TRƯỜNG ĐẠI HỌC KỸ THUẬT CÔNG NGHIỆP

KHOA ĐIỆN TỬ

BỘ MÔN: CÔNG NGHỆ THÔNG TIN

☆☆☆

BÀI TIỂU LUẬN

MÔN HỌC

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Họ tên : **Nguyễn Văn Thứ**

Mã số sinh viên : **K225480106062**

Ngành học : **Kỹ thuật Máy tính**

Lớp : **K58KMT.K01**

Giảng viên hướng dẫn : **ThS. Nguyễn Thị Hương**

Thái Nguyên 2024

TRƯỜNG ĐẠI HỌC KTCN
KHOA ĐIỆN TỬ
BỘ MÔN CÔNG NGHỆ THÔNG TIN

CỘNG HOÀ XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc
--------o0o---------

PHIẾU GIAO BÀI TẬP TIỂU LUẬN

Môn: Cấu trúc dữ liệu và giải thuật

Họ tên sinh viên: Nguyễn Văn Thứ

MSSV: K225480106062

Lớp: K58KMT.K01 *Ngành: Kỹ thuật máy tính*

Giáo viên hướng dẫn: Ths. Nguyễn Thị Hương

Email: huongktpm@tnut.edu.vn

Chú ý: Kết quả chấm bài tập tiểu luận sẽ được sử dụng làm điểm đánh giá kết thúc học phần.

I. Nội dung bài tập tiểu luận

- Đề tài môn học đã giao cho sinh viên thực hiện
- Yêu cầu khảo sát, phân tích, thiết kế, cài đặt chương trình

II. Thời gian thực hiện: từ 08/10/2024 đến 18/11/2024.

III. Hình thức nộp

1. Báo cáo bản Word
2. File source code chương trình.

Thái Nguyên, ngày.....tháng.....năm 2024

GIÁO VIÊN HƯỚNG DẪN

(Ký và ghi rõ họ tên)

NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Thái Nguyên, ngày.....tháng.....năm 2024

GIÁO VIÊN HƯỚNG DẪN

(Ký và ghi rõ họ tên)

MỤC LỤC

LỜI NÓI ĐẦU

Kính thưa quý thầy cô và các bạn,

Trong thời đại công nghệ thông tin phát triển mạnh mẽ, việc nắm vững và áp dụng hiệu quả các cấu trúc dữ liệu cùng với giải thuật đóng vai trò quan trọng trong việc giải quyết các bài toán phức tạp. Môn học Cấu trúc dữ liệu và Giải thuật (CTDL&GT) không chỉ mang đến nền tảng kiến thức quan trọng mà còn tạo điều kiện để khám phá các ý tưởng sáng tạo trong lập trình và phát triển phần mềm.

Bài tiểu luận này được thực hiện với mục tiêu hệ thống hóa và trình bày rõ ràng những khái niệm cơ bản cũng như các ứng dụng thực tế của cấu trúc dữ liệu và giải thuật. Nội dung bài viết tập trung vào các cấu trúc dữ liệu như danh sách liên kết, cây, đồ thị và các giải thuật thông dụng như tìm kiếm, sắp xếp, tối ưu hóa. Đồng thời, em cũng phân tích độ phức tạp và đánh giá hiệu suất của từng giải thuật trong các ngữ cảnh khác nhau.

Trong quá trình nghiên cứu, em đã tham khảo nhiều tài liệu đáng tin cậy và cố gắng minh họa nội dung bằng các ví dụ thực tiễn, giúp người đọc dễ dàng tiếp cận và hiểu sâu hơn về các chủ đề được đề cập. Em hy vọng rằng bài tiểu luận sẽ cung cấp những kiến thức bổ ích, đồng thời trở thành nguồn tham khảo hữu ích cho các bạn sinh viên và những ai quan tâm đến lĩnh vực này.

Mặc dù em đã cố gắng hoàn thành bài viết một cách tốt nhất có thể, nhưng với kinh nghiệm còn hạn chế và kỹ năng lập trình chưa thực sự thành thạo, bài tiểu luận có thể vẫn còn thiếu sót. Kính mong thầy cô và các bạn đóng góp ý kiến để em có thể cải thiện và hoàn thiện bài tập tốt hơn.

Em xin chân thành cảm ơn!

LỜI CẢM ƠN

Em xin chân thành cảm ơn sâu sắc tới Ths. Nguyễn Thị Hương - bộ môn Công Nghệ Thông Tin – Khoa Điện tử - Trường Đại học Kỹ thuật Công Nghiệp – Đại học Thái Nguyên, người đã dành thời gian và tâm huyết để hướng dẫn và đồng hành cùng em trong quá trình học tập. Bằng kiến thức sâu rộng và sự tận tâm, cô Hương không chỉ là người hướng dẫn mà còn là nguồn động viên quý báu.

Em rất trân trọng sự hiểu biết và kiên nhẫn của cô Hương, cũng như những lời khuyên chi tiết và quý báu mà cô đã chia sẻ. Điều này thật sự là nguồn động viên lớn đối với em để không ngừng phấn đấu và hoàn thiện bản thân.

Chân thành cảm ơn cô, người đã là người hướng dẫn xuất sắc và đồng hành tận tâm trong hành trình học tập của em. Sự đóng góp của cô là không thể đong đếm được và để lại dấu ấn lâu dài trong sự phát triển của em. Cảm ơn cô!

Thái Nguyên, ngày.....tháng.....năm 2024

SINH VIÊN THỰC HIỆN

(Ký và ghi rõ họ tên)

CHƯƠNG 1 MỞ ĐẦU

1.1 Giải thuật và cấu trúc dữ liệu

Giải thuật (statements):

Là một dãy các câu lệnh chặt chẽ và rõ ràng, xác định một trình tự các thao tác trên một số đối tượng nào đó, sao cho sau một số hữu hạn các bước thực hiện, cho ta một kết quả mong muốn. Giải thuật chỉ phản ánh các phép xử lý.

Dữ liệu (Data):

Biểu diễn các thông tin cần thiết cho bài toán, là đối tượng để xử lý trên Máy tính.

Các phần tử của dữ liệu thường có mối quan hệ với nhau, việc tổ chức dữ liệu theo một cấu trúc thích hợp (CTDL) giúp cho việc thực hiện các phép xử lý trên dữ liệu được thuận lợi đạt hiệu quả cao hơn.

Mối quan hệ giữa CTDL và giải thuật:

Giải thuật tác động trên cấu trúc dữ liệu để đưa ra kết quả mong muốn. Giữa giải thuật và cấu trúc dữ liệu có mối quan hệ mật thiết với nhau. CTDL thay đổi giải thuật cũng thay đổi theo.

1.2 Ngôn ngữ diễn đạt và giải thuật

Vấn đề ngôn ngữ chúng ta có thể nghĩ ngay tới việc sử dụng một ngôn ngữ cấp cao hiện có, chẳng hạn PASCAL, C, C++,... Nhưng như vậy ta sẽ gặp một số vấn đề sau:

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

- Phải phụ thuộc vào cấu trúc dữ liệu tiền định của ngôn ngữ lên có lúc không thể hiện được đầy đủ các ý về cấu trúc mà ta muốn biểu đạt.

- Ngôn ngữ nào được chọn cũng không hẳn đã đươc sử dụng và yêu thích.

CHƯƠNG 2 THIẾT KẾ VÀ PHÂN TÍCH

2.1 Từ bài toán đến chương trình

2.1.1 Mô đun hoá và việc giải quyết bài toán

Mô đun hoá bài toán cho phép phân chia một bài toán lớn thành nhiều bài toán nhỏ độc lập, và tiếp tục phân chia bài toán nhỏ thành nhiều bài toán nhỏ hơn.... Mỗi bài toán tương đương với một mô đun.

Việc tổ chức lời giải của bài toán sẽ được thể hiện theo một cấu trúc phân cấp có dang:

Chiến thuật này được gọi là chiến thuật "chia để trị" (divice and conque). Thực hiện chiến thuật bằng cách thiết kế đỉnh xuống (top-down design)- Thiết kế từ khái quát→ Chi tiết.

Cách giải quyết bài toán sử dụng phương pháp mô đun hoá:

Phân tích tổng quát toàn bộ vấn đề (căn cứ vào dữ liệu và các mục tiêu đặt ra) -> đề cập đến công việc chủ yếu -> đi dần vào giải quyết bài toán cụ thể một cách chi tiết hơn. *[cách thiết kế từ khái quát đến chi tiết]*

2.1.2 Phương pháp tinh chỉnh từng bước (Stepwise refinement)

Là phương pháp thiết kế giải thuật và phát triển chương trình gắn liền với lập trình: Chương trình sẽ được thể hiện dần dần từ dạng ngôn ngữ tự nhiên, qua giả ngôn ngữ, rồi đến ngôn ngữ lập trình (gọi là các bước tinh chỉnh sự tinh chỉnh).

Đi từ mức "làm cái gì" đến mức "làm như thế nào". Ngày càng sát với các chức năng ứng với các câu lệnh của ngôn ngữ lập trình đã chọn.

Dữ liệu cũng được "tinh chế" từ dạng cấu trúc -> dạng lưu trữ, cài đặt cụ thể.

2.2 Phân tích giải thuật

2.2.1 Đặt vấn đề

Khi xây dựng được giải thuật và chương trình tương ứng của 1 bài toán có rất nhiều yêu cầu về phân tích thiết kế hệ thống:

Yêu cầu phân tích tính đúng đắn của giải thuật:

Một giải thuật gọi là đúng đắn nếu nó thực sự giải được yêu cầu của bài toán (Thể hiện đúng được lời giải của bài toán).

Yêu cầu về tính đơn giản của giải thuật:

Dễ hiểu, dễ lập trình, dễ chỉnh lý, phân tích (t) thực hiện giải thuật - Một trong các tiêu chuẩn để đánh giá hiệu lực của giải thuật

2.2.2 Phân tích thời gian thực hiện giải thuật

a. Thời gian thực hiện giải thuật phụ thuộc vào nhiều yếu tố:

1. Kích thước dữ liệu vào: Gọi n là số lượng dữ liệu đưa vào thì thời gian thực T của một giải thuật được biểu diễn bởi hàm T(n).

2. Các kiểu lệnh và tốc độ xử lý của máy tính.

3. Ngôn ngữ viết chương trình và chương trình dịch ngôn ngữ ấy.

Các yếu tố 2) và 3) không đồng đều với mọi máy tính. Không dựa vào chúng khi xác lập T(n). Cách đánh giá T(n) này cho ta khái niệm về "độ phức tạp tính toán của giải thuật" (Cấp độ lớn của thời gian thực hiện giải thuật).

b. Độ phức tạp tính toán của giải thuật:

Nếu thời gian thực hiện giải thuật là $T(n)=cn^2$. Độ phức tạp tính toán có cấp là n^2, ký hiệu là $T(n)=O(n^2)$. Một cách tổng quát có thể định nghĩa:

Một hàm f(n) được xác định là O(g(n)), f(n)=O(g(n)), được gọi là có cấp g(n)

Nếu tồn tại hằng số c và n_0 sao cho : f(n) <= c*g(n) khi n >= n_0

Thường các hàm thể hiện độ phức tạp tính toán của giải thuật có dạng: Log_2n, n, $nLog_2n$, n^2, n^3, 2^n, n!, nn

c. Xác định độ phức tạp tính toán:

Với một số giải thuật, ta có thể áp dụng một số quy tắc sau:

❖ **Quy tắc tính tổng:**

Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của 2 đoạn chương trình p_1, p_2 mà $T_1(n)=O(f(n))$, $T_2=O(g(n))$, thì thời gian thực hiện p_1 rồi p_2 tiếp theo sẽ là: $T_1(n)+ T_2(n)=O(max(f(n), g(n)))$.

❖ **Quy tắc nhân:**

p_1: $T_1(n)= O(f(n))$

p_2: $T_2(n)=O(g(n))$

Thời gian thực hiện p_1, p_2 lồng nhau sẽ là: $T_1(n)*T_2(n)=O(f(n), g(n))$

Chú ý: Khi đánh giá thời gian thực hiện giải thuật ta cần chú ý tới đến các bước tương ứng với một phép toán mà ta gọi là các phép toán tích cực(active operation).

CHƯƠNG 3: ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY

3.1 Định nghĩa và khái niệm về đệ quy

3.1.1 Khái niệm về đệ quy

Một đối tượng được gọi là đệ quy (recursive algorithm) nếu nó bao gồm chính nó như một bộ phận hoặc nó được định nghĩa dưới dạng của chính nó.

Ví dụ: Trong toán học ta cũng hay gặp các định nghĩa đệ quy.

1. Số tự nhiên

   a) 1 là một số tự nhiên

   b) x là số tự nhiên nếu x – 1 là số tự nhiên.

2. Hàm n giai thừa: n!

   a) 0! = 1

   b) Nếu n > 0 thì n! = n(n-1)!

3.1.2 Giải thuật đệ quy và thủ tục đệ quy

Nếu lời giải của bài toán T được thực hiện bởi lời giải của một bài toán T', có dạng như T thì đó là một ời giải đệ quy. Giải thuật tương ứng với lời gọi như vậy được gọi là giải thuật đệ quy (T' < T).

Đặc điểm của thủ tục đệ quy:

- Trong thủ tục đệ quy có lời gọi đến chính nó.

- Mỗi lần có lời gọi thì kích thước của bài toán đã thu nhỏ hơn trước

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

Đệ quy gồm hai loại:

- Đệ quy trực tiếp (Directly recursive): Thủ tục chứa lời gọi đến chính nó

- Đệ quy gián tiếp (Indirectly recursive): Thủ tục chứa lời gọi đến thủ tục khác mà thủ tục này lại chứa lời gọi đến chính nó.

3.1.3 Ưu điểm và hạn chế của giải thuật đệ quy

❖ **Ưu điểm:**

- Giải thuật đệ quy làm đơn giản hóa mã nguồn, giải thuật sáng sủa, dễ hiểu và nêu rõ bản chất vẫn đề bài toán.

- Dễ dàng bảo trì và mở rộng: các thuật toán phức tạp, đệ quy giúp giảm bớt sự phức tạp về mặt tổ chức mã, dễ dàng bảo trì và mở rộng hơn so với các thuật toán lặp phức tạp.

- Tiết kiệm thời gian thực hiện mã nguồn

❖ **Nhược điểm:**

- Giải thuật đệ quy gây tốn bộ nhớ, thời gian thực thi lâu: mỗi lần gọi đệ quy, một ngăn xếp mới được tạo them vào bộ nhớ dẫn đến việc tiêu tốn bộ nhớ nhiều hơn.

- Hiệu suất thấp, khó theo dõi và gỡ rối.

3.2 Các dạng giải thuật đệ quy

3.2.1 Dãy số Fibonacci

a. Định nghĩa

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

1) Các con thỏ không bao giờ chết

2) Hai tháng sau khi ra đời, mỗi cặp thỏ mới sẽ sinh ra một cặp thỏ con (một đực, một cái)

3) Khi đã sinh con rồi thì cứ mỗi tháng tiếp theo chúng lại sinh được một cặp con mới Giả sử từ đầu tháng 1 có một cặp mới ra đời thì đến giữa tháng thứ n sẽ có bao nhiêu cặp.

Ví dụ, n = 5, ta thấy:

Giữa tháng thứ 1:1 cặp (ab) (cặp ban đầu)

Giữa tháng thứ 2:1 cặp (ab) (cặp ban đầu vẫn chưa đẻ)

Giữa tháng thứ 3:2 cặp (AB)(cd) (cặp ban đầu đẻ ra thêm 1 cặp con)

Giữa tháng thứ 4:3 cặp (AB)(cd)(ef) (cặp ban đầu tiếp tục đẻ)

Giữa tháng thứ 5:5 cặp (AB)(CD)(ef)(gh)(ik) (cả cặp (AB) và (CD) cùng đẻ)

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

Nếu mỗi cặp thỏ ở tháng thứ n - 1 đều sinh ra một cặp thỏ con thì số cặp thỏ ở tháng thứ n sẽ là: F(n) = 2 * F(n – 1)

Nhưng vấn đề không phải như vậy, trong các cặp thỏ ở tháng thứ n - 1, chỉ có những cặp thỏ đã có ở tháng thứ n - 2 mới sinh con ở tháng thứ n được thôi. Do đó F(n) = F(n - 1) + F(n - 2) (= số cũ + số sinh ra). Vậy có thể tính được F(n) theo công thức sau:

F(n) = 1 nếu n ≤ 2

F(n) = F(n - 1) + F(n - 2) nếu n > 2

b. Thuật toán thủ tục đệ quy dãy Fibonacci

- ***Hàm đệ quy tính số cặp thỏ ở tháng thứ n (dãy Fibonacci)***

```
int Fibonacci(int n) {
   if (n <= 2) {
      return 1; // Phần neo
   } else {
      return Fibonacci(n - 1) + Fibonacci(n - 2); // Phần đệ quy
   }
}
```

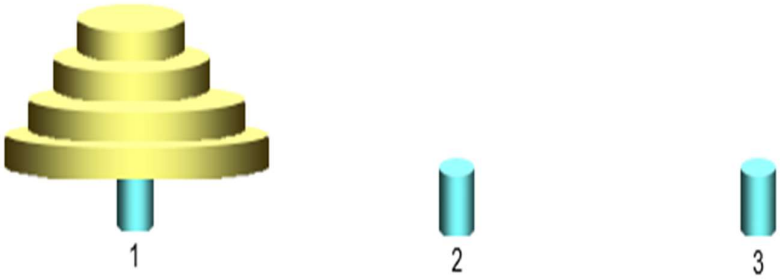
- ***Tính số cặp thỏ***

```
      int n;
      cout << "Nhap thang n: ";
      cin >> n;
      if (n <= 0) {
         cout << "Thang n phai lon hon 0!" << endl;
      } else {
         cout << "So cap tho o thang thu " << n << " la: " << Fibonacci(n)
<< endl;
      }
      break;
}
```

3.2.2 Bài toán Tháp Hà Nội (Tower of Hanoi)

a. Định nghĩa

Đây là một bài toán mang tính chất một trò chơi, tương truyền rằng tại ngôi đền Benares có ba cái cọc kim cương. Khi khai sinh ra thế giới, thượng đế đặt n cái đĩa bằng vàng chồng lên nhau theo thứ tự giảm dần của đường kính tính từ dưới lên, đĩa to nhất được đặt trên một chiếc cọc.



Hình 1: Mô phỏng tháp Hà Nội

b. Thuật toán thủ tục đệ quy

- ***Hàm đệ quy giải bài toán Tháp Hà Nội***

```
void ThapHaNoi(int n, int x, int y) {
   if (n == 1) {
      cout << "Chuyen 1 dia tu coc " << x << " sang coc " << y << endl;
   } else {
      int z = 6 - x - y; // Tính cột trung gian (6 = 1 + 2 + 3)
      ThapHaNoi(n - 1, x, z); // Chuyển n-1 đĩa từ cọc x sang cọc trung gian
      cout << "Chuyen 1 dia tu coc " << x << " sang coc " << y << endl;
      ThapHaNoi(n - 1, z, y); // Chuyển n-1 đĩa từ cọc trung gian sang cọc y
   }
}
```

- ***Giải bài toán Tháp Hà Nội***

```
   int n;
```

```
        cout << "Nhap so dia: ";
        cin >> n;
        if (n <= 0) {
            cout << "So dia phai lon hon 0!" << endl;
        } else {
            cout << "Cac buoc chuyen dia: " << endl;
            ThapHaNoi(n, 1, 3); // Gọi hàm Move với cột 1 (A) và cột 3 (C)
        }
        break;
    }
```

3.3 Ví dụ bài tập

Bài 1: Viết một hàm đệ quy tính ước số chung lớn nhất của hai số tự nhiên a, b không đồng thời bằng 0, chỉ rõ đâu là phần neo, đâu là phần đệ quy.

Giải

- ***Giải thuật hàm đệ quy tính USCLN***

```
int GCD(int a, int b) {
        if (b == 0) {
        return a; // Phần neo: Khi b = 0, trả về a }
        return GCD(b, a % b); // Phần đệ quy: Gọi hàm với (b, a % b)
}
```

- ***Giải thích***

Phần neo:

- Nếu b=0b = 0b=0, trả về aaa. Đây là điểm dừng vì khi b=0b = 0b=0, aaa là ước số chung lớn nhất của aaa và bbb.

Phần đệ quy:

- Gọi hàm với GCD(b,a%b)GCD(b, a \% b)GCD(b,a%b).

- Phép chia lấy dư a%ba \% ba%b giảm kích thước bài toán, đảm bảo tiến dần về phần neo.

Bài 2: Cho n là số nguyên dương n (0<n<=10), viết thủ tục đệ qui tính F(n) được cho bởi công thức sau:

$$\begin{cases} F(n) = 1 \text{ Nếu } n < 3 \\ F(n) = F(n-1) + F(n-3) \text{ nếu } n \geq 3 \end{cases}$$

Giải

- ***Giải thuật hàm đệ quy tính F(n)***

```
int F(int n) {
        if (n < 3) { return 1; // Phần neo: Khi n < 3, F(n) = 1
        }
        return F(n - 1) + F(n - 3); // Phần đệ quy: F(n) = F(n-1) + F(n-3)
}
```

- ***Giải thích***

Phần neo:

- Với n<3n < 3n<3, hàm trả về giá trị F(n)=1F(n) = 1F(n)=1. Đây là điều kiện dừng của thuật toán.

Phần đệ quy:

- Với n≥3n \geq 3n≥3, công thức xác định F(n)=F(n−1)+F(n−3)F(n) = F(n-1) + F(n-3)F(n)=F(n−1)+F(n−3).

- Hàm sẽ gọi lại chính nó với hai đối số nhỏ hơn (n−1n-1n−1 và n−3n-3n−3), sau đó cộng kết quả.

CHƯƠNG 4: MẢNG VÀ DANH SÁCH

4.1 Định nghĩa và khái niệm về mảng và danh sách

4.1.1 Mảng (Array)

a. Khái niệm mảng

Mảng (Array): Là một tập hợp có thứ tự gồm một số cố định, các phần tử không có phép bổ xung hoặc loại bỏ1 phần tử. Chỉ có các phép tạolập, tìm kiếm, lưu trữ. Mỗi phần tử của mảng ngoài giá trị (info) còn được đặc trưng bởi chỉ số để biểu hiện thứ tự của nó trong mảng. Có mảng 1 chiều, 2, 3 chiều...n chiều.

b. Ưu điểm và hạn chế của mảng

❖ **Ưu điểm**

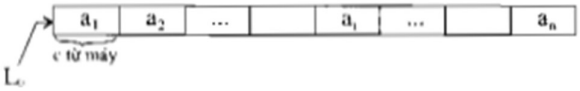
- Truy cập nhanh: nhờ vào việc lưu trữ liên tiếp trong bộ nhớ, mảng cho phép truy cập đến bất kỳ phần tử nào một cách trực tiếp (truy cập ngẫu nhiên) với độ phức tạp O(1).
- Sắp xếp và tìm kiếm dễ dàng: các thuật toán sắp xếp và tìm kiếm được áp dụng hiệu quả trên mảng vì mảng có thứ tự lưu trữ liên tục.
- Tối ưu hóa không gian so với các cấu trúc dữ liệu như danh sách liên kết (link list) vì mảng không cần lưu trữ thông tin về các liên kết giữa các phần tử/
- Đơn giản vì mảng là cấu trúc dữ liệu cơ bản.

❖ **Hạn chế**

- Kích thước cố định không thể thêm hoặc xóa phần tử trong mảng
- Không linh hoạt về loại dữ liệu: tất cả các phần tử trong mảng phải có cùng một kiểu dữ liệu, không thể lưu trữ hai kiểu dữ liệu khác nhau trong cùng một mảng.
- Tốn kém trong việc mở rộng: khi kích thước của mảng đầy, để mở rộng mảng, một mảng mới lớn hơn phải được tạo ra và tất cả các phần tử từ mảng cũ phải được sao chép sang mảng mới, làm tốn kém hiệu suất khi làm việc với các mảng có kích thước lớn.

c. Cấu trúc lưu trữ của mảng

- Một vector A có n phần tử, nếu mỗi phần tử a[i] (a<=i<=n) chiếm c từ máy thì nó sẽ được lưu trữ trong cn từ máy kế tiếp nhau (lưu trữ kế tiếp – Sequential storange allocation).
- Khi mảng được lưu trữ kế tiếp thì việc truy cập vào phần tử của mảng được thực hiện trực tiếp dựa vào địa chỉ được tính nên tốc độ truy cập nhanh và đồng đều đối với mọi phần tử.



Hình 2: Cấu trúc của mảng

4.1.2 Danh sách (List)

a. Khái niệm danh sách

Danh sách (list): là một tập có thứ tự nhưng bao gồm một số biến động các phần tử. Phép bổ sung và phép loại bỏ một phần tử là phép thường xuyên tác động lên danh sách.

Danh sách có khả năng thay đổi kíc0h thước, có thể thêm hoặc bớt phần tử của danh sách. Cho phép truy cập các phần tử thông qua chỉ số giống như mảng.

b. Ưu điểm và hạn chế của danh sách

- **Ưu điểm:** danh sách linh hoạt, có kích thước động nên có thể thêm hoặc xóa phần tử trong danh sách, các phần tử trong danh sách có thể có nhiều kiểu dữ liệu khác nhau.
- **Hạn chế:** khi chèn và xóa các phần tử ở giữa sẽ chậm và tốn bộ nhớ hơn so với mảng, và truy cập không nhanh bằng mảng trong một số trường hợp đặc biệt.

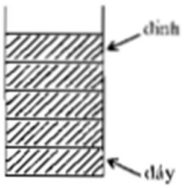
c. Cấu trúc lưu trữ của danh sách

- **Stack hay danh sách kiểu ngăn xếp**

Stack là một kiểu danh sách tuyến tính đặc biệt mà phép bổ sung và phép loại bỏ luôn luôn thực hiện ở một đầu gọi là đỉnh (top).

Cấu trúc của stack LIFO (last – in – firts – out) :

- Dùng một mảng để chứa các phần tử
- Dùng một số nguyên để lưu số phần tử tối đa trong stack
- Dùng một số nguyên để lưu chỉ số đỉnh của stack



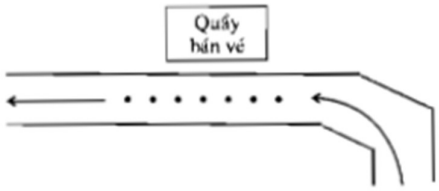
Hình 3: Ngăn xếp (stack)

❖ **Queue hay danh sách kiểu hàng đợi**

Hàng đợi là một danh sách tuyến tính trong đó phép bổ sung một phần tử mới được thực hiện thực hiện ở một đầu và phép toán loại bỏ phần tử cũ được thực hiện ở đầu còn lại của hàng đợi.

Cấu trúc của queue LIFO (firts – in – firts – out):

- Dùng một mảng để chứa các phần tử
- Dùng một số nguyên (QMax) để lưu số phần tử tối đa trong hàng đợi
- Dùng hai số nguyên (QFront, QRear) để xác định vị trí đầu, cuối của hàng đợi
- Dùng một số nguyên (QNumItem) để lưu số phần tử hiện có trong hàng đợi



Hình 4: Hàng đợi (Queue)

4.2 Các dạng thuật toán mảng và danh sách

4.2.1 Mảng (Array)

- ***Ham nhap mang***

```
void Nhapmang(int a[], int n){
    for(int i = 0; i<n; i++){
        cout<<"a["<<i<<"] = ";
        cin>>a[i];
    }
}
```

- ***Ham xuat mang***

```
void Xuatmang(int a[], int n){
    for(int i = 0; i<n; i++){
        cout<<a[i]<<" ";
    }
}
```

4.2.2 Stack kiểu ngăn xếp

- ***Lớp Stack (Ngăn xếp)***

```
class Cstack {
private:
    int* StkArr; // Mảng lưu trữ các phần tử của stack
    int StkMax;  // Kích thước tối đa của stack
    int StkTop;  // Vị trí đỉnh của stack
```

- ***Hàm khởi tạo stack với kích thước cho trước***

```
Cstack(int size) {
        StkArr = new int[size]; // Khởi tạo mảng có kích thước 'size'
        StkMax = size;          // Gán kích thước tối đa cho stack
        StkTop = -1;            // Khởi tạo stack rỗng (đỉnh là -1)
    }
```

- ***Hàm hủy để giải phóng bộ nhớ khi không cần sử dụng stack nữa***

```
Cstack() {
        delete[] StkArr; // Giải phóng mảng khi stack bị hủy
    }
```

- ***Kiểm tra xem stack có rỗng không***

```
bool isEmpty() {
        return (StkTop == -1); // Trả về true nếu stack rỗng
    }
```

- ***Kiểm tra xem stack có đầy không***

```
bool isFull() {
        return (StkTop == StkMax - 1); // Trả về true nếu stack đầy
    }
```

- ***Thêm (Push) một phần tử vào stack***

```
bool Push(int newItem) {
        if (isFull()) return false;     // Stack đầy, không thể thêm phần tử
```

```
        StkArr[++StkTop] = newItem;     // Thêm phần tử vào đỉnh stack
        return true;                    // Thêm thành công
    }
```

- ***Lấy ra (Pop) một phần tử từ đỉnh của stack***

```
    bool Pop(int& outItem) {
        if (isEmpty()) return false;    // Stack rỗng, không thể lấy phần tử
        outItem = StkArr[StkTop--];     // Gán phần tử đỉnh stack cho outItem
        return true;                    // Lấy ra thành công
    }
```

- ***Lấy phần tử đỉnh mà không xóa (Peek)***

```
    bool Peek(int& topItem) {
        if (isEmpty()) return false;    // Stack rỗng, không thể lấy phần tử đỉnh
        topItem = StkArr[StkTop];       // Gán phần tử đỉnh stack cho topItem
        return true;                    // Trả về thành công
    }
};
```

4.2.3 Queue kiểu hàng đợi

- ***Lớp Queue (Hàng đợi)***

```
    class Queue {
    private:
        int *QArray;  // Mảng chứa các phần tử của queue
        int QMax;     // Kích thước tối đa của queue
        int QFront;   // Vị trí đầu queue
        int QRear;    // Vị trí cuối queue
        int QNumItems; // Số phần tử hiện có trong queue
```

- ***Hàm khởi tạo Queue với kích thước cho trước***

```
    public:
        Queue(int size) {
```

```
    QArray = new int[size]; // Khởi tạo mảng chứa các phần tử của queue
    QMax = size;            // Gán kích thước tối đa cho queue
    QFront = QRear = -1;    // Khởi tạo queue rỗng
    QNumItems = 0;          // Số phần tử ban đầu là 0
}
```

- ***Hàm hủy để giải phóng bộ nhớ khi không cần sử dụng Queue nữa***

```
Queue() {
    delete[] QArray;  // Giải phóng mảng khi queue bị hủy
}
```

- ***Kiểm tra xem Queue có rỗng không***

```
bool IsEmpty() {
    return (QNumItems == 0); // Kiểm tra queue có rỗng không
}
```

- ***Kiểm tra xem Queue có đầy không***

```
bool IsFull() {
    return (QNumItems == QMax); // Kiểm tra queue có đầy không
}
```

- ***Thêm một phần tử vào cuối Queue***

```
bool Append(int newItem) {
    if (IsFull()) {
        cout << "Queue day!" << endl; // Nếu queue đầy
        return false;
    }
    if (QNumItems == 0) QFront = 0; // Nếu queue rỗng, khởi tạo QFront
    QRear = (QRear + 1) % QMax;  // Tăng QRear và vòng lại nếu cần
    QArray[QRear] = newItem;     // Thêm phần tử vào cuối queue
    QNumItems++;                 // Tăng số phần tử trong queue
    cout << "Them thanh cong!" << endl;
```

```
    return true;
}
```

- ***Xóa một phần tử trong Queue***

```
bool Take(int &itemOut) {
    if (IsEmpty()) {
        cout << "Queue rong!" << endl; // Nếu queue rỗng
        return false;
    }
    itemOut = QArray[QFront];  // Lấy phần tử từ đầu queue
    QFront = (QFront + 1) % QMax; // Tăng QFront và vòng lại nếu cần
    QNumItems--;  // Giảm số phần tử trong queue
    if (QNumItems == 0) QFront = QRear = -1;  // Nếu queue rỗng
    return true;
}
```

- ***Lấy một phần tử trong Queue***

```
void Display() {
    if (IsEmpty()) {
        cout << "Queue rong!" << endl; // Nếu queue rỗng
        return;
    }
    cout << "Cac phan tu trong Queue: ";
    int i = QFront;
    for (int j = 0; j < QNumItems; j++) {
        cout << QArray[i] << " ";  // In các phần tử trong queue
        i = (i + 1) % QMax;       // Vòng lại nếu cần
    }
    cout << endl;
}   };
```

4.3 Ví dụ bài tập

Bài 1: Cho số nguyên 39. Hãy biểu diễn giải thuật chuyển đổi số nguyên đó sang số nhị phân trên theo cơ chế hoạt động của Stack.

$39_{10}=100111_2$

Theo stack : vẽ các khối push, pop.

Giải

- ***Giải thuật***

```
void DecimalToBinary(int n) {
        stack<int> binaryStack;
        // Chuyển đổi và push phần dư vào stack
        while (n > 0) {
        binaryStack.push(n % 2); n /= 2;
        }
        // Lấy kết quả từ stack
        cout << "So nhi phan: ";
        while (!binaryStack.empty()) {
        cout << binaryStack.top(); binaryStack.pop();
        }
        cout << endl;
}
```

- ***Giải thích***

1. Khởi tạo một stack rỗng.
2. Thực hiện:
   - Chia số nguyên nnn cho 2.
   - Lấy phần dư của phép chia (n%2n \% 2n%2) và Push vào stack.
   - Lấy kết quả phép chia nguyên (n÷2n \div 2n÷2) để tiếp tục.
3. Lặp lại cho đến khi n=0n = 0n=0.
4. Khi kết thúc, thực hiện Pop từng phần tử trong stack và ghép lại để tạo số nhị phân.

CHƯƠNG 5: DANH SÁCH MÓC NỐI

5.1 Định nghĩa và khái niệm về danh sách móc nối

5.1.1 Khái niệm danh sách móc nối

Danh sách móc nối (link list): là một cấu trúc dữ liệu bao gồm một nhóm các nút (nodes) tạo thành một chuỗi. Thông thường mỗi node gồm dữ liệu (data) ở nút đó và tham chiếu (reference) đến nút kế tiếp trong chuỗi. Là cấu trúc dữ liệu đơn giản và phổ biến.

5.1.2 Ưu điểm và nhược điểm của danh sách móc nối

❖ **Ưu điểm:**

- Cung cấp giải pháp để chứa cấu trúc dữ liệu tuyến tính
- Dễ dàng thêm hoặc xóa các phần tử trong danh sách mà không cần phải cấp phát hoặc tổ chức lại trật tự của mảng
- Cấp phát bộ nhớ động

❖ **Nhược điểm:**

- Một danh sách liên kết đơn giản không cho phép truy cập ngẫu nhiên dữ liệu. Chính vì lý do này mà một số phép tính như tìm phần tử cuối cùng, xóa phần tử ngẫu nhiên hay chèn thêm, tìm kiếm có thể phải duyệt tất cả các phần tử.

5.1.3 Các loại danh sách móc nối

a. Danh sách móc nối đơn (Singly Linked List)

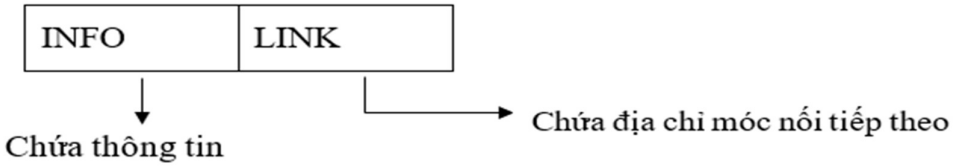
Danh sách móc nối đơn: là danh sách có nhiều nút móc nối đơn lại với nhau theo một chiều. Nút của danh sách bao gồm một số từ máy kế tiếp nhau, nút có thể nằm ở vị trí bất kỳ trong bộ nhớ, mỗi nút chứa nhiều thông tin ứng với một phần tử và địa chỉ phần tử đứng sau.

Mỗi phần tử của d/s được lưu trữ trong một phần tử nhớ gọi là nút

Mỗi nút bao gồm một số từ máy kế tiếp nhau

Các nút có thế nằm ở vị trí bất kỳ trong bộ nhớ

Mỗi nút chứa những thông tin ứng với một phần tử và địa chỉ phần tử đứng sau d/s:



Hình 2: Cấu trúc móc nối đơn

Nút cuối cùng : Không có nút đứng sau → LINK chứa "Địa chỉ đặc biệt" dùng để đánh dấu nút kết thúc DS, ta gọi là mối nối không, ký hiệu NULL

b. Danh sách móc nối kép (Doubly Linked List)

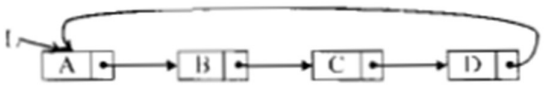
Danh sách móc nối kép: cũng là một danh sách liên kết nhưng mỗi phần tử liên kết với phần tử đứng trước và sau nó trong danh sách.

Tổ chức dữ liệu:

- Với danh sách liên kết đôi, mỗi phần tử sẽ liên kết với phần tử đứng trước và sau nó trong danh sách
- Mỗi phần tử trong danh sách (node) gồm biến lưu trữ dữ liệu và hai con trỏ liên kết đến phần tử trước và sau nó

c. Danh sách móc nối vòng (Curcularly linked list)

Tương tự như danh sách móc nối đơn, nhưng nút cuối cùng trỏ lại nút đầu tiên, tạo thành một vòng.



Hình 5: Danh sách móc nối vòng

5.2 Các thuật toán và giải thuật

5.2.1 Danh sách móc nối đơn

- ***Định nghĩa cấu trúc một nút trong danh sách***

```
struct Node {
    int data;       // Giá trị của nút
    Node* next;     // Con trỏ tới nút kế tiếp
};
```

- ***Con trỏ tới danh sách liên kết***

```
Node* L = NULL;
```

- ***Tạo một nút mới***

```
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

- ***Xóa một nút ở cuối danh sách***

```
void delete_cuoi() {
    if (L == NULL) {
        cout << "Danh sach trong\n";
    } else if (L->next == NULL) {  // Nếu chỉ có một phần tử
        delete L;
        L = NULL;
    } else {
        Node* temp = L;
        Node* prev = NULL;
        while (temp->next != NULL) {
            prev = temp;
            temp = temp->next; // Tìm nút cuối
        }
        prev->next = NULL;
        delete temp; // Xóa nút cuối
    }
    cout << "Danh sach sau khi xoa cuoi: ";
    void display();   }
```

- ***Xóa một nút ở đầu danh sách***

```
void delete_dau() {
    if (L == NULL) {
        cout << "Danh sach trong\n";
    } else {
        Node* temp = L;
        L = L->next;
        delete temp;
    }
    cout << "Danh sach sau khi xoa dau: ";
    void display();
}
```

- ***Thêm một nút vào cuối danh sách***

```
void insert_cuoi(int value) {
    Node* newNode = createNode(value);
    if (L == NULL) {
        L = newNode;
    } else {
        Node* temp = L;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    cout << "Danh sach sau khi them cuoi: ";
    void display();
}
```

- ***Thêm một nút vào đầu danh sách***

```
void insert_dau(int value) {
    Node* newNode = createNode(value);
    if (L == NULL) {
        L = newNode;
    } else {
        newNode->next = L;
        L = newNode;
    }
    cout << "Danh sach sau khi them dau: ";
    void display();
}
```

- ***Xóa nút đầu tiên có giá trị x trong danh sách***

```
void delete_value(int value) {
    if (L == NULL) {
        cout << "Danh sach trong\n";
        return;
    }

    if (L->data == value) {  // Xóa nút có giá trị đầu danh sách
        Node* temp = L;
        L = L->next;
        delete temp;
        cout << "Danh sach sau khi xoa gia tri " << value << ": ";
        void display();
        return;
    }
```

```
    Node* temp = L;
    Node* prev = NULL;
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) {
        cout << "Gia tri " << value << " khong ton tai trong danh sach\n";
        return;
    }

    prev->next = temp->next;
    delete temp;
    cout << "Danh sach sau khi xoa gia tri " << value << ": ";
    void display();
}
```

5.2.2 Danh sách móc nối kép

- **Khai báo cấu trúc Node**

```
struct Node {
    int data;       // Gia tri cua Node
    Node* next;     // Con tro toi Node tiep theo
    Node* prev;     // Con tro toi Node truoc do
};

// Khai bao con tro dau va cuoi cua danh sach
Node* head = NULL;
Node* tail = NULL;
```

- ***Hàm tạo Node mới***

```
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
```

- ***Hàm hiển thị danh sách***

```
void display() {
    if (head == NULL) {
        cout << "Danh sach trong." << endl;
        return;
    }
    Node* temp = head;
    cout << "Danh sach: ";
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
```

- ***Thêm một Node vào đầu danh sách***

```
void insert_dau(int value) {
    Node* newNode = createNode(value);
    if (head == NULL) { // Neu danh sach trong
        head = tail = newNode;
    } else {
```

```
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    cout << "Da them vao dau danh sach." << endl;
    display(); // Hien thi danh sach sau khi them
}
```

- ***Thêm một Node vào cuối danh sách***

```
void insert_cuoi(int value) {
    Node* newNode = createNode(value);
    if (head == NULL) { // Neu danh sach trong
        head = tail = newNode;
    } else {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
    cout << "Da them vao cuoi danh sach." << endl;
    display(); // Hien thi danh sach sau khi them
}
```

- ***Xóa Node ở đầu danh sách***

```
void delete_dau() {
    if (head == NULL) {
        cout << "Danh sach trong, khong the xoa." << endl;
        return;
    }
    Node* temp = head;
    if (head == tail) { // Neu chi co mot phan tu
```

```
        head = tail = NULL;
    } else {
        head = head->next;
        head->prev = NULL;
    }
    delete temp;
    cout << "Da xoa Node o dau danh sach." << endl;
    display(); // Hien thi danh sach sau khi xoa
}
```

- ***Xóa Node ở cuối danh sách***

```
void delete_cuoi() {
    if (head == NULL) {
        cout << "Danh sach trong, khong the xoa." << endl;
        return;
    }
    Node* temp = tail;
    if (head == tail) { // Neu chi co mot phan tu
        head = tail = NULL;
    } else {
        tail = tail->prev;
        tail->next = NULL;
    }
    delete temp;
    cout << "Da xoa Node o cuoi danh sach." << endl;
    display(); // Hien thi danh sach sau khi xoa
}
```

- ***Xóa Node có giá trị cho trước***

```
void delete_value(int value) {
```

```
    if (head == NULL) {
        cout << "Danh sach trong, khong the xoa." << endl;
        return;
    }
    Node* temp = head;
    while (temp != NULL && temp->data != value) {
        temp = temp->next;
    }
    if (temp == NULL) {
        cout << "Gia tri " << value << " khong ton tai trong danh sach." << endl;
        return;
    }
    if (temp == head) { // Xoa dau
        delete_dau();
        return;
    }
    if (temp == tail) { // Xoa cuoi
        delete_cuoi();
        return;
    }
```

- ***Xóa ở giữa***

```
    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
    delete temp;
    cout << "Da xoa Node co gia tri " << value << "." << endl;
    display(); // Hien thi danh sach sau khi xoa
}
```

5.2.3 Danh sách nối vòng

- ***Cấu trúc Node***

```
struct Node {
    int data;       // Dữ liệu của Node
    Node* next;     // Con trỏ tới Node tiếp theo
};
```

- ***Con trỏ đầu và cuối danh sách***

```
Node* head = NULL;
Node* tail = NULL;
```

- ***Hàm hiển thị danh sách***

```
void display() {
    if (head == NULL) {
        cout << "Danh sach trong." << endl;
        return;
    }
    Node* temp = head;
    cout << "Danh sach: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head); // Lặp lại đến khi quay lại Node đầu tiên
    cout << endl;
}
```

- ***Tạo một Node mới***

```
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = NULL;
```

```
    return newNode;
}
```

- ***Thêm Node vào đầu danh sách***

```
void insert_dau(int value) {
    Node* newNode = createNode(value);
    if (head == NULL) { // Nếu danh sách rỗng
        head = tail = newNode;
        tail->next = head; // Kết nối lại thành vòng
    } else {
        newNode->next = head;
        head = newNode;
        tail->next = head; // Kết nối lại tail với head
    }
    cout << "Da them vao dau danh sach." << endl;
    display(); // Hiển thị danh sách sau khi thêm
}
```

- ***Thêm Node vào cuối danh sách***

```
void insert_cuoi(int value) {
    Node* newNode = createNode(value);
    if (head == NULL) { // Nếu danh sách rỗng
        head = tail = newNode;
        tail->next = head; // Kết nối lại thành vòng
    } else {
        tail->next = newNode;
        tail = newNode;
        tail->next = head; // Kết nối lại tail với head
    }
    cout << "Da them vao cuoi danh sach." << endl;
```

```
    display(); // Hiển thị danh sách sau khi thêm
}
```

- ***Xóa Node ở đầu danh sách***

```
void delete_dau() {
    if (head == NULL) {
        cout << "Danh sach trong, khong the xoa." << endl;
        return;
    }
    Node* temp = head;
    if (head == tail) { // Nếu chỉ có một phần tử
        head = tail = NULL;
    } else {
        head = head->next;
        tail->next = head; // Kết nối lại tail với head mới
    }
    delete temp;
    cout << "Da xoa Node o dau danh sach." << endl;
    display(); // Hiển thị danh sách sau khi xóa
}
```

- ***Xóa Node ở cuối danh sách***

```
void delete_cuoi() {
    if (head == NULL) {
        cout << "Danh sach trong, khong the xoa." << endl;
        return;
    }
    Node* temp = head;
    if (head == tail) { // Nếu chỉ có một phần tử
        head = tail = NULL;
```

```
    } else {
        while (temp->next != tail) {
            temp = temp->next;
        }
        temp->next = head; // Kết nối lại Node cuối mới với head
        delete tail;
        tail = temp;
    }
    cout << "Da xoa Node o cuoi danh sach." << endl;
    display(); // Hiển thị danh sách sau khi xóa
}
```

- ***Xóa Node có giá trị cụ thể***

```
void delete_value(int value) {
    if (head == NULL) {
        cout << "Danh sach trong, khong the xoa." << endl;
        return;
    }
    if (head->data == value) { // Nếu Node đầu có giá trị cần xóa
        delete_dau();
        return;
    }
    Node* temp = head;
    Node* prev = NULL;
    do {
        prev = temp;
        temp = temp->next;
        if (temp->data == value) {
            prev->next = temp->next;
```

```
            if (temp == tail) { // Nếu Node cuối có giá trị cần xóa
                tail = prev;
            }
            delete temp;
            cout << "Da xoa Node co gia tri " << value << "." << endl;
            display(); // Hiển thị danh sách sau khi xóa
            return;
        }
    } while (temp != head);
    cout << "Gia tri " << value << " khong ton tai trong danh sach." << endl;
}
```

5.3 Vi dụ bài tập

Bài 1: Cho danh sách móc nối đơn có nút đầu danh sách trỏ bởi con trỏ P, với trường info trong các nút là các số nguyên dương. Hãy trình bày giải thuật bổ xung phần tử mới vào cuối danh sách.

Giải

- **Giải thuật**

Hàm bổ sung phần tử mới vào cuối danh sách

```
void insertAtEnd(Node*& P, int value) {
        Node* newNode = new Node(value); // Tạo nút mới
        if (P == nullptr) {
        // Trường hợp danh sách rỗng
        P = newNode; }
        else { Node* temp = P; // Bắt đầu từ nút đầu danh sách
        while (temp->next != nullptr) {
        temp = temp->next; // Duyệt tới nút cuối
        }
        temp->next = newNode; // Gắn nút mới vào cuối danh sách
        }       }
```

- **Giải thích**

Tạo một nút mới với giá trị value:

- *Khởi tạo một nút mới (newNode) với giá trị bằng value và con trỏ next trỏ tới nullptr.*

Kiểm tra danh sách có rỗng hay không:

- *Nếu danh sách rỗng (P == nullptr), nút mới (newNode) sẽ trở thành nút đầu tiên của danh sách. Gán P = newNode.*

Danh sách không rỗng:

- *Nếu danh sách không rỗng, thực hiện duyệt danh sách để tìm nút cuối:*
- *Khởi tạo con trỏ temp trỏ tới nút đầu của danh sách (temp = P).*
- *Lặp lại:*
- *Kiểm tra xem nút hiện tại (temp) có con trỏ next là nullptr không:*
- *Nếu temp->next != nullptr, di chuyển temp tới nút kế tiếp (temp = temp->next).*
- *Dừng khi tìm được nút cuối (nút có temp->next == nullptr).*

Gắn nút mới vào cuối danh sách:

- *Thiết lập con trỏ next của nút cuối (temp->next) trỏ tới newNode.*

Kết thúc:

- *Danh sách được cập nhật với nút mới được thêm vào cuối.*

CHƯƠNG 6: CÂY (TREE)

6.1 Định nghĩa và khái niệm về cây

6.1.1 Khái niệm cây

Cây (Tree): là tập hữu hạn các nút (tree node), sao cho có một nút gọi là nút *gốc* (root). Giữa các nút có mối quan hệ phân cấp gọi là quan hệ "cha – con".

Có thể định nghĩa cây bằng các đệ quy như sau:

- Mỗi nút là một cây, nút đó cũng là gốc của cây ấy

- Nếu n là một nút và n_1, n_2, …, n_k lần lượt là gốc của các cây T_1, T_2, …, T_k; các cây này đôi một không có nút chung. Thì nếu cho nút n trở thành cha của các nút n_1, n_2, …, n_k ta sẽ được một cây mới T. Cây này có nút n là gốc còn các cây T_1, T_2, …, T_k trở thành các cây con (subtree) của gốc.

Người ta quy ước: Một cây không có nút nào được gọi là *cây rỗng* (Null tree).

6.1.2 Các loại cây:

a. Cây nhị phân (Binary tree)

Mọi nút trên cây chỉ có tối đa là hai con, *cây con trái* (left subtree) và *cây con phải* (right subtree). Như vậy cây nhị phân là cây có thứ tự tính đến thứ tự của các nhánh con :và là dạng quan trọng của cấu trúc cây. Nó có đặc điểm là mọi nút trên cây chỉ có tối đa 2 nhánh con.

b. Cây nhị phân tìm kiếm (Binary search tree – BST)

Cây nhị phân tìm kiếm là cây nhị phân sao cho mỗi một nút có một khóa duy nhất và mọi nút P của cây đều thỏa mãn tính chất:

- Tất cả các nút thuộc cây con trái đều có giá trị khóa nhỏ hơn giá trị khóa của P (nhỏ hơn gốc P)
- Tất cả các nút thuộc cây con phải đều có giá trị khóa lớn hơn giá trị khóa của P

c. Cây nhị phân liên kết vòng (Threaded binary tree)

Sử dụng liên kết NULL để lưu trữ liên kết tới nút kế tiếp trong phép duyệt cây nhị phân để phép duyệt được thực hiện trở nên dễ dạng hơn. Sử dụng giá trị kiểm tra liên kết thật (đến nút trong cây) hay liên kết giả (nút trong phép duyệt).

LTYPE = True, nếu liên kết trái là liên kết thật

RTYPE = True, nếu liên kết phải là liên kết thật

LPTR	LTYPE	INFO	RTYPE	RPTR

d. Cây tổng quát (General tree)

Cây tổng quát (general tree): là một cấu trúc dữ liệu dạng cây, trong đó mỗi nút có thể có nhiều con. Cây tổng quát là một cây mở rộng của cây nhị phân, trong cây tổng quát không có giới hạn về số lượng con của mỗi nút.

Biểu diễn cây nhị phân bằng liên kết động với m nút con (cây con):

- Mỗi nút có m + 1 trường, với m mối nối
- Với cây có m phân đầy đủ, sẽ có tới n(m – 1) + 1 mối nối liên kết "NULL".

6.2 Các thuật toán và giải thuật

6.2.1 Tạo cây nhị phân

- ***Hàm tạo cây nhị phân từ mảng giá trị***

```
Node* createTreeFromArray(const vector<int>& values) {
    if (values.empty() || values[0] == -1) {
        return nullptr; // Trả về nullptr nếu mảng rỗng hoặc giá trị gốc là -1
    }

    Node* root = new Node(values[0]); // Tạo nút gốc
    queue<Node*> q;  // Dùng hàng đợi để quản lý các nút
    q.push(root);

    int i = 1;
    while (i < values.size()) {
        Node* current = q.front();
        q.pop();
```

- ***Gán con trái***

```
        if (values[i] != -1) {
            current->left = new Node(values[i]);
```

```
            q.push(current->left);
        }
        i++;
    }
```

- ***Gán con phải***

```
        if (i < values.size() && values[i] != -1) {
            current->right = new Node(values[i]);
            q.push(current->right);
        }
        i++;
    }
    return root;
}
```

6.2.2 Phép duyệt trước (Preorder)

- ***Hàm Duyệt trước (Preorder)***

```
void Preorder(Node* root) {
    if (root != nullptr) {
        cout << root->value << " ";  // In giá trị của nút hiện tại
        Preorder(root->left);        // Duyệt con trái
        Preorder(root->right);       // Duyệt con phải
    }
}
```

6.2.3 Phép duyệt giữa (Inorder)

- ***Hàm Duyệt giữa (Inorder)***

```
void Inorder(Node* root) {
    if (root != nullptr) {
        Inorder(root->left);        // Duyệt con trái
        cout << root->value << " "; // In giá trị của nút hiện tại
```

```
        Inorder(root->right);       // Duyệt con phải
    }
}
```

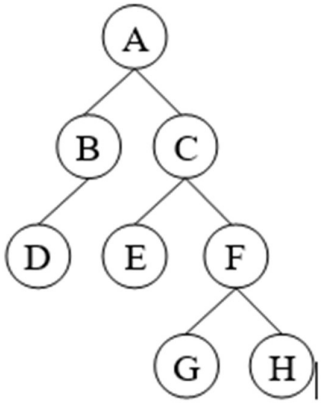
6.2.4 Phép duyệt sau (Postorder)

- ***Hàm Duyệt sau (Postorder)***

```
void Postorder(Node* root) {
    if (root != nullptr) {
        Postorder(root->left);      // Duyệt con trái
        Postorder(root->right);     // Duyệt con phải
        cout << root->value << " "; // In giá trị của nút hiện tại
    }
}
```

6.3 Ví dụ bài tập

Bài 1: Cho cây nhị phân như hình vẽ. Hãy viết thứ tự các nút được thăm khi duyệt cây theo thứ tự trước? Theo thứ tự giữa? Theo thứ tự sau?



Giải

Để giải bài toán này, chúng ta cần hiểu rõ về các loại duyệt cây:

Duyệt trước (Pre-order): Thăm nút gốc trước, sau đó thăm cây con bên trái, cuối cùng thăm cây con bên phải.

Duyệt giữa (In-order): Thăm cây con bên trái, sau đó thăm nút gốc, cuối cùng thăm cây con bên phải.

Duyệt sau (Post-order): Thăm cây con bên trái, sau đó thăm cây con bên phải, cuối cùng thăm nút gốc.

Duyệt trước (Pre-order):

Thăm A - Thăm B - Thăm D - Thăm E - Thăm C - Thăm F - Thăm G - Thăm H
Thứ tự: A B D E C F G H

Duyệt giữa (In-order):

Thăm D - Thăm B - Thăm E - Thăm A - Thăm G - Thăm F - Thăm C - Thăm H

Thứ tự: D B E A G F C H

Duyệt sau (Post-order):

Thăm D - Thăm E - Thăm B - Thăm G - Thăm H - Thăm F - Thăm C - Thăm A
Thứ tự: D E B G H F C A

CHƯƠNG 7: ĐỒ THỊ VÀ CẤU TRÚC PHI TUYẾN KHÁC

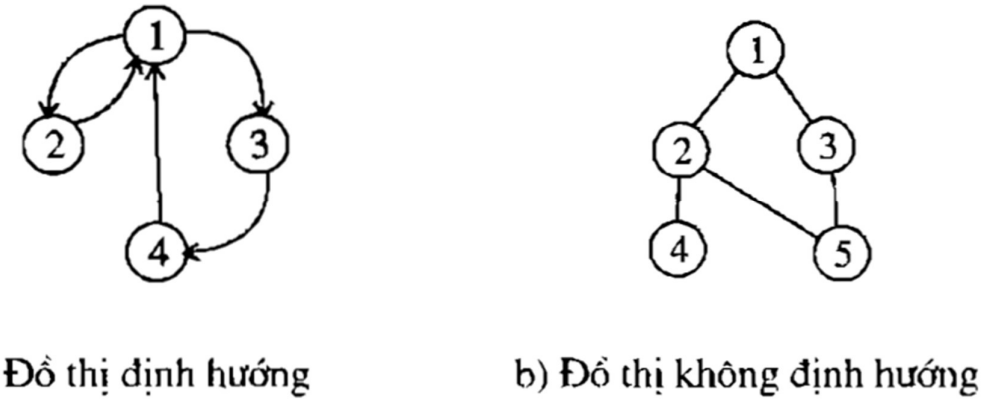
7.1 Định nghĩa và khái niệm về đồ thị

7.1.1 Định nghĩa và các khái niệm về đồ thị

Một đồ thị (Graph) G(V,E) bao gồm một tập hợp hữu hạn V các *nút*, hay *đỉnh* (vertices) và một tập hợp hữu hạn E các cặp đỉnh mà ta gọi là *cung* (edges).

Nếu (v1, v2) là các cặp đỉnh thuộc E thì ta nói: có một cung nối v1 và v2. Nếu cung (v1, v2) khác với cung (v2, v1) thì ta có một *đồ thị định hướng* (directed graph hay digraph). Lúc đó (v1, v2) được gọi là cung định hướng từ v1, v2. Nếu thứ tự các nút trên cây không được coi trọng thì ta gọi *đồ thị không định hướng* (undirected graph).

Bằng hình vẽ ta có thể biểu diễn bằng đồ thị như hình sau:



Hình 6: Các dạng đồ thị

7.1.2 Cây là trường hợp đặc biệt của đồ thị

- Nếu (v1, v2) là một cung thuộc E => v1, v2 gọi là lân cận của nhau.
- Một đường đi từ đỉnh vp đến đỉnh vq trong đồ thị G là một dãy các đỉnh vp, vi1, vi2…,vin,vp là các cung trong E(G), Số lượng các cung trên đường đi gọi là độ dài đường đi.
- Một đường đi đơn: mọi đỉnh trên đường đi đều khác nhau, trừ đỉnh đầu và đỉnh cuối.
- Chu trình : là đường đi đơn mà đỉnh đầu và đỉnh cuối trùng nhau.
- Với đồ thị định hướng, để cho rõ người ta thường thêm vào các từ " định hướng " sau các thuật ngữ nêu trên.

7.2 Các thuật toán và giải thuật

7.2.1 Cây khung tối thiểu (Kruskal)

- ***Hàm tìm kiếm và hợp nhất cho thuật toán Kruskal***

```
int find(vector<int>& parent, int i) {
    if (parent[i] == -1)
        return i;
    return find(parent, parent[i]);
}

void unionSet(vector<int>& parent, int x, int y) {
    parent[x] = y;
}
```

- ***Thuật toán Kruskal***

```
void kruskal() {
    sort(canh.begin(), canh.end(), [](Canh a, Canh b) {
        return a.w < b.w; // Sắp xếp các cạnh theo trọng số
    });

    vector<int> parent(dinh, -1);

    cout << "\nCay khung toi thieu (Kruskal):" << endl;
    for (auto& edge : canh) {
        int x = find(parent, edge.u);
        int y = find(parent, edge.v);

        if (x != y) {
            cout << edge.u << " - " << edge.v << " : " << edge.w << endl;
            unionSet(parent, x, y);
```

```
            }
        }
    }
```

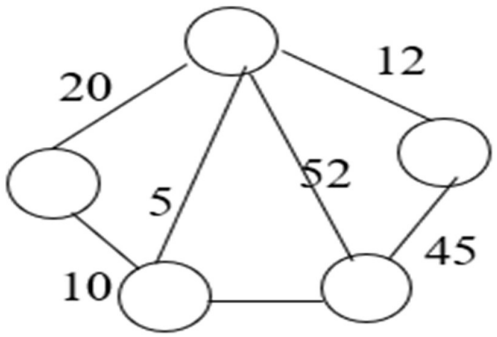
7.2.2 Tìm kiếm đường đi ngắn nhất trên đồ thị (Dijkstra)

- ***Thuật toán Dijkstra***

```
void dijkstra(int start) {
    vector<int> dist(dinh, INT_MAX);
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({ 0, start });
    dist[start] = 0;

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        for (auto& p : adjList[u]) {
            int v = p.first;
            int weight = p.second;

            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({ dist[v], v });
            }
        }
    }
```

7.3 Ví dụ bài tập

Bài 1: Cho đồ thị vô hướng có trọng số (như hình vẽ) hãy dựng cây khung có trọng số nhỏ nhất theo giải thuật Kruskal.



Giải

- **Giải thuật**

Danh sách các cạnh (u, v, weight)

```
vector<Edge> edges = {
   Edge(1, 2, 20),
   Edge(1, 3, 12),
   Edge(1, 4, 10),
   Edge(2, 4, 5),
   Edge(3, 4, 45),
   Edge(2, 3, 52)
};
int n = 4; // Số đỉnh
sort(edges.begin(), edges.end(), compareEdge); // Sắp xếp cạnh theo trọng số
vector<int> parent(n + 1), rank(n + 1, 0);
for (int i = 1; i <= n; i++) parent[i] = i;
vector<Edge> mst; // Lưu trữ cây khung nhỏ nhất
int totalWeight = 0;
```

Xây dựng cây khung nhỏ nhất

```
    for (const auto& edge : edges) {
        if (findParent(edge.u, parent) != findParent(edge.v, parent)) {
            mst.push_back(edge);                // Thêm cạnh vào cây khung
            totalWeight += edge.weight;         // Cộng trọng số
            unionSets(edge.u, edge.v, parent, rank); // Hợp các tập hợp
        }
    }
```

- **Giải thích**

Tạo danh sách các cạnh:

- Danh sách edges lưu trữ tất cả các cạnh của đồ thị, mỗi cạnh được tạo bằng Edge(u, v, weight).

Sắp xếp cạnh theo trọng số:

- Sử dụng sort() với hàm compareEdge() để sắp xếp các cạnh theo thứ tự trọng số tăng dần.

Khởi tạo cấu trúc Union-Find:

- Mảng parent để lưu cha của mỗi nút.
- Mảng rank để tối ưu hóa quá trình hợp các tập hợp.

Duyệt qua các cạnh và xây dựng cây khung:

- Kiểm tra nếu hai đỉnh của cạnh thuộc các tập hợp khác nhau (findParent()), thêm cạnh đó vào cây khung và hợp hai tập hợp lại (unionSets()).
- Bỏ qua cạnh nếu nó tạo chu trình.

In kết quả:

- In các cạnh thuộc cây khung nhỏ nhất (mst) và tổng trọng số (totalWeight).

CHƯƠNG 8: SẮP XẾP (SORTING)

8.1 Định nghĩa và khái niệm về sắp xếp (sorting)

8.1.1 Khái niệm về sắp xếp

Sắp xếp (Sorting): là quá trình bố trí lại các phần tử của một tập đối tượng nào đó, theo một thứ tự ấn định. Chẳng hạn như thứ tự tăng dần (hay giảm dần) đối với một dãy số, thứ tự từ điển với các từ v.v... Yêu cầu về sắp xếp thường xuyên xuất hiện trong các ứng dụng Tin học với các mục đích khác nhau: sắp xếp dữ liệu trong máy tính để tìm kiếm cho thuận lợi, sắp xếp các kết quả xử lý để in ra trên bảng biểu v.v...

8.1.2 Các loại thuật toán sắp xếp

a. Sắp xếp kiểu lựa chọn (Selection Sort)

❖ **Ưu điểm:**

- Đơn giản, dễ hiểu và dễ triển khai.
- Không yêu cầu bộ nhớ phụ.
- Ít thao tác hoán đổi, chỉ thực hiện một lần mỗi vòng lặp.

❖ **Nhược điểm:**

- Độ phức tạp thời gian $O(n^2)$, không hiệu quả với tập dữ liệu lớn.
- Không phải là thuật toán tối ưu cho sắp xếp mảng lớn.

b. Sắp xếp kiểu chèn (Insertion Sort)

❖ **Ưu điểm:**

- Đơn giản và dễ hiểu.
- Hiệu quả khi mảng đã gần như được sắp xếp hoặc với tập dữ liệu nhỏ.
- Sắp xếp tại chỗ, không cần bộ nhớ phụ.
- Hoạt động tốt với mảng nhỏ hoặc danh sách có dữ liệu gần như đã sắp xếp.

❖ **Nhược điểm:**

- Độ phức tạp thời gian $O(n^2)$, chậm khi mảng có số lượng phần tử lớn.
- Dịch chuyển nhiều phần tử, giảm hiệu suất với dữ liệu không sắp xếp.

c. Sắp xếp kiểu nổi bọt (Bubble Sort)

❖ **Ưu điểm:**

- Đơn giản và dễ hiểu.
- Dễ dàng phát hiện mảng đã được sắp xếp sớm (tối ưu khi không có hoán đổi).

❖ **Nhược điểm:**

- Độ phức tạp thời gian $O(n^2)$, không hiệu quả đối với mảng lớn.

- Quá nhiều vòng lặp, hiệu suất kém so với các thuật toán sắp xếp khác.
- Không hiệu quả với mảng lớn, dù có thể dừng sớm nếu mảng đã được sắp xếp.

d. Sắp xếp kiểu vun đống (Heap Sort)

❖ **Ưu điểm:**

- Độ phức tạp thời gian O(n log n) trong tất cả các trường hợp, hiệu quả cho dữ liệu lớn.
- Sắp xếp tại chỗ, không cần bộ nhớ phụ.
- Thích hợp cho các ứng dụng cần sắp xếp nhanh.

❖ **Nhược điểm:**

- Các thao tác chèn và loại bỏ từ heap không thể thực hiện theo cách tối ưu (do cấu trúc cây đống).
- Cấu trúc cây đống không phải là một cấu trúc tự nhiên, dễ gây khó khăn cho việc triển khai và hiểu.

e. Sắp xếp kiểu nhanh (Quick Sort)

❖ **Ưu điểm:**

- Độ phức tạp thời gian trung bình O(n log n), nhanh chóng.
- Được sử dụng rộng rãi trong các ứng dụng thực tế vì tính hiệu quả và khả năng phân tách tốt.
- Sắp xếp tại chỗ, không cần bộ nhớ phụ.

❖ **Nhược điểm:**

- Độ phức tạp thời gian có thể đạt $O(n^2)$ trong trường hợp xấu nhất (khi phân hoạch không cân bằng).
- Phụ thuộc vào việc chọn phân chia tốt (pivot), nếu không có thể làm giảm hiệu suất.
- Không thích hợp với các bộ dữ liệu đã được sắp xếp hoặc gần như đã sắp xếp.

f. Sắp xếp hòa nhập (Merge Sort)

❖ **Ưu điểm:**

Độ phức tạp thời gian O(n log n) ổn định, luôn hiệu quả với các tập dữ liệu lớn. Tốt cho sắp xếp các tập dữ liệu lớn hoặc sắp xếp ngoài bộ nhớ (như sắp xếp file). Sắp xếp ổn định, giữ nguyên thứ tự của các phần tử có giá trị bằng nhau.

❖ **Nhược điểm:**

Cần bộ nhớ phụ O(n) cho các mảng tạm. Không phải sắp xếp tại chỗ, cần nhiều bộ nhớ để thực hiện thao tác hợp nhất.

8.2 Các thuật toán và giải thuật

8.2.1 Sắp xếp kiểu lựa chọn (Selection Sort)

- ***Hàm Selection Sort***

```
void selectionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        int jmin = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[jmin]) {
                jmin = j;
            }
        }
        if (jmin != i) {
            swap(arr[i], arr[jmin]);
        }
    }
    cout << "Mang da duoc sap xep (Selection Sort)." << endl;
}
```

8.2.2 Sắp xếp kiểu chèn (Insertion Sort)

- ***Hàm Insertion Sort***

```
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int tmp = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > tmp) {
            arr[j + 1] = arr[j];
```

```
            j--;
        }
        arr[j + 1] = tmp;
    }
    cout << "Mang da duoc sap xep (Insertion Sort)." << endl;
}
```

8.2.3 Sắp xếp kiểu nổi bọt(Bubble Sort)

- **Hàm Bubble Sort**

```
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        for (int j = n - 1; j >= i; j--) {
            if (arr[j] < arr[j - 1]) {
                swap(arr[j], arr[j - 1]);
            }
        }
    }
    cout << "Mang da duoc sap xep (Bubble Sort)." << endl;
}
```

8.2.4 Sắp xếp kiểu vun đống (Heap Sort)

- **Hàm Heapify cho Heap Sort**

```
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;
```

```
    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```

- ***Hàm Heap Sort***

```
void heapSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
    cout << "Mang da duoc sap xep (Heap Sort)." << endl;
}
```

8.2.5 Sắp xếp kiểu nhanh (Quick Sort)

- ***Hàm Partition cho Quick Sort***

```
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
```

```
            if (arr[j] < pivot) {
                i++;
                swap(arr[i], arr[j]);
            }
        }
        swap(arr[i + 1], arr[high]);
        return i + 1;
    }
```

- ***Hàm Quick Sort***

```
    void quickSort(vector<int>& arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
```

8.2.6 Sắp xếp kiểu hòa nhập (Merge Sort)

- ***Hàm Merge cho Merge Sort***

```
    void merge(vector<int>& arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        vector<int> L(n1), R(n2);

        for (int i = 0; i < n1; i++)
            L[i] = arr[left + i];
        for (int j = 0; j < n2; j++)
            R[j] = arr[mid + 1 + j];
```

```
    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

- ***Hàm Merge Sort***

```
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
```

```
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

8.3 Ví dụ bài tập

Bài 1: Viết chương trình sắp xếp một mảng 1 chiều bằng phương pháp nổi bọt và tìm kiếm phần tử có giá trị bằng 5?

Giải

- **Giải thuật**

Hàm thực hiện sắp xếp nổi bọt (bubble sort)

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Hoán đổi nếu phần tử trước lớn hơn phần tử sau
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Hàm tìm kiếm phần tử có giá trị bằng 5

```
int searchElement(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x)
            return i;  // Trả về vị trí của phần tử nếu tìm thấy
    }
```

```
    return -1;  // Trả về -1 nếu không tìm thấy
}
```

- **Giải thích**
  - Người dùng nhập mảng từ bàn phím: Trước khi sắp xếp, chương trình sẽ yêu cầu người dùng nhập số phần tử của mảng và giá trị của từng phần tử.
  - Hàm bubbleSort: Sắp xếp các phần tử của mảng theo thứ tự tăng dần.
  - Hàm searchElement: Tìm kiếm phần tử có giá trị bằng 5 trong mảng và trả về vị trí nếu tìm thấy.

Bài 2: Cho dãy khoá sau:

15 90 31 27 4 59 27

Hãy trình bày giải thuật sắp xếp dãy số trên theo phương pháp sắp xếp vun đồng (heap sort)

Giải

- **Giải thuật**

Hàm heapify (đưa một nút về đúng vị trí trong Max-Heap)

```
void heapify(int arr[], int n, int i) {
    int largest = i;        // Giả sử nút hiện tại là lớn nhất
    int left = 2 * i + 1;   // Chỉ số nút con trái
    int right = 2 * i + 2;  // Chỉ số nút con phải
```

Kiểm tra nếu nút con trái lớn hơn nút hiện tại

```
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }
```

Kiểm tra nếu nút con phải lớn hơn nút lớn nhất hiện tại

```
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }
```

Nếu nút hiện tại không phải lớn nhất, hoán đổi và đệ quy

```
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
```

```
        }
    }
```

Hàm Heap Sort

```
    void heapSort(int arr[], int n) {
```

Bước 1: Xây dựng Max-Heap

```
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }
```

Bước 2: Lấy từng phần tử từ Heap và sắp xếp

```
        for (int i = n - 1; i > 0; i--) {
            // Đưa phần tử lớn nhất (ở gốc) về cuối mảng
            swap(arr[0], arr[i]);

            // Giảm kích thước Heap và điều chỉnh lại đống
            heapify(arr, i, 0);
        }
    }
```

- **Giải thích**

Đầu vào:

- Một mảng arr[] gồm n phần tử cần sắp xếp.

Đầu ra:

- Mảng arr[] được sắp xếp theo thứ tự tăng dần (hoặc giảm dần tùy biến).

Bước 1: Xây dựng Max-Heap

1. Một Max-Heap là cây nhị phân mà giá trị tại nút cha luôn lớn hơn hoặc bằng giá trị tại các nút con.
2. Duyệt từ nút không phải lá cuối cùng (ở chỉ số n/2 - 1) về gốc (chỉ số 0).
3. Với mỗi nút, áp dụng hàm Heapify để điều chỉnh cấu trúc Heap sao cho tuân thủ tính chất Max-Heap.

Bước 2: Trích xuất phần tử lớn nhất

1. Sau khi xây dựng Max-Heap, phần tử lớn nhất nằm ở gốc (chỉ số 0).
2. Hoán đổi phần tử gốc với phần tử cuối mảng (di chuyển phần tử lớn nhất đến cuối).
3. Giảm kích thước Heap (bỏ qua phần tử đã sắp xếp) và áp dụng lại Heapify cho phần tử gốc.
4. Lặp lại cho đến khi toàn bộ mảng được sắp xếp.

CHƯƠNG 9: TÌM KIẾM (SEARCHING)

9.1 Định nghĩa và khái niệm về tìm kiếm

9.1.1 Khái niệm về tìm kiếm

Tìm kiếm (Searching): là một quá trình xác định vị trí hoặc tìm ra phần tử mong muốn trong một tập hợp các phần tử, chẳng hạn như một mảng, danh sách, hoặc cấu trúc dữ liệu khác. Mục tiêu là tìm thấy phần tử theo một điều kiện nhất định (ví dụ như một giá trị khóa hoặc một thuộc tính nào đó). Quá trình tìm kiếm đóng vai trò quan trọng trong việc truy xuất dữ liệu nhanh chóng và hiệu quả.

9.1.2 Các dạng thuật toán tìm kiếm

a. Tìm kiếm tuần tự (Sequential Search)

Nội dung có thể tóm tắt như sau: "Bắt đầu từ bản ghi đầu tiên, lần lượt so sánh khoá tìm kiếm với khoá tương ứng của các bản ghi trong danh sách, cho tới khi tìm thấy bản ghi mong muốn hoặc đã duyệt hết danh sách mà chưa thấy".

❖ **Ưu điểm:**

- Đơn giản, dễ hiểu và dễ triển khai.
- Không yêu cầu dữ liệu phải được sắp xếp trước.
- Phù hợp với các tập dữ liệu nhỏ hoặc trong trường hợp không biết trước dữ liệu đã được sắp xếp hay chưa.

❖ **Nhược điểm:**

- Hiệu suất thấp với tập dữ liệu lớn, vì phải duyệt qua từng phần tử.
- Không tối ưu cho các ứng dụng yêu cầu truy xuất nhanh.

b. Tìm kiếm nhị phân (Binary Search)

❖ **Ưu điểm:**

- Tốc độ nhanh và hiệu quả hơn tìm kiếm tuần tự.
- Thích hợp với các tập dữ liệu lớn đã được sắp xếp, giúp tiết kiệm thời gian truy xuất dữ liệu.

❖ **Nhược điểm:**

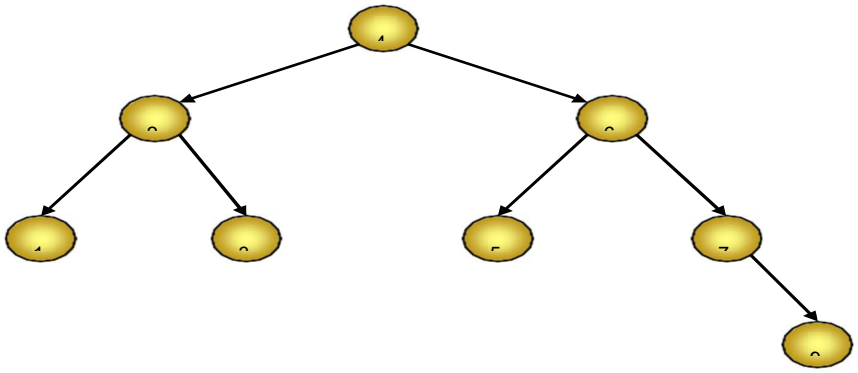
- Chỉ áp dụng được với dữ liệu đã sắp xếp. Nếu dữ liệu chưa sắp xếp, cần sắp xếp trước, làm tăng chi phí tính toán.
- Cần có cấu trúc dữ liệu cho phép truy cập ngẫu nhiên (như mảng); không phù hợp cho các cấu trúc như danh sách liên kết.

c. Cây nhị phân tìm kiếm (Binary Search Tree - BST)

Cho n khoá k_1, k_2, …, k_n, trên các khoá có quan hệ thứ tự toàn phần. Cây nhị phân tìm kiếm ứng với dãy khoá đó là một cây nhị phân mà mỗi nút chứa giá trị một khoá trong n khoá đã cho, hai giá trị chứa trong hai nút bất kỳ là khác nhau. Đối với mọi nút trên cây, tính chất sau luôn được thoả mãn:

- Mọi khoá nằm trong cây con trái của nút đó đều nhỏ hơn khoá ứng với nút đó.

- Mọi khoá nằm trong cây con phải của nút đó đều lớn hơn khoá ứng với nút đó.



Hình 7: Cây nhị phân tìm kiếm

Thuật toán tìm kiếm trên cây có thể mô tả chung như sau:

Trước hết, khoá tìm kiếm X được so sánh với khoá ở gốc cây, và 4 tình huống có thể xảy ra: Không có gốc (cây rỗng): X không có trên cây, phép tìm kiếm thất bại

- X trùng với khoá ở gốc: Phép tìm kiếm thành công
- X nhỏ hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con trái của gốc với cách làm tương tự
- X lớn hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con phải của gốc với cách làm tương tự

❖ **Ưu điểm:**

- Tìm kiếm, chèn và xóa nhanh chóng nếu cây cân bằng.
- BST có thể mở rộng, cho phép sắp xếp và thực hiện nhiều thao tác tìm kiếm phức tạp.

❖ **Nhược điểm:**

- Hiệu suất có thể giảm xuống nếu cây bị mất cân đối (cây "lệch" chỉ có một nhánh dài).

- Cần các thao tác cân bằng cây (như dùng cây AVL hoặc cây đỏ-đen) để đảm bảo hiệu suất ổn định khi cây dễ bị mất cân đối.

9.2 Các thuật toán và giải thuật

9.2.1 Tìm kiếm tuần tự

- ***Hàm tìm kiếm tuần tự (Sequential Search)***

```
int SequentialSearch(const vector<int>& arr, int X) {
    int n = arr.size();
    // Thêm một phần tử phụ bằng X vào cuối mảng để xử lý khi không tìm thấy
    vector<int> tempArr = arr;
    tempArr.push_back(X);

    int i = 0;
    while (tempArr[i] != X) {
        i++;
    }
    return (i == n) ? 0 : i + 1; // Trả về 0 nếu không tìm thấy, ngược lại trả về chỉ số
}
```

9.2.2 Tìm kiếm nhị phân

- ***Hàm tìm kiếm nhị phân (Binary Search)***

```
int BinarySearch(const vector<int>& arr, int X) {
    int inf = 0, sup = arr.size() - 1;

    while (inf <= sup) {
        int median = (inf + sup) / 2;
        if (arr[median] == X) {
            return median + 1; // Trả về chỉ số + 1 (theo yêu cầu đề bài)
```

```
        } else if (arr[median] < X) {
            inf = median + 1;
        } else {
            sup = median - 1;
        }
    }
    return 0; // Không tìm thấy
}
```

9.3 Ví dụ bài tập

Bài 1: Hãy trình bày giải thuật tìm kiếm có bao nhiêu phần tử có giá trị bằng 5?

15 5 2 9 38 5

Giải

- **Giải thuật**

Khởi tạo mảng với dãy số đã cho

```
int arr[] = {15, 5, 2, 9, 38, 5};
int n = sizeof(arr) / sizeof(arr[0]);  // Kích thước của mảng
int value_to_find = 5;  // Giá trị cần tìm
bool found = false;     // Biến kiểm tra xem có tìm thấy không
```

Duyệt qua từng phần tử của mảng

```
cout << "Cac vi tri co gia tri bang 5: ";
for (int i = 0; i < n; i++) {
    if (arr[i] == value_to_find) {
        cout << i << " ";  // In ra vị trí của phần tử bằng 5
        found = true;      // Đánh dấu là đã tìm thấy
    }
}
```

Nếu không tìm thấy giá trị nào bằng 5

```
if (!found) {
    cout << "Khong co phan tu nao co gia tri bang 5.";
}
```

- **Giải thích**
  - Nhập số lượng phần tử từ bàn phím: Người dùng nhập số lượng phần tử của mảng.
  - Nhập giá trị của từng phần tử từ bàn phím: Dãy số sẽ được nhập từ bàn phím và lưu vào mảng arr[].
  - Duyệt qua mảng: Chương trình kiểm tra từng phần tử trong mảng. Nếu phần tử có giá trị bằng 5, chương trình sẽ in ra vị trí của nó.
  - Nếu không có phần tử bằng 5: Chương trình sẽ in thông báo rằng không có phần tử nào bằng 5.

TÀI LIỆU THAM KHẢO

[1] Đỗ Xuân Lôi: *"Cấu trúc dữ liệu"*. Đại học Bách khoa Hà Nội xuất bản - 1976.

[2] Đỗ Xuân Lôi: *"Sắp xếp và tìm kiếm dữ liệu trên máy tính điện tử"*. Đại học Bách khoa Hà Nội xuất bản - 1980.

[3] Nguyễn Xuân Huy: *"Thuật toán"*. Nhà xuất bản Thống kê - 1988.

[4] Đoàn Nguyên Hải - Nguyễn Trung Trực - Nguyễn Anh Dũng: *"Lập trình căn bản"*. Trung tâm điện toán - Đại học Bách khoa TP. HCM - 1991.

[5] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Algorithms.* Addison-Wesley, Reading, MA, 1975.

[6] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching.* Second edition, Addison-Wesley, Reading, MA, 1997.

[7] https://chatgpt.com/