Student Name: Ashna Mittal

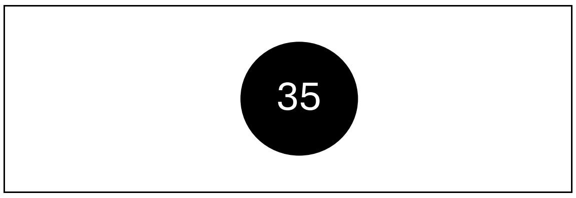
Student Number: 251206758

CS 3340B: Assignment 2

Solution 1: The steps for inserting 35, 40, 37, 19, 12, 8 into an initially red-black tree are:

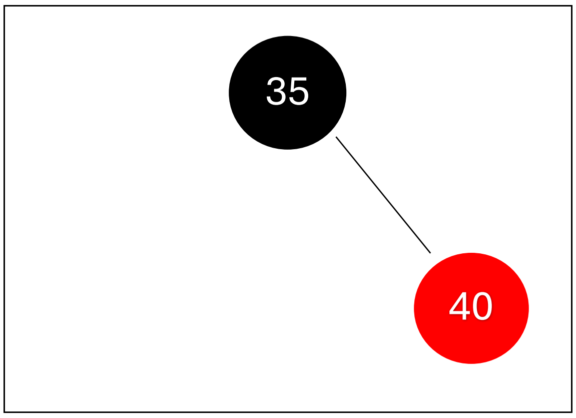
- Step 1: Insert 35

  Since the tree is initially empty, a new node will be created as the root node with the color black.

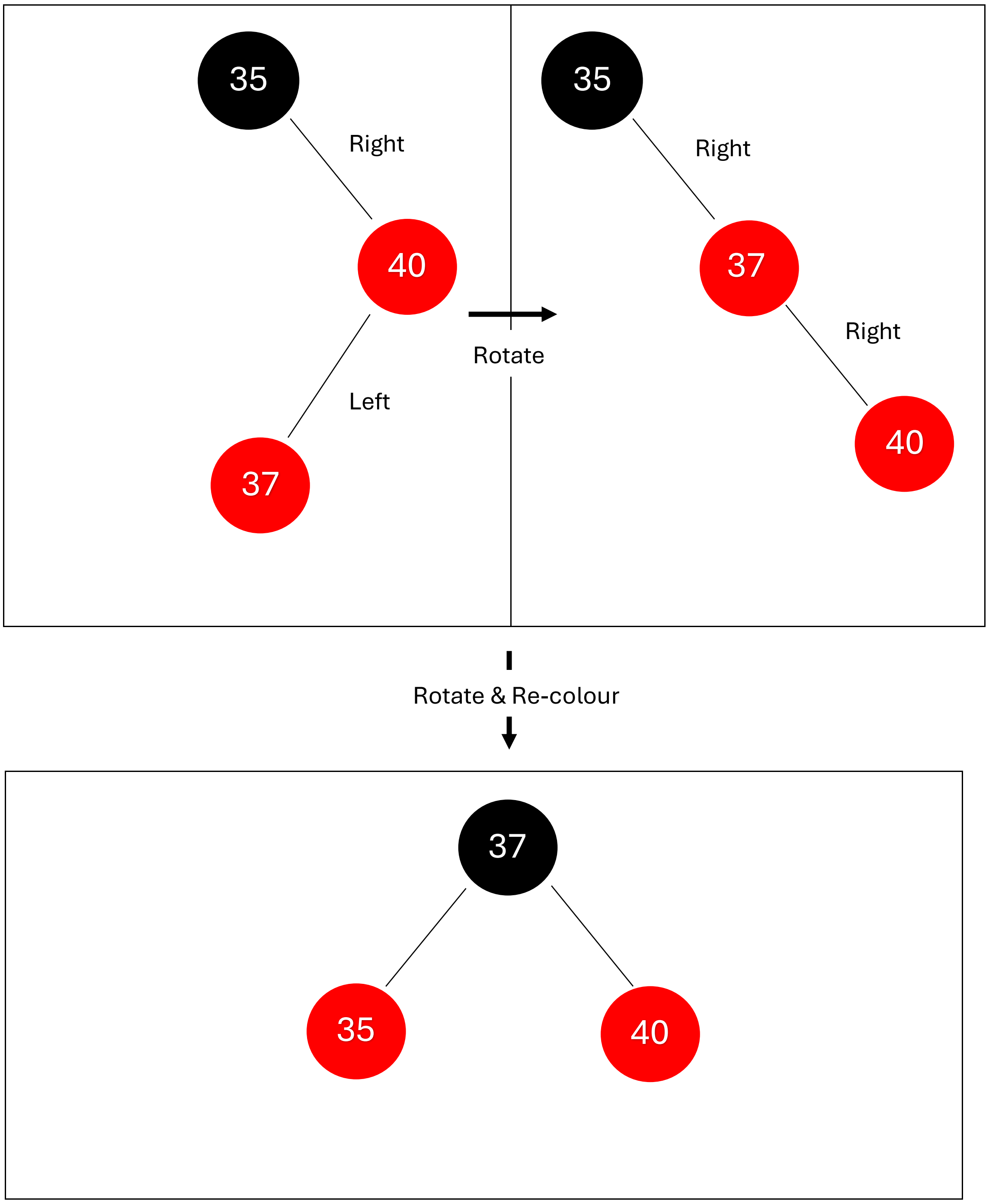
- Step 2: Insert 40

  Since the tree is not empty, a new leaf node will be created with the color red. Since 40 > 35, It will be inserted in the right subtree as per the properties of the red-black tree.

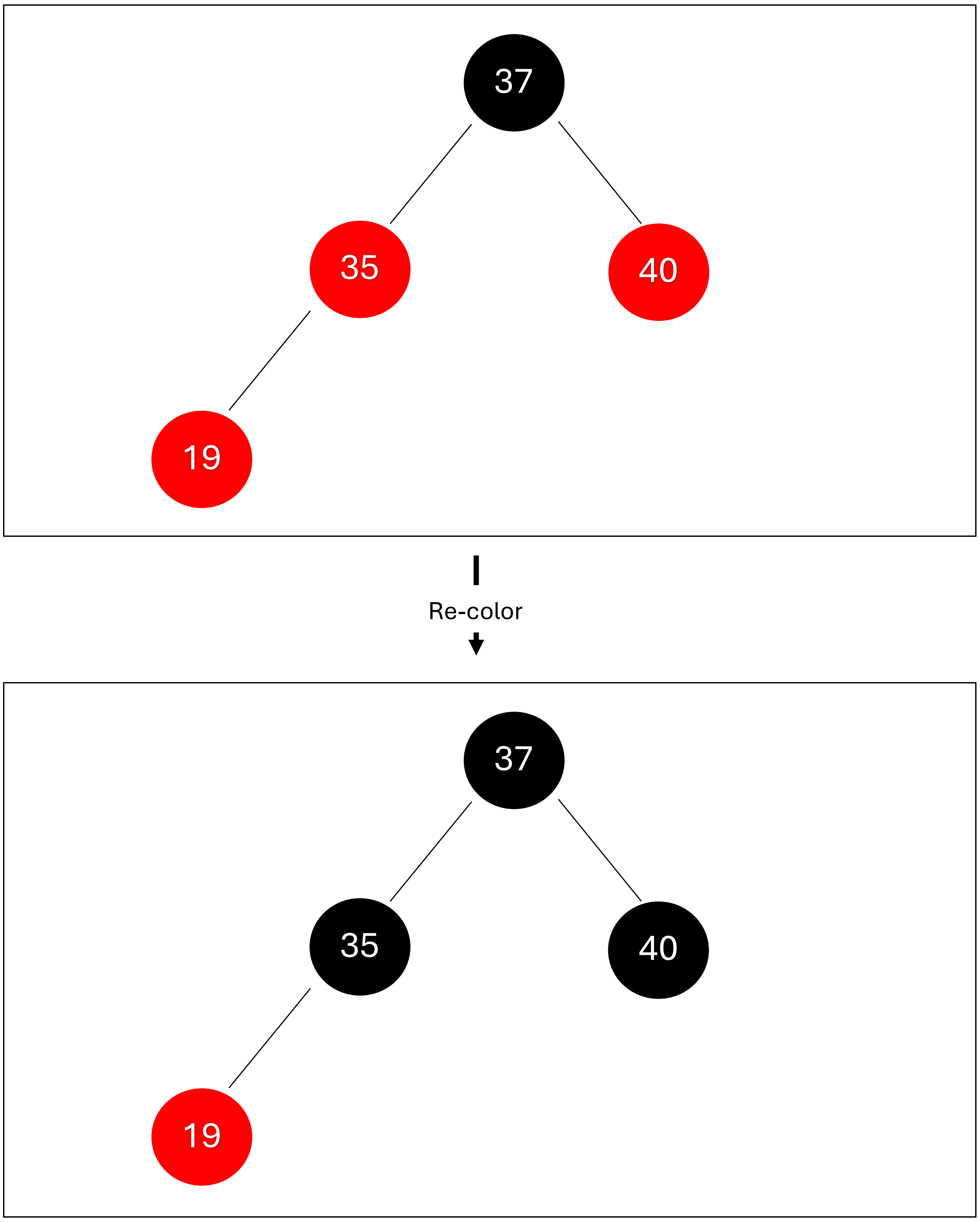
- Step 3: Insert 37

  Since the tree is not empty, a new leaf node will be created with the color red. Since 37 > 35 but 37 < 40, It will be inserted in the left subtree of 40 but the right subtree of 35 as per the properties of the red-black tree. Now since no two adjacent red nodes can be red and the parent of 37, which is 40, is also red, we will check the sibling of the 40. In this case, since 40 doesn't have a sibling, and has null as its sibling, we will do a right rotation. After the right rotation, we will do a left rotation since we still have two adjacent red nodes and null as the sibling. After the rotation is complete, we will re color the nodes where 37 takes the root's position and black color, and 35 and 40 become the left and right red children respectively.



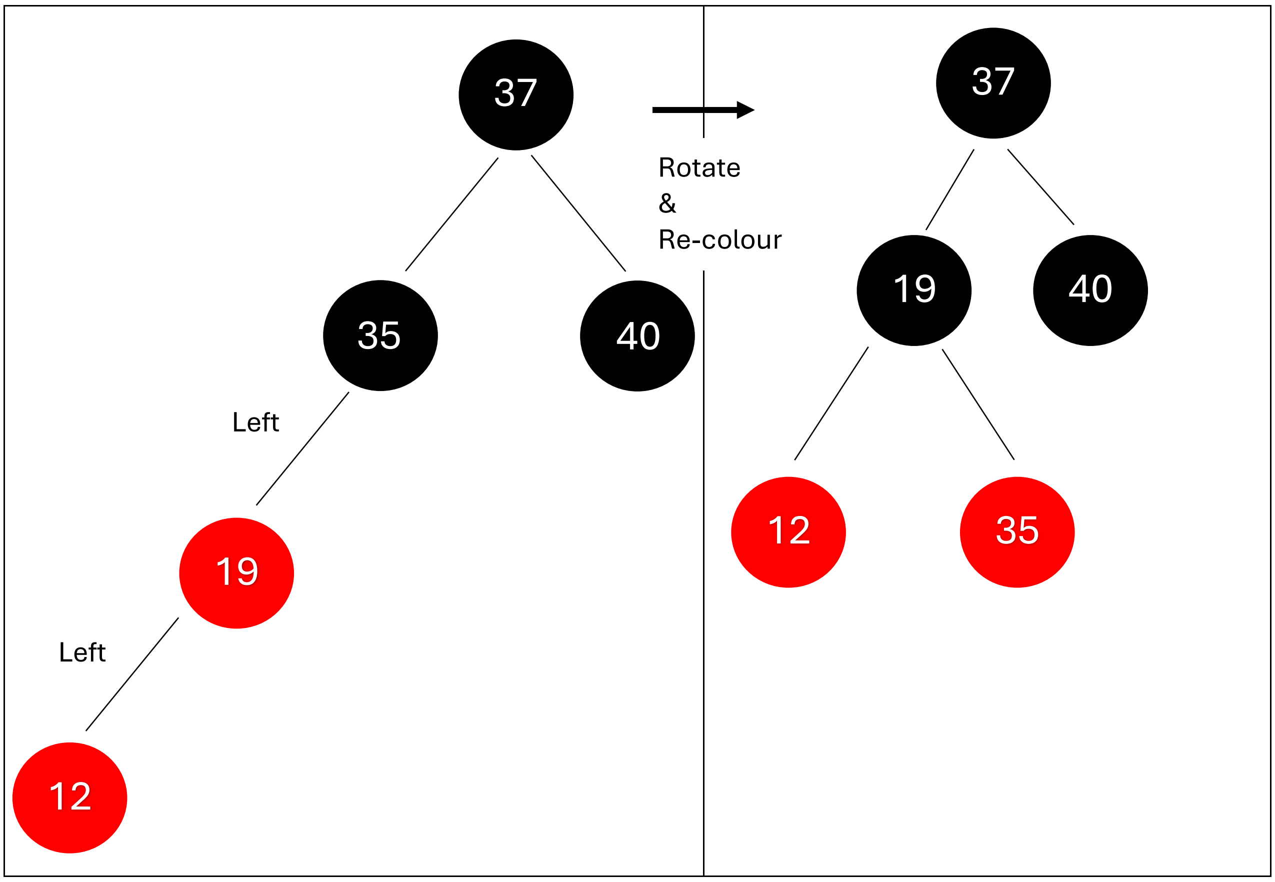
- Step 4: Insert 19

Since the tree is not empty, a new leaf node will be created with the color red. Since 19 < 37 and 19 < 35, It will be inserted in the left subtree of 35 as per the properties of the red-black tree. Now since no two adjacent red nodes can be red and the parent of 19, which is 35, is also red, we will check the sibling of the 35. In this case, since the sibling 40 is also red, we will re color the nodes 35 and 37 as black.



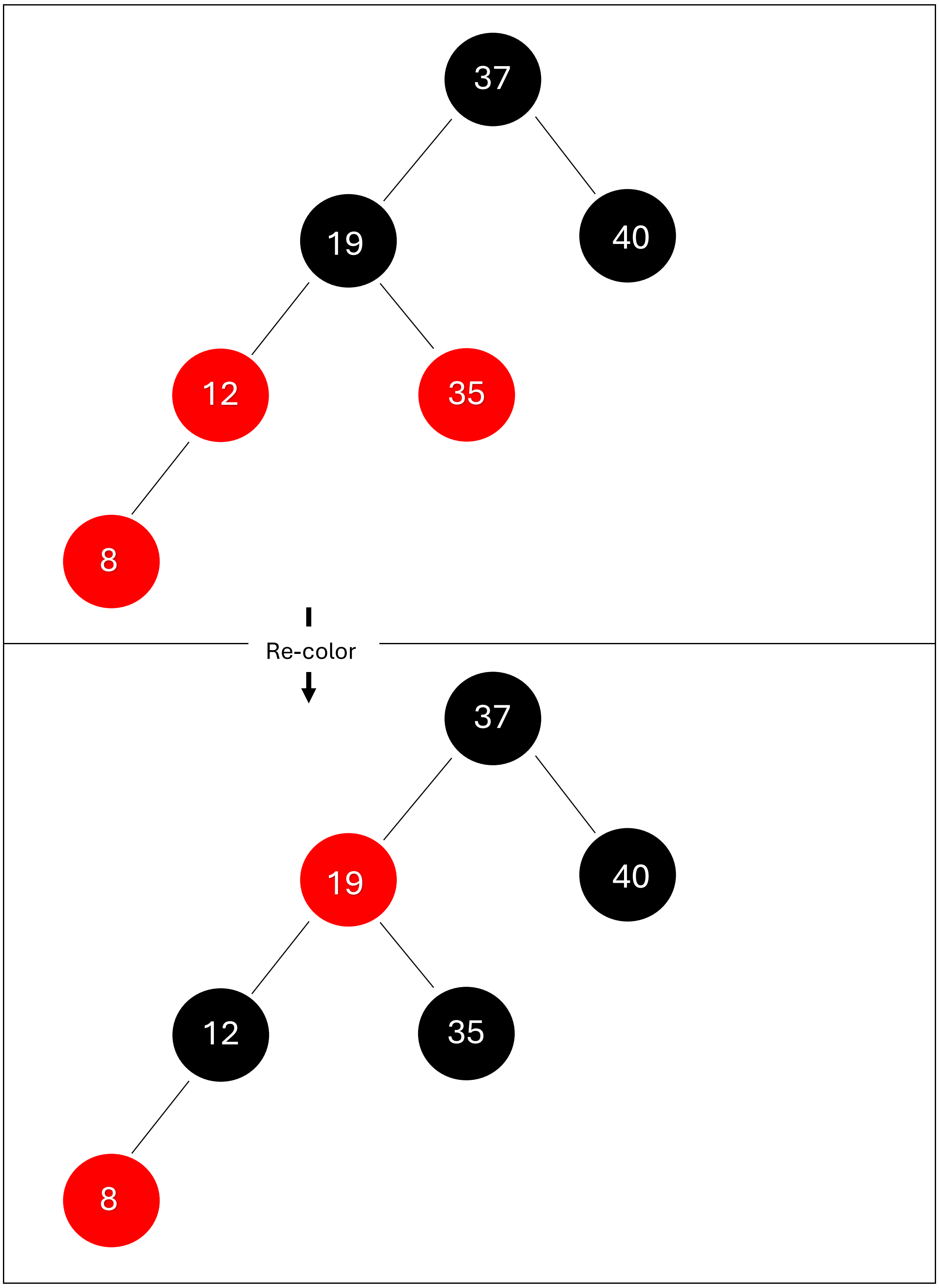
- Step 5: Insert 12

  Since the tree is not empty, a new leaf node will be created with the color red. Since 12 < 19, It will be inserted in the left subtree of 19 as per the properties of the red-black tree. Now since no two adjacent red nodes can be red and the parent of 12, which is 19, is also red, we will check the sibling of the 19. In this case, since 19 doesn't have a sibling, and has null as its sibling, we will do a right rotation. After the rotation is complete, we will re color the nodes where 19 takes the root's position and black color, and 12 and 35 become the left and right red children respectively.



- Step 6: Insert 8

  Since the tree is not empty, a new leaf node will be created with the color red. Since 8 < 12, It will be inserted in the left subtree of 12 as per the properties of the red-black tree. Now since no two adjacent red nodes can be red and the parent of 8, which is 12, is also red, we will check the sibling of the 12. In this case, since the sibling 35 is also red, we will re color the nodes 12 and 35 as black. Now, since the parent's parent, which is 19, is also black but not the root node, we will recolor it as red.

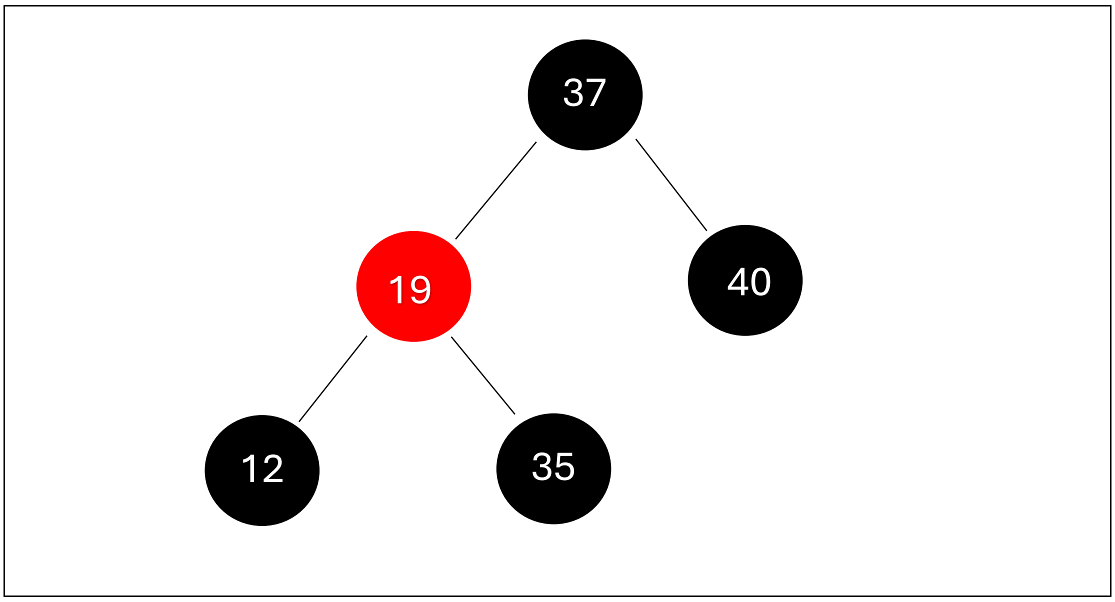


The red-black tree above is our final tree after the insertion is complete.

Solution 2: Deleting the keys 8, 12, 19, 37, 40, 35 from the red-black tree created in Question1.

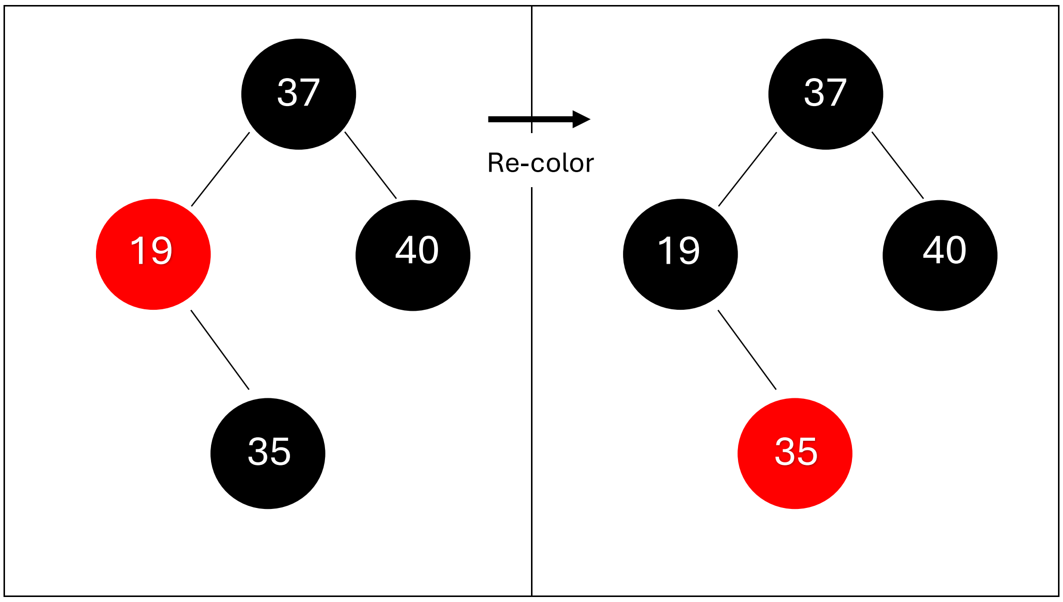
- Step 1: Delete 8

  Since the node 8 which is to be deleted is red, we can just delete it and remove from the red-black tree.



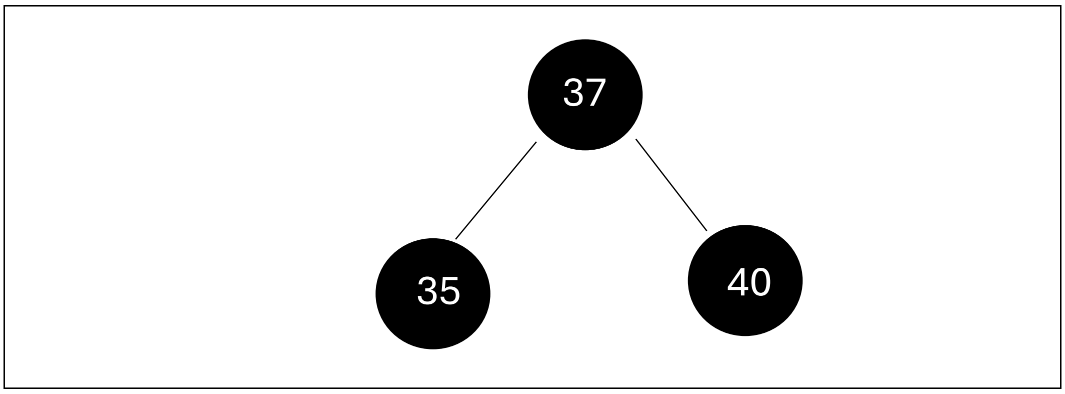
- Step 2: Delete 12

  Since the node 12 which is to be deleted is black, we delete it and then re-color the red-black tree. Now, since the sibling of the deleted node, which is 35, is black, we will recolor the red parent 19 as black and recolor the sibling 35 as red.

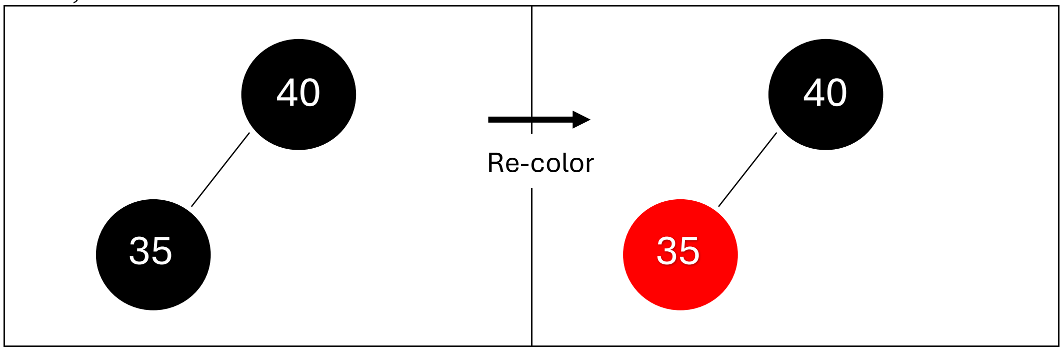


- Step 3: Delete 19

Since the node 19 which is to be deleted is black, we delete it and then re-color the red-black tree. Now, since the right child of the deleted node, which is 35, is red, we will recolor it as black.

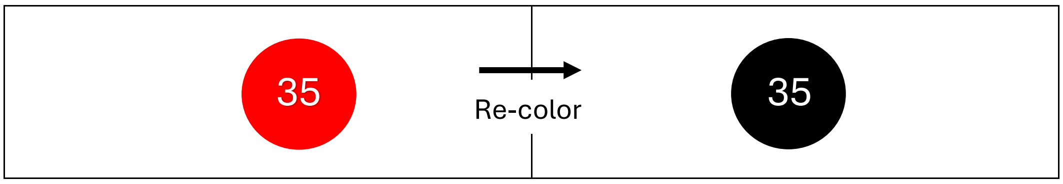


- Step 4: Delete 37
- Since the node 37 which is to be deleted is black, we delete it and then as per the instructions use the smallest in the right subtree. So, we make 40 as the root node and then re-color the red-black tree. Now, since the left child of root node, which is 35, is black, we will recolor it as red.



- Step 5: Delete 40

  Since the node 40 which is to be deleted is black, we delete it and then make 35 as the root node and then re-color the red-black tree. Now, since the leftover root node, which is 35, is red, we will recolor it as black.



- Step 6: Delete 35

  We now delete the last leftover root node 35 and finish the deletion process.

Solution 3:

A min heap is a binary tree where the parent node is always less than or equal to its child nodes. This property makes min heaps ideal for efficiently finding and removing the smallest element. The straightforward approach for this question is to pick the smallest of the top elements in each list, repeatedly. This takes $k - 1$ comparisons per element, in total $O(k \cdot n)$. The detailed steps necessary for this algorithm are:

Step 1: Initialize a min heap
We create an empty min heap and then for each of the k sorted lists, we take the first element (which is the smallest for that list) and insert it into the min heap. While inserting, we store not just the value but also information about which list this element came from and its index within that list. This is crucial for the next steps to know from where to fetch the next element.

Step 2: Process the min heap
We now remove the root of the min heap, which is the smallest element among the current set of elements in the heap. This element is the next in the merged sorted list. Once the element is removed, we need to maintain the size of the heap to be k (or the current number of non-empty lists). To do this, we would have to find the next element from the same list that the removed element belonged to and insert it into the min heap. If the list from which we removed the element is now empty, we simply continue with the remaining elements in the heap. We keep repeating this process of removing the smallest element and inserting the next element from the same list until all elements from all lists are processed and the heap is empty.

Step 3: Keep track of the sorted list
As we remove elements from the min heap, append them to the output list. This list will become our final merged sorted list because at each step, we are always removing the smallest of the available elements from the top of the heap.

Analysis: The heap always gives the smallest element in $O(1)$ time and removing it plus rearranging to maintain the heap property takes $O(lg\ k)$ time since there are at most k elements in the heap. Now, since there are total n elements, and for each element, a delete (and possibly an insert) operation is performed at $O(lg\ k)$ cost, the total time complexity becomes $O(k\ +\ n\ lg\ k) =\ O(n\ lg\ k)$.

Solution 4:

Given n integers in the range $0\ to\ k$, we want to preprocess these integers so that later we can answer any query about how many of the n integers fall into a range [a:b] in $O(1)$ time. The preprocessing time should be $O(n + k)$.

Pre-processing ($O(n + k)$): We construct an array A of length $k + 1$ where each entry $A[i]$ represents the number of integers from the input that are less than or equal to i. This array is initialized with all zeros. We iterate over all n input integers, and for each integer x, we increment $A[x]$. Next, we convert array A into a prefix sum array, where each entry $A[i]$ is transformed to represent the total count of integers less than or equal to i. This is done by iterating over the array and updating each entry $A[i]$ to be the sum of $A[i]$and $A[i - 1]$. This process ensures that after preprocessing, $A[i]$ contains the count of numbers in the range $[0{:}\, i]$.

Answering Queries ($O(1)$): To answer a query for a range $[a{:}\, b]$, we use the prefix sum array A as follows-

- If a > 0, the number of integers in the range$[a{:}\, b]$ is $A[b]$–$A[a - 1]$. This is because $A[b]$ includes all integers up to b, and $A[a - 1]$ includes all the integers less than a. So, their difference gives us the count in the range $[a{:}\, b]$.
- If a = 0, the number of integers in the range $[0{:}\, b]$ is directly $A[b]$, since there are no integers less than 0 to subtract.

Time complexity analysis: Pre-processing step iterates over n integers and then over k entries in the array A, so it takes $O(n + k)$. Each query is answered in constant time $O(1)$ because it involves only a constant number of operations regardless of the size n or k. This meets the requirement of $O(n + k)$ pre-processing time and $O(1)$ query time.

Solution 5:

To prove that every node has rank at most ⌊log n⌋, where n is the number of nodes in a binary tree, we can use induction on the sequence of operations:

Base Case:

Consider a binary tree with a single node (n=1). The rank of this node is 0, which is indeed less than or equal to ⌊log 1⌋ = 0.

Inductive Hypothesis:

Assuming that for any binary tree with k nodes, where k ≤ n, every node has a rank of at most ⌊log k⌋. We need to show that after the k+1th operation, the property still holds.

Union operation: When two sets are unioned, the rank of the resulting set's root is at most one greater than the rank of the highest-ranked root of the original two sets. By the inductive hypothesis, before the union, the ranks of the roots of the two sets are at most ⌊log n1⌋ and ⌊log n2⌋, where n1 and n2 are the sizes of the two sets, respectively.

- If the ranks are equal, the rank of the new root increases by one. Since the size of the new set is n1+n2, we must show that ⌊log (n1+n2)⌋ ≥ ⌊log n1⌋ + 1. This inequality holds because the logarithm function is monotonically increasing, and doubling the size of a set increases the floor of its logarithm by at most 1.
- If the ranks are unequal, the rank of the new root does not increase, and the property trivially holds.

By the principle of mathematical induction, the hypothesis is true for any sequence of operations, proving that every node has a rank of at most ⌊log n⌋.

Alternatively, Inductive Step:

We need to show that for a binary tree with k+1 nodes, every node has a rank of at most ⌊log(k+1)⌋.

Considering the operations that can change the structure of the tree:

1. Insertion of a new node: When a new node is inserted into the tree, it becomes a leaf node. Since we're dealing with a binary tree, each node has at most two children.

Thus, the new leaf node will be added at a level no greater than the height of the tree, which is ⌊log(k+1)⌋. Hence, the rank of this newly inserted node is at most ⌊log(k+1)⌋.

2. Deletion of a node: When a node is deleted from the tree, the tree might undergo rebalancing operations (in case of self-balancing trees like AVL or Red-Black trees). However, these operations maintain the property that the height of the tree remains within certain bounds. Hence, after deletion and rebalancing, every node still has a rank of at most ⌊log(k+1)⌋.

Since both insertion and deletion operations maintain the property that every node has a rank of at most ⌊log(k+1)⌋, by induction, we can conclude that every node in a binary tree with n+1 nodes have a rank of at most ⌊log(n+1)⌋.

Therefore, by induction, we have proved that every node in a binary tree with n nodes has a rank of at most ⌊log n⌋.

Solution 6:

The notation Θ implies that the number of bits required grows proportionally to (Iog(Iog(n))). The rank of a node in a union-find data structure is a measure of its potential height. We've established in the previous question that the rank is at most ⌊log n⌋.

Now, to know how many bits are required to store this rank, where the highest rank is ⌊log n⌋, we need to consider how many different values this rank can take. It can be any integer value from 0 to ⌊log n⌋, so there are ⌊log n⌋ + 1 possible values (including zero).

To determine how many bits we need to represent a range of values, we can use the logarithm base 2. If there are k possible values that the rank can take, we need $\lfloor log_2(k) \rfloor$ bits to represent those values.

Since k in this case is ⌊log n⌋ + 1, we need to find out $\lceil log_2(\lfloor \log n \rfloor + 1) \rceil$ bits.

The Θ (Iog(Iog(n))) notation abstracts away the floors and ceilings for the purpose of asymptotic analysis. It's stating that as n grows, the number of bits required to represent the rank grows like the logarithm of the logarithm of n.

Example:
Let's compute the exact number of bits needed for a specific value of n, say n = 1000, to better understand the calculation.
1. Compute the maximum rank, which is $\lfloor log_2(1000) \rfloor$.
2. Determine the number of possible rank values, which is $\lfloor log_2(1000) \rfloor + 1$.
3. Calculate the number of bits as $\lceil log_2(\lfloor log_2(1000) \rfloor + 1) \rceil$.

Let's perform these calculations.

For n = 1000, the maximum rank is 9, which means there are 10 possible values for the rank (from 0 to 9 inclusive). To represent these 10 possible values, we need 4 bits.

So, in general, the number of bits required to store the rank of each node, when the value is at most $\lfloor \log n \rfloor$, is $\lceil log_2(\lfloor \log n \rfloor + 1) \rceil$, which is asymptotically Θ (Iog(Iog(n))) bits. This asymptotic notation indicates that the number of bits grows slowly relative to the logarithm of the logarithm of the size of n.

In regard to question 8A –
A byte provides 256 different values (from 0 to 255). This range would be enough to cover the rank in nearly all practical applications for the following reasons:

Theoretical Maximum Rank: The maximum rank is limited by the number of elements, but even in a disjoint set with millions of elements, the rank is unlikely to exceed this range. This is because, to achieve a rank of k, the tree must have at least 2^kelements (since the rank increases only when two trees of the same rank are united). For instance, to just reach a rank of 20, a tree must have more than 1 million elements, and achieving a rank of 255 would require an astronomically large number of elements, far beyond practical computational scenarios.

Practical Consideration: In most practical applications, the number of elements n in a disjoint set does not reach a level where a rank would exceed 255. Additionally, path compression keeps the trees very flat, meaning that the rank is an even less accurate measure of tree height and is unlikely to grow very large.

Therefore, Yes, using one byte to store the rank in a disjoint set data structure is generally sufficient due to the slow growth rate of rank and the effectiveness of path compression in keeping the tree height minimal. It's highly unlikely for the rank to exceed 255 in real-world applications, making a byte an efficient and practical choice for this purpose.

Solution 7:

Using a binary tree representation for encoding an optimal prefix code with a given set of characters in a compact way.

A full binary tree with n leaves has exactly 2n - 1 nodes (since every node except the leaves has exactly two children). We can represent each node by a bit: 0 for internal nodes and 1 for leaf nodes. When performing a preorder traversal of the tree (visiting the root node first, then recursively applying the procedure to the left and right subtrees), each internal node will be followed by exactly two other nodes (its children), and each leaf node will not be followed by any children (since leaf nodes have no children).

Basic Step: For a single character set C with n =1, the tree is trivial and consists of only one node, which is a leaf. There's no need to encode the structure since there's only one node, and it's implicit that it's a leaf. The encoding size for the character is $\lceil lg\ 1 \rceil$ = 0 bits. The total encoding size is 0 bits, which satisfies the formula 2n -1 + n$\lceil lg\ n \rceil$ = 2*1 – 1 +1*0 = 1 bit (Which is for the implicit leaf node).

Inductive Step: Assume that for a set C with n characters, we can represent the tree using 2n -1 + n$\lceil lg\ n \rceil$ bits. Now let is consider a set C’ with n+1 characters. By adding one

character, we need to add one leaf node and possibly one internal node to the tree (unless the added leaf is a right child of an existing leaf, in which case the parent leaf becomes an internal node and only one new leaf is added).
With n+1 characters, the number of internal nodes increases to n, and we have n+1 leaf nodes. The tree encoding now takes 2(n+1) – 1 bits. Each character requires $\lceil lg\ (n+1) \rceil$ bits. The total encoding size becomes 2(n+1) – 1 + (n+1)$\lceil lg\ (n+1) \rceil$ bits, which satisfies the induction hypothesis.

We will follow these steps:

1. Tree Structure Encoding (Using 2n - 1 bits):
   - Perform a preorder traversal of the binary tree T.
   - For each node visited, output a 0 if it is an internal node and a 1 if it is a leaf node.
   - This sequence of 0s and 1s will represent the structure of the tree because in a full binary tree, knowing whether each node is internal, or a leaf is enough to reconstruct the tree.
2. Character Encoding (Using n$\lceil lg\ n \rceil$ bits):

- Each character from the set C can be represented using $\lceil lg\ n \rceil$ bits. This is because $\lceil lg\ n \rceil$ is the smallest number of bits that can represent n distinct values (from 0 to (n-1)).
- Encode each character in the order it appears in the preorder traversal of T.
- Since we have n characters, the total number of bits used to encode all characters will be n $*\lceil lg\ n \rceil$

3. Total Bits Calculation:

- The total number of bits used is the sum of the bits used for the tree structure and the bits used for encoding the characters.
- Therefore, the total number of bits is 2n - 1 (for the tree structure) plus n$\lceil lg\ n \rceil$ (for the characters).

The total is indeed 2n - 1 + n$\lceil lg\ n \rceil$ bits.

Suppose n = 4, we would have:

- Tree Structure Encoding: For example, a tree could be encoded as 00101011, where 0s are internal nodes and 1s are leaves.

- Character Encoding: If $\lceil lg\ 4 \rceil = 2$, each character is encoded in 2 bits. Suppose the characters in preorder traversal are 2, 0, 1, 3, they would be encoded as 10 00 01 11.

Combining these, the entire encoding for the tree and characters would be 00101011 10 00 01 11.

Solution 8:

(a) Correctness: The final_sets() method operates correctly within the context of a disjoint-set data structure for the following reasons:
- Identifying Root Elements: It correctly identifies the root elements of disjoint sets by checking if i == find_set(i). This condition ensures that it only works with the representative elements of each set, which are their own parents.
- Assigning New Representatives: It assigns new, sequential representative numbers to each root element. This step ensures that each disjoint set is represented by a unique, compact identifier, starting from 1.
- Updating Parent References: It then updates the parent of each element to directly point to the new representative. This is done by iterating over all elements and setting their parent to the new representative obtained through find_set(i), ensuring that the structure remains consistent and all elements within the same set point to the same new representative.
- Returning the Count: Finally, it returns the current count of disjoint sets, which is maintained and updated correctly during union operations. Since the method does not perform any operation that would change the number of disjoint sets, the count remains accurate.

Complexity:
- The method iterates over all elements twice. The first iteration is to assign new representative numbers to root elements. The second iteration is to update all elements to point to the new representatives. The find_set() method is called for each element during both iterations. Assuming path compression is used (as indicated in the find_set() method), the average time complexity of find_set() is nearly constant, O(α (n)), where α (n) is the inverse Ackermann function, which grows very slowly and is considered practically constant.
  Therefore, the overall time complexity of the final_sets() method can be approximated as O(n), where n is the number of elements in the set.
- The method uses an additional array, newRepresentatives, of size n to keep track of the new representative numbers. This means the space complexity is O(n).

  The final_sets() method is correct in its functionality to consolidate disjoint sets and return the final count of these sets. Its time complexity is linear, O(n), making it efficient for use in consolidating and finalizing the state of the disjoint set data structure. The space complexity is also linear due to the temporary array used for new representative assignments.

(b) The algorithm employs a two-pass approach using a disjoint-set data structure (also known as a union-find data structure) for efficient component identification and labelling.

Algorithm description:

- Initialization:
    - Read the binary image from a file. The image is stored as a list of strings, where each string represents a row of the image.
    - Initialize a disjoint-set data structure with a size equal to the total number of pixels in the image. Each pixel is initially considered as a separate component.
- First Pass - Identifying Connected Components:
    - Iterate through each pixel in the image. If a pixel belongs to the foreground ('+'), check its neighbours in all eight directions (to account for 8-connectedness).
    - For each neighbouring foreground pixel, union the current pixel's component with the neighbour's component using the disjoint-set data structure. This effectively groups all connected foreground pixels into the same component.
- Second Pass - Labelling Components:
    - Initialize a labelled image matrix and a map to associate each component's root with a unique label.
    - Iterate through each pixel again. If a pixel is part of the foreground, find its component's root using the disjoint-set and assign or retrieve its label from the map. Label the pixel in the labelled image matrix accordingly.
    - Background pixels are labelled with a distinct character (e.g., a space) to differentiate them from the foreground components.
- Final Output:
    - Print the original binary image, the labelled image, and lists of components sorted by their sizes. Additionally, print filtered labelled images showing only components larger than specified sizes.

Correctness of the Algorithm
The algorithm is correct because it systematically groups all connected foreground pixels into distinct components using the union-find data structure. By checking all eight neighbouring pixels for connections, it ensures that all pixels belonging to the same component, even diagonally, are identified as part of the same set. The second pass then correctly assigns unique labels to these identified components, ensuring that all pixels in a component receive the same label, while distinct components receive distinct labels.

Complexity:

- Time Complexity:
    - Reading the image: $O(n)$, where n is the total number of pixels.
    - First pass (union operations): $O(n \cdot \alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows very slowly and is practically constant for all reasonable values of n.
    - Second pass (find operations): Similarly, $O(n \cdot \alpha(n))$.
    - Printing and additional operations are also $O(n)$.

Overall, the time complexity is $O(n . \alpha(n))$, which is almost linear for all practical purposes.

- Space Complexity:
    - The space complexity is $O(n)$ for storing the disjoint-set data structure and the labelled image matrix.

Therefore, the algorithm efficiently identifies and labels connected components in a binary image with near-linear time complexity and linear space complexity, making it suitable for processing large images.