Q.1) Write a C program to implement Depth First Search (DFS) and Breadth First Search in graph.

Answer:

Depth First Search (DFS) Algorithm :

- Visit the current node.
- Recursively traverse the left subtree.
- Recursively traverse the right subtree.

Breadth First Search (BFS) Algorithm:

1. Initialize an empty queue and enqueue the starting node.
2. Mark the starting node as visited.
3. While the queue is not empty,do the following:
    a. Dequeue a node (current node) and visit it.
    b. For each unvisited adjacent node of the current node:
        i. Mark it as visited and enqueue it.
4. Otherwise, stop.

Example:

Program:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
   int data;
   struct Node *left, *right;
} Node;
typedef struct Queue {
```

```
    Node **array;
    int front, rear, size;
} Queue;
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}
Queue* createQueue(int size) {
    Queue* q = (Queue*)malloc(sizeof(Queue));
    q->array = (Node**)malloc(size * sizeof(Node*));
    q->front = q->rear = -1;
    q->size = size;
    return q;
}
void enqueue(Queue* q, Node* node) {
    if (q->rear == q->size - 1) return;
    q->array[++q->rear] = node;
    if (q->front == -1) q->front = 0;
}
Node* dequeue(Queue* q) {
    if (q->front == -1) return NULL;
    Node* temp = q->array[q->front];
    if (q->front == q->rear) q->front = q->rear = -1;
    else q->front++;
    return temp;
}
// BFS Traversal
void BFS(Node* root) {
    if (root == NULL) return;
    Queue* q = createQueue(10);
    enqueue(q, root);
    while (q->front != -1) {
```

```
        Node* current = dequeue(q);
        printf("%d ", current->data);
        if (current->left != NULL) enqueue(q, current->left);
        if (current->right != NULL) enqueue(q, current->right);
    }
    free(q->array);
    free(q);
}
// DFS Traversal
void DFS(Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    DFS(root->left);
    DFS(root->right);
}
Node* buildTree() {
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(5);
    root->left->left = createNode(3);
    root->left->right = createNode(4);
    root->right->left = createNode(6);
    root->right->right = createNode(7);
    return root;
}
int main() {
    Node* root = buildTree();
    printf("DFS Traversal: ");
    DFS(root);
    printf("\nBFS Traversal: ");
    BFS(root);
    printf("\n");
    return 0;
}
```

Output:

```
user@DolindraBahadurRaut:~/DSA$ gcc searchGraph.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
DFS Traversal: 1 2 3 4 5 6 7
BFS Traversal: 1 2 5 3 4 6 7
```

Conclusion:

Hence, We successfully created a C program to implement Depth First Search (DFS) and Breadth First Search in graph.

Q.2) Write programs to implement Prim's and Kruskal Algorithm for minimum spanning tree.

Answer:

Prim's Algorithm:

- Initialize the minimum spanning tree with a single vertex selected at random from the graph.
- Find all edges that connect the tree to vertices that are not yet in the tree.
- Select the edge with the smallest weight from the set of edges found in step 2 and add it to the minimum spanning tree.
- Add the new vertex that is connected to the selected edge to the minimum spanning tree.
- Repeat steps 2–4 until all vertices are part of the minimum spanning tree.

Kruskal's Algorithm:

- Sort all edges in increasing order of weight.
- Initialize each vertex as a separate set (Disjoint Set/Union-Find).
- For each edge in sorted order:
    - If the edge connects two different sets, add it to the MST and merge the sets.
- Repeat until MST has (V - 1) edges.

Example:

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define V 4  // Number of vertices
// ---------- For Kruskal's ----------
struct Edge {
    int src, dest, weight;
};

struct subset {
    int parent;
    int rank;
};

int find(struct subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y) {
    int rootX = find(subsets, x);
    int rootY = find(subsets, y);

    if (subsets[rootX].rank < subsets[rootY].rank)
        subsets[rootX].parent = rootY;
    else if (subsets[rootX].rank > subsets[rootY].rank)
        subsets[rootY].parent = rootX;
    else {
        subsets[rootY].parent = rootX;
        subsets[rootX].rank++;
    }
}

int compareEdges(const void *a, const void *b) {
    return ((struct Edge *)a)->weight - ((struct Edge *)b)->weight;
}

void kruskalMST(struct Edge edges[], int E) {
    struct Edge result[V];  // To store result MST
    int e = 0, i = 0;
```

```
    qsort(edges, E, sizeof(edges[0]), compareEdges);

    struct subset *subsets = (struct subset *)malloc(V * sizeof(struct subset));
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    while (e < V - 1 && i < E) {
        struct Edge next = edges[i++];
        int x = find(subsets, next.src);
        int y = find(subsets, next.dest);

        if (x != y) {
            result[e++] = next;
            Union(subsets, x, y);
        }
    }

    printf("\n*******Kruskal's MST*******\nEdge \tWeight\n");
    for (i = 0; i < e; i++)
        printf("%d - %d \t%d\n", result[i].src, result[i].dest, result[i].weight);

    free(subsets);
}
// ---------- For Prim's ----------
int minKey(int key[], int mstSet[]) {
    int min = INT_MAX, minIndex = -1;
    for (int v = 0; v < V; v++) {
        if (mstSet[v] == 0 && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }
    return minIndex;
}

void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    int mstSet[V];

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

```
        key[i] = INT_MAX;
        mstSet[i] = 0;
    }
    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = 1;

        for (int v = 0; v < V; v++) {
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    printf("*******Prim's MST*******\nEdge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
}

int main() {
    int graph[V][V] = {
        {0, 10, 6, 5},
        {10, 0, 15, 0},
        {6, 15, 0, 4},
        {5, 0, 4, 0}
    };

    struct Edge edges[] = {
        {0, 1, 10},
        {0, 2, 6},
        {0, 3, 5},
        {1, 3, 15},
        {2, 3, 4}
    };

    int E = sizeof(edges) / sizeof(edges[0]);

    primMST(graph);
    kruskalMST(edges, E);
```

```
    return 0;
}
```

Output:

```
user@DolindraBahadurRaut:~/DSA$ gcc MST.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
*******Prim's MST*******
Edge    Weight
0 - 1   10
3 - 2   4
0 - 3   5

*******Kruskal's MST*******
Edge    Weight
2 - 3   4
0 - 3   5
0 - 1   10
```

Conclusion:

Hence, we successfully implemented a C program to find the minimum spanning tree using Prim's and Kruskal's algorithm.

Q.3) Write a program to implement Dijkstra's Algorithm.

Answer:

Algorithm:

- Initialize a distance array dist[] where dist[i] represents the shortest distance from the source to vertex i. Set dist[source] = 0 and all others to infinity (INF).
- Use a priority queue (min-heap) to track unvisited nodes, starting with the source.
- While the queue is not empty:
    - Extract the node u with the smallest tentative distance.
    - For each adjacent node v, if dist[u] + weight(u, v) < dist[v], update dist[v] and add v to the queue.
- Repeat until all nodes are visited.

Example:

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define V 5
void dijkstra(int graph[V][V], int src) {
    int dist[V];
    int visited[V] = {0};
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
    }
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u, min_dist = INT_MAX;
        for (int v = 0; v < V; v++) {
            if (!visited[v] && dist[v] < min_dist) {
                min_dist = dist[v];
                u = v;
            }
        }
        visited[u] = 1;
        for (int v = 0; v < V; v++) {
            if (!visited[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
```

```
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
    printf("Vertex\t\tDistance from Source\n");
    for (int i = 0; i < V; i++) {
        printf("%d \t\t\t %d\n", i, dist[i]);
    }
}
int main() {
    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}
    };
    int source;
    printf("Enter source vertex: ");
    scanf("%d",&source);
    printf("Source = %d\n",source);
    dijkstra(graph, source);
    return 0;
}
```


Output:

```
user@DolindraBahadurRaut:~/DSA$ gcc dijkstra.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Enter source vertex: 0
Source = 0
Vertex          Distance from Source
0                       0
1                       2
2                       5
3                       6
4                       7
```

Conclusion:

Hence, we successfully implemented a C program to find the shortest path using Dijkstra's algorithm.

Q.4) Write a program to implement Binary Search Trees basic operations.

Answer:

Insertion Algorithm :

1. Start at the root.
2. If the tree is empty, create a new node with the given key and set it as the root.
3. If the tree is not empty:
    a. Compare the key with the current node.
    b. If the key is smaller than the current node's key, move to the left child.
    c. If the key is larger than the current node's key, move to the right child.
4. Repeat step 3 until you find a null spot (left or right child is null).
5. Insert the new node at the found null spot.

Deletion Algorithm:

1. Case 1: Node has on child.
    a. Directly delete the node.
2. Case 2: Node has 1 child.
    a. Replace with its child.
3. Case 3: Node has 2 child:
    a. Replace with its in order successor
    b. Delete the successor.

Traversal Algorithm:

Inorder Traversal (Left, Node, Right):

- Visit the left subtree.
- Visit the current node.
- Visit the right subtree.

Preorder Traversal (Node, Left, Right):

- Visit the current node.
- Visit the left subtree.
- Visit the right subtree.

Postorder Traversal (Left, Right, Node):

- Visit the left subtree.
- Visit the right subtree.
- Visit the current node.

Example:

Initial Tree: Empty tree (root is NULL).

Insert: 25, 35, 10, 7, 15, 28, 40

Delete 7

Delete 10 (node with one child).

Delete 25(root with two children).

Inorder traversal: 15, 28, 35, 40

Pre-order traversal: 28, 15, 35, 40

Post-order traversal: 15, 40, 35, 28

Program:

```
#include <stdio.h>
#include <stdlib.h>
// Definition of a BST node
struct Node {
    int data;
    struct Node *left, *right;
};
// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}
// Function to insert a node into the BST
struct Node* insert(struct Node* root, int data) {
    // If the tree is empty, create a new node
    if (root == NULL) {
```

```
        return newNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    // Loop down to find the leftmost leaf
    while (current && current->left != NULL) {
        current = current->left;
    }
    return current;
}
// Function to delete a node from the BST
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL) {
        return root;
    }
    if (key < root->data) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->data) {
        root->right = deleteNode(root->right, key);
    } else {
        // Case 1: Node has no children (leaf node)
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
        }
        // Case 2: Node has one child
        else if (root->left == NULL) {
```

```
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }
        // Case 3: Node has two children
        else {
            struct Node* temp = minValueNode(root->right); // In-order successor
            root->data = temp->data; // Copy the inorder successor's content to this node
            root->right = deleteNode(root->right, temp->data); // Delete the inorder successor
        }
    }
    return root;
}
// Inorder traversal of the BST
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
// Preorder traversal of the BST
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
// Postorder traversal of the BST
```

```
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
int main() {
    struct Node* root = NULL;

    // Insert values into the BST
    root = insert(root, 25);
    root = insert(root, 35);
    root = insert(root, 10);
    root = insert(root, 7);
    root = insert(root, 15);
    root = insert(root, 28);
    root = insert(root, 40);

    printf("Inorder Traversal before deletion: ");
    inorder(root);  // Perform Inorder Traversal
    printf("\n");

    root = deleteNode(root, 7);    // Deleting node 7 (leaf node)
    printf("Inorder Traversal after deleting 7: ");
    inorder(root);
    printf("\n");

    root = deleteNode(root, 10);   // Deleting node 10 (node with one child)
    printf("Inorder Traversal after deleting 10: ");
    inorder(root);
    printf("\n");
    root = deleteNode(root, 25);    // Deleting node 25 (root node with two children)
    printf("Inorder Traversal after deleting 25: ");
    inorder(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorder(root); // Perform Preorder Traversal
```

```
    printf("\n");

    printf("Postorder Traversal: ");
    postorder(root);   // Perform Postorder Traversal
    printf("\n");
    return 0;
}
```

Output:

```
user@DolindraBahadurRaut:~/DSA$ gcc BST.c
user@DolindraBahadurRaut:~/DSA$ ./a.out
Inorder Traversal before deletion: 7 10 15 25 28 35 40
Inorder Traversal after deleting 7: 10 15 25 28 35 40
Inorder Traversal after deleting 10: 15 25 28 35 40
Inorder Traversal after deleting 25: 15 28 35 40
Preorder Traversal: 28 15 35 40
Postorder Traversal: 15 40 35 28
```

Conclusion:

Therefore, we successfully implemented insertion, deletion and traversal operation on Binary search Tree.