Course title- Data structures and Algorithms Lab
Course Code- PMCA501P

CYCLESHEET 3

Q1.Program to implement breadth first traversal.

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 100

struct Node {

   int vertex;

   struct Node* next;

};

struct Graph {

   int numVertices;

   struct Node** adjLists;

   int* visited;

};

struct Queue {

   int items[MAX];

   int front;

   int rear;

};


struct Node* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

   return newNode;

}

struct Graph* createGraph(int numVertices) {

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

   graph->numVertices = numVertices;

   graph->adjLists = malloc(numVertices * sizeof(struct Node*));
```

```
    graph->visited = malloc(numVertices * sizeof(int));


    for (int i = 0; i<numVertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;  // Mark all vertices as unvisited initially
    }
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
newNode = createNode(src);
newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
struct Queue* createQueue() {
    struct Queue* queue = malloc(sizeof(struct Queue));
    queue->front = -1;
    queue->rear = -1;
    return queue;
}
int isEmpty(struct Queue* queue) {
    if (queue->rear == -1)
        return 1;
    else
        return 0;
}
void enqueue(struct Queue* queue, int value) {
    if (queue->rear == MAX - 1)
printf("\nQueue is full!!");
    else {
```

```
        if (queue->front == -1)
            queue->front = 0;
        queue->rear++;
        queue->items[queue->rear] = value;
    }
}
int dequeue(struct Queue* queue) {
    int item;
    if (isEmpty(queue)) {
printf("Queue is empty");
        item = -1;
    } else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
    }
    return item;
}
// Function to implement BFS traversal
void bfs(struct Graph* graph, int startVertex) {
    struct Queue* queue = createQueue();
    // Mark the starting vertex as visited and enqueue it
    graph->visited[startVertex] = 1;
    enqueue(queue, startVertex);

printf("Breadth-First Traversal starting from vertex %d: ", startVertex);

    while (!isEmpty(queue)) {
        // Dequeue a vertex from the queue and print it
        int currentVertex = dequeue(queue);
```

```
printf("%d ", currentVertex);

    // Get all adjacent vertices of the dequeued vertex
    struct Node* temp = graph->adjLists[currentVertex];

    while (temp) {
      int adjVertex = temp->vertex;

      // If the adjacent vertex has not been visited, mark it visited and enqueue it
      if (!graph->visited[adjVertex]) {
        graph->visited[adjVertex] = 1;
        enqueue(queue, adjVertex);
      }
      temp = temp->next;
    }
  }
}

int main() {
  struct Graph* graph = createGraph(6);

addEdge(graph, 0, 1);
addEdge(graph, 0, 2);
addEdge(graph, 1, 2);
addEdge(graph, 1, 3);
addEdge(graph, 2, 4);
addEdge(graph, 3, 4);
addEdge(graph, 3, 5);
bfs(graph, 0);

  return 0;
}
```

OUTPUT

```
c:\Users\despa\OneDrive\Desktop\c dsa\output>.\"graph_BFT.exe"
Breadth-First Traversal starting from vertex 0: 0 2 1 4 3 5
c:\Users\despa\OneDrive\Desktop\c dsa\output>
```

Q2.Program to implement depth first traversal.

```
#include <stdio.h>

#include <stdlib.h>

// Structure to represent a node in the adjacency list

struct Node {

   int vertex;

   struct Node* next;

};


// Structure to represent the adjacency list

struct Graph {

   int numVertices;

   struct Node** adjLists;

   int* visited;  // Array to keep track of visited vertices

};


// Function to create a node

struct Node* createNode(int vertex) {

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

newNode->vertex = vertex;

newNode->next = NULL;

   return newNode;

}


// Function to create a graph with numVertices vertices

struct Graph* createGraph(int numVertices) {

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

```
    graph->numVertices = numVertices;

    // Create an array of adjacency lists, initialized to NULL
    graph->adjLists = malloc(numVertices * sizeof(struct Node*));
    graph->visited = malloc(numVertices * sizeof(int));

    for (int i = 0; i<numVertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;  // Mark all vertices as unvisited
    }

    return graph;
}

// Function to add an edge to the graph
void addEdge(struct Graph* graph, int src, int dest) {
    // Add edge from src to dest
    struct Node* newNode = createNode(dest);
newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src (for undirected graph)
newNode = createNode(src);
newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Function to perform Depth-First Search (DFS)
void dfs(struct Graph* graph, int vertex) {
    // Mark the current vertex as visited and print it
    graph->visited[vertex] = 1;
printf("%d ", vertex);
```

```
    // Traverse all adjacent vertices recursively
    struct Node* temp = graph->adjLists[vertex];

    while (temp) {
        int adjVertex = temp->vertex;

        // If the adjacent vertex hasn't been visited, perform DFS on it
        if (!graph->visited[adjVertex]) {
dfs(graph, adjVertex);
        }
        temp = temp->next;
    }
}

int main() {
    struct Graph* graph = createGraph(6);

addEdge(graph, 0, 1);
addEdge(graph, 0, 2);
addEdge(graph, 1, 3);
addEdge(graph, 1, 4);
addEdge(graph, 2, 4);
addEdge(graph, 3, 4);
addEdge(graph, 4, 5);

    // Perform DFS starting from vertex 0
printf("Depth-First Traversal starting from vertex 0: ");
dfs(graph, 0);

    return 0;
}
```

OUTPUT

```
c:\Users\despa\OneDrive\Desktop\c dsa\output>.\"graph_DFT.exe"
Depth-First Traversal starting from vertex 0: 0 2 4 5 3 1
c:\Users\despa\OneDrive\Desktop\c dsa\output>
```

Q3.Program to implement Dijkstra's Algorithm.

```
#include <stdio.h>

#include <limits.h>


#define V 9  // Number of vertices in the graph


// Function to find the vertex with the minimum distance value, from the set of vertices not yet processed

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

    int min = INT_MAX, min_index;


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

        if (sptSet[v] == 0 &&dist[v] <= min) {

            min = dist[v];

min_index = v;

        }

    }


    return min_index;

}


// Function to print the shortest path from the source to all vertices

void printSolution(int dist[]) {

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

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

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

    }
```

```
}

// Function to implement Dijkstra's algorithm for a graph represented using adjacency matrix
void dijkstra(int graph[V][V], int src) {
   int dist[V];  // Output array: dist[i] will hold the shortest distance from src to i
   int sptSet[V];  // sptSet[i] will be 1 if vertex i is included in the shortest path tree or shortest distance from src to i is finalized

   // Initialize all distances as INFINITE and sptSet[] as 0
   for (int i = 0; i< V; i++) {
dist[i] = INT_MAX;
sptSet[i] = 0;
   }

   // Distance of source vertex from itself is always 0
dist[src] = 0;

   // Find the shortest path for all vertices
   for (int count = 0; count < V - 1; count++) {
      // Pick the minimum distance vertex from the set of vertices not yet processed
      int u = minDistance(dist, sptSet);

      // Mark the picked vertex as processed
sptSet[u] = 1;

      // Update dist value of the adjacent vertices of the picked vertex
      for (int v = 0; v < V; v++) {
         // Update dist[v] only if it's not in sptSet, there is an edge from u to v, and the total weight of path from src to v through u is smaller than the current value of dist[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];
         }
      }
```

```
    }

    // Print the constructed distance array
printSolution(dist);
}

int main() {
    // Create a graph as an adjacency matrix
    int graph[V][V] = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };

    // Call Dijkstra's algorithm from vertex 0 (or any source vertex)
dijkstra(graph, 0);

    return 0;
}
```

OUTPUT

```
c:\Users\despa\OneDrive\Desktop\c dsa\output>cd "c:\Users\despa\OneDrive\Desktop\c dsa\output"

c:\Users\despa\OneDrive\Desktop\c dsa\output>.\"DijkstraAlgorithm.exe"
Vertex   Distance from Source
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8
8                14

c:\Users\despa\OneDrive\Desktop\c dsa\output>
```

Q4.Program to implement Prim's Algorithm

```
#include <stdio.h>

#include <limits.h>


#define V 5  // Number of vertices in the graph


// Function to find the vertex with the minimum key value, from the set of vertices not yet included in MST

int minKey(int key[], int mstSet[]) {

   int min = INT_MAX, min_index;


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

      if (mstSet[v] == 0 && key[v] < min) {

         min = key[v], min_index = v;

      }

   }


   return min_index;

}


// Function to print the constructed MST stored in parent[]

void printMST(int parent[], int graph[V][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[V][V]) {
    int parent[V];  // Array to store the constructed MST
    int key[V];     // Key values used to pick the minimum weight edge in cut
    int mstSet[V];  // To represent set of vertices included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i< V; i++) {
        key[i] = INT_MAX, mstSet[i] = 0;
    }

    // Always include the first vertex in MST
    key[0] = 0;     // Make key 0 so that this vertex is picked first
    parent[0] = -1; // First node is always the root of MST

    // 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);

        // Add the picked vertex to the MST set
mstSet[u] = 1;

        // Update key value and parent index of the adjacent vertices of the picked vertex
        for (int v = 0; v < V; v++) {
            // graph[u][v] is non-zero only for adjacent vertices of u
            // 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] == 0 && graph[u][v] < key[v]) {
                parent[v] = u, key[v] = graph[u][v];
            }
        }
    }

    // Print the constructed MST
printMST(parent, graph);
}

int main() {
    // Graph represented as an adjacency matrix
    int graph[V][V] = {
        {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},
    };

    // Run Prim's algorithm to find the minimum spanning tree
primMST(graph);

    return 0;
}
```

OUTPUT

```
C:\Users\despa\OneDrive\Desktop\c dsa>cd "c:\Users\despa\OneDrive\Desktop\c dsa\output"

c:\Users\despa\OneDrive\Desktop\c dsa\output>.\"primsAlgoGraph.exe"
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5

c:\Users\despa\OneDrive\Desktop\c dsa\output>
```

Q5.Program to implement Strassen's Matrix Multiplication Algorithm using divide and conquer

```
#include <stdio.h>

#include <stdlib.h>


// Function to add two matrices

void add(int A[2][2], int B[2][2], int C[2][2]) {

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

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

            C[i][j] = A[i][j] + B[i][j];

        }

    }

}


// Function to subtract two matrices

void subtract(int A[2][2], int B[2][2], int C[2][2]) {

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

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

            C[i][j] = A[i][j] - B[i][j];

        }

    }

}


// Function to perform Strassen's Matrix Multiplication

void strassen(int A[2][2], int B[2][2], int C[2][2]) {
```

```
    int M1, M2, M3, M4, M5, M6, M7;
    int temp1[2][2], temp2[2][2];

    // Calculate M1 to M7 based on Strassen's formula
    M1 = (A[0][0] + A[1][1]) * (B[0][0] + B[1][1]);
    M2 = (A[1][0] + A[1][1]) * B[0][0];
    M3 = A[0][0] * (B[0][1] - B[1][1]);
    M4 = A[1][1] * (B[1][0] - B[0][0]);
    M5 = (A[0][0] + A[0][1]) * B[1][1];
    M6 = (A[1][0] - A[0][0]) * (B[0][0] + B[0][1]);
    M7 = (A[0][1] - A[1][1]) * (B[1][0] + B[1][1]);

    // Calculate C matrix using M1 to M7
    C[0][0] = M1 + M4 - M5 + M7;
    C[0][1] = M3 + M5;
    C[1][0] = M2 + M4;
    C[1][1] = M1 - M2 + M3 + M6;
}

// Function to print the matrix
void printMatrix(int matrix[2][2]) {
    for (int i = 0; i< 2; i++) {
        for (int j = 0; j < 2; j++) {
printf("%d ", matrix[i][j]);
        }
printf("\n");
    }
}

int main() {
    int A[2][2] = {{1, 2}, {3, 4}};
    int B[2][2] = {{5, 6}, {7, 8}};
```

```
    int C[2][2];  // To store result

printf("Matrix A:\n");
printMatrix(A);

printf("\nMatrix B:\n");
printMatrix(B);

    // Perform Strassen's matrix multiplication
strassen(A, B, C);

printf("\nResult matrix C after Strassen's multiplication:\n");
printMatrix(C);

    return 0;
}
```

OUTPUT

```
c:\Users\despa\OneDrive\Desktop\c dsa\output>.\"BFT.exe"
Matrix A:
1 2
3 4

Matrix B:
5 6
7 8

Result matrix C after Strassen's multiplication:
19 22
43 50

c:\Users\despa\OneDrive\Desktop\c dsa\output>
```