Heaps

Heaps

A **heap** is a certain kind of complete binary tree.

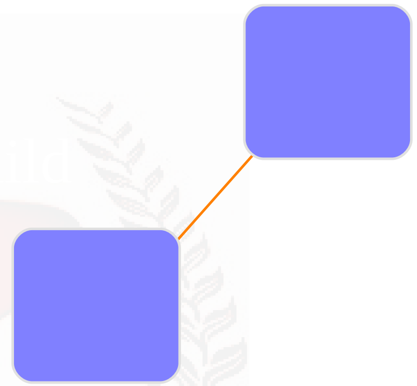
Heaps

A heap is a certain kind of complete binary tree.

When a complete binary tree is built, its first node must be the root.

Heaps

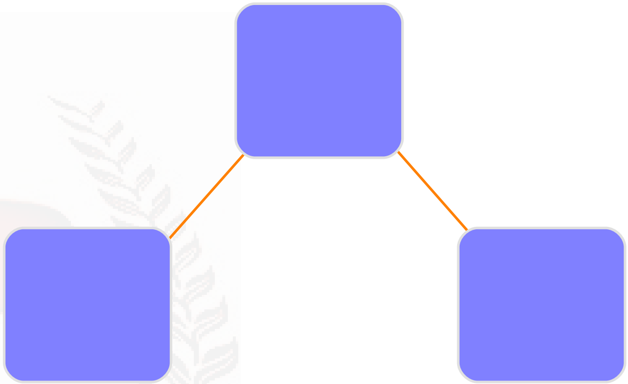
Complete binary tree.



The second node is always the left child of the root.

Heaps

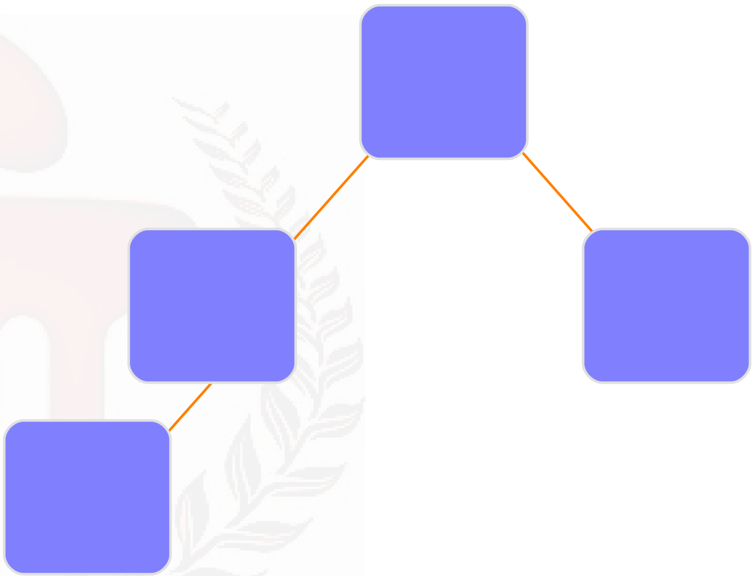
Complete binary tree.



The third node is always the right child of the root.

Heaps

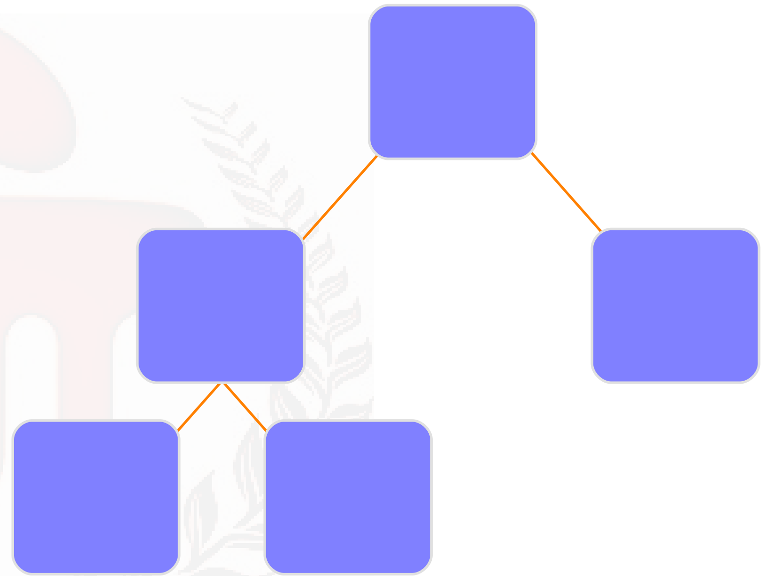
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

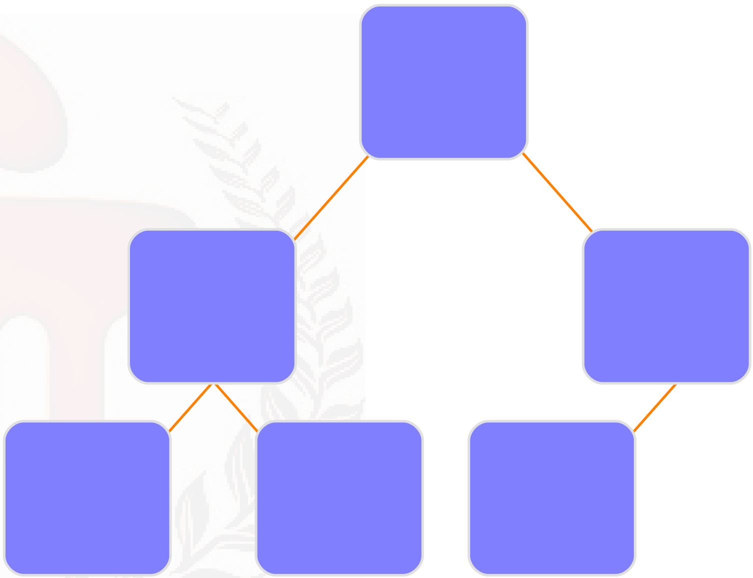
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

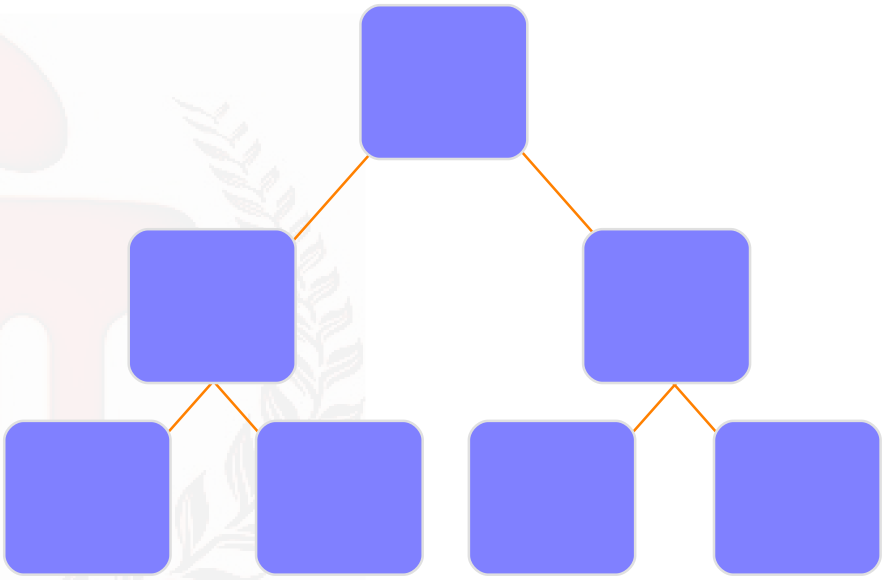
Complete binary tree.



The next nodes always fill the next level from left-to-right.

Heaps

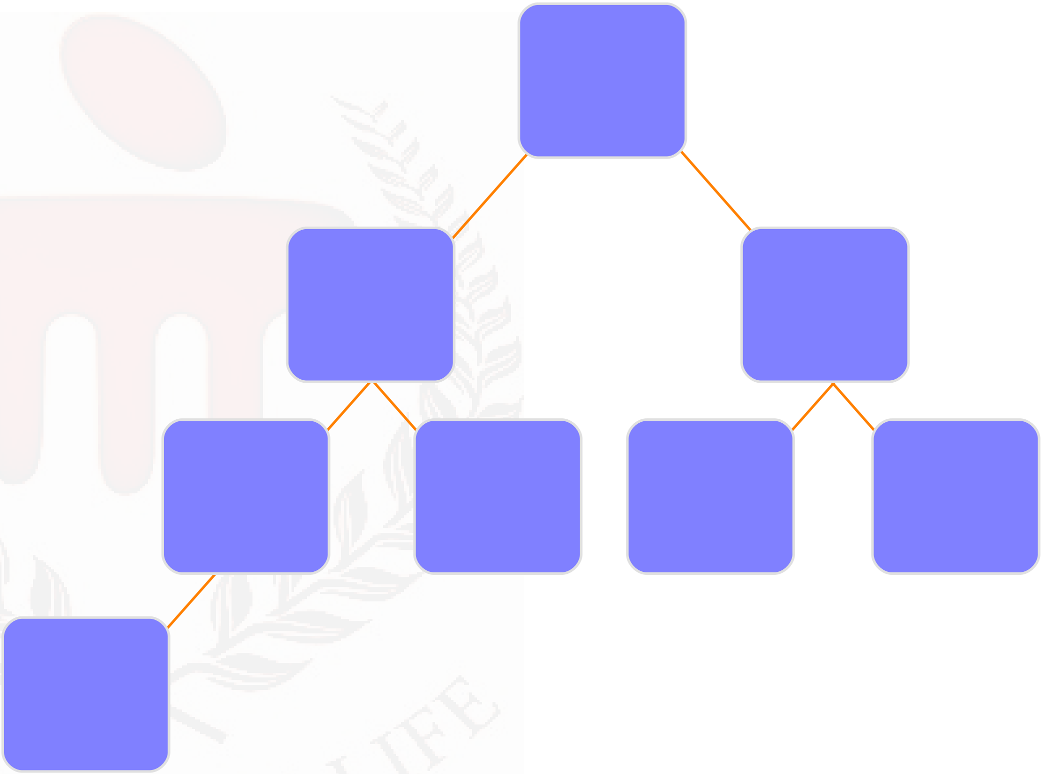
Complete binary tree.



The next nodes always fill the next level from left-to-right.

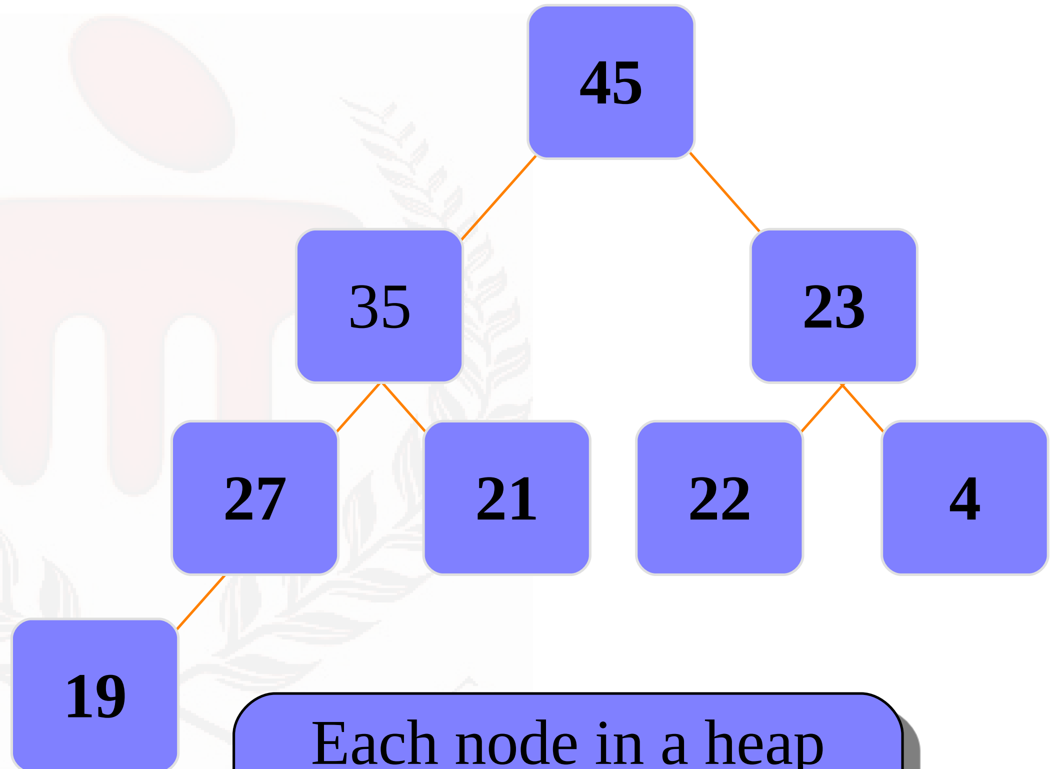
Heaps

Complete binary tree.



Heaps

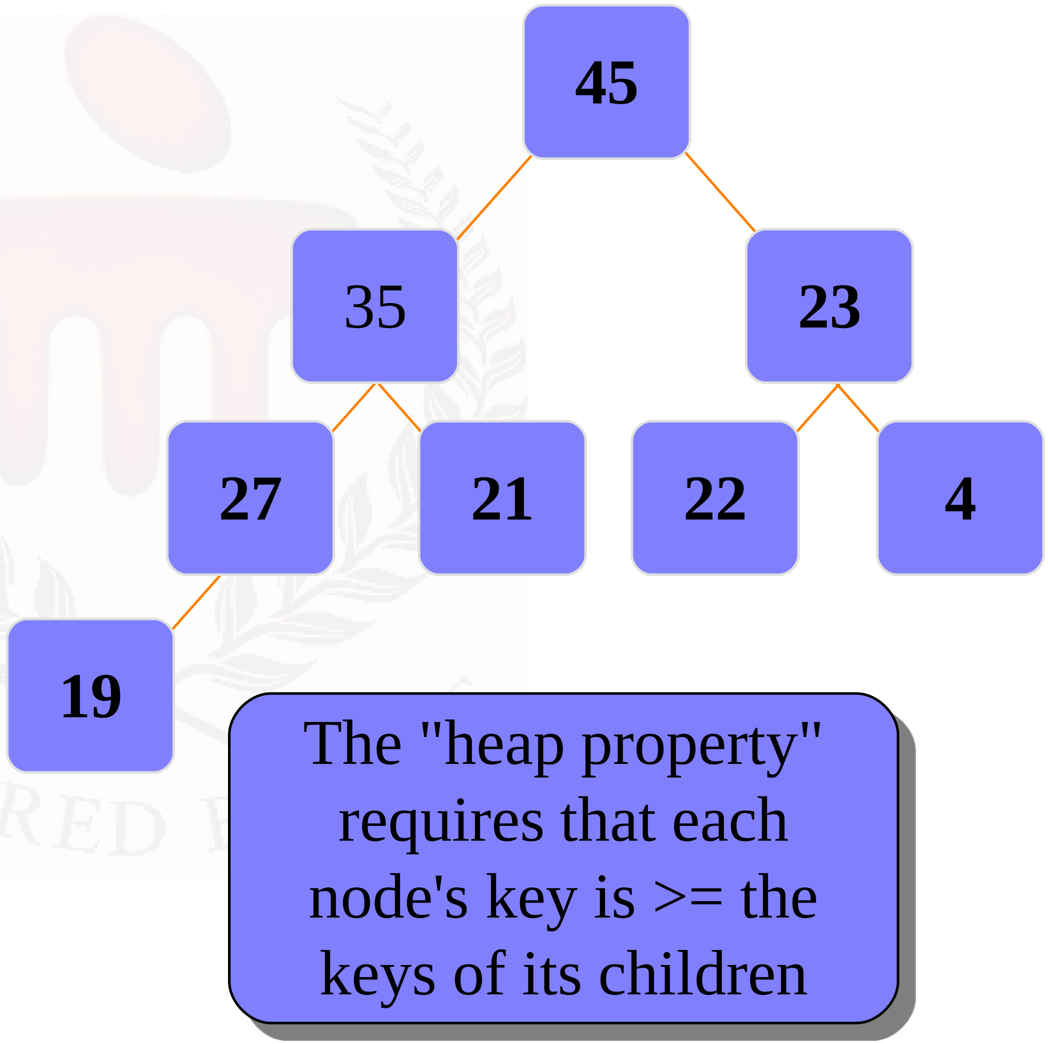
A heap is a **certain** kind of complete binary tree.



Each node in a heap contains a key that can be compared to other nodes' keys.

Heaps

A heap is a **certain** kind of complete binary tree.

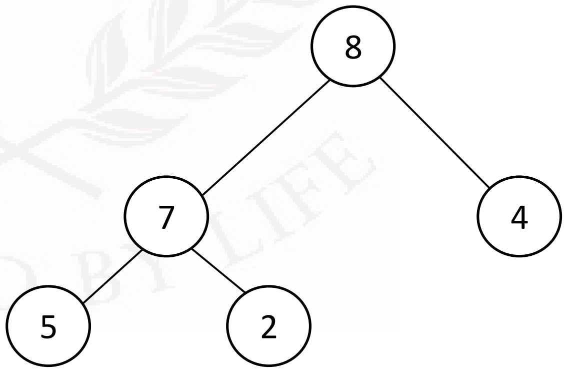


Heaps

- *Def:* A **heap** is a 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

    Parent(x) ≥ x(max heap) , Parent(x)<=x(Min heap)

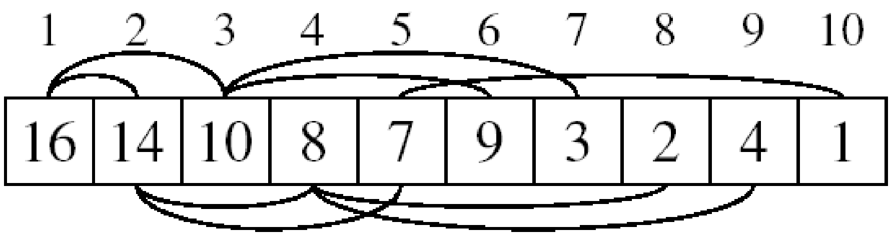
Heap

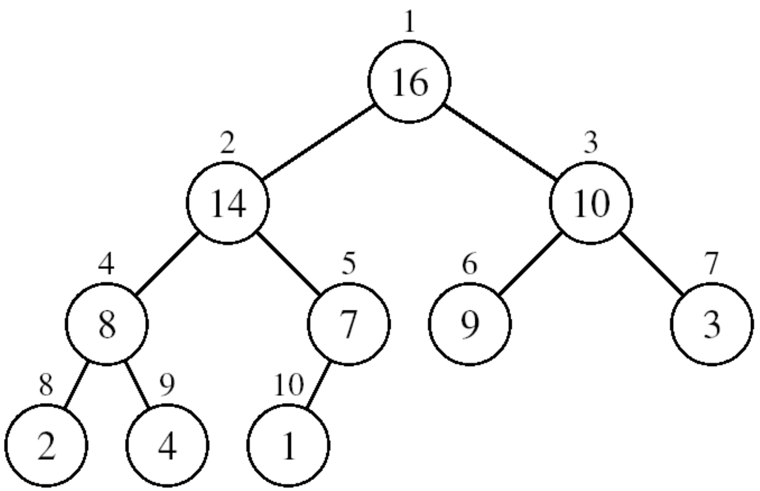


A heap is a binary tree that is filled in order

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

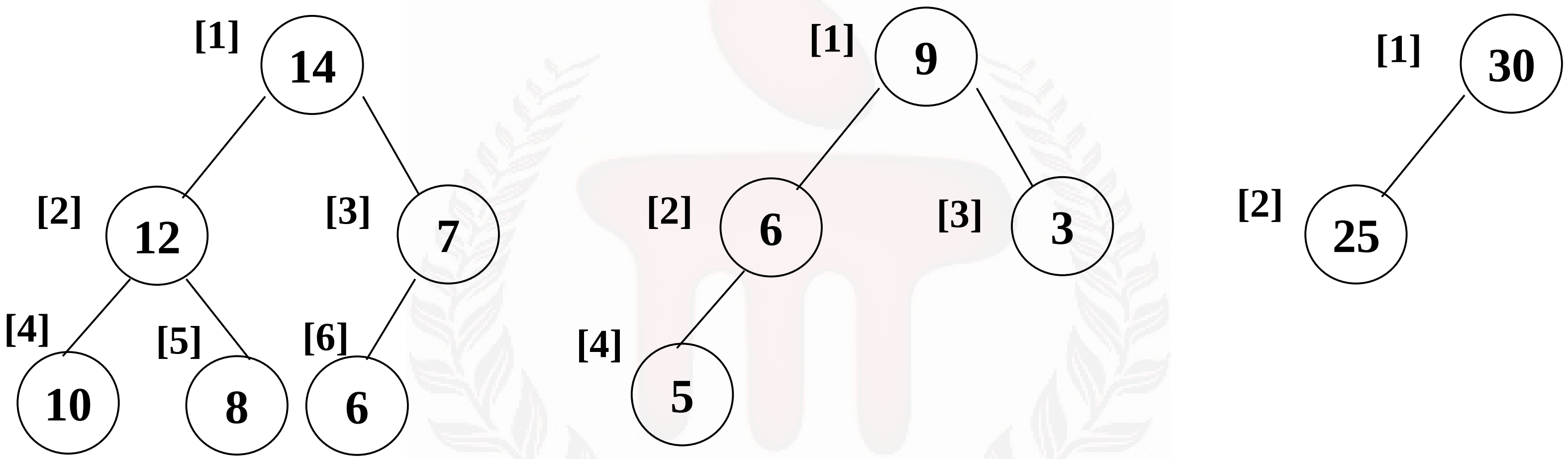




Heaps

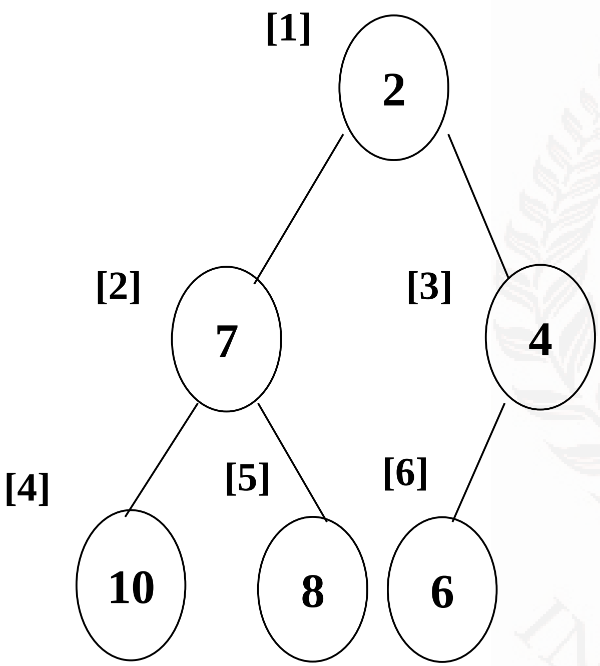
- A *max tree* is a tree in which the key value in each node is greater than(equal to) the key values in its children. A *max heap* is a complete binary tree that is also a max tree.

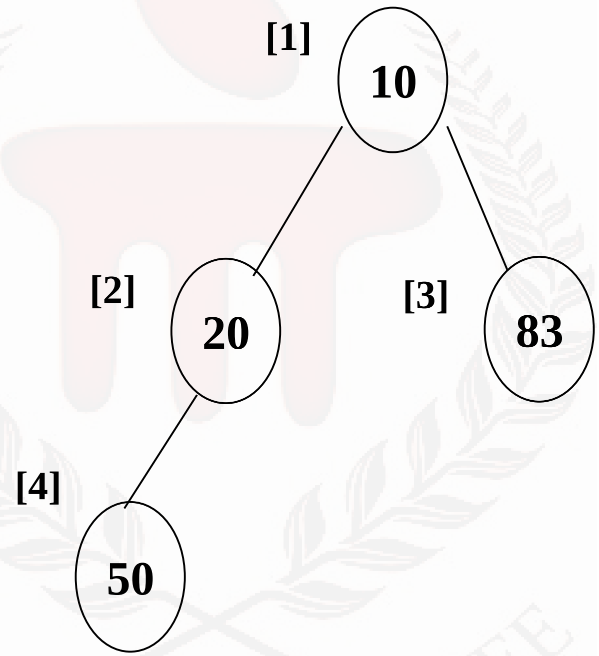
- A *min tree* is a tree in which the key value in each node is smaller than(equal to) the key values in its children. A *min heap* is a complete binary tree that is also a min tree.

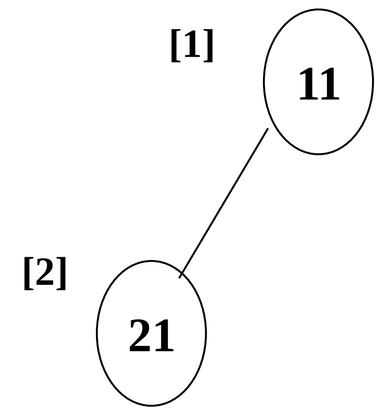


Property:

The root of max heap (min heap) contains the largest (smallest).


[1]
2
[2]
7
[3]
4
[4]
10
[5]
8
[6]
6


[1]
10
[2]
20
[3]
83
[4]
50

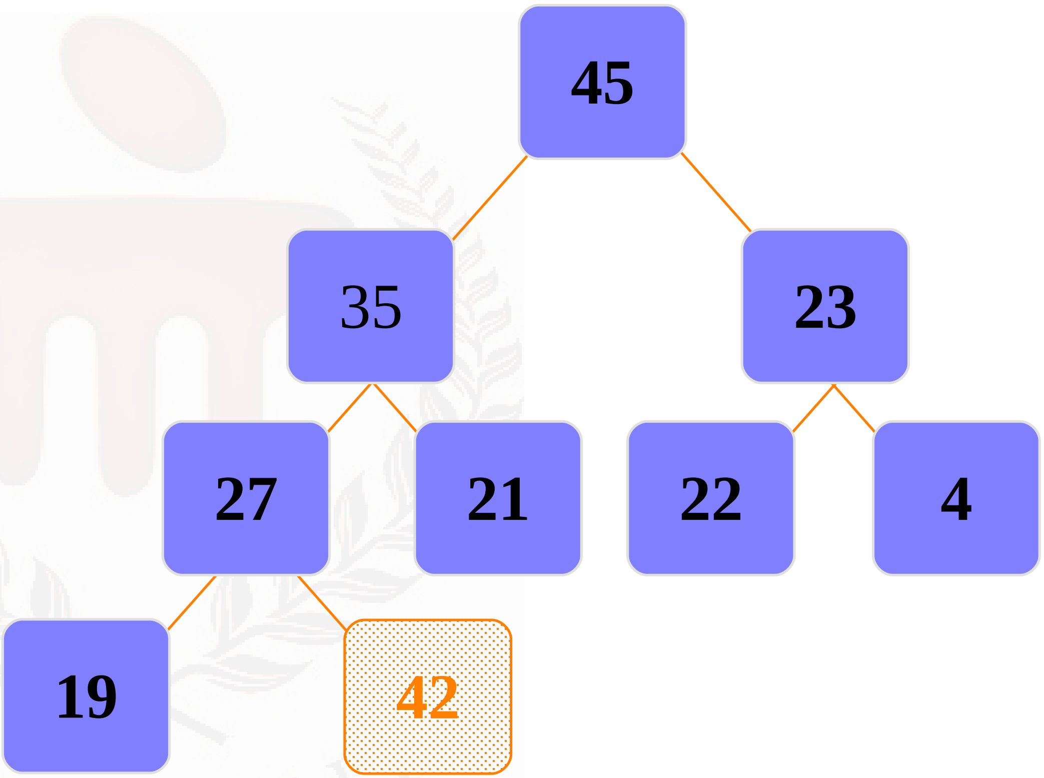

[1]
11
[2]
21

Steps to construct max heap

- 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.

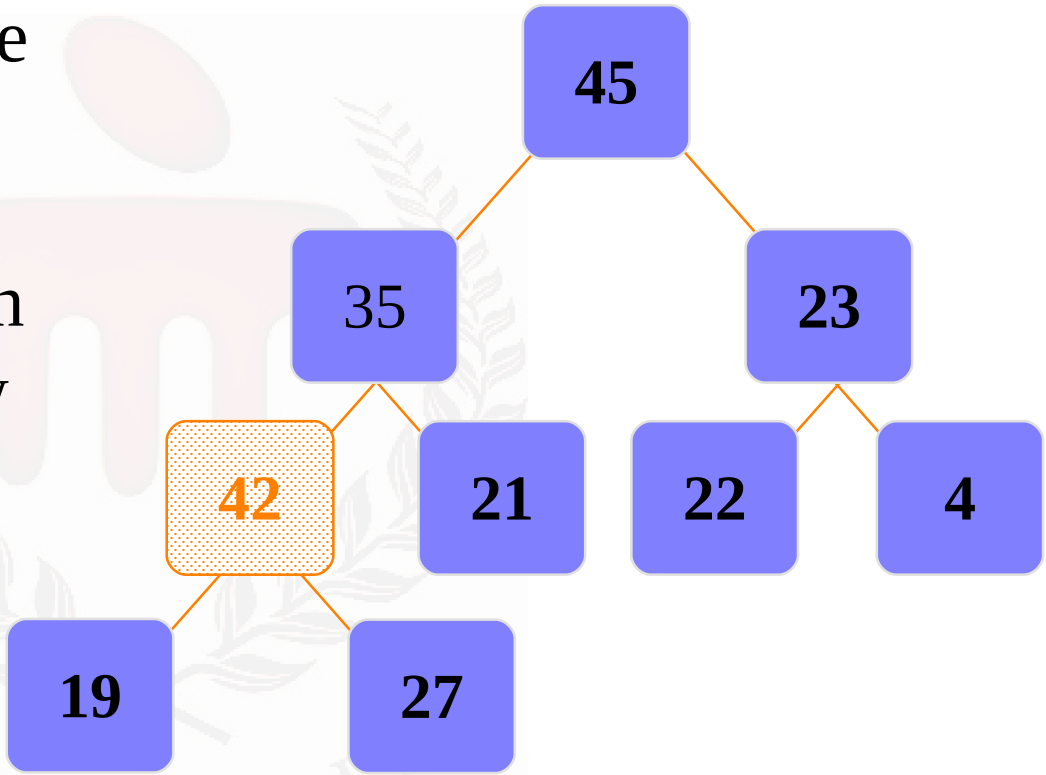
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



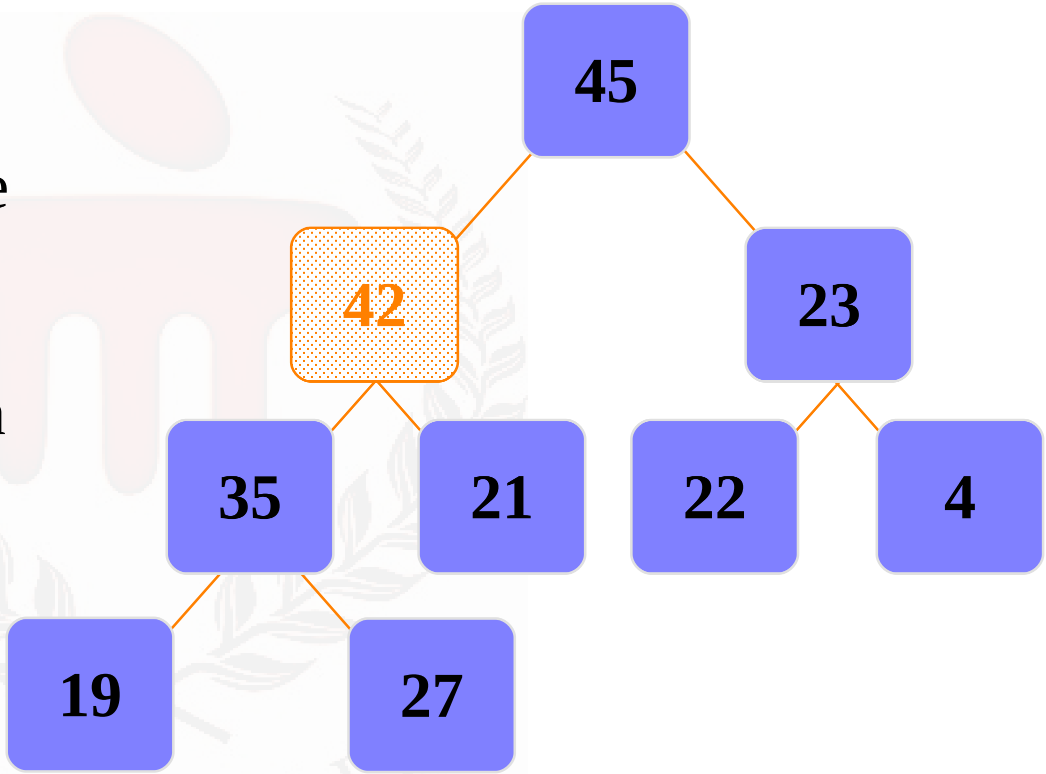
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



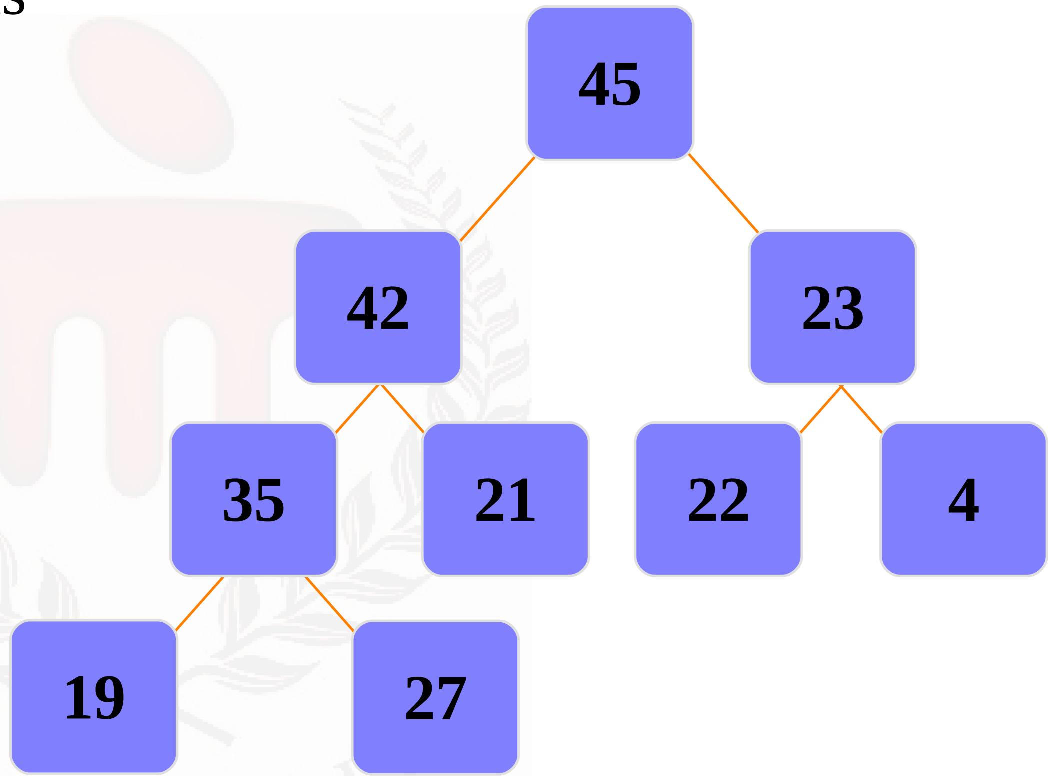
Adding a Node to a Heap

- Put the new node in the next available spot.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



Adding a Node to a Heap

- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.

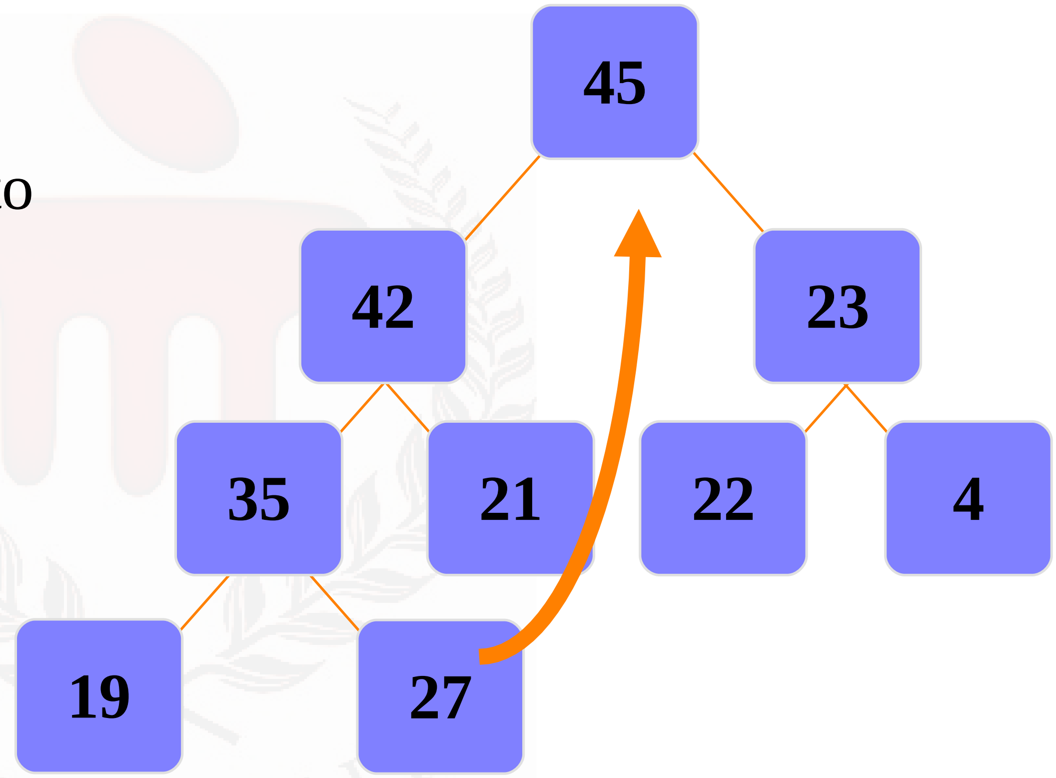


Deletion algorithm-Max - heap

- 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.

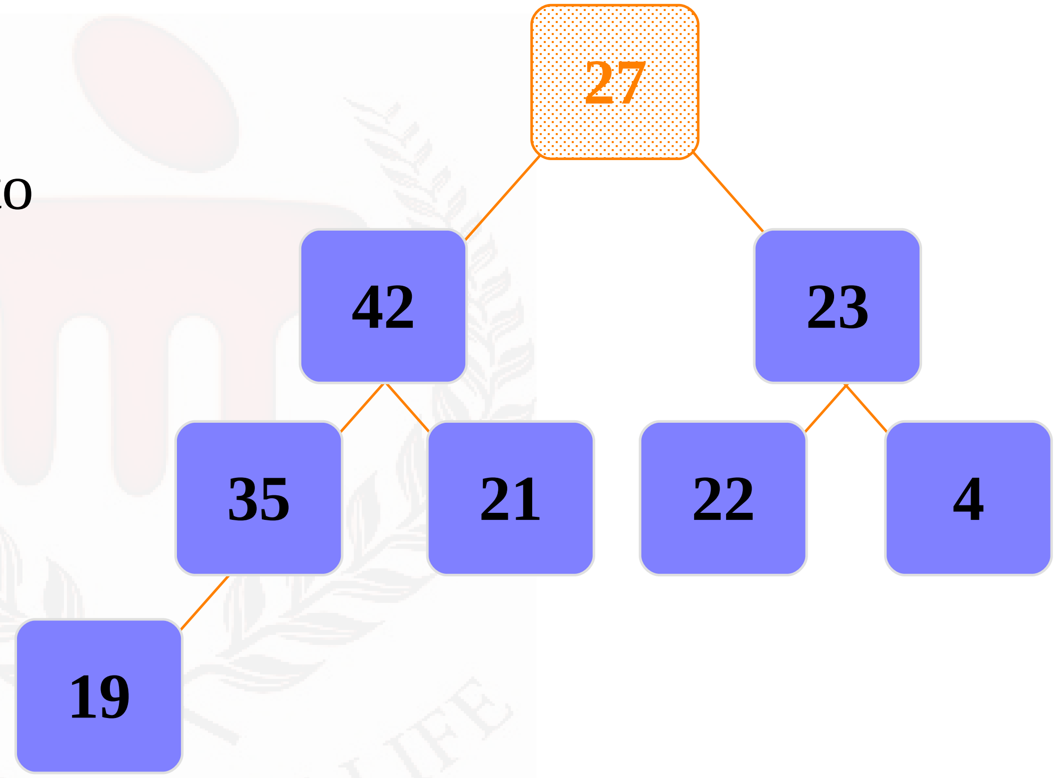
Removing the Top of a Heap

- Move the last node onto the root.



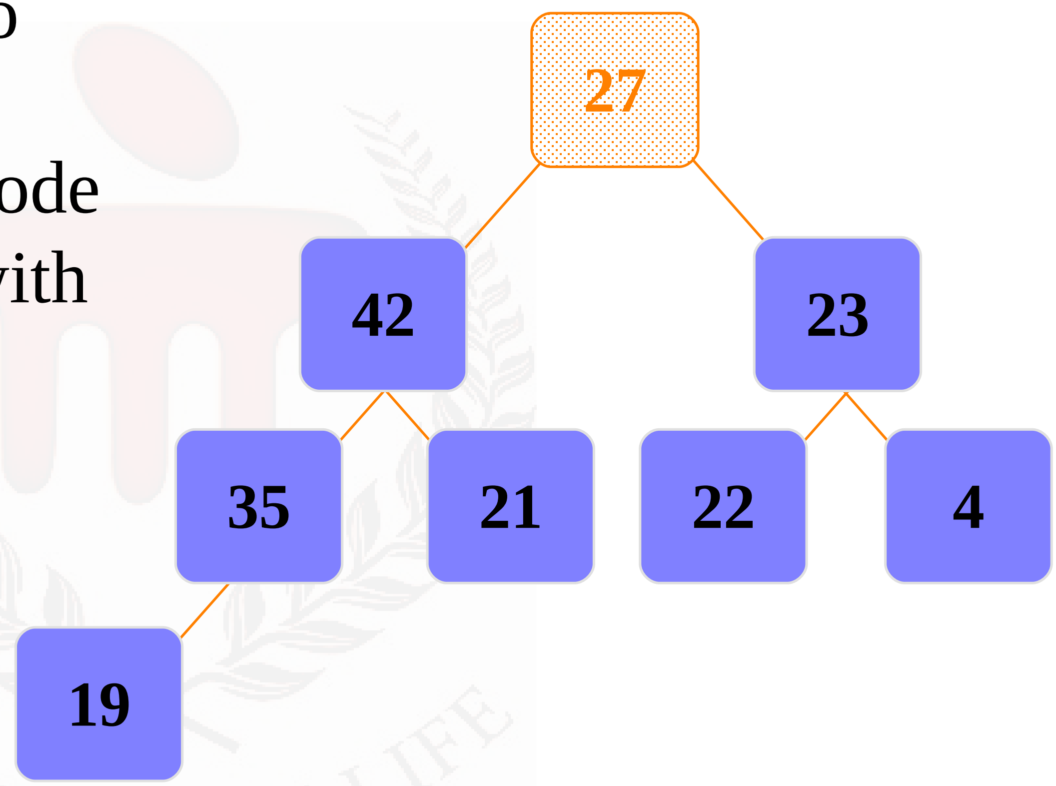
Removing the Top of a Heap

- Move the last node onto the root.



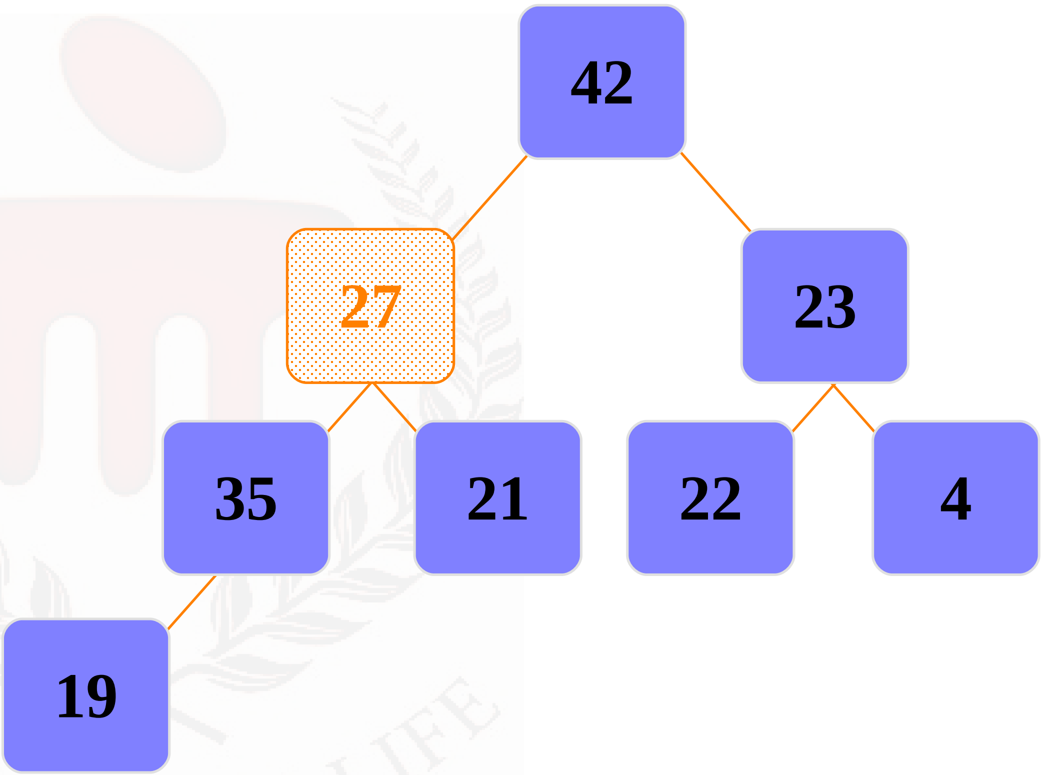
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



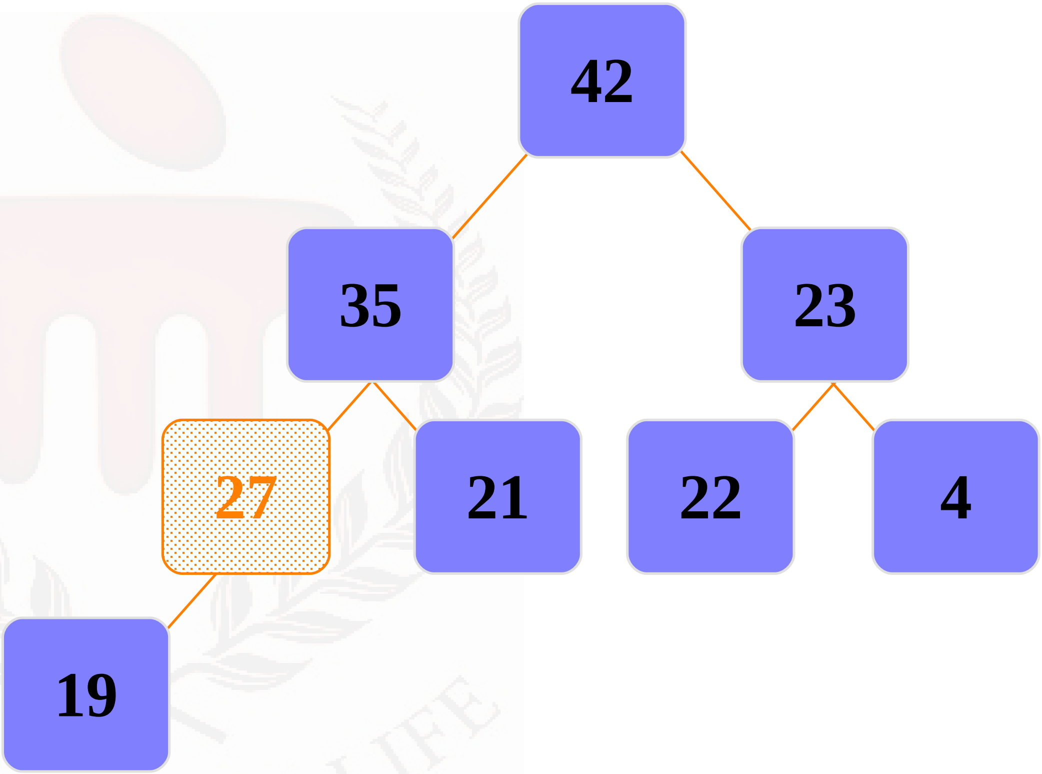
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



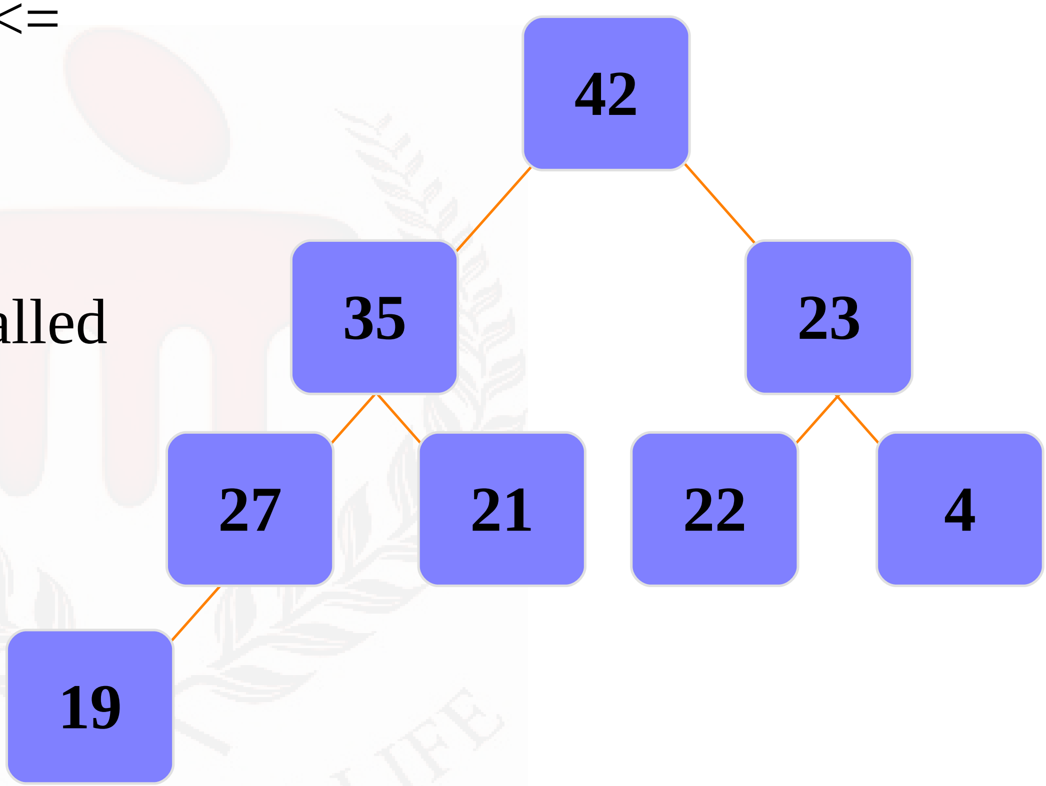
Removing the Top of a Heap

- Move the last node onto the root.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.



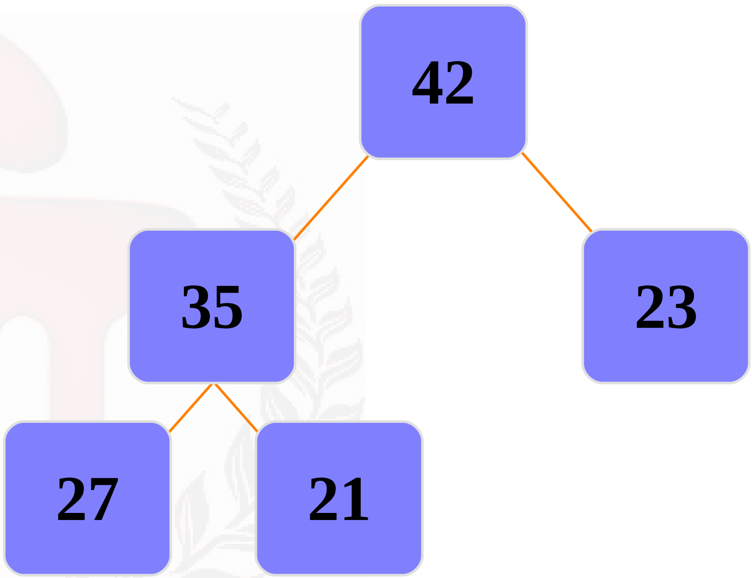
Removing the Top of a Heap

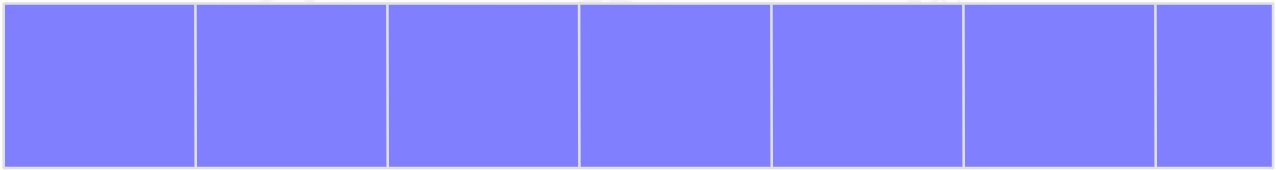
- The children all have keys <= the out-of-place node, or
- The node reaches the leaf.
- The process of pushing the new node downward is called **reheapification downward**.



Implementing a Heap

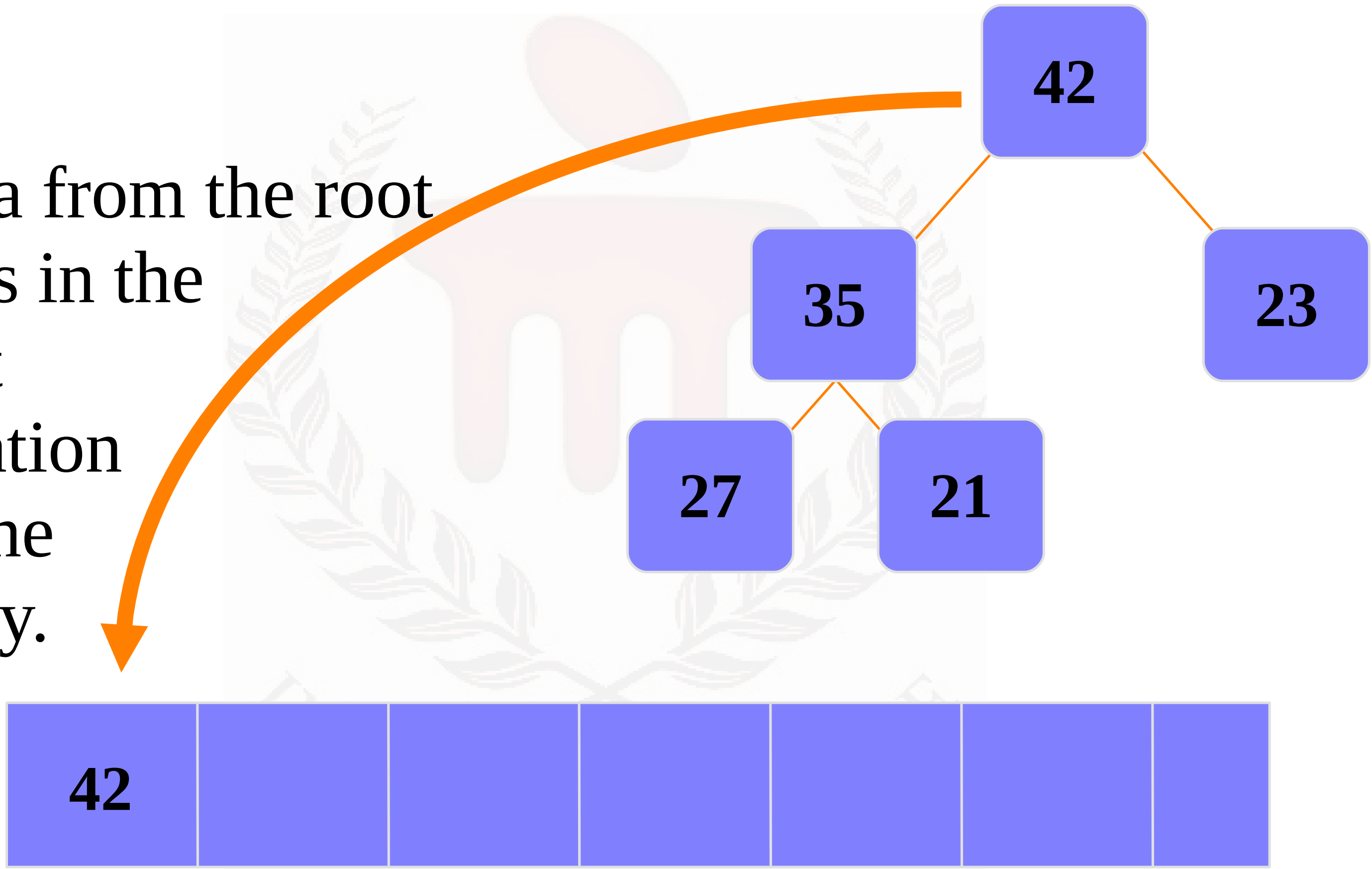
- We will store the data from the nodes in a partially-filled array.





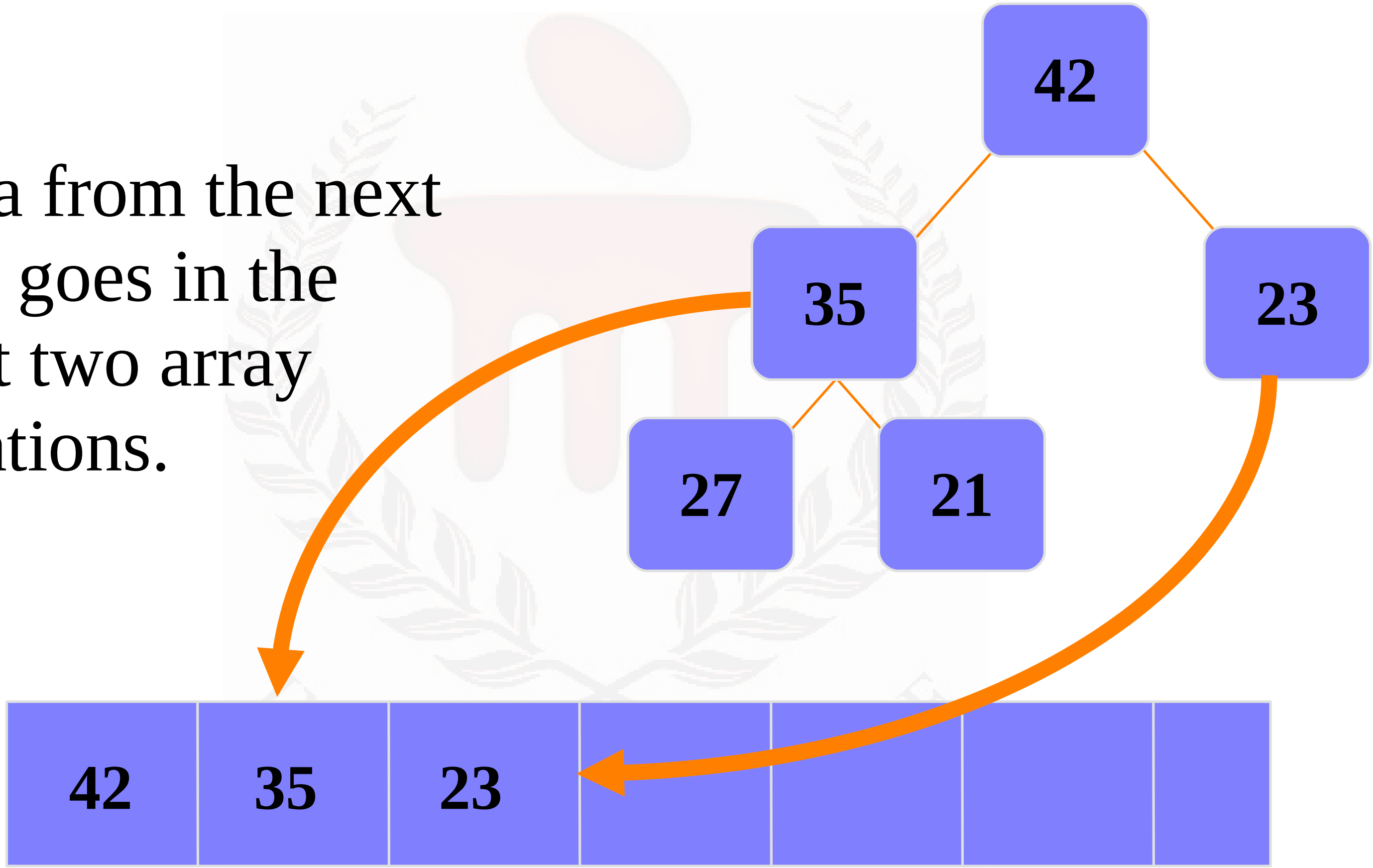
Implementing a Heap

- Data from the root goes in the first location of the array.



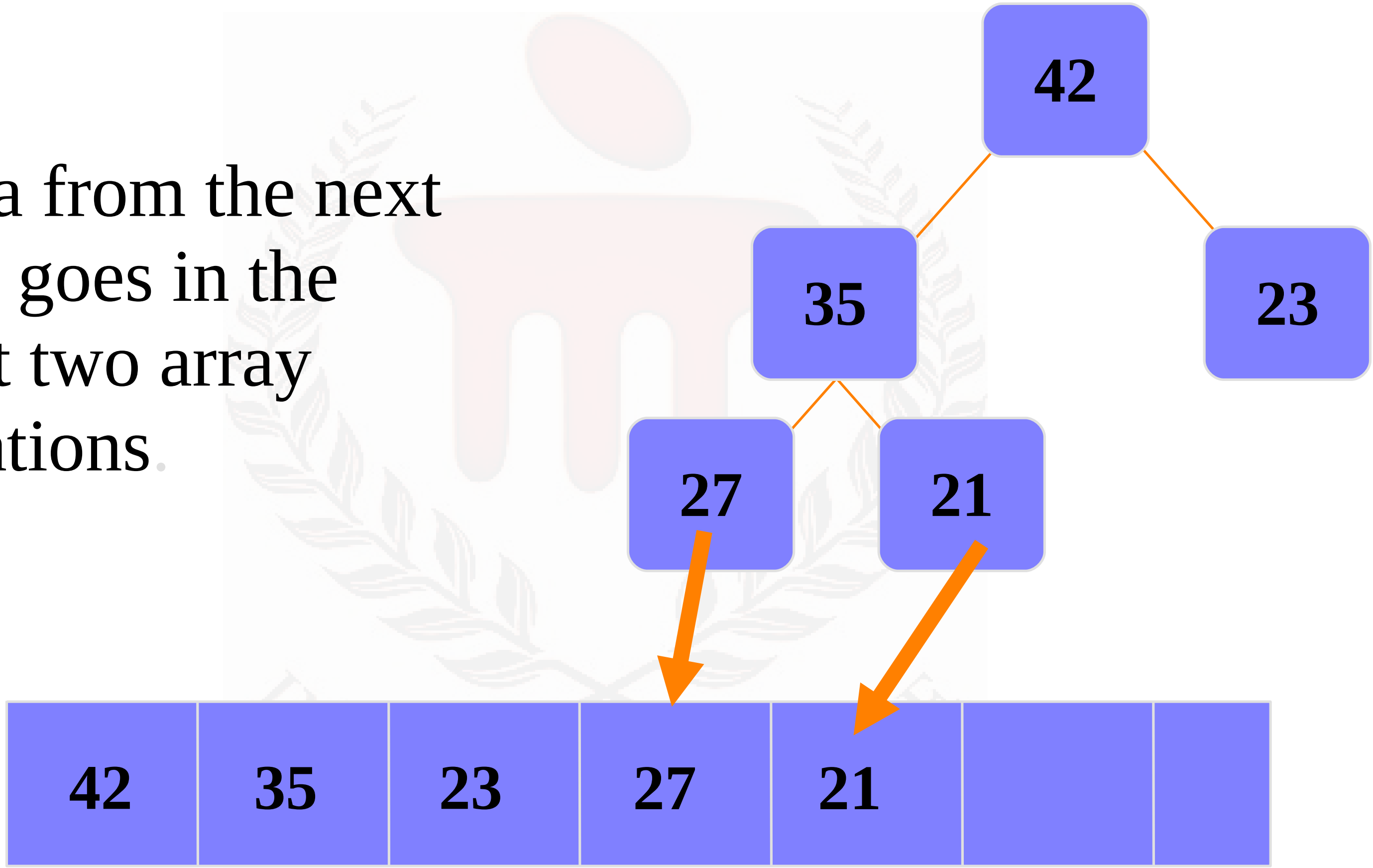
Implementing a Heap

- Data from the next row goes in the next two array locations.



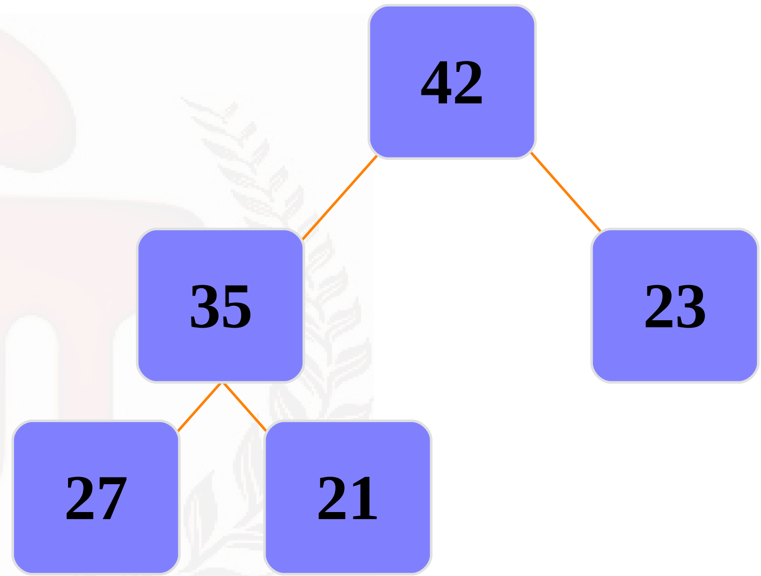
Implementing a Heap

- Data from the next row goes in the next two array locations.



Implementing a Heap

- Data from the next row goes in the next two array locations.

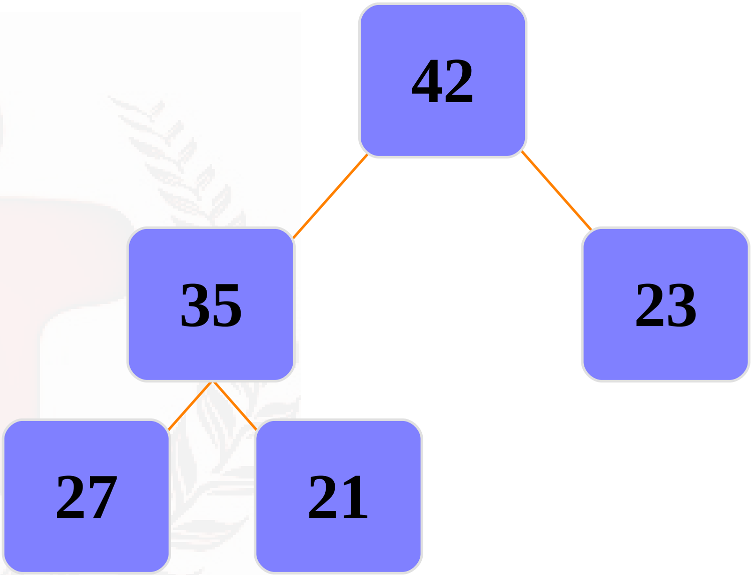


42	35	23	27	21		

We don't care what's in this part of the array.

Important Points about the Implementation

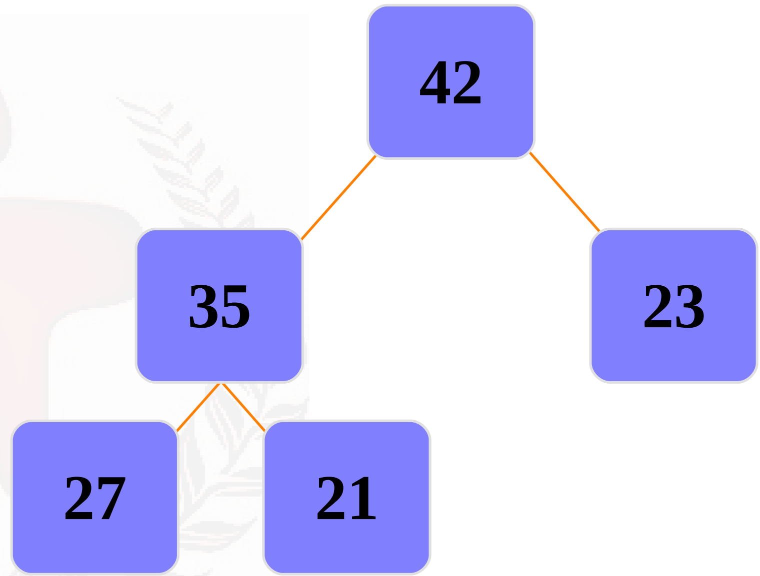
- The links between the tree's nodes are not actually stored as pointers, or in any other way.
- The only way we "know" that "the array is a tree" is from the way we manipulate the data.



42	35	23	27	21		

Important Points about the Implementation

- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.



42	35	23	27	21		

Summary

- A heap is a complete binary tree, where the entry at each node is greater than or equal to the entries in its children.
- To add an entry to a heap, place the new entry at the next available spot, and perform a reheapification upward.
- To remove the biggest entry, move the last node onto the root, and perform a reheapification downward.

Suppose the Heap is a Max-Heap as: *Element to be inserted is 15.*

```
        10
      /    \
   5         3
  / \
 2   4
```

Step 1: Insert the new element at the end.

```
       10
      /    \
    5       3
   / \     /
  2   4   15
```

Step 2: Heapify the new element following bottom-up approach.

-> 15 is more than its parent 3, swap them.

```
       10
      /    \
    5       15
   / \     /
  2   4   3
```

-> 15 is again more than its parent 10, swap them.

```
       15
      /    \
    5       10
   / \     /
  2   4   3
```

Therefore, the final heap after insertion is:

```
       15
      /    \
    5       10
   / \     /
  2   4   3
```

Add the following values one at a time to an initially empty binary search tree using the simple algorithm:

90 20 9 98 10 28 -25

What is the resulting tree?