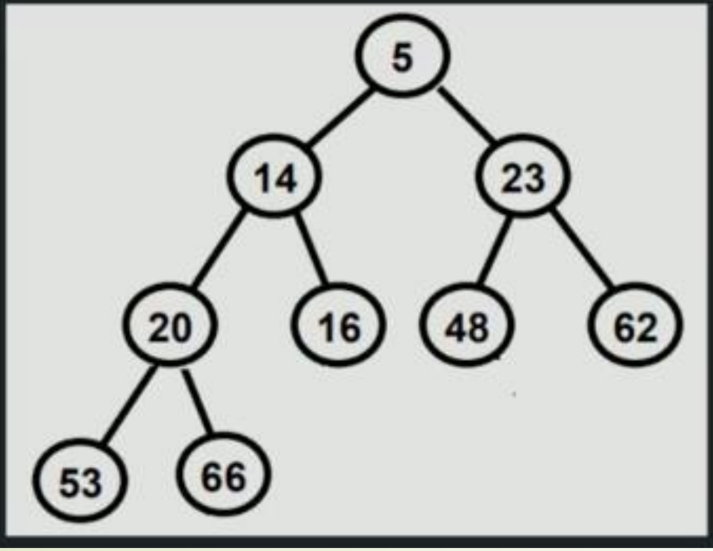
Data Structure and Algorithms

Session-33

Dr. Subhra Rani Patra
SCOPE, VIT Chennai

What is Binary Heap:

- ✓ **Definition:** Binary Heap is a Binary Tree with some special properties.
  - ✓ Heap property –
    - ✓ Value of any given node must be <= value of its children(Min-Heap)
    - ✓ Value of any given node must be >= value of its children(Max-Heap)
  - ✓ Complete Tree –
    - ✓ All levels are completely filled except possibly the last level and the last level has all keys as left as possible.
    - ✓ This makes Binary Heap ideal candidate for Array Implementation.



Why should we learn Binary Heap ?

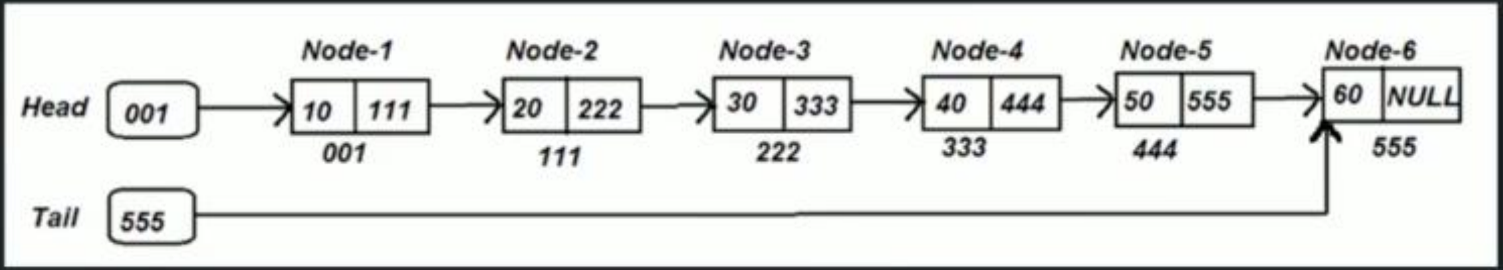
There are cases when we want to find 'min/max' number among set of numbers in log(n) time. Also, we want to make sure that Inserting additional numbers does not take more than O(log n) time.

Possible Solutions:

1. *Store the numbers in sorted array*
   - ✓ *Issue here is that once we insert/delete a new number, our array needs to be adjusted again to keep it sorted which will take O(n) time.*

10	20	30	40	50		

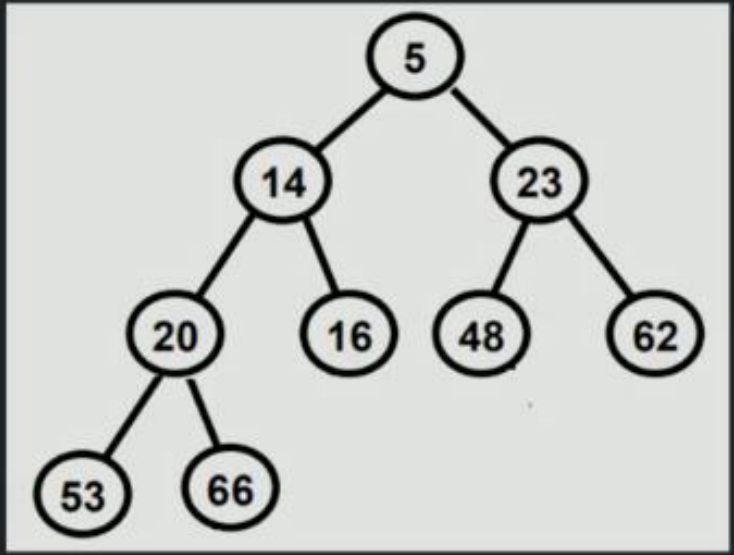
2. *Store the numbers in Linked list in sorted manner*

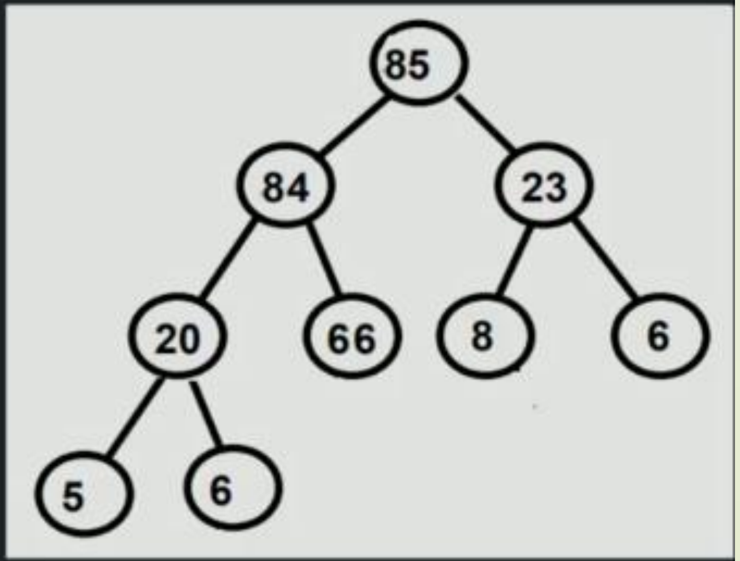


Types of Binary Heap:

✓**Min-Heap:** *If the value of each node is less than or equal to value of both of its children.*

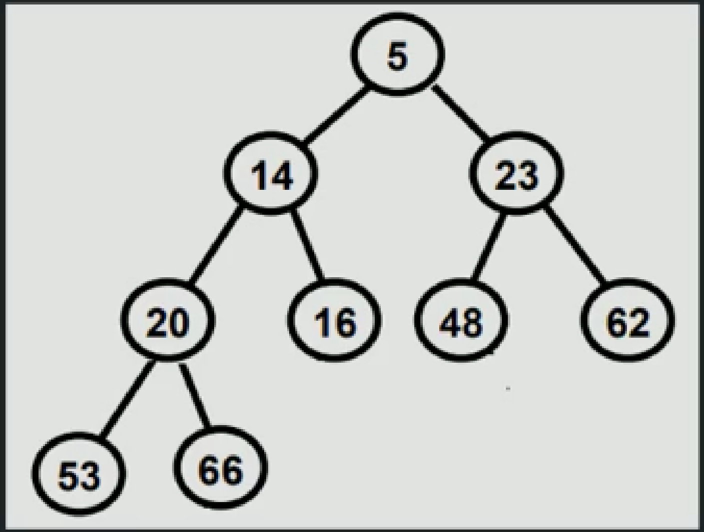
✓**Max-Heap:** *If the value of each node is more than or equal to value of both of its children.*





Common operations:

- ✓createHeap – *creates a blank Array to be used for storing Heap*
- ✓peekTopOfHeap – *returns min/max from Heap*
- ✓extractMin / extractMax – *extracts Min/Max from Heap. We can extract **only** this node.*
- ✓sizeOfHeap – *returns the size of the Heap*
- ✓insertValueInHeap – *Inserts value in Heap*
- ✓deleteHeap – *Deletes the entire Heap*

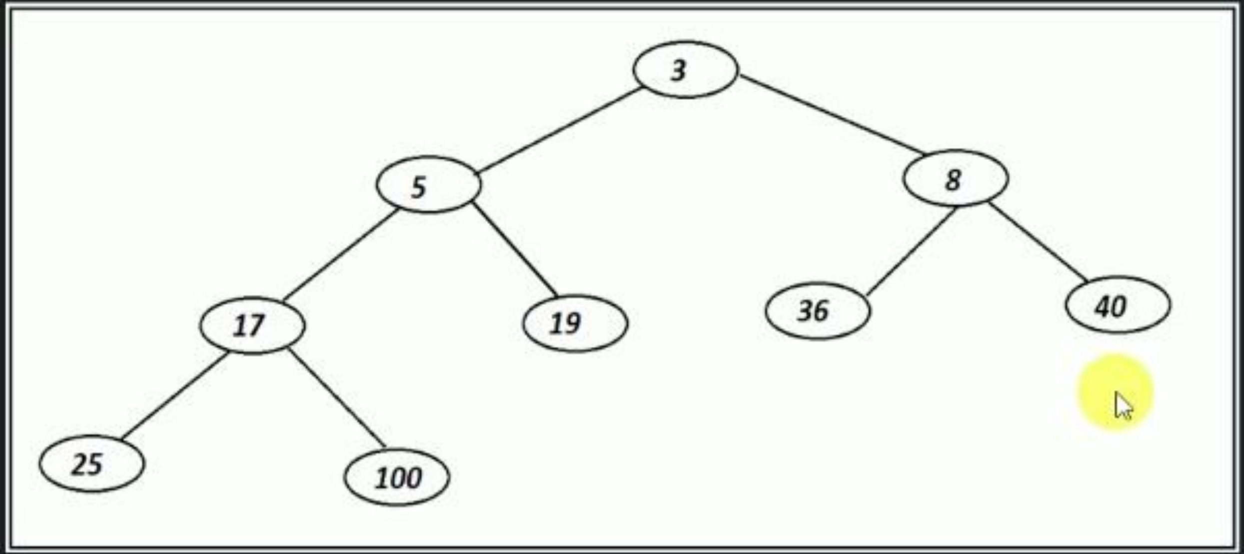


Implementation options:

- ✓ *Array based Implementation:*
- ✓ *Reference/Pointer based Implementation:*

Binary Heap - Array Representation:

✓How does Binary Heap looks like at logical level ?



✓How does Binary Heap looks when implemented via Array:

Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value	✗	3	5	8	17	19	36	40	25	100								

Left Child – cell [2x]

Right Child – cell [2x + 1]

Creation of Heap:

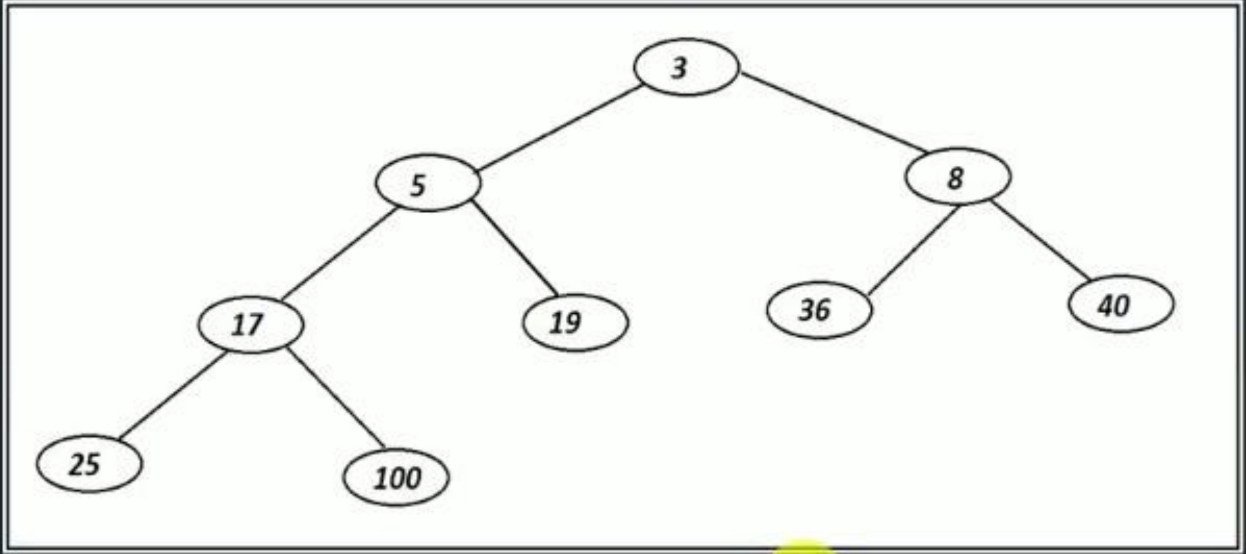
Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value	✕																	

createHeap(size)

create a blank array of 'size+1'

initialize sizeOfHeap with 0

Peek of Heap:



peekTopOfHeap ()

if tree does not exists

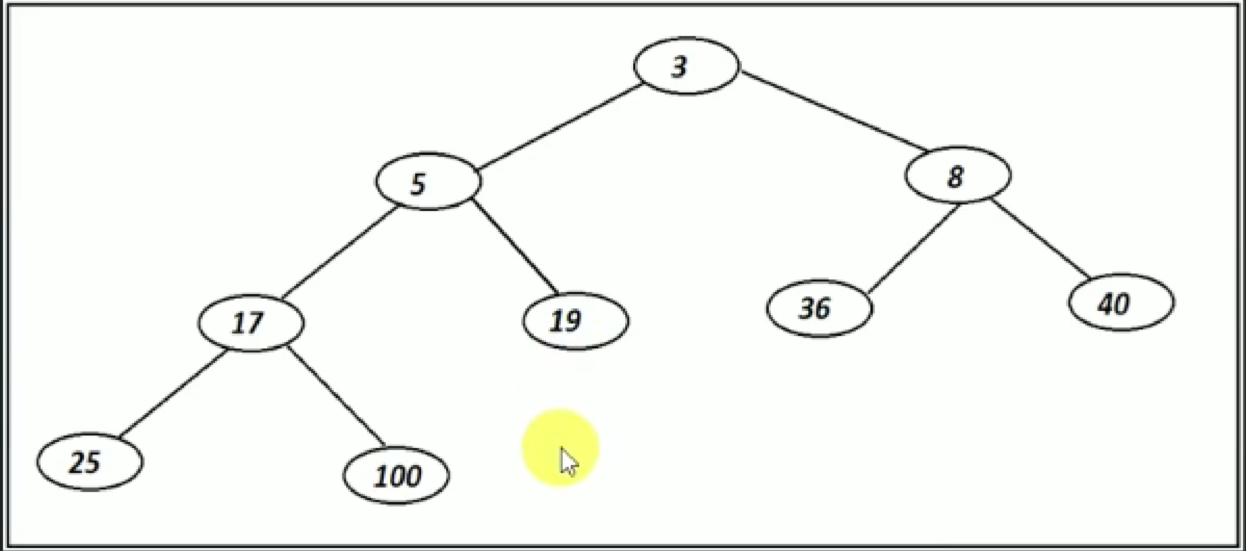
return error message

else

return 1st cell of array

Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value	✖	3	5	8	17	19	36	40	25	100								

Size of Heap:

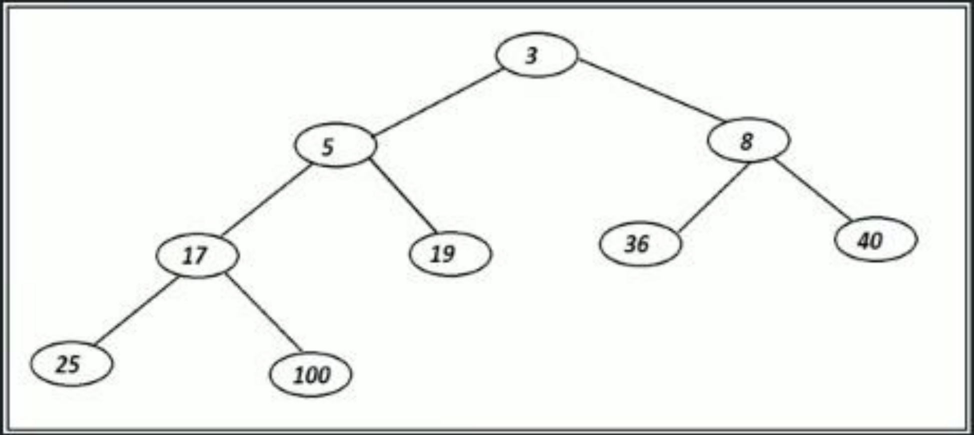


Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value	X	3	5	8	17	19	36	40	25	100								

sizeOfHeap()

return sizeOfHeap

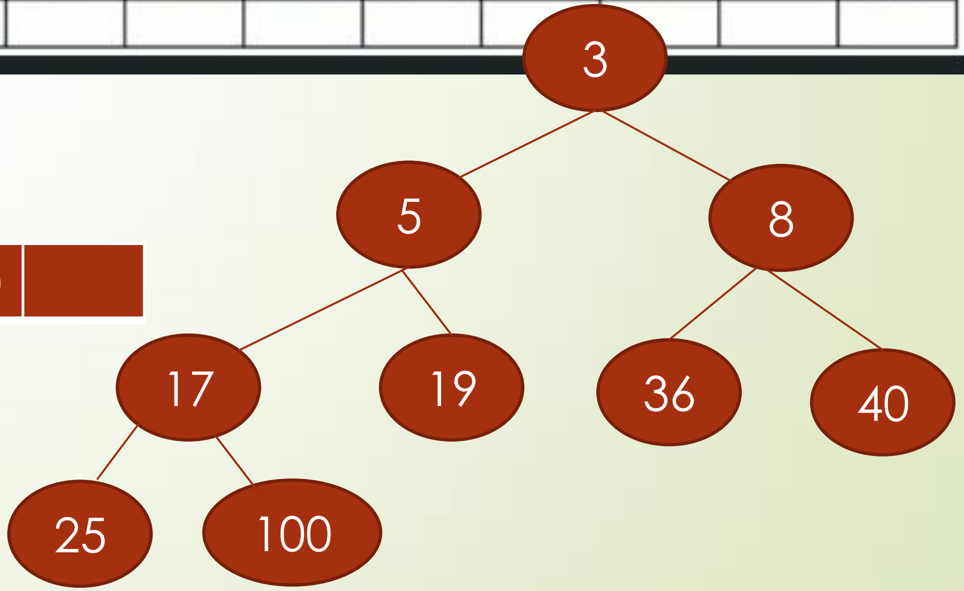
Insertion in Heap:




```
insertValueInHeap(value)
    if tree does not exists
        return error message
    else
        insert 'value' in first unused cell of array
        sizeOfHeap ++
        heapifyBottomToTop(sizeOfHeap)
```

Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value	X	3	5	8	17	19	36	40	25	100								

3	5	8	17	19	36	40	25	100	



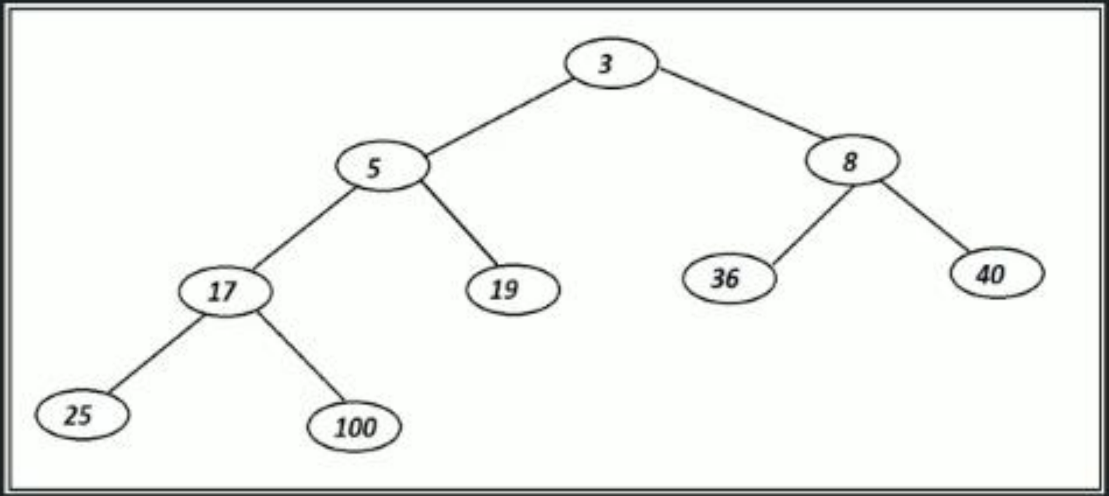
Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.

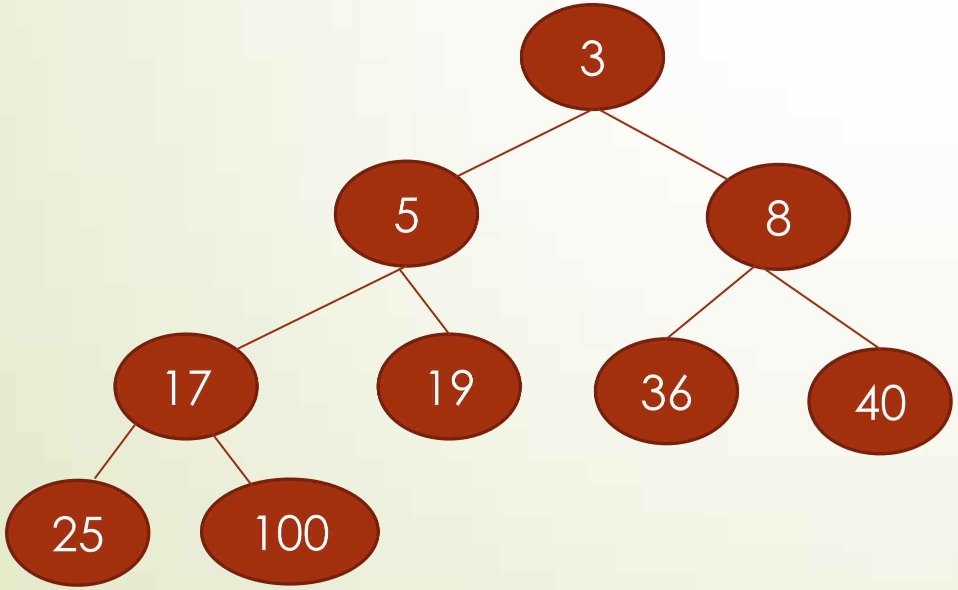
Time & Space Complexity – Insertion in Heap:

```
insertValueInHeap(value)
    if tree does not exists ------------------------------------ O(1)
        return error message ----------------------------------- O(1)
    else ------------------------------------------------------- O(1)
        insert 'value' in first unused cell of array ----------- O(1)
        sizeOfHeap ++ ------------------------------------------- O(1)
        heapifyBottomToTop(sizeOfHeap) -------------------------- O(log n)
```

Time Complexity – O(log n)

ExtractMin from Heap:



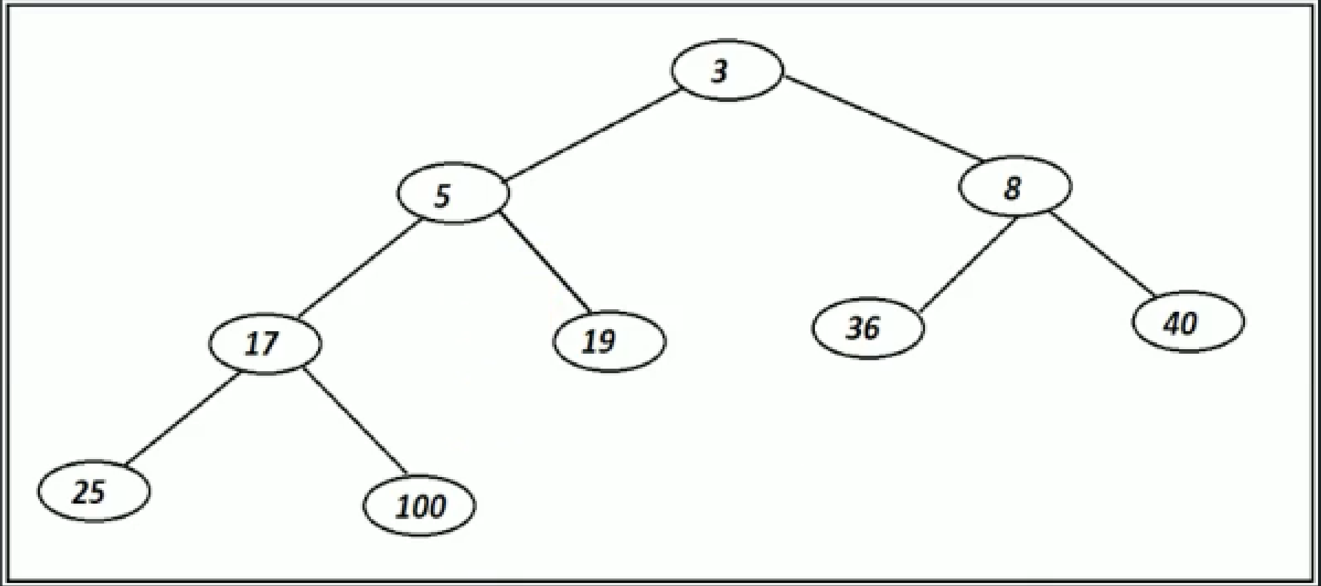



```
extractMin()
    if tree does not exists
        return error message
    else
        extract 1st cell of array
        promote last element to first
        sizeOfHeap --
        heapifyTopToBottom(1)
```

```
extractMin()

    if tree does not exists ------------------------------------------------ O(1)

        return error message ------------------------------------------- O(1)

    else ------------------------------------------------------------------- O(1)

        extract 1st cell of array ------------------------------------------- O(1)

        promote last element to first ---------------------------------- O(1)

        sizeOfHeap --    ------------------------------------------------------ O(1)

        heapifyTopToBottom(1) -------------------------------------------- O(log n)
```

<u>Time Complexity</u> – O(log n)

Delete Heap:



Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value	✖	3	5	8	17	19	36	40	25	100								

deleteHeap()

set array to null

Heap sort

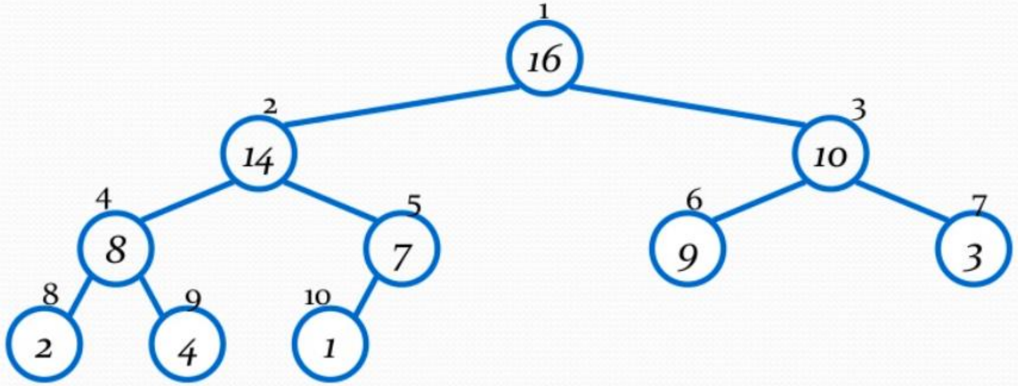
1. Build Max Heap from unordered array;
2. Find maximum element A[1];
3. Swap elements A[n] and A[1] :

   now max element is at the end of the array! .
4. Discard node n from heap

   (by decrementing heap-size variable).
5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.
6. Go to Step 2 unless heap is **empty**.

HeapSort() Example

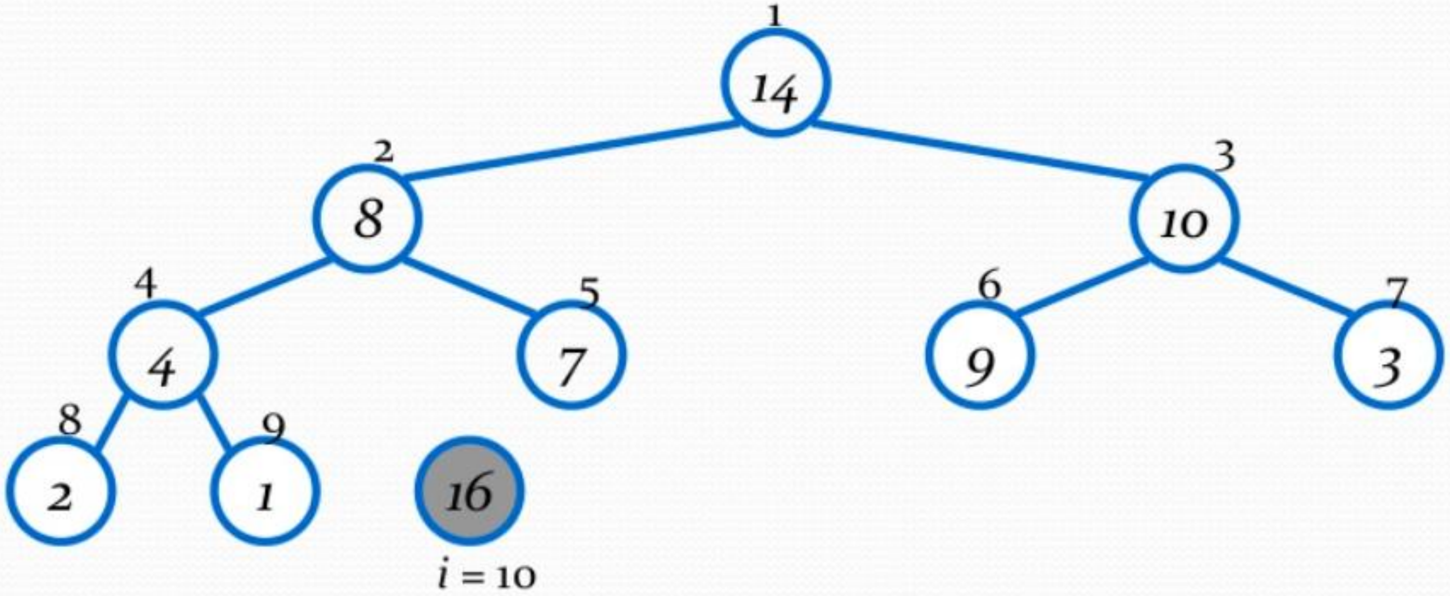
- A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}



16	14	10	8	7	9	3	2	4	1

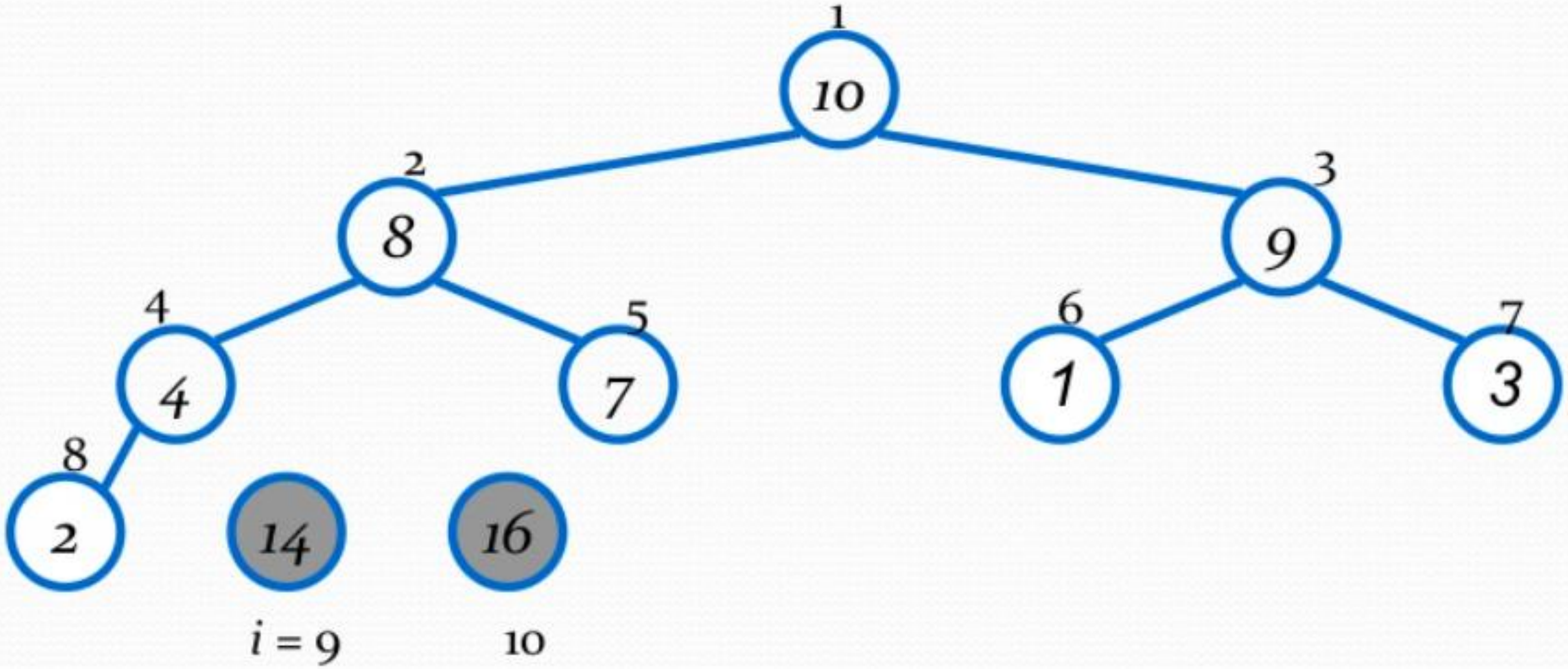
HeapSort() Example

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



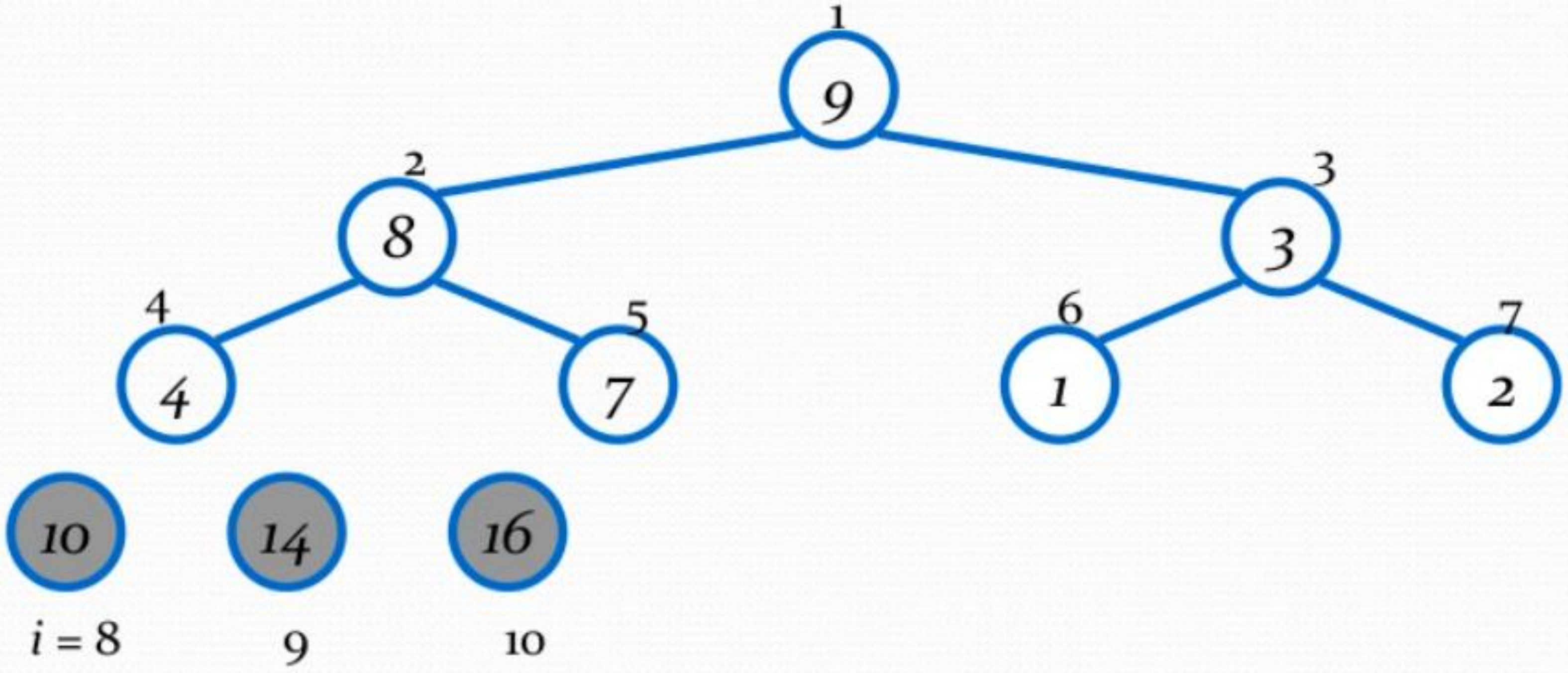
HeapSort() Example

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



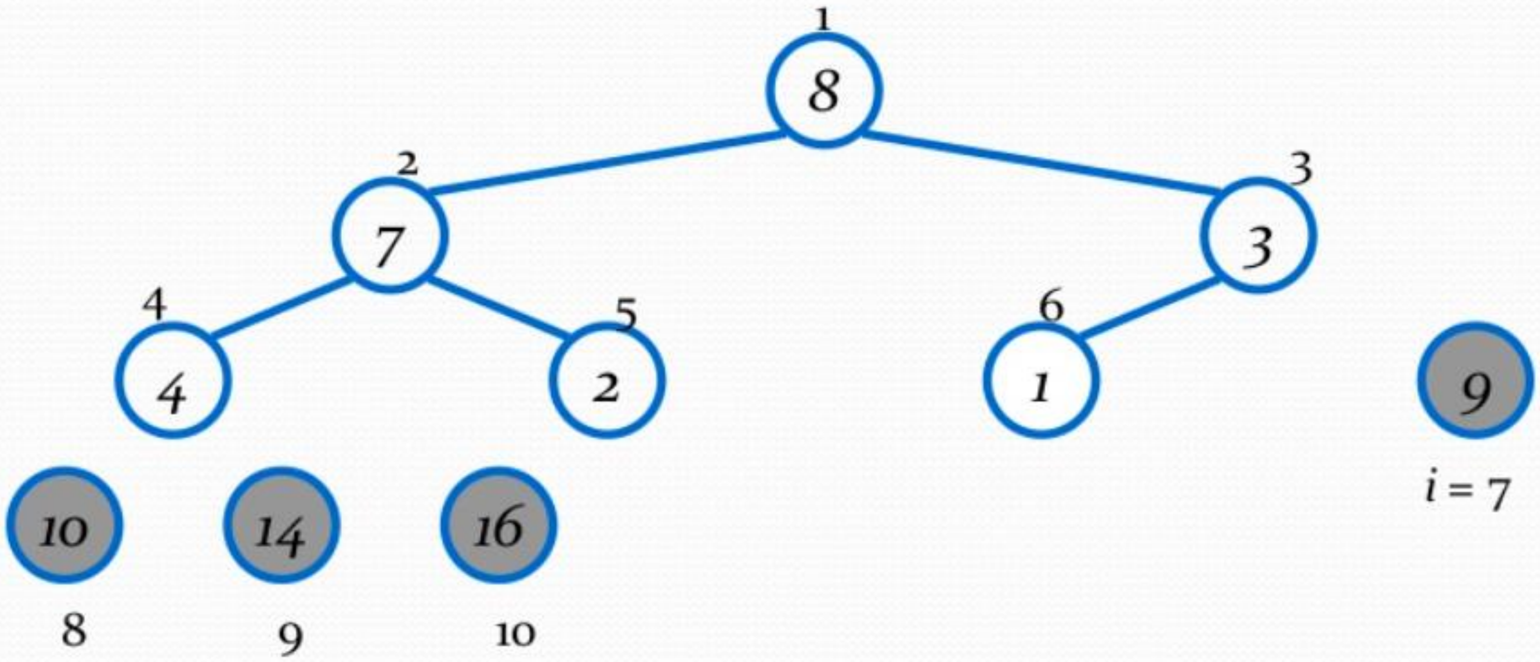
HeapSort() Example

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



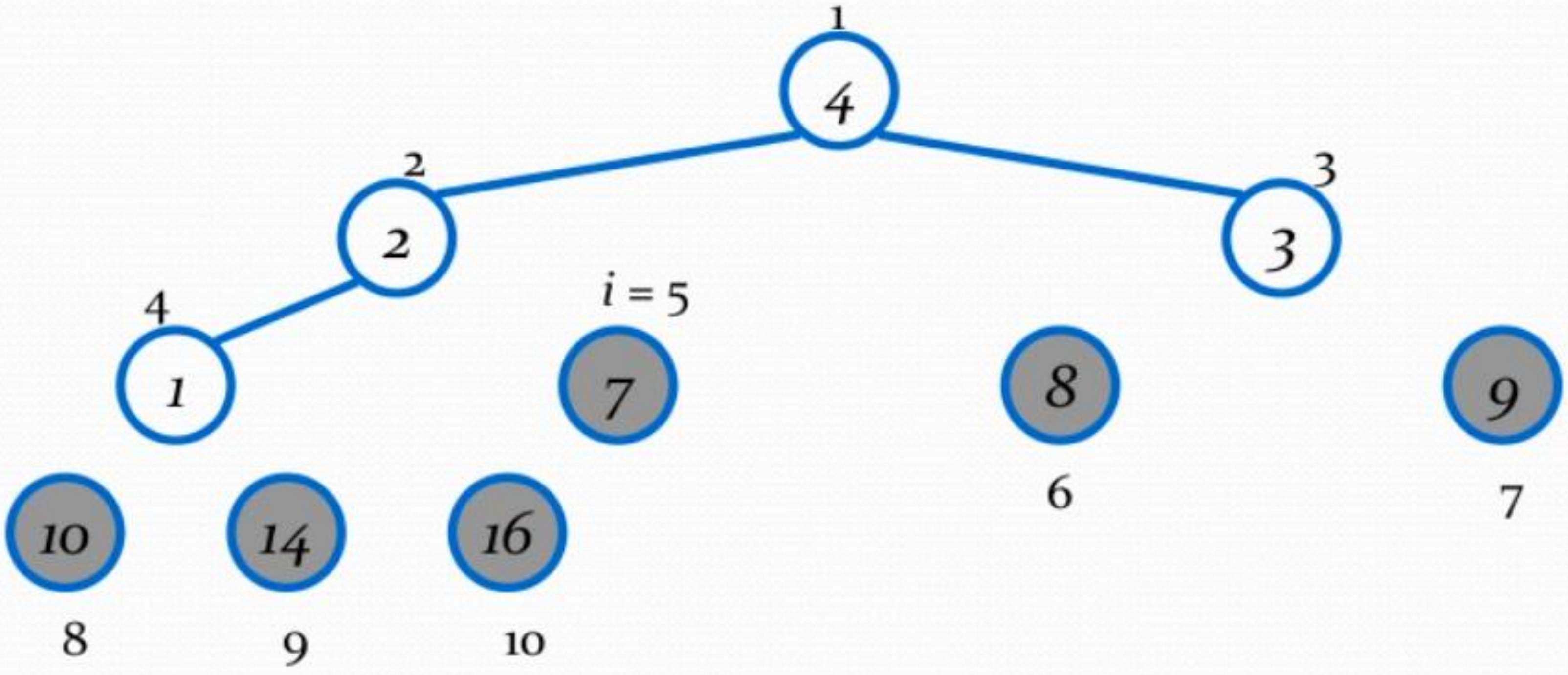
HeapSort() Example

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



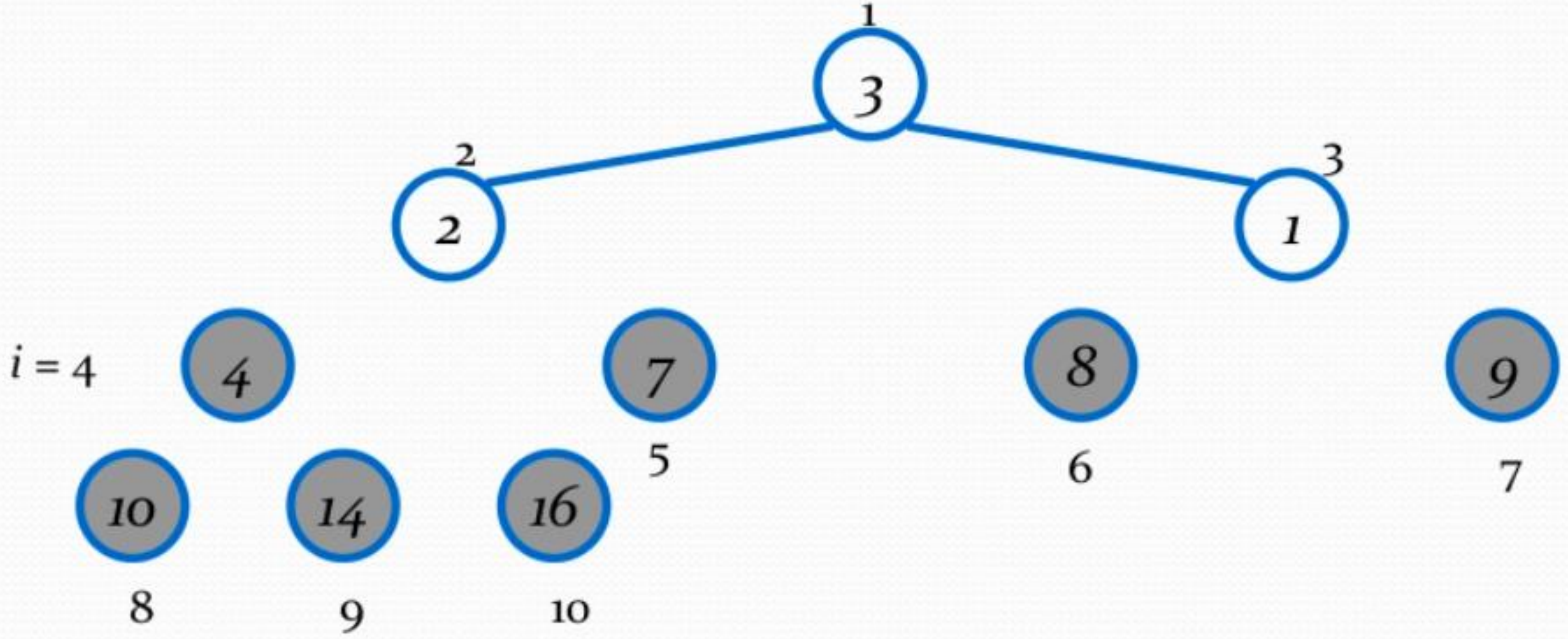
HeapSort() Example

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



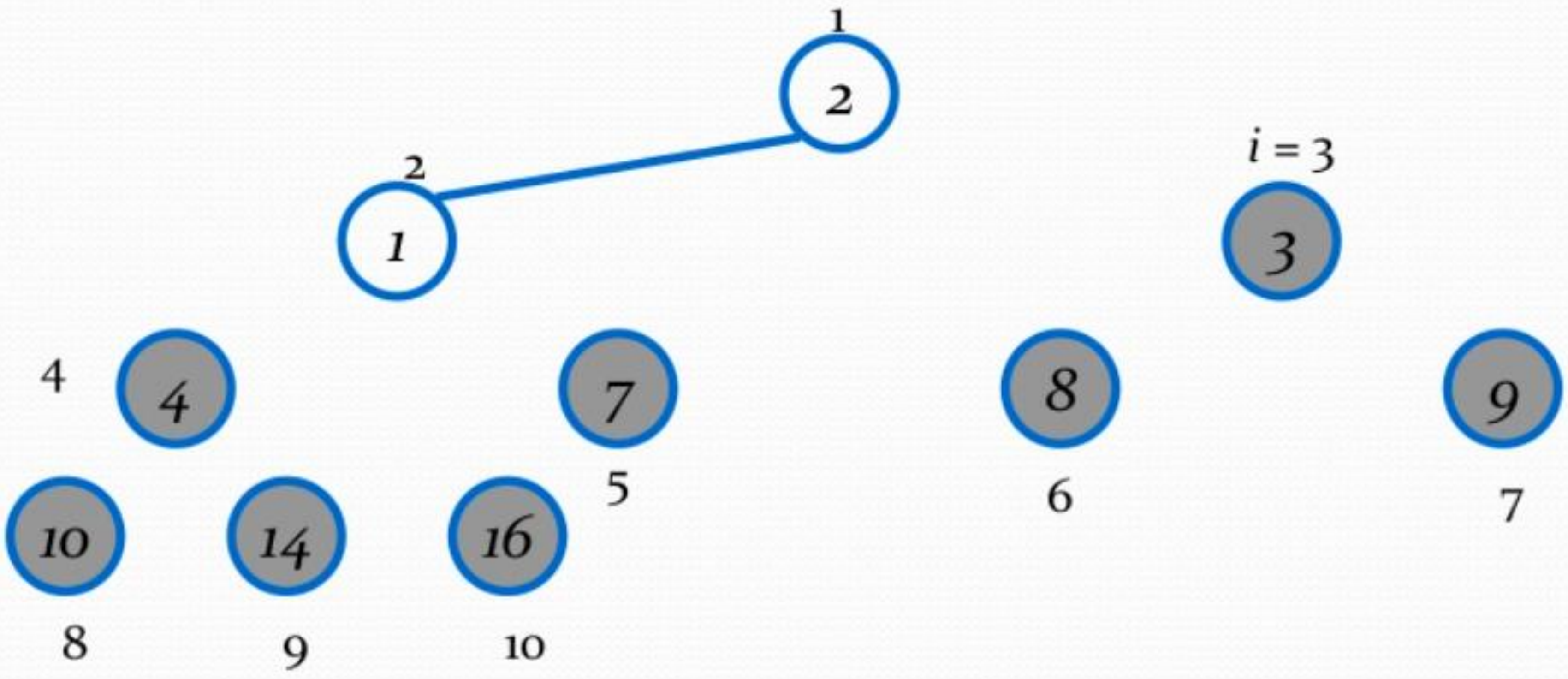
HeapSort() Example

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



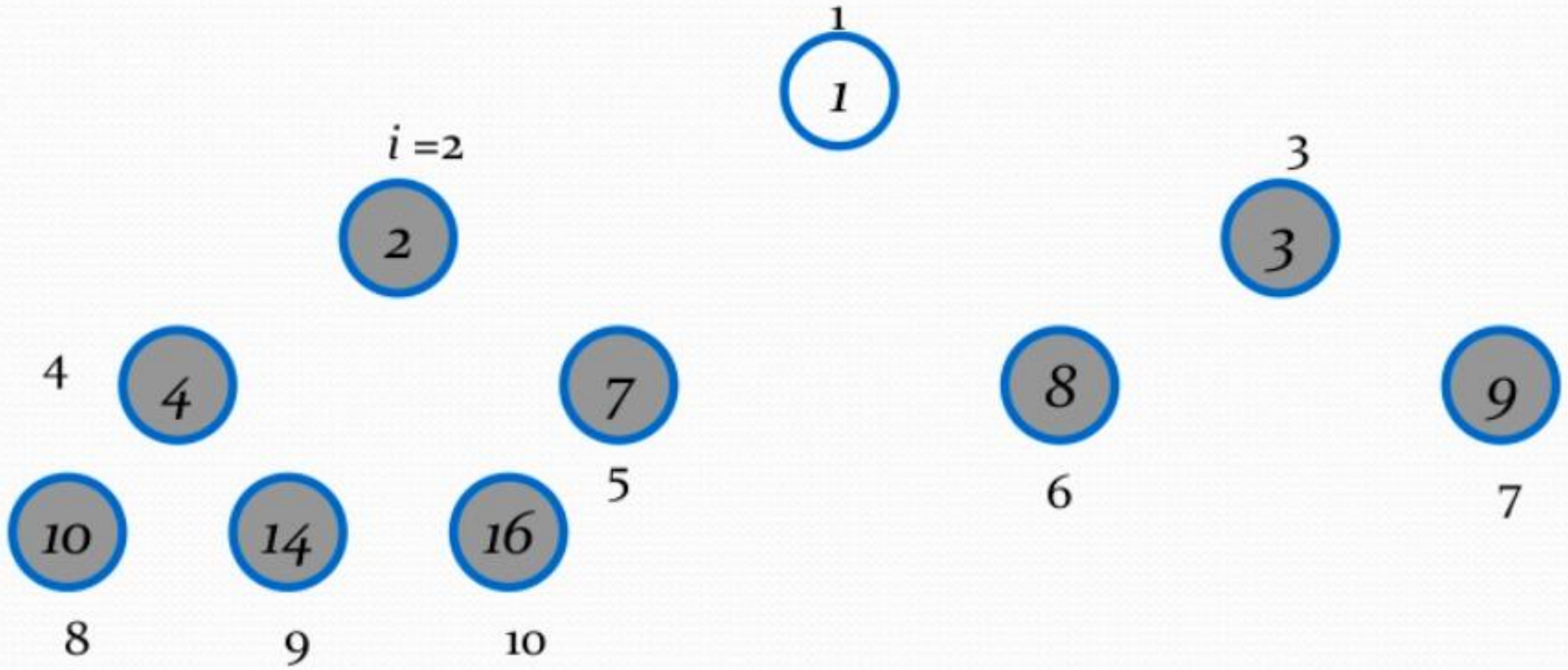
HeapSort() Example

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



HeapSort() Example

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




Thank you