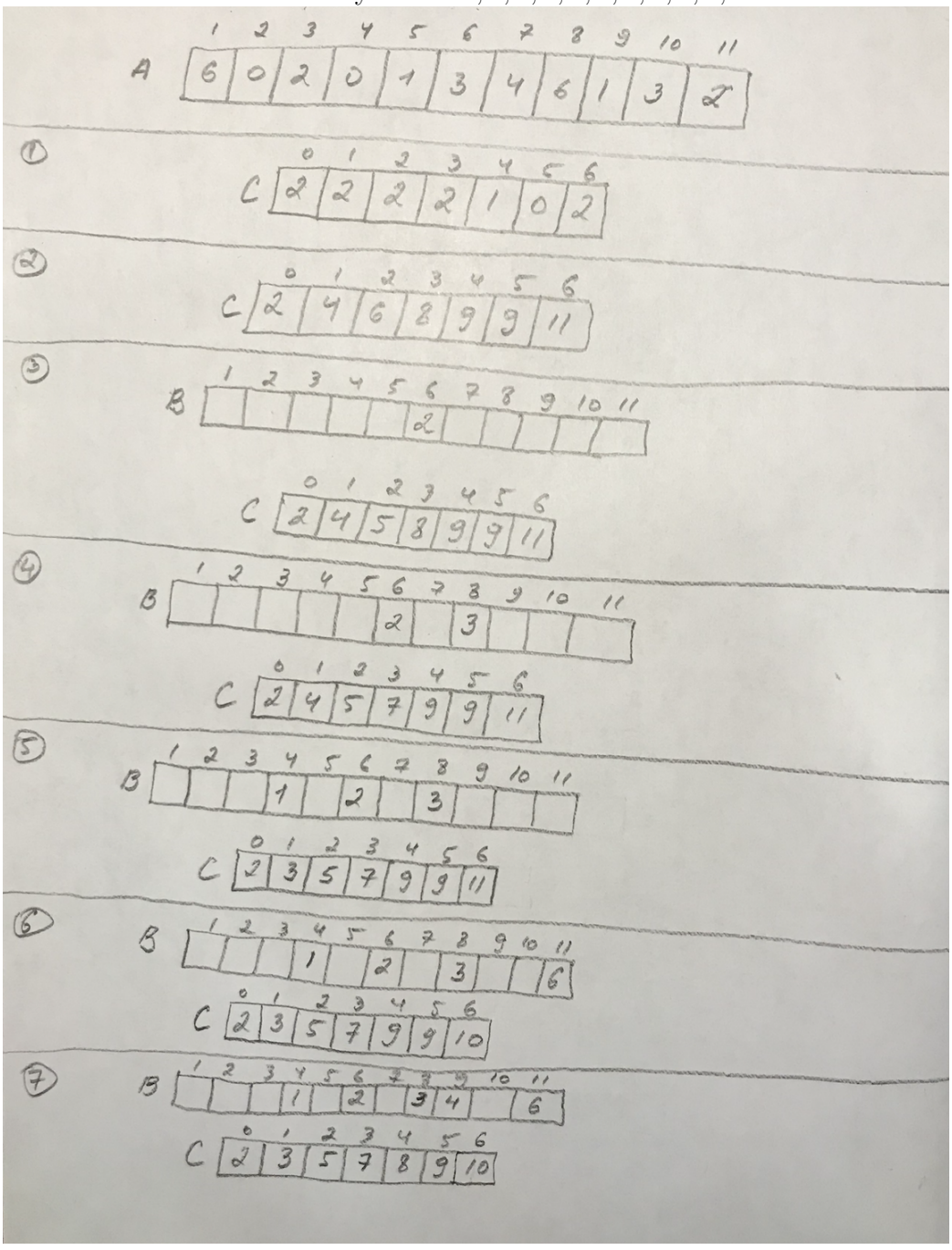
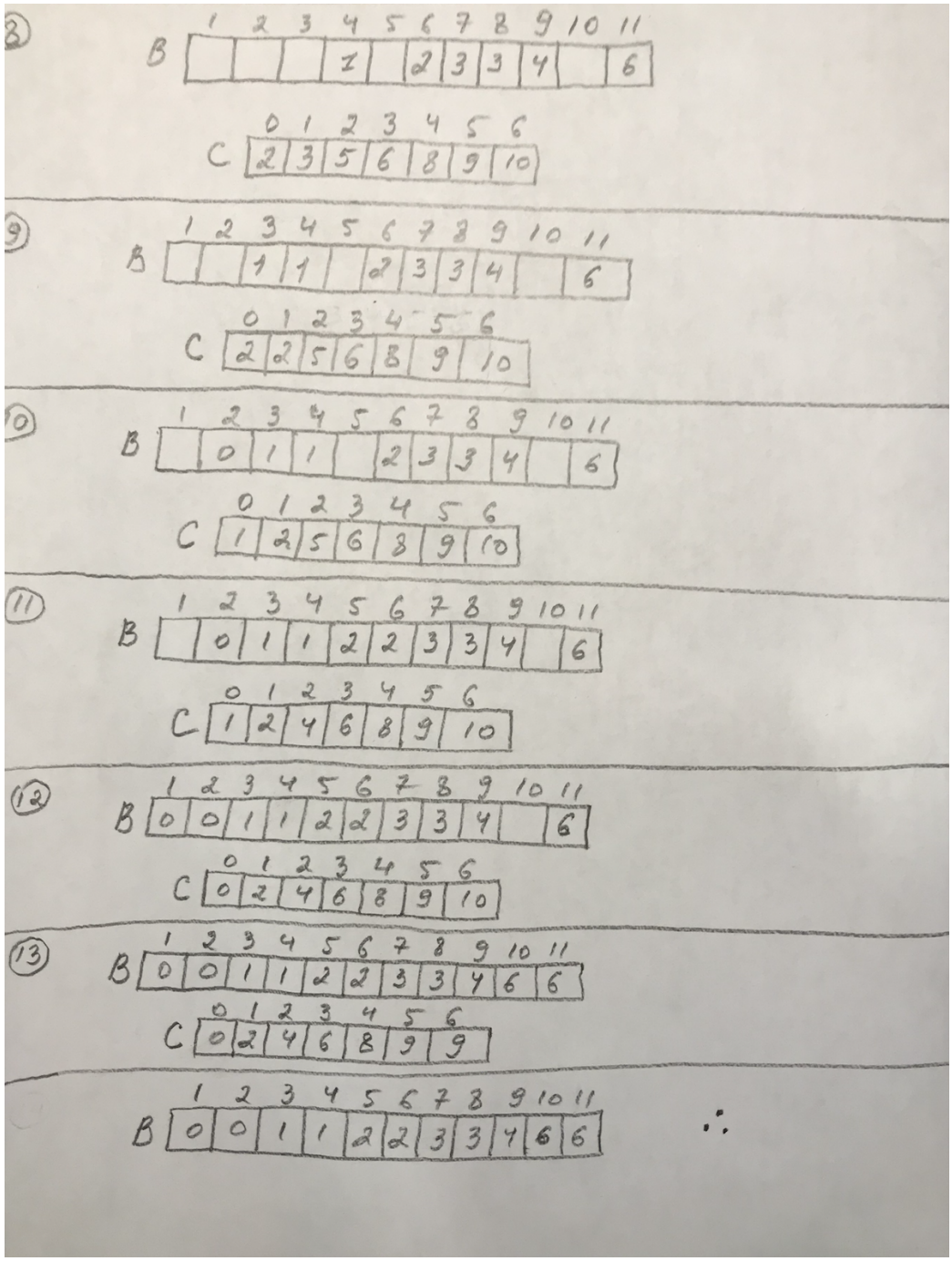
CS 3340 Assignment 2

Matvey Skripchenko 250899673

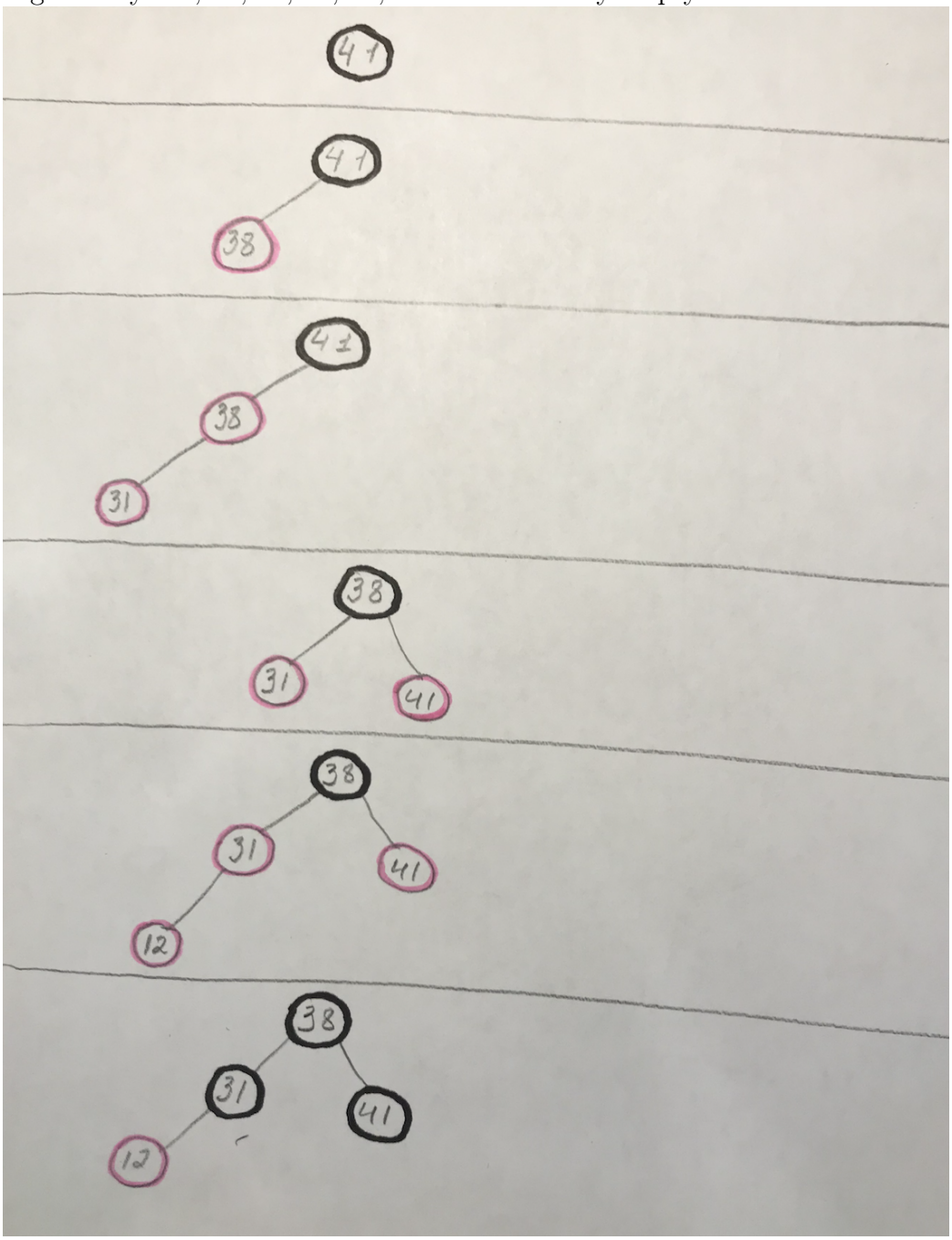
Due: March 2nd, 2021

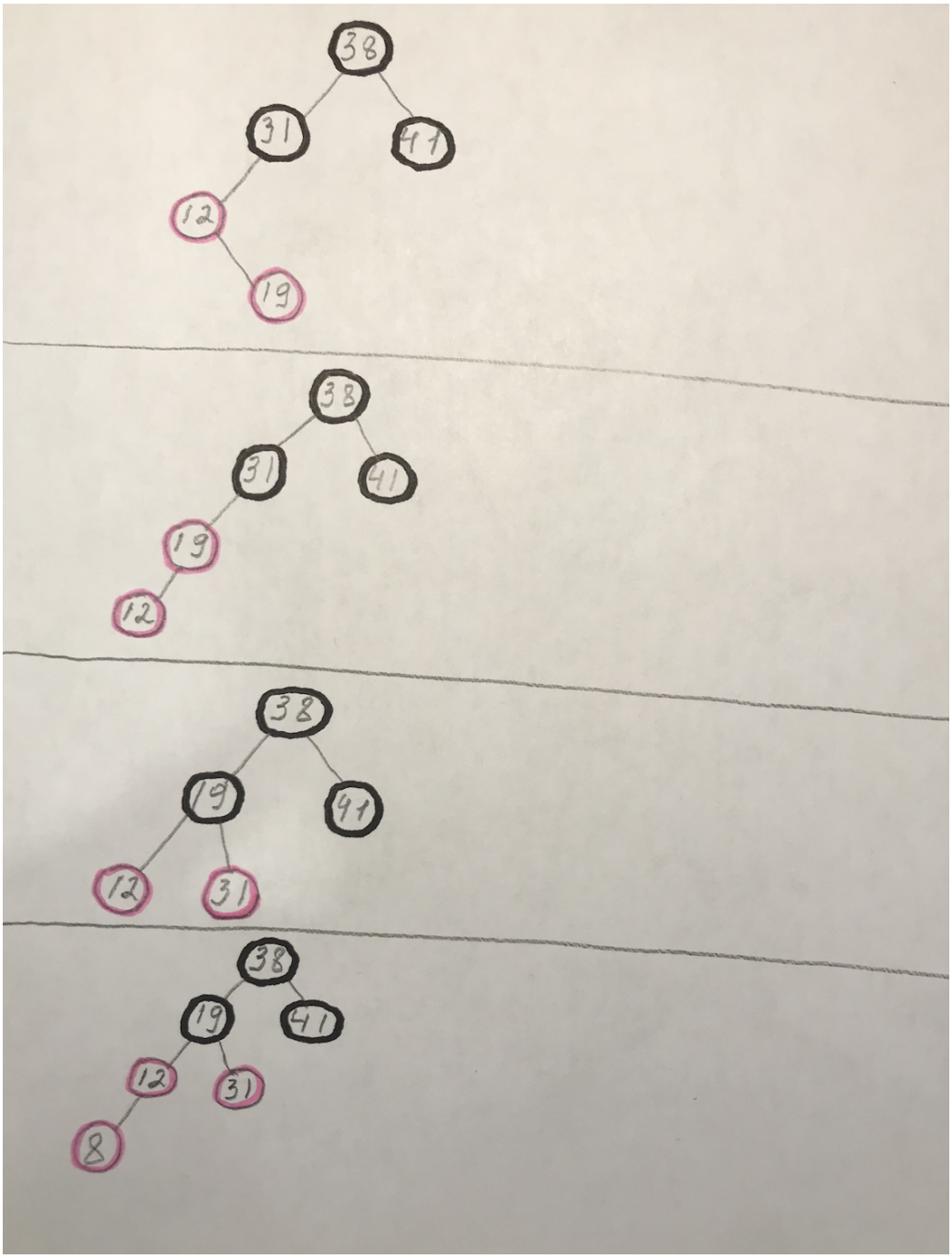
Question 1. Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array $A =< 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 >$.

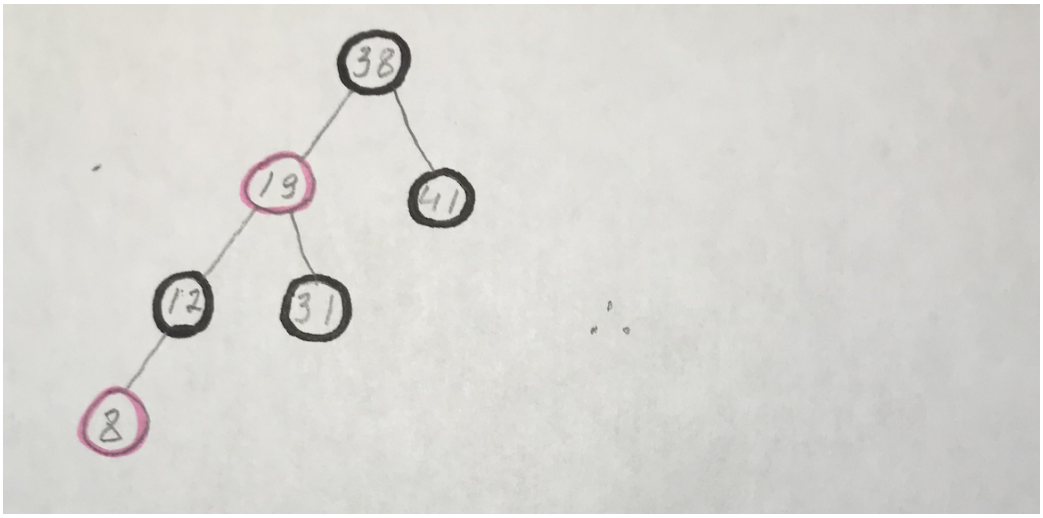



8
B 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 2 3 3 6
C 0 1 2 3 4 5 6
2 3 5 6 8 9 10
9
B 1 2 3 4 5 6 7 8 9 10 11
1 1 2 3 3 4 6
C 0 1 2 3 4 5 6
2 2 5 6 8 9 10
10
B 1 2 3 4 5 6 7 8 9 10 11
0 1 1 2 3 3 4 6
C 0 1 2 3 4 5 6
1 2 5 6 8 9 10
11
B 1 2 3 4 5 6 7 8 9 10 11
0 1 1 2 2 3 3 4 6
C 0 1 2 3 4 5 6
1 2 4 6 8 9 10
12
B 1 2 3 4 5 6 7 8 9 10 11
0 0 1 1 2 2 3 3 4 6
C 0 1 2 3 4 5 6
0 2 4 6 8 9 10
13
B 1 2 3 4 5 6 7 8 9 10 11
0 0 1 1 2 2 3 3 4 6 6
C 0 1 2 3 4 5 6
0 2 4 6 8 9 9
B 1 2 3 4 5 6 7 8 9 10 11
0 0 1 1 2 2 3 3 4 6 6
∴

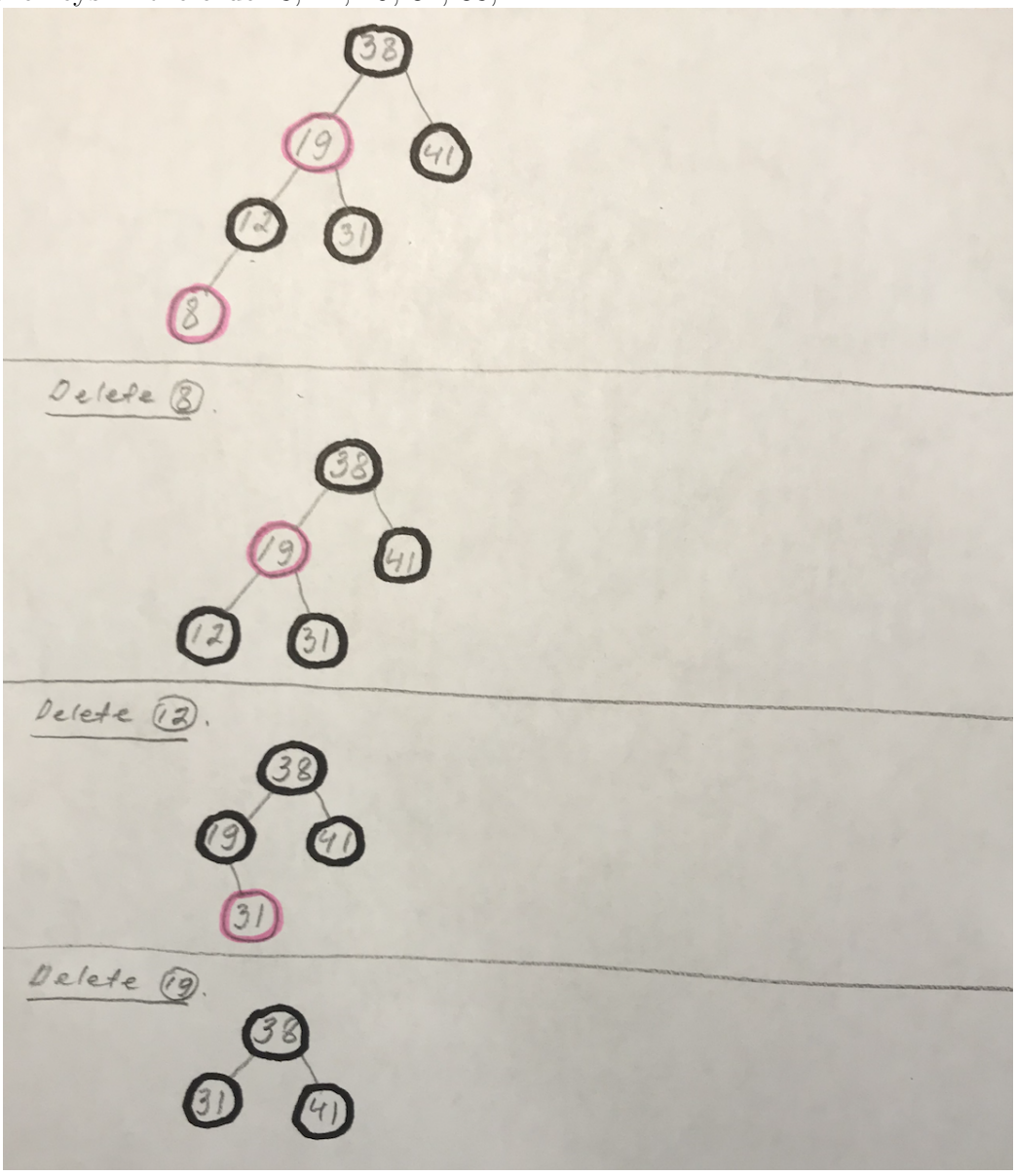
Question 2. Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

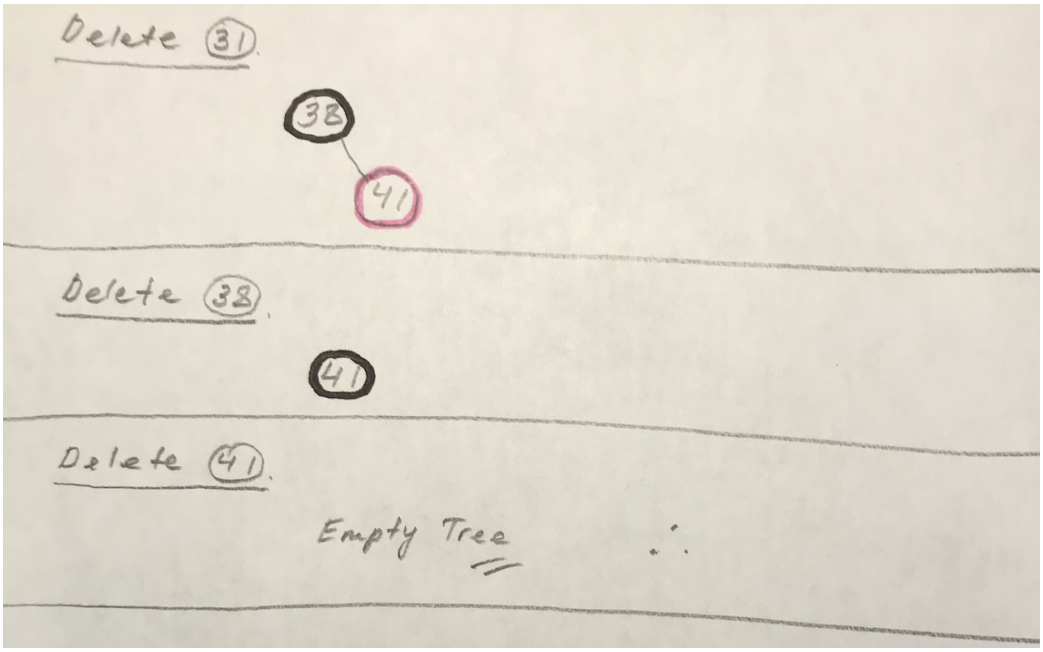



38
31
41
12
19
38
31
41
19
12
38
19
41
12
31
38
19
41
12
31
8


38
19
41
12
31
8

Question 3. In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.




Delete 31
38
41
Delete 38
41
Delete 41
Empty Tree
∴

Question 4. Given n elements and an integer k. Design an algorithm to output a sorted sequence of smallest k elements with time complexity $\theta(n)$ when $klog(n) \leq n$.

Solution:

I will explain the logic behind my algorithm design. I will use Max Heap data structure.

Firstly, we will pick first k elements of the array A from $A[0]$ to $A[k-1]$ and then make a max heap over these elements. This part of the algorithm will take $\theta(k)$, since we pick first k elements of the array A.

Secondly, we compare each element of the array A after kth element, so from $A[k]$ to $A[n-1]$ with the root of the max heap. Here, if the current element of the array is $<$ than the root element of the max heap, then we make the current element the root and not forget to heapify the max heap to obey its' property. Else, we just ignore the current element and go on to traverse other elements. At this point we will have $\theta((n-k)logk)$ time complexity.

Lastly, the root of the max heap will eventually become the kth smallest element and, thus, all other elements of the max heap will be the smallest k elements of array A.

Thus, we have the $\theta((n-k)logk)$ so far, however, this is for the unsorted output. The time complexity for the sorted output will then be $\theta((k+(n-k)logk)$.

Question 5. Design an **efficient** data structure using (modified) red-black trees that supports the following operations:

Insert(x): insert the key x into the data structure if it is not already there.

Delete(x): delete the key x from the data structure if it is there.

Find_Smallest(k): find the kth smallest key in the data structure.

What are the time complexities of these operations?
Solution:

As mentioned in the forums on OWL by one of the TAs, we need to explain how to modify red-black trees and describe each operation with the time complexity.

This is definitely a tough question, but I will try my best to answer it. I assume that since we have to use the red-black trees, then we can modify/create its' operations in a way that we still get an efficient time complexity. I think, it is important to start with operation finding the smallest key and then it should be more clear as to what should be done with insertion and deletion. In order to hopefully guarantee efficiency of the data structure, we could add a new piece of information to each node in a red-black tree such as the size field. This piece of information should help to allocate the kth node and, thus, the smallest one. So, if the number of elements in the left subtree is $k-1$, then root will be the kth element. If the number of elements in the left subtree is $> k-1$, then the kth element will be the kth element in the left subtree. Finally, if the number of elements in the left subtree is, say n, and $n < k-1$, then the kth element would be the $k-(n+1)$th element in the right subtree. I think that based on this, we will be able to allocate the smallest in $\theta(lgn)$ time complexity since $\theta(height)$ will be equal to $\theta(lgn)$. Forgot to mention that we should also have the rank assigned to each node so that we are able to find the smallest one, i.e. with the smallest rank. But, then we should be able to update this information without having to implement any additional operations. Well, we could do the updates when inserting/deleting the nodes. Thus, when inserting, the leaf will be added and the information on the path from leaf to root can be updated as well. This should still give the $\theta(lgn)$ time complexity. For deleting, we could update the information

on a path from the node that was deleted to the root. This will also give the $\theta(lgn)$ time complexity. Thus, we get $\theta(lgn)$ time complexity for all operations.

Question 6. Prove that every node has rank at most $\lfloor \lg n \rfloor$.

Proof.

Induction base: our base case will be the node of rank 0, which is the root of the subtree that at least contains itself. Therefore, the size of it is at least 1.

Induction hypothesis: we can assume that a node in the tree that has rank r, the size of the sub-tree with the node as root will be at least 2^r.

So, we assume that a node X, can have rank $(r + 1)$ if and only if it had the rank r and thus, it was the root of the tree at the prior step that was joined with another tree whose root had rank r.

Therefore, the X node will now be the root of the union of two trees. And, by the induction hypothesis we know that each tree that has at least 2^r size and thus, X being the root of the tree whose size is then:

$$2^r + 2^r = 2^{r+1}$$

Then, in the forest, we will have the total number of nodes equal to n and have at least 2^r nodes in every tree with the rank being equal to r. Therefore, putting it all together, we get:

$$n \geq 2^r$$
$$r \leq \lfloor \log_2 n \rfloor$$

Therefore, we proved the for every node in the union find algorithm, the rank will be at most $\lfloor \lg n \rfloor$.

Question 7. In light of Exercise 21.4-2, how many bits are necessary to store $x.rank$ for each node x?

Well, we already know that the rank of a node is at most the height of the subtree that is rooted at such node meaning that the number of nodes in a subtree will be N at most. Thus, the x field in each node has to be $\theta(logN)$ bits wide in order to store numbers from 0 to N. Therefore, we can represent the nodes using $\theta(log(logn))$ bits and thus, we might have to use that many bits for the number that will be taking that many values.

Will come back later for number 10.a.

Question 8. Prove that if we order the characters in an alphabet so that their frequencies are monotonically decreasing, then there exists an optimal code whose codeword lengths are monotonically increasing.

This proof can be done by the contradiction. We can assume that:

$$c_1.freq \geq c_2.freq \geq ... \geq c_n.freq$$

and

$$d_T(c_1) > d_T(c_2) > ... > d_T(c_n)$$

Thus, we can take c_n and c_{n-1}, because they will have the greatest depth and this it implies the following:

$$d_T(c_{n-2}) \leq d_T(c_{n-1})$$

what will give us the needed contradiction. Hence, proved.

Also, if we just look at this analytically, we are dealing with a similar situation as the normal Huffman, however, the difference is that we initially begin with choosing the characters x with the smallest frequency. Then, we make x the child of a new node, say z, as well as the second to last character, say y, with the smallest frequency the two children of another new node, say w. Then, we would repeat with the third to last, fourth to last, etc. The resulting output will produce the increased code length for decreasing efficiency.

Question 9. Suppose we have an optimal prefix code on a set $C = \{0, 1, ..., n-1\}$ of characters and we wish to transmit this code using as few bits as possible. Show how to represent any optimal prefix code on C using only $2n - 1 + n\lceil \text{lg n} \rceil$ bits. (*Hint*: Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

We need to show that any optimal prefix-free code on C can be represented using only $2n - 1 + n\lceil \text{lg n} \rceil$ bits.

An optimal prefix-free code on C will have a related full binary tree that has has n leaves and $n-1$ internal vertices. We observe that any full binary tree will have exactly $2n - 1$ nodes and the height of such tree will be $\lceil \text{lg n} \rceil$. Then, in order to associate the n members of $C = \{0, 1, ..., n-1\}$ with the n leaves of the tree, it can be achieved by listing them in the order that preorder traversal will go through.

Thus, $\lceil \text{lg n} \rceil$ bits should be enough to represent each of the n members of C and, therefore, no delimiters will be required if each member is allocated using $\lceil \text{lg n} \rceil$ bits. So, for n leaves it will need $\text{n}\lceil \text{lg n} \rceil$ bits for representation.

Since, it will require n for the leaves, $n - 1$ for the internal node, then the total amount of bits that will be needed for representation of the optimal prefix-free code on C is:

$$\begin{aligned} &= n + n - 1 + n\lceil lgn \rceil \\ &= 2n - 1 + n\lceil lgn \rceil \end{aligned}$$

Question 10.
Unfortunately, I ran out of time and was only able to partially do part a. I started in advance, but have 3 exams this week what makes it pointless to even use the self-report or take the late penalty. Apologize for extra info..