HEAP AND HEAP SORT

GOALS

- To explore the implementation, testing and performance of heap sort algorithm

HEAP

- A heap is a data structure that stores a collection of objects (with keys), and has the following properties:
  - Complete Binary tree
  - Heap Order

- It is implemented as an array where each node in the tree corresponds to an element of the array.

HEAP ORDER PROPERTY

- For every node *v*, other than the root, the key stored in *v* is greater or equal (smaller or equal for max heap) than the key stored in the parent of *v*.

- In this case the maximum value is stored in the root

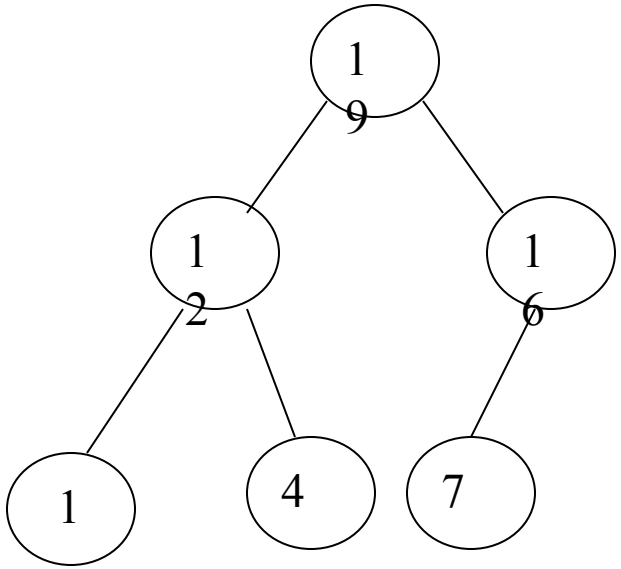
DEFINITION

- Max Heap
  - Store data in ascending order
  - Has property of

    $A[Parent(i)] \geq A[i]$
- Min Heap
  - Store data in descending order
  - Has property of

    $A[Parent(i)] \leq A[i]$

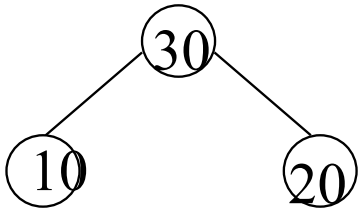
MAX HEAP EXAMPLE

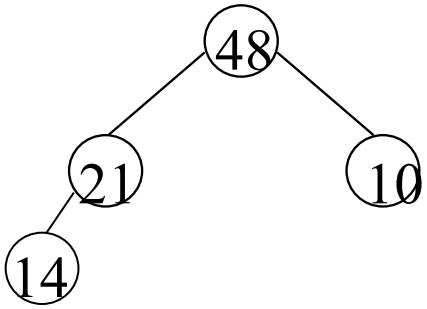


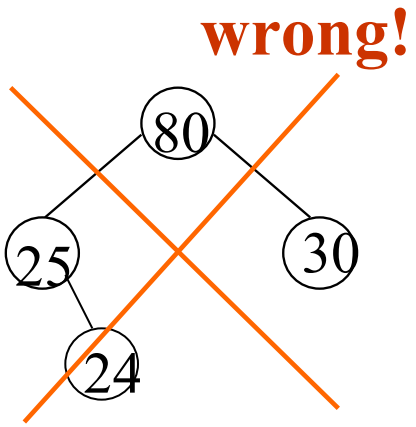
19	12	16	1	4	7

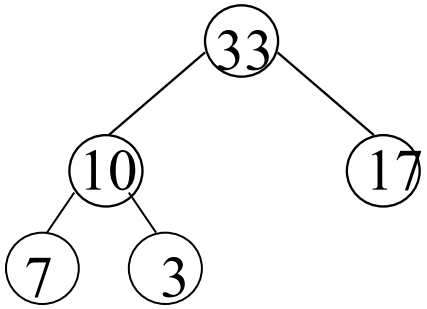
Array A

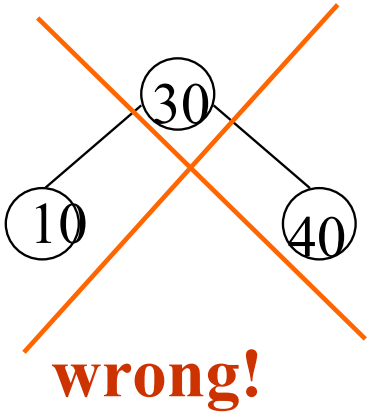
WHICH ARE MAX-HEAPS?

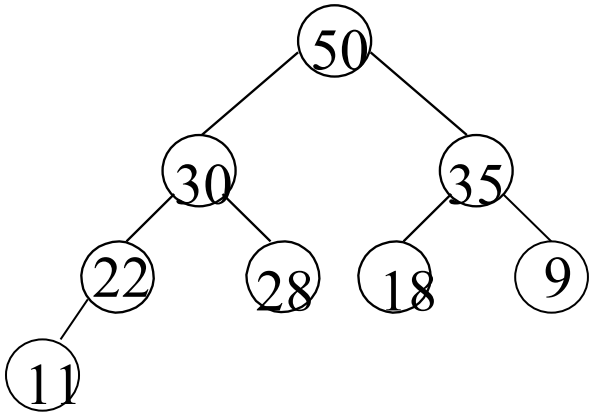




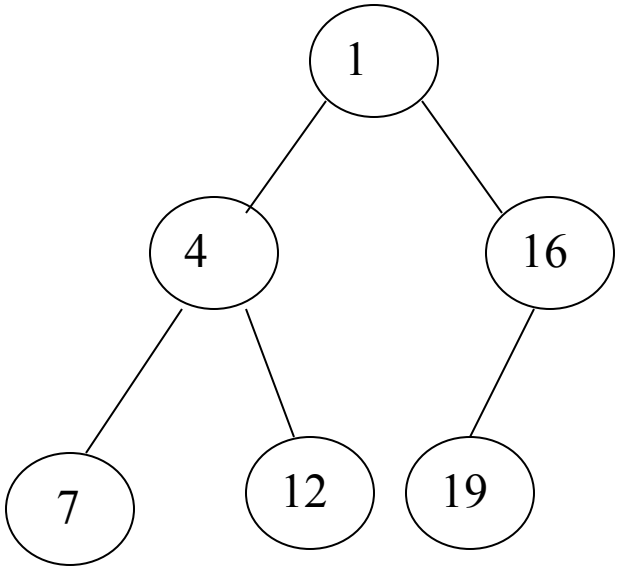








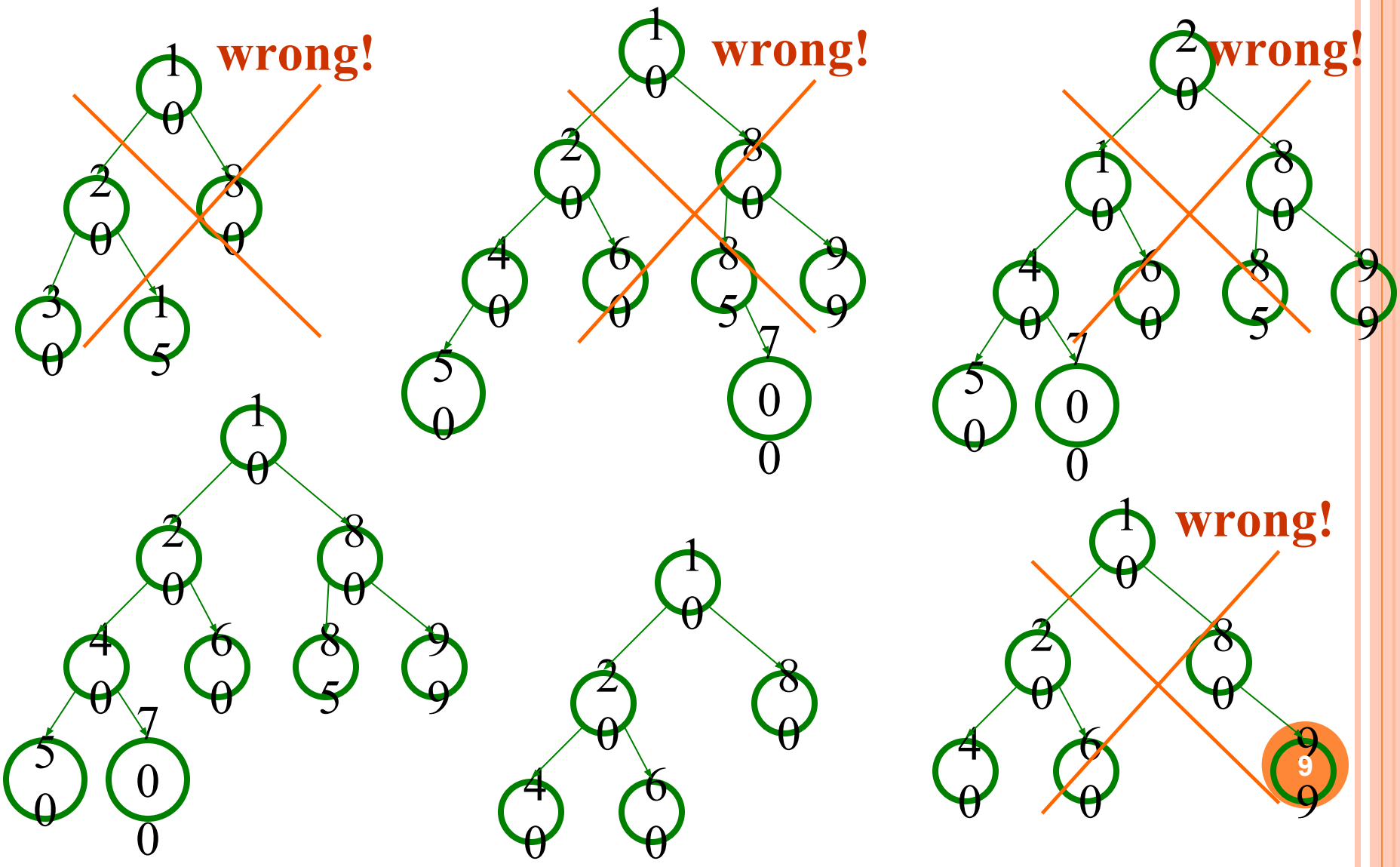
MIN HEAP EXAMPLE



1	4	16	7	12	19

Array A

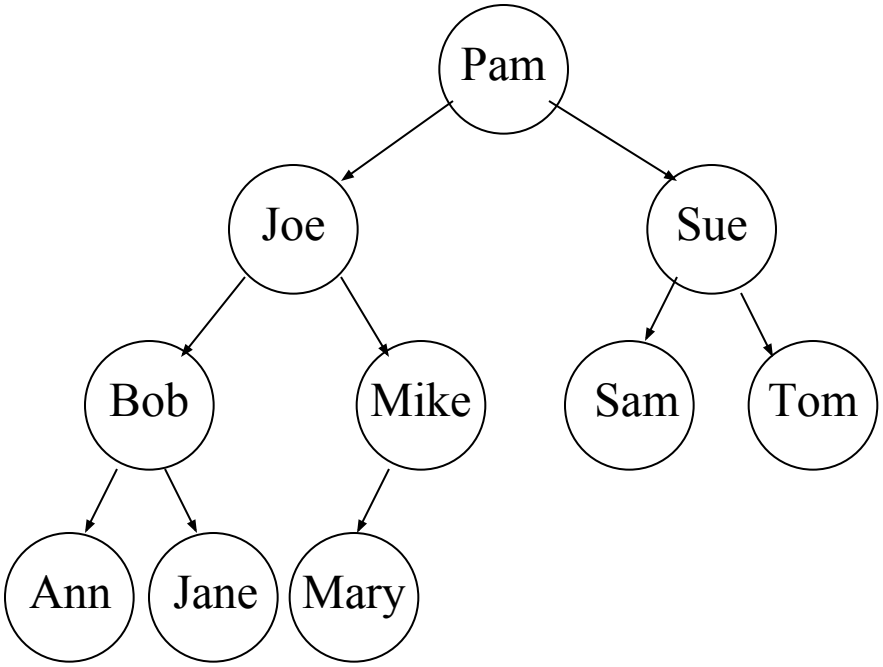
WHICH ARE MIN-HEAPS?



ARRAY TREE IMPLEMENTATION

- corollary: a complete binary tree can be implemented using an array (the example tree shown is *not* a heap)

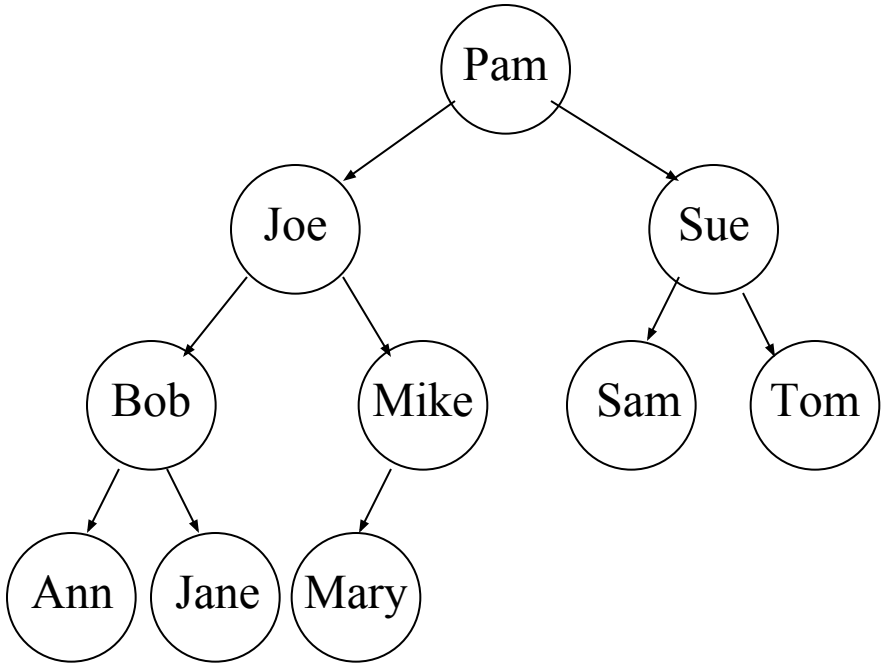
i	Item	Left Child	2*i	Right Child
1	Pam	2	2	3
2	Joe	4	4	5
3	Sue	6	6	7
4	Bob	8	8	9
5	Mike	10	10	-1
6	Sam	-1	11	-1
7	Tom	-1	13	-1
8	Ann	-1	15	-1
9	Jane	-1	17	-1
10	Mary	-1	19	-1



- LeftChild(i) = 2*i
- RightChild(i) = 2*i + 1

ARRAY BINARY TREE - PARENT

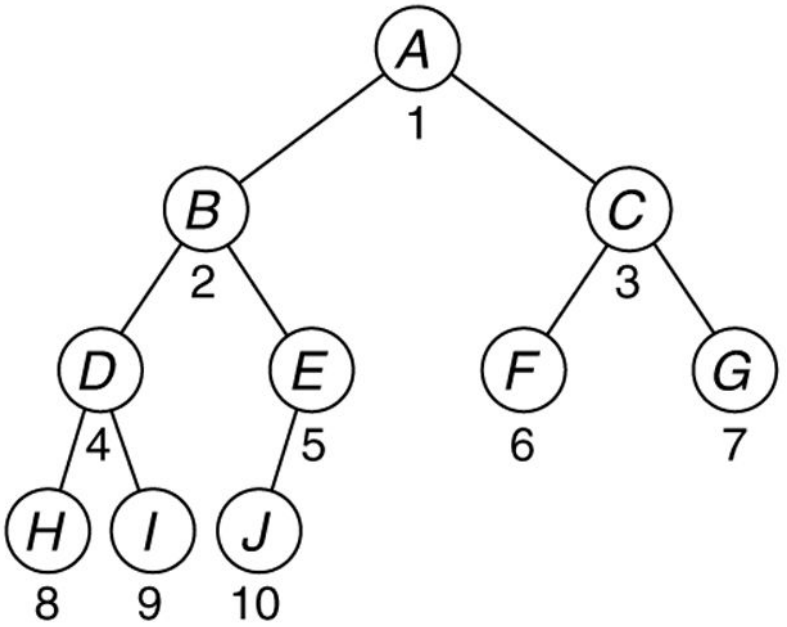
i	Item	Parent	i / 2
1	Pam	-1	0
2	Joe	1	1
3	Sue	1	1
4	Bob	2	2
5	Mike	2	2
6	Sam	3	3
7	Tom	3	3
8	Ann	4	4
9	Jane	4	4
10	Mary	5	5

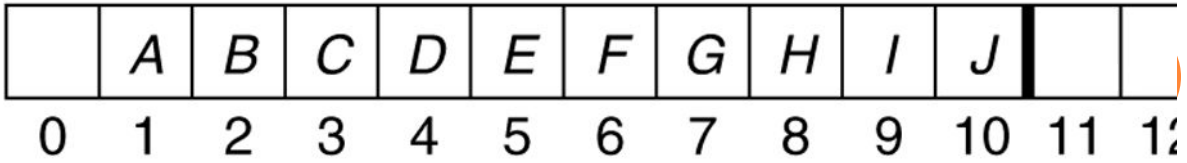


- Parent(i) = $\lfloor i / 2 \rfloor$

IMPLEMENTATION OF A HEAP

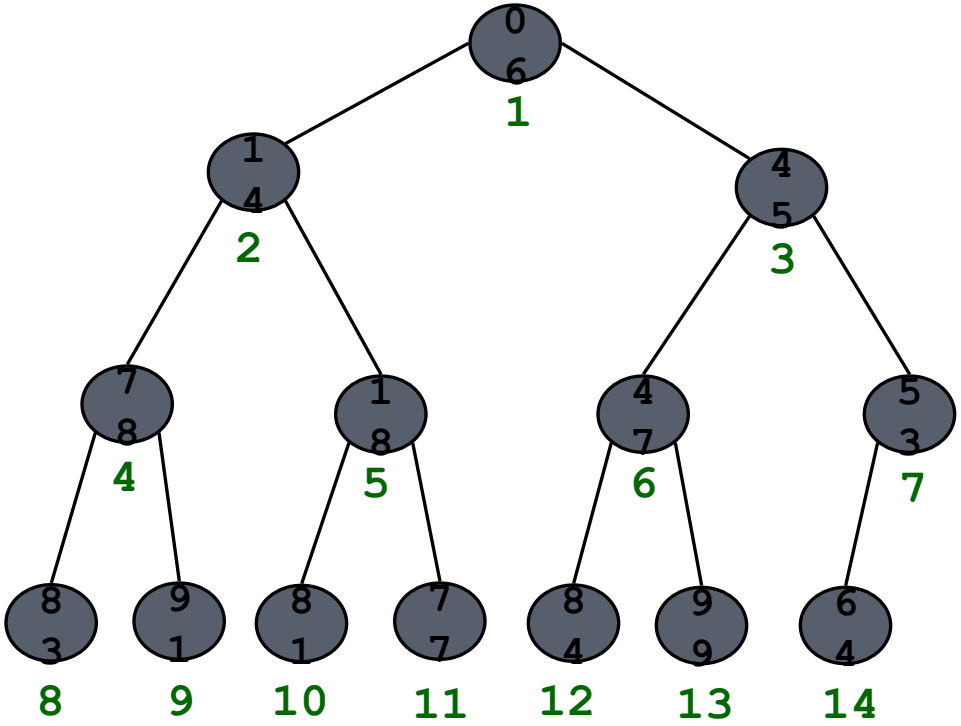
- when implementing a complete binary tree, we actually can "cheat" and just use an array
  - index of root = 1 (leave 0 empty for simplicity)
  - for any node *n* at index *i*,
    - index of *n*.left = 2*i*
    - index of *n*.right = 2*i* + 1





ARRAY IMPLEMENTATION

- Implementing binary heaps.
  - `Parent(i) = ⌊i/2⌋`
  - `Left(i)   = 2i`
  - `Right(i)  = 2i + 1`



INSERTION (MAX HEAP)

- Algorithm
  - Step 1 − Create a new node at the end of heap.
  - Step 2 − Assign new value to the node.
  - 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.

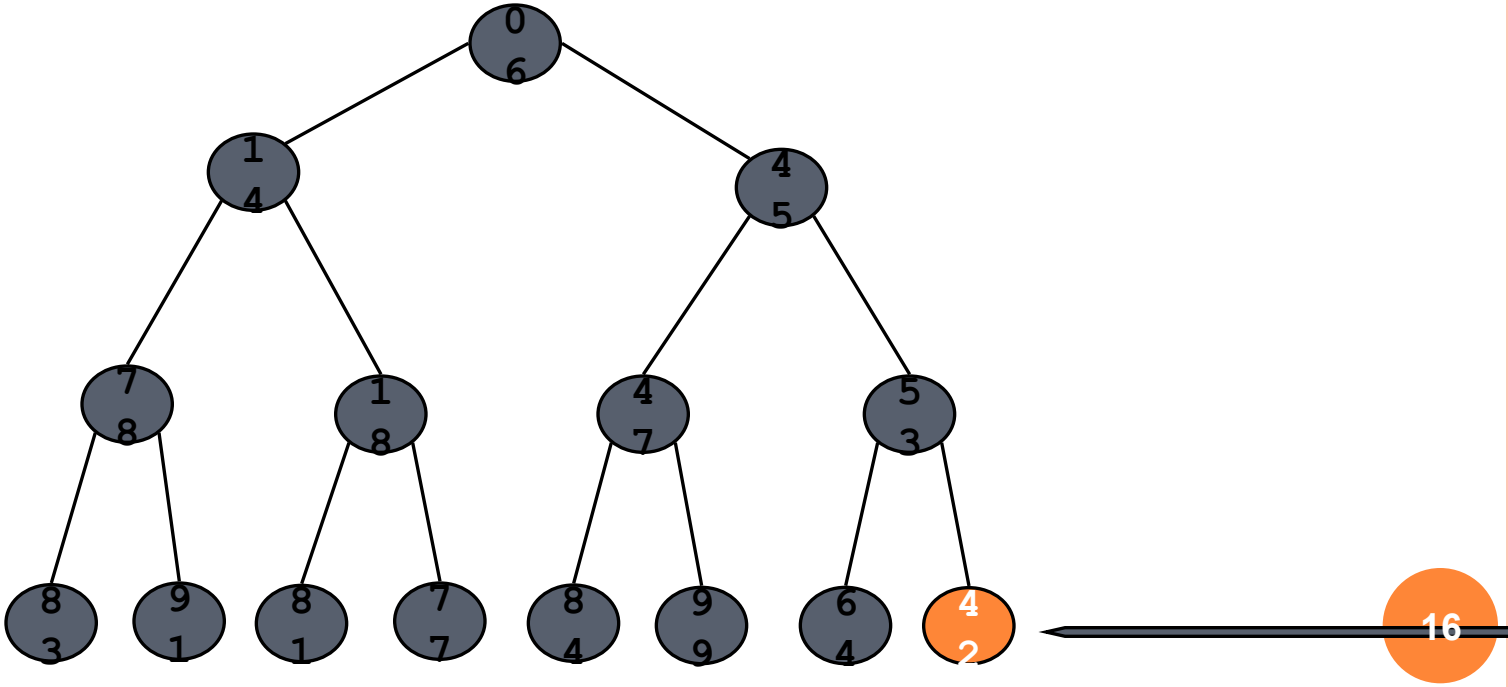
INSERTION (MAX HEAP)

- Example

Input 35 33 42 10 14 19 27 44 26 31

INSERTION (MIN HEAP)

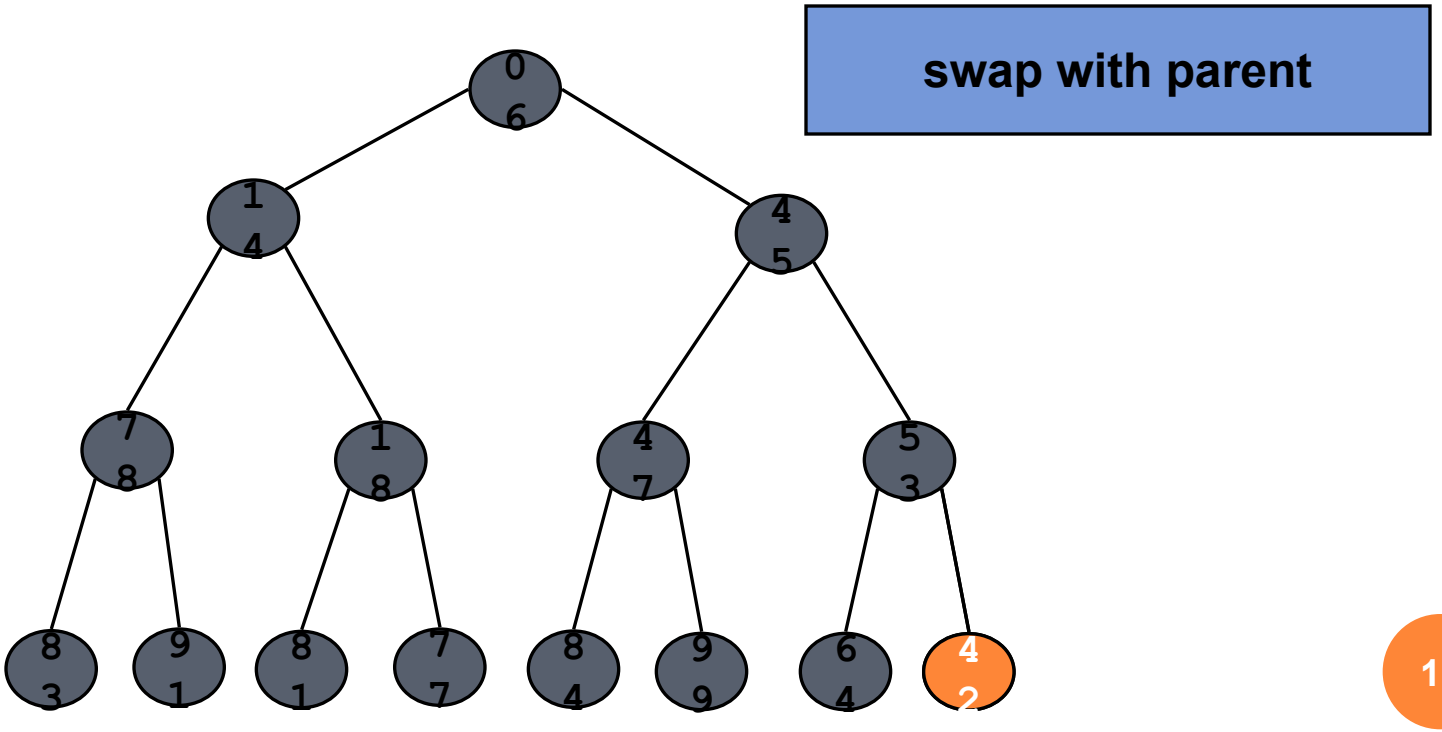
- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.



next free

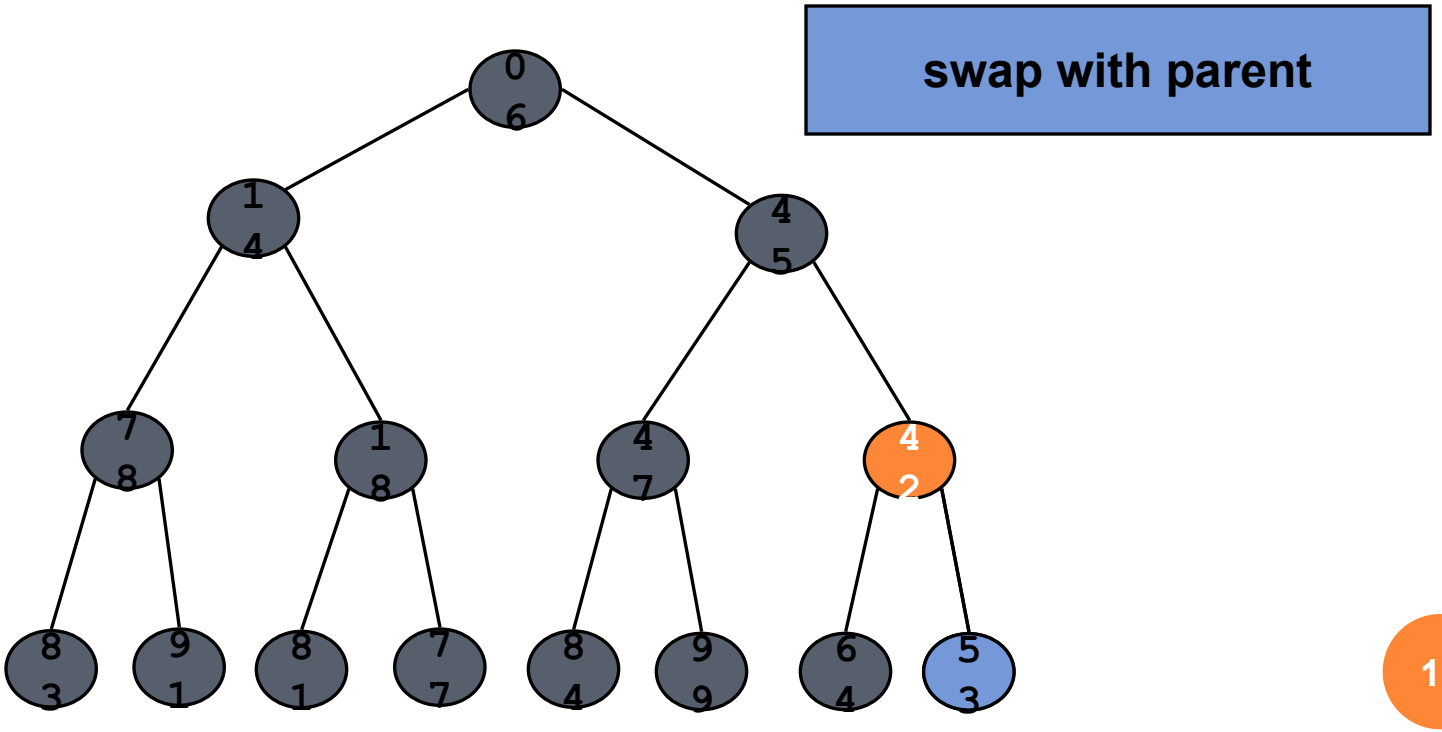
INSERTION (MIN HEAP)

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.



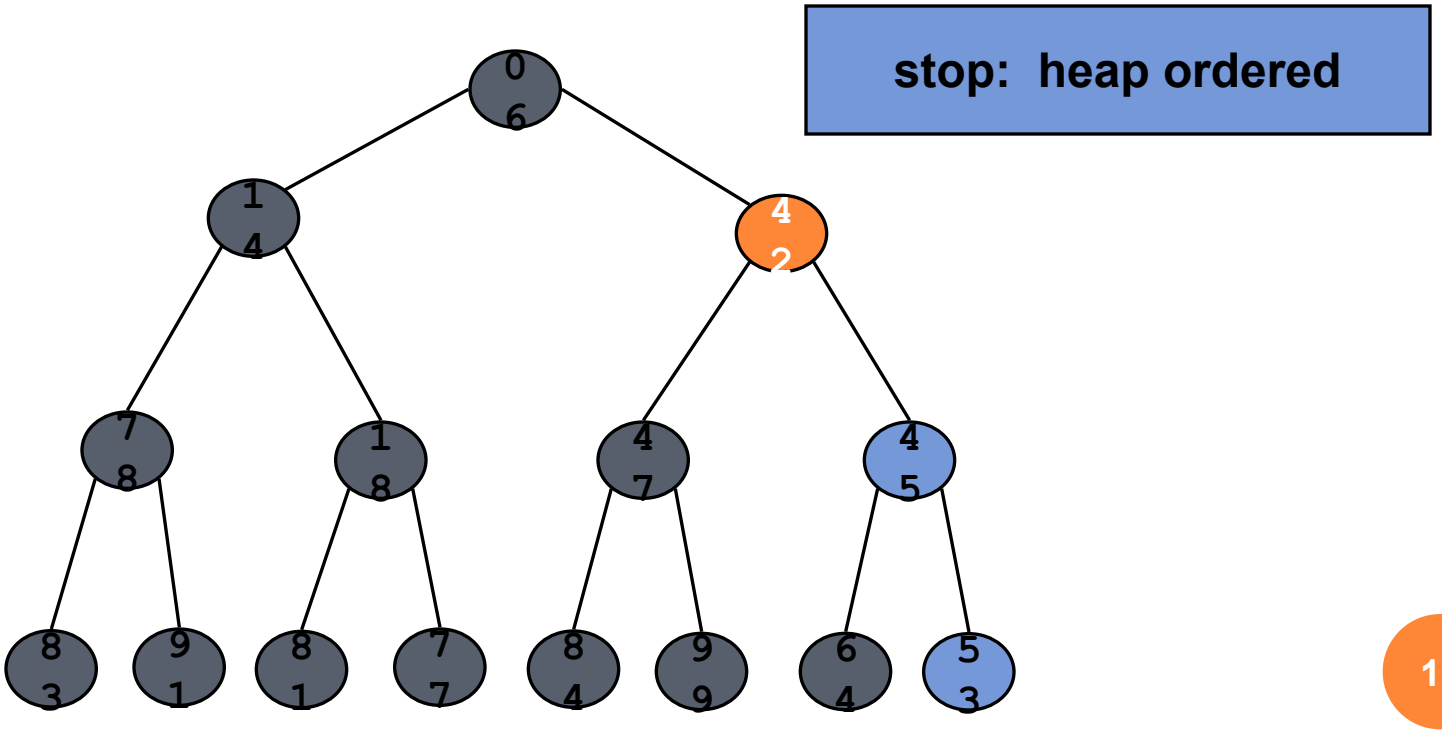
INSERTION (MIN HEAP)

- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.

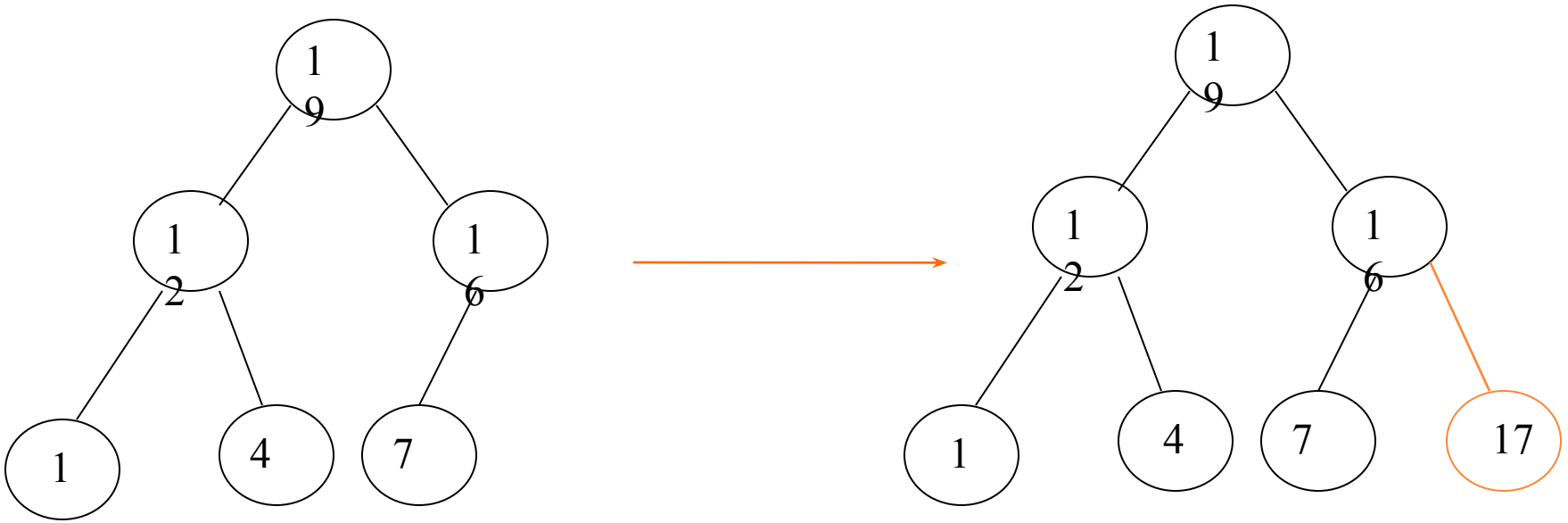


INSERTION (MIN HEAP)

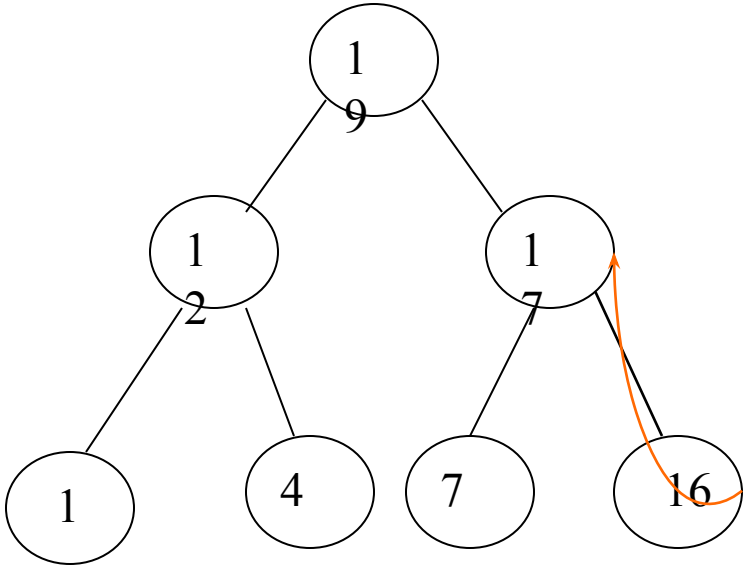
- Insert element x into heap.
  - Insert into next available slot.
  - Bubble up until it's heap ordered.
  - O(log N) operations.



INSERTION (ANOTHER EXAMPLE)



Insert 17



swap

Percolate up to maintain the heap property

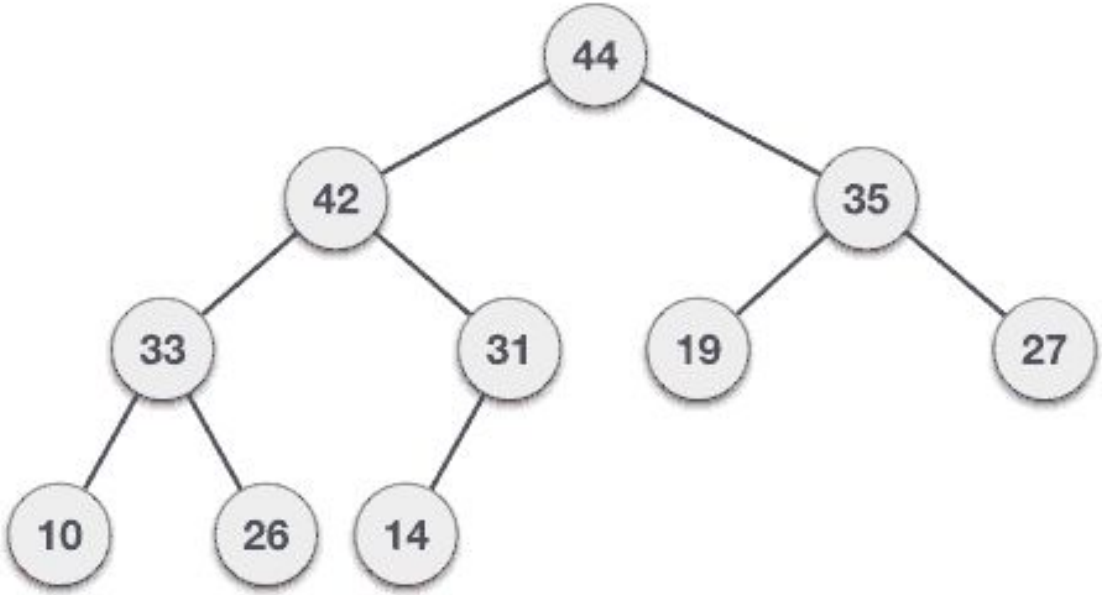
DELETION (MAX HEAP)

- Deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted with the last element and delete the last element of the Heap.

- Step 1 − Remove root node or element to be deleted with the last element.
- 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.

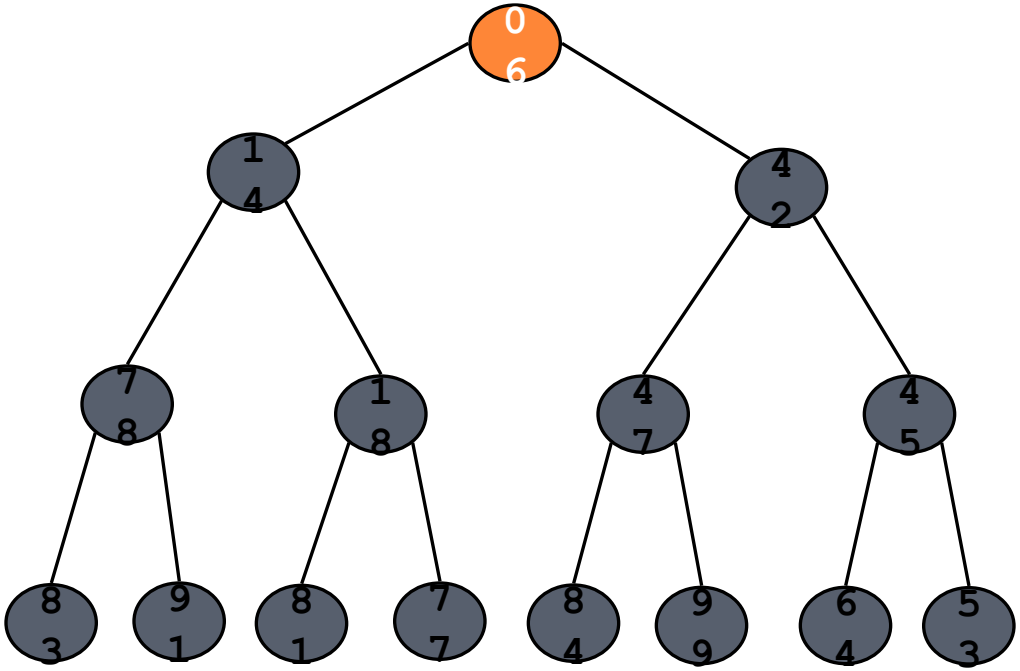
DELETION (MAX HEAP)

- Example



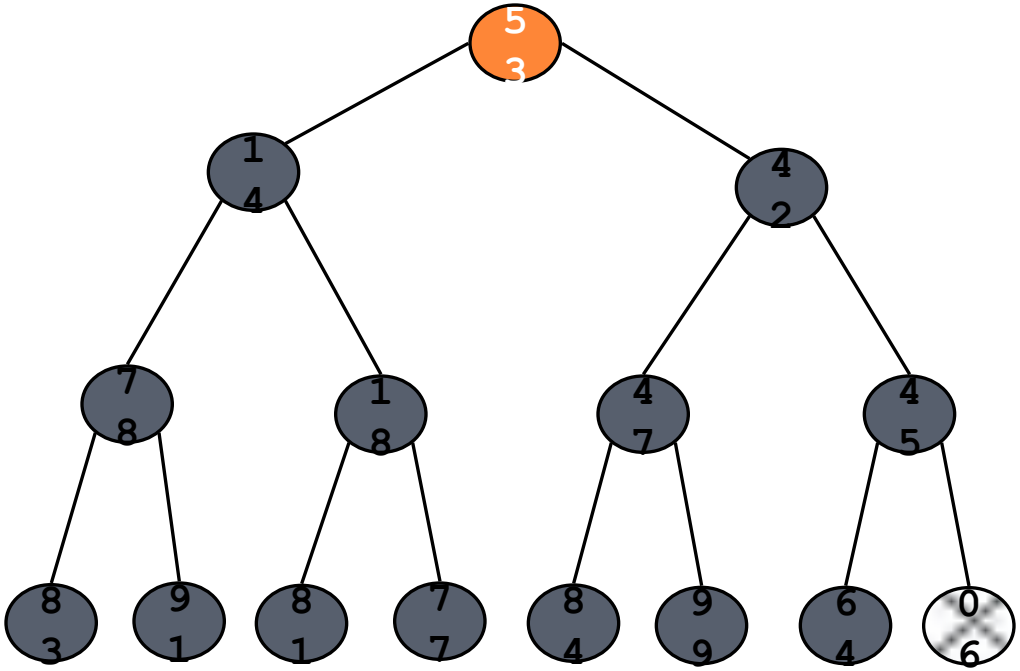
DELETE (MIN HEAP)

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.



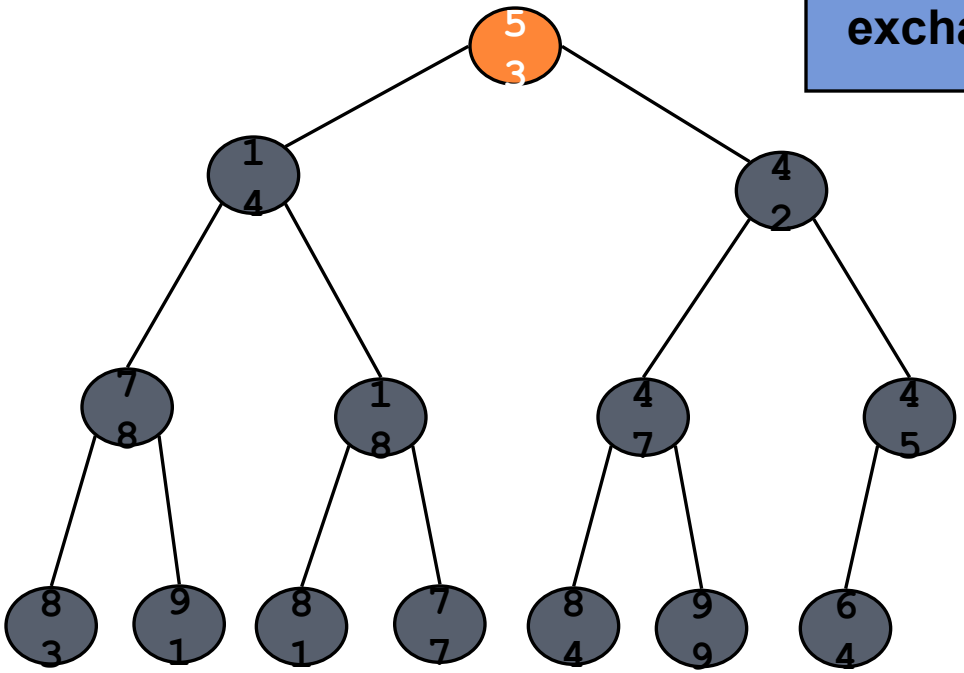
DELETE (MIN HEAP)

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.



DELETE (MIN HEAP)

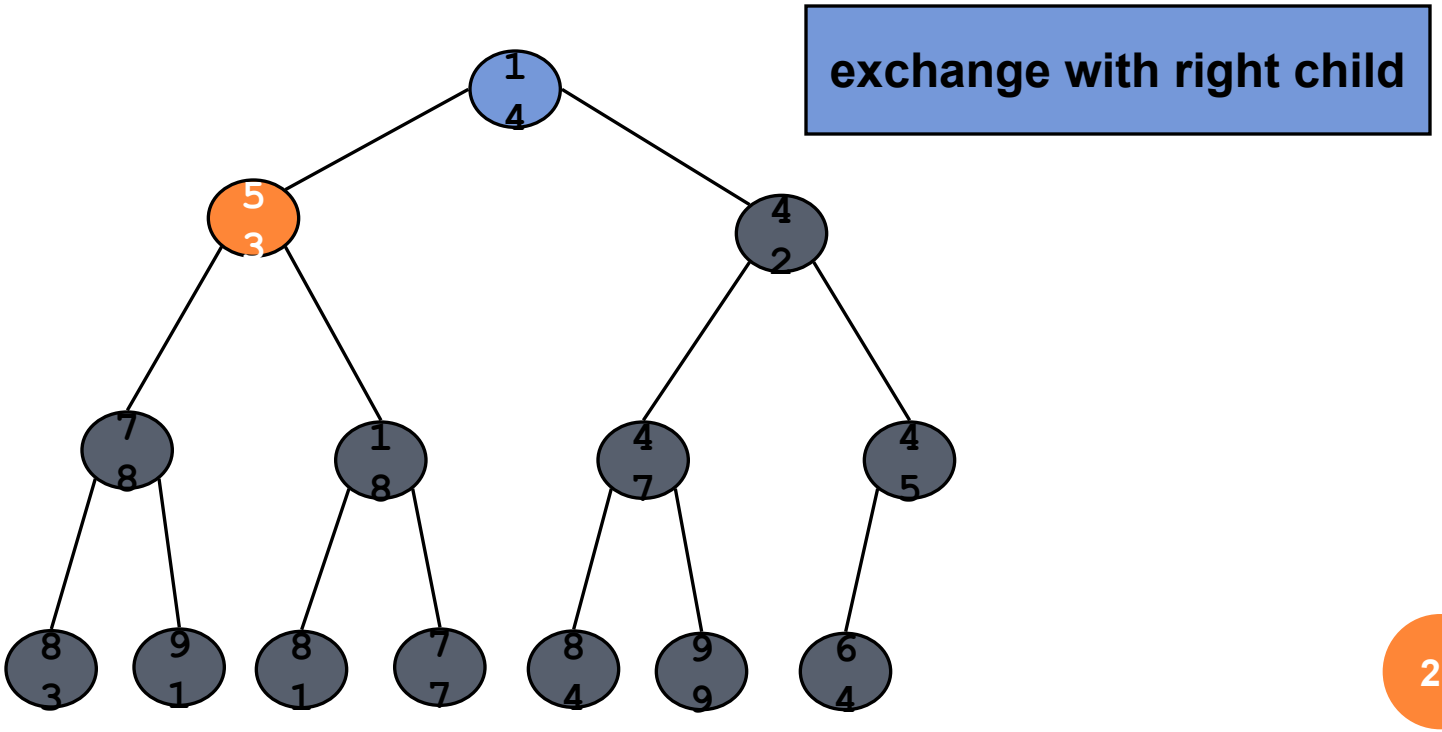
- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.



exchange with left child

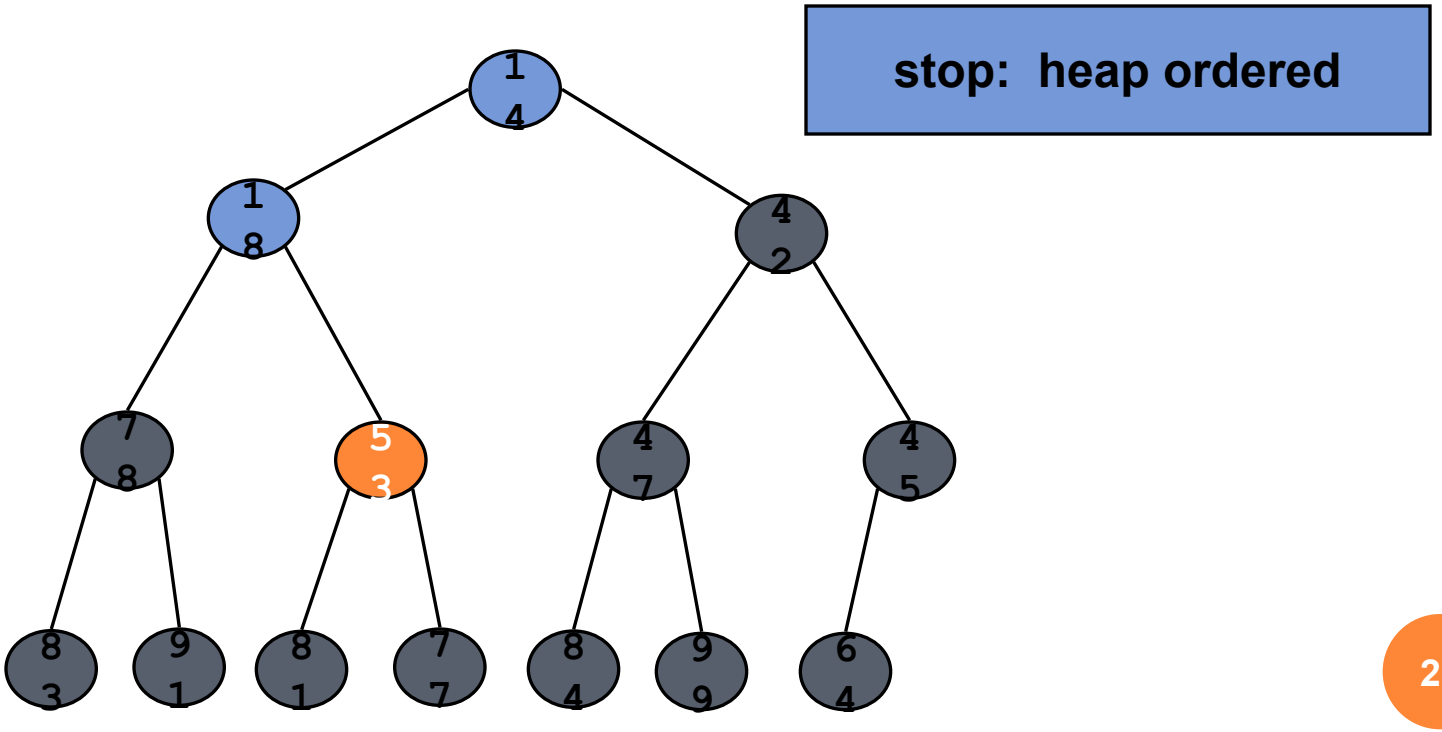
DELETE (MIN HEAP)

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.



DELETE (MIN HEAP)

- Delete minimum element from heap.
  - Exchange root with rightmost leaf.
  - Bubble root down until it's heap ordered.
  - O(log N) operations.



HEAP SORT

A sorting algorithm that works by first organizing the data to be sorted into a special type of binary tree called a heap.

PROCEDURES ON HEAP

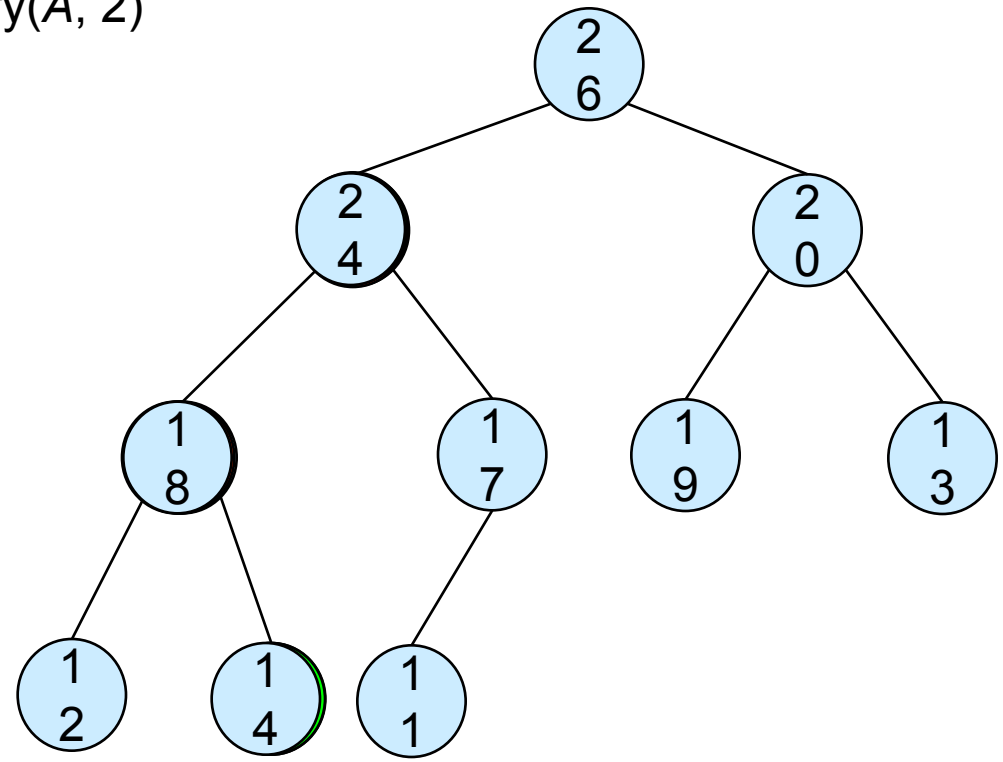
- Heapify
- Build Heap
- Heap Sort

HEAPIFY

- Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

MAXHEAPIFY – EXAMPLE

MaxHeapify(*A*, 2)



HEAPIFY

```
Heapify(A, i)
{
l □ left(i)
r □ right(i)
if l <= heapsize[A] and A[l] > A[i]
    then largest □l
    else largest □ i
if r <= heapsize[A] and A[r] > A[largest]
    then largest □ r
if largest != i
    then swap A[i] □□ A[largest]
        Heapify(A, largest)
}
```

BUILD HEAP

- We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[*n*/2 +1 . . *n*] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

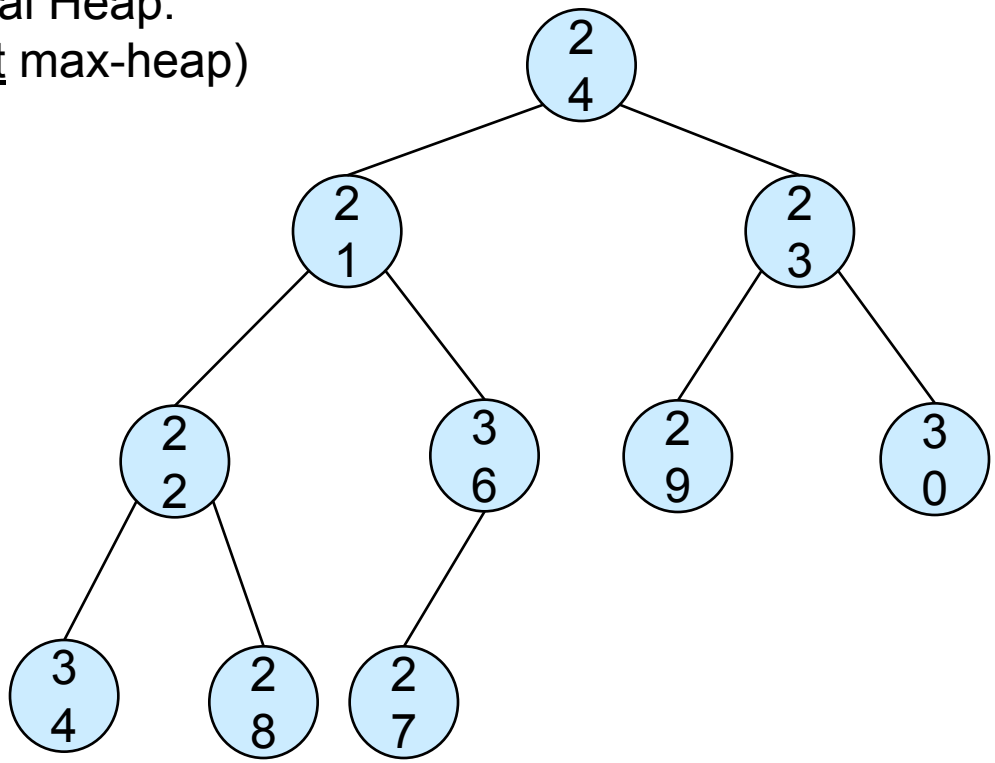
```
Buildheap(A)
{
    heapsize[A] □length[A]
    for i □|length[A]/2  //down to 1
       do Heapify(A, i)
}
```

BUILD MAXHEAP – EXAMPLE

Input Array:

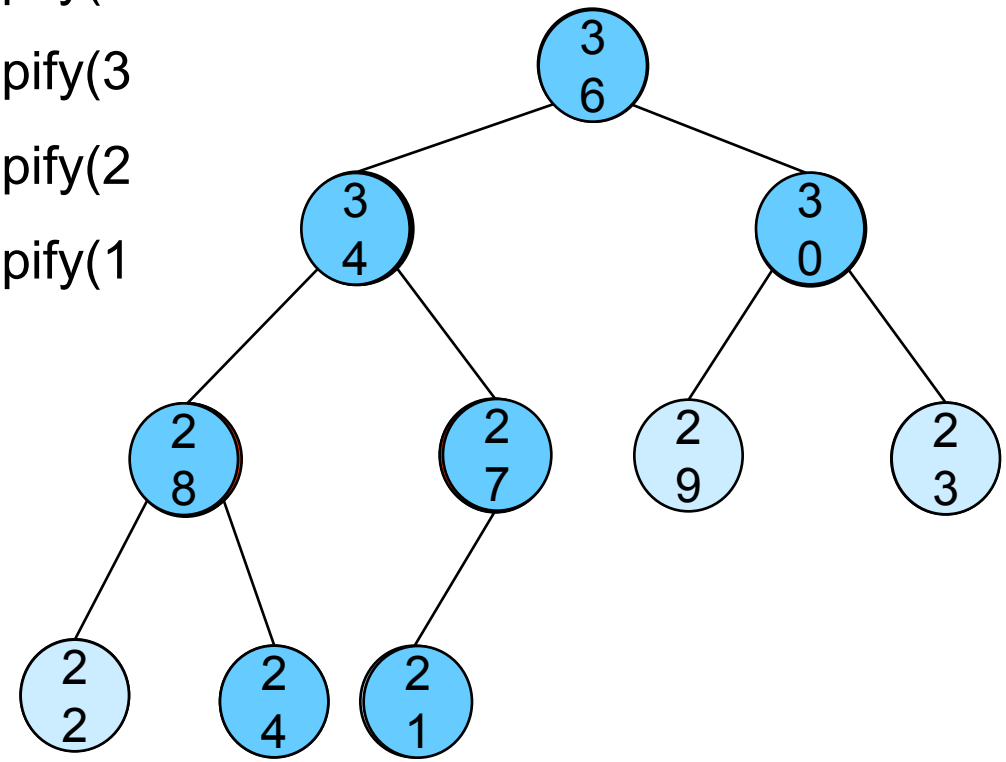
24	21	23	22	36	29	30	34	28	27

Initial Heap:
(not max-heap)



BUILD MAXHEAP – EXAMPLE

MaxHeapify(⌊10/2⌋ = 5)
MaxHeapify(4)
MaxHeapify(3)
MaxHeapify(2)
MaxHeapify(1)



HEAP SORT ALGORITHM

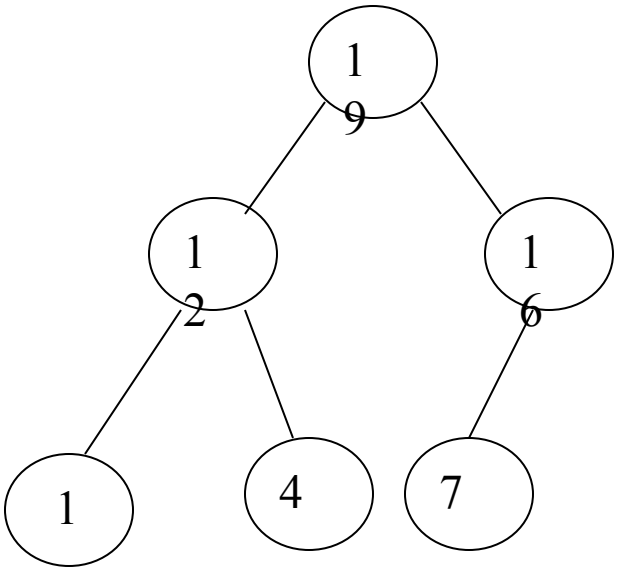
- The heap sort algorithm starts by using procedure BUILD-HEAP to build a heap on the input array A[1 . . *n*]. Since the maximum element of the array stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*). If we now discard node n from the heap than the remaining elements can be made into heap. Note that the new element at the root may violate the heap property. All that is needed to restore the heap property.

```
Heapsort(A)
{
      Buildheap(A)
      for i □ length[A] //down to 2
         do swap A[1] □□ A[i]
         heapsize[A] □ heapsize[A] - 1
         Heapify(A, 1)
}
```

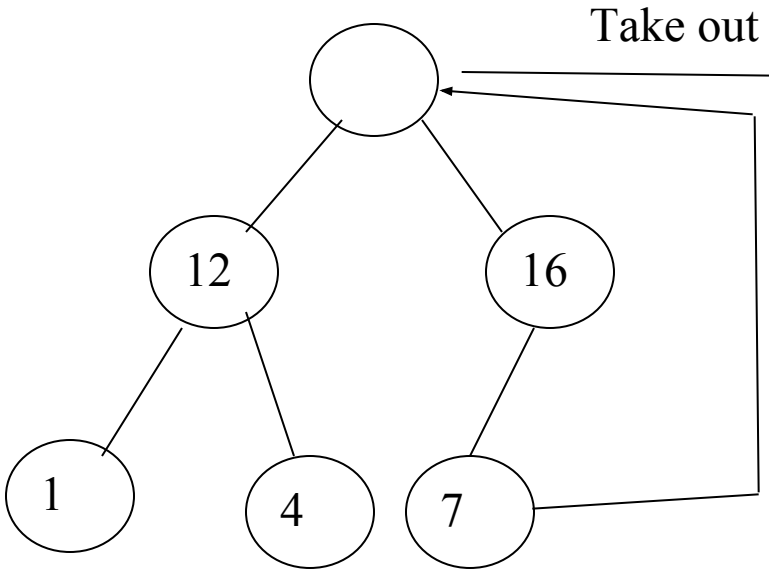
HEAP SORT

- The heapsort algorithm consists of two phases:
  - build a heap from an arbitrary array
  - use the heap to sort the data

- To sort the elements in the decreasing order, use a min heap
- To sort the elements in the increasing order, use a max heap



EXAMPLE OF HEAP SORT



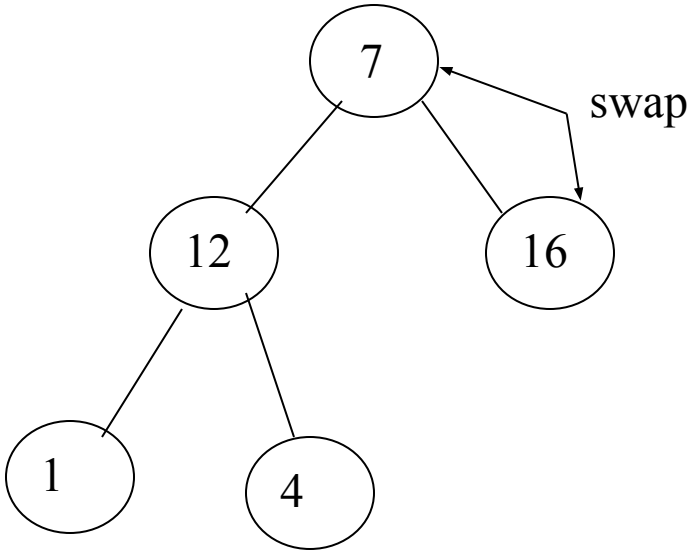
Array A

12	16	1	4	7

Sorted:

19

HEAPIFY()

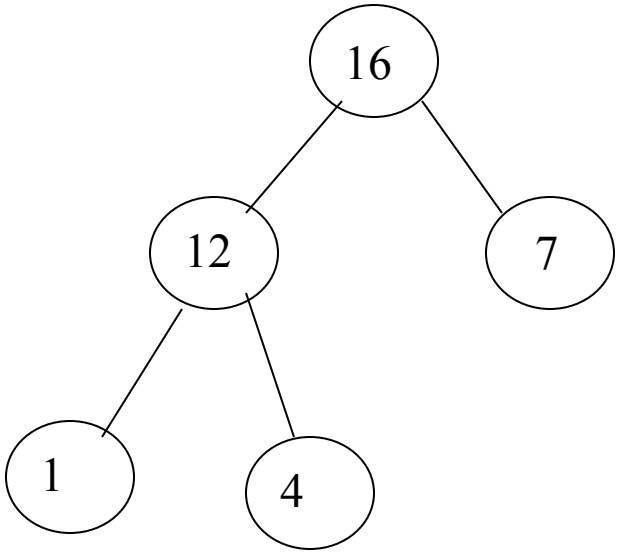


Array A

7	12	16	1	4

Sorted:

19

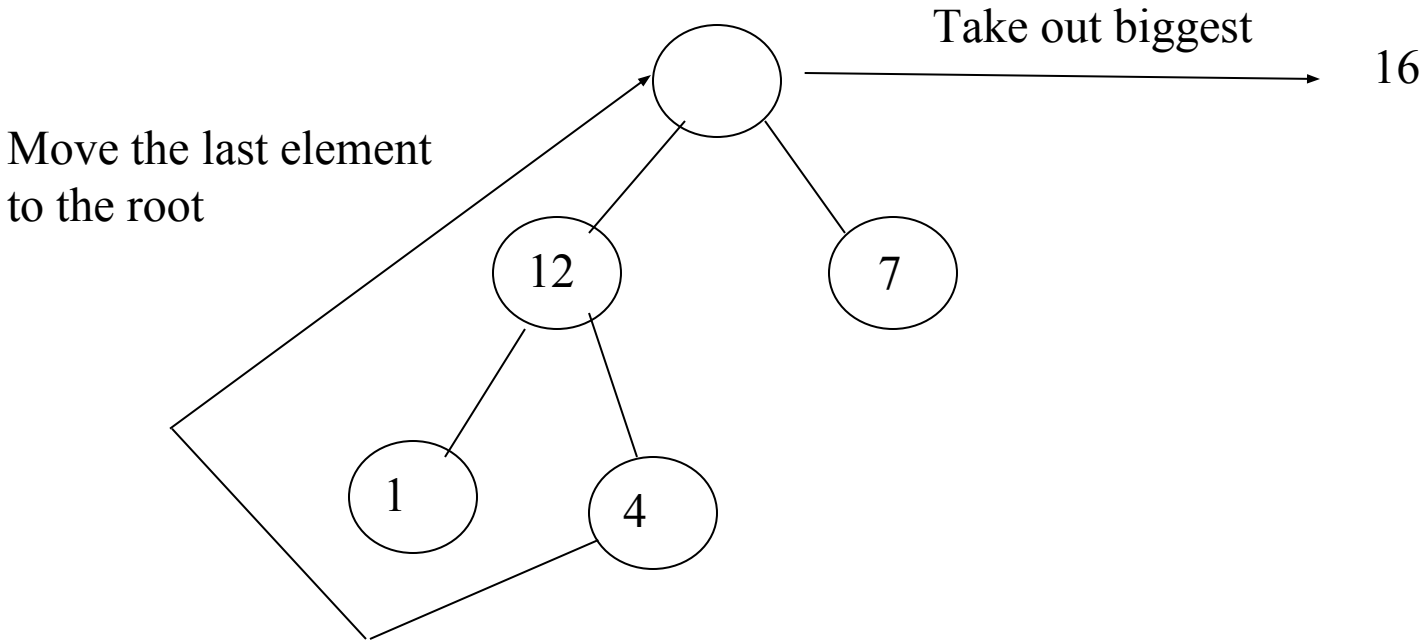


Array A

16	12	7	1	4

Sorted:

19

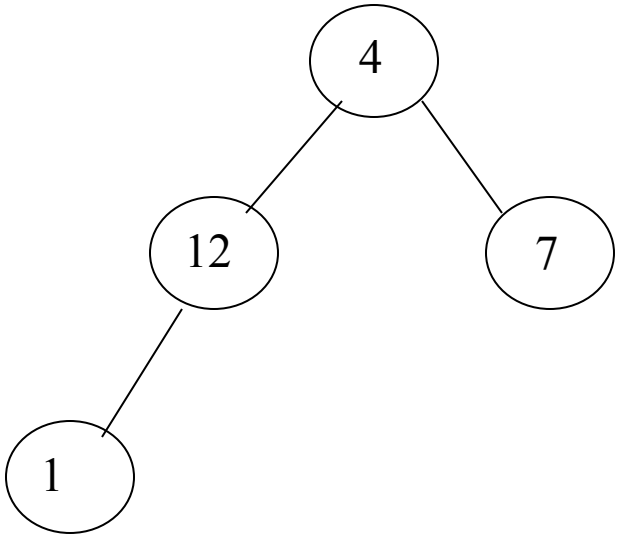


Array A

12	7	1	4

Sorted:

16	19



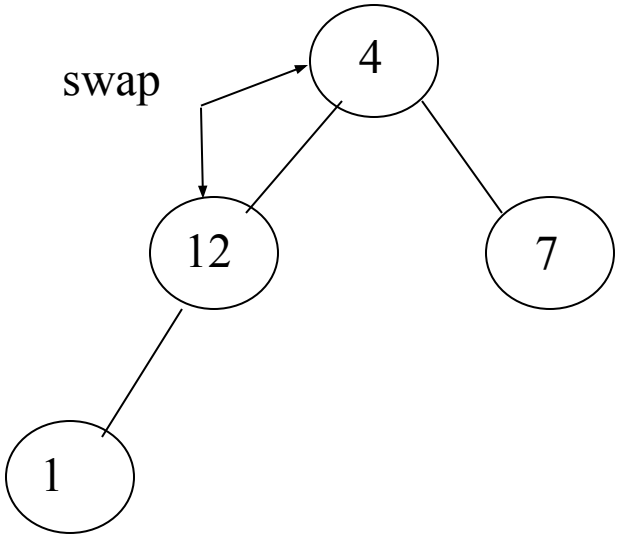
Array A

4	12	7	1

Sorted:

16	19

HEAPIFY()

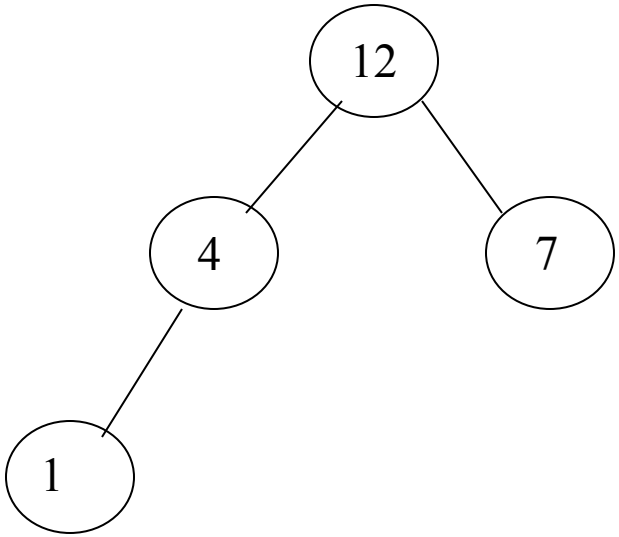


Array A

4	12	7	1

Sorted:

16	19

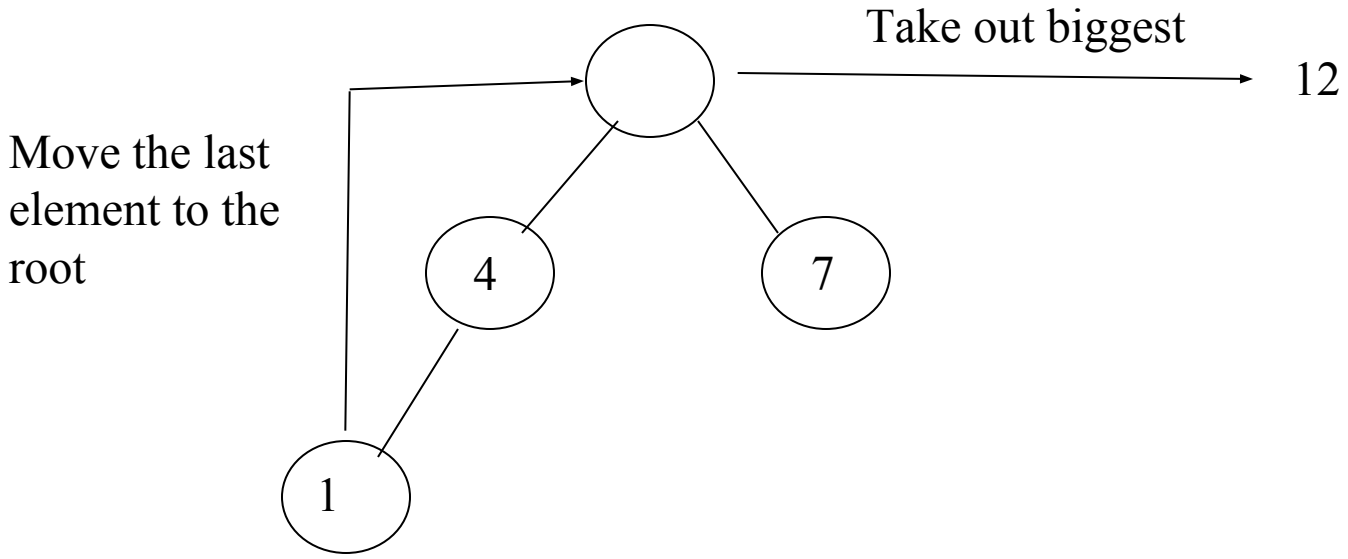


Array A

12	4	7	1

Sorted:

16	19

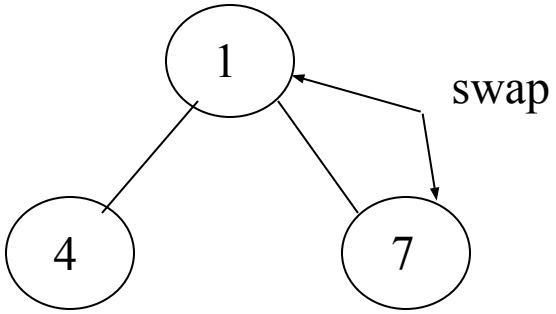


Array A

4	7	1

Sorted:

12	16	19

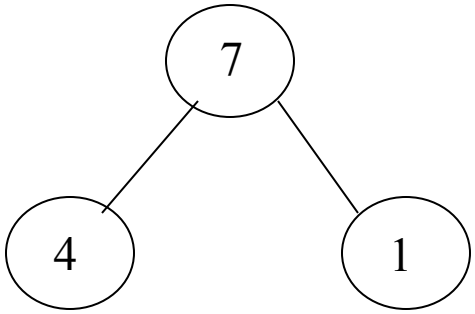


Array A

1	4	7

Sorted:

12	16	19

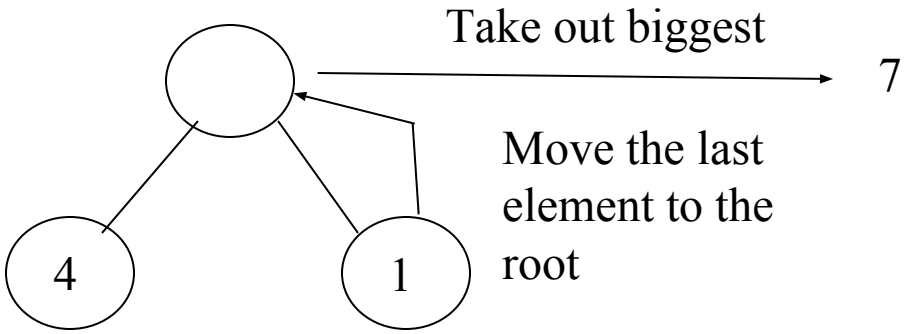


Array A

7	4	1

Sorted:

12	16	19



Array A

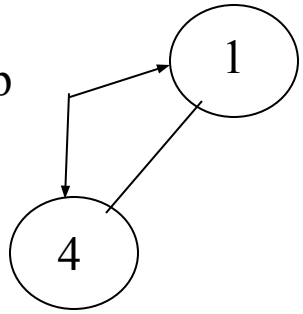
1	4

Sorted:

7	12	16	19

HEAPIFY()

swap



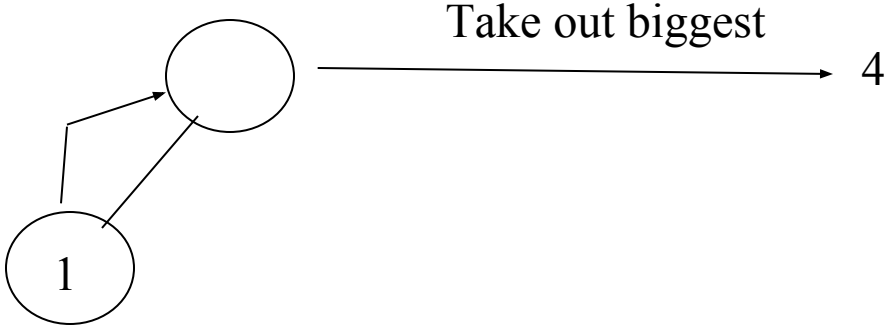
Array A

4	1

Sorted:

7	12	16	19

Move the last element to the root

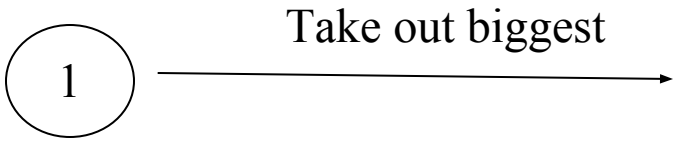


Array A

1

Sorted:

4	7	12	16	19

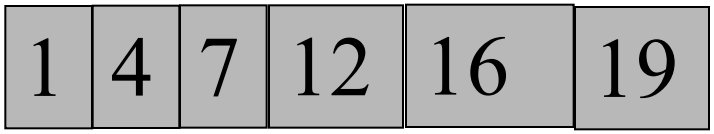


Array A

Sorted:

1	4	7	12	16	19

Sorted:



TIME ANALYSIS

- Build Heap Algorithm will run in O(n) time
- There are *n*-1 calls to Heapify each call requires O(log *n*) time
- Heap sort program combine Build Heap program and Heapify, therefore it has the running time of O(n log n) time
- Total time complexity: O(n log n)

POSSIBLE APPLICATION

- When we want to know the task that carry the highest priority given a large number of things to do
- Interval scheduling, when we have a lists of certain task with start and finish times and we want to do as many tasks as possible
- Sorting a list of elements that needs and efficient sorting algorithm