DAY 12 LAB:

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>

#define MAX\_KEYS 3

#define MIN\_KEYS 1

typedef struct Node {

int keys[MAX\_KEYS + 1];

struct Node \*children[MAX\_KEYS + 2];

int numKeys;

int isLeaf;

} Node;

Node\* createNode(int isLeaf) {

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

node->numKeys = 0;

node->isLeaf = isLeaf;

for (int i = 0; i < MAX\_KEYS + 2; i++)

node->children[i] = NULL;

return node;

}

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

Node \*fullChild = parent->children[index];

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

int median = 1;

newChild->numKeys = MIN\_KEYS;

for (int i = 0; i < MIN\_KEYS; i++)

newChild->keys[i] = fullChild->keys[i + MIN\_KEYS + 1];

if (!fullChild->isLeaf) {

for (int i = 0; i <= MIN\_KEYS; i++)

newChild->children[i] = fullChild->children[i + MIN\_KEYS + 1];

}

fullChild->numKeys = MIN\_KEYS;

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

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

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

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

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

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

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 == MAX\_KEYS) {

splitChild(node, i);

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

i++;

}

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

}

}

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

Node \*r = \*root;

if (r->numKeys == MAX\_KEYS) {

Node \*s = createNode(0);

\*root = s;

s->children[0] = r;

splitChild(s, 0);

insertNonFull(s, key);

} else {

insertNonFull(r, key);

}

}

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

if (root == NULL)

return;\

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++)

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

}

}

int main() {

Node \*root = createNode(1);

insert(&root, 10);

insert(&root, 20);

insert(&root, 5);

insert(&root, 6);

insert(&root, 15);

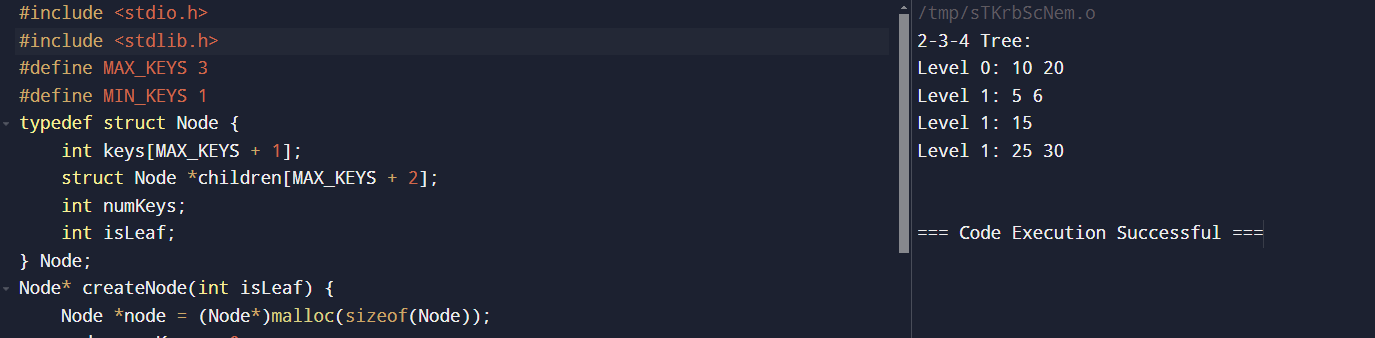
insert(&root, 30);

insert(&root, 25);

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

printTree(root, 0);

}



2.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>

typedef struct SplayTreeNode {

int key;

struct SplayTreeNode \*left, \*right;

} SplayTreeNode;

SplayTreeNode\* rightRotate(SplayTreeNode\* root) {

SplayTreeNode\* newRoot = root->left;

root->left = newRoot->right;

newRoot->right = root;

return newRoot;

}

SplayTreeNode\* leftRotate(SplayTreeNode\* root) {

SplayTreeNode\* newRoot = root->right;

root->right = newRoot->left;

newRoot->left = root;

return newRoot;

}

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

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

return root;

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 (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);

}

}

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

if (root == NULL) {

root = (SplayTreeNode\*)malloc(sizeof(SplayTreeNode));

root->key = key;

root->left = root->right = NULL;

return root;

}

root = splay(root, key);

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

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

newNode->key = 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;

}

SplayTreeNode\* minValueNode(SplayTreeNode\* root) {

SplayTreeNode\* current = root;

while (current && current->left != NULL)

current = current->left;

return current;

}

SplayTreeNode\* delete(SplayTreeNode\* root, int key) {

if (root == NULL) return NULL;

root = splay(root, key);

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

if (root->left == NULL) {

SplayTreeNode\* temp = root;

root = root->right;

free(temp);

} else {

SplayTreeNode\* temp = root;

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

root->right = temp->right;

free(temp);

}

return root;

}

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

return splay(root, key);

}

void printTree(SplayTreeNode\* root) {

if (root != NULL) {

printTree(root->left);

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

printTree(root->right);

}

}

int main() {

SplayTreeNode\* root = NULL;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 5);

root = insert(root, 6);

root = insert(root, 15);

printf("Splay Tree after insertions:\n");

printTree(root);

printf("\n");

root = delete(root, 10);

printf("Splay Tree after deleting 10:\n");

printTree(root);

printf("\n");

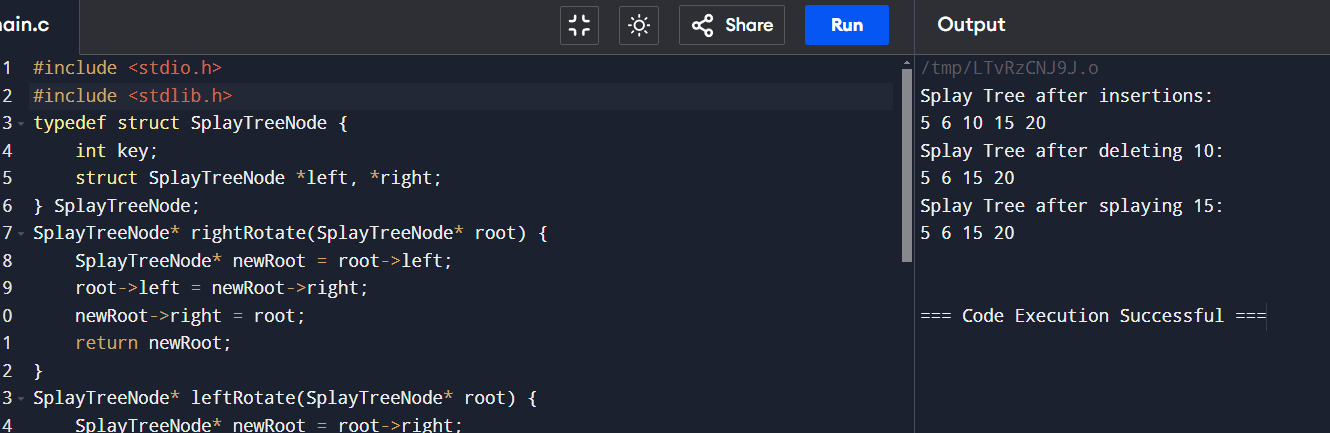
root = search(root, 15);

printf("Splay Tree after splaying 15:\n");

printTree(root);

printf("\n");

}



3.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 \*word) {

TrieNode \*node = root;

while (\*word) {

int index = \*word - 'a';

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

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

node = node->children[index];

word++;

}

node->isEndOfWord = 1;

}

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

TrieNode \*node = root;

while (\*word) {

int index = \*word - 'a'; // Convert character to index

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

return 0;

node = node->children[index];

word++;

}

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

}

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

if (root == NULL)

return 0;

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

if (root->isEndOfWord)

root->isEndOfWord = 0;

if (hasNoChildren(root)) {

free(root);

return 1;

}

return 0;

}

int index = \*word - 'a';

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

free(root->children[index]);

root->children[index] = NULL;

return (root->isEndOfWord == 0 && hasNoChildren(root));

}

return 0;

}

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

deleteHelper(root, word, 0);

}

int hasNoChildren(TrieNode \*node) {

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

if (node->children[i] != NULL)

return 0;

return 1;

}

void printWords(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] != NULL) {

prefix[length] = i + 'a';

printWords(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 word[100];

strcpy(word, prefix);

printWords(node, word, strlen(prefix));

}

int main() {

TrieNode \*root = createNode();

insert(root, "hello");

insert(root, "hell");

insert(root, "heaven");

insert(root, "heavy");

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

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

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

prefixSearch(root, "he");

delete(root, "hello");

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

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

prefixSearch(root, "he");

}

