KHOA KỸ THUẬT VÀ CÔNG NGHỆ

**BỘ MÔN CÔNG NGHỆ THÔNG TIN**



**THỰC TẬP ĐỒ ÁN CƠ SỞ NGÀNH**

**HỌC KỲ I, NĂM HỌC 2024-2025**

**ỨNG DỤNG CẤU TRÚC CHỒNG (STACK) ĐỂ KHỬ ĐỆ QUY**

**KHI DUYỆT CÂY NHỊ PHÂN**

**TÌM KIẾM**

*Giảng viên hướng dẫn:*

ThS. Lê Minh Tự

*Sinh viên thực hiện:*

Họ tên: Dương Bảo Khanh

MSSV: 110122009

Lớp: DA22TTA

***Trà Vinh, tháng 12 năm 2024***

KHOA KỸ THUẬT VÀ CÔNG NGHỆ

**BỘ MÔN CÔNG NGHỆ THÔNG TIN**



**THỰC TẬP ĐỒ ÁN** **CƠ SỞ NGÀNH**

**HỌC KỲ I, NĂM HỌC 2024-2025**

**ỨNG DỤNG CẤU TRÚC CHỒNG (STACK) ĐỂ KHỬ ĐỆ QUY**

**KHI DUYỆT CÂY NHỊ PHÂN**

**TÌM KIẾM**

*Giảng viên hướng dẫn:*

ThS. Lê Minh Tự

*Sinh viên thực hiện:*

Họ tên: Dương Bảo Khanh

MSSV: 110122009

Lớp: DA22TTA

***Trà Vinh, tháng 12 năm 2024***

**NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN**

*Trà Vinh, ngày ….. tháng …… năm ……*

**Giáo viên hướng dẫn**

*(Ký tên và ghi rõ họ tên)*

*Trà Vinh, ngày ….. tháng …… năm ……*

**Giáo viên hướng dẫn**

*(Ký tên và ghi rõ họ tên)*

**LỜI CẢM ƠN**

**NHẬN XÉT CỦA THÀNH VIÊN HỘI ĐỒNG**

*Trà Vinh, ngày ….. tháng …… năm ……*

**Thành viên hội đồng**

*(Ký tên và ghi rõ họ tên)*

Em xin gửi lời cảm ơn chân thành và sâu sắc nhất đến thầy Lê Minh Tự, người đã tận tình chỉ bảo, định hướng và hỗ trợ em trong suốt quá trình thực hiện đồ án. Những kiến thức và kinh nghiệm quý báu mà thầy chia sẻ không chỉ giúp em hoàn thành tốt đồ án này mà còn mang đến cho em nhiều bài học hữu ích, làm nền tảng vững chắc cho quá trình học tập và làm việc sau này.

Sự tận tâm và nhiệt tình của thầy trong việc hướng dẫn đã giúp em hiểu rõ hơn về các vấn đề phức tạp, tháo gỡ những khó khăn mà em gặp phải trong suốt quá trình thực hiện, đồng thời giúp em tự tin hơn trong việc tiếp cận và giải quyết vấn đề một cách khoa học và hiệu quả. Em thực sự trân trọng sự đồng hành và những đóng góp quý giá từ thầy.

Em xin chân thành cảm ơn!

**MỤC LỤC**

[CHƯƠNG 1: TỔNG QUAN 11](#_Toc187275995)

[1.1 Tổng quan về cây nhị phân tìm kiếm 11](#_Toc187275996)

[1.2 Tổng quan về Stack và khử đệ quy 11](#_Toc187275997)

[CHƯƠNG 2: NGHIÊN CỨU LÝ THUYẾT 12](#_Toc187275998)

[2.1 Lý thuyết cây nhị phân tìm kiếm 12](#_Toc187275999)

[2.1.1 Khái niệm cây nhị phân tìm kiếm 12](#_Toc187276000)

[2.1.2 Biểu diễn cây nhị phân tìm kiếm 12](#_Toc187276001)

[2.1.3 Nút (Node) trong cây nhị phân tìm kiếm 12](#_Toc187276002)

[2.1.4 Các phương pháp duyệt cây nhị phân tìm kiếm 13](#_Toc187276003)

[2.1.5 Thao tác chèn trong cây nhị phân tìm kiếm 13](#_Toc187276004)

[2.2 Lý thuyết về Stack 13](#_Toc187276005)

[2.2.1 Khái niệm Stack 13](#_Toc187276006)

[2.2.2 Các thao tác trên Stack 13](#_Toc187276007)

[2.2.3 Thuật toán khử đệ quy 14](#_Toc187276008)

[CHƯƠNG 3: HIỆN THỰC HÓA NGHIÊN CỨU 15](#_Toc187276009)

[3.1 Cấu trúc dữ liệu cây nhị phân tìm kiếm 15](#_Toc187276010)

[3.2 Cấu trúc Stack và các thao tác cơ bản 15](#_Toc187276011)

[3.3 Hàm thêm phần tử vào cây nhị phân tìm kiếm 17](#_Toc187276012)

[3.4 Tạo cây nhị phân với dữ liệu ngẫu nhiên 19](#_Toc187276013)

[3.5 Các phương pháp duyệt cây 20](#_Toc187276014)

[3.6 Hàm Solve 25](#_Toc187276015)

[3.7 Hàm main 32](#_Toc187276016)

[CHƯƠNG 4: KẾT QUẢ NGHIÊN CỨU 33](#_Toc187276017)

[4.1 Giao diện màn hình chính 33](#_Toc187276018)

[4.2 Giao diện màn hình chọn phương pháp duyệt 33](#_Toc187276019)

[4.3 Giao diện màn hình chọn cách duyệt 33](#_Toc187276020)

[4.4 Giao diện màn hình khi duyệt xong 34](#_Toc187276021)

[CHƯƠNG 5: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN 35](#_Toc187276022)

[5.1 Kết luận 35](#_Toc187276023)

[5.2 Hạn chế 35](#_Toc187276024)

[5.3 Hướng phát triển 35](#_Toc187276025)

[DANH MỤC TÀI LIỆU THAM KHẢO 36](#_Toc187276026)

**DANH MỤC HÌNH ẢNH**

[Hình 2.1.1: Ví dụ cây nhị phân tìm kiếm 11](#_Toc187066341)

[Hình 4.1: Giao diện màn hình chính 32](#_Toc187066342)

[Hình 4.2: Giao diện màn hình chọn phương pháp duyệt 32](#_Toc187066343)

[Hình 4.3: Giao diện màn hình con chọn cách duyệt 33](#_Toc187066344)

[Hình 4.4: Giao diện hiển thị kết quả 33](#_Toc187066345)

**DANH MỤC TỪ VIẾT TẮT**

|  |  |
| --- | --- |
| **Từ viết tắt** | **Ý nghĩa** |
| BST | Binary Search Tree |
| NLR | Pre-Order |
| LNR | In-Order |
| LRN | Post-Order |

**TÓM TẮT ĐỒ ÁN CƠ SỞ NGÀNH**

**Mục tiêu của đồ án**

Đồ án tập trung vào ứng dụng cấu trúc chồng (stack) để khử đệ quy khi duyệt cây nhị phân tìm kiếm hướng đến việc nghiên cứu và triển khai cấu trúc dữ liệu cây nhị phân tìm kiếm (Binary Search Tree - BST) để lưu trữ và quản lý dữ liệu số nguyên. Cây nhị phân tìm kiếm không chỉ giúp lưu trữ dữ liệu có trật tự mà còn hỗ trợ các thao tác cơ bản như thêm phần tử và duyệt cây một cách hiệu quả. Đồng thời, đồ án sử dụng cấu trúc dữ liệu Stack để thay thế phương pháp đệ quy trong các thuật toán duyệt cây phổ biến như Pre-order (NLR), In-order (LNR), và Post-order (LRN). Mục tiêu chính là chứng minh tính hiệu quả của Stack trong việc khử đệ quy.

**Nội dung thực hiện**

Nội dung của đồ án tập trung vào việc cài đặt và thử nghiệm các thao tác cơ bản trên cây nhị phân tìm kiếm. Phần đầu triển khai cấu trúc cây với các tính năng như thêm phần tử mới và thực hiện các thuật toán duyệt cây. Sau đó, cấu trúc dữ liệu Stack được xây dựng và sử dụng để khử đệ quy cho các phương pháp duyệt cây. Stack cho phép lưu trữ trạng thái của các nút một cách tuần tự, giúp thay thế hiệu quả ngăn xếp hệ thống (call stack) mà đệ quy sử dụng.

**Kết quả đạt được**

Đồ án đã xây dựng thành công cây nhị phân tìm kiếm với các thao tác cơ bản và thực hiện thành công các thuật toán duyệt cây theo cả hai cách: đệ quy và khử đệ quy. Qua thử nghiệm, phương pháp sử dụng Stack giúp tối ưu hóa bộ nhớ, đảm bảo tính ổn định và phù hợp với các bài toán xử lý dữ liệu lớn. Đồ án không chỉ minh họa tính ứng dụng của cây nhị phân tìm kiếm mà còn mở ra hướng phát triển các cấu trúc dữ liệu nâng cao hơn.

**Hướng phát triển**

Kết quả từ đồ án có thể được mở rộng bằng cách phát triển các tính năng như xóa nút trên cây, cân bằng cây tự động, hoặc áp dụng các cấu trúc nâng cao như AVL Tree hay Red-Black Tree. Ngoài ra, việc nghiên cứu thêm các thuật toán tìm kiếm và xử lý dữ liệu trên cây lớn hơn cũng là hướng đi triển vọng.

**MỞ ĐẦU**

**Lý do chọn đề tài**

Cấu trúc dữ liệu và giải thuật là nền tảng của khoa học máy tính, đóng vai trò quan trọng trong việc phát triển các hệ thống phần mềm. Trong số đó, cây nhị phân tìm kiếm là một cấu trúc dữ liệu phổ biến, được sử dụng để lưu trữ và quản lý dữ liệu có thứ tự. Với tính chất đặc biệt, BST hỗ trợ hiệu quả các thao tác tìm kiếm, chèn và xóa phần tử với độ phức tạp trung bình là O(log n) nếu cây được duy trì cân bằng.

Tuy nhiên, khi kích thước cây lớn, các phương pháp duyệt cây nhị phân thường được triển khai bằng đệ quy có thể dẫn đến tiêu tốn bộ nhớ hoặc lỗi tràn ngăn xếp (stack overflow). Để giải quyết vấn đề này, việc ứng dụng cấu trúc dữ liệu Stack để khử đệ quy trong các thuật toán duyệt cây không chỉ giúp tối ưu hóa hiệu suất mà còn đảm bảo tính ổn định và khả năng mở rộng trong các bài toán lớn. Do đó, việc nghiên cứu và triển khai các thuật toán khử đệ quy trên cây nhị phân tìm kiếm mang lại nhiều giá trị thực tiễn và học thuật.

**Mục tiêu nghiên cứu**

Mục tiêu của đề tài là xây dựng và triển khai cấu trúc dữ liệu ây nhị phân tìm kiếm để lưu trữ và quản lý dữ liệu số nguyên. Các thao tác cơ bản trên cây như thêm phần tử và duyệt cây sẽ được triển khai bằng cả phương pháp đệ quy và khử đệ quy. Cụ thể, đề tài tập trung vào việc sử dụng cấu trúc dữ liệu Stack để khử đệ quy cho ba phương pháp duyệt cây cơ bản: Pre-order (NLR), In-order (LNR) và Post-order (LRN). Ngoài ra, đề tài còn hướng đến so sánh hiệu quả giữa hai phương pháp đệ quy và khử đệ quy, từ đó đánh giá tính ưu việt của phương pháp khử đệ quy trong việc xử lý dữ liệu lớn.

**Phạm vi nghiên cứu**

Đề tài tập trung triển khai các thao tác cơ bản trên cây nhị phân tìm kiếm với tập dữ liệu là các số nguyên. Việc khử đệ quy được thực hiện bằng cách sử dụng cấu trúc dữ liệu Stack để thay thế đệ quy truyền thống trong các thuật toán duyệt cây. Phạm vi nghiên cứu không bao gồm các cây nhị phân nâng cao như AVL Tree hoặc Red-Black Tree mà chỉ giới hạn ở cây nhị phân tìm kiếm thông thường. Ngoài ra, đề tài chỉ tập trung so sánh hiệu quả của hai phương pháp duyệt cây trên cùng một tập dữ liệu đầu vào.

**Phương pháp nghiên cứu**

Để thực hiện đề tài, các phương pháp nghiên cứu lý thuyết và thực nghiệm được áp dụng. Đầu tiên, nghiên cứu các khái niệm cơ bản về cây nhị phân tìm kiếm, cấu trúc dữ liệu Stack và các phương pháp duyệt cây phổ biến. Sau đó, triển khai các thao tác thêm phần tử và duyệt cây bằng cả đệ quy và khử đệ quy. Tiếp theo, thực nghiệm các thuật toán trên cùng một tập dữ liệu để thu thập kết quả và đánh giá hiệu quả giữa hai phương pháp. Cuối cùng, phân tích kết quả thực nghiệm để rút ra những kết luận về ưu và nhược điểm của phương pháp khử đệ quy so với đệ quy truyền thống.

# CHƯƠNG 1: TỔNG QUAN

## Tổng quan về cây nhị phân tìm kiếm

Cây nhị phân tìm kiếm là một cấu trúc dữ liệu cây nhị phân đặc biệt, trong đó mỗi nút có tối đa hai nút con: nút con trái và nút con phải. Dữ liệu trong cây được tổ chức theo quy tắc: giá trị của nút con trái nhỏ hơn giá trị của nút gốc, và giá trị của nút con phải lớn hơn giá trị của nút gốc. Tính chất này được áp dụng đệ quy cho tất cả các nút trong cây.

Cây nhị phân tìm kiếm hỗ trợ hiệu quả các thao tác cơ bản như thêm, tìm kiếm và xóa phần tử. Khi cây được duy trì ở trạng thái cân bằng, các thao tác này có độ phức tạp trung bình là O(log n), giúp xử lý dữ liệu nhanh chóng và hiệu quả. Tuy nhiên, nếu cây mất cân bằng (tất cả các nút chỉ nằm về một phía), hiệu suất của BST có thể giảm xuống O(n), tương tự như danh sách liên kết.

## Tổng quan về Stack và khử đệ quy

Stack (ngăn xếp) là một cấu trúc dữ liệu tuyến tính hoạt động theo nguyên tắc LIFO (Last In, First Out), nghĩa là phần tử được đưa vào sau cùng sẽ được lấy ra trước. Stack cung cấp các thao tác cơ bản như thêm phần tử (Push), lấy phần tử (Pop), xem phần tử trên cùng (Peek), và kiểm tra ngăn xếp có rỗng không (IsEmpty).

Trong các thuật toán duyệt cây nhị phân, Stack được sử dụng để thay thế cơ chế đệ quy bằng cách lưu trữ trạng thái của các nút một cách thủ công. Điều này giúp khắc phục hạn chế của đệ quy, đặc biệt là lỗi tràn ngăn xếp khi xử lý cây lớn. Với sự hỗ trợ của Stack, quá trình duyệt cây trở nên rõ ràng và hiệu quả hơn, đảm bảo tính ổn định và khả năng mở rộng của thuật toán.

# CHƯƠNG 2: NGHIÊN CỨU LÝ THUYẾT

## Lý thuyết cây nhị phân tìm kiếm

* + 1. Khái niệm cây nhị phân tìm kiếm

Một cây tìm kiếm nhị phân (Binary Search Tree - BST) là một cây mà trong đó tất cả các nút đều có các đặc điểm sau:

* Cây con bên trái của một nút có khóa (key) nhỏ hơn hoặc bằng giá trị khóa của nút cha (của cây con này).
* Cây con bên phải của một nút có khóa lớn hơn hoặc bằng giá trị khóa của nút cha (của cây con này).
  + 1. Biểu diễn cây nhị phân tìm kiếm

Cây tìm kiếm nhị phân là một tập hợp bao gồm các nút được sắp xếp theo cách để chúng có thể duy trì hoặc tuân theo các đặc điểm của cây tìm kiếm nhị phân. Mỗi một nút thì đều có một khóa và giá trị liên kết với nó. Trong khi tìm kiếm, khóa cần tìm được so sánh với các khóa trong cây tìm kiếm nhị phân và nếu tìm thấy, giá trị liên kết sẽ được thu nhận.



Hình 2.1.1: Ví dụ cây nhị phân tìm kiếm

Từ hình ví dụ minh họa trên ta thấy rằng, khóa của nút gốc có giá trị 27 và tất cả khóa bên trái của cây con bên trái đều có giá trị nhỏ hơn 27 này và tất cả các khóa bên phải của cây con bên phải đều có giá trị lớn hơn 27.

* + 1. Nút (Node) trong cây nhị phân tìm kiếm

Một nút có một vài dữ liệu, tham chiếu tới các nút con bên trái và nút con bên phải của nó.

* + 1. Các phương pháp duyệt cây nhị phân tìm kiếm

In-order traversal: Với phương pháp này, chúng ta bắt đầu từ cây con bên trái và tiếp tục đến cây con gốc và cây con bên phải.

Pre-order traversal: Trong phương pháp này, nút gốc được truy cập đầu tiên, tiếp theo là cây con bên trái và cây con bên phải.

Post-Order Traversal: truy cập vào nút gốc cuối cùng. Chúng ta bắt đầu từ cây con bên trái, sau đó đến cây con bên phải, và sau đó là nút gốc.

* + 1. Thao tác chèn trong cây nhị phân tìm kiếm

Thao tác chèn sử dụng thao tác tìm kiếm để xác định vị trí nơi nút mới sẽ được chèn. Quá trình này bắt đầu từ nút gốc và quá trình tìm kiếm bắt đầu cho đến khi đến đích. Có ba trường hợp cần xem xét:

* Trường hợp 1: Khi không có nút nào tồn tại. Nút được chèn vào sẽ trở thành nút gốc.
* Trường hợp 2: Không có con. Trong trường hợp này, nút được chèn sẽ được so sánh với nút gốc. Nếu nó lớn hơn, nó sẽ trở thành nút con bên phải; nếu không, nó sẽ trở thành nút con trái.
* Trường hợp 3: Khi có cả nút gốc và con của nó. Nút mới sẽ được so sánh với mỗi nút trên đường dẫn của nó để xác định nút nào nó sẽ truy cập tiếp theo. Nếu nút này lớn hơn nút gốc, nó sẽ đi qua cây con bên phải, nếu không thì bên trái. Tương tự, so sánh được thực hiện ở mỗi cấp độ để xác định xem nó sẽ đi sang phải hay sang trái cho đến khi đến đích.
  1. Lý thuyết về Stack
     1. Khái niệm Stack

Stack là một danh sách có thứ tự trong đó việc chèn và xóa được thực hiện ở một đầu, được gọi là top - đỉnh. Phần tử cuối cùng được chèn là phần tử đầu tiên sẽ bị xóa. Do đó, nó được gọi là Last in First out (LIFO) hoặc First in Last out (FILO) list.

* + 1. Các thao tác trên Stack

Push: Thêm một phần tử vào đỉnh Stack.

Pop: Lấy phần tử trên cùng của Stack ra và xóa nó khỏi Stack.

Peek: Xem phần tử trên cùng mà không xóa.

IsEmpty: Kiểm tra xem Stack có rỗng hay không.

* + 1. Thuật toán khử đệ quy

**Duyệt Pre-order (NLR):**

Trong duyệt NLR, nút gốc được xử lý trước, tiếp theo là cây con trái và cuối cùng là cây con phải. Sử dụng Stack, thuật toán hoạt động như sau:

- Bắt đầu từ nút gốc, lưu nút hiện tại vào Stack và xử lý.

- Duyệt cây con trái trước, khi không còn nút con trái thì lấy nút trên cùng của Stack để tiếp tục duyệt cây con phải.

**Duyệt In-order (LNR):**

Trong duyệt LNR, thuật toán lần lượt đi đến nút trái nhất trước khi xử lý nút gốc và cây con phải. Sử dụng Stack, các bước thực hiện bao gồm:

- Lưu nút hiện tại vào Stack và tiếp tục duyệt cây con trái.

- Khi không còn nút con trái, lấy nút trên cùng của Stack, xử lý nó và chuyển sang cây con phải.

**Duyệt Post-order (LRN):**

Duyệt LRN yêu cầu xử lý các nút con trước khi xử lý nút gốc. Thuật toán khử đệ quy cho Post-order phức tạp hơn, cần kết hợp Stack và trạng thái đánh dấu nút đã được duyệt qua:

- Sử dụng Stack để đi đến nút trái nhất.

- Khi nút không còn con trái hoặc con phải, xử lý nút hiện tại. Nếu nút đã có con phải được duyệt trước, xử lý nút gốc.

CHƯƠNG 3: HIỆN THỰC HÓA NGHIÊN CỨU

* 1. Cấu trúc dữ liệu cây nhị phân tìm kiếm

Cây nhị phân tìm kiếm được xây dựng dựa trên cấu trúc NODE. Mỗi nút trong cây lưu trữ một giá trị kiểu số nguyên (data) và có hai con trỏ trỏ đến cây con bên trái (pLeft) và cây con bên phải (pRight). Cây được khởi tạo bằng con trỏ TREE, trỏ đến nút gốc (root) của cây, từ đó quản lý toàn bộ các nút trong cây. Khởi tạo cây ban đầu bằng cách gán root là NULL, điều này đảm bảo cây hoàn toàn trống trước khi các nút được thêm vào.

|  |
| --- |
| struct NODE  {  int data;  struct NODE \*pRight,\*pLeft;  };  typedef NODE\*TREE;  void Init(TREE &root)  {  root = NULL;  } |

* 1. Cấu trúc Stack và các thao tác cơ bản

Để khử đệ quy trong các phương pháp duyệt cây, chương trình sử dụng Stack (ngăn xếp) để lưu trạng thái của các nút trong quá trình duyệt. Stack được cài đặt dưới dạng mảng tĩnh với kích thước tối đa cố định max. Các phần tử trong Stack là con trỏ trỏ đến các nút trong cây.

Cấu trúc Stack bao gồm:

- a[100]: Mảng lưu trữ các con trỏ đến các nút.

- top: Biến theo dõi vị trí của phần tử trên cùng trong Stack.

|  |
| --- |
| struct STACK  {  NODE \*a[1000];  int top;  }; |

Các thao tác cơ bản trên Stack được triển khai như sau:

- Kiểm tra rỗng (IsEmpty): Hàm kiểm tra xem Stack có rỗng không bằng cách kiểm tra giá trị của top. Nếu top == -1, Stack đang rỗng.

|  |
| --- |
| int IsEmpty(STACK stack)  {  return stack.top == -1;  } |

- Thêm phần tử vào Stack (Push): Hàm Push thêm một con trỏ đến nút cây vào Stack, đồng thời tăng giá trị của top.

|  |
| --- |
| void Push(STACK &stack, NODE \*p)  {  stack.a[++stack.top] = p;  } |

- Lấy phần tử ra khỏi Stack (Pop): Hàm Pop lấy phần tử trên cùng của Stack ra, đồng thời giảm giá trị của top. Nếu Stack rỗng, hàm trả về NULL.

|  |
| --- |
| NODE \*Pop(STACK &stack)  {  if(IsEmpty(stack))  {  return NULL;  }  return stack.a[stack.top--];  } |

* 1. Hàm thêm phần tử vào cây nhị phân tìm kiếm

Thao tác thêm phần tử vào cây được thực hiện qua hàm InsertTree.

|  |
| --- |
| void InsertTree(TREE &root,int x)  {  if(root != NULL)  {  if(root->data == x)  {  return;  }  if(root->data > x)  {  InsertTree(root->pLeft,x);  }  else  {  InsertTree(root->pRight,x);  }  }  else  {  root = new NODE;  root->data= x;  root->pLeft = root->pRight = NULL;  }  } |

Để xây dựng toàn bộ cây nhị phân, chương trình sử dụng hàm BuildTree, cho phép người dùng nhập số lượng nút và giá trị từng nút, sau đó thêm vào cây bằng hàm InsertTree.

|  |
| --- |
| void BuildTree(TREE &root)  {  int x,n;  printf("Nhap so nut:");  scanf("%d",&n);  for(int i = 1;i <= n;i++)  {  printf("Nhap nut thu %d:",i);  scanf("%d",&x);  InsertTree(root,x);  }  printf("Da tao cay voi %d phan tu.",n);  } |

* 1. Tạo cây nhị phân với dữ liệu ngẫu nhiên

Ngoài việc tự nhập dữ liệu, chương trình còn hỗ trợ tạo cây nhị phân với các giá trị ngẫu nhiên. Hàm CreateRandomTree sử dụng hàm rand để sinh ngẫu nhiên các giá trị trong một khoảng xác định. Số lượng phần tử, giá trị nhỏ nhất và giá trị lớn nhất được nhập bởi người dùng.

|  |
| --- |
| void CreateRandomTree(TREE &root)  {  int n, minValue, maxValue;  printf("Nhap so luong nut: ");  scanf("%d", &n);  printf("Nhap khoang gia tri:");  scanf("%d %d", &minValue, &maxValue);  srand(time(0));  for (int i = 0; i < n; i++)  {  int x = minValue + rand() % (maxValue - minValue + 1);  InsertTree(root, x);  }  printf("Da tao cay voi %d phan tu ngau nhien.\n", n);  } |

* 1. Các phương pháp duyệt cây

Trong chương trình, ba phương pháp duyệt cây phổ biến được triển khai: Pre-order (NLR), In-order (LNR) và Post-order (LRN). Mỗi phương pháp đều được thực hiện bằng cả đệ quy và khử đệ quy.

**Duyệt Pre-order (NLR)**

Duyệt Pre-order thăm nút gốc trước, sau đó đến cây con trái và cuối cùng là cây con phải.

Đệ quy:

|  |
| --- |
| void NLRRecursive(TREE root)  {  if(root!=NULL)  {  printf("%d\t",root->data);  NLRRecursive(root->pLeft);  NLRRecursive(root->pRight);  }  } |

Khử đệ quy (dùng Stack):

|  |
| --- |
| void NLRIterative(TREE root)  {  NODE\* p = root;  STACK stack;  stack.top = -1;  while (p != NULL || !IsEmpty(stack))  {  if (p != NULL)  {  printf("%d\t", p->data);  Push(stack, p);  p = p->pLeft;  }  else  {  p = Pop(stack);  p = p->pRight;  }  }  } |

**Duyệt In-order (LNR)**

Duyệt In-order duyệt cây con trái trước, sau đó đến nút gốc, cuối cùng duyệt cây con phải. Phương pháp này trả về thứ tự tăng dần trong cây BST.

Đệ quy:

|  |
| --- |
| void LNRRecursive(TREE root)  {  if(root != NULL)  {  LNRRecursive(root->pLeft);  printf("%d\t",root->data);  LNRRecursive(root->pRight);  }  } |

Khử đệ quy (dùng Stack):

|  |
| --- |
| void LNRIterative(TREE root)  {  NODE \*p = root;  STACK stack;  stack.top = -1;  while(p != NULL || !IsEmpty(stack))  {  if(p != NULL)  {  Push(stack,p);  p = p->pLeft;  }  else  {  p = Pop(stack);  printf("%d\t",p->data);  p = p->pRight;  }  }  } |

**Duyệt Post-order (LRN)**

Duyệt Post-order thăm cây con trái, cây con phải, sau đó thăm nút gốc.

Đệquy:

|  |
| --- |
| void LRNRecursive(TREE root)  {  if(root != NULL)  {  LRNRecursive(root->pLeft);  LRNRecursive(root->pRight);  printf("%d\t",root->data);  }  } |

Khử đệ quy (dùng Stack):

|  |
| --- |
| void LRNIterative(TREE root)  {  NODE \*p = root;  STACK stack;  stack.top = -1;  NODE \*lastVisited = NULL;  while(p != NULL || !IsEmpty(stack))  {  if(p != NULL)  {  Push(stack,p);  p = p->pLeft;  }  else  {  if(stack.a[stack.top]->pRight != NULL && lastVisited != stack.a[stack.top]->pRight)  {  p = stack.a[stack.top]->pRight;  }  else  {  printf("%d\t",stack.a[stack.top]->data);  lastVisited = Pop(stack);  }  }  }  } |

* 1. Hàm Solve

Chức năng chính của hàm Solve là cung cấp một menu tương tác để người dùng lựa chọn các thao tác khác nhau. Đầu tiên, người dùng có thể chọn tạo cây bằng cách nhập dữ liệu thủ công hoặc sử dụng dữ liệu ngẫu nhiên. Nếu chọn nhập thủ công, hàm BuildTree sẽ được gọi để thêm lần lượt các giá trị người dùng nhập vào cây. Nếu chọn tạo cây ngẫu nhiên, người dùng cần cung cấp số lượng nút, giá trị nhỏ nhất và lớn nhất, sau đó hàm CreateRandomTree sẽ tự động sinh cây từ các giá trị ngẫu nhiên trong khoảng đã định.

Sau khi cây được tạo, người dùng có thể chọn thực hiện các phương pháp duyệt cây, bao gồm Pre-order (NLR), In-order (LNR), và Post-order (LRN). Với mỗi phương pháp, chương trình cung cấp thêm tùy chọn thực hiện bằng đệ quy, khử đệ quy, hoặc cả hai cách để so sánh kết quả. Nếu người dùng chọn duyệt bằng cả hai cách, chương trình sẽ lần lượt thực hiện cả đệ quy và khử đệ quy, đồng thời in kết quả của từng phương pháp lên màn hình.

Cuối cùng, chương trình hỗ trợ tùy chọn thoát để kết thúc vòng lặp và dừng chương trình. Trong trường hợp người dùng chọn tùy chọn không hợp lệ, chương trình sẽ thông báo lỗi và yêu cầu nhập lại.

|  |
| --- |
| void Solve()  {  TREE root;  Init(root);  int option;  while (true)  {  printf("\n===== MENU =====\n");  printf("1. Nhap du lieu thu cong de tao cay\n");  printf("2. Tao cay voi du lieu ngau nhien\n");  printf("3. Duyet cay\n");  printf("4. Thoat\n");  printf("Lua chon: ");  scanf("%d", &option);  if (option == 1)  {  printf("\n===== TAO CAY THU CONG =====\n");  BuildTree(root);  }  else if (option == 2)  {  printf("\n===== TAO CAY TU DONG =====\n");  CreateRandomTree(root);  }  else if (option == 3)  {  if (root == NULL)  {  printf("Cay hien tai dang rong. Vui long tao cay truoc.\n");  }  else  {  printf("\n===== MENU =====\n");  int traverseOption;  printf("1. Duyet Pre-order (NLR)\n");  printf("2. Duyet In-order (LNR)\n");  printf("3. Duyet Post-order (LRN)\n");  printf("Lua chon: ");  scanf("%d", &traverseOption);  int methodOption;  printf("\n===== MENU =====\n");  printf("1. De quy\n");  printf("2. Khu de quy\n");  printf("3. Ca hai\n");  printf("Lua chon: ");  scanf("%d", &methodOption);  if (traverseOption == 1)  {  if (methodOption == 1)  {  printf("\n===== KET QUA =====\n");  printf("Duyet Pre-order (De quy): ");  NLRRecursive(root);  }  else if (methodOption == 2)  {  printf("\n===== KET QUA =====\n");  printf("Duyet Pre-order (Khu de quy): ");  NLRIterative(root);  }  else if (methodOption == 3)  {  printf("\n===== KET QUA =====\n");  printf("Duyet Pre-order (De quy):/t");  NLRRecursive(root);  printf("\nDuyet Pre-order (Khu de quy): ");  NLRIterative(root);  }  else  {  printf("Lua chon khong hop le.\n");  }  }  else if (traverseOption == 2)  {  if (methodOption == 1)  {  printf("\n===== KET QUA =====\n");  printf("Duyet In-order (De quy): ");  LNRRecursive(root);  }  else if (methodOption == 2)  {  printf("\n===== KET QUA =====\n");  printf("Duyet In-order (Khu de quy): ");  LNRIterative(root);  }  else if (methodOption == 3)  {  printf("\n===== KET QUA =====\n");  printf("Duyet In-order (De quy):\t");  LNRRecursive(root);  printf("Duyet In-order (Khu de quy): ");  LNRIterative(root);  }  else  {  printf("Lua chon khong hop le.\n");  }  }  else if (traverseOption == 3)  {  if (methodOption == 1)  {  printf("\n===== KET QUA =====\n");  printf("Duyet Post-order (De quy): ");  LRNRecursive(root);  }  else if (methodOption == 2)  {  printf("\n===== KET QUA =====\n");  printf("Duyet Post-order (Khu de quy): ");  LRNIterative(root);  }  else if (methodOption == 3)  {  printf("\n===== KET QUA =====\n");  printf("Duyet Post-order (De quy):\t");  LRNRecursive(root);  printf("\nDuyet Post-order (Khu de quy): ");  LRNIterative(root);  }  else  {  printf("Lua chon khong hop le.\n");  }  }  else  {  printf("Lua chon khong hop le.\n");  }  printf("\n");  }  }  else if (option == 4)  {  printf("Dang thoat chuong trinh. Tam biet!\n");  break;  }  else  {  printf("Lua chon khong hop le. Vui long thu lai.\n");  }  }  } |

* 1. Hàm main

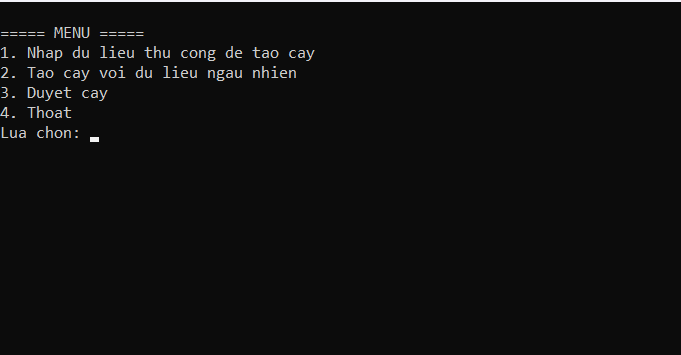
Hàm main gọi hàm Solve để bắt đầu chương trình.

|  |
| --- |
| int main()  {  Solve();  return 0;  } |

CHƯƠNG 4: KẾT QUẢ NGHIÊN CỨU

* 1. Giao diện màn hình chính

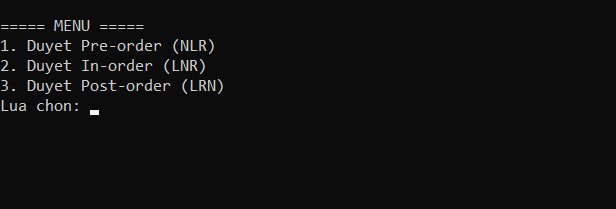
Giao diện chính của chương trình bao gồm 4 chức năng: nhập cây thủ công, tạo cây ngẫu nhiên, duyệt cây theo các phương pháp khác nhau và thoát chương trình.



Hình 4.1: Giao diện màn hình chính

* 1. Giao diện màn hình chọn phương pháp duyệt

Chương trình cung cấp menu cho phép người dùng lựa chọn ba phương pháp duyệt cây nhị phân tìm kiếm: duyệt Pre-order (NLR), duyệt In-order (LNR) và duyệt Post-order (LRN).



Hình 4.2: Giao diện màn hình chọn phương pháp duyệt

* 1. Giao diện màn hình chọn cách duyệt

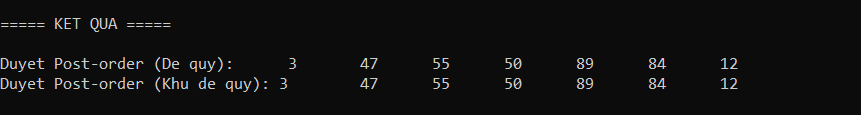
Chương trình cung cấp menu cho phép người dùng lựa chọn phương pháp duyệt cây: sử dụng đệ quy, khử đệ quy, hoặc cả hai.



Hình 4.3: Giao diện màn hình con chọn cách duyệt

* 1. Giao diện màn hình khi duyệt xong

Giao diện hiển thị kết quả của chương trình.



Hình 4.4: Giao diện hiển thị kết quả

# CHƯƠNG 5: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

* 1. Kết luận

Ứng dụng cấu trúc chồng (stack) để khử đệ quy khi duyệt cây nhị phân tìm kiếm đã đạt được những mục tiêu đề ra. Cụ thể, các thuật toán duyệt cây Pre-order, In-order và Post-order đã được thực hiện bằng cả đệ quy và khử đệ quy, đảm bảo tính chính xác và hiệu quả. Việc sử dụng cấu trúc dữ liệu Stack để thay thế đệ quy trong các thuật toán duyệt cây đã khắc phục được hạn chế của phương pháp đệ quy, đặc biệt trong các trường hợp dữ liệu lớn.

* 1. Hạn chế

Cây nhị phân tìm kiếm được xây dựng thủ công, chưa tích hợp các thuật toán tự động cân bằng như trong cây AVL hoặc Red-Black Tree, dẫn đến hiệu suất suy giảm nếu cây bị lệch.

Phương pháp khử đệ quy sử dụng Stack yêu cầu bộ nhớ bổ sung để lưu trạng thái, có thể không phù hợp trong các hệ thống hạn chế bộ nhớ.

* 1. Hướng phát triển

Tích hợp các thuật toán cân bằng cây, như AVL Tree hoặc Red-Black Tree, để đảm bảo độ phức tạp luôn là O(logn) ngay cả trong trường hợp dữ liệu không đều.

Cải tiến thuật toán khử đệ quy để tối ưu hóa bộ nhớ và thời gian thực thi, chẳng hạn sử dụng Stack động hoặc kỹ thuật đánh dấu trạng thái nút.

Nghiên cứu và triển khai các thuật toán khác trên cây nhị phân, chẳng hạn tìm kiếm, xóa phần tử, hoặc tính toán các thuộc tính của cây như chiều cao, số lượng nút, và đường đi dài nhất.

Ứng dụng cây nhị phân tìm kiếm trong các bài toán thực tế, như hệ thống quản lý dữ liệu, xử lý truy vấn, hoặc tìm kiếm thông tin.

DANH MỤC TÀI LIỆU THAM KHẢO

[1] vietjack.com, “Cây tìm kiếm nhị phân trong cấu trúc dữ liệu và giải thuật,” *www.vietjack.com*. <https://vietjack.com/cau-truc-du-lieu-va-giai-thuat/cay-tim-kiem-nhi-phan.jsp>

[2] Đại học trực tuyến FUNiX, “Binary Search Tree (Cây tìm kiếm nhị phân) là gì? | Học trực tuyến CNTT, học lập trình từ cơ bản đến nâng cao,” *Học Trực Tuyến CNTT, Học Lập Trình Từ Cơ Bản Đến Nâng Cao*, May 26, 2023. https://funix.edu.vn/chia-se-kien-thuc/binary-search-tree-cay-tim-kiem-nhi-phan-la-gi/

[3] Đ. An, “Chương 4: STACKS - Lý thuyết cơ bản,” *Viblo*, Jan. 06, 2025. https://viblo.asia/p/chuong-4-stacks-ly-thuyet-co-ban-y3RL1Dd7Jao