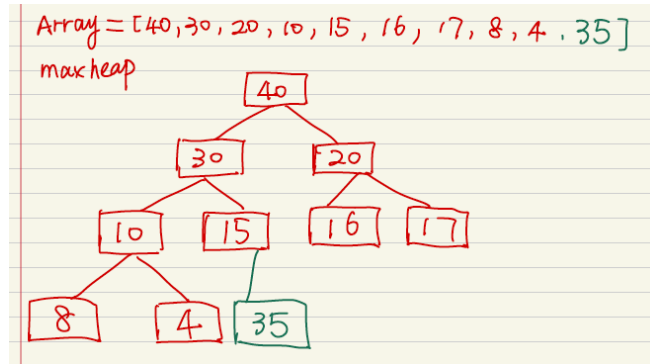
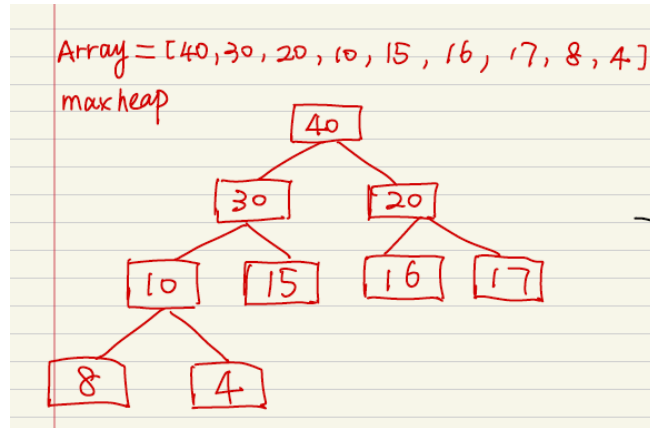
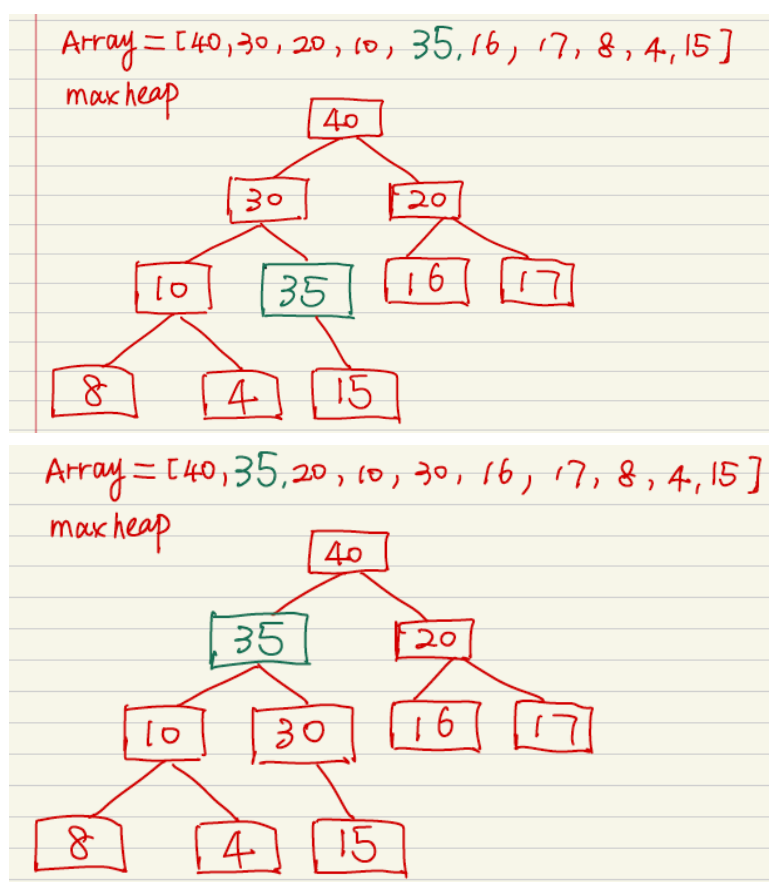


# Question 1



1. The array is shown as array[], and the maxheap is as shown

2. Adding 35 to the end of the array[]; 35 is the left node of the node 15;



3. As a left child, 35 is larger than 15, we need to swap their positions; now 35 becomes the parent node of 15.

4. As the left child of 30, 35 is larger than 30, we need to swap their position.

5. Now it complies with the heap order, we stop here.

Complexity: Since every time we need to compare and swap the inserted node with its parent node, at the worst case, the maximum times we need to swap is the layers of the maxheap. Therefore, the insertion time complexity is  $O(\log N)$ .

## Question 2

The time complexity to find the smallest element in a maxheap containing  $n$  elements is  $O(n)$ .

Due to the property of Maxheap, the parent node is always larger than the child node, which means that the leaf nodes is always not larger than their parent nodes. As a complete binary tree, there are around  $n/2$  leaf nodes. To find the smallest element in the maxheap, we just need to search it in the later half of the array, which has a length of  $n/2$ . Therefore, the time complexity is  $O(n/2)=O(n)$ .

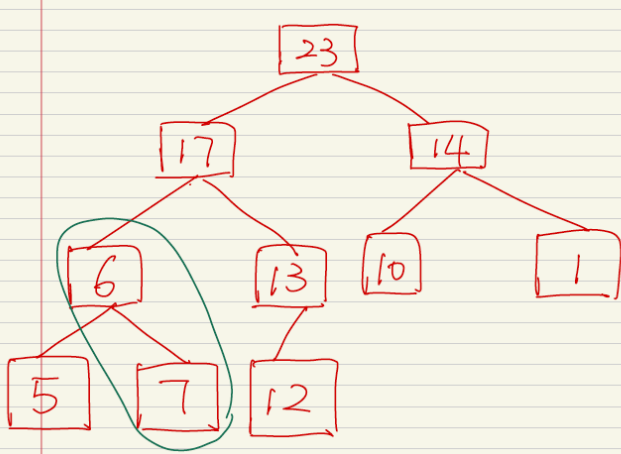
### Question 3

No, it's not a maxheap.

When we build the maxheap according to the array, we need to satisfy the rules of a heap which is a complete binary tree.

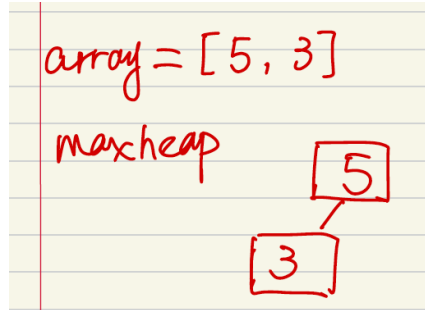
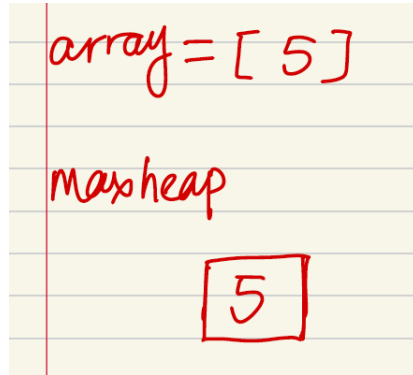
For the complete binary tree, every level is completely filled except for the last leaf nodes, and all nodes are filled from left to right.

When examine the heap as below, we find that node 6 is a parent node of node 7 but it's smaller than node 7, which is not complied with rules of a maxheap (parent node is always no smaller than child node). Therefore, it's not a maxheap.



## Question 4

### Maxheap----Top to down



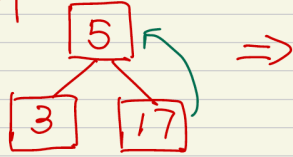
1. Insert 5.

2. Insert 3 as the leaf node of maxheap, since 3 is smaller than 5, maxheap is satisfied.

3. Insert 17 as the leaf node of maxheap, since 17 is larger than its root 5, we need to heapify and swap them. Now maxheap is satisfied

array = [5, 3, 17]

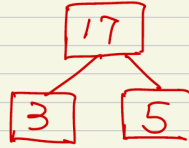
maxheap



⇒

array = [17, 3, 5]

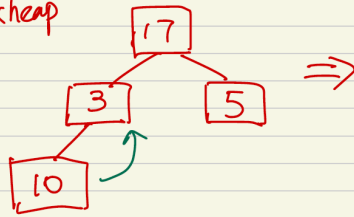
maxheap



4. Insert 10 as the leaf node of maxheap, since 10 is larger than its root 3, we need to swap them. Now maxheap is satisfied.

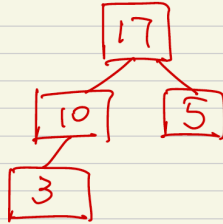
array = [17, 3, 5, 10]

maxheap



⇒

array = [17, 10, 5, 3]

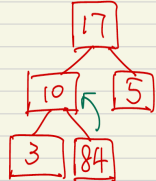


5. Insert 84 as the leaf node of maxheap. Since 84 is larger than its parent node 10, we need to swap them. And then 84 is larger than its parent node 17, we need further to swap them. Now maxheap is satisfied.

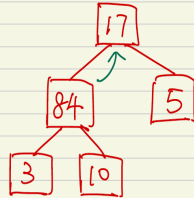
array = [17, 10, 5, 3, 84]

array = [17, 84, 5, 3, 10]

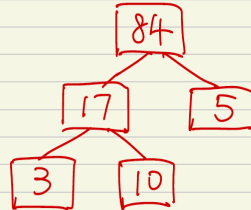
array = [84, 17, 5, 3, 10]

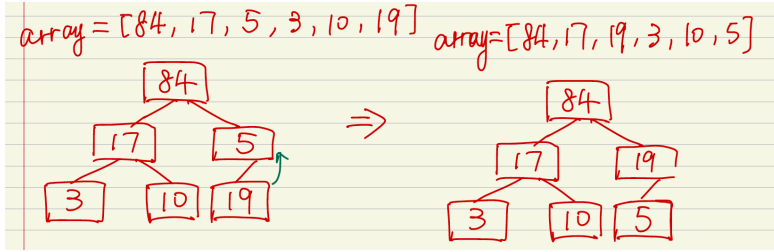


⇒

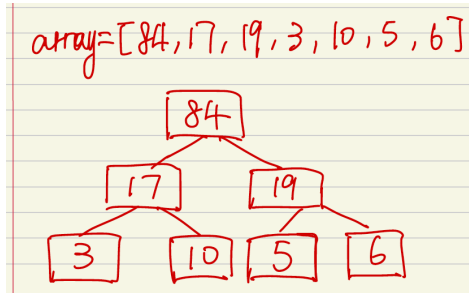


⇒

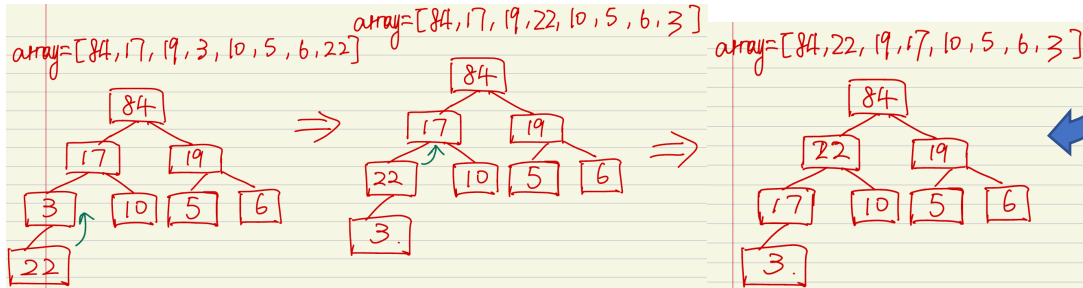





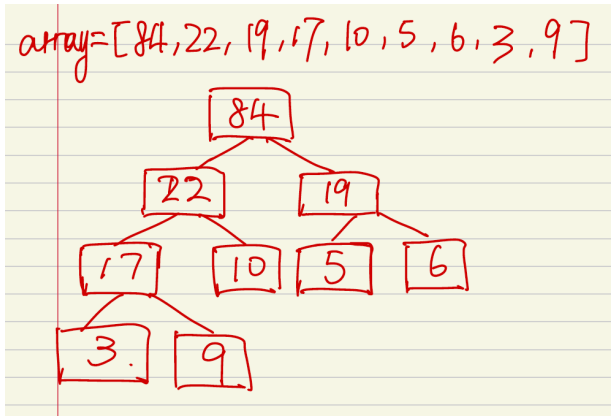
6. Insert 19 as the leaf node of the maxheap, since 19 is larger than 5, we need to swap them.



7. Insert 6 as the leaf node of the maxheap, now maxheap is satisfied. No swap.



8. Insert 22 as the leaf node of the maxheap, since 22 is larger than 3, swap them. Then 22 is larger than 17, swap them. Now maxheap is satisfied.



9. Insert 9 as the leaf node of maxheap. Now the maxheap is satisfied. No swap.

Complexity:

Using the top-down method, we need to insert  $n$  elements into the heap. And at the worst case, for the  $N$ th element, after the insertion, we need to heapify  $\log(N)$  to satisfy the maxheap rules. And for each layer of the maxheap, the sum of swap at the worst case is:

$$1 \cdot h_0 + 2 \cdot h_1 + 4 \cdot h_2 + \dots + \frac{n}{4} \cdot \log\left(\frac{n}{4}\right) + \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = O(n \log n)$$

Therefore, the time complexity is  $O(N \log N)$ .

Space complexity is  $\theta(n)$ .

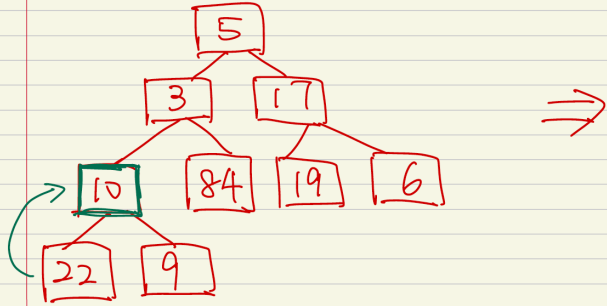


# Question 4

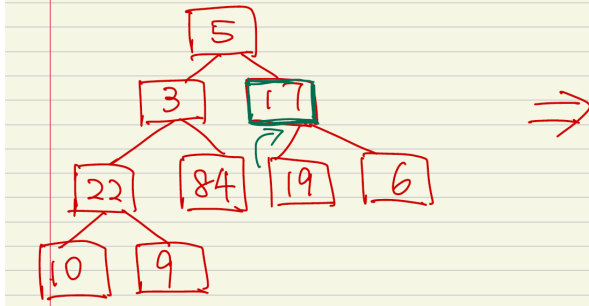
## Maxheap----Down to Top

1. First of all, we need to build a complete binary tree according to the array. Then we iterate from the last parent node to the root in a reverse way.

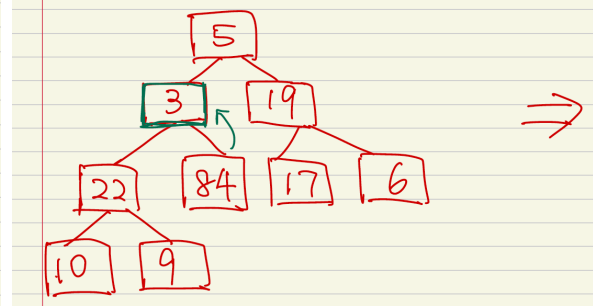
array = [5, 3, 17, 10, 84, 19, 6, 22, 9]



array = [5, 3, 17, 22, 84, 19, 6, 10, 9]



array = [5, 3, 19, 22, 84, 17, 6, 10, 9]

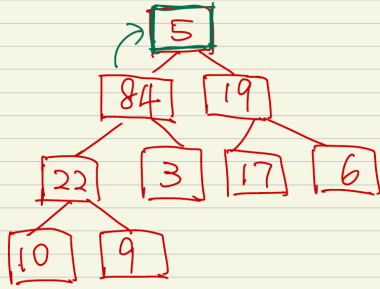


2. Start from the last parent node, skip over all the leaf nodes, starting from the last parent node 10, we find that 10 is smaller than its child node 22, we swap them. Now node 22 satisfies maxheap rules.

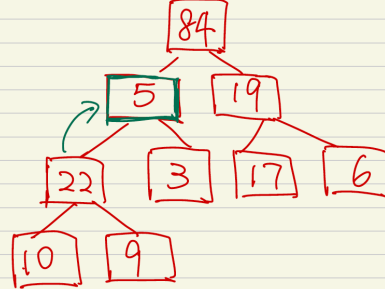
3. Jump to the next last parent node 17, it is smaller than its child node 19, we swap them. Now node 19 satisfies maxheap rules

4. Jump to the next last parent node 3, it is smaller than its child node 84, swap them. Now node 84 satisfies maxheap rules.

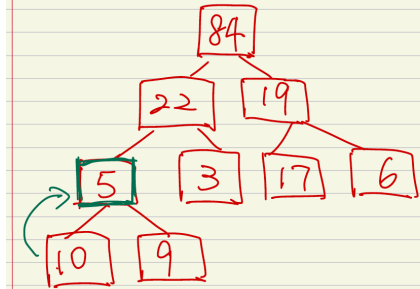
array=[5,84,19,22,3,17,6,10,9]



array=[84,5,19,22,3,17,6,10,9]



array=[84,22,19,5,3,17,6,10,9]

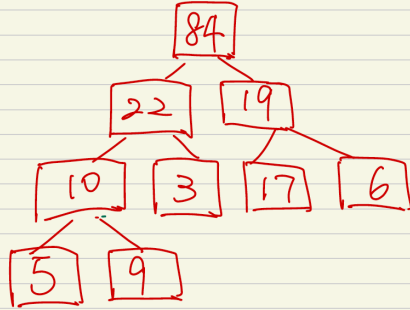


5. Jump to node 5, the root node, it's smaller than its child 84, we swap them. But node 84 right now does satisfy the maxheap rules because its child node 5 breaks the rules.

6. Node 5 is smaller than its child node, we swap them. Still, node 22 does not satisfy maxheap due to its child node 5 breaks the rules.

7. We then keep swap node 5 with its child node 10 (the larger one compared with node 9).

array = [84, 22, 19, 10, 3, 17, 6, 5, 9]



8. Now, each parent node in the heap complies with the maxheap rules.

Complexity:

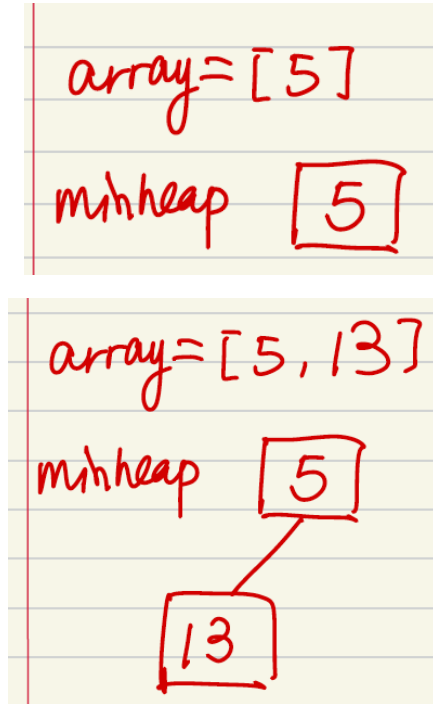
Using down to top method to build the maxheap, we can skip all the leaf nodes, and considering only the  $n/2$  parent nodes. For the  $n/2$  parent nodes, At the worst case, the root node need to swap  $\log(n)$  times to bottom, and the second layer nodes need to swap  $\log(n/2)$ . Therefore the total swap times would be:

$$1 \cdot \log(n) + 2 \cdot \log(n/2) + 4 \cdot \log(n/4) + \dots + n/4 \cdot 2 + n/2 \cdot 1 = O(n)$$

Therefore, the time complexity is  $O(n)$ . Space complexity is  $\theta(n)$ .

## Question 5

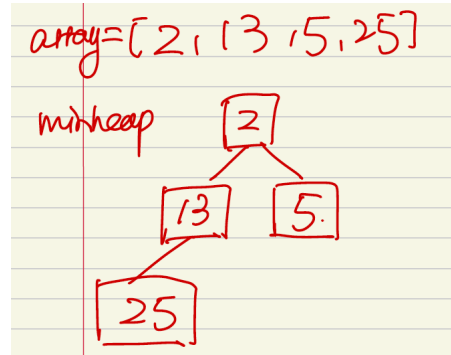
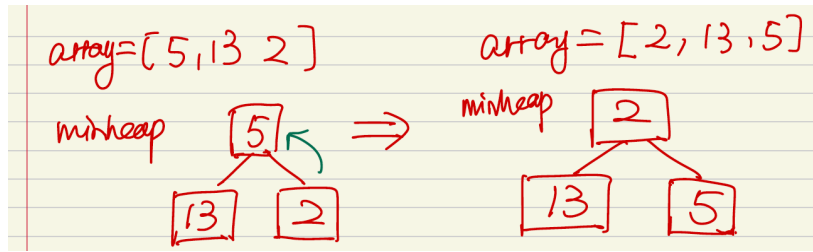
### Minheap---Top to down



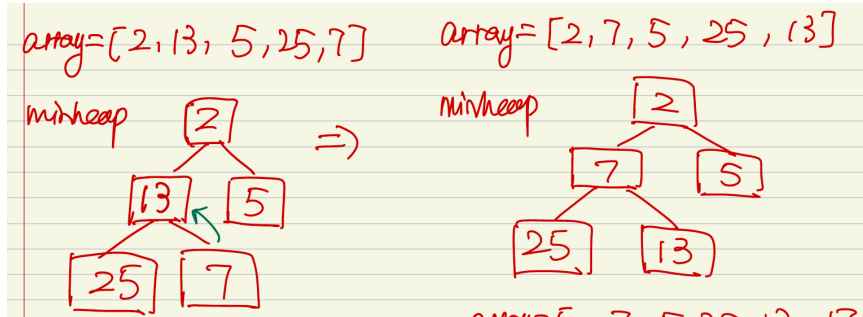
1. Insert 5.

2. Insert 13 as the leaf node of minheap, since 13 is larger than 5, minheap is satisfied.

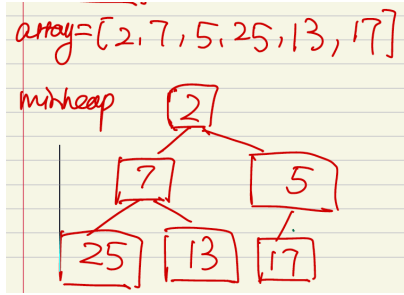
3. Insert 2 as the leaf node of minheap, since 5 is larger than its child node 2, we need to swap them. Now minheap is satisfied



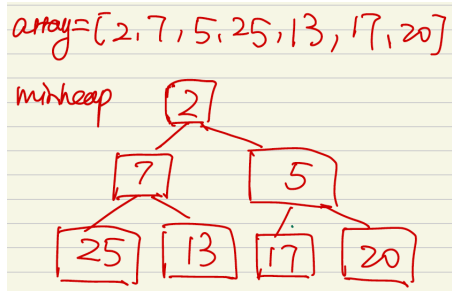
4. Insert 25 as the leaf node of minheap, now minheap is satisfied. No swap.



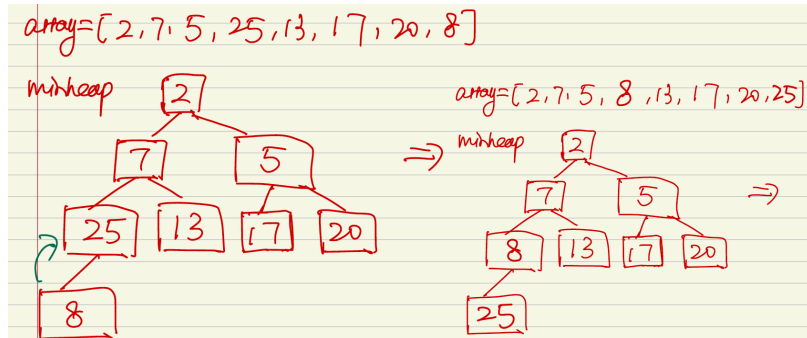
5. Insert 7 as the leaf node of minheap. Since 7 is smaller than its parent node 13, we need to swap them. Now minheap is satisfied.



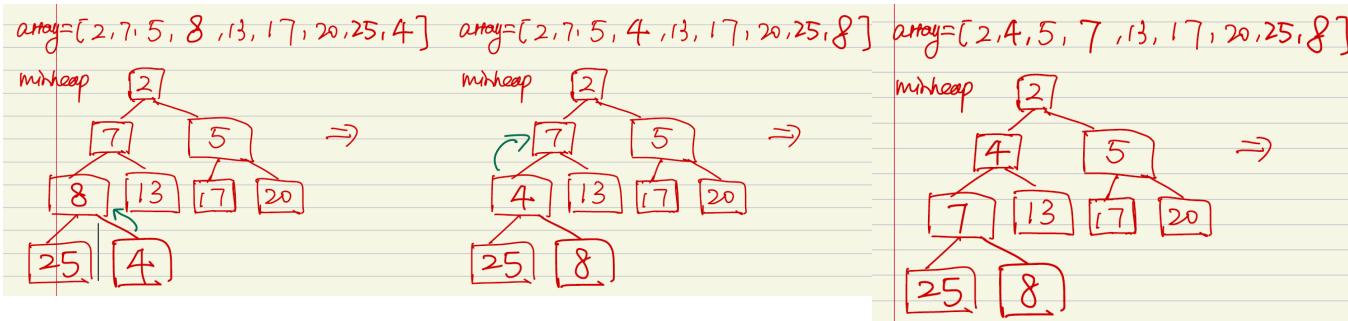
6. Insert 17 as the leaf node of the minheap, since 17 is larger than 5, minheap is satisfied. No swap.



7. Insert 20 as the leaf node of the minheap, now minheap is satisfied. No swap.



8. Insert 8 as the leaf node of the minheap, since 8 is smaller than 25, swap them. Now minheap is satisfied.



9. Insert 4 as the leaf node of minheap. Since 4 is smaller than its parent node 8, we swap them. And then 4 is smaller than its parent node 7, we swap them. Now the minheap is satisfied.

### Complexity:

Using the top-down method, we need to insert  $n$  elements into the heap. Since the insert is adding elements into the end of array. The adding time complexity for each element is  $\theta(1)$ . Therefore, the insertion time complexity is  $\theta(n)$ .

And at the worst case, for the  $N$ th element, after the insertion, we need to heapify/swap  $\log(N)$  times to satisfy the minheap rules. And for each layer of the minheap, the sum of swap at the worst case is:

$$1 \cdot h_0 + 2 \cdot h_1 + 4 \cdot h_2 + \dots + \frac{n}{4} \cdot \log(n/4) + \frac{n}{2} \cdot \log(n/2) = O(n \log n)$$

Therefore, the time complexity is  $\theta(N) + O(N \log N) = O(N \log N)$ . Space complexity is  $\theta(n)$ .

## Question 5

Delete minheap

array=[2, 4, 5, 7, 13, 17, 20, 25, 8]

1. For the minheap, when we delete min until the minheap is empty, we always do the below steps:
  - (1) swap the root and the last leaf node, which is array[0] and array[end];
  - (2) Delete the array[end], which is the target min.
  - (3) Sippy down the root node, until it satisfies the minheap.



delete 2

array = [2, 4, 5, 7, 13, 17, 20, 25, 8]

array = [8, 4, 5, 7, 13, 17, 20, 25]

array = [4, 8, 5, 7, 13, 17, 20, 25]

array = [4, 7, 5, 8, 13, 17, 20, 25]

delete 4

array = [4, 7, 5, 8, 13, 17, 20, 25]

array = [25, 7, 5, 8, 13, 17, 20]

array = [5, 7, 25, 8, 13, 17, 20]

array = [5, 7, 17, 8, 13, 25, 20]

2. When root is 2, we swap 2 and 8; Delete 2 from the minheap; swap 8 with its child node down until it follows the minheap again.

3. When root is 4, we swap 4 and 25; Delete 4 from the minheap; swap 25 with its child node down until it follows the minheap again.

delete 5

array = [5, 7, 17, 8, 13, 25, 20]  
array = [20, 7, 17, 8, 13, 25]  
array = [7, 20, 17, 8, 13, 25]  
array = [7, 8, 17, 20, 13, 25]

delete 7

array = [7, 8, 17, 20, 13, 25]  
array = [25, 8, 17, 20, 13]  
array = [8, 25, 17, 20, 13]  
array = [8, 13, 17, 20, 25]

4. When root is 5, we swap 5 and 20; Delete 5 from the minheap; swap 20 with its child node down until it follows the minheap again.

5. When root is 7, we swap 7 and 25; Delete 7 from the minheap; swap 25 with its child node down until it follows the minheap again.

delete 8  $\left\{ \begin{array}{l} \text{array} = [8, 13, 17, 20, 25] \\ \text{array} = [25, 13, 17, 20] \\ \text{array} = [13, 25, 17, 20] \\ \text{array} = [13, 20, 17, 25] \end{array} \right.$   
 delete 13  $\left\{ \begin{array}{l} \text{array} = [13, 20, 17, 25] \\ \text{array} = [25, 20, 17] \\ \text{array} = [17, 20, 25] \end{array} \right.$   
 delete 17  $\left\{ \begin{array}{l} \text{array} = [25, 20] \\ \text{array} = [20, 25] \end{array} \right.$   
 delete 20  $\text{array} = [25]$   
 delete 25  $\text{array} = []$

6. The rest follows the same pattern.

Complexity:

To delete  $N$  elements from the minheap, since each element is deleted from the end of the array, the time is  $O(1)$  for each one element, the total time is  $\theta(n)$ ;

However, after each delete, we need to sippy down the root element to maintain the minheap, at the worst case, each element might need  $\log(n)$  swaps. Therefore, the time complexity is  $O(n \log(n))$ .

The total time complexity is:  
 $\theta(n) + O(n \log(n)) = O(n \log n)$

## Question 5

### Maxheap----Top to down

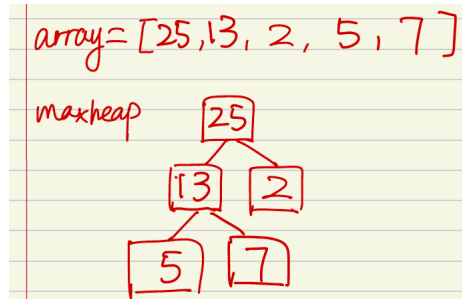
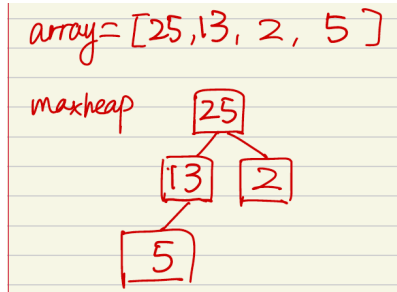
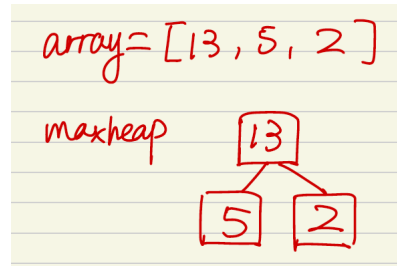
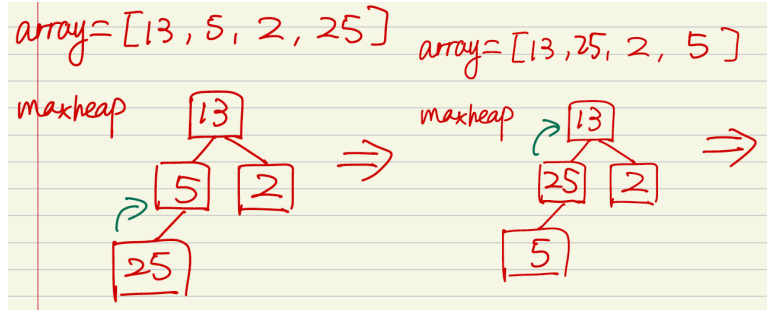
array = [5]  
maxheap 5

array = [5, 13]      array = [13, 5]  
maxheap 5 ⇒ maxheap 13  
13 5

1. Insert 5.

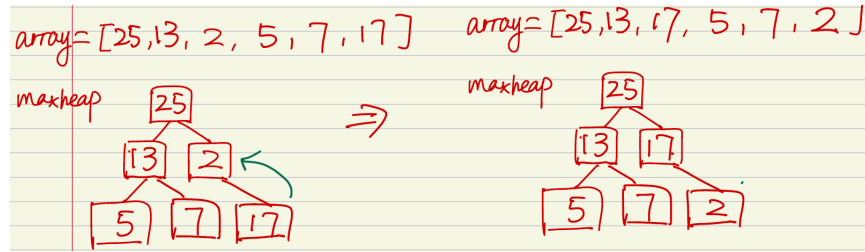
2. Insert 13 as the leaf node of maxheap, since 13 is larger than 5, we need to swap them. Then maxheap is satisfied.

3. Insert 2 as the leaf node of maxheap, since 13 is larger than its child node 2, maxheap is satisfied. No swap

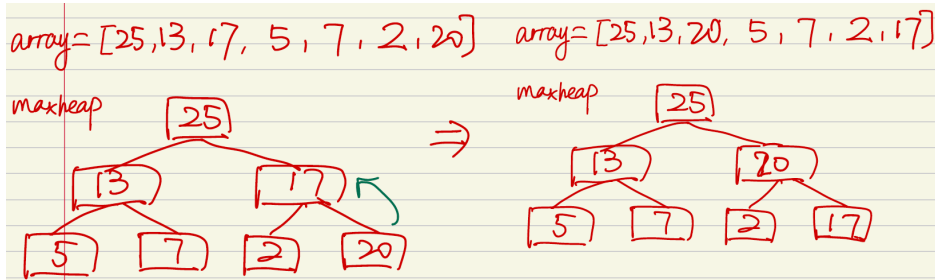


4. Insert 25 as the leaf node of minheap, since 25 is larger than 5, we need to swap them. Then 25 is larger than 13, we need to keep swap until maxheap is satisfied.

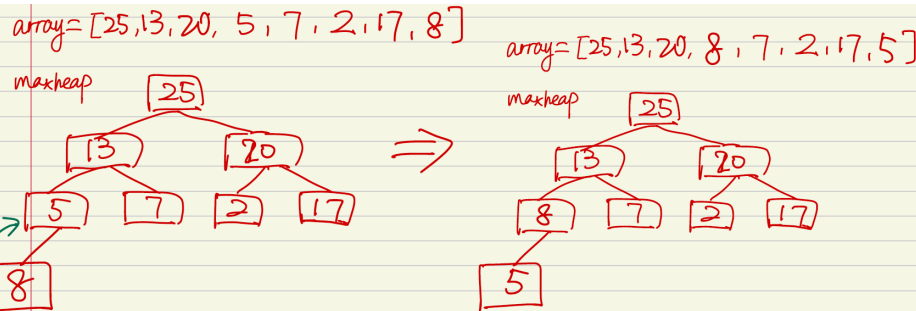
5. Insert 7 as the leaf node of maxheap. Since 7 is smaller than its parent node 13, maxheap is satisfied. No swap.



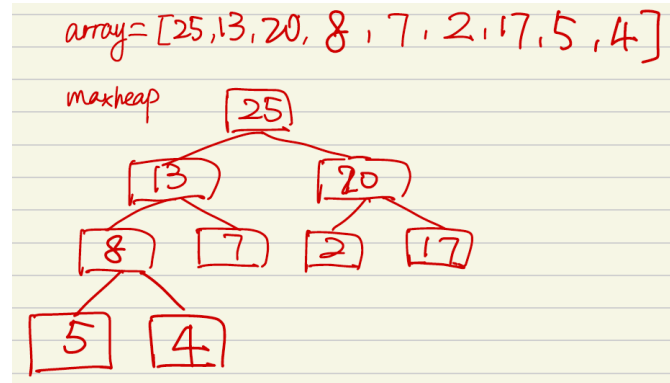
6. Insert 17 as the leaf node of the maxheap, since 17 is larger than 2, we need to swap them. Now maxheap is satisfied.



7. Insert 20 as the leaf node of the maxheap, since 20 is larger than 17, we need to swap. now maxheap is satisfied.



8. Insert 8 as the leaf node of the maxheap, since 8 is larger than 5, swap them. Now maxheap is satisfied.



9. Insert 4 as the leaf node of maxheap. Now the maxheap is satisfied. No swap.

### Complexity:

Using the top-down method, we need to insert  $n$  elements into the heap. Since the insert is adding elements into the end of array. The adding time complexity for each element is  $\theta(1)$ . Therefore, the insertion time complexity is  $\theta(n)$ .

And at the worst case, for the  $N$ th element, after the insertion, we need to heapify/swap  $\log(N)$  times to satisfy the minheap rules. And for each layer of the minheap, the sum of swap at the worst case is:

$$1 \cdot h_0 + 2 \cdot h_1 + 4 \cdot h_2 + \dots + \frac{n}{4} \cdot \log(n/4) + \frac{n}{2} \cdot \log(n/2) = O(n \log n)$$

Therefore, the time complexity is  $\theta(N) + O(N \log N) = O(N \log N)$ .

Space complexity is  $\theta(n)$ .

## Question 5

Delete maxheap

array = [25, 13, 20, 8, 7, 2, 17, 5, 4]

1. For the maxheap, when we delete max until the maxheap is empty, we always do the below steps:
  - (1) swap the root and the last leaf node, which is array[0] and array[end];
  - (2) Delete the array[end], which is the target max.
  - (3) Sippy down the root node, until it satisfies the maxheap.



delete 25

array = [25, 13, 20, 8, 7, 2, 17, 5, 4]

array = [4, 13, 20, 8, 7, 2, 17, 5]

array = [20, 13, 4, 8, 7, 2, 17, 5]

array = [20, 13, 17, 8, 7, 2, 4, 5]

delete 20

array = [20, 13, 17, 8, 7, 2, 4, 5]

array = [5, 13, 17, 8, 7, 2, 4]

array = [17, 13, 5, 8, 7, 2, 4]

2. When root is 25, we swap 25 and 4; Delete 25 from the minheap; swap 4 with its child node down until it follows the minheap again.

3. When root is 20, we swap 20 and 5; Delete 20 from the minheap; swap 5 with its child node down until it follows the minheap again.

delete  
17

array = [17, 13, 5, 8, 7, 2, 4]  
array = [4, 13, 5, 8, 7, 2]  
array = [13, 4, 5, 8, 7, 2]  
array = [13, 8, 5, 4, 7, 2]

4. When root is 17, we swap 17 and 4; Delete 17 from the minheap; swap 4 with its child node down until it follows the minheap again.

delete  
13

array = [13, 8, 5, 4, 7, 2]  
array = [2, 8, 5, 4, 7]  
array = [8, 2, 5, 4, 7]  
array = [8, 7, 5, 4, 2]

5. When root is 13, we swap 13 and 2; Delete 13 from the minheap; swap 2 with its child node down until it follows the minheap again.

delete 8

- array = [8, 7, 5, 4, 2]
- array = [2, 7, 5, 4]
- array = [7, 2, 5, 4]
- array = [7, 4, 5, 2]

delete 7

- array = [7, 4, 5, 2]
- array = [2, 4, 5]
- array = [5, 4, 2]

delete 5

- array = [5, 4, 2]
- array = [2, 4]
- array = [4, 2]

delete 4

- array = [2]

delete 2

- array = []

6. The rest follows the same pattern.

Complexity:

To delete  $N$  elements from the maxheap, since each element is deleted from the end of the array, the time is  $O(1)$  for each one element, the total time is  $\theta(n)$ ;

However, after each delete, we need to sippy down the root element to maintain the maxheap, at the worst case, each element might need  $\log(n)$  swaps. Therefore, the time complexity is  $O(n\log(n))$ .

The total time complexity is:  
 $\theta(n) + O(n\log(n)) = n\log n$