**PRACTICAL 1**

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

**C**

#include <stdio.h>

#define MAX\_NODES 10

char tree[MAX\_NODES];

int root(char key) {

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

printf("Tree already had root");

else

tree[0] = key;

return 0;

}

int left\_set(char key, int parent) {

if (tree[parent] == '\0' || (parent \* 2) + 1 >= MAX\_NODES)

printf("\nCan't set left child, invalid parent index or array bounds exceeded");

else

tree[(parent \* 2) + 1] = key;

return 0;

}

int right\_set(char key, int parent) {

if (tree[parent] == '\0' || (parent \* 2) + 2 >= MAX\_NODES)

printf("\nCan't set right child, invalid parent index or array bounds exceeded");

else

tree[(parent \* 2) + 2] = key;

return 0;

}

int print\_tree() {

printf("\n");

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

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

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

else

printf("-");

}

return 0;

}

int main() {

int numNodes;

char key;

int parent;

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

scanf("%d", &numNodes);

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

printf("Enter key for node %d: ", i);

scanf(" %c", &key);

if (i == 0) {

root(key);

} else {

printf("Enter parent index for node %c: ", key);

scanf("%d", &parent);

if (parent < 0 || parent >= MAX\_NODES) {

printf("Invalid parent index. Please enter a valid index.\n");

i--;

} else {

if (i % 2 == 1) {

left\_set(key, parent);

} else {

right\_set(key, parent);

}

}

}

}

print\_tree();

return 0;

}

**C++**

#include <iostream>

#include <vector>

#define MAX\_NODES 10

using namespace std;

vector<char> tree(MAX\_NODES, '\0');

void root(char key) {

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

cout << "Tree already has root\n";

else

tree[0] = key;

}

void left\_set(char key, int parent) {

if (tree[parent] == '\0' || (parent \* 2) + 1 >= MAX\_NODES)

cout << "\nCan't set left child, invalid parent index or array bounds exceeded\n";

else

tree[(parent \* 2) + 1] = key;

}

void right\_set(char key, int parent) {

if (tree[parent] == '\0' || (parent \* 2) + 2 >= MAX\_NODES)

cout << "\nCan't set right child, invalid parent index or array bounds exceeded\n";

else

tree[(parent \* 2) + 2] = key;

}

void print\_tree() {

cout << "Obtained Tree is:\n";

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

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

cout << tree[i];

else

cout << "-";

}

cout << " \n";

}

int main() {

int numNodes;

char key;

int parent;

cout << "Enter the number of nodes in the tree: ";

cin >> numNodes;

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

cout << "Enter key for node " << i << ": ";

cin >> key;

if (i == 0) {

root(key);

} else {

cout << "Enter parent index for node " << key << ": ";

cin >> parent;

if (parent < 0 || parent >= MAX\_NODES) {

cout << "Invalid parent index. Please enter a valid index.\n";

--i;

} else {

if (i % 2 == 1) {

left\_set(key, parent);

} else {

right\_set(key, parent);

}

}

}

}

print\_tree();

return 0;

}

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

**C**

#include<stdio.h>

#include<stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

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

if (node != NULL) {

node->data = data;

node->left = NULL;

node->right = NULL;

printf("New node created with data: %d\n", data);

} else {

printf("Memory allocation failed for the new node\n");

}

return node;

}

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

if (root == NULL) return;

if (data < root->data) {

if (root->left == NULL) {

root->left = newNode(data);

} else {

insertNode(root->left, data);

}

} else {

if (root->right == NULL) {

root->right = newNode(data);

} else {

insertNode(root->right, data);

}

}

}

struct Node\* buildBinaryTree() {

struct Node\* root = NULL;

int numNodes, data;

printf("Enter the number of nodes you want in the binary tree: ");

scanf("%d", &numNodes);

if (numNodes <= 0) {

printf("Invalid number of nodes. Exiting.\n");

exit(1);

}

printf("Enter the value for the root: ");

scanf("%d", &data);

root = newNode(data);

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

printf("Enter the value for node %d: ", i + 1);

scanf("%d", &data);

insertNode(root, data);

}

return root;

}

void Level\_Order(struct Node\* node) {

if (node == NULL)

return;

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

Level\_Order(node->left);

Level\_Order(node->right);

}

int main() {

struct Node\* root = buildBinaryTree();

printf("Traversal of the binary tree: \n");

Level\_Order(root);

return 0;

}

**C++**

#include <iostream>

#include <memory>

#include <queue>

using namespace std;

struct Node {

int data;

shared\_ptr<Node> left;

shared\_ptr<Node> right;

Node(int data) : data(data), left(nullptr), right(nullptr) {

cout << "New node created with data: " << data << endl;

}

};

shared\_ptr<Node> newNode(int data) {

return make\_shared<Node>(data);

}

void insertNode(shared\_ptr<Node>& root, int data) {

if (!root) return;

if (data < root->data) {

if (!root->left) {

root->left = newNode(data);

} else {

insertNode(root->left, data);

}

} else {

if (!root->right) {

root->right = newNode(data);

} else {

insertNode(root->right, data);

}

}

}

shared\_ptr<Node> buildBinaryTree() {

shared\_ptr<Node> root = nullptr;

int numNodes, data;

cout << "Enter the number of nodes you want in the binary tree: ";

cin >> numNodes;

if (numNodes <= 0) {

cout << "Invalid number of nodes. Exiting." << endl;

exit(1);

}

cout << "Enter the value for the root: ";

cin >> data;

root = newNode(data);

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

cout << "Enter the value for node " << i + 1 << ": ";

cin >> data;

insertNode(root, data);

}

return root;

}

void levelOrder(const shared\_ptr<Node>& root) {

if (!root) return;

queue<shared\_ptr<Node>> q;

q.push(root);

while (!q.empty()) {

shared\_ptr<Node> current = q.front();

q.pop();

cout << current->data << " ";

if (current->left) {

q.push(current->left);

}

if (current->right) {

q.push(current->right);

}

}

}

int main() {

shared\_ptr<Node> root = buildBinaryTree();

cout << "Level-order traversal of the binary tree: " << endl;

levelOrder(root);

return 0;

}

**PRACTICAL 2**

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

**C**

#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) {

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

return NULL;

}

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;

}

void inorderTraversal(struct Node\* root) {

if (root == NULL) {

return;

}

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

void preorderTraversal(struct Node\* root) {

if (root == NULL) {

return;

}

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

preorderTraversal(root->left);

preorderTraversal(root->right);

}

void postorderTraversal(struct Node\* root) {

if (root == NULL) {

return;

}

postorderTraversal(root->left);

postorderTraversal(root->right);

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

}

int main() {

struct Node\* root = NULL;

int numNodes, value;

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

scanf("%d", &numNodes);

printf("Enter the values of the nodes:\n");

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

scanf("%d", &value);

root = insertNode(root, value);

}

printf("Inorder traversal: ");

inorderTraversal(root);

printf("\n");

printf("Preorder traversal: ");

preorderTraversal(root);

printf("\n");

printf("Postorder traversal: ");

postorderTraversal(root);

printf("\n");

return 0;

}

**C++**

#include <iostream>

#include <memory>

using namespace std;

struct Node {

int data;

shared\_ptr<Node> left;

shared\_ptr<Node> right;

Node(int value) : data(value), left(nullptr), right(nullptr) {}

};

shared\_ptr<Node> createNode(int value) {

return make\_shared<Node>(value);

}

shared\_ptr<Node> insertNode(shared\_ptr<Node> root, int value) {

if (!root) {

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;

}

void inorderTraversal(const shared\_ptr<Node>& root) {

if (!root) {

return;

}

inorderTraversal(root->left);

cout << root->data << " ";

inorderTraversal(root->right);

}

void preorderTraversal(const shared\_ptr<Node>& root) {

if (!root) {

return;

}

cout << root->data << " ";

preorderTraversal(root->left);

preorderTraversal(root->right);

}

void postorderTraversal(const shared\_ptr<Node>& root) {

if (!root) {

return;

}

postorderTraversal(root->left);

postorderTraversal(root->right);

cout << root->data << " ";

}

int main() {

shared\_ptr<Node> root = nullptr;

int numNodes, value;

cout << "Enter the number of nodes to insert: ";

cin >> numNodes;

cout << "Enter the values of the nodes:" << endl;

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

cin >> value;

root = insertNode(root, value);

}

cout << "Inorder traversal: ";

inorderTraversal(root);

cout << endl;

cout << "Preorder traversal: ";

preorderTraversal(root);

cout << endl;

cout << "Postorder traversal: ";

postorderTraversal(root);

cout << endl;

return 0;

}

**PRACTICAL 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)**

**C**

#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));

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 0;

}

int leftHeight = findHeight(root->left);

int rightHeight = findHeight(root->right);

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

}

void level\_Order(struct Node\* root) {

if (root == NULL)

return;

struct Node\* queue[100];

int front = -1, rear = -1;

queue[++rear] = root;

while (front < rear) {

struct Node\* temp = queue[++front];

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

if (temp->left)

queue[++rear] = temp->left;

if (temp->right)

queue[++rear] = temp->right;

}

}

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);

}

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);

}

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 printMirror(struct Node\* root) {

if (root != NULL) {

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

printMirror(root->left);

printMirror(root->right);

}

}

int main() {

struct Node\* root = NULL;

int choice, value;

do {

printf("\n1. Insert Element\n");

printf("2. Find Height\n");

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

printf("4. Count Nodes\n");

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

printf("6. Create Mirror Image\n");

printf("7. Print Leaf Nodes\n");

printf("0. Exit\n");

printf("\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

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

scanf("%d", &value);

root = insertNode(root, value);

break;

case 2:

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

break;

case 3:

printf("Level order Traversal: ");

level\_Order(root);

printf("\n");

break;

case 4:

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

break;

case 5:

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

break;

case 6:

root = createMirror(root);

printf("Mirror Image created.\n");

printMirror(root);

break;

case 7:

printf("Leaf Nodes: ");

printLeafNodes(root);

printf("\n");

break;

case 8:

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

break;

default:

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

}

} while (choice != 0);

return 0;

}

**C++**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

};

Node\* createNode(int value) {

Node\* newNode = new Node();

newNode->data = value;

newNode->left = newNode->right = nullptr;

return newNode;

}

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

if (root == nullptr) {

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(Node\* root) {

if (root == nullptr) {

return 0;

}

int leftHeight = findHeight(root->left);

int rightHeight = findHeight(root->right);

return 1 + max(leftHeight, rightHeight);

}

int countNodes(Node\* root) {

if (root == nullptr) {

return 0;

}

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

}

int countLeafNodes(Node\* root) {

if (root == nullptr) {

return 0;

}

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

return 1;

}

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

}

Node\* createMirror(Node\* root) {

if (root == nullptr) {

return nullptr;

}

Node\* temp = root->left;

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

root->right = createMirror(temp);

return root;

}

void printLeafNodes(Node\* root) {

if (root == nullptr) {

return;

}

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

cout << root->data << " ";

}

printLeafNodes(root->left);

printLeafNodes(root->right);

}

void printMirror(Node\* root) {

if (root != nullptr) {

printMirror(root->left);

cout << root->data << " ";

printMirror(root->right);

}

}

int main() {

Node\* root = nullptr;

int choice, value;

do {

cout << "\n1. Insert Element\n";

cout << "2. Find Height\n";

cout << "3. Count Nodes\n";

cout << "4. Count Leaf Nodes\n";

cout << "5. Create Mirror Image\n";

cout << "6. Print Leaf Nodes\n";

cout << "0. Exit\n";

cout << "\nEnter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the value to insert: ";

cin >> value;

root = insertNode(root, value);

break;

case 2:

cout << "Height of the BST: " << findHeight(root) << endl;

break;

case 3:

cout << "Number of nodes in the BST: " << countNodes(root) << endl;

break;

case 4:

cout << "Number of leaf nodes in the BST: " << countLeafNodes(root) << endl;

break;

case 5:

root = createMirror(root);

cout << "Mirror Image created.\n";

printMirror(root);

cout << endl;

break;

case 6:

cout << "Leaf Nodes: ";

printLeafNodes(root);

cout << endl;

break;

case 0:

cout << "Exiting the program.\n";

break;

default:

cout << "Invalid choice. Please enter a valid option.\n";

}

} while (choice != 0);

return 0;

}

**PRACTICAL 4**

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

**C**

#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("\n1. Insert\n2. Delete\n3. Search\n4. Inorder Traversal\n5. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

root = insert(root, data);

break;

case 2:

printf("Enter data to delete: ");

scanf("%d", &data);

root = delete(root, data);

break;

case 3:

printf("Enter data to search: ");

scanf("%d", &data);

if (search(root, data) != NULL)

printf("Element %d found in the tree.\n", data);

else

printf("Element %d not found in the tree.\n", data);

break;

case 4:

printf("Inorder Traversal: ");

inorder(root);

printf("\n");

break;

case 5:

exit(0);

default:

printf("Invalid choice!\n");

}

}

return 0;

}

**C++**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* right\_child;

Node\* left\_child;

};

Node\* new\_node(int x) {

Node\* temp = new Node();

temp->data = x;

temp->left\_child = nullptr;

temp->right\_child = nullptr;

return temp;

}

Node\* search(Node\* root, int x) {

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

return root;

else if (x > root->data)

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

else

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

}

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

if (root == nullptr)

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;

}

Node\* find\_minimum(Node\* root) {

if (root == nullptr)

return nullptr;

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

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

return root;

}

Node\* deleteNode(Node\* root, int x) {

if (root == nullptr)

return nullptr;

if (x > root->data)

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

else if (x < root->data)

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

else {

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

delete root;

return nullptr;

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

Node\* temp = (root->left\_child == nullptr) ? root->right\_child : root->left\_child;

delete root;

return temp;

} else {

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

root->data = temp->data;

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

}

}

return root;

}

void inorder(Node\* root) {

if (root != nullptr) {

inorder(root->left\_child);

cout << " " << root->data << " ";

inorder(root->right\_child);

}

}

int main() {

Node\* root = nullptr;

int choice, data;

while (true) {

cout << "\n1. Insert\n2. Delete\n3. Search\n4. Inorder Traversal\n5. Exit\nEnter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter data to insert: ";

cin >> data;

root = insert(root, data);

break;

case 2:

cout << "Enter data to delete: ";

cin >> data;

root = deleteNode(root, data);

break;

case 3:

cout << "Enter data to search: ";

cin >> data;

if (search(root, data) != nullptr)

cout << "Element " << data << " found in the tree.\n";

else

cout << "Element " << data << " not found in the tree.\n";

break;

case 4:

cout << "Inorder Traversal: ";

inorder(root);

cout << "\n";

break;

case 5:

exit(0);

default:

cout << "Invalid choice!\n";

}

}

return 0;

}

**PRACTICAL 5**

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

**C**#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("\nBST 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.\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;

}

}

}

}

}

**C++**

**PRACTICAL 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.**

**C**

#include <stdio.h>

#include <stdlib.h>

// An AVL tree node

struct Node {

int key;

struct Node \*left;

struct Node \*right;

int height;

};

// A utility function to get the height of the tree

int height(struct Node \*N) {

if (N == NULL)

return 0;

return N->height;

}

// A utility function to get maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Helper function that allocates a new node with the given key and NULL left and right pointers

struct Node\* newNode(int key) {

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

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1; // new node is initially added at leaf

return (node);

}

// A utility function to right rotate subtree rooted with y

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

struct Node \*x = y->left;

struct Node \*T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

// Return new root

return x;

}

// A utility function to left rotate subtree rooted with x

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

struct Node \*y = x->right;

struct Node \*T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

// Return new root

return y;

}

// Get Balance factor of node N

int getBalance(struct Node \*N) {

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

// Function to insert a key in the subtree rooted with node and returns the new root of the subtree

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

/\* 1. Perform the normal BST insertion \*/

if (node == NULL)

return (newNode(key));

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

else // Equal keys are not allowed in BST

return node;

/\* 2. Update height of this ancestor node \*/

node->height = 1 + max(height(node->left), height(node->right));

/\* 3. Get the balance factor of this ancestor node to check whether this node became unbalanced \*/

int balance = getBalance(node);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case

if (balance > 1 && key < node->left->key) {

printf("Left Left Rotation is required");

return rightRotate(node);

}

// Right Right Case

if (balance < -1 && key > node->right->key) {

printf("Right Right Rotation is required\n");

return leftRotate(node);

}

// Left Right Case

if (balance > 1 && key > node->left->key) {

printf("Left Right Rotation is required\n");

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

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && key < node->right->key) {

printf("Right-Left rotation is required\n");

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

return leftRotate(node);

}

/\* return the (unchanged) node pointer \*/

return node;

}

// A utility function to print preorder traversal of the tree

void preOrder(struct Node \*root) {

if (root != NULL) {

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

preOrder(root->left);

preOrder(root->right);

}

}

int main() {

struct Node \*root = NULL;

int n, key;

printf("Enter the number of nodes to insert in the AVL tree: ");

scanf("%d", &n);

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

printf("Enter node %d: ", i + 1);

scanf("%d", &key);

root = insert(root, key);

}

printf("Preorder traversal of the constructed AVL tree is \n");

preOrder(root);

printf("\n");

return 0;

}

**C++**

#include <iostream>

#include <cstdlib>

using namespace std;

// An AVL tree node

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

// A utility function to get the height of the tree

int height(Node\* N) {

if (N == nullptr)

return 0;

return N->height;

}

// A utility function to get maximum of two integers

int maxi(int a, int b) {

return (a > b) ? a : b;

}

// Helper function that allocates a new node with the given key and NULL left and right pointers

Node\* newNode(int key) {

Node\* node = new Node();

node->key = key;

node->left = nullptr;

node->right = nullptr;

node->height = 1; // new node is initially added at leaf

return node;

}

// A utility function to right rotate subtree rooted with y

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = maxi(height(y->left), height(y->right)) + 1;

x->height = maxi(height(x->left), height(x->right)) + 1;

// Return new root

return x;

}

// A utility function to left rotate subtree rooted with x

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = maxi(height(x->left), height(x->right)) + 1;

y->height = maxi(height(y->left), height(y->right)) + 1;

// Return new root

return y;

}

// Get Balance factor of node N

int getBalance(Node\* N) {

if (N == nullptr)

return 0;

return height(N->left) - height(N->right);

}

// Function to insert a key in the subtree rooted with node and returns the new root of the subtree

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

/\* 1. Perform the normal BST insertion \*/

if (node == nullptr)

return (newNode(key));

if (key < node->key)

node->left = insert(node->left, key);

else if (key > node->key)

node->right = insert(node->right, key);

else // Equal keys are not allowed in BST

return node;

/\* 2. Update height of this ancestor node \*/

node->height = 1 + maxi(height(node->left), height(node->right));

/\* 3. Get the balance factor of this ancestor node to check whether this node became unbalanced \*/

int balance = getBalance(node);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case

if (balance > 1 && key < node->left->key) {

cout << "Left Left Rotation is required" << endl;

return rightRotate(node);

}

// Right Right Case

if (balance < -1 && key > node->right->key) {

cout << "Right Right Rotation is required" << endl;

return leftRotate(node);

}

// Left Right Case

if (balance > 1 && key > node->left->key) {

cout << "Left Right Rotation is required" << endl;

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

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && key < node->right->key) {

cout << "Right-Left rotation is required" << endl;

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

return leftRotate(node);

}

/\* return the (unchanged) node pointer \*/

return node;

}

// A utility function to print preorder traversal of the tree

void preOrder(Node\* root) {

if (root != nullptr) {

cout << root->key << " ";

preOrder(root->left);

preOrder(root->right);

}

}

int main() {

Node\* root = nullptr;

int n, key;

cout << "Enter the number of nodes to insert in the AVL tree: ";

cin >> n;

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

cout << "Enter node " << i + 1 << ": ";

cin >> key;

root = insert(root, key);

}

cout << "Preorder traversal of the constructed AVL tree is" << endl;

preOrder(root);

cout << endl;

return 0;

}

**PRACTICAL 7**

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

**C (Adjacency List)**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int vertex;

struct Node\* next;

} Node;

Node\* createNode(int v) {

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

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Number of vertices in the graph

int numVertices;

Node\*\* adjacencyList;

void initializeGraph(int vertices) {

numVertices = vertices;

adjacencyList = malloc(vertices \* sizeof(Node\*));

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

adjacencyList[i] = NULL;

}

}

void addEdge(int src, int dest) {

Node\* newNode = createNode(dest);

newNode->next = adjacencyList[src];

adjacencyList[src] = newNode;

// As it's an undirected graph, add edge from dest to src

newNode = createNode(src);

newNode->next = adjacencyList[dest];

adjacencyList[dest] = newNode;

}

void printGraph() {

for (int v = 0; v < numVertices; v++) {

Node\* temp = adjacencyList[v];

printf("Vertex %d:", v);

while (temp) {

printf(" -> %d", temp->vertex);

temp = temp->next;

}

printf("\n");

}

}

int main() {

int numEdges, src, dest;

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

scanf("%d", &numVertices);

initializeGraph(numVertices);

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

scanf("%d", &numEdges);

printf("Enter source and destination vertices for each edge:\n");

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

printf("Edge %d: ", i + 1);

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

addEdge(src, dest);

}

printf("\nAdjacency List Representation:\n");

printGraph();

return 0;

}

**C++ (Adjacency List)**

#include <iostream>

#include <vector>

using namespace std;

struct Node {

int vertex;

Node\* next;

};

Node\* createNode(int v) {

Node\* newNode = new Node();

newNode->vertex = v;

newNode->next = nullptr;

return newNode;

}

// Number of vertices in the graph

int numVertices;

vector<Node\*> adjacencyList;

void initializeGraph(int vertices) {

numVertices = vertices;

adjacencyList.resize(vertices, nullptr);

}

void addEdge(int src, int dest) {

Node\* newNode = createNode(dest);

newNode->next = adjacencyList[src];

adjacencyList[src] = newNode;

// As it's an undirected graph, add edge from dest to src

newNode = createNode(src);

newNode->next = adjacencyList[dest];

adjacencyList[dest] = newNode;

}

void printGraph() {

for (int v = 0; v < numVertices; v++) {

Node\* temp = adjacencyList[v];

cout << "Vertex " << v << ":";

while (temp) {

cout << " -> " << temp->vertex;

temp = temp->next;

}

cout << endl;

}

}

int main() {

int numEdges, src, dest;

cout << "Enter the number of vertices: ";

cin >> numVertices;

initializeGraph(numVertices);

cout << "Enter the number of edges: ";

cin >> numEdges;

cout << "Enter source and destination vertices for each edge:" << endl;

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

cout << "Edge " << i + 1 << ": ";

cin >> src >> dest;

addEdge(src, dest);

}

cout << "\nAdjacency List Representation:" << endl;

printGraph();

return 0;

}

**C (Adjacency matrix)**

#include<stdio.h>

#define max 20

int adj[max][max];

int n;

void create\_graph() {

int i, max\_edges, origin, destin;

printf("Enter number of nodes : ");

scanf("%d", &n);

max\_edges = n \* (n - 1);

for (i = 1; i <= max\_edges; i++) {

printf("Enter edge %d (0 0) to quit : ", i);

scanf("%d %d", &origin, &destin);

if ((origin == 0) && (destin == 0))

break;

if (origin > n || destin > n || origin <= 0 || destin <= 0) {

printf("Invalid edge!\n");

i--;

} else

adj[origin][destin] = 1;

}

// Print the adjacency matrix

printf("Adjacency Matrix:\n");

for (i = 1; i <= n; i++) {

for (int j = 1; j <= n; j++)

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

printf("\n");

}

}

int main() {

create\_graph();

return 0;

}

**C++ (Adjacency matrix)**

#include<iostream>

using namespace std;

#define max 20

int adj[max][max];

int n;

void create\_graph() {

int i, max\_edges, origin, destin;

cout << "Enter number of nodes : ";

cin >> n;

max\_edges = n \* (n - 1);

for (i = 1; i <= max\_edges; i++) {

cout << "Enter edge " << i << " (0 0) to quit : ";

cin >> origin >> destin;

if ((origin == 0) && (destin == 0))

break;

if (origin > n || destin > n || origin <= 0 || destin <= 0) {

cout << "Invalid edge!" << endl;

i--;

} else

adj[origin][destin] = 1;

}

// Print the adjacency matrix

cout << "Adjacency Matrix:" << endl;

for (i = 1; i <= n; i++) {

for (int j = 1; j <= n; j++)

cout << adj[i][j] << " ";

cout << endl;

}

}

int main() {

create\_graph();

return 0;

}

**PRACTICAL 8**

**8.1 Write C/C++ program for graph traversals using BFS**

**C**

#include<stdio.h>

#include<stdbool.h>

#define MAX\_VERTICES 100

int adj[MAX\_VERTICES][MAX\_VERTICES];

void addEdge(int x, int y) {

adj[x][y] = 1;

adj[y][x] = 1;

}

void bfs(int start, int num\_vertices) {

bool visited[MAX\_VERTICES] = {false};

int queue[MAX\_VERTICES];

int front = 0, rear = -1;

queue[++rear] = start;

visited[start] = true;

while (front <= rear) {

int vis = queue[front++];

printf("%d ", vis);

for (int i = 0; i < num\_vertices; i++) {

if (adj[vis][i] == 1 && !visited[i]) {

queue[++rear] = i;

visited[i] = true;

}

}

}

}

int main() {

int num\_vertices, num\_edges;

int start\_vertex;

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

scanf("%d", &num\_vertices);

for (int i = 0; i < num\_vertices; i++) {

for (int j = 0; j < num\_vertices; j++) {

adj[i][j] = 0;

}

}

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

scanf("%d", &num\_edges);

printf("Enter the edges (u v) one by one:\n");

for (int i = 0; i < num\_edges; i++) {

int u, v;

scanf("%d %d", &u, &v);

addEdge(u, v);

}

printf("Enter the starting vertex for BFS: ");

scanf("%d", &start\_vertex);

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

bfs(start\_vertex, num\_vertices);

printf("\n");

return 0;

}

**C++**

#include <iostream>

#include <queue>

using namespace std;

#define MAX\_VERTICES 100

int adj[MAX\_VERTICES][MAX\_VERTICES];

void addEdge(int x, int y) {

adj[x][y] = 1;

adj[y][x] = 1;

}

void bfs(int start, int num\_vertices) {

bool visited[MAX\_VERTICES] = {false};

queue<int> q;

q.push(start);

visited[start] = true;

while (!q.empty()) {

int vis = q.front();

q.pop();

cout << vis << " ";

for (int i = 0; i < num\_vertices; i++) {

if (adj[vis][i] == 1 && !visited[i]) {

q.push(i);

visited[i] = true;

}

}

}

}

int main() {

int num\_vertices, num\_edges;

int start\_vertex;

cout << "Enter the number of vertices: ";

cin >> num\_vertices;

for (int i = 0; i < num\_vertices; i++) {

for (int j = 0; j < num\_vertices; j++) {

adj[i][j] = 0;

}

}

cout << "Enter the number of edges: ";

cin >> num\_edges;

cout << "Enter the edges (u v) one by one:" << endl;

for (int i = 0; i < num\_edges; i++) {

int u, v;

cin >> u >> v;

addEdge(u, v);

}

cout << "Enter the starting vertex for BFS: ";

cin >> start\_vertex;

cout << "BFS starting from vertex " << start\_vertex << ": ";

bfs(start\_vertex, num\_vertices);

cout << endl;

return 0;

}

**8.2 Write C/C++ program for graph traversals using DFS  
C**

#include <stdio.h>

#include <stdlib.h>

int\*\* adj;

void addEdge(int x, int y) {

adj[x][y] = 1;

adj[y][x] = 1;

}

void dfs(int start, int\* visited, int v) {

printf("%d ", start);

visited[start] = 1;

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

if (adj[start][i] == 1 && !visited[i]) {

dfs(i, visited, v);

}

}

}

int main() {

int v, e;

printf("Enter number of vertices: ");

scanf("%d", &v);

printf("Enter number of edges: ");

scanf("%d", &e);

adj = (int\*\*)malloc(v \* sizeof(int\*));

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

adj[i] = (int\*)malloc(v \* sizeof(int));

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

adj[i][j] = 0;

}

}

printf("Enter the edges (format: x y):\n");

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

int x, y;

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

addEdge(x, y);

}

int\* visited = (int\*)malloc(v \* sizeof(int));

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

visited[i] = 0;

}

printf("DFS starting from node 0:\n");

dfs(0, visited, v);

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

free(adj[i]);

}

free(adj);

free(visited);

return 0;

}

**C++**

#include <iostream>

#include <cstdlib>

using namespace std;

int\*\* adj;

void addEdge(int x, int y) {

adj[x][y] = 1;

adj[y][x] = 1;

}

void dfs(int start, int\* visited, int v) {

cout << start << " ";

visited[start] = 1;

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

if (adj[start][i] == 1 && !visited[i]) {

dfs(i, visited, v);

}

}

}

int main() {

int v, e;

cout << "Enter number of vertices: ";

cin >> v;

cout << "Enter number of edges: ";

cin >> e;

adj = new int\*[v];

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

adj[i] = new int[v];

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

adj[i][j] = 0;

}

}

cout << "Enter the edges (format: x y):\n";

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

int x, y;

cin >> x >> y;

addEdge(x, y);

}

int\* visited = new int[v];

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

visited[i] = 0;

}

cout << "DFS starting from node 0:\n";

dfs(0, visited, v);

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

delete[] adj[i];

}

delete[] adj;

delete[] visited;

return 0;

}

**PRACTICAL 9**

**9.1 Write C/C++ program to demonstrate Hashing using linear probing**

**C**

#include <stdio.h>

#include <stdlib.h>

#define EMPTY -1

struct HashTable {

int \*table;

int size;

};

int hashFunction(int key, int size) {

return key % size;

}

void initHashTable(struct HashTable \*ht, int size) {

ht->size = size;

ht->table = (int \*)malloc(size \* sizeof(int));

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

ht->table[i] = EMPTY;

}

}

void insert(struct HashTable \*ht, int key) {

int index = hashFunction(key, ht->size);

int i = 0;

while (ht->table[(index + i) % ht->size] != EMPTY) {

i++;

if (i == ht->size) {

printf("Hash table is full!\n");

return;

}

}

ht->table[(index + i) % ht->size] = key;

}

int search(const struct HashTable \*ht, int key) {

int index = hashFunction(key, ht->size);

int i = 0;

while (ht->table[(index + i) % ht->size] != EMPTY) {

if (ht->table[(index + i) % ht->size] == key) {

return 1;

}

i++;

if (i == ht->size) {

break;

}

}

return 0;

}

void display(const struct HashTable \*ht) {

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

if (ht->table[i] == EMPTY) {

printf("Empty\n");

} else {

printf("%d\n", ht->table[i]);

}

}

}

int main() {

int size;

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

scanf("%d", &size);

struct HashTable ht;

initHashTable(&ht, size);

int choice, key;

while (1) {

printf("1. Insert\n2. Search\n3. Display\n4. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter key to insert: ");

scanf("%d", &key);

insert(&ht, key);

break;

case 2:

printf("Enter key to search: ");

scanf("%d", &key);

if (search(&ht, key)) {

printf("%d found in hash table.\n", key);

} else {

printf("%d not found in hash table.\n", key);

}

break;

case 3:

display(&ht);

break;

case 4:

free(ht.table);

return 0;

default:

printf("Invalid choice!\n");

}

}

}

**C++**

#include <iostream>

#include <cstdlib>

using namespace std;

#define EMPTY -1

struct HashTable {

int \*table;

int size;

};

int hashFunction(int key, int size) {

return key % size;

}

void initHashTable(HashTable &ht, int size) {

ht.size = size;

ht.table = new int[size];

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

ht.table[i] = EMPTY;

}

}

void insert(HashTable &ht, int key) {

int index = hashFunction(key, ht.size);

int i = 0;

while (ht.table[(index + i) % ht.size] != EMPTY) {

i++;

if (i == ht.size) {

cout << "Hash table is full!" << endl;

return;

}

}

ht.table[(index + i) % ht.size] = key;

}

int search(const HashTable &ht, int key) {

int index = hashFunction(key, ht.size);

int i = 0;

while (ht.table[(index + i) % ht.size] != EMPTY) {

if (ht.table[(index + i) % ht.size] == key) {

return 1;

}

i++;

if (i == ht.size) {

break;

}

}

return 0;

}

void display(const HashTable &ht) {

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

if (ht.table[i] == EMPTY) {

cout << "Empty" << endl;

} else {

cout << ht.table[i] << endl;

}

}

}

int main() {

int size;

cout << "Enter size of hash table: ";

cin >> size;

HashTable ht;

initHashTable(ht, size);

int choice, key;

while (true) {

cout << "1. Insert\n2. Search\n3. Display\n4. Exit\nEnter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter key to insert: ";

cin >> key;

insert(ht, key);

break;

case 2:

cout << "Enter key to search: ";

cin >> key;

if (search(ht, key)) {

cout << key << " found in hash table." << endl;

} else {

cout << key << " not found in hash table." << endl;

}

break;

case 3:

display(ht);

break;

case 4:

delete[] ht.table;

return 0;

default:

cout << "Invalid choice!" << endl;

}

}

}

**9.1 Write C/C++ program to demonstrate Hashing using quadratic probing**

**C**

#include <stdio.h>

#include <stdlib.h>

#define EMPTY -1

typedef struct HashTable {

int \*table;

int size;

} HashTable;

int hashFunction(int key, int size) {

return key % size;

}

void initHashTable(HashTable \*ht, int size) {

ht->size = size;

ht->table = (int \*)malloc(size \* sizeof(int));

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

ht->table[i] = EMPTY;

}

}

void insert(HashTable \*ht, int key) {

int index = hashFunction(key, ht->size);

int i = 0;

while (ht->table[(index + i \* i) % ht->size] != EMPTY) {

i++;

if (i == ht->size) {

printf("Hash table is full!\n");

return;

}

}

ht->table[(index + i \* i) % ht->size] = key;

}

int search(HashTable \*ht, int key) {

int index = hashFunction(key, ht->size);

int i = 0;

while (ht->table[(index + i \* i) % ht->size] != EMPTY) {

if (ht->table[(index + i \* i) % ht->size] == key) {

return 1;

}

i++;

if (i == ht->size) {

break;

}

}

return 0;

}

void display(HashTable \*ht) {

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

if (ht->table[i] == EMPTY) {

printf("Empty\n");

} else {

printf("%d\n", ht->table[i]);

}

}

}

int main() {

int size;

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

scanf("%d", &size);

HashTable ht;

initHashTable(&ht, size);

int choice, key;

while (1) {

printf("1. Insert\n2. Search\n3. Display\n4. Exit\nEnter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter key to insert: ");

scanf("%d", &key);

insert(&ht, key);

break;

case 2:

printf("Enter key to search: ");

scanf("%d", &key);

if (search(&ht, key)) {

printf("%d found in hash table.\n", key);

} else {

printf("%d not found in hash table.\n", key);

}

break;

case 3:

display(&ht);

break;

case 4:

free(ht.table);

return 0;

default:

printf("Invalid choice!\n");

}

}

}

**C++**

#include <iostream>

#include <cstdlib>

using namespace std;

#define EMPTY -1

struct HashTable {

int \*table;

int size;

};

int hashFunction(int key, int size) {

return key % size;

}

void initHashTable(HashTable &ht, int size) {

ht.size = size;

ht.table = new int[size];

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

ht.table[i] = EMPTY;

}

}

void insert(HashTable &ht, int key) {

int index = hashFunction(key, ht.size);

int i = 0;

while (ht.table[(index + i \* i) % ht.size] != EMPTY) {

i++;

if (i == ht.size) {

cout << "Hash table is full!" << endl;

return;

}

}

ht.table[(index + i \* i) % ht.size] = key;

}

int search(const HashTable &ht, int key) {

int index = hashFunction(key, ht.size);

int i = 0;

while (ht.table[(index + i \* i) % ht.size] != EMPTY) {

if (ht.table[(index + i \* i) % ht.size] == key) {

return 1;

}

i++;

if (i == ht.size) {

break;

}

}

return 0;

}

void display(const HashTable &ht) {

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

if (ht.table[i] == EMPTY) {

cout << "Empty" << endl;

} else {

cout << ht.table[i] << endl;

}

}

}

int main() {

int size;

cout << "Enter size of hash table: ";

cin >> size;

HashTable ht;

initHashTable(ht, size);

int choice, key;

while (true) {

cout << "1. Insert\n2. Search\n3. Display\n4. Exit\nEnter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter key to insert: ";

cin >> key;

insert(ht, key);

break;

case 2:

cout << "Enter key to search: ";

cin >> key;

if (search(ht, key)) {

cout << key << " found in hash table." << endl;

} else {

cout << key << " not found in hash table." << endl;

}

break;

case 3:

display(ht);

break;

case 4:

delete[] ht.table;

return 0;

default:

cout << "Invalid choice!" << endl;

}

}

}

**PRACTICAL 10**

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

**C**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int \*heap;

int size;

int capacity;

} MaxHeap;

void initMaxHeap(MaxHeap \*h, int capacity) {

h->heap = (int \*)malloc(capacity \* sizeof(int));

h->size = 0;

h->capacity = capacity;

}

void heapifyUp(MaxHeap \*h, int index) {

int parentIndex = (index - 1) / 2;

if (index > 0 && h->heap[parentIndex] < h->heap[index]) {

int temp = h->heap[parentIndex];

h->heap[parentIndex] = h->heap[index];

h->heap[index] = temp;

heapifyUp(h, parentIndex);

}

}

void insert(MaxHeap \*h, int element) {

if (h->size == h->capacity) {

printf("Max-Heap is full\n");

return;

}

h->heap[h->size] = element;

heapifyUp(h, h->size);

h->size++;

}

void display(MaxHeap \*h) {

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

printf("%d ", h->heap[i]);

}

printf("\n");

}

void freeMaxHeap(MaxHeap \*h) {

free(h->heap);

}

int main() {

MaxHeap maxHeap;

int capacity = 100; // Arbitrary capacity for the heap

initMaxHeap(&maxHeap, capacity);

int choice, element;

while (1) {

printf("1. Insert element into Max-Heap\n");

printf("2. Display Max-Heap\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter element to insert: ");

scanf("%d", &element);

insert(&maxHeap, element);

break;

case 2:

printf("Max-Heap elements: ");

display(&maxHeap);

break;

case 3:

freeMaxHeap(&maxHeap);

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

**C++**

**10.2 10.1 Write C/C++ program to implement Heap data structure. Create Min-heap and insert elements into it.**

**C**

#include<stdio.h>

#include<limits.h>

int heap[1000000], heapSize;

void Init() {

heapSize = 0;

heap[0] = -INT\_MAX;

}

void Insert(int element) {

heapSize++;

heap[heapSize] = element;

int now = heapSize;

while (heap[now / 2] > element) {

heap[now] = heap[now / 2];

now /= 2;

}

heap[now] = element;

}

int DeleteMin() {

int minElement, lastElement, child, now;

minElement = heap[1];

lastElement = heap[heapSize--];

for (now = 1; now \* 2 <= heapSize; now = child) {

child = now \* 2;

if (child != heapSize && heap[child + 1] < heap[child]) {

child++;

}

if (lastElement > heap[child]) {

heap[now] = heap[child];

} else

{

break;

}

}

heap[now] = lastElement;

return minElement;

}

void Display() {

int i;

for (i = 1; i <= heapSize; i++) {

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

}

printf("\n");

}

int main() {

int choice, element;

Init();

while (1) {

printf("Program to demonstrate Heap:\n");

printf("1. Insert\n");

printf("2. Display\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

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

scanf("%d", &element);

Insert(element);

break;

case 2:

printf("Heap elements: ");

Display();

break;

case 3:

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

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

}

**C++**

#include <iostream>

#define HEAP\_SIZE 1000000

using namespace std;

int heap[HEAP\_SIZE + 1];

int heapSize;

void Init()

{

heapSize = 0;

heap[0] = -2147483647; // -INT\_MAX

}

void Insert(int element)

{

heapSize++;

heap[heapSize] = element;

int now = heapSize;

while (heap[now / 2] > element)

{

heap[now] = heap[now / 2];

now /= 2;

}

heap[now] = element;

}

int DeleteMin()

{

int minElement, lastElement, child, now;

minElement = heap[1];

lastElement = heap[heapSize--];

for (now = 1; now \* 2 <= heapSize; now = child)

{

child = now \* 2;

if (child != heapSize && heap[child + 1] < heap[child])

{

child++;

}

if (lastElement > heap[child])

{

heap[now] = heap[child];

}

else

{

break;

}

}

heap[now] = lastElement;

return minElement;

}

void Display()

{

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

{

cout << heap[i] << " ";

}

cout << endl;

}

int main()

{

int choice, element;

Init();

while (true)

{

cout << "Program to demonstrate Heap:\n";

cout << "1. Insert\n";

cout << "2. Display\n";

cout << "3. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter the element to insert: ";

cin >> element;

Insert(element);

break;

case 2:

cout << "Heap elements: ";

Display();

continue; // This will skip the rest of the loop body and restart the loop

case 3:

cout << "Exiting...\n";

return 0;

default:

cout << "Invalid choice. Please try again.\n";

}

}

}

**Extra Bonus  
  
1) Dijkstra’s Algorithm**

**C**

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

#include <stdlib.h>

int minDistance(int dist[], bool sptSet[], int V) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (sptSet[v] == false && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

void printSolution(int dist[], int V) {

printf("Vertex \t\t Distance from Source\n");

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

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

}

void dijkstra(int \*\*graph, int src, int V) {

int dist[V];

bool sptSet[V];

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

dist[i] = INT\_MAX, sptSet[i] = false;

dist[src] = 0;

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet, V);

sptSet[u] = true;

for (int v = 0; v < V; v++)

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

printSolution(dist, V);

}

int main() {

int V;

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

scanf("%d", &V);

int \*\*graph = (int \*\*)malloc(V \* sizeof(int \*));

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

graph[i] = (int \*)malloc(V \* sizeof(int));

printf("Enter the adjacency matrix for the graph:\n");

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

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

printf("Enter weight of edge from vertex %d to vertex %d (or 0 if no edge): ", i, j);

scanf("%d", &graph[i][j]);

}

}

int source;

printf("Enter the source vertex: ");

scanf("%d", &source);

if (source < 0 || source >= V) {

printf("Invalid source vertex.\n");

return 1;

}

dijkstra(graph, source, V);

// Free allocated memory

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

free(graph[i]);

free(graph);

return 0;

}

**C++**

#include <iostream>

int minDistance(int \*dist, bool \*sptSet, int V) {

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (sptSet[v] == false && dist[v] <= min)

min = dist[v], min\_index = v;

return min\_index;

}

void printSolution(int \*dist, int V) {

std::cout << "Vertex \t\t Distance from Source\n";

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

std::cout << i << " \t\t\t\t " << dist[i] << "\n";

}

void dijkstra(int \*\*graph, int src, int V) {

int \*dist = new int[V];

bool \*sptSet = new bool[V];

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

dist[i] = INT\_MAX, sptSet[i] = false;

dist[src] = 0;

for (int count = 0; count < V - 1; count++) {

int u = minDistance(dist, sptSet, V);

sptSet[u] = true;

for (int v = 0; v < V; v++)

if (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX && dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

printSolution(dist, V);

delete[] dist;

delete[] sptSet;

}

int main() {

int V;

std::cout << "Enter the number of vertices in the graph: ";

std::cin >> V;

int \*\*graph = new int\*[V];

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

graph[i] = new int[V];

std::cout << "Enter the adjacency matrix for the graph:\n";

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

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

std::cout << "Enter weight of edge from vertex " << i << " to vertex " << j << " (or 0 if no edge): ";

std::cin >> graph[i][j];

}

}

int source;

std::cout << "Enter the source vertex: ";

std::cin >> source;

if (source < 0 || source >= V) {

std::cout << "Invalid source vertex.\n";

return 1;

}

dijkstra(graph, source, V);

// Free allocated memory

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

delete[] graph[i];

delete[] graph;

return 0;

}

**2) Prim’s Algorithm**

**C**

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

// A utility function to find the vertex with

// minimum key value, from the set of vertices

// not yet included in MST

int minKey(int key[], bool mstSet[], int V)

{

// Initialize min value

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

// A utility function to print the

// constructed MST stored in parent[]

void printMST(int parent[], int \*\*graph, int V)

{

printf("Edge \tWeight\n");

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

printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

}

// Function to construct and print MST for

// a graph represented using adjacency

// matrix representation

void primMST(int \*\*graph, int V)

{

// Array to store constructed MST

int parent[V];

// Key values used to pick minimum weight edge in cut

int key[V];

// To represent set of vertices included in MST

bool mstSet[V];

// Initialize all keys as INFINITE

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

key[i] = INT\_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.

// Make key 0 so that this vertex is picked as first

// vertex.

key[0] = 0;

// First node is always root of MST

parent[0] = -1;

// The MST will have V vertices

for (int count = 0; count < V - 1; count++)

{

// Pick the minimum key vertex from the

// set of vertices not yet included in MST

int u = minKey(key, mstSet, V);

// Add the picked vertex to the MST Set

mstSet[u] = true;

// Update key value and parent index of

// the adjacent vertices of the picked vertex.

// Consider only those vertices which are not

// yet included in MST

for (int v = 0; v < V; v++)

// graph[u][v] is non zero only for adjacent

// vertices of m mstSet[v] is false for vertices

// not yet included in MST Update the key only

// if graph[u][v] is smaller than key[v]

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

// print the constructed MST

printMST(parent, graph, V);

}

// Driver's code

int main()

{

int V;

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

scanf("%d", &V);

int \*\*graph = (int \*\*)malloc(V \* sizeof(int \*));

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

graph[i] = (int \*)malloc(V \* sizeof(int));

printf("Enter the adjacency matrix:\n");

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

for (int j = 0; j < V; j++)

scanf("%d", &graph[i][j]);

// Print the solution

primMST(graph, V);

// Free allocated memory

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

free(graph[i]);

free(graph);

return 0;

}

**C++**