An efficient algorithm to delete any non-leaf node of a given Heap and to Trace the content of every node of the Heap during the execution

IIT2016505	IIT2016113	IIT2016120	IIT2016129
Nairit Banerjee	Siddhanth Rao	Avanish Chand	Rohan Bhalerao

I. INTRODUCTION AND LITERATURE SURVEY

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 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 parents) 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. Balancing a heap is done by shift-up or shift-down operations (swapping elements which are out of order).

II. ALGORITHM DESIGN

We are given heap as a input.So first we need to detect whether it is 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 binary tree 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. Given below are various parts of algorithm.The shiftUpMax and shiftUpMin are almost same with only difference that one is for minheap and other for maxheap.Same is the case for shiftDown.

.

Algorithm 1 Main Algorithm

```
function main
//we are given a heap
    int n ← input //size of heap
    i ← 1
    while i<=n do
        heap[i] ← input
        trace[i] ← NULL
        i++
    end while
    flag ← checkheap(n) //max heap or min heap
    k ← input // index of node to be deleted
    swap( heap[k] , heap[n] )
    trace[k] ← insertnode(n,trace[k],heap[k]);
    if (flag==0) then
        if (k! = 1&&heap[k] > heap[k/2]) then
            shiftUpMax( k )
        else
            shiftDownMax( k ,n-1 )
        end if
    else
        if (k! = 1&&heap[k] < heap[k/2]) then
            shiftUpMin( k )
        else
            shiftDownMin( k ,n-1 )
        end if
        shiftUpMin( k )
    end if
end function
```

Algorithm 2 shiftUpMax

```
function shiftUpMax(int nodeIndex)

    if (nodeIndex! = 1) then
        parent ← (nodeIndex/2)

        if (heap[parent] < heap[nodeIndex]) then
            swap(heap[parent], heap[nodeIndex])
            trace[parent] ←
insertnode(n, trace[parent], heap[parent])
            trace[nodeIndex] ←
insertnode(n, trace[nodeIndex], heap[nodeIndex])
            shiftUpMax(parent)
        end if
        end if
    end function
```

Algorithm 3 shiftUpMin

```
function shiftUpMin(int nodeIndex)

    if (nodeIndex!=1) then

parent ← nodeIndex/2
        if (heap[parent] > heap[nodeIndex]) then
            swap(heap[parent], heap[nodeIndex])
            trace[parent] ←
insertnode(n, trace[parent], heap[parent])

            trace[nodeIndex] ←
insertnode(n, trace[nodeIndex], heap[nodeIndex])

            shiftUpMin(parent)
        end if
        end if
    end function
```

Algorithm 4 shiftDownMax

```
function shiftDownMax(int i, int n)
    left ← 2 * i
    right ← 2 * i + 1
    if (left <= n&heap[left] > heap[i]) then
        largest ← left
    else
        largest ← i
    end if
    if (right <= n&heap[right] > heap[largest]) then
        largest ← right
    end if
    if (largest! = i) then
        swap(heap[i], heap[largest])
        trace[i] ← insertnode(n, trace[i], heap[i])
       trace[l] ← insertnode(n, trace[largest], heap[largest])
        shiftDownMax(largest, n)
    end if
end function
```

Algorithm 5 shiftDownMin

```
function shiftDownMin(int i, int n)
    left ← 2 * i
    right ← 2 * i + 1
    if left <= n&heap[left] < heap[i] then
        smallest ← left
    else
        smallest ← i
    end if
    if right <= n&heap[right] < heap[s] then
        s ← right
    end if
    if s! = i then
        swap(heap[i], heap[smallest])
        trace[i] ← insertnode(n, trace[i], heap[i])
              trace[smallest] ←
insertnode(n, trace[smallest], heap[smallest])
        shiftDownMin(smallest, n)
    end if
end function
```

Algorithm 6 insertnode

```
function insertnode * (int n, node * trace, intvalue)
    temp ← new node
    temp− > data ← value
    if (trace==NULL) then

        trace ← temp
        temp− > next ← NULL          return trace
    else
        p ← trace
        while (p− > next! = NULL) do
            p ← p− > next
        end while
        p− > next ← temp
        temp− > next ← NULL
          return trace
    end if
end function
```

III. ANALYSIS

In this algorithm we are deleting in O(1) and then to satisy heap property we go on swapping the element until it fits in appropriate position.During this we are also adding positions of it in linked list which can take O(n).Overall the complexity

Algorithm 7 checkheap

```
function checkheap(int n, )

    flag ← 0
    i ← 1
    while i<=n/2 do
        left  ← 2 * i
        right  ← 2 * i + 1
        if left <= n then

            left ← heap[left]
        end if
        if right <= n then

            right ← heap[right]
        end if
        if left > heap[i]||right > heap[i] then

            flag ← 1
        end if
        i  ← i + 1
    end whilereturn flag
end function
```

of the algorithm will be O(logn). The analysis can be divided into three parts :

A. Worst case:

The worst case will be when swapping goes throughout the height of the heap. That is if we have to delete the root and replacing it with last element so the element needs to moved to last level.

$$t_{\text{checkheap}} \;\alpha\; 2 + 14 * (n/2)$$

$$t_{\text{shiftUp}} \;\alpha\; (log(n) - 1) * log(n) + 6 * (log(n) - 1)$$

$$t_{\text{shiftDown}} \;\alpha\; (log(n) - 1) * log(n) + 8 * (log(n) - 1)$$

Taking shiftDown ,

$$t_{\text{main}} \;\alpha\; 2 + 7 * n + log(n)^2 + log(n) + 8 * log(n) - 8$$

$$t_{\text{main}} \;\alpha\; 2 + 7 * n + log(n)^2 + log(n) + 6 * log(n) - 6$$

$$t_{\text{worst}} \;\alpha\; O(n)$$

The deletion has time complexity O(log(n)).The n factor is due to checking whether it is min heap or max heap.

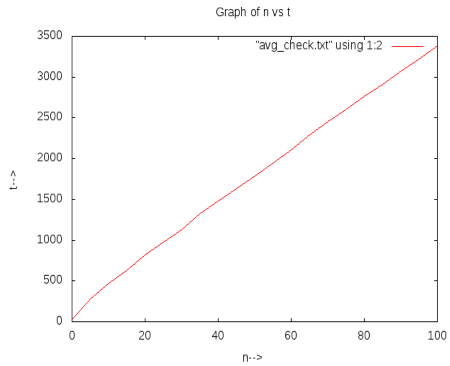
B. Best Case:

Best case will occur when we need only one swap to satisfy the heap property.

$$t_{\text{checkheap}} \;\alpha\; 2 + 14 * (n/2)$$

$$t_{\text{shiftUp}} \;\alpha\; 8$$

$$t_{\text{shiftDown}} \;\alpha\; 10$$



$$t_{\text{Best}} \;\alpha\; \text{2+7*n+8+10}$$

The n factor is same due to checking of heap.Best case heapify will be O(1).

C. Average case:

Average case can be considered as any input in which there is less swapping as compared to worst case.The deleting and heapifying will take O(log(n)).And initially for checking heap O(n).So the average case is $\Theta(n)$

$$t_{\text{Avg}} \;\alpha\; \Omega(n)$$

IV. EXPERIMENTAL STUDY

We try to analyze time or number of steps for various inputs.Below given are some tables and graphs that show number of elements and time complexity relationship and we see a linear relationship among the two.

n	time
5	277
10	475
15	625
20	823
25	973
30	1134
35	1321
40	1482
45	1632
50	1793
55	1943
60	1793
65	1793
70	1793
75	2602
80	2763
85	2913
90	3074
95	3224
100	3385

.

V. DISCUSSION

A. For Heap

Heaps are usually use for operations like findingMin (or Max), deleteMin (or Max) or Insert types of operations due to their optimal performance in time complexities of order log(n).
However, due to tracing part complexity got increases.

B. Alternative for tracing :

We were asked in the question to trace contents of every node of the heap. There are two approaches to do this. Either we can use a 2D array or we could use a linked list.

i) For 2D array each row would denote the index of the heap and elements of that row tells us the numbers that have been there at that index. This approach would require us to declare a 2D array of fixed size initially.This would waste our memory since it is not necessary that whole of the 2D array gets used up. So this is not a space efficient algorithm.

ii) For Linked List we could make an array of linked list where each index of the array of linked list denotes the index of a node in the heap. The value at nodes in ith linked list are the values of the elements that have been there at that index. In this approach we only create a node when we require it. So no memory gets wasted. And thus it is space efficient and better than using a 2D array.

VI. CONCLUSION

The main module of the algorithm design requires us to delete a non-leaf node from a heap while tracing the contents at every node of the heap.
The algorithm has been discussed in detail with a detailed analysis over a given set of n .

REFERENCES

[1] tex.stackexchage.com
[2] wikipedia.com
[3] mathcs.emory.edu