**Ds day12**

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

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int \*keys;

struct Node \*\*children;

int numKeys;

int isLeaf;

} Node;

Node\* createNode(int isLeaf) {

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

newNode->keys = (int\*)malloc(3 \* sizeof(int));

newNode->children = (Node\*\*)malloc(4 \* sizeof(Node\*));

newNode->numKeys = 0;

newNode->isLeaf = isLeaf;

return newNode;

}

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

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

newChild->numKeys = 1;

newChild->keys[0] = child->keys[2];

if (!child->isLeaf) {

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

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

}

}

child->numKeys = 1;

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

for (int i = parent->numKeys; i > index; i--) {

parent->children[i + 1] = parent->children[i];

}

parent->children[index] = child;

for (int i = parent->numKeys - 1; i >= index; i--) {

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

}

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

parent->numKeys++;

}

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);

}

}

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

if ((\*root)->numKeys == 3) {

Node\* newRoot = createNode(0);

newRoot->children[0] = \*root;

splitChild(newRoot, 0, \*root);

int i = 0;

if (newRoot->keys[0] < key) {

i++;

}

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

\*root = newRoot;

} else {

insertNonFull(\*root, key);

}

}

void traverse(Node\* root) {

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

if (!root->isLeaf) {

traverse(root->children[i]);

}

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

}

if (!root->isLeaf) {

traverse(root->children[root->numKeys]);

}

}

int main() {

Node\* root = createNode(1);

insert(&root, 10);

insert(&root, 20);

insert(&root, 5);

insert(&root, 6);

insert(&root, 12);

insert(&root, 30);

insert(&root, 7);

insert(&root, 17);

printf("Traversal of the 2-3-4 tree is: ");

traverse(root);

return 0;

}

Output:



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.

#include <stdio.h>

#include <stdlib.h>

struct Node {

int key;

struct Node \*left, \*right;

};

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

struct Node \*y = x->left;

x->left = y->right;

y->right = x;

return y;

}

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

struct Node \*y = x->right;

x->right = y->left;

y->left = x;

return y;

}

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

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

return root;

if (root->key > key) {

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

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

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

root = rightRotate(root);

} else if (root->left->key < 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 (root->right == NULL) return root;

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

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

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

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

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

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

root = leftRotate(root);

}

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

}

}

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

if (root == NULL) {

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

newNode->key = key;

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

return newNode;

}

root = splay(root, key);

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

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

newNode->key = key;

if (root->key > key) {

newNode->right = root;

newNode->left = root->left;

root->left = NULL;

} else {

newNode->left = root;

newNode->right = root->right;

root->right = NULL;

}

return newNode;

}

struct Node\* delete(struct Node \*root, int key) {

struct Node \*temp;

if (root == NULL) return NULL;

root = splay(root, key);

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

if (root->left == NULL) {

temp = root;

root = root->right;

} else {

temp = root;

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

root->right = temp->right;

}

free(temp);

return root;

}

struct Node\* search(struct Node \*root, int key) {

return splay(root, key);

}

void inorder(struct Node \*root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

struct Node \*root = NULL;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

printf("Inorder traversal of the Splay tree: ");

inorder(root);

printf("\n");

root = delete(root, 30);

printf("Inorder traversal after deleting 30: ");

inorder(root);

printf("\n");

struct Node \*found = search(root, 20);

if (found != NULL) {

printf("Found key: %d\n", found->key);

} else {

printf("Key not found.\n");

}

return 0;

}

Output:



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:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define ALPHABET\_SIZE 26

typedef struct TrieNode {

struct TrieNode \*children[ALPHABET\_SIZE];

int isEndOfWord;

} TrieNode;

TrieNode\* createNode() {

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

node->isEndOfWord = 0;

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

node->children[i] = NULL;

}

return node;

}

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

TrieNode\* pCrawl = root;

for (int level = 0; level < strlen(key); level++) {

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

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

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

}

pCrawl = pCrawl->children[index];

}

pCrawl->isEndOfWord = 1;

}

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

TrieNode\* pCrawl = root;

for (int level = 0; level < strlen(key); level++) {

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

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

return 0;

}

pCrawl = pCrawl->children[index];

}

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

}

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

if (!root) {

return 0;

}

if (depth == strlen(key)) {

if (root->isEndOfWord) {

root->isEndOfWord = 0;

return (root->children[0] == NULL);

}

return 0;

}

int index = key[depth] - 'a';

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

free(root->children[index]);

root->children[index] = NULL;

return (root->isEndOfWord == 0 && root->children[0] == NULL);

}

return 0;

}

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

deleteHelper(root, key, 0);

}

void prefixSearchHelper(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';

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

}

}

}

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

TrieNode\* pCrawl = root;

for (int level = 0; level < strlen(prefix); level++) {

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

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

printf("No words found with the given prefix.\n");

return;

}

pCrawl = pCrawl->children[index];

}

char buffer[100];

prefixSearchHelper(pCrawl, buffer, 0);

}

int main() {

TrieNode\* root = createNode();

insert(root, "hello");

insert(root, "helium");

insert(root, "hero");

insert(root, "heron");

printf("Search for 'hello': %d\n", search(root, "hello"));

printf("Search for 'her': %d\n", search(root, "her"));

delete(root, "hero");

printf("Search for 'hero' after deletion: %d\n", search(root, "hero"));

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

prefixSearch(root, "he");

return 0;

}

Output:

