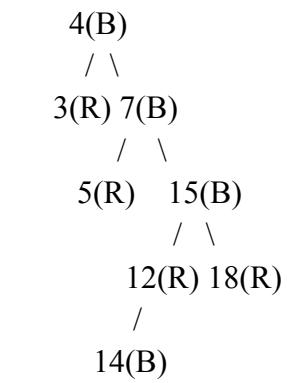
CS 430 – Fall 2024
Introduction To Algorithms
Homework #4

1)
```
function isInRange(node, a, b):
    if node is null:
        return False
    if a <= node.key <= b:
        return True
    if node.key < a:
        return isInRange(node.right, a, b)
    if node.key > b:
        return isInRange(node.left, a, b)

    return False
```

The algorithm performs a binary search-like traversal of the tree. Since the tree is balanced, the height of the tree is O(log n), where n is the number of nodes. In the worst case, the algorithm will traverse from the root to a leaf, which means the time complexity is O(log n).

2a)

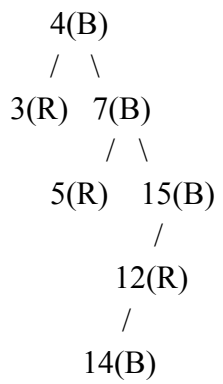


When 15 is inserted as a red child of 12. This results in two consecutive red nodes (12 and 15), which violates the red-black tree property. To fix this, 7 becomes red, and 12 and 15 become black.
When 5 is inserted as a red child of 7. This results in two consecutive red nodes (7 and 5). To fix this, 7 becomes black, 5 becomes red, and 4 remains black. No rotations are needed since the root is still valid.
When 14 is inserted as a red child of 15. This results in two consecutive red nodes (15 and 14). To fix this, 12 becomes red, 15 and 14 become black and we need to perform right rotation around 12 to maintain balance.

2b)

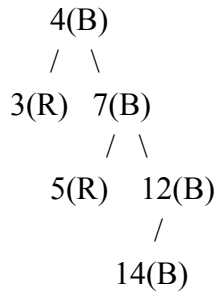
Step 1: Delete 18

18 is a red node and can be deleted without any violations.



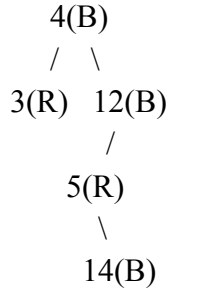
Step 2: Delete 15

15 is black. Replacing it with 12 results in two consecutive red nodes (12 and 14). To fix this, we change 12 to be black.



Step 3: Delete 7

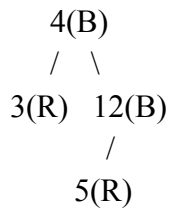
7 is black, so its black sibling, 12, can replace it



Step 4: Delete 14

14 is black. It can be deleted directly without any violations.

Final tree after deletions:



3)

Red-Black Tree

Pros:

- Red-black trees are self-balancing, ensuring that the tree height is always O(log n), where n is the number of elements in the tree.
- All operations (insertion, deletion, and searching for the largest element) can be performed in O(log n).
- In a red-black tree, the elements are always stored in a sorted manner, making it easy to find the largest element.

Cons:

- Red-black trees are more complex to implement compared to heaps.
- Although the tree remains balanced, the rebalancing operations can add constant overhead in practice compared to heap-based structures.

Runtime:

- The first step takes O(i log i), and for each of the remaining n − i elements, you perform insertion and deletion, each taking O(log i). So the total runtime is O(n log i).

Min-Heap

Pros:

- Insertion in a heap takes O(log n)
- It is easy to implement and generally requires less overhead than a red-black tree.
- Heaps are compact and memory efficient compared to binary search trees.

Cons:

- Since the heap property guarantees that the smallest element is at the root, finding the largest element requires traversing the entire heap, which takes O(n) time. This makes it slower compared to max-heaps or red-black trees.

Runtime:

- The first step takes O(i log i), but for each of the remaining n − i elements, finding and removing the largest element costs O(i), making the total runtime O(ni).

Max-Heap
Pros:

- In a max-heap, the largest element is always at the root, so finding the largest element takes O(1) and removing it takes O(log n).
- Like the min-heap, the max-heap is also compact and simple to implement.

Cons:

- Unlike a red-black tree, a heap does not maintain a sorted order for all elements, but this is not a critical drawback for this problem.

Runtime:

- The first step takes O(i log i), and for each of the remaining n − i elements, you insert and remove the largest element in O(log i), making the total runtime O(n log i).

Max-Heap is the best choice for the data structure S. It provides efficient insertion, quick access to the largest element, and removal of the largest element. The overall runtime is O(n log i), which matches the efficiency of the red-black tree, but with simpler operations and less overhead.

4)
The height of the tree is maximized when there are as many red nodes as possible. In this case, the total height of the tree would be 2k because there can be at most one red node between every pair of black nodes, this would contain $2^{2k} - 1$ internal nodes.

The height is k, and the tree contains only black nodes, so the smallest number of internal nodes corresponds to a perfect binary tree of height k. This has $2^k - 1$ internal nodes.

5)
Find the rank of x:
Start by finding the rank of x in the order-statistic tree. This can be done in O(log n) time by recursively comparing x with the root of the tree and adjusting for the size of the left subtree as we move down.

Find the i-th successor:
Once we know the rank of x, we need to find the node with rank(x) + i. This can also be done using the size fields in the order-statistic tree.

- Starting from node x, we need to follow the tree according to the target rank.
- At each node u, compare the desired rank with the rank of the current node u, which can be determined using the size of its left subtree:
    - If the target rank is less than the rank of u, go to the left child.
    - If the target rank is greater than the rank of u, subtract u's rank from the target rank and go to the right child.

6)

```
def Count(T, x):
    count = 0
    node = T.root

    while node is not None:
        if x < node.key:
            count += 1
            if node.right is not None:
                count += node.right.size
            node = node.left
        else:
            node = node.right

    return count
```

Explanation:

- Start with the count set to 0.
- If x is smaller than the current node, the current node and all nodes in the right subtree of this node are larger than x, so we add them to the count.
- If x is greater than or equal to the current node, we simply move to the right subtree since we are looking for nodes larger than x.
- The loop terminates when we reach a leaf.

Time Complexity:

- Since the red-black tree is balanced, the height of the tree is O(log n), and the traversal takes O(log n) time.
- The size information at each node is used in constant time, meaning that adding the size of the right subtree (when moving left) only takes O(1) time.
- Thus, the total time complexity of the Count(x) function is O(log n).