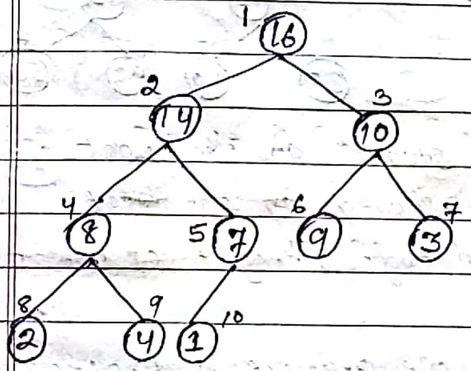
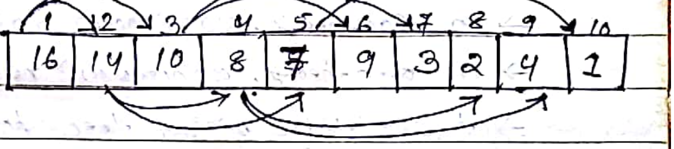
HEAP SORT

Binary Heap: The (binary) heap data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest.

complete binary tree: A binary tree is said to be complete if all the levels are completelly filled except the last but one level. And the last but one level is filled from left to right.



The array corresponding to the heap is:



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

→ If an array A contains key values of nodes on a heap, Length[A] is the total number of elements.

heap-size[A] = Length[A] = Number of elements.

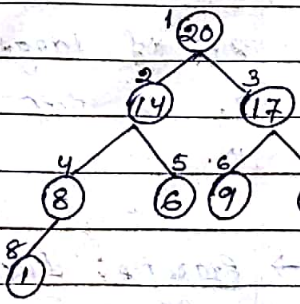
→ The root of the tree is A[1]

→ for any node i,

1. PARENT(i)
   return ⌊i/2⌋
2. LEFT(i)
   return 2i
3. RIGHT(i)
   return 2i+1

→ Example:

1	2	3	4	5	6	7	8
20	14	17	8	6	9	4	1



Root node is 20 having index 1.

Left child = 2*1 = 2 , Value = 14

Right child = 2*1+1 = 3 , Value = 17

PARENT(2) = 2/2 = 1

LEFT(2) = 2*2 = 4

RIGHT(2) = 2*2+1 = 5

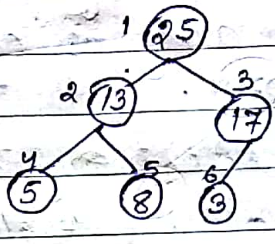
83

→ There are two kinds of heap
(1) Max-heaps (2) Min-heaps

→ A complete binary tree is said to be heap if it satisfies the heap properties.

Heap property

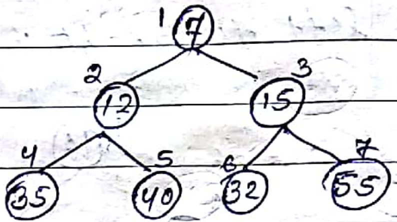
(1) Max-heap: In a max-heap, for every node i other than the root, the value of a node is ~~greater~~ less than or equal to the value of its parents.



$A[\text{PARENT}(i)] \geq A[i]$

(2) Min-heap: For every node i, other than root node,

$A[\text{PARENT}(i)] \leq A[i]$



→ In max-heap, maximum value is stored on the root.

→ In min-heap, minimum value is stored on the root.

→ A complete binary tree having n nodes, height of the tree is $\log_2 n$.

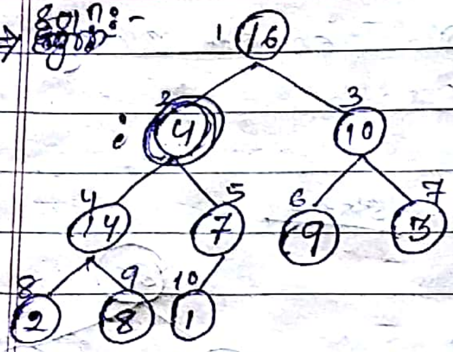
→ To maintain the heap property, we require a procedure called MAX-HEAPIFY(A, i), which is called for a particular node to make that node satisfy the heap property.

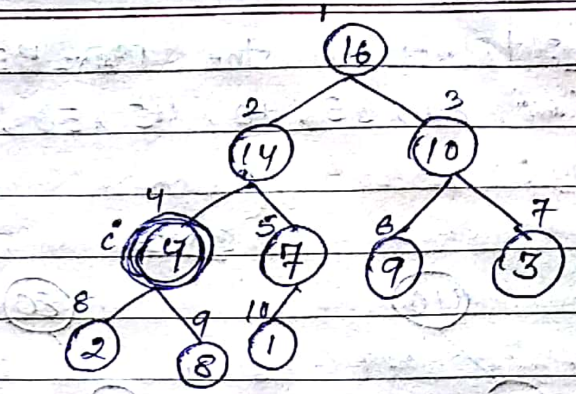
```
MAX-HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
      then largest ← l
      else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
      then largest ← r
if largest ≠ i
      then exchange A[i] ↔ A[largest]
            MAX-HEAPIFY(A, largest)
```

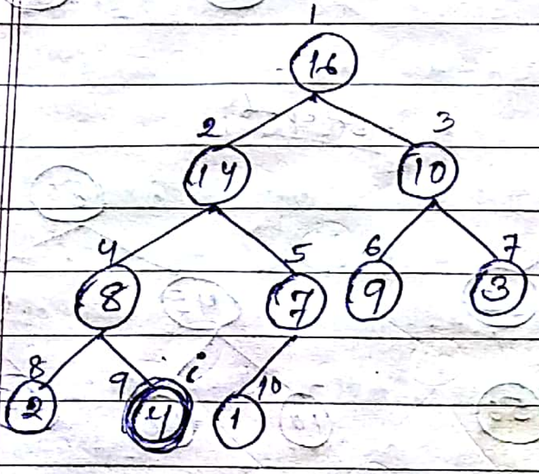
→ Example: Illustrate the operation of MAX-HEAPIFY(A, 2) on the array

A = {16, 4, 10, 14, 7, 9, 3, 2, 8, 1}

⇒ Soln:-



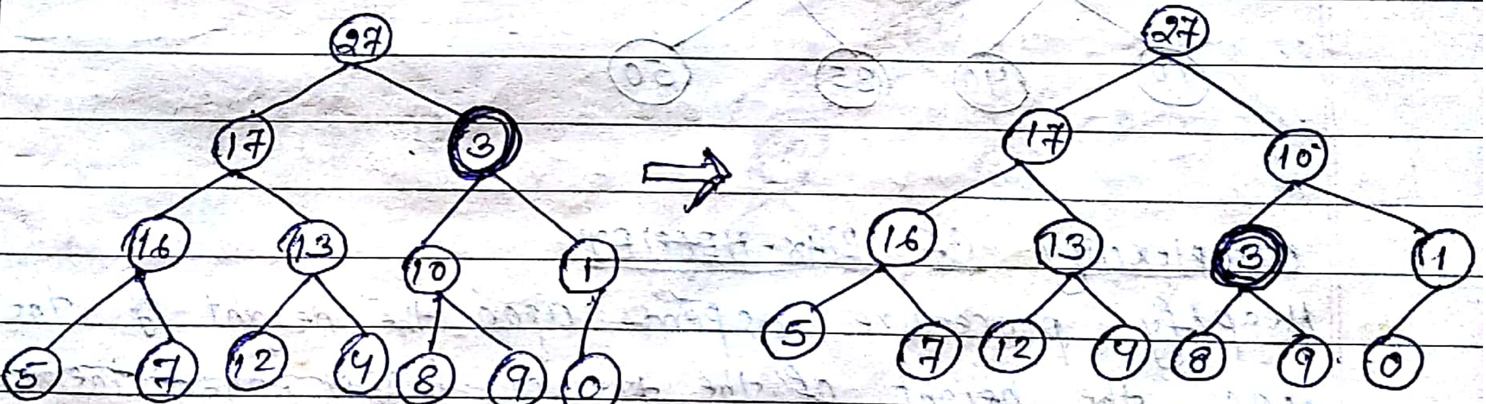


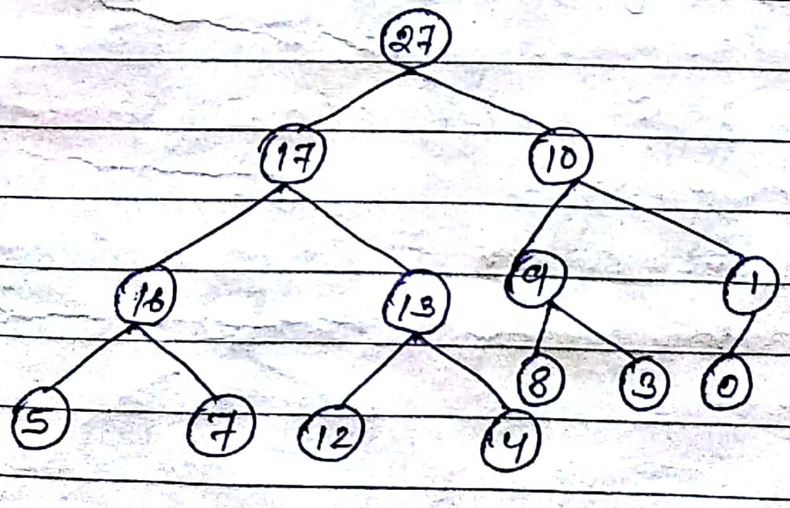


⇒ Example: Illustrate the operation of MAX-HEAPIFY (A, 3) on t

array A = < 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 >

Solution: The following happens when MAX-HEAPIFY (A, 3) is called.

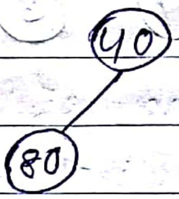




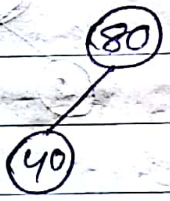
→ **Example:** Illustrate the MAX-HEAPIFY on the array
A = < 40, 80, 35, 10, 45, 50, 70 >

Solution:

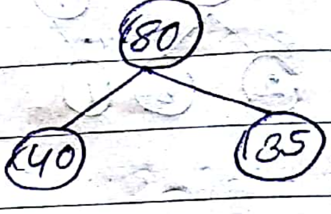
step-1



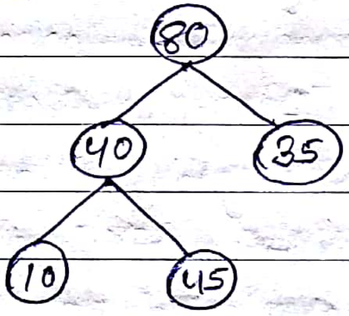
step-2



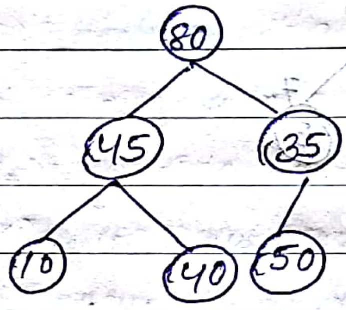
step-3



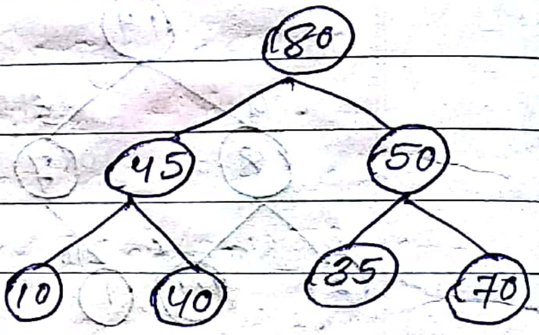
step-4



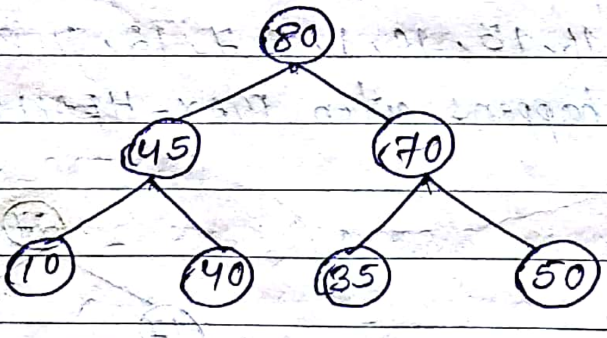
step-5



step-6



step-7



Complexity of MAX-HEAPIFY

Heapify procedure depends upon the height of the tree. Since the height of the tree is $\log_2 n$, then the heapify procedure takes ~~Θ~~ $O(\log n)$

Building A Heap

→ The MAX-HEAPIFY procedure is called from down to top manner to make the entire tree into heap.

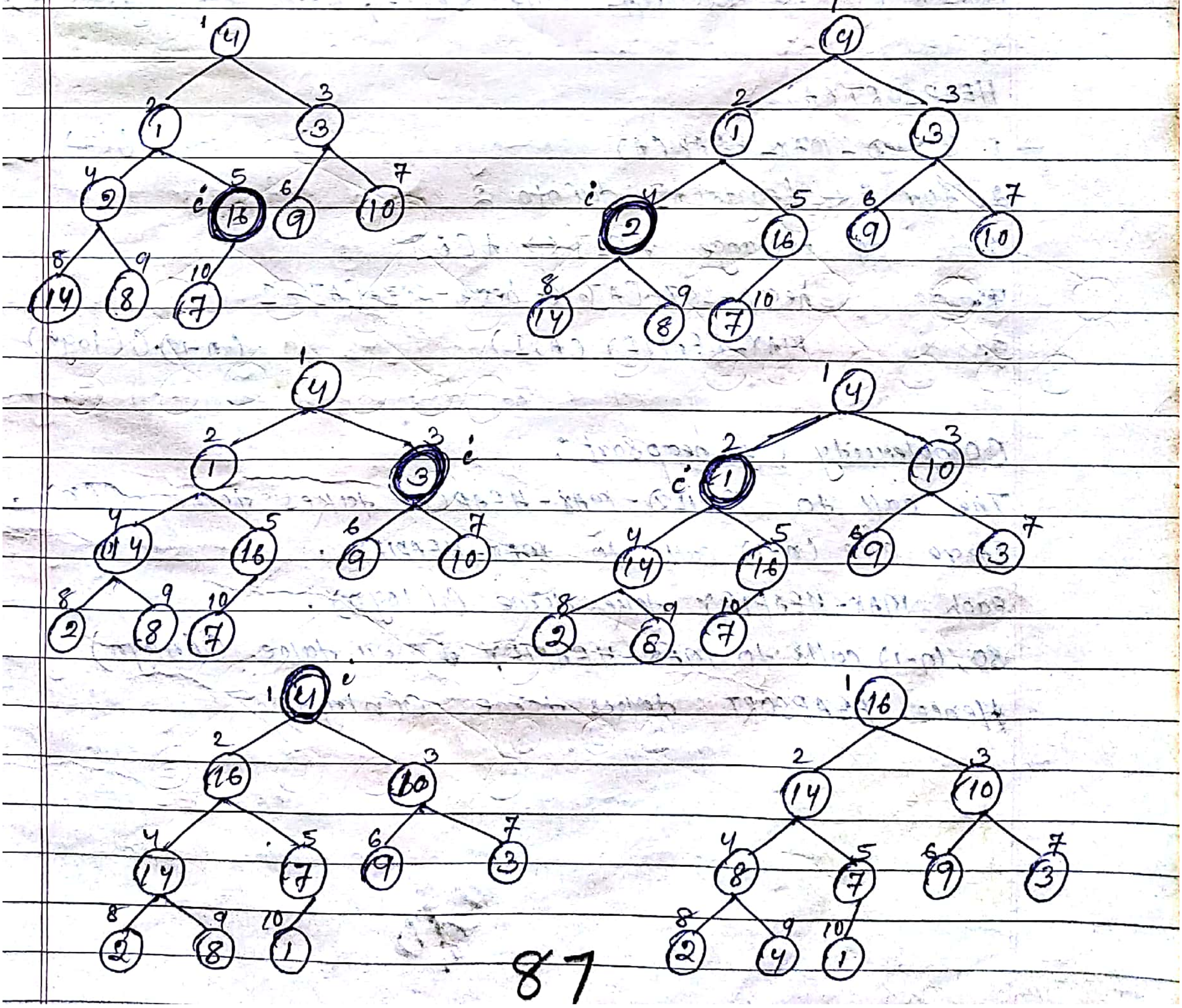
→ In a complete binary tree there is n nodes must have $\lceil \frac{n}{2} \rceil$ number of leaf nodes.

→ Hence, to make entire tree into heap, we call the heapify procedure starting from node no. n/2 down to 1. Because leaf nodes indicates that it is already in a heap.

```
BUILD-MAX-HEAP(A)
1.  heap-size[A] ← length[A]
2.  for i ← ⌊length[A]/2⌋ downto 1
3.        do MAX-HEAPIFY(A, i)
```

→ **Example:**

4	1	3	2	16	9	10	14	8	7



1 4, 2 1, 3 3, 4 2, 5 16 i, 6 9, 7 10, 8 14, 9 8, 10 7
1 4, 2 1, 3 3, 4 2 i, 5 16, 6 9, 7 10, 8 14, 9 8, 10 7
1 4, 2 1, 3 3 i, 4 14, 5 16, 6 9, 7 10, 8 2, 9 8, 10 7
1 4, 2 1 i, 3 10, 4 14, 5 16, 6 9, 7 3, 8 2, 9 8, 10 7
1 4 i, 2 16, 3 10, 4 14, 5 7, 6 9, 7 3, 8 2, 9 8, 10 1
1 16, 2 14, 3 10, 4 8, 5 7, 6 9, 7 3, 8 2, 9 4, 10 1

→ Complexity Of Build heap = O(nlogn)

→ Each call to MAX-HEAPIFY costs O(logn) time and to BUILD-MAX-HEAP, there are O(n) MAX-HEAPIFY calls. Thus the running time is O(nlogn)

THE HEAP SORT ALGORITHM :

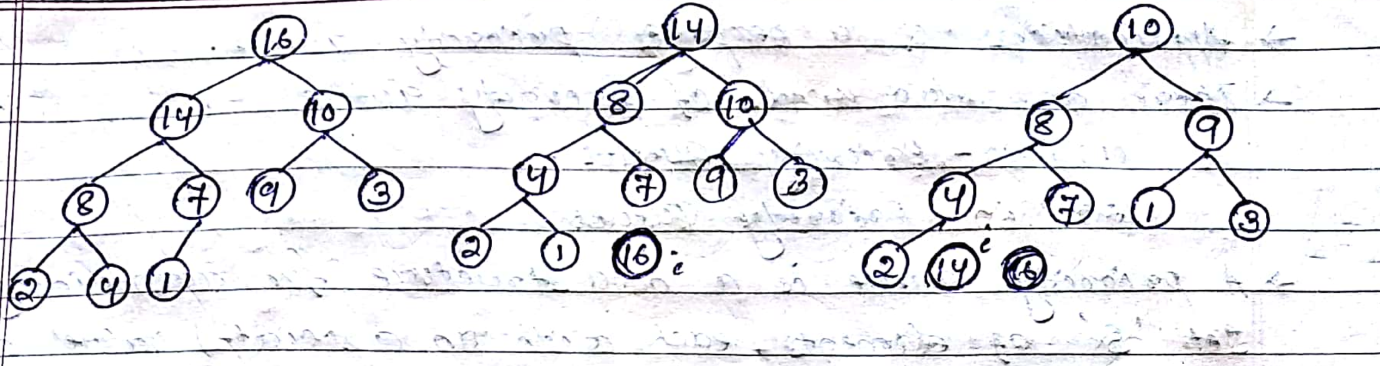
The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array A[1..n]. where n = length[A]

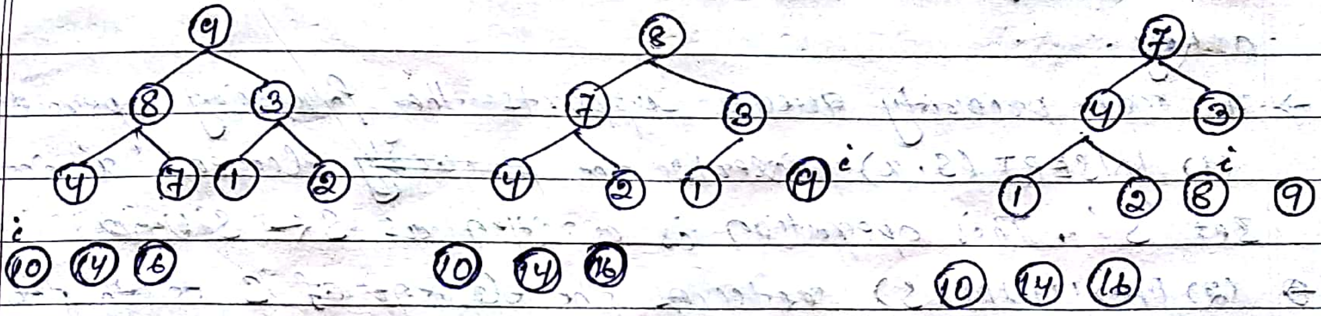
Since the maximum element of the array is stored at the root A[1], it will exchange with A[n] and we reduce the heap size by one (i.e. node n is discard from the heap). Now heap becomes A[1..(n-1)] can be converted to max heap by calling MAX-HEAPIFY(A,1) procedure at the first node. This process is continue until all the elements are sorted (i.e. n-1 down to 2)

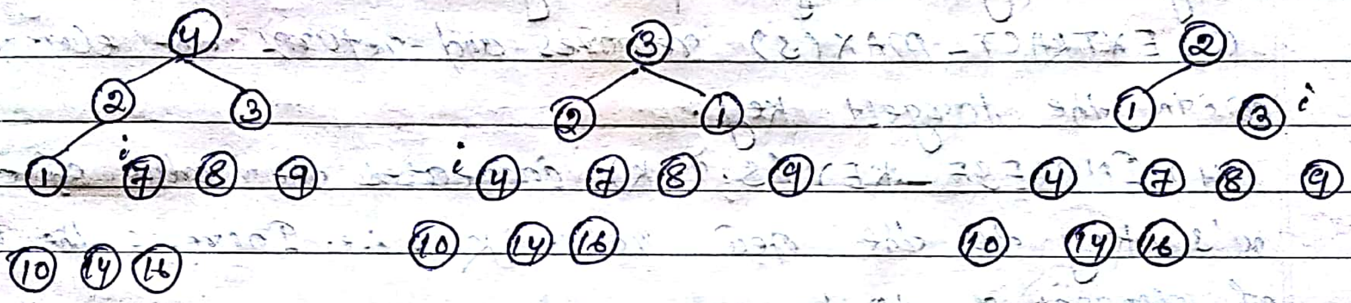
```
HEAPSORT (A)
BUILD-MAX-HEAP (A)                                  O(n)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
       heap-size[A] ← heap-size[A] - 1
       MAX-HEAPIFY (A,1)                            (n-1) O(logn)
```

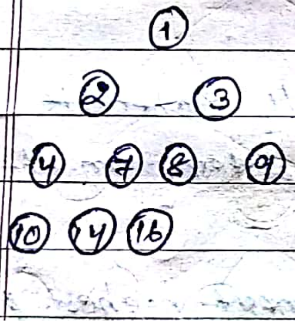
Complexity Of heapSort :

The call to BUILD-MAX-HEAP(A) takes time O(n).
There are (n-1) calls to MAX-HEAPIFY.
Each MAX-HEAPIFY takes time O(logn).
So, (n-1) calls to MAX-HEAPIFY will take O(nlogn)
Hence, HEAPSORT takes time O(nlogn)









A	1	2	3	4	7	8	9	10	14	16