Algorithms

Assignment 6

Ashwin Sathiyanarayanan
002227609

Question 1:

```
def Recursive_Tree_Insert(T, current_parent, new_node):
    if T.root is None:
        T.root = new_node
        new_node.parent = None
    else:
        if new_node.key < current_parent.key:
            next_node = current_parent.left
        else:
            next_node = current_parent.right
        if next_node is None:
            if new_node.key < current_parent.key:
                current_parent.left = new_node
            else:
                current_parent.right = new_node
            new_node.parent = current_parent
        else:
            Recursive_Tree_Insert( T, next_node, new_node)
```

Explanation:

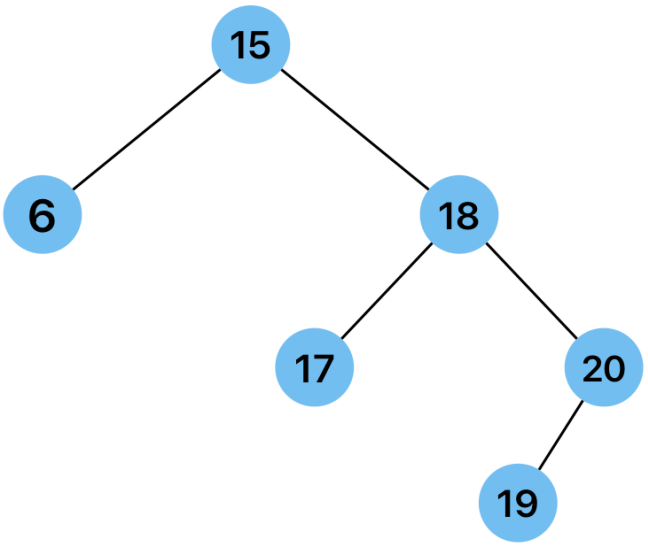
1. In order to convert the iterative tree insert into a recursive one, first we have to find the terminating condition of our recursion. In this case, we have two terminating conditions. One is if the root of the tree is None, (i.e.) the tree is empty. The next condition is that when the next_node variable is None, (i.e.) we have reached the place where we have to insert our element.
2. Now, first, we check if the root of the tree is none, and if it is true, it means that the tree is empty. Then in that case, we set the root of the tree to the new node and and set the parent of the new_node to None.
3. Then, if the tree is populated, we first check where the position of the new_node should be fixed. For this, we compare the values of keys both the new_node and the current_parent, and move the next_node variable either to the left or to the right of the current_parent depending on the value comparison.

4. Then we check if the value of the next_node variable is None or not. If it is None, it means that we have reached the node to which new_node should be a child of. So in this case we just compare the values of keys of new_node and current_parent and set the new_node either to the left or to the right of the current_parent node. And finally we set the parent of the new_node to the current_parent.
5. If and when the next_node is not None, we recursively call the method with the current_parent being the next_node. This goes on until the next_node is None.

Question 2:

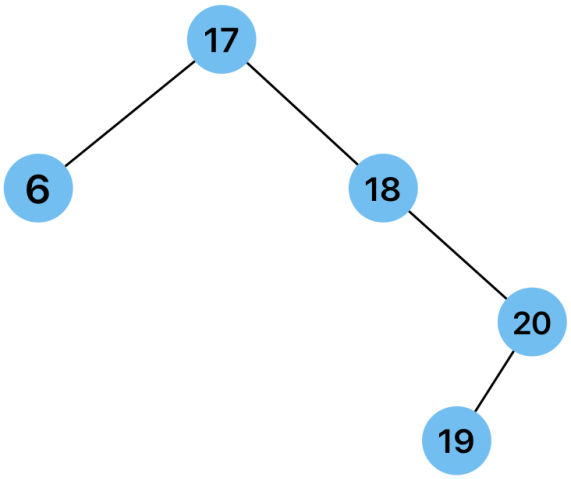
The operation of BST delete is not a commutative operation. This means that deleting node A and deleting node B may give a different resultant tree from first deleting node B and then node A. It is true that in some cases it may yield the same tree but in other cases, the tree is not the same. Let us see this with the help of an example:

Let us consider a BST as below:

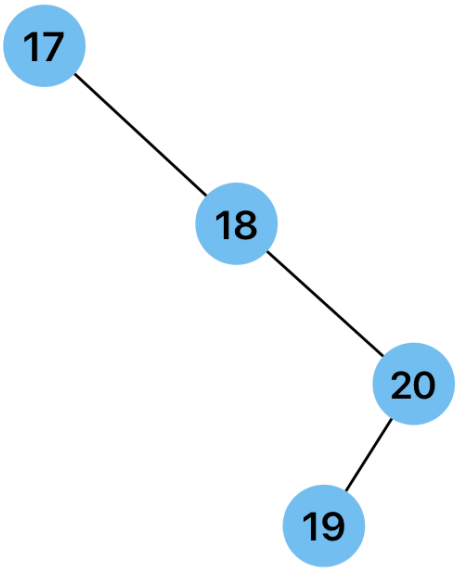


Now, let us delete node 15 first and then node 6.

After deleting node 15:



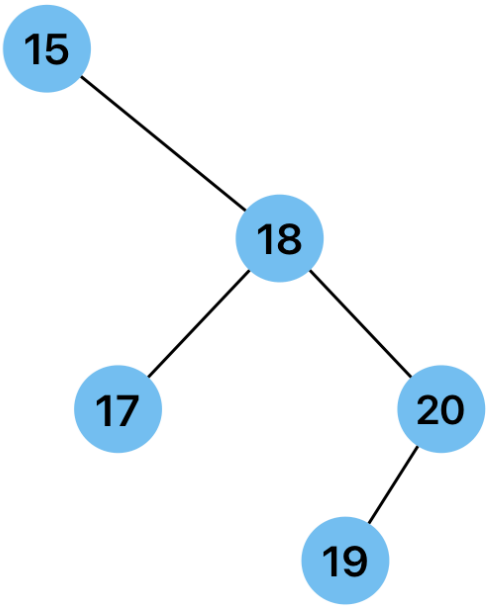
After deleting node 6:



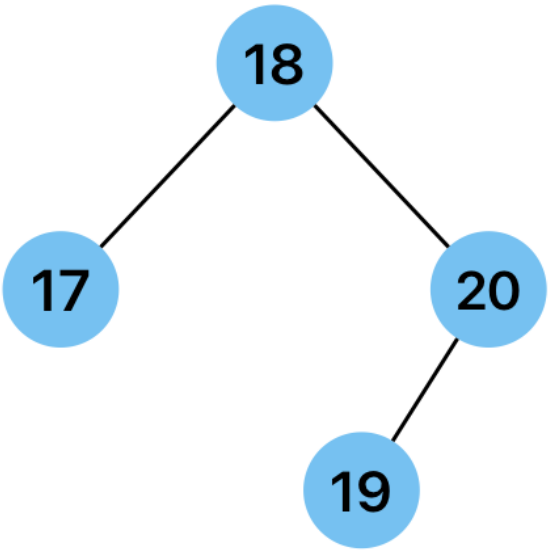
Now, this is the final tree after deleting the nodes in the order 15 and 6. Now let us change the order.

Let us first delete node 6 and then node 15 from the original tree.

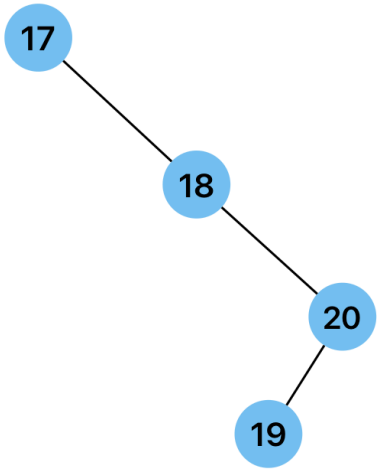
After deleting node 6:

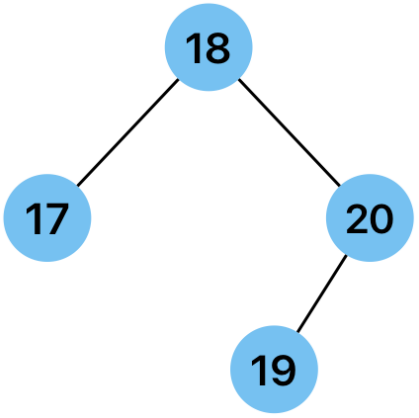


After deleting node 15:



Now let us compare the two resultant trees after deleting nodes in the respective orders as above.

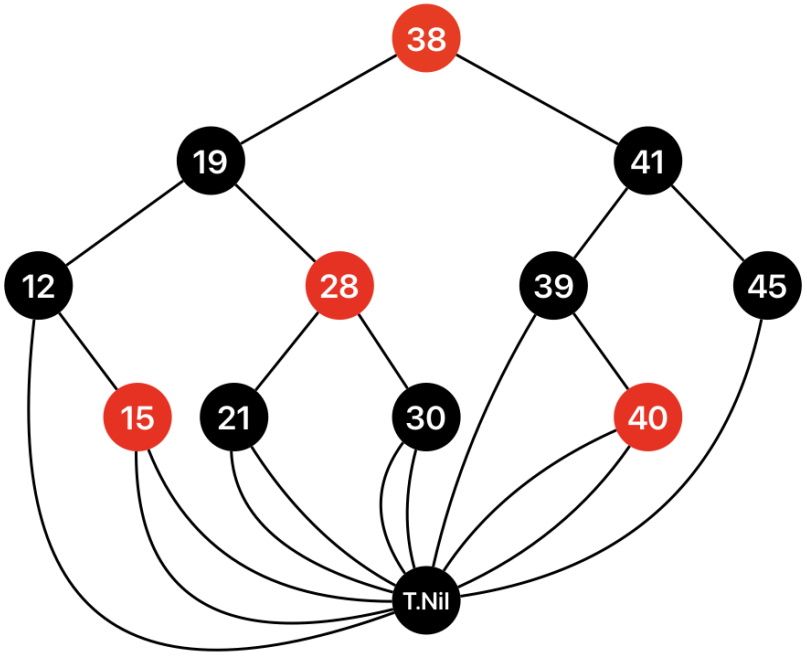




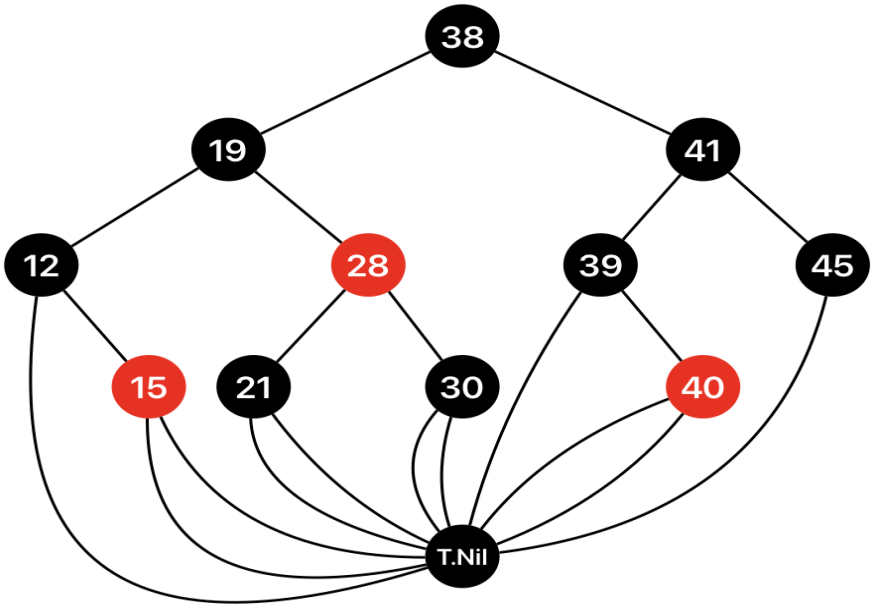
As we can see above, the two trees are completely different. So the order of the deletion matters in a BST delete method. Therefore the deletion operation of a BST is not commutative in nature.

Question 3:

Let us consider a relaxed red-black tree with root red that satisfies all the properties 1, 3, 4, and 5 of RBT like below:



Now let us change the root node to black with no other changes, and check whether it is still a valid RBT or not.



Now let us verify whether the tree is still a valid RBT or not by checking the 5 properties of RBT.

Property 1:
Property 1 states that every node is either red or black.
Property 1 is still valid because as we can see all the nodes in the tree are either black or red. This is because we have only changed the color of one node from black to red. So property 1 is true.

Property 3:
Property 3 states that every leaf (NIL) is black.
All the false leaves of the tree are connected to the T.nil node which is a black node. So property 3 is valid as the original leaf of all the false leaves is black. This is because we have not introduced any new child node and only changed the color of the root node. So property 3 is valid.

Property 4:
Property 4 states that if a node is red, then both its children are black.
As we can see from the tree above all the children of all the red nodes, such as 28, 15, and 40, are only black. This is true because of the same reason as above. We have not added any new red nodes to the tree. So Property 4 holds and is valid.

Property 5:
Property 5 states that for each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
We can infer from the trees above that the black height of the tree with a red node is 2 and the black height of the tree with the black node is 3. Both of them are valid since by changing the color of the root from red to black we have increased the black height of all the subtrees by 1. Therefore, the property 5 holds and is valid.

Therefore, the relaxed red-black tree after changing the root of the tree to black is still a valid RBT.

Question 4:

```
Right-Rotate(T, x):
    y = x.left
    x.left = y.right
    if y.right ≠ T.nil
        y.right.p = x
    y.p = x.p
    if x.p == T.nil
        T.root = y
    else if x == x.p.right
        x.p.right = y
    else
        x.p.left = y
    y.right = x
    x.p = y
```

Explanation:

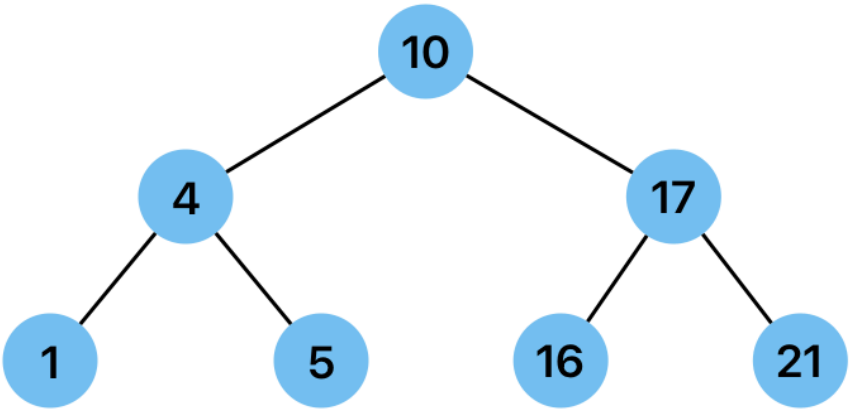
1. The method of right-rotating a node uses the same logic as the left-rotate method.
2. First, we set y to the node to the left of the node to be rotated.
3. Then, starting now, we change the pointers of the two nodes and their children.
4. First, we set the node to the left of x to be the node to the right of y.
5. Then, we check if y.right is equal to T.nil and if it is not, we set the parent of the right of y to be x.
6. In the above two methods, we turn y's right subtree into x's left subtree.
7. Then we link the parent of x to y.
8. Following that, first, we check if the parent of x is Nil. If it is Nil, it means that x is the root of the tree. In this case, if it is true, we set the root of the tree to y.
9. If not true, we check if x has a right-child relationship with its parent (i.e.) we check if x is the right child of its parent and if true, we set the right child of x's parent to y.

10. Finally, if all these are false, it means that x is a left child to its parent, and in this case, we set the left child of x's parent to y.
11. At last, we put x on y's right and set the parent of x to y. With this, we completed the right rotation of node x.

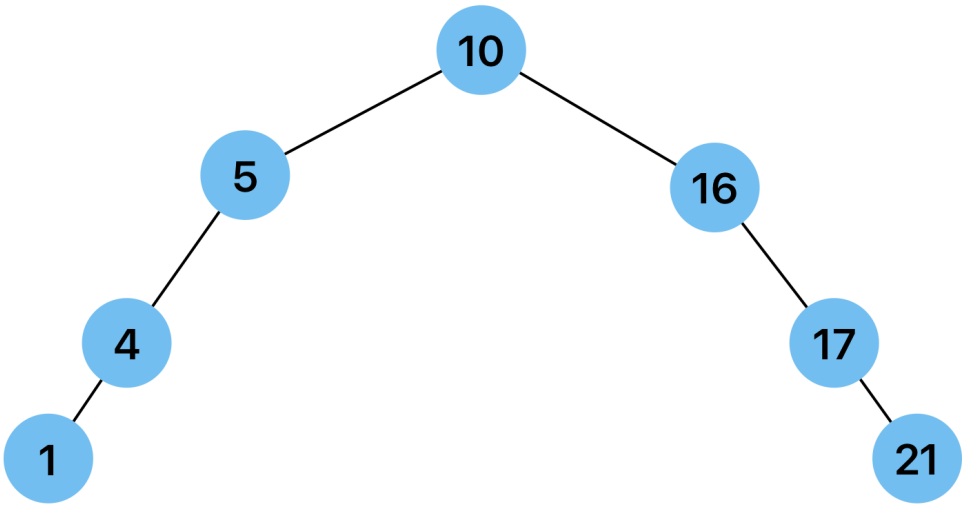
Question 5:

Elements: {1, 4, 5, 10, 16, 17, 21}

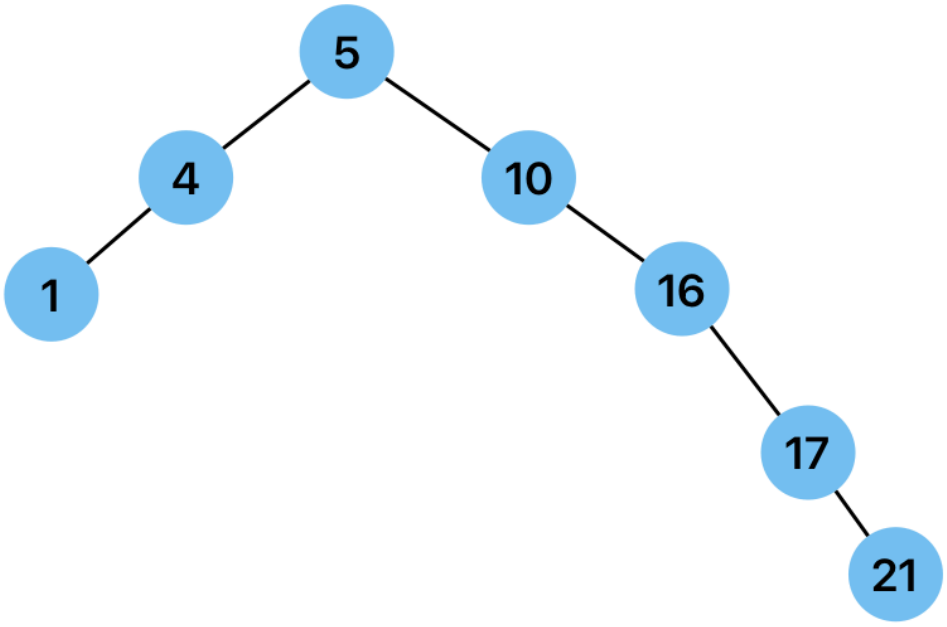
I. Tree of Height 2:



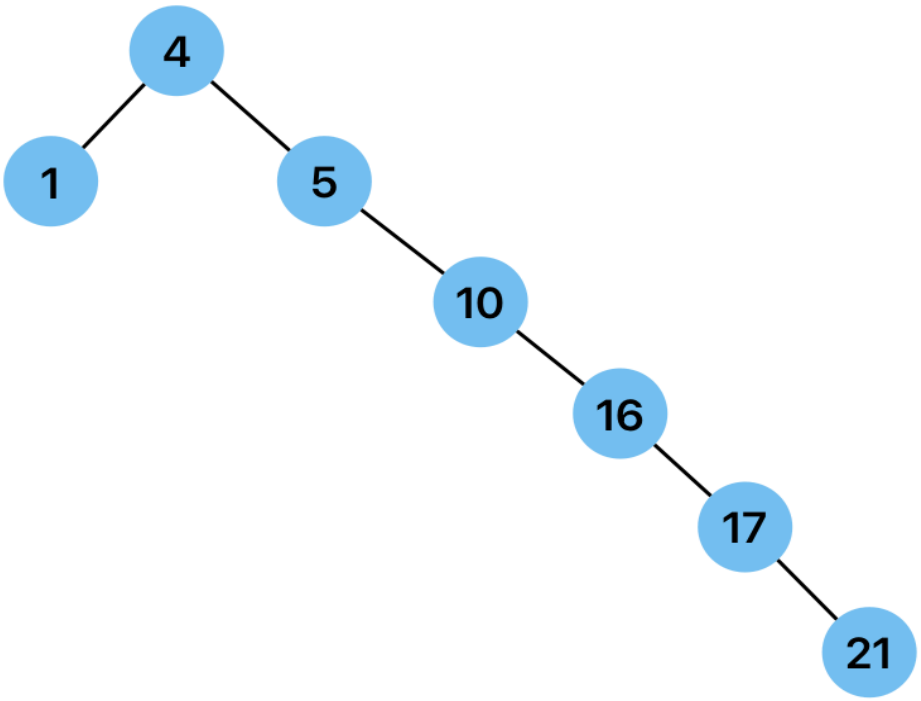
II. Tree of Height 3:



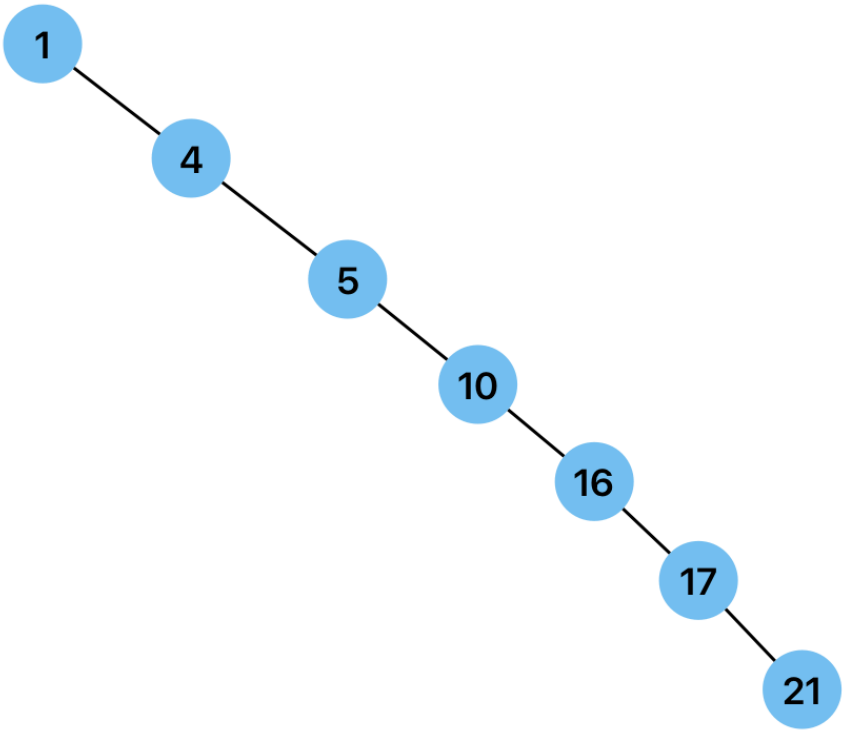
III. Tree of Height 4:



IV. Tree of Height 5:



V. Tree of Height 6:



Question 6:

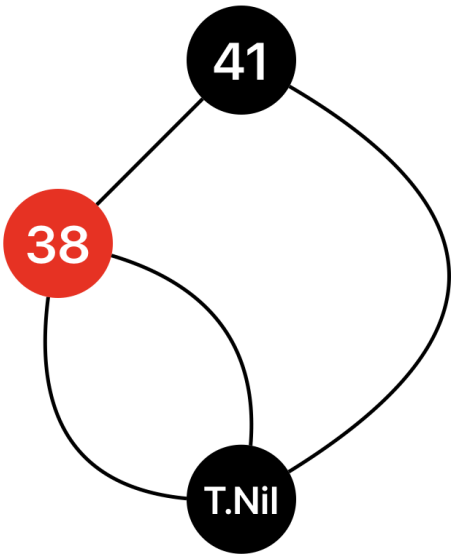
During the insertion of a new node into the red-black tree, if we choose to set z's color to black, it is true that property 4 of RBT, which states that if a node is red, then both its children are black, would not be violated. But by setting z's color to black, we would for sure violate the 5th property of RBT. It states that for each node, all simple paths from the node to descendant leaves contain the same number of black nodes. When we insert a new node, z, into the tree and set its color as black, we increase the number of black nodes on z's side of the tree by one, which clearly violates property 5 of RBT. Conversely, if we set the new node, z's color to be red and call the insert fixup method, we will not be violating property 5, and also after executing the fixup method, the property 4 violation, that we faced earlier while setting z's color to red, will be rectified. Additionally, the insert fixup method also rotates some subtrees which makes the tree more balanced.

Question 7:

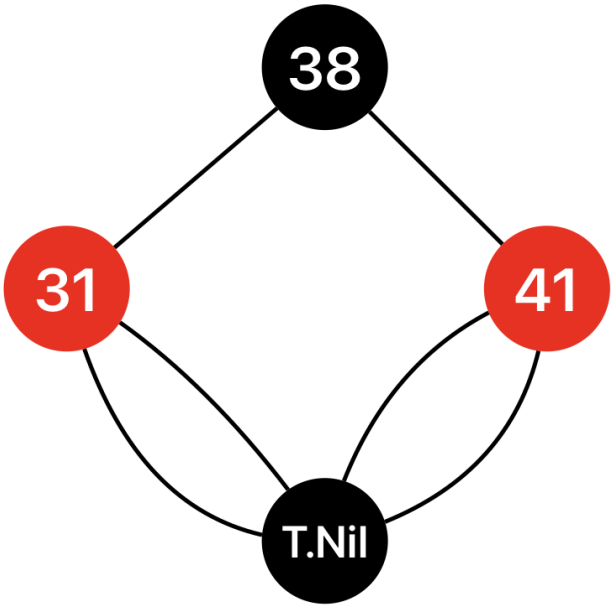
Keys to be inserted: 41, 38, 31, 12, 19, and 8

I. Inserting 41:

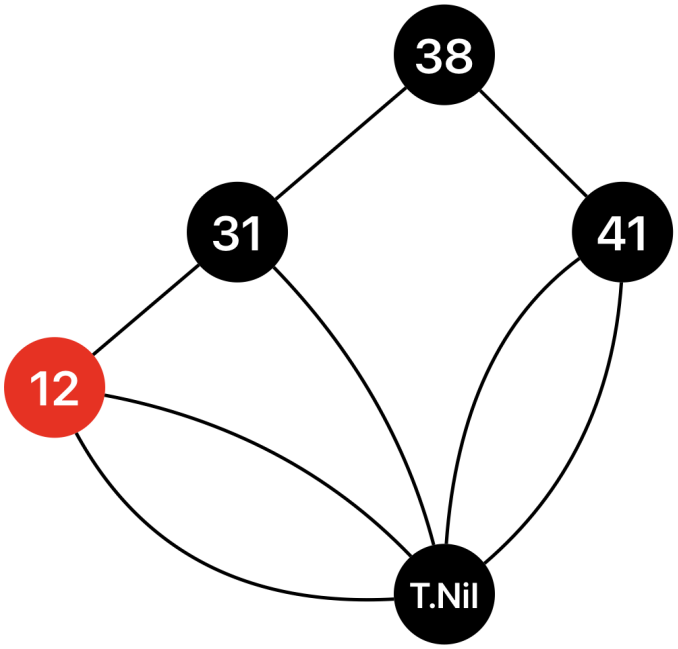
II. Inserting 38:



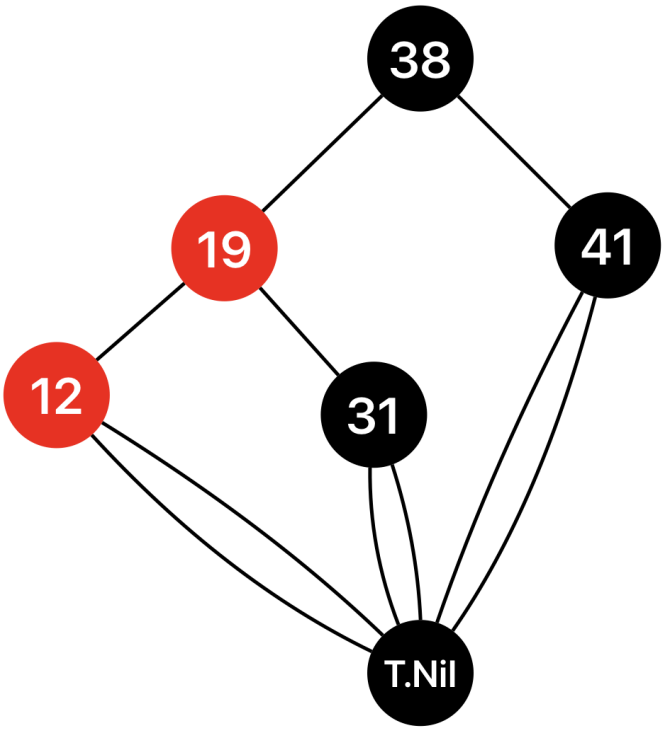
III. Inserting 31:



IV. Inserting 12:



V. Inserting 19:



VI. Inserting 8: