Assignment 3
Part 1

Question 1

Assume the utilization of *linear probing* for hash-tables. To enhance the complexity of the operations performed on the table, a special *AVAILABLE* object is used. Assuming that all keys are positive integers, the following two techniques were suggested to enhance complexity:

i) In case an entry is removed, instead of marking its location as AVAILABLE, indicate the key as the negative value of the removed key (i.e., if the removed key was 16, indicate the key as -16). Searching for an entry with the removed key would then terminate once a negative value of the key is found (instead of continuing to search if AVAILABLE is used).

ii) Instead of using AVAILABLE, find a key in the table that should have been placed in the location of the removed entry, then place that key (the entire entry of course) in that location (instead of setting the location as AVAILABLE). The motive is to find the key faster since it now in its hashed location. This would also avoid the dependence on the AVAILABLE object.

Will either of these proposals have an advantage of the achieved complexity? You should analyse both time-complexity and space-complexity. Additionally, will any of these approaches result in misbehaviours (in terms of functionalities)? If so, explain clearly through illustrative examples.

Proposal 1:

Proposal 1 says that, if the entry is removed from the hash table, then instead of marking its location as *AVAILABLE*, the key is indicated as the negative value of the removed key. This approach is helpful as there will not be a need for creating an extra variable. So, it reduces the need of extra space.

But this approach has one major drawback. Let's see one example.

10	23	-16	15	28	16	-15	20
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Let's assume that we have given the hash table as shown above. Now, we want to search the value 16 inside this hash table.

It might be possible that, when we inserted the value 16, we checked at location 2, as the assumed hash value of 16 is 2, but the space was already occupied. So, following the concept of linear probing, it was inserted at index 5.

As mentioned earlier, the assumed hash value of 16 is 2. So, when we start searching for 16, we will directly go to the index 2. But this location has -16 present already. According to the proposal, we should terminate searching as we have found the negative value of the key. But required value, 16, is already present at index 5. Even though it is present, out proposal returns false and terminates searching without going further from index 2.

So, this proposal is not suitable. We should look for more better approach.

Proposal 2:

Proposal 2 says that, instead of using AVAILABLE, we should find a key in the table that should have been placed in the location of the removed entry. The motive is to find the key faster since it now in its hashed location. This would also avoid the dependence on the AVAILABLE object. This approach is helpful as there will not be a need for creating an extra variable. So, it reduces the need of extra space.

Suppose we have given the hash table having the size of 1000. Out of these 1000 positions, first 500 elements are placed at its correct position. Remaining 500 elements are also same as first 500 elements, but these elements are found after the 500th element.

1	2	3	4	5	6	7	8	……	1	2	3	4	5	6	7	8	……
[0]	[1] ……..								[500]	[501]……..							[999]

If we remove 2 from index position 1, then as mentioned in the proposal, the algorithm searches for the value from the table that can fit in that location. If the hash table is shown above then, algorithm must traverse at least half of the hash table. So, it will take **O(n)** time, whenever the value is removed from the hash table. Which is not feasible, if there are large number of remove operations on the hash table.

Question 2

Show the steps that a radix sort takes when sorting the following array of 3-tuple Integer keys (notice that each digit in the following values represent a key):

783 992 472 182 264 543 356 295 692 491 947

Radix sort arranges numbers using the least significant digit to most significant digit. So, we need to know the range of iterations. For that, we need to find the number of digits in the maximum number.

Step - 1: Find maximum number and count digits.

After one iteration we will find maximum number.

MAX = 992, which has 3-digits. So, N = 3.

Step - 2: Arrange the numbers according to the least significant digit.

Input Array: 783 992 472 182 264 543 356 295 692 491 947

New Array: 491 992 472 182 692 783 543 264 295 356 947

Step – 3: Arrange the numbers according to the next significant digit.

Input Array: 491 992 472 182 692 783 543 264 295 356 947

New Array: 543 947 356 264 472 182 783 491 992 692 295

Step – 4: Arrange the numbers according to the next significant digit.

Input Array: 543 947 356 264 472 182 783 491 992 692 295

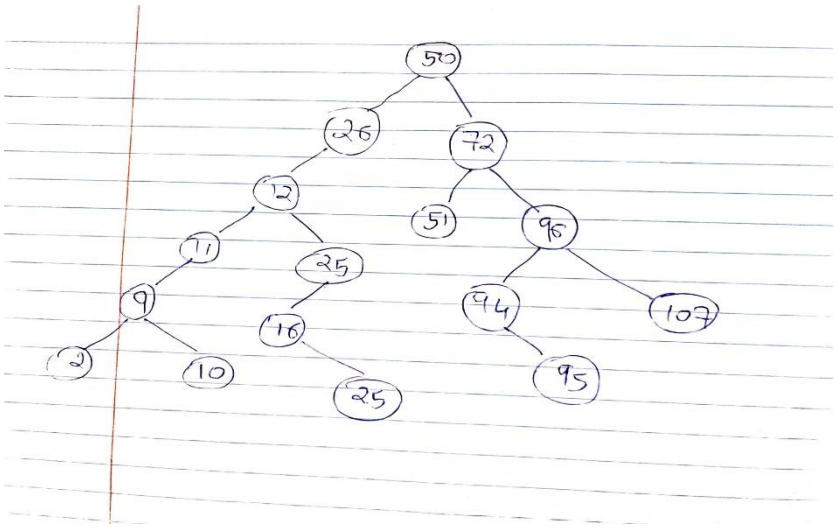
New Array: 182 264 295 356 472 491 543 692 783 947 992

Here, we have found the sorted sequence. So, we will stop here.

Question 3

Draw the binary search tree whose elements are inserted in the following order:

50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95



Question 4

Describe an efficient algorithm for computing the height of a given AVL tree. Your algorithm should run in time *O(log n)* on an AVL tree of size *n*. In the pseudocode, use the following terminology: T.left, T.right, and T.parent indicate the left child, right child, and parent of a node T and T.balance indicates its balance factor (-1, 0, or 1).

For example if T is the root we have T.parent=nil and if T is a leaf we have T.left and T.right equal to nil. The input is the root of the AVL tree. Justify correctness of the algorithm and provide a brief justification of the runtime.

Question 5

Given the following elements:

29 38 74 78 24 75 42 33 21 62 18 77 30 16

Trace the steps when sorting these values into ascending order using:

a) Merge Sort,

b) Quick Sort (using (middle + 1) element as pivot point),

c) Bucket Sort – We know that the numbers are less than 99 and there are 10 buckets.

a) Merge Sort:

Given array: 29 38 74 78 24 75 42 33 21 62 18 77 30 16

There is a total of 8 elements in the array. Thus, mid = 7. So, we will first divide this array into two arrays of size 4 as shown:

29 38 74 78 24 75 42 33 21 62 18 77 30 16

Next, we recursively divide these arrays into further halves. The half of 7 is 4. So, now we have 4 arrays.

29 38 74 78 24 75 42 33 21 62 18 77 30 16

Next, we recursively divide these arrays into further halves. The half of 4 and 3 is 2. So, now we have 8 arrays.

29 38 74 78 24 75 42 33 21 62 18 77 30 16

Again, we divide these arrays into further halves. The half of 2 is 1. So now we have 14 arrays.

29 38 74 78 24 75 42 33 21 62 18 77 30 16

After this, we have the combining step. We will compare each element with its consecutive elements and arrange them in a sorted manner. So, we will have 7 sorted arrays.

29 38 74 78 24 75 33 42 21 62 18 77 16 30

Again, we will compare each element with its consecutive elements and arrange them in a sorted manner. So, we will have 4 sorted arrays.

29 38 74 78 24 33 42 75 18 21 62 77 16 30

Again, we will compare each element with its consecutive elements and arrange them in a sorted manner. So, we will have 2 sorted arrays.

24 29 33 38 42 74 75 78 16 18 21 30 62 77

Again, we will compare each element with its consecutive elements and arrange them in a sorted manner. Now, we will get our sorted array.

16 18 21 24 29 30 33 38 42 62 74 75 77 78

B) Quick sort

Given array: 29 38 74 78 24 75 42 33 21 62 18 77 30 16

- **Step - 1**

Here, it is given that pivot element is (mid + 1). So, pivot is 33.

Pivot = 33

29 38 74 78 24 75 42 **33** 21 62 18 77 30 16

- **Step - 2**

Start pointers to left and right.

29 38 74 78 24 75 42 **33** 21 62 18 77 30 16

- **Step - 3**

Since 29 < 33, move left pointer.

29 38 74 78 24 75 42 **33** 21 62 18 77 30 16

- **Step - 4**

Since 38 > 33 and 16 < 33, swap values at pointer. And move pointers.

29 16 74 78 24 75 42 **33** 21 62 18 77 30 38

- **Step – 5**

Since 74 > 33 and 30 < 33, swap values at pointer. And move pointers.

29 16 30 78 24 75 42 **33** 21 62 18 77 74 38

- **Step – 6**

Since 77 > 33, move right pointer.

29 16 30 78 24 75 42 **33** 21 62 18 77 74 38

- **Step – 7**

Since 78 > 33 and 18 < 33, swap values at pointer. And move pointers.

29 16 30 18 24 75 42 **33** 21 62 78 77 74 38

- **Step – 8**

Since 24 < 33 and 62 > 33, move left and right pointers.

29 16 30 18 24 75 42 **33** 21 62 78 77 74 38

- **Step – 9**

Since 75 > 33 and 21 < 33, swap values at pointer. And move pointers.

29 16 30 18 24 21 42 **33** 75 62 78 77 74 38

- **Step – 10**

Since 42 > 33, swap values at pointer. And move pointers.

29 16 30 18 24 21 **33** 42 75 62 78 77 74 38

- **Step – 11**

We have two sub arrays.

29 16 30 18 24 21 42 75 62 78 77 74 38

- **Step – 12**

We will follow same process (step 1 to step 11) until all the elements are sorted.

At the end we will get following sorted array,

16 18 21 24 29 30 33 38 42 62 74 75 77 78

C) Bucket sort

Given array: 29 38 74 78 24 75 42 33 21 62 18 77 30 16

- **Step – 1**

Here, we have all the elements having value less than 99. And the number of buckets is 10. So, each bucket can store 10 values.

Bucket	Range
1	0 - 9
2	10 - 19
3	20 - 29
4	30 - 39
5	40 - 49
6	50 - 59
7	60 - 69
8	70 - 79
9	80 - 89
10	90 – 99

- **Step – 2**

Now, scatter the array elements to these buckets.

1 (0 – 9)	2 (10 – 19)	3 (20 – 29)	4 (30 – 39)	5 (40 – 49)	6 (50 – 59)	7 (60 – 69)	8 (70 – 79)	9 (80 – 89)	10 (90 – 99)
	18, 16	29, 24, 21	38, 33, 30	42		62	74, 78, 75, 77		

- **Step – 3**

Sort each bucket individually.

1 (0 – 9)	2 (10 – 19)	3 (20 – 29)	4 (30 – 39)	5 (40 – 49)	6 (50 – 59)	7 (60 – 69)	8 (70 – 79)	9 (80 – 89)	10 (90 – 99)
	16, 18	21, 24,29,	30, 33, 38	42		62	74, 75, 77, 78		

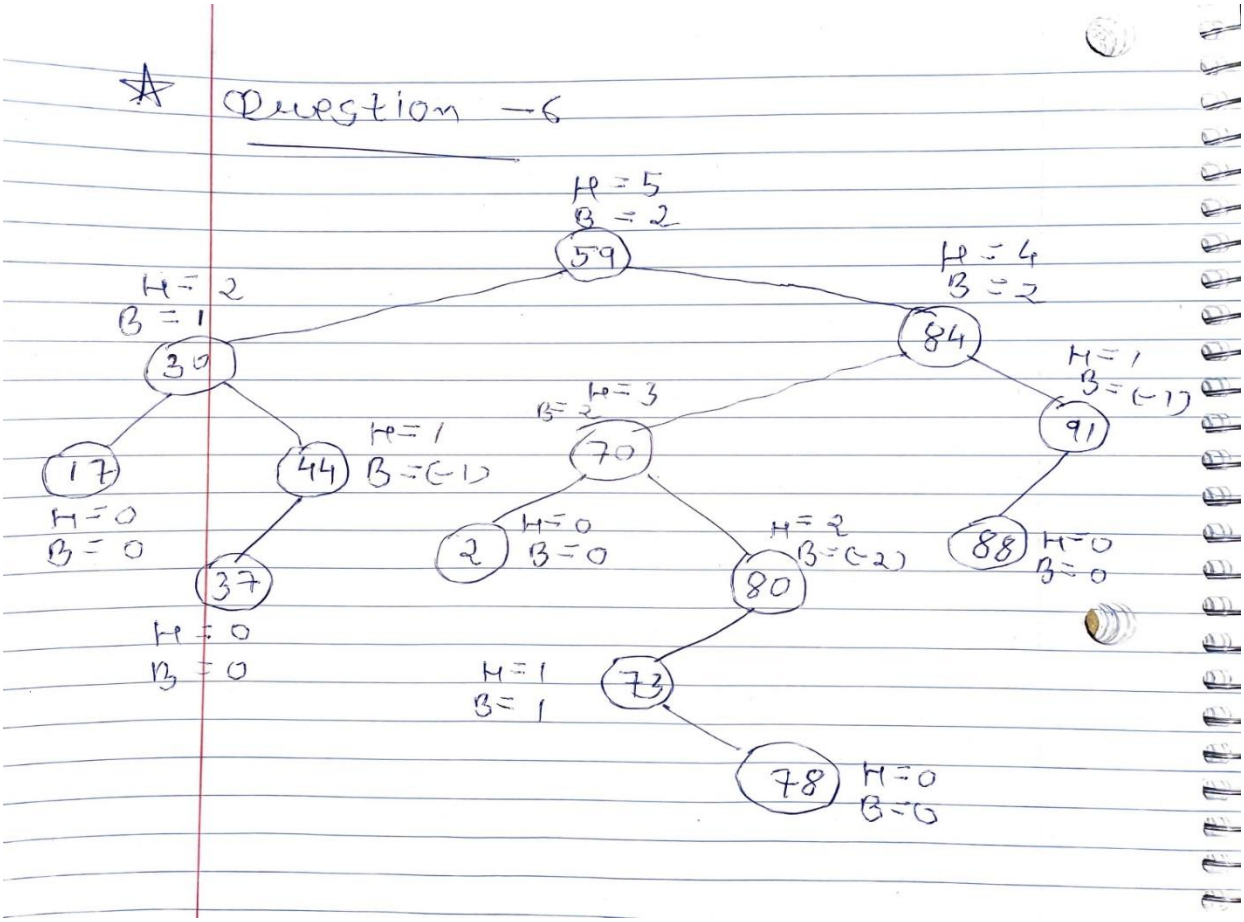
- **Step – 4**

Gather the sorted elements from buckets to original array

Sorted array: 16 18 21 24 29 30 33 38 42 62 74 75 77 78

Question 6

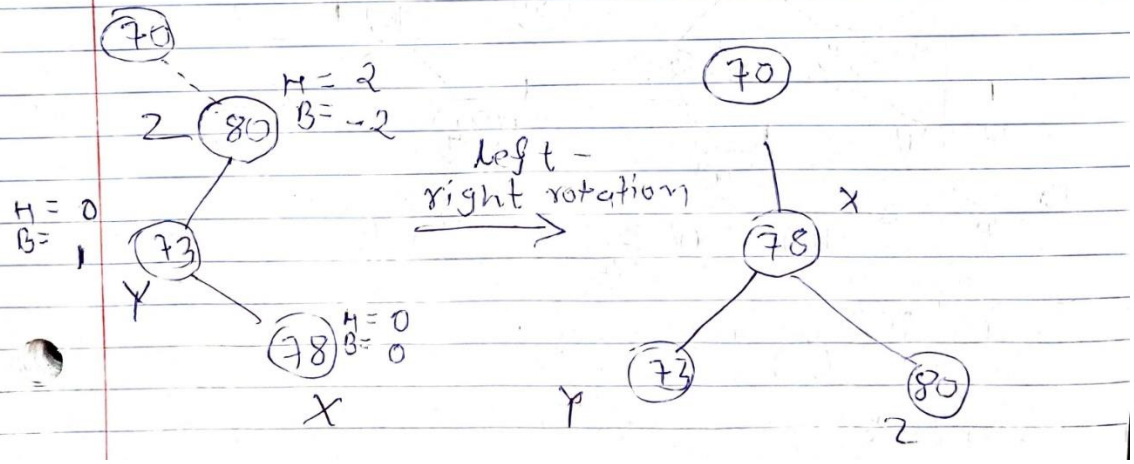
Given the following tree, which is assumed to be an AVL tree:



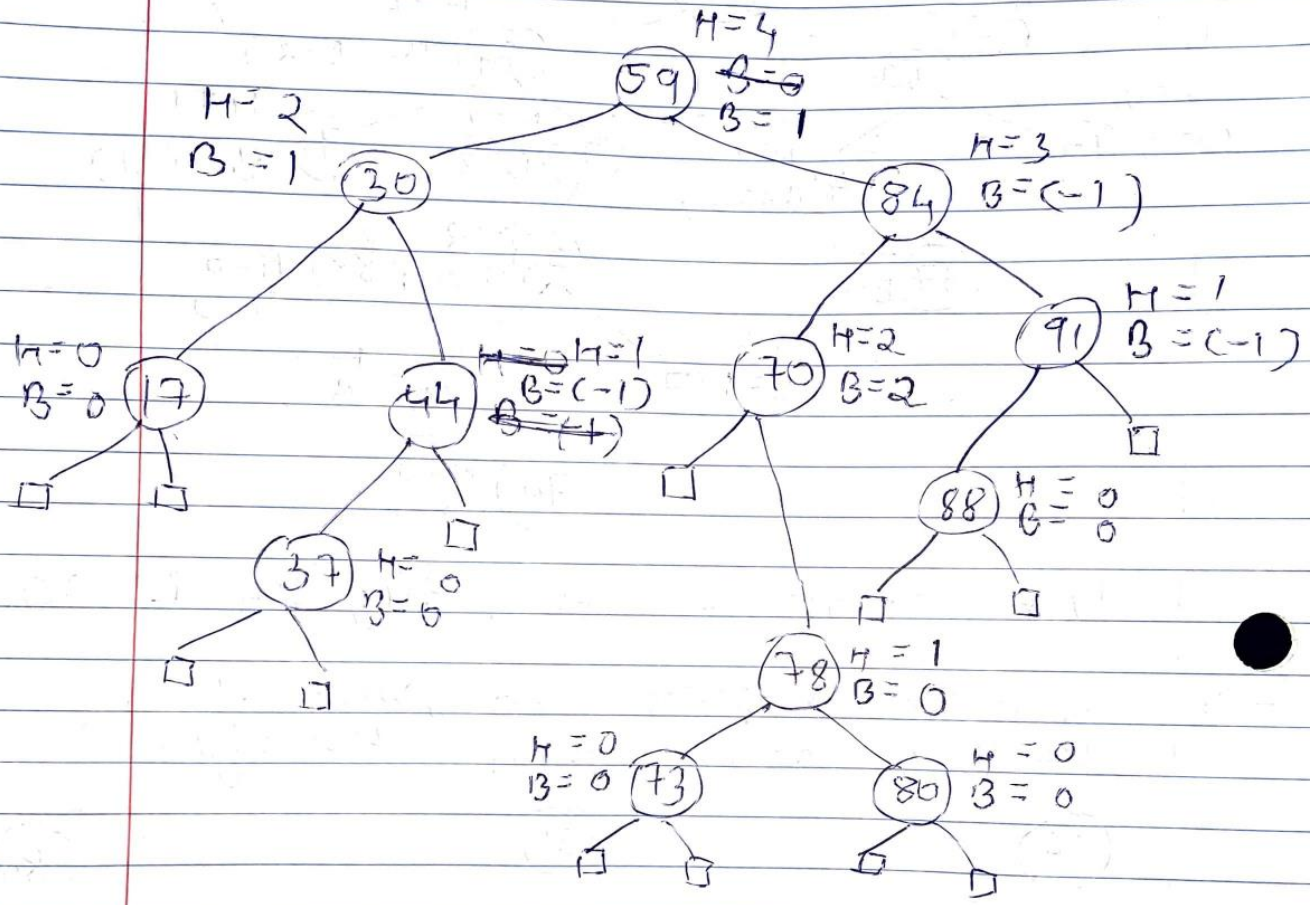
Hence, we have errors with

59, 84, 70 & 80 nodes.

→ To correct, we start with 80 the farthest node from root & work way up towards root.

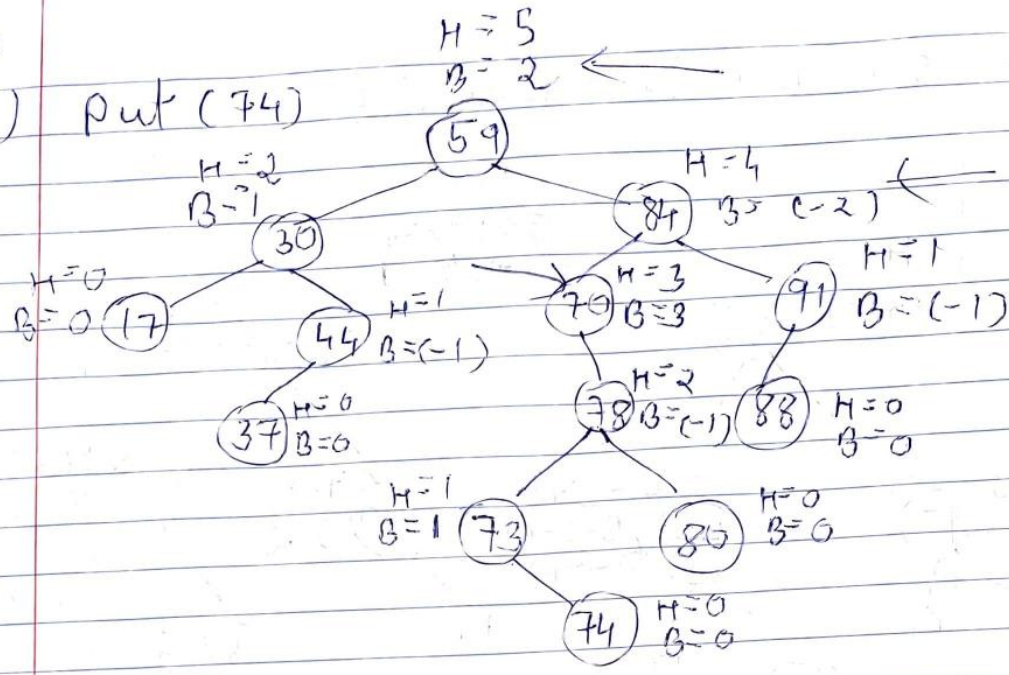


→ So, the tree would be,



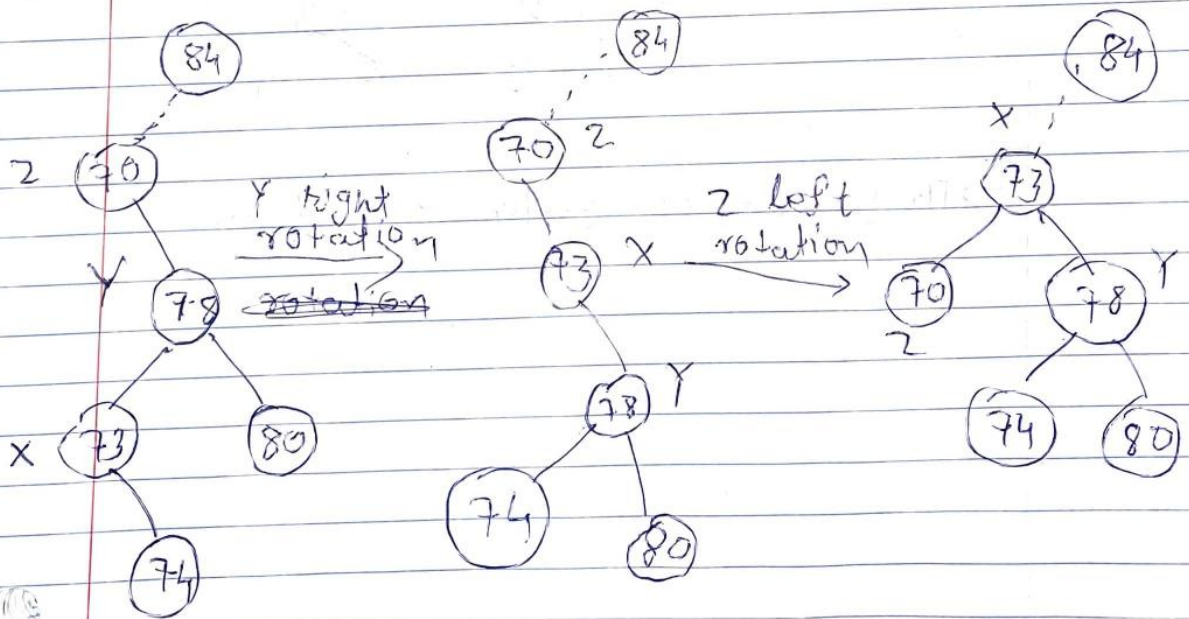
→ This is a balanced tree.

(ii) put (74)

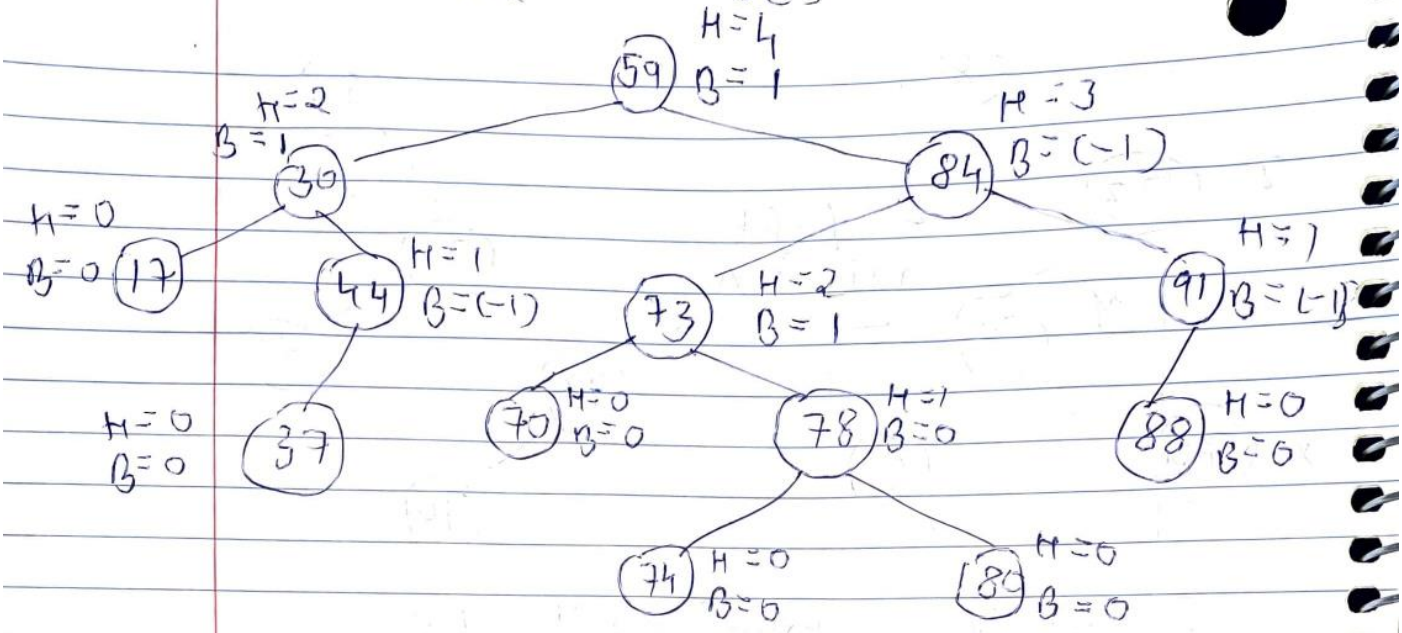


The error is in 59, 84 & 70.

→ we will take 70, since it is farthest root from the right.

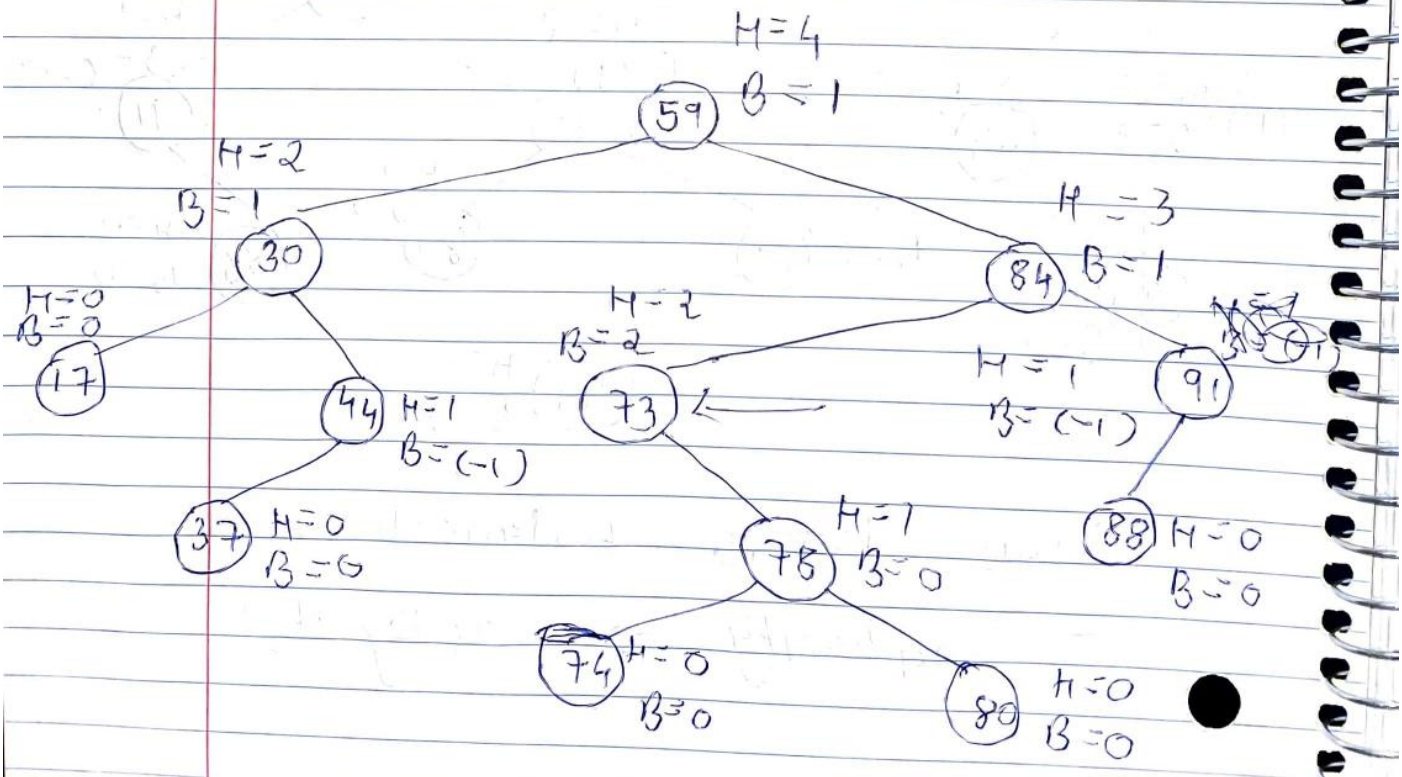


→ So the tree will be,

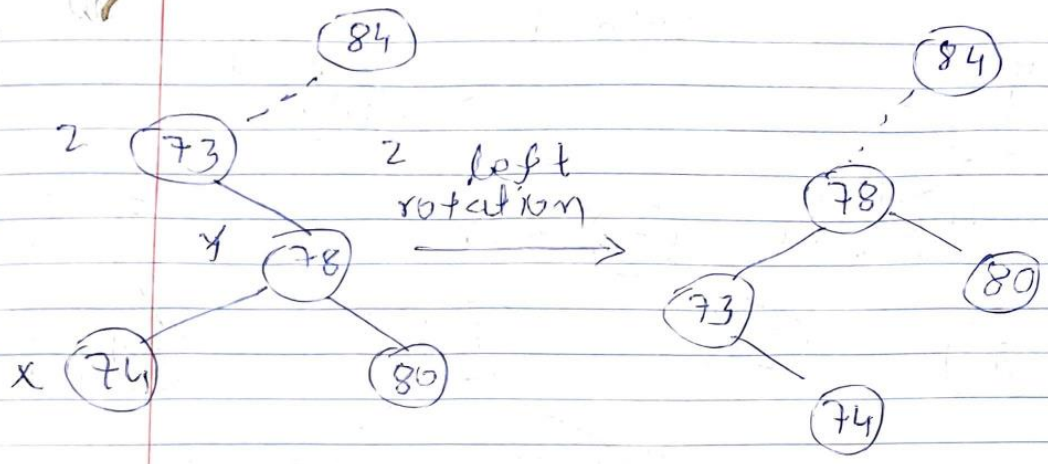


→ Now, the tree is balanced.
→ Complexity → O(log n)

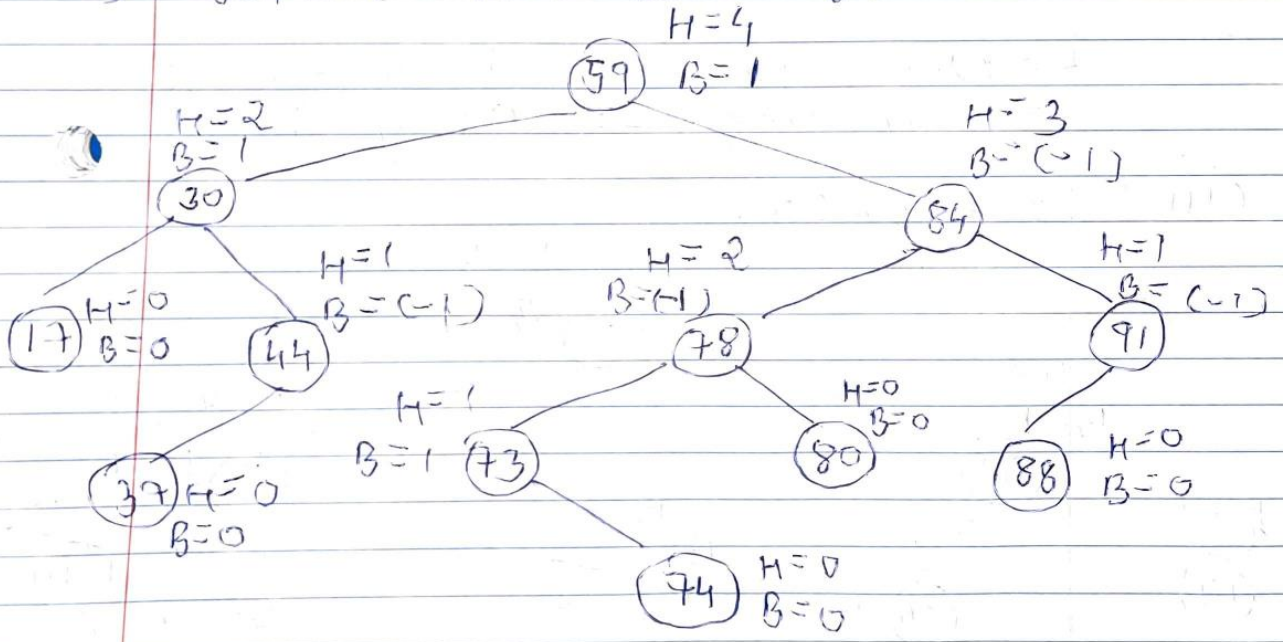
(iii) remove(70)



→ Problem is with the node, 73



→ So, the tree will be

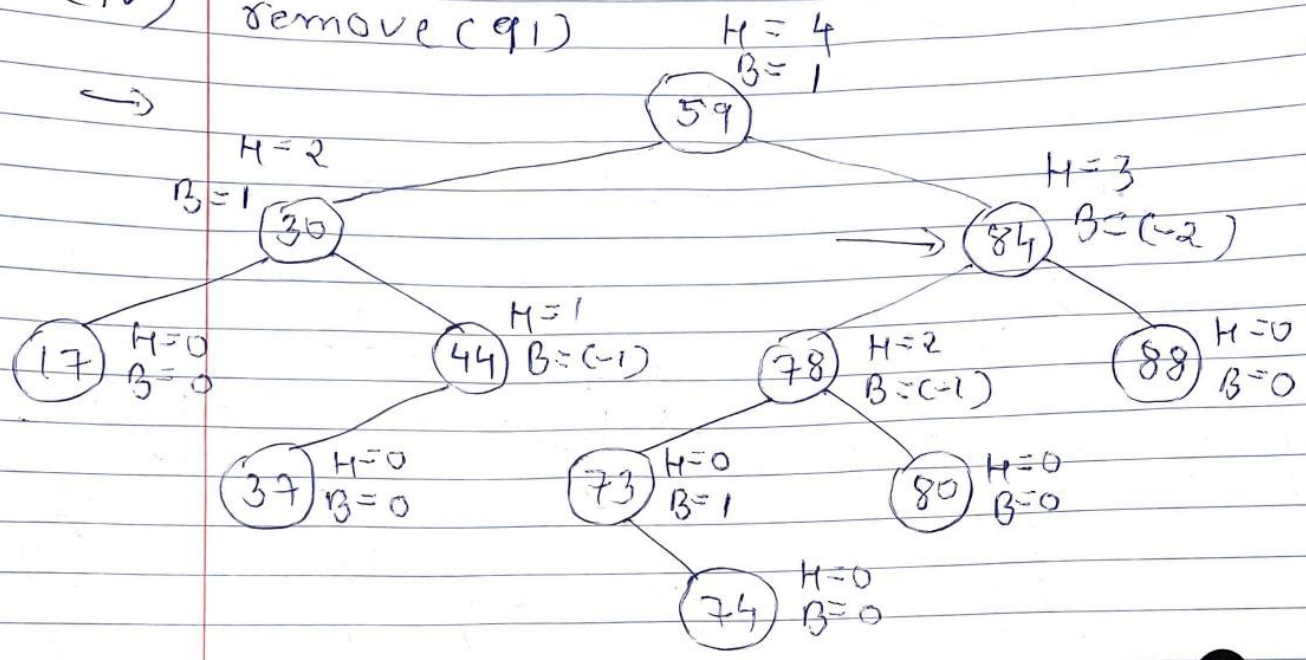


→ The tree is balanced.

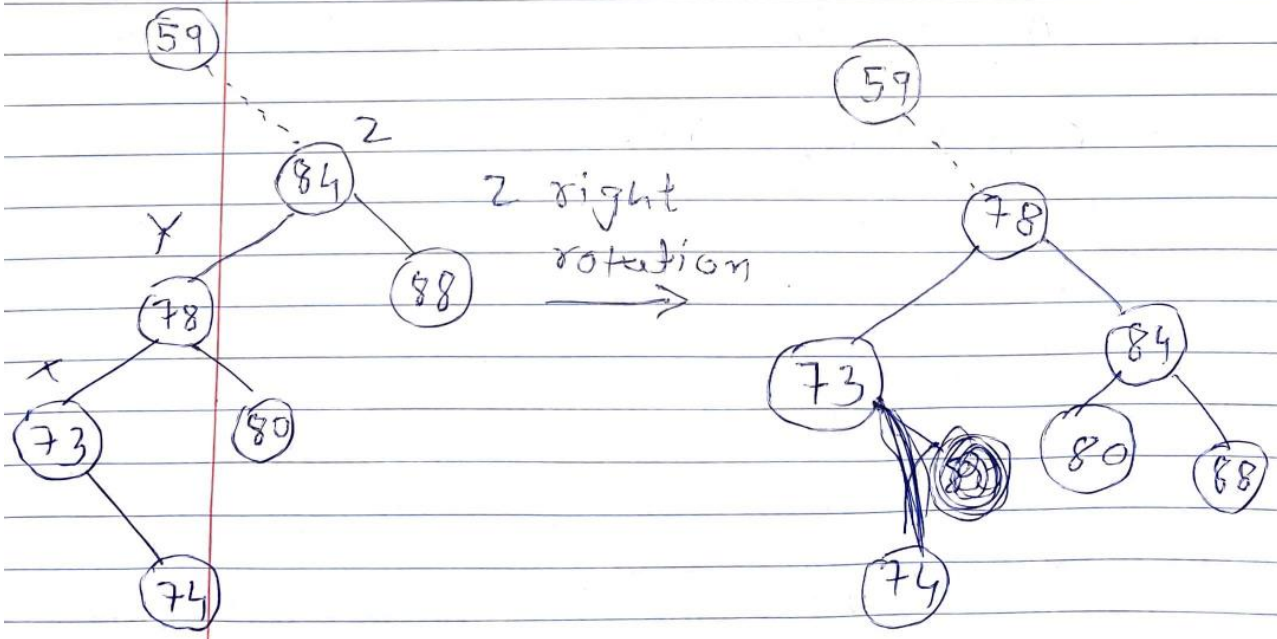
→ Completity → $O(\log n)$

(iv) remove(91)

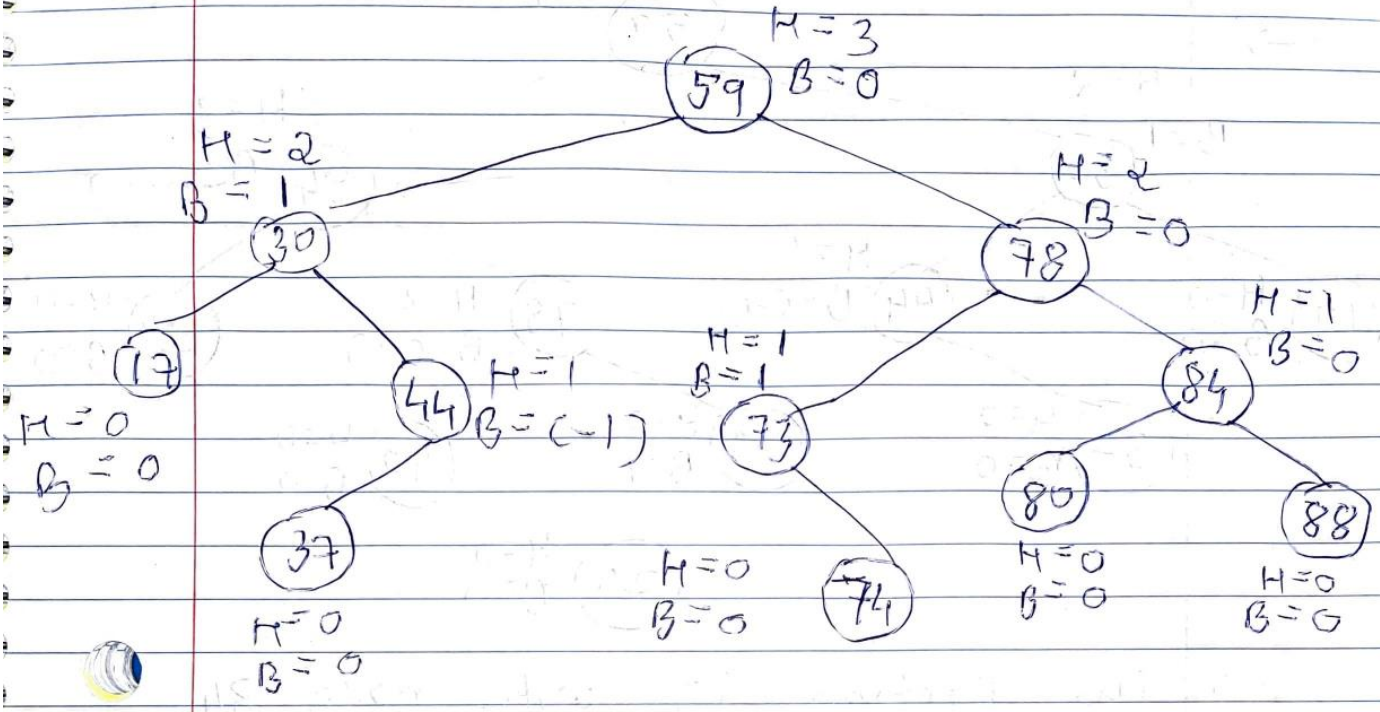
→



→ The problem is with node 84,



→ So, the tree will be,



→ Now, the tree is balanced,

→ Complexity → O (log n)