**TRƯỜNG ĐẠI HỌC KỸ THUẬT CÔNG NGHIỆP**

**KHOA ĐIỆN TỬ**

Bộ môn: Công nghệ thông tin

**BÀI TẬP LỚN**

MÔN HỌC

**CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

Sinh viên: …NGUYỄN LAM SƠN………………………………………

MSV: ….k225480106076………………………………………………

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: LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG**

BỘ MÔN : TIN HỌC CÔNG NGHIỆP

*Sinh viên: Nguyễn Lam Sơn*

*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: 18/11/2024*

**NHẬN XÉT CỦA GIÁO VIÊN HƯỚNG DẪN**

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

GIÁO VIÊN HƯỚNG DẪN

*(Ký ghi rõ họ tên)*

MỤC LỤC

[**CHƯƠNG I+II. GIẢI THUẬT** 5](#_Toc181885256)

[**1.1 Giải thuật và cấu trúc dữ liệu** 5](#_Toc181885257)

[**CHƯƠNG 3. ĐỆ QUY VÀ GIẢI THUẬT ĐỆ QUY** 6](#_Toc181885258)

[**3.1. Khái niệm của đệ quy** 6](#_Toc181885259)

[**3.3. Giải thuật và kết quả chạy** 7](#_Toc181885260)

[**CHƯƠNG 4. MẢNG VÀ DANH SÁCH** 8](#_Toc181885261)

[**4.1. Mảng và danh sách** 8](#_Toc181885262)

[**4.1.1. Khái niệm mảng** 8](#_Toc181885263)

[**4.1.2. Danh sách (List)** 8](#_Toc181885264)

[**4.2. Thuật toán và kết quả chạy** 10](#_Toc181885265)

[**4.2.3. Thuật toán danh sách.** 10](#_Toc181885266)

[**CHƯƠNG 5: DANH SÁCH MÓC NỐI ĐƠN VÀ DANH SÁCH MÓC NỐI KÉP** 14](#_Toc181885267)

[**5.1. Danh sách móc nối đơn và danh sách móc nối kép** 14](#_Toc181885268)

[**5.1.1. Danh sách Móc Nối Đơn (Singly Linked List)** 14](#_Toc181885269)

[**5.1.2. Danh sách Móc Nối Kép (Doubly Linked List)** 15](#_Toc181885270)

[**5.2. Thuật toán và kết quả chạy** 16](#_Toc181885271)

[**CHUONG 6: CÂY NHỊ PHÂN** 27](#_Toc181885272)

[**6.1. Khái niệm cây nhị phân** 27](#_Toc181885273)

[**6.2. Ưu điểm và nhược điểm của cây nhị phân** 28](#_Toc181885274)

[**6.3. Thuật toán và kết quả chạy** 29](#_Toc181885275)

[**CHUONG 7: ĐỒ THỊ** 34](#_Toc181885276)

[**7.1. khái niệm đồ thị** 34](#_Toc181885277)

[**7.2. Phép Duyệt Đồ Thị** 34](#_Toc181885278)

[**7.3. thuật toán và kết quả chạy** 36](#_Toc181885279)

[**CHƯƠNG 8: XẮP SẾP** 39](#_Toc181885280)

[**8.1.khái niệm xắp sếp** 39](#_Toc181885281)

[**8.2. thuật toán và kế quả chạy** 41](#_Toc181885282)

[**CHƯƠNG 9: TÌM KIẾM** 48](#_Toc181885283)

[**9.1. Khái niệm** 48](#_Toc181885284)

[**9.2. thuật toán và kết quả chạy** 51](#_Toc181885285)

# **CHƯƠNG I+II. GIẢI THUẬT**

## **1.1 Giải thuật và cấu trúc dữ liệu**

Giải thuật là tập hợp các bước hướng dẫn rõ ràng và chính xác, nhằm đạt được kết quả mong muốn sau một số hữu hạn bước thực hiện trên dữ liệu. Các giải thuật không chỉ là các phép tính toán hay xử lý trên dữ liệu mà còn phản ánh cách biểu diễn và tổ chức dữ liệu để tối ưu hóa hiệu suất của máy tính.

Cấu trúc dữ liệu (CTDL) là cách tổ chức, lưu trữ và biểu diễn các thông tin trong bài toán, giúp cho việc thực hiện các thao tác xử lý trên dữ liệu trở nên hiệu quả hơn. Các phần tử của dữ liệu thường có mối quan hệ logic với nhau, vì vậy việc chọn cấu trúc phù hợp sẽ ảnh hưởng trực tiếp đến hiệu quả của các giải thuật.

**Mối quan hệ giữa Giải thuật và Cấu trúc dữ liệu:**

Giải thuật và CTDL có mối quan hệ mật thiết. Một giải thuật sẽ hoạt động trên cấu trúc dữ liệu để biến đổi hoặc trích xuất thông tin nhằm đạt được kết quả mong muốn. Khi cấu trúc dữ liệu thay đổi, giải thuật cũng cần được điều chỉnh để phù hợp với cấu trúc mới, nhằm đạt hiệu suất tốt nhất. Điều này có nghĩa là việc chọn đúng CTDL sẽ giúp giải thuật hoạt động hiệu quả và tối ưu hơn.

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

## **3.1. Khái niệm của đệ quy**

**Đệ quy (Recursion algorithm)** : Đệ quy là một kỹ thuật trong lập trình, trong đó một hàm tự gọi lại chính nó để giải quyết một vấn đề. Quá trình này thường được sử dụng để giải quyết các bài toán có cấu trúc lặp lại hoặc phân chia và chinh phục.

Đặc điểm của Đệ Quy

Điểm Dừng (Base Case): Điều kiện để dừng việc gọi đệ quy. Nếu không có điểm dừng, hàm sẽ gọi vô hạn và gây ra lỗi tràn ngăn xếp.

Lời Gọi Đệ Quy (Recursive Call): Phần của hàm mà nó tự gọi lại chính nó với một tập dữ liệu nhỏ hơn hoặc đơn giản hơn.

**3.2. Ưu điểm và hạn chế của giải thuật đệ quy**

Ưu Điểm của Đệ Quy:

Ngắn Gọn và Dễ Hiểu: Mã nguồn thường ngắn hơn và dễ đọc hơn, đặc biệt với các bài toán có cấu trúc lặp lại như cây, đồ thị.

Giải Quyết Vấn Đề Phức Tạp: Thích hợp cho các bài toán phân chia và chinh phục, như tính giai thừa, giải bài toán tháp Hà Nội, duyệt cây.

Nhược Điểm của Đệ Quy:

Hiệu Suất Bộ Nhớ: Có thể tiêu tốn nhiều bộ nhớ do mỗi lời gọi hàm được lưu trữ trên ngăn xếp.

Dễ Gây Lỗi Tràn Ngăn Xếp: Nếu không có điều kiện dừng rõ ràng, hàm có thể gọi vô hạn, gây lỗi tràn ngăn xếp.

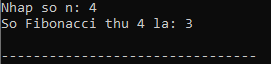
Thường Chậm Hơn: So với cách tiếp cận lặp, đệ quy có thể chậm hơn do chi phí của các lời gọi hàm.

## **3.3. Giải thuật và kết quả chạy**

**3.3.1. Giải thuật:** tính giai thừa

|  |
| --- |
| int fibonacci(int n) {  if (n <= 1)  return n;  return fibonacci(n - 1) + fibonacci(n - 2);  } |

**3.3.2. Kết quả sau khi chạy chương trình:**



# **CHƯƠNG 4. MẢNG VÀ DANH SÁCH**

## **4.1. Mảng và danh sách**

### **4.1.1. Khái niệm mảng**

**Mảng** là một cấu trúc dữ liệu bao gồm các phần tử có cùng kiểu dữ liệu và được lưu trữ liên tiếp trong bộ nhớ. Mỗi phần tử trong mảng được truy cập bằng chỉ số (index) của nó.

**Đặc điểm**:

* Kích thước mảng được cố định tại thời điểm khai báo và không thể thay đổi.
* Truy cập các phần tử trong mảng rất nhanh vì các phần tử có vị trí liền kề trong bộ nhớ.
* Chỉ có thể chứa các phần tử cùng kiểu dữ liệu.

**Ưu điểm**:

* Truy cập phần tử nhanh, do các phần tử nằm liên tiếp trong bộ nhớ.
* Dễ sử dụng và phù hợp cho các bài toán cần cố định số lượng phần tử.

**Nhược điểm**:

* Kích thước cố định, không thể thay đổi sau khi khai báo.
* Nếu cần thêm hoặc xóa phần tử, phải tạo mảng mới hoặc thực hiện thủ công, gây tốn bộ nhớ và công sức.

### **4.1.2. Danh sách (List)**

* **Danh sách** là một cấu trúc dữ liệu động, cho phép lưu trữ các phần tử có cùng kiểu dữ liệu và có thể thay đổi kích thước linh hoạt. Danh sách thường có hai loại phổ biến: danh sách liên kết và danh sách mảng động.

**4.1.2.1. Danh sách liên kết (Linked List)**

**Danh sách liên kết** là một cấu trúc dữ liệu mà các phần tử được lưu trữ ở các vị trí khác nhau trong bộ nhớ, mỗi phần tử (gọi là một "nút") chứa một con trỏ để liên kết tới nút kế tiếp.

**Các loại danh sách liên kết**:

* Danh sách liên kết đơn (Single Linked List): Mỗi nút chỉ chứa một con trỏ tới nút tiếp theo.
* Danh sách liên kết đôi (Doubly Linked List): Mỗi nút chứa hai con trỏ, một trỏ tới nút trước và một trỏ tới nút sau.
* Danh sách liên kết vòng (Circular Linked List): Nút cuối cùng trỏ lại nút đầu tiên, tạo thành vòng khép kín.

**Ưu điểm**:

* Kích thước linh hoạt, dễ thêm hoặc xóa phần tử ở bất kỳ vị trí nào.
* Không cần biết trước số lượng phần tử.

**Nhược điểm**:

* Truy cập phần tử chậm hơn so với mảng vì cần phải duyệt từng phần tử.
* Sử dụng thêm bộ nhớ cho các con trỏ.

**4.1.2.2.Danh sách mảng động (Dynamic Array)**

**Danh sách mảng động** (như std::vector trong C++ hoặc ArrayList trong Java) là một loại mảng có thể tự động thay đổi kích thước khi thêm hoặc xóa phần tử.

**Ưu điểm**:

* Tính năng linh hoạt của mảng động, dễ thêm hoặc xóa phần tử.
* Tốc độ truy cập phần tử nhanh do phần tử nằm liền kề trong bộ nhớ.

**Nhược điểm**:

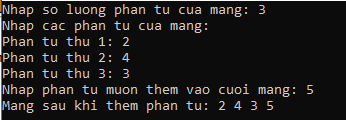
* Khi cần thay đổi kích thước lớn, đôi khi sẽ gây tốn tài nguyên do cần phải cấp phát lại bộ nhớ.

### **4.2. Thuật toán và kết quả chạy**

**4.2.1. Thuật toán mảng**

|  |
| --- |
| int newElement;  cout << "Nhap phan tu muon them vao cuoi mang: ";  cin >> newElement;  mang[n] = newElement;  n++;  cout << "Mang sau khi them phan tu: ";  for (int i = 0; i < n; i++) {  cout << mang[i] << " "; |

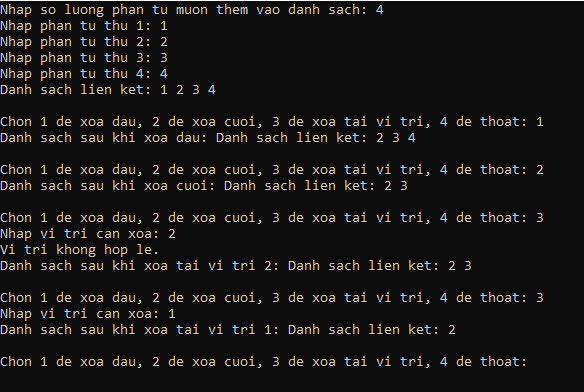
**4.2.2. Kết quả chạy mảng**



## **4.2.3. Thuật toán danh sách.**

|  |
| --- |
| // Hàm thêm phần tử vào cuối danh sách liên kết  void themVaoCuoi(Node\*& head, int newElement) {  Node\* newNode = new Node();  newNode->data = newElement;  newNode->next = nullptr;  if (head == nullptr) {  head = newNode;  } else {  Node\* temp = head;  while (temp->next != nullptr) {  temp = temp->next;  }  temp->next = newNode;  }  }  // Hàm xóa phần tử đầu tiên  void xoaDau(Node\*& head) {  if (head == nullptr) {  cout << "Danh sach rong, khong the xoa." << endl;  return;  }  Node\* temp = head;  head = head->next;  delete temp; // Giải phóng bộ nhớ của phần tử bị xóa  }  // Hàm xóa phần tử cuối cùng  void xoaCuoi(Node\*& head) {  if (head == nullptr) {  cout << "Danh sach rong, khong the xoa." << endl;  return;  }  if (head->next == nullptr) {  delete head;  head = nullptr;  return;  }  Node\* temp = head;  while (temp->next != nullptr && temp->next->next != nullptr) {  temp = temp->next;  }  delete temp->next;  temp->next = nullptr;  }  // Hàm xóa phần tử tại một vị trí  void xoaTaiViTri(Node\*& head, int viTri) {  if (head == nullptr) {  cout << "Danh sach rong, khong the xoa." << endl;  return;  }  if (viTri == 0) {  xoaDau(head);  return;  }  Node\* temp = head;  for (int i = 0; i < viTri - 1; i++) {  if (temp == nullptr) {  cout << "Vi tri khong hop le." << endl;  return;  }  temp = temp->next;  }  if (temp == nullptr || temp->next == nullptr) {  cout << "Vi tri khong hop le." << endl;  return;  }  Node\* nodeToDelete = temp->next;  temp->next = temp->next->next;  delete nodeToDelete;  } |

**2.3.2. Kết quả chạy danh sách**



# **CHƯƠNG 5: DANH SÁCH MÓC NỐI ĐƠN VÀ DANH SÁCH MÓC NỐI KÉP**

## **5.1. Danh sách móc nối đơn và danh sách móc nối kép**

### **5.1.1. Danh sách Móc Nối Đơn (Singly Linked List)**

**Khái niệm:**

Danh sách móc nối đơn là một cấu trúc dữ liệu mà mỗi phần tử (node) trong danh sách chỉ chứa dữ liệu và một con trỏ (next) trỏ đến phần tử tiếp theo trong danh sách.

Mỗi phần tử trong danh sách liên kết này có thể chỉ trỏ đến một phần tử duy nhất phía sau nó.

**Cấu trúc của Node:**

* Mỗi phần tử (node) trong danh sách móc nối đơn có hai phần:
* Dữ liệu (data): Lưu trữ giá trị của phần tử.
* Con trỏ (next): Chỉ trỏ đến phần tử tiếp theo trong danh sách. Nếu phần tử là cuối danh sách, next sẽ trỏ đến nullptr (điều này báo hiệu kết thúc danh sách).

**Đặc điểm:**

* Một chiều: Chỉ có thể di chuyển từ phần tử đầu tiên đến phần tử cuối cùng của danh sách, không thể quay lại.
* Dễ dàng thay đổi kích thước: Có thể thêm hoặc xóa phần tử ở đầu, giữa, hoặc cuối mà không cần phải di chuyển các phần tử khác trong danh sách.
* Dễ cài đặt: Cấu trúc danh sách móc nối đơn đơn giản hơn so với danh sách móc nối kép.

**Ứng dụng:**

* Sử dụng trong các tình huống cần cấu trúc dữ liệu có thể thay đổi linh hoạt.
* Làm cơ sở cho các cấu trúc dữ liệu phức tạp hơn như danh sách đơn, cây nhị phân.

**Các thao tác cơ bản:**

* Thêm phần tử vào đầu danh sách: Tạo một node mới và gán con trỏ next của nó trỏ đến node hiện tại, sau đó cập nhật head trỏ đến node mới.
* Thêm phần tử vào cuối danh sách: Duyệt danh sách đến phần tử cuối cùng và gán con trỏ next của nó trỏ đến node mới.
* Xóa phần tử đầu tiên: Cập nhật con trỏ head để trỏ đến phần tử tiếp theo trong danh sách.
* Duyệt danh sách: Di chuyển từ đầu đến cuối danh sách, in ra giá trị của từng node.

**Ví dụ:**

Giả sử bạn có một danh sách chứa các số nguyên: 10 → 20 → 30 → 40. Mỗi node sẽ chứa một giá trị và một con trỏ trỏ đến node tiếp theo.

### **5.1.2. Danh sách Móc Nối Kép (Doubly Linked List)**

**Khái niệm:**

Danh sách móc nối kép là một cấu trúc dữ liệu mà mỗi phần tử (node) không chỉ chứa con trỏ next trỏ đến phần tử tiếp theo mà còn có thêm một con trỏ prev trỏ đến phần tử trước đó.

Điều này cho phép duyệt danh sách theo cả hai chiều (từ đầu đến cuối và từ cuối lên đầu).

**Cấu trúc của Node:**

* Mỗi phần tử (node) trong danh sách móc nối kép có ba phần:
* Dữ liệu (data): Lưu trữ giá trị của phần tử.
* Con trỏ next: Chỉ trỏ đến phần tử tiếp theo trong danh sách.
* Con trỏ prev: Chỉ trỏ đến phần tử trước đó trong danh sách.

**Đặc điểm:**

* Hai chiều: Bạn có thể di chuyển từ đầu danh sách đến cuối danh sách và ngược lại, nhờ vào con trỏ prev.
* Tốn thêm bộ nhớ: Mỗi node cần phải lưu hai con trỏ (next và prev), do đó chiếm nhiều bộ nhớ hơn so với danh sách móc nối đơn.
* Dễ dàng thao tác: Việc xóa hoặc thêm phần tử ở đầu và cuối danh sách trở nên dễ dàng hơn vì bạn không cần phải duyệt qua tất cả các phần tử.

**Ứng dụng:**

* Duyệt danh sách hai chiều: Các ứng dụng yêu cầu việc di chuyển qua lại giữa các phần tử, như trong các phần mềm quản lý dữ liệu, ứng dụng tìm kiếm, vv.
* Cấu trúc dữ liệu hai chiều: Dùng trong các vấn đề cần truy cập qua lại như các thao tác trên cây, đồ thị.

**Các thao tác cơ bản:**

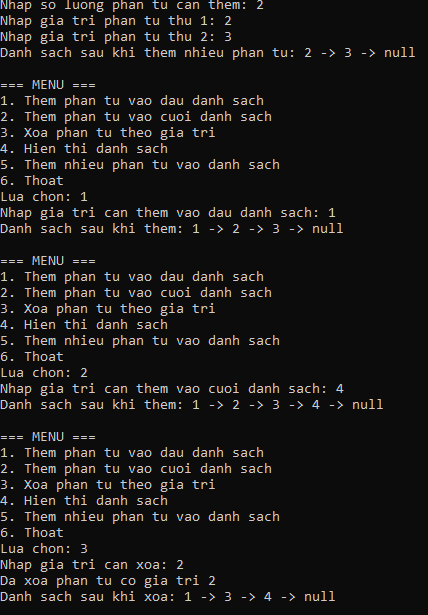
* Thêm phần tử vào đầu danh sách: Cập nhật con trỏ prev của phần tử đầu tiên để trỏ về node mới, và sau đó cập nhật head trỏ đến node mới.
* Thêm phần tử vào cuối danh sách: Duyệt qua danh sách cho đến khi gặp phần tử cuối cùng, cập nhật con trỏ next của phần tử cuối trỏ đến node mới và cập nhật con trỏ prev của node mới.
* Xóa phần tử đầu tiên: Cập nhật con trỏ head để trỏ đến phần tử tiếp theo và cập nhật con trỏ prev của phần tử đầu tiên nếu cần.
* Xóa phần tử cuối cùng: Duyệt danh sách đến phần tử cuối cùng, cập nhật con trỏ next của phần tử trước đó thành nullptr và xóa phần tử cuối cùng.

## **5.2. Thuật toán và kết quả chạy**

**5.2.1 thuật toán danh sách móc nối dơn**

|  |
| --- |
| // Hàm tạo một Node mới  Node\* createNode(int value) {  Node\* newNode = new Node();  newNode->data = value;  newNode->next = nullptr;  return newNode;  }  // Hàm thêm Node vào đầu danh sách  void insertAtHead(Node\*& head, int value) {  Node\* newNode = createNode(value);  newNode->next = head;  head = newNode;  }  // Hàm thêm Node vào cuối danh sách  void insertAtTail(Node\*& head, int value) {  Node\* newNode = createNode(value);  if (head == nullptr) { // Nếu danh sách rỗng  head = newNode;  return;  }  Node\* temp = head;  while (temp->next != nullptr) { // Duyệt đến node cuối  temp = temp->next;  }  temp->next = newNode; // Thêm node mới vào cuối  }  // Hàm xóa Node có giá trị cụ thể  void deleteNodeByValue(Node\*& head, int value) {  if (head == nullptr) {  cout << "Danh sach rong." << endl;  return;  }  // Xóa phần tử đầu nếu nó có giá trị cần xóa  if (head->data == value) {  Node\* temp = head;  head = head->next;  delete temp;  cout << "Da xoa phan tu co gia tri " << value << endl;  return;  }  // Tìm và xóa phần tử có giá trị cần xóa  Node\* temp = head;  while (temp->next != nullptr && temp->next->data != value) {  temp = temp->next;  }  if (temp->next == nullptr) {  cout << "Khong tim thay gia tri " << value << " trong danh sach." << endl;  return;  }  Node\* nodeToDelete = temp->next;  temp->next = temp->next->next;  delete nodeToDelete;  cout << "Da xoa phan tu co gia tri " << value << endl;  }  // Hàm hiển thị các phần tử trong danh sách  void displayList(Node\* head) {  if (head == nullptr) {  cout << "Danh sach rong." << endl;  return;  }  Node\* temp = head;  while (temp != nullptr) {  cout << temp->data << " -> ";  temp = temp->next;  }  cout << "null" << endl;  }  // Hàm nhập nhiều phần tử từ bàn phím và thêm vào danh sách  void inputMultipleElements(Node\*& head) {  int n, value;  cout << "Nhap so luong phan tu can them: ";  cin >> n;  for (int i = 0; i < n; i++) {  cout << "Nhap gia tri phan tu thu " << i + 1 << ": ";  cin >> value;  insertAtTail(head, value);  }  } |

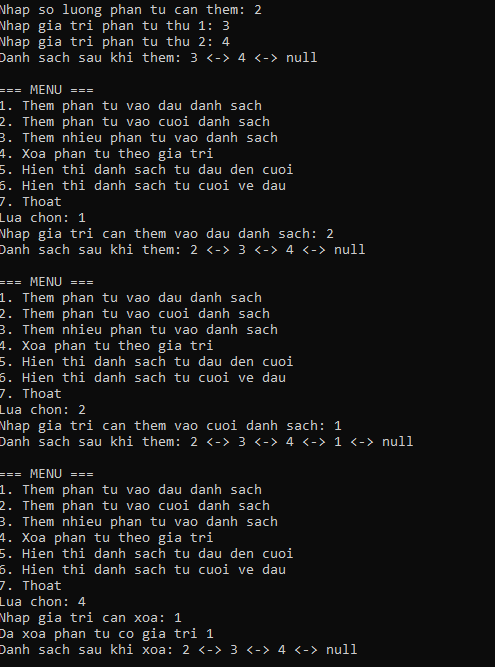
**5.2.2. Kết quả chạy danh sách móc nối đơn**



**5.2.3. thuật toán danh sách móc nối kép**

|  |
| --- |
| // Định nghĩa cấu trúc Node cho danh sách móc nối kép  struct Node {  int data; // Dữ liệu  Node\* prev; // Con trỏ đến node trước  Node\* next; // Con trỏ đến node sau  };  // Hàm tạo một Node mới  Node\* createNode(int value) {  Node\* newNode = new Node();  newNode->data = value;  newNode->prev = nullptr;  newNode->next = nullptr;  return newNode;  }  // Hàm thêm Node vào đầu danh sách  void insertAtHead(Node\*& head, int value) {  Node\* newNode = createNode(value);  newNode->next = head;    if (head != nullptr) {  head->prev = newNode; // Cập nhật con trỏ prev của node đầu  }    head = newNode; // Cập nhật head  }  // Hàm thêm Node vào cuối danh sách  void insertAtTail(Node\*& head, int value) {  Node\* newNode = createNode(value);  if (head == nullptr) {  newNode->prev = nullptr; // Nếu danh sách rỗng  head = newNode; // Đặt head trỏ đến node mới  return;  }  Node\* temp = head;  while (temp->next != nullptr) {  temp = temp->next; // Duyệt đến node cuối  }  temp->next = newNode; // Cập nhật con trỏ next của node cuối  newNode->prev = temp; // Cập nhật con trỏ prev của node mới  }  // Hàm xóa Node có giá trị cụ thể  void deleteNodeByValue(Node\*& head, int value) {  if (head == nullptr) {  cout << "Danh sach rong." << endl;  return;  }  Node\* temp = head;    // Tìm node cần xóa  while (temp != nullptr && temp->data != value) {  temp = temp->next;  }  // Nếu không tìm thấy node cần xóa  if (temp == nullptr) {  cout << "Khong tim thay gia tri " << value << " trong danh sach." << endl;  return;  }  // Nếu node cần xóa là node đầu  if (temp == head) {  head = head->next; // Cập nhật head  if (head != nullptr) {  head->prev = nullptr; // Cập nhật con trỏ prev của node mới đầu  }  } else {  temp->prev->next = temp->next; // Cập nhật con trỏ next của node trước  }  if (temp->next != nullptr) {  temp->next->prev = temp->prev; // Cập nhật con trỏ prev của node sau  }  delete temp; // Giải phóng bộ nhớ  cout << "Da xoa phan tu co gia tri " << value << endl;  }  // Hàm hiển thị các phần tử trong danh sách từ đầu đến cuối  void displayForward(Node\* head) {  if (head == nullptr) {  cout << "Danh sach rong." << endl;  return;  }    Node\* temp = head;  while (temp != nullptr) {  cout << temp->data << " <-> ";  temp = temp->next;  }  cout << "null" << endl;  }  // Hàm hiển thị các phần tử trong danh sách từ cuối đến đầu  void displayBackward(Node\* head) {  if (head == nullptr) return;  Node\* temp = head;  while (temp->next != nullptr) {  temp = temp->next; // Duyệt đến node cuối  }  while (temp != nullptr) {  cout << temp->data << " <-> ";  temp = temp->prev; // Quay lại node trước  }  cout << "null" << endl;  }  // Hàm nhập các phần tử từ bàn phím và thêm vào danh sách  void inputMultipleElements(Node\*& head) {  int n, value;  cout << "Nhap so luong phan tu can them: ";  cin >> n;  for (int i = 0; i < n; i++) {  cout << "Nhap gia tri phan tu thu " << i + 1 << ": ";  cin >> value;  insertAtTail(head, value);  }  cout << "Danh sach sau khi them: ";  displayForward(head); // Hiển thị danh sách sau khi thêm  } |

**5.2.4. kết quả chạy sách móc nối kép**



# **CHUONG 6: CÂY NHỊ PHÂN**

## **6.1. Khái niệm cây nhị phân**

Cây nhị phân (Binary Tree) là một cấu trúc dữ liệu mà mỗi nút (node) trong cây có tối đa hai con (children), thường được gọi là con trái (left child) và con phải (right child). Cây nhị phân thường được sử dụng để tổ chức và lưu trữ dữ liệu theo cách mà cho phép thực hiện các phép toán tìm kiếm, chèn và xóa một cách hiệu quả.

Phép duyệt cây là một kỹ thuật thiết yếu trong lập trình để thăm và xử lý tất cả các nút trong một cây, đặc biệt là cây nhị phân. Kỹ thuật này cho phép người lập trình thực hiện các thao tác như tìm kiếm, chèn, và xóa các nút trong cây. Duyệt cây giúp người dùng truy cập thông tin một cách có hệ thống, đảm bảo rằng không bỏ sót bất kỳ nút nào. Tùy thuộc vào mục đích và cách thức, có nhiều phương pháp duyệt cây khác nhau, mỗi phương pháp đều có những ứng dụng riêng.

**Duyệt theo thứ tự trước (Pre-order Traversal)**

Duyệt theo thứ tự trước là một trong những phương pháp phổ biến nhất để truy cập các nút trong cây nhị phân. Trong phép duyệt này, nút gốc (root) được thăm trước tiên, tiếp theo là các nút con bên trái và sau cùng là các nút con bên phải. Phép duyệt này thường được sử dụng trong các tình huống cần sao chép hoặc biểu diễn lại cấu trúc cây. Ví dụ, trong một cây nhị phân có cấu trúc đơn giản, nếu chúng ta thực hiện duyệt theo thứ tự trước, chúng ta sẽ nhận được kết quả lần lượt từ nút gốc cho đến các nút con, điều này giúp cho việc lập trình và xử lý dữ liệu trở nên dễ dàng hơn.

**Duyệt theo thứ tự giữa (In-order Traversal)**

Duyệt theo thứ tự giữa là một phương pháp hữu ích khác trong việc truy cập các nút của cây nhị phân. Với phép duyệt này, các nút con bên trái được truy cập trước, sau đó là nút gốc, và cuối cùng là các nút con bên phải. Phép duyệt này đặc biệt quan trọng khi làm việc với cây nhị phân tìm kiếm, vì nó cho phép chúng ta thu thập các giá trị theo thứ tự tăng dần. Khi áp dụng duyệt theo thứ tự giữa, các giá trị sẽ được thu thập một cách có trật tự, đảm bảo tính hợp lệ của dữ liệu khi cần xử lý hoặc hiển thị.

**Duyệt theo thứ tự sau (Post-order Traversal)**

Duyệt theo thứ tự sau là một phương pháp khác để thăm các nút trong cây nhị phân, trong đó các nút con bên trái và bên phải được truy cập trước, và cuối cùng là nút gốc. Phép duyệt này rất hữu ích trong các tình huống mà người lập trình cần thực hiện các thao tác giải phóng tài nguyên hoặc xóa các nút trong cây. Bằng cách truy cập các nút con trước, chúng ta có thể đảm bảo rằng tất cả các tài nguyên của nút gốc được giải phóng trước khi nút gốc bị xóa. Điều này giúp quản lý bộ nhớ hiệu quả và duy trì tính toàn vẹn của cây trong quá trình xử lý.

lệch chiều cao giữa các cây con trái và phải của bất kỳ nút nào không vượt quá 1.

## **6.2. Ưu điểm và nhược điểm của cây nhị phân**

**Ưu điểm:**

**Tìm kiếm nhanh**: Cây nhị phân tìm kiếm cho phép tìm kiếm, thêm và xóa các phần tử với độ phức tạp trung bình là O(log n).

**Tổ chức dữ liệu linh hoạt**: Dễ dàng thực hiện các phép toán như duyệt cây (traversing) theo thứ tự khác nhau (tiền thứ tự, trung thứ tự, hậu thứ tự).

**Cấu trúc đệ quy**: Có thể dễ dàng áp dụng các thuật toán đệ quy trên cây.

**Nhược điểm:**

**Cây không cân bằng**: Nếu các phần tử được chèn vào theo thứ tự đã sắp xếp, cây có thể trở thành một danh sách liên kết (linked list), dẫn đến độ phức tạp tìm kiếm O(n).

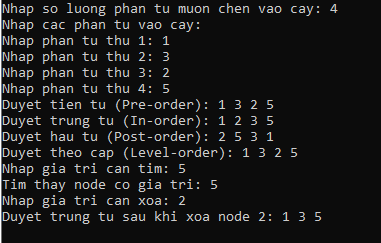
**Chiếm bộ nhớ**: Cây nhị phân yêu cầu bộ nhớ cho cả con trái và con phải của mỗi nút, điều này có thể tốn kém nếu cây có nhiều nút.

## **6.3. Thuật toán và kết quả chạy**

**6.3.1 Thuật toán**

|  |
| --- |
| // Hàm duyệt cây theo tiền tự (Pre-order)  void preOrder(Node\* root) {  if (root == nullptr) return;  cout << root->data << " "; // Xử lý node hiện tại  preOrder(root->left); // Duyệt cây con trái  preOrder(root->right); // Duyệt cây con phải  }  // Hàm duyệt cây theo trung tự (In-order)  void inOrder(Node\* root) {  if (root == nullptr) return;  inOrder(root->left); // Duyệt cây con trái  cout << root->data << " "; // Xử lý node hiện tại  inOrder(root->right); // Duyệt cây con phải  }  // Hàm duyệt cây theo hậu tự (Post-order)  void postOrder(Node\* root) {  if (root == nullptr) return;  postOrder(root->left); // Duyệt cây con trái  postOrder(root->right); // Duyệt cây con phải  cout << root->data << " "; // Xử lý node hiện tại  }  // Hàm duyệt cây theo cấp (Level-order) sử dụng Queue  void levelOrder(Node\* root) {  if (root == nullptr) return;  queue<Node\*> q;  q.push(root);  while (!q.empty()) {  Node\* current = q.front();  q.pop();  cout << current->data << " "; // Xử lý node hiện tại  if (current->left != nullptr) q.push(current->left); // Thêm con trái vào hàng đợi  if (current->right != nullptr) q.push(current->right); // Thêm con phải vào hàng đợi  }  }  // Hàm chèn phần tử vào cây nhị phân tìm kiếm (BST)  Node\* insert(Node\* root, int value) {  if (root == nullptr) {  return new Node(value);  }  if (value < root->data) {  root->left = insert(root->left, value);  } else {  root->right = insert(root->right, value);  }  return root;  }  // Hàm tìm kiếm phần tử trong cây nhị phân tìm kiếm (BST)  Node\* search(Node\* root, int value) {  if (root == nullptr || root->data == value) {  return root;  }  if (value < root->data) {  return search(root->left, value); // Tìm trong cây con trái  } else {  return search(root->right, value); // Tìm trong cây con phải  }  }  // Hàm tìm phần tử thay thế (node nhỏ nhất trong cây con phải)  Node\* minNode(Node\* root) {  Node\* current = root;  while (current && current->left != nullptr) {  current = current->left;  }  return current;  }  // Hàm xóa phần tử khỏi cây nhị phân tìm kiếm (BST)  Node\* deleteNode(Node\* root, int value) {  if (root == nullptr) return root;  if (value < root->data) {  root->left = deleteNode(root->left, value);  } else if (value > root->data) {  root->right = deleteNode(root->right, value);  } else {  // Node cần xóa là node hiện tại  if (root->left == nullptr) {  Node\* temp = root->right;  delete root;  return temp;  } else if (root->right == nullptr) {  Node\* temp = root->left;  delete root;  return temp;  }  // Node có hai con, thay thế bằng node nhỏ nhất trong cây con phải  Node\* temp = minNode(root->right);  root->data = temp->data;  root->right = deleteNode(root->right, temp->data);  }  return root;  } |

**3.2. Kết quả chạy**



# **CHUONG 7: ĐỒ THỊ**

## **7.1. khái niệm đồ thị**

**Đồ thị** (Graph) là một cấu trúc dữ liệu được sử dụng để mô tả mối quan hệ giữa các đối tượng. Một đồ thị thường được định nghĩa bởi hai thành phần chính:

**Tập hợp các đỉnh (Vertices)**: Đây là các đối tượng trong đồ thị. Mỗi đỉnh có thể đại diện cho một thực thể, chẳng hạn như người, địa điểm, hoặc bất kỳ loại đối tượng nào.

**Tập hợp các cạnh (Edges)**: Các cạnh kết nối các đỉnh với nhau. Chúng có thể là có hướng (directed) hoặc không có hướng (undirected). Nếu một cạnh là có hướng, nó chỉ cho phép đi từ đỉnh A đến đỉnh B mà không cho phép ngược lại.

Các khái niệm liên quan đến đồ thị bao gồm:

* **Đồ thị có hướng (Directed Graph)**: Đồ thị trong đó các cạnh có hướng, chỉ cho phép di chuyển từ đỉnh này đến đỉnh kia.
* **Đồ thị không có hướng (Undirected Graph)**: Đồ thị trong đó các cạnh không có hướng, cho phép di chuyển qua lại giữa các đỉnh.
* **Đồ thị có trọng số (Weighted Graph)**: Đồ thị trong đó các cạnh có gán trọng số (weight), thể hiện chi phí hoặc khoảng cách giữa các đỉnh.
* **Đồ thị liên thông (Connected Graph)**: Đồ thị mà trong đó có đường đi giữa mọi cặp đỉnh.
* **Cây (Tree)**: Một trường hợp đặc biệt của đồ thị, nơi không có chu trình và tất cả các đỉnh đều được kết nối.

## **7.2. Phép Duyệt Đồ Thị**

**Phép duyệt đồ thị** là quá trình truy cập tất cả các đỉnh trong đồ thị một cách có hệ thống. Mục tiêu của phép duyệt là khám phá toàn bộ cấu trúc của đồ thị để thu thập thông tin hoặc thực hiện các thao tác trên các đỉnh đó.

**Các Phương Pháp Duyệt Đồ Thị**

Có hai phương pháp chính để duyệt đồ thị: **Duyệt theo chiều sâu (Depth-First Search - DFS)** và **Duyệt theo chiều rộng (Breadth-First Search - BFS)**.

**7.2.1. Duyệt theo Chiều Sâu (DFS)**

**DFS** là một thuật toán duyệt bắt đầu từ một đỉnh, đi sâu vào các nhánh của đồ thị cho đến khi không còn đỉnh nào để truy cập. Sau đó, thuật toán sẽ quay lại và khám phá các nhánh khác. DFS có thể được triển khai bằng đệ quy hoặc ngăn xếp (stack).

**Cách thức hoạt động của DFS**:

* Bắt đầu từ một đỉnh, đánh dấu nó là đã được thăm.
* Tiến hành truy cập tất cả các đỉnh kề của đỉnh hiện tại mà chưa được thăm.
* Lặp lại quá trình cho đến khi tất cả các đỉnh đã được thăm.

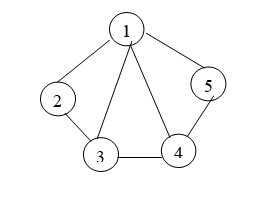
**2.2. Duyệt theo Chiều Rộng (BFS)**

**BFS** là một thuật toán duyệt mở rộng hơn, truy cập tất cả các đỉnh ở một độ sâu nhất định trước khi tiến hành khám phá các đỉnh ở độ sâu tiếp theo. BFS thường được triển khai bằng hàng đợi (queue).

**Cách thức hoạt động của BFS**:

* Bắt đầu từ một đỉnh, đánh dấu nó là đã được thăm và thêm vào hàng đợi.
* Khi đỉnh được lấy ra từ hàng đợi, kiểm tra tất cả các đỉnh kề và thêm vào hàng đợi nếu chưa được thăm.
* Tiếp tục cho đến khi hàng đợi trống.

## **7.3. thuật toán và kết quả chạy**

**3.1 thuật toán** 

|  |
| --- |
| // Duyet do thi theo chieu sau (DFS) va hien thi tung buoc  void DFS(int dinhBatDau) {  vector<bool> daTham(soDinh + 1, false);  stack<int> nganXep;  nganXep.push(dinhBatDau);  cout << "Duyet DFS tu dinh " << dinhBatDau << ": ";  while (!nganXep.empty()) {  int dinh = nganXep.top();  nganXep.pop();  if (!daTham[dinh]) {  cout << dinh << " ";  daTham[dinh] = true;  }  // Day cac dinh ke vao ngan xep neu chua duoc tham  for (auto it = danhSachKe[dinh].rbegin(); it != danhSachKe[dinh].rend(); ++it) {  if (!daTham[\*it]) {  nganXep.push(\*it);  }  }  }  cout << endl;  }  // Duyet do thi theo chieu rong (BFS) va hien thi tung buoc  void BFS(int dinhBatDau) {  vector<bool> daTham(soDinh + 1, false);  queue<int> hangDoi;  hangDoi.push(dinhBatDau);  daTham[dinhBatDau] = true;  cout << "Duyet BFS tu dinh " << dinhBatDau << ": ";  while (!hangDoi.empty()) {  int dinh = hangDoi.front();  hangDoi.pop();  cout << dinh << " ";  // Dua cac dinh ke vao hang doi neu chua duoc tham  for (int ke : danhSachKe[dinh]) {  if (!daTham[ke]) {  daTham[ke] = true;  hangDoi.push(ke);  }  }  }  cout << endl;  }  }; |

3.2. kết quả chạy



# **CHƯƠNG 8: XẮP SẾP**

## **8.1.khái niệm xắp sếp**

**8.1.1 Giải thuật sắp xếp chọn (Selection Sort)**

Khái niệm sắp xếp trong cấu trúc dữ liệu và giải thuật là quá trình sắp xếp lại các phần tử trong một danh sách hoặc một mảng theo một thứ tự nhất định, như thứ tự tăng dần hoặc giảm dần. Sắp xếp là một trong những bài toán cơ bản trong khoa học máy tính và được sử dụng rộng rãi trong nhiều ứng dụng để tổ chức và xử lý dữ liệu hiệu quả.

Các loại sắp xếp phổ biến

1. Sắp xếp nổi bọt (Bubble Sort):

* Là thuật toán sắp xếp đơn giản, hoạt động bằng cách so sánh từng cặp phần tử liền kề và hoán đổi nếu chúng không theo thứ tự mong muốn.
* Thời gian chạy: O(n^2)
* Ưu điểm: Dễ cài đặt, phù hợp cho mảng có kích thước nhỏ.
* Nhược điểm: Hiệu suất thấp khi làm việc với dữ liệu lớn.

1. Sắp xếp chọn (Selection Sort):

* Tìm phần tử nhỏ nhất (hoặc lớn nhất) trong danh sách và đặt nó vào vị trí đầu tiên, sau đó lặp lại cho phần còn lại của danh sách.
* Thời gian chạy: O(n^2)
* Ưu điểm: Đơn giản, dễ hiểu.
* Nhược điểm: Không hiệu quả với mảng lớn.

1. Sắp xếp chèn (Insertion Sort):

* Phần tử được lấy ra và chèn vào vị trí thích hợp trong phần đã được sắp xếp của danh sách.
* Thời gian chạy O(n^2), nhưng có thể nhanh hơn khi dữ liệu đã gần như được sắp xếp.
* Ưu điểm: Tốt cho danh sách nhỏ và danh sách đã gần sắp xếp.
* Nhược điểm: Không tối ưu với danh sách lớn.

1. Sắp xếp nhanh (Quick Sort):

* Chọn một phần tử làm "pivot" (chốt) và chia danh sách thành hai phần: một phần với các phần tử nhỏ hơn pivot và một phần với các phần tử lớn hơn pivot. Lặp lại quá trình này cho đến khi danh sách được sắp xếp.
* Thời gian chạy trung bình: (n \log n)
* Ưu điểm: Hiệu quả cao cho mảng lớn.
* Nhược điểm: Trường hợp xấu nhất là O(n^2) khi mảng đã được sắp xếp.

1. Sắp xếp trộn (Merge Sort):

* Chia mảng thành hai nửa, sắp xếp từng nửa và sau đó trộn chúng lại để có được mảng đã sắp xếp.
* Thời gian chạy: O(n \log n)
* Ưu điểm: Đảm bảo hiệu suất tốt trong mọi trường hợp.
* Nhược điểm: Sử dụng thêm bộ nhớ.

1. Sắp xếp vun đống (Heap Sort):

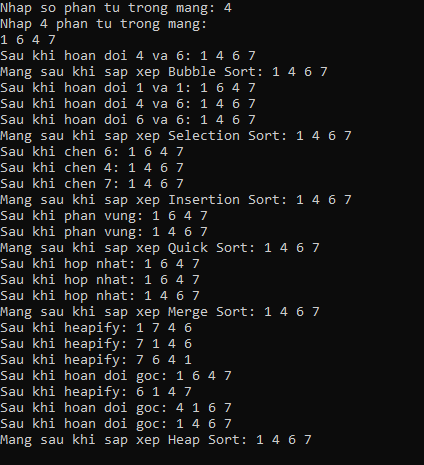
* Sử dụng cấu trúc dữ liệu Heap để tìm và đặt phần tử lớn nhất (hoặc nhỏ nhất) vào vị trí cuối cùng của mảng, sau đó tiếp tục với phần còn lại.
* Thời gian chạy: O(n \log n)
* Ưu điểm: Hiệu quả và không cần bộ nhớ phụ trợ.
* Nhược điểm: Khó cài đặt hơn các thuật toán khác.

## **8.2. thuật toán và kế quả chạy**

**8.2.1 thuật toán**

|  |
| --- |
| // Hàm sắp xếp nổi bọt (Bubble Sort)  void bubbleSort(vector<int>& arr) {  int n = arr.size();  for (int i = 0; i < n - 1; ++i) {  for (int j = 0; j < n - i - 1; ++j) {  if (arr[j] > arr[j + 1]) {  swap(arr[j], arr[j + 1]);  // In ra mảng sau mỗi lần đổi chỗ  cout << "Sau khi hoan doi " << arr[j] << " va " << arr[j + 1] << ": ";  for (int num : arr) cout << num << " ";  cout << endl;  }  }  }  }  // Hàm sắp xếp chọn (Selection Sort)  void selectionSort(vector<int>& arr) {  int n = arr.size();  for (int i = 0; i < n - 1; ++i) {  int minIdx = i;  for (int j = i + 1; j < n; ++j) {  if (arr[j] < arr[minIdx]) {  minIdx = j;  }  }  swap(arr[i], arr[minIdx]);  // In ra mảng sau mỗi lần hoán đổi  cout << "Sau khi hoan doi " << arr[i] << " va " << arr[minIdx] << ": ";  for (int num : arr) cout << num << " ";  cout << endl;  }  }  // Hàm sắp xếp chèn (Insertion Sort)  void insertionSort(vector<int>& arr) {  int n = arr.size();  for (int i = 1; i < n; ++i) {  int key = arr[i];  int j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  --j;  }  arr[j + 1] = key;  // In ra mảng sau mỗi lần chèn  cout << "Sau khi chen " << key << ": ";  for (int num : arr) cout << num << " ";  cout << endl;  }  }  // Hàm sắp xếp nhanh (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;  }  void quickSort(vector<int>& arr, int low, int high) {  if (low < high) {  int pi = partition(arr, low, high);  // In ra mảng sau mỗi lần phân vùng  cout << "Sau khi phan vung: ";  for (int num : arr) cout << num << " ";  cout << endl;    quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  }  }  // Hàm sắp xếp trộn (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++];  } else {  arr[k++] = R[j++];  }  }  while (i < n1) {  arr[k++] = L[i++];  }  while (j < n2) {  arr[k++] = R[j++];  }  // In ra mảng sau mỗi lần hợp nhất  cout << "Sau khi hop nhat: ";  for (int num : arr) cout << num << " ";  cout << endl;  }  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);  }  }  // Hàm sắp xếp vun đống (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]);  // In ra mảng sau mỗi lần heapify  cout << "Sau khi heapify: ";  for (int num : arr) cout << num << " ";  cout << endl;  heapify(arr, n, largest);  }  }  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 >= 1; --i) {  swap(arr[0], arr[i]);  // In ra mảng sau mỗi lần hoán đổi gốc  cout << "Sau khi hoan doi goc: ";  for (int num : arr) cout << num << " ";  cout << endl;  heapify(arr, i, 0);  }  } |

**8.2.2 kết quả chạy**



# **CHƯƠNG 9: TÌM KIẾM**

## **9.1. Khái niệm**

**Tìm kiếm** trong khoa học máy tính là quá trình xác định vị trí hoặc xác minh sự tồn tại của một phần tử trong một cấu trúc dữ liệu. Quá trình này có thể được thực hiện trên các mảng, danh sách, cây, đồ thị, hoặc các cấu trúc dữ liệu khác. Tìm kiếm là một thao tác cơ bản trong việc xử lý dữ liệu và là một phần quan trọng trong nhiều thuật toán và ứng dụng.

Có nhiều phương pháp tìm kiếm khác nhau, tùy thuộc vào kiểu dữ liệu và yêu cầu về hiệu suất. Dưới đây là một số phương pháp tìm kiếm phổ biến:

**Tìm kiếm tuyến tính (Linear Search)**

* **Mô tả**: Đây là thuật toán tìm kiếm đơn giản nhất, trong đó bạn sẽ duyệt qua từng phần tử trong mảng (hoặc danh sách) và kiểm tra xem phần tử đó có phải là phần tử cần tìm hay không.
* **Độ phức tạp**:

Thời gian: O(n) (vì phải kiểm tra tất cả phần tử trong danh sách)

Không yêu cầu mảng phải được sắp xếp.

* **Ứng dụng**: Tìm kiếm trong các danh sách không có thứ tự hoặc các cấu trúc dữ liệu mà không yêu cầu hiệu suất quá cao.

**Tìm kiếm nhị phân (Binary Search)**

* **Mô tả**: Tìm kiếm nhị phân được áp dụng trên các mảng đã được **sắp xếp**. Thuật toán này chia mảng thành hai phần, so sánh phần tử cần tìm với phần tử ở giữa mảng, sau đó tiếp tục tìm kiếm trong một nửa mảng còn lại.
* **Độ phức tạp**:

Thời gian: O(log n)

Yêu cầu mảng phải được sắp xếp.

* **Ứng dụng**: Tìm kiếm hiệu quả trong các mảng đã sắp xếp.

**Tìm kiếm theo cây (Tree Search)**

* **Mô tả**: Dùng để tìm kiếm trong các cấu trúc cây như **cây nhị phân tìm kiếm** (BST), trong đó các phần tử con trái luôn nhỏ hơn phần tử cha và các phần tử con phải luôn lớn hơn phần tử cha.
* **Độ phức tạp**:

Thời gian: O(log n) đối với cây cân bằng, nhưng có thể O(n) trong trường hợp cây mất cân bằng.

* **Ứng dụng**: Cây nhị phân tìm kiếm, cây AVL, cây đỏ đen.

**Tìm kiếm theo đồ thị (Graph Search)**

* **Mô tả**: Được áp dụng khi tìm kiếm trong các cấu trúc đồ thị, với các thuật toán phổ biến như **Tìm kiếm theo chiều rộng (BFS)** và **Tìm kiếm theo chiều sâu (DFS)**.

BFS: Tìm kiếm theo chiều rộng, sẽ tìm kiếm từ một đỉnh bắt đầu, duyệt qua tất cả các đỉnh gần nhất trước rồi tiếp tục với các đỉnh xa hơn.

DFS: Tìm kiếm theo chiều sâu, sẽ đi "sâu" vào mỗi nhánh của đồ thị trước khi quay lại và tiếp tục tìm kiếm.

* **Độ phức tạp**:

BFS và DFS đều có độ phức tạp thời gian: O(V + E) với V là số đỉnh và E là số cạnh của đồ thị.

* **Ứng dụng**: Tìm kiếm trong các bài toán đồ thị như tìm đường đi, xác định các thành phần liên thông, tìm kiếm chu trình trong đồ thị.

**Tìm kiếm băm (Hashing)**

* **Mô tả**: Sử dụng bảng băm để ánh xạ khóa tìm kiếm thành chỉ mục trong bảng băm. Từ đó có thể tìm kiếm phần tử rất nhanh.
* **Độ phức tạp**:

Thời gian trung bình: O(1), nhưng có thể O(n) trong trường hợp xấu (nếu có nhiều xung đột băm).

* **Ứng dụng**: Sử dụng trong các bài toán cần tìm kiếm nhanh như tìm kiếm trong từ điển, bộ đệm, hoặc bảng tra cứu.

**Tìm kiếm trong mảng đã sắp xếp (Interpolation Search)**

* **Mô tả**: Là một biến thể của tìm kiếm nhị phân, nhưng thay vì chia mảng thành hai phần đều, thuật toán này sẽ dựa vào giá trị của phần tử cần tìm để xác định "vị trí" của nó trong mảng, giả định rằng các phần tử được phân phối đều.
* **Độ phức tạp**:

Thời gian: O(log log n) trong trường hợp tốt nhất (khi mảng được phân phối đều).

Thời gian xấu nhất: O(n).

* **Ứng dụng**: Thích hợp cho các mảng có dữ liệu phân phối đều.

**Tìm kiếm trong mảng đã sắp xếp với các phép toán mở rộng (Exponential Search)**

* **Mô tả**: Là sự kết hợp giữa tìm kiếm nhị phân và một bước kiểm tra mở rộng. Nó bắt đầu tìm kiếm theo cấp số mũ (tăng gấp đôi mỗi lần), sau đó chuyển sang tìm kiếm nhị phân khi tìm được khoảng chứa phần tử cần tìm.
* **Độ phức tạp**:

Thời gian: O(log n)

* **Ứng dụng**: Tìm kiếm hiệu quả trong các mảng rất lớn.

## **9.2. thuật toán và kết quả chạy**

**9.2.1. thuật toán**

|  |
| --- |
| // 1. Tim kiem tuyen tinh (Linear Search)  int linearSearch(const vector<int>& arr, int target) {  for (int i = 0; i < arr.size(); ++i) {  if (arr[i] == target) {  return i; // Tra ve chi so neu tim thay  }  }  return -1; // Khong tim thay  }  // 2. Tim kiem nhi phan (Binary Search)  int binarySearch(const vector<int>& arr, int target) {  int left = 0, right = arr.size() - 1;  while (left <= right) {  int mid = left + (right - left) / 2;  if (arr[mid] == target) {  return mid; // Tra ve chi so neu tim thay  }  if (arr[mid] < target) {  left = mid + 1;  } else {  right = mid - 1;  }  }  return -1; // Khong tim thay  }  // 3. Tim kiem trong cay nhi phan tim kiem (Binary Search Tree)  struct Node {  int data;  Node\* left;  Node\* right;  Node(int val) : data(val), left(nullptr), right(nullptr) {}  };  // Ham chen mot phan tu vao cay  Node\* insert(Node\* root, int val) {  if (root == nullptr) return new Node(val);  if (val < root->data) {  root->left = insert(root->left, val);  } else {  root->right = insert(root->right, val);  }  return root;  }  // Ham tim kiem trong cay  bool search(Node\* root, int target) {  if (root == nullptr) return false;  if (root->data == target) return true;  return (target < root->data) ? search(root->left, target) : search(root->right, target);  }  // 4. Tim kiem do thi (BFS, DFS)  void bfs(const vector<vector<int>>& graph, int start, int target) {  vector<bool> visited(graph.size(), false);  queue<int> q;  q.push(start);  visited[start] = true;  while (!q.empty()) {  int node = q.front();  q.pop();  if (node == target) {  cout << "Tim thay " << target << " trong BFS.\n";  return;  }  for (int neighbor : graph[node]) {  if (!visited[neighbor]) {  visited[neighbor] = true;  q.push(neighbor);  }  }  }  cout << "Khong tim thay " << target << " trong BFS.\n";  }  void dfs(const vector<vector<int>>& graph, int node, int target, vector<bool>& visited) {  if (node == target) {  cout << "Tim thay " << target << " trong DFS.\n";  return;  }  visited[node] = true;  for (int neighbor : graph[node]) {  if (!visited[neighbor]) {  dfs(graph, neighbor, target, visited);  }  }  } |

**9.2.2 kết quả chạy**

