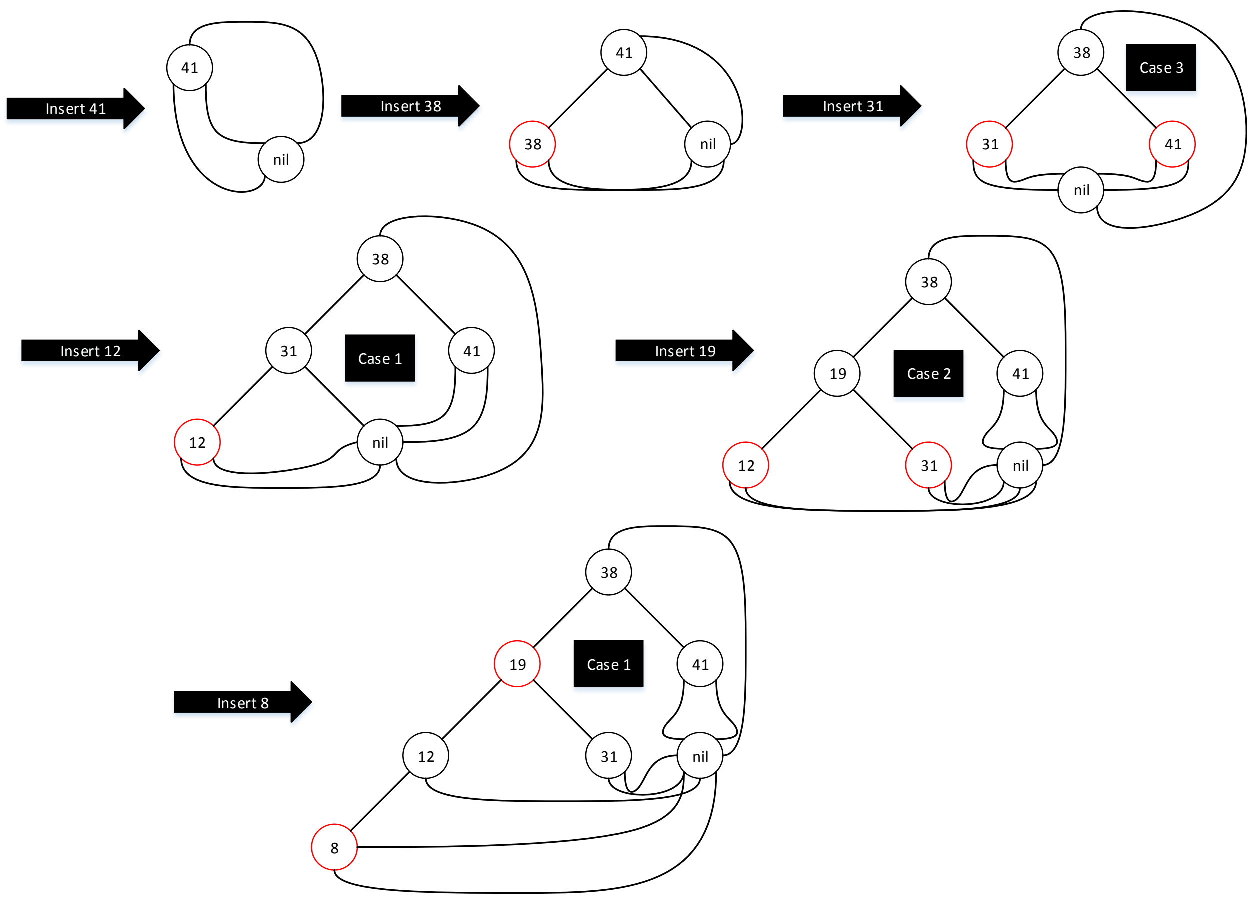
Assignment #2

Student #: [redacted]

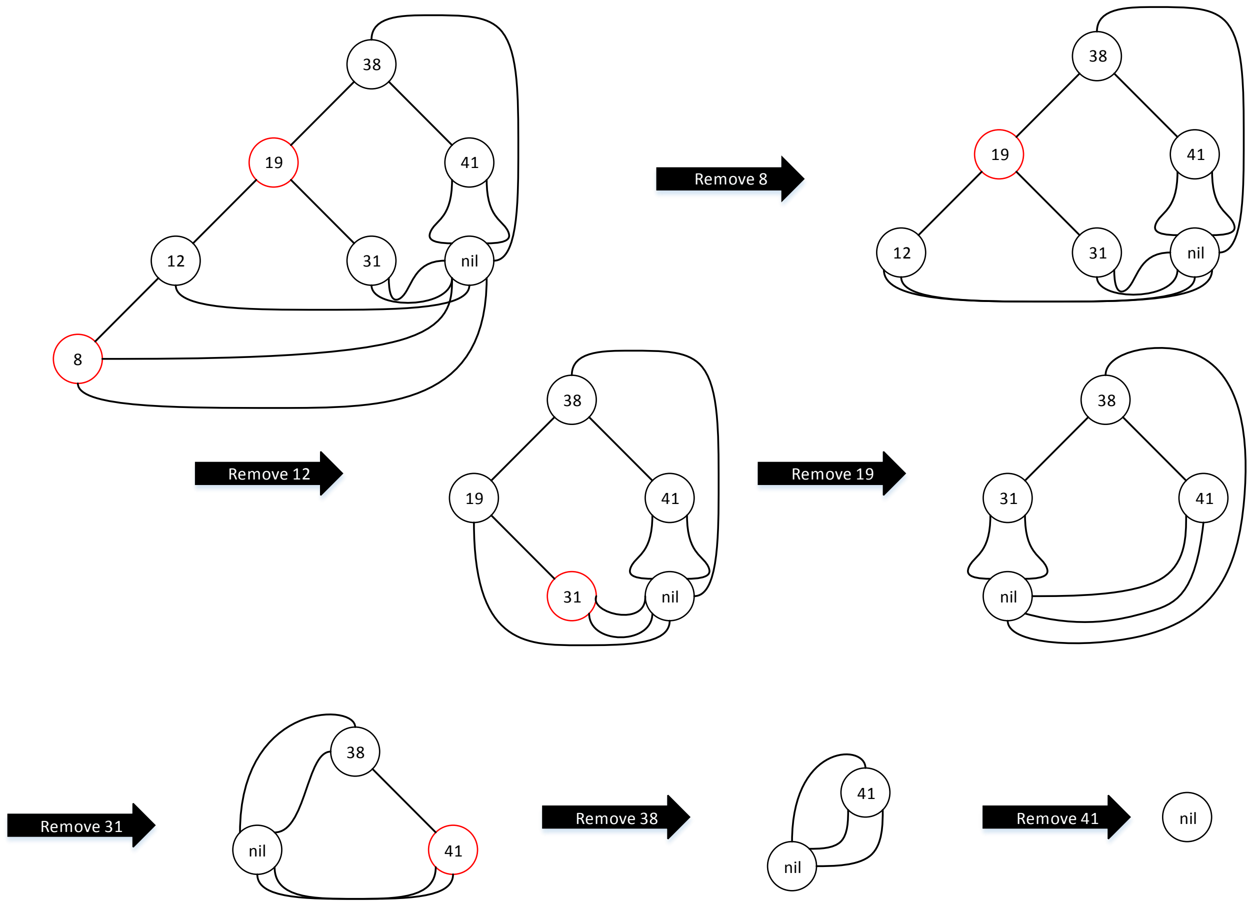
Name: Zaid Albirawi

UWO email: zalbiraw@uwo.ca

1. 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.



2. Show the red-black tree that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41 from the red-black tree resulted from question 1.



3. An AVL tree is binary search tree that is height balanced: for each node x, the heights of the left and right sub-trees of x differ by a most 1. To implement an AVL tree, we maintain an extra attribute in each node: x.h is the height of node x. As for any other binary search tree T, we assume that T.root points to the root node.

a. Prove that an AVL tree with n nodes has height $O(\log n)$. (Hint: Prove that an AVL tree of height h has at least F_h nodes, where F_h is the h^{th} Fibonacci number.)

First, we recall that the heights of the right and left sub-trees of the AVL tree differ by 1, at most. We also recall the tree identity, $n(T) = n(T_1) + n(T_2) + 1$, where $T1$ and $T2$ are T's right and left sub-trees and n is the number of nodes in each tree. Therefore, a tree with maximum height and a minimal number of nodes, must have a maximum height difference between its sub-trees, 1. Hence, by applying the tree identity to our findings, we conclude that $(T_h) = n(T_{h-1}) + n(T_{h-2}) + 1$.

Proof by induction:

Base case:

$n(0) = F_0 = 0$
$n(1) = F_1 = 1$

Induction case: $n(T_h) \geq F_h$

$n(T_{h+1}) \geq F_{h+1}$
$n(T_h) + n(T_{h-1}) \geq F_h + F_{h-1}$

Since $(T_h) \geq n(T_{h-1})$and $F_h \geq F_{h-1}$ then,

$(T_h) \geq F_h$

Proven by induction.

Now that we have proven that an AVL tree of height h has at least F_h nodes. We can prove that the tree has a height of $(\log n)$.

Since $n(T_h) \geq F_h$, and $F_h \geq 1.6^h$ then,

$n(T_h) \geq F_h \geq 1.6^h$

$n(T_h) \geq 1.6^h$

$h = \log_{1.6} n(T_h)$

Therefore, h is $\log n(T_h)$

4. Design an efficient data structure using (modified) red-black trees for an abstract data type that supports the following operations.

Insert(x): insert the key x into the data structure if it is not already there.
Delete(x): delete the key x from the data structure if it is there.
Find_Smallest (k): find the kth smallest key in the data structure.

What are the time complexities of these operations?

The data structure that will be used to solve this problem is a modified red-black tree structure. This structure will differ from a normal red-black tree in its node class. The new node of the red-black tree will contain an extra integer attribute that will hold the number of nodes that each sub-tree has. Also, to keep track of the number of nodes in each sub-tree, we modify the insert and delete algorithms to use an algorithm called, tracker, that will take in the parent of the node that was inserted or removed, and preforms a sequence of recursive calls that will increment or decrement the number of nodes attribute for each sub-tree root. The algorithms for, insert, delete, insertBalance, deleteBalance, find and tracker, follow.

Algorithm: find (key)
In: A key object that is supposedly in the tree.
Out: The node to where the key is or supposed to be placed.
find(key)
node ← root
while node != NIL **do**
  if key(node)>key **then** node ← left(node)
  else if key(node)<key **then** node ← right(node)
  else return node
return NIL

Algorithm: insert (key)
In: A key object that will be inserted into the tree.
Out: A boolean to wither the key object was inserted or not.
insert(key)
if root = NIL **then**
  root ← node(key, black)
  return true
else if find(key)= NIL **then**
  find(key) ← node(key, red)
  InsertBalance(node)
  tracker(P(node), insert)
  return true
else return false

Algorithm: delete (key)
In: A key object that is to be deleted from the tree.
Out: A boolean to whether the node containing the key has been deleted or not.

```
delete(key)
if find(key)=NIL then return false
else
        node ← find(key)
        if node = red & right(node)=NIL & left(node)=NIL then
                tracker(P(node), delete)
                node ← NIL
                return true
        else if node = black & right(node)!=NIL & left(node)=NIL then
                tracker(P(node), delete)
                right(node) ← black
                P(right(node)) ← P(node)
                node ← NIL
                return true
        else if node = black & right(node)=NIL & left(node)!=NIL then
                tracker(P(node), delete)
                left(node) ← black
                P(left(node)) ← P(node)
                node ← NIL
                return true
        else if node = black & right(node)!=NIL & left(node)!=NIL then
                tracker(P(node), delete)
                node ← NIL
                deleteBalance(node)
                return true
```

Algorithm: insertBalance (node)
In: The node that was recently inserted into the tree
insertBalance(node)
if node = root **then** root ← black
else if P(node) = red **then**
    if P(P(node))=black **&** Sibling(P(node)) = red **then**
        P(P(node)) ← red
        P(node) ← black
        Sibling(P(node)) ← black
        InsertBalance (P(node))
    if left(P(node))=true **&** isRight(node)=true **then** leftRotation(node)
    else if right(P(node))=true **&** isLeft(node)=true **then** rightRotation(node)
    if P(P(node))=black **&** Sibling(P(node)) = black **then**
        P(P(node)) ← red
        P(node) ← black
        if left(P(node))=true **then** rightRotation(P(node))
        else leftRotation(P(node)) root ← black
return insertBalance (P(node))

Algorithm: deleteBalance(node)
In: A key object that is to be deleted from the tree.
deleteBalance(node)
sibling ← Sibling(node)
if node = root **then** root ← black
else if sibling=red **then**
    P(node) ← red
    P(sibling) ← black
    if isLeft(node)=true **then** leftRightRotation(node)
    else rightLeftRotation(node)
if sibling=black **&** left(sibling)=black **&** right(sibling)=black **then**
    sibling ← red
    if P(node)=red **then** P(node) ← black
    else if P(node)!=root **then** deleteBalance(P(node))
if sibling=black **&** left(sibling)=red **&** right(sibling)=black **&** isRight(sibling)=true **then**
    sibling ← red
    left(sibling) ← black
    rightLeftRotation(node)
else if sibling=black **&** left(sibling)=black **&** right(sibling)=black **&** isLeft(sibling)=true **then**
    sibling ← red
    right(sibling) ← black
    leftRightRotation(node)
if sibling=black **&** right(sibling)=red **&** isRight(sibling)=true **then**
    right(sibling) ← black
    leftRotation(sibling)
else if sibling=black **&** left(sibling)=red **&** isLeft(sibling)=true **then**

left(sibling) ← black
rightRotation(sibling)
return deleteBalance(P(node))

Algorithm: tracker (node, str)
In: the parent of the node that has been inserted or deleted, a string specifying if it's an insert or a delete operation.
tracker (node, str)
if String = delete **then** i ← -1
else i ← 1
if node = root **then** n (node) ← #nodes(node) + i
else
n (node) ← #nodes(node) + i
tracker(P(node), str)

Furthermore, now that each sub-tree has an attribute for the number of nodes it contains, we can introduce the find_smallest (k). This algorithm will determine where the kth smallest element is located in the tree based on the number of nodes in each sub-tree. Hence, since we are looking for the smallest element then we always start our search with the left sub-tree but if k > than the number of nodes in the left sub-tree then we go into the right sub-tree and decrement the value of k by the size of the left sub-tree, algorithm.

Algorithm: find_smallest (k)
In: a number that will determine the kth smallest element.
Out: The kth smallest element in the tree.
find_smallest(k)
node ← root
while node != NIL **do**
if (n(left(node))>k) **then**
node ← left(node)
else if (n(left(node))<k) **then**
node ← right(node)
k ← k – n(left(node))
else return node
return NIL

The time complexities for the operations insert(key) and delete(key) are $2\log n + c$ because they use the find(key) operation to find the location of the key. The find operation takes $(\log n)$ because of the height of the tree. Furthermore, $\log n + c$ is obtained from the use of the operations insertBalance() and deleteBalance() where every rotation and color change operation preformed in those functions take $O(\log n), O(1)$, respectively. Therefore, the insert and delete operations are $O(\log n)$. The find_smallest operation is also $O(\log n)$because it iterates through the tree to find the kth smallest value in the tree. Therefore, worst case will be the height of the tree.

5. Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into rang [a…b] in $O(1)$time. Your algorithm should use (n+k) preprocessing time.

To preprocess the data given we use an algorithm named, preprocessCounter. This algorithm will count the number of elements in array A, by incrementing the index correspondent value of array B, B[A[x]], whenever the loop encounters that index. Furthermore, counting all the elements in array A, the algorithm then adds every value in B, B[x] to the previous value, B[x-1]. This will make each index value equal to its number of elements, and all the elements that are to it as well, algorithm

Algorithm: preprocessCounter(A, B, k)
In: An array with n integers, A. An empty array of size k, B. An integer k.
Out: An Array that contains the counted number of elements less than i.

```
preprocessCounter(A, B, k)
for i ← 0 i to k
        B[i] ← 0
for j ← 0 j to A.length-1
        B[A[j]] ← B[A[j]] + 1
for I ← 1 i to k
        B[i] ← B[i]+ B[i-1]
 return B
```

Secondly, the query algorithm will determine the number of elements that fall into range (a, b) based on the results obtained from the preprocessCounter algorithm. The algorithm checks to make sure that both a and b are within the array bounds and performs a $O(1)$ to fetch the number of elements in the range (a, b) by B[b] – B[a], algorithm

Algorithm: query(a, b, A, B)
In: Integers a and b that set the range, an array containing the count of elements less than i, B.
Out: An integer that represents the number of integers that fall in the range (a, b)

```
query(a, b)
B ← preprocessCounter(A, B, A.maxValue)
if a < 0 then a ← 0
if b > k then b ← k
return B[b] – B[a]
```

6. Given k sorted sequences each of which has n elements, design an algorithm to merge them into one sorted sequence. What is the time complexity of your algorithm?

Firstly, the algorithm will be divided into two parts, an algorithm named merge that will contain a queue of all the lists to be merged, and merger, which will merges two lists are sends it back to merge to enqueue it back into the queue, merge will keep looping until there is only one list left in the queue, algorithms

Algorithm: merger(A, B)
In: A and B are two sorted arrays.
Out: A sorted array of A and B's merged components

```
merger(A, B)
C ← array [2n]
j ←0
z ← 0
n ← A.size + B.size
for i ← 0 i to n
        if j < A.size & z < B.size then
                if A[j] < B[z] then
                        C[i] ← A[j]
                        j ← j + 1
                else
                        C[i] ← B[z]
                        z ← z + 1
        else if j < A.size then
                C[i] ← A[j]
                j ← j + 1
        else
                C[i] ← B[z]
                z ← z + 1
                i ← i + 1
return C
```

Algorithm: merge(queue)
In: A and B are two sorted arrays.
Out: A sorted array of all the lists in the queue.

```
merge(queue)
while queue.size > 1 do
        A ← queue.dequeue
        B ← queue.dequeue
        queue.inqueue(merger(A, B))
return queue.dequeue
```

The merge algorithm is divided into two parts, the merge algorithm itself and its helper the merger algorithm. In the merge algorithm the inqueue and dequeue operation are of $O(1)$. While the merger helper algorithm is of $O(kn \log k)$ where kn is equal to the total number of elements and log k is the height of the recursive binary tree that is created by merging different the lists. Therefore the time complexity of the algorithm is $O(kn \log k)$

7. Suppose we have an optimal prefix code on a set $C = \{0,1, \ldots, n-1\}$ of the characters and we wish to transmit this code using as few bits as possible. Show how to represent any optimal prefix code on C using only $2n - 1 + n[\lg n]$ bits. (Hint: Use $2n - 1$ bits to specify the structure of the tree, as discovered by the walk of the tree.)

Since we know that a tree with n leaves also has $n-1$ internal nodes. Then $2n-1$ bits are sufficient to store a full structure of a tree of n leaves; this structure is obtainable by running a tree traversal on the tree. Furthermore, the characters of C are going to be represented by tree traversal sequences, as in Huffman's coding, therefore, every character will need $\lg n$ bits to be store, the sequence that each character is represented in depends on the type of tree traversal we choose. Therefore, we need $n[\lg n]$ bits to store all the characters, which adds up to $2n - 1 + n[\lg n]$.

8. Prove that every node has rank at most $[\lg n]$.

Proof by induction.

A node of rank r is a node of at least size 2^r

Base case: $2^0 = 1$

A node of rank 0 is a node of size 1.

Inductive case: $2^r \leq n$

The rank of any node is only increased when it is unionized with another node of the same rank, therefore, $2^{r+1} = 2^r + 2^r$

By inductive hypothesis, $2^r \leq n$

$2^r + 2^r \leq n + n$
$2^r(1+1) \leq 2n$
$\cancel{2} * 2^r \leq \cancel{2}n$
$2^r \leq n$
$r \leq \log n$

Proven by induction.

9. How many bits are necessary to store $x.rank$ for each node x?

Since $x.rank$ needs $\log n$ bits to be stored and the node also needs $\log n$ bits to be stored. Therefore, the amount of bits that are necessary to store $x.rank$ are $\log \log n$.