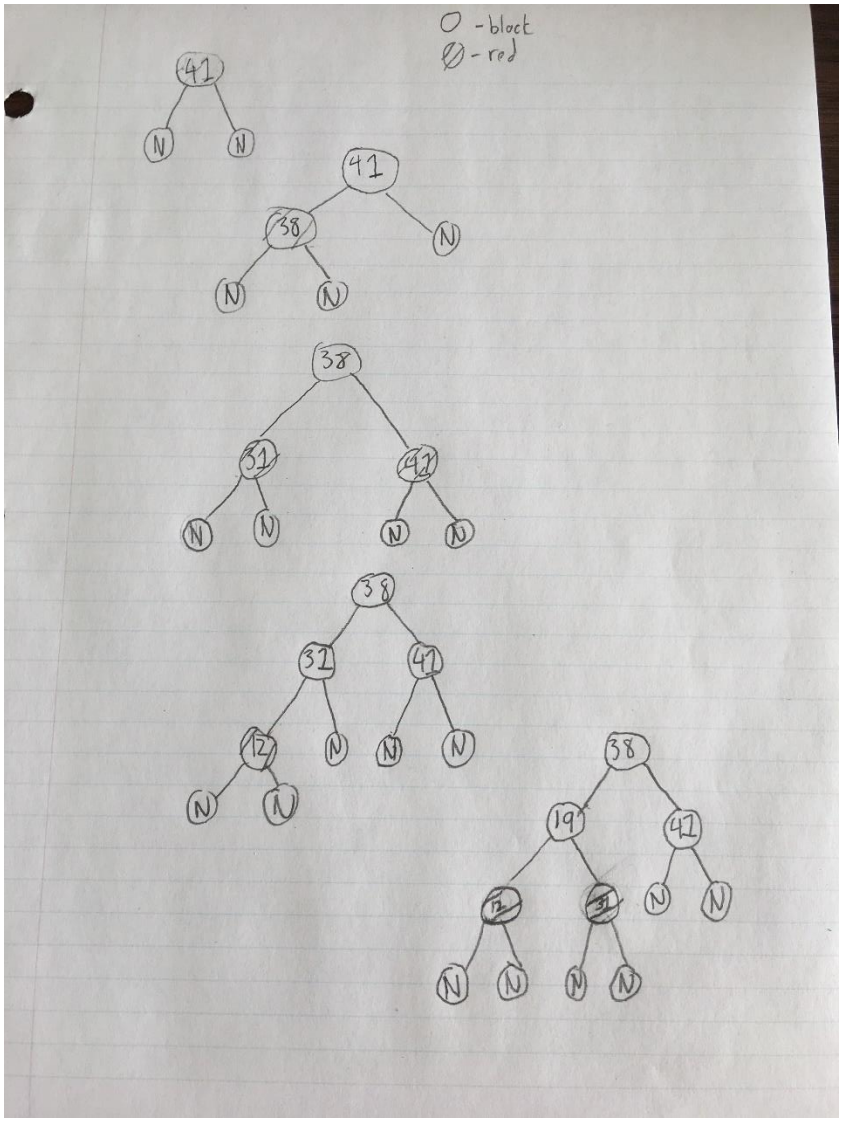
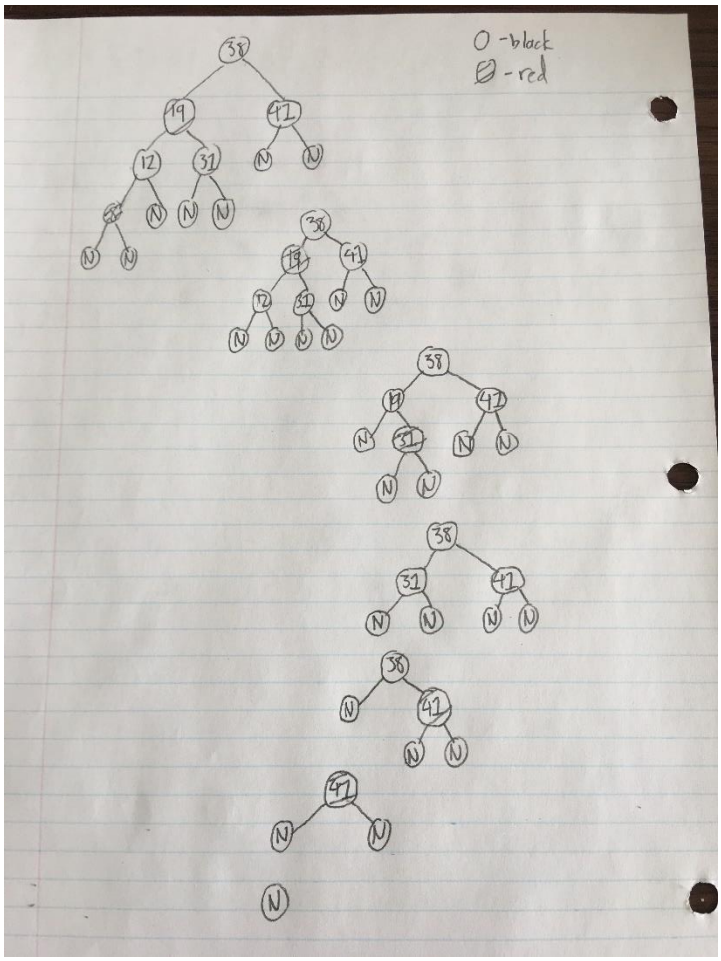
CS3340 Assignment 2 By Michael Song

Student #: 251101048

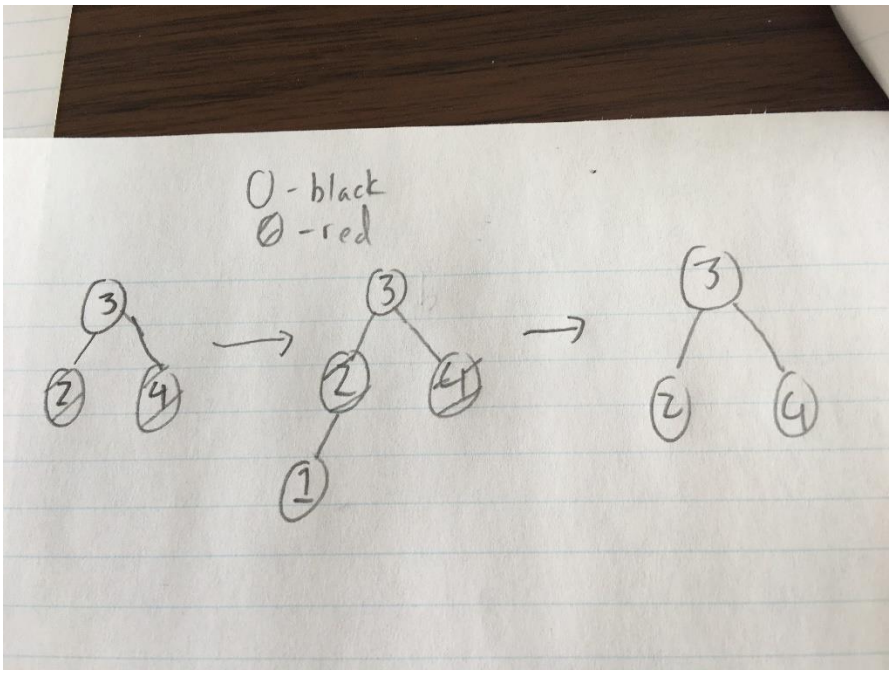
1. The algorithm would use the first couple of steps of COUNTING-SORT in preprocessing. This would allow it to create an array C such that C[i] contains the number of elements that are less or equal to i. If asked about how many integers fall into the range [a…b], the answer could be found through C[b] – C[a – 1].
2.



3.



4. The resulting Red-Black Tree is not the same as the initial tree. This can be demonstrated with an example of:



5. If height is h, S(h) = minimum size of an AVL tree and maximum value of children's heights = h – 1.

   We know that: $S(h) >= S(h-1) + S(h-2) + 1$

   We want the minimum size of $S(h) = S(h-1) + S(h-2) + 1$

   As $S(1) = 1$, $S(0) = 0$, $S(h) = [(\phi^h/\sqrt{5}) + (1/2)] <= n$

   $(\phi^h/\sqrt{5}) - (1/2) <= n$

   $\phi^h <= \sqrt{5}(n + 1/2)$

   $h <= (\lg(\sqrt{5}) + \lg(n + 1/2))/\lg(\phi) \in O(\lg(n))$.

   In conclusion, AVL trees with n nodes have height $O(\lg(n))$.

6. First, create an additional array that is k times the length of each of the arrays (n) by themselves (kn). We proceed to fill this array with zeroes in order to initialize it. Create counter variables for each of the k arrays that are initialized at 1 to count which part of each array you are on. We need to use a min heap in order to create a proper algorithm. First, we add the first element of each of the arrays into the min heap from which we derive the smallest element. The smallest element is added to the new array. We increment the counter of the array which had the smallest value and add the new element to the min heap and repeat the process until all elements of all k arrays are used up. The time complexity of this algorithm should be

O(kn*log(n)) due to the fact that there are kn elements in total and we need to analyze every element once in order to put each into the new array. The min heap will have 1 + floor(log(k)) levels, thus giving us the log(n) part of time complexity.

7. Strong Induction:

   Base Case:
   If n = 1, then that node has rank equal to 0 = [lg1].

   Induction Step:
   If there are n + 1 nodes, then we use placeholder a and b nodes respectively, where a, b are less or equal to n.

   Root of the first set has rank at most [lg a]
   Root of the second set has rank at most [lg b].

   Assuming the ranks are equal, then the rank of the union increases by 1, and the resulting set has rank $[\lg a] + 1 \leq [\lg(n + 1)/2] + 1 = [\lg(n + 1)]$.

8. Because the rank of each node is at most [lg(n)], we can show them using $\Theta(\lg(\lg(n)))$ bits at maximum.

9. The Huffman code exists to prioritize characters that appear more frequently by giving them shorter identification numbers and thus having every character be the same creates a situation in which every character is represented using 8 bits. As a result, the total file length fails to decrease at all even when the compression is implemented as the amount of bits used for each character remains the same.

10. A) DisjointSetsLinkedList(int n) is used in place of wandf(n) and it creates an array to store nodes and calls on make_set() to make the requested number of sets in accordance to n.

make_set(int a) sets the parameters of the head, tail, and size for a node and adds the node to the array of nodes created previously.

find_set(int a) retrieves a node by finding if it is in the array of nodes and returning it.

union_sets(LinkedSet set1, LinkedSet set2) will join the tail of the set to the head of the input set.

final_sets() will have the size recorded by a map that allows for a size to be called with pMap.get(i).size().

B) For the first part: A scanner will be used and the characters will be printed out as they are read by the scanner.

For the second part: A HashMap will be created to identify the positioning of each of the separate attributes.

For the third part: Every head value will be printed out and every node associated with it will follow.

For the fourth part: The second part will be repeated except if (pMap.get(i).size() > 4) will be added.