Design and Analysis of Algorithm for deleting any non-leaf node from a heap and tracing the contents of each node

Nairit Banerjee
IIT2016505

Kamal Nayan Chaubey
IIT2016510

Ashutosh Vishwakarma
IIT2016017

Mandeep Chakma
IIT2016012

March 26, 2018

Abstract

This document is a final report of our study on deleting the non-leaf node of a heap and the resultant structure must also be a heap alongwith the tracing of each node during the execution of our algorithm.

Keywords: *Min-Heap, Max-Heap, Heapify, Trace, Non-Leaf Node, Build Heap*

1 Introduction And Literature Survey

In Computer Science, Heap is a specialised tree based data Structure that satisfies the heap property : if P is a parent node of C then the key(value) of P is either greater than or equal to(in a max-heap) or less than or equal to(in a min-heap) the key of C. The node at the top of the heap (with no parent) is called the Root Node.
Heaps are usually implemented in an array (fixed size or dynamic array). After an element is inserted into or deleted from a heap, the heap property may be violated and the heap must be balanced by internal operations. Binary heaps may be represented in a very space-efficient way(as an implicit data structure) using an array alone. The first (or last) element will contain the root. This allows moving up or down the tree by doing simple index computations.

2 Algorithm Design

We are given a Heap as an input. So first we need to detect whether it a min-heap or max-heap. Deleting any given node(index) creates hole in the node so to fill it we insert the last element of the heap. When inserted the heap becomes complete heap but it no longer remains heap.
To heapify we have to move the inserted element in its suitable position. By comparing its value with parent node and chidren nodes we move it upwards or downwards until it satisfies heap property. Now to trace movement of content in every node we keep track of the content by creating linked list for every node. Whenever there is position change or swap we simply store the content to the linked list of both nodes.Here trace[i] denotes the linked list storing the contents of ith node when they are changed.

2.1 Pseudo Code for the Algorithm

Algorithm 1 Check

```
1: procedure CHECK(heap, n)
2:     flag ← 0 //assumes heap to be max
3:     if heap[1] > heap[2] then
4:         flag ← 1
5:     end if
6: end procedure
```

Algorithm 2 Insertnode

```
1: procedure STRUCT NODE* INSERTNODE(n, trace, value) struct node* tmp
2:     tmp ← (structnode*)malloc(sizeof(structnode))
3:     tmp−>data ← value
4:     tmp−>next ← trace
5:     trace ← tmp    return trace
6: end procedure
```

Algorithm 3 Maxheapify

```
1: procedure MAXHEAPIFY(arr, n, i)
2:     largest ← i
3:     l ← 2 * i //left child
4:     r ← 2 * i + 1 //right child
5:     if l ≤ n AND arr[l] > arr[largest] then
6:         largest ← l
7:     end if
8:     if r ≤ n AND arr[r] > arr[largest] then
9:         largest ← r
10:    end if
11:    if largest ≠ i then
12:        tmp ← arr[i]
13:        arr[i] ← arr[largest]
14:        arr[largest] ← tmp
15:        trace[i] ← INSERTNODE(n,trace[i],arr[i])
16:        trace[largest] ← INSERTNODE(n,trace[largest],arr[largest])
           MAXHEAPIFY(arr, n, largest)
17:    end if
18: end procedure
```

Algorithm 4 Minheapify

```
1: procedure MINHEAPIFY(arr, n, i)
2:     smallest ← i
3:     l ← 2 * i //left child
4:     r ← 2 * i + 1 //right child
5:     if l ≤ n AND arr[l] < arr[smallest] then
6:         smallest ← l
7:     end if
8:     if r ≤ n AND arr[r] < arr[smallest] then
9:         smallest ← r
10:    end if
11:    if smallest ≠ i then
12:        tmp ← arr[i]
13:        arr[i] ← arr[smallest]
14:        arr[smallest] ← tmp
15:        trace[i] ← INSERTNODE(n,trace[i],arr[i])
16:        trace[smallest] ← INSERTNODE(n,trace[smallest],arr[smallest])
           MINHEAPIFY(arr, n, smallest)
17:    end if
18: end procedure
```

Algorithm 5 Main Program

```
1: procedure MAIN
2:     flag ← 0
3:     i ← 1
4:     while i <= n do
5:         trace[i] ← NULL
6:     end while
7:     flag ← CHECKHEAP(heap,n)
8: k is the index to delete(non leaf)
9:     if k ≤ 0 OR k ≥ n + 1 OR then
10:      print("Invalid Node")
11:      return 0
12:    end if
13:    if 2 * k ≥ n + 1 then
14:      print("It's a leaf node")
15:      return 0
16:    end if
17:    l ← heap[k]
18:    heap[k] ← heap[n]
19:    heap[n] ← l
20:    trace[k] ← INSERTNODE(n,trace[k],heap[k])
21:    if flag==1 then
22:      MAXHEAPIFY(heap,n-1, k)
23:      print the trace array of link list
24:      return 0
25:    end if
26:    if flag==0 then
27:      MINHEAPIFY(heap,n-1, k)
28:      print the trace array of link list
29:      return 0
30:    end if
31: end procedure
```

3 Analysis and Discussions

3.1 Time Complexity

The time complexity for the above stated algorithm can be computed by adding units of time taken by each statement multiplied by the number of times the statement is executed. Hence,

$T = \Sigma(cost(i) * no.of.times(i))$,for all statements

Based on the position of the non-leaf node we wish to remove, the running time can be divided in three classes. Best, if only one swap is required to heapify after removal. Worst, if the root node is removed, and whole heap is required to be heapified, Average, for any intermediate case.

$$T_{best} \propto n(26 + 4c_1) + 67$$

$$T_{average} \propto n(26 + 4c_1) + (log(n) - c_2)43 + 24$$

$$T_{worst} \propto n(26 + 4c_1) + log(n)43 + 24$$

Hence the order of growth for best case, average case and worst case are

$$\Omega(n), \Theta(n), O(n) \text{ respectively.}$$

Here n is the number of nodes in the heap. Varying the value of n over a range 7 to 21, we get the following running times, for the three classes of inputs.

Table 1: Data for plotting running time graph

n	T_{best}	T_{avg}	T_{worst}
7	159	167	175
11	235	243	259
15	311	319	335
21	425	433	457

We will get the following running time graph for n ranging from 7 to 21. The x-axis represents the number of node and y-axis represents running time.

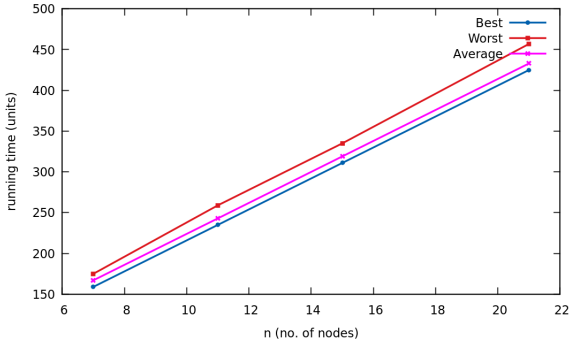

Figure 1: Running time graph.

3.2 Space Complexity

Apart from a few variables, we are generating an array of linked lists whose sizes are not fixed. Whenever a new value appears at a node, the size of the linked list increases by one. Initially all the linked lists have one node, with the initial value of the node in the heap. We can say that,

$$Space(n) \propto n + a_1 + a_2 + ... + a_n$$

where each

$$a_1, a_2, a_3.....a_n$$

is the number of nodes added in that list. Thus we have,

$$Space(n) \propto n + c$$

4 Experimental Study

We may assume the following examples to study the various running times possible,

Best Case: The best case will be when there is only one swap and only two comparison required to heapify , and this condition will hold true only for the internal non leaf node which is the parent of a leaf node. For example if we delete 4th index from 20, 19, 18, 17, 16, 15, 14.

Worst Case:This will include the deletion of root node and while heapifying, the whole heap is traversed up to the leaf node and hence the swapping takes place at each step of traversal.For example if we delete 1st index from 20, 19, 18, 17, 16, 15, 14.

Average Case: This will include the deletion of nodes other than the root node and the non-leaf node which is the parent of a leaf node. And hence the number of swaps and comparisons required for the same is more than the best case and less than the worst case inputs.For example if we delete 2nd index from 20, 19, 18, 17, 16, 15, 14.

5 Comparison with Previous Group

Apart from shortening the length of code by quite some extent. The running time expression of the previous group was of the form,

$$nc_1 + log(n)^2 + c_2$$

Whereas we obtained a running time expression of the form,

$$nc_3 + log(n) + c_4$$

$$\text{where } c_3 << c_1 \text{ and } c_4 < c_2$$

This is because, our method to check for a heap to be of max or min type takes constant time, whereas there's take time of the order of n. Moreover, our algorithm does way lesser number of comparisons than there's.

References

[1] Introduction To Algorithms (Second Edition) by Thomas H.Cormen, Charles E.Leiserson , R.L.Rivest and Clifford Stein.

[2] How to Solve it by Computer by R.G.Dromey.

[3] The C Programming Language by Brian Kernighan and Dennis Ritchie.

[4] Geeks for Geeks : https://www.geeksforgeeks.org/

[5] Stack Overflow : https://www.stackoverflow.com/

[6] Wikipedia