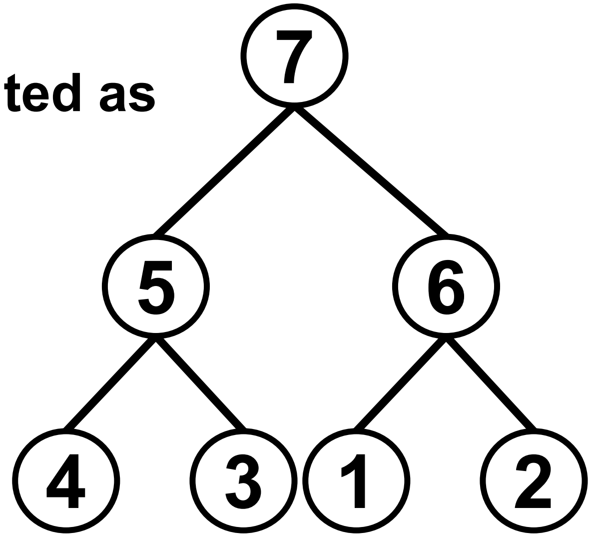
HEAPS

MAX HEAP DATA STRUCTURE

HEAPS

Definition: **A heap is a tree-based data structure that satisfies the heap property:**

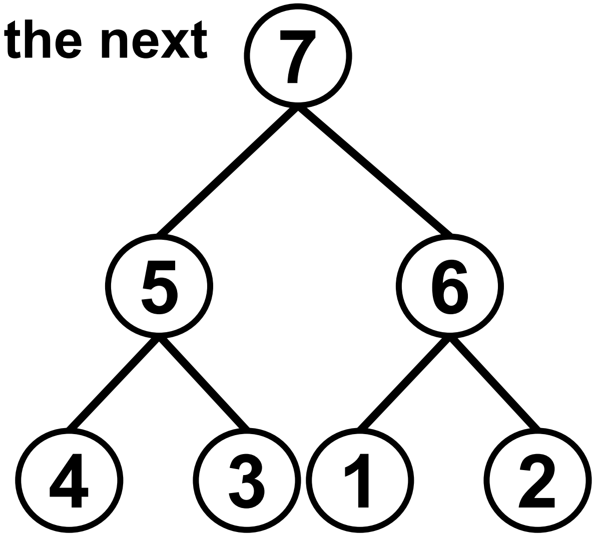
- **In a max heap, a parent node is always larger than (or equal to) its children**
- **In a min heap, a parent node is always smaller than (or equal to) its children.**
- **Heaps are commonly implemented as complete binary trees.**



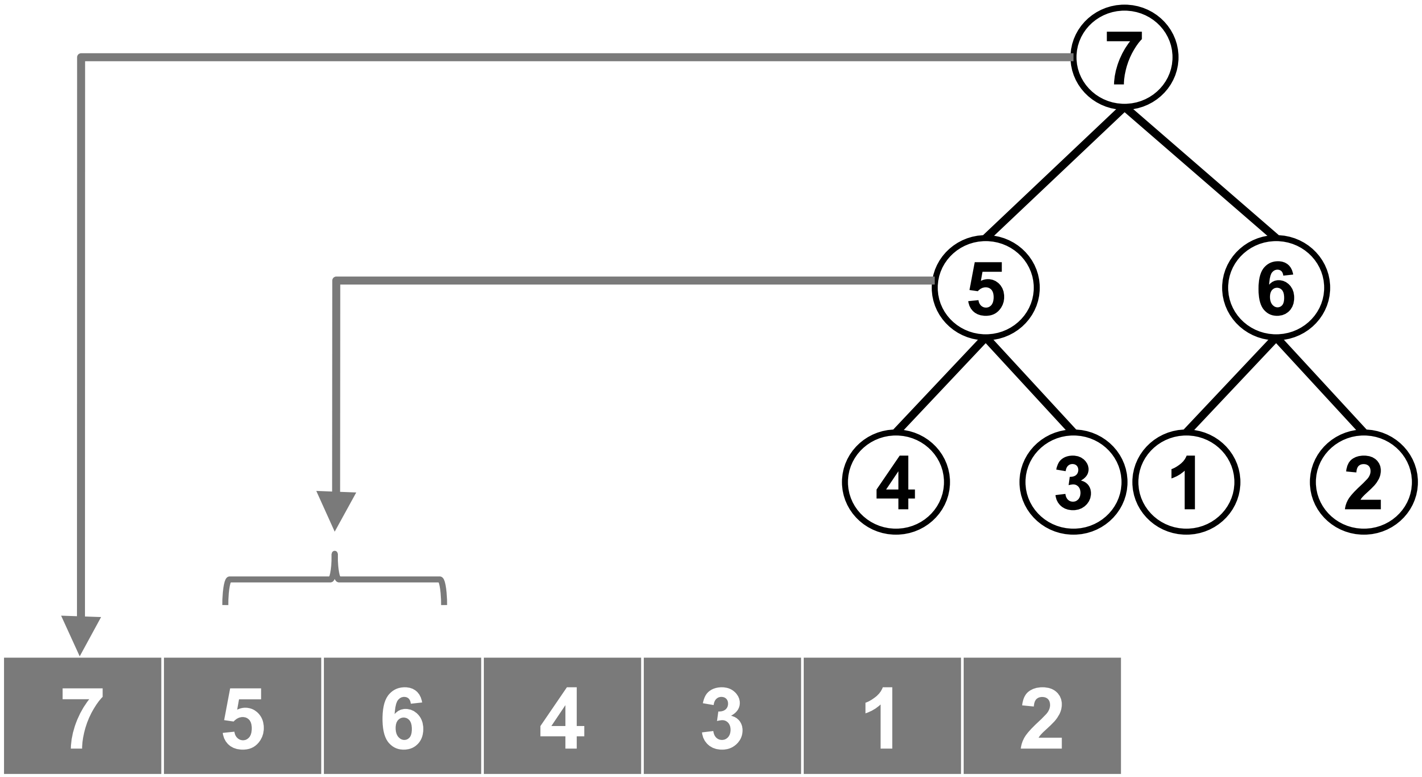
HEAPS AS ARRAYS

A balanced binary tree can be efficiently implemented as an array.

- **The root of the tree in place at index 0**
- **The children of the root are placed as the next two array elements (indices 1 and 2)**
- **Depth 2 elements are added as the next four elements … and so on.**



HEAPS AS ARRAYS



NODES IN THE ARRAY

It is easy to find a node's children or parent in an array implementation:

- **The root is always at index 0**
- **For a non-root node at index i**
  - The parent is at index **(i-1)/2**
- **For any node at index i, its children (if they exist) are:**
  - Left child: Index **(2i +1)**
  - Right child: Index **(2i +2)**

INSERTION

To add a new element to a heap, there are three things to keep in mind:

- **We have to keep the tree complete**
- **We have to re-arrange the elements to preserve the heap property.**
- **Make the smallest number of movements to be more efficient.**

INSERTION

Pseudocode for insertion:

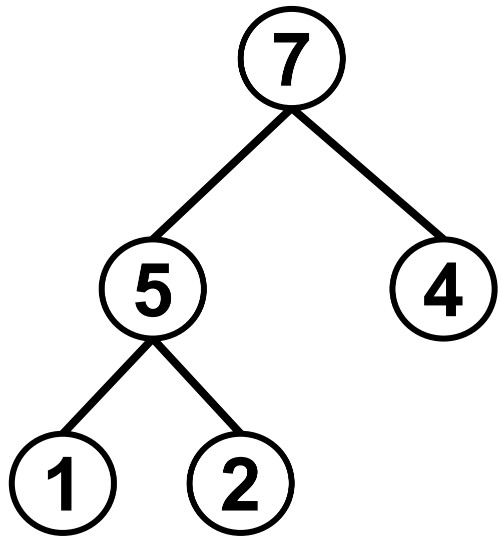
- **Place the new element in the next available location in the array.**
  - This keeps the structure as a complete binary tree but might no longer be a heap.
- **While the new element has a higher priority than its parent**

→ **Swap the element with its parent.**

  - This will stop when the new element reaches the root or when its parent has a higher priority.

INSERTION - DEMO

Insert **6**

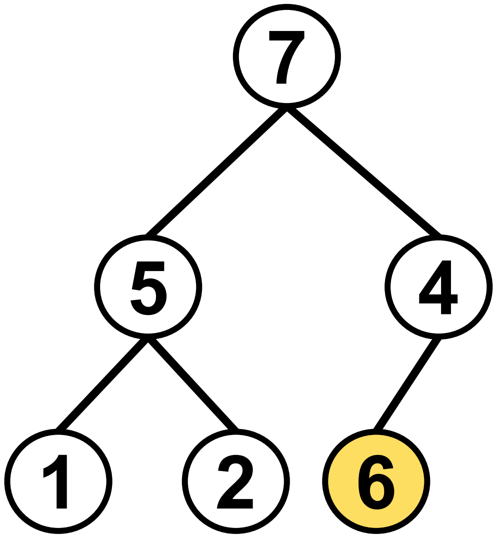


7	5	4	1	2				

INSERTION - DEMO

Insert 6

Next available location



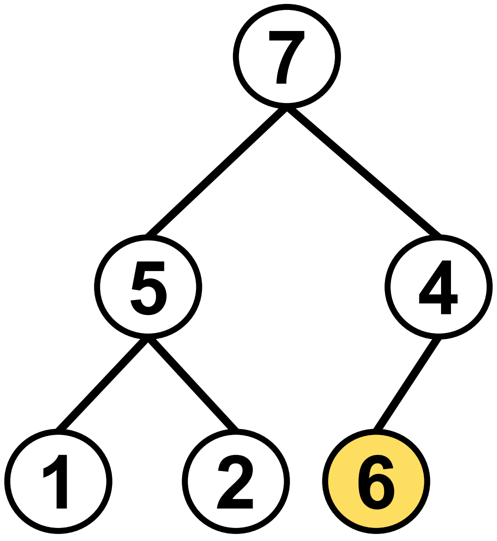
7	5	4	1	2	6			

INSERTION - DEMO

Insert 6

Compare with parent

Parent index = (i-1) /2 =2

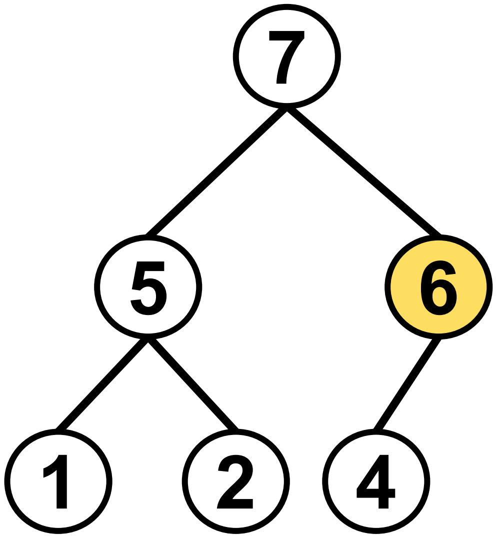


7	5	4	1	2	6			

INSERTION - DEMO

Insert **6**

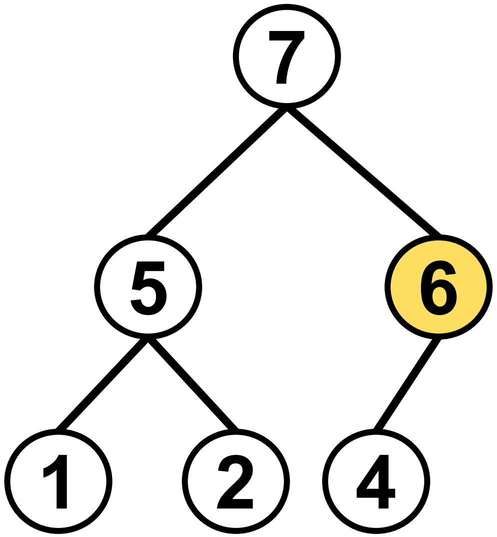
Swap with element at index 2 (parent)



7	5	6	1	2	4			

INSERTION - DEMO

Insert 6



Compare with new parent

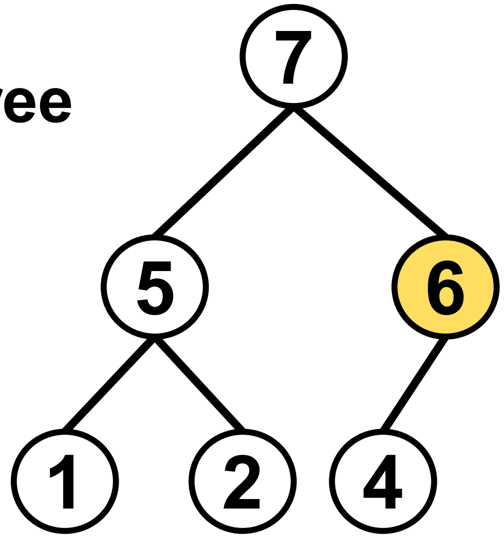
Parent index = (i-1) /2 = 0

7	5	6	1	2	4			

INSERTION - DEMO

The process of moving up the tree

Is called reheapification upward



7	5	6	1	2	4			

DELETION

In a heap, we always delete the root node and return its value. This corresponds to the largest element in a max heap.

- If the root is the only element, we only need to modify the size of the heap
- If there are other elements in the heap, we need to replace the root with another element.
- We have to preserve the heap property by the end of this process.

DELETION

Pseudocode for deletion:

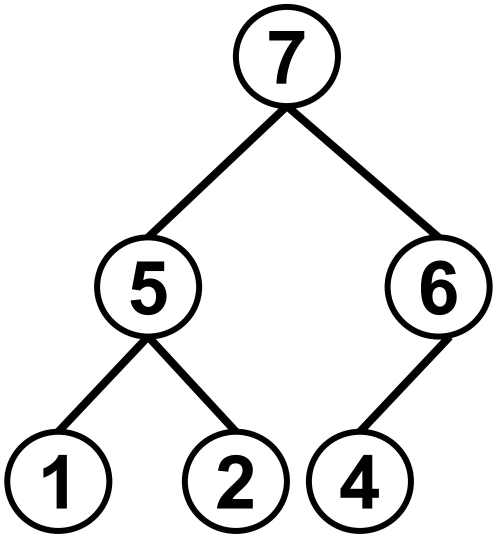
- **Save the value of the root element.**
- **Replace the root with the last element in the heap**
  - This keeps the structure as a complete binary tree but might no longer be a heap.
- **While the new element has a lower priority than one of its children**

→ **Swap the element with its largest child.**

  - This will stop when the new element reaches a leaf or when its priority is at least as high as both of its children.
- **Return the value saved in the first step**

DELETION - DEMO

Delete

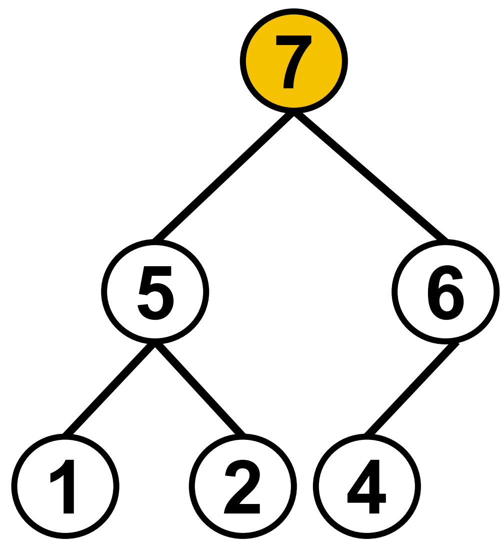


7	5	6	1	2	4			

DELETION - DEMO

Delete

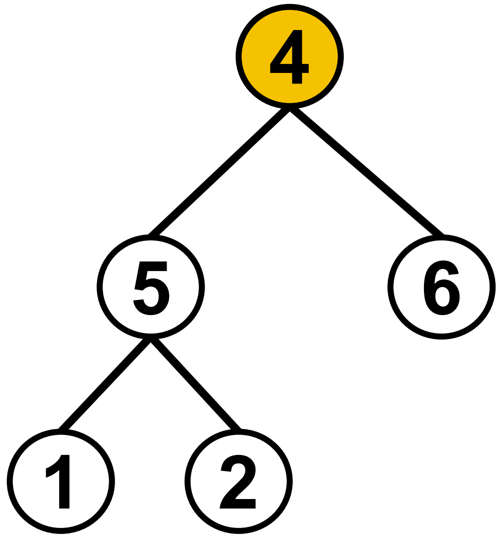
Store the value 7 in temp



7	5	6	1	2	4			

DELETION - DEMO

Delete



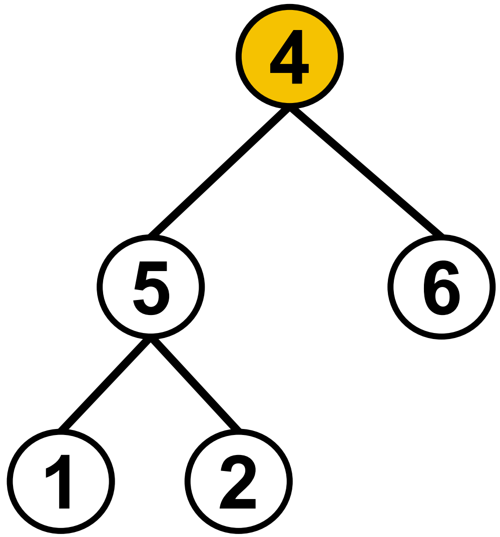
Store the value 7 in temp

Set the root to the last element

4	5	6	1	2				

DELETION - DEMO

Delete



Store the value 7 in temp

Compare with both children

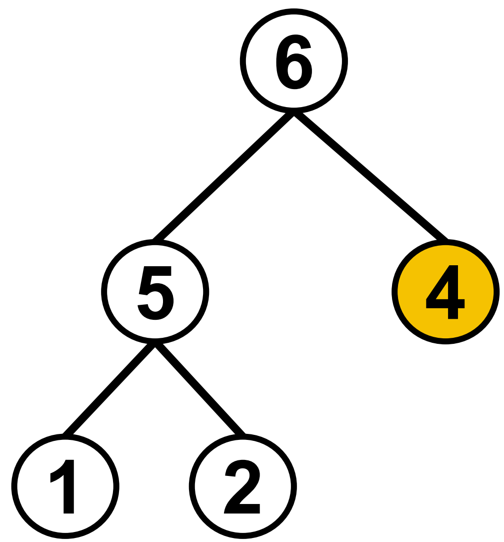
right child index = (2i +1)=1

Left child index = (2i +2)=2

4	5	6	1	2				

DELETION - DEMO

Delete



Store the value 7 in temp

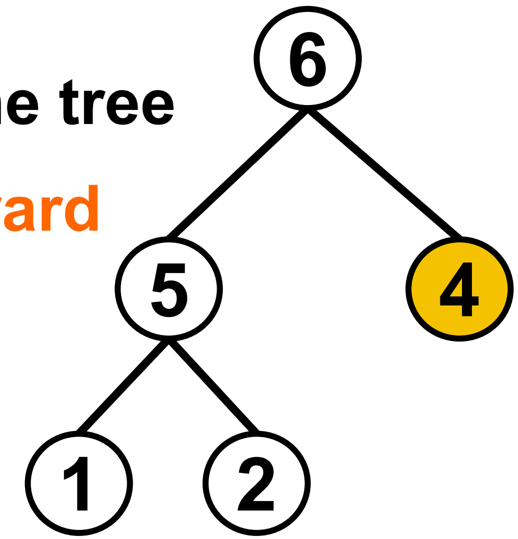
Swap with larger child

6	5	4	1	2				

DELETION - DEMO

The process of moving down the tree

Is called reheapification downward



6	5	4	1	2				