DAA ASSIGNMENT-5
GROUP-3
Find k'th largest element using Heap Sort and track content of each node

IIT2016039, IIT2016007, IIT2016019, IIT2016052

***Abstract*—Here, we have to find the kth largest element from a set of numbers using heap sort. An $O(n*log(n))$, $\Omega(n)$ algorithm has been designed to solve this problem.**

***Index Terms*—Sorting, storing positions, time complexity, heaps**

I. Introduction and Literature Survey

Sorting is any process of arranging items in a sequence ordered by some criterion. Here we are sorting the integers based on where they appear on the real number axis.

Here we have to insert the n unsorted elements into a heap.

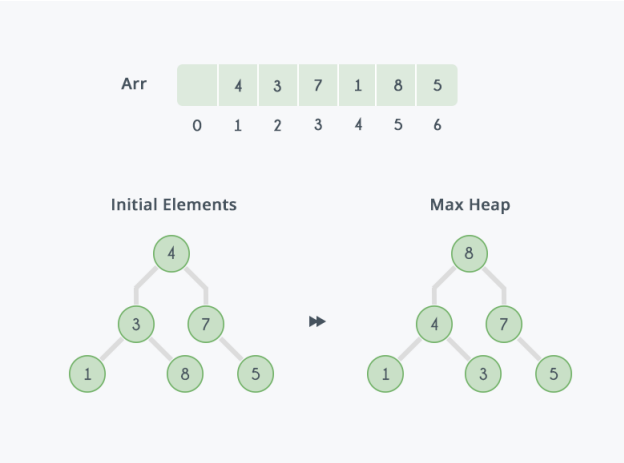

Figure 1. heap

Then find the kth largest element using heap sort. We are going to extract the largest element from the heap (k-2) times and then find the larger of the two children of the root to store the kth largest element. This works because the 3 largest elements of a heap are among the root and it's 2 children, and since the root element is larger than any of it's children, the 2nd largest element is the larger of the two children of the root. Hence, after extracting (k-2) largest elements from the heap, we find the 2nd largest element in the remaining heap which is going to be the kth largest element. We are also tracking each position by storing which elements came to that particular position while sorting process is going on, so every time we swap an element two nodes are created(one for each position.)

II. Algorithm Design

Finding the kth largest element of an array using heap

The idea of our algorithm is to build maxheap from given input array and then extracting the element from top k-2 times as the top of maxheap is always maximum element.After this we will find the maximum of the two children of the root node. This works because the root node contains the (k-1)th largest element and since it's a max heap, the kth largest element will be the largest of the two children of the root node.Our algorithm also gives on which position which element exists after each max heapify operation is performed in order to trace the nodes being swapped. We divided algorithm into 3 main parts:

1. *MAXHEAPIFY :-* Its inputs are an array A and an index i into the array. When it is called, MAXHEAPIFY assumes that the binary trees rooted at LEFT[i] and RIGHT[i] are maxheaps,but that A[i] might be smaller than its children, thus violating the max-heap property. MAX-HEAPIFY lets the value at A[i] float down in the max-heap so that the subtree rooted at index i obeys the max-heap property.

2. *BUILDHEAP :-* The procedure BUILD-MAX- HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

3. *EXTRACTMAX :-* In this algorithm we will delete top most element in heap and then calling BUILDHEAP to create maxheap again on remaining elements.

Tracing the content in each node

The ith index of posarr array of shows the new element stored in that position after any swaps are made due to the maxheapify or buildheap functions.

Algorithm 1 $Kth-largest-element-using-heap-sort$

```
Struct node {
val
struct node *left;
struct node *right;        ▷ the struct node datatype is used to store a node along with it's left and right child in the complete binary tree.
};
procedure SWAP(posarr, arr, index, x, y)
    z ← arr[x]
    arr[x] ← arr[y]
    arr[y] ← z
INSERTNODE(posarr, x, arr[x])
INSERTNODE(posarr, y, arr[y])

procedure MAXHEAPIFY(arr, index, n)
    largest ← index
    left ← (2 · index + 1)
    right ← 2 · index + 2
    if left < n and arr[largest] < arr[left] then
        largest ← left
    if right < n and arr[largest] < arr[right] then
        largest ← right
    if largest ≠ index then
SWAP(posarr, arr, index, largest)‘
MAXHEAPIFY(posarr, arr, largest, n)
```

Algorithm 2 *INSERTNODE*

```
Input: arr, i, value
*temp.pos ← value
*temp.link ← NULL
*arr[i].tail.link ← temp
arr[i].tail ← temp
```

Algorithm 3 *main*

```
Input: arr, i, value
print(”Initially the contents of the Heap”)
for i = 0 to i ¡ n do
print(Node:
arr[i] ← i)
end for
print(”Building the Heap”)
BUILD-HEAP(Arr,size)
print(”Heap is built”);
for i = 0 to i ¡ n do
print(Node:
arr[i] ← i)
end for
for i = 0 to i ¡ k-2 do
EXTRACTMAX(arr,n-i)
end for
if k = 1 then
print arr[0]
return 0
if k = n or arr[1]¿arr[2] then
print arr[1]
return 0
print arr[2]
return 0
```

III. Time Complexity Analysis

The time complexity for the above mentioned algorithm depends on two factors one is the time require to create heap for the given array and other is to find the Kth largest element from the heap.Time complexity for building heap is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right).$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$$

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n).$$

Figure 2. Build heap time complexity

Here h is the height of the tree. *Worst Case* :The worst case is when we need to find the smallest element in the heap i.e nth largest in a heap of size n. In this case we have to delete n-2 elements from the heap, So maximum number of swaps occur.In swapping we insert two nodes one for each position which contains the element and iteration number. In worst case k is n so we have to remove n-2 elements from the heap. After removing each element we have to call heapify again which takes log(n) time. so time complexity to find the n largest element is (n-2)log(n).

So time complexity in worst case is proportional to n+(n-2)log(n) i.e O(n*log(n)).

Best Case :In the best case the given array is already is a max heap and the element we need to find is the 1st largest element. so we for heapify it takes n/2 steps and the fist element if found in O(1) time.

So time complexity in worst case is proportional to n/2 i.e O(n).

Average Case : The expression for time complexity in the average case is bounded below by the expression for time complexity obtained in the best case and bounded above by the expression for time complexity obtained in the worst case.

g(x) < f(x) < h(x) where g(x) is the tightest lower bound of the function and h(x) is the least upper bound of the function. here g(x) = $\Omega(n)$ and h(x) = O(n*log(n))

IV. Space Complexity Analysis

For Building heap no extra space is required as all the elements gets swapped in the same array. Here as we tracking each element in a particular position we require some more space.

As we cannot know the exact number of elements that will come to a particular position space will be unnecessarily wasted if we use arrays. So here we have implemented it using linked lists to reduce the space complexity.

Worst Case : The worst case is when the given array is sorted in ascending order and when we need to find the smallest in the array. so that it takes maximum swaps for the array to become a max heap.Initially n nodes are created one for each position, There will be at max n swaps so 2n nodes will be created for building heap 3*n nodes are required and in finding the nth largest element it takes n*log(n) swaps at max so in worst case extra 3*n+n*log(n) nodes are to be created.

Best Case : The best case for space complexity is when we need to given array is in descending order and we need to find the 1st largest number so that there will be no swaps only initial positions will be stored. so only n nodes are required. *Average Case* : The expression for space complexity in the average case is bounded below by the expression for space complexity obtained in the best case and bounded above by the expression for space complexity obtained in the worst case.

V. Experimental Study

We have found the time complexity of the algorithm by calculating it in each step. Now to will verify it with the experimental results.

We have plotted T VS N graph where is T is the total number of computations performed for a given N. The T and N values for calculating in rows graphs for them are plotted accordingly.

Table I
N VS T GRAPH

n	t_{best}	$t_{average}$	t_{worst}
10	55	351	448
20	110	387	1173
30	165	976	1976
40	220	545	2909
50	275	3375	3868
60	330	4290	4801
70	385	1385	5838
80	440	2221	6901
90	495	1740	8016
100	550	2113	9053

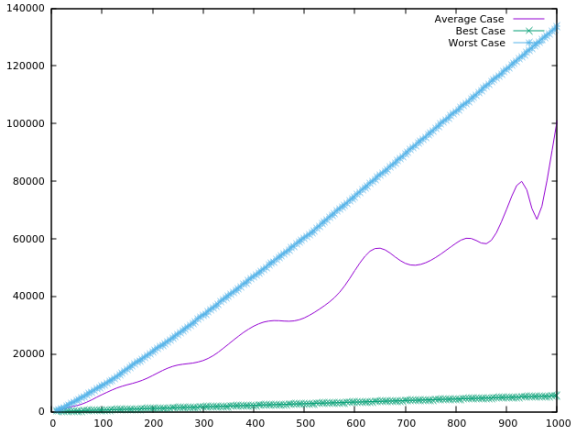

Figure 3. time(y-axis) VS N(x-axis)

Now to took each node creation uses one unit of space then the values and graph for space complexity is as follows:

Table II
N VS SPACE GRAPH

n	t_{best}	$t_{average}$	t_{worst}
10	10	26	126
20	20	284	336
30	30	178	570
40	40	204	844
50	50	506	1126
60	60	736	1400
70	70	998	1706
80	80	752	2020
90	90	458	2350
100	100	1056	2656

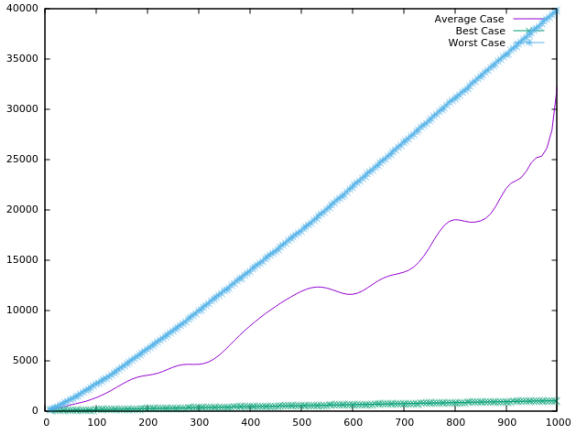

Figure 4. Nodes(y-axis) VS N(x-axis)

VI. Discussions and Comparison with other group

Thus from time complexity analysis and experimental study we can see the overall complexity of the algorithm is always proportional to n in best case and $n * log(n)$ in worst case.

The space occupied in best case is proportional to n and in worst case it is proportional to n*log(n). Thus the experimental

study proves that our time complexity and space complexity analysis is correct.

Now if we compare the algorithm of other group with our group. There are basically two differences one major and one minor. The one minor is that in our algorithm we are stopping the delete process at k-2 so that we can easily find the required element comparing the 2nd and 3rd element in the heap.
The major difference is that in tracing each node the other group is just printing the numbers that are swapped ,when the input is very large you cannot find the data of a particular node easily in their case, But in our algorithm we are maintaining a linked list for each node so that every time a new element comes to that node we insert it in corresponding linked list so that if we need particular information of a node we can get that easily.

If we had multiple queries for kth largest element then we keep track of how many elements are sorted and when we get a query in which the given k is less that sorted part then we can directly print it so then for any number of queries the worst case complexity is O(n*log(n)).

VII. Conclusion

we have tried to design an efficient algorithm that sorts a given list by finding the largest and smallest element along every iteration and swapping it up with the first and last indexed element of the remaining list. A simple approach of traversing across the array (n/2) times and decreasing size of remaining array by 2 with each iteration has been used, the details of implementation and pseudo code along with relevant calculations of time complexities and experimental details is done above to the best of our abilities. Hence, it can be concluded that the given question can be solved in $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$. Moreover, all the possible cases have been discussed for the formation of as efficient algorithm as possible.

References

[1] https://www.hackerearth.com/practice/notes/heaps-and-priority-queues/
[2] https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/
[3] https://www.geeksforgeeks.org/heap-data-structure/
[4] Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford; Introduction to algorithm, Tenth Edition, 2008