Task 1: Binary Tree

You have to implement the Binary Tree class using link list data structure.

Suppose, we have the following Node and Binary Tree class.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def set_value(self, value):
        self.value = value

    def set_left(self, left):
        self.left = left

    def set_right(self, right):
        self.right = right

    def get_value(self):
        return self.value

    def get_left(self):
        return self.left

    def get_right(self):
        return self.right
```

```
class BinaryTree:
    def __init__(self):
        self.root = None
```

Implement the following functions in the BinaryTree Class

1. `def insert_element_root(self, element):`
2. `def insert_left_child(self, parent, child):`
3. `def insert_right_child(self, parent, child):`
4. `def delete_element(self, element):`
5. `def display_pre_order(self, node): ): # This function does the recursive PreOrder traversal of the tree.`

6. `def display_in_order(self, node): #This function does the recursive InOrder traversal of the`
7. `def display_post_order(self, node): #This function does the recursive PostOrder traversal of the`
8. `def count_nodes(self):  #This function returns the number of nodes in the tree.`

9. `def min_value(self, node): #This function returns the minimum value from the tree.`
10. `def count_leaf_nodes(self, node): #This function returns the number of leaf nodes in the tree.`
11. `def non_rec_pre_order(self): # This function does the non-recursive PreOrder traversal of the tree.`
12. `def non_rec_post_order(self): #This function does the non-recursive PostOrder traversal of the tree.`
13. `def non_rec_in_order(self):  #This function does the non-recursive InOrder traversal of the tree.`
14. `def find_balance_factor(self, node): #This function finds the balance factor of a given node.`
    `#Balance Factor of a particular node is defined as the difference between the height of its left and right sub tree.`
15. `def display_ancestors(self, node_data): # display ancestors of the given node`
16. `def display_descendants(self, node_data): #display descendants of the given node`
17. `def height_of_tree(self): #Calculate the height of the tree`

Task 2: BST

You have to implement the Binary Search Tree class using link list data structure.

Suppose, we have the following Node and Binary Tree class.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def set_value(self, value):
        self.value = value

    def set_left(self, left):
        self.left = left

    def set_right(self, right):
        self.right = right

    def get_value(self):
        return self.value

    def get_left(self):
        return self.left

    def get_right(self):
        return self.right
```

```python
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert_element(self, value):


    def delete_element(self, value):

    def display_pre_order(self):
    def display_in_order(self):
    def display_post_order(self):
    def total_elements(self):
```

Implement the following functions in the Binary Search Tree Class

1. def insert_element(self, value):
2. def delete_element(self, value):
3. def display_pre_order(self):
4. def display_in_order(self):
5. def display_post_order(self):
6. def total_elements(self):

Task 3:

Today, we are going to balance a tree with off-line data. The idea is to keep adding data in Binary Search Tree without any balance checking. You may jump to bullet points instead of reading this paragraph further. Occasionally, **count number of nodes** on left and right of root and take difference. If the **difference** is more than **one** then balance the tree. There is a different way of balancing i.e. **take an array** according to the **count** (already taken, while finding difference) plus one (keep 0 index empty). While doing **in-order store data in array**, in sorted order (Do not do sorting after putting data in array, rather get sorted data by doing in-order traversal and managing index). After getting sorted data, **remove elements** from BST, assign **NULL to root**. Add elements in BST using Binary Search approach that is add element from middle of the array, next add element from middle of left half, next add element from middle of first quarter and so on. You will see that you will get BST, with almost half elements in left sub-tree and half in right sub-tree. You have to do following tasks:

a. Download "Task3.py" having all the functions either implemented or to be implemented with appropriate guidelines in comments
b. Complete function to count nodes, you may check your function by constructing Binary Tree from in-order and pre-order traversal

c. Complete function to remove nodes in post-order traversal that is delete node after deleting left and right sub-tree
d. Complete function ***checkAndBalance***, this function is just checking balance counting nodes on left and right sub-trees and calling another function to do actual task
e. Complete function ***checkAndBalanceTree***, this function will declare array, call another function to store values from tree to array in ascending order, next this function will remove the tree by calling function create in step "***C***" and call another function to reconstruct balanced tree by adding nodes in tree appropriately
f. Complete function ***inOrderArray***, this function will call itself recursively with appropriate index and store tree data into array.
g. Complete function ***addBinarySearch***, this function will add array elements into empty tree in the same way as we do binary search that is first add middle element, then add middle element from first half, next add middle element from first quarter and so on
h. Lastly, uncomment one commented statement from main, run your code and try to verify your functionality from reconstruction, if code successfully runs (Big If, otherwise do debugging to find errors and ENJOY)