top; //1
  top = p;       //2
}
```

- **Lấy một phần tử ra khỏi ngăn xếp (Pop)**

```
int Stack::Pop()
{
  NODE *p = top;
  top = p->Next;
  int x = p->Data;
  delete p;
  return x;
}
```

- **Kiểm tra ngăn xếp rỗng? (isEmpty)**

```
int Stack::isEmpty()
{
  return top==NULL;
}
```

5.2.3.3. Sử dụng lớp stack của C++

```
#include <stack>
```

Lớp này đã cài đặt sẵn các phương thức:

- ***empty()***: kiểm tra stack rỗng?
- ***push(x)***: bổ sung x vào stack
- ***pop()***: xóa phần tử ở đỉnh stack
- ***top()***: lấy phần tử ở đỉnh stack

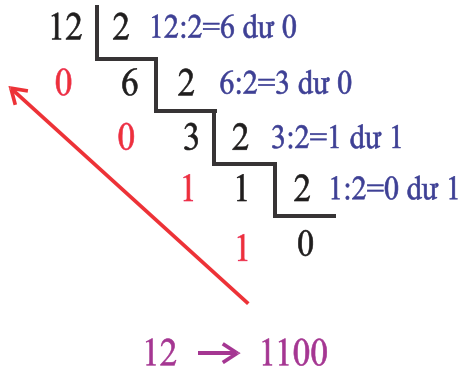
5.2.4. Ứng dụng của ngăn xếp

5.2.4.1. Khử đệ quy

Một chương trình sử dụng giải thuật đệ quy sẽ ngắn gọn, dễ cài đặt, tuy nhiên sự hạn chế của bộ nhớ dành cho chương trình không cho phép xây dựng chương trình bằng đệ quy. Vì vậy, việc chuyển chương trình từ dạng đệ quy sang dạng không đệ quy (chỉ sử dụng các cấu trúc lặp) gọi là khử đệ quy.

Khử đệ quy giúp giữ được nguyên bản thuật toán đệ quy nhưng trong chương trình không hề có lời gọi đệ quy, như thế chương trình có thể chạy được trong bất kỳ môi trường lập trình nào. Ngăn xếp là một trong những công cụ hữu hiệu cho việc khử đệ quy.

Ví dụ 5.1: Ứng dụng ngăn xếp để biểu diễn nhị phân của một số nguyên dương n (Ví dụ: n = 12 sẽ in ra màn hình 1100).



Hình 5.3: *Cách chuyển số thập phân sang dạng nhị phân*

Thuật toán chuyển đổi có thể thực hiện theo các bước sau:

Bước 1: Khai báo stack S rỗng

Bước 2: Trong khi n>0 thì

- PUSH(x%2,S);
- n = n/2;

Bước 3: Lấy tất cả các giá trị trong stack in ra màn hình.

```
#include <iostream>
using namespace std;
template <class T>
class Stack
{
  int top;
  T a[1000];
```

```
public:
    Stack()
  {
    top = 0;
  }

  void push(T x)
  {
    a[++top] = x;
  }

  T pop()
  {
    return a[top--];
  }

  bool isEmpty()
  {
    return top==0;
  }
};
void Bin(int n)
{
  //B1: Khởi tạo stack rỗng
  Stack<int> S;
  //B2: Lấy phần dư n%2 bỏ vào stack
  while(n>0)
  {
    S.push(n%2);
    n = n/2;
  }
  //B3: Lấy hết các phần tử trong
```

```
    //stack in ra màn hình
    while(!S.isEmpty())
    {
      cout<<S.pop();
    }
}

int main()
{
    int n;
    cin>>n;  //13
    Bin(n); //1101
}
```

Cách viết khác: có thể sử dụng lớp stack của C++, sau khi khai báo

```
#include <stack>
```

ta có thể sử dụng stack mà không cần phải định nghĩa lại lớp Stack như trên.

5.2.4.2. Định giá trị biểu thức toán học

Giải thuật định giá trị biểu thức toán học gồm hai giai đoạn sau:

Giai đoạn 1: Chuyển biểu thức toán học từ dạng trung tố sang dạng hậu tố.

Giai đoạn 2: Tính giá trị biểu thức hậu tố vừa tìm được ở giai đoạn 1.

Giai đoạn 1: Áp dụng ký pháp nghịch đảo Balan để chuyển biểu thức toán học từ dạng trung tố sang dạng hậu tố:

Biểu thức toán học bao gồm các ***toán tử*** (các phép toán: +, - , *, /) và các **toán hạng** (hằng, biến, hàm…). Một biểu thức có thể được biểu diễn ở ba dạng khác nhau:

- **Tiền tố (prefix)**: các toán tử đi trước các toán hạng.
- **Trung tố (infix)**: các toán tử đi giữa các toán hạng.
- **Hậu tố (postfix)**: các toán tử đi sau các toán hạng.

Ví dụ 5.2: Biểu thức trung tố (A+B)*C biến đổi thành

Dạng hậu tố: **AB+C***

Dạng tiền tố: ***+ABC**

Có thể nhận thấy rằng, ***ở dạng tiền tố và hậu tố, dấu ngoặc là không cần thiết***.

Giải thuật: Chuyển biểu thức ở dạng trung tố **Infix** sang dạng hậu tố **Posfix** như sau:

Thêm cặp dấu ( ) vào đầu và cuối biểu thức trung tố rồi thực hiện các bước sau:

Bước 1:

- Khởi gán: **Posfix** = ″ ″;
- Khởi tạo ngăn xếp S = ∅;

Bước 2: Xét từng phần tử X thuộc **Infix** (từ trái sang phải):

- Nếu X là dấu ″(″: `PUSH(X,S);`
- Nếu X là dấu ″)″:

```
Token = POP(Stack);
while(Token!="(")
{
  Posfix = Posfix + Token;
  Token = POP(Stack);
}
```

✍***Chú ý****: dấu mở ngoặc ″(″cũng lấy ra khỏi S nhưng không cho vào* **Posfix**.

- Nếu X là toán hạng: **`Posfix = Posfix + X;`**
- Nếu X là toán tử:
  + ***while*** (toán tử ở đỉnh Stack ≥ X) **`Posfix = Posfix + POP(S);`**
  + `PUSH(X,S);`

Bước 3: Lấy các phần tử còn lại trong stack S bổ sung vào cuối **Posfix**.

Ví dụ 5.3: Chuyển biểu thức **(A+B)*(C-D)** từ dạng trung tố sang dạng hậu tố:

Infix	Stack	Posfix
(A+B)*(C-D)	{Empty}	” ”
A+B)*(C-D)	(	” ”
+B)*(C-D)	(	A
B)*(C-D)	(+	A
)*(C-D)	(+	A B
***(C-D)**	{Empty}	A B +
(C-D)	*	A B +
C-D)	* (	A B +
-D)	* (	A B + C
D)	* (-	A B + C
)	* (-	A B + C D
” ”	*	A B + C D -
” ”	{Empty}	**A B + C D - ***

Giai đoạn 2: Tính giá trị biểu thức hậu tố.

Bước 1: Khởi tạo stack S = ∅;

Bước 2: Xét từng phần tử X của **Posfix**:

- Nếu X là toán hạng: PUSH(X,S);
- Nếu X là toán tử:
  + b = POP(S);
  + a = POP(S);
  + Tính z = a **X** b với X là toán tử của **Posfix**.
  + PUSH(z,S);

Bước 3: return POP(S);

Ví dụ 5.4: Tính giá trị biểu thức hậu tố 3 2 + 6 4 - *

Posfix	**Stack**
3 2 + 6 4 - *	∅
2 + 6 4 - *	3
+ 6 4 - *	3 2
6 4 - *	5
4 - *	5 6
- *	5 6 4
*	5 2
” ”	**10**

5.3. HÀNG ĐỢI

5.3.1. Định nghĩa

Hàng đợi là danh sách có các thao tác thêm một phần tử vào danh sách được thực hiện ở một đầu và thao tác lấy một phần tử ra khỏi danh sách được thực hiện ở đầu kia.

Như vậy hàng đợi hoạt động theo cơ chế **FIFO** (**F**irst **I**n **F**irst **O**ut – Vào trước ra trước).

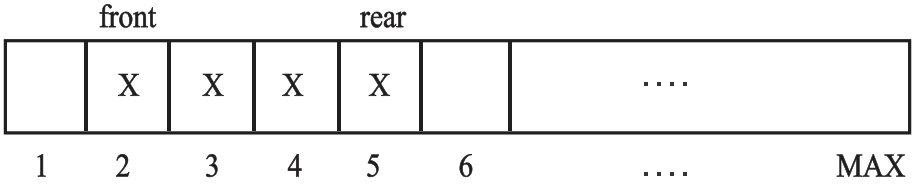
5.3.2. Các phép toán trên hàng đợi

- Khởi tạo hàng đợi
- Bổ sung một phần tử vào hàng đợi
- Lấy một phần tử ra khỏi hàng đợi
- Kiểm tra hàng đợi có rỗng không

5.3.3. Cài đặt hàng đợi

5.3.3.1. Cài đặt hàng đợi bằng mảng

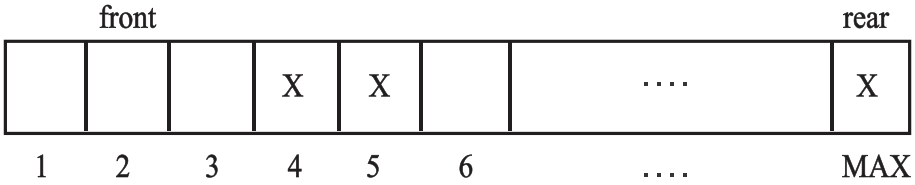
Có thể dùng mảng một chiều có tối đa MAX phần tử để tổ chức hàng đợi. Lối ra và lối vào của hàng đợi được quản lý bởi hai biến chỉ số **front** và **rear** (Hình 5.4).



Hình 5.4: *Tổ chức queue bằng mảng một chiều*

Với cách tổ chức này, vị trí để bổ sung phần tử mới vào queue là vị trí sau **rear**, còn vị trí để lấy phần tử ra là vị trí **front**.

Trong quá trình bổ sung vào và lấy ra có thể xảy ra tình huống không thể bổ sung vào trong khi phần đầu của mảng vẫn còn chỗ trống (Hình 5.5):



Hình 5.5: *Tình huống không thể bổ sung vào queue*

Với tình huống như Hình 5.5, ta có thể khắc phục bằng cách tịnh tiến các phần tử trong queue về phía trước sao cho **front** ở vị trí 1. Điều này có thể dễ dàng thực hiện bằng đoạn mã sau:

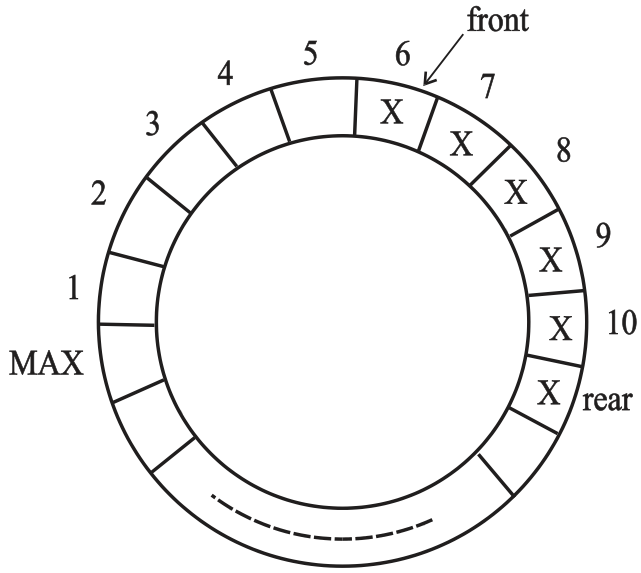
```
for(int i = front;i<=rear;i++)
    Q[i-front+1] = Q[front];
front = 1;
rear = rear - front + 1;
```

Như vậy việc điều chỉnh lại vị trí của front và rear sẽ được thực hiện rất nhiều lần trong quá trình thao tác trên queue. Để khắc phục điều này, ta có thể cải tiến queue bằng cách dùng ***mảng nối vòng***.

Phương án xử lý quy ước cụ thể như sau:

- Sau vị trí i là vị trí i+1, với i = 1, 2, ... , MAX-1.
- Sau vị trí MAX là vị trí 1 (xoay vòng).

Khi đó, **front** và **rear** được dịch chuyển về phía sau theo chiều kim đồng hồ như danh sách nối vòng (Hình 5.6).



Hình 5.6: *Tổ chức hàng đợi nối vòng*

Với cách tổ chức này, ta có thể đặc tả về vị trí đứng sau vị trí **i** là **i mod MAX + 1**.

Từ đó, có thể đặc tả cho các trường hợp queue rỗng và đầy như sau:

- Queue rỗng khi: **front = rear = 0**.
- Queue đầy khi phía sau **rear** là **front**, nghĩa là ***rear mod MAX + 1 = front***.

Với các phân tích trên, các thao tác trên queue có thể được cài đặt như sau:

- **Khởi tạo hàng đợi**

```
class Queue
{
  #define MAX 1000
  int A[MAX];
  int front,rear;
public:
  Queue()
  {
    front = rear = 0;
    }
};
```

- **Kiểm tra hàng đợi có rỗng không**

```
bool Queue::isEmpty()
{
  return front == 0;
}
```

- **Kiểm tra hàng đợi đã đầy hay chưa**

```
bool Queue::isFull()
{
  return (rear%MAX + 1) == front;
}
```

- **Thêm một phần tử vào cuối hàng đợi**

```
void Queue::Push(int x)
{
  if(rear%MAX + 1 == front)
    cout<<"Queue is full";
  else
  {
    rear = rear%MAX + 1;
  A[rear] = x;
  if(front==0)
    front = 1; //do ban dau Queue rong
  }
}
```

- **Lấy ra một phần tử ở đầu hàng đợi**

```
int Queue::Pop()
{
  int x = A[front];
  if(front==rear)
    front = rear = 0;
  else
    front = front%MAX + 1;
  return x;
}
```

5.3.3.2. Cài đặt hàng đợi bằng danh sách liên kết kép

- **Khởi tạo hàng đợi**

```
class Queue
{
  struct NODE
  {
```

```
    int Data;
    NODE *Next, *prev;
  };
  NODE *front, *rear;
public:
  Queue()
  {
    front = rear = NULL;
  }
};
```

- **Kiểm tra hàng đợi có rỗng không**

```
bool Queue::isEmpty()
{
  return front == NULL;
}
```

- **Thêm một phần tử vào hàng đợi**

```
void Queue::Push(int x)
{
  //B1: Tao nut moi p chua x
  NODE *p = new(NODE);
  p->Data = x;
  p->Next = NULL;
  p->prev = NULL;
  //B2: Chen p vao dau danh sach
  if(front==NULL) //Danh sach rong
    front = rear = p;
  else //Danh sach khong rong
  {
```

```
    p->Next = front; //1
    front->prev = p; //2
    front = p;       //3
  }
}
```

- **Lấy ra một phần tử từ hàng đợi**

```
int Queue::Pop()
{
  NODE *p = rear;
  if(front == rear) //Danh sach chi co 1 nut
    front = rear = NULL;
  else
  {
    rear = p->prev;
    rear->Next = NULL;
  }
  int x = p->Data;
  delete p;
  return x;
}
```

5.3.4. Ứng dụng của hàng đợi

Hàng đợi có nhiều ứng dụng trong lĩnh vực khoa học máy tính và mạng. Trong lý thuyết đồ thị, có thể ứng dụng hàng đợi để cài đặt phép duyệt đồ thị theo chiều rộng (BFS).

Trong chương tiếp theo, chúng ta sẽ ứng dụng hàng đợi để duyệt cây nhị phân theo mức.

BÀI TẬP CHƯƠNG 5

Bài 5.1: Cho hai ví dụ thực tế để minh họa cho stack.

Bài 5.2: Cho hai ví dụ thực tế để minh họa cho queue.

Bài 5.3: Sử dụng stack để khử đệ quy cho bài toán tháp Hà Nội.

Bài 5.4: Nhập vào số nguyên dương N ở dạng thập phân. Sử dụng stack để in ra màn hình dạng biểu diễn của N ở hệ cơ số a với a $\in$ [2, 3, 4, 8, 16].

Bài 5.5: Cho xâu ngoặc *S* chỉ chứa ký tự '(' và ')'. Hãy xác định xâu S là xâu ngoặc đúng hay sai.

***Input:** xâu ngoặc *S*.

***Output:** giá trị TRUE ứng với xâu S đúng, FALSE nếu S sai.

Ví dụ:

INPUT	OUTPUT
(()())	TRUE
())()	FALSE

Bài 5.6: Cho xâu ngoặc *S* chứa các cặp dấu ngoặc ( ) [ ] { }. Hãy xác định xâu S là xâu ngoặc đúng hay sai.

***Input:** xâu ngoặc *S*.

***Output:** giá trị TRUE ứng với xâu S đúng, FALSE nếu S sai.

Ví dụ:

INPUT	OUTPUT
{[()]}	TRUE
{[(])}	FALSE

Bài 5.7: Gọi xâu chỉ chứa các ký tự ngoặc tròn (,), ngoặc vuông [,] và ngoặc nhọn {, } là xâu ngoặc.

Xâu ngoặc đúng được định nghĩa như sau:

- Xâu rỗng được coi là xâu ngoặc đúng,
- Nếu a là xâu ngoặc đúng, (a), $[a]$, $\{a\}$ cũng là các xâu ngoặc đúng,
- Nếu a và b là các xâu ngoặc đúng, ab cũng là xâu ngoặc đúng.

Cho xâu $S = s_1s_2\ldots s_n$ độ dài n. Với $k > 1$, xâu $s_ks_{k+1}s_{k+2}\ldots s_ns_1s_2\ldots s_{k-1}$ được gọi là xâu đẩy vòng thứ k của S. Ta quy ước bản thân xâu S cũng là một xâu đẩy vòng của S với $k=1$.

Yêu cầu: Cho xâu ngoặc S có độ dài không quá 1000. Hãy xác định có tồn tại một xâu đẩy vòng của S là xâu ngoặc đúng hay không. Trong trường hợp câu trả lời là khẳng định hãy đưa ra vị trí k nhỏ nhất.

*** Input:** xâu S

*** Output:** vị trí k tìm được, trong trường hợp không có lời giải, ghi ra số -1.

Ví dụ:

INPUT	OUTPUT
} { }( ) {	2
} { ( ) }	-1

Chương 6
CÂY

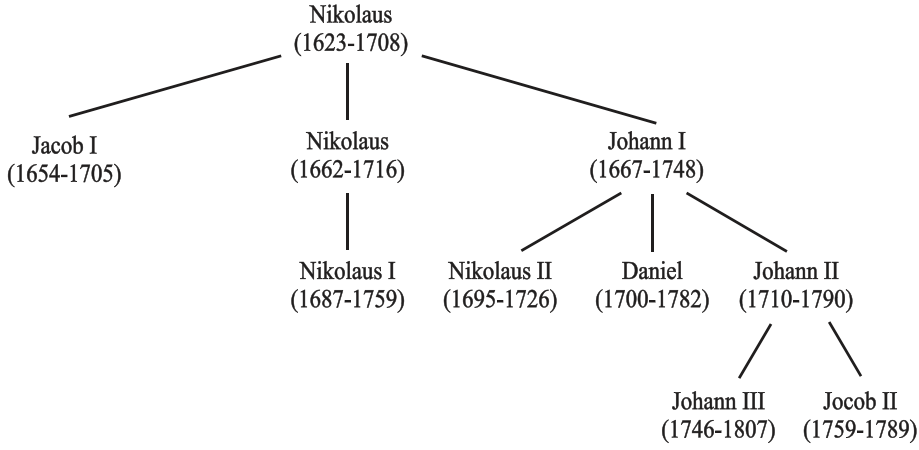
- Giới thiệu tổng quan về cây: định nghĩa và một số khái niệm.
- Cây nhị phân: tổ chức cây, các phép duyệt và ứng dụng.
- Cây tìm kiếm nhị phân (BST), cây AVL.

6.1. GIỚI THIỆU

Trong khoa học máy tính, cây là mô hình trừu tượng của một cấu trúc phân cấp.

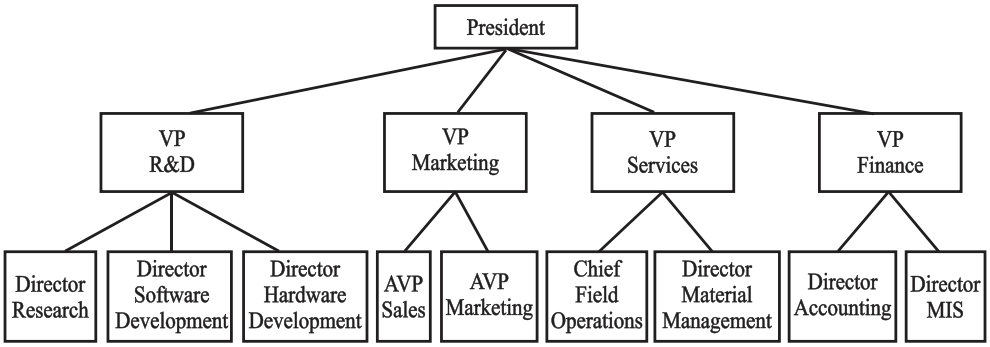
Cây có nhiều ứng dụng thực tiễn:

- Cây gia phả (Hình 6.1)

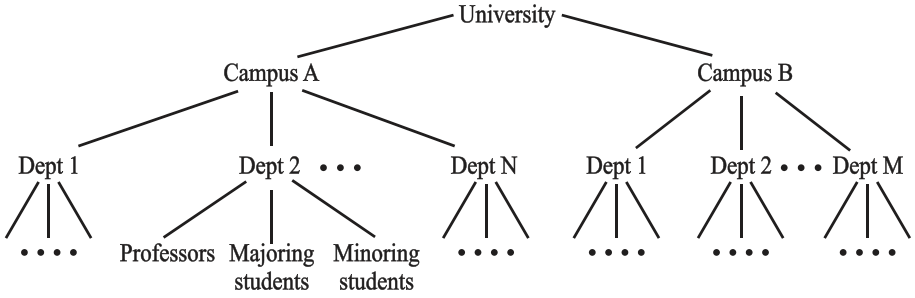


Hình 6.1: *Gia phả của gia đình nhà toán học Bernoulli* [7]

- Các sơ đồ tổ chức chính phủ, doanh nghiệp, trường học (Hình 6.2, Hình 6.3)

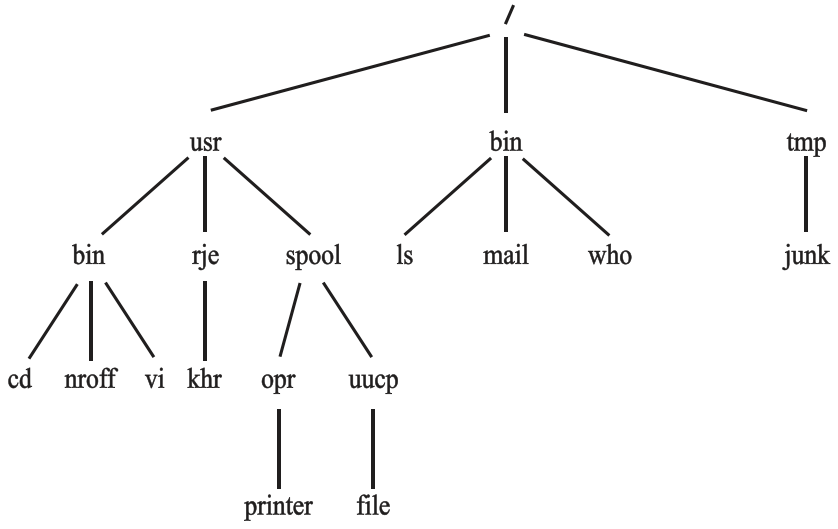


Hình 6.2: *Sơ đồ tổ chức doanh nghiệp* [7]



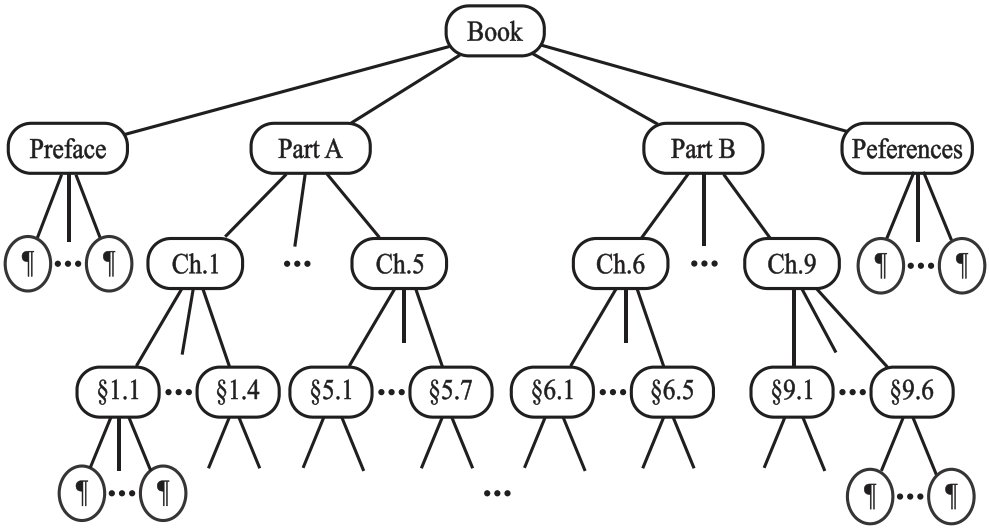
Hình 6.3: *Sơ đồ tổ chức trường học* [7]

- Hệ thống lưu trữ file và thư mục (Hình 6.4)



Hình 6.4: *Hệ thống cây thư mục* [7]

- Mục lục của cuốn sách (Hình 6.5)



Hình 6.5: *Tổ chức mục lục của cuốn sách* [7]

6.2. ĐỊNH NGHĨA VÀ MỘT SỐ KHÁI NIỆM

6.2.1. Định nghĩa

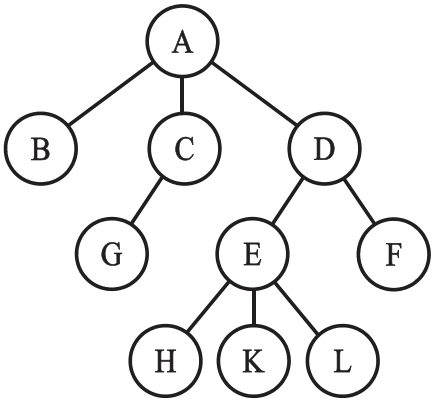
Cây là tập hợp các phần tử (nút) có cùng kiểu dữ liệu, giữa các nút có một quan hệ phân cấp gọi là quan hệ "cha-con", thỏa mãn các điều kiện:

- Mỗi nút có thể có nhiều hoặc không có nút con nào.
- Mỗi nút có duy nhất một nút cha, ngoại trừ một nút đặc biệt không có nút cha gọi là nút *gốc* (*root*).
- Nếu B là nút con của A thì A là nút cha của B và ngược lại.

Quy ước: cây không có nút nào được gọi là ***cây rỗng***.

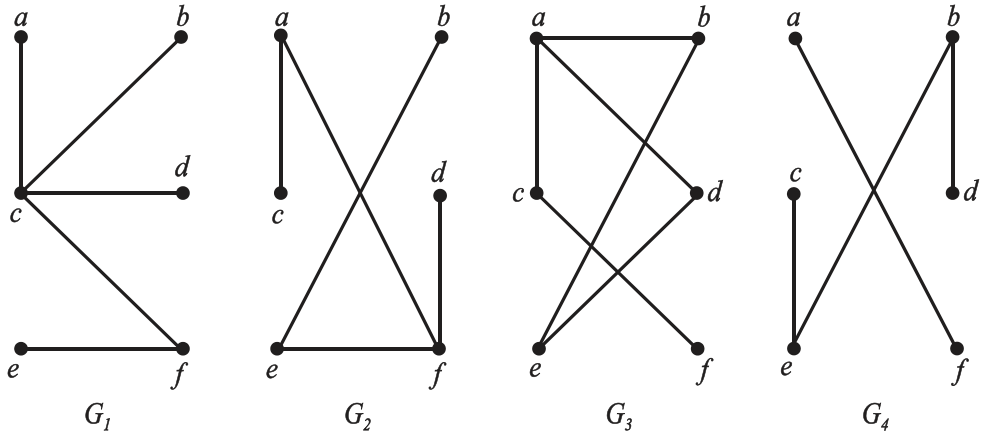
- A là nút gốc của cây.
- B, C, D là các nút con của A (A là nút cha của B, C, D).
- D có hai nút con là E và F.

- Các nút B, G, F, H, K, L không có nút con nào.



Hình 6.6: *Minh họa cây*

Ngoài ra, theo lý thuyết đồ thị, cây là một đồ thị vô hướng liên thông không có chu trình.



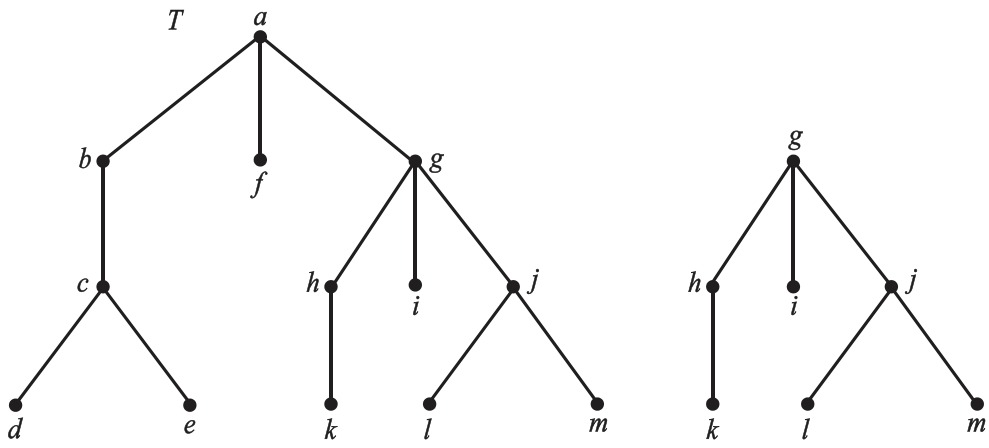
Hình 6.7: *Minh họa về cây và đồ thị* [7]

Theo Hình 6.7:

- G1 và G2 là cây
- G3 không phải cây vì có chu trình
- G4 cũng không phải cây vì đồ thị không liên thông

6.2.2. Các khái niệm

- ***Nút gốc (root)***: Nút gốc là nút duy nhất không có nút cha.
- ***Nút nội bộ (Internal node)***: là nút có ít nhất một nút con.
- ***Nút lá (leaf)***: là nút không có nút con.
- ***Tiền bối (Ancestors)*** của một nút: cha mẹ, ông bà... của nút đó.
- ***Hậu duệ (Descendants)*** của một nút: con, cháu... của nút đó.
- ***Cây con (sub-tree)***: là cây bao gồm một nút và các hậu duệ của nó.



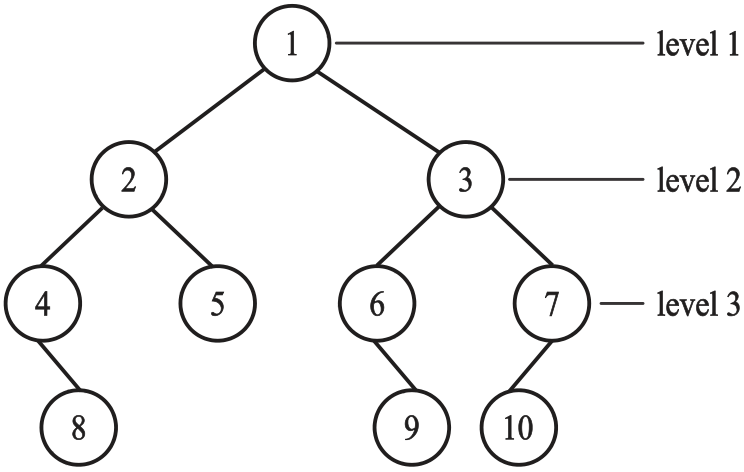
Hình 6.8: *Cây T có nút gốc là* ***a*** *và cây con của T có nút gốc là* ***g*** [7]

- ***Mức của nút (level)***

- Mức của nút gốc là 1.
- Nếu nút cha có mức là i thì nút con có mức là i+1.

- ***Chiều cao của cây (Height of a tree)***: mức cao nhất của các nút trên cây.
  - Cây rỗng có chiều cao bằng 0.
  - Cây chỉ có một nút có chiều cao bằng 1.

- ***Cấp/bậc (order) của một nút***: là số nút con của nút đó.
- ***Bậc của cây***: là bậc lớn nhất của các nút trên cây. Nếu cây có bậc bằng n được gọi là **cây bậc n** hay **cây n_phân**. Ví dụ: cây T ở Hình 6.8 là ***cây tam phân*****,** cây ở Hình 6.9 là ***cây nhị phân***.



Hình 6.9: *Mức của nút*

- **Cây có thứ tự (Ordered Tree)**: là cây mà tại mỗi nút, thứ tự của các nút con của nó được coi trọng (được xếp thứ tự). Ví dụ: Hình 6.5 là cây có thứ tự.
- ***Rừng cây (Forest)***: Rừng là tập hợp các cây. Như vậy một cây khi loại bỏ nút gốc sẽ cho ta một rừng cây.

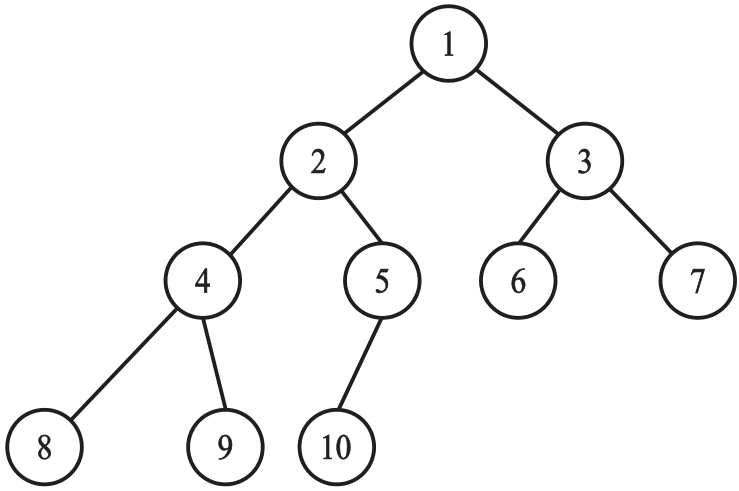
6.3. CÂY NHỊ PHÂN

6.3.1. Định nghĩa

Cây nhị phân là cây mà mọi nút trên cây chỉ ***có tối đa hai nút con***. Với mỗi nút có cây con trái và cây con phải của nút đó.

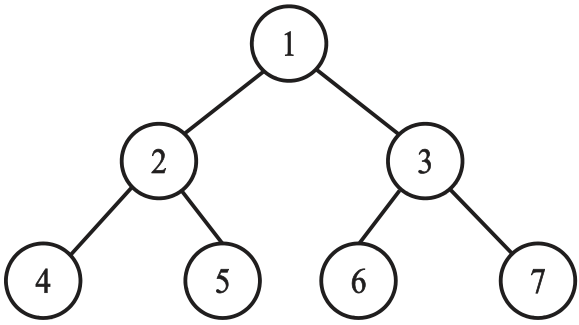
6.3.2. Các khái niệm bổ sung

Cây nhị phân hoàn chỉnh (Full Binary Tree): là cây nhị phân mà mọi nút ***tại mức nhỏ hơn*** mức (h -1) đều có đúng hai nút con - với h là chiều cao của cây (Hình 6.10).



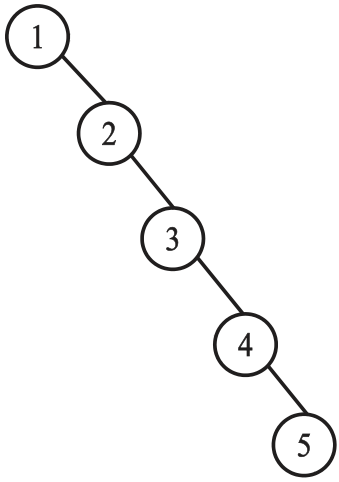
Hình 6.10: *Cây nhị phân hoàn chỉnh*

Cây nhị phân đầy đủ (Complete Binary Tree): là cây nhị phân mà mọi nút tại mức nhỏ hơn hoặc bằng mức (h -1) đều có đúng hai nút con - với h là chiều cao của cây (Hình 6.11).



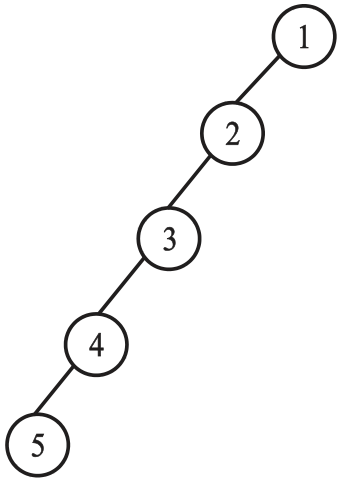
Hình 6.11: *Cây nhị phân đầy đủ*

Cây lệch phải là cây nhị phân chỉ có cây con phải, tức là các nút trên cây chỉ có cây con phải mà không có cây con trái (Hình 6.12).



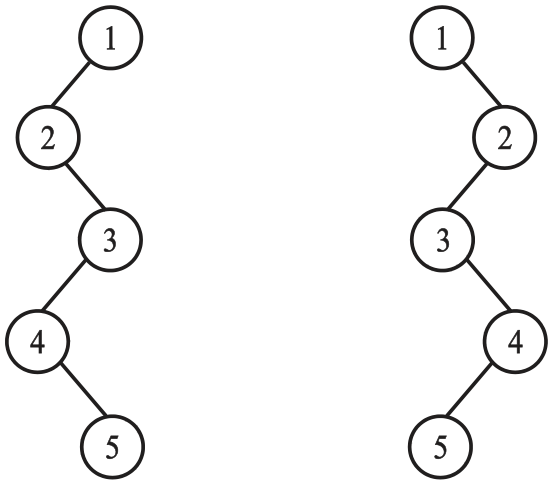
Hình 6.12: *Cây nhị phân lệch phải*

Cây lệch trái là cây nhị phân chỉ có cây con trái, tức là các nút trên cây chỉ có cây con trái mà không có cây con phải (Hình 6.13).



Hình 6.13: *Cây nhị phân lệch trái*

Cây zíc zắc là cây nhị phân mà các nút trái và nút phải của cây đan xen nhau thành một hình zíc zắc (Hình 6.14).



Hình 6.14: *Cây zíc zắc*

Các loại cây lệch phải, cây lệch trái, cây zíc zắc được gọi là ***cây nhị phân suy biến***.

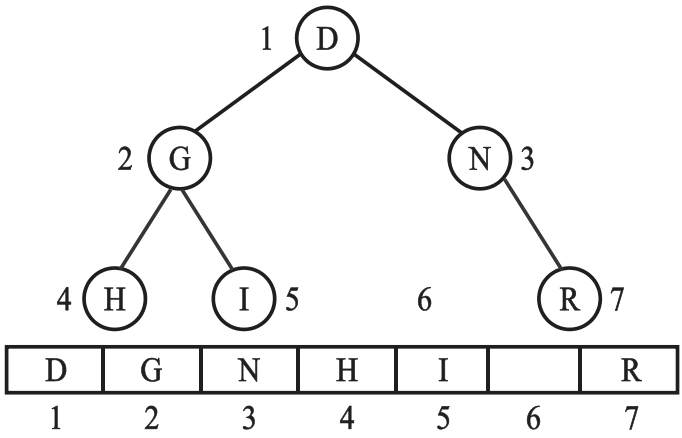
6.3.3. Tổ chức lưu trữ cây nhị phân

Để lưu trữ cây nhị phân, có thể sử dụng mảng hoặc danh sách liên kết.

6.3.3.1. Lưu trữ cây nhị phân bằng mảng

Nguyên tắc: Đánh số trên cây nhị phân theo thứ tự từ trên xuống và từ trái sang phải (tính luôn cả những nút bị khuyết để tạo thành một cây nhị phân hoàn chỉnh). Chỉ số được đánh chính là chỉ số của phần tử mảng chứa node đó.

Ví dụ 6.1: Cho cây nhị phân như Hình 6.15



Hình 6.15: *Tổ chức cây nhị phân bằng mảng*

Nhận xét: Theo cách lưu trữ trên mảng, ta nhận thấy:

- Nếu nút cha ở vị trí thứ **i** thì nút con trái của nó có vị trí là **2*i**, và nút con phải ở vị trí là **2*i+1**.
- Nếu nút con trái ở vị trí **i**, thì nút cha của nó ở vị trí là **i/2**.

Việc lưu trữ dữ liệu cây nhị phân trên mảng chỉ thích hợp với cây nhị phân đầy đủ hay cây nhị phân hoàn chỉnh. Còn đối với các dạng khác, việc lưu trữ bằng mảng sẽ gây ra hiện tượng lãng phí bộ nhớ. Vì vậy, người ta thường tổ chức lưu trữ cây nhị phân bằng danh sách liên kết.

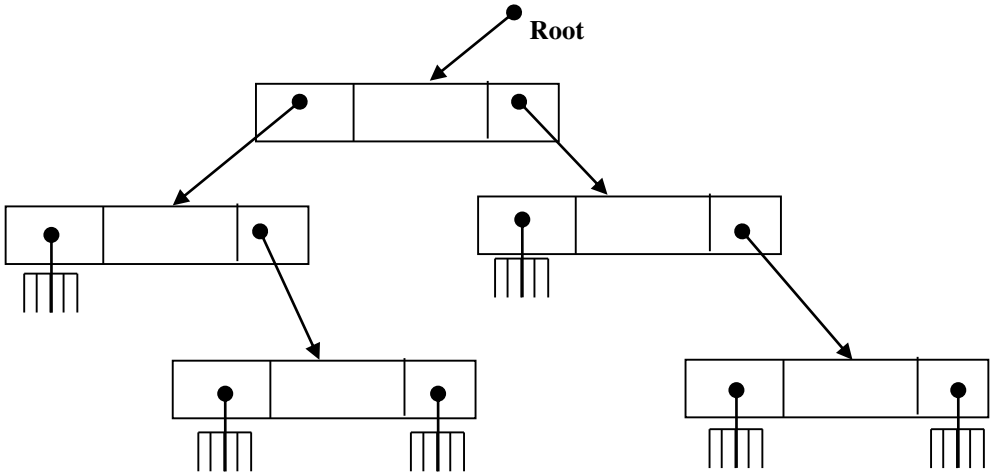
6.3.3.2. Lưu trữ cây nhị phân bằng danh sách liên kết

Ngoài việc lưu trữ cây nhị phân bằng mảng người ta còn có thể lưu trữ cây nhị phân bằng danh sách liên kết kép. Với mỗi nút, ngoài trường Data để chứa dữ liệu, dùng thêm hai trường liên kết Left và Right để trỏ đến địa chỉ nút con bên trái và nút con bên phải.

Cấu trúc một nút được mô tả như sau:

Left	**Data**	**Right**

Như vậy, có thể hình dung một cây nhị phân được lưu trữ dưới dạng danh sách liên kết sẽ được tổ chức như Hình 6.16:



Hình 6.16: *Tổ chức cây nhị phân bằng danh sách liên kết*

Ta có thể khai báo cấu trúc dữ liệu một nút của cây nhị phân như sau:

```
struct NODE
{
  <Type> Data;
  NODE *Left, *Right;
};
typedef NODE* TREE;
```

Khi đó, có thể khai báo một biến con trỏ ***Root*** trỏ vào nút gốc của cây nhị phân:

```
TREE Root;
```

6.3.4. Các phép duyệt trên cây nhị phân

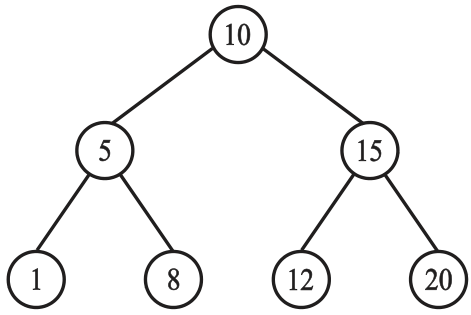
6.3.4.1. Duyệt trái (NLR)

Theo cách duyệt này, nút gốc sẽ được thăm trước, sau đó mới đến duyệt thứ tự hai cây con trái và phải.

- Thăm nút gốc (N).

- Duyệt cây con bên trái (L).
- Duyệt cây con bên phải (R).

Ví dụ 6.2: Cho cây nhị phân như Hình 6.17.



Hình 6.17.

Duyệt theo NLR sẽ cho kết quả: 10, 5, 1, 8, 15, 12, 20

6.3.4.2. Duyệt giữa (LNR)

- Duyệt cây con bên trái (L).
- Thăm nút gốc (N).
- Duyệt cây con bên phải (R).

Duyệt theo LNR sẽ cho kết quả: 1, 5, 8, 10, 12, 15, 20.

6.3.4.3. Duyệt phải (LRN)

- Duyệt cây con bên trái (L).
- Duyệt cây con bên phải (R).
- Thăm nút gốc (N).

Duyệt theo LRN sẽ cho kết quả: 1, 8, 5, 12, 20, 15, 10.

6.3.4.4. Duyệt cây theo mức

Ý tưởng của phương pháp này là sử dụng hàng đợi để duyệt.

Bước 1: Khởi tạo hàng đợi Q = NULL;

Bước 2: Đưa nút gốc vào Q;

Bước 3: Trong khi Q khác rỗng:

- Lấy 1 nút p từ Q, thăm p->Data;
- Đưa 2 nút con của p (nếu có) vào Q.

Giải thuật trên có thể được cài đặt như sau:

```
void DuyetMuc(TREE Root)
{
  Queue Q = NULL;
  PUSH(Root,Q);
  while(Q)
  {
    TREE p = POP(Q);
    <Thăm p->Data>;
    if(p->Left) PUSH(p->Left,Q);
    if(p->Right) PUSH(p->Right,Q);
  }
}
```

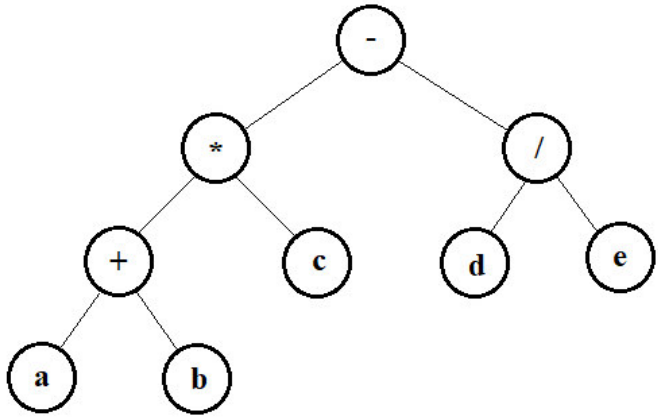
Kết quả duyệt cây theo mức: 10, 5, 15, 1, 8, 12, 20.

6.4. CÂY BIỂU THỨC

Một trong những ứng dụng của cây nhị phân là lưu trữ biểu thức số học.

Cây biểu thức là một dạng cây nhị phân, có các nút lá là các toán hạng, các nút nội bộ là các toán tử.

Ví dụ 6.3: Xét biểu thức số học đơn giản: **(a+b)*c-d/e.** Biểu thức này có thể được biểu diễn dưới dạng cây nhị phân (Hình 6.18).



Hình 6.18: *Cây biểu thức*

Quy tắc biểu diễn một biểu thức số học trên cây như sau

- Mỗi nút lá là một toán hạng.
- Mỗi nút nội bộ là một toán tử.

Khi duyệt cây biểu thức, thứ tự các phép duyệt:

- NLR: cho biểu thức dạng tiền tố (prefix).
- LNR: cho biểu thức dạng trung tố (infix) không có dấu ngoặc.
- LRN: cho biểu thức dạng hậu tố (posfix).

Ví dụ 6.4: (ứng dụng) Cho biểu thức số học dưới dạng tiền tố.

1. Tổ chức cây nhị phân để lưu trữ biểu thức.
2. Tính giá trị của biểu thức.

Sau đây là chương trình chuyển biểu thức dạng tiền tố sang cấu trúc cây nhị phân, từ đó tính giá trị biểu thức được lưu trữ dưới dạng cây nhị phân:

```
#include <iostream>
#include <string>
#include <conio.h>
#include "windows.h"
```

```
using namespace std;
#ifdef WIN32
void gotoxy(int x, int y)
  {
    COORD cur = {x, y};

SetConsoleCursorPosition(GetStdHandle(STD_OUTPU
T_HANDLE), cur);
}
#else
void gotoxy(int x, int y)
{
  printf("\033[%dG\033[%dd", x+1, y+1);
}
#endif
struct NODE
{
  char x;
  NODE *Left,*Right;
};
typedef NODE* Tree;
class ExpTree
{
    int i=-1;
    char Token;
  public:
    Tree Root;
    string st;
    ExpTree()
    {
      Root=NULL;
    }
```

```
    ExpTree(string s)
    {
      Root=NULL;
      st = s;
    }
    void CreateTree(string st, Tree &Root)
    {
      i++;
      Token = st[i];
      Root = new(NODE);
      Root->x = Token;
      if((Token>='0')&&(Token<='9'))
      {
        Root->Left = NULL;
        Root->Right = NULL;
      }
      else  //Token is [+,-,*,/]
      {
        CreateTree(st,Root->Left);
            CreateTree(st,Root->Right);
          }
    }
    void View(Tree Root,int x,int y,int k) //Pre
order
    {
        if(Root)
        {
            gotoxy(x,y); cout<<Root->x;
            XemCay(Root->Left,x - k/2,y+2,k-3);
            XemCay(Root->Right,x + k/2,y+2,k-3);
        }
    }
```

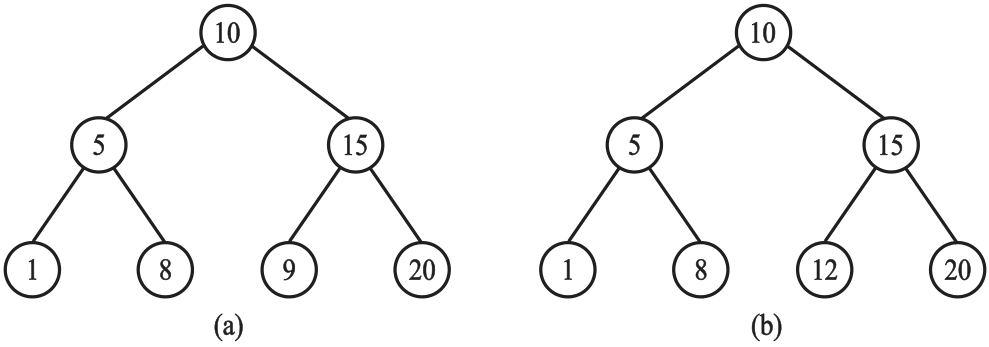
```
    float Value(Tree Root)
    {
        switch(Root->x)
        {
          case '+': return Value(Root->Left)
            + Value(Root->Right);
          case '-': return Value(Root->Left)
            - Value(Root->Right);
          case '*': return Value(Root->Left)
            * Value(Root->Right);
          case '/': return Value(Root->Left)
            / Value(Root->Right);
          default : return Root->x - 48;
        }
    }
};
int main()
{
  ExpTree t("*-62+34");
  t.CreateTree(t.st,t.Root);
    t.View(t.Root,40,5,10);
    cout<<endl<<"Gia tri: "<<t.Value(t.Root);
}
```

6.5. CÂY TÌM KIẾM NHỊ PHÂN

6.5.1. Định nghĩa

Cây tìm kiếm nhị phân (BST – Binary Search Tree) là cây nhị phân mà tại mỗi nút, khóa của nút đang xét lớn hơn khóa của tất cả các nút thuộc cây con trái và nhỏ hơn khóa của tất cả các nút thuộc cây con phải.

Ví dụ 6.5:



Hình 6.19: *Cây trong hình (a) không phải là cây BST, cây trong hình (b) là cây BST*

Đối với cây BST, khi duyệt cây theo thứ tự giữa (LNR) sẽ cho ra một dãy có thứ tự tăng dần.

6.5.2. Các thao tác trên cây BST

6.5.2.1. Khởi tạo cây

```
BST Root = NULL;
```

6.5.2.2. Duyệt cây

Để duyệt cây ta có 3 cách duyệt nhưng ở phần này chỉ xét đến phần duyệt theo thứ tự LNR để có thể thấy rõ hơn tính chất của cây BST. Hàm hiển thị nội dung cây theo thứ tự LNR như sau:

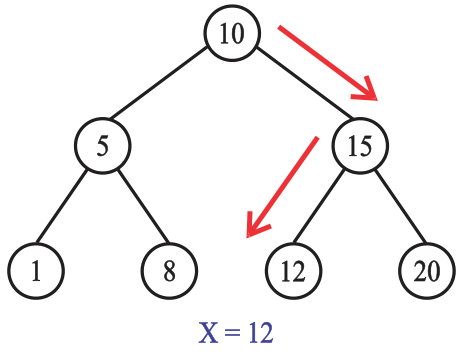
```
void LNR(BST Root)
  {
     if (Root)
    {
    LNR(Root->Left);
    <Thăm Root->Data>;
```

```
    LNR(Root->Right);
    }
  }
```

6.5.2.3. Tìm kiếm một giá trị X trên cây BST

Muốn tìm kiếm giá trị X trên cây BST, thực hiện các bước như sau:

- Nếu cây rỗng, không tìm thấy.
- Nếu X = giá trị tại nút gốc: tìm thấy, trả về địa chỉ của nút gốc.
- Nếu X < giá trị tại nút gốc: tìm kiếm X bên cây con trái. Ngược lại, tìm kiếm X bên cây con phải.



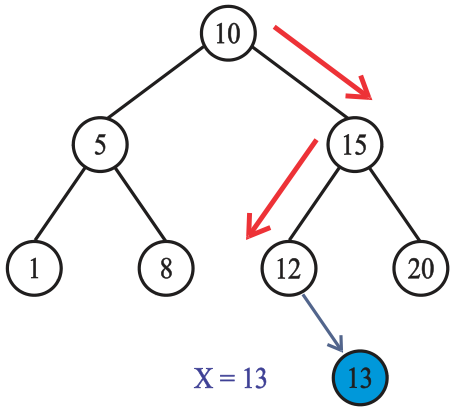
Hình 6.20: *Tìm giá trị X = 12 trên cây BST*

```
BST Search(int x,BST Root)
{
  if(Root==NULL) return NULL;
  else if(x == Root->Data) return Root;
    else if(x < Root->Data)
      return Search(x,Root->Left);
        else
          return Search(x,Root->Right);
}
```

6.5.2.4. Thêm một nút mới vào cây BST

Để thêm một nút mới chứa giá trị X vào cây, thực hiện theo các bước như sau:

- Nếu cây rỗng: tạo nút mới chứa X, nút này cũng là nút gốc.
- Nếu X = giá trị tại nút gốc: dừng giải thuật vì không cần thêm nút vào cây.
- Nếu X > giá trị tại nút gốc: gọi đệ quy chèn vào cây con phải, ngược lại gọi đệ quy chèn vào cây con trái.



Hình 6.21: *Chèn giá trị X = 13 vào cây BST*

```
void Insert(int x,BST Root)
{
  if(Root==NULL)
  {
    new(Root);
    Root->Data = x;
    Root->Left = NULL;
    Root->Right =NULL;
  }
  else if(x < Root->Data)
```

```
    Insert(x,Root->Left);
  else if(x > Root->Data)
    Insert(x,Root->Right);
}
```

6.5.2.5. Xóa một nút có khóa X

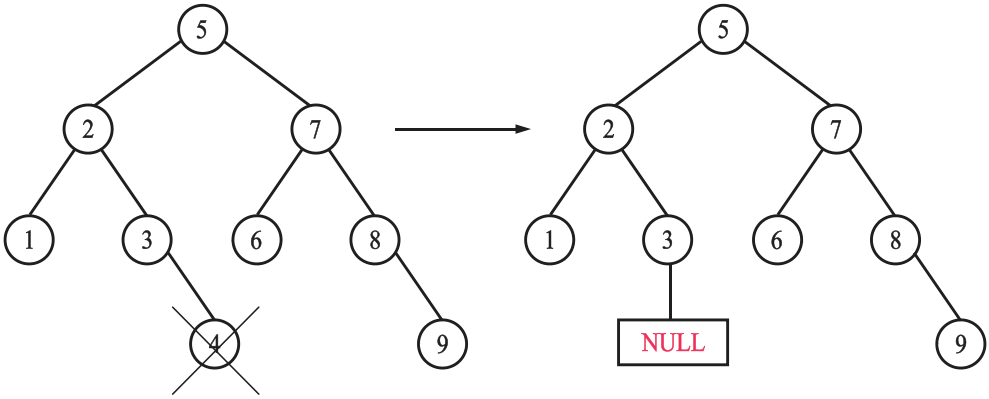
Muốn xóa một nút chứa khóa X, ta tiến hành theo các bước sau đây:

Bước 1: Tìm kiếm nút chứa khóa X trên cây, nếu không có, kết thúc giải thuật, ngược lại chuyển qua bước 2.

Bước 2: Xét ba trường hợp sau:

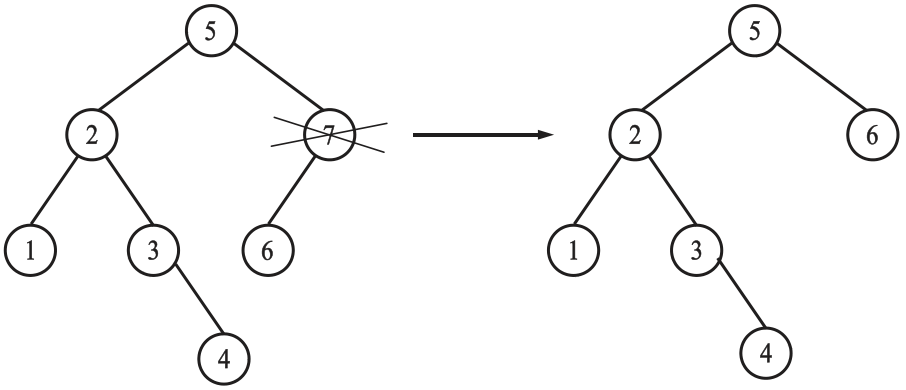
1) X là nút lá
2) X chỉ có một cây con (trái hoặc phải).
3) X có đủ cả hai cây con.

Trường hợp 1: Nếu X là nút lá, cho con trỏ của nút cha của X trỏ vào NULL, sau đó xóa X.



Hình 6.22: *Xóa nút lá*

Trường hợp 2: Trước khi xóa X, móc nối cha của X với nút con duy nhất của X.



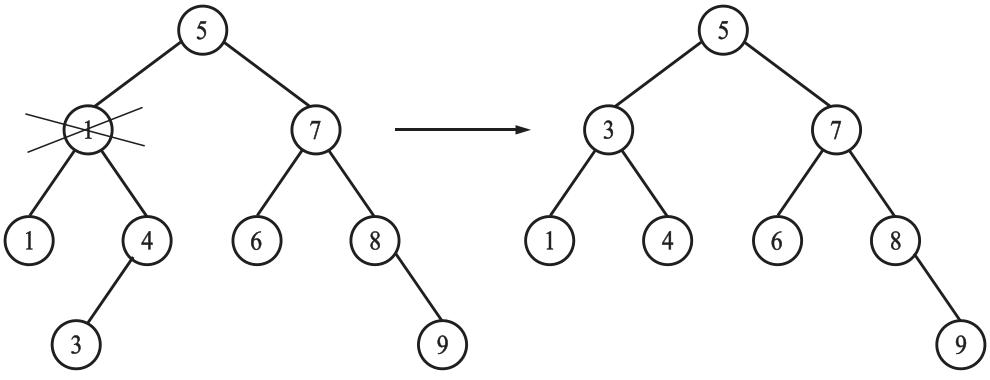
Hình 6.23: *Xóa nút chỉ có một cây con*

Trường hợp 3: Thay vì xóa X, tìm một phần tử thế mạng Y. Phần tử này có tối đa một con. Thông tin lưu tại Y sẽ được chuyển lên lưu tại X.

Vấn đề là phải chọn Y sao cho khi lưu Y vào vị trí của X, cây vẫn là cây BST.

Có 2 cách chọn thỏa mãn yêu cầu:

- Thay X bởi nút **cực phải** của ***cây con trái***.
- Thay X bởi nút **cực trái** của ***cây con phải***.

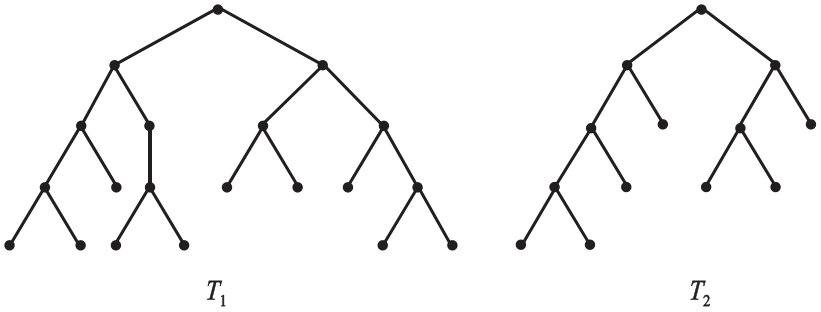


Hình 6.24: *Xóa nút có đủ hai cây con*

6.6. CÂY TÌM KIẾM NHỊ PHÂN CÂN BẰNG

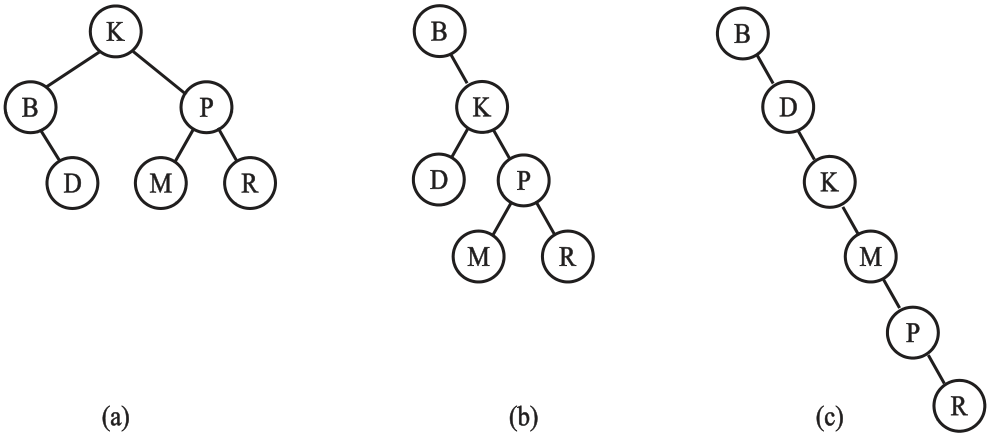
6.6.1. Định nghĩa

Một cây nhị phân được gọi là ***cân bằng*** (***balanced***) nếu mọi nút lá của cây đều ở mức h hoặc h-1 (h: chiều cao của cây).



Hình 6.25: *T_1 là cây cân bằng, T_2 không phải là cây cân bằng*

Mục đích tạo ra cây BST cân bằng là để tối ưu trong việc tìm kiếm. Nếu một cây BST bị lệch trái hoặc lệch phải, việc tìm kiếm trên cây sẽ có độ phức tạp O(n), còn nếu cây BST cân bằng thì độ phức tạp tìm kiếm sẽ là O(logn).

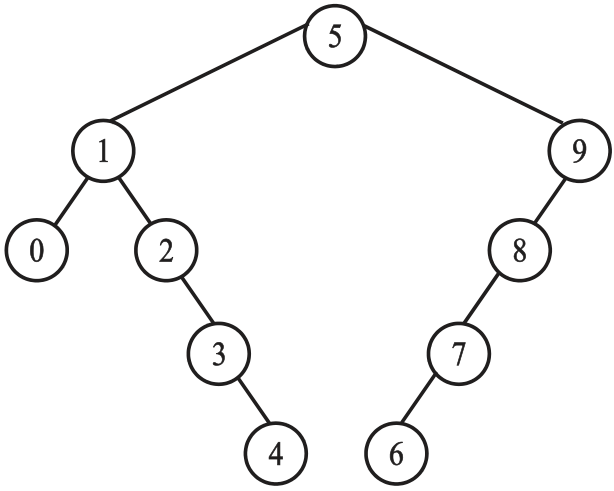


Hình 6.26: *Các cây BST khác nhau chứa cùng thông tin*

6.6.2. Thuật toán cân bằng đơn giản

Giả sử nhập vào cây BST các giá trị theo thứ tự: 5 1, 9, 8, 7, 0, 2, 3, 4, 6.

Cây BST sẽ được tạo ra theo thứ tự nhập (Hình 6.27).



Hình 6.27: *Cây BST lưu các giá trị theo thứ tự:*
5 1, 9, 8, 7, 0, 2, 3, 4, 6

Mảng lưu trữ dữ liệu:

0	1	2	3	4	5	6	7	8	9

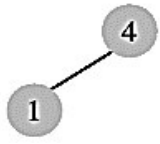
Tạo cây mới bằng cách lấy phần tử ở giữa mảng bổ sung vào cây.

(a) Bổ sung 4 vào cây

0 1 2 3 [4] 5 6 7 8 9

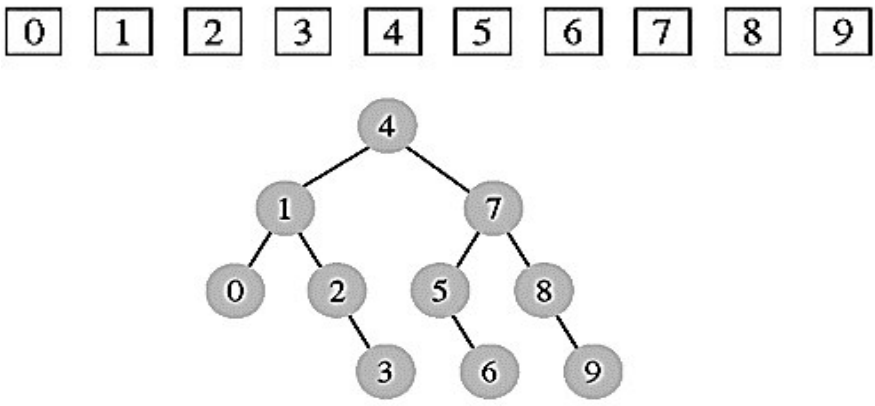
(b) Bổ sung 1 vào cây

0 [1] 2 3 [4] 5 6 7 8 9



(c) Bổ sung 0, 2 vào cây

(d) Bổ sung các phần tử còn lại vào cây



Hình 6.28: *Cây BST cân bằng mới được tạo ra*

Cây BST mới được tạo ra là cây cân bằng (Hình 6.28). Thuật toán tạo ra cây BST cân bằng được thực hiện như sau:

Bước 1: Sao chép tất cả các nút của cây vào mảng.

Bước 2: Sắp xếp mảng tăng dần.

Bước 3: Xóa cây.

Bước 4: Gọi hàm **Balance()** để xây dựng lại cây.

```
void Balancing(T data[], int left, int right)
{
  if (left <= right)
  {
    int mid = (left + right)/2;
    Insert(data[mid],Root);
```

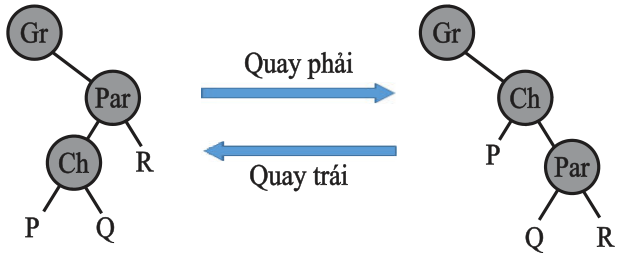
```
    Balancing(data, left,mid-1);
    Balancing(data,mid+1, right);
  }
}
void Balance(T data[])
{
  balancing(data,0,data.length-1);
}
```

6.6.3. Các phép xoay để cân bằng cây BST

6.6.3.1. Phép xoay phải

Quay nút ***Par*** qua phải nút ***Ch*** - nút con trái của ***Par*** (Hình 6.29).

- ***Ch*** trở thành nút gốc mới của cây con
- Cây con phải của ***Ch*** trở thành cây con trái của ***Par***
- Cây con ***Par*** trở thành cây con phải của ***Ch***



Hình 6.29: *Phép xoay phải nút Par và xoay trái nút Ch*

6.6.3.2. Phép xoay trái

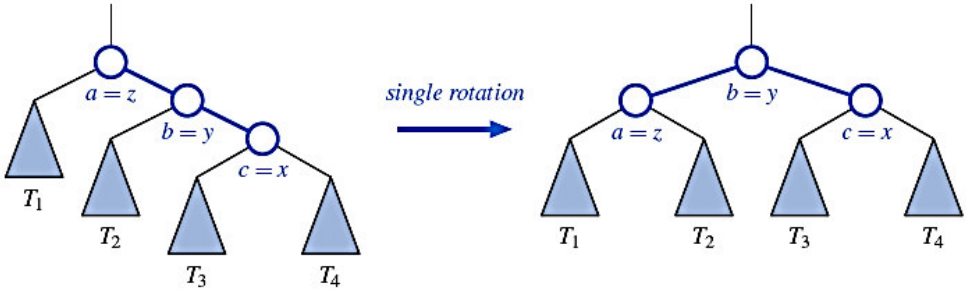
Quay nút ***Ch*** qua trái nút ***Par*** - nút con phải của ***Ch*** (Hình 6.29).

- ***Par*** trở thành nút gốc mới của cây con
- Cây con trái của ***Par*** trở thành cây con phải của ***Ch***
- Cây con ***Ch*** trở thành cây con trái của ***Par***

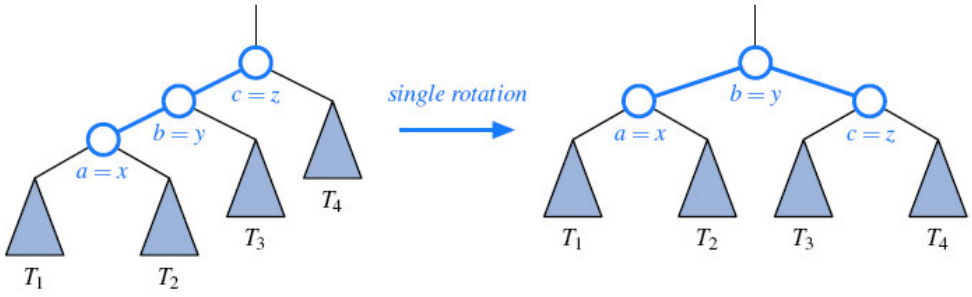
6.6.3.3. Phép xoay đơn

Các phép xoay trái và xoay phải như trên được gọi là các phép ***xoay đơn (single rotation)***.

Ví dụ 6.6: Hình 6.30 và Hình 6.31 là các phép xoay đơn để cân bằng cây.



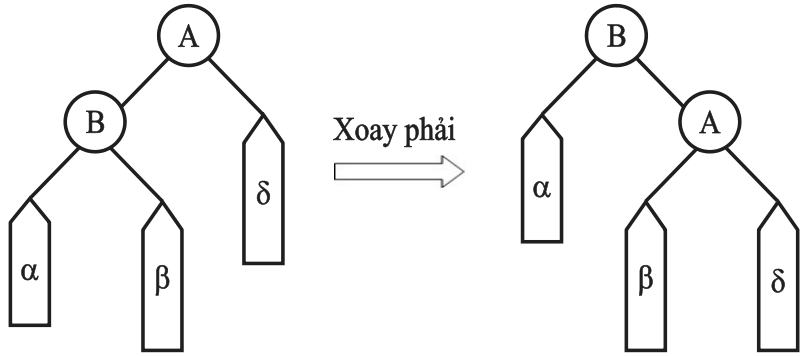
Hình 6.30: *Phép xoay trái nút a = z* [5]



Hình 6.31: *Phép xoay phải nút c = z* [5]

6.6.3.4. Phép xoay kép

Giả sử cây con có gốc A cần thực hiện phép xoay phải như Hình 6.32.



Hình 6.32: *Phép xoay kép*

Gọi h(α), h(β) và h(δ) là chiều cao của các cây con α, β và δ. Nếu h(β)>h(α), chiều cao của cây con trái gốc A là h(β)+1, khi đó [h(β)+1)] - h(δ) > 1 (vì A là nút bất thường có chiều cao cây con trái lớn hơn). Khi đó, cây sau khi xoay có gốc B, có chiều cao của cây con trái là h(α) và chiều cao của cây con phải là h(β)+2. Vì h(β) > h(α) nên h(β) - h(α) > 1, do đó cây có gốc B lại là nút bất thường, điều đó cho thấy chưa cải thiện được tính cân bằng của cây.

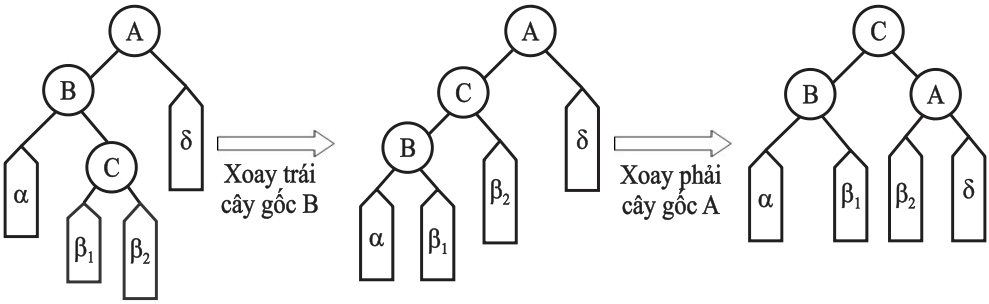
Để khắc phục điều này, cần làm như sau:

- Xoay trái cây con trái gốc B để được cây con trái có chiều cao không lớn hơn cây con phải.
- Xoay phải cây con có gốc A.

Công việc trên được gọi là ***xoay phải kép*** (***double rotation***) cây có gốc A.

Vì h(β)>h(α) nên cây con β phải có ít nhất một nút. Gọi C là gốc của cây β và β_1, β_2 là cây con trái và cây con phải của cây con β.

Việc xoay phải kép cây A ở trên được minh họa như Hình 6.33.



Hình 6.33: *Phép xoay kép*

Sau đây là giải thuật xoay phải kép với cây con có nút gốc trỏ bởi p:

```
void RightRotation(p)
{
  - Xoay trái cây con trái của p;
  - Xoay phải p;
}
```

Đối với phép xoay trái kép cũng tương tự:

```
void LeftRotation(p)
{
  - Xoay phải cây con phải của p;
  - Xoay trái p;
}
```

6.7. CÂY AVL

Lớp cây này được đề xuất bởi hai nhà toán học người Nga G.M. **A**delsen **V**elskii và E.M. **L**endis vào năm 1962, được gọi là ***cây cân bằng AVL***.

Cây AVL là cây tìm kiếm nhị phân sao cho tại mỗi nút của cây, chiều cao của cây con trái và cây con phải chênh lệch nhau không quá 1.

Cho một cây tìm kiếm nhị phân T, gọi p là một nút nào đó trên cây, ký hiệu lh(p) và rh(p) là chiều cao của cây con trái và cây con phải của nút p. Khi đó:

Nếu lh(p) = rh(p): p là nút cân bằng.

Nếu lh(p) = rh(p) + 1: p là nút lệch trái.

Nếu rh(p) = lh(p) + 1: p là nút lệch phải.

Nếu |lh(p) - rh(p)| > 1: p là nút bất thường.

Như vậy, một cây T là cân bằng thì trong đó không có nút nào là bất thường. Các thao tác (thay đổi lại các mối nối) trên cây sao cho cây không còn nút bất thường được gọi là "*cân bằng lại cây*". Đối với cây tìm kiếm nhị phân, sau khi cân bằng lại cây mà cây vẫn là cây BST, cây đó được gọi là cây AVL.

6.7.1. Chèn một nút mới vào cây AVL

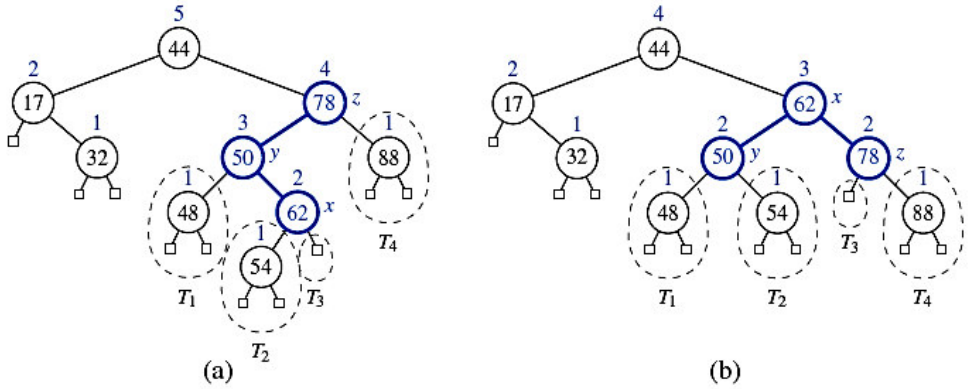
Bước 1: Chèn nút mới như ở cây BST.

Bước 2: Tính toán lại hệ số cân bằng của các nút từ nút được chèn ngược về nút gốc. Hệ số cân bằng của nút p: **bf(p) =** rh(p) – lh(p).

- Nếu không có nút bất thường trên cây (tức bf(p) = -2 hoặc 2): dừng.
- Nếu tìm thấy nút p mà bf(p) = 2 hoặc -2, quay p quanh nút con q của nó.
  - Nếu bf(p) và bf(q) cùng dấu, chỉ cần duy nhất một phép xoay đơn.
  - Ngược lại, xoay kép: đầu tiên là xoay q, sau đó là xoay p.

Quy luật xoay: nếu nút p lệch trái, xoay phải và ngược lại.

Ví dụ 6.7:



Hình 6.34: *Cân bằng lại cây sau khi chèn nút 54 vào cây* [5].

Hình 6.34 (a): chèn nút chứa khóa 54 vào cây BST.

Hình 6.34 (b): cân bằng lại cây.

- Xoay trái nút chứa khóa 50
- Xoay phải nút chứa khóa 78.

6.7.2. Xóa một nút khỏi cây AVL

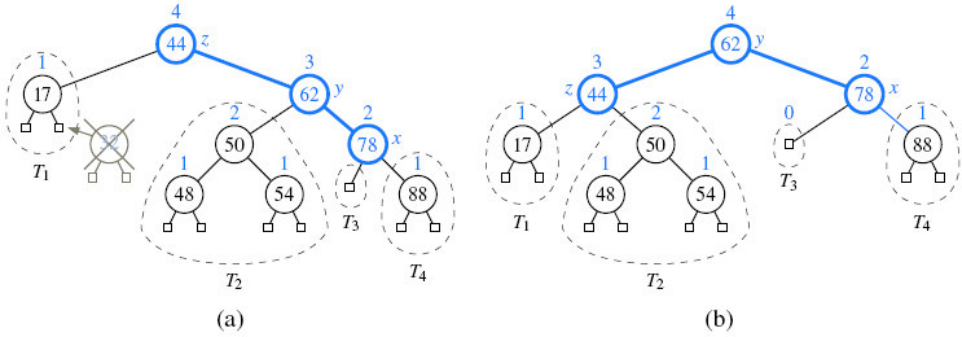
Bước 1: Xóa nút như ở cây BST.

Bước 2: Tính toán lại hệ số cân bằng của các nút từ nút bị xóa ngược về nút gốc. Hệ số cân bằng của nút p: **bf(p)** = rh(p) – lh(p).

- Nếu không có nút bất thường trên cây (tức bf(p) = -2 hoặc 2): dừng.
- Nếu tìm thấy nút p mà bf(p) = 2 hoặc -2, quay p quanh nút con q của nó.
  - Nếu bf(p) và bf(q) cùng dấu, chỉ cần duy nhất một phép xoay đơn.
  - Ngược lại, xoay kép: đầu tiên là xoay q, sau đó là xoay p.

Quy luật xoay: nếu nút p lệch trái thì xoay phải và ngược lại.

Ví dụ 6.8:



Hình 6.35: *Cân bằng lại cây sau khi xóa nút 32* [5].

Hình 6.35 (a): Xóa nút chứa khóa 32 trên cây BST.

Hình 6.35 (b): cân bằng lại cây bằng cách xoay trái nút chứa khóa 44.

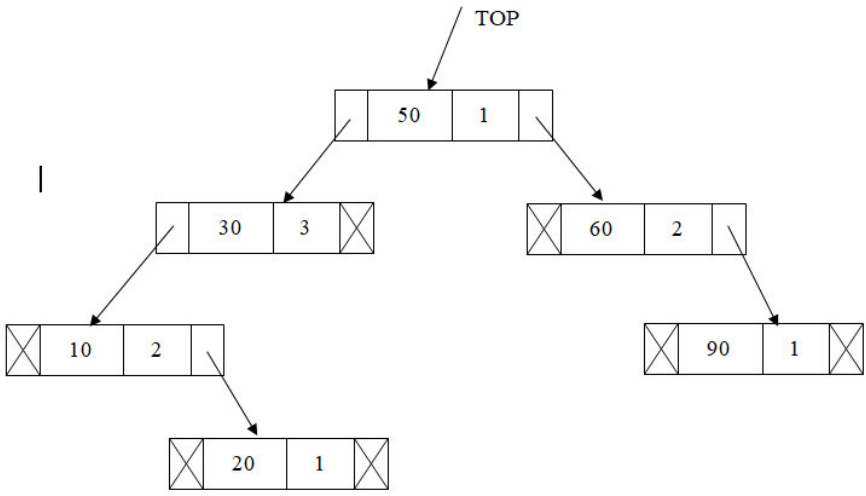
BÀI TẬP CHƯƠNG 6

Bài 6.1: Tổ chức cây tìm kiếm nhị phân chứa các số nguyên, sau đó lập trình để thực hiện các công việc sau:

1. Viết hàm **Insert(int X)** để chèn một nút mới có giá trị X vào cây.
2. Viết hàm **Input()** để nhập vào một số nút cho cây nhị phân.
3. Viết hàm **Display()** để duyệt cây và in các giá trị của các nút ra màn hình theo thứ tự LNR.
4. Viết hàm **int Sum()** để tính tổng giá trị các nút của cây nhị phân.
5. Viết hàm **int LeafCount()** để đếm xem cây có bao nhiêu nút lá.
6. Viết hàm **int Hight()** để tính chiều cao của cây.
7. Viết hàm **Delete(int X)** để xoá nút trên cây nhị phân chứa giá trị X.

Bài 6.2: Người ta dùng một cây nhị phân tìm kiếm để lưu trữ dãy các số nguyên mà mỗi nút của cây gồm 4 trường sau: 2 trường liên kết TRAI và PHAI; 2 trường còn lại là GIATRI và SOLUONG. Trường GIATRI là trường khóa lưu giá trị nguyên trong dãy, trường SOLUONG lưu số lần xuất hiện của giá trị khóa tương ứng trong dãy.

Ví dụ: Với dãy 50, 30, 60, 10, 30, 20, 10, 90, 60, 30 sẽ được lưu trữ như sau:



1. Hãy xây dựng cấu trúc dữ liệu cho cây nhị phân nói trên.
2. Viết hàm để bổ sung một giá trị X vào cây nhị phân: Nếu X chưa có trên cây, tạo nút mới để bổ sung X vào, ngược lại, chỉ cần tăng trường SOLUONG lên 1.
3. Viết hàm nhập vào một cây BST (lấy số liệu ở ví dụ trên).
4. Viết hàm để duyệt cây theo thứ tự giữa với trường khóa là GIATRI.
5. Viết hàm để duyệt cây theo trường khóa GIATRI sao cho các giá trị in ra theo thứ tự giảm dần.

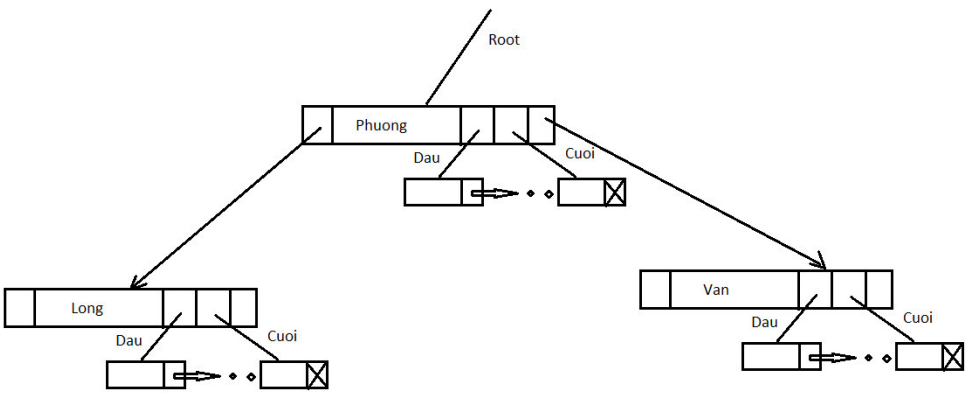
Bài 6.3: Quản lý Thư viện

Người ta quản lý các thông tin của một thư viện dưới dạng cây BST với khóa là *TenTG* (tên tác giả). Mỗi nút của cây là một cấu trúc

gồm trường *TenTG* và 4 trường con trỏ: 2 con trỏ *Trái* và *Phai* lần lượt trỏ tới các nút con trái và nút con phải, 2 con trỏ *Dau* và *Cuoi* lần lượt trỏ tới phần tử đầu tiên và cuối cùng của một danh sách liên kết đơn dùng để ghi nhận các sách có trong thư viện của tác giả đó. Mỗi phần tử của danh sách liên kết là một cấu trúc gồm 2 trường: *TenSach* (tên sách) và *Tieptheo* (chứa địa chỉ nút tiếp theo).

Người ta khai báo CTDL cho bài toán trên như sau:

```
struct SACH
{
  string TenSach;
  SACH *Tieptheo;
};
  typedef SACH* TroSach;
  struct TACGIA
{
  string TenTG;
  TroSach Dau, Cuoi;
  TACGIA *Trai, *Phai;
};
typedef TACGIA* TroTG;
```



1. Viết hàm **TroTG Search(string Name, TroTG Root)** để tìm đến nút trên cây BST có nút gốc được trỏ bởi con trỏ Root chứa tác giả có tên là *Name*. Nếu không tìm thấy, hàm trả về giá trị NULL.
2. Viết hàm **void Insert(string Title, string Name, TroTG &Root)** để bổ sung tác giả có tên *Name* với cuốn sách có tiêu đề *Title* vào cây BST có nút gốc được trỏ bởi con trỏ *Root* theo cách sau:
   - Nếu *Name* và *Title* đã có trong thư viện thì không làm gì nữa.
   - Nếu *Name* đã có và *Title* chưa có, bổ sung *Title* vào cuối danh sách liên kết đơn tương ứng với nút có *TenTG* = *Name*.
   - Nếu *Name* chưa có, bổ sung một nút mới vào cây BST với *TenGT* = *Name* và *TenSach* = *Title*.
3. Viết hàm **void Duyet(TroTG Root)** để hiển thị lên màn hình tất cả các tác giả và các tiêu đề sách của các tác giả đó.
4. Viết hàm **int Dem(TroTG Root)** để đếm số tác giả viết từ 2 cuốn sách trở lên.

Chương 7
BẢNG BĂM

Tóm tắt chương

- Tại sao lại tổ chức bảng băm (hash table)
- Hàm băm
- Xử lý va chạm trong bảng băm
- Xóa phần tử trong bảng băm

7.1. GIỚI THIỆU

Nếu dữ liệu được tổ chức dưới dạng mảng đã được sắp xếp, việc tìm kiếm một phần tử có thể mất thời gian là O(log n) với thuật toán tìm kiếm nhị phân.

Tuy nhiên với một mảng đã sắp xếp, việc chèn và xóa một phần tử được thực hiện với thời gian là O(n).

Nếu dữ liệu được tổ chức là cây tìm kiếm nhị phân cân bằng, việc chèn, tìm kiếm và xóa được thực hiện trong thời gian O(log n).

Như vậy, có cấu trúc dữ liệu nào để chèn, xóa và tìm kiếm hiệu quả hơn không? Câu trả lời là "Có", đó chính là bảng băm (hash table).

Cấu trúc dữ liệu bảng băm chỉ hỗ trợ một vài thao tác của cây BST.

"Băm" (Hashing) là kỹ thuật được sử dụng để thực hiện việc chèn, xóa và tìm kiếm với thời gian trung bình không đổi (O(1)).

Tuy nhiên, cấu trúc dữ liệu bảng băm không hiệu quả đối với các hoạt động yêu cầu lấy thông tin bất kỳ thứ tự nào giữa các phần tử, chẳng hạn như tìm giá trị bé nhất, tìm giá trị lớn nhất hoặc in toàn bộ bảng theo thứ tự đã sắp xếp.

7.2. HÀM BĂM

Cấu trúc bảng băm lý tưởng chỉ đơn thuần là một mảng có kích thước cố định, chứa các ***mục (items)*** dữ liệu.

Một mục lưu trữ cần phải có một ***khóa (key)***, được sử dụng để tính toán giá trị chỉ mục cho mục đó. Khóa có thể là một số nguyên, một chuỗi,...

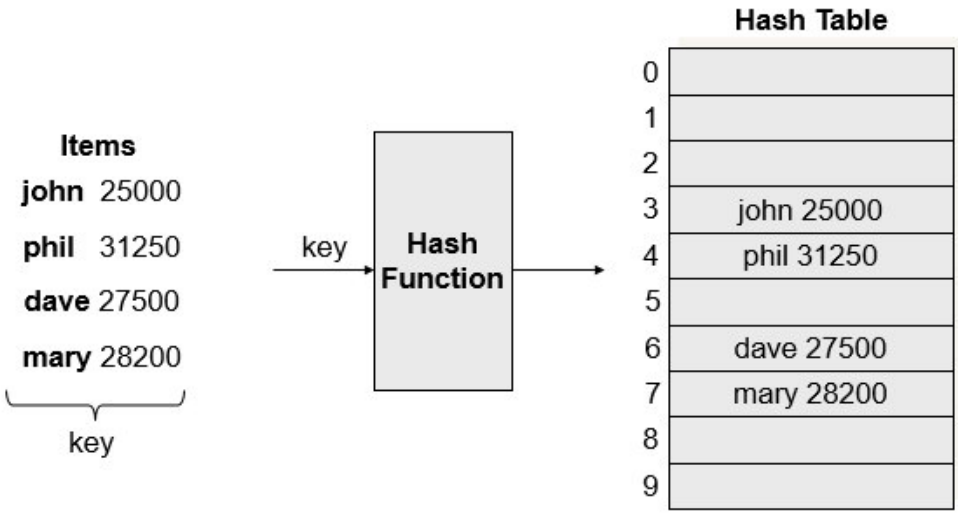
Kích thước của mảng (bảng băm) là ***TableSize***.

Các mục được lưu trữ trong bảng băm được lập chỉ mục theo các giá trị từ ***0 đến TableSize - 1***.

Mỗi khóa được ***ánh xạ*** thành một số trong phạm vi từ 0 đến ***TableSize - 1***.

Ánh xạ này được gọi là một ***hàm băm (hash function)***.

Ví dụ 7.1:



Hình 7.1: *Ví dụ về hàm băm*

Đối với hàm băm:

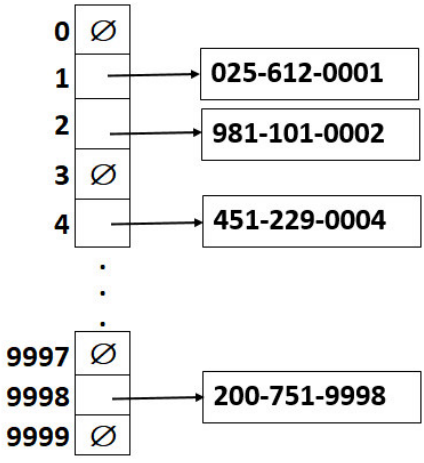
- Phải đơn giản để tính toán.
- Phải phân phối đều các khóa giữa các ô.

Một số vấn đề thường gặp khi thiết kế hàm băm:

- Các khóa có thể không phải là số.
- Số lượng khóa có thể lớn hơn nhiều so với không gian chỉ mục trong trong bảng băm.
- Các khóa khác nhau có thể ánh xạ vào cùng một vị trí.
  - Hàm băm không phải là ánh xạ một - một, điều này dẫn đến ***va chạm (collision)***.
  - Nếu có quá nhiều va chạm, hiệu suất của bảng băm sẽ bị ảnh hưởng nghiêm trọng.
- Nếu khóa là số nguyên, chiến lược chung để tạo hàm băm là ***Key mod TableSize***. Trừ khi khóa xảy ra một số thuộc tính không mong muốn (ví dụ: tất cả các khóa đều kết thúc bằng 0 và hàm băm sử dụng ***mod 10***).
- Nếu các khóa là chuỗi, hàm băm cần được quan tâm nhiều hơn, trước tiên là phải chuyển nó thành một giá trị số.

Ví dụ 7.2: Thiết kế bảng băm với một ánh xạ lưu trữ các mục dữ liệu là (ID, Name), trong đó ID là một số nguyên dương có mười chữ số.

Bảng băm sử dụng một mảng có kích thước N = 10.000 và hàm băm **h(x) = bốn chữ số cuối của x** (Hình 7.2).



Hình 7.2: *Bảng băm với kích thước N = 10.000*

Chúng ta luôn mong muốn có một hàm băm dễ tính toán và giảm thiểu số lần va chạm.

Các hàm băm không được thiên vị:

- Nghĩa là, nếu chọn ngẫu nhiên một khóa x, từ không gian khóa, xác suất để f(x) = i là 1/M, trong đó M là kích thước của bảng băm.
- Ta gọi hàm băm thỏa mãn thuộc tính không thiên vị là hàm băm thống nhất.

7.2.1. Hàm băm chia

Phép chia $\mathbf{h_D(x) = x \bmod M}$:

- Sử dụng toán tử chia lấy dư (mod).
- Chia khóa x cho M và sử dụng phần còn lại làm chỉ số băm cho x.
- Điều này cung cấp cho các chỉ mục nằm trong khoảng từ 0 đến M - 1, trong đó M = kích thước bảng băm.
- Sự lựa chọn M rất quan trọng:
  - Nếu M chia hết cho 2, khóa lẻ cho chỉ số lẻ và khóa chẵn cho chỉ số chẵn. (thiên vị!)
  - Nếu M là lũy thừa của 2, tức là $M = 2^p$, h(k) chỉ là p bit bậc thấp nhất của k. (thiên vị!)
  - Nếu M = pH, các khóa trong tập hợp {H, 2H, 3H,…, (p-1)H, pH, (p + 1) H,…, kH,…} ánh xạ tới vị trí p {H, 2H, 3H,…, (p-1) H, 0} (thiên vị!)
  - Một lựa chọn tốt cho M: M là ***số nguyên tố*** sao cho M không chia hết rk ± a, với k nhỏ và a.

7.2.2. Hàm băm xếp

Cách xếp (folding):

- Phân chia khóa x thành nhiều phần.

- Tất cả các phần ngoại trừ phần cuối cùng có cùng độ dài.
- Cộng các phần lại với nhau để thu được giá trị y, chỉ số băm khi đó là **h(x) = y mod M**.

Có hai khả năng (chia x thành nhiều phần):

- Dịch chuyển xếp (Shift folding): Dịch chuyển tất cả các phần ngoại trừ phần cuối cùng, sao cho bít cuối của mỗi phần thẳng hàng với bít tương ứng của phần cuối cùng. Giả sử x = 723 203 541 213 24 thì $x_1 = 723$, $x_2 = 203$, $x_3 = 541$, $x_4 = 213$, $x_5 = 24$, index = $(x_1 + x_2 + x_3 + x_4 + x_5)$ % 1000 = 1704 % 1000 = 704.
- Xếp theo ranh giới: đảo ngược mọi phân vùng khác, trừ phần cuối cùng (nếu phần đảo lớn hơn nó) trước khi cộng vào chỉ số index. Ví dụ: $x_1 = 723$, $\mathbf{x_2 = 302}$, $x_3 = 541$, $\mathbf{x_4 = 312}$, $x_5 = 24$, index = 1902 % 1000 = 902.

7.2.3. Các hàm băm khác

Phương pháp mid-square

Có thể lấy giá trị khóa bình phương và phần giữa của kết quả được sử dụng làm địa chỉ index.

Ví dụ: key = 3121^2 = 9 7**40 6**41 thì index = 406 % TableSize.

Vì các bít ở giữa của hình vuông thường phụ thuộc vào tất cả các ký tự trong một khóa, nên có khả năng cao là các khóa khác nhau sẽ tạo ra các chỉ số băm khác nhau.

Phương pháp trích xuất (extraction)

Trong phương pháp trích xuất, chỉ một phần của khóa được sử dụng để tính toán địa chỉ:

Ví dụ: 123-45-6789 có thể trích xuất 2 ký tự đầu và cuối 12345-6789 thì index = 1289 mod TableSize.

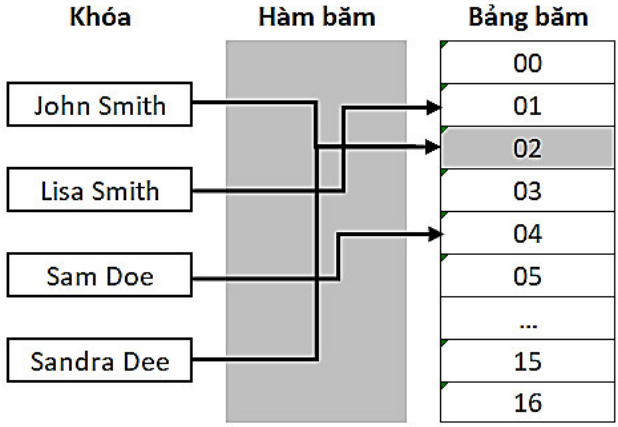
Phép biến đổi cơ số (radix transformation)

Khóa K được biến đổi thành một cơ số khác, K được biểu diễn trong một hệ thống số sử dụng một cơ số khác.

Ví dụ: $345_{10} = 423_9$ thì index = 423 mod TableSize.

7.3. XỬ LÝ VA CHẠM

Va chạm (collision) xảy ra khi các phần tử khác nhau được ánh xạ vào cùng một ô (Hình 7.3).



Hình 7.3: *Tình huống va chạm*

7.3.1. Xử lý va chạm bằng địa chỉ mở

Trong phương pháp địa chỉ mở (open addressing), khi một khóa x va chạm với một khóa khác, xung đột được giải quyết bằng cách tìm một mục có sẵn khác với vị trí (địa chỉ) mà khóa xung đột ban đầu được băm. Do đó, nếu vị trí k = h(x) được sử dụng, các vị trí sau được thử:

$k = h_i(x) = h(x) + p(i) \bmod M$, i = 1, 2, ... (M = TableSize)

- Phương pháp đơn giản nhất là **thăm dò tuyến tính**, với p(i) = i và đối với đầu dò thứ i, vị trí cần thử là (h(x) + i) mod M, i = 1,2,…
- Phương pháp **thăm dò bậc hai (quadratic)**: $p(i) = i^2$ do đó vị trí cần thử là $(h(x) + i^2)$ % M, i = 1, 2,…

Thuật toán tìm kiếm một mục trong bảng băm sử dụng phép thăm dò tuyến tính:

Get(k): Xét bảng băm A sử dụng thăm dò tuyến tính

- Bắt đầu tại ô h(k)
- Thăm dò các vị trí liên tiếp cho đến khi một trong những điều sau xảy ra:
  - Một mục có khóa k được tìm thấy, hoặc
  - Một ô trống được tìm thấy, hoặc
  - N ô đã được thăm dò không thành công

Algorithm *get(k)*

```
i = h(k);
p = 0;
do
{
  c = A[i];
  if(c==null) return null;
  else if(c.key = k)
    return c.element();
      else
      {
        i = (i + 1)% N;
        P = p + 1;
      }
}
while (p != N);
return null;
```

7.3.2. Các yếu tố ảnh hưởng đến hiệu suất tìm kiếm

Chất lượng của hàm băm phụ thuộc vào:

- Dữ liệu thực tế
- Chiến lược giải quyết va chạm

- Hệ số tải của bảng băm: N/Tsize, hệ số tải càng thấp thì hiệu suất tìm kiếm càng tốt.

Ví dụ 7.3: Thăm dò bậc hai.

Nếu vị trí k = h(x) được sử dụng, các vị trí sau sẽ được thử:

$k = h_i(x) = h(x) + i^2 \bmod M$, i = 1, 2, ... (M = Tsize)

h (x) = x % 10

Chèn các khóa 89, 18, 49, 58, 69 theo thứ tự này

0	1	2	3	4	5	6	7	8	9
49		58	69					18	89

Ưu điểm và nhược điểm của thăm dò bậc hai

Một vấn đề đối với thăm dò bậc hai là các chuỗi thăm dò không thăm dò tất cả các vị trí trong bảng. Ví dụ, nếu M = 11, k = h(x) = x % 11. Sau đó, đối với các khóa x, trong đó h(x) = 3 và xảy ra va chạm, chỉ các vị trí 3, 4, 7, 1, 8, 6 được thăm dò.

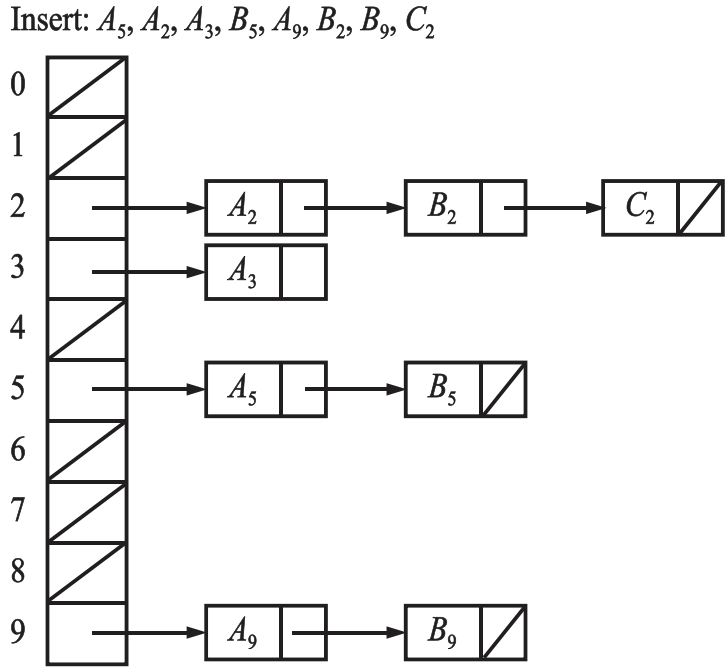
Khi M là số nguyên tố, có thể đảm bảo qua định lý sau:

Định lý 7.1: *Nếu M là số nguyên tố và bảng trống ít nhất một nửa, thăm dò bậc hai sẽ luôn tìm thấy một vị trí trống. Hơn nữa, không có địa điểm nào được kiểm tra hai lần.*

7.3.3. Xử lý va chạm bằng phương pháp chuỗi

Các khóa không phải được lưu trữ trong bảng chính. Trong chuỗi (chaining), mỗi vị trí của bảng được kết nối với một danh sách liên kết lưu trữ các khóa.

Phương pháp này được gọi là chuỗi riêng và một bảng tham chiếu (con trỏ) là bảng phân tán. Trong phương pháp này, bảng không bao giờ tràn vì danh sách liên kết có thể mở rộng.



Hình 7.4: *Trong chuỗi, các khóa va chạm được đưa vào cùng một danh sách liên kết*

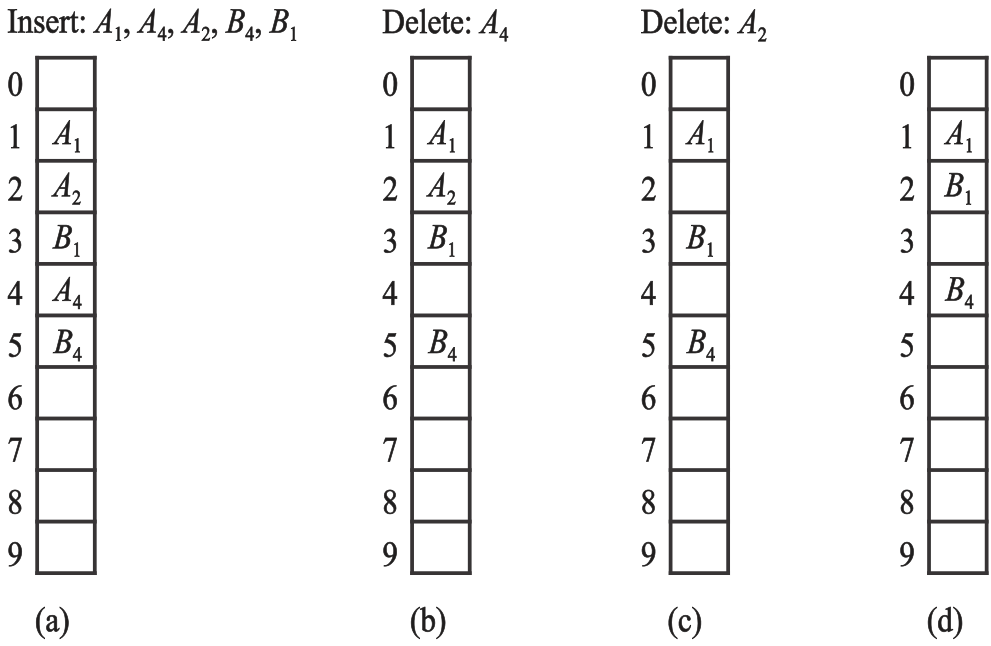
7.4. XÓA PHẦN TỬ TRONG BẢNG BĂM

Xét bảng băm trong đó các khóa được lưu trữ bằng cách sử dụng thăm dò tuyến tính. Giả sử cần xóa khóa A4 và sau đó cố gắng tìm B4. Vì khi tìm kiếm B ta băm nó đến vị trí 4 và thấy vị trí này trống và kết luận không tìm thấy B4 (điều này không đúng).

Để tránh tình trạng này, chỉ cần đánh dấu các vị trí đã xóa. Khi chèn phần tử mới vào vị trí này, cần cập nhật lại thông tin cho phần tử mới. Khi có quá nhiều phần tử đã xóa được đánh dấu trong bảng, bảng sẽ được làm mới (d).

Nếu hàm băm h biến đổi các khóa khác nhau thành các số khác nhau, nó được gọi là ***hàm băm hoàn hảo (perfect hash function)***.

Nếu một hàm chỉ yêu cầu bao nhiêu ô trong bảng bằng số dữ liệu để không có ô trống nào còn lại sau khi hoàn tất quá trình băm, hàm đó được gọi là ***hàm băm hoàn hảo tối thiểu.***



Hình 7.5: *Tìm kiếm tuyến tính trong trường hợp chèn và xóa khóa*

BÀI TẬP CHƯƠNG 7

Bài 7.1: Giải thích việc chèn các khóa 5, 28, 19, 15, 20, 33, 12, 17, 10 vào một bảng băm có các xung đột được giải quyết bởi kỹ thuật kết nối. Cho bảng có 9 vị trí, và cho hàm băm là h(k) = k mod 9.

Bài 7.2: Xét tiến trình chèn các khóa 10, 22, 31, 4, 15, 28, 17, 88, 59 vào một bảng băm có chiều dài $m = 11$ dùng kỹ thuật định địa chỉ mở với hàm băm sơ cấp $h'(k) = k \mod m$. Minh họa kết quả của việc chèn các khóa này bằng kỹ thuật thăm dò tuyến tính, thăm dò bật hai với $c_1=1$ và $c_2=3$, và dùng kỹ thuật băm kép với $h_2(k) = 1 + (k \mod (m-1))$.

Bài 7.3: Quản lý Thư viện

Người ta quản lý các thông tin của một thư viện dưới dạng bảng băm. Mỗi tên tác giả ứng với mã băm từ 0 đến 25 (tương ứng với 'A' đến 'Z'). Mỗi mã cho cách truy nhập tới một **danh sách liên kết đơn đã được sắp xếp** gồm tất cả tên tác giả có cùng mã.

Mỗi nút của danh sách liên kết gồm 4 trường: *TenTG*, 2 trường con trỏ *First* và *Last* lần lượt trỏ tới phần tử đầu tiên và cuối cùng của một danh sách liên kết đơn dùng để ghi nhận các sách của tác giả đó, *Next* là con trỏ trỏ đến nút tiếp theo của danh sách tác giả.

Người ta khai báo CTDL cho bài toán trên như sau:

```
struct SACH
{
  string TenSach;
  SACH *Link;
};
  typedef SACH* TroSach;
  struct TACGIA
{
  string TenTG;
  TroSach First, Last;
  TACGIA *Next;
};
  typedef TACGIA* TroTG;
  TroTG Thuvien[26];
```

Giả sử các tác giả có tên khác nhau và các cuốn sách của cùng tác giả cũng có tên khác nhau.

1. Viết hàm **int MaTG(string Name)** để trả về mã của tác giả trong bảng băm có tên Name.
2. Viết hàm **TroTG TimTG(string Name, TroTG TG)** để tìm đến nút trong danh sách *TG* chứa tác giả có tên là *Name*. Nếu không tìm thấy, hàm trả về giá trị NULL.
3. Viết hàm **TroSach TimSach(string TenSach,TroSach First)** để kiểm tra cuốn sách với tựa đề *TenSach* có trong danh sách First hay không. Nếu có trả về địa chỉ nút tìm được, ngược lại trả về NULL.

4. Viết hàm **int SoSachTG(string Name, TroTG Thuvien[])** để đếm số đầu sách của tác giả có tên *Name* trong thư viện.
5. Viết hàm **void Insert(string Title, string Name, TroTG Thuvien[])** để bổ sung tác giả có tên *Name* với cuốn sách có tiêu đề *Title* vào thư viện theo cách sau:
   - Nếu *Name* và *Title* đã có trong thư viện: không làm gì nữa.
   - Nếu *Name* đã có và *Title* chưa có: bổ sung *Title* vào cuối danh sách liên kết đơn tương ứng với nút có *TenTG = Name*.
   - Nếu *Name* chưa có: bổ sung một nút mới vào thư viện với *TenGT = Name* và *TenSach = Title*.
6. Viết hàm **void Xem(TroTG Thuvien[])** để liệt kê tất cả các tác giả và các sách của họ ra màn hình.
7. Viết hàm **int DemTG(TroTG Thuvien[])** để đếm số tác giả có trong thư viện.
8. Viết hàm **int DemSach(TroTG Thuvien[])** để đếm số đầu sách có trong thư viện.
9. Viết hàm **void XoaSach(string TenSach,TroSach &First,TroSach &Last)** để xóa cuốn sách với tựa đề *TenSach* có trong danh sách được trỏ bởi First và Last.
10. Viết hàm **void XoaTG(string Name,TroTG &TG)** để xóa tác giả với tên *Name* khỏi danh sách *TG*.
11. Viết hàm **void Delete(string TenSach, string Name, TroTG Thuvien[])** để xóa cuốn sách với tựa đề *TenSach* của tác giả có tên *Name* trong thư viện. Nếu tác giả không còn cuốn sách nào nữa thì xóa tên tác giả khỏi thư viện.

.

Phụ lục A
CÀI ĐẶT CÁC THAO TÁC TRÊN DANH SÁCH LIÊN KẾT

A1. DANH SÁCH LIÊN KẾT ĐƠN

```
#include <bits/stdc++.h>
using namespace std;
class SingleLinkList
{
    struct NODE
    {
      int Data;
      NODE *Next;
    };
    NODE *First, *Last;
  public:
    SingleLinkList() //Khoi tao danh sach rong
    {
      First = Last = NULL;
    }
    void View() //Duyet danh sach
    {
      NODE *p = First;
      while(p)
      {
        cout<<p->Data<<" ";
        p = p->Next;
      }
      cout<<endl;
```

```
}
void FirstInsert(int x)
{
  //B1: Tao nut moi p chua x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chen p vao dau danh sach
  if(First==NULL) //Danh sach rong
  {
    p->Next = NULL;
    First = Last = p;
  }
  else //Danh sach khong rong
  {
    p->Next = First; //1
    First = p;       //2
  }
}
void LastInsert(int x)
{
  //B1: Tao nut moi p chua x
  NODE *p = new(NODE);
  p->Data = x;
  p->Next = NULL;
  //B2: Chen p vao cuoi danh sach
  if(Last==NULL) //Danh sach rong
    First = Last = p;
  else //Danh sach khong rong
  {
    Last->Next = p; //1
    Last = p;       //2
```

```
  }
}
void Insert(int x,NODE *q)
{
  //B1: Tao nut moi p chua x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chen p vao sau q
  p->Next = q->Next; //1
  q->Next = p;       //2
}
NODE *GetNode(int n)
{
  NODE *p = First;
  for(int i=1;i<n;i++) p = p->Next;
  return p;
}
void FirstDel()
{
  NODE *p = First;
  if(First==Last) //Danh sach chi co 1 nut
    First = Last = NULL;
  else
    First = p->Next;
  delete p;
}
void LastDel()
{
  NODE *p = Last;
  if(First==Last) //Danh sach chi co 1 nut
    First = Last = NULL;
```

```
      else
      {
        //Tim den nut q dung truoc nut Last
        NODE *q = First;
        while(q->Next!=Last) q = q->Next;
        //q la nut cuoi danh sach
        q->Next = NULL;
        Last = q;
      }
      delete p;
    }
    void Del(NODE *T) //xoa nut dung sau T
    {
      NODE *p = T->Next; //nut can xoa
      T->Next = p->Next;
      delete p;
    }
};
void Nhap(int &n,SingleLinkList &L)
{
  cout<<"So phan tu cua danh sach: ";
  cin>>n;
  for(int i=1;i<=n;i++)
  {
    int x;
    cin>>x;
    L.LastInsert(x);
  }
}
int main()
{
```

```
  SingleLinkList L;
  int n;
  Nhap(n,L);
  L.View();
  L.Del(L.GetNode(2));
  L.View();
}
```

A2. DANH SÁCH LIÊN KẾT ĐƠN NỐI VÒNG

```
#include <bits/stdc++.h>
using namespace std;
class CircleLink
{
    struct NODE
    {
      int Data;
      NODE *Next;
    };
  public:
    NODE *First;

    CircleLink()//Khoi tao danh sach rong
    {
      First = NULL;
    }

    void View()
    {
      if(First)
      {
        NODE *p = First;
```

```
    do
    {
      cout<<p->Data<<" ";
      p = p->Next;
    }
    while(p!=First);
  }
  cout<<endl;
}
NODE* Search(int x)
{
  if(First)
  {
    NODE *p = First;
    do
    {
      if(p->Data!=x);
        p = p->Next;
    }
    while(p->Data!=x && p!=First);
    if(p!=First) return p;
  }
  return NULL;
}
void FirstInsert(int x)
{
  //B1: tao nut moi p chua x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chen
  if(First==NULL)
```

```
  {
    First = p;             //1
    First->Next = First;   //2
  }
  else
  {
    //B3: Tim den nut cuoi danh sach q
    NODE *q = First;
    while (q->Next != First) q = q->Next;
    //B4: Chen p vao dau danh sach
    p->Next = First;
    First = p;
    q->Next = First; //noi vong
  }
}

void LastInsert(int x)
{
  //B1: tao nut moi p chua x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chen
  if(First==NULL)
  {
    First = p;             //1
    First->Next = First;   //2
  }
  else
  {
    //B3: Tim den nut cuoi danh sach q
    NODE *q = First;
```

```
    while (q->Next != First) q = q->Next;
    //B4: Chen p vao cuoi danh sach
    q->Next = p;
    p->Next = First; //noi vong
  }
}
void Insert(int x,NODE *T)
{
  //B1: tao nut moi p chua x
  NODE *p = new(NODE);
  p->Data = x;
  //B2: Chen p vao sau T
  p->Next = T->Next;      //1
  T->Next = p;            //2
}

void FirstDel()
{
  NODE *p = First;
  if(First->Next==First) //danh sach co 1 nut
    First = NULL;
  else
  {
    //Tim toi nut cuoi danh sach
    NODE *q = First;
    while(q->Next!=First) q = q->Next;
    //cat nut p ra khoi danh sach
    First = p->Next;
    q->Next = First;
  }
```

```
      delete p;
    }
    void Delete(NODE *q)
    {
      NODE *p = q->Next;
      q->Next = p->Next;
      delete p;
    }
};
void Nhap(int &n,CircleLink &L)
{
  cin>>n;
  for(int i=1;i<=n;i++)
  {
    int x;
    cin>>x;
    L.LastInsert(x);
  }
}
int main()
{
  CircleLink L;
  int n;
  Nhap(n,L);
  L.View();
  L.FirstDel();
  L.Delete(L.First);
  L.View();
}
```

A3. DANH SÁCH LIÊN KẾT KÉP

```
#include <bits/stdc++.h>
using namespace std;
class DoubleLinkList
{
    struct NODE
    {
      int Data;
      NODE *Next, *prev;
    };
    NODE *First, *Last;
  public:
    DoubleLinkList() //Khoi tao danh sach rong
    {
      First = Last = NULL;
    }
    void View() //Duyet danh sach
    {
      NODE *p = First;
      while(p)
      {
        cout<<p->Data<<" ";
        p = p->Next;
      }
      cout<<endl;
    }
    void LastInsert(int x)
    {
      //B1: Tao nut moi p chua x
      NODE *p = new(NODE);
      p->Data = x;
```

```
  p->Next = NULL;
  p->prev = NULL;
  //B2: Chen p vao cuoi danh sach
  if(Last==NULL) //Danh sach rong
    First = Last = p;
  else
  {
    Last->Next = p; //1
    p->prev = Last; //2
    Last = p;       //3
  }
}
void FirstInsert(int x)
{

}

void Input()
{
  int n;
  cout<<"So phan tu cua danh sach: ";
  cin>>n;
  for(int i=1;i<=n;i++)
  {
    int x;
    cin>>x;
    LastInsert(x);
  }
}
void Insert(NODE *T,int x)
{
```

```
  //B1: Tao nut moi
  NODE *p = new(NODE);
  p->Data = x;
  //B2: chen p vao sau T
  p->Next = T->Next; //1
  p->prev = T;       //2
  T->Next->prev = p; //3
  T->Next = p;       //4
}
NODE *GetNode(int n)
{
  NODE *p = First;
  for(int i=1;i<n;i++) p = p->Next;
  return p;
}

void FirstDel()
{
  //B1: dinh vi den nut can xoa
  NODE *p = First;
  //B2: Xu ly truoc khi xoa
  if(First==Last) //danh sach chi co 1 nut
    First = Last = NULL;
  else
  {
    First = p->Next; //First qua nut ke tiep
    First->prev = NULL;
  }
  //B3: Xoa
  delete p;
}
```

```
void LastDel()
{
  //B1: dinh vi den nut can xoa
  NODE *p = Last;
  //B2: Xu ly truoc khi xoa
  if(First==Last) //danh sach chi co 1 nut
    First = Last = NULL;
  else
  {
    Last = p->prev; //Last qua nut ke tiep
    Last->Next = NULL;
  }
  //B3: Xoa
  delete p;
}

void Del(NODE *T) //xoa nut dung sau nut T
{
  //B1: dinh vi den nut can xoa
  NODE *p = T->Next;
  //B2: xu ly truoc khi xoa
  T->Next = p->Next; //1
  p->Next->prev = T; //2
  //B3: xoa p
  delete p;
}
NODE *FirstNode()
{
  return First;
}
```

```
    NODE *LastNode()
    {
      return Last;
    }
};
int main()
{
  DoubleLinkList L;
  L.Input();
  L.View();
}
```

Phụ lục B
CÀI ĐẶT STACK

B1. CÀI ĐẶT STACK BẰNG DANH SÁCH LIÊN KẾT

```
#include <iostream>
#include <conio.h>
using namespace std;
class Stack
{
    struct NODE
    {
     int Data;
     NODE *Next;
    };
    NODE *top;
   public:
    Stack()
    {
      top = NULL;
    }

    void Push(int x)
    {
      //B1: Tao nut moi p chua x
      NODE *p = new(NODE);
      p->Data = x;
      //B2: Chen p vao dau danh sach
      p->Next = top; //1
      top = p;       //2
    }
```

```
    int Pop()
    {
      NODE *p = top;
        top = p->Next;
      int x = p->Data;
      delete p;
      return x;
    }
    bool isEmpty()
    {
      return top==NULL;
    }
};
void Bin(int n)
{
  //B1: Khoi tao stack
  Stack S;
  //B2: Dua cac so du n%2 vao stack
  while(n>0)
  {
     S.Push(n%2);
     n=n/2;
  }
  //B3: Lay cac phan tu trong stack in ra man hinh
  while(!S.isEmpty())
  {
    cout<<S.Pop();
  }
}
int main()
{
  Bin(13);
}
```

B2. TÍNH GIÁ TRỊ BIỂU THỨC SỐ HỌC

```
#include<stack>
#include<iostream>
#include<string>
#include <stdlib.h>
using namespace std;
bool Sosanh(string x,string y)
{
      int a,b;
      if(x=="(") a=0;    if(y=="(") b=0;
      if(x=="-") a=1;    if(y=="-") b=1;
      if(x=="+") a=1;    if(y=="+") b=1;
      if(x=="/") a=2;    if(y=="/") b=2;
      if(x=="*") a=2;    if(y=="*") b=2;
      return (a>=b);
}
void toArrayString(string st, int &n,string a[])
//Tách chuỗi st thành mảng các chuỗi a[]
{
  n=0;
  a[0]="(";
  while(!st.empty())
  {
    string temp = "";
    if(st[0]=='+'|| st[0]=='-'|| st[0]=='*'||
      st[0]=='/'|| st[0]=='(' || st[0]==')')
    {
      temp = temp + st[0];
      st.erase(0,1);
      a[++n] = temp;
    }
```

```
    else
    {
      int i=0;
      while((i<st.length())&&((st[i]>='0')
        &&(st[i]<='9')||(st[i]=='.')))
      {
        temp = temp + st[i];
        i++;
      }
      st.erase(0,temp.length());
      a[++n] = temp;
    }
  }
  a[++n]=")";
}
string Hauto(string infix)
//Chuyển biểu thức trung tố infix sang dạng hậu tố
{
  //B1: khoi tao
  string st[1000];
  int n;
  toArrayString(infix,n,st);
  stack<string> S;
    string posfix="", Top;
    //Buoc 2:
    for(int i=0;i<=n;i++)
    {
      string X=st[i];
        if(X=="(") S.push(X);
          else if(X==")")
          {
```

```
            string Token=S.top();
            S.pop();
            while(Token!="(")
            {
              posfix = posfix + Token + ' ';
              Token=S.top();
              S.pop();
            }
      }
else
       if(isdigit(X[0])) //toan hang
            posfix = posfix + X + ' ';
       else //toan tu
       {
            if(S.empty()) S.push(X);
            else
            {

Top=S.top();

while(Sosanh(Top,X))

{

S.pop();

posfix = posfix + Top + ' ';

Top=S.top();
          }
          S.push(X);
            }
```

```
            }
        }
  //Buoc 3:
    while(!S.empty())
    {
      posfix = posfix + S.top() + ' ';
      S.pop();
    }
    return posfix;
}
double Value(string posfix)
//Tính giá trị biểu thức hậu tố posfix
{
  stack<float> S;
  while(!posfix.empty())
  {
    string temp = posfix.substr(0,posfix.find(" "));
    posfix.erase(0,temp.length()+1);
    if(isdigit(temp[0]))
    {
      float x = atof(temp.c_str());
      S.push(x);
    }
    else
    {
      float b = S.top();
      S.pop();
      float a = S.top();
      S.pop();
      if(temp=="+") S.push(a+b);
      if(temp=="-") S.push(a-b);
```

```
      if(temp=="*") S.push(a*b);
      if(temp=="/")
S.push(a/b);
    }
  }
  return S.top();
}
int main()
{
   string s;
   cout<<"Nhap bieu thuc: "; cin>>s;
   string st = Hauto(s);
   cout<<"Chuyen sang hau to : "<<st<<endl;
   cout<<"Gia tri bieu thuc: "<<Value(st);
}
```

Phụ lục C
CÀI ĐẶT HÀNG ĐỢI

C1. CÀI ĐẶT HÀNG ĐỢI BẰNG MẢNG

```
#include <iostream>
using namespace std;
class Queue
{
    #define MAX 1000
    int A[MAX];
    int front,rear;
  public:
    Queue()
    {
      front = rear = 0;
    }
    bool isEmpty()
    {
      return front == 0;
    }
    bool isFull()
    {
      return (rear%MAX + 1) == front;
    }
    void Push(int x)
    {
      if(rear%MAX + 1 == front)
        cout<<"Queue is full";
      else
```

```
    {
      rear = rear%MAX + 1;
      A[rear] = x;
      if(front==0)
        front = 1; //do ban dau Queue rong
    }
  }
  int Pop()
  {
    int x = A[front];
    if(front==rear)
      front = rear = 0;
    else
      front = front%MAX + 1;
    return x;
  }
};
void Daoso(int n)
{
  //Khoi tao Queue
  Queue Q;
  //Dua cac so du n%10 vao Queue
  while(n>0)
  {
     Q.Push(n%10);
     n=n/10;
  }
  //Lay cac phan tu trong Queue in ra man hinh
  while(!Q.isEmpty())
  {
    cout<<Q.Pop();
```

```
  }
}
int main()
{
  Daoso(12345);
}
```

C2. CÀI ĐẶT HÀNG ĐỢI BẰNG DANH SÁCH LIÊN KẾT ĐƠN

```
#include <iostream>
using namespace std;
class Queue
{
    struct NODE
    {
        int Data;
        NODE *Next;
    };
    NODE *front, *rear;
  public:
    Queue()
    {
      front = rear = NULL;
    }
    bool isEmpty()
    {
      return front == NULL;
    }
    void Push(int x)
    {
      //B1: Tao nut moi p chua x
      NODE *p = new(NODE);
```

```
  p->Data = x;
  //B2: Chen p vao dau danh sach
  if(front==NULL) //Danh sach rong
  {
    p->Next = NULL;
    front = rear = p;
  }
  else //Danh sach khong rong
  {
    p->Next = front; //1
    front = p;       //2
  }
}
int Pop()
{
  NODE *p = rear;
  if(front == rear) //Danh sach chi co 1 nut
    front = rear = NULL;
  else
  {
    //Tim den nut q dung truoc nut rear
    NODE *q = front;
    while(q->Next!=rear) q = q->Next;
    //q la nut cuoi danh sach
    q->Next = NULL;
    rear = q;
  }
  int x = p->Data;
  delete p;
  return x;
}
```

```
};
void Daoso(int n)
{
  //Khoi tao queue
  Queue Q;
  //Dua cac so du n%10 vao Queue
  while(n>0)
  {
     Q.Push(n%10);
     n=n/10;
  }
  //Lay cac phan tu trong Queue in ra man hinh
  while(!Q.isEmpty())
  {
    cout<<Q.Pop();
  }
}
int main()
{
  Daoso(12345);
}
```

C3. CÀI ĐẶT HÀNG ĐỢI BẰNG DANH SÁCH LIÊN KẾT KÉP

```
#include <iostream>
using namespace std;
class Queue
{
    struct NODE
    {
        int Data;
        NODE *Next, *prev;
```

```
  };
  NODE *front, *rear;
public:
  Queue()
  {
    front = rear = NULL;
  }
  bool isEmpty()
  {
    return front == NULL;
  }
  void Push(int x)
  {
    //B1: Tao nut moi p chua x
    NODE *p = new(NODE);
    p->Data = x;
    p->Next = NULL;
    p->prev = NULL;
    //B2: Chen p vao dau danh sach
    if(front==NULL) //Danh sach rong
      front = rear = p;
    else //Danh sach khong rong
    {
      p->Next = front; //1
      front->prev = p; //2
      front = p;       //3
    }
  }
  int Pop()
  {
    NODE *p = rear;
```

```
      if(front == rear) //Danh sach chi co 1 nut
        front = rear = NULL;
      else
      {
        rear = p->prev; //rear qua nut ke tiep
        rear->Next = NULL;
      }
      int x = p->Data;
      delete p;
      return x;
    }
};
void Daoso(int n)
{
  //Khoi tao queue
  Queue Q;
  //Dua cac so du n%10 vao Queue
  while(n>0)
  {
     Q.Push(n%10);
     n=n/10;
  }
  //Lay cac phan tu trong Queue in ra man hinh
  while(!Q.isEmpty())
  {
    cout<<Q.Pop();
  }
}
int main()
{
  Daoso(12345);
}
```

TÀI LIỆU THAM KHẢO

[1]. Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, NXB Khoa học và kỹ thuật, 2002.

[2]. Đinh Mạnh Tường, *Cấu trúc dữ liệu và giải thuật*, NXB Khoa học và kỹ thuật, 2008.

[3]. Nguyễn Trung Trực, *Cấu trúc dữ liệu*, Trung tâm điện toán Trường Đại học Bách khoa TP Hồ Chí Minh, 2004.

[4]. Mark Allen Weiss, Data Structures & Algorithm Analysis in C++, *Pearson*; 4th edition, 2013.

[5]. Michael T. Goodrich, Roberto Tamassia andMichael H. Goldwasser, *Data structures and Algorithms in Java*, Addison Wiley, 2011.

[6]. Dinesh P. Mehta and Sartaj Sahni, *Data structures and Applications,* Chapman & Hall/CRC, 2005.

[7]. Richard Johnsonbaugh, *Discrete Mathematics*, Macmillan Publishing Company, New york 1992.

[8]. https://www.geeksforgeeks.org/sorting-algorithms/

Giáo trình Cấu trúc dữ liệu và giải thuật

TS. Phạm Anh Phương (Chủ biên), TS. Trần Văn Hưng
TS. Nguyễn Đình Lầu, TS. Quách Hải Thọ,
Trường Đại học Sư phạm - Đại học Đà Nẵng

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

Phòng 501, Nhà Điều hành ĐHQG-HCM, P. Linh Trung, TP Thủ Đức, TP.HCM.
ĐT: 028 62726361
E-mail: vnuhp@vnuhcm.edu.vn
Website: vnuhcmpress.edu.vn

Chịu trách nhiệm xuất bản và nội dung
PGS.TS NGUYỄN MINH TÂM

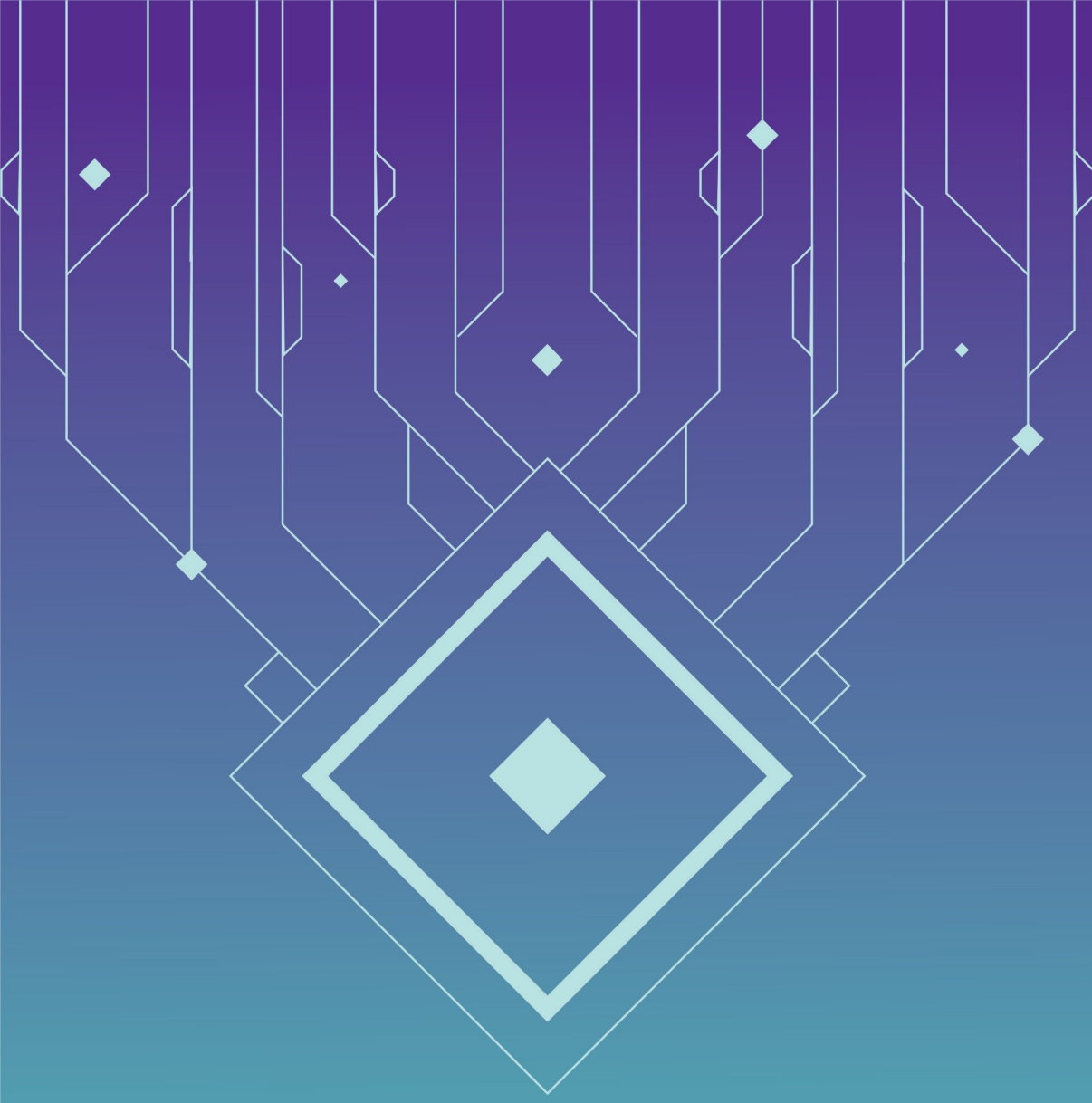
Biên tập
LÊ THỊ MINH HUỆ

Sửa bản in
ANH TÀI

Trình bày bìa
NGỌC TRẦN

Đối tác liên kết
TRƯỜNG ĐẠI HỌC SƯ PHẠM - ĐẠI HỌC ĐÀ NẴNG

Xuất bản lần thứ 1. Dung lượng: 6.80 MB, khổ 16 x 24 cm. Số XNĐKXB: 2009-2024/CXBIPH/14-20/ĐHQGTPHCM. QĐXB số: 110/QĐ-NXB-ĐT cấp ngày 17/6/2024. Đăng tải tại: **www.vnuhcmpress.edu.vn**. Nộp lưu chiểu: Năm 2024. ISBN: **978-604-479-657-4.**



ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC SƯ PHẠM

ISBN: 978-604-479-657-4
NXB ĐHQG-HCM
Giá: 70.000 đ