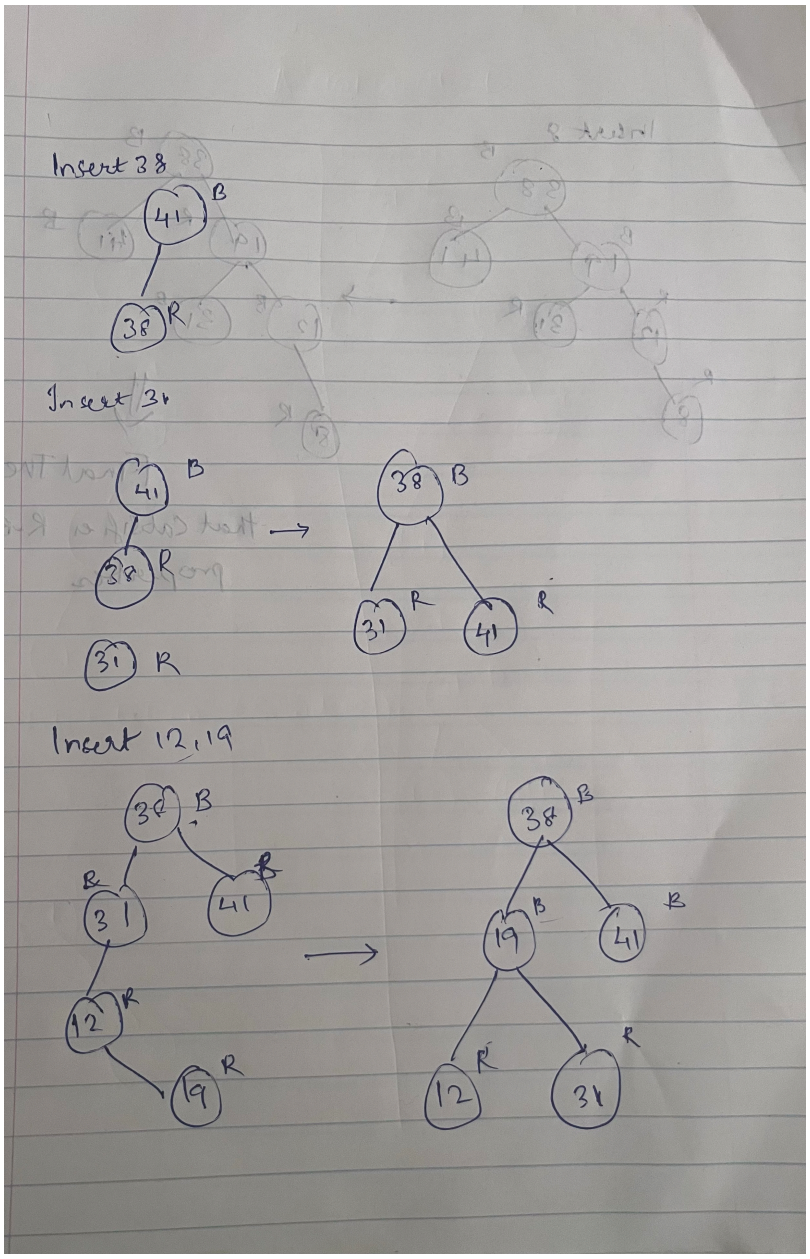
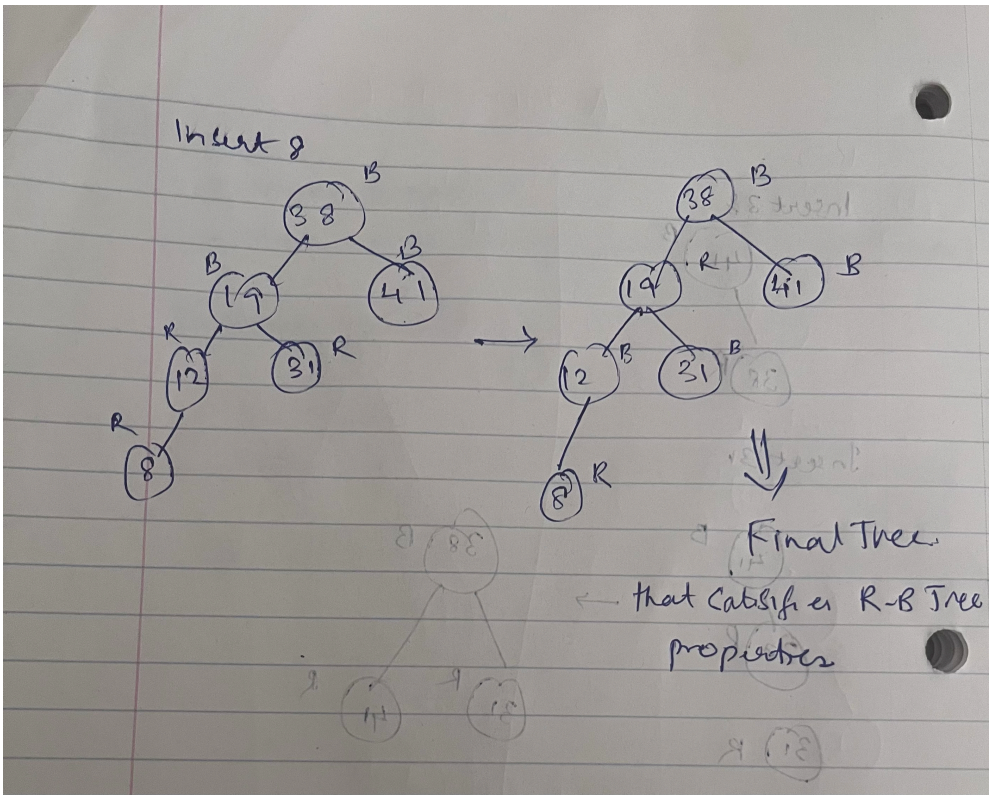
ASSIGNMENT 9 **Vikramaditya Reddy**

Z1973679

1. Show the red-black trees that result after successively inserting each of the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree. For each key, show each tree after inserting each key.





2.Argue that any arbitrary *n*-node binary search tree can be transformed into any other arbitrary *n*- node binary search tree using *O*(*n*) rotations. Hint: First show that at most *n*-1 right rotations suffice to transform the tree into a right-going chain.)

Consider any arbitrary binary search tree with n nodes. We can transform it into a right-going chain in the following way:

- Start at the root node and move down the left child as far as possible until we reach a leaf node.
- Perform a right rotation to bring the leaf node up to the root of the right subtree.
- Repeat above steps until the tree becomes a right-going chain.

Each iteration reduces the height of the left subtree by one and finally we end up with a right-going chain with n nodes. Since the original tree has a height of at most n-1, we need at most n-1 iterations and therefore at most n-1 right rotations to transform it into a right-going chain.

Now, let's consider two right-going chains, A and B, each with n nodes. To transform A into B, we can use the following algorithm:

- Start at the root of A and B.
- If the roots have the same key, move to their right children.
- If the root of A has a smaller key than the root of B, perform a left rotation on the root of B and its right child. Then move to the left child of the new root of B.
- If the root of A has a larger key than the root of B, perform a right rotation on the root of A and its left child. Then move to the right child of the new root of A.
- Repeat until the roots of A and B have the same key.

Here we can perform each rotation in O(1) time.

Therefore, the total number of rotations required to transform A into B is at most 2(n-1), which is O(n).

3.Can the black-heights of nodes in a red-black tree be maintained as a data field in the nodes of the tree without affecting the asymptotic performance of insertion and search in the red-black tree? Show how, or argue why not.

Ans:
Yes, the black-heights of nodes in a red-black tree can be maintained as a data field in the nodes of the tree without affecting the asymptotic performance of insertion and search in the red-black tree.
Red-black trees are a type of self-balancing binary search tree that ensure that the tree is balanced and that the worst-case performance of operations is O(log n).
The black-height of any node in the tree is the same for all nodes in its subtree.The black-height of a node is defined as the number of black nodes on the path from that node to a leaf node, not including the node itself.
Maintaining the black-heights of nodes as a data field in the nodes of the tree does not affect the asymptotic performance of insertion and search in the red-black tree. This is because the black-heights of nodes can be updated during insertion or deletion operations in O(1) time, which does not affect the O(log n) worst-case performance of these operations.

4.The interval tree is also a red-black tree. When inserting or deleting a node from interval tree, rotation will be involved. The `max` data fields of involved nodes will need update. Use Left- Rotate to explain how the `max` data fields of involved nodes can be updated in *O*(1) time.

Left- Rotate method to update max data fields involved nodes in O(1) time

- Let a be the node that will be rotated to the left, and b be its right child.
- Save the max data fields of a, b, and b's left child (if it exists).
- Perform the left rotation as usual, with a becoming b's left child and b's left child becoming a's right child.
- Update the max data fields of a and b based on their new children.
- If b has a left child, update its max data field based on its new children.
- Restore the saved max data fields from step 2 to the appropriate nodes.

5.Describe an efficient algorithm that, given an interval *i*, returns an interval overlapping *i* that has the minimum low endpoint, or null if no such interval exists.

- Create an empty variable to store the minimum low endpoint interval.
- Iterate through each interval in the dataset
- if overlaps with i, check if variable is empty or if the low endpoint of the current interval is lower than the low endpoint of the result interval.
- If either condition in previous step is true, update the result variable to be the current interval.
- Once all intervals have been checked, return the result variable if it is not empty, otherwise return null.

Algorithm:

```
function Overlap(i, intervals):
    x= null
    for interval in intervals:
        if i.high >= interval.low and interval.high >= i.low:
            if x is null or interval.low < x.low:
                x = interval
    return x
```