PROJECT: DICTIONARY APPLICATION

Introduction

The Dictionary Application is designed to provide efficient storage, retrieval, and management of word definitions using a Binary Search Tree (BST) data structure. This application allows users to interactively add new words, search for word meanings, update existing definitions, delete words, and view the entire dictionary.

Objective

The objective of this project is to create a robust and user-friendly dictionary application that leverages the BST data structure for quick lookup operations. By organizing words alphabetically, the application ensures efficient insertion, deletion, and search functionalities, providing a seamless experience for users accessing word definitions.

Features

- **Add a New Word**: Users can add new words along with their definitions to expand the dictionary.
- **Search for a Word**: Provides the ability to search for a specific word and display its meaning.
- **Update Meaning of a Word**: Allows users to modify the definition of an existing word.
- **Delete a Word**: Enables deletion of a word and its associated definition from the dictionary.
- **Print All Words and Meanings**: Displays all words and their respective definitions in alphabetical order.
- **Interactive Menu**: User-friendly interface with a menu-driven approach for intuitive interaction.

Implementation Details

Technologies Used:

- **Languages:** C++
- **Libraries:** `#include <iostream> #include <string>`
- **Development Environment:** Any C++ IDE or compiler supporting standard libraries

Data Structure: Implemented using a Binary Search Tree (BST) where each node contains a word (key) and its meaning.

- **Insertion**: Words are inserted into the BST based on their alphabetical order.
- **Search**: Utilizes BST search algorithm to quickly locate and retrieve word meanings.
- **Deletion**: Supports deletion of nodes using standard BST deletion techniques, handling cases with zero, one, or two children.
- **Update**: Allows for updating the meaning associated with an existing word.
- **Traversal**: Inorder traversal of the BST is used to print all words and meanings in alphabetical order.

- **User Input Handling**: Utilizes `cin` for user input, handling both single-word inputs and multi-line definitions using `getline`.

Future Enhancements

- **User Authentication**: Implement user authentication to manage access and modifications to the dictionary.
- **Persistent Storage**: Integrate file I/O operations to save and load dictionary contents from disk.
- **Search Optimization**: Implement techniques like AVL trees or Red-Black trees for better balance and improved performance.
- **Multi-Language Support**: Extend dictionary capabilities to support multiple languages with efficient lookup mechanisms.
- **GUI Application**: Develop a graphical user interface (GUI) version for enhanced user experience and accessibility.

CODE:

```
#include <iostream>
#include <string>
using namespace std;

// Definition of a Node in BST
struct Node {
    string key;
    string meaning;
    Node* left;
    Node* right;

    Node(string k, string m) : key(k), meaning(m), left(NULL), right(NULL) {}
};

// Class for Binary Search Tree
class Dictionary {
private:
    Node* root;

    // Helper function to insert a new node into the BST
    Node* insert(Node* node, string key, string meaning) {
        if (node == NULL) {
            node = new Node(key, meaning);
        } else if (key < node->key) {
            node->left = insert(node->left, key, meaning);
        } else if (key > node->key) {
            node->right = insert(node->right, key, meaning);
        }
        return node;
    }
```

```cpp
    // Helper function to search for a key in the BST
    Node* search(Node* node, string key) {
        if (node == NULL || node->key == key) {
            return node;
        } else if (key < node->key) {
            return search(node->left, key);
        } else {
            return search(node->right, key);
        }
    }

    // Helper function to delete a node from the BST
    Node* deleteNode(Node* node, string key) {
        if (node == NULL) {
            return node;
        }

        if (key < node->key) {
            node->left = deleteNode(node->left, key);
        } else if (key > node->key) {
            node->right = deleteNode(node->right, key);
        } else {
            // Node to be deleted found

            // Case 1: Node has no children or only one child
            if (node->left == NULL) {
                Node* temp = node->right;
                delete node;
                return temp;
            } else if (node->right == NULL) {
                Node* temp = node->left;
                delete node;
                return temp;
            }

            // Case 2: Node has two children
            Node* temp = minValueNode(node->right);
            node->key = temp->key;
            node->meaning = temp->meaning;
            node->right = deleteNode(node->right, temp->key);
        }
        return node;
    }

    // Helper function to find the node with minimum key value in BST
    Node* minValueNode(Node* node) {
        Node* current = node;
        while (current && current->left != NULL) {
```

```cpp
            current = current->left;
        }
        return current;
    }

    // Helper function to print the BST inorder
    void inorder(Node* node) {
        if (node != NULL) {
            inorder(node->left);
            cout << node->key << ": " << node->meaning << endl;
            inorder(node->right);
        }
    }

public:
    Dictionary() : root(NULL) {}

    // Function to insert a key-value pair into the dictionary
    void insert(string key, string meaning) {
        root = insert(root, key, meaning);
    }

    // Function to search for a key in the dictionary
    string search(string key) {
        Node* result = search(root, key);
        if (result != NULL) {
            return result->meaning;
        } else {
            return "Key not found";
        }
    }

    // Function to delete a word from the dictionary
    void deleteWord(string key) {
        root = deleteNode(root, key);
    }

    // Function to update the meaning of a word in the dictionary
    void updateMeaning(string key, string newMeaning) {
        Node* node = search(root, key);
        if (node != NULL) {
            node->meaning = newMeaning;
            cout << "Meaning updated successfully!\n";
        } else {
            cout << "Word not found. Cannot update meaning.\n";
        }
    }
```

```cpp
    // Function to print all keys and meanings in the dictionary
    void printDictionary() {
        inorder(root);
    }
};

// Main function for testing the Dictionary class
int main() {
    Dictionary dict;

    // Inserting initial key-value pairs
    dict.insert("apple", "A fruit that grows on trees.");
    dict.insert("banana", "A long curved fruit that grows in clusters.");
    dict.insert("cat", "A small domesticated carnivorous mammal with soft
fur.");

    // Menu for user interaction
    cout << "Dictionary Application\n";
    cout << "----------------------\n";
    cout << "1. Add a new word\n";
    cout << "2. Search for a word\n";
    cout << "3. Update meaning of a word\n";
    cout << "4. Delete a word\n";
    cout << "5. Print all words and meanings\n";
    cout << "6. Exit\n";

    int choice;
    string key, meaning, newMeaning;

    do {
        cout << "\nEnter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "\nEnter word to add: ";
                cin >> key;
                cout << "Enter meaning: ";
                cin.ignore(); // to ignore newline character left in the
stream
                getline(cin, meaning);
                dict.insert(key, meaning);
                cout << "Word '" << key << "' added successfully!\n";
                break;
            case 2:
                cout << "\nEnter word to search: ";
                cin >> key;
```

```
                cout << "Meaning of '" << key << "': " << dict.search(key) <<
endl;
                break;
            case 3:
                cout << "\nEnter word to update meaning: ";
                cin >> key;
                cout << "Enter new meaning: ";
                cin.ignore(); // to ignore newline character left in the
stream
                getline(cin, newMeaning);
                dict.updateMeaning(key, newMeaning);
                break;
            case 4:
                cout << "\nEnter word to delete: ";
                cin >> key;
                dict.deleteWord(key);
                cout << "Word '" << key << "' deleted successfully!\n";
                break;
            case 5:
                cout << "\nDictionary contents:\n";
                dict.printDictionary();
                break;
            case 6:
                cout << "\nExiting...\n";
                break;
            default:
                cout << "Invalid choice. Please enter a valid option.\n";
        }
    } while (choice != 6);

    return 0;
}
```

OUTPUT:

C:\Users\Reen\Documents\Dev-Cpp\CODES\DSAPROJECT.exe

```
Dictionary Application
----------------------
1. Add a new word
2. Search for a word
3. Update meaning of a word
4. Delete a word
5. Print all words and meanings
6. Exit

Enter your choice: 1

Enter word to add: incentive
Enter meaning: additional profit
Word 'incentive' added successfully!

Enter your choice: 2

Enter word to search: incentive
Meaning of 'incentive': additional profit

Enter your choice: 3

Enter word to update meaning: additional gain
Enter new meaning: Word not found. Cannot update meaning.

Enter your choice: 3

Enter word to update meaning: incentive
Enter new meaning: additional gain
Meaning updated successfully!

Enter your choice: 5
```

```
Enter your choice: 5

Dictionary contents:
apple: A fruit that grows on trees.
banana: A long curved fruit that grows in clusters.
cat: A small domesticated carnivorous mammal with soft fur.
incentive: additional gain

Enter your choice: 4

Enter word to delete: cat
Word 'cat' deleted successfully!

Enter your choice: 5

Dictionary contents:
apple: A fruit that grows on trees.
banana: A long curved fruit that grows in clusters.
incentive: additional gain

Enter your choice: 6

Exiting...
```

Time Complexity of Operations Performed in the Dictionary Application:

1. Insert Operation

- **Upper Bound (Worst-case)**: O(n) complexity
- **Lower Bound (Best-case)**: O(1), if the tree is empty, the new node is inserted as the root.
- **Average-case**: O(log n), assuming the BST is reasonably balanced.

2. Search Operation

- **Upper Bound (Worst-case)**: O(n) complexity
- **Lower Bound (Best-case)**: O(1), if the node being searched is the root.
- **Average-case**: O(log n), assuming the BST is balanced.

3. Delete Operation

- **Upper Bound (Worst-case:** O(n) complexity
- **Lower Bound (Best-case)**: O(1), if the node to be deleted has no children (leaf node).
- **Average-case**: O(log n), assuming the BST is balanced.

4. Update Operation

- The update operation involves a search followed by modification of the node's meaning, making its complexity dependent on the search operation.
- **Overall Complexity**: O(log n) for update, assuming the BST is balanced.

5. Inorder Traversal (Printing All Words)

- **Upper Bound (Worst-case)**: O(n), where n is the number of nodes. Inorder traversal visits all nodes.
- **Lower Bound (Best-case)**: O(n), since all nodes must be visited to print them in order.
- **Tight Bound (Average-case)**: O(n), as every node must be processed exactly once to produce a sorted output.

Summary of Time Complexity:

- **Insert**: O(log n) to O(n)
- **Search**: O(log n) to O(n)
- **Delete**: O(log n) to O(n)
- **Update**: O(log n)
- **Inorder Traversal**: O(n)

CONCLUSION:

The Dictionary Application presented here demonstrates the effective use of a Binary Search Tree (BST) data structure to manage and manipulate word definitions. By leveraging BST's properties of efficient insertion, deletion, and search operations, the application provides a streamlined experience for users interacting with the dictionary.