ASSIGNMENT – 3

Introduction to Algorithms (CS 430)

1. Given a node x in binary search tree T, present an algorithm TREE −PREDECESSOR(x) that can find the predecessor of x in T. You may assume that x is not the tree minimum while creating this algorithm.

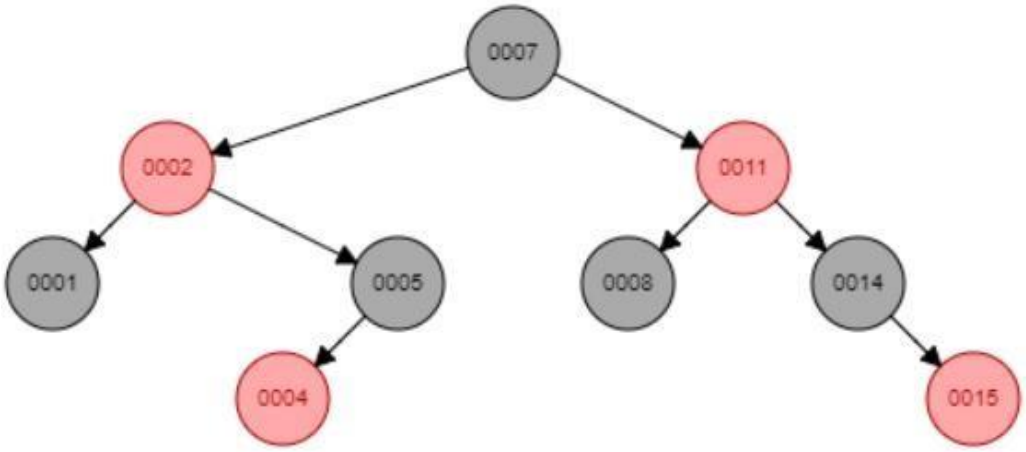
- To find the predecessor of a node x in a BST T, we need to consider, whether x has a left subtree or not. If x has a left subtree, the predecessor is the maximum node in that subtree. If x doesn't have a left subtree, we need to find the nearest ancestor for which x will be in right subtree.

  Here's the pseudo-code for the **TREE −PREDECESSOR** algorithm:

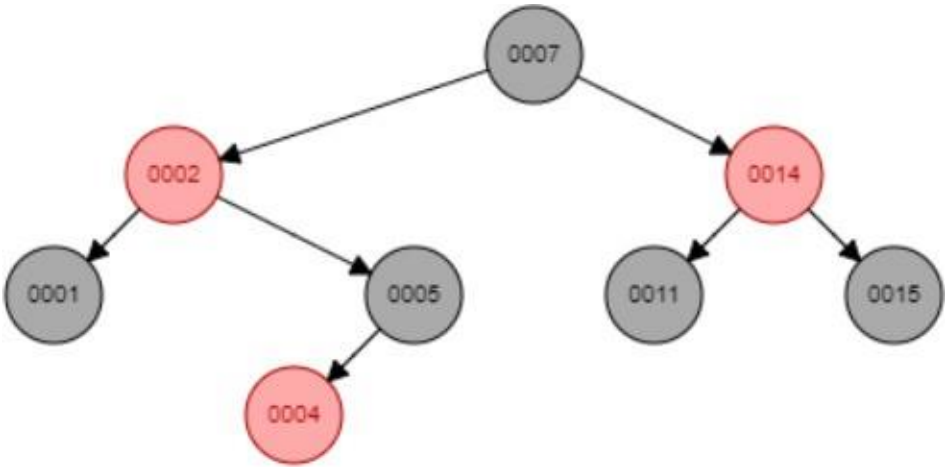
```
1. function TREE-PREDECESSOR(x)
2.       if x.left is not NIL
3.             Return TREE-MAXIMUM(x.left)
4.       else
5.             y = x.parent
6.             while y is not NIL and x == y.left
7.                    x = y
8.                    y = y.parent
9.              return y
```

2. Given the following red-black tree *T* (the "grey" nodes are red, and the black nodes are black), consider the following successive operations on;
***T : delete 8, insert 13, delete 7, insert 3,insert 6, delete 14*.**

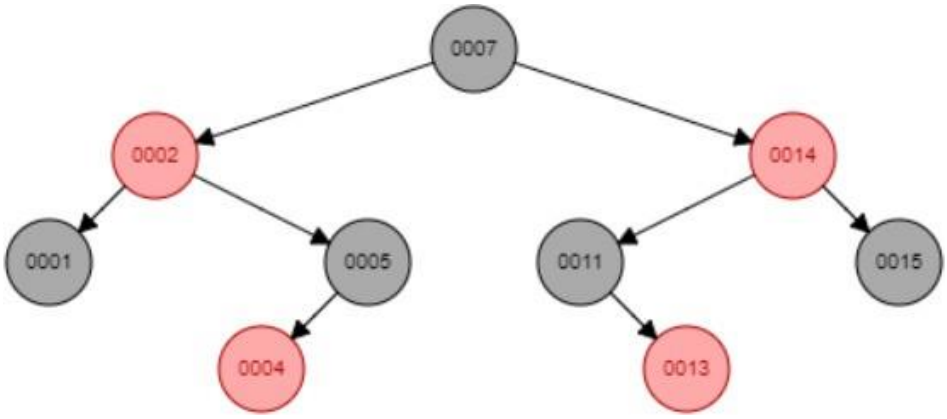
Given, Red and Black Tree



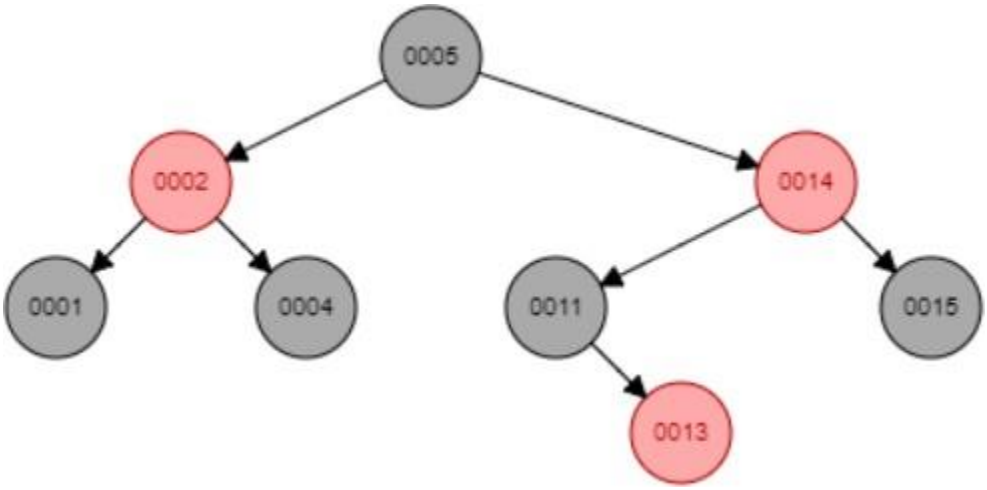
DELETE 8:



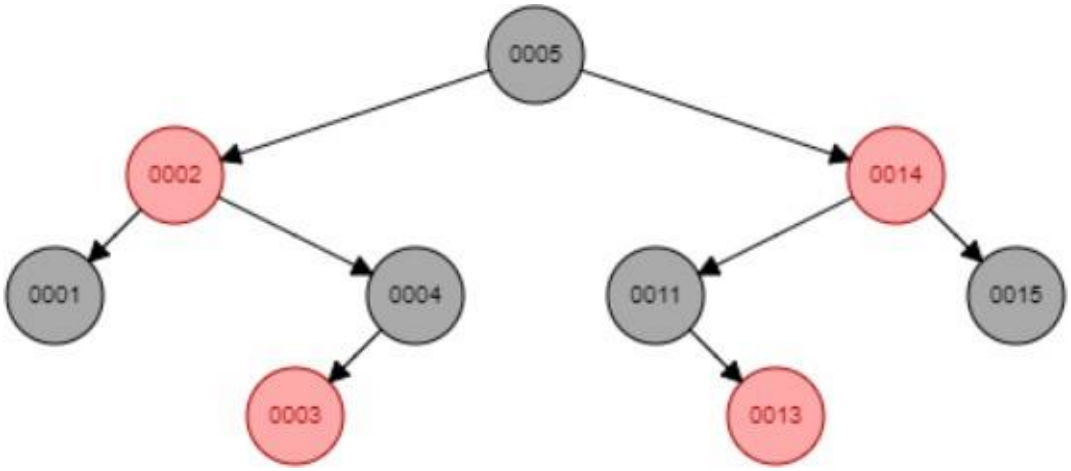
INSERT 13:



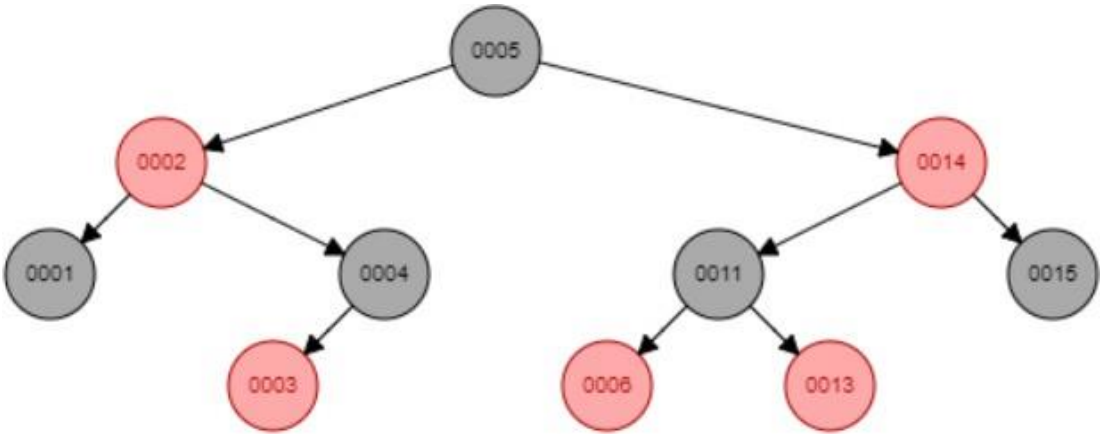
DELETE 7:



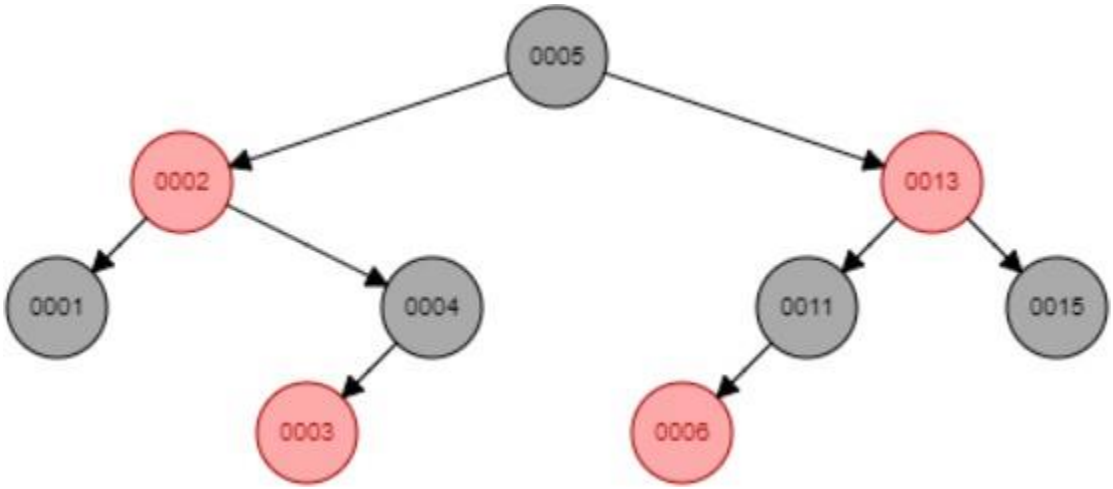
INSERT 3:



INSERT 6:



DELETE 14:



3.a)

To find the i_{th} successor of a node x in an n-node order-statistic tree T, we can use the following algorithm:

```
1. function OS-ITH-SUCCESSOR(T, x, i):
2.     rank_x = OS-RANK(T, x)
3.     if rank_x + i <= T.root.left.size:
4.         return OS-SELECT(T.root.left, rank_x + i)
5.     else:
6.         return OS-SELECT( T.root.right, i - ( T.root.left.size + 1 - rank_x ) )
```

Explanation:

i. We find the rank of node x using OS-RANK.

ii. If rank is <= size of left subtree of root, it means the i_{th} successor is in left subtree. Now, we use OS-SELECT on left subtree to find the node with rank equal to rank_x + i.

iii. If it is not above case the, the i_{th} successor is in right subtree. To find the rank of i_{th} successor, we subtract the size of the left subtree from i. Then we use, OS-SELECT on the right subtree with adjusted rank.

Time complexity of this algorithm is <u>O(lg n)</u> because OS-RANK and OS-SELECT takes O(lg n) time in an Order-Statistic Tree of size n.

b)

Yes, we can maintain the rank in nodes of a red-black tree as attributes without affecting the $O(\lg n)$ performance of insertion and deletion operations.

In a red-black tree, insertion and deletion operations only require updating the ranks of nodes along the path from the modified node to the root. This can be completed in $O(\lg n)$ time, as the height of a red-black tree is $O(\lg n)$.

Specifically, after inserting or deleting a node, we can update the ranks of the nodes along the path to the root as follows:

- For insertion, increment the ranks of all nodes on the path from the inserted node to the root, except for the root itself.
- For deletion, decrement the ranks of all nodes on the path from the deleted node to the root, except for the root itself.

Rank updates can be done during rebalancing without affecting the overall time complexity of $O(\lg n)$.

By maintaining node ranks as attributes and updating them during insertions and deletions, we can support both OS-RANK and OS-SELECT operations in $O(\lg n)$ time, without impacting the asymptotic performance of the red-black tree operations.

4.

To implement a binary counter that supports both Increment(A) and Decrement(B) operation with an amortized cost of O(1) per operation. Although, individual operations may appear to cost more than O(1) in the worst-case scenario (since a single increment or decrement may require flipping several bits), the average cost over a sequence of m operations will remain constant.

Pseudo code:

```
1. Initialize a k-bit binary counter A to all 0s
2. k = 10  # Example value for k
3. A = [0] * k
4. Function Increment(A):
5.     i = k - 1
6.     while i >= 0 and A[i] == 1:
7.         A[i] = 0  # Flip 1 to 0
8.         i -= 1
9.     if i >= 0:
10.        A[i] = 1  # Flip the rightmost 0 to 1
11. Function Decrement(A):
12.    i = k - 1
13.    while i >= 0 and A[i] == 0:
14.        A[i] = 1  # Flip 0 to 1
15.        i -= 1
16.    if i >= 0:
17.        A[i] = 0  # Flip the rightmost 1 to 0
```

Increment Algorithm:

To increment a k-bit binary counter A:

1. Initialize ‘i’ to the index of the least significant bit (rightmost bit).
2. While loop:
    a. Check if the current bit is 1.
    b. If it is, flip it to 0 and move to the next more significant bit (left).
    c. If it is 0, flip it to 1 and break the loop.
3. If the loop ends without breaking, all bits were 1 and are now 0, so no further action is needed.

Decrement Algorithm:

To decrement a k-bit binary counter A:

1. Initialize ‘i’ to the index of the least significant bit (rightmost bit).
2. While loop:

    a. Check if the current bit is 0.
    b. If it is, flip it to 1 and move to the next more significant bit (left).
    c. If it is 1, flip it to 0 and break the loop.

3. If the loop ends without breaking, all bits were 0 and are now 1, so no further action is needed.

Amortized Cost:

The amortized analysis considers the cost of a sequence of operations, rather than individual operations.

- **Increment Operation:**
    - Each bit-flip from 0 to 1 is ‘reset’ for that bit. That means it wont be flipped again until other bits are flipped.
    - At-most, each bit is flipped once per increment operation on average.
- *Decrement Operation:*
    - Each bit-flip from 0 to 1 is ‘reset’ for that bit.
    - At-most, each bit is flipped once per increment operation on average.

Because of this, the amortized cost of each operation remains constant:- O(1)

By carefully analyzing the sequence of flips, we can see that over a sequence of m operations, each bit is flipped at most m times, ensuring that the total cost is O(m). Hence, the amortized cost per operation is O(1).

5.

No matter what node we start at in a height-*h* binary search tree, m successive calls to TREE-SUCCESSOR take O(m+h) time.

Analysis:
Let's consider different scenarios in a height-h binary search tree and analyze the cost of m successive TREE-SUCCESSOR calls:

1. Starting at root:

    - In the worst case, we might need to traverse the entire height (h) of the tree to find the m_{th} successor if all nodes have a right child (left-skewed tree).

- However, each TREE-SUCCESSOR call after the first one only needs constant time $\Theta(1)$ to access the right child pointer.
- Therefore, the total cost would be dominated by the traversal down the tree (h) and some constant time lookups (m), resulting in O(h + m).

2. Starting at an internal node:

    - Similar to the root case, the worst-case scenario involves traversing down the right subtree for h steps in the first few calls.
    - Subsequent calls might involve some constant time lookups and potentially shorter traversals depending on the tree structure.
    - Again, the cost is dominated by the initial traversal (h) and some constant time operations (m), leading to O(h + m).

3. Starting at a leaf node:

    - There's no right child for a leaf node.
    - The first TREE-SUCCESSOR call might need to traverse up the tree to find a parent with the right child. In the worst case, this could take up to h steps.
    - Subsequent calls would likely fail to find a successor and return null, resulting in constant time operations.
    - Here, the cost is dominated by the potential traversal (h) in the first call and some constant time operations (m - 1), still resulting in O(h + m).

In all these scenarios:

- The worst-case cost for finding the first successor involves traversing the tree height (h).
- Subsequent calls involve constant time lookups (m) that don't depend on the tree height.
- Therefore, the overall time complexity is dominated by the sum of the maximum tree height (h) and the number of calls (m), which is O(h + m).

Based on the analysis of different starting nodes and scenarios, it's highly likely that m successive calls to TREE-SUCCESSOR in a height-h binary search tree take O(m + h) time.