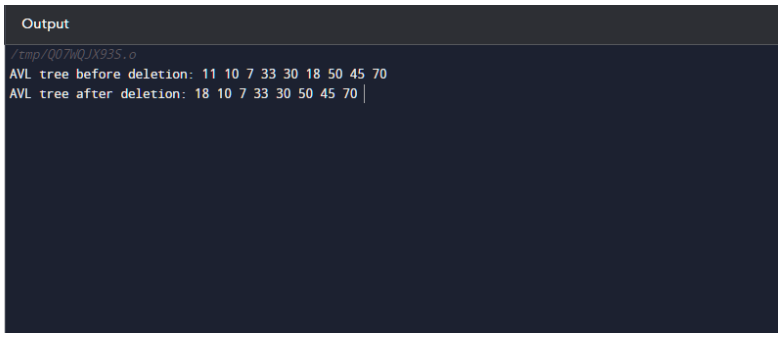
<u>**Assignment 3 - Output**</u>

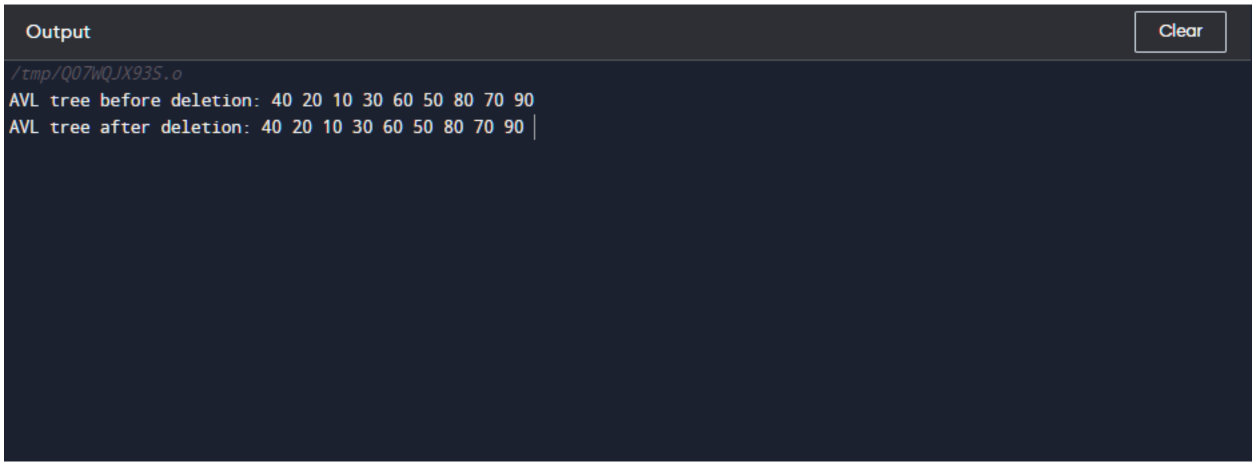
1. <u>Deletion in AVL Tree</u>
   Output Text screens

Deleting 11



```
Output
/tmp/Q07WQJX93S.o
AVL tree before deletion: 11 10 7 33 30 18 50 45 70
AVL tree after deletion: 18 10 7 33 30 50 45 70
```

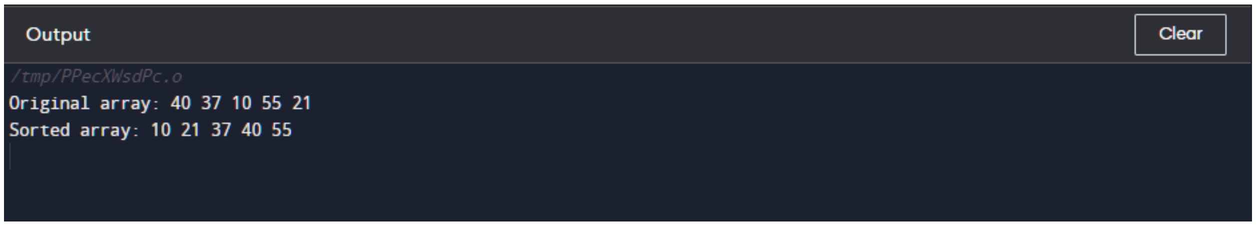
Deleting 80



```
Output                                                   Clear
/tmp/Q07WQJX93S.o
AVL tree before deletion: 40 20 10 30 60 50 80 70 90
AVL tree after deletion: 40 20 10 30 60 50 80 70 90
```

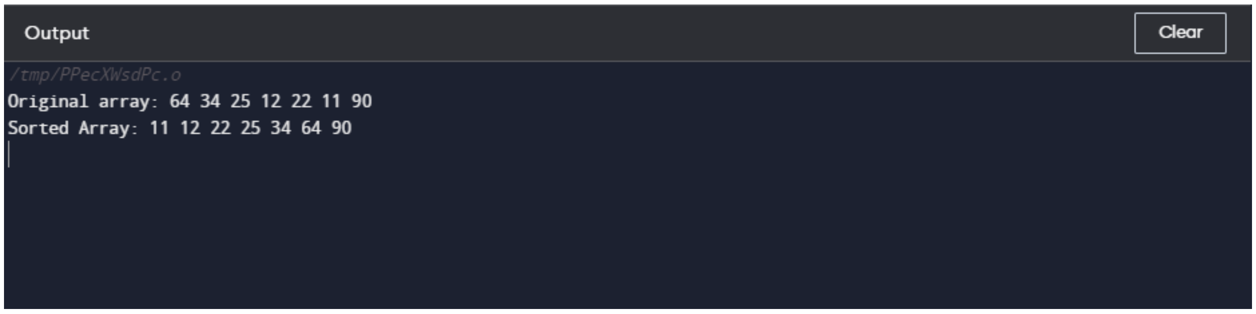
2. Insertion sort and Bubble Sort comparison

Insertion Sort - Output text screen



```
Output                                                   Clear
/tmp/PPecXWsdPc.o
Original array: 40 37 10 55 21
Sorted array: 10 21 37 40 55
```

Bubble Sort - Output text screen



```
/tmp/PPecXWsdPc.o
Original array: 64 34 25 12 22 11 90
Sorted Array: 11 12 22 25 34 64 90
```

<u>Q2) Bubble Sort:</u>
Bubble sort starts at the beginning of the array, and compares elements two at a time and swaps them if the right one is larger than the left one. Take the following example:

7	6	4	3

Start by comparing 7 and 6 and swapping them if arr[1] > arr[0]

6	7	4	3

Compare the next two adjacent elements and swap if arr[left] > arr[right]

6	4	7	3

6	4	3	7

4	6	3	7

4	3	6	7

4	3	6	7

3	4	6	7

Once arr[0] and arr[1] are switched, the loop goes through the entire array once and if there are no swaps, the loop breaks.

Bubble Sort time complexity:

- The best case time complexity for bubble sort is O(n), where n = to the number of elements in the array. The best case may occur where the iteration only occurs n number of times.

- The worst case complexity is $O(n^2)$, when the elements are in reverse order and the number of comparisons add up to

$$= (n-1) + (n-2) + (n-3) + (n-4) + \dots + 1$$
$$\approx (n-1)(n-2)/2 \Rightarrow O(n^2)$$

<u>Insertion Sort:</u>
Take the same array: {7,6,4,3}

7	6	4	3

Start by comparing arr[0] and arr[1]

6	7	4	3

6 is now in a separate subarray, and 7 and 4 are compared next.

6	4	7	3

4 is lesser than 7 so it is swapped and then compared with 6

4	6	7	3

4, 6, 7 now stored in subarray and 7 and 3 are compared

4	6	3	7

4	3	6	7

3 is now compared with elements on its left till it is in the correct position → arr[left] < arr[right]

3	4	6	7

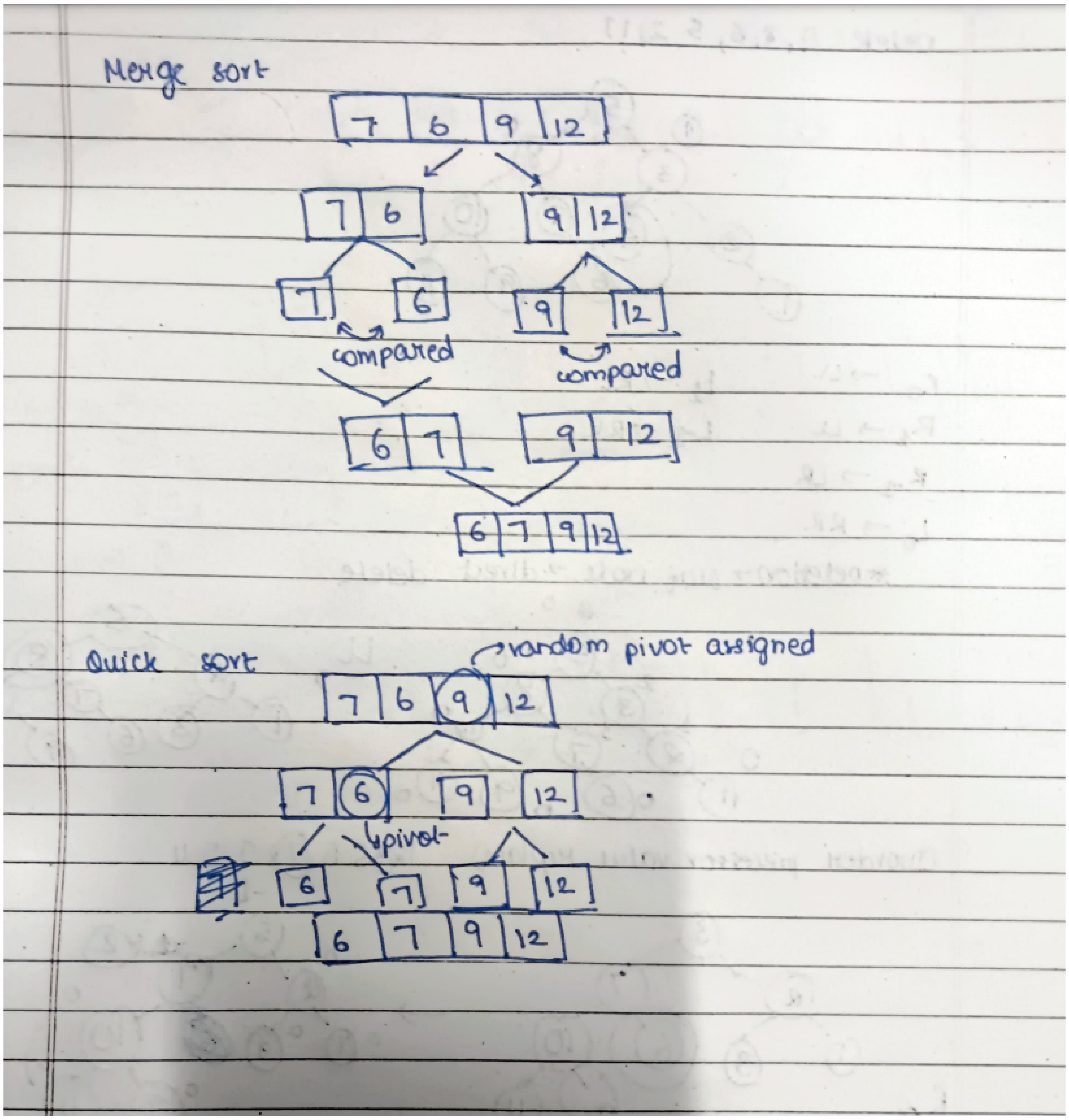
Insertion sort complexity:
- The best time complexity is also $O(n)$, where the array is ordered and the loop only goes through the entire array once.
- The worst case complexity $O(n^2)$, which can happen when the array is completely in reverse order. The loop starts at $j = 1$. When there is a comparison and swap in this position, the time taken is $(1 + 1)$. At $j = 2$, there are 2 comparisons done with its indexes on the right, and there are 2 swaps. Similarly this goes on for n elements.

$j = 1 \Rightarrow (1+1) = 2(1)$
$j = 2 \Rightarrow (2+2) = 2(2)$
$j = n \Rightarrow (n+n) = 2(n)$

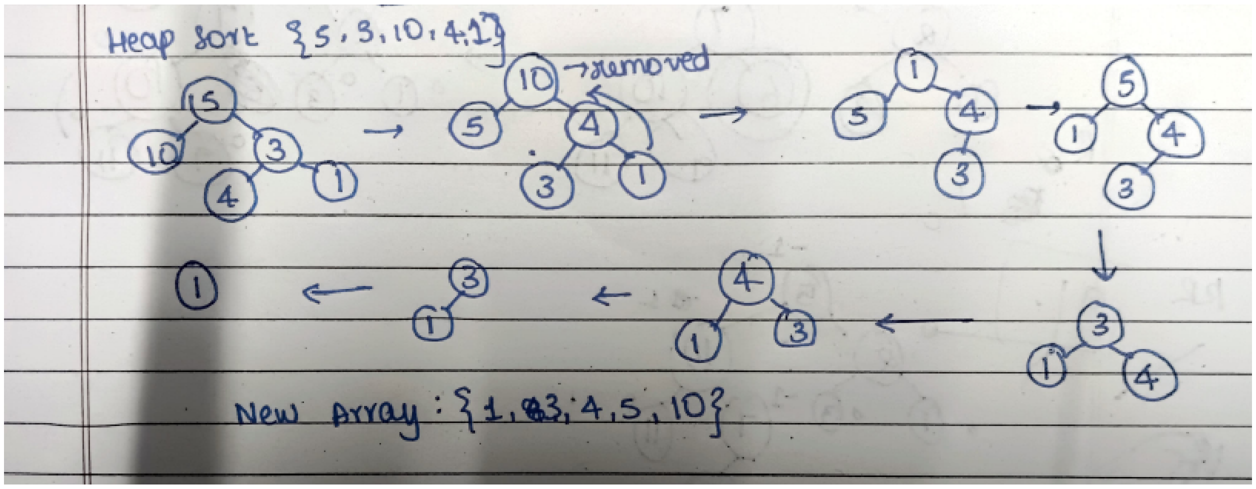
$$\Rightarrow 2(1) + 2(2) + \dots + 2(n)$$
$$\Rightarrow 2(1 + 2 + 3 + \dots + n)$$ - sum of all natural numbers

$$\Rightarrow 2n(n + 1)/2 \Rightarrow O(n^2)$$

Q3) <u>Merge Sort and QuickSort vs Heap Sort</u>



<u>Heap Sort:</u>



<u>Merge Sort Time Complexity:</u> For merge sort, consider the time for sorting and merging elements in the array, where n is the number of elements in the array. It equates to log(n) because this is the number of possible divisions that the array may make before sorting.

$$T(n) = 2*T(n/2) + O(n) \approx O(n\log(n))$$

<u>Quick Sort Time Complexity:</u> The best case time complexity is **O(log n)** and the worst case is **$O(n^2)$**. The best case time complexity occurs when the pivot of the array is the median of the array. However, applying median finding algorithm to quick sort becomes inefficient. The worst case complexity occurs if the pivot at each iteration is the least/greatest value of the array.

<u>Heap Sort Time Complexity:</u> The best and worst case is O(log n) as the time it takes to sort is dependent on the height of the binary search tree which is a height of log(n).