ASSIGNMENT - 2

Q1.

```
#include<stdio.h>
#include<stdlib.h>
struct node{
   int data;
   struct node *left, *right;
};
struct node *insertBST(struct node *root, int value){
   if(root == NULL){
      struct node *newNode;
      newNode = (struct node*)malloc(sizeof(struct node));
      newNode->data = value;
      newNode->left = newNode->right = NULL;
      return newNode;
   }else{
      if(value < root->data)
         root->left = insertBST(root->left, value);
      else if(value > root->data)
         root->right = insertBST(root->right, value);
      else{
         printf("Duplicate Entries are not allowed!!");
         exit(0);
      }
   }
}
void inorder(struct node *root){
   if(root == 0)
      return;
   inorder(root->left);
   printf("%d ",root->data);
   inorder(root->right);
}
int nonleafnode = 0, leafnode = 0;
void count_nodes(struct node *root){
   if(root == NULL)
      return;
   else{
      if(root->left != NULL || root->right != NULL)
         nonleafnode++;
      else
         leafnode++;
      count_nodes(root->left);
      count_nodes(root->right);
   }
}
int main(){
   struct node *root = NULL;
   while(1){
```

```
        int x;
        printf("enter the data: ");
        scanf("%d",&x);
        if(x == -1)
            break;
        root = insertBST(root, x);
    }
    inorder(root);
    count_nodes(root);
    printf("\nThe number of leaf nodes is %d and non leaf nodes is
%d",leafnode,nonleafnode);
}
```

Q2.

```
#include<stdio.h>
#include<stdlib.h>

struct node {
    int data;
    struct node *left, *right;
};

struct node *create() {
    int x;
    struct node *newnode = (struct node*)malloc(sizeof(struct node));
    printf("\nEnter the data(-1 for NULL): ");
    scanf("%d", &x);
    if (x == -1)
        return NULL;
    newnode->data = x;
    printf("\nEnter the left child of the %d", newnode->data);
    newnode->left = create();
    printf("\nEnter the right child of the %d", newnode->data);
    newnode->right = create();
    return newnode;
}

struct node* mir(struct node *root) {
    if (root == NULL)
        return NULL;
    else {
        struct node *mirror = (struct node*)malloc(sizeof(struct node));
        mirror->data = root->data;
        mirror->left = mir(root->right);
```

```
        mirror->right = mir(root->left);
        return mirror;
    }
}

void inorder(struct node *root) {
    if (root == NULL)
        return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

int main() {
    struct node *root = NULL;
    root = create();
    printf("\nInorder traversal of the original tree: ");
    inorder(root);
    struct node *mirror = NULL;
    mirror = mir(root);
    printf("\nInorder traversal of the mirror tree: ");
    inorder(mirror);
    free(root);
    free(mirror);
    return 0;
}
```

Q3.
Algorithm for Displaying Nodes within a Given Range:

1.Start the function BSTinRange with parameters root, k1, and k2.
2.Check if the root is NULL (base case):
 If root is NULL, return.
3.Check if the value of root->data lies within the range defined by k1 and k2 (inclusive):
 If root->data is greater than k1 and less than k2, print root->data.
4.Recursively call the BSTinRange function for the left subtree with parameters root->left, k1, and k2.
5.Recursively call the BSTinRange function for the right subtree with parameters root->right, k1, and k2.
6.End the function.
(it is assumed that bst tree is already created).

Q4.
Algorithm for Topological Sort:

1.Represent the graph using an adjacency matrix or adjacency list.
2.Create a function called DFS (Depth-First Search) to perform the topological sort. The function should take the current vertex, a list to track visited vertices, a stack to store the result, and the graph as parameters.
3.Inside the DFS function:
a. Mark the current vertex as visited.
b. For each unvisited neighbour (vertex) of the current vertex, recursively call the DFS function on that neighbour .
c. After visiting all neighbours, push the current vertex onto the stack.
4.Create another function called topologicalSort that takes the graph and the number of vertices as parameters.
5. Initialise a list to keep track of visited vertices and set all elements to "not visited."
6. Initialise a stack to store the topological sort result.
7. For each unvisited vertex in the graph:
   Call the DFS function on that vertex to explore its connected vertices.
8. After visiting all vertices, the stack will contain the topological sort result in reverse order.
9.Print the topological sort result by popping elements from the stack one by one.

Q5.
Algorithm for Finding Shortest Path between Two Vertices:

1.Represent the graph using a list of edges, where each edge has a starting vertex, an ending vertex, and a weight (distance).
2.Create a function called FindShortestPath that takes the graph, the number of vertices in the graph, the starting vertex, and the ending vertex as inputs.
3.Inside the FindShortestPath function:
  a. Set all distances to "infinity" (except the starting vertex, which has a distance of 0).
  b. Create a priority queue (a special list where the smallest distance element is always at the front).
  c. Add the starting vertex to the priority queue with distance 0.
4.While the priority queue is not empty:
  a. Take out the vertex with the smallest distance from the priority queue (let's call it "currentVertex").
  b. If the "currentVertex" is the ending vertex, we found the shortest path. Stop the algorithm.
  c. For each neighbour of the "currentVertex":
    i. Calculate the distance from the starting vertex to this neighbour through the "currentVertex".
    ii. If this distance is smaller than the current distance stored for this neighbour , update the distance.
    iii. Add the neighbour to the priority queue with the updated distance.

5.After the loop ends, we have the shortest distance from the starting vertex to all other vertices in the graph, including the ending vertex.
6.To find the actual shortest path, backtrack from the ending vertex to the starting vertex using the information stored during the algorithm.
7.Return the shortest path distance and the path itself.
