**TRƯỜNG ĐẠI HỌC BẠC LIÊU**

**KHOA CÔNG NGHỆ THÔNG TIN**



**ĐỒ ÁN 1**

**TÊN ĐỀ TÀI**

**XÂY DỰNG CÂY TÌM KIẾM NHỊ PHÂN**

***Sinh viên thực hiện* *Mã Sinh viên***

**Phương Tấn Đạt 227480201090**

***Giảng viên hướng dẫn* Ts.Ngô Đức Lưu**

**HỌC KỲ I , 2024-2025**

**TRƯỜNG ĐẠI HỌC BẠC LIÊU**

**KHOA KỸ THUẬT & CÔNG NGHỆ**

A blue and white logo

Description automatically generated

**ĐỒ ÁN 1**

**TÊN ĐỀ TÀI**

**XÂY DỰNG CÂY TÌM KIẾM NHỊ PHÂN**

***Sinh viên thực hiện* *Mã Sinh viên***

**Phương Tấn Đạt 227480201090**

***Giảng viên hướng dẫn* Ts. Ngô Đức Lưu**

**HỌC KỲ I , 2024-2025**

**NHẬN XÉT CỦA GIẢNG VIÊN**

Bạc Liêu, *ngày … tháng … năm ……*

**GIẢNG VIÊN NHẬN XÉT**

**ĐÁNH GIÁ KẾT QUẢ THỰC HIỆN ĐỒ ÁN 1**

(Học kỳ I , Niên khóa 2024-2025)

**TÊN ĐỀ TÀI:** **XÂY DỰNG CÂY TÌM KIẾM NHỊ PHÂN**

**GIÁO VIÊN HƯỚNG DẪN:**

|  |  |  |
| --- | --- | --- |
| STT | HỌ VÀ TÊN | MSCB |
| 1 | Ts.Ngô Đức Lưu |  |

**SINH VIÊN THỰC HIỆN:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| STT | HỌ VÀ TÊN | MSSV | THƯỞNG  *(Tối đa 1,0 điểm)* | ĐIỂM |
| 1 | PHƯƠNG TẤN ĐẠT | 227480201090 |  |  |

|  |  |
| --- | --- |
| **I. HÌNH THỨC** *(Tối đa 0,5 điểm)* |  |
| **Bìa** *(tối đa 0,25 điểm)* |  |
| * Các tiêu đề: Trường ĐHBL, Khoa Kỹ Thuật và Công Nghệ * Loại đồ án: 1 * Tên đề tài * Thông tin về các sinh viên thực hiện: họ tên, mã số sinh viên * Giáo viên hướng dẫn: chức danh, họ tên * Học kỳ, niên khóa thực hiện |  |
| **Bố cục** *(tối đa 0,25 điểm)* |  |
| * Nhận xét, đánh giá của giảng viên chấm * Mục lục: cấu trúc chương, mục và tiểu mục * Phụ lục (nếu có) * Tài liệu tham khảo |  |
| **II. NỘI DUNG** *(Tối đa 3,5 điểm)* |  |
| **Tổng quan** *(tối đa 0,5 điểm)* |  |
| * Mô tả bài toán, mục tiêu cần đạt được *(0,25 điểm)* * Hướng giải quyết và kế hoạch thực hiện *(0,25 điểm)* |  |
| **Lý thuyết** *(tối đa 0,5 điểm)* |  |
| * Các khái niệm sử dụng trong đề tài *(0,25 điểm)* * Kết quả vận dụng lý thuyết vào đề tài *(0,25 điểm)* |  |
| **Ứng dụng** *(tối đa 2,0 điểm)* |  |
| * Phân tích yêu cầu bài toán, xây dựng các cấu trúc dữ liệu cần thiết *(tối đa 0,5 điểm)* * Giải thuật (Lưu đồ-Ngôn ngữ giả) *(1,0 điểm)* * Giới thiệu chương trình *(0,5 điểm)* |  |
| **Kết luận** *(tối đa 0,5 điểm)* |  |
| * Nhận xét kết quả đạt được * Hạn chế * Hướng phát triển |  |
| **III. CHƯƠNG TRÌNH DEMO** *(Tối đa 5,0 điểm)* |  |
| **Giao diện thân thiện với người dùng** *(1,0 điểm)* |  |
| **Hướng dẫn sử dụng** *(0.5 điểm)* |  |
| **Kết quả thực hiện đúng với kết quả của phần ứng dụng** *(3,5 điểm)* |  |

Bạc Liêu, *ngày … tháng … năm ……*

**GIÁNG VIÊN CHẤM**

**LỜI MỞ ĐẦU**

Trong lĩnh vực khoa học máy tính, việc tổ chức và quản lý dữ liệu một cách hiệu quả là yếu tố cốt lõi. Cây tìm kiếm nhị phân, với cấu trúc đặc biệt và các thuật toán đi kèm, đã trở thành một công cụ không thể thiếu trong việc thực hiện các phép toán tìm kiếm, chèn, xóa trên tập hợp dữ liệu lớn.Từ việc quản lý danh bạ điện thoại, tìm kiếm sản phẩm trên các trang thương mại điện tử đến xây dựng các hệ thống cơ sở dữ liệu lớn, cây tìm kiếm nhị phân đóng vai trò quan trọng trong việc tối ưu hóa hiệu suất và cải thiện trải nghiệm người dùng. Mặc dù cây tìm kiếm nhị phân đã được nghiên cứu và ứng dụng rộng rãi, vẫn còn nhiều vấn đề mở cần được giải quyết, đặc biệt là trong việc cải thiện hiệu suất và khả năng mở rộng của cấu trúc dữ liệu này.

Vấn đề đặt ra: Đề tài này sẽ đi sâu vào việc phân tích, đánh giá và ứng dụng cây tìm kiếm nhị phân, nhằm làm rõ vai trò quan trọng của cấu trúc dữ liệu này trong việc giải quyết các bài toán thực tế.

**MỤC LỤC**

[CHƯƠNG I: TỔNG QUAN BÀI TOÁN 1](#_Toc174885199)

[I. CƠ SỞ LÝ THUYẾT 1](#_Toc174885200)

[1. Cấu trúc dữ liệu cây nhị phân 1](#_Toc174885201)

[2. Các loại cây tìm kiếm nhị phân 1](#_Toc174885202)

[3. Các thuật toán cơ bản trên cây nhị phân 1](#_Toc174885203)

[4. Các bước vẽ cây tìm kiếm nhị phân 1](#_Toc174885204)

[5. Giới thiệu thuật toán 2](#_Toc174885205)

[II. MỤC TIÊU CỦA ĐỀ TÀI 3](#_Toc174885206)

[III. HƯỚNG GIẢI QUYẾT 3](#_Toc174885207)

[CHƯƠNG II:NỘI DUNG ĐỀ TÀI CÂY TÌM KIẾM NHỊ PHÂN 4](#_Toc174885208)

[I. MÔ TẢ THUẬT TOÁN 4](#_Toc174885209)

[II.GIẢI QUYẾT THUẬT TOÁN TOÁN. 5](#_Toc174885210)

[1.Phương pháp. 5](#_Toc174885211)

[2. Phân tích thiết kế thuật toán. 7](#_Toc174885212)

[3.Công cụ lập trình 9](#_Toc174885213)

[III.CHƯƠNG TRÌNH CÂY TÌM KIẾM NHỊ PHÂN 9](#_Toc174885214)

[1. Chương trình 9](#_Toc174885215)

[2. Yêu cầu khi chạy 13](#_Toc174885216)

[CHƯƠNG III:DEMO CỦA CÂY TÌM KIẾM NHỊ PHÂN 14](#_Toc174885217)

[I. Dữ liệu đầu vào 14](#_Toc174885218)

[II.Kết quả 14](#_Toc174885219)

[1. Hiển thị được cây tìm kiếm nhị phân với giao diện đồ họa 14](#_Toc174885220)

[2.Kết quả bài toán 15](#_Toc174885221)

[CHƯƠNG IV:KẾT LUẬN 16](#_Toc174885222)

[I. Ưu điểm 16](#_Toc174885223)

[II. Khuyết điểm 16](#_Toc174885224)

[III. Hướng phát triển 16](#_Toc174885225)

[TÀI LIỆU THAM KHẢO 17](#_Toc174885226)

CHƯƠNG I: TỔNG QUAN BÀI TOÁN

I. CƠ SỞ LÝ THUYẾT

1. Cấu trúc dữ liệu cây tìm kiếm nhị phân

1.1.1.Định nghĩa

Cây tìm kiếm nhị phân là một cấu trúc dữ liệu cây trong đó mỗi nút có tối đa hai nút con: nút con trái và nút con phải. Các nút trong BST được sắp xếp theo một thứ tự nhất định:

* Con trái (Left Child): Chứa các giá trị nhỏ hơn nút cha.
* Con phải (Right Child): Chứa các giá trị lớn hơn nút cha

1.1.2. Thành phần cơ bản

* Nút (Node): Đơn vị cơ bản của cây, chứa dữ liệu và hai liên kết đến con trái và con phải.
* Nút gốc (Root): Nút đầu tiên trong cây.
* Nút lá (Leaf): Nút không có con trái lẫn con phải.
* Cây con (Subtree): Một phần của cây nhị phân, cũng là một cây nhị phân.

1.1.3. Đặc điểm

* Một cây nhị phân có tính chất đệ quy: Mỗi cây con của nó cũng là một cây nhị phân.
* Trong cây nhị phân tìm kiếm, các giá trị được sắp xếp sao cho dễ dàng tìm kiếm, thêm, và xóa.

2. Các loại cây tìm kiếm nhị phân

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

* Định nghĩa:
* Một cây nhị phân trong đó:
* Tất cả các giá trị trong cây con trái nhỏ hơn giá trị tại nút cha.
* Tất cả các giá trị trong cây con phải lớn hơn giá trị tại nút cha.
* Ưu điểm: Dễ dàng tìm kiếm, thêm, xóa với độ phức tạp trung bình O(log n).
* Nhược điểm: Có thể trở thành cây lệch nếu dữ liệu không được chèn cân đối.

2.2.2.Cây nhị phân cân bằng

* Các loại cây:
* Cây AVL: Duy trì sự cân bằng về chiều cao giữa các cây con bằng cách xoay cây khi cần.
* Cây đỏ-đen (Red-Black Tree): Cân bằng cây bằng cách áp dụng quy tắc màu sắc.
* Ưu điểm: Đảm bảo thời gian xử lý luôn là O(log n).

2.2.3.Cây nhị phân đầy đủ và hoàn chỉnh

* Cây đầy đủ (Full Binary Tree): Tất cả các nút có 0 hoặc 2 con.
* Cây hoàn chỉnh (Complete Binary Tree): Tất cả các tầng đều đầy đủ, ngoại trừ tầng cuối, các nút lấp từ trái sang phải.

3. Các thuật toán cơ bản trên cây nhị phân

3.3.1.Thuật toán tìm kiếm

* Ý tưởng:
* Bắt đầu từ nút gốc, so sánh giá trị cần tìm:
* Nếu bằng giá trị tại nút hiện tại → Trả về kết quả.
* Nếu nhỏ hơn → Tìm trong cây con trái.
* Nếu lớn hơn → Tìm trong cây con phải.
* Độ phức tạp:
* Trung bình: O(log n).
* Trường hợp xấu nhất (cây lệch): O(n).

3.3.2.Thuật toán chèn

* Ý tưởng:
* So sánh giá trị cần chèn với giá trị tại nút hiện tại:
* Nếu nhỏ hơn → Chèn vào cây con trái.
* Nếu lớn hơn → Chèn vào cây con phải.
* Nếu gặp nút trống → Thêm nút mới.
* Độ phức tạp: O(log n).

3.3.3.Thuật toán xóa

* Ý tưởng:
* Tìm nút cần xóa.
* Xử lý các trường hợp:
* Nút không có con → Xóa trực tiếp.
* Nút có một con → Kết nối nút cha với nút con.
* Nút có hai con → Thay thế giá trị bằng giá trị nhỏ nhất của cây con phải (hoặc lớn nhất của cây con trái) rồi xóa nút này.
* Độ phức tạp: O(log n).

3.3.4.Thuật toán duyệt cây

* Duyệt theo thứ tự (In-order): Trái → Gốc → Phải (In-order traversal).
* Kết quả: Các giá trị được duyệt theo thứ tự tăng dần.
* Duyệt trước (Pre-order): Gốc → Trái → Phải.
* Duyệt sau (Post-order): Trái → Phải → Gốc.
* Duyệt theo tầng (Level-order): Duyệt từng tầng từ trái sang phải.

3.3.5.Thuật toán tính chiều cao

* Ý tưởng: Tính chiều cao của cây bằng cách đệ quy:
* Chiều cao cây = Max(chiều cao cây con trái, cây con phải) + 1.
* Độ phức tạp: O(n).

4. Các bước vẽ cây tìm kiếm nhị phân

4.4.1.Xây dựng cấu trúc cây

* Cài đặt cấu trúc dữ liệu cây nhị phân và các thuật toán cơ bản (tìm kiếm, chèn, xóa, duyệt cây).
* Sử dụng các lớp (class) để biểu diễn:
* Node: Biểu diễn một nút trong cây.
* BinarySearchTree: Chứa logic quản lý cây.

4.4.2.Tính toán vị trí các nút

* Xác định tọa độ (x, y) của từng nút dựa trên:
* Độ sâu của cây (tầng y).
* Khoảng cách giữa các nút (trục x).
* Duyệt cây để xác định vị trí của từng nút.

4.4.3.Vẽ đồ họa

* Sử dụng thư viện đồ họa:
* Python: Tkinter, matplotlib, hoặc pygame.
* Java: Swing hoặc JavaFX.
* Web: HTML Canvas hoặc D3.js.
* Các thành phần cần vẽ:
* Nút: Biểu diễn bằng hình tròn.
* Cạnh: Biểu diễn bằng các đường thẳng nối giữa nút cha và nút con.

4.4.4.Cập nhật cây

* Khi có thay đổi (thêm, xóa nút), vẽ lại toàn bộ cây để hiển thị trạng thái mới.

5. Giới thiệu thuật toán

5.5.1.Thuật toán xác định vị trí nút

* Ý tưởng:
* Với mỗi nút:
* Xác định tọa độ x dựa trên khoảng cách giữa các nút cùng tầng.
* Xác định tọa độ y dựa trên độ sâu của nút.
* Duyệt cây bằng đệ quy để xác định vị trí tất cả các nút.

5.5.2.Thuật toán vẽ cây

* Ý tưởng:
* Vẽ các nút:
* Duyệt cây và vẽ hình tròn tại vị trí xác định.
* Vẽ các cạnh:
* Kết nối vị trí nút cha với các nút con bằng đường thẳng.

5.5.3.Thuật toán duyệt cây để cập nhật

* Kết hợp duyệt cây (In-order hoặc Level-order) để xác định thứ tự vẽ lại các nút khi cây thay đổi.

Dưới đây là một ví dụ về bài toán xây dựng cây tìm kiếm nhị phân

Bài toán cụ thể: Xây dựng cây tìm kiếm nhị phân để quản lý danh sách sinh viên:

Đề bài

Một lớp học cần quản lý danh sách sinh viên dựa trên mã số sinh viên (ID). Yêu cầu xây dựng chương trình sử dụng cây tìm kiếm nhị phân (BST) để hỗ trợ các chức năng sau:

1. Thêm sinh viên mới vào cây, trong đó mỗi sinh viên có thông tin:

• ID (số nguyên, duy nhất): Là khóa (key) trong cây.

• Họ tên (name).

• Điểm trung bình (GPA).

2. Tìm kiếm thông tin sinh viên dựa trên ID. Nếu tìm thấy, in ra thông tin của sinh viên đó.

3. Xóa một sinh viên khỏi cây dựa trên ID.

4. Hiển thị danh sách sinh viên theo thứ tự tăng dần của ID.

5. In cây dưới dạng đồ họa để trực quan hóa cấu trúc cây.

Đầu vào và đầu ra

• Đầu vào:

• Dữ liệu ban đầu: Một danh sách sinh viên với các thông tin (ID, Name, GPA).

• Các thao tác yêu cầu từ người dùng:

• Thêm sinh viên.

• Tìm kiếm sinh viên.

• Xóa sinh viên.

• Hiển thị danh sách sinh viên.

• Đầu ra:

• Thông tin của sinh viên khi tìm kiếm.

• Danh sách sinh viên theo thứ tự tăng dần của ID.

• Trạng thái cây sau mỗi thao tác, hiển thị dưới dạng cấu trúc đồ họa.

Ví dụ minh họa

Dữ liệu ban đầu

Danh sách sinh viên:



Các thao tác yêu cầu

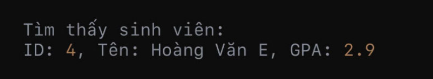
1. Thêm sinh viên có ID = 6, tên = “Vũ Văn F”, GPA = 3.6.

• Cây sau khi thêm:



2. Tìm kiếm sinh viên với ID = 4.

• Kết quả:



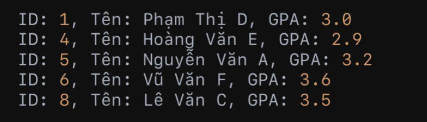
3. Xóa sinh viên với ID = 3.

• Cây sau khi xóa:



4. Hiển thị danh sách sinh viên:

• Kết quả:



5. Hiển thị cây dưới dạng đồ họa:

• Cây được vẽ với mỗi nút hiển thị ID và GPA của sinh viên.

Yêu cầu bài toán

1. Cài đặt cấu trúc cây tìm kiếm nhị phân:

• Mỗi nút trong cây chứa thông tin: ID, Name, GPA, con trái, con phải.

2. Viết các thuật toán:

• Thêm: Thêm một nút vào cây dựa trên ID.

• Tìm kiếm: Duyệt cây để tìm sinh viên có ID yêu cầu.

• Xóa: Xóa nút trong cây và đảm bảo tính chất BST.

• Duyệt cây: Theo thứ tự tăng dần (In-order Traversal).

• Vẽ cây: Hiển thị cây dưới dạng sơ đồ.

3. Đảm bảo chương trình chạy chính xác và hiệu quả:

• Độ phức tạp cho các thao tác: O(h), với h là chiều cao của cây.

II. MỤC TIÊU CỦA ĐỀ TÀI

Nắm vững cơ sở lý thuyết về cấu trúc dữ liệu. Các kỹ thuật thiết kế giải thuật.

1. Quản lý dữ liệu hiệu quả:

• Lưu trữ thông tin sinh viên (hoặc các đối tượng khác) một cách khoa học dựa trên mã số sinh viên (ID).

• Đảm bảo khả năng tìm kiếm, thêm, xóa và duyệt dữ liệu nhanh chóng.

2. Tối ưu hóa thao tác truy vấn dữ liệu:

• Sử dụng cấu trúc cây tìm kiếm nhị phân để giảm độ phức tạp các thao tác xuống mức tối ưu ￼, với ￼ là chiều cao của cây.

3. Cung cấp giao diện trực quan:

• Hiển thị cấu trúc cây tìm kiếm nhị phân dưới dạng đồ họa, giúp dễ dàng quan sát mối quan hệ giữa các nút trong cây.

4. Áp dụng vào bài toán thực tế:

• Giải quyết bài toán quản lý danh sách sinh viên theo mã ID, giúp dễ dàng truy xuất thông tin hoặc tổ chức dữ liệu theo thứ tự tăng dần.

5. Xây dựng nền tảng cho các bài toán phức tạp hơn:

• Tạo cơ sở cho việc phát triển các thuật toán nâng cao, như cân bằng cây hoặc ứng dụng cây trong hệ thống quản lý lớn hơn.

**III. HƯỚNG GIẢI QUYẾT**

***Về lý thuyết****:* Tìm hiểu các kỹ thuật thiết kế giải thuật…và các kiến thức về lập trình trên ngôn ngữ sử dụng để giải quyết yêu cầu đề tài.

***Về chương trình*:** Sử dụng ngôn ngữ lập trình chính là **C++** hoặc Python để viết chương trình và môi trường cài đặt DevC hoặc có thể sử dụng Visual Studio , cài đặt các thuật toán thực hiện các yêu cầu của đề tài, nghiên cứu và cài đặt các thủ tục hàm đồ họa để hỗ trợ giao diện người dùng sử dụng phần mềm.

CHƯƠNG II:LÝ THUYẾT

I.CÁC KHÁI NIỆM

Khi xây dựng và làm việc với cây tìm kiếm nhị phân (BST), một số khái niệm cơ bản cần hiểu rõ để triển khai và áp dụng thuật toán hiệu quả. Dưới đây là các khái niệm quan trọng:

1. Cây nhị phân (Binary Tree)

Cây nhị phân là một cấu trúc dữ liệu trong đó mỗi nút có tối đa hai nút con: nút con trái và nút con phải. Mỗi nút trong cây chứa một giá trị và hai con trỏ (hoặc tham chiếu) đến các nút con của nó.

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

Cây tìm kiếm nhị phân là một dạng đặc biệt của cây nhị phân, trong đó:

• Tất cả các giá trị ở cây con bên trái của một nút đều nhỏ hơn giá trị của nút đó.

• Tất cả các giá trị ở cây con bên phải của một nút đều lớn hơn giá trị của nút đó.

3. Nút (Node)

Một nút trong cây tìm kiếm nhị phân bao gồm ba thành phần:

• Dữ liệu (data): Giá trị lưu trữ tại nút.

• Con trái (left): Con trỏ hoặc tham chiếu đến nút con bên trái.

• Con phải (right): Con trỏ hoặc tham chiếu đến nút con bên phải.

4. Gốc (Root)

Nút gốc là nút đầu tiên của cây, từ đó các thao tác như tìm kiếm, chèn, xóa bắt đầu. Cây tìm kiếm nhị phân chỉ có một nút gốc duy nhất.

5. Nút lá (Leaf Node)

Nút lá là những nút không có con. Chúng là các nút nằm ở cuối cây, không có nút con bên trái hoặc bên phải.

6. Nút nội (Internal Node)

Là các nút trong cây có ít nhất một nút con (hoặc con trái, hoặc con phải, hoặc cả hai).

7. Chiều cao (Height)

Chiều cao của một nút trong cây là số lượng các cạnh trên đường đi dài nhất từ nút đó đến một nút lá. Chiều cao của cây là chiều cao của nút gốc.

8. Độ sâu (Depth)

Độ sâu của một nút là số lượng các cạnh từ nút gốc đến nút đó. Độ sâu của nút gốc là 0.

9. Cây con trái và cây con phải

• Cây con trái: Là phần cây được tạo thành từ các nút có giá trị nhỏ hơn giá trị của nút gốc.

• Cây con phải: Là phần cây được tạo thành từ các nút có giá trị lớn hơn giá trị của nút gốc.

10. Duyệt cây (Tree Traversal)

Các phương pháp duyệt cây giúp duyệt qua tất cả các nút trong cây theo một thứ tự nhất định:

• Duyệt theo tiền thứ tự (Pre-order): Duyệt nút gốc trước, sau đó cây con trái và cây con phải.

• Duyệt theo thứ tự trung gian (In-order): Duyệt cây con trái, sau đó nút gốc, rồi đến cây con phải. Phương pháp này cho phép truy cập các phần tử của cây theo thứ tự tăng dần.

• Duyệt theo hậu thứ tự (Post-order): Duyệt cây con trái, cây con phải, sau đó đến nút gốc.

11. Chèn (Insert)

Thao tác chèn là quá trình thêm một giá trị mới vào cây. Khi chèn, ta so sánh giá trị cần chèn với các giá trị của cây, và dựa vào đó để quyết định chèn vào cây con trái hay phải của một nút.

12. Tìm kiếm (Search)

Thao tác tìm kiếm là quá trình tìm kiếm một giá trị trong cây. Ta bắt đầu từ nút gốc và so sánh giá trị cần tìm với giá trị của nút hiện tại. Nếu giá trị cần tìm nhỏ hơn giá trị của nút, tìm kiếm tiếp trong cây con trái; nếu lớn hơn, tìm kiếm trong cây con phải.

13. Xóa (Delete)

Thao tác xóa là quá trình loại bỏ một nút khỏi cây. Có ba trường hợp:

• Nút lá (không có con): Xóa nút này trực tiếp.

• Nút có một con: Thay thế nút cần xóa bằng nút con của nó.

• Nút có hai con: Thay thế nút cần xóa bằng nút kế tiếp hoặc nút kế trước (theo thứ tự tăng dần hoặc giảm dần) và xóa nút kế tiếp hoặc kế trước.

14. Cây con (Subtree)

Một cây con của một nút là cây con bắt đầu từ nút đó. Ví dụ, cây con trái là cây con của nút gốc nằm bên trái nút đó.

15. Cân bằng cây (Balanced Tree)

Một cây tìm kiếm nhị phân được gọi là cân bằng nếu chiều cao của cây con trái và cây con phải của bất kỳ nút nào không chênh lệch quá một đơn vị. Cây AVL và cây Red-Black là các ví dụ về cây tìm kiếm nhị phân tự cân bằng.

16. Độ phức tạp (Time Complexity)

Độ phức tạp của các thao tác trên cây tìm kiếm nhị phân phụ thuộc vào chiều cao của cây. Trong trường hợp cây cân bằng, các thao tác tìm kiếm, chèn và xóa có độ phức tạp là O(log n). Tuy nhiên, trong trường hợp cây không cân bằng, độ phức tạp có thể lên đến O(n).

**II.KẾT QUẢ VẬN DỤNG LÝ THUYẾT VÀO ĐỀ TÀI**

Khi áp dụng lý thuyết về cây tìm kiếm nhị phân (BST) vào việc xây dựng một chương trình thực tế, kết quả đạt được sẽ phản ánh việc tổ chức và quản lý dữ liệu theo một cấu trúc tối ưu cho các phép toán như tìm kiếm, chèn, xóa, và duyệt cây. Dưới đây là những kết quả cụ thể khi vận dụng lý thuyết về cây BST vào đề tài:

1. Tổ chức dữ liệu hiệu quả

Áp dụng lý thuyết về cây tìm kiếm nhị phân giúp tổ chức dữ liệu theo một cấu trúc có trật tự. Mỗi nút trong cây chỉ có tối đa hai nút con, đồng thời đảm bảo rằng các giá trị ở cây con trái luôn nhỏ hơn nút gốc và các giá trị ở cây con phải luôn lớn hơn nút gốc. Điều này giúp:

• Tìm kiếm dữ liệu nhanh chóng: Dữ liệu được sắp xếp theo thứ tự, giúp việc tìm kiếm một giá trị có thể thực hiện trong thời gian O(log n) đối với cây cân bằng.

• Chèn và xóa dữ liệu dễ dàng: Khi thêm hoặc loại bỏ dữ liệu, các thao tác này có thể thực hiện mà không làm mất trật tự của cây, giúp duy trì cấu trúc cây tìm kiếm nhị phân.

2. Cải thiện hiệu suất tìm kiếm

Khi áp dụng lý thuyết BST, việc tìm kiếm một phần tử trong tập hợp được thực hiện bằng cách so sánh giá trị cần tìm với giá trị của nút gốc và di chuyển vào cây con trái hoặc cây con phải tùy thuộc vào kết quả so sánh. Điều này giảm thiểu số lần so sánh cần thiết. So với các phương pháp tìm kiếm tuyến tính (O(n)), BST cho phép tìm kiếm nhanh hơn với độ phức tạp O(log n) trong trường hợp cây cân bằng.

3. Duy trì trật tự và tính nhất quán của dữ liệu

Lý thuyết về cây tìm kiếm nhị phân đảm bảo rằng:

• Các giá trị trong cây luôn được sắp xếp: Việc duy trì thứ tự các phần tử trong cây giúp dễ dàng truy xuất và làm việc với các dữ liệu theo một thứ tự cụ thể (tăng dần hoặc giảm dần).

• Duyệt cây theo thứ tự: Các phép duyệt cây như Duyệt theo thứ tự trung gian (In-order) cho phép lấy các giá trị trong cây theo thứ tự tăng dần. Điều này có ứng dụng rộng rãi trong các hệ thống cơ sở dữ liệu và tìm kiếm.

4. Xử lý dữ liệu linh hoạt với các thao tác cơ bản

Ứng dụng lý thuyết cây BST cho phép thực hiện các thao tác cơ bản như:

• Chèn (Insert): Khi cần thêm phần tử mới vào cây, ta thực hiện việc tìm vị trí thích hợp cho giá trị đó trong cây bằng cách so sánh giá trị cần thêm với các nút trong cây, đồng thời duy trì tính chất BST.

• Tìm kiếm (Search): Các phép toán tìm kiếm có thể thực hiện nhanh chóng và chính xác thông qua các phép so sánh, giảm thiểu số lần truy cập cần thiết.

• Xóa (Delete): Việc xóa một nút cũng dễ dàng thực hiện theo lý thuyết, đặc biệt là với ba trường hợp khi xóa một nút có con hoặc không có con.

5. Hiển thị cây dưới dạng đồ họa

Lý thuyết về cây BST cho phép mô hình hóa cây dưới dạng đồ họa, giúp người dùng dễ dàng quan sát và hiểu cách cây phát triển. Các thao tác như duyệt cây hoặc xóa nút có thể được trực quan hóa giúp đánh giá hiệu quả của các thao tác. • Ứng dụng: Cây BST có thể được sử dụng trong các hệ thống quản lý dữ liệu (như cơ sở dữ liệu, tìm kiếm tài liệu) hoặc trong các hệ thống yêu cầu tính năng tìm kiếm nhanh như tìm kiếm tài liệu, tìm kiếm trong trò chơi, hoặc ứng dụng các thuật toán học máy.

6. Đánh giá và tối ưu hiệu suất

Việc áp dụng lý thuyết cây tìm kiếm nhị phân giúp tối ưu hóa hiệu suất của hệ thống:

• Tối ưu hóa không gian và thời gian: Các thao tác tìm kiếm, chèn, và xóa trên BST, nếu cây được duy trì cân bằng, sẽ có hiệu suất tốt với độ phức tạp là O(log n) cho các thao tác tìm kiếm, chèn và xóa.

• Quản lý bộ nhớ hiệu quả: Việc tổ chức dữ liệu theo cấu trúc cây giúp quản lý bộ nhớ hiệu quả hơn, tránh việc phải duy trì các mảng hoặc danh sách không cần thiết.

7. Kết luận

Vận dụng lý thuyết về cây tìm kiếm nhị phân trong việc xây dựng chương trình không chỉ giúp giải quyết bài toán tìm kiếm và sắp xếp dữ liệu hiệu quả mà còn tạo ra một cấu trúc dữ liệu linh hoạt, có thể mở rộng và tối ưu cho các bài toán thực tế. Với các thuật toán tìm kiếm, chèn và xóa hiệu quả, BST là một trong những công cụ quan trọng trong khoa học máy tính và phát triển phần mềm. Các kết quả này thể hiện được tính ứng dụng cao của cây tìm kiếm nhị phân trong việc quản lý và xử lý dữ liệu, đồng thời cung cấp nền tảng cho việc triển khai các thuật toán tối ưu hóa và các cấu trúc dữ liệu nâng cao như cây AVL, cây Red-Black, hoặc các thuật toán tìm kiếm trong cơ sở dữ liệu.

**CHƯƠNG III:ỨNG DỤNG**

1. **PHÂN TÍCH YÊU CẦU BÀI TOÁN** 
   * 1. **Đặc tả bài toán**

**Nội dung bài toán :** xây dựng 1 cây tìm kiếm nhị phân có giao diện đồ họa

* + 1. **Yêu cầu của đề tài**
  + Nắm vững cơ sở lý thuyết về cấu trúc dữ liệu. Các kỹ thuật thiết kế giải thuật.
  + Chương trình cần có các chức năng sau:tổ chức cấu trúc dữ liệu,cùng các phép toán:
    - Khởi tạo cây
    - Chèn một phần tử vào cây
    - Xóa một phần tử trên cây
    - Tìm một nút trên cây
    - Đếm số nút trên cây
    1. **Phân tích chi tiết yêu cầu**

Input:

• Danh sách các phần tử (n): Một danh sách chứa các phần tử mà người dùng muốn chèn vào cây tìm kiếm nhị phân. Đây là các giá trị số nguyên, có thể là bất kỳ dãy số nào.

• Dữ liệu tìm kiếm: Một giá trị cần tìm trong cây.

• Tên phần tử (Tên Phần Tử): Mỗi phần tử có thể đi kèm với một tên, giúp dễ dàng nhận diện các phần tử khi xuất kết quả.

Output:

• Phương án tìm kiếm: Kết quả của thao tác tìm kiếm trên cây, bao gồm:

• Xác định có tìm thấy phần tử cần tìm hay không.

• Nếu tìm thấy, xuất tên của phần tử cùng với giá trị của nó.

• Nếu không tìm thấy, xuất thông báo không tìm thấy phần tử đó.

* + 1. **Yêu cầu chức năng**

Nhập dữ liệu:

• Nhập từ bàn phím hoặc từ tập tin:

• Dữ liệu bao gồm: danh sách các phần tử, tên của từng phần tử.

• Người dùng có thể nhập dữ liệu thủ công từ bàn phím hoặc tải dữ liệu từ một tập tin chứa các phần tử cần chèn vào cây.

// Ví dụ nhập dữ liệu từ bàn phím

int n; // Số lượng phần tử

cout << "Nhập số lượng phần tử: ";

cin >> n;

vector<int> elements(n);

vector<string> names(n);

// Nhập các phần tử và tên của chúng

for (int i = 0; i < n; i++) {

cout << "Nhập giá trị phần tử thứ " << i + 1 << ": ";

cin >> elements[i];

cout << "Nhập tên phần tử thứ " << i + 1 << ": ";

cin >> names[i];

}

Xử lý và giải thuật:

1. Khởi tạo cây tìm kiếm nhị phân (BST):

• Tạo một cây với gốc là nullptr.

• Chèn từng phần tử vào cây theo thứ tự đã nhập.

* + 1. **Chèn phần tử vào cây**

• Với mỗi phần tử trong danh sách, tiến hành chèn vào cây theo thuật toán cây tìm kiếm nhị phân:

• Nếu phần tử nhỏ hơn nút hiện tại, chèn vào cây con trái.

• Nếu phần tử lớn hơn nút hiện tại, chèn vào cây con phải.

// Hàm chèn phần tử vào cây BST

Node\* insert(Node\* root, int value, string name) {

if (root == nullptr) {

Node\* newNode = new Node{value, name, nullptr, nullptr};

return newNode;

}

if (value < root->data) {

root->left = insert(root->left, value, name);

} else if (value > root->data) {

root->right = insert(root->right, value, name);

}

return root;

}

* + 1. **Tìm kiếm trong cây (search)**

• Xác định xem phần tử cần tìm có tồn tại trong cây không. Nếu có, xuất tên của phần tử đó.

• Nếu không tìm thấy, xuất thông báo không tìm thấy.

// Hàm tìm kiếm trong cây BST

bool search(Node\* root, int value) {/-strong/-heart:>:o:-((:-h if (root == nullptr) return false;

if (root->data == value) {

cout << "Đã tìm thấy phần tử: " << root->name << " với giá trị: " << root->data << endl;

return true;

}

if (value < root->data) {

return search(root->left, value);

}

return search(root->right, value);

}

* + 1. **Duyệt cây để xuất kết quả**

• Duyệt cây theo thứ tự trung gian (In-order) để xuất các phần tử đã chèn vào cây theo thứ tự tăng dần.

// Hàm duyệt cây theo thứ tự trung gian (In-order Traversal)

void inorder(Node\* root) {

if (root == nullptr) return;

inorder(root->left);

cout << root->name << ": " << root->data << endl;

inorder(root->right);

}

Xuất kết quả:

• Phương án tìm kiếm: Xuất tên của phần tử và giá trị khi tìm thấy.

• Cây tìm kiếm: Xuất các phần tử trong cây theo thứ tự tăng dần.

cout << "Cây tìm kiếm nhị phân theo thứ tự tăng dần: " << endl;

inorder(root);

**II.XÂY DỰNG CẤU TRÚC DỮ LIỆU CẦN THIẾT**

**2. Cấu trúc dữ liệu cần thiết**

Để xây dựng cây tìm kiếm nhị phân và triển khai các thao tác trên, ta cần thiết kế các cấu trúc dữ liệu sau:

**2.1. Cấu trúc Nút (Node)**

Mỗi nút trong cây sẽ lưu trữ một giá trị (dữ liệu) và con trỏ đến hai nút con (trái và phải). Cấu trúc nút là thành phần cơ bản của cây BST.

struct Node {

int data; // Dữ liệu của nút

Node\* left; // Con trỏ đến cây con trái

Node\* right; // Con trỏ đến cây con phải

};

• Dữ liệu (data): Lưu trữ giá trị của phần tử trong cây.

• Con trái (left): Lưu trữ con trỏ đến cây con trái của nút.

• Con phải (right): Lưu trữ con trỏ đến cây con phải của nút.

**2.2. Cấu trúc Cây Tìm Kiếm Nhị Phân (BST)**

Cây tìm kiếm nhị phân sẽ có một con trỏ đến gốc của cây. Mỗi cây sẽ chứa con trỏ này để dễ dàng thực hiện các thao tác trên cây.

struct BST {

Node\* root; // Con trỏ đến gốc của cây

};

**3. Các chức năng cần có trong chương trình**

Chương trình cần có các chức năng cơ bản sau để tổ chức cấu trúc dữ liệu và thực hiện các phép toán trên cây tìm kiếm nhị phân:

**3.1. Khởi tạo cây (Initialize Tree)**

Khi khởi tạo cây, cây BST bắt đầu với con trỏ gốc là nullptr (rỗng).

Mỗi khi tạo một cây mới, cần phải khởi tạo một đối tượng cây với con trỏ gốc trống.

BST\* createTree() {

BST\* tree = new BST;

tree->root = nullptr; // Cây bắt đầu với gốc là null

return tree;

}

**3.2. Chèn một phần tử vào cây (Insert a Node)**

Khi chèn một phần tử vào cây, ta cần phải tìm đúng vị trí để chèn sao cho cây vẫn giữ được tính chất của cây tìm kiếm nhị phân:

• Nếu giá trị phần tử cần chèn nhỏ hơn giá trị của nút hiện tại, ta sẽ tiếp tục tìm kiếm ở cây con trái.

• Nếu giá trị phần tử cần chèn lớn hơn giá trị của nút hiện tại, ta sẽ tiếp tục tìm kiếm ở cây con phải.

• Khi tìm được vị trí thích hợp, ta sẽ chèn nút mới vào đó.

Node\* insert(Node\* root, int value) {

if (root == nullptr) {

// Tạo nút mới và trả về

Node\* newNode = new Node;

newNode->data = value;

newNode->left = newNode->right = nullptr;

return newNode;

}

if (value < root->data) {

// Chèn vào cây con trái

root->left = insert(root->left, value);

} else if (value > root->data) {

// Chèn vào cây con phải

root->right = insert(root->right, value);

}

return root; // Trả về gốc cây đã cập nhật

}

**3.3. Xóa một phần tử khỏi cây (Delete a Node)**

Khi xóa một phần tử khỏi cây, ta cần xử lý ba trường hợp sau:

1. Nút không có con (nút lá): Xóa nút này trực tiếp.

2. Nút có một con: Thay thế nút cần xóa bằng nút con của nó.

3. Nút có hai con: Thay thế nút cần xóa bằng nút kế tiếp (nút nhỏ nhất trong cây con phải hoặc lớn nhất trong cây con trái), rồi xóa nút kế tiếp.

Node\* deleteNode(Node\* root, int value) {

if (root == nullptr) return root; // Nếu cây rỗng

// Tìm nút cần xóa

if (value < root->data) {

root->left = deleteNode(root->left, value);

} else if (value > root->data) {

root->right = deleteNode(root->right, value);

} else {

// Nếu nút cần xóa là nút gốc

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; }

// Nếu nút có hai con, tìm giá trị nhỏ nhất trong cây con phải

Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

Node\* findMin(Node\* root) {

while (root && root->left) {

root = root->left;

}

return root;

}

**3.4. Tìm một nút trong cây (Search a Node)**

Để tìm một nút trong cây, ta thực hiện so sánh giá trị cần tìm với giá trị của các nút trong cây:

• Nếu giá trị cần tìm nhỏ hơn giá trị của nút hiện tại, tìm kiếm tiếp trong cây con trái.

• Nếu giá trị cần tìm lớn hơn giá trị của nút hiện tại, tìm kiếm tiếp trong cây con phải.

• Nếu giá trị cần tìm bằng giá trị của nút hiện tại, trả về true (nút đã tìm thấy).

bool search(Node\* root, int value) {

if (root == nullptr) return false;

if (root->data == value) return true;

if (value < root->data) return search(root->left, value);

return search(root->right, value);

}

**3.5. Đếm số nút trong cây (Count Nodes)**

Để đếm số lượng nút trong cây, ta có thể thực hiện duyệt qua tất cả các nút trong cây và đếm mỗi nút.

int countNodes(Node\* root) {

if (root == nullptr) return 0;

return 1 + countNodes(root->left) + countNodes(root->right);

}

**III.THIẾT KẾ GIẢI THUẬT (LƯU ĐỒ- NGÔN NGỮ GIẢ )**

**1 THIẾT KẾ GIẢI THUẬT (LƯU ĐỒ - NGÔN NGỮ GIẢ )**

* + 1. **Chèn một phần tử vào cây (insert)**

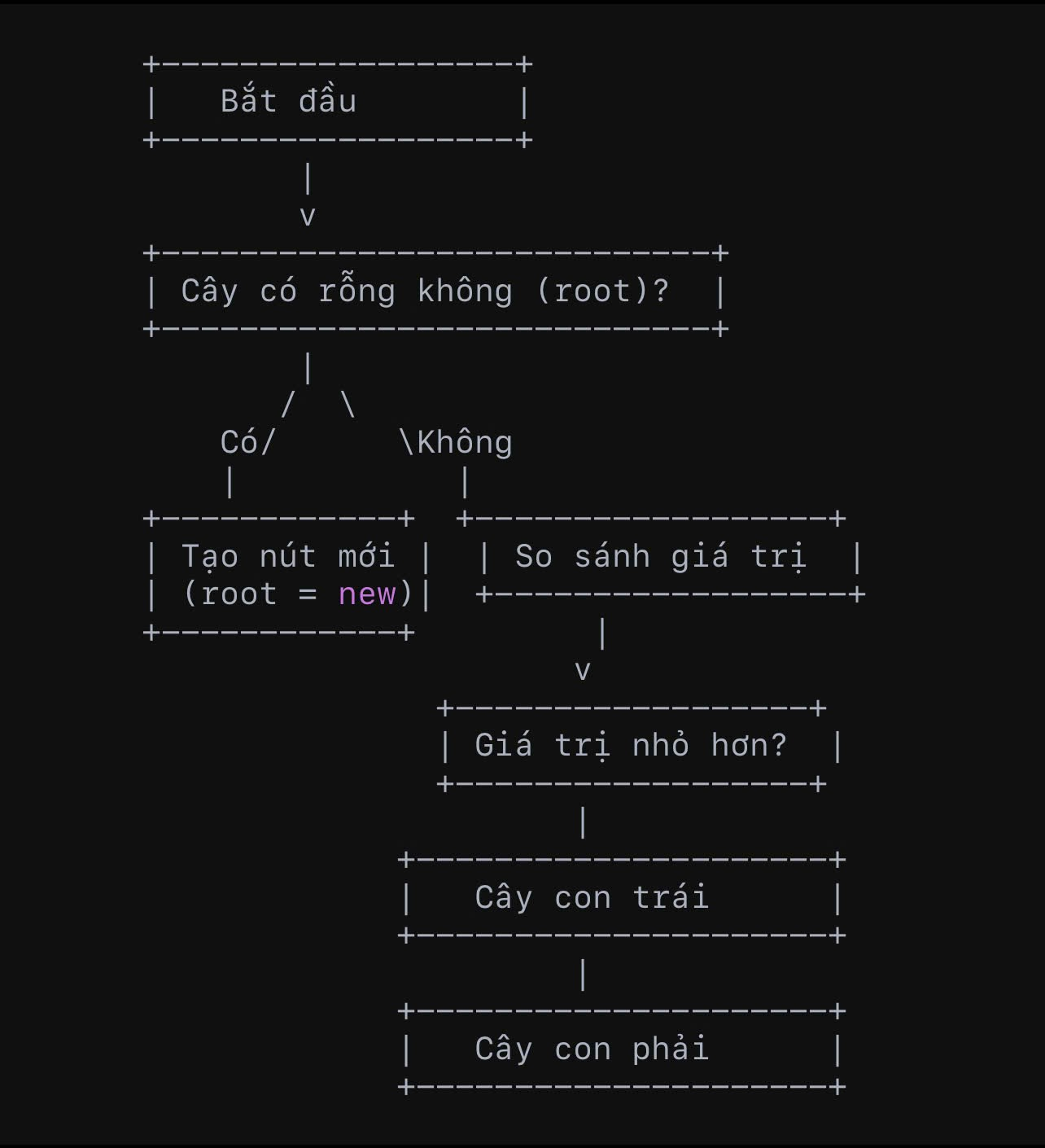
Chèn một phần tử vào cây tìm kiếm nhị phân yêu cầu ta tìm đúng vị trí thích hợp để chèn phần tử sao cho cây vẫn giữ được tính chất BST.

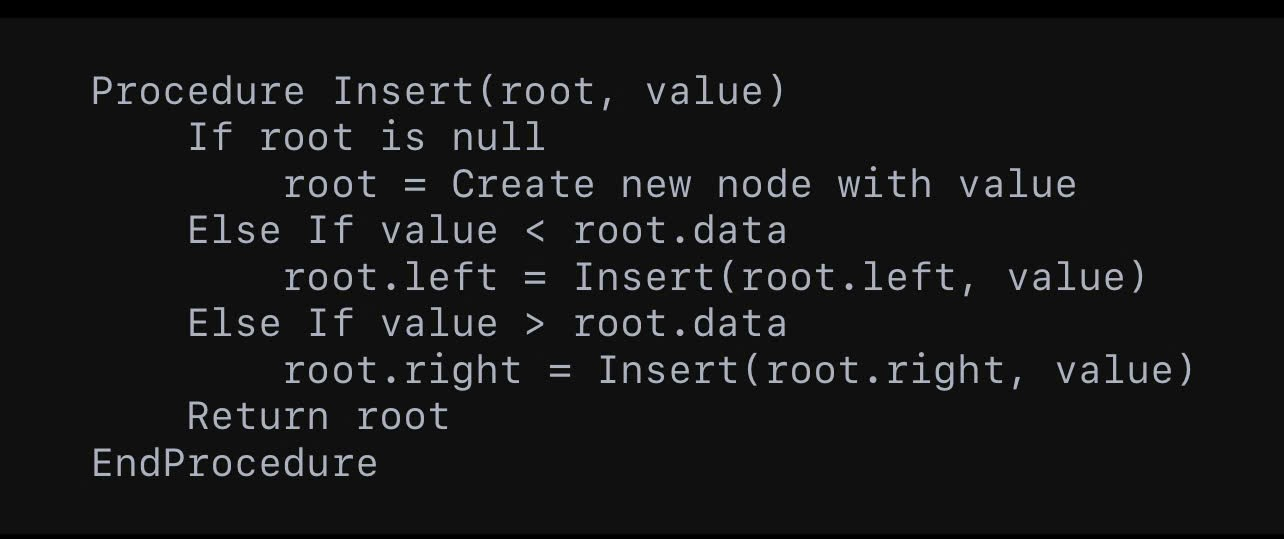
Giải thuật:

1. Nếu cây rỗng, tạo một nút mới và gán nó làm gốc cây.

2. Nếu phần tử cần chèn nhỏ hơn giá trị của nút hiện tại, tiếp tục chèn vào cây con trái.

3. Nếu phần tử cần chèn lớn hơn giá trị của nút hiện tại, tiếp tục chèn vào cây con phải.

\*lưu đồ chèn: 

**\* ngôn ngữ giả **

**1.1.2. Tìm kiếm phần tử trong cây (Search)**

Tìm kiếm một phần tử trong cây yêu cầu so sánh giá trị cần tìm với giá trị của nút hiện tại và đi tiếp vào cây con trái hoặc phải.

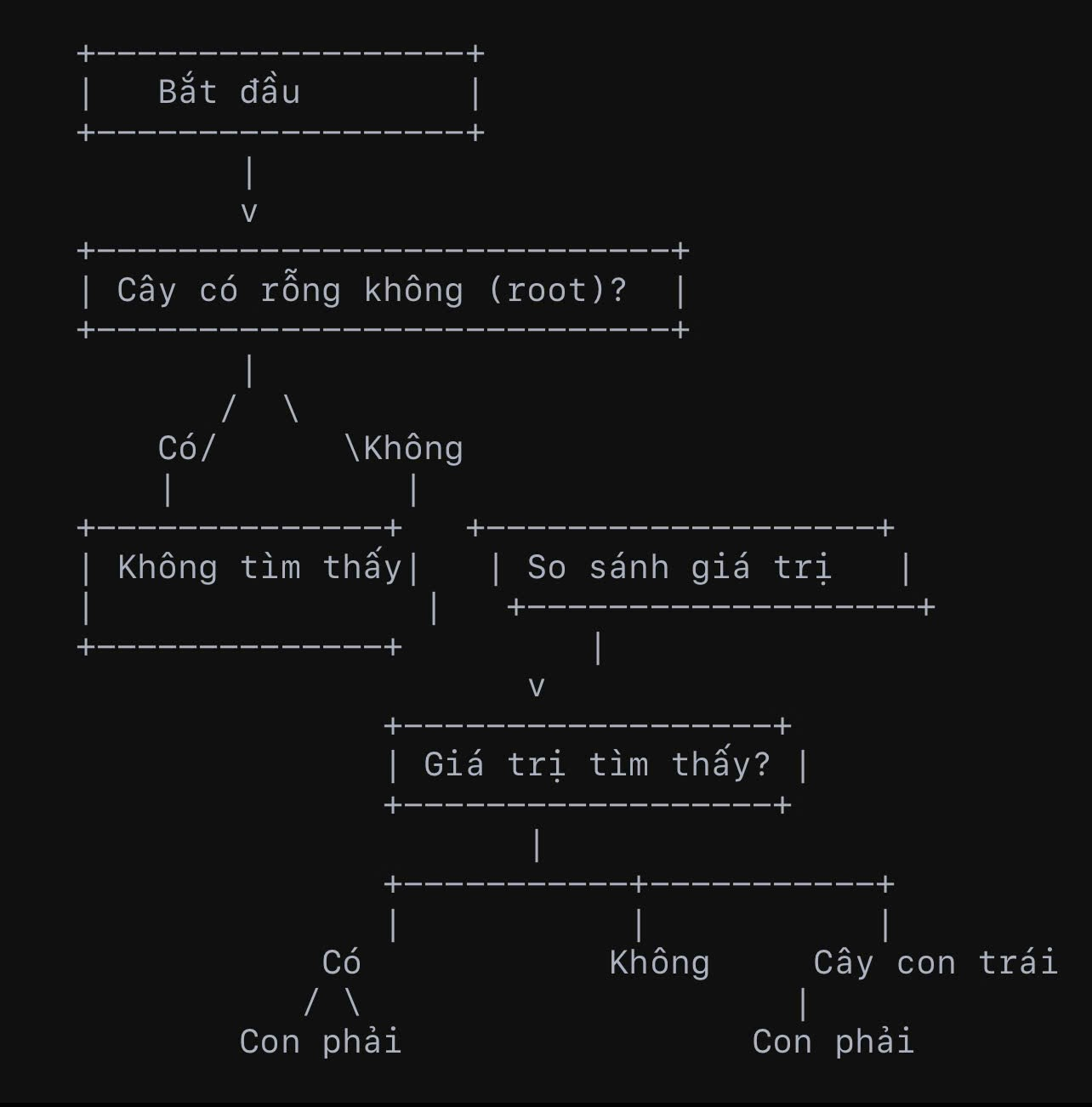
Giải thuật:

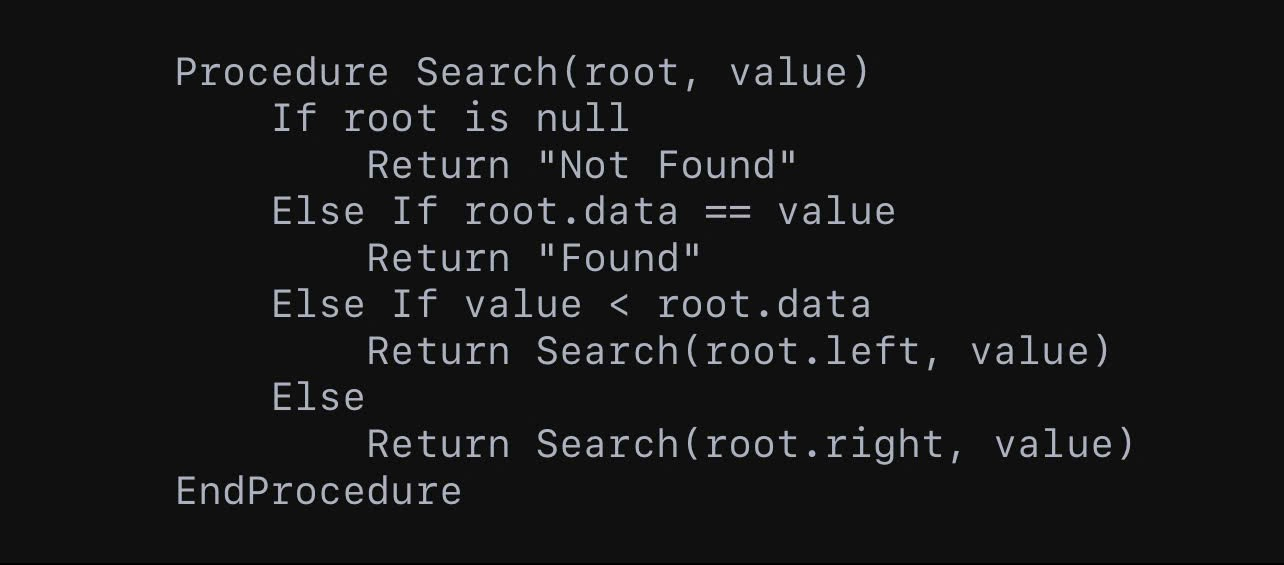
1. Nếu cây rỗng, trả về false (không tìm thấy).

2. Nếu giá trị cần tìm nhỏ hơn giá trị của nút hiện tại, tìm trong cây con trái.

3. Nếu giá trị cần tìm lớn hơn giá trị của nút hiện tại, tìm trong cây con phải.

4. Nếu giá trị cần tìm bằng giá trị của nút hiện tại, trả về true.

\*lưu đồ tìm: 

**\*ngôn ngữ giả **

**1.1.3. Xóa một phần tử khỏi cây (Delete)**

Khi xóa một nút, có ba trường hợp cần xử lý:

1. Nút không có con: Xóa nút này trực tiếp.

2. Nút có một con: Thay thế nút cần xóa bằng nút con của nó.

3. Nút có hai con: Tìm nút kế tiếp (nút nhỏ nhất trong cây con phải hoặc lớn nhất trong cây con trái), sao chép giá trị của nó vào nút cần xóa, sau đó xóa nút kế tiếp.

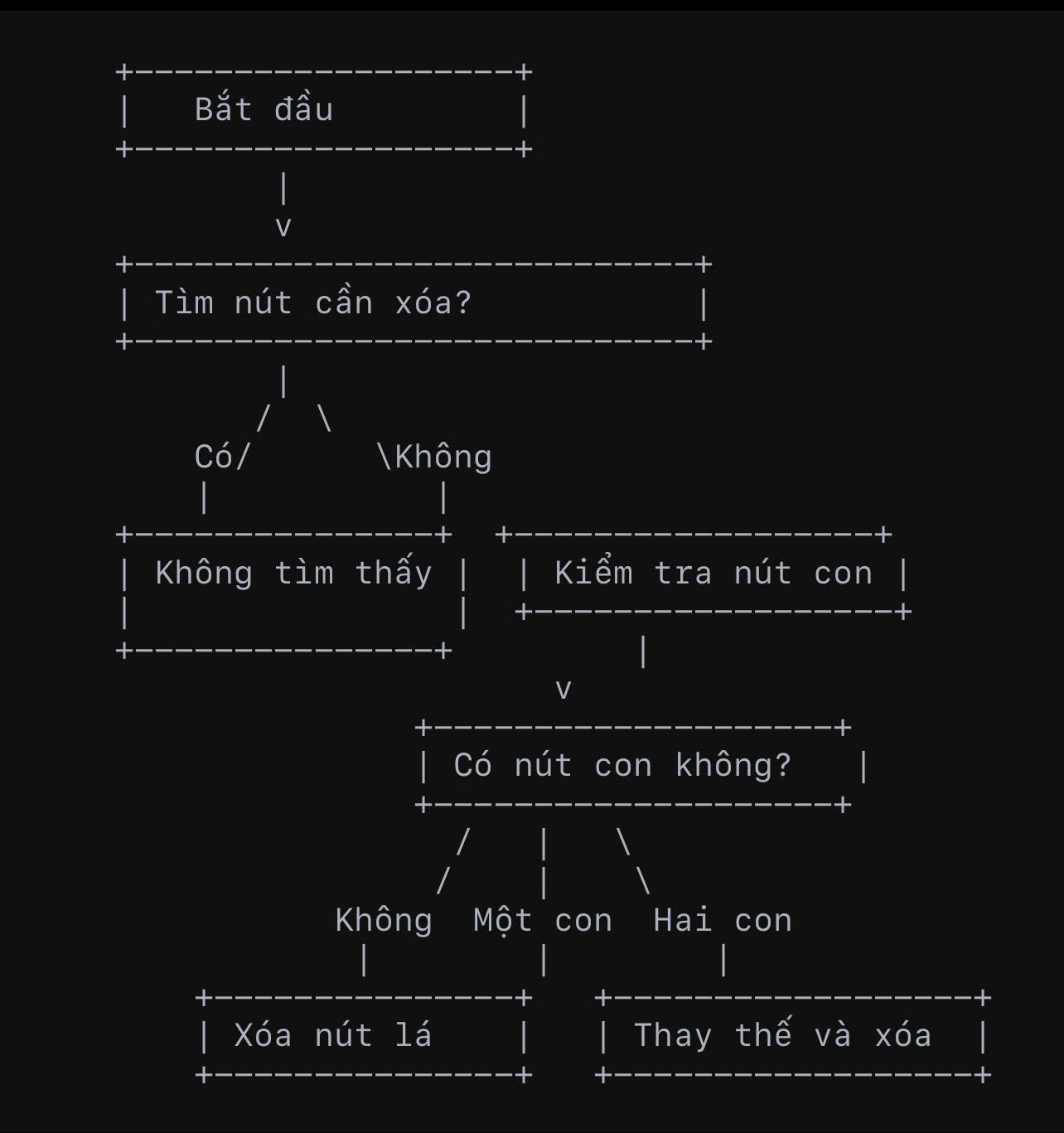
Giải thuật:

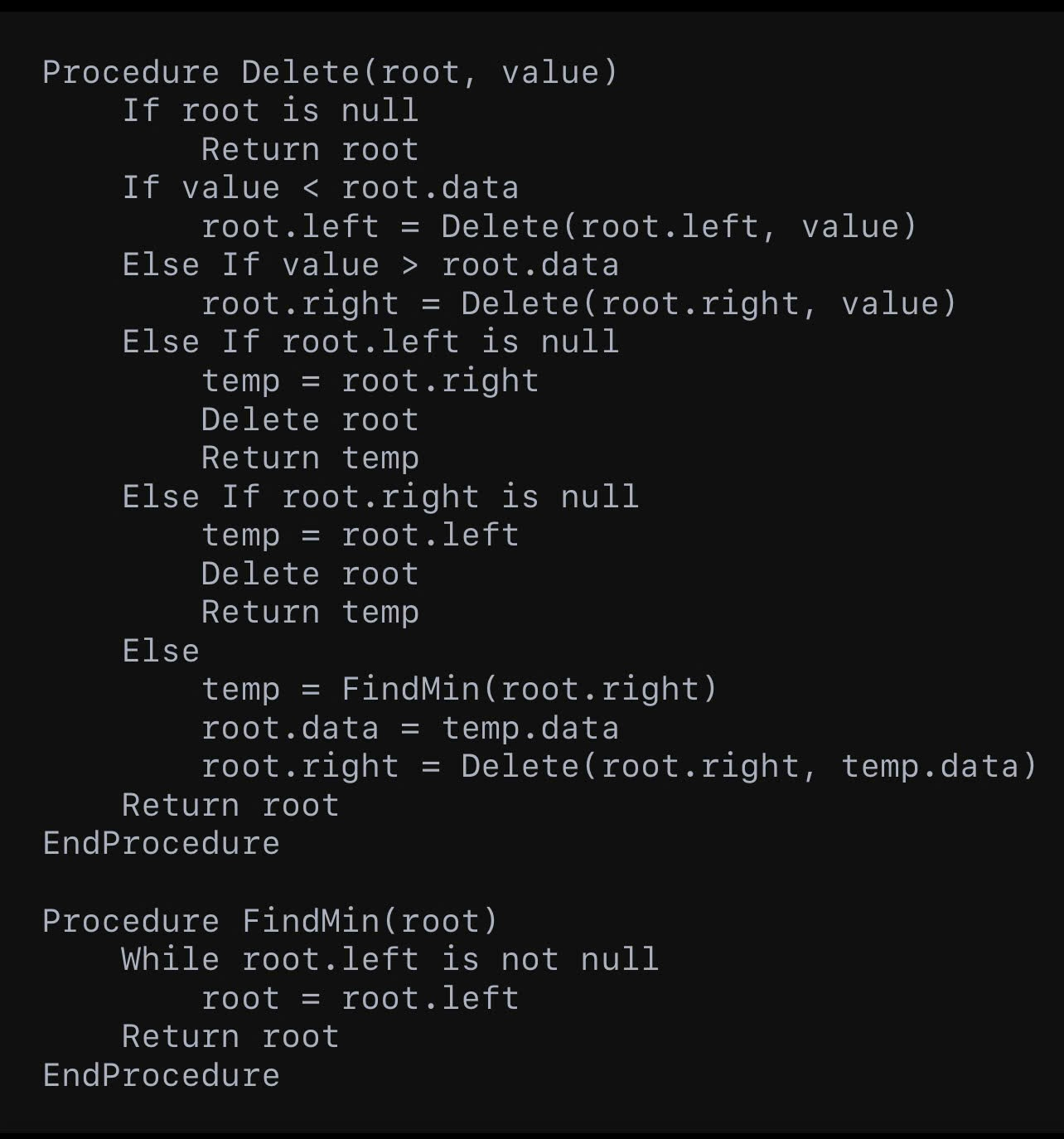
1. Nếu giá trị cần xóa nhỏ hơn giá trị nút hiện tại, tìm và xóa trong cây con trái.

2. Nếu giá trị cần xóa lớn hơn giá trị nút hiện tại, tìm và xóa trong cây con phải.

3. Nếu nút cần xóa có một hoặc không có con, xóa nó và thay thế với con của nó.

4. Nếu nút cần xóa có hai con, tìm giá trị thay thế (nút kế tiếp), sao chép và xóa nút kế tiếp.

**\*lưu đồ xóa : **

**\*ngôn ngữ giả : **

**IV.GIỚI THIỆU CHƯƠNG TRÌNH**

**Đây là giao diện khi băt đầu** 

**Giao diện này được thiết kế bằng Visual studio chúng ta cần thiết kế một giao diện để bắt đầu chạy cái phần chương trình mà mình đã viết khi thiết kế xong và viết chương trình cho nó thì chúng ta bắt đầu chạy**

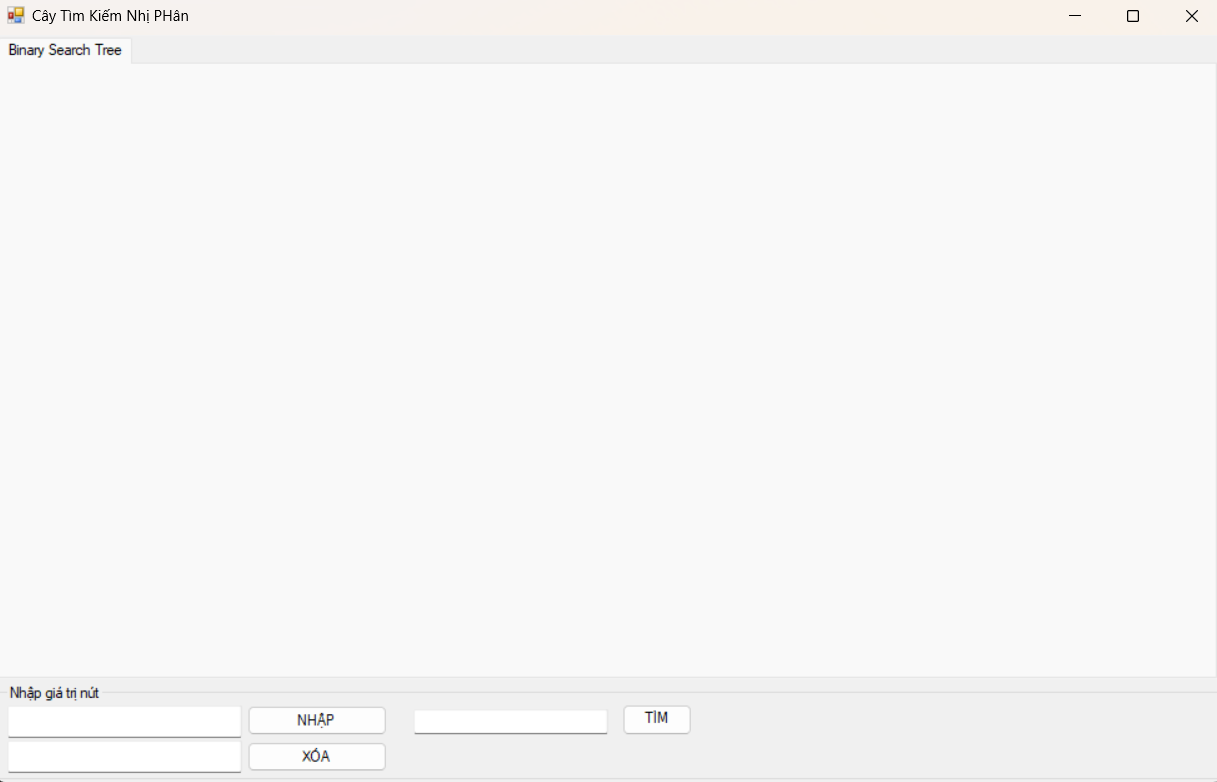


**Đây là giao diện tổng quan của chúng ta , muốn chạy chương trình chúng ta cần bấm vào nút start** 

**Sau đó nó sẽ bắt đầu chạy và hiện ra cái giao diện để cho mình thao tác**



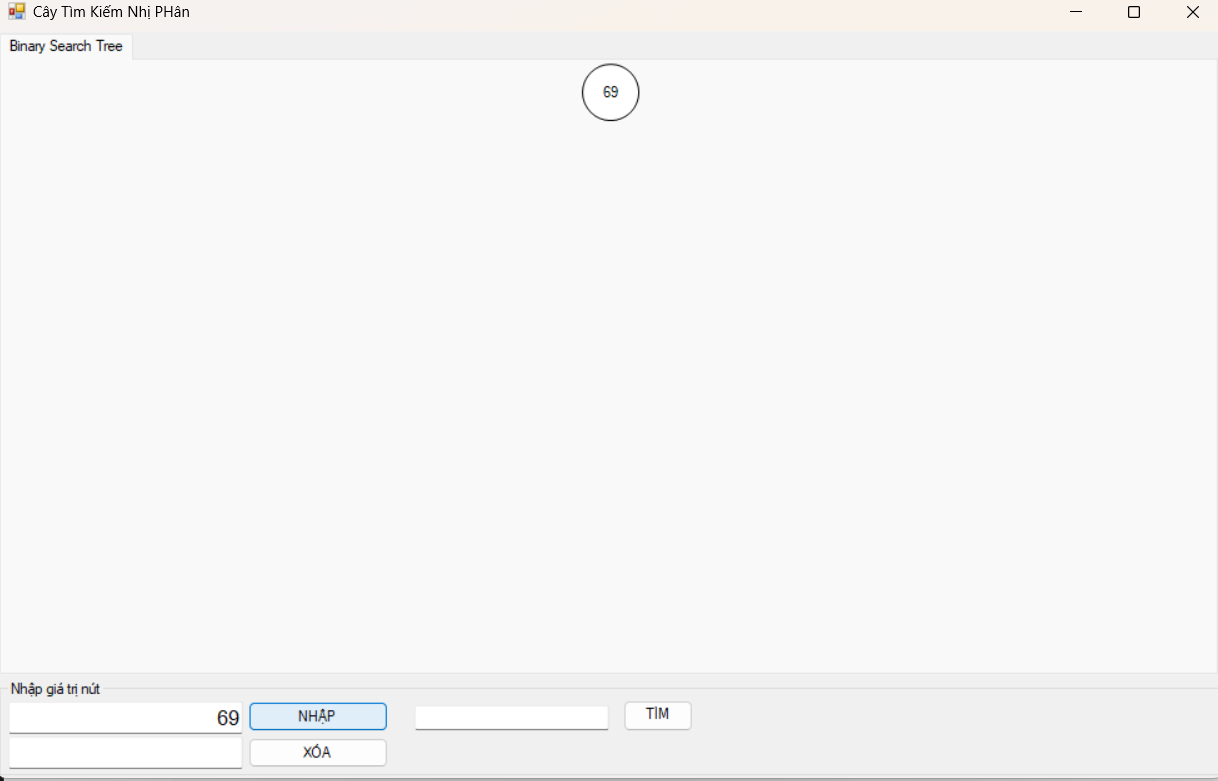
**Tiếp theo chúng ta cần bấm bắt đầu để thực hiện thao tác tìm kiếm cây nhị phân**



**Đây là nơi để chúng ta thao tác tìm kiếm cây nhị phân**

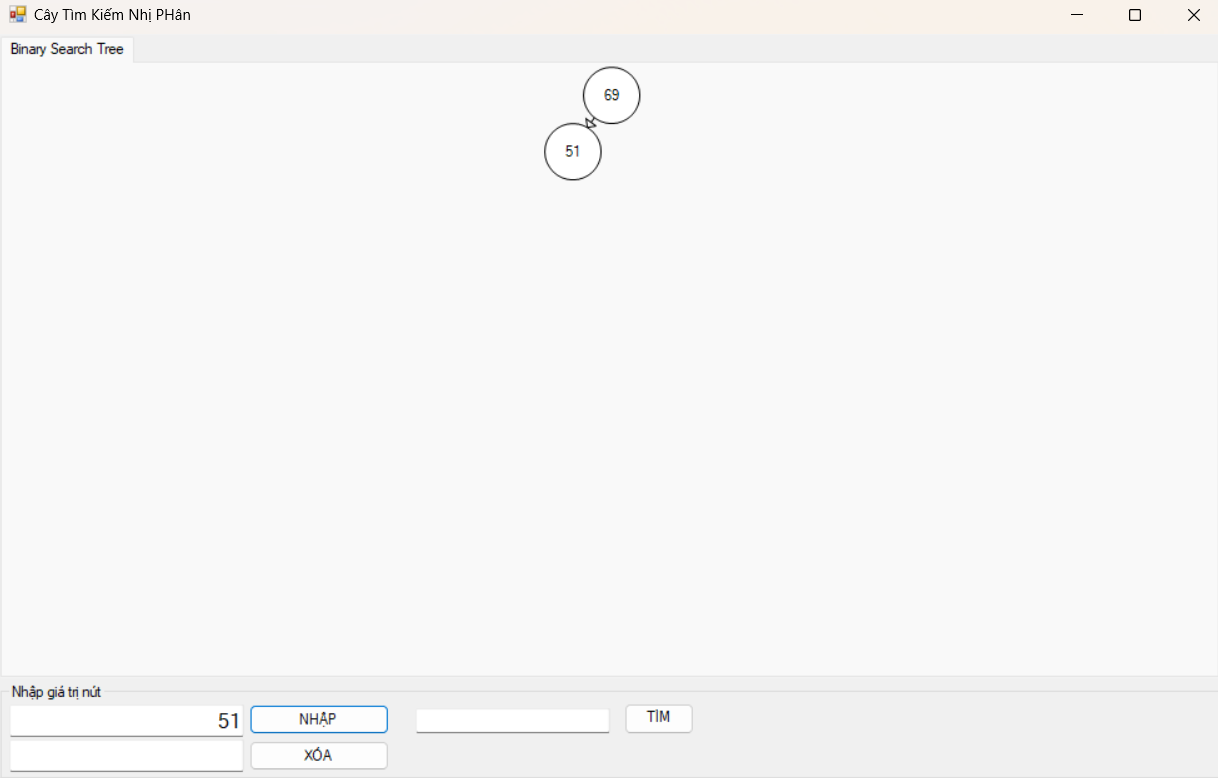
**Tôi sẽ ví dụ thao tác :**

**Đầu tiên ta cần nhập giá trị vào ô nhập : lấy 69 làm nút gốc**

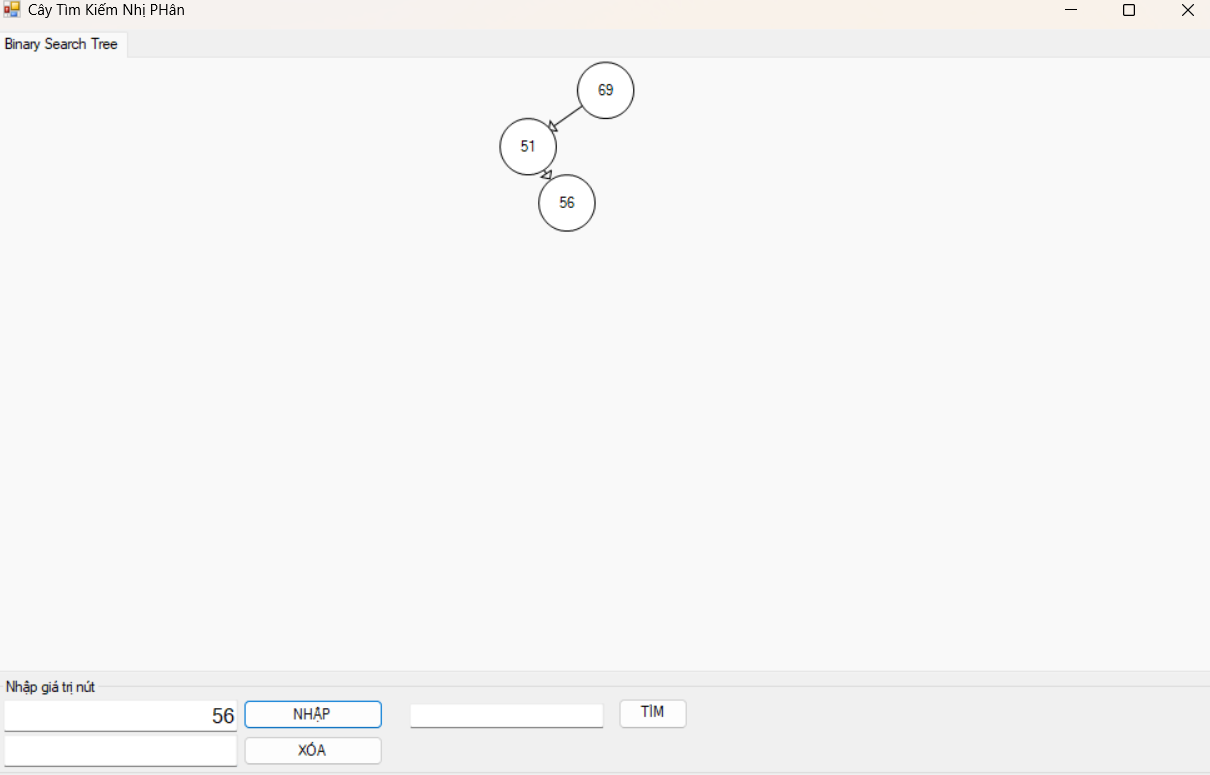


**Tiếp theo tôi sẽ chọn một nút khi nút đó nhỏ hơn nút gốc thì nó sẽ về bên trái và khi nút đó lớn hơn nút gốc thì nó sẽ về bên phải**

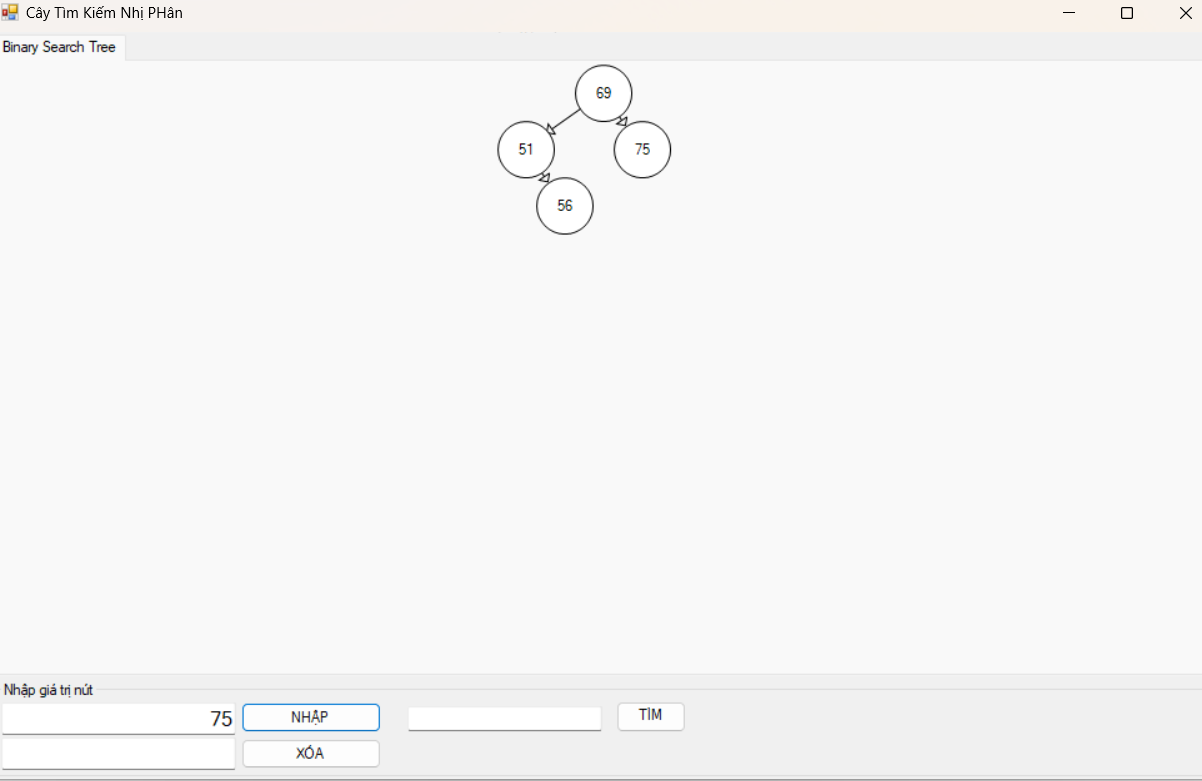
**Tôi sẽ chọn 51 thì chắc chắn 51 sẽ nhỏ hơn 69 nên nó sẽ về bên trái**



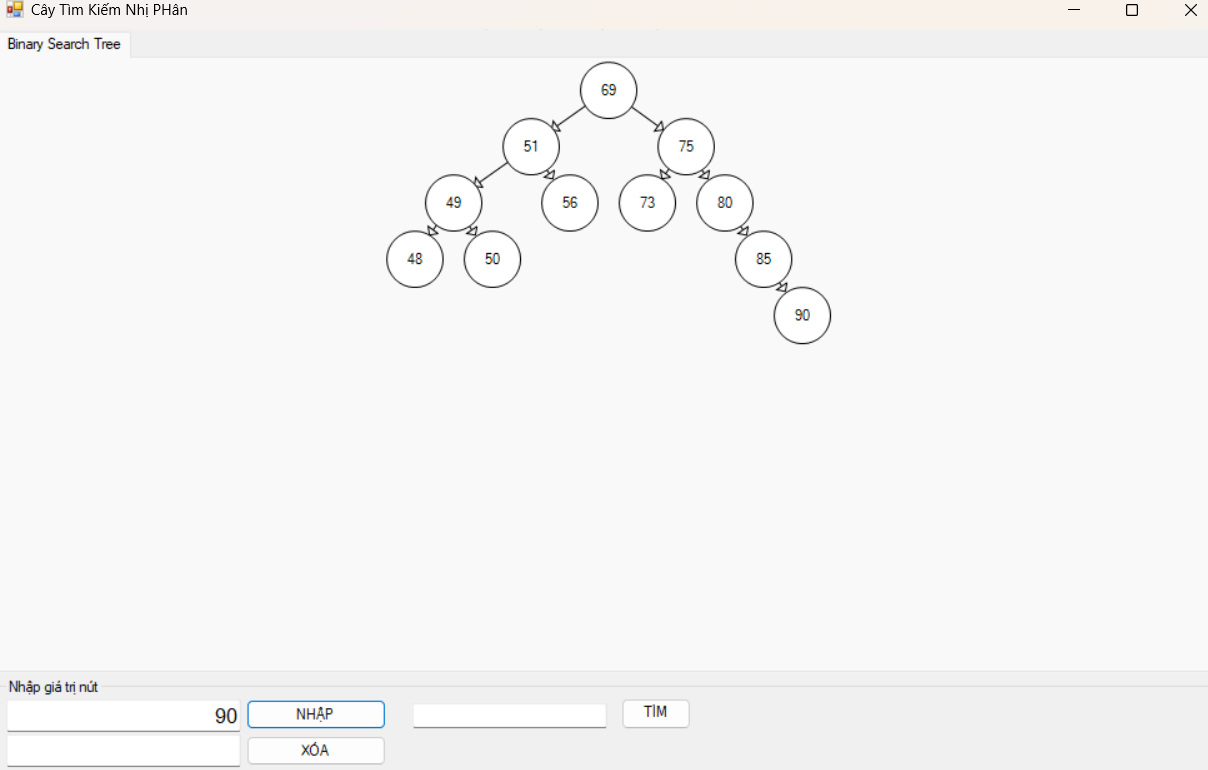
**Tôi chọn 56 vì 56 nhỏ hơn nút gốc 69 nhưng 56 lại lớn hơn nút lá 51 nên nút 56 sẽ ở bên phải nút lá 51**



**Tôi chọn tiếp một số để cho nó lớn hơn nút gốc , tôi sẽ chọn 75 vì 75 lớn hơn nút gốc 69 nên 75 sẽ nằm bên phải nút gốc 69**

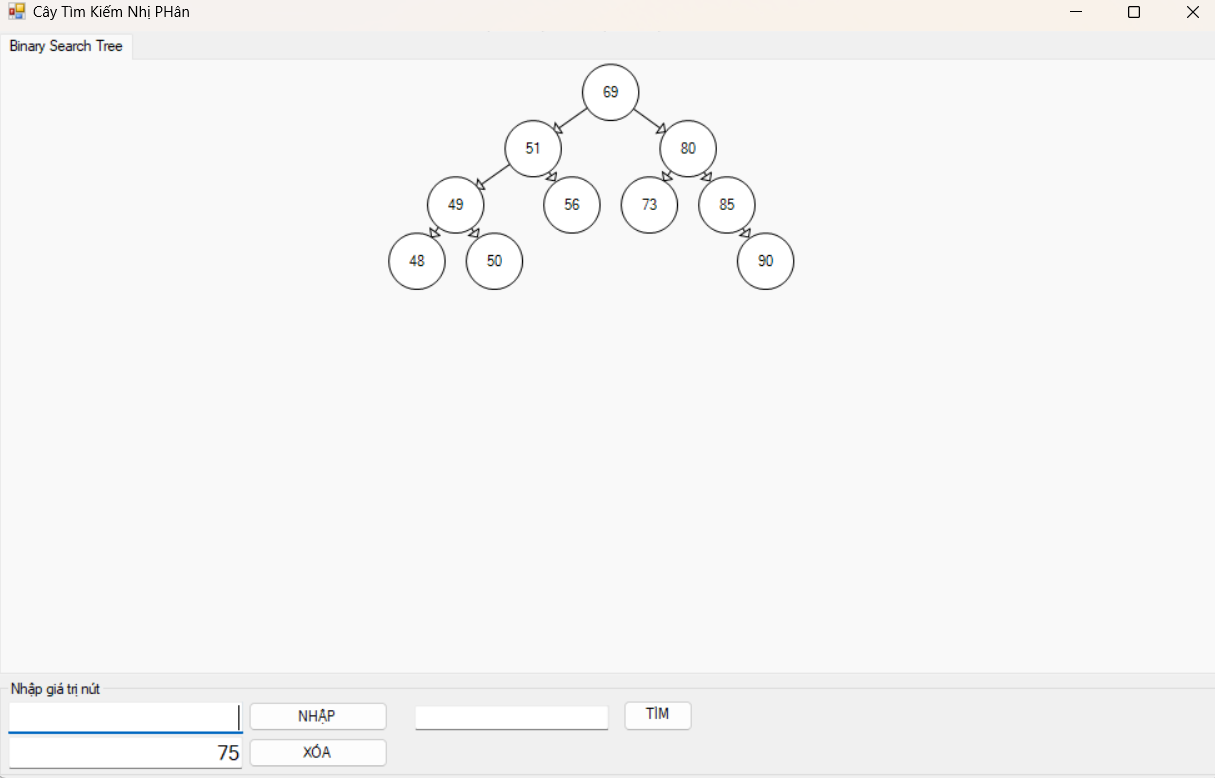


**Tương tự như vậy tôi sẽ thêm 49,50,48,73,80,85,90,để hoàn thành cây**



**Đây là cây sau khi hoàn thành .**

**Tiếp theo tới xóa một phần tử trong cây vừa hoàn thành tôi sẽ chọn75 để xóa**



**Sau khi xóa nút 75 đi thì nút 80 nó thay thế vào nút vừa bị xóa .**

**CHƯƠNG IV:KẾT LUẬN**

Sau một thời gian tìm hiểu, nghiên cứu và thực hiện đề tài. Các yêu cầu chính của đề tài cơ bản đã hoàn tất với các nội dung sau:

**I. Ưu điểm**

• Xây dựng được chương trình “tìm kiếm cây nhị phân có giao diện đồ họa “

• Chương trình sử lý nhanh và tương đối chính xác.

**II. Khuyết điểm**

• Mặc dù rất cố gắng nhưng trong thời gian ngắn, kinh nghiệm còn hạn chế nên kết quả còn thiếu sót cần tiếp tục được hoàn thiện để có thể giải được các yêu cầu phức tạp hơn.

• Chương trình còn nhiều lỗi như: về vấn đề xử lý hay thuật toán truy vết (tìm kiếm kết quả) chưa tối ưu …

**III. Hướng phát triển**

• Xây dựng hoàn thiện các chức năng giúp người sử dụng dễ dàng hơn, phương pháp qui hoạch động tương đối tối ưu và hiệu quả hơn.

• Có thể sử dụng phương pháp để giải một số bài toán tương tự. Trên đây là kết quả đạt được cũng như còn một số tồn tại, hướng phát triển của đề tài.

**TÀI LIỆU THAM KHẢO**

[1]. Cẩm nang thuật toán – cuốn 1 – Robert Sedgewich – Trần Đan Thư.

[2]. Lập trình = Thuật toán + CTDL, N. Wirth.

[3]. Giải thuật - Nguyễn Văn Linh - Khoa CNTT