Heap Sort

Prof. Pramod Nath
Assistant Professor IT
KIET

The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree

TYPES:

•**Max-heaps** (largest element at root), have the *max-heap property:*

- for all nodes i, excluding the root:

  A[PARENT(i)] ≥ A[i]

•**Min-heaps** (smallest element at root), have the *min-heap property:*

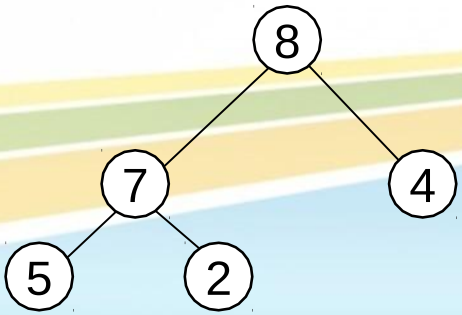
- for all nodes i, excluding the root:

  A[PARENT(i)] ≤ A[i]

The Heap Data Structure

- *Def:* A **heap** is a nearly complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Order (heap) property:** for any node x

$$\mathrm{Parent}(x) \geq x$$



Heap

From the heap property, it follows that:
"The root is the maximum element of the heap!"

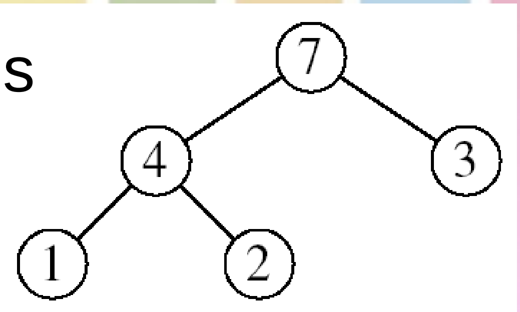
A heap is a binary tree that is filled in order

Why study Heapsort?

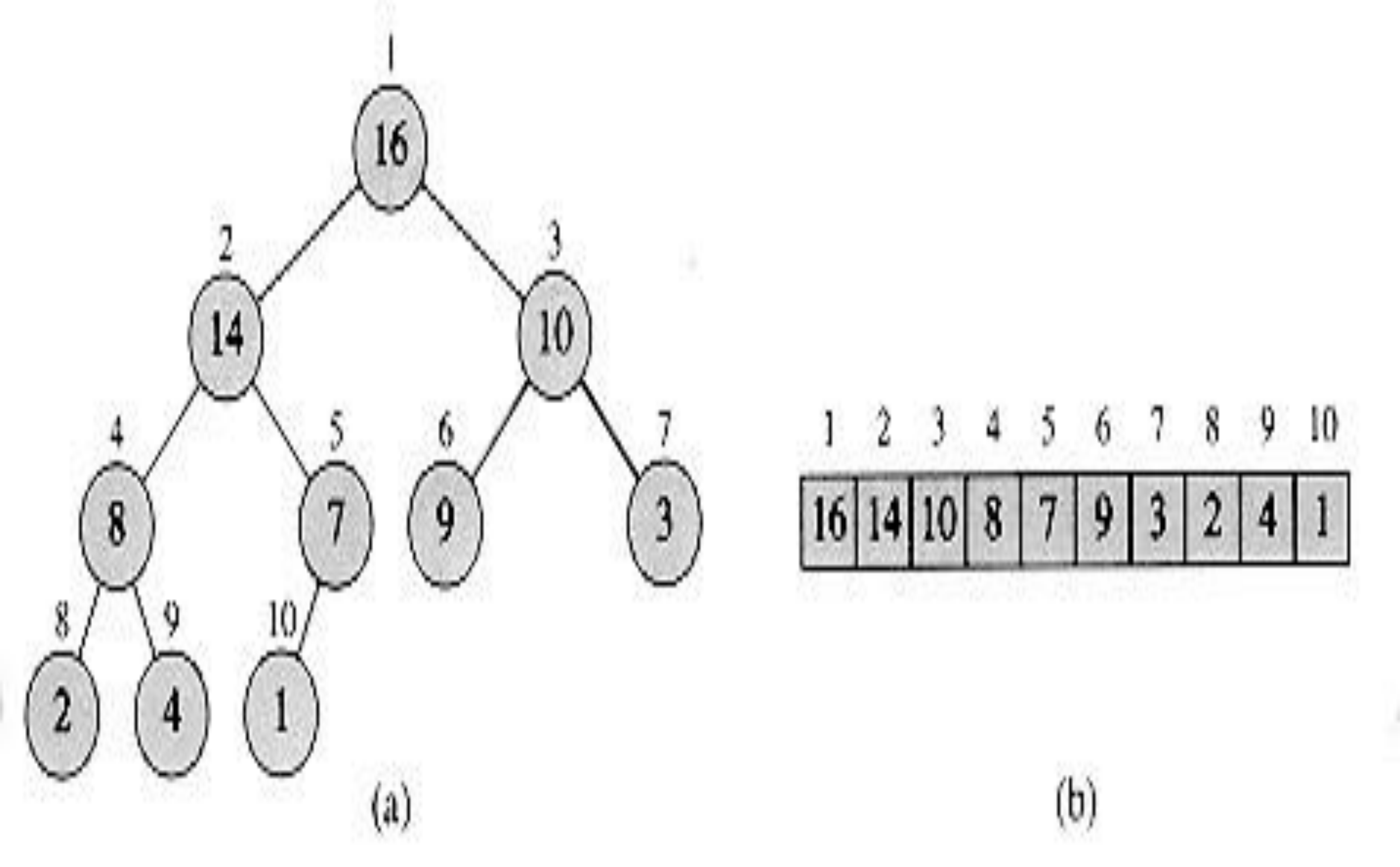
- It is a well-known, traditional sorting algorithm you will be expected to know
- Heapsort is *always* O(n log n)
  - Quicksort is usually O(n log n) but in the worst case slows to $O(n^2)$
  - Quicksort is generally faster, but Heapsort is better in time-critical applications
- Heapsort is a *really cool* algorithm!

Heapsort

- Goal:
  - Sort an array using heap representations



- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - "Discard" this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains



- Figure (a) Max Heap Tree
- Figure (b) Array Representation of Max Heap Tree

Operations on Heaps

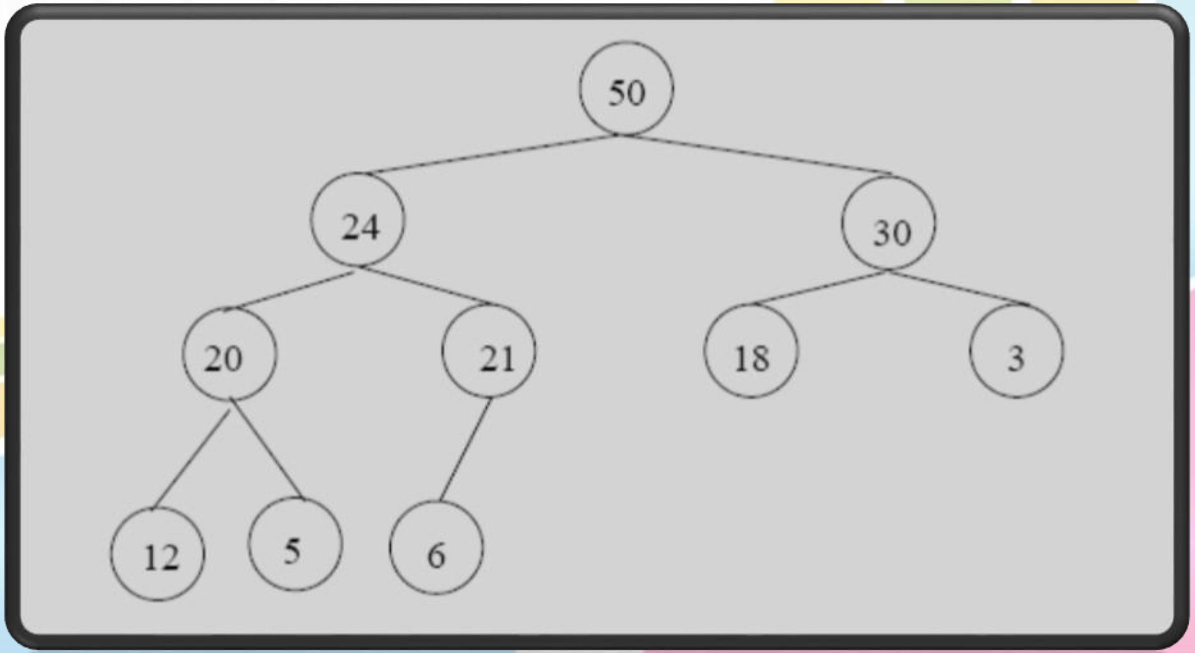
Insertion.

Heapify.

Deletion.

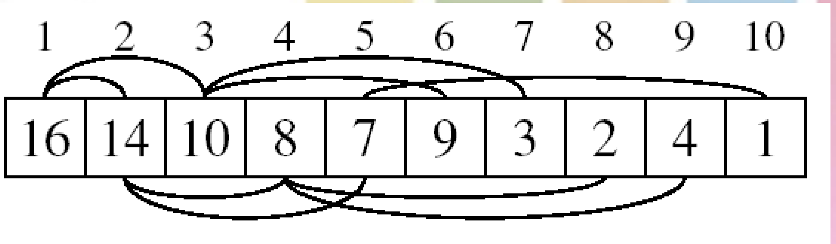
Rule of Adding/Deleting Nodes

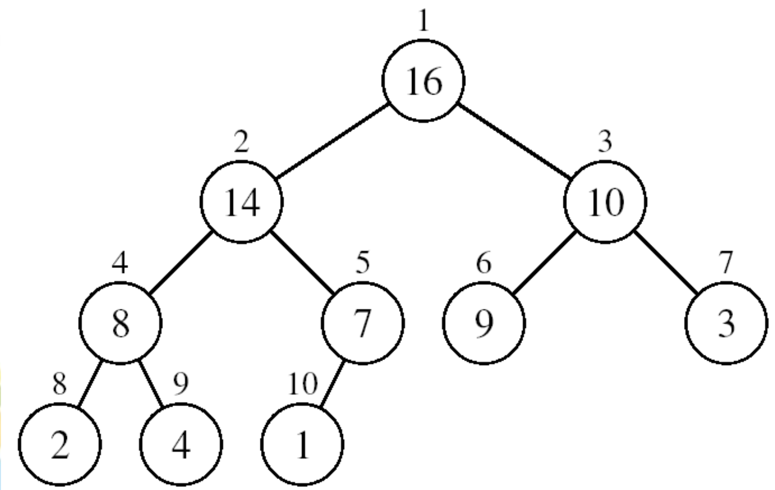
- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)



Array Representation of Heaps

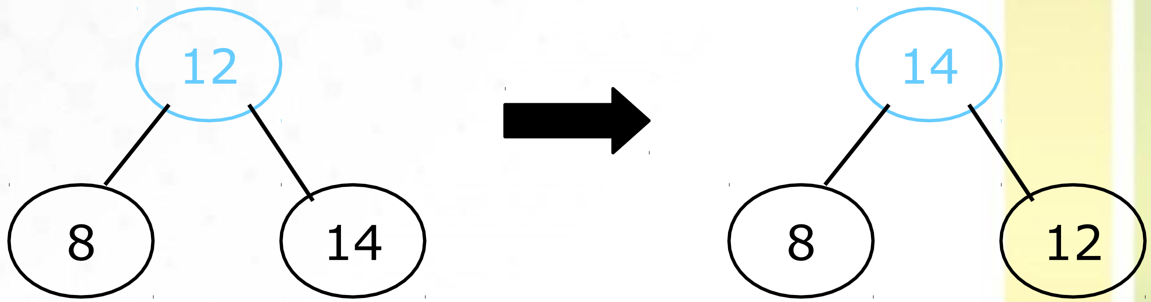
- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) .. n]$ are leaves





Heapify

- Given a node that does not have the heap property, you can give it the heap property by exchanging its value with the value of the larger child



Blue node does not have heap property

Blue node has heap property

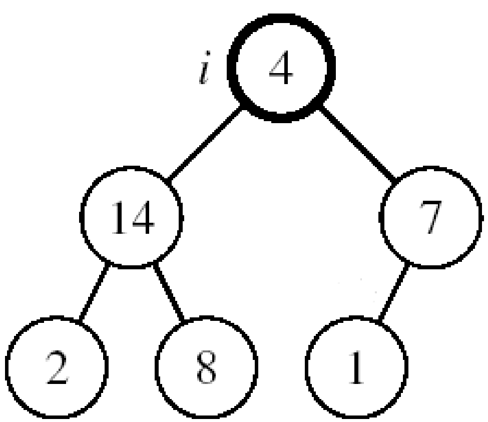
- This is sometimes called shifting up

Problem during Constructing a heap

- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we perform Heapify procedure.
- We repeat the Heapify process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of i are max-heaps
  - $A[i]$ may be smaller than its children



Alg: MAX-HEAPIFY(*A, i, n*)

```
l ← LEFT(i)              /* l = 2i */
r ← RIGHT(i)             /* r = l + 1 */
if l ≤ n and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ n and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] ↔ A[largest]
        MAX-HEAPIFY(A, largest, n)
```

Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap ($n = length[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: <u>BUILD-MAX-HEAP(A)</u>

1. n = length[A]
2. **for** i $\leftarrow \lfloor n/2 \rfloor$ **down to** 1
3. **do** MAX-HEAPIFY(A, i, n) $O(\log n)$

$O(n)$

$\Rightarrow$ Running time: $O(n\lg n)$

Heap Sort

HEAP-SORT(A)

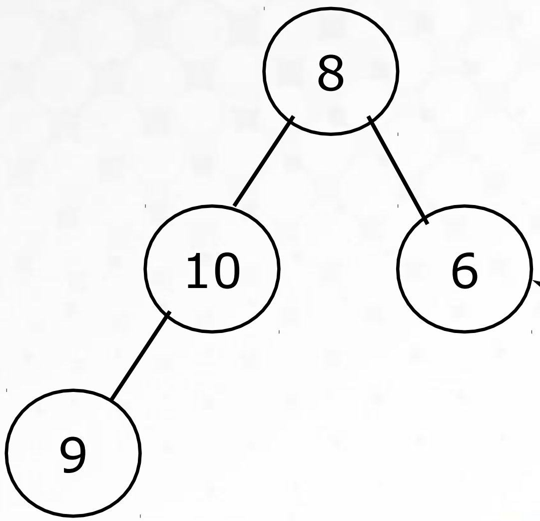
1. BUILD-MAX-HEAP(A)
2. For (i = n; i >= 2; i --)   / * i = length[A] down to 1 */

    swap (A[1], A[i])

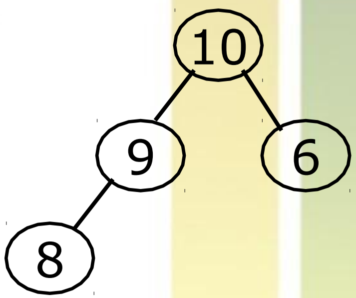
    do MAX-HEAPIFY(A, 1, i - 1)

Example

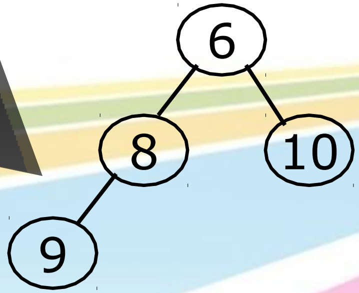
Make a tree from Array **8 10 6 9**



ACBT / Left Justify Tree



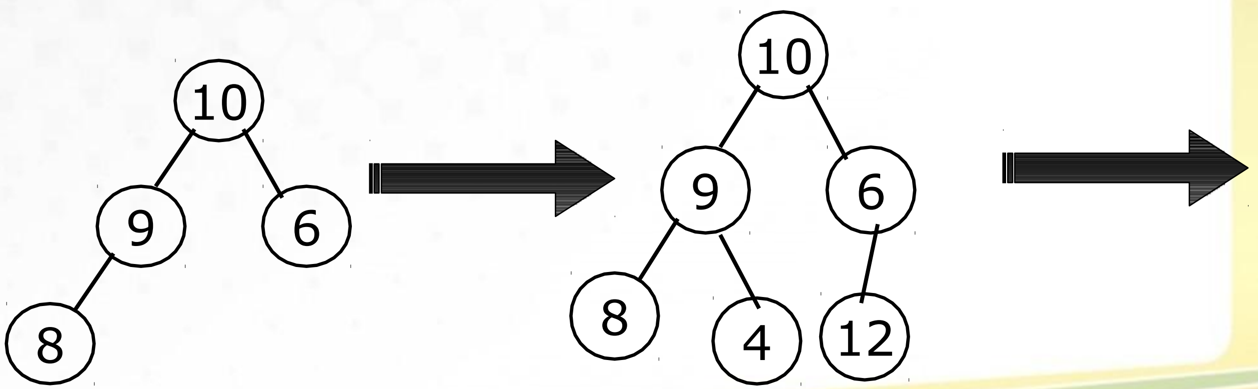
Max Heap-Sort Tree



Min Heap-Sort Tree

Insertion

Insert following number in tree: 4,12

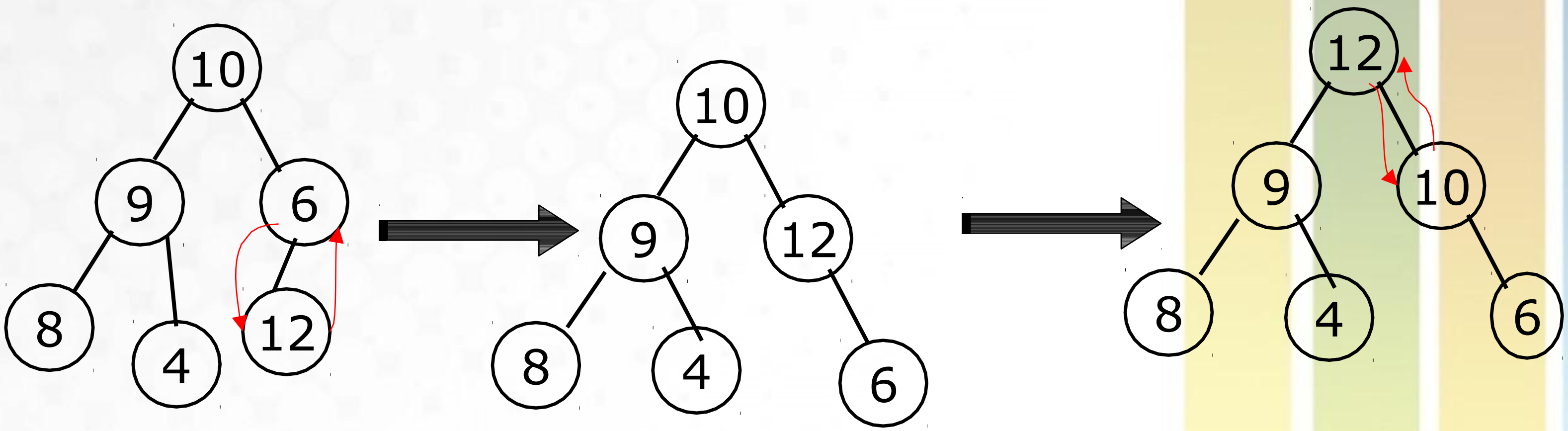


Insert 4

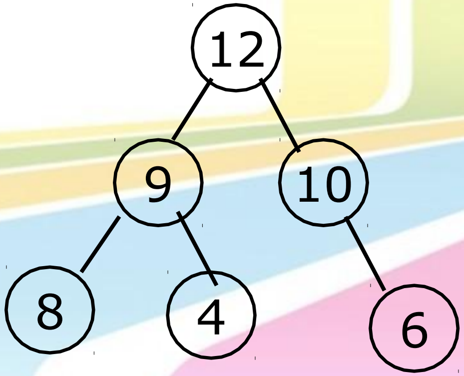
After inserting 4.12 heapify property of max heap is destroy, again make it max heap

After insertion this tree doesn't support the heapify property, to support the heapify property we should sort the tree

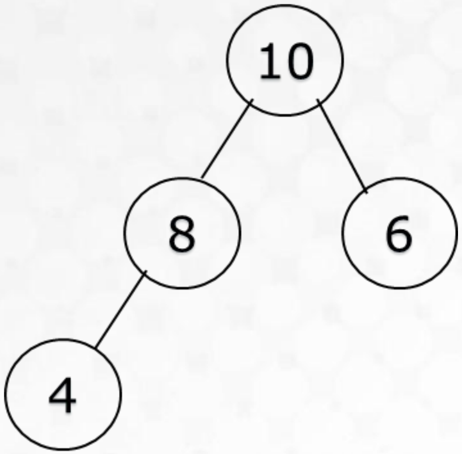
Insertion

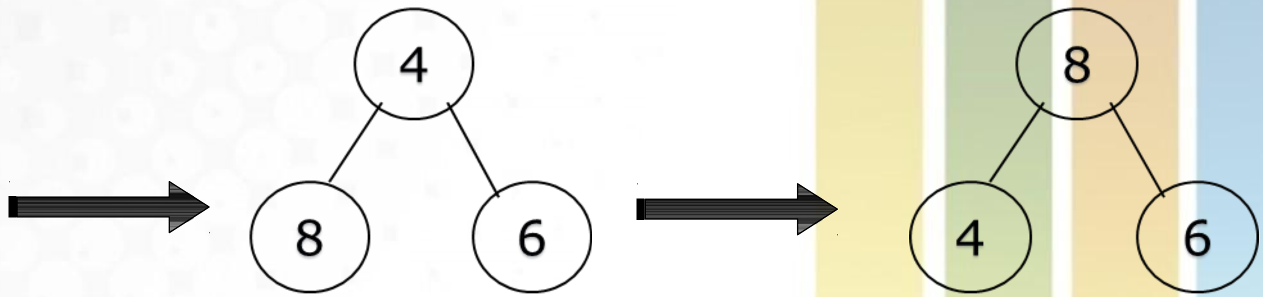


Sorted Tree in Max-heap.

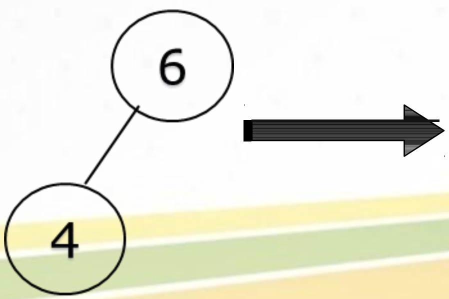


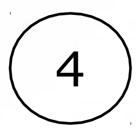
Deletion 10,8,6,4





For Sorted tree
swap 8,with 4





Delete 4 and also
goto array

4 6 8 10

Sorted array

Here tree is
already maxheap
so no swapping

MAX-HEAP-INSERT

- Goal:
  - Inserts a new element into a max- heap
- Idea:
  - Expand the max-heap with a new element whose key is x
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property

Running time: $\boldsymbol{O(\log(n))}$

Heapify()

- What do heaps have to do with sorting an array?
- Here's the neat part:
  - Because the binary tree is *balanced* and *ACBT / left justified*,
    it can be represented as an array
  - All our operations on binary trees can be represented as operations on *arrays*
  - To sort:

```
heapify the array;
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;
}
```

Time complexity

- We can perform Major operations on heaps:

¬**heapify** **which runs in O(log *n*) time.**

¬**Build-heap** **which returns in linear time.**

¬**Heap sort, which runs in O(*n* log *n*) time.**

Complexity of the Heap Sort Algorithm
To sort an unsorted list with '**n**' number of elements, following are the complexities...

Worst Case : O(n log n)
Best Case : O(n log n)
Average Case : O(n log n)

Q1. 5, 13, 2, 25, 7, 17, 20, 8, 4

Q2. 13, 4, 11, 15, 59, 27, 19, 3, 92, 5

Q3. 4, 1, 3, 2, 16, 9, 10, 14, 8, 7

Any Question…???