3340A2

Hamza Bana, 250857725

February 23, 2020

1 Q1

Algorithm 1: Count (array, k, a, b)

```
Result: Output the number of integers n that fall within a range [a..b]
n = array.length;
arr = [0...k];
i = 0 for j ← 0 to n do
    arr[array[j]] ++;
end
for j ← 1 to n do
    arr[j] += arr[j-1];
end
//Above algorithm is from counting sort
 return arr[b] - arr[a-1] //O(1) operation
```

2 Q2

See pdf

3 Q3

See pdf

4 Q4

No, the tree will not look the same. Inserting and deletion involves possible rotating, rebalancing and recoloring the tree nodes. See pdf for example

5 Q5

Prove that AVL tree with n nodes has height lg(n)

Height of left and right subtree at every node, x, should only different by 1 for the tree to be an AVL tree.

Let h_l be the height of the left subtree and h_r be the height of the right subtree. $|h_l - h_r| <= 1$

Let n_h defined the minimum number of nodes in an AVL tree with height h

Base case: $n_0 = 1$ $n_1 = 1$ $n_2 = 2$

For a tree with height h, left side must be h-1 and right side must be h-2.

So the min number of nodes at height h can be expressed as the min number of nodes with height h-1 and height h-2, which are respectively the left and right subtree: $n_h = (n_h - 1) + (n_h - 2) + 1$

$h - 1$ are all the nodes on the left of node x and $h - 2$ are all the nodes on the right of node x. The +1 in the above formula indicates the counting of node x, or the root nod.

Represents Fib sequence of f(h) = f(h-1) + f(h-2)

Solve the recurrence.

$n_h = 1 + n_h - 1 + n_h - 2$

$n_h > 1 + 2n_h - 2$

$n_h > 2n_h - 2$

n_h ¿ $\theta(2^{h/2})$

$h < 2log_2 n$

Hence at height h, $h < 2log_2 n$ and at most could have height logn where there are n nodes.

6 Q6

Algorithm 2: ksorted(A, k)

```
Result: output a sorted sequence of smallest k elements
heap = heapify(A, A.length, 1);
//O(n) operation result = [k];
j = 0;
for i ← 0 to k //O(k) for loop do
    result[i] = heap.ExtractMinimum() //O(logn) operation;
    i ++;
end
//O(klogn)
return result;
```

The above algorithm is completed in O(klogn + n) time.

7 Q7

Prove that every node has at most rank *logn*. Assume above statement is true.

If we have 1 node: n = 1. Rank = logn = 0. Rank of node 1 = 0

Proof by induction: Assume n+1 total nodes.

Assume n nodes are in a set with the highest rank being logn of the parent root. Now call union operation with (n+1)th node. Since the ranks are unequal of the parent node of both disjoint sets on which the union is being called, the rank of of logn is preserved when the (n+1)th node is called with the union operation.

Now assume there are two disjoint sets with a and b total nodes. The rank of the parent with a nodes is loga and rank with b nodes if logb. a and b ¡= n.

When calling union on a + b, assume that a == b and loga == logb. a + b = n+1 and the rank will get incremented by one this time, hence the new rank will be log(n+1), where n+1 is the total nodes in the disjoint set.

We have proved by induction that every node has rank at most logn where n is the total number of nodes.

8 Q8

Since logn bits are needed to address n bytes. The highest rank possible is logn bytes where n is the total number of nodes.

We need to sub logn in to the first equation of addressing n bytes and we get: log(logn)

9 Q9

Encoding can be represented using a tree. A complete binary tree with n leaves has 2n-1 total nodes. n characters can be address with logn bits. Since we need a tree of size 2n-1 bits to represent the message with n characters. Each character required logn bits to be represented and there are n characters total, equaling n(logn) total memory to represent the entire prefix code.

Total memory required = memory for the tree + memory for the message

Memory for the tree = 2n-1

Memory for the message = n(logn)

Total memory required = 2n-1 + n(logn)

10 Q10

Correctness and complexity of question 10 code:

uandf: O(n) - creating an array of size n

make_set: O(1) - constant time. Allocation of data and parent pointers

union_sets:

2 find_set operations: O(1)

Checking if the ranks are same or not of two different set representatives = O(1)

Changing parent pointers = O(1)

find_set: O(1) - check nullity of index in an array and return parent pointer

final_sets: Iterate from 1 to size of array (n). O(n) operation.

Comparisons of the if-loop conditions are O(1) time. Checking nullity, checking parent pointers, checking if data is equal to a certain number.

Total time complexity of final_sets() = O(n)