Assignment N*3

Name: Harlee Ramos **Due Date:** 06/23/2024

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

In the following pseudocode for finding the predecessor, two scenarios were considered: the first while loop is to check if the node x has a left subtree, and the second while condition will validate if the node x has a right subtree.

```
TREE-PREDECESSOR(x)
    1.  If x.left ≠ NIL:
    2.  Set y = x.left
    3.  While y.right ≠ NIL:
    4.  Set y = y.right
    5.  Return y
    6.  Else:
    7.  Set y = x.parent
    8.  While y ≠ NIL and x == y.left:
    9.  Set x = y
    10. Set y = y.parent
    11. Return y
```

Time complexity analysis.

Line 1. This line will perform a check on the node x and determine if it is not null. This step will have a time complexity of O(1).

Line 2. This will set the value of y to x's left child, and then this step will take a time complexity of O(1).

Line 3-5. In these lines, a while loop is being performed. The initial condition is to check if the y's right child is not null; if yes, y will be set to the y's right child. However, that condition will depend on the height of the subtree with a root at x.left, so the time complexity is O(h).

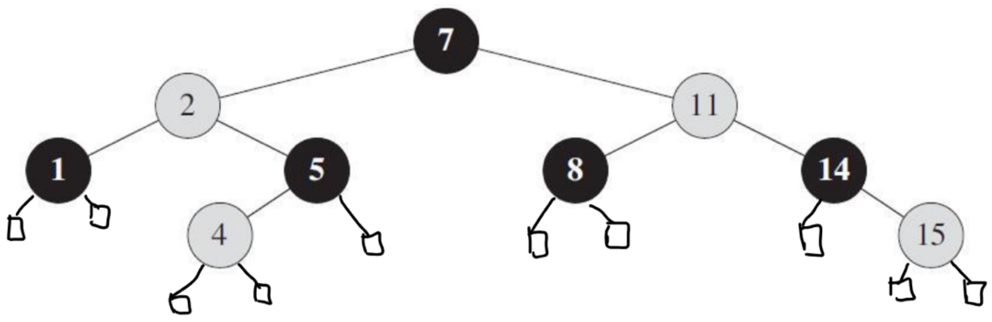
Line 6-7. These lines have a time complexity of O(1).

Line 8-10. This loop moves up the tree to find the predecessor of x. If the number of iterations in the worst case is proportional to the height of the tree, then the time complexity is O(h).

Line 11 has a time complexity of O(1).

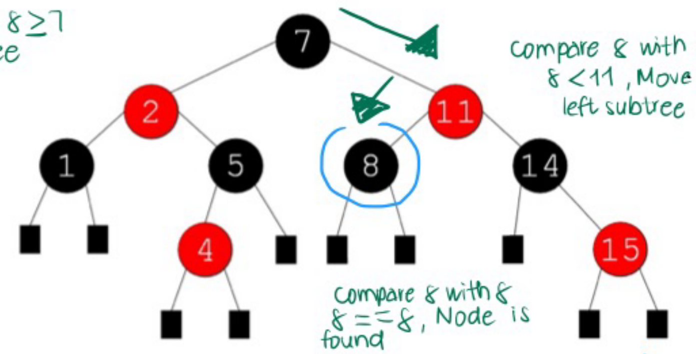
The TREE-PREDECESSOR(x) algorithm has a worst-case time complexity of O(h), where h represents the height of the binary search tree T. This is because, in the most unfavorable situation, we may have to cross the height of the tree, either in the left subtree or when ascending to the root, to locate the predecessor.

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: $delete\ 8$, $insert\ 13$, $delete\ 7$, $insert\ 3$, $insert\ 6$, $delete\ 14$. Draw the red-black tree after each of the above operations and show how it is produced.



Delete node 8

Compare 8 with 7 8≥7
Move to right subtree

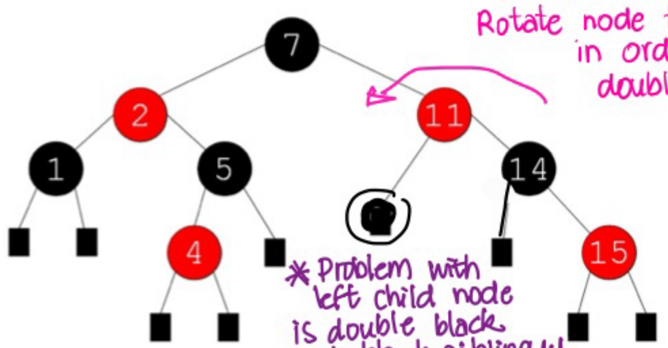


Compare 8 with 11
8<11, Move to left subtree

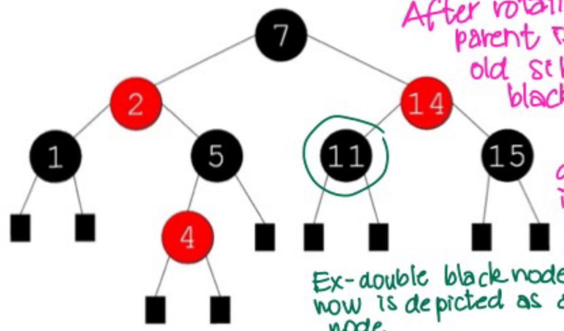
Compare 8 with 8
8==8, Node is found

Node 8 is a leaf node it can be easily removed

It's necessarily to Rotate node to left in order to avoid double blackness



* Problem with left child node is double black with black sibling 14 and its nephew 15 is red

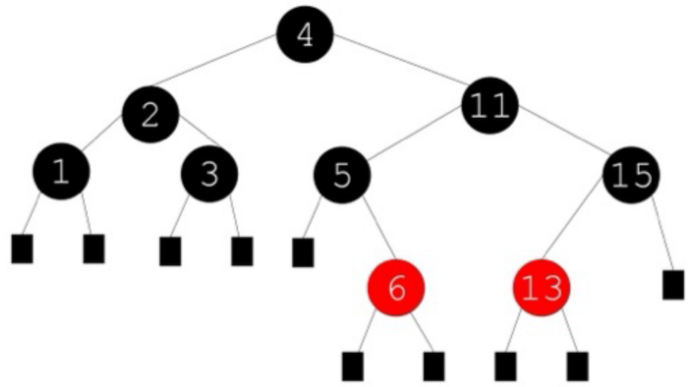


After rotation parent is red and old sibling is black now

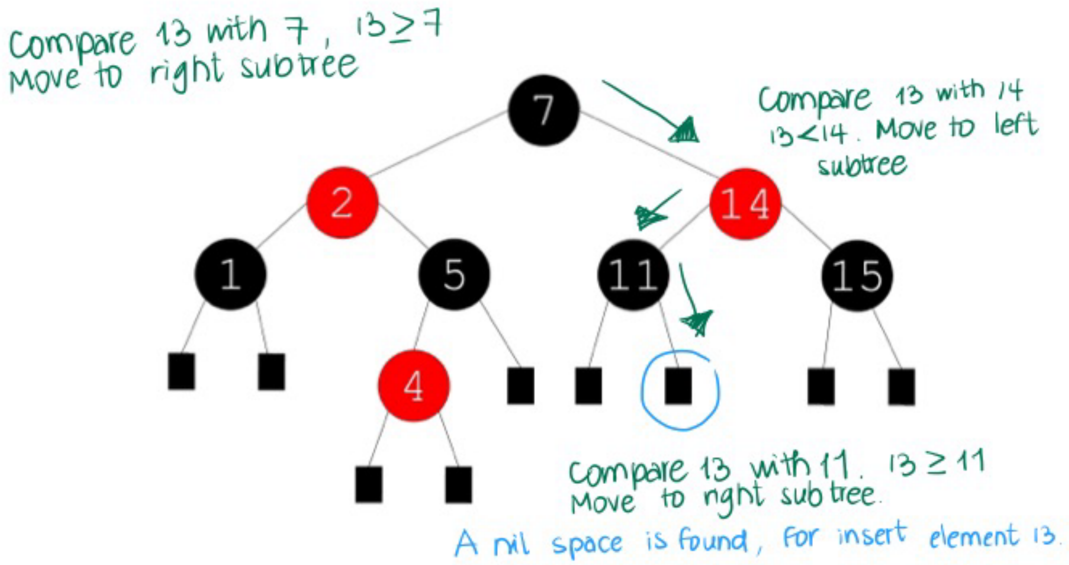
double black is colored to black

Ex-double black node now is depicted as a nil node.

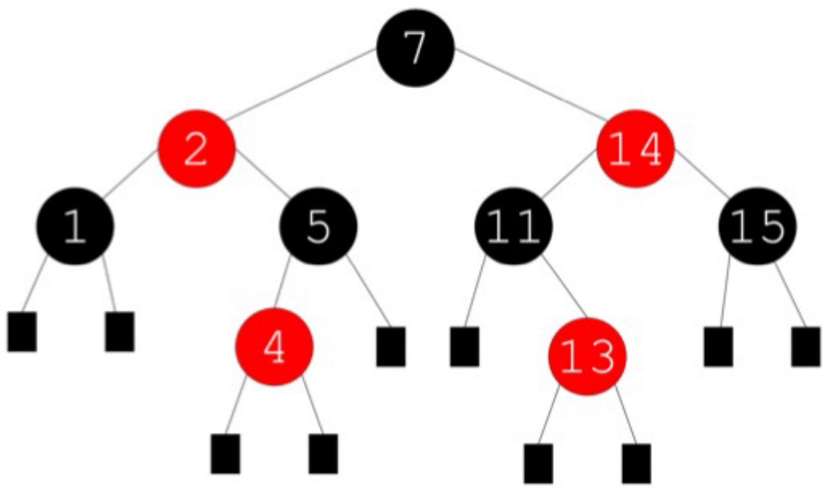
Final form of the tree:



Insert node 13


Compare 13 with 7, 13 ≥ 7
Move to right subtree
Compare 13 with 14
13<14. Move to left subtree
Compare 13 with 11. 13 ≥ 11
Move to right sub tree.
A nil space is found, for insert element 13.
7
2
14
1
5
11
15
4

Final form of the tree:



IN ORDER VIEW 1 2 4 $\boxed{5}$ ~~7~~ $\boxed{11}$ 13 14 15

Delete Node 7:

Compare 7 with 7 . 7 == 7.
Found node at root node

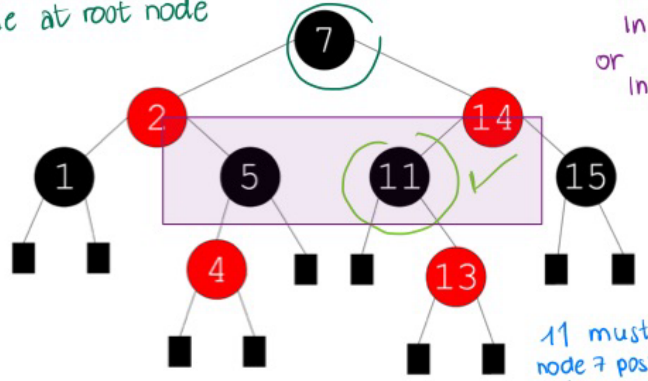
In this tree we have a decision problem.
Since the node has two children, the replacement node could be:

In-Order Predecessor (Maximun Node in Left subtree)
↳ Rightmost node in the left subtree.

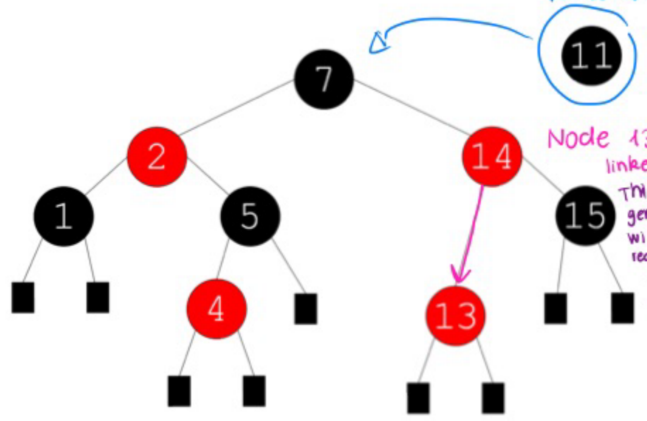
or

In-Order Successor (Minimum Node in Right Subtree)
↳ Leftmost node in the right subtree.

Then both are black nodes, so no matter the node. In this approach the chosen node is 11 which is the leftmost node in the right subtree.



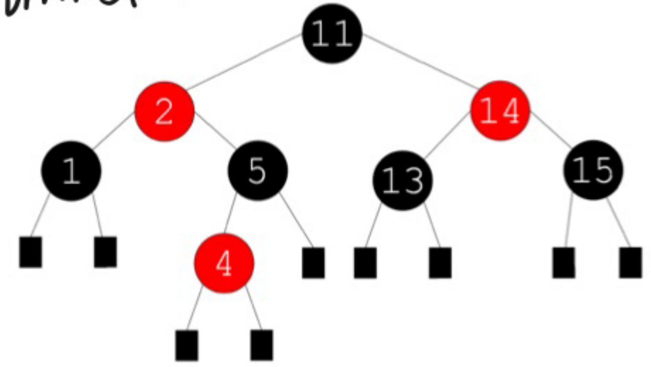
11 must be copied (swapped) node 7 position an erase it current position.



Node 13 must be linked to 14.

This opperation generates a problem with two consecutive red nodes So node 13 must be recolored to Black.

Final form of the tree:

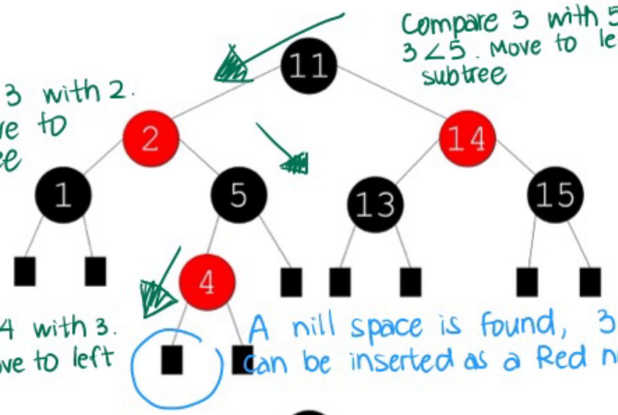


Insert Node 3

Compare 3 with 11 3<11.
Move to left subtree

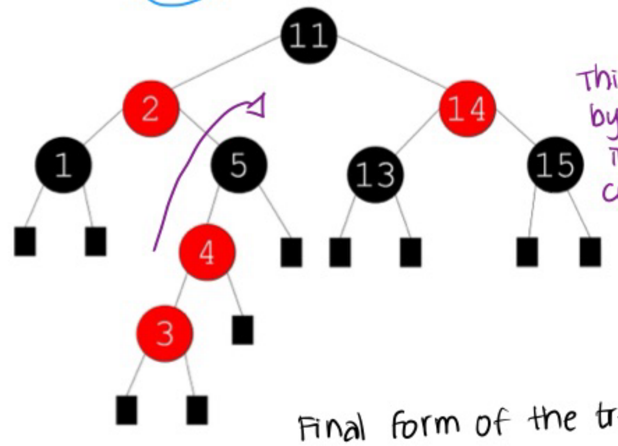
Compare 3 with 5. 3<5. Move to left subtree

Compare 3 with 2. 3≥2. Move to right subtree



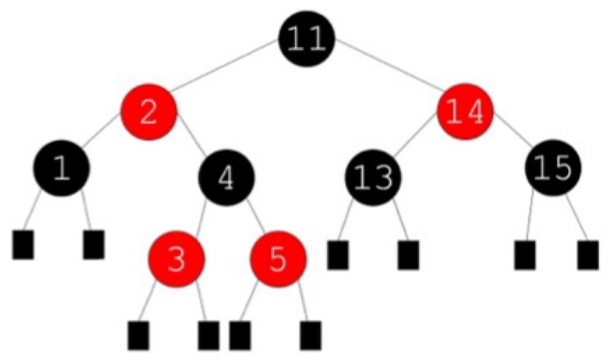
Compare 4 with 3. 3<4. Move to left subtree

A nill space is found, 3 can be inserted as a Red node



This problem is fixed by rotating to the right, in order to get the correct sequence of colors.

Final form of the tree:



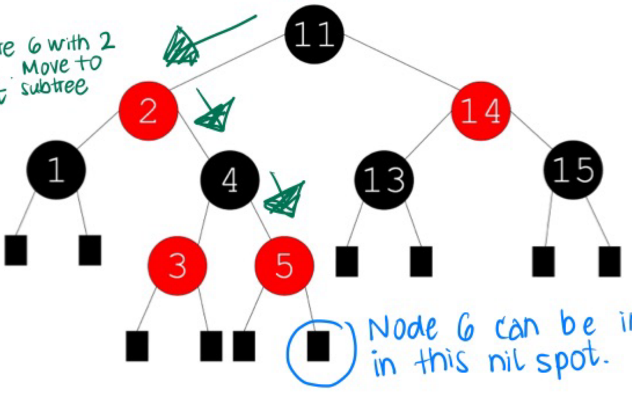
Insert Node 6

Compare 6 with 11. 6<11. Move to left subtree

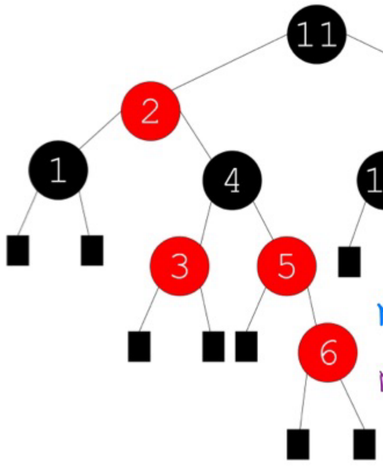
Compare 6 with 2 6≥2 . Move to right subtree

Compare 6 with 4 6≥4 . Move to right subtree

Compare 6 with 5 6≥5 . Move to right s btree

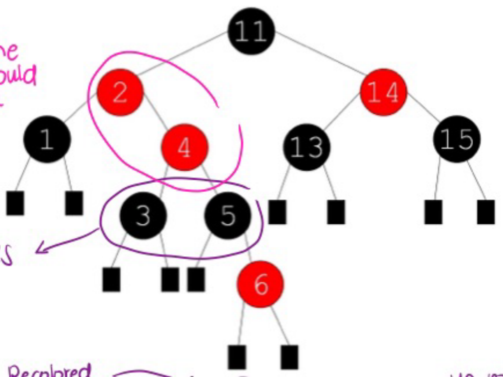


Node 6 can be inserted in this nil spot.



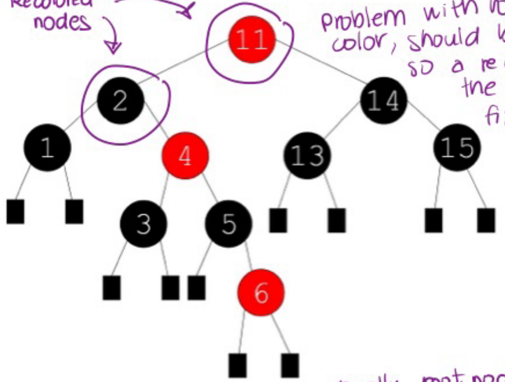
Node 6 is inserted in red color. Now there is a problem. Node 5 is red parent node with a red child, the solution is to change the father and uncle to black, the grandfather to red and so on.

In this situation the father and uncle should be turned to black color, and the grandfather red.



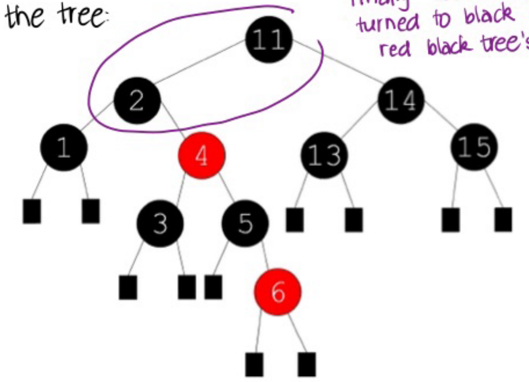
Recolored nodes

Recolored nodes



Problem with root node color, should be black, so a re color of only the root will fix it.

Final form of the tree:

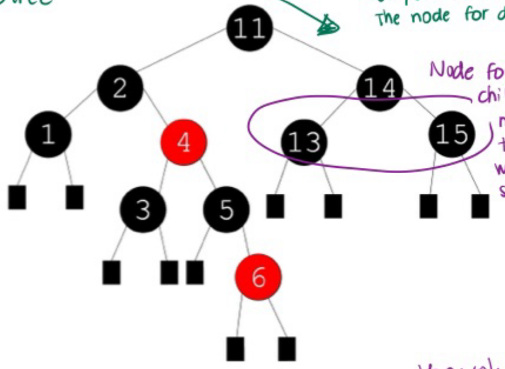


Finally root node is turned to black according to red black tree's rules.

Delete Node 14

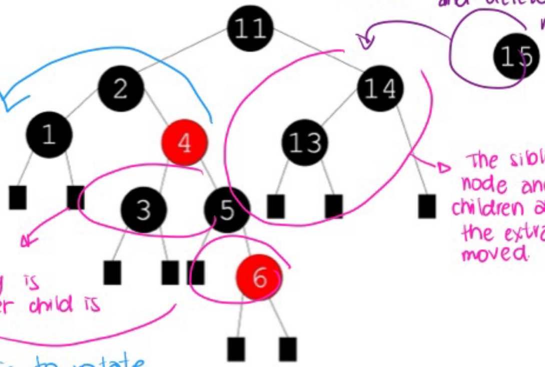
Compare 14 with 11. $14 \geq 11$. Move to right subtree

Compare 14 with 14. $14 == 14$. The node for deletion is found.



Node for deletion has two children. The leftmost node successor in the right subtree, will replace the space of this node

Copy the value of 15 in 14 and delete the previous node 15.



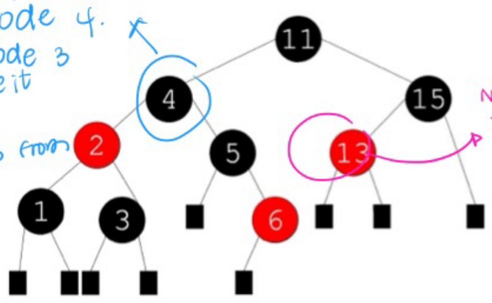
The sibling of the node and both his children are black, the extra black is moved

The node's sibling is black, the closer child is red.

The approach is to rotate left the node 4.

Now link node 3 to 2 and make it right child.

Unlink node 3 from 4.



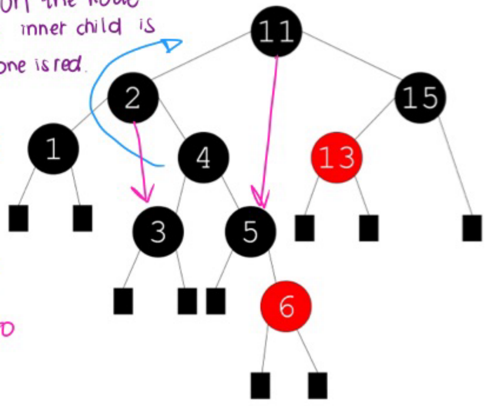
Node 13 is recolored to red.

In this situation the node sibling and his inner child is black, the other one is red.

It's necessarily to rotate in order to change colors.

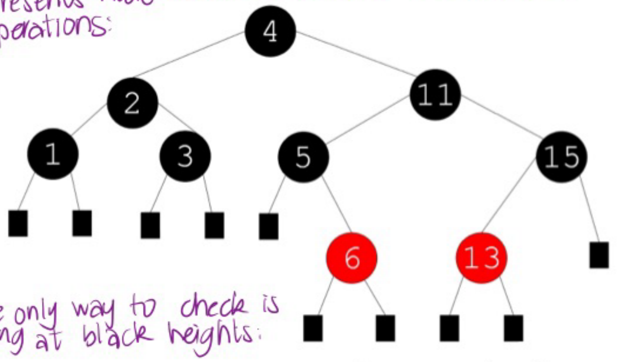
Link node 3 to node 2.

Link node 5 to node 11.



Final form of the tree:

This tree presents more black consecutive node, due to previous operations:



But the only way to check is by looking at black heights:

black heights: The tree is balanced with a height of 3.

$4 \rightarrow 2 \rightarrow 1 \rightarrow \text{NIL}$ $\boxed{3}$

$4 \rightarrow 2 \rightarrow 3 \rightarrow \text{NIL}$ $\boxed{3}$

$4 \rightarrow 11 \rightarrow 5 \rightarrow \text{NIL}$ $\boxed{3}$

$4 \rightarrow 11 \rightarrow 15 \rightarrow 13 \rightarrow \text{NIL}$ $\boxed{3}$

$4 \rightarrow 11 \rightarrow 5 \rightarrow 6 \rightarrow \text{NIL}$ $\boxed{3}$

$4 \rightarrow 11 \rightarrow 5 \rightarrow \text{NIL}$ $\boxed{3}$

3. Consider the following algorithm from the textbook:

```
**OS − RANK** $(T, x)$
1 $r = x.left.size + 1$
2 $y = x$
3 **while** $y \neq T.root$
4     **if** $y = y.p.right$
5         $r = r + y.p.left.size + 1$
6     $y = y.p$
7 **return** $r$
```

Like **OS − SELECT** (x, i) in Lecture 8, this algorithm is also used in a red-black tree with $size$ maintained in nodes as attributes (and textbook calls it an **order-statistic tree**). Here, we define the $rank$ of a node x in T as the output of **OS − RANK** (T, x); in other words, if $x.key$ is the i^{th} smallest key in T, then $x.rank = i$.

Answer the following questions.

a) Given a node x in an n-node **order-statistic tree** T and a natural number i, using procedures **OS − RANK** and/or **OS − SELECT** to create an algorithm that can determine the i^{th} successor of x (in other words, the i^{th} node to visit after x in an inorder-tree-walk of T) in $O(\lg n)$ time. You may assume that the i^{th} successor of x exists while creating this algorithm.

Pseudocodes provided by the textbook:
OS-SELECT: Retrieves the element with a given rank.

```
OS-SELECT(x, i)
1. r = x.left.size + 1
2. if i == r
3.    return x
4. elseif i < r
5.    return OS-SELECT(x.left, i)
6. else
7.    return OS-SELECT(x.right, i - r)
```

Time complexity in O(log n) time.

OS-RANK: Determines the rank of an element.

```
OS-RANK(T, x)
1. r = x.left.size + 1
2. y = x
3. while y ≠ T.root
4.    if y == y.p.right
5.       r = r + y.p.left.size + 1
6.    y = y.p
7. return r
```

Time complexity in O(log n) time.

To find the i^{th} successor of a node x in an in order-tree-walk of T in O(logn) time, it can be done by using a recursive approach for computing the rank with OS-RANK and then using OS-SELECT with the rank plus i.

```
OS-ITH-SUCCESSOR(T, x, i)
1. Compute r = OS-RANK(T, x).             //Compute the rank r of the node x using OS-RANK
2. Return OS-SELECT(T.root, r + i)        //Use OS-SELECT to find the node with the computed rank.
```

This algorithm runs in O(logn) time because both OS-RANK and OS-SELECT run in O(logn) time.

b) Can we maintain $rank$ in nodes of a red-black tree as attributes without affecting the $O(\lg n)$ performance of insertion and deletion operations? Justify your answer.

Yes, we can keep the rank of the nodes in a red-black tree as traits without changing how fast insertion and deletion work, which is O(logn). This is because the size property of subtrees makes it easy to figure out and keep track of the rank. Because of these actions:

When a node is added, we go through the path from the root to the new node and change the size of each node along the way. It takes O(logn) time to do this traversal. The rank is calculated from the size traits, so keeping track of the rank information doesn't take any extra asymptotic time.

When a node is removed, we go back through the path from the root to the node that is being deleted and change the size of each node along the way. It takes O(logn) time to do this too.

There are always the same number of nodes involved in the rotations needed to keep the red-black properties, and it takes O(1) time to change the size traits for these nodes.

As a result, keeping the rank of each node as a trait that comes from the size of its subtrees does not make adding and removing items in a red-black tree more difficult. The general performance stays the same (logn).

4. Suppose we wish not only to increment value in a k-digit binary counter A, but also to decrement the value. Counting the cost of each flip as 1, can you implement $\text{Increment}(A)$ and $\text{Decrement}(A)$ such that any sequence of m $\text{Increment}(A)$ and $\text{Decrement}(A)$ operations cost $O(m)$? In other words, each operation in the sequence has amortized cost $\Theta(1)$, which is a constant and independent from k. If you can, show how; if you think it is impossible, show why.

Yes, it is possible to implement increment(A) and decrement(A) operations in such a way that each sequence of m increment and decrement operations is O(m). This implies that each operation in the series has an amortized cost of θ(1), which is constant and independent of k.

The main idea is to apply a two's complement binary counter. A two's complement binary counter's increment process is simple and similar to that of a standard binary counter. However, the decrement procedure is a little more complex. It requires flipping all the bits of the number (one's complement) and then adding 1 to the result (two's complement). In binary, the two's complement of a number signifies its negative.

In the pseudocode, A is the binary counter represented as an array of bits, with A[i] being the i-th bit of A. The increment(A) function scans the bits of A from least significant to most significant, flipping each bit from 1 to 0 until it comes across a bit that is 0, which it converts to 1. The decrement(A) function does the inverse: it flips each bit from 0 to 1 until it finds a bit that is 1, then flips it to 0.

INCREMENT(A):
```
    1.  i = 0
    2.  while i < A.length and A[i] == 1:
    3.  A[i] = 0
    4.  i = i + 1
    5.  if i < A.length:
    6.  A[i] = 1
```
Time complexity analysis:
Line 1 presents a constant time operation. Time Complexity: O(1)
Line 3 - 4 The loop runs k times Time Complexity: O(k)
Line 5 presents a constant time operation. Time Complexity: O(1)
Line 6 presents a constant time operation. Time Complexity: O(1)

DECREMENT(A):
```
    1.  i = 0
    2.  while i < A.length and A[i] == 0:
    3.  A[i] = 1
    4.  i = i + 1
    5.  if i < A.length:
    6.  A[i] = 0
```

Time complexity analysis:
Line 1 presents a constant time operation. Time Complexity: O(1)
Line 3 - 4 The loop runs k times Time Complexity: O(k)
Line 5 presents a constant time operation. Time Complexity: O(1)
Line 6 presents a constant time operation. Time Complexity: O(1)

The increment(A) and decrement(A) functions each contain a loop that runs in O(k) time in the worst-case scenario, where k is the number of bits in A. However, across a sequence of m increment and decrement operations, each bit is only flipped twice: once from 0 to 1 and once from 1 to 0.

As a result, the total cost of m operations is O(m), and the amortized cost per operation is O(m)/m = O(1), a constant that is not affected by k. This fits the requirements of the problem.

5. **[Think about this question]** Show that no matter what node we start at in a height-h binary search tree, m successive calls to $\textbf{TREE} - \textbf{SUCCESSOR}$ take $O(m + h)$ time.

   Only think about question 5. You don't need to submit proof for question 5 since it is not easy to give a proof that is both convincing and elegant for this statement; so instead, simply consider this statement in all possible situations and convince yourself it is true.

Pseudocode for TREE-SUCCESSOR:

```
TREE-SUCCESSOR(x)
    1.  if x.right ≠ NIL
    2.  return TREE-MINIMUM(x.right)
    3.  y = x.p
    4.  while y ≠ NIL and x == y.right
    5.  x = y
    6.  y = y.p
    7.  return y
```

Looking at the textbook pseudocode and thinking that every node has a parent pointer; if there isn't a parent pointer, it might take longer to find the parent of a node. But if the BST is balanced, the height h grows exponentially with the number of tree nodes, which makes the time complexity efficient. When the if condition says that x.right ≠ NIL, it returns the function TREE-MINIMUM(x.right), which finds the lowest node in the right subtree. This process could take O(h) time in the worst case because it needs to go through the whole tree's height and down the right subtree.

The while condition of y ≠ NIL and x == y.right, on the other hand, means that the function goes up the tree to find the next node in the tree. It does this by going from the current node to its parent and then to the left child of its parent. It could take O(h) time for this process to complete because it must go up the tree in the worst case.

In the worst case, this means that a single call to TREE-SUCCESSOR takes O(h) time. To move to the right child or up to the parent, we make m calls to TREE-SUCCESSOR. Each time, we start at a different node and follow m edges in the tree. There is a total of O(m) edges because we only follow each one once. Because of this, calling TREE-SUCCESSOR again and again takes O(m) time. Therefore, for m calls, the first call costs O(h), and the next m-1 calls will usually involve moving to nodes next to each other, which can be done in O(1) time per call after the initial setup.

As a result, the total amount of time needed for m calls in a row can be written as O(h)+O(m)=O(m+h).