The objective of this lab is to:

- Understand and implement binary tree.
- Understand and implement binary search tree.

ALERT!

1. This is an individual lab, you are strictly **NOT** allowed to discuss your solution with fellow colleagues, even not allowed asking how is he/she is doing, it may result in negative marking. You can **ONLY** discuss with your TAs or with me.
2. Pay attention to **GOOD coding conventions.**
3. Anyone caught in act of plagiarism would be awarded an "F" grade in this Lab.

Task 01: [40 Marks]

Implement the binary tree class using array based implementation as we discussed in class.

Recalling array based implementation:

A binary tree can be implemented using arrays such that each index of array holds a tree node. In array based implementation there may be following questions to be answered:

- What is the initial size of array as the tree starts growing from root to leaf node by node?
  You can take height of the tree as input and can find the maximum possible nodes of given height using $2^h - 1$. So you can create the array of maximum possible nodes of a given height and store the tree nodes in level order.
- How to get children of a particular node?

On examining a tree and its array you may observe that all left children are on odd indices and all right children are on even indices. Thus, for each node at index i, the left child is at index $(2 \times i) + 1$, and the right child is at index $(2 \times i) + 2$. The parent of node i is at $\lfloor (i-1)/2 \rfloor$.

You may use following class definition and implement the functions given in the public part of the Tree. You may write other utility functions if you think required.

```
template<class T>
class BinaryTree
{
    private:
        int height;      // represent the maximum possible height of the tree.
        T* data;         // stores the tree nodes.
        bool* status;    // this array is used to find that whether there is a
                         // node on a particular index or not. If there is a valid
                         // node exitsts on a particular index the status array holds
                         // 'true' on corresponding index.
    public:
        BinaryTree(int h); //initializes all the data members
        ~BinaryTree();
        void setRoot(T val);  // set val at data[0] as a root of tree and set status[0]
= true;
        T getRoot();      // returns the root of tree if exists.
```


Madiha Khalid

```
        void setLeftChild(T parent, T val);     // sets tge left child of given Parent.
        void setRightChild(T parent, T val);
        int getParent(int node);   // returns the index of parent node.
        int getLeftChild (int par);
        int getRightChild (int par);
        void remove(T node);    // remove the given node and all its descendants.

        void preOrder();    // display tree nodes using preOrder Traversal.
        void postOrder();   // display tree nodes using postOrder Traversal.
        void inOrder();    // display tree nodes using inOrder Traversal.
        void levelOrder();  // display tree nodes using levelOrder Traversal.

        void displayAncestors(T node);
        void displayDescendents(T node);
        bool isPerfectTree();   // returns true if the tree is perfect, otherwise
false.
        bool iscompleteTree();  // returns true if the tree is perfect, otherwise
false.
};
```

Task 02: [54 marks]

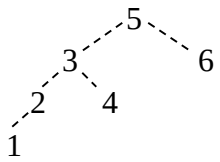
Implement a Binary Search Tree class using linked implementation.

```
// forward declaration of template class BTree
template<class T>
class BST;

template<class T>
class BSTNode
{
   friend BST<T>;

   T  data;
   BSTNode<T>*  left;
   BSTNode<T>*  right;
     // Methods…
};

template<class T>
class BST
{
    BSTNode<T>* root;
     // Methods…
};
```

Implement following functions for BST class:

1. **Constructor, destructor, Copy-constructor.** **[2+2+2]**
2. **void setRoot (T value);** **[1]**
3. **void insert (T value);** **[3]**
4. **BSTNode<T>* getLeftChild (BSTNode<T>* node);** **[1]**
5. **BSTNode<T>* getRightChild (BSTNode<T>* node);** **[1]**
6. **DNode<T>* search (T val)** **[3]**
7. **void deleteNode (BSTNode<T>* node)** **[4]**
8. **void printNodes (BSTNode<T>* root);** // Use any traversal to print the tree Nodes **[2]**
9. **boolean isBST (BSTNode<T>* root);** **[4]**
   Takes an Object of tree as a parameter and returns true if it is a BST, false otherwise.

10. **void printTree ( );** **[4]**
    This function should print tree in bellow form



11. **boolean isEqual (BSTNode<T>* r1, BSTNote<T>* r2);** **[4]**
    Takes roots of two trees and as input parameter and returns true if they are equal.

12. **boolean isInternalNode (BSTNote<T>* node);** **[2]**
    Returns true if given node is an internal node. Where, internal Node is one which has degree greater than zero.

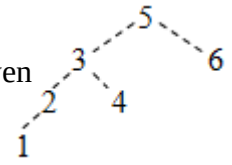
13. **boolean isExternalNode (BSTNote<T>* node);** **[2]**
    Returns true if given node is an external node. External Node is one which has degree equal to zero.

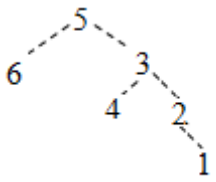
14. **int getHeight ( );** **[2]**
    Returns the height of tree.
    If u want to write recursive function for height calculation then call your recursive function in this function and make it helper/driver of recursive function.

15. **void displayDescedents (T val);** **[2]**
    Display decedents of the node containing given value.

16. **void displayAncestors (T val);** **[2]**
    Display Ancestors of the node containing given value.

17. **BST<T> getMirrorImage ( );** **[4]**
    Returns the mirror image of *this tree.

    For Example, given 

    you should return 

18. **int getNodeCount (BST<T>* tree);** **[3]**
    Return the number of nodes in a given tree.

19. **T findMin ();** **[2]**
    Returns the min value stored in your BST.

20. **T findMax ();** **[2]**
    Returns the max value stored in your BST.