#include <stdio.h>

#include <stdlib.h>

// Structure for a node in the Splay tree

struct Node {

char key;

struct Node\* left;

struct Node\* right;

};

// Function to create a new node with a given key

struct Node\* createNode(char key) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->key = key;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function to perform a right rotation (Zig operation)

struct Node\* rightRotate(struct Node\* root) {

struct Node\* newRoot = root->left;

root->left = newRoot->right;

newRoot->right = root;

return newRoot;

}

// Function to perform a left rotation (Zag operation)

struct Node\* leftRotate(struct Node\* root) {

struct Node\* newRoot = root->right;

root->right = newRoot->left;

newRoot->left = root;

return newRoot;

}

// Function to perform Splay operation to move the node with the given key to the root

struct Node\* splay(struct Node\* root, char key) {

if (root == NULL || root->key == key) {

return root;

}

if (key < root->key) {

// Key is in the left subtree

if (root->left == NULL) {

return root;

}

// Zig-Zig case

if (key < root->left->key) {

root->left->left = splay(root->left->left, key);

root = rightRotate(root);

}

// Zig-Zag case

else if (key > root->left->key) {

root->left->right = splay(root->left->right, key);

if (root->left->right != NULL) {

root->left = leftRotate(root->left);

}

}

return (root->left == NULL) ? root : rightRotate(root);

} else {

// Key is in the right subtree

if (root->right == NULL) {

return root;

}

// Zag-Zig case

if (key < root->right->key) {

root->right->left = splay(root->right->left, key);

if (root->right->left != NULL) {

root->right = rightRotate(root->right);

}

}

// Zag-Zag case

else if (key > root->right->key) {

root->right->right = splay(root->right->right, key);

root = leftRotate(root);

}

return (root->right == NULL) ? root : leftRotate(root);

}

}

// Function to insert a key into the Splay tree

struct Node\* insert(struct Node\* root, char key) {

if (root == NULL) {

return createNode(key);

}

// Perform Splay to bring the key to the root

root = splay(root, key);

// If the key is already at the root, return the root

if (root->key == key) {

return root;

}

struct Node\* newNode = createNode(key);

if (key < root->key) {

newNode->right = root;

newNode->left = root->left;

root->left = NULL;

} else {

newNode->left = root;

newNode->right = root->right;

root->right = NULL;

}

return newNode;

}

// Function to print the Splay tree in-order

void inOrder(struct Node\* root) {

if (root != NULL) {

inOrder(root->left);

printf("%c ", root->key);

inOrder(root->right);

}

}

int main() {

struct Node\* root = NULL;

char keys[] = {'F', 'S', 'Q', 'K', 'C', 'L', 'H', 'T', 'V', 'W', 'M', 'R', 'N', 'P', 'A', 'B', 'X', 'Y', 'D', 'Z', 'E'};

int i;

for (i = 0; i < sizeof(keys) / sizeof(keys[0]); i++) {

root = insert(root, keys[i]);

printf("Inserted: %c\n", keys[i]);

}

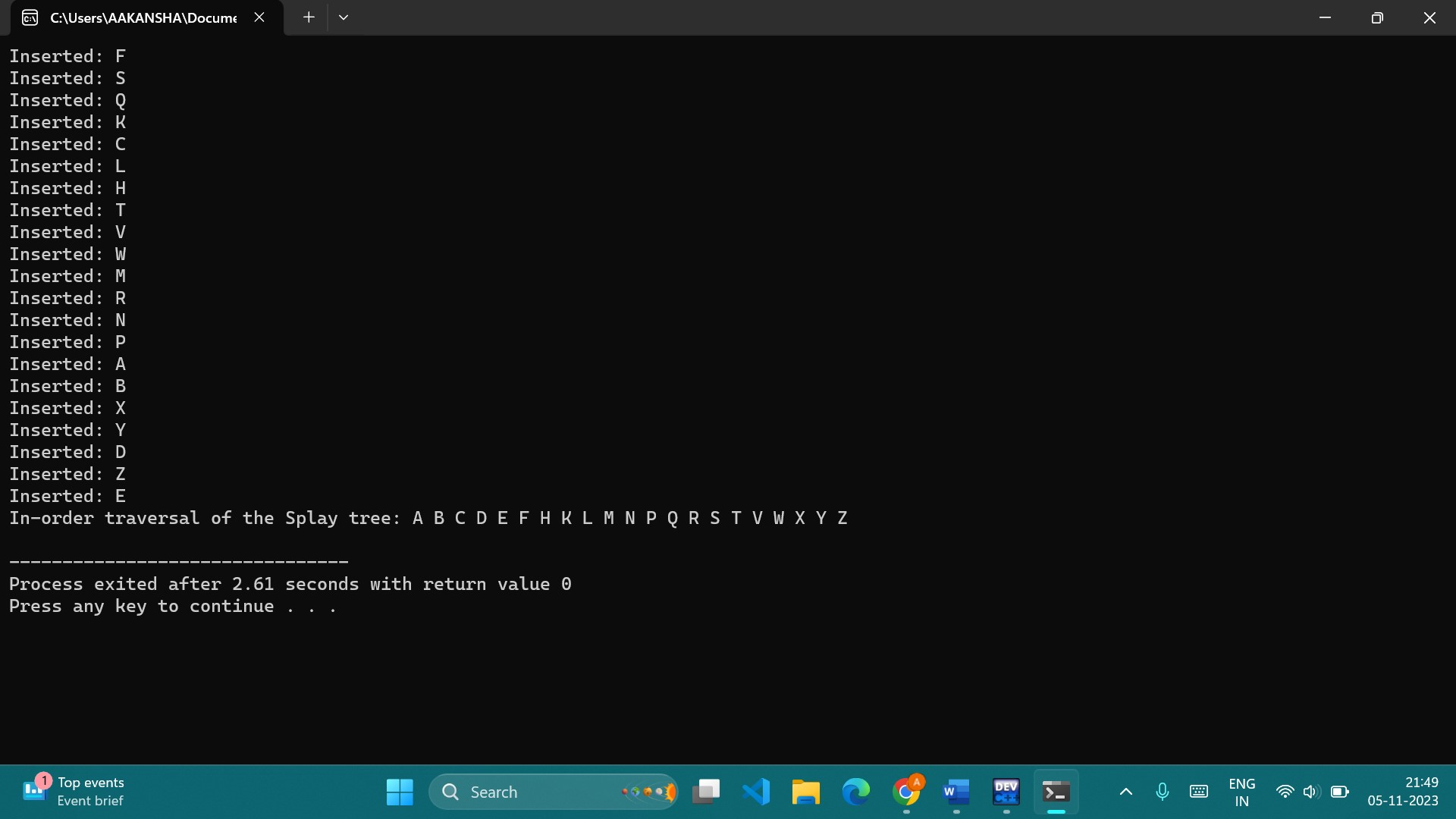
printf("In-order traversal of the Splay tree: ");

inOrder(root);

printf("\n");

return 0;

}



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define ALPHABET\_SIZE 26

struct TrieNode {

struct TrieNode\* children[ALPHABET\_SIZE];

int isEndOfWord;

};

struct TrieNode\* createNode() {

struct TrieNode\* newNode = (struct TrieNode\*)malloc(sizeof(struct TrieNode));

for (int i = 0; i < ALPHABET\_SIZE; i++) {

newNode->children[i] = NULL;

}

newNode->isEndOfWord = 0;

return newNode;

}

void insert(struct TrieNode\* root, const char\* word) {

struct TrieNode\* current = root;

for (int i = 0; word[i] != '\0'; i++) {

int index = word[i] - 'a';

if (!current->children[index]) {

current->children[index] = createNode();

}

current = current->children[index];

}

current->isEndOfWord = 1;

}

int search(struct TrieNode\* root, const char\* word) {

struct TrieNode\* current = root;

for (int i = 0; word[i] != '\0'; i++) {

int index = word[i] - 'a';

if (!current->children[index]) {

return 0; // Word not found

}

current = current->children[index];

}

return (current != NULL && current->isEndOfWord);

}

int main() {

struct TrieNode\* root = createNode();

const char\* words[] = {"apple", "banana", "cherry", "date", "grape"};

for (int i = 0; i < 5; i++) {

insert(root, words[i]);

}

const char\* searchWords[] = {"banana", "apple", "kiwi", "date"};

for (int i = 0; i < 4; i++) {

if (search(root, searchWords[i])) {

printf("%s found in the trie.\n", searchWords[i]);

} else {

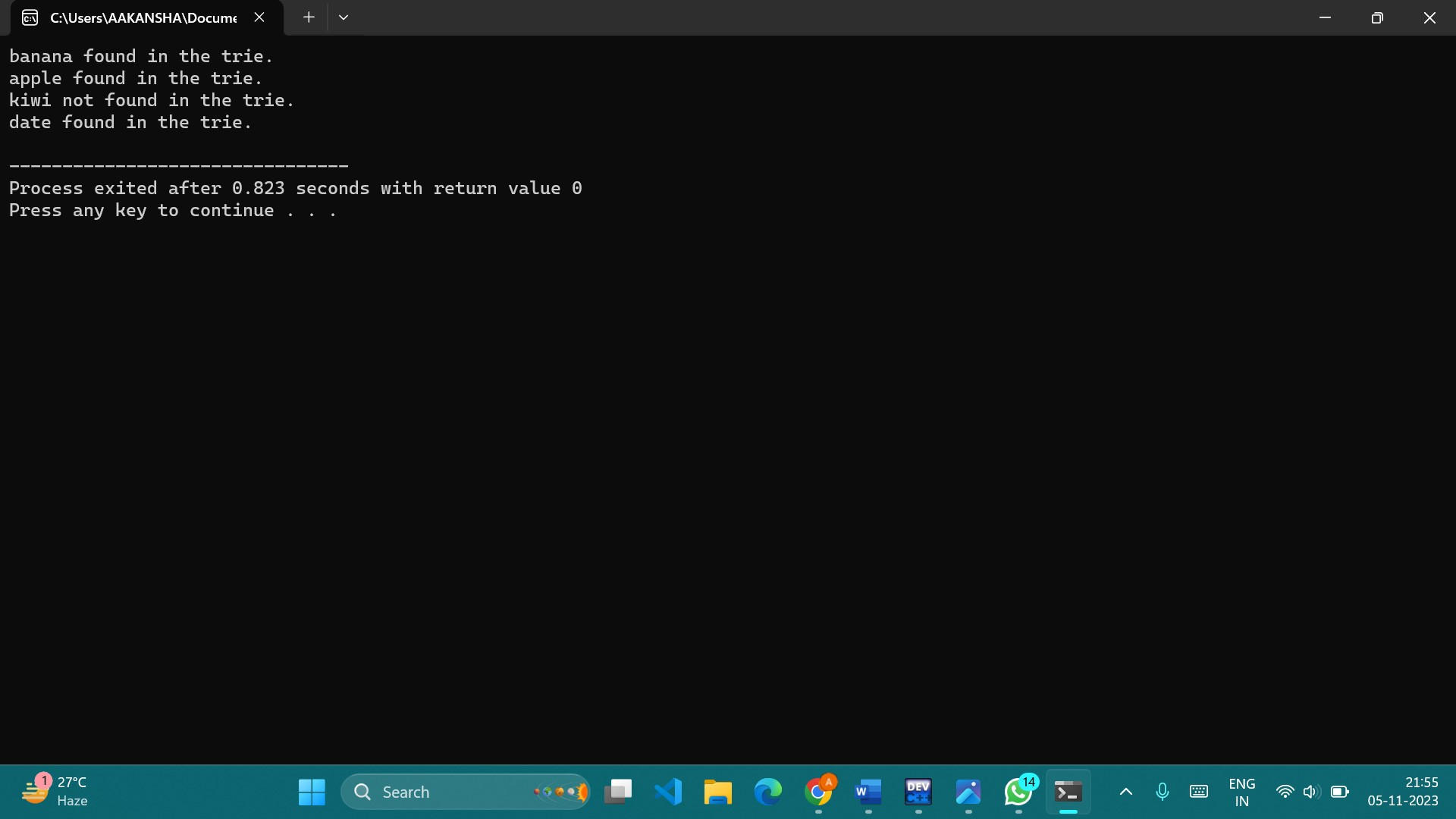
printf("%s not found in the trie.\n", searchWords[i]);

}

}

return 0;

}



3. void delete(struct TrieNode\* root, const char\* word) {

struct TrieNode\* current = root;

for (int i = 0; word[i] != '\0'; i++) {

int index = word[i] - 'a';

if (!current->children[index]) {

return; // Word not found

}

current = current->children[index];

}

if (current != NULL && current->isEndOfWord) {

current->isEndOfWord = 0;

}

}

void suggestionsUtil(struct TrieNode\* root, char prefix[], int level) {

if (root->isEndOfWord) {

prefix[level] = '\0';

printf("%s\n", prefix);

}

for (int i = 0; i < ALPHABET\_SIZE; i++) {

if (root->children[i]) {

prefix[level] = i + 'a';

suggestionsUtil(root->children[i], prefix, level + 1);

}

}

}

void autocomplete(struct TrieNode\* root, const char\* prefix) {

struct TrieNode\* current = root;

int level;

int length = strlen(prefix);

for (level = 0; level < length; level++) {

int index = prefix[level] - 'a';

if (!current->children[index]) {

printf("No suggestions found for %s\n", prefix);

return;

}

current = current->children[index];

}

char suggestion[100];

strcpy(suggestion, prefix);

suggestionsUtil(current, suggestion, level);

}

int main() {

struct TrieNode\* root = createNode();

const char\* words[] = {"apple", "banana", "cherry", "date", "grape"};

for (int i = 0; i < 5; i++) {

insert(root, words[i]);

}

char prefix[] = "da";

printf("Autocomplete suggestions for '%s':\n", prefix);

autocomplete(root, prefix);

return 0;

}

5. #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define ALPHABET\_SIZE 26

struct SuffixNode {

struct SuffixNode\* children[ALPHABET\_SIZE];

};

struct SuffixNode\* createNode() {

struct SuffixNode\* newNode = (struct SuffixNode\*)malloc(sizeof(struct SuffixNode));

for (int i = 0; i < ALPHABET\_SIZE; i++) {

newNode->children[i] = NULL;

}

return newNode;

}

void insertSuffix(struct SuffixNode\* root, const char\* suffix) {

struct SuffixNode\* current = root;

for (int i = 0; suffix[i] != '\0'; i++) {

int index = suffix[i] - 'a';

if (!current->children[index]) {

current->children[index] = createNode();

}

current = current->children[index];

}

}

int searchSuffix(struct SuffixNode\* root, const char\* suffix) {

struct SuffixNode\* current = root;

for (int i = 0; suffix[i] != '\0'; i++) {

int index = suffix[i] - 'a';

if (!current->children[index]) {

return 0; // Suffix not found

}

current = current->children[index];

}

return 1; // Suffix found

}

int main() {

struct SuffixNode\* root = createNode();

const char\* input = "banana";

int length = strlen(input);

// Build the suffix tree-like structure

for (int i = 0; i < length; i++) {

insertSuffix(root, input + i);

}

// Search for suffixes

const char\* searchSuffixes[] = {"ana", "nana", "apple"};

for (int i = 0; i < 3; i++) {

if (searchSuffix(root, searchSuffixes[i])) {

printf("%s found in the suffix tree.\n", searchSuffixes[i]);

} else {

printf("%s not found in the suffix tree.\n", searchSuffixes[i]);

}

}

return 0;

}

