Ben Dumais 250669195 3340-Asn 2

Question 1

Array { 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 }

1. Post-Initialization

A	6	0	2	0	1	3	4	6	1	3	2
B											
C	0	0	0	0	0	0	0				

2. Place count of elements into c

A	6	0	2	0	1	3	4	6	1	3	2
B											
C	2	2	2	2	1	0	2				

3. Increment count elements by previous element

A	6	0	2	0	1	3	4	6	1	3	2
B											
C	2	4	6	8	9	9	11				

4. First Pass

A	6	0	2	0	1	3	4	6	1	3	2
B						2					
C	2	4	5	8	9	9	11				

5. Second Pass

A	6	0	2	0	1	3	4	6	1	3	2
B						2		3			
C	2	4	5	7	9	9	11				

6. Third Pass

A	6	0	2	0	1	3	4	6	1	3	2
B				1		2		3			
C	2	3	5	7	9	9	11				

. . . . .

7. Final Pass

A	6	0	2	0	1	3	4	6	1	3	2
B	0	0	1	1	2	2	3	3	4	6	6
C	0	2	4	6	8	9	9				

Question 2

Inductive Hypothesis: For an RB-Tree containing k nodes, there is at least one red node in the tree.

Base Case: For n = 2, we are inserting one node into a tree consisting of only the root. The root is, by definition of RB-Tree, always black, and a node being inserted is set to red by default. Since the parent of the inserted node is black, the inserted node remains red. Thus at n=2 there is at least one red node in the tree

Inductive Step: At k+1 nodes, we are adding a node to a tree of size k (ie we have done k inserts to the tree). By hypothesis there is at least one red node in the tree that we are inserting another node into. By the RB-Insert algorithm, if the parent of the inserted node is black, the inserted node remains red, otherwise some operations must be done to rebalance tree. Let X be the inserted node

Case 1 - Parent(X) is black:

X remains colored red, and no changes are made to rest of tree. Since X remains red, there is at least one red node in the tree

Case 2 – Parent(X) is red:

Since the parent is red and X is red, we must do some operations to rebalance the black-height of the tree and satisfy RB-property. Several cases arise: (1) we recolor parent(x), parent(parent(X)) and X (2) we rotate parent left or right or (3) we recolor parent(X), parent(parent(X)), and then rotate parent(parent(X)). In (1), while 2 previously red nodes are recolored black, we are changing another node to be red, ensuring that at least 1 red node remains in the tree. In (2), we preserve the amount of red nodes in the tree via a rotation, and in (3) we gain a red node as we have inserted a red node, recolored one to red, and then recolor a red node to black.

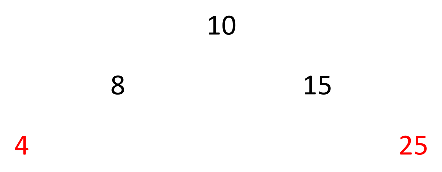
In all scenarios for this case, the amount of red nodes is at least guaranteed to be 1

Question 3

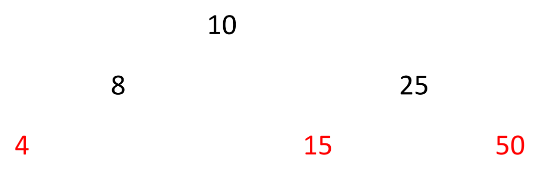
The only way to insert a node without changing the tree via rotations or recoloring is to insert a node into a position where its parent is black. This does not change the black-height of the tree and preserves the RB-property. If the parent is red, then one of the insert cases will have to happen, causing rotations or recoloring and changing the tree.

The same goes for removal: the only way to remove a node and not change the tree is to remove the node that is still red and its parent still being black.

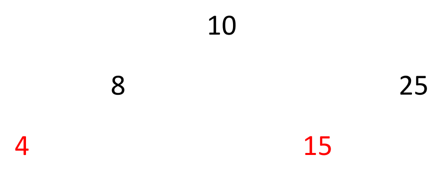
An insert followed by a removal in cases that DO cause the tree to rotate and recolor some nodes will not result in the same tree in all cases. Consider the following tree:



Inserting 50 causes a rotation and recoloring:



Now deleting 50 will not cause any further changes:



Therefore, insert then delete in an RB-Tree does not always result in the same tree

Question 4

With definition of AVL trees, a tree of height h is a root node connecting two AVL trees of height h-1, or a tree of height h-1 and height h-2 (as height can differ by 1)

Let N(h) represent number of nodes in AVL tree height h

$N(h) = 1 + N(h-1) + N(h-2)$

$= 1 + (1 + N(h-2) + N(h-3)) + N(h-2)$

$= 2 + 2N(h-2) + N(h-3)$

$> 2N(h-2)$

$> 4N(h-4)$

$> 8N(h-6)$

…

$> 2^{i}N(h-2i)$ -> $h-2i = 0$ when $i = h/2$

$> 2^{h/2}N(h-2(h/2))$

$> 2^{h/2}N(0)$

$> 2^{h/2} * 1$ -> Tree height 0 is one node, the root

$Lg(N(h)) > h/2$

$2lgN(h) > h$

$h < 2lgN(h)$

$= O(\log n)$

<u>Question 5</u>

“Tree List Structure”

Structure is a single-linked list of RB-Trees, ie a list of RB-Trees. The structure contains 2 pointers, one to the first node and one to the last node in the list, and an “order” property, an integer that denotes the maximum size of RB-trees in the list.

By the order property, each node in the list must be at least k more than the previous. For example, if we have 3 nodes and order 10, each node (RB-Tree) can have 10 nodes in it, the first can contain values 0-9, second 10-19, and third 20-29. Order can be initialized upon creation of tree.

Nodes in RB-Trees contain the value (key), pointers to left and right children, and pointer to parent node.

The structure will use indexing to locate which tree a node belongs to. The index function is 1 + Floor(key/order). For example, if we want to find the node containing 12 in a tree-list with order 10, we go to the 1 + Floor(12/10) node = 2nd node.

<u>Insert(x)</u>: Can be implemented by moving to [1+ floor(key/order)]th node and calling RB-Insert(x) on the resultant root. If the appropriate node in the list does not exist, it will be created and then x will be inserted as the root.

Complexity (worst case): let n be nodes in list, k be order, N be total number of nodes. n = floor(N/k). In worst case we calculate index, move n nodes up list, and call RB-Insert. This gives us:

C1 + C2*n + RB-Insert = C1 + C2*n + (C3*logk + C4) = n + log k = O(n) = **O(N/k)**

<u>Delete(x)</u>: Can be done similarly to insert - move to indexed node and call RB-Remove(x).

Complexity (worst case): As before, let n be nodes in list, k be order, N be total number of nodes. n = floor(N/k). In worst case we calculate index, move n nodes up list, and call RB-remove. This gives us:

C1 + C2*n + RB-Insert = C1 + C2*n + (C3*logk + C4) = n + log k = O(n) = **O(N/k)**

<u>Find_Smallest(k)</u>: Use input k as a proxy for key. Go to the [1 + floor(k/order)]th node in list and call RB-Smallest (essentially BST-Smallest method). This takes us the to tree that holds the kth smallest node, so now to find that node, we work our way back up the tree using parent pointers. The amount to move up the tree is given as the difference between the input k and the order*index. Specifically move k – (order*(floor(k/order))) nodes up from smallest node in that tree. If the indexed node does not exist, go to the last node in list and search from there.

For example, if we want the 2nd smallest node, we need to go to the 1st node (1 + floor(2/10)). We then go to the smallest node in the tree. The remaining difference is now 2 – (10(floor(2/10))) = 2 – 1 = 1, so we move one node up from the smallest. This is the kth smallest node.

Complexity (worst case): Let n be nodes in list, r be order, N be total number of nodes. n = floor(N/k). Constant time for index, n maximum moves down list to proper tree, a call to BST-Smallest then r moves up tree: C1 + C2*n + BST-Smallest + C3*r = C1 + C2*n + log(r)*C3 + C4*r = **O(N/r)**

<u>Question 6</u>

<u>Description of Algorithm:</u>

Merge_arrs(int k, int n, int arrs[k][n]) takes in the number of arrays (k), the size of arrays (n), and a 2d array of size k*n, where each position in the array length k is an array of size n.

```
1. Create 1d array, result_array, of length k*n
2. Create 1d array, positions, of length k
3. Create integers smallest and ptr
4. Set smallest to a large number
5. Initialize positions array elements to 0
6. Loop from 0 to k*n
    a. Loop from 0 to k
        i. If arrs[i][positions[j]] < than smallest
            1. Set smallest to arrs[i][positions[j]]
            2. Set ptr to j
7. Place smallest into result_array[i]
8. Increment positions[ptr]
9. Reset smallest to a large number
10. Return result_array after finishing loops
```

<u>Correctness:</u>

Input array: { 1, 5, 7
2, 3, 9
4, 6, 8 }

n = 3

k = 3

<u>Initial:</u>

Result_array = { }

Positions = { 0, 0, 0 }

Ptr = 0

Smallest = 1000

First Pass:

result_array = { 1 }

positions { 1, 0, 0 }

ptr = 0

smallest = 1

Second Pass:

results_array = { 1, 2 }

positions = { 1, 1, 0 }

ptr = 1

smallest = 2

Third Pass:

results_array = { 1, 2, 3 }

positions = { 1, 2, 0 }

ptr = 1

smallest = 3

Fourth Pass:

results_array = { 1, 2, 3, 4 }

positions = { 1, 2, 1 }

ptr = 2

smallest = 4

And so forth until…

Final Pass:

results_array = { 1, 2, 3, 4, 5, 6, 7, 8, 9}

positions = { 4, 4, 4 }

ptr = 1

smallest = 9

<u>Complexity:</u>

O(1) to create arrays and integers, k*constant to initialize *pointers*. Within loop, 1 comparison and 2 constant operations. Inner loop loops k times, outer loop k*n times. Post loops, constant time.

C1 + k*C2 + (k*n)*k*C3 + C4 = k + k^2n = **O(k^2n)**

Question 8

Let r represent rank of a node, n represent number of Make-Set operations.

Induction Hypothesis:

$r <= \lg(n)$, which can be written as $2^r <= n$

Base Case:

When n = 1, one node exists and has rank set as 0 by default

$0 <= \lg(1)$, or $2^0 = 1$

Inductive Step:

A node has rank r+1 when it is previously has rank = r and is joined to a tree rooted by a node with at least rank = r. Using n as number of sets and hypothesis $2^r <= n$:

$2^{(r+1)} <= n$
$2(2^r) <= n$ => by hypothesis, and $2 <= n$

Therefore rank of node is bounded by log(n), ie O(log(n))

Question 9

Since rank is bound by log(n), there are log(n) characters possible for rank. Given the nature of Union-Finds and Trees, nodes of rank = log(n), ie the maximum rank), will only occur once in the structure and there will be many leaf nodes. This means that frequency(rank) will decrease exponentially as rank increases (logarithmic nature of structure).

Let h represent height of a node, d represent depth of a node, r represent rank of a node, f(c) represent frequency of a character c, and d(c) represent length of code for character c.

Maximum number of nodes at a depth = 2^d, so: $f(r_i) <= f(2^{n.depth})$

Length of code for rank is equal to depth of a leaf, so: $d(r_i) = n.depth$

So number of bits needed to encode rank of a node is equal to the depth of that node

Total number of bits needed is $Sum_{i=0 \text{ to } logn}(f(r_i)d(n_i)) = Sum_{i=0 \text{ to } logn}(2^{ni.depth} \text{ x } n_i.depth)$

Question 10

a) See UandF.java, UnionFind.java and Node.java for implementation
b) Was not attempted due to time constraints arising from my dual degree/conflicting deadlines. Apologies…