Ads lab exam

1.1 Write C or C++ program to represent binary tree or BST using array.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

char tree[MAX\_SIZE];

void initializeTree() {

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

tree[i] = '\0';

}

}

int root(char key) {

if (tree[0] != '\0') {

printf("Tree already has root\n");

} else {

tree[0] = key;

}

return 0;

}

int insertNode(char key) {

int i = 0;

while (i < MAX\_SIZE && tree[i] != '\0') {

if (key < tree[i]) {

i = (2 \* i) + 1;

} else {

i = (2 \* i) + 2;

}

}

if (i < MAX\_SIZE) {

tree[i] = key;

} else {

printf("Cannot insert, tree is full\n");

}

return 0;

}

void inOrderTraversal(int index) {

if (index < MAX\_SIZE && tree[index] != '\0') {

inOrderTraversal((2 \* index) + 1);

printf("%c ", tree[index]);

inOrderTraversal((2 \* index) + 2);

}

}

void preOrderTraversal(int index) {

if (index < MAX\_SIZE && tree[index] != '\0') {

printf("%c ", tree[index]);

preOrderTraversal((2 \* index) + 1);

preOrderTraversal((2 \* index) + 2);

}

}

void postOrderTraversal(int index) {

if (index < MAX\_SIZE && tree[index] != '\0') {

postOrderTraversal((2 \* index) + 1);

postOrderTraversal((2 \* index) + 2);

printf("%c ", tree[index]);

}

}

int printTree() {

printf("\n");

for (int i = 0; i < 30; i++) {

if (tree[i] != '\0') {

printf("%c ", tree[i]);

} else {

printf("- ");

}

}

printf("\n");

return 0;

}

int main() {

initializeTree();

int choice;

char key;

do {

printf("\nBinary Search Tree using array:\n");

printf("1. Set Root\n");

printf("2. Insert Node\n");

printf("3. In-order Traversal\n");

printf("4. Pre-order Traversal\n");

printf("5. Post-order Traversal\n");

printf("6. Print Tree\n");

printf("7. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the root element: ");

scanf(" %c", &key);

root(key);

printf("\nRoot element inserted\n");

break;

case 2:

printf("Enter the element to be inserted: ");

scanf(" %c", &key);

insertNode(key);

printf("\nElement inserted\n");

break;

case 3:

printf("In-order traversal: ");

inOrderTraversal(0);

printf("\n");

break;

case 4:

printf("Pre-order traversal: ");

preOrderTraversal(0);

printf("\n");

break;

case 5:

printf("Post-order traversal: ");

postOrderTraversal(0);

printf("\n");

break;

case 6:

printf("Printing tree: ");

printTree();

break;

case 7:

printf("Exiting program...\n");

break;

default:

printf("Invalid choice, please try again\n");

}

} while (choice != 7);

return 0;

}

1.2 Write C or C++ program to represent binary tree or BST using linked list.

#include <stdio.h>

#include <stdlib.h>

struct binarytree {

struct binarytree \*left;

int data;

struct binarytree \*right;

};

struct binarytree \*root = NULL;

struct binarytree \*createNode(int data) {

struct binarytree \*newnode = (struct binarytree \*)malloc(sizeof(struct binarytree));

newnode->data = data;

newnode->left = NULL;

newnode->right = NULL;

return newnode;

}

struct binarytree \*insertNode(struct binarytree \*node, int data) {

if (node == NULL) {

return createNode(data);

}

if (data < node->data) {

node->left = insertNode(node->left, data);

} else if (data > node->data) {

node->right = insertNode(node->right, data);

}

return node;

}

void inOrderTraversal(struct binarytree \*root) {

if (root != NULL) {

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

}

void preOrderTraversal(struct binarytree \*root) {

if (root != NULL) {

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

preOrderTraversal(root->left);

preOrderTraversal(root->right);

}

}

void postOrderTraversal(struct binarytree \*root) {

if (root != NULL) {

postOrderTraversal(root->left);

postOrderTraversal(root->right);

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

}

}

int main() {

int data;

char choice;

printf("Binary search tree using linked list\n");

do {

printf("Enter data to be inserted in the binary tree: ");

scanf("%d", &data);

root = insertNode(root, data);

printf("Do you want to add more nodes? Y/N: ");

scanf(" %c", &choice);

} while (choice == 'Y' || choice == 'y');

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

inOrderTraversal(root);

printf("\n");

printf("Preorder traversal of the tree: ");

preOrderTraversal(root);

printf("\n");

printf("Postorder traversal of the tree: ");

postOrderTraversal(root);

printf("\n");

return 0;

}

2. Write C or C++ program to implement recursive tree traversal (inorder, preorder and post order)

#include <stdio.h>

#include <stdlib.h>

struct binarytree {

struct binarytree \*left;

int data;

struct binarytree \*right;

};

struct binarytree \*root = NULL;

struct binarytree \*createTree() {

struct binarytree \*newnode;

newnode = (struct binarytree \*)malloc(sizeof(struct binarytree));

int data;

char choice;

printf("Enter data to add in the binary tree: ");

scanf("%d", &data);

newnode->data = data;

// Clear the input buffer

while ((getchar()) != '\n');

printf("Do you want to insert left child to %d? Y/N: ", newnode->data);

scanf("%c", &choice);

if (choice == 'Y' || choice == 'y') {

newnode->left = createTree();

} else {

newnode->left = NULL;

}

// Clear the input buffer

while ((getchar()) != '\n');

printf("Do you want to insert right child to %d? Y/N: ", newnode->data);

scanf("%c", &choice);

if (choice == 'Y' || choice == 'y') {

newnode->right = createTree();

} else {

newnode->right = NULL;

}

// Clear the input buffer

while ((getchar()) != '\n');

return newnode;

}

void inOrderTraversal(struct binarytree \*root) {

if (root != NULL) {

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

}

void preOrderTraversal(struct binarytree \*root) {

if (root != NULL) {

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

preOrderTraversal(root->left);

preOrderTraversal(root->right);

}

}

void postOrderTraversal(struct binarytree \*root) {

if (root != NULL) {

postOrderTraversal(root->left);

postOrderTraversal(root->right);

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

}

}

int main() {

int choice;

while (1) {

printf("1. Create Binary Tree\n");

printf("2. Print Inorder Traversal of Binary Tree\n");

printf("3. Print Preorder Traversal of Binary Tree\n");

printf("4. Print Postorder Traversal of Binary Tree\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

root = createTree();

break;

case 2:

if (root == NULL) {

printf("First create a tree\n");

} else {

printf("The inorder Traversal of the tree is: ");

inOrderTraversal(root);

printf("\n");

}

break;

case 3:

if (root == NULL) {

printf("First create a tree\n");

} else {

printf("The Preorder traversal of the tree is: ");

preOrderTraversal(root);

printf("\n");

}

break;

case 4:

if (root == NULL) {

printf("First create tree\n");

} else {

printf("The Postorder traversal of the tree is: ");

postOrderTraversal(root);

printf("\n");

}

break;

case 5:

printf("Exiting the program...\n");

exit(0);

break;

default:

printf("Invalid option, enter a valid choice\n");

break;

}

}

}

3. Write C or C++ program to demonstrate various tree operations (count number of nodes, leaf nodes, printing leaf nodes, height of tree, mirror image of the tree)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int value) {

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

if (newNode == NULL) {

fprintf(stderr, "Memory allocation failed\n");

exit(1);

}

newNode->data = value;

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

return newNode;

}

struct Node\* insertNode(struct Node\* root, int value) {

if (root == NULL) {

return createNode(value);

}

if (value < root->data) {

root->left = insertNode(root->left, value);

} else if (value > root->data) {

root->right = insertNode(root->right, value);

}

return root;

}

int findHeight(struct Node\* root) {

if (root == NULL) {

return -1;

}

int leftHeight = findHeight(root->left);

int rightHeight = findHeight(root->right);

return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);

}

int countNodes(struct Node\* root) {

if (root == NULL) {

return 0;

}

return 1 + countNodes(root->left) + countNodes(root->right);

}

int countLeafNodes(struct Node\* root) {

if (root == NULL) {

return 0;

}

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

return 1;

}

return countLeafNodes(root->left) + countLeafNodes(root->right);

}

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

if (root == NULL) {

return NULL;

}

struct Node\* temp = root->left;

root->left = createMirror(root->right);

root->right = createMirror(temp);

return root;

}

void printLeafNodes(struct Node\* root) {

if (root == NULL) {

return;

}

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

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

}

printLeafNodes(root->left);

printLeafNodes(root->right);

}

void printMirror(struct Node\* root) {

if (root != NULL) {

printMirror(root->right);

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

printMirror(root->left);

}

}

int main() {

struct Node\* root = NULL;

int choice, value, n, i;

printf("Enter the number of elements in the tree: ");

scanf("%d", &n);

printf("Enter the elements: ");

for (i = 0; i < n; i++) {

scanf("%d", &value);

root = insertNode(root, value);

}

do {

printf("\n1. Find Height\n");

printf("2. Count Nodes\n");

printf("3. Count Leaf Nodes\n");

printf("4. Create Mirror Image and Print\n");

printf("5. Print Leaf Nodes\n");

printf("0. Exit\n");

printf("\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Height of the BST: %d\n", findHeight(root));

break;

case 2:

printf("Number of nodes in the BST: %d\n", countNodes(root));

break;

case 3:

printf("Number of leaf nodes in the BST: %d\n", countLeafNodes(root));

break;

case 4:

root = createMirror(root);

printf("Mirror Image created. Mirror In-order Traversal: ");

printMirror(root);

printf("\n");

break;

case 5:

printf("Leaf Nodes: ");

printLeafNodes(root);

printf("\n");

break;

case 0:

printf("Exiting the program.\n");

break;

default:

printf("Invalid choice. Please enter a valid option.\n");

}

} while (choice != 0);

return 0;

}

4. Write C or C++ program to create Binary search tree, insertion and searching and all deletion cases in BST

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*right\_child;

struct node \*left\_child;

};

struct node\* new\_node(int x) {

struct node \*temp;

temp = malloc(sizeof(struct node));

temp->data = x;

temp->left\_child = NULL;

temp->right\_child = NULL;

return temp;

}

struct node\* search(struct node \*root, int x) {

if (root == NULL || root->data == x)

return root;

else if (x > root->data)

return search(root->right\_child, x);

else

return search(root->left\_child, x);

}

struct node\* insert(struct node \*root, int x) {

if (root == NULL)

return new\_node(x);

else if (x > root->data)

root->right\_child = insert(root->right\_child, x);

else

root->left\_child = insert(root->left\_child, x);

return root;

}

struct node\* find\_minimum(struct node \*root) {

if (root == NULL)

return NULL;

else if (root->left\_child != NULL)

return find\_minimum(root->left\_child);

return root;

}

struct node\* delete(struct node \*root, int x) {

if (root == NULL)

return NULL;

if (x > root->data)

root->right\_child = delete(root->right\_child, x);

else if (x < root->data)

root->left\_child = delete(root->left\_child, x);

else {

if (root->left\_child == NULL && root->right\_child == NULL) {

free(root);

return NULL;

} else if (root->left\_child == NULL || root->right\_child == NULL) {

struct node \*temp;

if (root->left\_child == NULL)

temp = root->right\_child;

else

temp = root->left\_child;

free(root);

return temp;

} else {

struct node \*temp = find\_minimum(root->right\_child);

root->data = temp->data;

root->right\_child = delete(root->right\_child, temp->data);

}

}

return root;

}

void inorder(struct node \*root) {

if (root != NULL) {

inorder(root->left\_child);

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

inorder(root->right\_child);

}

}

int main() {

struct node \*root = NULL;

int choice, data;

while (1) {

printf("Binary Search Tree Operations: ");

printf("\n1. Insert\n2. Delete\n3. Inorder Traversal\n4. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the data to insert: ");

scanf("%d", &data);

root = insert(root, data);

break;

case 2:

printf("Enter the data to delete: ");

scanf("%d", &data);

root = delete(root, data);

break;

case 3:

printf("Inorder Traversal: ");

inorder(root);

printf("\n");

break;

case 4:

exit(0);

default:

printf("Invalid choice!\n");

}

}

return 0;

}

5. Write C or C++ program to implement **non-recursive tree traversal**(inorder, preorder and post order)

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node \*left, \*right;

} Node;

typedef struct Stack {

Node \*\*s;

int top;

int capacity;

} Stack;

void push(Stack \*s, Node \*t);

Node \*pop(Stack \*s);

int isEmpty(Stack \*s);

void inorder(Node \*root);

void preorder(Node \*root);

void postorder(Node \*root);

Node \*createNode(int data);

Stack \*createStack(int capacity);

void freeTree(Node \*root);

void insertNode(Node \*\*root, int data);

int main() {

int choice, data;

Node \*root = NULL;

do {

printf("\nBinary Tree Operations\n");

printf("1. Insert Node\n");

printf("2. Inorder Traversal\n");

printf("3. Preorder Traversal\n");

printf("4. Postorder Traversal\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data for the new node: ");

scanf("%d", &data);

insertNode(&root, data);

break;

case 2:

printf("Inorder Traversal: ");

inorder(root);

printf("\n");

break;

case 3:

printf("Preorder Traversal: ");

preorder(root);

printf("\n");

break;

case 4:

printf("Postorder Traversal: ");

postorder(root);

printf("\n");

break;

case 5:

printf("Exiting...\n");

freeTree(root);

break;

default:

printf("Invalid choice. Please enter a number between 1 and 5(both inclusive).\n");

}

} while (choice != 5);

return 0;

}

void push(Stack \*s, Node \*t) {

if (s->top == s->capacity - 1) {

printf("Stack Overflow\n");

return;

}

s->top++;

s->s[s->top] = t;

}

Node \*pop(Stack \*s) {

if (s->top == -1) {

printf("Stack Underflow\n");

return NULL;

}

Node \*t = s->s[s->top];

s->top--;

return t;

}

int isEmpty(Stack \*s) {

return s->top == -1;

}

void inorder(Node \*root) {

if (root == NULL) return;

Stack \*s = createStack(100);

Node \*T = root;

while (T != NULL || !isEmpty(s)) {

while (T != NULL) {

push(s, T);

T = T->left;

}

T = pop(s);

printf("%d ", T->data);

T = T->right;

}

}

void preorder(Node \*root) {

if (root == NULL) return;

Stack \*s = createStack(100);

Node \*T = root;

while (T != NULL || !isEmpty(s)) {

while (T != NULL) {

printf("%d ", T->data);

push(s, T);

T = T->left;

}

T = pop(s);

T = T->right;

}

}

void postorder(Node \*root) {

if (root == NULL) return;

Stack \*s1 = createStack(100);

Stack \*s2 = createStack(100);

push(s1, root);

while (!isEmpty(s1)) {

root = pop(s1);

push(s2, root);

if (root->left != NULL)

push(s1, root->left);

if (root->right != NULL)

push(s1, root->right);

}

while (!isEmpty(s2)) {

root = pop(s2);

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

}

}

Node \*createNode(int data) {

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

if (newNode == NULL) {

printf("Memory allocation failed\n");

exit(EXIT\_FAILURE);

}

newNode->data = data;

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

return newNode;

}

Stack \*createStack(int capacity) {

Stack \*stack = (Stack \*)malloc(sizeof(Stack));

if (stack == NULL) {

printf("Memory allocation failed\n");

exit(EXIT\_FAILURE);

}

stack->s = (Node \*\*)malloc(capacity \* sizeof(Node \*));

if (stack->s == NULL) {

printf("Memory allocation failed\n");

exit(EXIT\_FAILURE);

}

stack->top = -1;

stack->capacity = capacity;

return stack;

}

void freeTree(Node \*root) {

if (root != NULL) {

freeTree(root->left);

freeTree(root->right);

free(root);

}

}

void insertNode(Node \*\*root, int data) {

Node \*newNode = createNode(data);

if (\*root == NULL) {

\*root = newNode;

} else {

Node \*current = \*root;

Node \*parent = NULL;

while (1) {

parent = current;

if (data < current->data) {

current = current->left;

if (current == NULL) {

parent->left = newNode;

return;

}

} else {

current = current->right;

if (current == NULL) {

parent->right = newNode;

return;

}

}

}

}

}

6. Write C/C++ program to check whether the tree is balanced or not and tree is AVL or not. Also, your code should tell which rotation case is required if tree is imbalanced. Show all rotations cases. Input should be user choice**.**

#include <stdio.h>

#include <stdlib.h>

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

int height;

};

struct TreeNode\* createNode(int data) {

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

if (!newNode) {

printf("Memory allocation failed!\n");

exit(1);

}

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

int getHeight(struct TreeNode\* node) {

if (node == NULL)

return 0;

return node->height;

}

struct TreeNode\* rightRotate(struct TreeNode\* y) {

struct TreeNode\* x = y->left;

struct TreeNode\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = 1 + (getHeight(y->left) > getHeight(y->right) ? getHeight(y->left) : getHeight(y->right));

x->height = 1 + (getHeight(x->left) > getHeight(x->right) ? getHeight(x->left) : getHeight(x->right));

return x;

}

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

struct TreeNode\* y = x->right;

struct TreeNode\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = 1 + (getHeight(x->left) > getHeight(x->right) ? getHeight(x->left) : getHeight(x->right));

y->height = 1 + (getHeight(y->left) > getHeight(y->right) ? getHeight(y->left) : getHeight(y->right));

return y;

}

int getBalanceFactor(struct TreeNode\* node) {

if (node == NULL)

return 0;

return getHeight(node->left) - getHeight(node->right);

}

struct TreeNode\* insertNode(struct TreeNode\* root, int data) {

if (root == NULL)

return createNode(data);

if (data < root->data)

root->left = insertNode(root->left, data);

else if (data > root->data)

root->right = insertNode(root->right, data);

else

return root;

root->height = 1 + (getHeight(root->left) > getHeight(root->right) ? getHeight(root->left) : getHeight(root->right));

int balance = getBalanceFactor(root);

if (balance > 1 && data < root->left->data)

return rightRotate(root);

if (balance < -1 && data > root->right->data)

return leftRotate(root);

if (balance > 1 && data > root->left->data) {

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

return rightRotate(root);

}

if (balance < -1 && data < root->right->data) {

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

return leftRotate(root);

}

return root;

}

int isBalanced(struct TreeNode\* root) {

if (root == NULL)

return 1;

int leftHeight = getHeight(root->left);

int rightHeight = getHeight(root->right);

if (abs(leftHeight - rightHeight) <= 1 && isBalanced(root->left) && isBalanced(root->right))

return 1;

return 0;

}

int isAVL(struct TreeNode\* root) {

if (root == NULL)

return 1;

int balance = getBalanceFactor(root);

if (abs(balance) <= 1 && isAVL(root->left) && isAVL(root->right))

return 1;

return 0;

}

void printRotationCases(struct TreeNode\* root, int data) {

struct TreeNode\* newRoot = insertNode(root, data);

if (isBalanced(newRoot))

printf("The tree is balanced.\n");

else {

printf("The tree is imbalanced.\n");

int balance = getBalanceFactor(newRoot);

printf("Balance factor: %d\n", balance);

if (balance > 1 && data < newRoot->left->data)

printf("Left-Left rotation needed.\n");

else if (balance < -1 && data > newRoot->right->data)

printf("Right-Right rotation needed.\n");

else if (balance > 1 && data > newRoot->left->data) {

printf("Left-Right rotation needed.\n");

printf("Performing Left-Right rotation...\n");

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

newRoot = rightRotate(newRoot);

printf("Rotation performed.\n");

} else if (balance < -1 && data < newRoot->right->data) {

printf("Right-Left rotation needed.\n");

printf("Performing Right-Left rotation...\n");

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

newRoot = leftRotate(newRoot);

printf("Rotation performed.\n");

}

}

}

void freeTree(struct TreeNode\* root) {

if (root == NULL)

return;

freeTree(root->left);

freeTree(root->right);

free(root);

}

int main() {

struct TreeNode\* root = NULL;

int choice, data;

do {

printf("\n----- AVL Tree Operations -----\n");

printf("1. Insert Node in AVL tree\n");

printf("2. Check if AVL tree is Balanced\n");

printf("3. Check if it is AVL tree\n");

printf("4. Check Rotation Cases\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

root = insertNode(root, data);

break;

case 2:

if (isBalanced(root))

printf("The tree is balanced.\n");

else

printf("The tree is not balanced.\n");

break;

case 3:

if (isAVL(root))

printf("The tree is an AVL tree.\n");

else

printf("The tree is not an AVL tree.\n");

break;

case 4:

printf("Enter data to check rotation cases: ");

scanf("%d", &data);

printRotationCases(root, data);

break;

case 5:

printf("Exiting...\n");

break;

default:

printf("Invalid choice! Please enter a valid option.\n");

}

} while (choice != 5);

freeTree(root);

return 0;

}

7. Write C/C++ program for representation of graphs using adjacency matrix and adjacency lists

#include <stdio.h>

#include <stdlib.h>

struct Node {

int dest;

struct Node\* next;

};

struct Graph {

int V;

struct Node\*\* array;

};

struct Node\* createNode(int dest) {

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

newNode->dest = dest;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int V) {

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->V = V;

graph->array = (struct Node\*\*)malloc((V + 1) \* sizeof(struct Node\*));

for (int i = 1; i <= V; ++i)

graph->array[i] = NULL;

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) {

struct Node\* newNode = createNode(dest);

newNode->next = graph->array[src];

graph->array[src] = newNode;

newNode = createNode(src);

newNode->next = graph->array[dest];

graph->array[dest] = newNode;

}

void printGraph(struct Graph\* graph) {

for (int v = 1; v <= graph->V; ++v) {

struct Node\* pCrawl = graph->array[v];

printf("Adjacency list of vertex %d\nhead ", v);

while (pCrawl) {

printf("-> %d", pCrawl->dest);

pCrawl = pCrawl->next;

}

printf("\n\n");

}

}

void adjacencyMatrix() {

int N, M;

printf("Enter the number of vertices: ");

scanf("%d", &N);

printf("Enter the number of edges: ");

scanf("%d", &M);

int Adj[N + 1][N + 1];

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

for (int j = 0; j <= N; j++) {

Adj[i][j] = 0;

}

}

printf("Enter the edges (source destination):\n");

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

int x, y;

scanf("%d %d", &x, &y);

Adj[x][y] = 1;

Adj[y][x] = 1;

}

printf("\nGraph representation using adjacency matrix\n");

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

for (int j = 1; j <= N; j++) {

printf("%d ", Adj[i][j]);

}

printf("\n");

}

}

void adjacencyList() {

int V, E;

printf("Enter the number of vertices: ");

scanf("%d", &V);

printf("Enter the number of edges: ");

scanf("%d", &E);

struct Graph\* graph = createGraph(V);

printf("Enter the edges (source destination):\n");

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

int src, dest;

scanf("%d %d", &src, &dest);

addEdge(graph, src, dest);

}

printf("\nGraph representation using adjacency list\n");

printGraph(graph);

}

int main() {

int choice;

while (1) {

printf("1. Graph Representation using Adjacency matrix\n");

printf("2. Graph Representation using Adjacency list\n");

printf("3. Exit\n");

printf("Enter choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

adjacencyMatrix();

break;

case 2:

adjacencyList();

break;

case 3:

exit(0);

default:

printf("Invalid choice\n");

break;

}

}

return 0;

}

8. Write C/C++ program for graph traversals using BFS and DFS

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node \*next;

};

struct Queue {

struct Node \*front, \*rear;

};

struct Node \*createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

struct Queue \*createQueue() {

struct Queue \*queue = (struct Queue \*)malloc(sizeof(struct Queue));

queue->front = queue->rear = NULL;

return queue;

}

void enqueue(struct Queue \*queue, int data) {

struct Node \*newNode = createNode(data);

if (queue->rear == NULL) {

queue->front = queue->rear = newNode;

return; }

queue->rear->next = newNode;

queue->rear = newNode;

}

int dequeue(struct Queue \*queue) {

if (queue->front == NULL)

return -1;

struct Node \*temp = queue->front;

int data = temp->data;

queue->front = queue->front->next;

if (queue->front == NULL)

queue->rear = NULL;

free(temp);

return data;

}

int isEmpty(struct Queue \*queue) {

return queue->front == NULL;

}

void BFS(int N, int Adj[][N + 1], int start) {

int \*visited = (int \*)malloc((N + 1) \* sizeof(int));

for (int i = 1; i <= N; ++i)

visited[i] = 0;

struct Queue \*queue = createQueue();

visited[start] = 1;

printf("BFS traversal starting from vertex %d: ", start);

enqueue(queue, start);

while (!isEmpty(queue)) {

int vertex = dequeue(queue);

printf("%d ", vertex);

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

if (Adj[vertex][i] && !visited[i]) {

visited[i] = 1;

enqueue(queue, i);

}

} }

free(visited);

}

void DFSUtil(int N, int Adj[][N + 1], int vertex, int \*visited) {

visited[vertex] = 1;

printf("%d ", vertex);

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

if (Adj[vertex][i] && !visited[i]) {

DFSUtil(N, Adj, i, visited);

}

} }

void DFS(int N, int Adj[][N + 1], int start) {

int \*visited = (int \*)malloc((N + 1) \* sizeof(int));

for (int i = 1; i <= N; ++i)

visited[i] = 0;

printf("DFS traversal starting from vertex %d: ", start);

DFSUtil(N, Adj, start, visited);

free(visited);

}

int main() {

int N, M;

printf("Enter the number of vertices: ");

scanf("%d", &N);

printf("Enter the number of edges: ");

scanf("%d", &M);

int Adj[N + 1][N + 1];

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

for (int j = 0; j <= N; j++) {

Adj[i][j] = 0;

}

}

printf("Enter the edges (source destination):\n");

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

int x, y;

scanf("%d %d", &x, &y);

Adj[x][y] = 1;

Adj[y][x] = 1;

}

printf("\nGraph representation using adjacency matrix\n");

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

for (int j = 1; j <= N; j++) {

printf("%d ", Adj[i][j]);

}

printf("\n");

}

int choice;

int start;

while (1) {

printf("\nChoose traversal type:\n");

printf("1. BFS traversal\n");

printf("2. DFS traversal\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

if (choice == 3)

break;

printf("Enter the starting vertex: ");

scanf("%d", &start);

switch (choice) {

case 1:

BFS(N, Adj, start);

break;

case 2:

DFS(N, Adj, start);

break;

default:

printf("Invalid choice\n");

break; }

}

return 0; }

9. Write C/C++ program to demonstrate Hashing using linear or quadratic probing

#include <stdio.h>

#include <stdlib.h>

// Hash function

int hash(int key, int size) {

return key % size;

}

// Linear probing

int linearProbing(int key, int size, int hashTable[]) {

int index = hash(key, size);

int i = 0;

while (hashTable[(index + i) % size] != -1) {

i++;

if (i == size) {

return -1; // Hash table is full

}

}

return (index + i) % size;

}

// Quadratic probing

int quadraticProbing(int key, int size, int hashTable[]) {

int index = hash(key, size);

int i = 0;

while (hashTable[(index + i \* i) % size] != -1) {

i++;

if (i == size) {

return -1; // Hash table is full

}

}

return (index + i \* i) % size;

}

// Insert

void insert(int key, int size, int hashTable[], int technique) {

int index;

switch (technique) {

case 1:

index = linearProbing(key, size, hashTable);

break;

case 2:

index = quadraticProbing(key, size, hashTable);

break;

default:

printf("Invalid technique\n");

return;

}

if (index == -1) {

printf("Hash table is full, cannot insert key %d\n", key);

} else {

hashTable[index] = key;

}

}

// Search

int search(int key, int size, int hashTable[], int technique) {

int index = hash(key, size);

int i = 0;

switch (technique) {

case 1:

// Linear probing

while (hashTable[(index + i) % size] != key) {

if (hashTable[(index + i) % size] == -1 || i == size) {

return -1; // Key not found

}

i++;

}

return (index + i) % size;

case 2:

// Quadratic probing

while (hashTable[(index + i \* i) % size] != key) {

if (hashTable[(index + i \* i) % size] == -1 || i == size) {

return -1; // Key not found

}

i++;

}

return (index + i \* i) % size;

default:

printf("Invalid technique\n");

return -1;

}

}

void display(int size, int hashTable[]) {

printf("\nHash Table:\n");

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

if (hashTable[i] == -1) {

printf("-\t");

} else {

printf("%d\t", hashTable[i]);

}

}

printf("\n");

}

int main() {

int size;

printf("Enter the size of the hash table: ");

scanf("%d", &size);

int hashTable[size];

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

hashTable[i] = -1;

}

int technique;

printf("Enter the hashing technique to use:\n1. Linear Probing\n2. Quadratic Probing\nEnter your choice: ");

scanf("%d", &technique);

int choice;

do {

printf("\n1. Insert\n2. Search\n3. Display\n4. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: {

int key;

int n;

printf("Enter the number of keys to insert: ");

scanf("%d", &n);

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

printf("Enter the key to be inserted: ");

scanf("%d", &key);

insert(key, size, hashTable, technique);

}

break;

}

case 2: {

int key;

printf("Enter the key to be searched: ");

scanf("%d", &key);

int index = search(key, size, hashTable, technique);

if (index == -1) {

printf("Key not found\n");

} else {

printf("Key found at index %d\n", index);

}

break;

}

case 3: {

display(size, hashTable);

break;

}

case 4: {

printf("Exiting...\n");

break;

}

default: {

printf("Invalid choice\n");

break;

}

}

} while (choice != 4);

return 0;

}

10. Write C/C++ program to implement Heap data structure. Create Max-heap and insert elements into it.

#include <stdio.h>

#define MAX\_SIZE 100

int size = 0;

void swap(int \*a, int \*b) {

int temp = \*b;

\*b = \*a;

\*a = temp;

}

void heapify(int array[], int size, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// Compare with left child

if (left < size && array[left] > array[largest])

largest = left;

// Compare with right child

if (right < size && array[right] > array[largest])

largest = right;

// If largest is not the root, swap and heapify the affected subtree

if (largest != i) {

swap(&array[i], &array[largest]);

heapify(array, size, largest);

}

}

void insert(int array[], int newNum) {

if (size >= MAX\_SIZE) {

printf("Heap is full, cannot insert %d\n", newNum);

return;

}

if (size == 0) {

array[0] = newNum;

size += 1;

} else {

array[size] = newNum;

size += 1;

// Heapify from the last inserted element up to the root

for (int i = size / 2 - 1; i >= 0; i--) {

heapify(array, size, i);

}

}

}

int deleteElement(int array[], int element) {

int i;

for (i = 0; i < size; i++) {

if (element == array[i])

break;

}

if (i == size) {

printf("Element %d not found in the heap\n", element);

return -1;

}

int deletedValue = array[i];

array[i] = array[size - 1];

size -= 1;

heapify(array, size, i);

return deletedValue;

}

void printArray(int array[], int size) {

for (int i = 0; i < size; ++i)

printf("%d ", array[i]);

printf("\n");

}

int main() {

int array[MAX\_SIZE];

printf("Enter elements to insert into max-heap (-1 to stop):\n");

int num;

do {

printf("Enter element (or -1 to stop): ");

scanf("%d", &num);

if (num != -1) {

insert(array, num);

printf("Max-Heap array: ");

printArray(array, size);

}

} while (num != -1);

int choice;

do {

printf("\n1. Delete root element\n");

printf("2. Delete specific element\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: {

if (size > 0) {

int deletedValue = array[0];

deleteElement(array, array[0]);

printf("Deleted root element: %d\n", deletedValue);

printf("Max-Heap array after deletion: ");

printArray(array, size);

} else {

printf("Heap is empty, cannot delete\n");

}

break;

}

case 2: {

int element;

printf("Enter element to delete: ");

scanf("%d", &element);

int deletedValue = deleteElement(array, element);

if (deletedValue != -1) {

printf("Deleted element: %d\n", deletedValue);

printf("Max-Heap array after deletion: ");

printArray(array, size);

}

break;

}

case 3:

printf("Exiting...\n");

break;

default:

printf("Invalid choice\n");

break;

}

} while (choice != 3);

return 0;

}