DAA Assignment 4

Nitin Raj Singh IIT2016132
Chirag Meel ISM2016006.
Lekhika Dugtal ITM2016008
Saurabh Kumar IWM2016502

1 Problem

Given a Heap, write an efficient algorithm to locate the $1^{st} largest, 2^{nd} largest, ..., k^{th}$ largest element in the Heap, for some given input value of k. Do the necessary experimentation and analysis with your algorithm.

2 Introduction and Literary Section

In this question we have to locate the k largest elements in the heap. That means that if all the elements present in the heap were to be put in an array and sorted in decreasing order, then we have to select the first k elements in the array. We also have to tell about the node number where that particular element is present in the heap.

Heap Data Structure :

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

3 Algorithm Design

In a Binary tree structure, given a node i we can easily calculate its parents, left child and right child.

$$PARENT(i) \Rightarrow return \quad i/2$$

$$LEFT(i) \Rightarrow return \quad 2i$$

$$RIGHT(i) \Rightarrow return \quad 2i+1$$

There are two kinds of Binary Heap: *Max-heap and Min-heap*.

In both kinds the values of a node satisfy a heap property, the specifics of which depend on the kind of heap.

In a max-heap, the max-heap property is that for every node i, other than the root,

$$A[PARENT(i)] >= A[i]$$

that is, the value of the parent node is always greater than its children node. Following this property we can conclude that the root node has the maximum value and the subtree rooted at a node contains values no larger than that contained at the node itself.
A min-heap is organized in the opposite way; the min-heap property is that for every node i other than the root,

$$A[PARENT(i)] <= A[i]$$

The root is the smallest element in the heap.

To locate the node number at which any particular element is present, we use an index array which will be initialized to i initially.

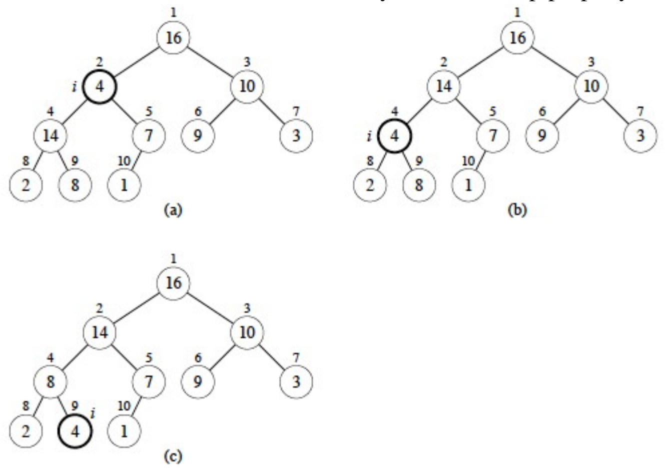
```
for $i \Leftarrow 1$ to $n$ do
    $Index[i] \Leftarrow i$
end
```

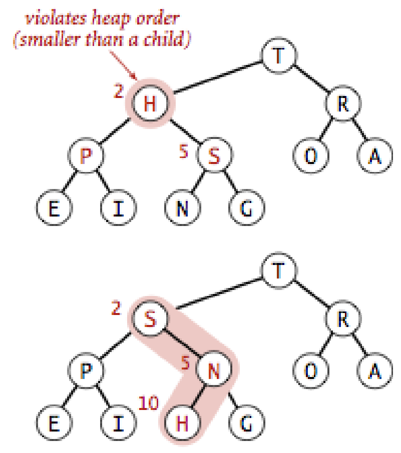
Now, to keep track of the changes that we make in the heap, we exchange the index array also along with the array elements in the MAX-HEAPIFY function and the FIND() function. In this way for any particular element i in the heap, its corresponding node number will be stored in the index[i].

3.1 MAX-HEAPIFY

One of the most important algorithm relating to heap is MAX-HEAPIFY(). We call this procedure in order to maintain Max Heap Property. In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY.

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.





```
MAX-HEAPIFY(A, i)
$l \Leftarrow LEFT(i)$
$r \Leftarrow RIGHT(i)$
if $l \leq A[heap-size]$ and $A[l] > A[i]$ then
    $largest \Leftarrow l$
end
else
    $largest = i$
end
if $r \leq A[heap-size]$ and $A[r] > A[largest]$ then
    $largest \Leftarrow r$
end
if $largest \neq i$ then
    exchange A[i] with A[largest]
    exchange index[i] with index[largest]
    MAX-HEAPIFY(A, largest)
end
```

3.2 Removing the Maximum Element from a Max-Heap

We know that the maximum element of the heap is present at the root of the heap and removing the maximum element is not simple, as we want to maintain the max-heap property even after removing that element. For that we first exchange the root element with the last element and then call the MAX-HEAPIFY procedure to maintain the max-heap property again.

```
FIND(k);
for $i \Leftarrow 1$ to $k$ do
    exchange A[1] with A[heap-size]
    exchange index[1] with index[heap-size]
    $heap-size \Leftarrow heap-size - 1$
    MAX-HEAPIFY(A, 1)
end
```

The k-largest elements would be present in the array A, from the last in decreasing order. Also the node number corresponding to every element as it was in the original heap would be present in the index array.

4 Analysis

4.1 Space Complexity

We are using an extra space to store the indexes of the elements in the heap. Thus in every case we are using the same amount of extra space.

$$O(n)$$

4.2 TIme Complexity

MAX-HEAPIFY() function

Hypothetically, the best case the complexity of MAX-HEAPIFY is $\Omega(1)$ as the i^{th} element can be the equal to both the children and thus the MAX-HEAPIFY function would not be called again. But the chances of this happening is very low, as we are replacing the node this the leaf element and thus it has to be small as seen from the max-heap property. One of the case when this would be true is when all the elements in the heap are equal.

$$T \propto (2+2+5+1+5+1+1+3)$$

$$T \propto (20)$$

$$T \quad \propto \quad \Omega(1)$$

A heap is a form of complete binary tree and the max height of a complete binary tree can not be more than logn (where n is the number of nodes in the tree). Thus the worst case complexity of the MAXHEAPIFY function will be proportional to logn. As we are exchanging the root with the leaf node there is a very high chance that the function will execute for logn as that number would be small and there is a very high chance that it will propagate down to the bottom again. Thus the average and worst case complexity would be logn.

$$T \propto (2+2+5+1+5+1+1+3) * logn$$

$$T \propto 20 * logn$$

$$T_{worst\ and\ average} \quad \propto \quad O(logn)$$

FIND() function

The find function in every case would run for k iterations(where k <= n) as we have to find k largest elements in the heap.

$$T \propto (3+1+1+1+x) * k$$

Where x = time taken by MAX-HEAPIFY function

$$T \quad \propto \quad (k)$$

In the best case the value of x would be 20,

$$T_{best} \propto 26 * k$$

In the worst case the value of x would be 20*logn, thus the complexity of the whole function becomes,

$$T \propto (6 + 20 * logn) * k$$

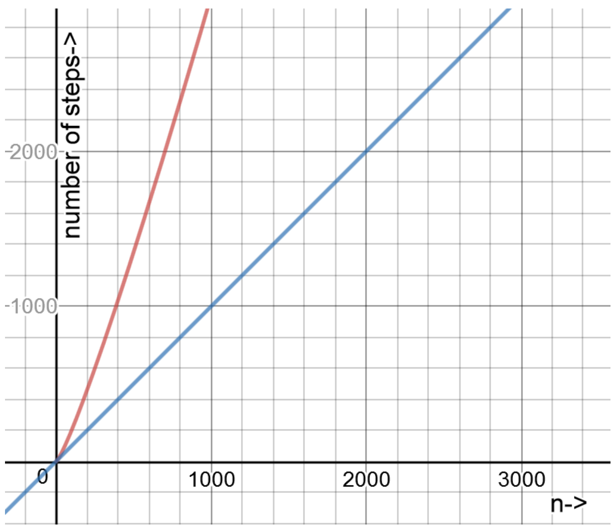
$$T \propto 6 * k + 20 * logn * k$$

$$T \quad \propto \quad O(k * logn)$$

Where k is n

$$T_{worst} \quad \propto \quad O(n * logn)$$

5 Experimental Analysis



This plot represents best and worst case through blue and pink lines. The best case is when all the elements are equal. And the worst case would be when all the numbers are in descreasing order.

6 Discussion and Conclusion

A heap is one of the important and frequently used data structure. The way we found the k largest elements in that similar way we can find the n largest elements and that is one way of sorting the elements, which is known as heap sort whose worst case complexity is O(nlogn). Also one of the other most important use of heap is the priority queues. Using priority queue we can get the one with the highest and the lowest priority in consstant time.

7 References

- How to solve it by Computer By R.G. Dromney
- Introduction to Algorithm 3rd Edition By Thomas H.Cormen
- overleaf.com
- tex.stackexchange.com
- latex.org