LAB ASSIGNMENT 2

1. Implement Binary Search Tree (Insertion & searching)

A. CODE:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
 struct node
  {
    int data;
    struct node *left,*right;
  };
 struct node *  insert(int, struct node *);
 void search(int, struct node *);
 void display(struct node *);

 void main()
 {
  int ch;
  int a;
  struct node * root=NULL,*temp;
  while(1)
          {
      printf("\n1. Insert\n 2. Find \n 3. Display\n 4. Exit\n Enter Your Choice : ");
      scanf("%d",&ch);
      switch(ch)
       {
        case 1:
        printf("Enter the Data : ");
        scanf("%d", &a);
        root = insert(a, root);
        break;
        case 2:
                  printf("\nEnter the data to be searched : ");
                  scanf("%d",&a);
                  search(a, root);
```

```
                    break;
                    case 3:
                    if(root==NULL)
                       printf("\nEmpty tree");
                    else
                       display(root);
                    break;
                    case 4:
                    exit(0);
                    default:printf("Invalid Choice");
                    }
          }
}
struct node * insert(int x,struct node * t)
  {
     if(t==NULL)
            {
                    t = (struct node *)malloc(sizeof(struct node *));
                    t->data = x;
                    t->left = t->right = NULL;
            }
          else
      {
              if(x < t->data)
                    t->left = insert(x, t->left);
                    else if(x > t->data)
                    t->right = insert(x, t->right);
      }
          return t;
  }


void search(int x, struct node * t)
{
   if(t==NULL)
           printf("Data is not found");
   else if(x<t->data)
           search(x,t->left);
   else if(x>t->data)
           search(x,t->right);
```

```
   else
          printf("Data is Found");
}

 void display(struct node * t)
 {
   if(t)
    {
           display(t->left);
      printf("%d\t",t->data);
      display(t->right);
    }
 }
```

SAMPLE INPUT AND SAMPLE OUTPUT:

```
1. Insert
 2. Find
 3. Display
 4. Exit
 Enter Your Choice : 1
Enter the Data : 26

1. Insert
 2. Find
 3. Display
 4. Exit
 Enter Your Choice : 1
Enter the Data : 89

1. Insert
 2. Find
 3. Display
 4. Exit
 Enter Your Choice : 2

Enter the data to be searched : 90
Data is not found
```

```
1. Insert
 2. Find
 3. Display
 4. Exit
 Enter Your Choice : 2

Enter the data to be searched : 89
Data is Found
1. Insert
 2. Find
 3. Display
 4. Exit
 Enter Your Choice : 3
26      89
1. Insert
 2. Find
 3. Display
 4. Exit
 Enter Your Choice : 4
```

Time Complexity Analysis:

1. Time complexity of all BST Operations = O(h)
   Here, h = Height of binary search tree

2. In worst case,

   - The binary search tree is a skewed binary search tree.
   - Height of the binary search tree becomes n.
   - So, Time complexity of BST Operations = O(n).

3, In best case,

   - The binary search tree is a balanced binary search tree.
   - Height of the binary search tree becomes log(n).
   - So, Time complexity of BST Operations = O(logn).

2. Implement Heap Sort Algorithm

A. __CODE__:

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

void main()
{
	int a[10],size,i,j,c,p,temp;
	printf("Enter the Size of an Array : \n");
	scanf("%d",&size);
	printf("Enter the Numbers into Array: \n");
	for(i=1;i<=size;i++)
	  scanf("%d",&a[i]);
	for(i=2;i<=size;i++)
	{
	  c=i;
	  do
	  {
		p=(c)/2;
		if(a[p]<a[c])
		{
			temp=a[p];
			a[p]=a[c];
			a[c]=temp;
		}
		c=p;
	  }while(c!=1);
	}
	for(j=size;j>=1;j--)
	{
	  temp=a[1];
	  a[1]=a[j];
	  a[j]=temp;
	  p=1;
	  do{
		c=2*p;
		if((a[c]<a[c+1]) && c<j-1)
		  c++;
```

```
                if(a[p]<a[c] && c<j)
                    {
                    temp=a[p];
                    a[p]=a[c];
                    a[c]=temp;
                }
                p=c;
           }while(c<j);
        }
        printf("The Sorted Array is: \n ");
        for(i=1;i<=size;i++)
           printf("%d ",a[i]);
        getch();
}
```

<u>SAMPLE</u> <u>INPUT</u> <u>AND</u> <u>SAMPLE</u> <u>OUTPUT</u>:

```
Enter the Size of an Array :
3
Enter the Numbers into Array:
2
56
89
The Sorted Array is:
 2 56 89
```

<u>Time</u> <u>Complexity</u> <u>Analysis</u>:

1. Avg Time complexity of Heap Sort is O(nlog n)

2. In Best case
    - Time Complexity is O(nlog n)

3. In Worst case
    - Time Complexity is O(nlog n)

3. Implement the solution for the fractional knapsack problem.

A. __CODE__:

```
#include <stdio.h>
int n = 5; /* The number of objects */
int c[10] = {12, 1, 2, 1, 4}; /* c[i] is the *COST* of the ith object; i.e. what  YOU PAY to
take the object */
int v[10] = {4, 2, 2, 1, 10}; /* v[i] is the *VALUE* of the ith object; i.e. what YOU GET for
taking the object */
int W = 15; /* The maximum weight you can take */
void simple_fill() {
    int cur_w;
    float tot_v;
    int i, maxi;
    int used[10];
    for (i = 0; i < n; ++i)
        used[i] = 0; /* I have not used the ith object yet */
    cur_w = W;
    while (cur_w > 0) { /* while there's still room*/
        /* Find the best object */
        maxi = -1;
        for (i = 0; i < n; ++i)
            if ((used[i] == 0) &&
                ((maxi == -1) || ((float)v[i]/c[i] > (float)v[maxi]/c[maxi])))
                maxi = i;

        used[maxi] = 1; /* mark the maxi-th object as used */
        cur_w -= c[maxi]; /* with the object in the bag, I can carry less */
        tot_v += v[maxi];
        if (cur_w >= 0)
            printf("Added object %d (%d$, %dKg) completely in the bag. Space left: %d.\n",
maxi + 1, v[maxi], c[maxi], cur_w);
        else {
            printf("Added %d%% (%d$, %dKg) of object %d in the bag.\n", (int)((1 +
(float)cur_w/c[maxi]) * 100), v[maxi], c[maxi], maxi + 1);
            tot_v -= v[maxi];
            tot_v += (1 + (float)cur_w/c[maxi]) * v[maxi];
```

```
        }
    }

    printf("Filled the bag with objects worth %.2f$.\n", tot_v);
}

int main(int argc, char *argv[]) {
    simple_fill();

    return 0;
}
```

SAMPLE INPUT AND SAMPLE OUTPUT:

```
Added object 5 (10$, 4Kg) completely in the bag. Space left: 11.
Added object 2 (2$, 1Kg) completely in the bag. Space left: 10.
Added object 3 (2$, 2Kg) completely in the bag. Space left: 8.
Added object 4 (1$, 1Kg) completely in the bag. Space left: 7.
Added 58% (4$, 12Kg) of object 1 in the bag.
Filled the bag with objects worth 17.33$.
```

Time Complexity Analysis:

Time complexity of the sorting + Time complexity of the loop to maximize profit = O(NlogN) + O(N) = O(NlogN)

4. Represent a graph using adjacency matrix and adjacency list and display them

A. <u>CODE</u>:

(i) Adjacency matrix

```
#include <stdio.h>
#define V 4

// Initialize the matrix to zero
void init(int arr[][V]) {
  int i, j;
  for (i = 0; i < V; i++)
    for (j = 0; j < V; j++)
      arr[i][j] = 0;
}

// Add edges
void addEdge(int arr[][V], int i, int j) {
  arr[i][j] = 1;
  arr[j][i] = 1;
}

// Print the matrix
void printAdjMatrix(int arr[][V]) {
  int i, j;

  for (i = 0; i < V; i++) {
    printf("%d: ", i);
    for (j = 0; j < V; j++) {
      printf("%d ", arr[i][j]);
    }
    printf("\n");
  }
}

int main() {
  int adjMatrix[V][V];
```

```
  init(adjMatrix);
  addEdge(adjMatrix, 0, 1);
  addEdge(adjMatrix, 0, 2);
  addEdge(adjMatrix, 1, 2);
  addEdge(adjMatrix, 2, 0);
  addEdge(adjMatrix, 2, 3);
  printAdjMatrix(adjMatrix);
  return 0;
}
```

(ii) Adjacency list

```
#include <stdio.h>
#include <stdlib.h>
struct node {
  int vertex;
  struct node* next;
};
struct node* createNode(int);
struct Graph {
  int numVertices;
  struct node** adjLists;
};
// Create a node
struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}
// Create a graph
struct Graph* createAGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;
  graph->adjLists = malloc(vertices * sizeof(struct node*));
  int i;
  for (i = 0; i < vertices; i++)
    graph->adjLists[i] = NULL;
  return graph;
}
// Add edge
```

```
void addEdge(struct Graph* graph, int s, int d) {
  // Add edge from s to d
  struct node* newNode = createNode(d);
  newNode->next = graph->adjLists[s];
  graph->adjLists[s] = newNode;
  // Add edge from d to s
  newNode = createNode(s);
  newNode->next = graph->adjLists[d];
  graph->adjLists[d] = newNode;
}
// Print the graph
void printGraph(struct Graph* graph) {
  int v;
  for (v = 0; v < graph->numVertices; v++) {
    struct node* temp = graph->adjLists[v];
    printf("\n Vertex %d\n: ", v);
    while (temp) {
      printf("%d -> ", temp->vertex);
      temp = temp->next;
    }
    printf("\n");
  }
}
int main() {
  struct Graph* graph = createAGraph(4);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 0, 3);
  addEdge(graph, 1, 2);
  printGraph(graph);
  return 0;
}
```

SAMPLE INPUT AND SAMPLE OUTPUT:

(i) Adjacency matrix

```
0: 0 1 1 0
1: 1 0 1 0
2: 1 1 0 1
3: 0 0 1 0
```

(ii) Adjacency list

```
 Vertex 0
: 3 -> 2 -> 1 ->

 Vertex 1
: 2 -> 0 ->

 Vertex 2
: 1 -> 0 ->

 Vertex 3
: 0 ->
```

Time Complexity Analysis:

(i) Adjacency matrix

Time Complexity of Adjacency matrix is O(n^2)

(ii) Adjacency list

Time Complexity of Adjacency List is O(m)

In Worst case

- Time Complexity is O(n^2)

5. Implement Kruskal's algorithm for Minimum Spanning Tree.

A. <u>CODE</u>:

```
#include <stdio.h>

#define MAX 30

typedef struct edge {
  int u, v, w;
} edge;

typedef struct edge_list {
  edge data[MAX];
  int n;
} edge_list;

edge_list elist;

int Graph[MAX][MAX], n;
edge_list spanlist;

void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();

// Applying Krushkal Algo
void kruskalAlgo() {
  int belongs[MAX], i, j, cno1, cno2;
  elist.n = 0;

  for (i = 1; i < n; i++)
    for (j = 0; j < i; j++) {
      if (Graph[i][j] != 0) {
        elist.data[elist.n].u = i;
        elist.data[elist.n].v = j;
```

```
        elist.data[elist.n].w = Graph[i][j];
        elist.n++;
      }
    }

  sort();

  for (i = 0; i < n; i++)
    belongs[i] = i;

  spanlist.n = 0;

  for (i = 0; i < elist.n; i++) {
    cno1 = find(belongs, elist.data[i].u);
    cno2 = find(belongs, elist.data[i].v);

    if (cno1 != cno2) {
      spanlist.data[spanlist.n] = elist.data[i];
      spanlist.n = spanlist.n + 1;
      applyUnion(belongs, cno1, cno2);
    }
  }
}

int find(int belongs[], int vertexno) {
  return (belongs[vertexno]);
}

void applyUnion(int belongs[], int c1, int c2) {
  int i;

  for (i = 0; i < n; i++)
    if (belongs[i] == c2)
      belongs[i] = c1;
}

// Sorting algo
void sort() {
  int i, j;
  edge temp;
```

```
  for (i = 1; i < elist.n; i++)
    for (j = 0; j < elist.n - 1; j++)
      if (elist.data[j].w > elist.data[j + 1].w) {
        temp = elist.data[j];
        elist.data[j] = elist.data[j + 1];
        elist.data[j + 1] = temp;
      }
}

// Printing the result
void print() {
  int i, cost = 0;

  for (i = 0; i < spanlist.n; i++) {
    printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
    cost = cost + spanlist.data[i].w;
  }

  printf("\nSpanning tree cost: %d", cost);
}

int main() {
  int i, j, total_cost;

  n = 6;

  Graph[0][0] = 0;
  Graph[0][1] = 4;
  Graph[0][2] = 4;
  Graph[0][3] = 0;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;

  Graph[1][0] = 4;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
  Graph[1][4] = 0;
```

```
  Graph[1][5] = 0;
  Graph[1][6] = 0;

  Graph[2][0] = 4;
  Graph[2][1] = 2;
  Graph[2][2] = 0;
  Graph[2][3] = 3;
  Graph[2][4] = 4;
  Graph[2][5] = 0;
  Graph[2][6] = 0;

  Graph[3][0] = 0;
  Graph[3][1] = 0;
  Graph[3][2] = 3;
  Graph[3][3] = 0;
  Graph[3][4] = 3;
  Graph[3][5] = 0;
  Graph[3][6] = 0;

  Graph[4][0] = 0;
  Graph[4][1] = 0;
  Graph[4][2] = 4;
  Graph[4][3] = 3;
  Graph[4][4] = 0;
  Graph[4][5] = 0;
  Graph[4][6] = 0;

  Graph[5][0] = 0;
  Graph[5][1] = 0;
  Graph[5][2] = 2;
  Graph[5][3] = 0;
  Graph[5][4] = 3;
  Graph[5][5] = 0;
  Graph[5][6] = 0;

  kruskalAlgo();
  print();
}
```

SAMPLE INPUT AND SAMPLE OUTPUT:

```
2 - 1 : 2
5 - 2 : 2
3 - 2 : 3
4 - 3 : 3
1 - 0 : 4
Spanning tree cost: 14
```

Time Complexity Analysis:

The time complexity Of Kruskal's Algorithm is: O(E log E)

6. Implement Prim's Algorithm for Minimum Spanning Tree.

A. <u>CODE</u>:

```
#include<stdio.h>
#include<stdbool.h>
#include<string.h>
#define INF 9999999

// number of vertices in graph
#define V 5

// create a 2d array of size 5x5
//for adjacency matrix to represent graph
int G[V][V] = {
  {0, 9, 75, 0, 0},
  {9, 0, 95, 19, 42},
  {75, 95, 0, 51, 66},
  {0, 19, 51, 0, 31},
  {0, 42, 66, 31, 0}};

int main() {
  int no_edge;  // number of edge

  // create a array to track selected vertex
  // selected will become true otherwise false
  int selected[V];

  // set selected false initially
  memset(selected, false, sizeof(selected));

  // set number of edge to 0
  no_edge = 0;

  // the number of egde in minimum spanning tree will be
  // always less than (V -1), where V is number of vertices in
  //graph
```

```
  // choose 0th vertex and make it true
  selected[0] = true;

  int x;  //  row number
  int y;  //  col number

  // print for edge and weight
  printf("Edge : Weight\n");

  while (no_edge < V - 1) {
    //For every vertex in the set S, find the all adjacent vertices
    // , calculate the distance from the vertex selected at step 1.
    // if the vertex is already in the set S, discard it otherwise
    //choose another vertex nearest to selected vertex  at step 1.

    int min = INF;
    x = 0;
    y = 0;

    for (int i = 0; i < V; i++) {
      if (selected[i]) {
        for (int j = 0; j < V; j++) {
          if (!selected[j] && G[i][j]) {  // not in selected and there is an edge
            if (min > G[i][j]) {
              min = G[i][j];
              x = i;
              y = j;
            }
          }
        }
      }
    }
    printf("%d - %d : %d\n", x, y, G[x][y]);
    selected[y] = true;
    no_edge++;
  }

  return 0;
}
```

SAMPLE INPUT AND SAMPLE OUTPUT:

```
Edge : Weight
0 - 1 : 9
1 - 3 : 19
3 - 4 : 31
3 - 2 : 51
```

Time Complexity Analysis:

The time complexity of Prim's algorithm is O(E log V)