Instructions: Please read carefully

- Please rename this file as only your ID number **(e.g. 18-*****-1.doc or 18-*****-1.pdf).**
- Submit the file before **11:59pm on 11/12/2020** in the Portal Lab Performance section labeled **Lab task 10. If you cannot complete the full task, do not worry. Just upload what you have completed.**

1. **Write a C++ code to implement Binary Search Tree operations (insertion, traversal and searching)**
   Do the following to write program for a BST:
   1. To construct a binary search tree of integers (**insert** one by one).
   2. To **traverse** the tree using all the methods i.e., in order, preorder and post order.
   3. To **search** an element on the BST.
   4. There are three cases when you delete a node.
      - Case 1: Node with zero child (Leaf node)
      - Case 2: Node with one child
      - Case 3: Node with both children

      Implement the logic of 3 cases one by one.

Hint: Your program should ask the user to input the choice what operation the user wants to perform.

1. Insert
2. Travers
3. Search
4. Delete

```
Class Node{
      Data
      lptr , rptr
      Node(){
      Everything null
      }
      Node(int a){
      Data =a
      Everything else null
       }
 }
```

```
Class BST{
    root
    BST(){
     Root = NULL
}
insert(x){
  if(Root == null){
    Node * nptr = new Node();
    Nptr->Data = x;
    Root = nptr;
……}// for inserting root node
  else {………..}// for inserting rest of the nodes
}
Preorder(Node * tptr){
    If(tptr !=Null){
        Print(tptr->Data)
        Preorder(tptr->lptr)
        Preorder(tptr->rptr)

}
}
}
```

```
main(){
BST b
b.insert(10)
b.insert(20)
b.Preorder(b.root)
}
```

Your code here:

```
#include<iostream>
#define SPACE 10

using namespace std;

class TreeNode

{
 public:
  int value;
 TreeNode * left;
 TreeNode * right;

 TreeNode()
 {
  value = 0;
  left = NULL;
  right = NULL;
 }

 TreeNode(int v)
 {
  value = v;
  left = NULL;
  right = NULL;
 }
};



class BST
{
 public:
  TreeNode * root;

 BST()
 {
  root = NULL;
 }

 bool isTreeEmpty()
 {
  if (root == NULL)
    {
      return true;
  }

  else
    {
      return false;
  }
 }
```

```
void insertNode(TreeNode * new_node)
{
 if (root == NULL)
 {
  root = new_node;
  cout << "Value Inserted as root node!" << endl;
 }

 else
   {
     TreeNode * temp = root;

 while (temp != NULL)
   {
     if (new_node -> value == temp -> value)
     {
       cout << "Value Already exist," <<"Insert another value!" << endl;
    return;
   }

   else if ((new_node -> value < temp -> value) && (temp -> left == NULL))
     {
       temp -> left = new_node;
   cout << "Value Inserted to the left!" << endl;
    break;
   }

   else if (new_node -> value < temp -> value)
     {
       temp = temp -> left;
   }
   else if ((new_node -> value > temp -> value) && (temp -> right == NULL))
     {
    temp -> right = new_node;
    cout << "Value Inserted to the right!" << endl;
    break;
   }
   else
     {
    temp = temp -> right;
   }
  }
 }
}



void print2D(TreeNode * r, int space)
{
 if (r == NULL) // Base case  1
  return;
 space += SPACE; // Increase distance between levels  2
 print2D(r -> right, space); // Process right child first 3
 cout << endl;
 for (int i = SPACE; i < space; i++)
  cout << " ";
 cout << r -> value << "\n";
 print2D(r -> left, space);
}

void printPreorder(TreeNode * r) //(current node, Left, Right)
{
 if (r == NULL)
  return;
 cout << r -> value << " ";

 printPreorder(r -> left);

 printPreorder(r -> right);
}

void printInorder(TreeNode * r) //  (Left, current node, Right)
{
 if (r == NULL)
  return;

 printInorder(r -> left);

 cout << r -> value << " ";

 printInorder(r -> right);
}
void printPostorder(TreeNode * r) //(Left, Right, Root)
{
 if (r == NULL)
  return;
 // first recur on left subtree
 printPostorder(r -> left);
 // then recur on right subtree
 printPostorder(r -> right);
 // now deal with the node
 cout << r -> value << " ";
}



TreeNode * iterativeSearch(int v)
{
 if (root == NULL)
   {
  return root;
 }
 else
   {
  TreeNode * temp = root;
  while (temp != NULL)
  {
   if (v == temp -> value)
   {
    return temp;
   }
 else if (v < temp -> value)
   {
    temp = temp -> left;
   }
 else
   {
    temp = temp -> right;
   }
  }
  return NULL;
 }
}



TreeNode * recursiveSearch(TreeNode * r, int val)
{
 if (r == NULL || r -> value == val)
  return r;

 else if (val < r -> value)
  return recursiveSearch(r -> left, val);

 else
  return recursiveSearch(r -> right, val);
}

int height(TreeNode * r)
{
 if (r == NULL)
  return -1;

 else
   {

  int lheight = height(r -> left);
  int rheight = height(r -> right);


  if (lheight > rheight)
   return (lheight + 1);
  else return (rheight + 1);
 }
}
```

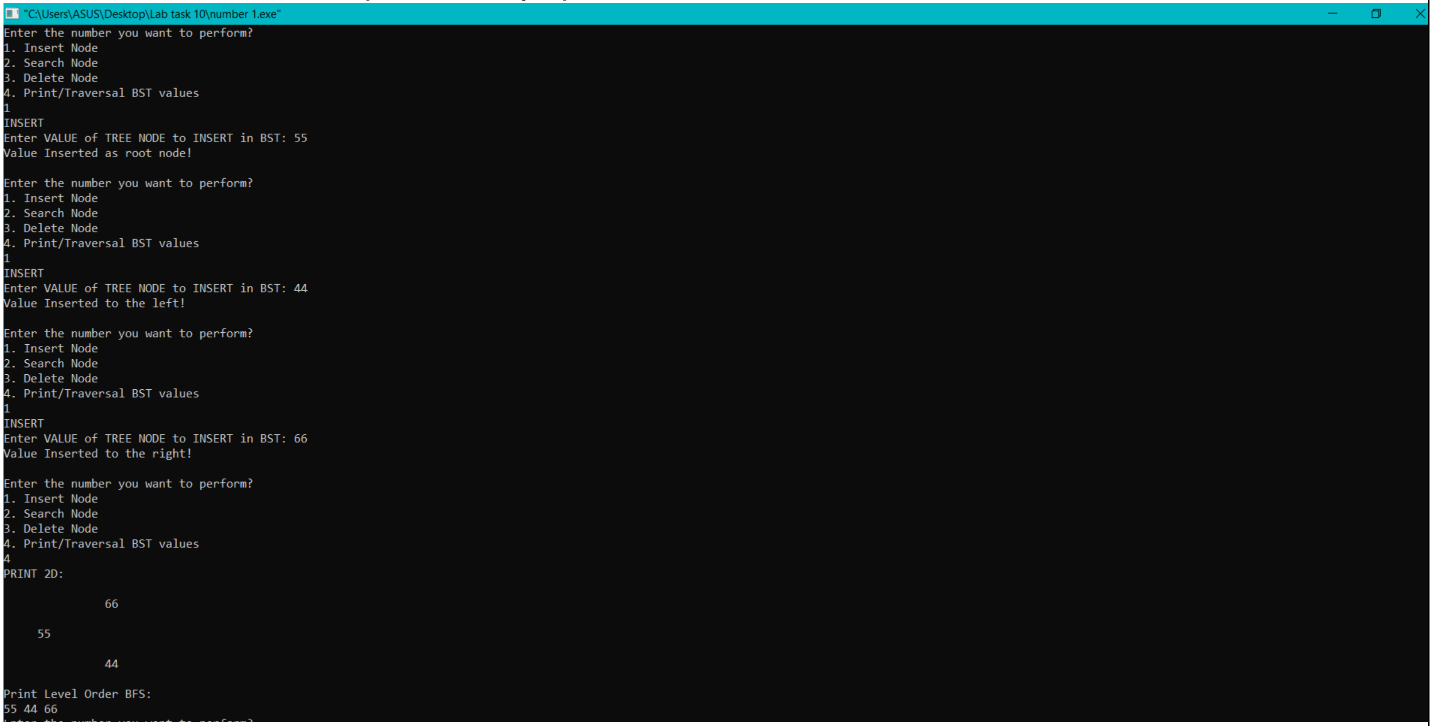
```
void printGivenLevel(TreeNode * r, int level)
{
 if (r == NULL)
  return;
 else if (level == 0)
  cout << r -> value << " ";
 else // level > 0
 {
  printGivenLevel(r -> left, level - 1);
  printGivenLevel(r -> right, level - 1);
 }
}

void printLevelOrderBFS(TreeNode * r)
 {
 int h = height(r);
 for (int i = 0; i <= h; i++)
  printGivenLevel(r, i);
}

TreeNode * minValueNode(TreeNode * node)
 {
 TreeNode * current = node;

 while (current -> left != NULL) {
  current = current -> left;
 }
 return current;
}

TreeNode * deleteNode(TreeNode * r, int v)
 {

 if (r == NULL) {
  return NULL;
 }
 // If the key to be deleted is smaller than the root's key,
 // then it lies in left subtree
 else if (v < r -> value)
   {
  r -> left = deleteNode(r -> left, v);
 }
 // If the key to be deleted is greater than the root's key,
 // then it lies in right subtree
 else if (v > r -> value)
 {
  r -> right = deleteNode(r -> right, v);
 }
 // if key is same as root's key, then This is the node to be deleted
 else
 {
  // node with only one child or no child
  if (r -> left == NULL)
```

```
    {
    TreeNode * temp = r -> right;
    delete r;
    return temp;
   }
  else if (r -> right == NULL)
    {
    TreeNode * temp = r -> left;
    delete r;
    return temp;
   }
  else
    {
    // node with two children: Get the inorder successor (smallest
    // in the right subtree)
    TreeNode * temp = minValueNode(r -> right);
    // Copy the inorder successor's content to this node
    r -> value = temp -> value;
    // Delete the inorder successor
    r -> right = deleteNode(r -> right, temp -> value);
    //deleteNode(r->right, temp->value);
   }
  }
  return r;
 }

};



int main()
{
 BST obj;
 int option, val;

 do {
  cout << "Enter the number you want to perform? "<<endl;
  cout << "1. Insert Node" << endl;
  cout << "2. Search Node" << endl;
  cout << "3. Delete Node" << endl;
  cout << "4. Print/Traversal BST values" << endl;

  cin >> option;
  //Node n1;
  TreeNode * new_node = new TreeNode();

  switch (option)
  {
  case 0:
   break;

  case 1:
   cout << "INSERT" << endl;
```

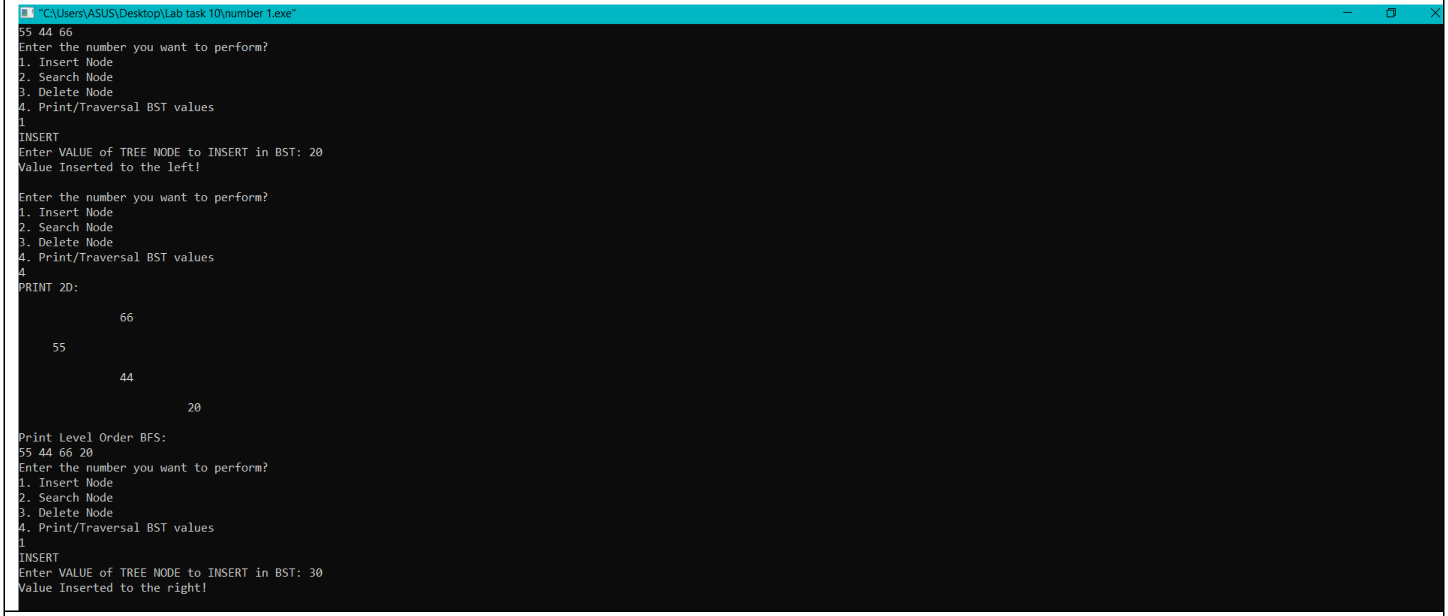
```
      cout << "Enter VALUE of TREE NODE to INSERT in BST: ";
      cin >> val;
      new_node -> value = val;
      obj.insertNode(new_node);
      cout << endl;
      break;

    case 2:
      cout << "SEARCH" << endl;
      cout << "Enter VALUE of TREE NODE to SEARCH in BST: ";
      cin >> val;
      //new_node = obj.iterativeSearch(val);
      new_node = obj.recursiveSearch(obj.root, val);
      if (new_node != NULL)
        {
        cout << "Value found" << endl;
      }
      else
        {
        cout << "Value NOT found" << endl;
      }
      break;

    case 3:
      cout << "DELETE" << endl;
      cout << "Enter VALUE of TREE NODE to DELETE in BST: ";
      cin >> val;
      new_node = obj.iterativeSearch(val);
      if (new_node != NULL)
        {
        obj.deleteNode(obj.root, val);
        cout << "Value Deleted" << endl;
      }
      else
       {
        cout << "Value NOT found" << endl;
      }
      break;

    case 4:
      cout << "PRINT 2D: " << endl;
      obj.print2D(obj.root, 5);
      cout << endl;
      cout << "Print Level Order BFS: \n";
      obj.printLevelOrderBFS(obj.root);
      cout << endl;

      break;

    default:
      cout << "Enter Proper Option number " << endl;
    }
```

```
  } while (option != 0);

  return 0;
}
```

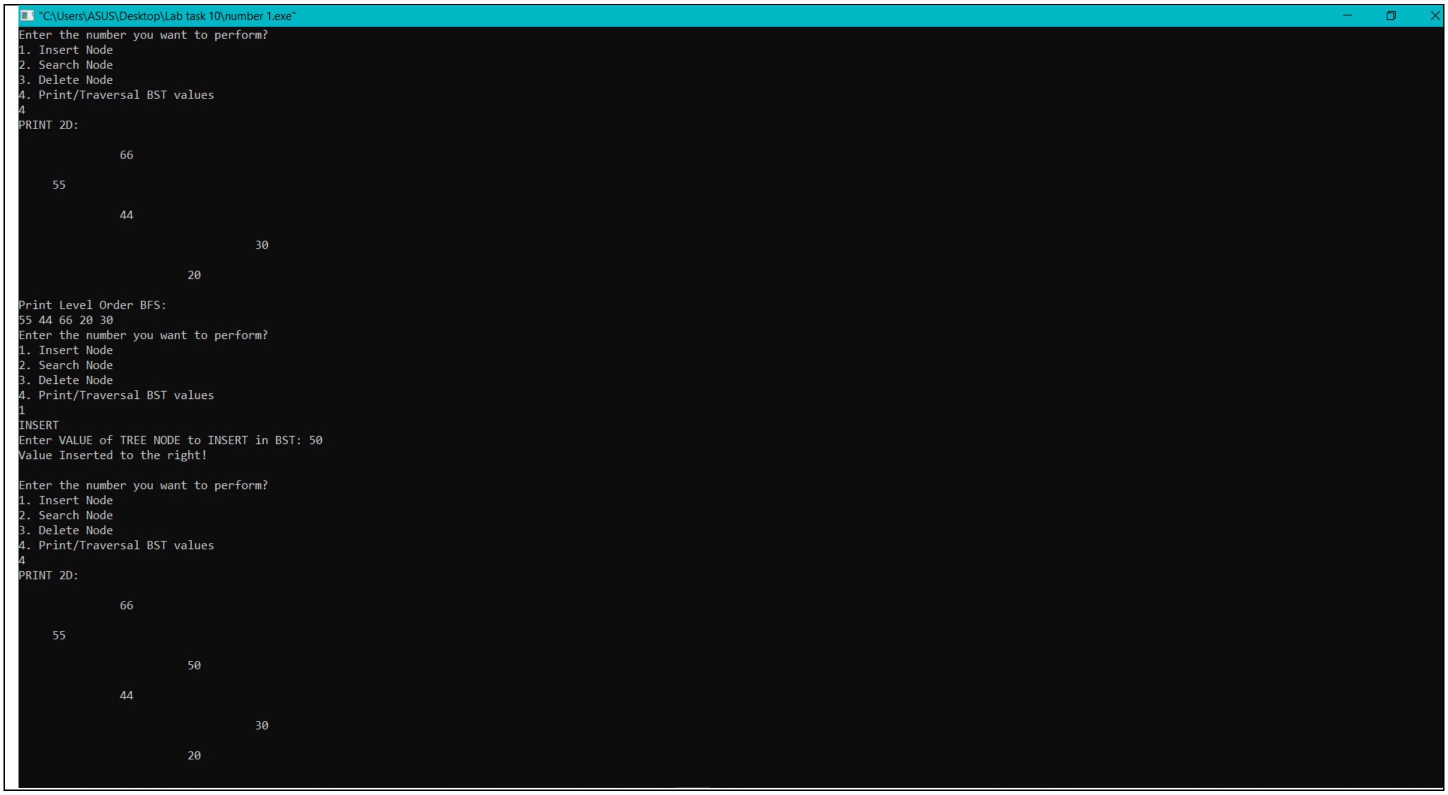
Your whole Screenshot here: (Console Output):



```
Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
1
INSERT
Enter VALUE of TREE NODE to INSERT in BST: 55
Value Inserted as root node!

Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
1
INSERT
Enter VALUE of TREE NODE to INSERT in BST: 44
Value Inserted to the left!

Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
1
INSERT
Enter VALUE of TREE NODE to INSERT in BST: 66
Value Inserted to the right!

Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
4
PRINT 2D:

               66

     55

               44

Print Level Order BFS:
55 44 66
```



```
55 44 66
Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
1
INSERT
Enter VALUE of TREE NODE to INSERT in BST: 20
Value Inserted to the left!

Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
4
PRINT 2D:

               66

     55

               44

                         20

Print Level Order BFS:
55 44 66 20
Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
1
INSERT
Enter VALUE of TREE NODE to INSERT in BST: 30
Value Inserted to the right!
```



```
Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
4
PRINT 2D:

               66

     55

               44

                                   30

                         20

Print Level Order BFS:
55 44 66 20 30
Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
1
INSERT
Enter VALUE of TREE NODE to INSERT in BST: 50
Value Inserted to the right!

Enter the number you want to perform?
1. Insert Node
2. Search Node
3. Delete Node
4. Print/Traversal BST values
4
PRINT 2D:

               66

     55

                         50

               44

                                   30

                         20
```