CS3340 Assignment #2

Tuesday, February 24th, 2015

James Crocker

250634027

113/135
Marked by Zhewei

8/10 QUESTION 1)

An algorithm that accomplishes this is a modified Counting-sort. Counting-sort will accumulate a running total of all the numbers such that the value of any element in the array, C, is the sum of all numbers below that element's index. In order to determine the number of integers falling in the range [a..b], you simply subtract C[b] – C[a], while taking into account that when a=0, C[a]=0.

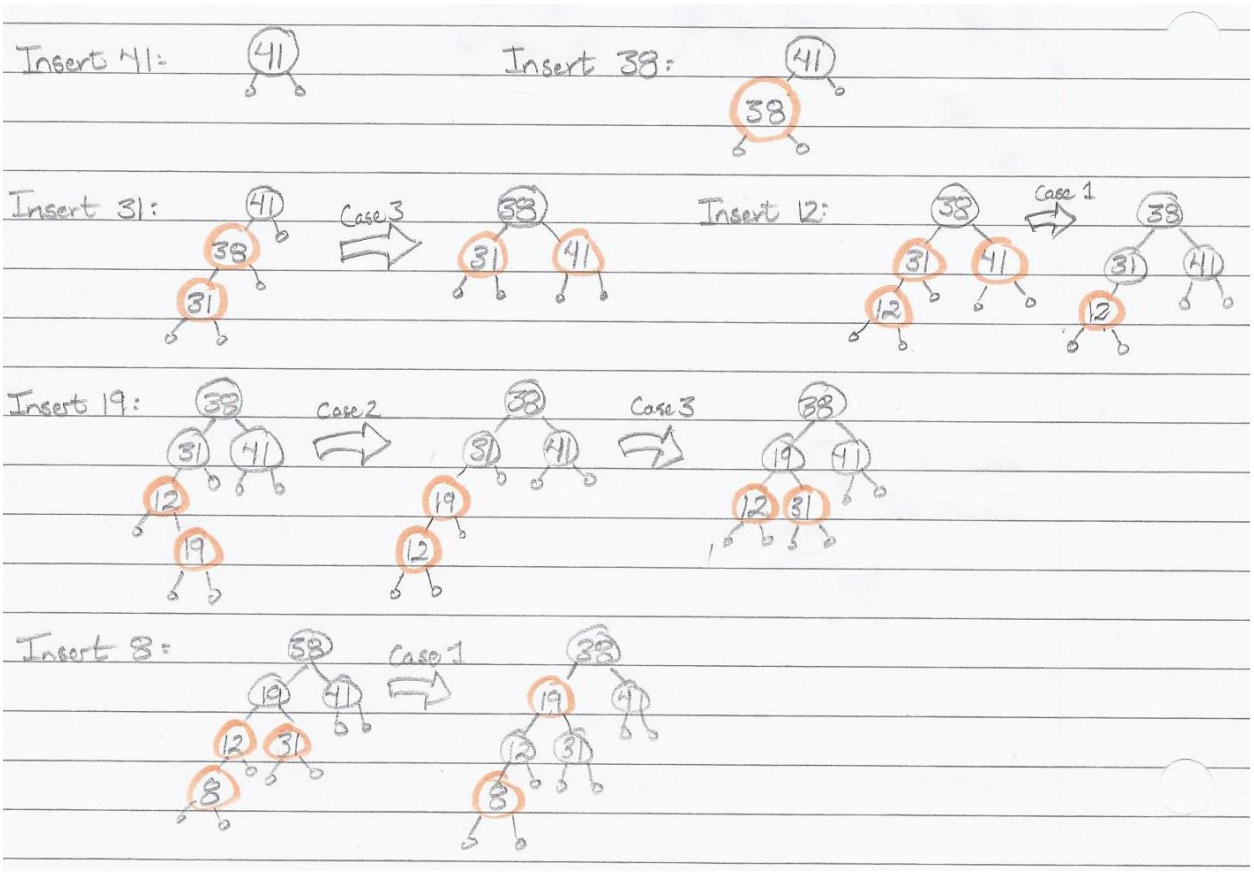
```
COUNTING-SORT(A,C,k, n)
Inputs: Array of inputs, A; the length of A, n; an array to hold
the running total, C; and the range of inputs k.
Output: Array C
For i=0 to k
     C[i]=0
For j=1 to n
     C[A[j]]=C[A[j]] + 1
For i=1 to k
C[i]=C[i] + C[i-1]
```

In order to calculate the number of integers in the range [a..b], we use the following algorithm:

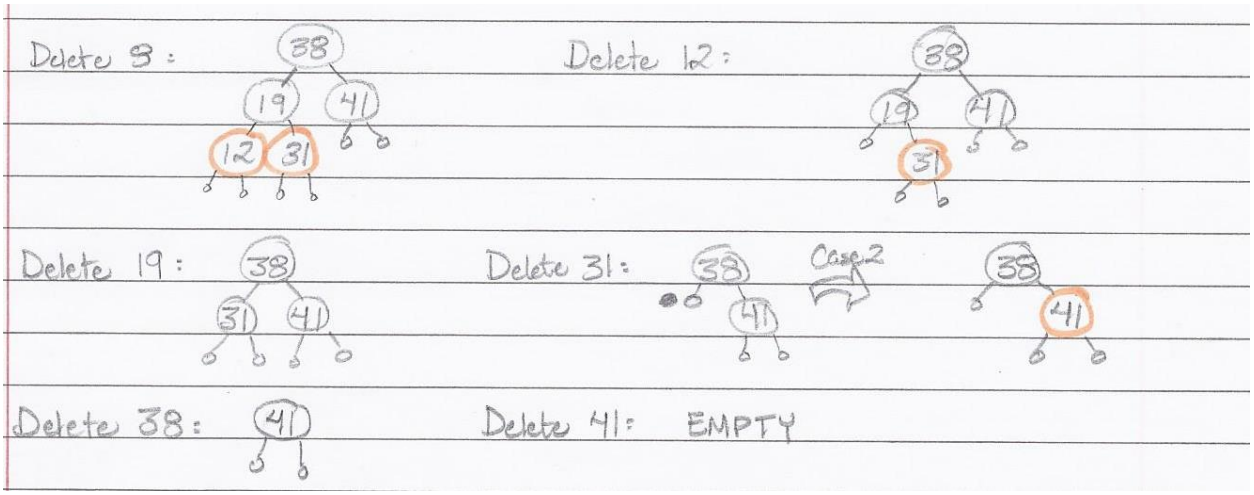
```
RANGE-COUNT(a, b, C)
Inputs: Lower bound, a; upper bound, b; and the array C
containing the running total.
Outputs: The number of integers in the range a to b.
if a=0 return C[b]
else return C[b] – C[a]
```

Finally, we see the algorithm is correct since C contains the running total of all integer inputs, and therefore the number of integers in the range a to b is simply the C[b]-C[a]. Additionally, COUNTING-SORT is O(n+k) which is required for preprocessing, and the query function RANGE-COUNT runs in constant time and is therefore O(1).

10/10 QUESTION 2)



10/10 QUESTION 3)



10/10

QUESTION 4)

a) The relationship expressing the number of nodes in an AVL tree is m(h) = m(h-1) + m(h-2), where m is the number of nodes. This is true since any node is simply the parent of two subtrees, each of the subtrees can differ in height by at most 1. Therefore,

```
m(h) > 2*m(h-2) = 2*2m(h-4) = 2*2*2m(h-6)
.
.
.
=2^i*m(h-2*i)
```

When the height of the tree is 0, there is only 1 node. Therefore, m(h-2*i) = 1, h = 2*i, and i = h/2.

m(h) > 2^i = 2^(h/2)
Since m(h) = n, n =2^(h/2) and therefore h=2*lgn.

Therefore, the height of an AVL tree is O(lgn).

b) BALANCE(x) will work by comparing the right and left subtree's, and based on whichever tree is taller by more than 1 level, will subsequently apply the appropriate rotation to the taller tree. Either a right or left rotation will be applied depending on the configuration of the unbalanced tree.

```
BALANCE(x)
Input: Node x
Output: { Balanced AVL tree }
diff <- x.right.h - x.left.h
if |diff| <= 1 return
else if diff > 1 {
      perform_rotation(x.right)

else
      perform_rotation(x.left)
```

This algorithm is correct since rotations are only made to nodes if there are subtrees differing in heights by more than 2. Additionally, it since a single rotation runs in constant time, BALANCE(x) will run in constant time.

c) AVL-INSERT(x,z) will operate similarly to TREE-INSERT(x,z) on pg 294. To insert a node into an AVL tree, we first insert the node as if it were a regular binary search tree. Then beginning at the node where z was inserted, we continue to call the BALANCE(x) function until we find an unbalanced pair of sub trees, or we reach the root.

```
AVL-INSERT(x,z)
Input: Root x, and node z to be inserted.
Output: { AVL tree with z inserted in it }
Y <- Nil
while x != Nil
      y <- x
      if z.key < x.key
            x <- x.left
      else x <- x.right
z.parent <- y
if y == nil
      T.root <- z
elseif z.key < y.key
      y.left <- z
else y.right <- z
// Check the tree for balance
a <- z
while z != nil
      if |z.left - z.right| > 1
            BALANCE(x)
            return
      else z <- z.parent
```

d) AVL-INSERT will go through the height of the tree until a leaf is reached where the node can be inserted. The height of an AVL tree is O(lgn), therefore this operation takes O(lgn) time. Next, at most, all the nodes are checked for balance until the root is reached. Therefore this 'check' portion of the algorithm takes at most O(lgn) time. Finally, if subtrees are found to be out of balance, BALANCE(x) is called and the AVL-INSERT function ends (so only a constant number of rotations can occur). Therefore, the algorithm takes O(lgn) time and O(1) rotations occur.

7/10

QUESTION 5)

The red-black tree data structure can accomplish Insert, Delete and Find_Smallest with the addition of some helper methods.

Insert(x) will remain unchanged from the standard Insert(x) method for red-black trees (see p315 of textbook) and it will therefore run in O(lgn) time.

Delete(x) will also remain unchanged from the standard Delete(x) method for red-black trees (see p324 of textbook) and it will therefore run in O(lgn) time as well.

To incorporate the Find_Smallest(k) function, we first make a global variable count = 0. We then subsequently call a modified In-Order traversal on the red-black tree. While executing the In-Order traversal, the global variable count is incremented by 1 for each node traversed. Once count = k, the kth smallest node has been found and therefore its key can be returned by the data structure.

```
count = 0 { global variable }

Find_Smallest(k)
Input: Integer k
Output: The kth key of the data structure.
return INORDER(r, k)

INORDER(r, k)
Input: Root r, integer k.
Output: The kth key of the data structure.
if r is a leaf
      count <- count + 1
      if count = k
            return r.key
      else return null
else
      a <- INORDER(r.left, k)
      if a != null return a
      else
            count <- count + 1
            if count = k return r.key
            else return INORDER(r.right, k)
```

Since this recursive call will visit at most, each node once and perform a constant number of operations per node, Find_Smallest is O(n). This algorithm is correct because it visits the nodes in ascending order (due to the characteristics of an in-order traversal) and returns the key of the kth element visited.

QUESTION 6)

6/10

To output a sorted sequence of the smallest k elements, we must create a maximum heap from the input elements. Once this is done, we can subsequently make k calls to Extract_Minimum(). The pseudocode for this algorithm is shown on the following page.

```
K-SORTED-SEQUENCE(A, n, k)
Input: Array A of size n, size k.
Output: Array B sorted with the k smallest elements
BUILD-MAX-HEAP(A)
for i = 0 to k
      B[i] <- A.Extract_Minimum()
return B
```

This algorithm is correct because each time Extract_Minimum is called on the maximum heap, the smallest element in the heap is removed and returned. Therefore, successive calls to Extract_Minimum return the 2^{nd} , 3^{rd}, 4^{th} smallest elements and we keep them ordered by inserting them into ascending positions in array B.

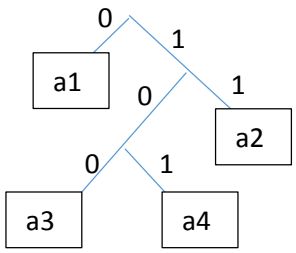
From p.159 of the text, we see BUILD-MAX-HEAP is O(n), and due to the tree structure of a heap Extract_Minimum is O(height) = O(logn). Since k calls are made to Extract_Minimum, the running time of this algorithm is O(n + klogn) but the condition in the question states that klogn < n. Therefore the running time of this algorithm is O(n).

For a general k, this algorithm will still run O(klogn + n) and since klogn <= klogk + n, then O(klogn + n) is O(klogk + 2n), which is O(klogk + n). Therefore this algorithm will run O(klogk + n) for a general k.

5/10 QUESTION 7)

For a set of characters a1, a2, .. a^n, whose frequencies are monotonically decreasing, we can prove the existence of an optimal code for codeword lengths monotonically increasing by drawing a n-leaf binary tree. On this tree, each left edge is labeled with a 0 and each right edge is labeled with a one. Then each character of high frequency appears as a leaf closer to the root, and each character of lower frequency appears closer to the bottom. Using the first 4 terms:

E.g.



We see that as the frequency of the character decreases, then the codeword length increases: a1:0, a2:11, a3:100, a4:101. For additional characters added to the tree, each level of nodes on the tree adds 1 to the codeword length of the character. This the optimal code because more common characters will occupy less space and less common characters will occupy more space, thus leading to a smaller file overall. Therefore, as the frequencies monotonically decrease, their optimal code lengths monotonically increase.

4/10

QUESTION 8)

To prove this we consider that trees are always formed by the merging of smaller trees. Additionally, the tree with the smaller rank points to the tree with the larger rank, and the rank will only ever increase if a union is formed between two trees of equal rank. For the rank = $\lg\lfloor n \rfloor$, n >= 2^rank. To prove this we recognize that when rank=0, n=1. For a rank r, we assume that n >=2^r. Using induction, we know that the union of two trees of rank r-1 will have a rank or r. Each subtree of rank r-1 will therefore have at least 2^(r-1) nodes and therefore the resulting tree will have at least 2*2^(r-1)=2^r nodes. Therefore every node will have a rank of at most $\lg\lfloor n \rfloor$.

8/10

QUESTION 9)

The number of bits required to store the rank of any node is $\lg(\lg\lfloor n \rfloor)$.

QUESTION 10)

b) The algorithm created operates by using a Union-Find data structure to identify connected components. After the input file is parsed, a 2D array is created to contain the image's data. Subsequently, for each element in the array, if it represents a '+' in that image, the Make_Set() function is called for that position. Next, if this position represents a '+' and the element directly to the left also represents a '+' in the array, then the Union_Sets() function is called on these two neighboring elements. Next, if the element directly above also represents a '+', then the Union_Sets() function is called on these two neighboring elements. By the time this process has been applied to all elements in the array, the Union_Set and Make_set operations will have 'joined' connected components by modifying all the pointers in a connected component to point to the parent

node of that connected component. Therefore all the connected components have been identified.

This algorithm is correct because for this problem a connected component is defined only for '4 connectivity' (i.e. diagonals are considered connected). Therefore by iterating through the 2D array and comparing an element to its left neighbor and upper neighbor, all elements will be checked for '4 connectivity'. The right and lower neighbors do not require checking because they will captured by applying the same operations in the next iterations and joined through the Union_Sets() operation. It is ensured that the connected components are actually 'joined' because of the inherent characteristics of the Union_Sets() function. When this function is called, connected components will be compared and they will be modified to point to one of their parents.

The complexity can be calculated by considering there are n elements that are iterated through and on which Make_Set() and Union_Set() operations are called on. Make_Set() is O(1), while Union_Set() is O(logn) since Find_Sets() must be called inside it. In the worst case, Make_Set() is called once for all n elements and therefore Union_Set() would be called twice for each (for the left and upper neighbors). Therefore the complexity of the algorithm is O(nlogn).

10: 45/45
Data type: 20/20
Program:12/12
Output: 10/10
Style:3/3