**TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT**



**THÀNH PHỐ HỒ CHÍ MINH**

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO CUỐI KÌ  
BỘ MÔN CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

**Giảng viên hướng dẫn: Thầy Trần Đắc Tốt.  
Nhóm : 18  
Gồm các thành viên :   
 Nguyễn Hoàng Anh Kiệt: 23110247  
 Nguyễn Tấn Thành : 23133068  
 Nguyễn Ngọc Huy : 22110332 Lê Đức Việt : 23162113**

Thành Phố Hồ Chí Minh, Tháng 11 Năm 2024

**BẢNG PHÂN CÔNG NHIỆM VỤ TRONG NHÓM**

|  |  |  |  |
| --- | --- | --- | --- |
| **Họ tên** | **MSSV** | **Nhiệm vụ** | **Hoàn thành** |
| Nguyễn Hoàng Anh Kiệt | 23110247 | Bài 1: xây dựng kiểu danh sách Stack và ứng dụng vào bài toán duyệt đồ thị theo chiều sâu  Bài 5: Minh họa cho bài 1 (DFS) | **100%** |
| Nguyễn Tấn Thành | 23133068 | Bài 2: Xây dựng cấu trúc dữ liệu cây nhị phân tìm kiếm cho kiểu cấu trúc dữ liệu tự chọn. Thực hiện các thao tác sau:  + Thêm 1 phần tử vào cây (Dùng đệ quy và không đệ quy)  + Xóa một phần tử khỏi cây (Dùng đệ quy và không đệ quy)  + Tính chiều cao của cây (Chiều cao là số nút trên đường đi dài nhất từ gốc đến lá của cây) (Dùng đệ quy)  Bài 5: Minh họa bằng hình ảnh cho bài 2 | **100%** |
| Nguyễn Ngọc Huy | 22110332 | Viết ứng dụng cho phép người dùng nhập vào một chuỗi của một biểu thức toán học. Sau đó tính giá trị của biểu thức toán này. | **100%** |
| Lê Đức Việt | 23162113 | Thực hiện bài 4 : tạo một từ điển tiếng anh dùng bẳng băm, phục vụ tra cứu từ, thêm, xóa, chỉnh sửa, dùng tệp nhị phân để lưu dữ liệu | **100%** |

**MỤC LỤC**

[**Bài 1: 1**](#_Toc26742)

[1.1. Cấu trúc dữ liệu: 1](#_Toc24847)

[1.2. Ý tưởng bài toán: 1](#_Toc4953)

[**Bài 2 2**](#_Toc14119)

[2.1. Giới thiệu 2](#_Toc12012)

[2.2 Cấu trúc dữ liệu 2](#_Toc9344)

[2.3 Các thao tác trên cây nhị phân tìm kiếm 3](#_Toc12747)

[**BÀI 3 8**](#_Toc30351)

[3.1. Giới thiệu 8](#_Toc21769)

[3.2. Phân tích bài toán 8](#_Toc15789)

[3.3. Thuật toán 8](#_Toc16951)

[3.4. Mã nguồn 9](#_Toc22819)

[3.5. Kết quả 12](#_Toc18161)

[3.6. Kết luận 13](#_Toc17733)

[**Bài 4 : 14**](#_Toc5315)

[4.1. Giới thiệu 14](#_Toc8651)

[4.2. Yêu cầu bài toán 15](#_Toc14173)

[4.3. Thiết kế và triển khai, thực hiện hóa ứng dụng. 17](#_Toc29466)

[4.4. Các hàm chức năng chính 23](#_Toc29499)

[4.5. Xử lý xung đột trong bảng băm 32](#_Toc7869)

[4.6. Kết quả và Đánh giá. 34](#_Toc1718)

[4.7. Hướng phát triển tương lai 39](#_Toc1216)

[4.8. Kết luận 41](#_Toc18549)

[**Bài 5.1: Minh họa cho bài 1 42**](#_Toc25802)

[**Bài 5.2: Minh họa bằng hình ảnh cho bài 2 42**](#_Toc13978)

[5.1 Giới thiệu 42](#_Toc29998)

[5.2 Cấu trúc ứng dụng 5.2.1 Giao diện người dùng 43](#_Toc25628)

[5.3. Kết luận 46](#_Toc833)

**Bài 1:**

* 1. **Cấu trúc dữ liệu:**

+ Stack: gồm mảng lưu giá trị và biến lưu vị trí của đỉnh stack

+ Một số hàm cơ bản gồm:

* Kiểm tra Stack rỗng/đầy: EmptyStack/IsFullStack
* Thêm phần tử vào Stack: Push(Stack,int)
* Lấy giá trị phần tử ở đỉnh Stack: Top(Stack)
* Xóa phần tử ở đỉnh Stack: Pop(Stack)

+ Áp dụng các hàm của Stack vào bài toán duyệt đồ thị theo chiều sâu

* 1. **Ý tưởng bài toán:**

+ Khởi tạo Stack

+ Lấy nút đầu tiên được chỉ định làm root cho vào Stack

While(!EmptyStack):

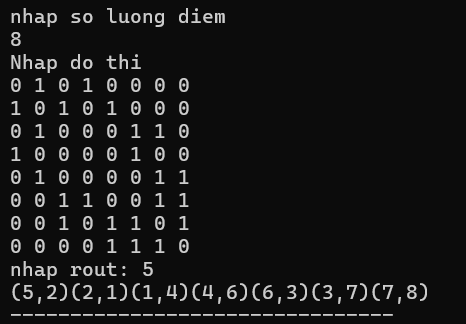
+ Lấy giá trị đỉnh Stack ra xét

+ Tìm kíếm nút kề đầu tiên với nút đang xét theo thứ tự từ nhỏ đến lớn

* Nếu có thì đưa vào Stack, sau đó đánh dấu lại nút đã duyệt, và in ra màn hình kết quả đường đi vừa xét
* Nếu không có thì xóa nút

+ lặp lại cho tới khi Stack rỗng

* Kết quả thu được là đường đi qua hết tất cả các đỉnh trong đồ thị



**Bài 2**

**2.1. Giới thiệu**

Cây nhị phân tìm kiếm (Binary Search Tree - BST) là một cấu trúc dữ liệu cây, trong đó mỗi nút có tối đa hai con (còn gọi là con trái và con phải). BST được thiết kế để hỗ trợ các thao tác tìm kiếm, thêm và xóa một cách hiệu quả.

- Đặc điểm cấu trúc:

+ Nút: Mỗi nút trong cây nhị phân tìm kiếm chứa một giá trị (hoặc khóa) và hai con trỏ (hoặc tham chiếu) đến hai nút con: một nút con bên trái và một nút con bên phải.

+ Gốc: Nút đầu tiên của cây được gọi là nút gốc (root).

+ Cây con: Mỗi nút có thể có 0, 1 hoặc 2 nút con, tạo thành các cây con.

- Tính chất của BST:

+ Thứ tự đối với mỗi nút trong cây

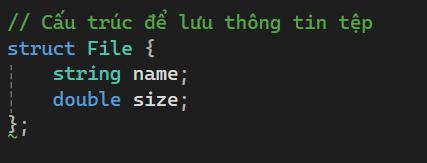
* + Tất cả các giá trị trong cây con bên trái nhỏ hơn giá trị của nút đó.
  + Tất cả các giá trị trong cây con bên phải lớn hơn giá trị của nút đó.

+ Cho phép thực hiện các thao tác tìm kiếm một cách hiệu quả, với độ phức tạp trung bình là O (log n) trong trường hợp cây cân bằng.

**2.2 Cấu trúc dữ liệu**

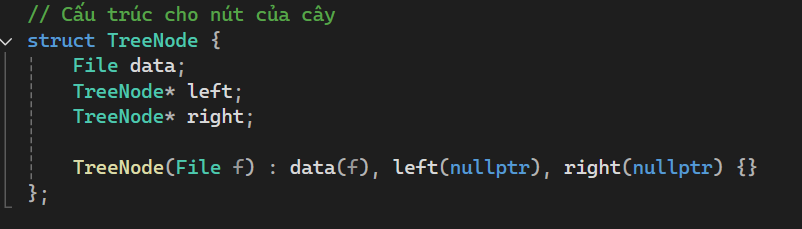
**2.2.1 Cấu trúc tệp**

Cấu trúc *File* được sử dụng để lưu trữ thông tin về một tệp, bao gồm tên và kích thước của tệp.



* + String name: thuộc tính lưu trữ tên của tệp.
  + Double size: thuộc tính lưu trữ kích thước của tệp.

**2.2.2 Cấu trúc nút trong cây**

Cấu trúc *TreeNode* đại diện cho một nút trong cây nhị phân tìm kiếm. Mỗi nút chứa dữ liệu về một tệp và có hai con trỏ để trỏ đến các nút con bên trái và bên phải.

**2.2.3 Cấu trúc cây nhị phân tìm kiếm**

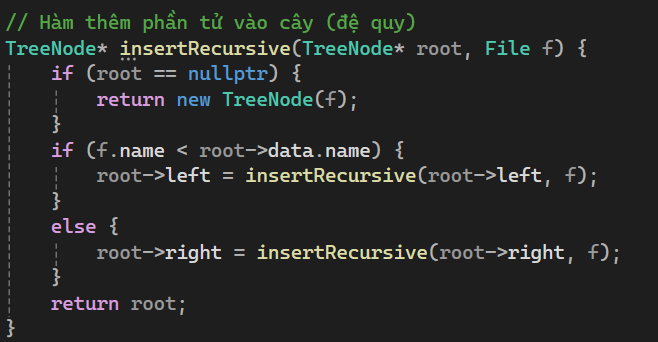
Cây nhị phân tìm kiếm được xây dựng từ các nút *TreeNode*, trong đó nút gốc là nút đầu tiên được thêm vào cây. Cây này duy trì các quy tắc của BST thông qua các thao tác thêm và xóa.

**2.3 Các thao tác trên cây nhị phân tìm kiếm**

**2.3.1 Thêm một phần tử vào cây**

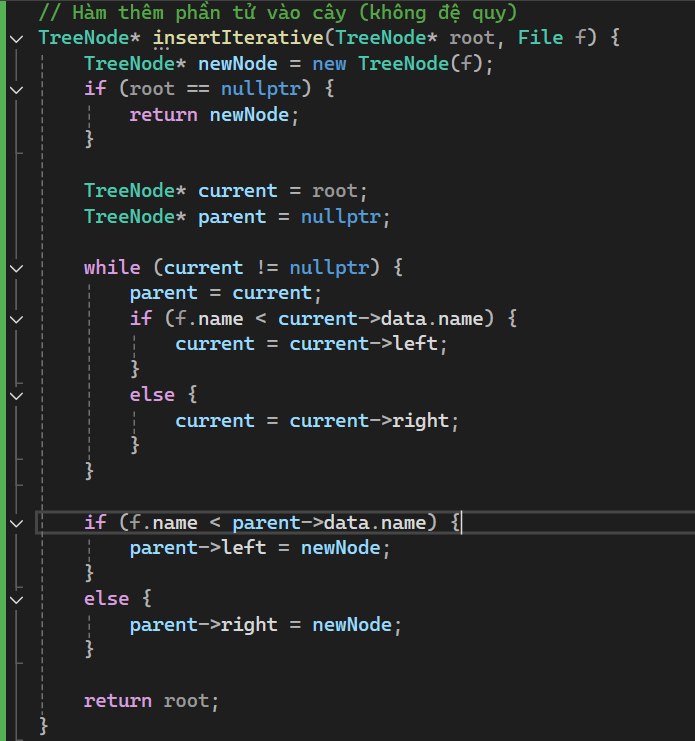
**2.3.1.1 Thêm (đệ quy)**

Hàm *insertRecursive* cho phép thêm một tệp mới vào cây theo cách đệ quy. Nó so sánh tên tệp để xác định vị trí thêm.



**2.3.1.2 Thêm (không đệ quy)**

Hàm *insertIterative* cho phép thêm một tệp mới vào cây theo cách không đệ quy. Nó sử dụng vòng lặp để tìm vị trí thích hợp.

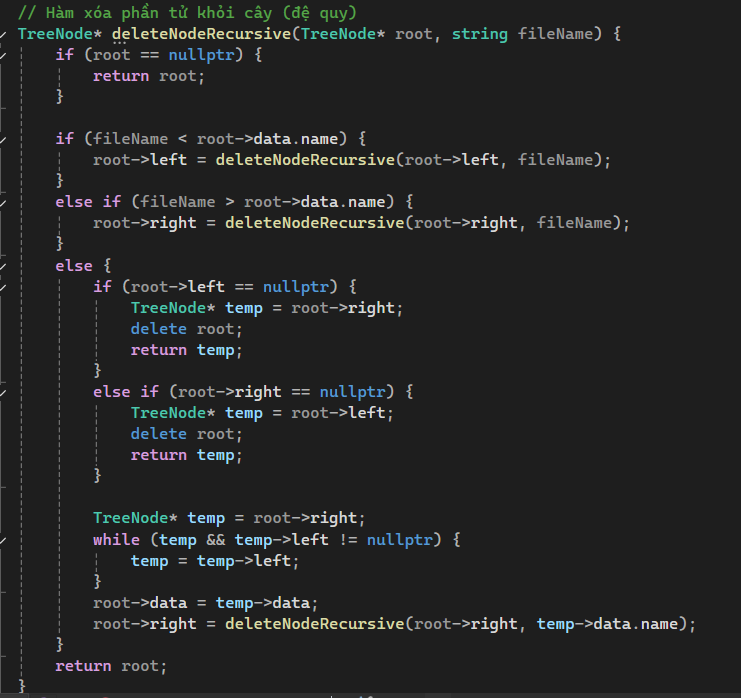


**2.3.2 Xóa một phần tử khỏi trên cây.**

**2.3.2.1 Xóa (đệ quy)**

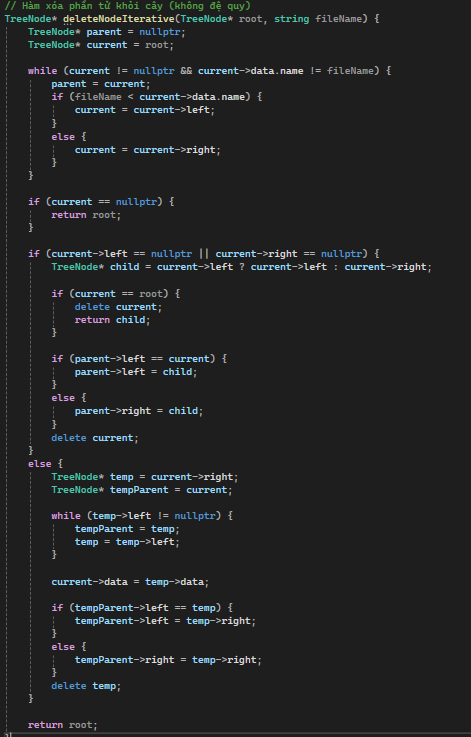
Hàm *deleteNodeRecursive* cho phép xóa một tệp khỏi cây theo cách đệ quy. Nó tìm kiếm tệp cần xóa và xử lý các trường hợp khác nhau:

+ Nếu nút cần xóa có một con, nó sẽ thay thế nút đó bằng con còn lại.

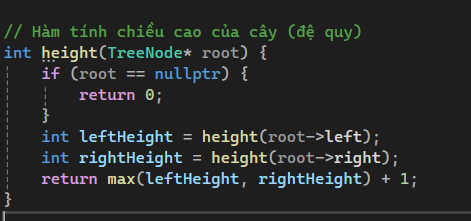
****+ Nếu nút cần xóa có hai con, nó sẽ tìm nút nhỏ nhất trong cây con bên phải (hoặc lớn nhất trong cây con bên trái), sao chép dữ liệu của nút đó vào nút cần xóa, và sau đó xóa nút nhỏ nhất.

**2.3.2.2 Xóa (không đệ quy)**

Hàm *deleteNodeIterative* cho phép xóa một tệp khỏi cây theo cách không đệ quy. Nó tìm kiếm tệp cần xóa và xử lý các trường hợp tương tự như hàm đệ quy, nhưng sử dụng vòng lặp để tìm nút cần xóa và nút cha của nó.



**2.3.3 Tính chiều cao của cây (đệ quy)** Hàm *height* tính chiều cao của cây nhị phân tìm kiếm bằng cách đệ quy. Chiều cao của một cây được xác định là số lượng nút trên đường dài nhất từ nút gốc đến nút lá. Nếu cây rỗng, chiều cao là 0. Nếu không, chiều cao là 1 cộng với chiều cao lớn nhất giữa các cây con bên trái và bên phải**.**



**BÀI 3**

# **3.1. Giới thiệu**

Bài toán yêu cầu viết ứng dụng cho phép người dùng nhập vào một chuỗi biểu thức toán học và tính toán giá trị của biểu thức đó. Biểu thức có thể chứa các toán tử cơ bản như cộng, trừ, nhân, chia, và lũy thừa cùng với các dấu ngoặc đơn để nhóm các toán hạng và phép toán. Để giải quyết bài toán này, chúng ta sử dụng một trong các phương pháp giải quyết biểu thức toán học phổ biến là **thuật toán Balan ngược** (Reverse Polish Notation - RPN).

**Thuật toán Balan ngược** là một phương pháp hiệu quả để đánh giá các biểu thức toán học, trong đó toán tử luôn đứng sau các toán hạng, không cần sử dụng dấu ngoặc để xác định thứ tự ưu tiên. Việc chuyển đổi biểu thức từ dạng **Infix** (chuỗi biểu thức thông thường) sang **Postfix** (chuỗi RPN) giúp việc tính toán trở nên đơn giản và hiệu quả hơn.

# **3.2. Phân tích bài toán**

Biểu thức toán học có thể chứa các số, toán tử (cộng, trừ, nhân, chia, lũy thừa), và dấu ngoặc. Chúng ta cần xử lý biểu thức để tính giá trị cuối cùng đúng với thứ tự ưu tiên của các phép toán.

# **3.3. Thuật toán**

Thuật toán này bao gồm hai bước chính:

**Chuyển đổi từ Infix sang Postfix**: Đây là bước cần thiết để làm giảm độ phức tạp của biểu thức và đảm bảo tính toán theo đúng thứ tự ưu tiên của các toán tử.

**Đánh giá biểu thức Postfix**: Sau khi chuyển đổi xong, biểu thức sẽ được tính toán từ trái qua phải bằng cách sử dụng một Stack (ngăn xếp).

**3.3.1. Chuyển đổi Infix sang Postfix**

Biểu thức Infix có thể chứa các dấu ngoặc và các toán tử với độ ưu tiên khác nhau. Để chuyển đổi biểu thức từ Infix sang Postfix:

Duyệt qua từng ký tự trong biểu thức:

- Nếu gặp **toán hạng** (số), thêm nó vào chuỗi Postfix.

- Nếu gặp **toán tử**, kiểm tra độ ưu tiên của nó so với toán tử trên đỉnh Stack. Nếu toán tử này có độ ưu tiên thấp hơn hoặc bằng toán tử trên đỉnh Stack, pop toán tử từ Stack vào chuỗi Postfix cho đến khi tìm thấy toán tử có độ ưu tiên thấp hơn.

- Nếu gặp **dấu ngoặc mở** (, push nó vào Stack.

- Nếu gặp **dấu ngoặc đóng** ), pop các toán tử từ Stack vào chuỗi Postfix cho đến khi gặp dấu ngoặc mở (, sau đó pop dấu ngoặc này.

**3.3.2. Đánh giá biểu thức Postfix**

Biểu thức Postfix không cần dấu ngoặc vì thứ tự các toán tử đã được xác định rõ ràng. Để tính toán biểu thức Postfix:

Duyệt qua từng ký tự:

- Nếu là **toán hạng** (số), push nó vào Stack.

- Nếu là **toán tử**, pop hai toán hạng từ Stack, thực hiện phép toán và push kết quả vào Stack.

Cuối cùng, kết quả cuối cùng sẽ là giá trị còn lại trên đỉnh Stack.

# **3.4. Mã nguồn**

Chương trình bao gồm các hàm chính sau:

***Kiểm tra tính hợp lệ của biểu thức***

******

***Chuyển đổi Infix sang Postfix***

******

***Tính toán giá trị Postfix***

******

# **3.5. Kết quả**

**Đúng đắn**: Chương trình đã có thể chuyển đổi biểu thức từ dạng Infix sang Postfix một cách chính xác và thực hiện tính toán chính xác theo thứ tự ưu tiên.

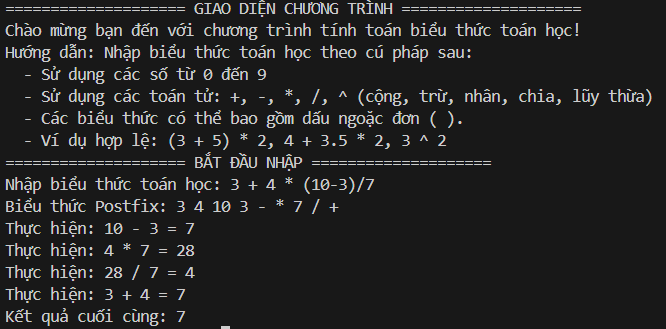
**Lỗi xử lý**: Trong trường hợp biểu thức nhập vào không hợp lệ (ví dụ: chia cho 0 hoặc biểu thức không đầy đủ), chương trình sẽ thông báo lỗi và yêu cầu người dùng nhập lại.

Mặc dù chương trình đã giải quyết được bài toán cơ bản, nhưng có thể mở rộng thêm một số tính năng như:

**Hỗ trợ các toán tử mở rộng**: Ví dụ, hàm lượng giác, căn bậc hai, hoặc các hàm toán học khác.

**Giao diện người dùng**: Có thể phát triển giao diện đồ họa để người dùng dễ dàng nhập biểu thức và nhận kết quả trực quan hơn.

VÍ DỤ:



# **3.6. Kết luận**

Chương trình đã thành công trong việc giải quyết bài toán tính giá trị của một biểu thức toán học, sử dụng thuật toán **Balan ngược** để chuyển đổi biểu thức từ dạng Infix sang Postfix và sau đó tính toán giá trị của biểu thức Postfix bằng cách sử dụng Stack. Thuật toán này giúp đơn giản hóa quá trình tính toán và giảm độ phức tạp khi xử lý các biểu thức toán học phức tạp. Chương trình có thể mở rộng thêm để xử lý các biểu thức với các toán tử phức tạp hơn hoặc hỗ trợ các tính năng khác.

**Bài 4 :**

**4.1. Giới thiệu**

**Đề bài :** Viết một phần mềm từ điển tiếng Anh đơn giản (không cần giao diện đồ họa).

- Sử dụng bảng băm

- Dữ liệu có thể được lưu trữ trên tập tin văn bản hoặc nhị phân.

Ví dụ: Nhập vào: school

Kết quả: Trường học (danh từ)

## **4.1.1. Mục tiêu của bài toán**

Xây dựng một phần mềm từ điển tiếng Anh đơn giản với các tính năng tra cứu từ, thêm, xóa, chỉnh sửa từ, lưu trữ dữ liệu bằng bảng băm, và sử dụng tệp nhị phân cho việc lưu trữ dữ liệu từ điển.

## **4.1.2. Phạm vi và yêu cầu của bài toán**

Phạm vi của bài toán: nhằm tạo ra một phần mềm từ điển tiếng Anh đơn giản, dễ sử dụng, không yêu cầu giao diện đồ họa, và có khả năng lưu trữ dữ liệu hiệu quả bằng bảng băm và tệp nhị phân.   
 Các chức năng cơ bản như tra cứu, thêm, sửa và xóa từ sẽ được thực hiện trong môi trường dòng lệnh, giúp người dùng dễ dàng thao tác mà không gặp phải sự phức tạp của giao diện đồ họa.  
 Lưu trữ dữ liệu trên file (.bin) - một tệp lưu trữ dữ liệu dưới dạng nhị phân, thay vì dạng văn bản như các tệp .txt. Dữ liệu nhị phân là các byte (8-bit), cho phép bạn lưu trữ thông tin ở bất kỳ định dạng nào (chữ cái, số nguyên, số thực, hình ảnh, video, âm thanh, v.v.) mà không cần chuyển đổi thành ký tự văn bản.

# **4.2. Yêu cầu bài toán**

## **4.2.1. Chức năng chính của phần mềm**

**Tra cứu từ**: Cho phép người dùng nhập vào một từ tiếng Anh và hiển thị nghĩa và từ loại của từ đó. dữ liệu từ điển ban đầu có gần 65000 từ được lưu trữ và xuất ra thông tin đáp ứng nhu càu người dùng,

**Thêm, xóa từ mới**: Cung cấp chức năng cho người dùng (người tạo ra dữ liệu thông tin từ điển) thêm từ mới vào từ điển, xóa từ ra khỏi từ điển.

**Chỉnh sửa từ trong từ điển** : Cho phép người dùng (người tạo ra dữ liệu thông tin từ điển) có thể chỉnh sửa hoặc bổ sung thêm loại từ và ngữ nghĩa cho từ đã có trong từ điển.

**Lưu trữ và khôi phục dữ liệu**: Sử dụng bảng băm trong bộ nhớ hoặc tập tin nhị phân để lưu trữ dữ liệu từ điển, nhằm sử dụng lâu dài.

## **4.2.2. Cấu trúc dữ liệu sử dụng**

Bài toán yêu cầu sử dụng bảng băm để lưu trữ và tra cứu các từ trong từ điển. Cấu trúc dữ liệu chủ yếu là cấu trúc Word (để lưu từ và nghĩa) và bảng băm với các danh sách liên kết xử lý va chạm. Dữ liệu có thể được lưu trữ và đọc từ tệp nhị phân để tái sử dụng, giúp phần mềm hoạt động hiệu quả.

**Cấu trúc Word**

Cấu trúc Word sẽ đại diện cho một từ và nghĩa của từ đó trong từ điển. Cấu trúc này bao gồm 3 trường:

- word: Từ tiếng Anh (kiểu string).

- type : Loại từ cảu từ (kiểu string)

- meaning: Nghĩa của từ (kiểu string).

**Bảng băm (Hash Table)**

Bảng băm sẽ lưu trữ tất cả các từ trong từ điển. Mỗi từ sẽ được ánh xạ tới một chỉ số trong bảng băm thông qua hàm băm. Trong bài toán này, sử dụng một mảng các danh sách liên kết (linked list) để xử lý các va chạm (collisions) trong bảng băm. (phương pháp nối tiếp).

**Hàm Băm**

Hàm băm sẽ ánh xạ từ khóa (từ tiếng Anh) sang một chỉ số trong bảng băm. Một hàm băm đơn giản có thể là tổng các mã ASCII của các ký tự trong từ tích với chỉ số của nó trong từ đó (từ add qua hàm băm và cho chỉ số bằng 65 \* 1 + 68 \* 2 + 68 \* 3), và sau đó lấy modulo với kích thước của bảng băm.

**Quản lý Bảng Băm**

Bảng băm sẽ là một mảng với các chỉ mục là kết quả của hàm băm, và mỗi chỉ mục sẽ trỏ tới một danh sách liên kết chứa các đối tượng Word. Khi gặp va chạm (nhiều từ có cùng giá trị băm), chúng ta sẽ thêm các mục vào danh sách liên kết tại chỉ mục tương ứng.

**4.2.3. Lưu trữ dữ liệu**

**Lưu trữ trên tệp nhị phân**

Dữ liệu sẽ được lưu trữ vào tệp nhị phân để khi khởi động lại chương trình, dữ liệu có thể được khôi phục và sử dụng lại.

Tệp nhị phân giúp tiết kiệm bộ nhớ hơn so với việc sử dụng tệp văn bản, cho hiệu suất cao hơn khi thao tác với file dữ liệu, dữ liệu được bảo mật, mã hóa.

# **4.3. Thiết kế và triển khai, thực hiện hóa ứng dụng.**

## **4.3.1. Các cấu trúc dữ liệu sử dụng**

### **4.3.1.1. Cấu trúc Word :**

Trong mỗi Word chứa 3 trường :

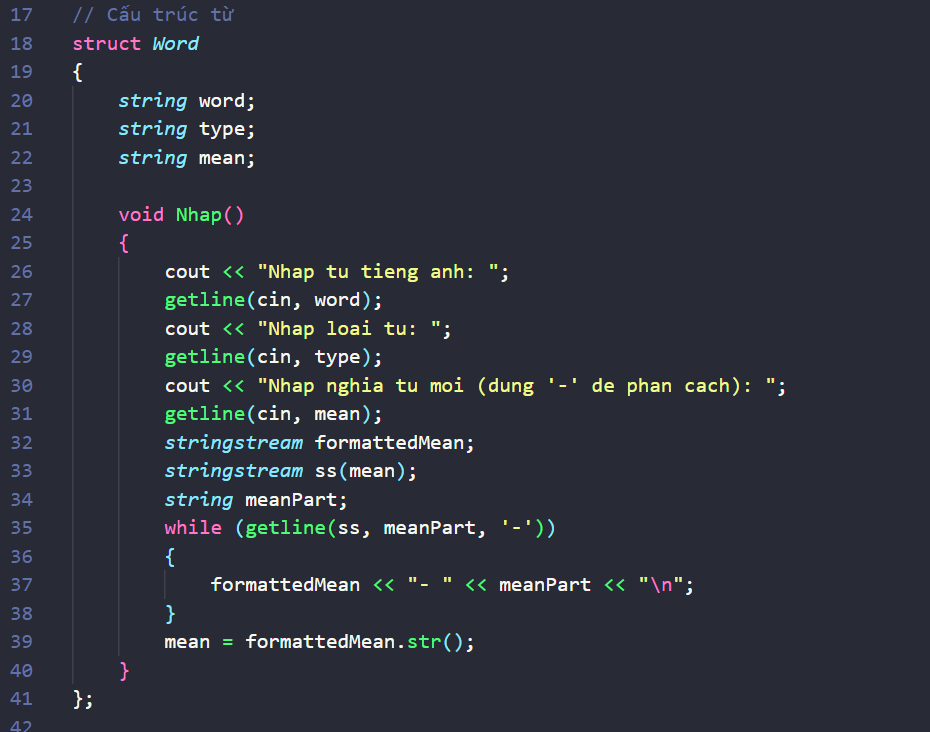
+ word : (string) - biến này dùng để lưu trữ từ tiếng anh trong cấu trúc Word

+ type : (string) - biến này dùng để lưu loại từ của từ đó, bao gồm noun (danh từ), verb (động từ), adjective (tính từ), adv (trạng từ), ...

+ mean : (string) - biến này dùng để lưu trữ phần nghĩa của từ đó, nhiều nghĩa cách nhau bởi dấu ‘-’.

+ Hàm nhap() : giúp cho người dùng thực hiện các thao tác nhập thông tin cho Word này dễ dàng, nhanh chóng, tiện lợi.

Các bucket trong bảng băm sẽ lưu trữ các Word này, dữ liệu sẽ được đóng gói và dễ lưu trữ, truy xuất dễ dàng hơn.

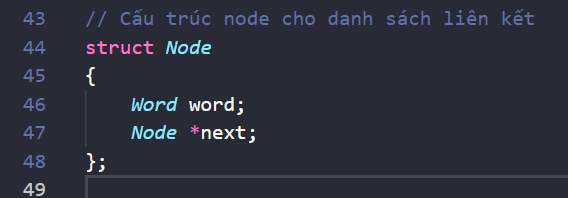


Hình 4.1 : Thiết kế cấu trúc Word.

### **4.3.1.2. Bảng băm (hash table)**

Bảng băm (Hash Table) là một cấu trúc dữ liệu được sử dụng để lưu trữ và truy xuất dữ liệu một cách hiệu quả, với thời gian truy xuất trung bình là O(1). Bảng băm trong chương trình này được triển khai để lưu trữ từ điển, trong đó các từ được băm và phân vào các bucket. Mỗi bucket được quản lý bởi một danh sách liên kết (linked list) nhằm xử lý xung đột.

**Cấu trúc Node**



Hình 2.2 : Cấu trúc Node

- Mục đích: Cấu trúc Node đại diện cho một nút trong danh sách liên kết, lưu trữ một từ trong từ điển.

- Thành phần:

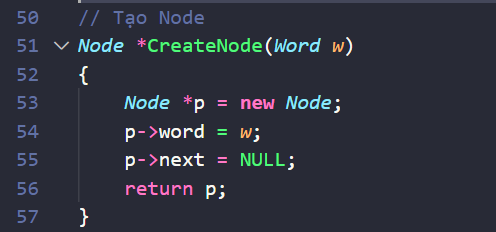
word: Là dữ liệu chính, chứa thông tin từ vựng (bao gồm từ tiếng Anh, loại từ, và nghĩa).

next: Là con trỏ trỏ đến nút tiếp theo trong danh sách liên kết. Nếu next == NULL, đó là nút cuối của danh sách.

- Vai trò trong bảng băm:

Mỗi bucket trong bảng băm là một danh sách liên kết, chứa nhiều Node để xử lý các trường hợp xung đột khi nhiều từ được băm vào cùng một vị trí.

**Hàm CreateNode**



Hình 2.3 : Hàm tạo node trong danh sách liên kết đơn

- Mục đích: Tạo một nút mới chứa thông tin về một từ.

- Các bước thực hiện:

Khởi tạo một con trỏ Node bằng cách cấp phát động bộ nhớ (new Node).

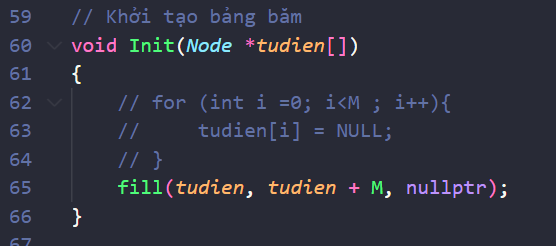
Gán dữ liệu w vào thành phần word của nút.

Thiết lập con trỏ next trỏ đến NULL, vì nút này ban đầu chưa liên kết với nút nào khác.

- Ứng dụng:

Dùng để thêm từ mới vào bảng băm khi cần.

**Hàm Init**



Hình 2.4 : Hàm Init

- Mục đích:

Khởi tạo bảng băm với tất cả các bucket đều rỗng (tức là tất cả các phần tử của mảng tudien[] được đặt là nullptr).

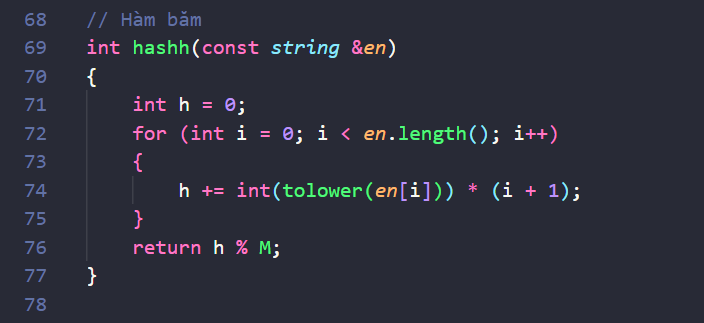
- Cách hoạt động:

Dùng hàm fill() của C++ để điền giá trị nullptr vào toàn bộ mảng con trỏ tudien[].

- Vai trò:

Đảm bảo rằng trước khi sử dụng bảng băm, tất cả các bucket đều được khởi tạo là rỗng.

**Hàm băm (hashh)**



Hình 2.5 : Hàm băm

- Mục đích:

Chuyển một từ tiếng Anh (en) thành một chỉ số nguyên (index) để xác định vị trí của từ đó trong bảng băm.

- Cách hoạt động:

Duyệt qua từng ký tự của từ (en).

Tính tổng trọng số của các ký tự, trong đó trọng số là mã ASCII của ký tự nhân với vị trí của nó trong chuỗi.

Lấy kết quả tổng đó chia dư cho kích thước bảng băm M để đảm bảo giá trị trả về nằm trong phạm vi hợp lệ [0,M−1]

- Ứng dụng:

Dùng để xác định bucket mà từ cần lưu trữ trong bảng băm.

- Xử lý xung đột:

Trong trường hợp nhiều từ có cùng giá trị băm (tức là cùng chỉ số bucket), các từ sẽ được thêm vào danh sách liên kết của bucket đó.

**Đặc điểm của bảng băm trong chương trình**

Kích thước cố định: Bảng băm có kích thước M cố định (ví dụ: 100 hoặc một số nguyên tố lớn). Điều này giúp giảm thiểu xung đột khi phân bố từ vào các bucket.

Giải quyết xung đột:

Sử dụng phương pháp danh sách liên kết. Khi nhiều từ có cùng giá trị băm, chúng được lưu trữ dưới dạng danh sách liên kết trong cùng một bucket.

Hiệu quả:

Hàm băm được tối ưu hóa bằng cách kết hợp mã ASCII và vị trí ký tự, giảm thiểu xung đột.

Việc tra cứu, thêm, và xóa từ nhanh chóng với độ phức tạp trung bình là O(1) khi bảng băm ít xung đột.

**Tổng quan vai trò của các hàm**

CreateNode: Tạo nút mới để thêm vào danh sách liên kết.

Init: Khởi tạo bảng băm rỗng để chuẩn bị sử dụng.

hashh: Xác định vị trí bucket dựa trên giá trị băm của từ.

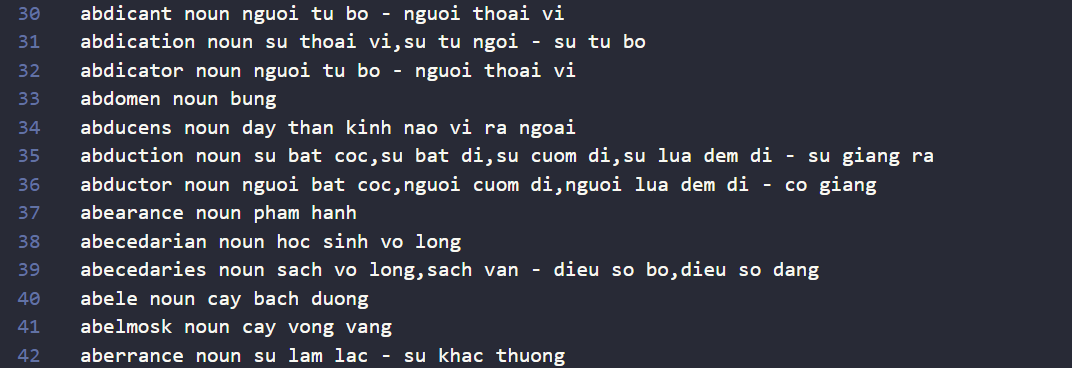
Kết hợp các hàm trên giúp bảng băm hoạt động như một công cụ lưu trữ và truy xuất dữ liệu hiệu quả.

## **4.4. Các hàm chức năng chính**

### **4.4.1. Hàm loadFromTextFile**

- Mục đích:

Tải dữ liệu từ một file văn bản (.txt) vào bảng băm. File chứa các từ tiếng Anh cùng loại từ và nghĩa được ghi trên 1 dòng.



Hình 2.6 : Cấu trúc của file .txt lưu trữ các từ vựng trong từ điển

- Cách hoạt động:

Mở file .txt bằng luồng ifstream. Nếu không mở được, thông báo lỗi và thoát hàm.

Đọc từng dòng trong file, mỗi dòng chứa dữ liệu dạng: word type mean.

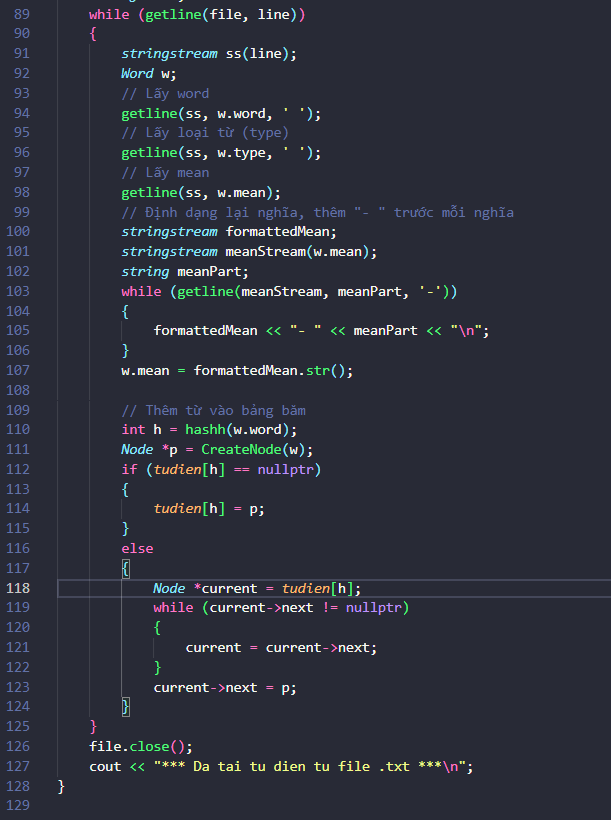
Sử dụng stringstream để phân tách dòng thành ba thành phần: từ vựng, loại từ, và nghĩa.

Định dạng lại nghĩa, thêm ký hiệu - trước mỗi phần nghĩa để hiển thị đẹp hơn.

Băm từ vựng bằng hàm hashh để tìm vị trí bucket.

Thêm từ mới vào danh sách liên kết trong bucket tương ứng.

Sau khi hoàn thành, đóng file và thông báo đã tải xong từ điển.



Hình 2.7 : Hàm loadFromTextFile

### **4.4.2. Hàm writeWordToFile**

- Mục đích:

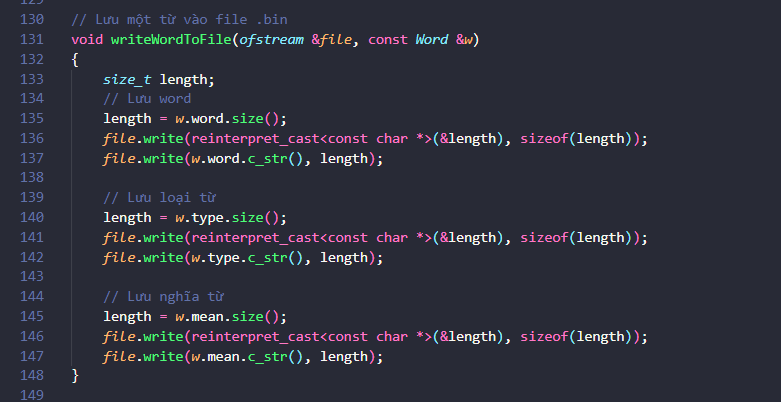
Lưu một từ (Word) vào file nhị phân.

- Cách hoạt động:

Ghi độ dài chuỗi của từng thành phần (word, type, mean) vào file để xác định kích thước dữ liệu.

Ghi nội dung của word, type, và mean vào file nhị phân theo thứ tự.

Đảm bảo dữ liệu được lưu trữ có cấu trúc và có thể đọc lại dễ dàng.



Hình 2.8 : hàm writeWordToFile

### **4.4.3. Hàm saveHashTableToBinaryFile**

- Mục đích:

Lưu toàn bộ bảng băm vào file nhị phân để duy trì dữ liệu lâu dài.

- Cách hoạt động:

Mở file nhị phân bằng luồng ofstream với chế độ ghi (ios::binary). Nếu không mở được, thông báo lỗi và thoát hàm.

Duyệt qua tất cả các bucket trong bảng băm.

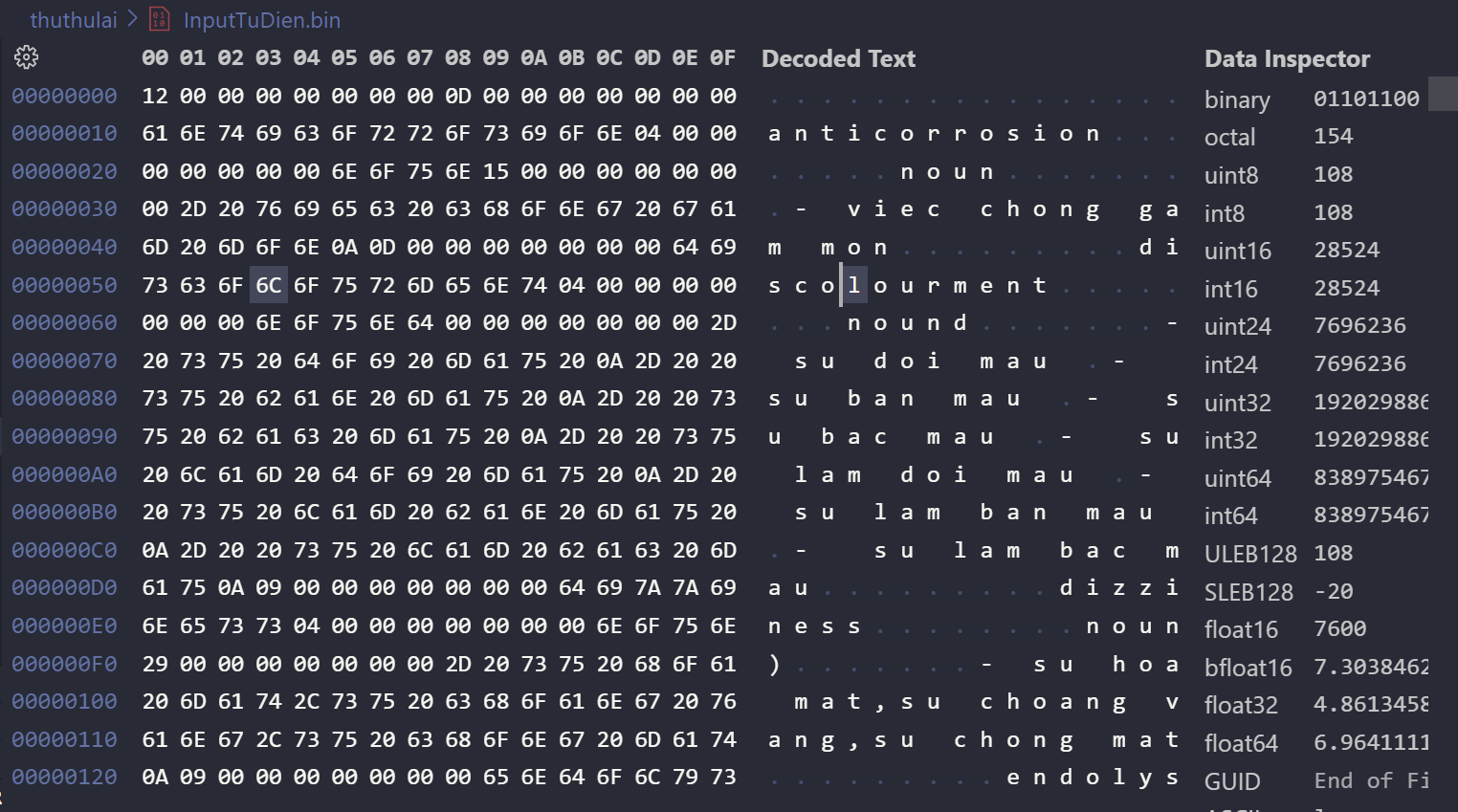
Ghi số lượng từ trong mỗi bucket vào file.

Ghi tuần tự từng từ trong bucket bằng cách sử dụng hàm writeWordToFile.

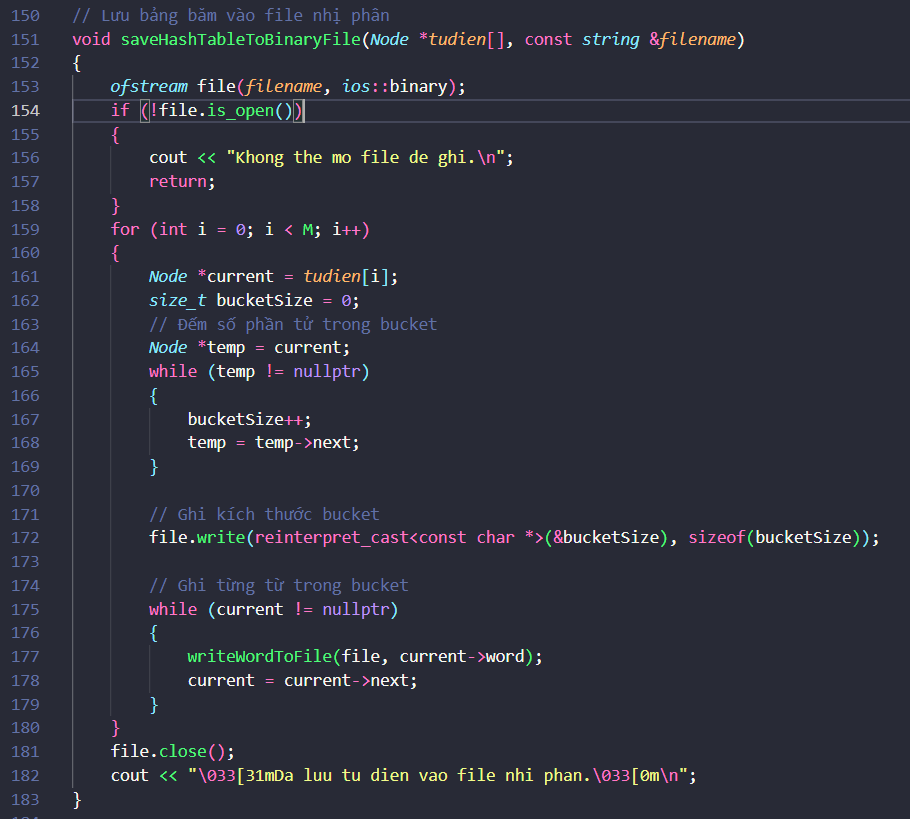
Sau khi hoàn thành, đóng file và thông báo thành công.

Dữ liệu sau khi được lưu vào file nhị phân chỉ toàn là các kí tự 0, 1

Chúng ta cần dùng một số công cụ đặc biệt mới có thể mở và xem, chỉnh sửa dữ liệu trong file. Dưới đây là hình ảnh file InPutTuDien.bin được mở với công cụ Hex Editor - extension trên trình soạn thảo code VS code.



Hình 2.8 : Dữ liệu sau khi được lưu vào tệp nhị phân

  
Hình 2.9 : Hàm saveHashTableToBinaryFile

### **4.4.4. Hàm loadHashTableFromBinaryFile**

- Mục đích:

Đọc dữ liệu bảng băm từ file nhị phân để tái sử dụng từ điển.

- Cách hoạt động:

Mở file nhị phân bằng luồng ifstream với chế độ đọc (ios::binary). Nếu không mở được, thông báo lỗi và thoát hàm.

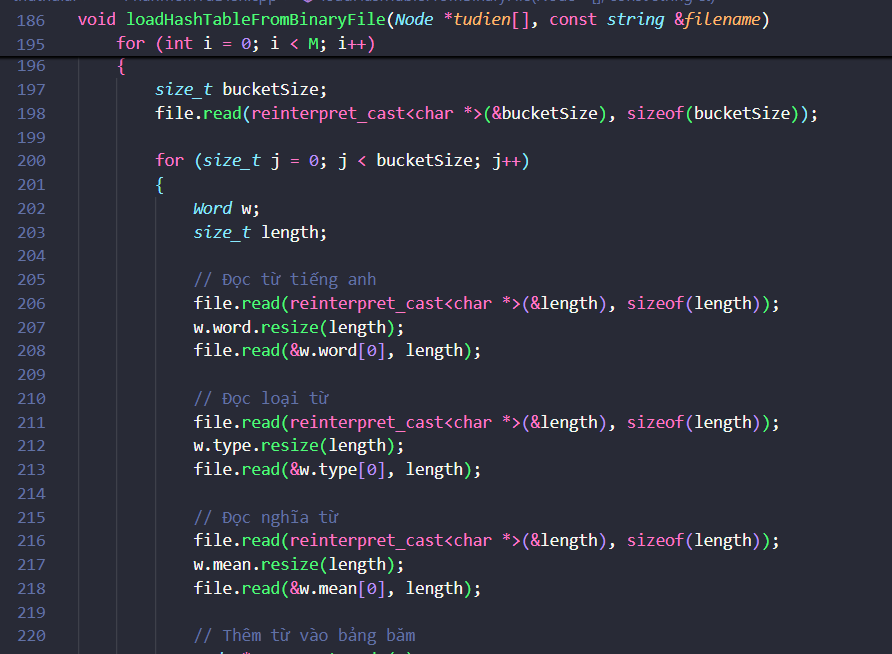
Duyệt qua từng bucket trong file:

Đọc số lượng từ trong bucket.

Đọc dữ liệu của từng từ (word, type, mean) từ file nhị phân.

Tạo các Node mới từ dữ liệu vừa đọc và thêm chúng vào bucket tương ứng.

Đóng file sau khi hoàn thành và thông báo đã tải thành công.

  
Hình 2.10 : Hàm loadHashTableFromBinaryFile

### **4.4.5. Hàm addWord**

- Mục đích:

Thêm một từ mới vào bảng băm và lưu cập nhật vào file nhị phân.

- Cách hoạt động:

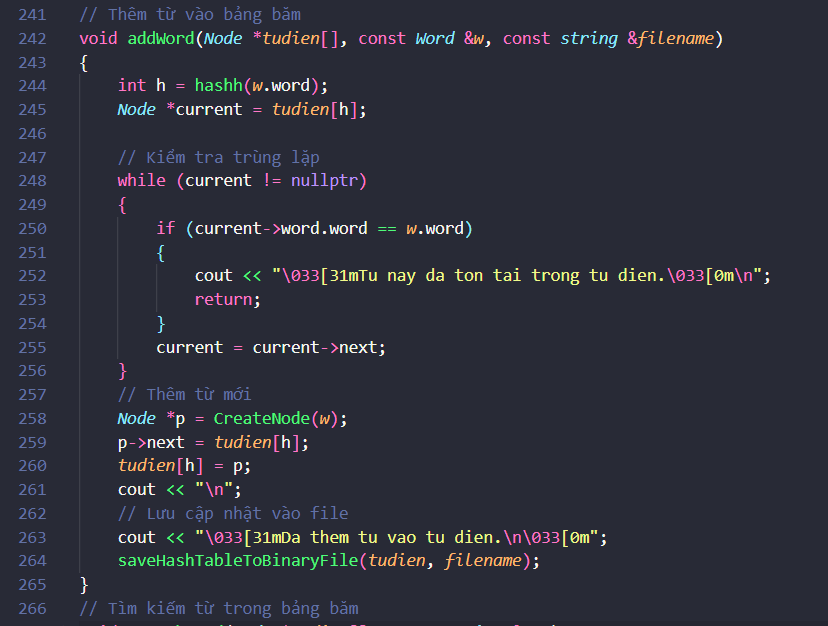
Băm từ mới để xác định bucket.

Kiểm tra trùng lặp: Duyệt qua danh sách liên kết trong bucket. Nếu từ đã tồn tại, thông báo và thoát hàm.

Nếu từ đó chưa tồn tại trong từ điển thì tạo một Node mới chứa từ cần thêm.

Chèn nút mới vào đầu danh sách liên kết trong bucket.

Gọi hàm saveHashTableToBinaryFile để lưu cập nhật vào file nhị phân.



Hình 2.11 : Hàm addWord

### **4.4.6. Hàm searchWord**

- Mục đích:

Tìm kiếm một từ trong bảng băm và hiển thị thông tin của từ.

- Cách hoạt động:

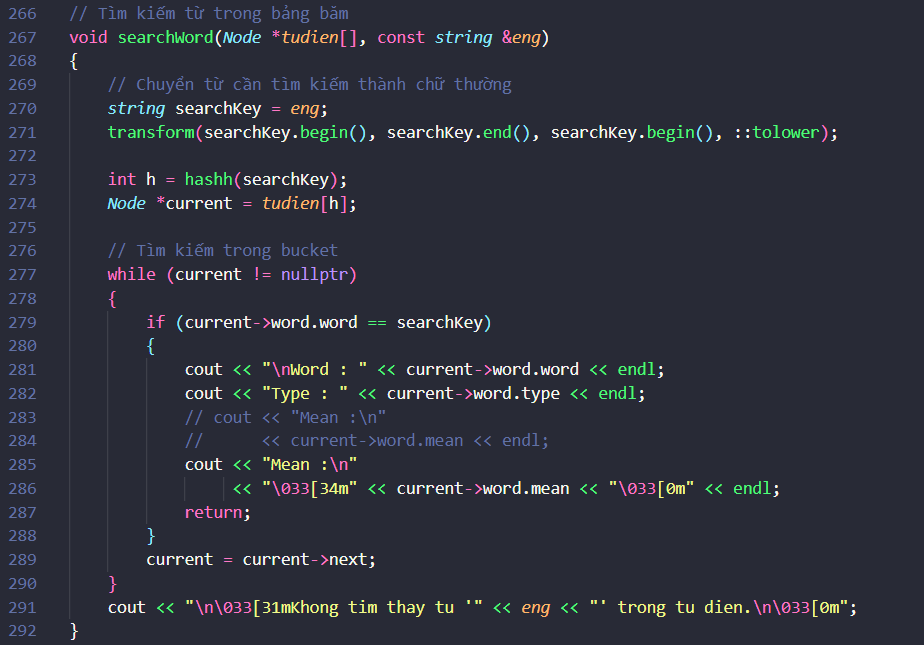
Chuyển từ cần tìm kiếm sang chữ thường để tránh phân biệt hoa/thường.

Băm từ để xác định bucket.

Duyệt qua danh sách liên kết trong bucket:

Nếu tìm thấy từ, hiển thị thông tin của từ (word, type, mean).

Nếu không tìm thấy, thông báo từ không tồn tại trong từ điển.



Hình 2.12 : hàm searchWord

### **4.4.7. Hàm deleteWord**

- Mục đích:

Xóa một từ khỏi bảng băm và lưu cập nhật vào file nhị phân.

- Cách hoạt động:

Chuyển từ cần xóa sang chữ thường.

Băm từ để xác định bucket.

Duyệt qua danh sách liên kết trong bucket:

Nếu tìm thấy từ: Cập nhật liên kết để bỏ qua nút chứa từ cần xóa.

Giải phóng bộ nhớ của nút đó.

Gọi hàm saveHashTableToBinaryFile để lưu cập nhật.

Thông báo đã xóa thành công và thoát hàm.

Nếu không tìm thấy, thông báo từ không tồn tại.



Hình 2.13 : Hàm deleteWord

### **4.4.8. Hàm editWord**

- Mục đích:

Chỉnh sửa thông tin của một từ trong bảng băm.

- Cách hoạt động:

Chuyển từ cần chỉnh sửa sang chữ thường.

Băm từ để xác định bucket.

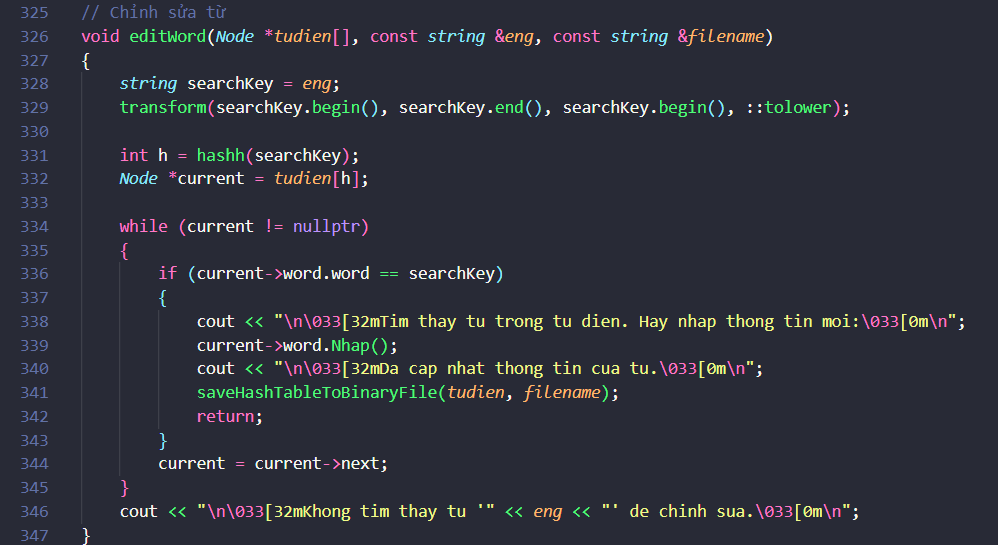
Duyệt qua danh sách liên kết trong bucket:

Nếu tìm thấy từ: Gọi phương thức Nhap để nhập thông tin mới cho từ.

Gọi hàm saveHashTableToBinaryFile để lưu cập nhật.

Thông báo đã chỉnh sửa thành công.

Nếu không tìm thấy, thông báo từ không tồn tại.



Hình 2.14 : Hàm editWord

## **4.5. Xử lý xung đột trong bảng băm**

Việc xảy ra xung đột trong trường hợp này là điều hoàn toàn có thể xảy ra, với số lượng từ lớn (gần 65000 từ) với kích thước của bảng băm là M = 5011 bucket tương đối nhỏ.

Xung đột (collision) trong bảng băm xảy ra khi hai hoặc nhiều từ tiếng Anh có cùng giá trị băm và được ánh xạ vào cùng một bucket trong bảng băm. Để đảm bảo dữ liệu được lưu trữ và truy cập chính xác, cần sử dụng một cơ chế xử lý xung đột.

Trong bài toán từ điển này, cơ chế được sử dụng để xử lý xung đột là danh sách liên kết (chaining). Dưới đây là chi tiết cách xử lý xung đột:

**Cơ chế danh sách liên kết**

-Mô tả:  
 Mỗi phần tử của bảng băm là một danh sách liên kết. Khi xảy ra xung đột, từ mới được thêm vào danh sách liên kết tương ứng của bucket đó.

- Hoạt động:

+ Khi thêm một từ mới:

Tính giá trị băm của từ bằng hàm hashh().

Nếu bucket tương ứng trống (nullptr), tạo một node mới và đặt vào bucket.

Nếu bucket đã chứa node (xung đột), thêm node mới vào cuối hoặc đầu danh sách liên kết.

+ Khi tìm kiếm:

Tính giá trị băm của từ.

Duyệt qua danh sách liên kết trong bucket tương ứng để tìm từ.

+ Khi xóa hoặc chỉnh sửa:

Tương tự, duyệt danh sách liên kết trong bucket để tìm và thực hiện thao tác.

- Ví dụ minh họa :

Giả sử bảng băm có kích thước M = 10, và hàm băm được tính bằng cách lấy mã ASCII của ký tự đầu tiên của từ chia dư cho M.

Thêm "apple":

Giá trị băm: hashh("apple") = 7.

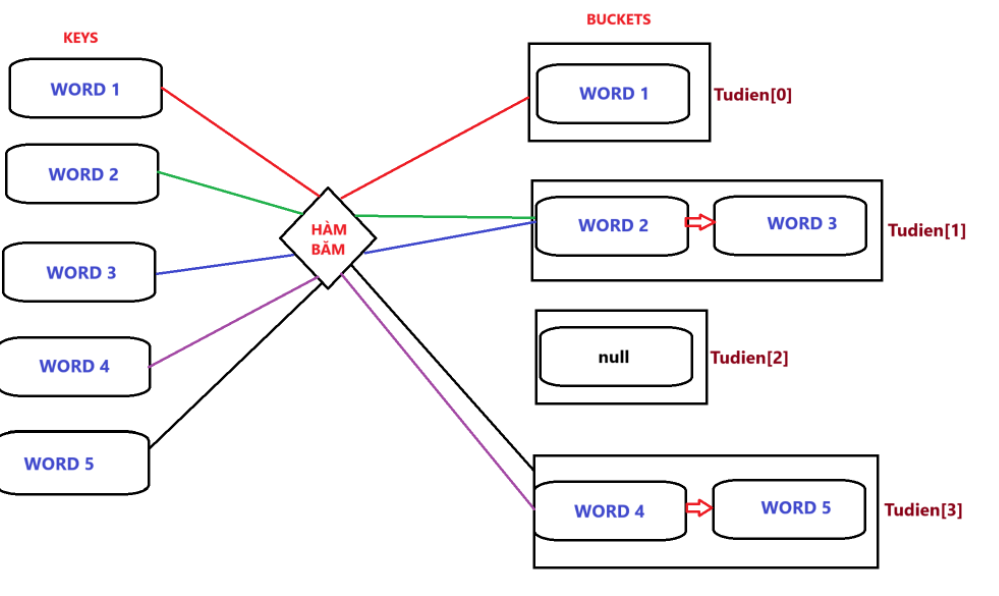
Bucket 7 trống, thêm node chứa "apple".

Thêm "ant":

Giá trị băm: hashh("ant") = 7.

Bucket 7 đã chứa "apple" → thêm "ant" vào danh sách liên kết của bucket 7.

- Dưới đây là mô tả sơ lược về cách lưu dữ liêu khi gặp xung đột.



Hình 2.15 : Mô phỏng hệ thống lưu trữ trên bảng băm

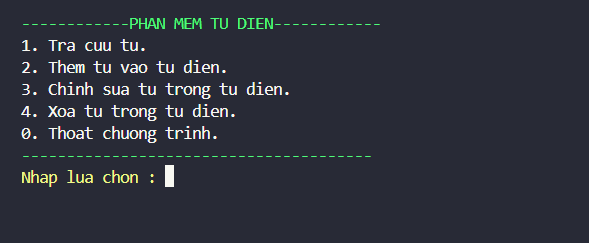
# **4.6. Kết quả và Đánh giá****.**

## **4.6.1.Kết quả đạt được**

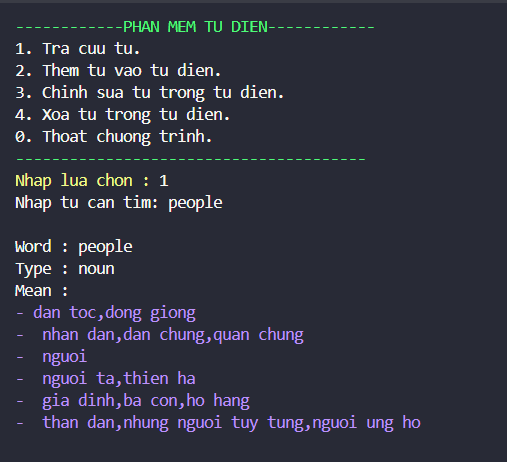
Chương trình chạy trên màn hình terminal, gọn gàng dễ sử dụng, đáp ứng yêu cầu từ đề bài : thực hiện chức năng tìm kiếm, thêm, chỉnh sửa, xóa từ.

Giao diện đơn giản, dễ sử dụng, sử dụng các thư viện ngoài như conio.h, chrono.h, thread.h nhằm tạo thêm màu sắc, hiệu ứng cho ứng dụng.

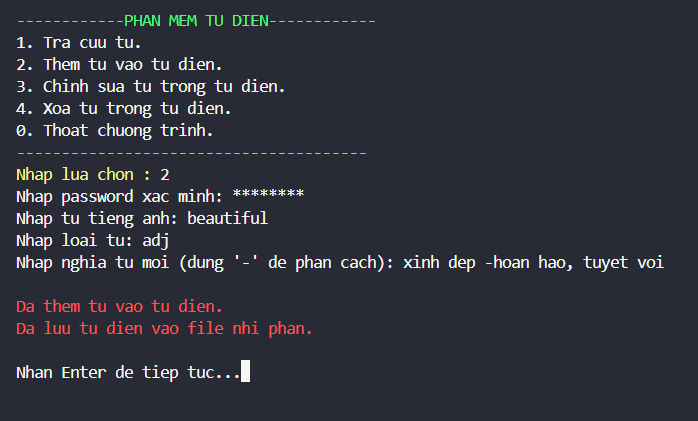




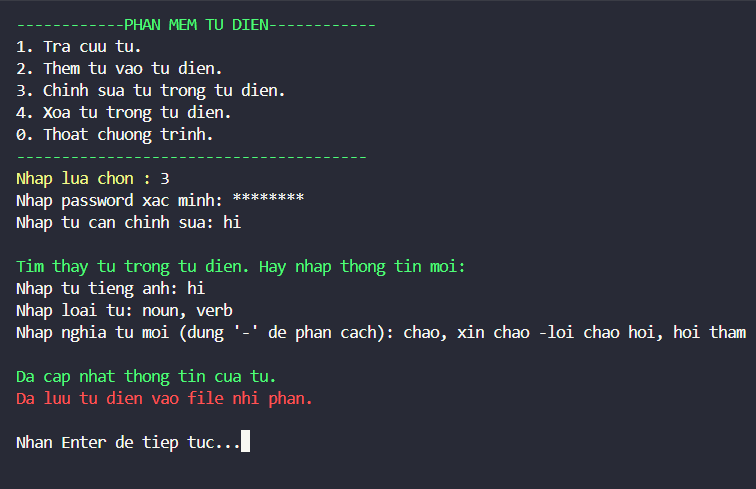
Hình 2.16 : Màn hình hiển thị các chức năng cơ bản của chương trình



Hình 2.17 : Giao diện khi thực hiện chức năng tìm kiếm



Hình 2.18 : Giao diện khi thực hiện chức năng thêm từ vào từ điển



Hình 2.19 : Giao diện khi thực hiện chức năng chỉnh sửa từ trong từ điển



Hình 2.20 : Giao diện khi thực hiện chức năng xóa từ khỏi từ điển

## **4.6.2. Đánh giá hiệu suất, độ ổn định chương trình.**

Hiệu suất của chương trình được đánh giá dựa trên các yếu tố sau:

**Thời gian thực thi**

- Thêm từ vào từ điển:

Thời gian thực hiện tỷ lệ thuận với chiều dài danh sách liên kết của bucket. Trong trường hợp xấu nhất (tất cả từ được ánh xạ vào cùng một bucket), thời gian thêm là O(n), với nnn là số từ trong bucket.

Trong trường hợp trung bình, với hàm băm phân phối đều, thời gian thêm là O(1)

- Tìm kiếm từ:

Thời gian tìm kiếm tương tự như thêm, O(n) trong trường hợp xấu nhất và O(1) trong trường hợp trung bình.

- Xóa từ:

Thời gian xóa tương tự tìm kiếm, do phải tìm vị trí của từ trước khi xóa.

- Tải dữ liệu từ file:

Tốc độ phụ thuộc vào kích thước file dữ liệu và cách lưu trữ:

File nhị phân (.bin): Nhanh hơn do lưu dữ liệu dạng cấu trúc.

**Sử dụng bộ nhớ**

- Cấu trúc bảng băm:

Bộ nhớ sử dụng tỷ lệ thuận với kích thước bảng băm MMM và số từ trong danh sách liên kết.

Cần thêm bộ nhớ cho danh sách liên kết tại mỗi bucket.

- Lưu trữ dữ liệu:

File nhị phân sử dụng bộ nhớ tối ưu hơn file văn bản do lưu trực tiếp các cấu trúc dữ liệu.

**Độ phức tạp của chương trình**

- Ưu điểm:

Với hàm băm hiệu quả, chương trình đạt hiệu suất cao trong việc thêm, tìm kiếm và xóa từ.

Cơ chế lưu file nhị phân giúp giảm thời gian đọc/ghi dữ liệu.

- Nhược điểm:

Trong trường hợp dữ liệu không phân phối đều (hàm băm không tốt), hiệu suất giảm đáng kể do danh sách liên kết dài.

**Độ ổn định**

- Khả năng xử lý lỗi:

Các chức năng như loadFromTextFile() và loadHashTableFromBinaryFile() đã xử lý tốt các lỗi như không mở được file hoặc file trống.

Hàm tìm kiếm, xóa, chỉnh sửa xử lý trường hợp không tìm thấy từ mà không gây ra lỗi chương trình.

- Độ tin cậy:

Chương trình hoạt động ổn định với dữ liệu nhỏ và vừa.

Với các thao tác như thêm từ, chương trình tự động cập nhật dữ liệu vào file nhị phân, giúp giảm nguy cơ mất dữ liệu.

# **4.7. Hướng phát triển tương lai**

## **4.7.1. Cải tiến giao diện người dùng**

Mục tiêu:

Cung cấp giao diện thân thiện, dễ sử dụng cho người dùng.

Hỗ trợ cả giao diện dòng lệnh (CLI) và giao diện đồ họa (GUI) cho nhiều đối tượng người dùng.

Kế hoạch thực hiện:

Xây dựng giao diện đồ họa bằng các thư viện như Qt, Tkinter, hoặc GTK+ để cung cấp trải nghiệm trực quan hơn.

Tích hợp tính năng nhập/xuất dữ liệu, tìm kiếm, chỉnh sửa và xóa từ thông qua các nút chức năng hoặc menu.

Hỗ trợ giao diện đa ngôn ngữ để mở rộng đối tượng sử dụng.

## **4.7.2. Tối ưu hóa mã nguồn và cải thiện hiệu suấ**t

Mục tiêu:

Tăng hiệu quả sử dụng tài nguyên hệ thống và giảm thời gian xử lý các thao tác.

Kế hoạch thực hiện:

Cải thiện hàm băm để phân phối dữ liệu đồng đều hơn trong bảng băm, giảm độ dài danh sách liên kết.

Chuyển đổi cơ chế xử lý xung đột từ danh sách liên kết sang Open Addressing hoặc Quadratic Probing để giảm chi phí bộ nhớ.

Áp dụng tối ưu hóa mã nguồn như giảm số vòng lặp không cần thiết và sử dụng các thuật toán hiệu quả hơn.

## **4.7.3. Tính năng mở rộng**

Mục tiêu:

Đáp ứng nhiều nhu cầu hơn từ người dùng và mở rộng phạm vi sử dụng từ điển.

Các tính năng có thể thêm vào:

Gợi ý từ gần đúng: Sử dụng các thuật toán như Levenshtein Distance để hỗ trợ tìm kiếm từ gần đúng khi người dùng nhập sai chính tả.

Học từ vựng: Tích hợp các bài kiểm tra, flashcard và thống kê mức độ học tập.

Cập nhật dữ liệu online: Tích hợp khả năng đồng bộ và cập nhật dữ liệu từ điển từ các nguồn trực tuyến.

Phân loại từ vựng: Cho phép phân loại theo chủ đề, trình độ, hoặc dạng ngữ pháp.

Tìm kiếm nâng cao: Hỗ trợ tìm kiếm theo loại từ hoặc nghĩa.

# **4.8. Kết luận**

## **4.8.1. Tóm tắt kết quả nghiên cứu và phát triển**

Đã xây dựng thành công một phần mềm từ điển tiếng Anh cơ bản, với các chức năng như:

Thêm, tìm kiếm, chỉnh sửa, xóa từ.

Lưu trữ và tải dữ liệu từ file văn bản và file nhị phân.

Chương trình đạt hiệu suất tốt với dữ liệu vừa và nhỏ, đảm bảo tính ổn định và khả năng xử lý lỗi cơ bản.

Đã áp dụng bảng băm để lưu trữ dữ liệu một cách hiệu quả, đảm bảo thời gian xử lý gần như O(1) trong trường hợp trung bình.

## **4.8.2. Các bài học và kinh nghiệm thu được**

Hiểu sâu hơn về cấu trúc dữ liệu bảng băm và các cơ chế xử lý xung đột.

Nâng cao kỹ năng làm việc với file văn bản và file nhị phân trong C++.

Tích lũy kinh nghiệm xử lý lỗi và tối ưu hóa mã nguồn.

Tầm quan trọng của việc thiết kế mô hình dữ liệu rõ ràng trước khi bắt đầu code.

Sự cần thiết của việc kiểm thử chương trình ở nhiều tình huống khác nhau để đảm bảo tính ổn định.

**Bài 5.1: Minh họa cho bài 1**

* Dựa vào source code từ bài 1 bổ sung thêm 1 số hàm vẽ hình ảnh gồm:
* Strut Point: lưu tọa độ các điểm
* char\* int\_to\_char(int x): dùng để chuyển đổi thông tin từ kiểu int sang char\*
* DrawPoint(Point &x,int value,int color=WHITE): vẽ các nút với giá trị truyền vào tương ứng, màu mặc định là TRẮNG

A white background with black and red text

Description automatically generated

* lineDDA(Point p1, Point p2, int thickness,int Delay,int color): cho phép vẽ đường thẳng kết nối 2 điểm với độ DELAY, màu sắc mong muốn

A screenshot of a computer program

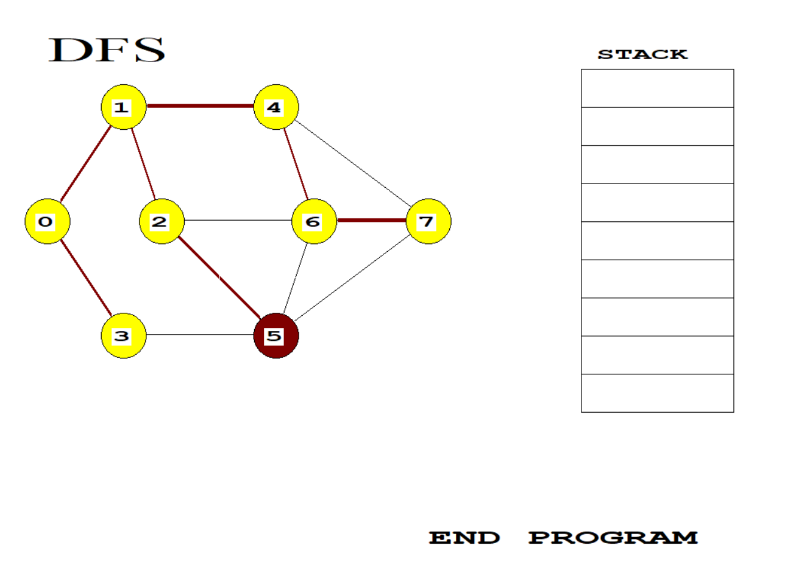
Description automatically generated

* Bổ sung các hàm trên vào hàm xử lí DFS:

+ Ý tưởng:

* Vẽ các nút với ma trận cho trước(áp dụng với ma trận số nút <=8) với tọa độ mặc định cho trước
* Mô phỏng Stack bằng việc vẽ các khối hình chữ nhật
* Tô màu mô phỏng quá trình duyệt đồ thị
* Tô đậm đường đi qua các đỉnh đã được duyệt
* Chỉnh tốc độ duyệt bằng lệnh delay()

+ Kết quả thu được đồ thị đã được duyệt với bằng đường đi được tô đậm và các nút được đánh dấu màu vàng.



**Bài 5.2: Minh họa bằng hình ảnh cho bài 2**

**5.1 Giới thiệu**

Dùng Winform C#.NET để xây dựng hệ thống quản lý tệp bằng cấu trúc dữ liệu cây nhị phân tìm kiếm (BST). Ứng dụng cho phép người dùng thêm, xóa và tìm kiếm tệp, đồng thời hiển thị thông tin về các tệp được lưu trữ trong cây.

**5.2 Cấu trúc ứng dụng**   
**5.2.1 Giao diện người dùng**

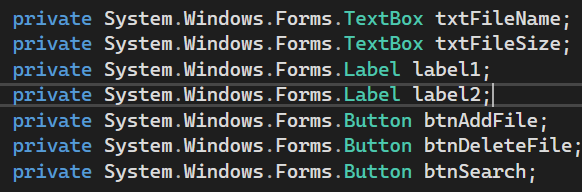
Giao diện người dùng được xây dựng bằng Windows Forms với các thành phần chính:

* **TextBox** để nhập tên tệp và kích thước tệp.
* **Button** để thực hiện các thao tác thêm, xóa và tìm kiếm tệp.
* **Label** để hiển thị chiều cao của cây.
* **ListBox** để hiển thị danh sách các tệp hiện có.

**5.2.2 Mã nguồn**

**5.2.2.1. File Form1.designer.cs**

File chứa mã thiết kế cho giao diện người dùng. Các thành phần được khởi tạo và cấu hình trong phương thức **InitializeComponent()**.

**5.2.2.2 File Form1.cs**

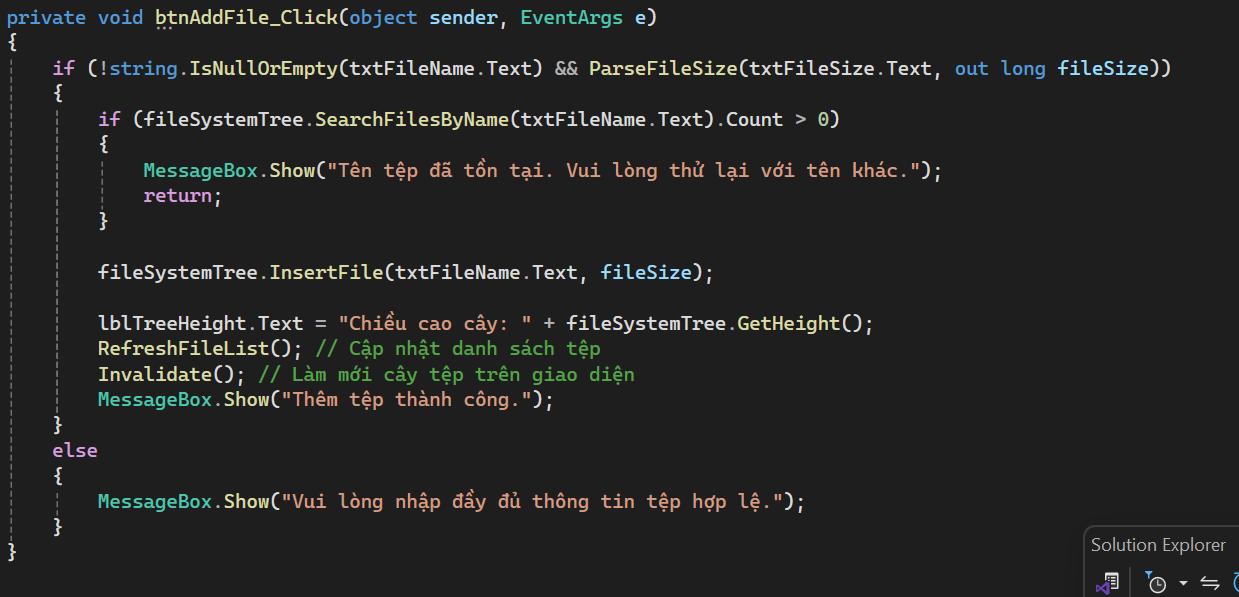
File này chứa mã logic cho ứng dụng. Các lớp chính bao gồm:

* **FileNode**: Đại diện cho một nút trong cây nhị phân tìm kiếm, lưu trữ tên tệp, kích thước tệp, con trái, con phải và chiều cao của nút.
* **FileSystemTree**: Chứa các phương thức để thêm, xóa và tìm kiếm tệp trong cây nhị phân tìm kiếm. Nó cũng bao gồm các phương thức để tính chiều cao của cây và lấy danh sách các tệp.

**5.2.3 Các thao tác chính**

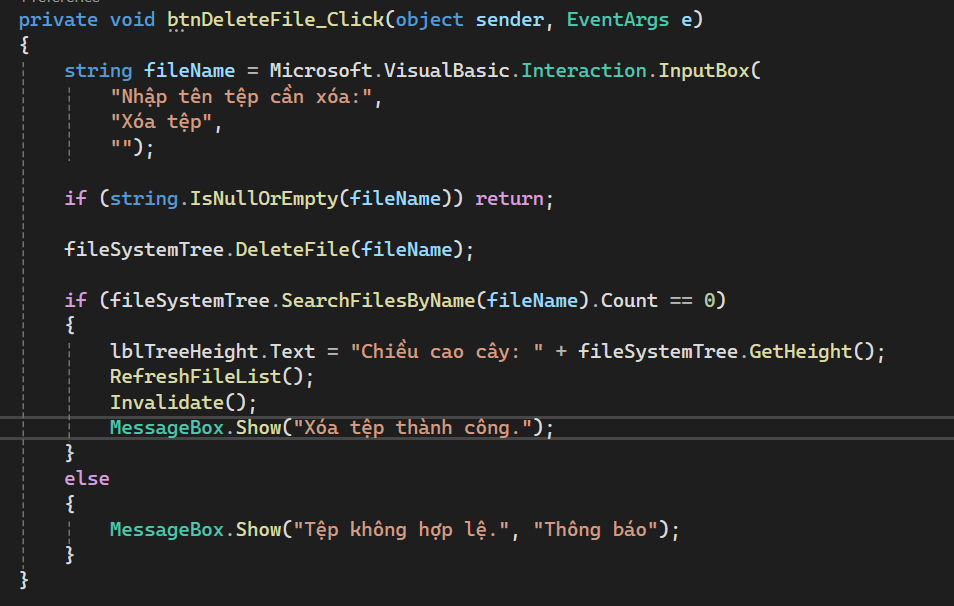
**5.2.3.1 Thêm tệp**

Khi người dùng nhập tên và kích thước tệp và nhấn nút "Thêm Tệp", phương thức **btnAddFile\_Click** sẽ được gọi. Nó kiểm tra tính hợp lệ của thông tin và thêm tệp vào cây nếu tệp chưa tồn tại.



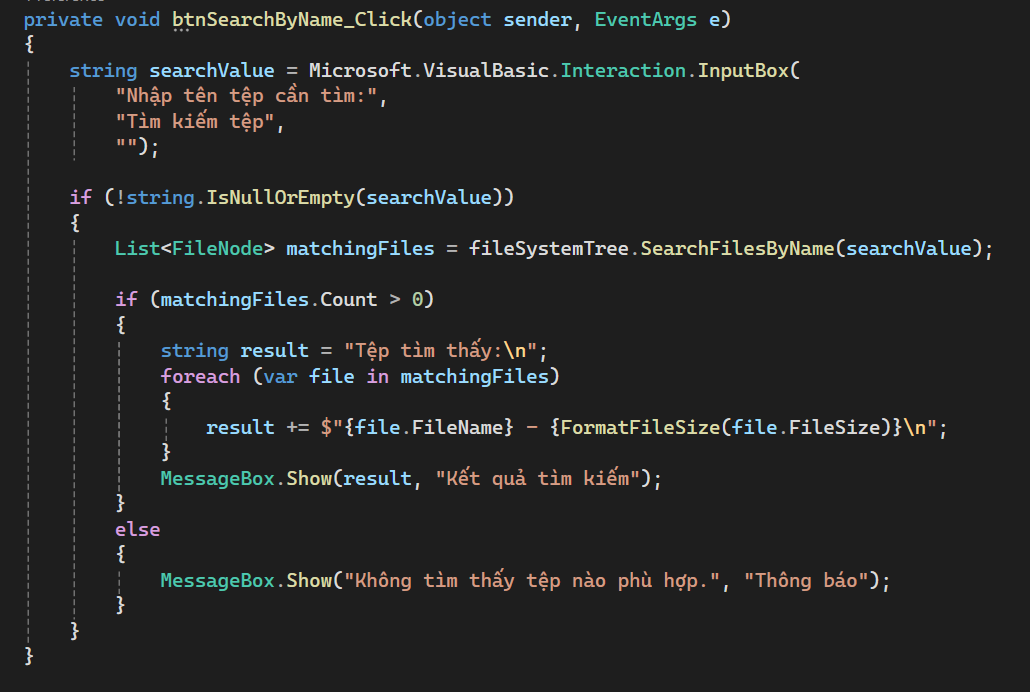
**5.2.3.2. Xóa tệp**

Khi người dùng nhấn nút "Xóa Tệp", phương thức **btnDeleteFile\_Click** sẽ được gọi. Nó yêu cầu người dùng nhập tên tệp cần xóa và thực hiện thao tác xóa.



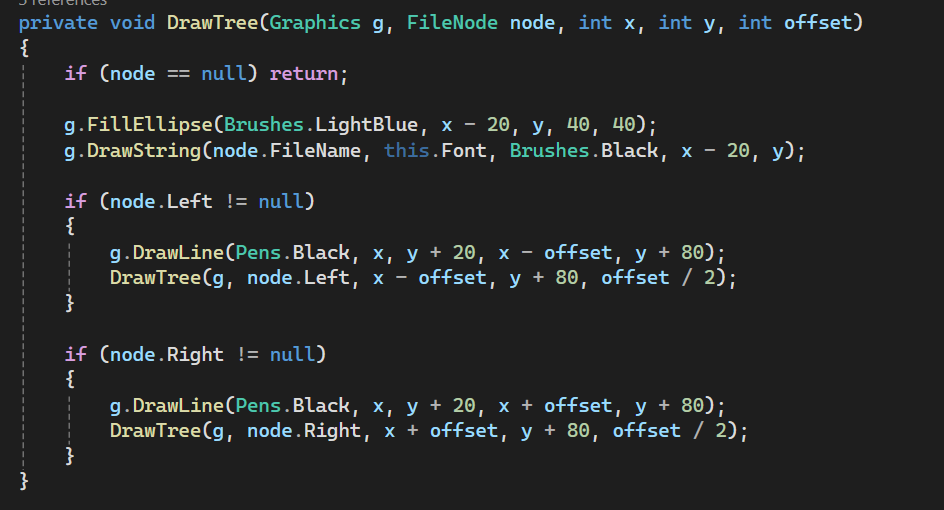
**5.2.3.3 Tìm kiếm tệp**

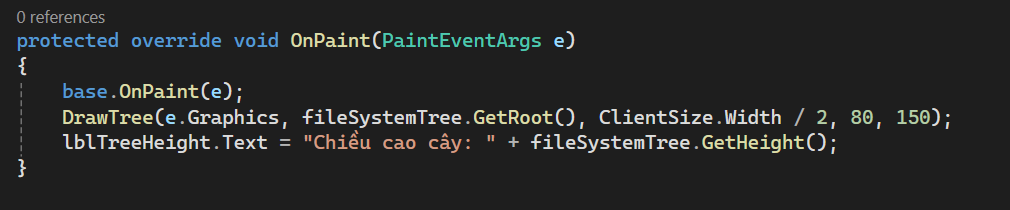
Người dùng có thể tìm kiếm tệp bằng cách nhấn nút "Tìm Kiếm". Phương thức **btnSearchByName\_Click** sẽ hiển thị danh sách các tệp phù hợp với tên đã nhập.



**5.2.3.4 Vẽ cây**

Phương thức **OnPaint** được ghi đè để vẽ cây nhị phân tìm kiếm trên giao diện người dùng. Nó sử dụng phương thức **DrawTree** để vẽ các nút và các kết nối giữa chúng.





**5.3. Kết luận**

Hệ thống quản lý tệp sử dụng cây nhị phân tìm kiếm (BST) là một ứng dụng hữu ích cho việc tổ chức và truy xuất dữ liệu tệp một cách hiệu quả. Qua việc triển khai các thao tác như thêm, xóa và tìm kiếm tệp, ứng dụng không chỉ giúp người dùng quản lý tệp một cách dễ dàng mà còn cung cấp cái nhìn trực quan về cấu trúc dữ liệu thông qua việc vẽ cây.