**Group 4**

**Group Member 1: Nehal Asif 22F-3638**

**Group Member 2: Ibrahim Khan 22F-3707**

**Group Member 3: Emaan Fatima 22F-3640**

**Dictionary Program Documentation**

# **Overview**

The provided C++ program implements a dictionary using a trie data structure. Users can search for words, insert new words, update word meanings, delete words, and view an interactive menu. The program utilizes a console-based interface for user interaction.

# **Classes and Data Structures**

# **1. TrieNode Class**

* **Attributes:**
  + **char data**: Represents the character associated with the node.
  + **TrieNode\*\* children**: Array of pointers to TrieNode, representing the children nodes.
  + **bool isTerminal**: Indicates whether the node marks the end of a word.
  + **string meaning**: Stores the meaning of the word associated with the node.
  + **string partOfSpeech**: Stores the part of speech of the word associated with the node.
* **Methods:**
  + **TrieNode(char data, string partOfSpeech, string meaning)**: Constructor to initialize a TrieNode.

# **2. Trie Class**

* **Attributes:**
  + **TrieNode\* root**: Pointer to the root node of the trie.
  + **int count**: Tracks the number of words in the trie.
* **Methods:**
  + **Trie()**: Constructor to initialize a Trie object.
  + **bool insertHelper(TrieNode\*& root, string word, string partOfSpeech, string meaning)**: Recursively inserts a word into the trie.
  + **void insertWord(string word, string partOfSpeech, string meaning)**: Public method to insert a word into the trie.
  + **string searchHelper(TrieNode\* root, string word)**: Recursively searches for a word in the trie and returns its meaning.
  + **string search(string word)**: Public method to search for a word in the trie.
  + **void updateHelper(TrieNode\*& root, string word, string meaning = "\0")**: Recursively updates the meaning of a word in the trie.
  + **void update(string word, string meaning)**: Public method to update the meaning of a word in the trie.
  + **void deleteWordHelper(TrieNode\*& root, string word)**: Recursively deletes a word from the trie.
  + **void deleteWord(string word)**: Public method to delete a word from the trie.
  + **void saveToFile(string filename = "Dictionary.txt")**: Saves the trie data to a file.
  + **void saveToFileHelper(TrieNode\* root, string currentWord, ofstream& output)**: Recursively saves trie data to a file.
  + **void traverseTrie(TrieNode\* node, const string& word)**: Traverses the trie and displays words.
  + **void searchPrefix(TrieNode\* root, const string& prefix, string currentWord)**: Searches for words with a given prefix.
  + **string autosuggestion()**: Provides an interactive word suggestion feature for the user.
  + **void LoadDictionary()**: Loads initial data from a file into the trie.

# **3. Console Class**

* **Methods:**
  + **void displayMenu()**: Displays the interactive console menu.
  + **int getUserChoice()**: Obtains and returns the user's menu choice.
  + **void executeUserChoice(Trie& dictionary, int choice, string prefix)**: Executes the user's selected menu option.

# **Time Complexity Analysis**

**Insertion**

* **Method Used:**
  + Recursive insertion in **insertHelper** function.
* **Time Complexity:**
  + O(L), where L is the length of the word being inserted.

**Search**

* **Method Used:**
  + Recursive search in **searchHelper** function.
* **Time Complexity:**
  + O(L), where L is the length of the word being searched.

**Update**

* **Method Used:**
  + Recursive update in **updateHelper** function.
* **Time Complexity:**
  + O(L), where L is the length of the word being updated.

**Deletion**

* **Method Used:**
  + Recursive deletion in **deleteWordHelper** function.
* **Time Complexity:**
  + O(L), where L is the length of the word being deleted.

**Autocomplete (Prefix Search)**

* **Method Used:**
  + Recursive search in **searchPrefix** function.
* **Time Complexity:**
  + O(P), where P is the length of the prefix being searched.

**File Save and Load**

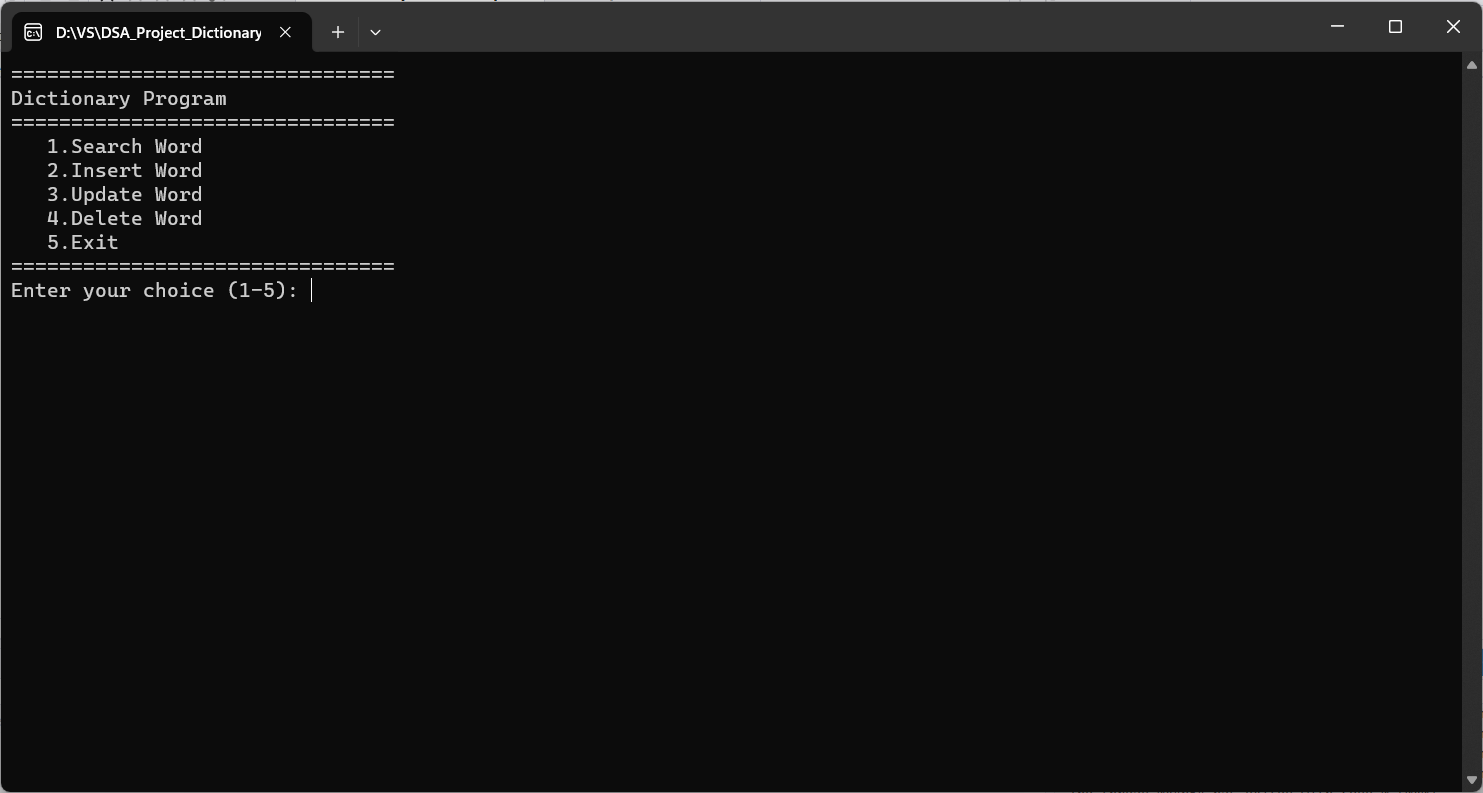
* **Methods Used:**
  + **saveToFile** and **LoadDictionary** functions.
* **Time Complexity:**
  + O(N), where N is the total number of nodes in the trie.

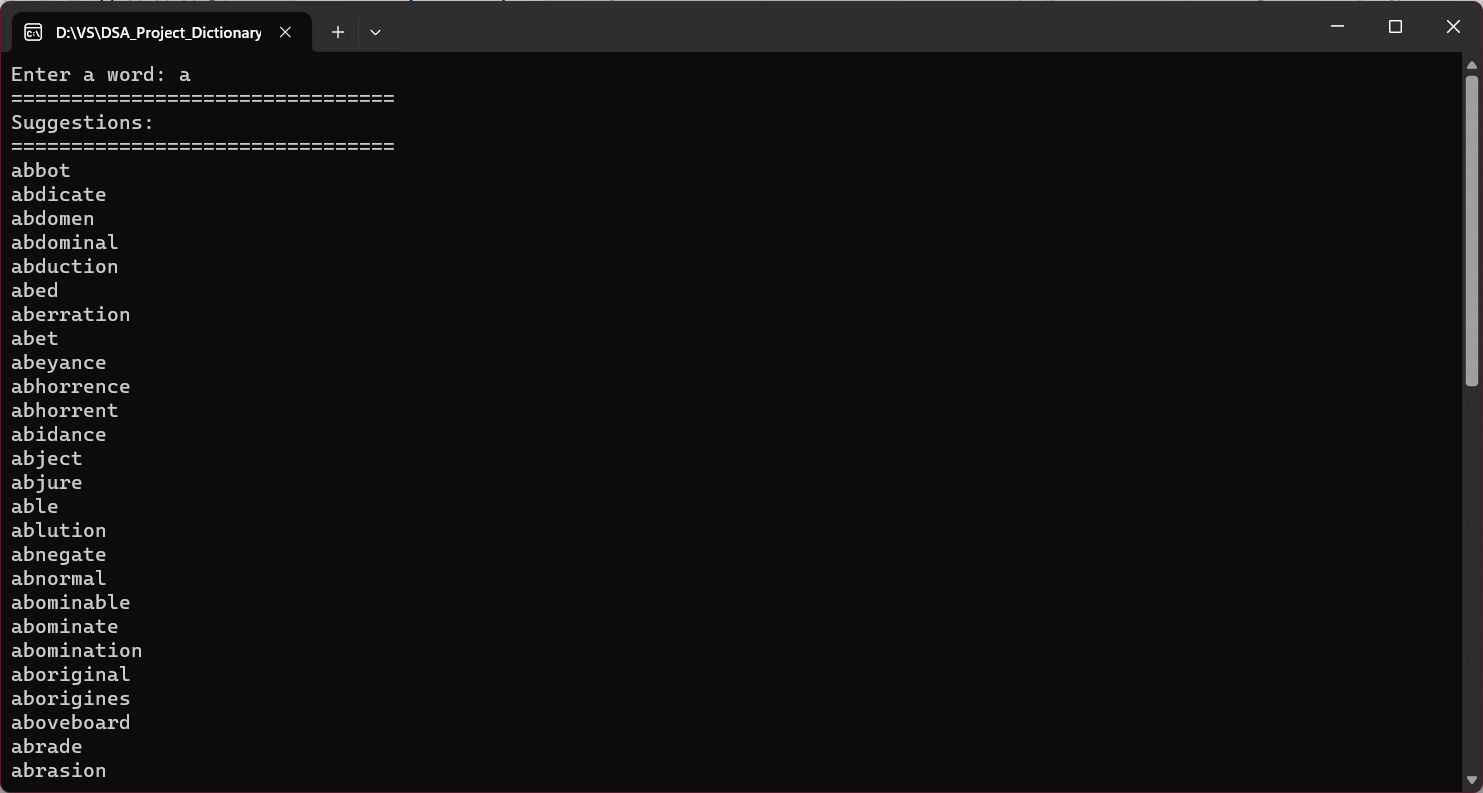
**Conclusion**

The trie data structure allows efficient word retrieval and modification operations in the dictionary. The time complexity of operations is generally proportional to the length of the words involved. The interactive console interface enhances user experience, and the program provides persistent storage by saving and loading data to and from a file.

|  |
| --- |
| **Running Code Overview** Code: #include<iostream>  #include<fstream>  #include<string>  #include<conio.h>  #include <iomanip>  using namespace std;  class TrieNode  {  public:  char data;  TrieNode\*\* children;  bool isTerminal;  string meaning;  string partOfSpeech;  TrieNode(char data, string partOfSpeech, string meaning)  {  this->data = data;  children = new TrieNode \* [27];  for (int i = 0; i < 27; i++)  {  children[i] = NULL;  }  isTerminal = false;  this->meaning = meaning;  this->partOfSpeech = partOfSpeech;  }  };  class Trie  {  TrieNode\* root;  public:  int count;  Trie()  {  this->count = 0;  root = new TrieNode('\0', "", "");  }  bool insertHelper(TrieNode\*& root, string word, string partOfSpeech, string meaning)  {  if (word.size() == 0)  {  if (!root->isTerminal)  {  root->isTerminal = true;  root->meaning = meaning;  root->partOfSpeech = partOfSpeech;  return true;  }  else  {  return false;  }  }  int index;  word[0] = tolower(word[0]);  if (word[0] == ' ' || word[0] == '-')  {  index = 26;  }  else  {  index = word[0] - 'a';  }  TrieNode\* child;  if (root->children[index] != NULL)  {  child = root->children[index];  }  else  {  child = new TrieNode(word[0], partOfSpeech, meaning);  root->children[index] = child;  }  return insertHelper(child, word.substr(1), partOfSpeech, meaning);  }  void insertWord(string word, string partOfSpeech, string meaning)  {  if (insertHelper(root, word, partOfSpeech, meaning))  {  this->count++;  }  saveToFile();  }  string searchHelper(TrieNode\* root, string word)  {  if (word.size() == 0 && root->isTerminal) {  return ("(" + root->partOfSpeech + ") " + root->meaning);  }  int index;  if (word[0] == ' ' || word[0] == '-')  {  index = 26;  }  else  {  index = word[0] - 'a';  }  if (root->children[index] == NULL)  {  return "Word not found!";  }  return searchHelper(root->children[index], word.substr(1));  }  string search(string word)  {  return searchHelper(root, word);  }  void updateHelper(TrieNode\*& root, string word, string meaning = "\0")  {  if (word.size() == 0 && root->isTerminal) {  cout << "Meaning update." << endl;  root->meaning = meaning;  }  int index;  if (word[0] == ' ' || word[0] == '-')  {  index = 26;  }  else  {  index = word[0] - 'a';  }  if (root->children[index] == NULL)  {  cout << "Word not found.";  return;  }  updateHelper(root->children[index], word.substr(1), meaning);  }  void update(string word, string meaning)  {  updateHelper(root, word,meaning);  saveToFile();  }  void deleteWordHelper(TrieNode\*& root, string word)  {  if (word.size() == 0)  {  if (root->isTerminal)  {  root->isTerminal = false;  return;  }  }  int index;  if (word[0] == ' ' || word[0] == '-')  {  index = 26;  }  else  {  index = word[0] - 'a';  }  if (root->children[index])  {  deleteWordHelper(root->children[index], word.substr(1));  // Check if the child node can be removed  if (!root->children[index]->isTerminal)  {  bool flag = true;  for (int i = 0; i < 27; i++)  {  if (root->children[index]->children[i])  {  flag = false;  break;  }  }  if (flag)  {  delete root->children[index];  root->children[index] = nullptr;  }  }  }  }  void deleteWord(string word)  {  deleteWordHelper(root, word);  saveToFile("Dictionary.txt");  cout << endl << "Word Deleted" << endl;  }  void saveToFile(string filename = "Dictionary.txt")  {  ofstream output(filename);  saveToFileHelper(root, "", output);  output.close();  }  void saveToFileHelper(TrieNode\* root, string currentWord, ofstream& output)  {  if (root->isTerminal)  {  output << currentWord << " " << root->partOfSpeech << " " << root->meaning << endl;  }  for (int i = 0; i < 27; i++)  {  if (root->children[i] != NULL)  {  if (i < 26)  {  saveToFileHelper(root->children[i], currentWord + char('a' + i), output);  }  else  {  saveToFileHelper(root->children[i], currentWord + ' ', output);  }  }  }  }  void traverseTrie(TrieNode\* node, const string& word) {  if (node->isTerminal) {  cout << word << endl;  }  for (int i = 0; i < 27; i++) {  if (node->children[i] != nullptr) {  char c;  if (i < 26)  c = 'a' + i;  else  c = ' ';  traverseTrie(node->children[i], word + c);  }  }  }  // Search for prefix in the trie and return suggestions  void searchPrefix(TrieNode\* root, const string& prefix, string currentWord) {  TrieNode\* curr = root;  for (char c : prefix) {  int index;  if (c == '-' || c == ' ')  index = 26;  else  index = c - 'a';  if (curr->children[index] == nullptr) {  return; // No suggestions found  }  curr = curr->children[index];  }  // Traverse the trie to find all suggestions  traverseTrie(curr, currentWord + prefix);  }  string autosuggestion() {  string prefix;  cout << "Enter a word: ";  while (true) {  char c = \_getche();  if (c == '\r') { // Enter key pressed  system("cls");  return prefix;  break;  }  else if (c == '\b') { // Backspace key pressed  if (!prefix.empty()) {  prefix.pop\_back();  system("cls"); // Clear console screen  cout << "Enter a word: " << prefix << endl;  cout << "================================" << endl;  cout << "Suggestions:" << endl;  cout << "================================" << endl;  searchPrefix(root, prefix, "");  }  }  else {  prefix += c;  system("cls"); // Clear console screen  cout << "Enter a word: " << prefix << endl;  cout << "================================" << endl;  cout << "Suggestions:" << endl;  cout << "================================" << endl;  searchPrefix(root, prefix, "");  }  }  }  void LoadDictionary()  {  string word;  string partOfSpeech;  string meaning;  ifstream infile;  infile.open("Dictionary.txt");  if (!infile)  {  cout << "Error in Opening File" << endl;  }  else  {  cout << "Loading Dictionary...." << endl;  while (infile >> word >> partOfSpeech)  {  // Read the rest of the line for the meaning  getline(infile, meaning);  // Insert the word into the dictionary  insertWord(word, partOfSpeech, meaning);  }  }  infile.close(); // Close the file after reading  }  };  class Console  {  public:  void displayMenu() {  system("cls"); // Clear screen for a clean display  cout << "================================" << endl;  cout << setw(15) << right << "Dictionary Program" << setw(15) << left << " " << endl;  cout << "================================" << endl;  cout << setw(5) << right << "1." << setw(25) << left << "Search Word" << endl;  cout << setw(5) << right << "2." << setw(25) << left << "Insert Word" << endl;  cout << setw(5) << right << "3." << setw(25) << left << "Update Word" << endl;  cout << setw(5) << right << "4." << setw(25) << left << "Delete Word" << endl;  cout << setw(5) << right << "5." << setw(25) << left << "Exit" << endl;  cout << "================================" << endl;  }  int getUserChoice() {  int choice;    cout << "Enter your choice (1-5): ";  cin >> choice;  cin.ignore();  return choice;  }  void executeUserChoice(Trie& dictionary, int choice,string prefix) {  string partOfSpeech, meaning;  switch (choice) {  case 1:  system("cls");  cout << "Meaning: " << dictionary.search(prefix) << endl;  break;  case 2:  cout << "Enter part of speech: ";  getline(cin, partOfSpeech);  cin.ignore();  cout << "Enter meaning: ";  getline(cin, meaning);  cin.ignore();  dictionary.insertWord(prefix, partOfSpeech, meaning);  system("cls");  cout << "Word inserted successfully!" << endl;  break;  case 3:  cout << "Enter new meaning: ";  getline(cin, meaning);  dictionary.update(prefix, meaning);  system("cls");  cout << "Meaning updated successfully!" << endl;  break;  case 4:  dictionary.deleteWord(prefix);  system("cls");  cout << "Word Deleted Successfully!" << endl;  break;  case 5:  cout << "Exiting the Program. Goodbye!" << endl;  exit(0);  default:  cout << "Invalid choice. Please Enter a Number between 1 and 5." << endl;  }  }  };  int main() {  Trie dictionary;  dictionary.LoadDictionary(); // Load initial data from the file  system("cls");  Console obj;  string prefix;  char choice;  while (true) {  system("cls");  obj.displayMenu();  choice = obj.getUserChoice();  // Use autosuggestion to get the word from the user  if(choice<5)  prefix = dictionary.autosuggestion();  obj.executeUserChoice(dictionary,choice,prefix);  // Press any key to continue...  system("pause");  system("cls");  }  return 0;  } |

**Output:**





A screenshot of a computer

Description automatically generated