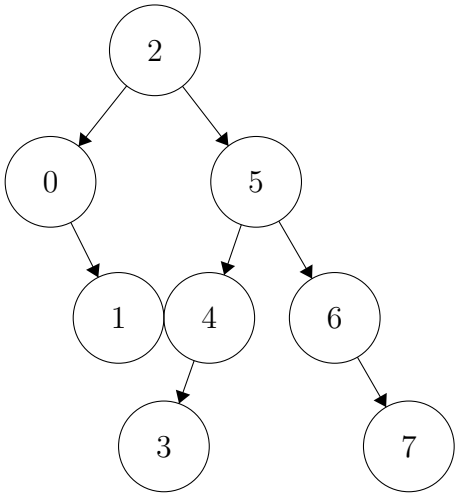
Vladimir Hugec

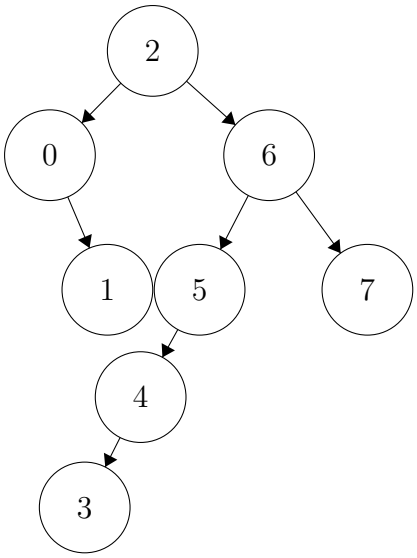
Question 1:

1.1 A

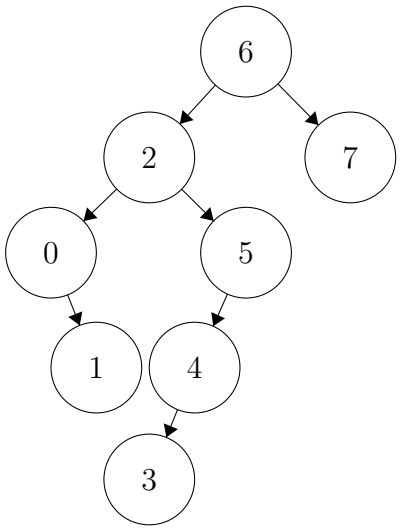


1.2 B

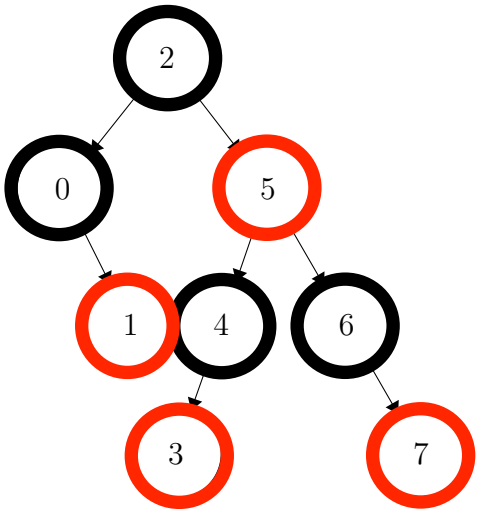
Left-rotation on 5:



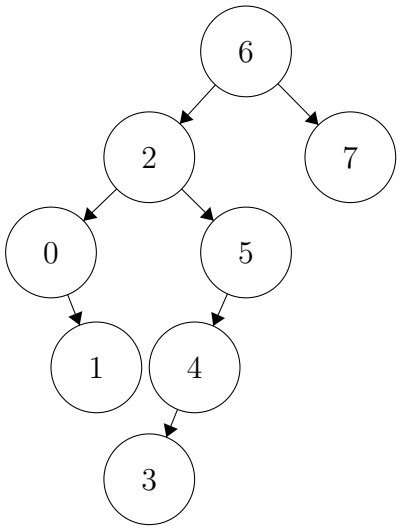
Followed by a left-rotation on $2 = T'$:



1.3 Red-Black T



1.4 Red-Black T'

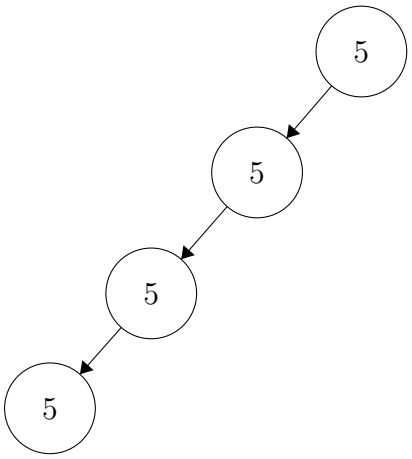


Coloring this tree red black is not possible since because of the (5,4,3) subtree you cannot color it so that the all the lengths from root-to-nil nodes contain the same number of black nodes.

Question 2:

2.1 A

You are not guaranteed to be able to balance the tree if you allow for duplicates. For example:



The tree is not balanced, and any attempts to balance the tree would invalidate the premise that all values that are equal to the parent are left side children. That being said, this does not mean that all tree's that allow for these duplicates cannot be balanced, just that you are not guaranteed to be able to.

2.2 B

Since by definition of our structure we know that duplicate items must be left side children of the desired value, we can eliminate searching the right side in our algorithm in the first case, in all subsequent levels both sides must be searched. The algorithm is as follows:

```
FUNCTION findAllOccurances (POINTER BSTnode, value, POINTER count)
    LOOP while BSTnode is not NULL
        IF value at node in BST equals our desired value
           increment count by one
           recursivly call findAllOccurances on left subtree
        ELSE IF value at node in BST is greater than our desired value
            recursivly call findAllOccurances on left subtree
        ELSE IF value at node in BST is less than our desired value
            recursivly call findAllOccurances on right subtree
        END-IF
    END-LOOP
        RETURN count
    END-FUNCTION
```

2.3 C

This algorithm is correct since it recursively goes through each possible spot a duplicate of the desired value can be found. If the duplicate value we are looking for is found, then all subsequent duplicates must be a left child of that node. At the next level, if the node, now our relative root, is not the desired value, then it is compared to the value and based on that comparison, either the left or right subtree is recursed on.

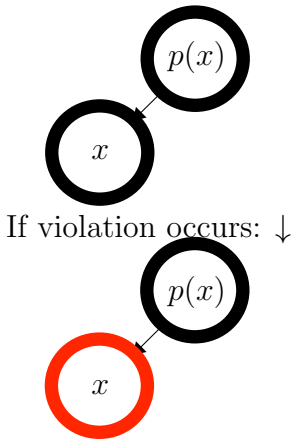
2.4 D

Since we know that we are never recursing on more than one side of a node at any point, at each recursive call we are effectively cutting the search space in half. If we start with n elements in the BST, and at each recursive call we narrow the possible landing spot of a duplicate by half we can say with reasonable certainty, assuming that the comparisons at each level run in constant time, that the algorithm would then be $O(\log n)$.

Question 3:

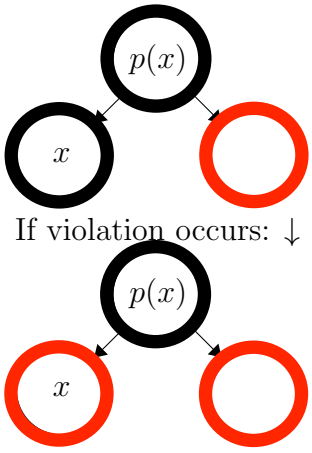
3.1 A

In this case, insert black x as left child of $p(x)$. If this addition violates that rule that all routes from root to leaf (NIL) are equal, recolor x red.



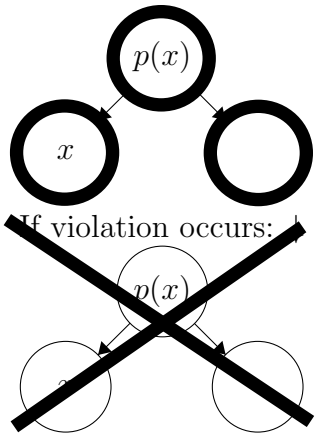
3.2 B

In this case, insert black x as left child of $p(x)$. If this addition violates that rule that all routes from root to leaf (NIL) are equal, recolor x red.



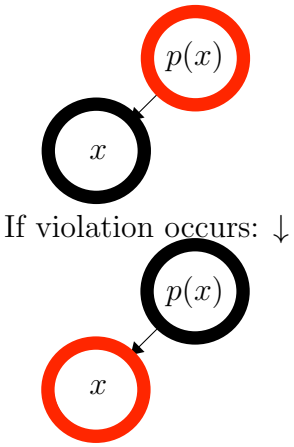
3.3 C

In this case, simply insert black x as left child of $p(x)$.



3.4 D

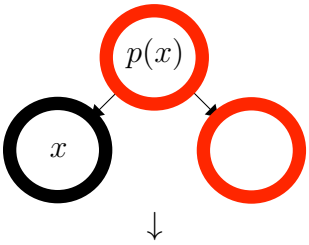
In this case, simply insert black x as left child of $p(x)$. If this addition violates that rule that all routes from root to leaf (NIL) are equal, recolor x red and $p(x)$ black. If $p(x)$ is the root, also recolor $p(x)$ black and x red.



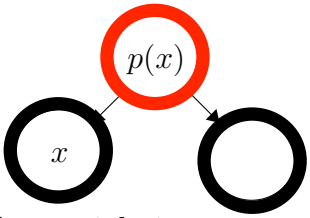
3.5 E

In this case, insert black x as left child of $p(x)$. The sibling of x violates the rule that children of a red-parent must be black, recolor x's sibling black. If

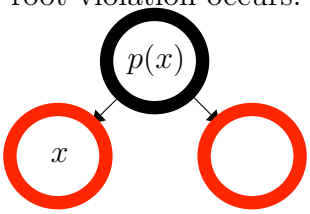
on the other-hand $p(x)$ is the root, recolor $p(x)$ black then both x and its' sibling red.



$\downarrow$

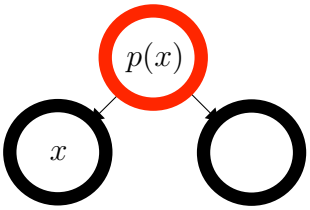


If root violation occurs: $\downarrow$



3.6 F

In this case, insert black x as left child of $p(x)$. If $p(x)$ is the root, recolor $p(x)$ black, and both x and its' sibling red.



If violation occurs: $\downarrow$