**Slip no 14**

1. **1. Implement Sequential and Binary Search**

#include <iostream>

using namespace std;

int sequentialSearch(int array[], int size, int key) {

for (int i = 0; i < size; i++) { if (array[i] == key) {

return i; // return the index of the key if found }}

return -1; // return -1 if key is not found

} int main() {

int array[] = {1, 2, 3, 4, 5};

int size = sizeof(array) / sizeof(array[0]);

int key = 3;

int index = sequentialSearch(array, size, key);

if (index != -1) {

cout << "Key found at index " << index << endl;

} else {

cout << "Key not found" << endl;

} return 0;}

**Q2. Create binary tree. Find height of the tree and print leaf nodes. Find mirror image, print original and mirror image using level-wise printing**

#include <iostream>using namespace std;

struct Node { int data;

Node \*left, \*right; };

Node\* newNode(int data)

{ Node\* node = new Node; node->data = data;

node->left = node->right = NULL;

return node; }

int height(Node\* root)

{ if (root == NULL) return 0;

return 1 + max(height(root->left), height(root->right)); ] }

void printLeafNodes(Node\* root)

{ if (root == NULL) return;

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

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

printLeafNodes(root->left);

printLeafNodes(root->right); }

Node\* mirror(Node\* root)

{ if (root == NULL) return NULL;

Node\* left = mirror(root->left);

Node\* right = mirror(root->right);

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

return root; }

void printLevelWise(Node\* root)

{ if (root == NULL) return;

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

printLevelWise(root->left);

printLevelWise(root->right);

} int main()

{ Node\* root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

cout << "Height of tree: " << height(root) << endl;

cout << "Leaf nodes: ";

printLeafNodes(root);

cout << endl;

Node\* mirrorRoot = mirror(root);

cout << "Original tree: ";

printLevelWise(root);

cout << endl;

cout << "Mirror image: ";

printLevelWise(mirrorRoot);

return 0;

}

**Slip16**

**Q. 1. Implement circular queue using arrays.**

#include <iostream>using namespace std;

const int MAX\_SIZE = 5; class CircularQueue {

private: int front, rear; int arr[MAX\_SIZE];

public: CircularQueue() {

front = -1; rear = -1; } bool is\_empty() {

return front == -1 && rear == -1; }bool is\_full() {

return (rear + 1) % MAX\_SIZE == front;

}void enqueue(int value) {

if (is\_full()) {

cout << "Error: Queue is full." << endl;

} else if (is\_empty()) { front = 0;

rear = 0; arr[rear] = value;

} else {

rear = (rear + 1) % MAX\_SIZE;

arr[rear] = value; }

} int dequeue() { if (is\_empty()) {

cout << "Error: Queue is empty." << endl;

return -1; } else if (front == rear) {

int value = arr[front];

front = -1; rear = -1;

return value; } else {

int value = arr[front];

front = (front + 1) % MAX\_SIZE;

return value; } }

void display() { if (is\_empty()) {

cout << "Queue is empty." << endl;

} else { cout << "Queue elements are: ";

int i = front; while (i != rear) {

cout << arr[i] << " ";

i = (i + 1) % MAX\_SIZE; }

cout << arr[rear] << endl; }}};

int main() { CircularQueue q;

q.enqueue(1); q.enqueue(2);

q.enqueue(3);q.enqueue(4);

q.enqueue(5); q.display(); // Output// 1 2 3 4 5

q.enqueue(6); // Output: Error: Queue is full.

q.dequeue(); q.dequeue();

q.display(); // Output: Queue elements are: 3 4 5

q.enqueue(6);

q.display(); // Output:: 3 4 5 6

return 0;}

**Q. 2. Create binary tree and perform recursive traversals.**

#include <iostream>

using namespace std;

struct Node {int data;

Node\* left; Node\* right; };

Node\* getNewNode(int data) {

Node\* newNode = new Node();

newNode->data = data;

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

return newNode;

}void preOrder(Node\* root) {

if (root == NULL) return;

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

preOrder(root->left); preOrder(root->right);

}void inOrder(Node\* root) {

if (root == NULL) return;

inOrder(root->left); cout << root->data << " ";

inOrder(root->right);

}void postOrder(Node\* root) {

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

postOrder(root->right);

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

}int main() {

Node\* root = getNewNode(1);

root->left = getNewNode(2);

root->right = getNewNode(3);

root->left->left = getNewNode(4);

root->left->right = getNewNode(5);

cout << "Pre-order traversal: ";

preOrder(root); cout << endl;

cout << "In-order traversal: ";

inOrder(root); cout << endl;

cout << "Post-order traversal: ";

postOrder(root); cout << endl;

return 0;}

**Slip 17**

**1. Implement minimum cost spanning tree algorithm.**

#include <iostream> #include <climits>

#include <vector> using namespace std;

const int MAX\_SIZE = 100; int prim\_algorithm(int graph[MAX\_SIZE][MAX\_SIZE], int num\_vertices) {

vector<bool> visited(num\_vertices, false);

vector<int> distances(num\_vertices, INT\_MAX);

distances[0] = 0;int min\_cost = 0;

for (int i = 0; i < num\_vertices - 1; i++) {

int min\_distance = INT\_MAX;int current\_vertex = -1; for (int j = 0; j < num\_vertices; j++) { if (!visited[j] && distances[j] < min\_distance) {

min\_distance = distances[j]; current\_vertex = j;}}

visited[current\_vertex] = true;

min\_cost += min\_distance;

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

if (graph[current\_vertex][k] != 0 && !visited[k] && graph[current\_vertex][k] < distances[k]) {

distances[k] = graph[current\_vertex][k]; }}}

return min\_cost;}

int main() {

int graph[MAX\_SIZE][MAX\_SIZE] = {

{0, 2, 0, 6, 0}; {2, 0, 3, 8, 5},

{0, 3, 0, 0, 7}, {6, 8, 0, 0, 9},

{0, 5, 7, 9, 0} };

int num\_vertices = 5;

int min\_cost = prim\_algorithm(graph, num\_vertices);

cout << "Minimum cost spanning tree: " << min\_cost << endl; // Output: 16

return 0;}

**Slip no 18**

**Q. 1. Implement shortest path algorithm.**

#include <iostream> #include <climits>

#include <vector> using namespace std;

const int MAX\_SIZE = 100; int

dijkstra\_algorithm(int graph[MAX\_SIZE][MAX\_SIZE], int num\_vertices, int start\_vertex, int end\_vertex)

{ vector<bool> visited(num\_vertices, false);

vector<int> distances(num\_vertices, INT\_MAX);

distances[start\_vertex] = 0;

for (int i = 0; i < num\_vertices - 1; i++)

{

int min\_distance = INT\_MAX;

int current\_vertex = -1;

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

if (!visited[j] && distances[j] < min\_distance) {

min\_distance = distances[j]; current\_vertex = j;}

}

visited[current\_vertex] = true;

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

if (graph[current\_vertex][k] != 0 && !visited[k] && distances[current\_vertex] + graph[current\_vertex][k] < distances[k]) { distances[k] = distances[current\_vertex] + graph[current\_vertex][k]; }}

}return distances[end\_vertex];

}int main() {

int graph[MAX\_SIZE][MAX\_SIZE] = {

{0, 2, 0, 6, 0},{2, 0, 3, 8, 5},

{0, 3, 0, 0, 7},{6, 8, 0, 0, 9},

{0, 5, 7, 9, 0}};

int num\_vertices = 5;

int start\_vertex = 0;

int end\_vertex = 4;

int shortest\_distance = dijkstra\_algorithm(graph, num\_vertices, start\_vertex, end\_vertex);

cout << "Shortest distance between vertex " << start\_vertex << " and vertex " << end\_vertex << ": " << shortest\_distance << endl; // Output: 14

return 0;

}

**Slip 19**

1. **1. Write a menu driven program to perform following operations on singly linked list: Create, Insert, Delete, and Display**

#include <iostream>using namespace std;

struct Node { int data; Node\* next; };

class LinkedList {

private:Node\* head;

int count;

public: LinkedList()

{head = NULL;

count = 0;} void create() {int data;

cout << "Enter the data for the node: ";

cin >> data;Node\* newNode = new Node();

newNode->data = data; newNode->next = head;

head = newNode; count++ ;}

void reverse() {

Node\* prev = NULL; Node\* current = head;

Node\* next = NULL; while (current != NULL) {

next = current->next; current->next = prev;

prev = current; current = next;}head = prev;}

int search(int key) {

Node\* current = head;

int index = 0; while (current != NULL) {

if (current->data == key) { return index;}

current = current->next; index++;}

return -1;} int countNodes() { return count;}

void display() {

Node\* current = head;

while (current != NULL) {cout << current->data << " "; current = current->next; }cout << endl;}};

int main() {

int choice; LinkedList list;

while (true) { cout << "1. Create Node" << endl;

cout << "2. Reverse List" << endl;

cout << "3. Search Element" << endl;

cout << "4. Count Nodes" << endl;

cout << "5. Display List" << endl;

cout << "6. Exit" << endl;

cout << "Enter your choice: "; cin >> choice;

switch (choice) {case 1: list.create(); break;

case 2: list.reverse();break; case 3: {int key;

cout << "Enter the element to be searched: ";

cin >> key; int index = list.search(key);

if (index == -1) {

cout << "Element not found." << endl; } else {

cout << "Element found at index: " << index << endl;

} break; }case 4:cout << "Number of nodes: " << list.countNodes() << endl;

break; case 5: list.display();

break;case 6: return 0; default:

cout << "Invalid choice. Please enter a valid choice." << endl; }} return 0;

}

Slip 20

**1. Implement Linked queue**

#include <bits/stdc++.h>

using namespace std;

struct QNode {int data;

QNode\* next; QNode(int d)

{ data = d; next = NULL; }};

struct Queue { QNode \*front, \*rear;

Queue() { front = rear = NULL; }

void enQueue(int x)

{ QNode\* temp = new QNode(x);

if (rear == NULL) {

front = rear = temp;

return; } rear->next = temp;

rear = temp; } void deQueue()

{ if (front == NULL)

return; QNode\* temp = front;

front = front->next; if (front == NULL)

rear = NULL; delete (temp);}};

int main() {

Queue q; q.enQueue(10); q.enQueue(20);

q.deQueue(); q.deQueue(); q.enQueue(30);

q.enQueue(40); q.enQueue(50); q.deQueue();

cout << "Queue Front : " << (q.front)->data << endl; cout << "Queue Rear : " << (q.rear)->data;

}