|  |
| --- |
| **SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES** |
| **COMPUTER SCIENCE AND ENGINEERING PROGRAMME** |

**SUB CODE: CSA0392 SUB NAME: Data Structures for Hashing Techniques**

**LIST OF PROGRAMS**

**DATE : 20.08.2024**

**Lab Questions to be practiced with test cases**

1. Write a C program to Implement a 2-3-4 tree as an extension of the 2-3 tree.

Answer:

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a 2-3-4 tree node

typedef struct Node {

int keys[3]; // Keys in the node

struct Node\* children[4]; // Pointers to children

struct Node\* parent; // Pointer to parent

int numKeys; // Number of keys in the node

int isLeaf; // 1 if leaf, 0 if internal node

} Node;

// Function prototypes

Node\* createNode(int isLeaf);

void traverse(Node\* root);

Node\* insert(Node\* root, int key);

void splitChild(Node\* parent, int index, Node\* fullChild);

void insertNonFull(Node\* node, int key);

void display(Node\* root, int level);

// Create a new 2-3-4 tree node

Node\* createNode(int isLeaf) {

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

newNode->numKeys = 0;

newNode->isLeaf = isLeaf;

newNode->parent = NULL;

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

newNode->children[i] = NULL;

}

return newNode;

}

// Traverse and print the 2-3-4 tree (in-order traversal)

void traverse(Node\* root) {

if (root != NULL) {

int i;

for (i = 0; i < root->numKeys; i++) {

if (!root->isLeaf) {

traverse(root->children[i]);

}

printf("%d ", root->keys[i]);

}

if (!root->isLeaf) {

traverse(root->children[i]);

}

}

}

// Split a full child node into two nodes

void splitChild(Node\* parent, int index, Node\* fullChild) {

Node\* newChild = createNode(fullChild->isLeaf);

parent->children[index + 1] = newChild;

parent->keys[index] = fullChild->keys[1];

parent->numKeys++;

newChild->numKeys = 1;

fullChild->numKeys = 1;

for (int i = 0; i < 1; i++) {

newChild->keys[i] = fullChild->keys[i + 2];

}

if (!fullChild->isLeaf) {

for (int i = 0; i < 2; i++) {

newChild->children[i] = fullChild->children[i + 2];

}

}

}

// Insert a new key into a non-full node

void insertNonFull(Node\* node, int key) {

int i = node->numKeys - 1;

if (node->isLeaf) {

while (i >= 0 && key < node->keys[i]) {

node->keys[i + 1] = node->keys[i];

i--;

}

node->keys[i + 1] = key;

node->numKeys++;

} else {

while (i >= 0 && key < node->keys[i]) {

i--;

}

i++;

if (node->children[i]->numKeys == 3) {

splitChild(node, i, node->children[i]);

if (key > node->keys[i]) {

i++;

}

}

insertNonFull(node->children[i], key);

}

}

// Insert a key into the 2-3-4 tree

Node\* insert(Node\* root, int key) {

Node\* r = root;

if (r->numKeys == 3) {

Node\* s = createNode(0);

s->children[0] = r;

r->parent = s;

splitChild(s, 0, r);

root = s;

}

insertNonFull(root, key);

return root;

}

// Display the 2-3-4 tree structure

void display(Node\* root, int level) {

if (root) {

printf("Level %d: ", level);

for (int i = 0; i < root->numKeys; i++) {

printf("%d ", root->keys[i]);

}

printf("\n");

if (!root->isLeaf) {

for (int i = 0; i <= root->numKeys; i++) {

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

}

}

}

}

// Main function to demonstrate 2-3-4 tree operations

int main() {

Node\* root = createNode(1); // Start with an empty leaf node

// Inserting elements into the 2-3-4 tree

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 5);

root = insert(root, 6);

root = insert(root, 15);

root = insert(root, 30);

root = insert(root, 25);

root = insert(root, 35);

printf("2-3-4 Tree elements in sorted order: ");

traverse(root);

printf("\n");

printf("Displaying 2-3-4 Tree structure:\n");

display(root, 0);

return 0;

}

1. Write a C program to perform the following operations:

a) Insert an element into a Splay tree.

b) Delete an element from a Splay tree.

c) Search for a key element in a Splay tree.

Answer:

#include <stdio.h>

#include <stdlib.h>

// Define the structure for a Splay Tree node

typedef struct SplayNode {

int key;

struct SplayNode\* left;

struct SplayNode\* right;

} SplayNode;

// Function prototypes

SplayNode\* createNode(int key);

SplayNode\* rightRotate(SplayNode\* root);

SplayNode\* leftRotate(SplayNode\* root);

SplayNode\* splay(SplayNode\* root, int key);

SplayNode\* insert(SplayNode\* root, int key);

SplayNode\* deleteNode(SplayNode\* root, int key);

SplayNode\* search(SplayNode\* root, int key);

void inorder(SplayNode\* root);

// Create a new Splay Tree node

SplayNode\* createNode(int key) {

SplayNode\* newNode = (SplayNode\*)malloc(sizeof(SplayNode));

newNode->key = key;

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

return newNode;

}

// Perform a right rotation on the root

SplayNode\* rightRotate(SplayNode\* root) {

SplayNode\* newRoot = root->left;

root->left = newRoot->right;

newRoot->right = root;

return newRoot;

}

// Perform a left rotation on the root

SplayNode\* leftRotate(SplayNode\* root) {

SplayNode\* newRoot = root->right;

root->right = newRoot->left;

newRoot->left = root;

return newRoot;

}

// Splay the tree to bring the key to the root

SplayNode\* splay(SplayNode\* root, int key) {

if (root == NULL) return NULL;

if (key < root->key) {

if (root->left == NULL) return root;

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

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

root = rightRotate(root);

} 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 if (key > root->key) {

if (root->right == NULL) return root;

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

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

root = leftRotate(root);

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

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

if (root->right->left != NULL) root->right = rightRotate(root->right);

}

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

} else {

return root;

}

}

// Insert a key into the Splay Tree

SplayNode\* insert(SplayNode\* root, int key) {

if (root == NULL) return createNode(key);

root = splay(root, key);

if (root->key == key) return root;

SplayNode\* 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;

}

// Delete a key from the Splay Tree

SplayNode\* deleteNode(SplayNode\* root, int key) {

if (root == NULL) return NULL;

root = splay(root, key);

if (root->key != key) return root;

SplayNode\* newRoot;

if (root->left == NULL) {

newRoot = root->right;

} else {

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

newRoot->right = root->right;

}

free(root);

return newRoot;

}

// Search for a key in the Splay Tree

SplayNode\* search(SplayNode\* root, int key) {

return splay(root, key);

}

// In-order traversal of the Splay Tree

void inorder(SplayNode\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

// Main function to demonstrate Splay Tree operations

int main() {

SplayNode\* root = NULL;

// Inserting elements into the Splay Tree

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 5);

root = insert(root, 6);

root = insert(root, 15);

root = insert(root, 30);

root = insert(root, 25);

root = insert(root, 35);

printf("Splay Tree in-order traversal: ");

inorder(root);

printf("\n");

// Search for a key

root = search(root, 15);

printf("After searching for 15, root key: %d\n", root->key);

// Delete a key

root = deleteNode(root, 10);

printf("Splay Tree in-order traversal after deleting 10: ");

inorder(root);

printf("\n");

return 0;

}

1. Implement a Trie data structure to store a set of strings. The Trie should support the following operations:

Insert: Insert a string into the Trie.

Search: Search for a given string in the Trie.

Delete: Delete a string from the Trie.

Prefix Search: Find all words in the Trie that start with a given prefix.

Answer:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define ALPHABET\_SIZE 26

// Trie node structure

typedef struct TrieNode {

struct TrieNode\* children[ALPHABET\_SIZE];

bool isEndOfWord;

} TrieNode;

// Create a new Trie node

TrieNode\* createNode() {

TrieNode\* node = (TrieNode\*)malloc(sizeof(TrieNode));

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

node->children[i] = NULL;

}

node->isEndOfWord = false;

return node;

}

// Insert a string into the Trie

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

TrieNode\* node = root;

while (\*key) {

int index = \*key - 'a';

if (node->children[index] == NULL) {

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

}

node = node->children[index];

key++;

}

node->isEndOfWord = true;

}

// Search for a string in the Trie

bool search(TrieNode\* root, const char\* key) {

TrieNode\* node = root;

while (\*key) {

int index = \*key - 'a';

if (node->children[index] == NULL) {

return false;

}

node = node->children[index];

key++;

}

return node != NULL && node->isEndOfWord;

}

// Delete a string from the Trie

bool deleteHelper(TrieNode\* root, const char\* key, int depth) {

if (root == NULL) return false;

if (\*key == '\0') {

if (!root->isEndOfWord) return false;

root->isEndOfWord = false;

return true;

}

int index = \*key - 'a';

if (deleteHelper(root->children[index], key + 1, depth + 1)) {

free(root->children[index]);

root->children[index] = NULL;

return !root->isEndOfWord && root->children[index] == NULL;

}

return false;

}

bool delete(TrieNode\* root, const char\* key) {

return deleteHelper(root, key, 0);

}

// Prefix search - collect all words with the given prefix

void collectWords(TrieNode\* root, char\* prefix, int length) {

if (root == NULL) return;

if (root->isEndOfWord) {

prefix[length] = '\0';

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

}

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

if (root->children[i]) {

prefix[length] = i + 'a';

collectWords(root->children[i], prefix, length + 1);

}

}

}

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

TrieNode\* node = root;

while (\*prefix) {

int index = \*prefix - 'a';

if (node->children[index] == NULL) return;

node = node->children[index];

prefix++;

}

char buffer[100];

collectWords(node, buffer, 0);

}

// Main function to demonstrate Trie operations

int main() {

TrieNode\* root = createNode();

// Insert words

insert(root, "hello");

insert(root, "hell");

insert(root, "he");

insert(root, "heat");

// Search for words

printf("Searching 'hello': %s\n", search(root, "hello") ? "Found" : "Not Found");

printf("Searching 'hell': %s\n", search(root, "hell") ? "Found" : "Not Found");

printf("Searching 'he': %s\n", search(root, "he") ? "Found" : "Not Found");

// Delete a word

delete(root, "hello");

printf("Searching 'hello' after deletion: %s\n", search(root, "hello") ? "Found" : "Not Found");

// Prefix search

printf("Words with prefix 'he':\n");

prefixSearch(root, "he");

// Free memory (not shown here, but important for real applications)

return 0;

}