East West University

Department of CSE

CSE-207
Data Structures

Project No : 14

Project Name

"Implementation of Radix tree (Trie)"

Semester & Year	Group No
Fall 2023	**02**

Name of Student & Id :	Course Instructor :
Osama Bin Qashem **2022-2-60-058** **KM Fahim A Bari** **2022-2-60-153** **Abrar Hossain Zahin** **2022-2-60-040**	Dr Maheen Islam Chairperson & Associate Professor Department of Computer Science and Engineering, EWU

Date of Report Submitted : 28 December, 2023

INTRODUCTION

This project presents a basic implementation of Radix Tree, a data structure commonly used for storing and searching strings. A Radix Tree is also known as a Trie, which is short for "retrieval". The term "Radix" refers to the base of a number system, and in the context of Radix Trees, it signifies the base of the keys being stored. The primary objective of this implementation is to highlight the fundamental operations of inserting, searching, and deleting strings from the tree.

What is a radix tree?

A Radix Tree is a data structure that efficiently stores and organizes strings by grouping together words with shared prefixes. It is a type of tree structure where each node represents a common prefix of the strings it contains.

Benefits of using the radix tree:

1. **Rapid Vocabulary Retrieval:** Radix trees make it super quick to find words, especially when many words share the same beginnings (like “Tea”,“Ten”,“Ted” “Tie”) It is like finding words in a dictionary with shortcuts.
2. **Optimal Memory Utilization:** By eliminating redundancy in word storage, radix trees ensure efficient memory usage. This is akin to employing concise abbreviations to economize space in written communication.

3. **Enhanced Predictive Text Capability:** They are fantastic for autocomplete and predictive text because they can suggest words that start with what you have typed so far, like when your phone suggests words as you type a message.

4. **Alphabetic Organization Maintenance:** Radix trees contribute to the systematic organization of words, ensuring they remain in alphabetical order. This is analogous to the meticulous arrangement of books on a shelf for easy reference.
5. **Seamless Updates:** Adding or removing words from a radix tree is a seamless process, avoiding disruptions to the overall structure. This is comparable to

the ease of incorporating or deleting entries in a smartphone's contact list without causing chaos.

6. **Prefix Sharing:** The concept of prefix sharing is crucial for optimizing memory usage. Common prefixes among keys are represented by shared paths in the tree, minimizing redundancy and improving storage efficiency.
7. **Application and Use Cases:** Radix Trees have diverse applications, such as IP routing tables, spell checkers, and database indexing. Their efficiency in managing dynamic sets of strings makes them versatile in various computing scenarios.

In simple terms, radix trees are like a smart way of organizing and finding words, saving both time and space.

Space complexity:

The space complexity of a radix tree depends on the number of unique strings stored in the tree and the length of these strings. In general, the space complexity of a radix tree is O(N * M), where: N is the number of unique strings stored in the tree. M is the average length of these strings.

Auxiliary Space: This refers to the extra space or memory used by the algorithm, excluding the input size. It includes all the additional data structures and variables that the algorithm needs during its execution. **Input Space:** This component refers to the space required by the input to the algorithm. It is not considered when analyzing space complexity, as it is assumed that the input is already present in the memory.

However, in the best-case scenario, where all the stored strings have long common prefixes, the space complexity can approach O(N + M), which is much more efficient than the worst-case estimate.

Algorithms to implement radix tree:

Here is a high-level description of an algorithm to implement a radix tree:

Insertion (insert function): The insert function is responsible for adding a new string to the Radix Tree. The tree is traversed from the root, comparing characters of the input string with the characters of the current node's value. At each step, it looks for the longest common prefix between the current node's value and the input string. If there is a common prefix, the tree is split at the point where the prefix ends. The current node is updated, and a new node is created for the remaining suffix of the input string. If there is no common prefix, a new node is created to represent the entire input string. If the input string is a prefix of an existing string in the tree, the existing node is marked as a data node.

Deletion (remove function): The remove function is responsible for removing a string from the Radix Tree. Similar to the insert operation, the tree is traversed until the node representing the input string is found. If the node is a leaf (represents the entire string), it is marked as a non-data node. If the node has children, it remains in the tree as a non-data node unless it has only one child. In that case, the node is combined with its child to maintain the compression property of the Radix Tree. The deletion process involves removing nodes up the tree until a node with more than one child is encountered.

Search (search function): The search function checks if a given string exists in the Radix Tree. It traverses the tree from the root, comparing characters of the input string with the characters of the current node's value. If, at any point, there is a mismatch, the search fails, indicating that the string is not present. If the search reaches the end of the input string and the current node is marked as a data node, the search succeeds.

Project Source Code

```
#include<iostream>
#include<conio.h>
#include<windows.h>
#include<algorithm>
#include<vector>
#include<io.h>
#include<fcntl.h>

using namespace std;

const int MAX_CHILD = 26;
const int SUGGESTION_LIMIT = 25;

class RadixTree {
   class Node;
   Node *_root = nullptr;
   void print_tree(Node*, int);
   void get_entries(Node*, string, vector<string>&);
   bool search(Node*, string);
   Node* insert(Node*, string, Node*);
   Node* remove(Node*, string, Node*);
public:
   void insert(string);
   void remove(string);
   bool search(string);
   void display();
   void autocomplete(Node*, int);
   Node* root() { return _root; }
} tree;

class RadixTree::Node {
   Node* next[MAX_CHILD]{};
   int cc = 0;
   string val;
   bool datanode;
public:
   // ctors
   Node(string value) : val(value), datanode(false) {}
   Node(string value, bool dn) : val(value), datanode(dn) {}

   Node* split(int ix, Node* parent) {
      if (ix < 0 || ix >= val.size())
         return this;
      Node *prefix = new Node(val.substr(0, ix));
      if (parent != nullptr)
         parent->set_child(prefix->value()[0]-'a', prefix);
```

```
        val = val.substr(ix);
        prefix->set_child(val[0]-'a', this); // val[0] is the edge with the parent
        return prefix;
    }

    Node* combine(Node *parent, Node *grandparent) {
        if (parent->child_count() != 1) return this;
        val = parent->value() + val;
        if (grandparent != nullptr)
            grandparent->set_child(val[0]-'a', this);
        delete parent;
        return this;
    }

    // getter and setters
    void set_child(int ix, Node* child) {
        if (ix < 0 || ix >= MAX_CHILD) return;
        if (next[ix] == nullptr) {
            if (child == nullptr) return;
            next[ix] = child;
            cc++;
        } else {
            if (child == nullptr) cc--;
            next[ix] = child;
        }
    }
    Node* child(int ix) {
        if (ix < 0 || ix >= MAX_CHILD) return nullptr;
        return next[ix];
    }
    int child_count() { return cc; }
    void setdatanode(bool b) { datanode = b; }
    bool isdatanode() { return datanode; }
    string value() { return val; }
};

void RadixTree::insert(string str) {
    transform(str.begin(), str.end(), str.begin(), ::tolower);
    _root = insert(_root, str, nullptr);

}

void RadixTree::remove(string str) {
    transform(str.begin(), str.end(), str.begin(), ::tolower);
    _root = remove(_root, str, nullptr);
}

bool RadixTree::search(string str) {
```

```
    transform(str.begin(), str.end(), str.begin(), ::tolower);
    return search(_root, str);
}

RadixTree::Node* RadixTree::insert(Node* root, string str, Node* parent = nullptr) {
    if (root == nullptr) {
        Node *node = new Node(str);
        node->setdatanode(true);
        root = node;
        return node;
    }
    string val = root->value();
    int i = 0, lim = min(str.size(), val.size());
    while (i < lim && str[i] == val[i]) i++;
    if (i == val.size()) {
        if (i == str.size()) {
            root->setdatanode(true);
            return root;
        }
        Node *child = root->child(str[i]-'a');
        root->set_child(
            str[i]-'a',
            insert(child, str.substr(i), root)
        );
        return root;
    }
    root = root->split(i, parent);
    if (i == str.size()) {
        root->setdatanode(true);
        return root;
    }
    string suffix = str.substr(i);
    root->set_child(suffix[0]-'a', new Node(suffix, true));
    return root;
}

bool RadixTree::search(Node* root, string str) {
    if (root == nullptr) return false;

    string val = root->value();
    int i = 0, lim = min(str.size(), val.size());
    while (i < lim && str[i] == val[i]) i++;

    if (i != val.size())
        return false;

    if (i != str.size()) {
        Node *child = root->child(str[i]-'a');
```

```
        return search(child, str.substr(i));
    }

    return true;
}

RadixTree::Node* RadixTree::remove(Node* root, string str, Node *parent) {
    if (root == nullptr) return nullptr;
    string val = root->value();
    int i = 0, lim = min(str.size(), val.size());
    while (i < lim && str[i] == val[i]) i++;
    if (i != val.size())
        return root;
    if (i != str.size()) {
        Node *child = root->child(str[i]-'a');
        root->set_child(
            str[i]-'a',
            remove(child, str.substr(i), root)
        );
        if (!root->isdatanode() && root->child_count() == 1) {
            int e = 0;
            while (e < MAX_CHILD && root->child(e) == nullptr) e++;
            root = root->child(e)->combine(root, parent);
        }
        return root;
    }

    // root->value() == str
    root->setdatanode(false);
    if (root->child_count() == 0) {
        delete root;
        return nullptr;
    }
    if (root->child_count() == 1) {
        int e = 0;
        while (e < MAX_CHILD && root->child(e) == nullptr) e++;
        root = root->child(e)->combine(root, parent);
    }
    return root;
}

void RadixTree::print_tree(Node *root, int level) {
    if (root == nullptr) return;
    static const wchar_t blank = L'\u252C', tri = L'\u251C', h_bar = L'\u2500', di = L'\u2514', v_bar =
L'\u2502';
    static wstring prefix;

    // print root first
```

```
    string val = root->value();
    if (root->isdatanode())
        transform(val.begin(), val.end(), val.begin(), ::toupper);
    if (val.empty())
        wcout << blank << endl;
    else
        wcout << wstring(val.begin(), val.end()) << endl;

    int cc = root->child_count();
    for (int i = 0; i < MAX_CHILD; i++) {
        Node *child = root->child(i);
        if (child == nullptr)
            continue;
        cc--;
        wcout << prefix;
        wcout << (cc? tri : di);
        wcout << h_bar << h_bar << h_bar;
        prefix.push_back(cc? v_bar : ' ');
        prefix.append(L"   ");
        print_tree(child, level+1);
        prefix.resize(prefix.size()-4);
    }
}

void RadixTree::display() {
    _setmode(_fileno(stdout), 0x00020000); // unicode mode
    print_tree(_root, 0);
    _setmode(_fileno(stdout), 0x00004000); // ascii mode
}

void RadixTree::get_entries(Node *root, string value, vector<string> &entries) {
    if (root == nullptr || entries.size() == SUGGESTION_LIMIT) return;
    if (root->isdatanode() && entries.size() < SUGGESTION_LIMIT)
        entries.push_back(value);
    for (int i = 0; i < MAX_CHILD; i++) {
        auto child = root->child(i);
        if (child == nullptr)
            continue;

        get_entries(child, value + child->value(), entries);
    }
}

void RadixTree::autocomplete(Node *root, int index=0) {
    static char c;
    static string buffer;
    do c = getch();
    while (!isprint(c) && c != '\r' && (c != '\b' || buffer.empty()));
```

```
    if (c == '\r') {
        cout << endl;
        return;
    }

    if (c == '\b') return;

    buffer.push_back(c);
    do {
        Node *child = root;
        system("cls");
        cout << buffer;
        if (root != nullptr) {
            string val = root->value();
            if (index == val.size()) {
                child = root->child(buffer.back()-'a');
                index = 0;
            }
            if (child != nullptr) {
                val = child->value();
                if (val[index] == buffer.back()) {
                    vector<string> entries;
                    get_entries(child, buffer + val.substr(index+1), entries);
                    for (auto &entry: entries) {
                        cout << endl << entry;
                    }
                }
            }
        }

        autocomplete(child, index+1);
    } while (c == '\0');
    if (c == '\b') c = '\0';
    buffer.pop_back();
    if (c == '\0' && buffer.empty()) {
        system("cls");
        autocomplete(root);
    }
}

void menu() {
    system("cls");
    cout << "1. Insert words" << endl;
    cout << "2. Remove words" << endl;
    cout << "3. Search a word" << endl;
    cout << "4. Display the tree" << endl;
    cout << "5. Auto-complete suggestion" << endl;
    cout << "0. Exit" << endl;
```

```
}

void main_menu() {
   menu();
   int choice;
   cin >> choice;
   while(choice != 0) {
      system("cls");
      if (choice == 1) {
         int n;
         cout << "Input the number of words you want to insert: ";
         cin >> n;
         cout << "Input the words (consist of lowercase latin letters only):" << endl;
         for (int i = 0; i < n; i++) {
            string word;
            cin >> word;
            tree.insert(word);
         }
      } else if (choice == 2) {
         int n;
         cout << "Input the number of words you want to remove: ";
         cin >> n;
         cout << "Input the words (consist of lowercase latin letters only):" << endl;
         for (int i = 0; i < n; i++) {
            string word;
            cin >> word;
            if (tree.search(word)) {
               tree.remove(word);
               cout << "Successfully removed!" << endl;
            } else
               cout << "The word you provided doesn't exist!" << endl;
         }
      } else if (choice == 3) {
         string word;
         cout << "Input a word: ";
         cin >> word;
         cout << (tree.search(word)? "Word exists!" : "Word doesn't exist!") << endl;
      } else if (choice == 4) {
         tree.display();
      } else if (choice == 5) {
         tree.autocomplete(tree.root());
      } else if (choice == 0) break;
      system("pause");
      menu();
      cin >> choice;
   }
}
```

```
int main() {
    main_menu();
}
```

CODE OVERVIEW

The essential features and functionalities of this project are:

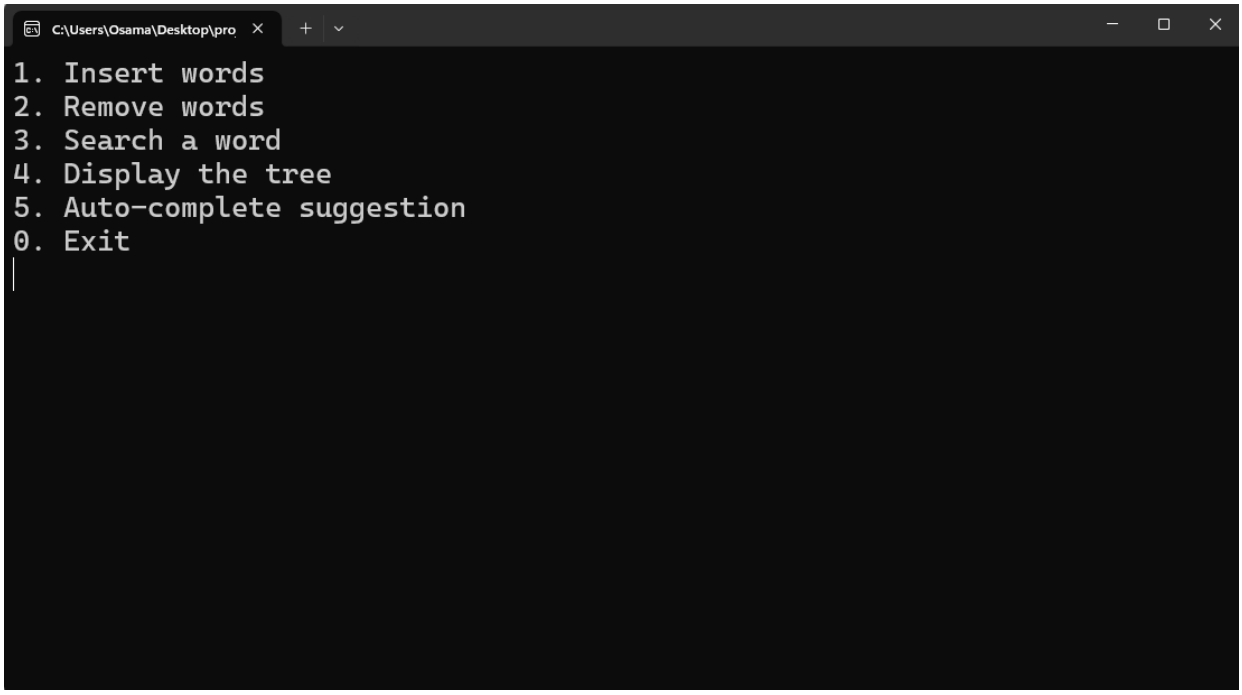
1. Insertion: The `insert` function of Radix Tree class inserts a string into the Radix Tree by traversing the tree, adding nodes as needed for each character in the input string.

2. Search: The `search` function of Radix Tree class checks whether a given string exists in the Radix Tree by traversing the tree and returning true if the string is found.

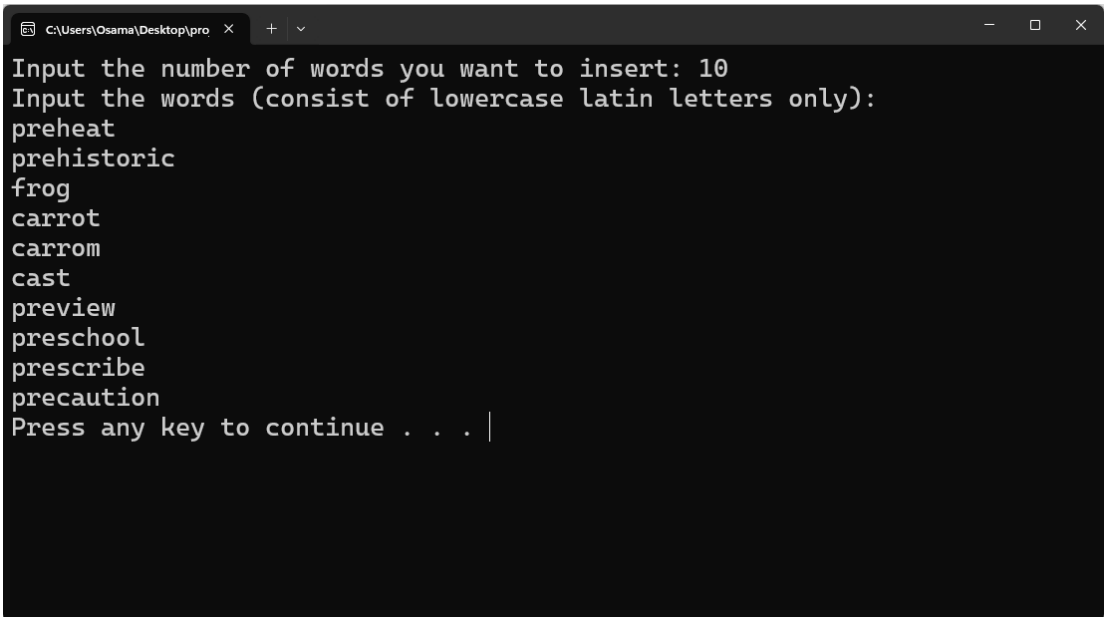
3. Deletion: The `remove` function deletes a string from the Radix Tree while maintaining the tree's integrity. It also removes unnecessary nodes that are no longer part of any valid string.

4. Displaying: The `display` function allows users to visualize the contents of the Radix Tree in a tree structure, displaying all inserted strings.

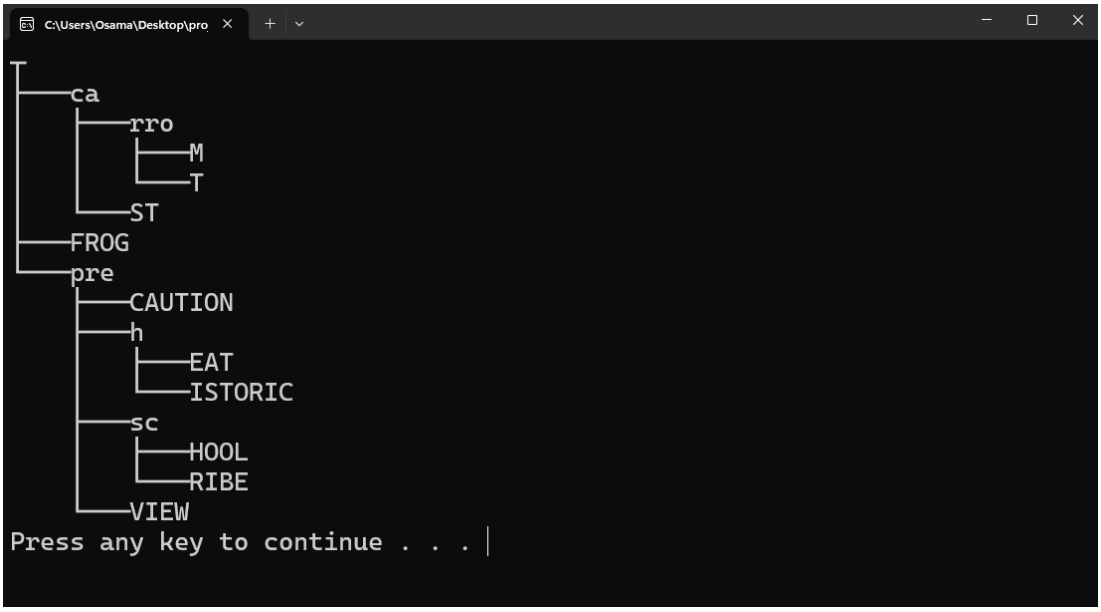
Output



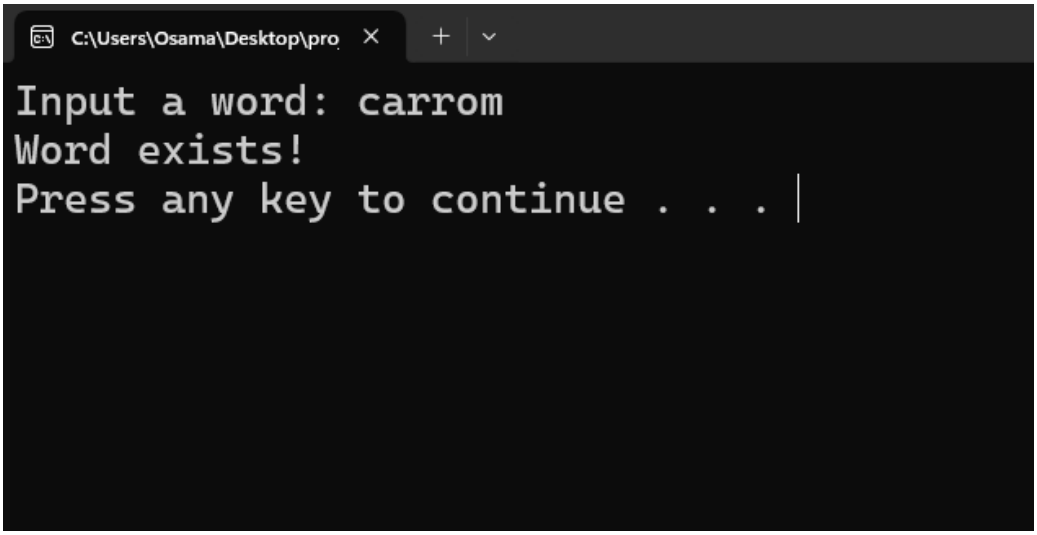
Insertion:



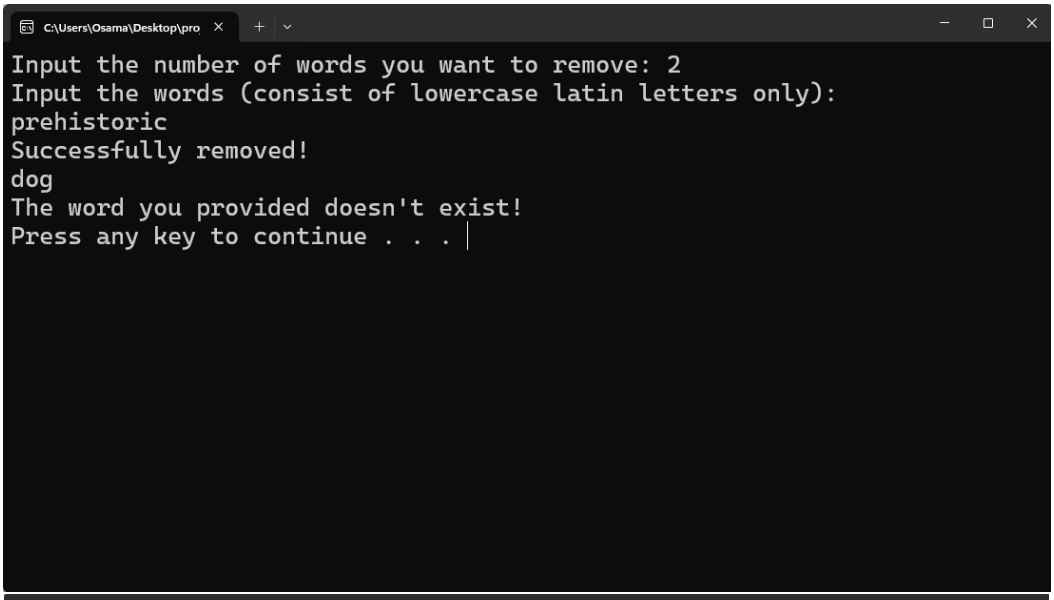
Display the Radix Tree:

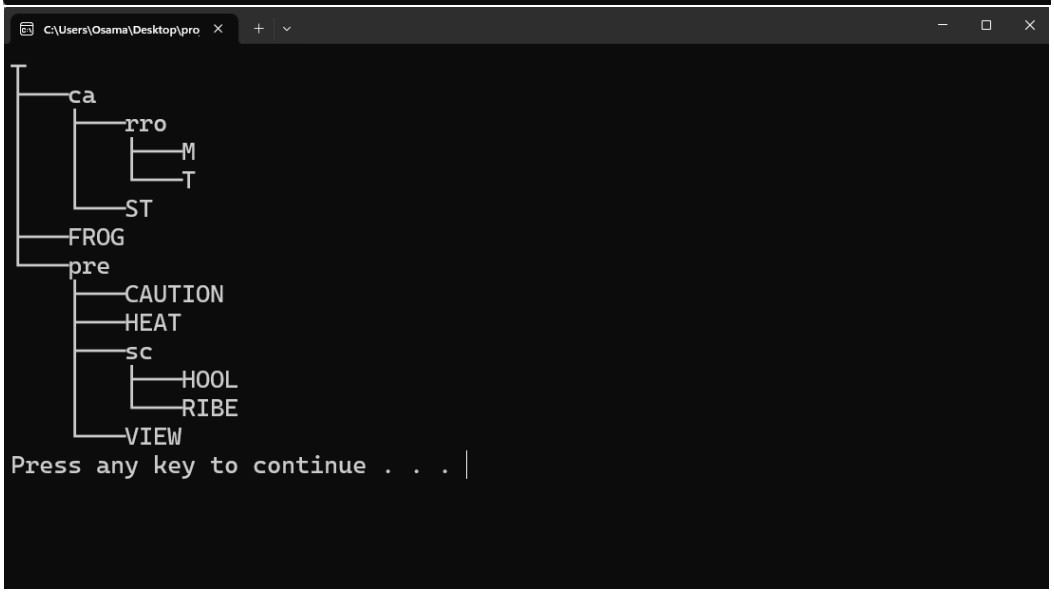


Searching a word in Radix Tree:

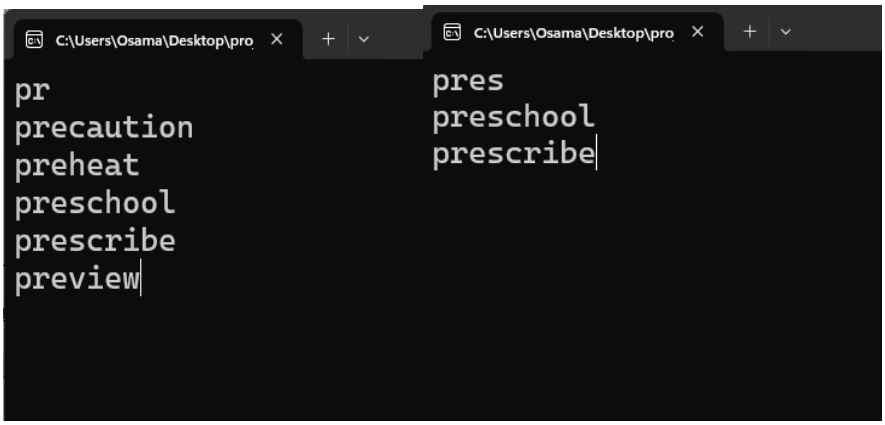


Deletion of a word from the Radix Tree and Display the final Radix Tree:





Auto-complete suggestions:



USAGE

The code provides a user-friendly menu-driven interface that allows users to interact with the Radix Tree efficiently. Users can perform the following operations:

- Insert words: Users can add strings/words to the tree.
- Delete words: Users can remove strings/words from the tree if it exists.
- Display the Radix Tree: Users can visualize the Radix Tree.
- Autocomplete suggestions: Users can see real-time suggestions from the Radix Tree as they type.
- Exit: Users can exit the program.

CONCLUSION

This project demonstrates the core functionality of a Radix Tree, emphasizing the insertion, deletion, and searching of strings. The user-friendly interface makes it easy to experiment with various string operations. Further enhancements could include additional features, optimizations, and error handling to create a more robust Radix Tree implementation. While other data structures exist for similar purposes, Radix Trees stand out for their unique ability to handle string-based keys with speed and efficiency. Understanding and leveraging the strengths of Radix Trees can benefit applications that require fast insertion, deletion, and search operations on dynamic sets of strings.