3.1 Write one single C++ program to create binary search tree for any structure and form the following through menu-driven program :

i. Insert
ii. Delete
iii. Search
iv. Display in Inorder
v. Display in postorder
vi. Display in preorder
vii. Print details with largest key
viii. Print details with smallest key
ix. Print no of nodes in the tree

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<assert.h>
struct data
{
	intnum;
	struct data *left,*right;
};
classbintree
{
	private:
	data *root;
	int count;
	public:
	bintree();
	voidcreatetree();
	data* inserttree(data*,int);
	data* deletetree(data*,int);
	data* searchtree(data*,int);
	voidinorder(data*);
	void preorder(data*);
	voidpostorder(data*);
	data* findlargest(data*);
	data* findsmallest(data*);
	inttreecount();
	voidcallmain();

};

bintree::bintree()
{
	root=NULL;
	count=0;
}
voidbintree::createtree()
{
	data* node;
	//node=new data;
	int d;
	char choice;
	do
```

```
	{
	cout<<"\nEnter a number:";
	cin>>d;
	root=inserttree(root,d);
	cout<<"\nContinue??";
	cin>>choice;

	}while(choice=='y');

}
data* bintree::inserttree(data *r,int d)
{
	if(r==NULL)
	{
		r=new data;
		count++;
		r->num=d;
		r->left=NULL;
		r->right=NULL;

	}
	else if(d<root->num)
	{
		r->left=inserttree(r->left,d);
	}
	else
	{
		r->right=inserttree(r->right,d);
	}
	return r;
}
data* bintree::deletetree(data* r,int d)
{
	data *t1,*t2;
	if(r==NULL)
	{
		return NULL;
	}
	else if(d<r->num)
	{
		r->left=deletetree(r->left,d);
	}
	else if(d>r->num)
	{
		r->right=deletetree(r->right,d);
	}
	else
	{
		if(r->left==r->right)
		{
			delete r;
			count--;
			return NULL;
		}
		if(!r->left)
		{
			t1=r->right;

			delete r;
```

```
                    count--;
                    return t1;

            }
            else if(!r->right)
            {
                    t1=r->left;

                    delete r;
                    count--;
                    return t1;
            }
            else
            {
                    t1=t2=r->right;
                    while(t1->left!=NULL)
                    {
                            t1=t1->left;
                    }
                    t1->left=r->left;
                    delete r;
                    count--;
                    return t2;
            }

    }
return r;

}
intbintree::treecount()
{
    return count;
}
data* bintree::searchtree(data* r,int d)
{
    if(!r)
    {
            return NULL;
    }
    if(r->num>d)
    {
            returnsearchtree(r->left,d);
    }
    else if(r->num<d)
    {
            returnsearchtree(r->right,d);
    }
    else
    {
            return r;
    }

}
voidbintree::inorder(data* r)
{
    if(r==NULL)
      {
            return;
      }
```

```
		inorder(r->left);
		cout<<"\t"<<r->num;
		inorder(r->right);

}
voidbintree::postorder(data* r)
{
		if(r==NULL)
		{
			return;
		}

		postorder(r->left);
		postorder(r->right);
		cout<<"\t"<<r->num;

}
voidbintree::preorder(data* r)
{
		if(r==NULL)
		{
			return;
		}
		cout<<"\t"<<r->num;
		preorder(r->left);
		preorder(r->right);

}
data* bintree::findlargest(data *r)
{
		while(r->right!=NULL)
		{
			r=r->right;
		}
		return r;
}
data* bintree::findsmallest(data *r)
{
		while(r->left!=NULL)
		{
			r=r->left;
		}
		return r;
}

voidbintree::callmain()
{
		intch=0,user;
		while(ch!=10)
		{

		cout<<"\n1.Insert\n2.delete\n3.search\n4.inorder\n5.preorder\n6.postorder\n7.largest\n8.sm
allest\n9.count\n10.exit\nchoice:"; ;
			cin>>ch;

			switch(ch)
			{
				case 1:
```

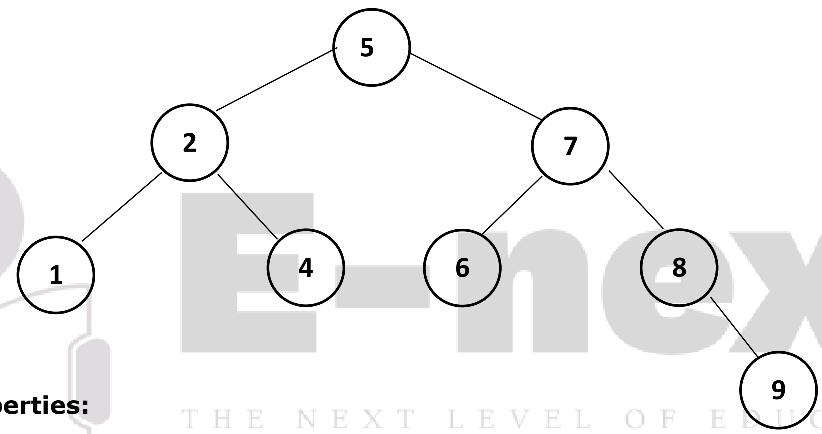
```
{
        cout<<"\nEnter the number to be inserted:";
        cin>>user;
        root=inserttree(root,user);
        if(root==NULL)
        {
                cout<<"\nNOt inserted";
        }


        break;
}
case 2:
{
        cout<<"\nEnter the number to be deleted:";
        cin>>user;
        root=deletetree(root,user);
        if(!root)
        {
                cout<<"\nnot found";
        }
        else
        {
                cout<<"\nDeleted!!";
        }
        break;
}
case 3:
{     data *n;
        cout<<"\nEnter the number to be searched:";
        cin>>user;
        n=searchtree(root,user);
        if(n==NULL)
        {
                cout<<"\nNot found";
        }
        else
        {
                cout<<"\nFound!!";
        }

        break;
}
case 4:
{
        cout<<"\nInorder:";
        inorder(root);
        break;
}
case 5:
{
        cout<<"\npreorder:";
        preorder(root);
        break;
}
case 6:
{
        cout<<"\nPost order:";
        postorder(root);
```

```
                break;
            }
            case 7:
            {
                data* node;
                node=findlargest(root);
                cout<<"\nLargest:" <<node->num;
                break;
            }
            case 8:
            {

                data* node;
                node=findsmallest(root);
                cout<<"\nsmallest:" <<node->num;
                break;
            }
            case 9:
            {
                cout<<"\nCount:"<<treecount();
                break;
            }

            case 10:
            {
                break;
            }
            default:
            {
                cout<<"\nINvalid Choice";
                break;

            }

        }
    }
}
void main()
{
clrscr();
bintree b1;
b1.createtree();
b1.callmain();
getch();
}
```

3.2 Write the properties of Binary Search Tree (BST) and depict it with a simple diagram and write algorithms for the following BST operations :

i. Insert
ii. Delete
iii. Search
iv. Inorder Traversal
v. Postorder Traversal
vi. Preorder Traversal
vii. Find Largest Key
viii. Find Smallest Key
ix. Count Nodes in the tree

Example:



Basic Properties:

1. The left sub tree of a node contains only nodes with keys less than the node's key.
2. The right sub tree of a node contains only nodes with keys greater than the node's key.
3. Both the left and right sub trees must also be binary search trees.
4. There must be no duplicate nodes.

AlgorithmdeleteBST (ref root <pointer>, valdltKey<key>)

This program deletes a node from a BST

Pre root is a pointer to tree containing data to be deleted

Dltkey is key of node to be deleted

Post node deleted & memory recycled

If dltkey not found, root unchanges

Return True if not deleted, falsr if not found

```
1 if(root null)
    2      return false
2   if(dltkey< root->data.key)
    1      return   deleteBST(root->left, dltkey)
3   else if(dltkey> root->data.key)
    1      return   deleteBST(root->right,dltkey)
4   else
           Delete node found – Test for leaf node
    1      if(root->left==root->right)
           1      recycle (root)
           2      return NULL
    1      if(root -> left NULL)
           1      dltptr=root->right
           2      recycle(root)
           3      return (dltptr)
    3      if(root ->right NULL)
           1      dltptr=root->left
           2      recycle root
           3      return (dltptr)
    4      else
           1      dltptr1=dltptr2=root->right
           2      while(dltptr1->left not NULL)
```

1 dltptr1=dltptr1->left

3 dltptr1->left=root->left

4 recycle root

5 return dltptr2

5 return root

AlgorithminsertBST(ref root <pointer>, val new <pointer>)

Insert node containing new data into BST using recursion

Pre root is address of first node in a BST

New is address of node containing data to be inserted

Post new node inserted into the tree

1 if (root is NULL)

1 root = new
2 root -> left = NULL
3 root -> right = NULL

2 else

Location null subtree for insertion

1 If (new -> key < root -> key)

1 insertBST(root -> left, new)

2 else

1 insertBST(root -> right, new)

3 return

endinsertBST

AlgorithmsearchBST (val root<pointer>, val argument <key>)
Search a binary search tree for a given value.
Pre root is the root to a binary tree or subtree
argument is the key value requested
Return the node address if the value is found
null if the node is not in the tree

```
1   if  (root is null)
        1.1 returns null
2   if  (argument < root -> key)
        2.1 return searchBST  (root ->left , argument)
3   else if  (argument > root ->key )
            3.1  return searchBST (root ->right , argument)
4   else
        4.1 return root
endsearchBST
```

AlgorithmpreorderBST(ref root <pointer>)
This algorithm displays the inorder of a BST
Pre root is a pointer to a non-empty BST
Return null if the node is not in the tree

```
1    if  (root is null)
   1  return  null
2   print(root->number)
3 preorderBST(root->left)
4 preorder(root->right)
endpreorderBST
```

AlgorithmpostorderBST(ref root <pointer>)
This algorithm displays the postorder of a BST
Pre root is a pointer to a non-empty BST
Return null if the node is not in the tree

```
1  if  (root is null)
     1    return  null
2  postorderBST(root->left)
3   postorder(root->right)
4  print(root->number)

endpostorderBST
```

AlgorithmfindLargestBST(ref root <pointer>)
This algorithm finds the largest node in a BST
Pre root is a pointer to a non-empty BST
Return address of largest node

1 if (root -> right null)
    1 return (root)
2 return findLargestBST (root-> right)
endfindLargestBST

AlgorithmfindSmallestBST(ref root <pointer>)
This algorithm finds the smallest node in a BST
Pre root is a pointer to a non-empty BST
Return address of smallest node

1 if (root -> left null)
    1 return (root)
2 return findSmallestBST(root-> left)
endfindSmallestBST

AlgorithminorderBST(ref root <pointer>)
This algorithm displays the inorder of a BST
Pre root is a pointer to a non-empty BST
Return null if the node is not in the tree

1 if (root is null)
    1 return null
2 inorderBST(root->left)
3 print(root->number)
4 inorder(root->right)
endinorderBST

AlgorithmcountNodesBST(ref root <pointer>)
This algorithm displays the total number of nodes in a BST
Pre root is a pointer to a non-empty BST
Return null if the node is not in the tree

1 if (root is null)
    1 return null
2 countNodesBST(root->left)
3 countNodesBST(root->right)
4 count ++
endcountNodesBST