Trace the Movement of Elements & Find the Most Stable and Least Stable Elements During Heap Sort

Ajay Saini
Dept. of Information Technology,
Indian Institute of Information Technology, Allahabad

Ayonya Prabhakaran
Dept. of Information Technology,
Indian Institute of Information Technology, Allahabad

Ankit Mund
Dept. of Information Technology,
Indian Institute of Information Technology, Allahabad

***Abstract*—The paper details the algorithm, that traces the movement of every node in a max heap and finds the element that moved the most and the one that is most stable during heap sort. The proposed algorithm will output movement path of each node during heap sort and at print the element(s) that moved the most and the element(s) which is/are most stable.**

***Index Terms*—heaps, heap sort, binary heap, child node, parent node, paths, arrays.**

I. Introduction

A heap is a specialized tree-based data structure that satisfies the heap property: if P is a parent node of C, then the key (the value) of node P is greater than or lesser than the key of node C. A heap can be classified further as either a “max heap” or a “min heap”. In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node. In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node. A complete binary max heap can be seen in Fig 1. Heap sort can be performed by extracting top element of heap each time using heap property. The heap can be represented by binary tree or array. In this whole process of heap sort, every node moves many times according to heap operations and our algorithm will trace the path of each nodes movement and will print the most moved element and most stable element.

II. Methods/ Algorithm

A. Input / Output

The user inputs the total number of elements in heap and then all elements in heap one by one. The output is the sorted array and the path of each element during heap sort. The most stable and least stable elements are also printed.

B. Algorithm

In our proposed approach, the user inputs the number of elements and the the elements to be sorted. Since a binary heap is a complete binary tree, it can be easily represented as an array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (the indexing starts at 0). The input is an array of size N. To trace the path of each element we maintain a list that stores the path of each element. The elements in the list is initialized with the initial respective positions in the array. Following this call to the function *build_max_heap()* is made. This is like the main function that does the heap sort. This function in turn calls the *max_heapify()* function and extracts the top element from the max heap and moves it to the last position in the array so that the final array is sorted in ascending order. Successive calls to *max_heapify()* is made so as to sort the elements after the swap of the root with the last position in the array. This results in the array sorted in ascending order finally.

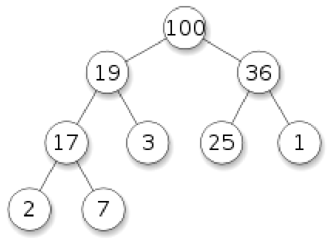

Fig. 1. Example of a complete binary max-heap with node keys being integers from 1 to 100.

The function *max_heapify()* is a recursive function that creates a max heap rooted at the specified index/node in the tree. The function checks to find the greater child among the two children (or one child in some cases) of the specified node. The larger child is then swapped with the specified node and *max_heapify()* is called with the child as the specified index now, so as to recursively build a max heap of the affected sub-tree. If both the children of the specified node(parent) are smaller in value than the parent, then return. The time complexity of heap sort in entirety is $O(Nlog(N))$.

Every time an element is swapped, the path list is updated for both the elements that are swapped. Once heap sort has completed, the sorted output is printed. A call to *find_stable()* is made to find the most and least stable elements. The most stable element was the one that moved the least during heap sort and the least stable the one that moved the most during heap sort. This is found by calculating the length of the path list for each element. The element(s) with the maximum length are the least stable elements and the the element(s) with the minimum length are the most stable elements(we can have multiple most and least stable elements).

The pseudo code for the algorithm proposed is given below:

Algorithm 1 Heap Sort

```
procedure MAIN
    input number of elements: n
    input array: arr
    initialize the list: path
    build_max_heap(arr, n)
    print sorted output
    find_stable(arr, n)

function BUILD_MAX_HEAP (arr, n)
    $i = n/2 - 1$
    while $i >= 0$ do
        max_heapify (arr, n, i)
        $i - -$
    $i = n - 1$
    while $i >= 0$ do
        path[i].add(0)
        path[0].add(i)
        swap(arr[i], arr[0])
        max_heapify (arr, i, 0)
        $i - -$

function MAX_HEAPIFY (arr, n, i)
    $max \leftarrow i$
    $l \leftarrow 2 * i + 1$
    $r \leftarrow 2 * i + 2$
    if $l < n$ and $arr[l] > arr[max]$ then
        $max \leftarrow l$
    if $r < n$ and $arr[r] > arr[max]$ then
        $max \leftarrow r$
    if $max \; ! = i$ then
        path[i].add(max)
        path[max].add(i)
        swap(arr[max],arr[i])
        max_heapify (arr, n, max)
function FIND_STABLE (arr, n)
    $max\_path \leftarrow INT\_MIN$
    $min\_path \leftarrow INT\_MAX$
    for each element $\in$ path do
        Compare with path[element].size() and
        obtain most stable and least stable elements
    print all the most stable and least stable elements
```

The time complexity of the above algorithm is $O(Nlog(N)) + O(N)$ for heap sort and finding the stable elements respectively, thus the overall time complexity is $O(Nlog(N))$. The space complexity is N for the array and the list containing the path which will be N*(length of each element path)*, hence giving a space complexity $O(N)$.

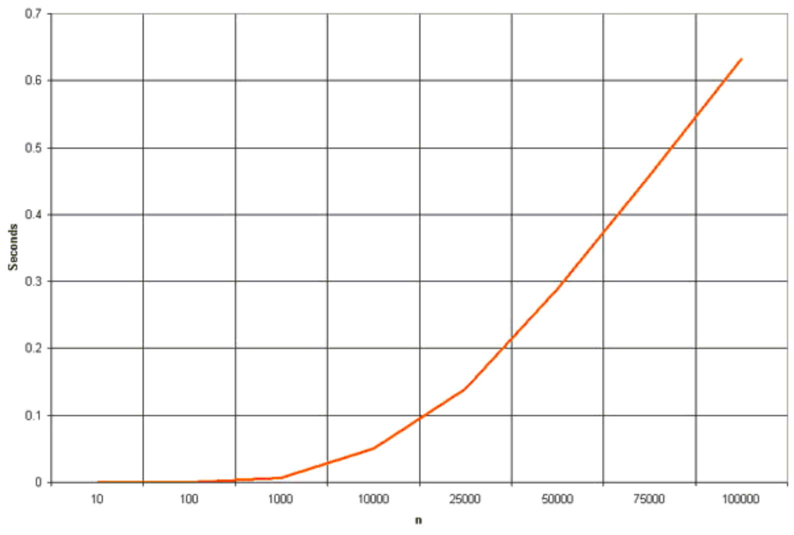

Fig. 2. Plot of time vs. number of element(N).

C. *Runtime Analysis*

As the number of inputs increases the time complexity increases propotionaly as a funcion of $N * log(N)$. Figure 2 shows the plot of Time vs the Number of input elements.

III. Conclusion

The above paper details the implementation of the algorithm which solves the problem tracing the movement of every number in the heap during heap sort and find which element moved the most and which one is most stable. The solution has been optimized to use the least amount of space and time possible.

References

[1] Stack Overflow (https://stackoverflow.com)
[2] CS Stack Exchange(https://cs.stackexchange.com)
[3] Geeks For Geeks (http://www.geeksforgeeks.org/)
[4] Google Images (https://images.google.com/)
[5] C Plus Plus Reference(http://www.cplusplus.com)
[6] Wikipedia (https://en.wikipedia.org/)
[7] Wolfram Mathworld (http://mathworld.wolfram.com/)