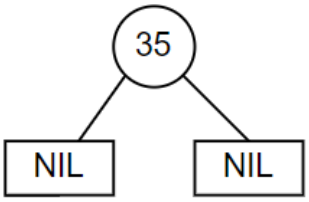
# 3340 Assignment 2

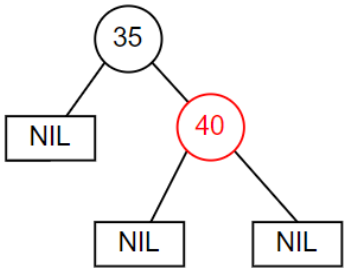
Arsalaan Ali

**Question 1:**

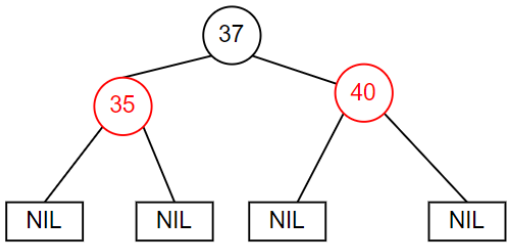
*Insert 35:*



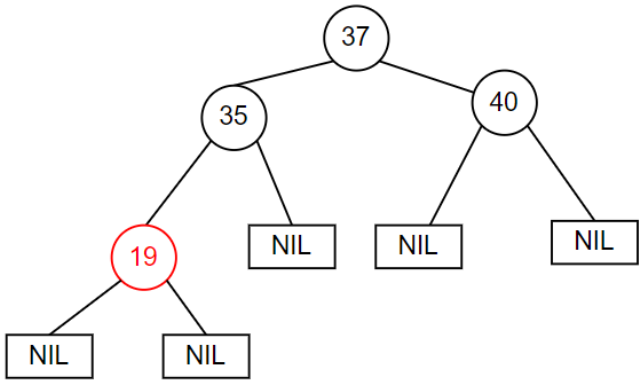

*Insert 40:*



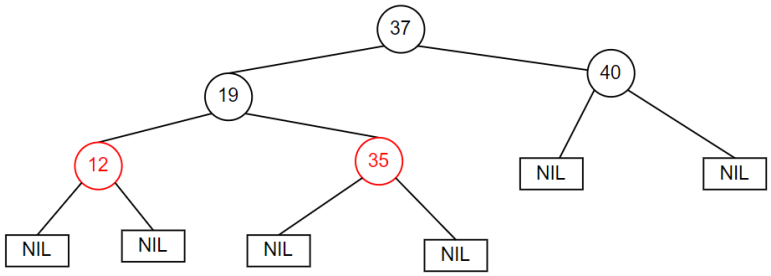
*Insert 37:*



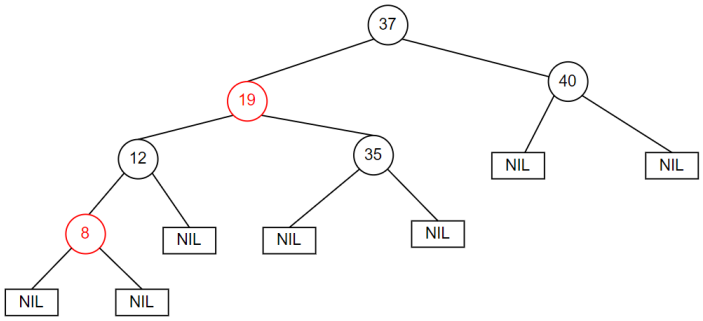

*Insert 19:*



*Insert 12:*

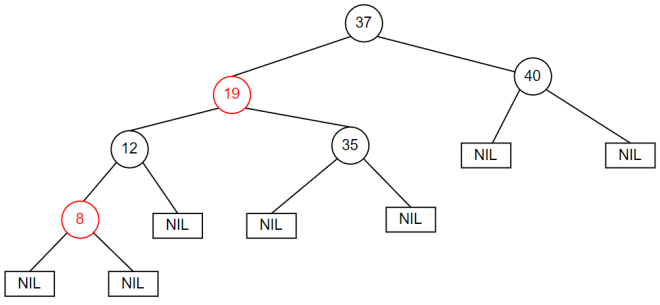



*Insert 8:*

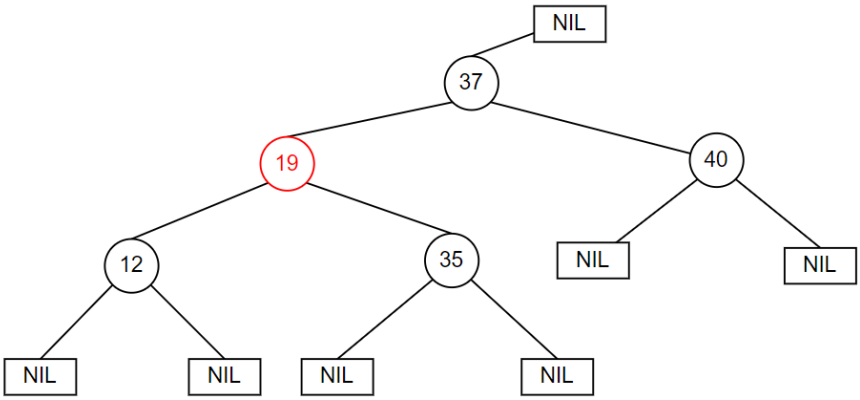



**Question 2:**

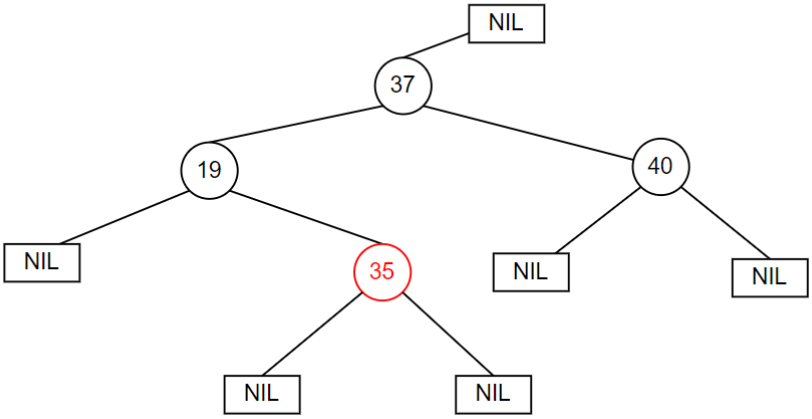
*Initial Tree:*



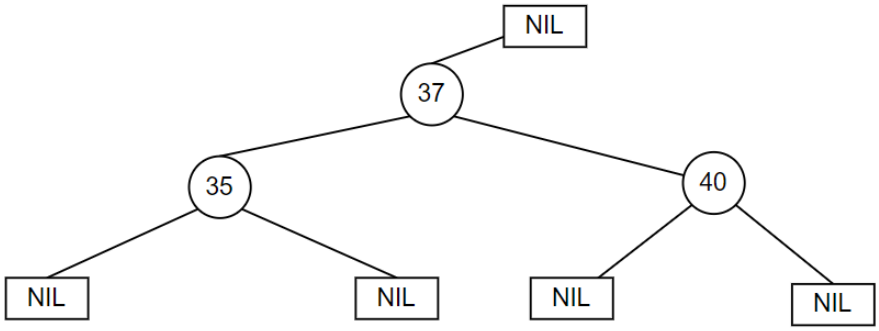

*Delete 8:*



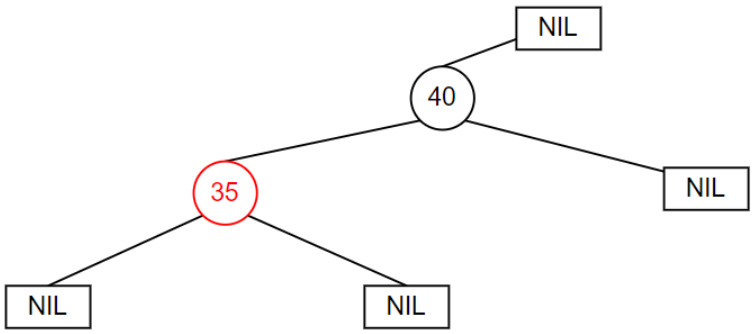

*Delete 12:*



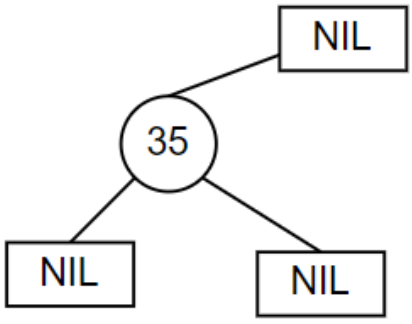

*Delete 19:*



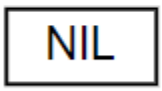

*Delete 37:*




*Delete 40:*




*Delete 35:*




***Question 3:***

1. Describe your algorithm in English (not with pseudo code)

My algorithm merges the k sorted lists into one list using a heap to keep track of the smallest elements in each list:

**Initialization**:

A heap is set up to track the smallest elements from each of the k sorted input lists

Initially, the first element of each list is inserted into the heap. The heap entries consist of the element's index within its list and the list's identifier to facilitate fetching subsequent elements.

**Merging Process**

The algorithm repeatedly removes the smallest element from the heap (which represents the smallest among the current heads of all lists) and adds it to the result list.

It then fetches the next element from the same list as the removed element and inserts it into the heap, maintaining the heap's order.

If the extracted element was not the last element in its respective list, insert the next element from that list into the min-heap.

This process continues until the heap is empty, indicating that all elements from all lists have been processed and inserted into the result list in sorted order.

2. Show why the algorithm is correct.

- Heap-Property: The min-heap is set up to carry, at any moment, not more than one item from each of the k lists. This setup ensures that the tiniest item among these lists is consistently on the heap's summit. This is essential for ensuring that we can always pick the smallest available item for the combined list.

- Maintaining sorted order: The algorithm maintains the sorted order of the merged lists by choosing which elements to insert into the heap based on the last element in each list. Whenever the smallest element is removed from the heap and added to the merged list, the next element from the same list is added into the heap. This approach ensures that the relative order of elements from any individual list is preserved in the merged list, thus maintaining the sorted order.

- Termination: The algorithm concludes once the min-heap is empty, meaning that all of the items from the lists have been merged.

3. Analyze the time-complexity of the algorithm.

- Heap Operations: The important operations on the min-heap are 'insert' and get-min', each of these has a time complexity of lg(k) due to the heap containing at most k elements at any point. This complexity arises because, in a heap, the time it takes to insert an element or to extract the minimum element is proportional to the height of the heap, which is for a heap containing elements.

- Total Operations: Since all n elements from the lists are inserted into and then gotten from the heap only once, there are a total of heap 2n operations done.

- Overall Complexity: Given that each heap operation takes log(k) time and these operations are done 2n times, the overall time complexity of the algorithm is O(n lg k).

**Question 4:**

1. Describe your algorithm in English (not with pseudo code)

- The algorithm takes a list of n integers, each falling between 0 and k, and sets it up to efficiently compute the quantity of numbers within a given range [a:b]. It begins by forming an array, referred to C, in which each index corresponds to a number between 0 and k, and the value at each index reflects the total count of integers from the original list that do not exceed that index number.
- This configuration is achieved by iterating over each integer in the original list and increasing the count in C for that integer and for every larger integer. Following this initial tallying stage, the algorithm transforms the counts in C into cumulative figures.
- For any query about the count of numbers within a specific range [a:b], the algorithm merely deducts the cumulative count immediately before a (which is C[a-1]) from the cumulative count at b (which is C[b]). This subtraction yields the precise count of numbers lying within the range [a:b]

2. Show why the algorithm is correct.

- The reason this algorithm is correct is because firstly, the counting stage guarantees that for each number i, C[i] precisely reflects the count of integers in the input list that are less than or equal to i. This is achieved by incrementing the count for each integer in the list at the corresponding index in C and for all subsequent indices, ensuring an accurate representation of each integer's occurrence.
- Secondly, during the cumulative sum phase, C is updated in a manner where C[i] encompasses not just its own count but also the cumulative totals of all previous counts

up to C[i]. As a result, for any specified range [a:b], C[b] encompasses the total count of integers up to and including b, whereas C[a-1] accounts for all integers below a.
- Therefore, when C[a-1] is subtracted from C[b], it effectively filters out all integers below the lower limit a, retaining only the count pertinent to the range [a:b]. The meticulous construction of C ensures that this straightforward subtraction operation provides an accurate count for the specified range.

3. Analyze the time-complexity of the algorithm.

- The time complexity analysis of the algorithm is split into two main components:
    - Initially, the algorithm traverses through the entire list of n integers to establish the initial counts, a process that requires Θ(n) time.
        - Following this, it executes a cumulative sum operation across the array C, which has a length of k+1, necessitating Θ(k) time.
        - Consequently, the preprocessing stage possesses a time complexity of Θ(n+k).
    - Given that the array C has undergone preprocessing, responding to each query becomes a matter of a simple subtraction, achievable in O(1) time.
- Hence, the time complexity of the algorithm is primarily influenced by its initial time, amounting to Θ(n+k), while the time required for answering queries remains constant, independent of the number of queries.

**Question 5:**

To prove that every node has rank at most [lg n] by using strong induction on the number of nodes we will use

If n=1, then that node has a rank equal to 0:

Therefore, lg 1. Now, suppose that this claim holds for 1, 2, … n nodes.

Given n+1 nodes, if we perform a union operation on two disjoint sets p and q nodes respectively, where p, q <= n.

Then the root of the first set has rank at most lg p.

And the root of the second set has rank at most lg q.

If the ranks are unequal, then the UNION operation keeps the rank and therefore we are done, since it is lg n, and therefore the ranks are equal.

If the rank increases of the union increases by 1, then the resulting set has the rank:

lg a + 1 <= [lg(n+1) /2]+1 <= [lg(n+1)]

Therefore, every node has rank at most

lg(n)

**Question 6:**

We know that the maximum rank of any node is $\lfloor \log_2(n) \rfloor$, it follows that they can be encoded using O(lg(lg(n))) bits. Consequently, representing a value that encompasses this range necessitates O(lg(lg(n))) bits as well.

Regarding question 8a), it is enough to hold the rank. Given that the upper limit for rank is $\lfloor \log_2(n) \rfloor$, and considering n equals 5400 (reflecting the image dimensions of 72*75), $\lfloor \log_2(n) \rfloor$ computes to 12. This value of 12 can be accommodated within a single byte, as a byte is capable of representing up to 256 distinct integers.

**Question 7:**

Given a set C consisting of n characters, represented as C = {0, 1, ..., n-1}, we can encode an optimal prefix-free code for C using a total of $2n-1 + n\lceil \log_2(n) \rceil$ bits. An optimal prefix-free code for C is characterized by a corresponding full binary tree with n leaves and n-1 internal nodes.

This tree can be represented by a sequence requiring 2n-1 bits. The height of this full binary tree is $\lceil \log_2(n) \rceil$, which means a path from the tree's root to any leaf can be encoded using $\lceil \log_2(n) \rceil$ bits.

To link the n elements of C {0, 1, ..., n-1} with the n leaves of the tree, we can list them in the order they are encountered during a preorder traversal of the tree. Consequently, $\lceil \log_2(n) \rceil$ bits are sufficient to represent each element in C, eliminating the need for delimiters by allocating $\lceil \log_2(n) \rceil$ bits for each element.

For the n leaves, representing each element in C requires $n * \lceil \log_2(n) \rceil$ bits. Hence, the total number of bits required to represent an optimal prefix-free code on C is:

$C = n\_leaves + n\_internal\_nodes + n * \lceil \log_2(n) \rceil$

$= 2n - 1 + n\lceil \log_2(n) \rceil$ bits

**Question 8a)**

1. The algorithm final_set(a) initiates by iterating over every node within the disjoint set. For each node, it executes the find_set() function. Owing to the implementation of path compression, the array holding the parents will have the parent of the ith node at the ith position. Whenever a set is encountered for the first time, its parent is stored, and a new label ranging from 1 to n is assigned to it in a hashmap. This hashmap serves to track the newly assigned labels for each set. Subsequently, the algorithm traverses the array once more, and this time, it updates the parent of each node to its newly assigned label, as specified in the hashmap.
2. The correctness of the algorithm is ensured through the application of the find_set() function to each node, which employs path compression across all nodes. This technique ensures that each entry in the parents' array directly points to the ultimate parent, guaranteeing that all members within a distinct set share the same parent. These

parents are then systematically mapped to a new label within the 1-n range. The rationale behind this approach's validity lies in the fact that, following the process, every node is directly associated with its newly assigned label, ensuring consistency and correctness in the representation of sets.

3. The time complexity of the algorithm is analyzed as O(n log n). This is due to the path compression operation, which has a time complexity of log n for a single element. Since path compression is applied to each of the n elements in the set, the overall time complexity becomes n * log n, leading to the conclusion that the algorithm operates in O(n log n) time.

**Question 8b)**

**Describe your algorithm in English (not with pseudo code)**

To create the disjoint set we create a disjoint set the size of 72*75 (number of characters in the image), we then use a nested for loop to go through each of the characters in the image, for each character we check if it is a '+' and if it is already part of a set, if it is not then make it its own set. We then loop the 8 adjacent characters to check if any of them are a '+', if they are then we create a union between the set of the adjacent character and the set of the current character.

**Show why the algorithm is correct.**

This algorithm goes through each of the characters in the image and connects them to the sets of the characters beside them, this means that any of the characters in the same connected component are going to be in the same disjoint set. So when we do path compression and the final_sets() function they are going to be pointing to the same label. We can use these labels to do parts 2-5.

**Analyze the time-complexity of the algorithm.**

The complexity of creating a disjoint set for each of the characters in the image is O(n), since you have to loop through each character in the image.

The time complexity of doing path compression on the nodes is O(lg n) and worse case we would have to do path compression on almost every node, so that would be O(n lg n)