1. Algorithm Description
    - Like the Counting Sort Algorithm, the algorithm will take in an input array filled with n integers from 0 to k, and then use a second array (C) to keep track of the number of elements in the array. Finally the sorted array will be transferred to Array B. C now contains the number of elements that are less or equal than the index i. Therefore when queried to see how many integers fall between [a..b] you need to calculate C[b]-C[a-1].

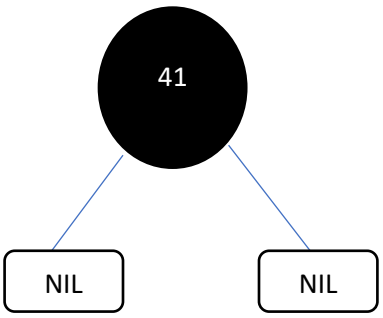
    Algorithm Correctness

    - Since we know the counting sort algorithm terminates, this algorithm will too since no new loops were added. The algorithm also produces a correct output since the values in each index in the C array will be the number of elements that are less than or equal to that index value. Therefore, when you take C[b] you have all the values less than or equal to b and when you subtract C[a-1] you are getting rid of all the values less than a in the set, this gives us the values in the range [a..b].

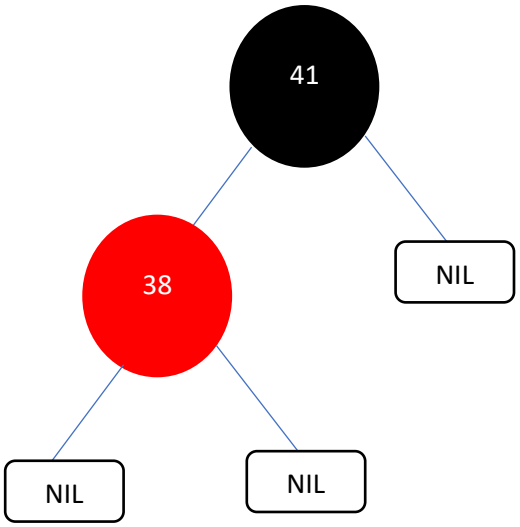
    Complexity of the Algorithm

    The counting sort part of the algorithm is of O(n) time and the part of the algorithm where we need to find the integers that fall in the range [a..b] is of O(1) time, since there is only 1 constant instruction.

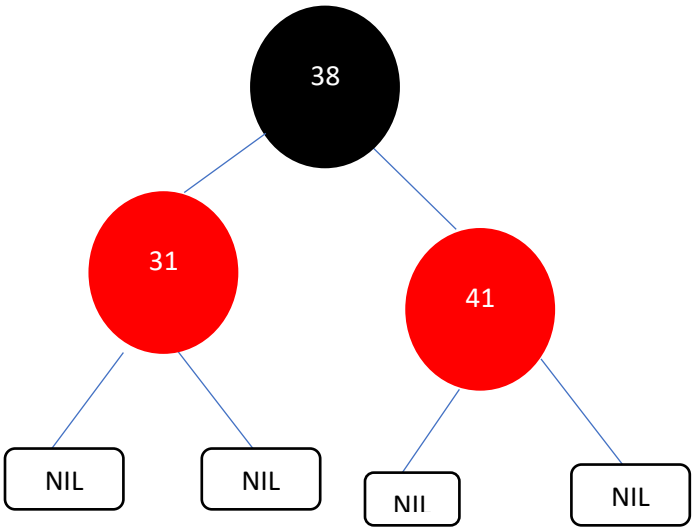
2. Insert 41



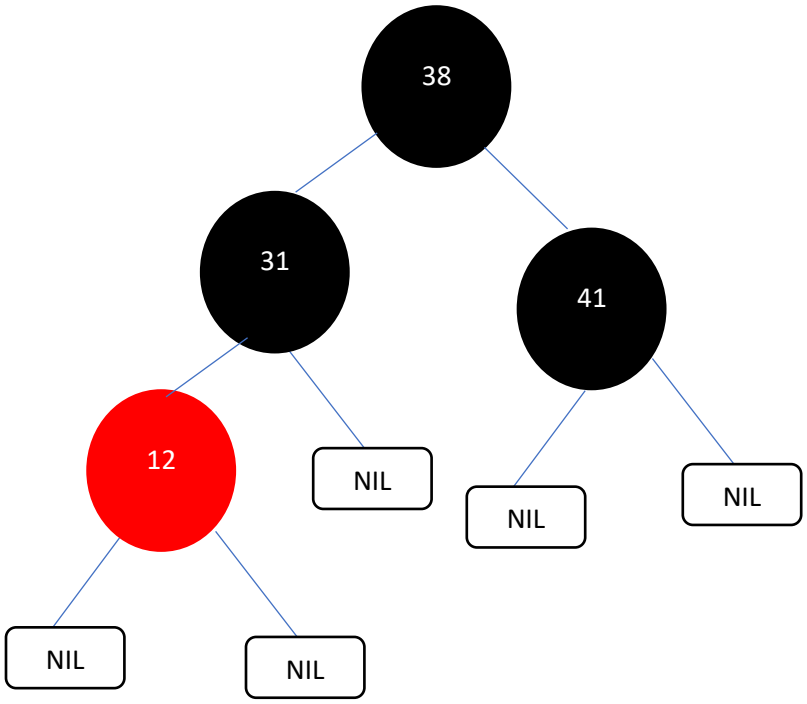
Insert 38



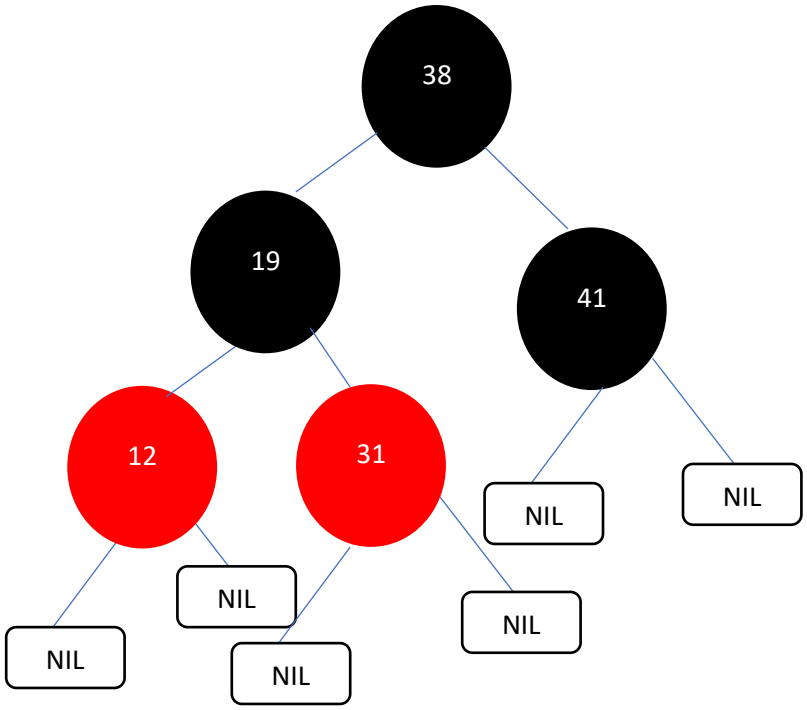
Insert 31



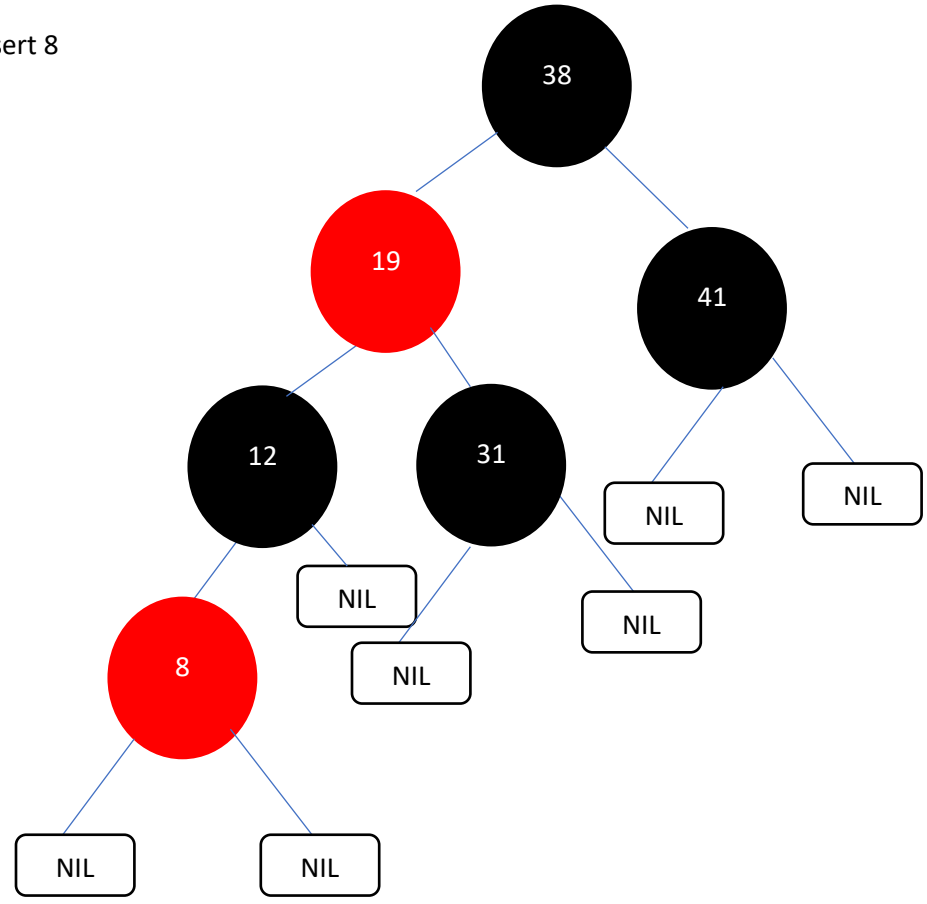
Insert 12



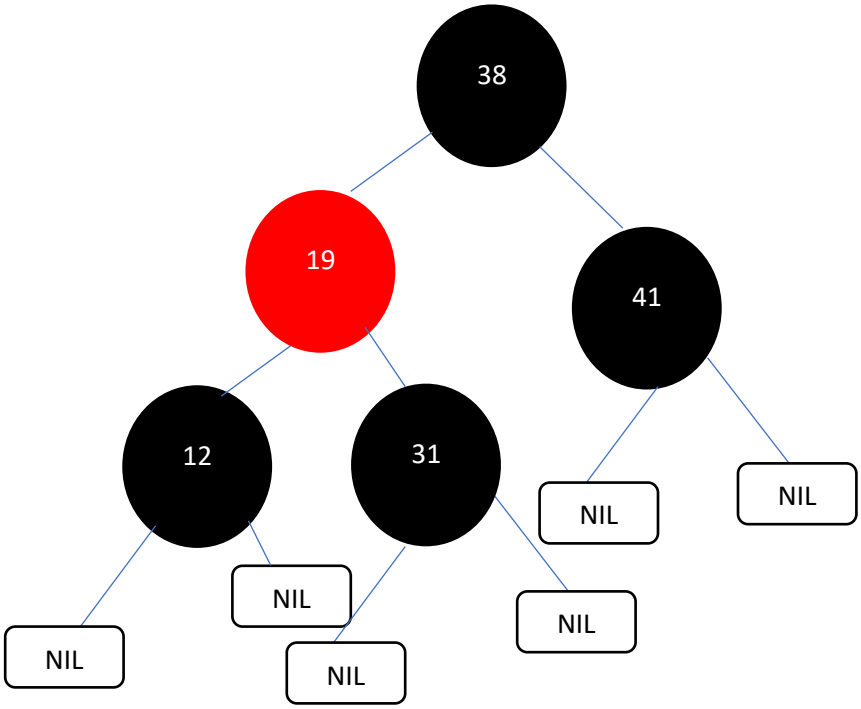
Insert 19



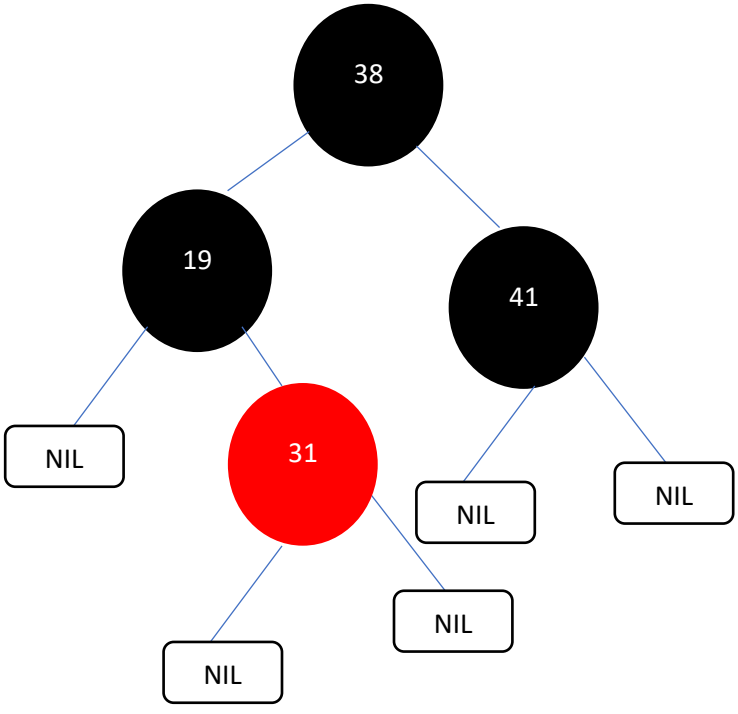
Insert 8



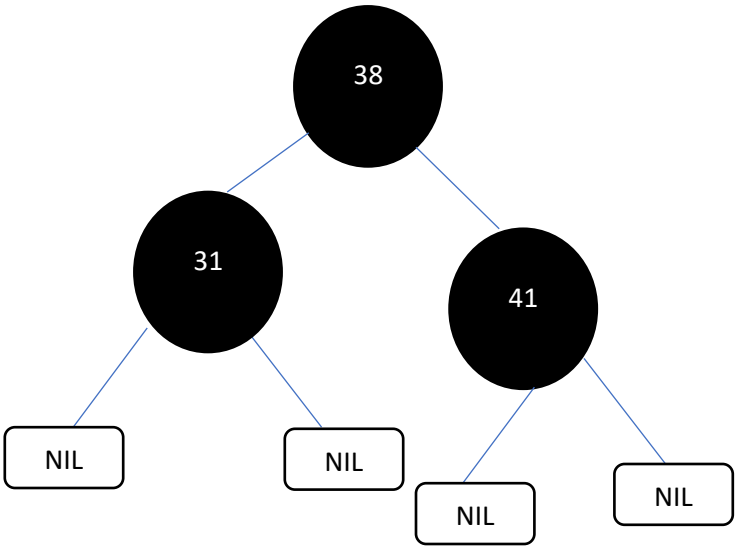
3. Delete 8



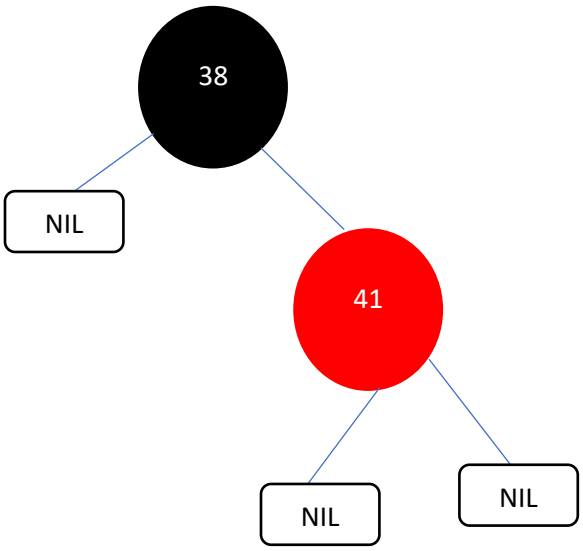
Delete 12



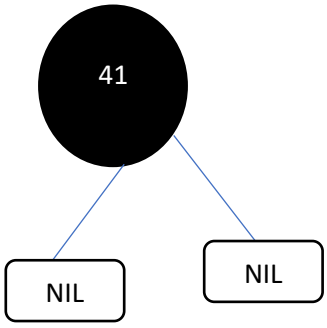
Delete 19



Delete 31



Delete 38



Delete 41

4. No. If we look at the answers to questions 2 and 3 in this assignment, we can see that when the number 8 was inserted, the colors of nodes 12 and 31 were red, however, when 8 was inserted the colors changed to black. Then, when 8 was deleted, the colors of 12 and 31 still remained black, meaning the tree isn't the same as the initial red/black tree.

5. If we let T(h) be the minimum size of an AVL tree of height h. We know that at each node the height will be the max height of it's children's heights +1. If we assume that the smallest child has the smallest possible height that is allowed by the range of the tree, so one less than the larger child, we get the following equation for the height of the tree:

   T(h)≥ T(h-1)+T(h-2)+1 -> Notice that this is just like the Fibonacci equation, but we are also adding 1 for the root at the end

   Base Case: T(0) = 0 and T(1)=1 -> this is just like the Fibonacci sequence, therefore

   T(h) = $\left\lfloor \frac{\phi^h}{\sqrt{5}} + \frac{1}{2} \right\rfloor \leq n$

* This equation came from equation 3.25 on pg 60 of the textbook. $\phi \; is \; the \; golden \; ratio$.

Rearrange for h:

$$\frac{\phi^h}{\sqrt{5}} \leq n$$

$$\phi^h \leq \sqrt{5}n$$

$$h \; \log_\phi \phi \leq \log_\phi(\sqrt{5}) + \log_\phi n$$

$$h \; \leq \frac{\log_\phi(\sqrt{5}) + \log_\phi n}{\log_\phi \phi}$$

$$\in O(\lg(n))$$

□

6. Algorithm Descriptions
   This algorithm will be based off of the heap sort algorithm. First a heap will be created for the first k elements in the array. Then for each element compare it with the kth element, if it is smaller, swap that value with the root. Keep going until all of the elements are analyzed. The heap will now have the k smallest elements in order.

Algorithm Correctness
The algorithm will terminate since each for loop will end after max n iterations. The algorithm will also produces the correct output since it follows the same algorithm as heap sort except it's only analyzing the first k elements, in theory this will be faster than a normal heap sort with n elements.

Complexity of Algorithm
Creating a heap of the first k elements will take O(k) time. Sorting the elements will take O((n-k)*logk). Therefore the total time complexity is O(k+(n-k)logk +klogk) = O(n+klogk).

7. Proof using Induction

   Let R(n) be the rank at the nth node.

   Base Case:
   R(1) = lg 1 = 0

   Induction hypothesis.
   Assume R(n) = lg(n)

   Inductive Proof for R(n+1) = lg(n+1)

   Assume we have 2 disjoint sets a and b and n+1 nodes. The sets a and b <=n. If we perform the UNION operation on the sets then:

   Case 1: Unequal ranks

   R(a)<=lg(a)
   R(b)<=lg(b)

   Case 2: Equal Ranks -- we choose one of the roots as the parent and increase the rank by 1
   R(a) = lg(a) + 1<= $\left\lfloor \frac{log(n+1)}{2} \right\rfloor + 1 = \lfloor log(n+1) \rfloor$

□

8. The rank of each node is $\lfloor lg\ n \rfloor$ so we can represent them using Θ(lg(lg(n)) bits. One byte should be enough to store the rank since Θ(lg(lg(n)) can take in up to $1x10^{78}$ nodes to reach 1 byte.