**TRƯỜNG ĐẠI HỌC KỸ THUẬT CÔNG NGHIỆP**

**KHOA ĐIỆN TỬ**

Bộ môn: Công nghệ thông tin

**A circular logo with a star and a blue star with a red and yellow circle

Description automatically generated**

**BÀI TẬP LỚN**

MÔN HỌC

**CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

**ĐỀ TÀI: CÁC GIẢI THUẬT CỦA CÁC CHƯƠNG**

Sinh viên: …Trần Thị Thu Hà …………………………………………

Mã số sinh viên:

Lớp: ……K58KMT.K01……………………………………………….

Giáo viên hướng dẫn: ………Th.S Nguyễn Thị Hương………………

**Thái Nguyên – 2024**

|  |  |
| --- | --- |
| **TRƯỜNG ĐHKTCN** | **CỘNG HOÀ XÃ HỘI CHỦ NGHĨA VIỆT NAM** |
| **KHOA ĐIỆN TỬ** | ***Độc lập - Tự do - Hạnh phúc*** |
|  |  |

**PHIẾU GIAO BÀI TẬP TIỂU LUẬN**

**MÔN HỌC: CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

BỘ MÔN : TIN HỌC CÔNG NGHIỆP

*Sinh viên: Trần Thị Thu Hà*

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

*Giáo viên hướng dẫn: Th.S Nguyễn Thị Hương*

*Ngày giao đề: 10/08/2024*  *Ngày hoàn thành: 05/11/2024*

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. Hình thức nộp

1. Báo cáo bản word

2. File source code chương trình

**NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN**

*Thái Nguyên, ngày….tháng…..năm 20....*

GIÁO VIÊN HƯỚNG DẪN

*(Ký ghi rõ họ tên)*

**MỤC LỤC**

[**Lời mở đầu** 6](#_Toc182774890)

[**CHƯƠNG 3. ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY** 7](#_Toc182774891)

[**I.** **Khái niệm và cách thức hoạt động của đệ quy** 7](#_Toc182774892)

[**1.** **Đệ quy (Recursion algorithm)**. 7](#_Toc182774893)

[**2.** **Cách thức hoạt động của đệ quy** 7](#_Toc182774894)

[**3.** **Ưu điểm và hạn chế của giải thuật đệ quy** 7](#_Toc182774895)

[**4.** **Giải thuật đệ quy** 8](#_Toc182774896)

[**CHƯƠNG 4. MẢNG VÀ DANH SÁCH** 11](#_Toc182774897)

[**I.** **Mảng** 11](#_Toc182774898)

[**1.** **Khái niệm mảng** 11](#_Toc182774899)

[**2.** **Ưu điểm và hạn chế của mảng** 11](#_Toc182774900)

[**3.** **Cấu trúc lưu trữ của mảng** 11](#_Toc182774901)

[**4.** **Các thao tác trên mảng** 12](#_Toc182774902)

[**II.** **Danh sách** 14](#_Toc182774903)

[**1.** **Khái niệm danh sách** 14](#_Toc182774904)

[**2.** **Ưu điểm và hạn chế của danh sách** 14](#_Toc182774905)

[**CHƯƠNG 5. DANH SÁCH MÓC NỐI (LINKLIST)** 22](#_Toc182774906)

[**1.** **Khái niệm danh sách móc nối** 22](#_Toc182774907)

[**2.** **Ưu điểm và nhược điểm của danh sách móc nối** 22](#_Toc182774908)

[**3.** **Các loại danh sách móc nối** 22](#_Toc182774909)

[**3.1.** **Danh sách móc nối đơn** 22](#_Toc182774910)

[**3.2.** **Các thuật toán của danh sách móc nối đơn** 22](#_Toc182774911)

[**3.3.** **Danh sách liên kết đôi** 27](#_Toc182774912)

[**3.4.** **Các thuật toán của danh sách kết nối đôi** 27](#_Toc182774913)

[**3.5.** **Danh sách móc nối vòng** 29](#_Toc182774914)

[**3.6.** **Các thuật toán của danh sách kết nối vòng** 30](#_Toc182774915)

[**CHƯƠNG 6. CÂY** 33](#_Toc182774916)

[**1.** **Khái niệm cây** 33](#_Toc182774917)

[**2.** **Các loại cây:** 33](#_Toc182774918)

[**2.1.** **Cây nhị phân (binary tree)** 33](#_Toc182774919)

[**2.2.** **Cây nhị phân tìm kiếm (binary search tree – BST)** 33](#_Toc182774920)

[**2.3.** **Cây tổng quát (general tree)** 33](#_Toc182774921)

[**3.** **Các thuật toán của cây** 33](#_Toc182774922)

[**3.1.** **Các phép duyệt cây nhị phân** 33](#_Toc182774923)

[**3.2.** **Các thao tác trên cây nhị phân tìm kiếm (binary search tree – BST)** 35](#_Toc182774924)

[**3.3.** **Các thao tác trên cây tổng quát** 39](#_Toc182774925)

[**CHƯƠNG 7. ĐỒ THỊ** 43](#_Toc182774926)

[**I.** **Định nghĩa đồ thị và các khái niệm** 43](#_Toc182774927)

[**1.** **Định nghĩa đồ thị** 43](#_Toc182774928)

[**2.** **Phân loại đồ thị** 43](#_Toc182774930)

[**3.** **Các khái niệm** 43](#_Toc182774931)

[**4.** **Biểu diễn đồ thị** 44](#_Toc182774932)

[**5.** **Các phép duyệt đồ thị** 45](#_Toc182774933)

[**CHƯƠNG 8. SẮP XẾP** 52](#_Toc182774934)

[**1.** **Khái niệm** 52](#_Toc182774935)

[**1.1.** **Thuật toán sắp xếp lựa chọn Selection sort** 52](#_Toc182774936)

[**1.2.** **Thuật toán sắp xếp chèn Inserttion sort** 53](#_Toc182774937)

[**1.3.** **Thuật toán sắp xếp nổi bọt Buble sort** 53](#_Toc182774938)

[**1.4.** **Thuật toán sắp xếp nhanh Quick sort** 54](#_Toc182774939)

[**1.5.** **Thuật toán sắp xếp Heap sort** 55](#_Toc182774940)

[**CHƯƠNG 9. TÌM KIẾM** 57](#_Toc182774941)

[**1.** **Bài toán tìm kiếm** 57](#_Toc182774942)

[**2.** **Các phép toán tìm kiếm** 57](#_Toc182774943)

[**2.1.** **Phép tìm kiếm tuần tự (sequential search)** 57](#_Toc182774944)

[**2.2.** **Phép tìm kiếm nhị phân (search binary)** 58](#_Toc182774945)

[**2.3.** **Cây nhị phân tìm kiếm (binary search tree)** 60](#_Toc182774946)

[**KẾT LUẬN** 62](#_Toc182774947)

[**TÀI LIỆU THAM KHẢO** 63](#_Toc182774948)

# **Lời mở đầu**

Môn học Cấu trúc dữ liệu và Giải thuật đóng vai trò nền tảng quan trọng trong việc phát triển kỹ năng lập trình và tư duy logic của sinh viên ngành Công nghệ thông tin. Việc hiểu rõ các cấu trúc dữ liệu cơ bản như mảng, danh sách liên kết, cây, đồ thị, cùng với các thuật toán tìm kiếm, sắp xếp, và tối ưu hóa sẽ giúp sinh viên xây dựng được các hệ thống phần mềm hiệu quả và tối ưu. Bài tập lớn cuối kỳ này được thiết kế để áp dụng các lý thuyết đã học vào thực tiễn, qua đó giúp sinh viên củng cố kiến thức, rèn luyện kỹ năng phân tích và giải quyết vấn đề.

Ngoài việc học các thuật toán đơn giản, bài tập còn giúp sinh viên nắm vững cách thức triển khai các cấu trúc dữ liệu phức tạp, cũng như tối ưu hóa thuật toán sao cho phù hợp với từng tình huống cụ thể. Từ đó, sinh viên sẽ nhận thức được tầm quan trọng của việc lựa chọn cấu trúc dữ liệu và thuật toán phù hợp trong việc xây dựng các hệ thống phần mềm có hiệu năng cao, khả năng mở rộng và bảo trì dễ dàng.

# **CHƯƠNG 3. ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY**

1. **Khái niệm và cách thức hoạt động của đệ quy**
2. **Đệ quy (Recursion algorithm)**

* Đệ quylà một phương pháp lập trình, trong đó một hàm gọi lại chính nó để giải quyết bài toán bằng cách chia một bài toán lớn thành nhiều bài toán nhỏ giống hệt nhau.
* Nếu lời giải của bài toán T được thực hiện bởi lời giải của bài toán T’ có dạng như T thì đó là một lời giải đệ quy. Giải thuật tương ứng với lời gọi như vậy được gọi là giải thuật đệ quy (T’ < T).
* Đệ quy có 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 rục này lại chứa lời gọi đến chính nó.

1. **Cách thức hoạt động của đệ quy:**

* Điều kiện dừng (Base case): đây là trường hợp đặc biệt không cần phải gọi đệ quy nữa, điều này giúp cho chương trình kết thúc và trả về kết quả trực tiếp mà không bị rơi vào vòng lặp vô hạn.
* Trường hợp đệ quy: đây là phần mà hàm tự gọi lại chính nó với một phiên bản nhỏ hơn của bài toán để giải quyết bài toán.
* Thực thi hàm: khi hàm được gọi, hệ thống sẽ tạo ra một khung ngăn xếp (stack frame) chứa các biến cục bộ và các giá trị cần thiết. Hàm kiểm tra điều kiện dừng, nếu thỏa mãn hàm sẽ dừng và trả về kết quả. Nếu không thỏa mãn hàm tiếp tục gọi lại chính nó với giá trị nhỏ hơn. Mỗi lần như vậy, một khung ngăn xếp mới sẽ được tạo ra và thêm vào ngăn xếp.
* Quay lui (Backtracking): khi hàm đệ quy đạt đến điều kiện dừng nó sẽ bắt đầu quay lui. Tại mỗi bước quay lui, kết quả sẽ được trả về cho lời gọi đệ quy trước đó và tiếp tục như vậy cho đến khi quay về lời gọi ban đầu.

1. **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

1. **Giải thuật đệ quy**
   1. **Tính giai thừa của một số n**

|  |
| --- |
| long Fact(int n){  if(n == 0) {  return 1;  }  else return n\*Fact(n-1);  } |

* Khi gọi hàm *Fact(n)* chương trình sẽ hoạt động như sau:

*Fact(n)* sẽ gọi tới *Fact(n-1)* và *Fact(n-1)* sẽ gọi tới *Fact(n-2)* cứ như vậy cho đến khi *Fact(0)*, đây chính là điều kiện dừng của hàm đệ quy tính giai thừa.

* 1. **Giải thuật đệ quy tính số Fibonaci**

|  |
| --- |
| //Ham de quy tinh Fibonaci  int fibonacci(int n) {  if (n == 0)  return 0;  else if (n == 1)  return 1;  else // Loi giai de quy  return fibonacci(n - 1) + fibonacci(n - 2);  } |

* Điều kiện dừng:
* Nếu n == 0, hàm trả về 0 vì Fibonacci(0) = 0
* Nếu n == 1, hàm trả về 1 vì Fibonacci(1) = 1
* Khi gọi Fibonacci(n), chương trình sẽ hoạt động như sau:
* Gọi chính nó: Fibonacci(n) sẽ gọi hai hàm con Fibonacci(n-1) và Fibonacci(n-2). Quá trình này lặp lại đệ quy, tiếp tục gọi hai hàm con cho đến khi đạt điều kiện dừng
* Quay ngược: khi đạt điều kiện dừng, các giá trị trả về sẽ được gộp lại tạo thành kết quả.
  1. **Giải thuật đệ quy giải bài toán tháp Hà Nội**

|  |
| --- |
| void hanoi(int n, char A, char B, char C) {  if (n == 1) {  cout << "Di chuyen dia 1 tu coc " << A << " sang coc " << C << endl;  return;  }  // Buoc 1: Di chuyen n-1 dia tu coc A sang coc B, su dung C lam coc trung gian  hanoi(n - 1, A, C, B);  // Buoc 2: Di chuyen dia thu n tu coc A sang coc C  cout << "Di chuyen dia " << n << " tu coc " << A << " sang coc " << C << endl;  // Buoc 3: Di chuyen n-1 dia tu coc B sang coc C, su dung A lam coc trung gian  hanoi(n - 1, B, A, C);  } |

**Hoạt động của hàm:**

* Điều kiện dừng: nếu chỉ còn 1 đĩa (n == 1), di chuyển trực tiếp đĩa này từ cọc A sang cọc C rồi dừng lại.
* Khi gọi đệ quy
* Bước 1: Di chuyển n-1 đĩa từ cọc A sang cọc B, sử dụng C làm trung gian
* Bước 2: Di chuyển đĩa lớn nhất (đĩa thứ n) từ cọc A sang cọc C.
* Bước 3: Di chuyển n-1 đĩa từ cọc B sang cọc C, sử dụng A làm trung gian

**Chi tiết từng bước:**

* Di chuyển n-1 đĩa từ A sang B:

Di chuyển đĩa 1 từ A sang C

Di chuyển đĩa 2 từ A sang B

Di chuyển đĩa 1 từ C sang B

* Di chuyển đĩa lớn nhất (3) từ A sang C
* Di chuyển n-1 đĩa từ B sang C:

Di chuyển đĩa 1 từ B sang A

Di chuyển đĩa 2 từ B sang C

Di chuyển đĩa 1 từ A sang C

**Độ phức tạp của thuật toán:**

* Mỗi lời gọi Hanoi cho n đĩa bao gồm 2 lời gọi đệ quy cho n-1 đĩa
* Độ phức tạp thời gian: T(n) = 2T(n-1) + O(1) => T(n) = O(2n)
* Độ phức tạp không gian: O(n)

# **CHƯƠNG 4. MẢNG VÀ DANH SÁCH**

1. **Mảng**
2. **Khái niệm mảng**

* Mảng (Array) là một tập có thứ tự gồm một số cố định các phần tử có cùng kiểu dữ liệu được sắp xếp liên tiếp trong bộ nhớ. Không có phép bổ sung phần tử hoặc loại bỏ phần tử được thực hiện đối với mảng. Thường chỉ có các phép tạo lập (create) mảng, tìm kiếm (retrieve) giá trị, một phần tử của mảng còn được đặc trưng bởi chỉ số (index) thể hiện thứ tự của phần đó trong mảng.

1. **Ưu điểm và hạn chế của mảng**
   1. **Ưu điểm của mảng**

* 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.
  1. **Hạn chế của mảng**
* 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.

1. **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.

1. **Các thao tác trên mảng**
   1. **Nhập mảng**

|  |
| --- |
| void nhapmang(int arr[], int &n) {  cout << "Nhap so luong phan tu cua mang: ";  cin >> n; // Cho phep nguoi dung nhap so luong phan tu  cout << "Nhap cac phan tu cua mang: " << endl;  for (int i = 0; i < n; i++) {  cout << "a[" << i << "] = ";  cin >> arr[i];  }  cout << endl;  } |

* Hàm nhapmang() đầu tiên sẽ nhập số lượng phần tử cần có trong mảng qua tham chiếu n
* Sử dụng vòng lặp for để nhập từng giá trị của các phần tử trong mảng
  1. **Tìm kiếm phần tử trong mảng**

|  |
| --- |
| int timKiem(int arr[], int n, int x) {  for (int i = 0; i < n; i++) {  if (arr[i] == x)  return i; // Tra ve vi tri tim thay  }  return -1; // Tra ve -1 neu khong tim thay  } |

* Hàm này sử dụng một vòng lặp for để duyệt qua từng phần tử của mảng. Tại mỗi vòng lặp sẽ kiểm tra xem phần tử arr[i] nào có giá trị bằng giá trị cần tìm
* Nếu tìm thấy giá trị cần tìm, hàm sẽ trả về giá trị i của phần tử đó. Nếu duyệt hết mảng mà không tìm thấy giá trị x, hàm sẽ trả về giá trị -1.
  1. **Thêm phần tử vào mảng**

|  |
| --- |
| void chenPhanTu(int arr[], int& n, int value, int pos) {  if (pos < 0 || pos > n) {  cout << "Vi tri khong hop le!" << endl;  return;  }  for (int i = n; i > pos; i--) {  arr[i] = arr[i - 1];  }  arr[pos] = value;  n++;  } |

* Cách hoạt động của hàm chèn: đầu tiên sẽ kiểm tra vị trí muốn chèn phần tử mới vào. Nếu pos (vị trị muốn chèn phần tử mới vào) < 0 hoặc > n (số lượng phần tử hiện tại của mảng) thì hàm sẽ in ra thông báo “vị trí không hợp lệ” và dừng.
* Nếu vị trí muốn chèn thỏa mãn điều kiện trên, hàm sẽ sử dụng vòng lặp for để dịch chuyển các phần tử từ vị trí pos trở về cuối mảng sang phải để tạo khoảng trống cho phần tử mới. Mỗi phần tử tại vị trí i-1 được gán sang i bắt đầu từ vị trí cuối mảng.
* Arr[pos] = value: chèn giá trị vào vị trí pos trong mảng sau đó tăng kích thược của n lên 1.
  1. **Xóa phần tử khỏi mảng**

|  |
| --- |
| void xoaPhanTu(int arr[], int& n, int pos) {  if (pos < 0 || pos >= n) {  cout << "Vi tri khong hop le!" << endl;  return;  }  for (int i = pos; i < n - 1; i++) {  arr[i] = arr[i + 1];  }  n--;  } |

* Hàm xoaPhanTu() đầu tiên cũng kiểm tra vị trí cần xóa có hợp lệ không. Nếu vị trí cần xóa (pos < 0 và pos > n) thì hàm sẽ in ra “Vi tri không hop le” và dừng
* Nếu vị trí cần xóa thỏa mãn điều kiện, hàm sẽ sử dụng một vòng lặp for để duyệt qua tất cả các phần tử trong mảng để tìm phần tử cần xóa, mảng sẽ giảm kích thước
  1. **Sắp xếp mảng**

|  |
| --- |
| void sapXepTangDan(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]) {  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  } |

1. **Danh sách**
2. **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 kích 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.

1. **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.

1. **Cấu trúc lưu trữ của danh sách**
   1. **Stack hay danh sách kiểu ngăn xếp**

* Cấu trúc của stack:
* 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
  1. **Giải thuật của Stack**
     1. **Khởi tạo stack rỗng:**

|  |
| --- |
| private:  int\* StkArr; // Mang luu tru cac phan tu cua stack  int StkTop; // Dinh cua stack  int StkMax; // Kich thuoc toi da cua stack  public:  Cstack(int size) {  StkArr = new int[size]; // Cap phat mang  StkMax = size; // Gan kick thuoc toi da  StkTop = -1; // Stack rong ban dau  } |

* + 1. **Kiểm tra stack rỗng**

|  |
| --- |
| Boolean isEmpty() {  If(StkTop == -1) return true; // Stack rong  Return false; //Stack khong rong  } |

* + 1. **Kiểm tra stack đầy**

|  |
| --- |
| Boolean isFull(){  If(StkTop == StkMax – 1)  Return true; //Stack day  Return False;//Stack khong day  } |

* + 1. **Thao tác Pop lấy ra một phần tử từ đỉnh Stack**

|  |
| --- |
| bool Pop(int outitem) {  if (isEmpty()) {  cout << "Stack rong, khong lay duoc phan tu" << endl;  return false; // Stack rong, khong lay ra duoc  } else {  outitem = StkArr[StkTop];  StkTop--;  cout << "Da lay duoc " << outitem << " ra khoi stack." << endl;  return true; // Lay ra thanh cong  }  } |

* Khi gọi hàm Pop() để lấy ra một phần tử từ đỉnh stack: đầu tiên sẽ gọi hàm isEmpty() để kiểm tra xem stack có rỗng không, nếu stack rỗng thì sẽ in ra màn hình một thông báo và trả về false không lấy phần tử ra được
* Nếu stack không rỗng, gán giá trị của phần tử trên đỉnh stack StkArr[StkTop] cho biến outitem sau đó giảm chỉ số đỉnh stack Stktop- - để loại bỏ phần tử vừa lấy ra và in ra màn hình thông báo
  + 1. **Thao tác push thêm một phần tử vào stack**

|  |
| --- |
| bool Push(int initem) {  if (isFull()) {  cout << "Stack da day, khong them duoc phan tu vao stack" << endl;  return false;  } else {  StkTop++; // Tang stack  StkArr[StkTop] = initem; // Them phan tu vao danh sach  cout << "Da them " << initem << " thanh cong vao stack!" << endl;  return true; // Them thanh cong  }  } |

* Khi gọi hàm Push() thêm một phần tử vào satck bước đầu tiên gọi hàm isFull() để kiểm tra stack đầy hay chưa nếu stack đầy thì in ra màn hình thông báo và trả về false không thể thêm
* Nếu stack chưa đầy, tăng chỉ số đỉnh stack StkTop++ để di chuyển đến vị trí trống tiếp theo trong stack sau đó gán giá trị initem phần tử cần thêm vào vị trí tương ứng trong mảng lưu trữ stack StkArr[StkTop] và in ra màn hình thông báo rồi kết thúc.
  1. **Bài tập stack**
* Chuyển đổi số thập phân sang nhị phân sử dụng cơ chế stack

|  |
| --- |
| #include <iostream>  #include <stack>  using namespace std;  // Ham chuyen doi so nguyen sang so nhi phan theo co che stack  void decimalToBinary(int n) {  stack<int> binaryStack; // Tao stack luu tru phan du  // Chia 2 va luu phan du vao stack  while (n > 0) {  int remainder = n % 2;  binaryStack.push(remainder); //Day phan du vao stack  n = n / 2;  }  // In ra kq nhi phan bang cach lay phan tu ra khoi Stack  cout << "So nhi phan la: ";  while (!binaryStack.empty()) {  cout << binaryStack.top(); // Lay phan tu tren dinh stack  binaryStack.pop(); // Xoa phan tu khoi stack  }  cout << endl;  }  int main() {  int n;  cout<<"Nhap so thap phan n = ";  cin>>n;  decimalToBinary(n); // Goi ham chuyen doi va in ket qua ra man hinh  return 0;  } |

* 1. **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:
* 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
  1. **Giải thuật của Queue**
     1. **Khai báo lớp Queue:**

|  |
| --- |
| class Queue{  private:  int\* QArray;  int QMax;  int QNumItems;  int QFront;  int QRear;  public:  Queue (int size){  QArray = new int [size];  QMax = size;  QFront = QRear = -1;//queue rong  QNumItems = 0;//chua co phan tu nao trong queue  } |

* + 1. **Kiểm tra queue rỗng:**

|  |
| --- |
| Boolean isEMpty(){  If(QNumItems == 0)  Return true;//Queue rong  Return false; //Queue khong rong  } |

* + 1. **Kiểm tra queue đầy:**

|  |
| --- |
| Boolean isFull(){  If(QNumItems == QMax)  Return true; //Queue day  Return false; //Queue khong day  } |

* + 1. **Thêm một phần tử vào queue:**

|  |
| --- |
| bool append (int newitem){  if(isFull()){  cout<<"\nQueue day, khong them duoc phan tu vao queue"<<endl;  return false;  }else{  QRear++;  QArray[QRear] = newitem;//them phan tu vao cuoi queue  QNumItems++;  cout<<"\nDa them "<<newitem<<" thanh cong vao Stack";  return true;//them thanh cong  }  } |

* Hàm thêm một phần tử vào queue append(): đầu tiên sẽ kiểm tra queue có đầy không, nếu queue đầy hàm sẽ in ra thông báo và dừng.
* Nếu queue không đầy, tăng giá trị chỉ số cuối QRear lên 1 để trỏ tới vị trí tiếp theo trong mảng sau đó gán giá trị của chỉ số mới newitem vào vị trí QArray[QRear].
* Cuối cùng là tăng số lượng phần tử trong hàng đợi QNumItems lên 1
  + 1. **Xóa một phần từ khỏi queue:**

|  |
| --- |
| bool take(int itemout){  if(isEmpty()){  cout<<"\nQueue rong, khong lay ra duoc phan tu tu queue";  return false;  }else{  itemout = QArray[QFront];//Lay ptu dau ra  QFront++;  QNumItems--;  if(QFront == QMax)  QFront = QRear = -1;//Quay tro ve dau mang  return true;//lay thanh cong  }  } |

* Hàm xóa một phần tử khỏi queue take(): đầu tiên sẽ kiểm tra xem queue có rỗng không (isEmpty() ). Nếu queue rỗng thì sẽ in ra thông báo và dừng.
* Nếu queue không rỗng: gán giá trị đầu tiên trong hàng đơi (QArray[QFront] vào biến itemout sau đó tăng chỉ số đầu hàng đợi (QFront) lên 1 để trỏ đến phân tử tiếp theo.
* Sau khi đã xóa sẽ cập nhật lại trạng thái hàng đợi: giảm số lượng phần tử hiện có (QNumItems). Nếu chỉ số QFront đã vượt quá kích thước tối đa của mảng (QMax) thì đăt lại trạng thái hàng đợi. QFront = QRear = -1 đặt hàng đợi về trạng thái rỗng và chuẩn bị cho việc sử dụng lại từ đầu mảng.

**CHƯƠNG 5. DANH SÁCH MÓC NỐI (LINKLIST)**

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.

1. **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.

1. **Các loại danh sách móc nối**
   1. **Danh sách móc nối đơn**

* Danh sách móc nối đơn là danh sách có nhiều nút móc nối đơn lại với nhau. 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
  1. **Các thuật toán của danh sách móc nối đơn**
     1. **Xóa một nút ở cuối danh sách móc nối đơn**

|  |
| --- |
| void delete\_cuoi() {  if (L == NULL) {  cout << "Danh sach trong\n";  } else if (L->next == NULL) { // Neu chua co mot phan tu  delete L;  L = NULL;  } else {  Node\* temp = L;  Node\* prev = NULL;  while (temp->next != NULL) {  prev = temp;  temp = temp->next; // Tim nut cuoi  }  prev->next = NULL;  delete temp; // Xoa nut cuoi  }  } |

* Hàm xóa một phần tử cuối delete\_cuoi(): đầu tiên sẽ kiểm tra danh sách liên kết, nếu trống sẽ in ra thông báo và dừng.
* Nếu danh sách liên kết có một phần tử, xóa phần tử đó và đặt L == NULL, danh sách rỗng.
* Nếu có nhiều phần tử: ta sử dung hai con trỏ temp duyệt qua từng nút trong danh sách và prev nút liền trước nút cuối. Sử dụng vòng lặp while để tìm nút cuối danh sách. Tại mỗi lần lặp, gắn prev = temp và temp = nút kế tiếp (temp ->next). Khi con trỏ temp đã đến vị trí cuối cùng ta sẽ gán prev->next = NULL và xóa temp, phần tử cuối của danh sách liên kết đơn và dừng lại.
  + 1. **Xóa một nút ở đầu danh sách móc nối đơn**

|  |
| --- |
| void delete\_dau() {  if (L == NULL) {  cout << "Danh sach trong\n";  } else {  Node\* temp = L;  L = L->next;  delete temp;  }  } |

* Hàm xóa nút đầu danh sách delete\_dau(): đầu tiên sẽ kiểm tra xem danh sách có rỗng không. Nếu rỗng thì in ra thông báo và dừng lại
* Nếu không rỗng, sử dụng con trỏ temp gán nó cho phần tử đầu tiên của danh sách L và gán L = L->next phần tử tiếp theo sau đó xóa temp.
  + 1. **Thêm một nút vào cuối danh sách móc nối đơn**

|  |
| --- |
| 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;  }  } |

* Hàm chèn một nút vào cuối danh sách insert\_cuoi(): tạo một nút mới newNode gán cho nó giá trị muốn thêm vào danh sách createNode(value). Nếu danh sách rỗng, đăt nút mới làm nút đâu tiên của danh sách luôn L = newNode
* Nếu danh sách không rỗng, sử dụng con trỏ temp gán nó bằng phần tử đầu tiên sau đó dùng vòng lặp while di chuyển temp đến nút cuối có con trỏ next = NULL
* Khi đã tìm được nút cuối, gán temp->next = newNode để liên kết nút cuối với nút mới
  + 1. **Thêm một nút vào đầu danh sách móc nối đơn**

|  |
| --- |
| void insert\_dau(int value) {  Node\* newNode = createNode(value);  if (L == NULL) {  L = newNode;  } else {  newNode->next = L;  L = newNode;  }  } |

* Insert\_dau() chèn nút vào đầu danh sách: đầu tiên tgọi hàm createNode(value) để tạo một nút mới newNode. Sau đó kiểm tra danh sách có rỗng không, nếu rỗng thì gán L = newNode làm nút đầu tiên của danh sách.
* Nếu không rỗng, gán con trỏ newNode->next = L liên kết nút mới newNode với nút đầu tiên hiện tại L sau đó cập nhập con trỏ L để trỏ tới newNode để nó thành nút đầu tiên
  + 1. **Xóa nút đầu tiên có giá trị x trong danh sách móc nối đơn**

|  |
| --- |
| void delete\_value(int value) {  if (L == NULL) {  cout << "Danh sach trong\n";  return;  }  if (L->data == value) { // Xoa nut gia tri dau danh sach  Node\* temp = L;  L = L->next;  delete temp;  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;  } |

* Delete\_value() hàm xóa nút có giá trị x: đầu tiên kiểm tra xem danh sách có rỗng không, nếu danh sách rỗng in ra thông báo và kết thúc, nếu danh sách không rỗng tiếp tục thực hiện
* Kiểm tra nút đầu danh sách L->data == value), nếu giá trị đầu bằng value, tạo con trỏ tạm thời temp trỏ tới nút đầu temp = L sau đó cập nhật con trỏ L trỏ đến nút tiếp theo L = L->next rồi xóa nút đầu và kết thúc hàm
* Nếu nút cần xóa nằm bên trong danh sách, sử dụng 2 con trỏ temp để duyệt qua danh sách và prev để trỏ đến nút liền trước temp, sử dụng vòng lặp while duyệt qua từng nút trong danh sách, nếu giá trị của nút hiện tại temp->data khác value tiếp tục cập nhật prev = temp và di chuyển temp đến nút tiếp theo
* Khi đã duyệt hết danh sách mà không tìm thấy giá trị value cần xóa thì in ra thông báo và kết thúc. Nếu tìm thấy nút cần xóa, liên kết nút trước nút cần cóa prev->next với nút sau nút cần xóa temp->next và xóa nút cần xóa bang delete temp.
  + 1. **Chèn một phần tử sau một giá trị cho trước**

|  |
| --- |
| void insertAfter(int target, int value) {  Node\* temp = head;  while (temp != NULL && temp->data != target) {  temp = temp->next;  }  if (temp != NULL) { // Tim thay gia tri can chen sau  Node\* newNode = new Node(value);  newNode->next = temp->next;  temp->next = newNode;  } else {  cout << "Gia tri " << target << " khong ton tai trong danh sach." << endl;  }  } |

* 1. **Danh sách liên kết đôi**
* Danh sách móc nối đôi 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
  1. **Các thuật toán của danh sách kết nối đôi**
     1. **Thêm node vào đầu danh sách liên kết đôi**

|  |
| --- |
| void insertAtHead(Node\*& head, int newData) {  Node\* newNode = new Node();  newNode->data = newData;  newNode->next = head;  newNode->prev = NULL;  if (head != NULL) {  head->prev = newNode;  }  head = newNode;  } |

* Dùng toán tử new để tạo một nút mới newNode trong bộ nhớ động, gán giá trị newData vào trương data của nút mới newNode->data = newData rồi đặt con trỏ next của nút mới trỏ tới nút đầu hiện tại của danh sách newNode->next = head. Đặt con trỏ prev của nút mới là NULL, nút mới sẽ trở thành nút đầu tiên không có nút trước newNode->prev = NULL
* Kiểm tra nếu danh sách rỗng head != NULL, cập nhật con trỏ của nút hiện tại để trỏ tới nút mới, nút mới sẽ trở thành nút đầu tiên của danh sách head->prev = newNode
  + 1. **Xóa một node khỏi danh sách liên kết đôi**

|  |
| --- |
| void deleteNode(Node\*& head, int value) {  if (head == NULL) {  cout << "Danh sach rong, khong the xoa." << endl;  return;  }  Node\* current = head;  // Duyet tim node chua gia tri can xoa  while (current != NULL && current->data != value) {  current = current->next;  }  // Neu khong tim thay gia tri  if (current == NULL) {  cout << "khong tim thay gia tri " << value << " trong danh sach." << endl;  return;  }  // Truong hop node can xoa la node dau  if (current == head) {  head = current->next; // Cap nhat head  if (head != NULL) {  head->prev = NULL; // Neu con node phia sau, cap nhat prev thành NULL  }  }  // Truong hop node nam giua hoac cuoi  else {  if (current->next != NULL) { // Neu khong phai node cuoi  current->next->prev = current->prev;  }  if (current->prev != NULL) { // Neu khong phai node dau  current->prev->next = current->next;  }  }  // Giai phong node can xoa  delete current;  cout << "Xoa thanh cong node co gia tri " << value << "." << endl;  } |

* 1. **Danh sách móc nối vòng**
* Danh sách móc nối vòng là một loại danh sách liên kết trong đó phần tử cuối cùng không trỏ tới NULL mà trỏ ngược lại phần tử đầu tiên của danh sách, tạo thành một vòng lặp. Vì vậy không có điểm bắt đầu và điểm kết thúc cố định và ta có thể di chuyển từ phần tử cuối về phần tử đầu mà không cần ngắt đoạn
* Đặc điểm của danh sách móc nối vòng:
* Kết nối tuần hoàn: phần tử cuối cùng của danh sách trỏ lại phần tử đầu tiên tạo thành một vòng khép kín
* Không có nút đầu và nút cuối rõ rang: vì danh sách là vòng tròn nên việc định nghĩa phần tử đầu hoặc cuối thường là tùy chọn
* Linh hoạt trong thao tác duyệt: khi duyệt danh sách ta có thể quay lại phần tử đầu tiên sau khi đã duyệt qua hết các phần tử mà không phải xử lý điều kiện kết thúc như trong danh sách liên kết đơn thông thường
  1. **Các thuật toán của danh sách kết nối vòng**
     1. **Hàm chèn thêm một phần tử vào cuối danh sách liên kết vòng**

|  |
| --- |
| void insert(Node\*\* head, int newData) {  Node\* newNode = new Node();  newNode->data = newData;  newNode->next = \*head;  if (\*head == NULL) { // Neu danh sach trong  \*head = newNode;  newNode->next = \*head;  } else { // Neu danh sach da co phan tu  Node\* temp = \*head;  while (temp->next != \*head) {  temp = temp->next;  }  temp->next = newNode;  newNode->next = \*head;  }  } |

* + 1. **Hàm thêm phần tử vào vị trí bất kỳ trong danh sách**

|  |
| --- |
| void addElement(Node\*\* head, int newData, int pos) {  Node\* newNode = new Node();  newNode->data = newData;  if (\*head == NULL) {  \*head = newNode;  newNode->next = \*head;  } else if (pos == 0) {  Node\* temp = \*head;  while (temp->next != \*head) {  temp = temp->next;  }  temp->next = newNode;  newNode->next = \*head;  \*head = newNode;  } else {  Node\* temp = \*head;  for (int i = 1; i < pos && temp->next != \*head; i++) {  temp = temp->next;  }  newNode->next = temp->next;  temp->next = newNode;  }  } |

* + 1. **Hàm xóa một phần tử có giá trị x trong danh sách liên kết vòng**

|  |
| --- |
| void deleteElement(Node\*\* head, int x) {  if (\*head == NULL) return;  Node\* temp = \*head;  Node\* prev = NULL;  // Neu phan tu dau la x  if ((\*head)->data == x) {  while (temp->next != \*head) {  temp = temp->next;  }  Node\* delNode = \*head;  temp->next = (\*head)->next;  \*head = (\*head)->next;  delete delNode;  return;  }  // Tim phan tu co gia tri x va xoa no  do {  prev = temp;  temp = temp->next;  } while (temp != \*head && temp->data != x);  if (temp->data == x) {  prev->next = temp->next;  delete temp;  } else {  cout << "Khong tim thay phan tu co gia tri " << x << " de xoa.\n";  }  } |

# **CHƯƠNG 6. CÂY**

1. **Khái niệm cây**

* Cây là tập hữu hạn các nút (tree node), sao cho có một nút gọi là nút gốc (root), các nút còn lại được phân hoạch thành n tập riêng biết T1, T2,…Tn, mỗi tập Ti là một cây. Giữa các nút có mối quan hệ phân cấp gọi là quan hệ cha con
* Cây không có nút gọi là cây rỗng (Null tree)

1. **Các loại cây:** 
   1. **Cây nhị phân (binary tree)**

* Cây nhị phân là một dạng quan trọng của cấu trúc cây. Nó có đặc điểm là mọi nút trên cây chỉ có tối đa là hai con. Đối với cây con của một nút người ta cũng phân biệt cây con trái và cây con phải.
  1. **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
  1. **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

1. **Các thuật toán của cây**
   1. **Các phép duyệt cây nhị phân**

* Phép duyệt cây là phép xử lý các nút trên cây, mỗi nút một lần
  + 1. **Duyệt cây theo thứ tự trước (NLR)**
* Thăm gốc (root)
* Duyệt cây con trái theo thứ tự trước
* Duyệt cây con phải theo thứ tự trước
* Giải thuật:

|  |
| --- |
| Void preorder (Node\* root)  {  if root != NULL){  cout<<root->data << " ";  Preorder (Root->left);  Preorder (Root->right);  }  } |

* + 1. **Duyệt cây nhị phân theo thứ tự giữa (LNR)**
* Duyệt cây con trái theo thứ tự giữa
* Thăm gốc
* Duyệt cây con phải theo thứ tự giữa
* Giải thuật

|  |
| --- |
| Void inorder(Node\* root)  {  If(root != NULL){  Inorder(root->left);  Cout<<root->data << " ";  Inorder(root->right);  }  } |

* + 1. **Duyệt cây nhị phân theo thứ tự sau (LRN)**
* Duyệt cây con trái theo thứ tự sau
* Duyệt cây con phải theo thứ tự sau
* Thăm gốc
* Giải thuật:

|  |
| --- |
| Void postorder(Node\* root)  {  If(root != NULL){  Postorder (root->left);  Postorder (root->right);  Cout<<root->data << " ";  }  } |

* 1. **Các thao tác trên cây nhị phân tìm kiếm (binary search tree – BST)**
     1. **Khai báo cây**

|  |
| --- |
| class BSTNode {  public:  int data;  BSTNode \*left;  BSTNode \*right;  BSTNode(int value) {  data = value;  left = NULL;  right = NULL;  }  } |

* + 1. **Thêm một phần tử vào cây nhị phân tìm kiếm**

|  |
| --- |
| int insert(int x, BSTNode\*& root) {  if (root == NULL) {  root = new BSTNode(x); // Cap phat bo nho cho node moi  if (root == NULL) {  return -1; // Khong the cap phat bo nho  } else {  return 1; // Them thanh cong  }  } else {  if (root->data == x) {  return 0; // Gia tri da ton tai trong cay  } else if (x < root->data) {  return insert(x, root->left); // Di vao nhanh trai de them  } else {  return insert(x, root->right); // Di vao nhanh phai de them  }  }  } |

* Đầu tiên kiểm tra nếu cây rỗng thì tạo một nút mới với giá trị x làm nút gốc và trả về 1 nếu tạo thành công, nếu tạo không thành công trả về -1.
* Kiểm tra nếu cây không rỗng, nếu giá trị đã tồn tại trong cây thì trả về 0, nếu giá trị nhỏ hơn nút gốc thì tiến vào nhánh trái của cây, nếu giá trị lớn hơn nút gốc thì tiến vào nhánh phải của cây.
  + 1. **Xóa một nút khỏi cây nhị phân tìm kiếm**

|  |
| --- |
| int Delete(int x, BSTNode\*& root) {  BSTNode\* p;  if (root == NULL)  return 0; // Cay rong, khong tim thay  else if (x < root->data) // Xoa tren cay con trai  return Delete(x, root->left);  else if (x > root->data) // Xoa tren cay con phai  return Delete(x, root->right);  else { // Tim ra nut can xoa  if (root->left && root->right) { // Co 2 con  p = Findmax(root->left); // Tim nut co khoa lon nhat tren cay con trai  root->data = p->data;  return Delete(root->data, root->left);  } else { // Co 1 con hoac khong co con  p = root;  if (root->left != NULL) // Xu ly neu chi co con trai  root = root->left;  else if (root->right != NULL) // Xu ly neu chi co con phai  root = root->right;  else  root = NULL;  delete p;  }  }  return 1;  } |

* + 1. **Tìm nút có khóa X**

|  |
| --- |
| BSTNode\* Find(int x, BSTNode\* root) {  if (root == NULL) return NULL;  if (x < root->data) {  return Find(x, root->left); // Tim o nhanh trai  } else if (x > root->data) {  return Find(x, root->right); // Tim o nhanh phai  } else {  return root; // Da tim thay  }  } |

* + 1. **Tìm một nút có khóa nhỏ nhất**

|  |
| --- |
| BSTNode\* Findmin(BSTNode\* root) {  if (root == NULL) return NULL;  else if (root->left == NULL) return root;  else return Findmin(root->left);  } |

* + 1. **Tìm một nút có khóa lớn nhất**

|  |
| --- |
| BSTNode\* Findmax(BSTNode\* root) {  if (root != NULL) {  while (root->right != NULL) {  root = root->right;  }  }  return root;  } |

* + 1. **Xóa một nút có khóa x trong cây**

|  |
| --- |
| int Delete(int x, BSTNode\*& root) {  BSTNode\* p;  if (root == NULL)  return 0; // Cay rong, khong tim thay  else if (x < root->data) // Xoa tren cay con trai  return Delete(x, root->left);  else if (x > root->data) // Xoa tren cay con phai  return Delete(x, root->right);  else { // Tim ra nut can xoa  if (root->left && root->right) { // Co 2 con  p = Findmax(root->left); // Tim nut co khoa lon nhat tren cay con trai  root->data = p->data;  return Delete(root->data, root->left);  } else { // Co 1 con hoac khong co con  p = root;  if (root->left != NULL) // Xu ly neu chi co con trai  root = root->left;  else if (root->right != NULL) // Xu ly neu chi co con phai  root = root->right;  else  root = NULL;  delete p;  }  }  return 1;  } |

* 1. **Các thao tác trên cây tổng quát**
     1. **Thêm một nút mới vào cây nhị phân**

|  |
| --- |
| void addNode(Node\*& root, int value) {  Node\* newNode = new Node(value);  if (root == NULL) {  root = newNode; // Neu cay rong, gan nut moi lam goc  return;  }  queue<Node\*> q;  q.push(root);  while (!q.empty()) {  Node\* temp = q.front();  q.pop();  if (temp->left == NULL) {  temp->left = newNode;  return;  } else {  q.push(temp->left);  }  if (temp->right == NULL) {  temp->right = newNode;  return;  } else {  q.push(temp->right);  }  }  } |

* + 1. **Xóa một nút ra khỏi cây nhị phân**

|  |
| --- |
| void deleteNode(Node\*& root, int value) {  if (root == NULL) {  cout << "Cay rong, khong the xoa!" << endl;  return;  }  queue<Node\*> q;  q.push(root);  Node\* targetNode = NULL; // Nut can xoa  Node\* lastNode = NULL; // Nut cuoi cung  Node\* parentOfLast = NULL; // Cha cua nut cuoi cung  while (!q.empty()) {  Node\* temp = q.front();  q.pop();  if (temp->data == value) {  targetNode = temp;  }  if (temp->left) {  parentOfLast = temp;  q.push(temp->left);  }  if (temp->right) {  parentOfLast = temp;  q.push(temp->right);  }  lastNode = temp;  }  if (targetNode == NULL) {  cout << "Khong tim thay gia tri can xoa!" << endl;  return;  }  // Thay the gia tri cua targetNode bang gia tri cua lastNode  targetNode->data = lastNode->data;  // Xoa lastNode khoi cay  if (parentOfLast->left == lastNode) {  delete parentOfLast->left;  parentOfLast->left = NULL;  } else if (parentOfLast->right == lastNode) {  delete parentOfLast->right;  parentOfLast->right = NULL;  } else {  // Neu nut cuoi chinh la goc  delete root;  root = NULL;  }  } |

* + 1. **Duyệt cây theo thứ tự trước**

|  |
| --- |
| void preorderTraversal(Node\* root) {  if (root == NULL) return;  cout << root->data << " ";  preorderTraversal(root->left);  preorderTraversal(root->right);  } |

* + 1. **Duyệt cây theo thứ tự giữa**

|  |
| --- |
| void inorderTraversal(Node\* root) {  if (root == NULL) return;  inorderTraversal(root->left);  cout << root->data << " ";  inorderTraversal(root->right);  } |

* + 1. **Duyệt cây theo thứ tự sau**

|  |
| --- |
| void postorderTraversal(Node\* root) {  if (root == NULL) return;  postorderTraversal(root->left);  postorderTraversal(root->right);  cout << root->data << " ";  } |

# **CHƯƠNG 7. ĐỒ THỊ**

1. **Định nghĩa đồ thị và các khái niệm**
2. **Định nghĩa 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 cặp đỉnh thuộc E thì ta nói: có một cung nối v1 và v2. Nếu cung (v1, v2) khác 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 v1, v2. Nếu thứ tự các nút trên cung không được coi trong thì ta gọi đồ thị không định hướng (undirected graph).

1. **Phân loại đồ thị**

* Có ba loại đồ thị:
* Đồ thị vô hướng: là đồ thị trong đó các cạnh không có hướng
* Đồ thị có hướng: là đồ thị mà các cạnh có hướng, mỗi cạnh đi từ một đỉnh đầu đến đỉnh cuối
* Đồ thị trọng số: là đồ thị mà các cạnh có một giá trị hay trọng số, biểu thị đồ dài, chi phí hoặc độ quan trọng của liên kết giữa các đỉnh

1. **Các khái niệm**

* Nếu (v1, v2) là một cung trong tập E(G) thì v1, v2 gọi là lân cận của nhau (adjacent)
* Một đường đi (path) từ đỉnh vp đến đỉnh vq trong đồ thị G là một dãy đỉnh vp, vi1, vi2,…, vin, vq mà (vp, vi1), (vi1, vi2)…, (vin, vq) là các xung trong E(G). Số lượng các cung trên đường đi ấy gọi là độ dài của đường đi (path length)
* Một đường đi đơn (simple path) là đường đi mà mọi đỉnh trên đó, trừ đỉnh đầu và đỉnh cuối, đều khác nhau
* Một chu trình (cycle) là một đường đi đơn mà đỉnh đầu và đỉnh cuối trùng nha
* Tính liên thông:
* Trong đồ thị G , hai đỉnh x, y gọi là liên thông (connected) nếu có một đường từ x đến y
* Đồ thị G liên thông nếu với mọi (x, y) thuộc E, tồn tại đường đi từ x đến y
* Đồ thị G gọi là trọng số nếu mỗi cung được gán một giá trị số đặc trưng

1. **Biểu diễn đồ thị** 
   1. **Biểu diễn đồ thị bằng ma trận kề**

* Ma trận kề là một phương pháp biểu diễn đồ thị bằng cách sử dụng một ma trận vuông kích thước n×n (với n là số lượng đỉnh của đồ thị). Các phần tử trong ma trận biểu diễn sự tồn tại hoặc không tồn tại của các cạnh giữa các đỉnh.
* Xét G = (V, E) với V = {x1,…, xn}
* Biểu diễn G bằng ma trận A – (Aij), i, j = 1,…, n
* aij = 1, nếu tồn tại (xi, xj) thuộc E
* aij = 0, nếu không tồn tại (xi, xj) thuộc E

**Ví dụ**: biểu diễn ma trận kề không hướng

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A[i][j] | 0 | 1 | 2 | 3 |
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 2 | 1 | 1 | 0 | 1 |
| 3 | 0 | 1 | 1 | 0 |

2

3

1

0

**Ví dụ**: biểu diễn ma trận kề có hướng

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A[i][j] | 0 | 1 | 2 | 3 |
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 0 | 1 |
| 3 | 0 | 0 | 0 | 0 |

2

3

1

0

* 1. **Biểu diễn đồ thị bằng danh sách kề**
* Danh sách kề là phương pháp biểu diễn đồ thị bằng cách sử dụng một danh sách để lưu trữ các đỉnh kề với mỗi đỉnh. Mỗi đỉnh trong đồ thị có một danh sách các đỉnh kết nối trực tiếp với nó.
* Ở đây n hàng của ma trận kề thay thế bằng n danh sách liên kết động
* Mỗi đỉnh của G có một danh sách, mỗi nút trong danh sách thể hiện các đỉnh lân cận của nút này
* Cấu trúc mỗi nút: ID là tên đỉnh (chỉ số, danh hiệu), next là con trỏ đến nút kế tiếp

1. **Các phép duyệt đồ thị** 
   1. **Tìm kiếm theo chiều sâu**

Tìm kiếm theo chiều sâu đối với một đồ thị không định hướng được thực hiện như sau:

* Xuất phát từ đỉnh V (coi như được thăm). Xét một đỉnh w là lân cận của V nhưng chưa được thăm
* Nếu một đỉnh V đã được thăm mà mọi đỉnh lân cận của nó đều được thăm thì ta sẽ quay ngược lên đỉnh cuối cùng vừa được thăm mà nó có đỉnh w lân cận chưa được thăm và lặp lại cho tới khi không một nút nào được thăm
* Thuật toán của phép duyệt theo chiều sâu (DFS):

|  |
| --- |
| void DFS(int start) {  vector<bool> visited(V, false); // Khoi tao mang danh dau cac dinh chua duoc tham  cout << "DFS: ";  DFSUtil(start, visited);  cout << endl;  }  // Ham tien ich cho DFS (DFS su dung de quy)  void DFSUtil(int u, vector<bool>& visited) {  visited[u] = true; // Danh dau dinh u la da tham  cout << u << " "; // In dinh u  for (size\_t i = 0; i < adjList[u].size(); ++i) { // Su dung vong lap thong thuong  int v = adjList[u][i];  if (!visited[v]) { // Neu v chua duoc tham  DFSUtil(v, visited); // Goi de quy DFS tren dinh v  }  }  } |

* 1. **Tìm kiếm theo chiều rộng**
* Chọn một đỉnh xuất phát giả sử là V, ta lần lượt thăm tất cả các đỉnh lân cận đỉnh V. Sau đó lại lần lượt thăm tất cả các lân cận của các lân cận của V mà cũng chưa được thăm
* Thuật toán của phép tìm kiếm theo chiều rộng BFS:

|  |
| --- |
| void BFS(int start) {  vector<bool> visited(V, false); // Khoi tao mang danh dau cac dinh chua duoc tham  queue<int> q; // Khoi tao hang doi duyet BFS  visited[start] = true; // Ðanh dau dinh bat dau la da tham  q.push(start); // Dua dinh bat dau vao hang doi  cout << "BFS: ";  while (!q.empty()) {  int u = q.front();  q.pop();  cout << u << " "; // In dinh u  for (size\_t i = 0; i < adjList[u].size(); ++i) { // Su dung vong lap thong thuong  int v = adjList[u][i];  if (!visited[v]) { // Neu v chua duoc tham  visited[v] = true; // Ðanh dau v la da tham  q.push(v); // Ðua v vao hang doi  }  }  }  cout << endl;  } |

* Kết quả chạy thử:

A black screen with white text

Description automatically generated

* 1. **Cây khung (Spanning Tree) và cây khung với giá trị cực tiểu (minimum cost spanning tree)**
     1. **Cây khung (Spanning Tree)**
* T bao gồm các cung thuộc một phép duyệt từ một đỉnh đến các đỉnh còn lại trong V
* Giá của cây khung T bằng tổng trọng số của các cung thuộc E
* Cây khung là một đường đi đơn mà điểm đầu và cuối không trùng nhau
* Một đồ thị G có thể có nhiều cây khung
* Cây khung theo chiều rộng, cây khung theo chiều sâu
* Các cung trong cây khung không tạo nên chu trình . Giữa hai đỉnh trong một cây khung chỉ tồn tại duy nhất một đường đi từ đỉnh này đến đỉnh kia
* Nếu đồ thị có n đỉnh thì cây khung có n-1 cạnh
  + 1. **Cây khung cực tiểu**
* **Cây khung cực tiểu** (Minimum Spanning Tree - MST) là một cây con của một đồ thị vô hướng liên thông có trọng số, chứa tất cả các đỉnh của đồ thị, sao cho tổng trọng số của các cạnh trong cây là nhỏ nhất. Nói cách khác, đây là cây khung có tổng trọng số tối thiểu bao phủ tất cả các đỉnh của đồ thị.

**Các tính chất của cây khung cực tiểu:**

* Số đỉnh và số cạnh: Với đồ thị có VVV đỉnh, cây khung sẽ có V−1V - 1V−1 cạnh. Nếu có nhiều hơn V−1V - 1V−1 cạnh, đồ thị sẽ có chu trình, và nếu có ít hơn V−1V - 1V−1 cạnh, đồ thị sẽ không liên thông.
* Tính duy nhất: Cây khung cực tiểu không nhất thiết là duy nhất. Nếu đồ thị có các cạnh với trọng số bằng nhau, có thể tồn tại nhiều cây khung cực tiểu khác nhau với cùng tổng trọng số.
* Chu trình: Cây khung không bao giờ có chu trình, vì chu trình trong cây sẽ làm tăng số cạnh và phá vỡ tính chất cây
  + 1. **Thuật toán Kruskal**
* Sắp xếp các cung theo thứ tự không giảm đối với trọng số
* Bắt đầu từ T = rỗng
* Lặp lại cho đến khi không còn đỉnh nào (|E’| = n-1)
* Lấy ra cung w có trọng số nhỏ nhất
* Thêm cung w vào T với điều kiện không tạo chu trình trong T
* Để kiểm tra xem có tạo ra chu trình trong T hay không chúng ta xem hai đỉnh của cung được thêm có thuộc tập các đỉnh hiện có trong T không, nếu có nghĩa là sẽ tạo chu trình
* Thuật toán kruskal:

|  |
| --- |
| void kruskal(int n, vector<Edge>& edges) {  // Sap xep cac canh theo trong so tang dan  sort(edges.begin(), edges.end(), compare);  vector<int> parent(n + 1);  vector<int> rank(n + 1, 0);  // Khoi tao parent cua moi dinh la chinh no  for (int i = 0; i <= n; i++) {  parent[i] = i;  }  vector<Edge> mst;  int mstWeight = 0;  // Duyet cac canh theo trong so tang dan  for (size\_t i = 0; i < edges.size(); i++) {  int u = edges[i].u;  int v = edges[i].v;  int weight = edges[i].weight;  // Kiem tra xem co tao chu trinh hay khong  if (findParent(u, parent) != findParent(v, parent)) {  mst.push\_back(edges[i]);  mstWeight += weight;  unionSets(u, v, parent, rank);  }  }  // In ra ket qua  cout << "Canh trong cay khung toi thieu: \n";  for (size\_t i = 0; i < mst.size(); i++) {  cout << mst[i].u << " - " << mst[i].v << " : " << mst[i].weight << endl;  }  cout << "Tong trong so cay khung toi thieu: " << mstWeight << endl;  } |

* + 1. **Bài toán đường đi ngắn nhất dijkstra**

|  |
| --- |
| void dijkstra(const vector<vector<Edge> >& graph, int start) {  int n = graph.size();  vector<int> distances(n, INT\_MAX);  vector<bool> visited(n, false);  // Su dung priority\_queue de luu tru (khoang cach dinh)  priority\_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > > pq;  distances[start] = 0;  pq.push(make\_pair(0, start)); // Su dung make pair thay cho khoi tao danh sach  while (!pq.empty()) {  int node = pq.top().second;  int dist = pq.top().first;  pq.pop();  if (visited[node]) continue;  visited[node] = true;  for (size\_t i = 0; i < graph[node].size(); ++i) { // Su dung vong lap thong thuong  const Edge& neighbor = graph[node][i];  int newDist = dist + neighbor.weight;  if (newDist < distances[neighbor.node]) {  distances[neighbor.node] = newDist;  pq.push(make\_pair(newDist, neighbor.node)); // Su dung make\_pair  }  }  }  // Hien thi ket qua  cout << "Do dai duong di ngan nhat tu dinh " << start << ":\n";  for (int i = 0; i < n; i++) {  if (distances[i] == INT\_MAX) {  cout << "Dinh " << i << ": Khong the di toi\n";  } else {  cout << "Dinh " << i << ": " << distances[i] << endl;  }  }  } |

# **CHƯƠNG 8. SẮP XẾP**

1. **Khái niệm**

* Sắp xếp là quá trình tổ chức lại tập dữ liệu theo một trật tự tăng dần hay giảm dần
* Các phương pháp sắp xếp: có 5 phương pháp sắp xếp:
* Thuật toán Selection sort
* Thuật toán insertion sort
* Thuật toán buble sort
* Thuật toán heap sort
* Thuật toán quick sort
  1. **Thuật toán sắp xếp lựa chọn Selection sort**
* Giải thuật Selection sort sắp xếp một danh sách các giá trị bằng cách lặp lại việc đặt lại một giá trị cụ thể vào đúng vị trí thích hợp cho nó trong dãy sắp xếp
* Nói cách khác, với mỗi giá trị trong danh sách giải thuật đi tìm giá trị phù hợp cho vị trí đó
* Thuật toán:

|  |
| --- |
| void Selectionsorting(int a[], int n) {  int tmp;  for(int i = 0; i < n - 1; i++) {  int minIndex = i;  for(int j = i + 1; j < n; j++) {  if(a[minIndex] > a[j]) {  minIndex = j;  }  }  // Hoan doi gia tri i voi minIndex  tmp = a[minIndex];  a[minIndex] = a[i];  a[i] = tmp;  }  } |

* Độ phức tạp tính toán: ở bước i, có (n-i) lần so sánh, với i=1,…, n-1; (n-1)+(n-2)+…+1= n(n-1)/2 = O(n2).
  1. **Thuật toán sắp xếp chèn Inserttion sort**
* Ý tưởng: dựa theo ý tưởng của người chơi bài:
* Giả sử ở bước thứ i, các phần tử đã được sắp xếp theo thứ tự ki1, ki2,…kii-1
* Xét phần tử thứ i có khóa ki1, ta lần lượt so sánh với các phần tử đã được sắp xếp để tìm vị trí chèn thích hợp
* Thuật toán:

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

* Độ phức tạp tính toán:
* Ở bước thứ i, có tối đa i-1, tối thiểu 1 phép so sánh
* Thời gian thực hiện giải thuật T(n)~O(n2)
* Trường hợp xấu nhất có: 1+2+3+…+(n-1)= n(n-1)/2 = O(n2) phép so sánh và dịch chuyển
* Trường hợp tốt nhất (mảng đã có thứ tự tăng dần) O(n) phép so sánh và O phép dịch chuyển
  1. **Thuật toán sắp xếp nổi bọt Buble sort**
* Ý tưởng: ở bước i kể từ phần tử thứ i
* So sánh hai phần tử đứng cạnh nha, nếu khóa của phần tử trước lơn hơn khóa của phần tử sau thì đổi chỗ cho nhau
* Cuối cùng được phần tử có khóa lớn nhất đặt tại vị trí n-i-1
* Thuật toán:

|  |
| --- |
| void bublesort(int a[], int n){  int tmp;  for(int i = 0; i<n; i++){  for(int j = 0; j<n-i-1; j++){  if(a[j+1] < a[j]){  tmp = a[j+1];  a[j+1] = a[j];  a[j] = tmp;  }  }  }  } |

* Độ phức tạp tính toán:
* Ở bước thứ i, có n-i phép so sánh
* Thời gian thực hiện giải thuật T(n)~O(n2)
  1. **Thuật toán sắp xếp nhanh Quick sort**
* Ý tưởng: xét một dãy n phần tử a1, a2, an
* (1) chọn phần tử x = a[(n+1)div2] hàm khóa
* (2) đi từ hai đầu của dãy, nếu gặp một cặp a[i]>=x>=a[j] thì hoán vị hai phần tử này
* (3) tăng i=i+1, giảm j=j-1
* (4) lặp lại (2) cho đến khi i>j (kết quả thu được phân đoạn AxB)
* (5) lặp lại (1)-(4) với hai phân đoạn A và B
* Cuối cùng được phần tử có khóa lớn nhất đặt tại vị trí n-i+1
* Thuật toán:

|  |
| --- |
| void Quicksort(int a[], int l, int r) {  int i = l, j = r;  int x = a[(l + r) / 2]; // Chon phan tu o giua lam chot  do{  while(a[i] < x) i++; // Tim phan tu lon hon hoac bang chot tu trai sang phai  while(a[j] > x) j--; // Tim phan tu nho hon hoac bang chot tu phai sang trai  if(i <= j){  int tmp;  tmp = a[i];  a[i] = a[j];  a[j] = tmp;  i++;  j--;  }  }while(i <= j);  // De quy sap xep hai phan  if(l < j) Quicksort(a, l, j); // Sap xep phan ben trai  if(i < r) Quicksort(a, i, r); // Sap xep phan ben phai  } |

* Độ phức tạp tính toán:
* ở bước thứ i có n-i phép so sánh
* thời gian thực hiện giải thuật T(n)~O(n2)
  1. **Thuật toán sắp xếp Heap sort**
* Định nghĩa: dãy h1, h2,…, hn gọi tên là heap, nếu thỏa mãn
* hi>h2i+1
* hi>h2i+2
* thuật toán:

|  |
| --- |
| void setupHeap(int a[], int k, int n) {  int x = a[k];  while (k < n / 2) {  int j = 2 \* k + 1;  if (j + 1 < n && a[j] < a[j + 1]) j++;  if (x >= a[j]) break;  a[k] = a[j];  k = j;  }  a[k] = x;  }  // Ham de xay dung heap tu mot mang  void makeHeap(int a[], int n) {  for (int i = n / 2 - 1; i >= 0; i--) {  setupHeap(a, i, n);  }  }  // Thuat toan sap xep Heap Sort  void heapSort(int a[], int n) {  makeHeap(a, n);  for (int i = n - 1; i > 0; i--) {  int tmp = a[0];  a[0] = a[i];  a[i] = tmp;  setupHeap(a, 0, i);  }  *}* |

* độ phức tạp tính toán: thời gian thực hiện Heapsort là O(nlogn)

# **CHƯƠNG 9. TÌM KIẾM**

1. **Bài toán tìm kiếm**

* Tìm kiếm (Searching) là một đòi hỏi rất thường xuyên trong đời sống hàng ngày cũng như trong xử lý tin học.
* Tìm kiếm là quá trình xác định vị trí của một phần tử cụ thể trong một tập hợp dữ liệu. Tìm kiếm là một trong những thao tác cơ bản trong khoa học máy tính, dùng để truy xuất thông tin hoặc kiểm tra sự tồn tại của một giá trị trong cấu trúc dữ liệu như mảng, danh sách, liên kết, cây hoặc đồ thị.

1. **Các phép toán tìm kiếm** 
   1. **Phép tìm kiếm tuần tự (sequential search)**

* Tìm kiếm tuần tự là kỹ thuật tìm kiếm rất đơn giản và cổ điển. Nội dung có thể tóm tắt như sau:

“Bắt đầu từ bản ghi thứ nhất, lần lượt so sánh khóa tìm kiếm với khóa tương ứng của các bản ghi trong bảng, cho tới khi tìm được bản ghi mong muốn hoặc đã hết bảng mà chưa tìm thấy”.

* Ưu, nhược điểm của tìm kiếm tuần tự:
* Ưu điểm: dễ cài đặt, không cần mảng đã sắp xếp, hoạt động tốt với mảng nhỏ hoặc khi tìm kiếm chỉ diễn ra một lần
* Nhược điểm: không hiệu quả cho dữ liệu lớn do độ phức tạp thời gian cao
* Giải thuật:

|  |
| --- |
| int SEQUEN\_SEARCH(const vector<int>& k, int n, int X) {  int i = 0; // Buoc 1: Khoi dau  vector<int> arr = k; // Sao chep mang de khong anh huong den du lieu goc  arr.push\_back(X); // Them X vao vi tri cuoi mang de dung vong lap neu khong tim thay    while (arr[i] != X) { // Buoc 2: Tim khoa trong day  i++;  }  if (i == n) { // Buoc 3: Kiem tra ket qua  return 0; // Neu khong tim thay  } else {  return i + 1; // Tra ve vi tri tim thay (bat dau tu 1)  }  } |

* Kết quả sau khi chạy mô phỏng:

A black background with white text

Description automatically generated

* Độ phức tạp của thuật toán
* Trường hợp xấu nhất: O(n), khi phần tử cần tìm nằm ở cuối mảng hoặc không có trong mảng
* Trường hợp trung bình: O(n/2)
* Độ phức tạp không gian (Space Complexity): O(1) do chỉ cần một biến đếm và không sử dụng bộ nhớ phụ
  1. **Phép tìm kiếm nhị phân (search binary)**
* **Tìm kiếm nhị phân (Binary Search)** là một thuật toán tìm kiếm hiệu quả cho các dãy dữ liệu đã được sắp xếp. Thay vì kiểm tra từng phần tử như trong tìm kiếm tuần tự, thuật toán tìm kiếm nhị phân sẽ chia dãy dữ liệu thành hai nửa và so sánh giá trị cần tìm với phần tử ở giữa dãy. Từ đó, nó sẽ tiếp tục tìm kiếm trong nửa có thể chứa phần tử cần tìm, làm giảm đáng kể số lần so sánh
* Phương pháp tìm kiếm nhị phân: chọn khóa ở giữa dãy khóa đang xét để thực hiện so sánh với khóa tìm kiếm.
* Nếu "Khoá"< khoá ở giữa dãy tìm kiếm, sẽ tiếp tục thực hiện với "khoá ở giữa"-1
* Nếu "Khoá"> khoá ở giữa dãy tìm kiếm, sẽ tiếp tục thực hiện với "khoá ở giữa"+1, ..đến hết dãy.

Quá trình tìm kiếm được tiếp tục đến khi tìm thấy khóa mong muốn hoặc dãy khóa xét đó trở nên rỗng.

* Giải thuật tìm kiếm nhị phân:

|  |
| --- |
| int binarySearch(vector<int>& k, int n, int X) {  int l = 0; // Chi so bat dau (l) cua mang  int r = n - 1; // Chi so cuoi (r) cua mang  int m;  while (l <= r) {  m = (l + r) / 2; // Tinh chi so giua m  if (X < k[m]) {  r = m - 1; // Neu X nho hon phan tu giua, tim trong nua trai  } else if (X > k[m]) {  l = m + 1; // Neu X lon hon phan tu giua, tim trong nua phai  } else {  return m; // Neu tim thay X, tra ve chi so cua no  }  }  return -1; // Neu khong tim thay X trong mang  } |

* Kết quả sau khi chạy mô phỏng:

A black background with white text

Description automatically generated

* Độ phức tạp của thuật toán:
* Độ phức tạp thời gian: O(log n), là mỗi lần tìm kiếm sẽ giảm một nửa dãy dữ liệu. Giúp tiết kiệm thời gian đáng kể khi tìm kiếm trong mảng lớn so với tìm kiếm tuần tự
* Điều kiện là mảng phải được sắp xếp trước (tăng hoặc giảm dần), nếu mảng chưa được xắp xếp trước thì thuật toán không thể áp dụng được.
  1. **Cây nhị phân tìm kiếm (binary search tree)**
* Định nghĩa: cây nhị phân tìm kiếm ứng với n khóa k1, k2,…,kn là cây: mỗi nút của nó đều được định danh bởi một khóa nào đó trong các khóa đã cho, mọi nút trên cây có tính chất:
* Mọi khóa thuộc cây con trái nút đó đều nhỏ hơn khóa ứng với nút đó
* Mọi khóa thuộc cây con phải nút đó đều lớn hơn khóa ứng với nút đó
* Thuật toán:

|  |
| --- |
| Node\* search(Node\* root, int X) {  // Neu cay rong hoac tim thay khoa X  if (root == NULL || root->data == X)  return root;  // Neu X nho hon goc, tim kiem trong cay con trai  if (X < root->data)  return search(root->left, X);  // Neu X lon hon goc, tim kiem trong cay con phai  return search(root->right, X);  } |

* Kết quả sau khi chạy mô phỏng:

A black background with white text

Description automatically generated

# **KẾT LUẬN**

Bài tập lớn môn Cấu trúc dữ liệu và Giải thuật không chỉ giúp sinh viên củng cố kiến thức về các cấu trúc dữ liệu và thuật toán, mà còn tạo cơ hội để ứng dụng những lý thuyết đã học vào giải quyết các bài toán thực tiễn. Qua việc thiết kế, triển khai và tối ưu hóa các thuật toán, sinh viên đã học được cách phân tích các vấn đề một cách hệ thống và đưa ra các giải pháp hiệu quả, phù hợp với từng tình huống cụ thể.

Quá trình thực hiện bài tập giúp sinh viên nhận thức được vai trò quan trọng của việc lựa chọn cấu trúc dữ liệu và thuật toán tối ưu trong việc phát triển phần mềm. Những kỹ năng này sẽ đóng vai trò then chốt trong việc giải quyết các bài toán phức tạp trong ngành công nghiệp phần mềm, nơi yêu cầu hiệu suất và khả năng mở rộng cao.

Cuối cùng, bài tập không chỉ giúp sinh viên phát triển kỹ năng lập trình, mà còn rèn luyện khả năng tư duy phân tích, làm việc độc lập và khả năng hợp tác trong môi trường nhóm. Đây là những yếu tố thiết yếu để thành công trong nghề nghiệp lập trình viên trong tương lai.

**TÀI LIỆU THAM KHẢO**

1. *Tài liệu môn học “Cấu trúc dữ liệu và giải thuật” do thầy Đỗ Xuân Lôi biên soạn – Nhà xuất bản Đại học quốc gia Hà Nội*
2. *Các giải thuật phổ biến trong lập trình được trích dẫn từ các nguồn mở trực tuyến*