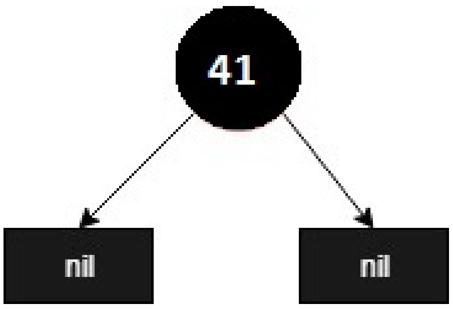
Algorithms Homework 4

Liam Dillingham

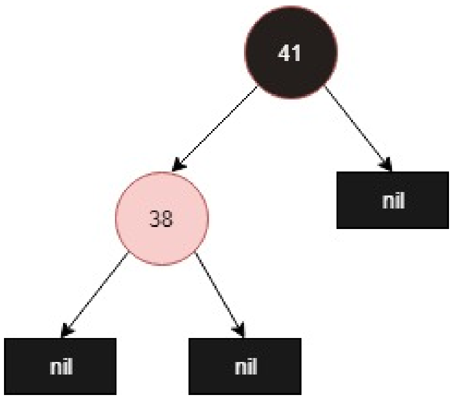
October 26, 2018

1 Question 13.3-2

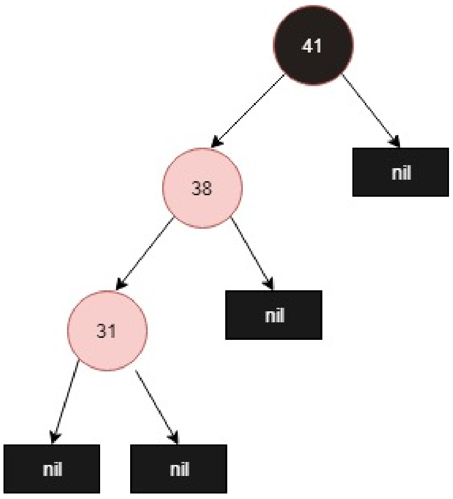
Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.



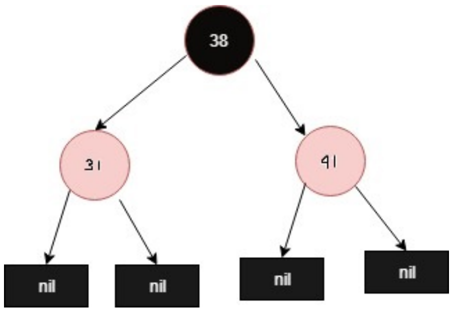
(a) Inserting 41



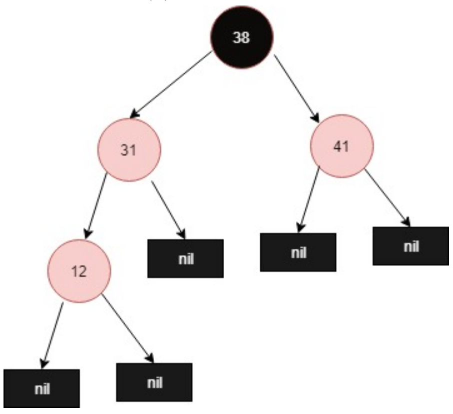
(b) Inserting 38



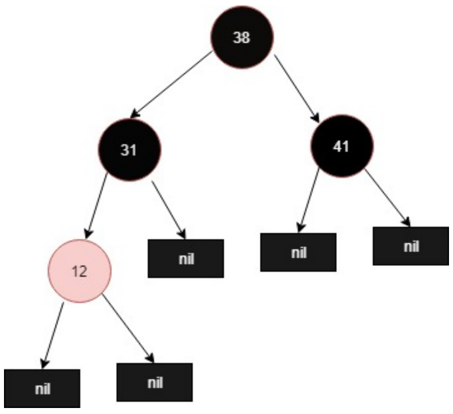
(c) Inserting 31



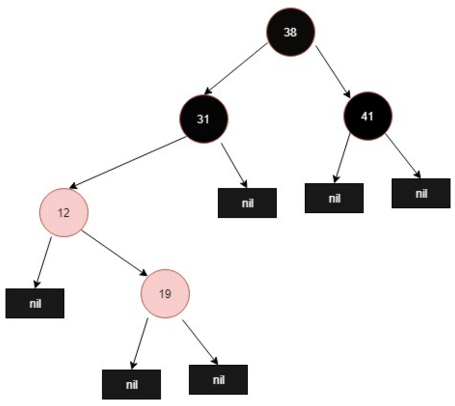
(d) Re-balancing



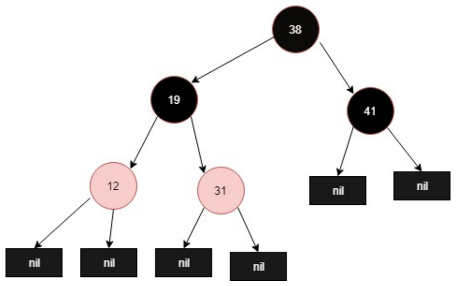
(e) Inserting 12



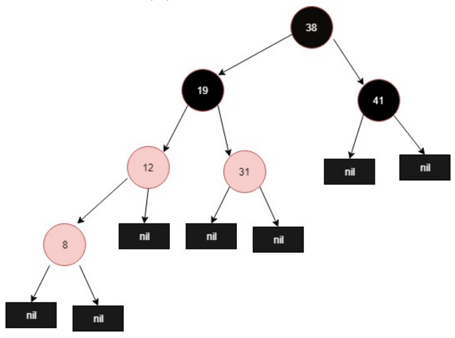
(f) Re-balancing



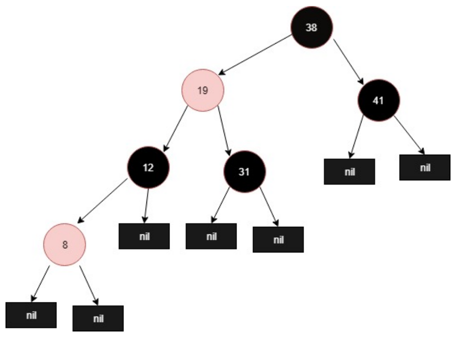
(a) Inserting 12



(b) Re-balancing



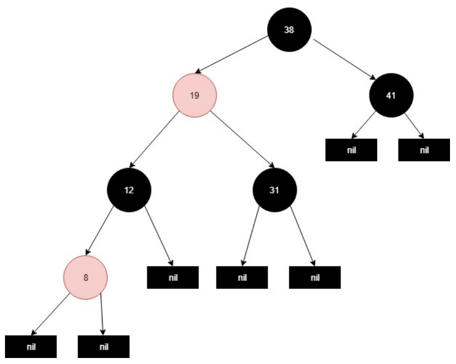
(c) Inserting 12



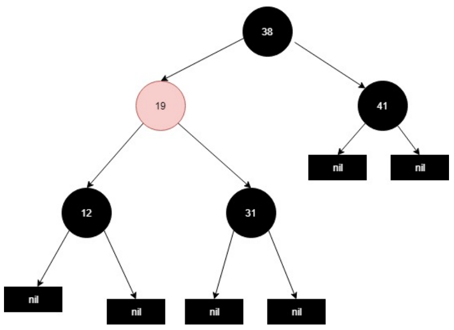
(d) Re-balancing

2 Question 13.4-3

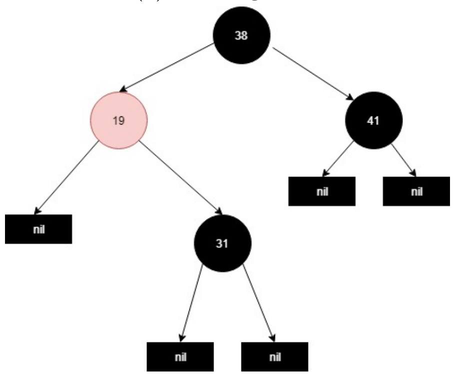
In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.



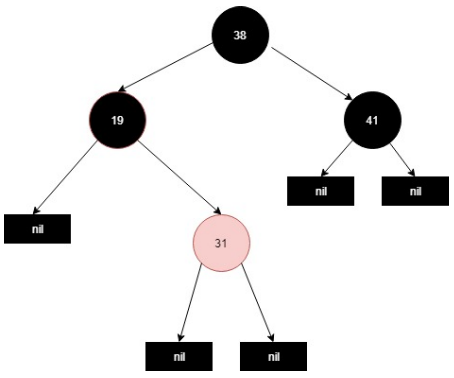
(a) Starting state



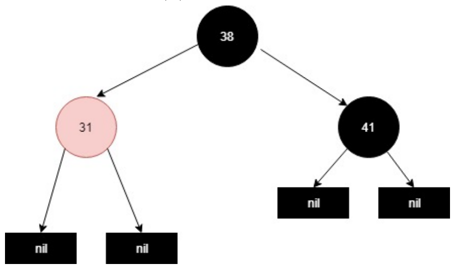
(b) Deleting 8



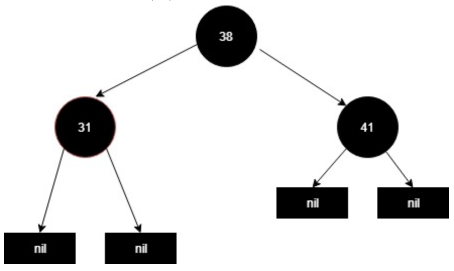
(c) Delete 12



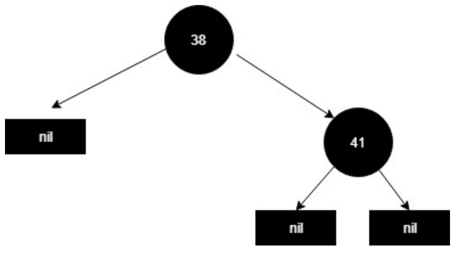
(d) Re-coloring



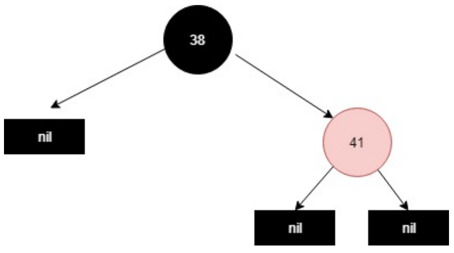
(e) Delete 19



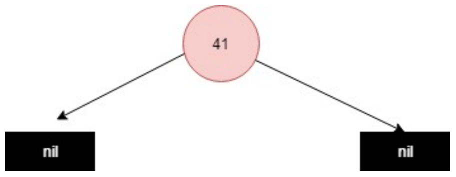
(f) Re-color



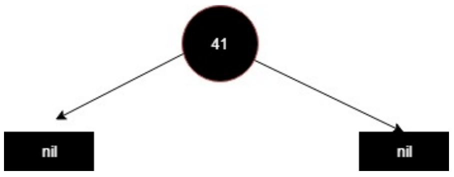
(g) Delete 31



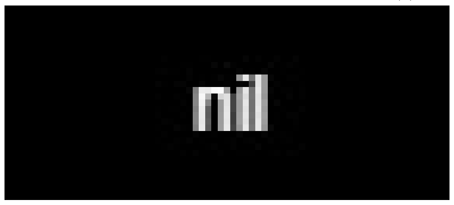
(h) Re-color



(i) Delete 38



(j) Re-color



(k) Delete 41

3 Question 14.1-3

Write a non-recursive version of OS-SELECT.

```
OS-SELECT(x, i)
   while x != nil
      r = x.left.size + 1
      if i == r
         return x
      elseif i < r
         x = x.left
      else
         x = x.right
         i = i - r
```

with the initial call being OS-SELECT(T.root, i).
Note that the key change here that causes this function to differ from its recursive counterpart, is that when we meet one of the three conditions (the third being the 'else' clause) we do a reassignment instead of a function call. If we think about it, a recursive function call is sort of a re-assignment, except we are changing the value of the parameters in the top of a new function call. Here we are changing the parameters while inside the same call.

4 Question 14.1-5

Given an element x in an n-node order-statistic tree and a natural number i, how can we determine the ith successor of x in linear order of the tree in $\mathcal{O}(\lg n)$ time?

---

determining the ith successor of a node x is similar to determining ith smallest element, except that we pretend that x is the smallest node in the tree. So determining the ith successor in x is essentially taking the rank of x and adding it to the ith smallest element greater than x. So for example, if the rank of x is 4, then finding the 3rd successor of x is equivalent to OS-SELECT(T.root, x.rank + 3) = OS-SELECT(T.root, 7). or to word this generally, we can use the pseudocode function from the book OS-RANK(T,x) and plug it into our OS-SELECT function.

```
OS-SELECT(T.root, OS-RANK(T, x) + i)
```

Note that OS-RANK runs in time $\mathcal{O}(\lg n)$. In addition, OS-SELECT runs in time $\mathcal{O}(\lg n)$. Thus the run time of determining the ith successor is $\mathcal{O}(\lg n)$ time

5 Question 14.1-6

Observe that whenever we reference the size attribute of a node in either OS-SELECT or OS-RANK, we use it only to compute a rank. Accordingly, suppose we store in each node its rank in the subree of which it is the root. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations).

Once the new node is inserted, we will need to first call PREDECESSOR on our newly inserted node, and set the rank of our new as PREDECESSOR.rank + 1. The we will need to call the SUCCESSOR function for the new node, and on each subsequent node, and increase the rank of each node by 1.

Handling the rotations resulting from deletions and insertions is simple; we can simply stick this new logic at the end of the functions. The modified code for deletions and insertions is shown below:

```
RB-Insert(T, z)
   y = T.nil
   x = T.root

   while x != T.nil
      y = x
      if z.key < x.key
         x = x.left
      else x = x.right
   z.p = y
   if y == T.nil
      T.root = z
   elseif z.key < y.key
      y.left = z
   else y.right = z
   z.left = T.nil
   z.right = T.nil
   z.color = RED
   RB-INSERT-FIXUP(T,z)

   // Beginning new code
   // Get rank of predecessor and increase the rank of the new node by 1
   z.rank = PREDECESSOR(T, z).rank + 1

   w = SUCCESSOR(T, z) // Get z's successor
   while w != nil // While there is a successor
      w.rank = w.rank + 1 // increase its rank
      w = SUCCESSOR(T, w) // Get the next successor
```

Deletion of a node involves some early book-keeping, but since the rank of the nodes is in linear order, so long as we bookmark the location in the rank list, we can carry on like normal

```
RB-DELETE(T,z)
   // Save z's successor
   w = SUCCESSOR(T, z)

   y = z
   y-original-color = y.color
   if z.left == T.nil
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
   elseif z.right == T.nil
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
   else
      y = TREE-MINIMUM(z.right)
      y-original-color = y.color
      x = y.right
      if y.p == z
         x.p = y
      else
         RB-TRANSPLANT(T, y, y.right)
         y.right = z.right
         y.right.p = y
      RB-TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
      y.color = z.color
   if y-original-color == BLACK
      RB-DELETE-FIXUP(T, x)

   while w != nil
      w.rank = w.rank - 1
      w = SUCCESSOR(T, w)
```

6 Question 14.1-7

Show how to use an order-statistic tree to count the number of inversions (see problem 2-4) in an array of size n in time $\mathcal{O}(n \lg n)$

Problem 2-4 states:
Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.

To count the total number of inversion for an array is simple. For each j, count the number of elements greater than $A[j]$ that precede it. Do this for all $j \in 1..n$, and that is the total number of inversions. Note that by using an order-statistic tree, we can easily obtain the rank of any element $A[j]$, essentially "sorting" it. Note also that if an element preceding j is less than j, then it is not an inversion. That is, the rank of j is its index minus the number of inversions. $rank(j) = j - NumInversion(j)$.

If we re-arrange this equation, we can get $NumInversion(j) = j - rank(j)$. So, by inserting j into the tree, we can find its rank. then we can subtract the rank from j, and calculate the number of inversions. Note that both insertion and rank calculation take $\mathcal{O}(\lg n)$ time each. since we have to do this for n elements (from the array), then to calculate the total number of inversions using an order-statistic tree, it will take $\mathcal{O}(n \lg n)$ time.