```
void printArray (int array[], int size)
{
  for (int i=0; i<size; i++)
  printf("%d", array[i]);
  printf("\n")
}
```

output

```
Enter the number of element
5
Enter the element
3
Enter the element
8
Enter the element
2
Enter the element
1
Enter the element
4
Before sorting  8 3 2 1 4
After sorting   1 2 3 4 8
```

Page No. 21. | Date | Exp. No.

6. write a program to implement Heap sort.

```
#include <stdio.h>
int size=0;
void heapsort (int array[], int size);
void heapify (int array [], int size, int i);
void create (int array []);
void printArray (int array[], int size);
int main()
{ int array[10];
  Create (array);
  printArray (array, size);
  heapsort (array, size);
  printArray (array, size);
}
void create (int array [])
{ int n, i, newNum;
  printf ("Enter the number of elements in");
  scanf ("%d", &n);
  for (size=0; size<n; size++)
  { printf ("Enter the element in");
    scanf ("%d", &newNum);
    if (size == 0)
      array [0] = newNum;
    else
    { array[size] = newNum;
      for (int i = size/2 - 1; i >= 0; i--)
      {
```

7. write a c program to implimend BST

```
#include <stdio.h>
#include< malloc.h>
Struct node
{
    int data;
    Struct node *right child;
    Struct node *left child;
}*root =NULL, *temp;
Struct node * insert(struct node *, int);
Struct node * search(struct node *, int);
void inorder (struct node*);
void preorder (struct node*);
void postorder (struct node*);
Struct node * delete (struct node *, int);
Strud node * find-min(struct node*);
void main()
{
    root =insert (root, 1);
    insert(root, 4);
    insert (root, 5);
    Insert (root, 3);
    root =delete (root, 6),
    if ((Ssearch (root, 5)) ==NULL)
      printf (" Element not found\n");
    else
      printf (" Element found\n");
    printf ("Inorder traversal is \n");
      inorder(root);
      printf ("in Preorder trouersal is\n");
      preorder(root);
```

Page No. 24. Date Exp. No.

```
printf("\n postorder traversal is \n");
postorder(root);
}
struct node * insert(struct node * root, int x)
{
  if (root==NULL)
  {
    temp=(struct node *)malloc(sizeof(struct node));
    temp->data=x;
    temp->leftChild=temp->rightchild=NULL;
    root=temp;
  }
  else if(x>root->data)
    root->rightchild = insert(root->rightchild,x);
  else
    root->leftchild = insert(root->leftchild,x);
  return root;
}
struct node * search(struct node * root, int x)
{
  if(root==NULL || root->data==x)
    return root;
  else if(x>root->data)
    return search(root->rightchild,x);
  else
    return search(root->leftchild,x);
}
void inorder(struct node * root)
{
  if(root!=NULL)
  {
    inorder(root->leftchild);
```

Page No. 25 | Date | Exp. No.

```
        printf("%d", root -> data);
        inorder(root->rightchild);
    }
}
void preorder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d", root->data);
        preorder(root->leftchild);
        preorder(root->rightchild);
    }
}
void postorder(struct node *root)
{
    if(root!=NULL)
    {
        postorder(root->left child);
        postorder(root->right child);
        printf("%d", root->data);
    }
}
struct node * delete(struct node *root, int x)
{
    if(root==NULL)
        return root;
    if(x > root -> data)
        root -> right child = delete(root -> right child, x);
    else
        if(x < root -> data)
            root -> left child = delete(root -> left child, x);
    else
    {
        if(root->leftchild==NULL && root->rightchild=NULL
```

```
{
    free(root);
    return NULL;
}
else if(root->left child == NULL // root->right child == NULL)
    {
    struct node *temp;
    if (root->left child == NULL)
        temp = root->right child;
    else
        temp = root->left child;
    free(root);
    return temp;
}
else
{
    struct node *temp = find min(root->right child);
    root->data = temp->data;
    root->right child = delete (root->right child, temp->data);
}
}
return root;
}
struct node *find_min(struct node * root)
{
    if (root == NULL)
        return NULL;
    else if(root->left child != NULL)
        return find min(root->left child);
    return root;
}
```

```
    heapify(array, size, i);
  }
}
void heapsort(int array[], int size)
{
  int i, temp;
  for(i = size/2 - 1; i >= 0; i--)
    heapify(array, size, i)
  for(i = size-1; i >= 0; i--)
  {
    temp = array[0];
    array[0] = array[i];
    array[i] = temp;
    heapify(array, i, 0);
  }
}
void heapify(int array[], int size, int i)
{
  int temp;
  int largest = i;
  int l = 2*i + 1;
  int r = 2*i + 2;
  if(l < size && array[l] > array[largest])
    largest = l;
  if(r < size && array[r] > array[largest])
    largest = r;
  if(largest != i)
  {
    temp = array[i];
    array[i] = array[largest];
    array[largest] = temp;
  }
  heapify(array, size, largest);
}
```

output

Element found

Inorder traversal is:

1 3 5

preorder traversal is:

1 5 3

postorder traversal is:

3 5 1

Page No. 28 | Date | Exp. No.

8. write a C program to implement Breadth First Search (BFS)

```
#include <stdio.h>
#define initial 1
#define waiting 2
#define visited 3
#define MAX 5
int queue[MAX];
int front=-1, rear=-1, n;
int G[MAX][MAX], state[10];
void insert_queue(int);
int delete_queue();
void bf_traversal();
void create_graph();
void bfs(int v);
void main()
{
Create_graph();
bf_traversal();
}
void create_graph()
{
  int origin, destin, c, max_edge;
  printf("Enter number of vertices:");
  scanf("%d",&n);
  max_edge= n*(n-1);
  for(c=1; c<max_edge; c++)
  {
    printf("Enter edge %d(-1 -1 to quit ): ",c);
    scanf("%d %d",&origin,&destin);
    if(origin==-1&&destin==-1)
      break;
```

Page No. 29 | Date | Exp. No.

```
else if (origin>=n || destin>=n || origin<0 || destin<0)
  {
   printf("Invalid edge\n");
    i--;
  }
  else
   G[origin][destin] = 1;
}
}
void bf_traversal()
    { int v;
     for(v=0; v<n; v++)
       state[v] = initial;
     printf("Enter start vertex for BFS\n");
     scanf("%d", &v);
     bfs(v);
}
void bfs(int v)
 { int i;
  insertQueue(v);
  state[v] = waiting;
  while(rear!=-1 && front!=rear+1)
   {
    v = deletequeue();
    printf("%d", v);
    state[v] = visited;
    for(i=0; i<n; i++)
   { if(G[v][i]==1 && state[i]==initial)
       { insertqueue(i);
         state[i] = waiting;
```

Page No. 30 | Date | Exp. No.

```
3 3
 3
3 void insert_queue(int x)
{ if (front == -1)
   front++;
  queue[++rear] = x;
}
int delete_queue()
 { return(queue[front++]);
}
```

output

```
Enter number of vertices : 5
Enter edge 1 (-1 -1 to quit) : 0
1
Enter edge 2 (-1 -1 to quit) : 0
2
Enter edge 3 (-1 -1 to quit) : 1
3
Enter edge 4 (-1 -1 to quit) : 2
4
Enter edge 5 (-1 -1 to quit) -1
-1
Enter start vertex for BFS
0
0 1 2 3 4
```

Page No. 31 | Date | Exp. No.

9. write a C program to implement Depth first Search (DFS)

```
#include<stdio.h>
#define MAX 10
int n;
int visited[MAX];
int adj[MAX][MAX];
void create_graph();
void dfs(int);
void main()
{
int v;
create_graph();
printf("Enter the starting vertex\n");
scanf("%d",&v);
dfs(v);
}
void create_graph()
{
int origin, destin, c, max_edge;
printf("Enter number of vertices:");
scanf("%d",&n);
max_edge = n*(n-1);
for(c=1; c<=max_edge; c++)
{
printf("Enter edge %d(-1 -1 to quit):", c);
scanf("%d %d",&origin,&destin);
if(origin==-1&&destin==-1)
break;
else if(origin>=n||destin>=n||origin<0||destin<0)
{
```

```
        printf("invalid edge!\n");
        c--;
      }
      else
      adj[origin][destin] = 1;
   }
}
void dfs(int i)
{
   int j;
   printf("%d ", i);
   visited[i] = 1;
   for(j=0; j<n; j++)
   {
   if(!visited[j] && adj[i][j]==1)
      dfs(j);
   }
```

output

```
Enter number of vertices : 5
Enter edge 1 (-1 -1 to quit): 0 1
Enter edge 2 (-1 -1 to quit): 0 2
Enter edge 3 (-1 -1 to quit): 1 3
Enter edge 4 (-1 -1 to quit): 2 4
Enter edge 5 (-1 -1 to quit): 3 4
Enter edge 6 (-1 -1 to quit): -1 -1
Enter the starting vertex: 0

DFS traversal starting from vertex 0:
0 1 3 4 2
```

Page No. 33 | Date | Exp. No.

10. write a c program to implement AVL Tree, operation - Insertion, deletion and searching.

```
#include <stdio.h>
#include <stdlib.h>
Struct node
{
 int data;
 Struct node * left;
 Struct nod * right;
};
Struct node * newnode (int);
int height (struct node *);
int getBalance (struct node *);
Struct node * rightRotate (Struct node *)
Struct node * leftRotate (Struct node *);
Struct node * insert (Struct node *, int);
Void inorder (struct node *);
Void main()
{
Struct node * root = NULL;
 int n, i, x;
printf ("Enter the no. of nodes \n");
 Scanf ("%d", &n);
 for (i=0; i<n; i++)
 { Scanf ("%d", &x);
   root = insert (root, x);
 }
 printf ("Inorder traversal of the constructed AVL Tree")
   inorder (root);
}
```

```
Struct node *newnode(int key)
{
  Struct node * node = (struct node *)malloc(sizeof(struct node));
   node -> data = key;
   node -> left = NULL;
   node -> right = NULL;
   return node;
}
int height(struct node* node)
  {
   if (node == NULL)
      return 0;
   int leftHeight = height(node -> left);
   int rightHeight = height(node -> right);
   return(leftHeight > rightHeight) ? (leftHeight + 1) : (rightHeight + 1);
}
int getBalance(struct node * node)
   {
    if(node == NULL)
      return 0;
    return height(node -> left) - height(node -> right);
   }
Struct node * rightRotate(struct nod * y)
    {
      Struct node *x = y -> left;
      Struct node *T2 = x -> left;
      x -> right = y;
      y -> left = T2;
      return x;
    }
```

```
struct node * leftRotate (struct node * x)
{
  struct node * y = x->right;
  struct node * T2 = y->left;
  y->left = x;
  x->right = T2;
  return y;
}
struct node * insert (struct node * node, int key)
{
  if (node == NULL)
    return newnode(key);
  if(key<node->data)
    node->left = insert(node->left, key);
  else if (key > node->data)
    node->right = insert(node->right, key);
  else
    return node;
int balance = getBalance (node);
if(balance > 1 && key < node->left->data)
  return rightRotate(node);
if(balance < -1 && key > node->right->data)
  return leftRotate(node);
if(balance > 1 && key > node->left->data)
  {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }
```

```
if (balance < -1 && key < node->right->data)
    { node->right = rightRotate(node->right);
      return leftRotate(node);
    }
    return node;
}
void inorder(struct node *root)
    { if(root != NULL)
        { inorder(root->left);
          printf("%d", root->data);
          inorder(root->right);
    } }
```

output

```
Enter the no. of nodes
7
Enter the values for each node
30
20
40
10
25
35
45
Inorder traversal of the constructed AVL tree:
10 20 25 30 35 40 45
```