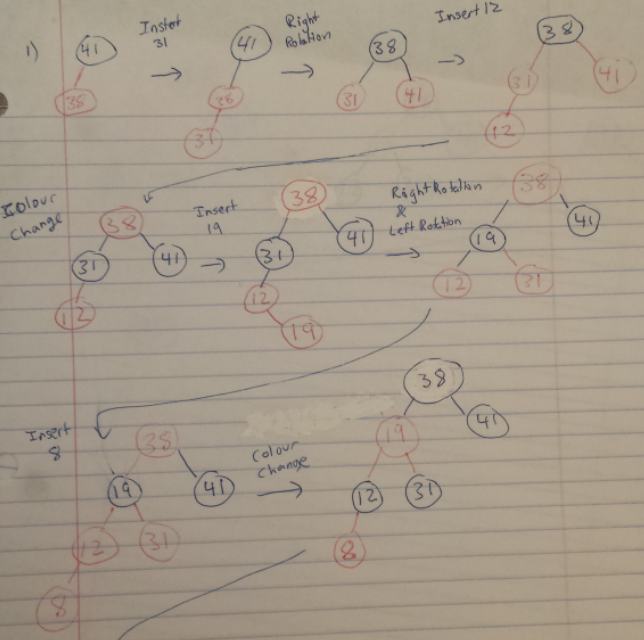
Yaqzan Ali

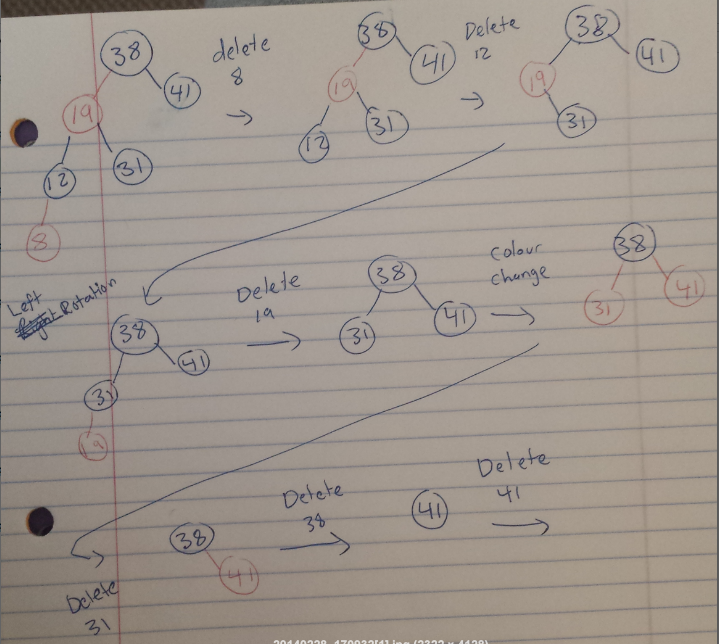
250644709

CS3340 Assignment 2.

1)



2)



13-3 A)

The Fibonacci Sequence goes 0 1 1 2 3 5 8 13 21 34 …. Etc

Let F(x) represent the xth Fibonacci number.

**Basis:** h = 0; a tree of height h has 0 or 1 nodes. F(0) = 0;

Let T be an AVL tree of height h+1. Let T(L) be the left subtree and T(R) be the right subtree. Both T(L) and T(R) are both AVL Trees, by definition of an AVL Tree. Since T has height h+1, at least one of T(L), T(R) has a height of h. Neither have a height greater than h. Thus, by definition of an AVL Tree, either both have height h, or one has height h, and the other has height h-1. By induction, the number of nodes in T is at least **F(h) + F(h-1) + 1**. Which is at least F(h+1).

Now let *φ* = 1.61803399 (The Golden Ratio). Then, based on the properties of the Fibonacci numbers,

F(i+1) ≥ *φ*i ≥ 0

Combine these equations and we see that an AVL tree of height h ≥ 2 has at least *φ*h-2 many nodes.

This implies that an AVL tree with n nodes has at most  **+2 ,** which is **O(log n)**

4) **Insert(x)** – We insert the key into the tree, like any other red black tree. However, we also add 2 other values for this node: The number of nodes in the left subtree. Now, whenever we add a node, every node gets updated on how many nodes are in its right and left subtrees. Insertion takes O(Log2n) (Binary tree insertion)+O(log2n)(colour changes) + O(1) (for rotations) + O(log2n) (for updating values at each node). Therefore, the time complexity is O(log2n).

**Delete(x)** – Same as any other red black tree deletion, except we also recursively update the parent of the deleted node’s parent until the root. We update the amount of nodes in its left subtrees. We decrement the value accordingly. Deletion takes O(Log2n) (Binary tree deletion)+O(log2n)(colour changes) + O(1) (for rotations) + O(log2n) (for updating values at each parent node). Therefore, the time complexity is O(log2n).

**Find\_Smallest(k)** – We use a Breadth first search. We start at the root, comparing k, to the value of the number of elements in the left subtree. If k > (nodes in left subtree), then we can ignore the left subtree, because they are all smaller than the number we are looking for, and proceed to the right subtree. If k < (nodes in left subtree), we proceed to the left subtree, and repeat the procedure. If k = (nodes in left subtree), then we’ve found the kth smallest element. This operation takes O(Log2n) time.

5)

For this, we assume an array A[1… n] and we need two additional arrays B[1… k] and C[1… k]. Our algorithm will first initialize all elements of array B to 0. This step requires O(k) time. Next, for each element of array A, we will increment B[A[i]]. This step will require O(n) time and B[j] for j = 1,2, … k. Now contains the number of elements of A having value j. Finally, make C[1] = B[1] and for each element t of array C, where t = 2,3, …, k, we will compute C[t] = B[t] + B[t-1]. This step takes O(k) time. Now, whenever we want to find how many of n integers fall into a range [a … b], we simply compute C[b] - C[a] + B[a]. This only takes O(1) time.

6)

For this, we use merge sort on each sequence, 2 at a time. So we conduct merge sort on the first 2 sequences, then we conduct merge sort on this newly formed sequence and the next sequence. And repeat, until all the sequences are merged. This takes O(k2 \* n) .

7)

An optimal prefix-free code on C has an associated [full binary tree](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees) with n leaves and n−1internal vertices; such a tree can be coded by a sequence of 2n−1, as given by the hint. It only remains to associate the n members of C with the n leaves of the tree, which can be done by listing them in the order in which preorder traversal encounters them. ⌈logn⌉ is enough bits to represent each of the n members of C, and no delimiters are needed if each is alloted ⌈logn⌉ bits. That comes to a total of 2n−1+n⌈logn⌉ bits.

8)

A node with rank r, then it is a root of a subtree of size at least 2r

Base Case:

A node with rank 0 has size of 1 (itself)

Inductive Case:

A node X can only have a rank of (r+1) if at some previous point, it had a rank r, and it was the root of a tree that was joined (via union) with another tree whose root had rank r. Then X becomes the root of the union of the two trees. Each tree, by Inductive hypothesis, is of size at least 2r, and so now X is the root of a tree of size at least 2r + 2r = 2(r+1). Now the number of nodes in the forest is n and we have at least 2r nodes in every tree with rank r. So then, n ≥ 2r = **r ≤ (logn)**

9)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Max. Rank (R) | 16 | 8 | 4 | 2 | 1 |
| # of bits (n) | 5 | 4 | 3 | 2 | 1 |

We can see that n = (LogR + 1)

If logn is the maximum rank, then the number of bits needed is (log R) + 1, where R is the rank.