Objective:

- Implementing the Representation of Binary Trees using Array structure and Link structure.
- Applying/adding/implementing some easy to hard practice problems which will not only strengthen your tree related concept but will also enhance your recursive thinking.

Task-1: ADT for Array Based implementation of Binary Tree. Array Based binary representation already discussed in class.

```
template<typename T>
class ArrayBasedBinaryTree
{
    int height; //represents the maximum possible (capacity = 2^height –1) height of tree.
    int maxNodes; //stores 2^height –1, so doesnt need to be recalculate
    T * data;    // stores the nodes of the trees
    bool * nodeStatus;
            //It is used to find that whether there is a node on a particular index of
            //array pointed by 'data'.
            // Note: we are not using the approach of using a sentinel value in 'data'
            //array because in a template-based data T could be of any type.
public:
    ArrayBasedBinaryTree(int h=4);
        // initializes the nodeStatus array with 0 and creates data array of size 2^h –1

    void setRoot(T val);
        //stores val at data[0] as root of tree and also sets the nodeStatus[0] =true.
    T getRoot(); //returns the root of tree if exists.

    void setLeftChild(T parent, T child);
    void setRightChild(T parent, T child);
    T getParent(T node);
    bool searchNode(T node);
    void remove(T node);    //removes the given node and all its descendants from tree.

    void displayAncestors(T node);    //display ancestors of the given node
    void displayDescendents(T node);     //display descendants of the given node
    int heightOfTree(); //returns the height (actual height) of tree.
    void preOrder();    // do the VLR of tree.
    void postOrder();    // do the LRV of tree.
    void inOrder();    // do the LVR of tree.
    void levelOrder();    // do the level order traversal of tree.
    void displayLevel(int levelNo);    // display the nodes on a particular level number.
    int findLevelOfNode(T node);    // returns the level/depth of given node.
};
```

Task-2: ADT for Link Based implementation of Binary Tree. Some functions already implemented in class.

```
template<typename T>
struct BTNode
{
    T info;
    BTNode<T> * left;
    BTNode<T> * right;
    BTNode()
    {left=right=nullptr;}
    BTNode(T val)
    {info=val; left=right=nullptr;}
};

template<typename T>
class LinkBinaryTree
{
    BTNode<T> * root;
public:
```

```
    LinkBinaryTree();
    void setRoot(T val);
    T getRoot();
    bool searchNode(T key);
    void setLeftChild(T parent, T child);
    void setRightChild(T parent, T child);
    T getParent(T node);
    void remove(T node);
    void displayAncestors(T node);
    void displayDescendents(T node);
    int heightOfTree();
    void preOrder();
    void postOrder();
    void inOrder();
    void levelOrder();
    void displayLevel(int levelNo);
    int findLevelOfNode(T node);
    void displayParenthesizedView();//see below the explanation.
    void displayExplorerView();//see below the explanation.
    ~LinkBinaryTree();
};
```

The ***parenthesize view*** of the tree given above will be

```
A ( B ( D ( , H ) , E ( I ( K , ) , J ) ) , C ( , F ) )
```

Note: this parenthesize view will help us store this hierarchal structure in file.

The ***explorer view*** of the tree given above will be

```
A
    B
        D
            H
        E
            I
                K
            J
    C
        F
```

Note: the explorer/expanded view gives us basic idea about how a directory strutucre in window explorer is display in expanded form.

When you are done with the above ADTs, add the following functions in your Link Binary Tree ADT.

1. ***int LinkBinaryTree<T>::countNodes( )***
   Purpose: This function returns the number of nodes in the *this object.

2. ***T LinkBinaryTree<T>::minValue( )***
   Purpose: This function returns the minimum value from the tree.

3. ***int LinkBinaryTree<T>::countLeafNodes( )***
   Purpose: This function returns the number of leaf nodes in the *this object.

4. ***void LinkBinaryTree<T>::nonRecPreOrder( )***
   Purpose: This function does the non-recursive PreOrder traversal of the tree.

5. ***void LinkBinaryTree<T>::nonRecPostOrder( )***
   Purpose: This function does the non-recursive PostOrder traversal of the tree.

6. ***void LinkBinaryTree<T>::nonRecInOrder( )***
   Purpose: This function does the non-recursive InOrder traversal of the tree.

7. ***int LinkBinaryTree<T>::isComplete( )***
   Purpose: This function determines whether the tree is complete or not.

8. ***int LinkBinaryTree<T>::isFull( )***
   Purpose: This function determines whether the tree is full or not.

9. ***int LinkBinaryTree<T>::findBalanceFactor(T)***
   Purpose: This function finds the balance factor of a given node 'T'.
   Balance Factor of a particular node is defined as the difference between the height of its left and right sub-tree.

10. ***int LinkBinaryTree<T>::isIsomorphic(LinkBinaryTree<T>&)***
    Purpose: This function determines the equality of two rooted trees.

11. ***int LinkBinaryTree<T>::isBTIsomorphic(LinkBinaryTree<T>&)***
    Purpose: This function determines the equality of two binary trees.

12. ***LinkBinaryTree<T> LinkBinaryTree<T>::createClone( )***
    Purpose: This function return the deep clone of *this.

13. ***LinkBinaryTree<T> LinkBinaryTree<T>::createMirrorClone( )***
    Purpose: This function return the mirror image of *this.
    Note: *this object must not be changed

14. ***int LinkBinaryTree<T>::isMirror(LinkBinaryTree<T>&)***
    Purpose: This function returns true if the *this and received object are mirror image of each other other false.

15. ***void LinkBinaryTree<T>::displayLongestPath( )***
    Purpose: This function displays the longest path from root to leaf.
    For Example: In the following tree

The longest path is
A---B---E---I---K

16. ***void LinkBinaryTree<T>::displayAllPaths( )***
    Purpose: This function displays the all possible paths from root to leaves. Obviously the number of paths will be equal to number of leaves (external node) in the tree.
    For Example: In the tree given above; possible paths are as follows:

Path: A---B---D---H
Path: A---B---E---I---K
Path: A---B---E---J
Path: A---C---F

17. ***void LinkBinaryTree<T>::displayAllPathsLength( )***
    Purpose: This function displays the all possible paths from root to leaves. Obviously the number of paths will be equal to number of leaves (external node) in the tree.
    The only difference in the previous and this function is the output which also shows length of each path.

Path: A---B---D---H = 4
Path: A---B---E---I---K = 5

Path: A---B---E---J = 4
Path: A---C---F = 3

18. T LinkBinaryTree<T>::lowestCommonAncestor(T a,T b)

Purpose: This function returns the lowest common ancestor of nodes 'a' and 'b'.
The lowest common ancestor is defined between two nodes v and w as the lowest node in tree that has both v and w as descendants (where we allow a node to be a descendant of itself).
For Example: For the tree given above
The lowest common ancestor of nodes 'H' and 'K' is 'B'.
The lowest common ancestor of nodes 'I' and 'J' is 'E'.
The lowest common ancestor of nodes 'B' and 'K' is 'B'.

19. int LinkBinaryTree<T>::findDistance(T a,T b)

Purpose: This function returns the distance between the nodes 'a' and 'b'.
Let d(a) denote the depth of 'a' in tree. The distance between two nodes a and b in tree is d(a) + d(b) – 2∗d(x), where 'x' is the node which is (lowest common ancestor) of 'a' and 'b'.
For Example: For the tree given above
The distance between nodes 'H' and 'K' is 5.
The distance between nodes 'B' and 'D' is 1.

20. int LinkBinaryTree<T>::findDiameter()

Purpose: This function returns the diameter of binary tree.
The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.
For Example:

The diameter of the tree is 3.	The diameter of the tree is 7.
 Longest Paths: C–A–B–D and C–A–B–E	 Longest Path: U–S–H–D–B–E–I–T

21. Attempt the competition problem: **http://uva.onlinejudge.org/external/1/112.html**