DAA Assignment No. : 7 Trace the movements of each elements and values held by all nodes in a heap

Lovely Digra
IIT2017090

Abhishek Vishwakarma
IIT2017091

Anuj Raj
IIT2017092

***Abstract*—This paper introduces us to an algorithm to trace the movements of each elements and values held by all nodes in a heap. We solve the problem by using max_heapify function. While designing our problem we calculated the time complexity for the algorithm which comes to be O(n*n) and plotted graphs and found the nature of graph is quadratic.**

I. INTRODUCTION

a special kind of binary tree that follows a strict set of rules, and are used to implement things like priority queues and background jobs. But these arent the only things that heaps are good for. It turns out that binary heaps are often used for no other purpose than efficient sorting. Many programs will rely on heap sort since it happens to be one of the most efficient ways to sort an array. They are of two kinds Min Heap and Max heap. For this problem we would be talking about min_heap. Therefore, further discussion consider heap as min_heap. In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree. Similarly, for max_heap the key present in max_heap at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

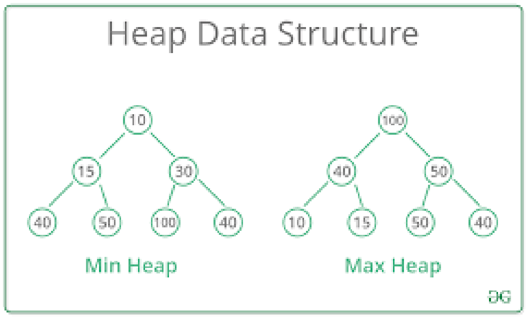

Fig. 1: Heaps

II. ALGORITHM DESIGN AND EXPLANATION

Heap data structure are be implemented generally in two ways. One is nodes tree structure and the other is arrays. Implementation in arrays is widely used because of its easy implementation and space efficiency. Thus the later implementation is followed in the proposed algorithm. Input elements are stored in normal array. Then the array is converted to heap. Array to heap conversion is done using two sub programs. Heapify and Build-Heap. The movement of each node while building heap(min_heap) . This problem can be solved in two ways either by using iterative approach or by implementing recursive method.

Approach 1: Using Recursion

Let an arr[0...n-1] be an input array of size 'n' which stores elements of heap. To trace the movements of each elements and values held by all nodes in a heap first there is need to build heap for that call function build_heap(). As, heap is almost complete binary tree which means there are two types of nodes that are leaf nodes and non-leaf nodes. And we know that leaf nodes are nodes whose left and right children are null. Therefore, leaf nodes always satisfy the condition of min_heap. Thus in build_heap() function iterate an array from last non-leaf node to the root of heap and call min_heapify() function for each element. In min_heapify() function calculate index of left and right children and compare their value with key. And now, check the key present at the current root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree. If not so then swap the key with the smallest element and print the heap to track the movements of each element in a heap. As heap is modified now you need to check for its left and right sub-trees whether they satisfy the condition of min_heap or not for that call min_heapify() function on the location of smallest element. At last, print the final created heap that is min_heap.

This approach is similar to iterative approach but there is a bit difference that is in recursion as we need to call function again and again which consumes a lot of space complexity. In this case space complexity taken by min_heapify() function is o(log(n)) but in case of iterative approach there is a base condition instead of calling function again and again.

Approach 2: Using iterative method

The optimized solution of this problem is Iterative approach. Similar to recursive approach take input from user in an array format and for building heap call build_heap() function. But there is a difference in min_heapify() function. In case of iterative approach there is a base condition which need to be checked every time that is location of smallest element should be always less than the size of actual heap if not so then come out of the loop otherwise keep updating the value of smallest position with the index of smallest element out of left or right child. And at last, print the final min_heap.

Algorithm 1 Trace movement of elements in min-heapify (Iterative)

```
1:  procedure MINHEAPIFY($int\ A[], int\ size, int\ pos$)
2:      $smallest \leftarrow pos$
3:      while $pos < size$ do
4:          $left \leftarrow 2 * pos + 1$
5:          $right \leftarrow 2 * pos + 2$
6:          if $left < size$ and $A[left] < A[smallest]$ then
7:              $smallest \leftarrow left$
8:          if $right < size$ and $A[right] < A[smallest]$ then
9:              $smallest \leftarrow right$
10:         if $smallest =\ pos$ then
11:             $temp \leftarrow A[pos]$
12:             $A[pos] \leftarrow A[smallest]$
13:             $A[smallest] \leftarrow temp$
14:             Print the array
15:             $pos \leftarrow smallest$
16:         else
17:             break
18: procedure BUILDHEAP($A[], size$)
19:     $s \leftarrow (size/2) - 1$
20:     for $I$ from $s \rightarrow 0$ do
21:         $minheapify(A, size, I)$
22: procedure MAIN
23:     $Taking\ the\ inputs$
24:     $Calling\ buildHeap()$
```

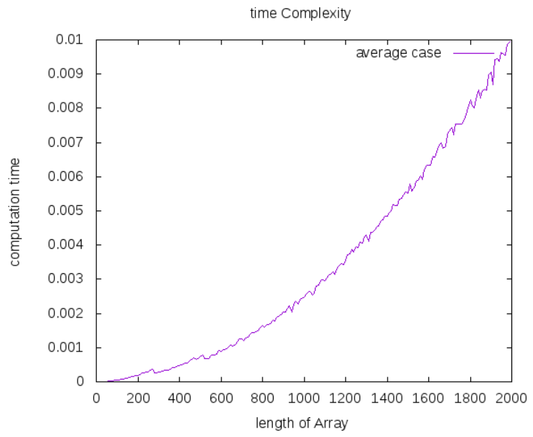

Fig. 2: Average case

Algorithm 2 Trace movement of elements in min-heapify (Recursive)

```
    procedure MINHEAPIFY($int\ A[], int\ size, int\ pos$)
2:      $smallest \leftarrow pos$
        $left \leftarrow 2 * pos + 1$
4:      $right \leftarrow 2 * pos + 2$
        if $left < size$ and $A[left] < A[smallest]$ then
6:          $smallest \leftarrow left$
        if $right < size$ and $A[right] < A[smallest]$ then
8:          $smallest \leftarrow right$
        if $smallest\ != \ pos$ then
10:         $temp \leftarrow A[pos]$
            $A[pos] \leftarrow A[smallest]$
12:         $A[smallest] \leftarrow temp$
            Print the array
14:         $minHeapify(A, size, smallest)$
    procedure BUILDHEAP($A[], size$)
16:     $s \leftarrow (size/2) - 1$
        for $I$ from $s \rightarrow 0$ do
18:         $minheapify(A, size, I)$
    procedure MAIN
20:     $Taking\ the\ inputs$
        $Calling\ buildHeap()$
```

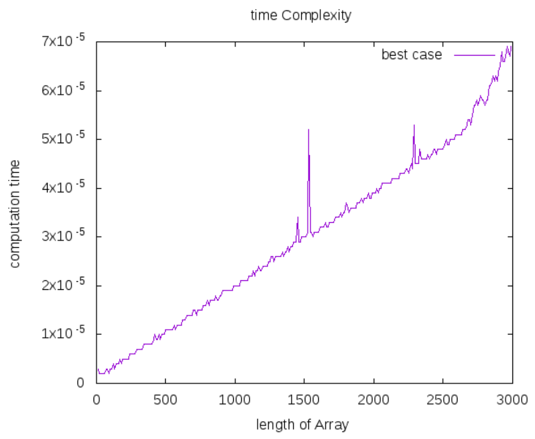

Fig. 3: Best Case

III. EXPERIMENTAL STUDY

For the experimental analysis of this algorithm, size of the input is array varied, to see how the execution time varies with input size. The inputs are taken from the standard random generator in the range $1 < E < 1000$. Where E is the element. The graph of length vs execution time. is plotted using .dat file generated from the source code. see Fig 2.

IV. Time Complexity and Discussion

In the algorithm design section, it is discussed that array is converted into Heap (min Heap). This is done using the by applying buildHeap and minHeapify. Every time when there is a swap that is there is a movement of elements in the array, it is traced by printing the array after the swap. Thus time complexity of the algorithm will be no. of swaps x complexity of the printing the whole array of size n.

No. of swaps is actually time complexity of BuildHeap.time complexity of buildHeap is $O(n)$. This can be understood using following facts. Heap is a complete binary tree in which every level except the last one has $\frac{n}{2^{h+1}}$ elements. All the $\frac{n}{2}$ leaf nodes are already heaps of size 1. The next level has $\frac{n}{4}$ nodes which can have maximum of one swaps. This goes like:

$$1.\frac{n}{4} + 2.\frac{n}{8} + ... + (h.1) = n$$

Thus worst case total complexity of the algorithm is $O(n^2)$. This is depicted by the graph shown in Fig 2. Best case can be the case when the input is already in min Heap. In that case, time complexity only depends number of times the heapify function is called in the build heap function. i.e $\frac{n}{2} - 1$. Thus execution takes linear time as depicted in Fig 3.

Space Complexity: resursive algorithm has a runtime stack overhead of $O(\log_2 n)$, but this has been improved by iterative algorithm . Therefore, space complexity is $O(1)$.

V. Conclusion

We designed an algorithm to trace the movements of each elements and values held by all nodes in a heap. We also analyzed the algorithm that resulted in general time complexity of $O(n^2)$ and plotted the graph of size of an array vs time elapsed and found that the nature of graph is convex.

References

[1] https://www.geeksforgeeks.org/heap-data-structure/
[2] https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/
[3] https://www.geeksforgeeks.org/heap-sort/