**TRƯỜNG ĐẠI HỌC KỸ THUẬT THẦYNG NGHIỆP**



**KHOA ĐIỆN TỬ**

**BỘ MÔN: CÔNG NGHỆ THÔNG TIN**

✯✯✯

Ảnh có chứa biểu tượng, Nhãn hiệu, phù hiệu, vòng tròn

Mô tả được tạo tự động

**TIỂU LUẬN CẤU TRÚC DỮ LIỆU & GIẢI THUẬT**

|  |  |  |
| --- | --- | --- |
| **Giảng viên hướng dẫn** | **:** | **Nguyễn Thị Hương** |
| **Họ tên** | **:** | **Tạ Phạm Đình Hòa** |
| **MSSV** | **:** |  |
| **Ngành học** | **:** | **Kỹ thuật Máy tính** |
| **Lớp** | **:** | **K58KMT.01** |
|  |  |  |
|  |  |  |

**Thái Nguyên 2023**

|  |  |
| --- | --- |
| TRƯỜNG ĐẠI HỌC KTCN | CỘNG HOÀ XÃ HỘI CHỦ NGHĨA VIỆT NAM |

**PHIẾU GIAO BÀI TẬP TIỂU LUẬN**

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

Họ tên sinh viên : Tạ Phạm Đình Hòa

MSSV :

Lớp : K58KMT Ngành : Kỹ thuật máy tính

Giáo viên hướng dẫn:. Nguyễn Thị Hương

1. 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

1. Thời gian thực hiện: từ 08/10/2024 đến 05/11/2024
2. Hình thức nộp:
3. Báo cáo word
4. File source code chương trình

MỤC LỤC

[CHƯƠNG 3: ĐỆ QUY 4](#_Toc12377)

[3.1 Khái niệm về đệ quy: 5](#_Toc20097)

[3.2. Ưu điểm và nhược điểm : 5](#_Toc2172)

[3.3. Giải thuật đệ quy: 6](#_Toc14563)

[3.4. Độ phức tạp của thuật toán: 6](#_Toc31129)

[3.5. Ví dụ: 6](#_Toc15074)

[CHƯƠNG 4: MẢNG VÀ DANH SÁCH 6](#_Toc16696)

[4.1. Khái niệm: 6](#_Toc4692)

[4.3. Giải thuật 8](#_Toc27287)

[4.4. Độ phức tạp của thuật toán: 9](#_Toc22326)

[4.5. Ví dụ: 9](#_Toc26771)

[CHƯƠNG 5: DANH SÁCH MÓC NỐI 9](#_Toc18557)

[5.1. Khái niệm: 9](#_Toc24495)

[5.2.Ưu điểm và nhược điểm: 9](#_Toc23673)

[5.3. Giải thuật 10](#_Toc10604)

[5.4. Độ phức tạp: 12](#_Toc6482)

[5.5. Ví dụ: 12](#_Toc9043)

[CHƯƠNG 6: CÂY (TREE) 12](#_Toc7781)

[6.1. Khái niệm: 12](#_Toc12127)

[6.2.Ưu điểm và nhược điểm của cây: 12](#_Toc19851)

[6.3. Giải thuật 13](#_Toc14509)

[6.4. Độ phức tạp: 13](#_Toc15243)

[6.5. Ví dụ: 14](#_Toc26255)

[CHƯƠNG 7: ĐỒ THỊ 14](#_Toc2388)

[7.1. Khái niệm: 14](#_Toc14725)

[7.2. Ưu điểm và nhược điểm của cây 14](#_Toc5027)

[7.3. Giải thuật 14](#_Toc26787)

[7.4. Độ phức tạp: 15](#_Toc5446)

[DFS: 15](#_Toc19302)

[BFS: 16](#_Toc30753)

[7.5. Ví dụ: 16](#_Toc15119)

[CHƯƠNG 8: SẮP XẾP 16](#_Toc13206)

[8.1. Khái niệm: 16](#_Toc28668)

[8.2. Ưu điểm và nhược điểm: 16](#_Toc19209)

[8.3. Giải thuật 17](#_Toc17045)

[8.4. Độ phức tạp: 19](#_Toc21900)

[8.5. Ví dụ: 20](#_Toc13505)

[Chương 9: Tìm kiếm 20](#_Toc3)

[9.1. Khái niệm: 20](#_Toc4107)

[9.2. Ưu điểm và nhược điểm 21](#_Toc17719)

[9.3. Giải thuật 22](#_Toc1497)

[9.4. Độ phức tạp: 23](#_Toc13234)

[9.5. Ví dụ: 23](#_Toc27230)

**CHƯƠNG 3: ĐỆ QUY**

**3.1 Khái niệm về đệ quy:**

Đệ quy là một kỹ thuật trong lập trình, trong đó một hàm tự gọi lại chính nó cho đến khi đạt đến một điều kiện dừng cụ thể. Khi một hàm đệ quy được gọi, nó tiếp tục thực hiện các bước xử lý và gọi lại chính nó với một phiên bản nhỏ hơn hoặc đơn giản hơn của bài toán cho đến khi đáp ứng điều kiện dừng. Khi đó, chương trình sẽ quay ngược lại qua các lời gọi hàm, hoàn thành từng bước và kết thúc.

**3.2. Ưu điểm và nhược điểm :**

**Ưu điểm của đệ quy:**

Đơn giản và dễ hiểu cho các bài toán phân cấp: Đệ quy giúp chương trình rõ ràng và ngắn gọn khi xử lý các bài toán có cấu trúc lặp lại hoặc phân cấp, như cây nhị phân hoặc bài toán chia để trị.

Giảm thiểu mã lệnh: Đệ quy giúp giảm số lượng mã lệnh cần viết, đặc biệt trong những bài toán phức tạp như duyệt cây, tìm kiếm trong đồ thị, tính giai thừa, hoặc chuỗi Fibonacci.

Giải quyết các bài toán phức tạp: Đệ quy thường hiệu quả khi áp dụng cho các bài toán có tính lặp lại và phụ thuộc vào kết quả của các bước trước, chẳng hạn như bài toán tháp Hà Nội hay tìm đường trong mê cung.

**Nhược điểm của đệ quy:**

Tốn bộ nhớ và dễ gây tràn ngăn xếp: Mỗi lần gọi đệ quy, hệ thống cần lưu trạng thái của hàm vào ngăn xếp. Nếu có quá nhiều lời gọi đệ quy, đặc biệt là không có điều kiện dừng hợp lý, sẽ dễ gây tràn ngăn xếp (stack overflow).

Hiệu suất kém trong một số trường hợp: Đệ quy thường gây lặp lại các tính toán, nhất là khi không dùng kỹ thuật ghi nhớ (memoization), dẫn đến hiệu suất chậm. Ví dụ, tính chuỗi Fibonacci bằng đệ quy sẽ chậm hơn rất nhiều so với sử dụng vòng lặp hoặc kỹ thuật lưu trữ.

1. Khó hiểu và khó debug hơn: Đệ quy có thể gây khó hiểu cho người lập trình khi lần theo từng bước thực thi, và việc gỡ lỗi (debug) cũng phức tạp hơn so với vòng lặp.

**3.3. Giải thuật đệ quy:**

**- Tính tổng:**

function TinhTong(n):

if n == 1:

return 1

else:

return n + TinhTong(n - 1)

**3.4. Độ phức tạp của thuật toán:**

Độ phức tạp thời gian của một thuật toán đệ quy phụ thuộc vào số lần mà hàm tự gọi lại chính nó và công việc thực hiện trong mỗi lần gọi

**3.5. Ví dụ:**

Tính tổng S = 1+2+3+.......+ n với n là số nguyên dương được nhâp từ bàn phím

**CHƯƠNG 4: MẢNG VÀ DANH SÁCH**

**4.1. Khái niệm:**

**1. Stack (Ngăn xếp):**

Stack là một cấu trúc dữ liệu tuân theo nguyên tắc LIFO (Last In, First Out), tức là phần tử được thêm vào sau cùng sẽ được lấy ra đầu tiên. Trong stack, ta chỉ có thể thao tác với phần tử ở đỉnh (top) của nó.

**Ưu điểm:**

Dễ triển khai và quản lý khi chỉ cần thao tác với phần tử ở đỉnh.

Thao tác thêm (push) và xóa (pop) có thời gian xử lý nhanh, thường là O(1)

Rất hữu ích cho các vấn đề liên quan đến đệ quy, kiểm tra tính cân bằng của dấu ngoặc, xử lý biểu thức, và thao tác hoàn tác.

**Nhược điểm:**

Hạn chế về truy cập, chỉ làm việc được với phần tử ở đỉnh nên không linh hoạt nếu cần truy cập ngẫu nhiên.

Dễ gây tràn ngăn xếp (stack overflow) nếu stack có kích thước cố định hoặc gọi đệ quy quá sâu.

**Ứng dụng:** Stack thường được sử dụng trong việc gọi hàm đệ quy, kiểm tra dấu ngoặc, duyệt cây nhị phân không đệ quy, thuật toán sắp xếp, v.v.

**2. Queue (Hàng đợi):**

Queue là một cấu trúc dữ liệu tuân theo nguyên tắc FIFO (First In, First Out), nghĩa là phần tử được thêm vào đầu tiên sẽ được lấy ra đầu tiên. Queue có hai đầu: một đầu để thêm phần tử (rear) và một đầu để lấy phần tử (front).

**Ưu điểm:**

Hữu ích cho việc xử lý dữ liệu theo thứ tự xuất hiện, giữ đúng trình tự của các phần tử.

Các thao tác thêm và xóa ở đầu và cuối thường có độ phức tạp O(1)O(1)O(1), đảm bảo hiệu quả cho các ứng dụng xử lý theo hàng đợi.

**Nhược điểm:**

**Truy cập ngẫu nhiên không được hỗ trợ, chỉ có thể truy cập đầu và cuối, gây khó khăn cho các thao tác yêu cầu linh hoạt hơn.**

**Nếu sử dụng queue có kích thước cố định (như mảng), có thể dễ dàng gây tràn hoặc lãng phí bộ nhớ.**

**Ứng dụng:**

**Queue thường được sử dụng trong các hệ thống cần xử lý dữ liệu theo thứ tự xuất hiện như lập lịch CPU, quản lý tác vụ in ấn, tìm kiếm theo chiều rộng (BFS), và quản lý hàng đợi khách hàng.**

**4.3. Giải thuật**

**4.3.1. Stack:**

**Thêm (Push) một phần tử vào stack**

Push(stack, element):

if stack is full:

print "Stack Overflow"

else:

top = top + 1

stack[top] = element

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

Pop(stack):

if stack is empty:

print "Stack Underflow"

else:

element = stack[top]

top = top - 1

return element

**4.3.2. Queue:**

**Thêm phần tử vào cuối queue**

Enqueue(queue, element):

if queue is full:

print "Queue Overflow"

else:

rear = rear + 1

queue[rear] = element

**Lấy ra phần tử ở đầu queue**

Dequeue(queue):

if queue is empty:

print "Queue Underflow"

else:

element = queue[front]

front = front + 1

return element

**4.4. Độ phức tạp của thuật toán:**

Độ phức tạp của thuật toán đều là O(1) riêng hàm khởi tạo stack và queue là O(n).

**4.5. Ví dụ**:

Thực hiên thêm và lấy phần tử ở đỉnh Stack và Queue

**CHƯƠNG 5: DANH SÁCH MÓC NỐI**

**5.1. Khái niệm:**

Danh sách móc nối (Linked List) là một cấu trúc dữ liệu tuyến tính, trong đó các phần tử được lưu trữ không liên tiếp trong bộ nhớ. Mỗi phần tử của danh sách này, được gọi là một node (điểm nút), chứa hai phần chính:

Dữ liệu: Phần chứa giá trị của phần tử.

Liên kết: Phần này chứa địa chỉ của phần tử tiếp theo trong danh sách.

**5.2.Ưu điểm và nhược điểm:**

**Ưu điểm:**

Dễ dàng thay đổi kích thước: Danh sách móc nối không yêu cầu cấp phát bộ nhớ liên tiếp, do đó nó có thể mở rộng hoặc thu nhỏ một cách linh hoạt, mà không cần lo lắng về việc thiếu bộ nhớ liên tiếp như trong mảng.

Chèn và xóa dễ dàng: Các thao tác chèn và xóa phần tử trong danh sách móc nối (ở đầu, cuối, hay giữa) có thể thực hiện một cách nhanh chóng (O(1)) nếu ta đã có vị trí của phần tử cần thao tác.

Không gian linh hoạt: Các node trong danh sách có thể được phân bổ không liên tiếp trong bộ nhớ.

**Nhược điểm:**

Truy xuất không hiệu quả: Để truy cập phần tử tại vị trí bất kỳ trong danh sách, ta phải duyệt qua các node từ đầu danh sách đến node cần truy cập, có độ phức tạp O(n), trong khi mảng có thể truy xuất trực tiếp theo chỉ số.

Tiêu tốn bộ nhớ: Mỗi node cần thêm một con trỏ (hoặc hai trong danh sách móc nối đôi), làm tăng bộ nhớ sử dụng so với mảng chỉ cần lưu trữ dữ liệu.

Khó quản lý hơn mảng: Vì các node không nằm liền nhau trong bộ nhớ, việc quản lý và thao tác với danh sách móc nối có thể phức tạp hơn khi so với mảng

**5.3. Giải thuật**

**1. Chèn một phần tử vào đầu danh sách (Insert at Beginning)**

void insertAtBeginning(Node\*& head, int value) {

Node\* newNode = new Node();

newNode->data = value;

newNode->next = head;

head = newNode; // Cập nhật lại con trỏ head

}

**2. Xóa phần tử khỏi danh sách (Delete a Node)**

void deleteNode(Node\*& head, int value) {

if (head == NULL) return; // Danh sách rỗng, không làm gì

// Nếu cần xóa phần tử đầu tiên

if (head->data == value) {

Node\* temp = head;

head = head->next;

delete temp;

return;

}

// Duyệt qua các node để tìm phần tử cần xóa

Node\* temp = head;

while (temp->next != NULL && temp->next->data != value) {

temp = temp->next;

}

// Nếu tìm thấy phần tử cần xóa

if (temp->next != NULL) {

Node\* delNode = temp->next;

temp->next = temp->next->next; // Bỏ qua node cần xóa

delete delNode;

}

}

**3. Tìm kiếm phần tử trong danh sách (Search a Node)**

Node\* search(Node\* head, int value) {

Node\* temp = head;

while (temp != NULL) {

if (temp->data == value) {

return temp; // Trả về con trỏ node tìm được

}

temp = temp->next;

}

return NULL; // Nếu không tìm thấy

}

**4.Duyệt và in danh sách (Print the List)**

void printList(Node\* head) {

Node\* temp = head;

while (temp != NULL) {

cout << temp->data << " ";

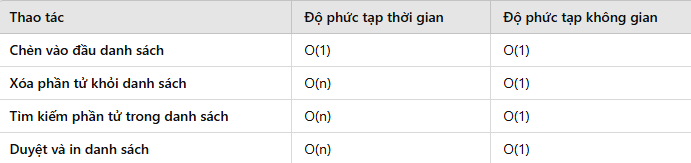
temp = temp->next;

}

cout << endl;

}

**5.4. Độ phức tạp:**



**5.5. Ví dụ:**

**Cho danh sách nối đơn co nút đầu danh sách trỏ bởi con trỏ Q. Hãy viết chương trình bổ sung một phần tử mới vào sau nút trỏ bởi con trỏ M.**

**CHƯƠNG 6: CÂY (TREE)**

**6.1. Khái niệm:**

**Cây (Tree) là một cấu trúc dữ liệu có cấu trúc phân cấp, được sử dụng để biểu diễn các mối quan hệ giữa các phần tử trong một hệ thống phân cấp hoặc phân loại. Mỗi phần tử trong cây gọi là một nút (node), và cây bắt đầu từ một nút gốc gọi là nút gốc (root). Các nút con được kết nối với nhau thông qua các cạnh (edges).**

**6.2.Ưu điểm và nhược điểm của cây:**

**Ưu điểm:**

**Cây là cấu trúc dữ liệu phân cấp tự nhiên, rất hữu ích để biểu diễn các mối quan hệ cha-con (ví dụ, hệ thống thư mục, quản lý cây quyết định).**

**Tìm kiếm, chèn, xóa trong cây (đặc biệt là cây nhị phân tìm kiếm) có thể thực hiện nhanh với độ phức tạp O(log n) trong trường hợp cân bằng.**

**Cây giúp tổ chức dữ liệu theo cấu trúc phân cấp, dễ dàng quản lý và truy xuất.**

**Nhược điểm:**

**Việc duy trì sự cân bằng trong cây (đặc biệt là các cây nhị phân tìm kiếm) có thể phức tạp và yêu cầu chi phí tính toán thêm.**

**Các thao tác trên cây có thể không hiệu quả nếu cây bị mất cân bằng (ví dụ, cây chỉ có một nhánh dài, giống như danh sách liên kết).**

**6.3. Giải thuật**

**6.3.1. Duyệt trước (Pre-order Traversal)**

void preOrder(Node\* root) {

if (root == nullptr) return; // Kiểm tra nút rỗng

cout << root->data << " "; // In giá trị của nút gốc

preOrder(root->left); // Duyệt nút trái

preOrder(root->right); // Duyệt nút phải

}

**6.3.2. Duyệt giữa (In-order Traversal)**

void inOrder(Node\* root) {

if (root == nullptr) return; // Kiểm tra nút rỗng

inOrder(root->left); // Duyệt nút trái

cout << root->data << " "; // In giá trị của nút gốc

inOrder(root->right); // Duyệt nút phải

}

**6.3.3. Duyệt sau (Post-order Traversal)**

void postOrder(Node\* root) {

if (root == nullptr) return; // Kiểm tra nút rỗng

postOrder(root->left); // Duyệt nút trái

postOrder(root->right); // Duyệt nút phải

cout << root->data << " "; // In giá trị của nút gốc

**6.4. Độ phức tạp:**

- Duyệt trước (Pre-order): O(n)

- Duyệt giữa (In-order): O(n)

- Duyệt sau (Post-order): O(n)

**6.5. Ví dụ:**

Viết chuong trình thực hiện các thao tác duyệt các nút theo thứ tự trước, giữa, sau của cây nhị phân

**CHƯƠNG 7: ĐỒ THỊ**

**7.1. Khái niệm:**

Đồ thị (Graph) là một cấu trúc dữ liệu quan trọng trong toán học và tin học, dùng để mô tả các mối quan hệ giữa các đối tượng. Đồ thị gồm các đỉnh (vertices) và các cạnh (edges) nối giữa các đỉnh. Đồ thị thường được sử dụng để mô hình hóa các mối quan hệ trong các bài toán như mạng xã hội, mạng lưới giao thông, mối quan hệ giữa các trang web, v.v.

**7.2. Ưu điểm và nhược điểm của cây**

**Ưu điểm của đồ thị:**

Mô hình hóa quan hệ phức tạp: Đại diện các mối quan hệ giữa các đối tượng.

Linh hoạt: Có thể đại diện quan hệ có hướng và không có hướng, với trọng số.

Dễ mở rộng: Dễ dàng thêm hoặc xóa đỉnh, cạnh.

Ứng dụng rộng: Dùng trong tìm đường, mạng xã hội, giao thông, v.v.

**Nhược điểm của đồ thị:**

Độ phức tạp duy trì: Cập nhật đồ thị lớn hoặc động tốn tài nguyên.

Yêu cầu bộ nhớ lớn: Đặc biệt với đồ thị dày đặc.

Thuật toán phức tạp: Một số thuật toán tốn nhiều thời gian, đặc biệt với đồ thị lớn.

Khó xử lý đồ thị không liên thông: Cần thuật toán đặc biệt.

Cần kiến thức chuyên sâu: Yêu cầu hiểu rõ lý thuyết đồ thị và các thuật toán.

**7.3. Giải thuật**

**Duyệt theo chiều sâu:**

void DFS(int v) {

visited[v] = true; // Đánh dấu đỉnh đã thăm

cout << v << " "; // Xử lý đỉnh

for (int i : adj[v]) { // Duyệt tất cả đỉnh kề

if (!visited[i]) {

DFS(i); // Đệ quy cho đỉnh kề

}

}

}

**Duyệt theo chiều rộng:**

void BFS(int start) {

queue<int> q; // Tạo hàng đợi

visited[start] = true; // Đánh dấu đỉnh đã thăm

q.push(start); // Đưa đỉnh vào hàng đợi

while (!q.empty()) {

int v = q.front(); // Lấy đỉnh ở đầu hàng đợi

q.pop(); // Xóa đỉnh khỏi hàng đợi

cout << v << " "; // Xử lý đỉnh

for (int i : adj[v]) { // Duyệt tất cả đỉnh kề

if (!visited[i]) {

visited[i] = true; // Đánh dấu đỉnh đã thăm

q.push(i); // Đưa đỉnh vào hàng đợi

}

}

}

}

**7.4. Độ phức tạp:**

**DFS:**

- Thời gian: O(V + E) (thăm tất cả các đỉnh và cạnh

- Không gian: O(V) (để lưu trữ mảng đã thăm và ngăn xếp trong đệ quy

**BFS:**

- Thời gian: O(V + E) (thăm tất cả các đỉnh và cạnh)

- Không gian: O(V) (để lưu trữ mảng đã thăm và hàng đợi)

**7.5. Ví dụ:**

Cho đồ thị vô hướng với số đỉnh và số cạnh được nhập từ bàn phím. Thực hiện duyệt đồ thị theo phương pháp duyệt theo chiều sâu

**CHƯƠNG 8: SẮP XẾP**

**8.1. Khái niệm:**

Sắp xếp (Sorting) là một khái niệm quan trọng trong cấu trúc dữ liệu và giải thuật, liên quan đến việc tổ chức lại các phần tử của một tập hợp (thường là mảng hoặc danh sách) theo một thứ tự nhất định. Thứ tự này có thể là tăng dần (từ nhỏ đến lớn) hoặc giảm dần (từ lớn đến nhỏ).

**8.2. Ưu điểm và nhược điểm:**

**1. Sắp xếp nổi bọt (Bubble Sort)**

Ưu điểm: Dễ hiểu, không cần bộ nhớ thêm.

Nhược điểm: Hiệu suất kém (O(n²)), không hiệu quả với dữ liệu lớn.

**2. Sắp xếp chọn (Selection Sort)**

Ưu điểm: Đơn giản, không cần bộ nhớ thêm.

Nhược điểm: Hiệu suất thấp (O(n²)), không ổn định.

**3. Sắp xếp chèn (Insertion Sort)**

Ưu điểm: Hiệu quả với dữ liệu gần như đã sắp xếp (O(n)), ổn định.

Nhược điểm: Không hiệu quả với mảng lớn (O(n²)).

**4. Sắp xếp nhanh (Quick Sort)**

Ưu điểm: Hiệu suất cao (O(n log n)), nhanh cho dữ liệu lớn.

Nhược điểm: Không ổn định, trường hợp tệ nhất có thể là O(n²).

**5. Sắp xếp hợp nhất (Merge Sort)**

Ưu điểm: Ổn định, hiệu suất ổn định (O(n log n)).

Nhược điểm: Cần thêm bộ nhớ, phức tạp hơn để triển khai.

**8.3. Giải thuật**

**1. Sắp xếp nổi bọt (Bubble Sort)**

function bubble\_sort(arr):

n = length(arr)

for i from 0 to n-1:

for j from 0 to n-i-2:

if arr[j] > arr[j+1]:

swap(arr[j], arr[j+1])

**2. Sắp xếp chọn (Selection Sort)**

function selection\_sort(arr):

n = length(arr)

for i from 0 to n-1:

min\_index = i

for j from i+1 to n-1:

if arr[j] < arr[min\_index]:

min\_index = j

swap(arr[i], arr[min\_index])

**3. Sắp xếp chèn (Insertion Sort)**

function insertion\_sort(arr):

n = length(arr)

for i from 1 to n-1:

key = arr[i]

j = i - 1

while j >= 0 and key < arr[j]:

arr[j + 1] = arr[j]

j = j - 1

arr[j + 1] = key

**4. Sắp xếp nhanh (Quick Sort)**

function quick\_sort(arr):

if length(arr) <= 1:

return arr

pivot = arr[length(arr) // 2]

left = []

middle = []

right = []

for x in arr:

if x < pivot:

left.append(x)

else if x == pivot:

middle.append(x)

else:

right.append(x)

return quick\_sort(left) + middle + quick\_sort(right)

**5. Sắp xếp hợp nhất (Merge Sort)**

function merge\_sort(arr):

if length(arr) <= 1:

return arr

mid = length(arr) // 2

left = merge\_sort(arr[0:mid])

right = merge\_sort(arr[mid:length(arr)])

return merge(left, right)

function merge(left, right):

result = []

i = 0

j = 0

while i < length(left) and j < length(right):

if left[i] < right[j]:

result.append(left[i])

i = i + 1

else:

result.append(right[j])

j = j + 1

result.extend(left[i:])

result.extend(right[j:])

return result

**8.4. Độ phức tạp:**

**Bubble Sort**

Tốt nhất: O(n) - (nếu mảng đã được sắp xếp)

Trung bình: O(n^2)

Xấu nhất: O(n^2)

Không gian: O(1)

**Selection Sort**

Tốt nhất: O(n^2)

Trung bình: O(n^2)

Xấu nhất: O(n^2)

Không gian: O(1)

**Insertion Sort**

Tốt nhất: O(n) - (nếu mảng đã được sắp xếp)

Trung bình: O(n^2)

Xấu nhất: O(n^2)

Không gian: O(1)

**Quick Sort**

Tốt nhất: O(n log n)

Trung bình: O(n log n)

Xấu nhất: O(n^2) - (xảy ra khi pivot chọn không tốt)

Không gian: O(log n) - (do sử dụng đệ quy)

**Merge Sort**

Tốt nhất: O(n log n)

Trung bình: O(n log n)

Xấu nhất: O(n log n)

Không gian: O(n) - (do cần bộ nhớ để lưu các mảng con trong quá trình hợp nhất)

**8.5. Ví dụ:**

Viết chương trình thực hiện sắp xếp 1 mảng gồm n phần tử nhập từ bàn phím sử dụng các giải thuật sắp xếp.

**Chương 9: Tìm kiếm**

**9.1. Khái niệm:**

Tìm kiếm là một quá trình trong khoa học máy tính và lập trình, nhằm mục đích xác định vị trí hoặc nhận diện một phần tử cụ thể trong một tập hợp dữ liệu (như mảng, danh sách, hoặc cơ sở dữ liệu). Tìm kiếm có thể được thực hiện trên các cấu trúc dữ liệu khác nhau và có nhiều thuật toán khác nhau tùy thuộc vào tính chất của dữ liệu và yêu cầu của bài toán.

**9.2. Ưu điểm và nhược điểm**

**1. Tìm kiếm tuần tự (Sequential Search)**

Ưu điểm:

- Đơn giản và dễ cài đặt: Thuật toán rất dễ hiểu và thực hiện.

- Không yêu cầu sắp xếp: Có thể được sử dụng trên bất kỳ tập hợp dữ liệu nào, ngay cả khi chưa được sắp xếp.

Nhược điểm:

- Hiệu suất kém với dữ liệu lớn: Thời gian tìm kiếm tỉ lệ thuận với số lượng phần tử, dẫn đến hiệu suất thấp khi xử lý các tập dữ liệu lớn.

- Tối ưu không tốt: Không có cách nào để tối ưu hóa hơn nữa nếu dữ liệu không được sắp xếp.

**2. Tìm kiếm nhị phân (Binary Search)**

Ưu điểm:

- Hiệu suất cao: Thời gian tìm kiếm rất nhanh, đặc biệt cho các tập dữ liệu lớn. Độ phức tạp thời gian là O(log n).

- Tối ưu hóa tốt cho dữ liệu đã sắp xếp: Nếu dữ liệu được sắp xếp, tìm kiếm nhị phân là một trong những phương pháp hiệu quả nhất.

Nhược điểm:

- Yêu cầu dữ liệu phải được sắp xếp: Nếu dữ liệu chưa được sắp xếp, bạn cần sắp xếp nó trước khi áp dụng tìm kiếm nhị phân, điều này có thể tốn thời gian.

- Khó khăn trong việc cài đặt: So với tìm kiếm tuần tự, thuật toán tìm kiếm nhị phân phức tạp hơn trong việc cài đặt và quản lý các chỉ số.

**3. Tìm kiếm trong cây (Tree Search)**

Ưu điểm:

- Tìm kiếm nhanh: Nếu cây được cân bằng (như cây AVL hay cây đỏ đen), thời gian tìm kiếm có thể đạt O(log n).

- Hỗ trợ thêm thao tác: Ngoài việc tìm kiếm, cây còn hỗ trợ các thao tác khác như chèn và xóa một cách hiệu quả.

Nhược điểm:

- Cần cấu trúc dữ liệu phức tạp: Việc cài đặt và quản lý cây có thể phức tạp hơn so với tìm kiếm tuần tự hay nhị phân.

- Có thể không cân bằng: Nếu không được giữ cân bằng, độ phức tạp tìm kiếm có thể xấu đi đến O(n).

**9.3. Giải thuật**

**Tìm kiếm tuần tự (Sequential Search)**

function sequentialSearch(arr, key)

for i from 0 to length(arr) - 1 do

if arr[i] == key then

return i

return -1

**Tìm kiếm nhị phân (Binary Search)**

function binarySearch(arr, key)

left = 0

right = length(arr) - 1

while left <= right do

mid = left + (right - left) / 2

if arr[mid] == key then

return mid

else if arr[mid] < key then

left = mid + 1

else

right = mid - 1

return -1

**Tìm kiếm trong cây nhị phân (Binary Tree Search)**

function search(root, key)

if root == null then

return false

if root.data == key then

return true

if key < root.data then

return search(root.left, key)

else

return search(root.right, key)

**9.4. Độ phức tạp:**

**1. Tìm kiếm tuần tự (Sequential Search)**

Độ phức tạp thời gian: O(n)

Độ phức tạp không gian: O(1)

**2. Tìm kiếm nhị phân (Binary Search)**

Độ phức tạp thời gian: O(log n)

Độ phức tạp không gian: O(1)

**3. Tìm kiếm trong cây nhị phân (Binary Tree Search)**

Độ phức tạp thời gian: O(h) với h là chiều cao của cây

Độ phức tạp không gian: O(h) trong trường hợp tìm kiếm đệ quy, O(1) trong trường hợp tìm kiếm không đệ quy

**9.5. Ví dụ:**

Viết chương trình thực hiện tìm kiếm vị trí của các phần tử muốn tìm sử dụng các thuật toán tìm kiếm trên