Recursive Data Structures: Trees

-- Binary Heaps --

University of Virginia
CS 2110
Prof. N. Basit

Announcements

- Homework #6 – *Binary Search Trees*
  - Due: by 11:30pm on Tuesday, April 28, 2020
  - Remember to submit your **JUnit** tests
  - Web-CAT: 100% auto graded (score out of **100**)
- Final Exam Review
  - On Monday, April 27 – *bring questions to the live stream!*
- Final Exam
  - On Saturday, May 2, 2020
  - *Similar format to Exam 2, however specific details will be provided during our Final Exam Review session*

Scenario:
Hospital Waiting Room

Efficient *patient registration*

AND

Efficient *removal (to see a Dr.) based on priority level?*

How can we achieve BOTH??

Heaps (“binary heaps”)

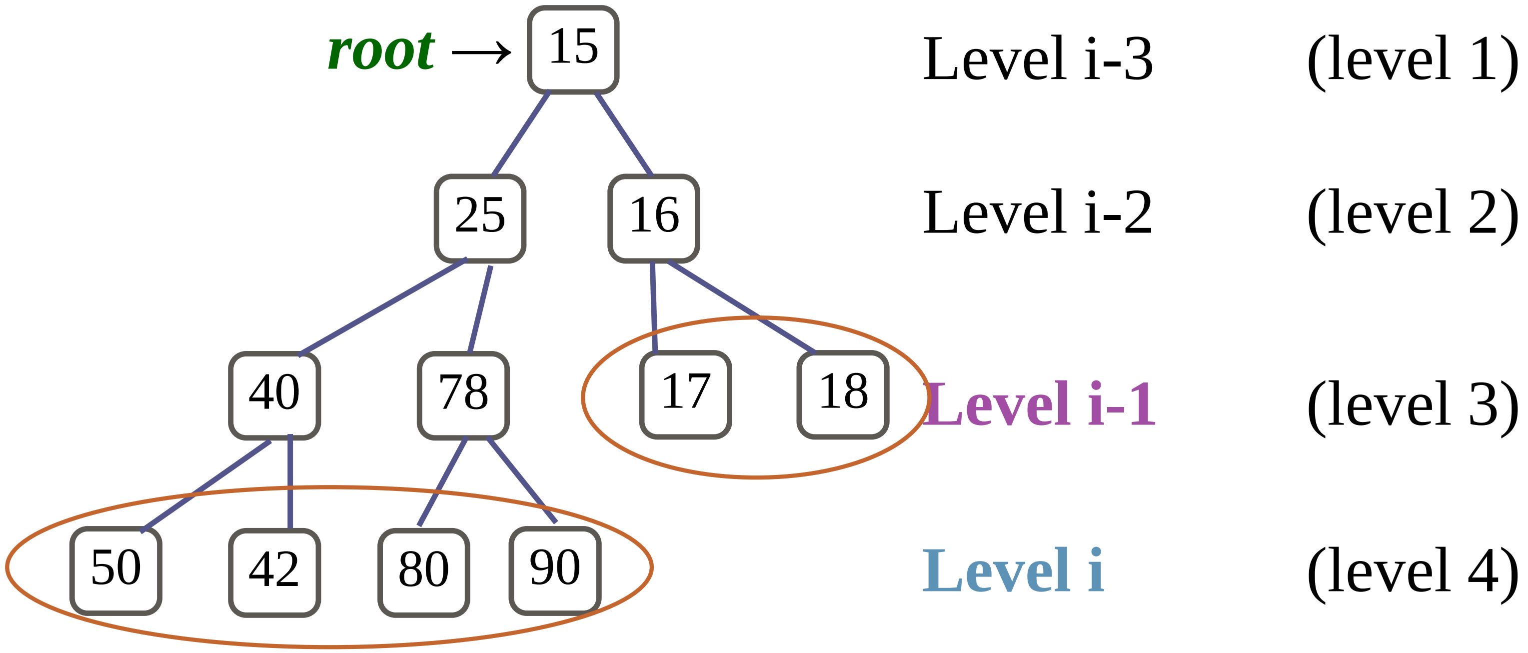
- The **heap** data structure is an example of a ***balanced binary tree***
- Useful in solving three types of problems:
  - Finding the **min or max** value within a collection
  - **Sorting** numerical values into ascending or descending order
  - Implementing another important data structure called a **priority queue**

Heaps (“binary heaps”)

- A **binary heap** is a heap data structure created using a binary tree
- It can be seen as a binary tree *with* ***two*** *additional constraints*:
- **Shape property:**
  - A heap is a **complete binary tree**, a binary tree of height (i) in which all leaf nodes are located on level (i) or level (i-1), and all the leaves on level (i) are as far to the left as possible
- **Order (heap) property:**
  - The data value stored in a node is **less than or equal to** the data values stored in all of that node’s descendants
  - (Value stored in the root is always the **smallest** value in the heap)

Leaf nodes on level (i) or level (i-1)?

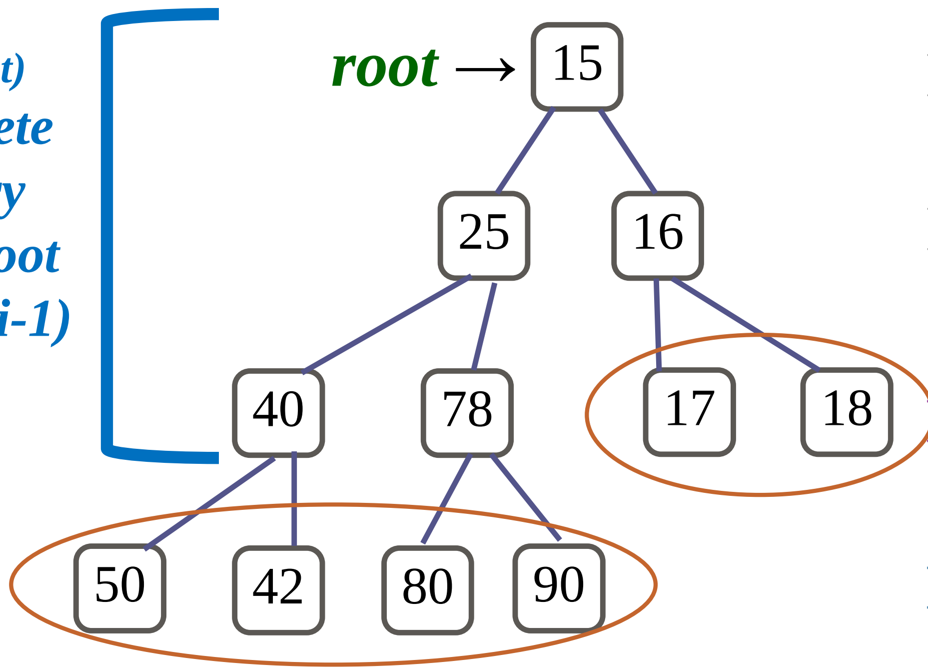
- Notice that all leaves are located on **level (i)** or **level (i-1)**
- Where **level (i)** is the *furthest away* from the **root**



Leaf nodes on level (i) or level (i-1)?

- Notice that all leaves are located on **level (i)** or **level (i-1)**
- Where **level (i)** is the *furthest away* from the **root**

(at least) Complete Binary Tree (root to level i-1) →



Level i-3	(level 1)
Level i-2	(level 2)
Level i-1	(level 3)
Level i	(level 4)

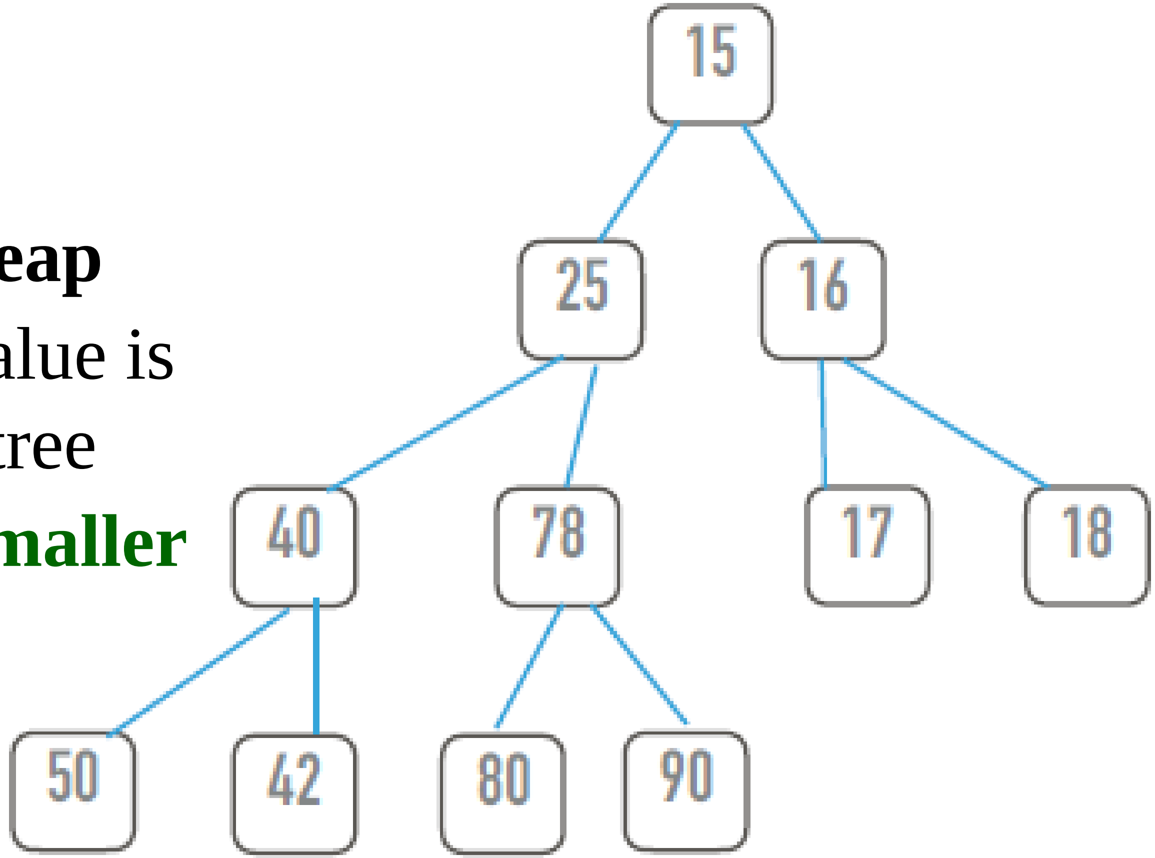
Minheap vs Maxheap

- We could just as easily define a heap in which a node’s value is ***greater than or equal to*** the data values stored in all of that node’s descendants.

- In this case, all algorithms would simply change the < operator to a >, and every occurrence of the word smallest would be replaced by largest.

Minheap

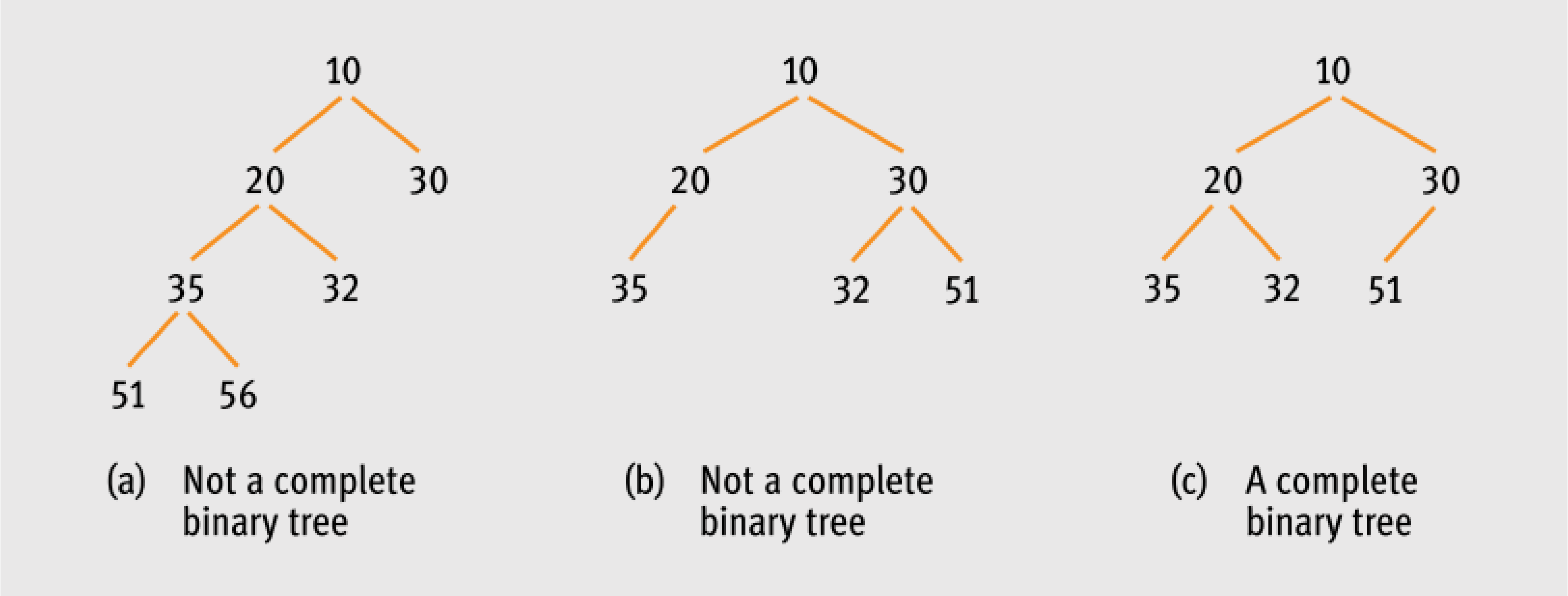
- This is a **min heap**
- The **smallest** value is the **root** of the tree
- All nodes are **smaller** than ALL its descendants



- Note: a heap is NOT a binary search tree – values larger than the root can appear on either side as children

Complete Binary Tree

Which of these trees is a complete binary tree?



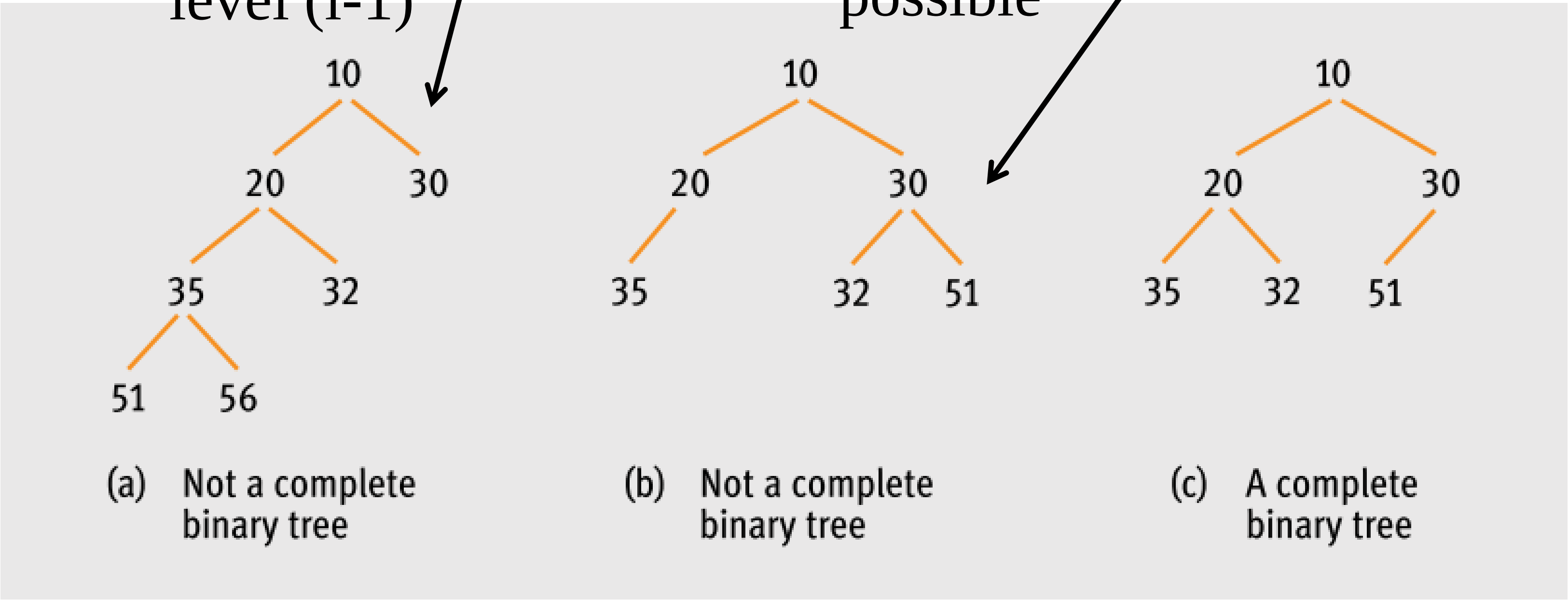
[FIGURE 7-29] Examples of valid and invalid complete binary trees

(complete except for the 'last' level)

Why First Two Are Invalid?

All leaf nodes are NOT located on level (i) or level (i-1)

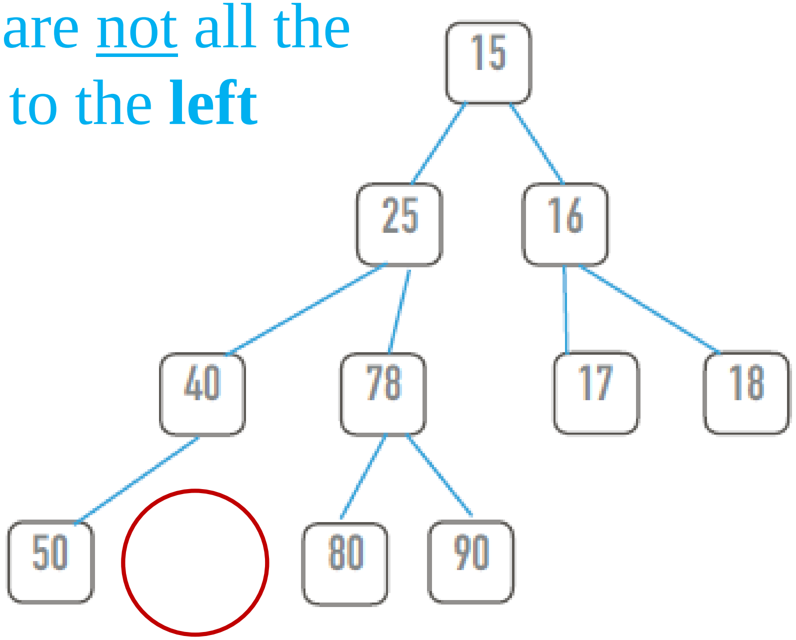
Leaves on level i are NOT as far to the left as possible

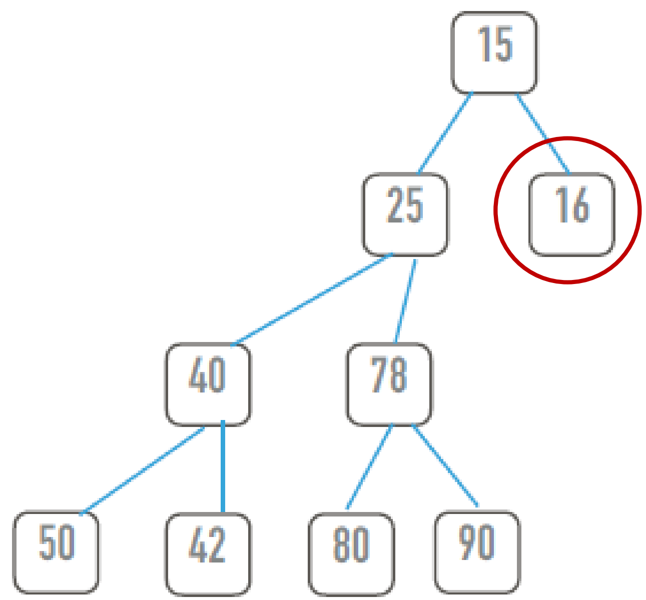


[FIGURE 7-29] Examples of valid and invalid complete binary trees

Examples of Invalid Heaps

Nodes on the bottom row are <u>not</u> all the way to the **left**





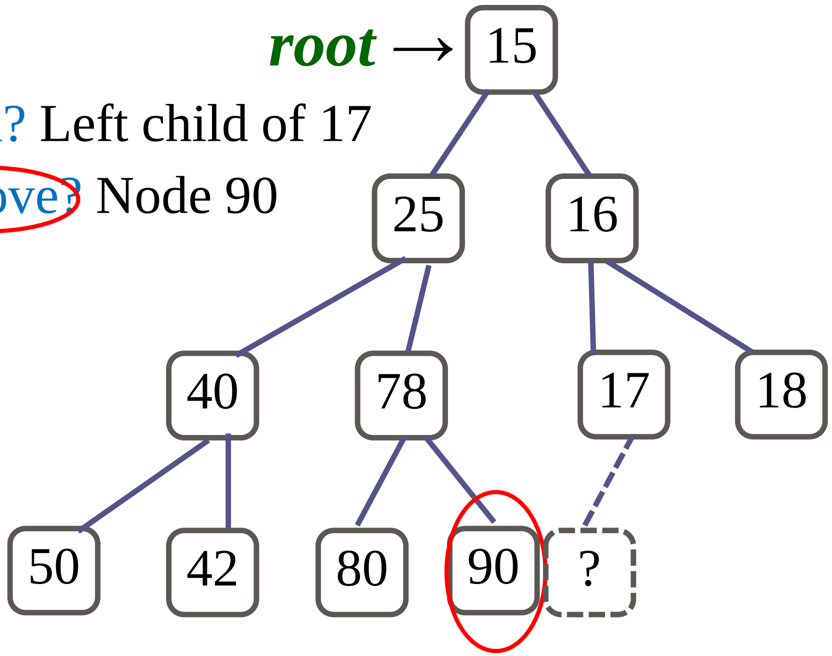
The right leaf is **not balanced**
(Leaf nodes appear at an *inappropriate* level – not level (i) or (i-1))

Implementatation

- Heap must be a *complete* tree

- all leaves are on the *lowest two levels*
- **nodes are <u>added</u> on the *lowest level, from left to right***
- **nodes are <u>removed</u>** (to replace the root) ***from the lowest level, from right to left***

Where are nodes added or removed?



- Where to add? Left child of 17
- What to remove? Node 90

- Nodes added: → → → → *from left to right (no gaps)* → → → →
- Nodes removed: ← ← ← ← *from right to left (no gaps)* ← ← ← ←

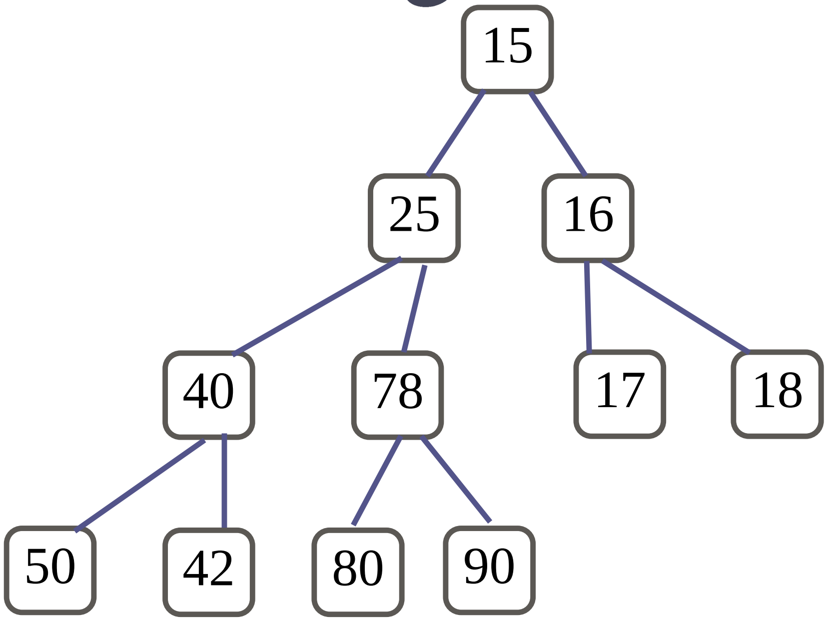
Binary Heap

- The two most important mutator methods on heaps are:
- (1) **inserting** a new value into the heap and
- (2) **retrieving the smallest** value from the heap (in other words, *removing the root*).
- The `insertHeapNode()` method adds a new data value to the heap. It must ensure that the insertion maintains both the **order** and **shape** properties of the heap. The retrieval method, `getSmallest()`, removes and *returns the smallest value* in the heap, which must be the value stored in the **root**. This method also **rebuilds the heap** because it removes the root, and all nonempty trees must have a root by definition

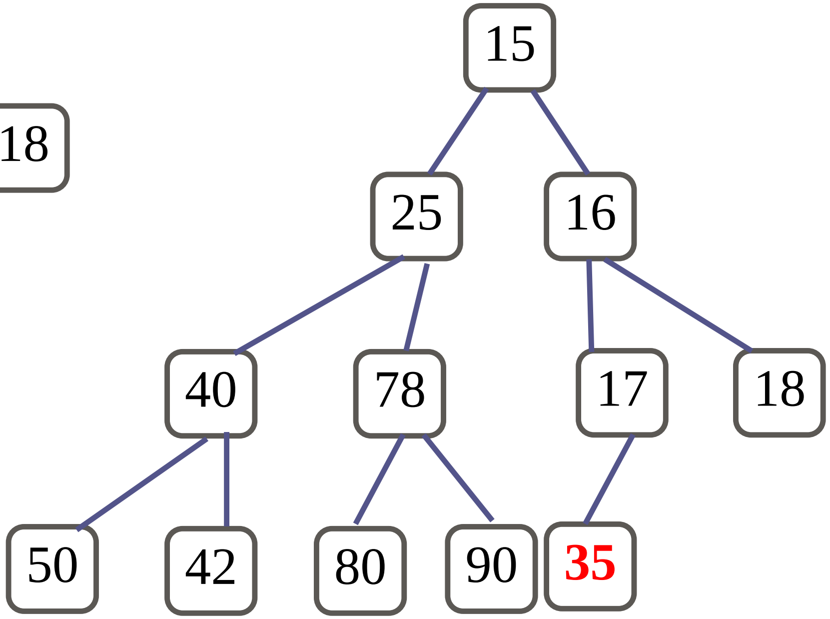
Inserting a node into a Heap

- Add the element to the ***bottom level of the heap*** – *maintaining the* **shape property**
- Compare the added element with its parent; if they are in the correct order, stop
- If not, **swap** the element with its parent and return to the previous step (the parent must be less than or equal to its children – *maintaining the* **order property**)
- The number of operations required is dependent on the number of levels the new element must rise to satisfy the heap property
- **Time complexity:** **O(log *n*)**

Adding to a heap

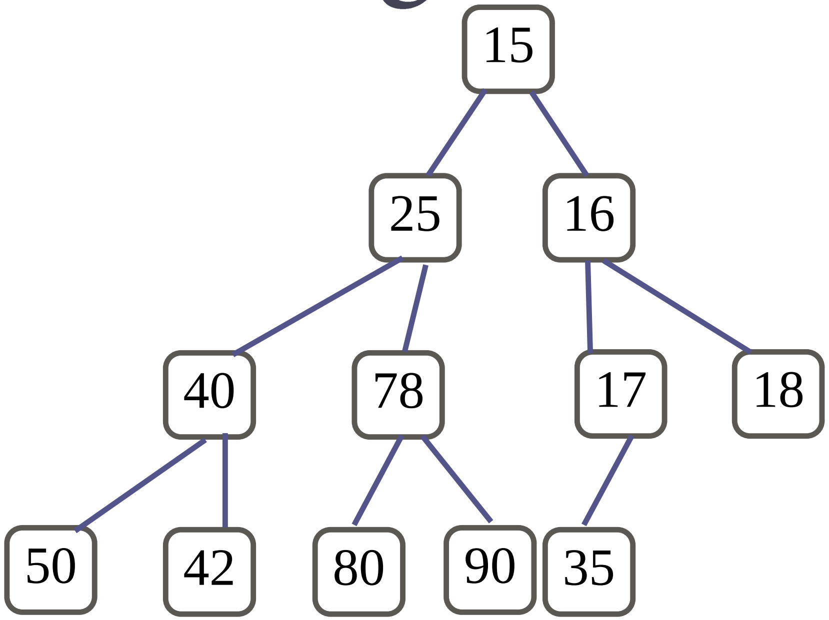


Let’s add the value 35

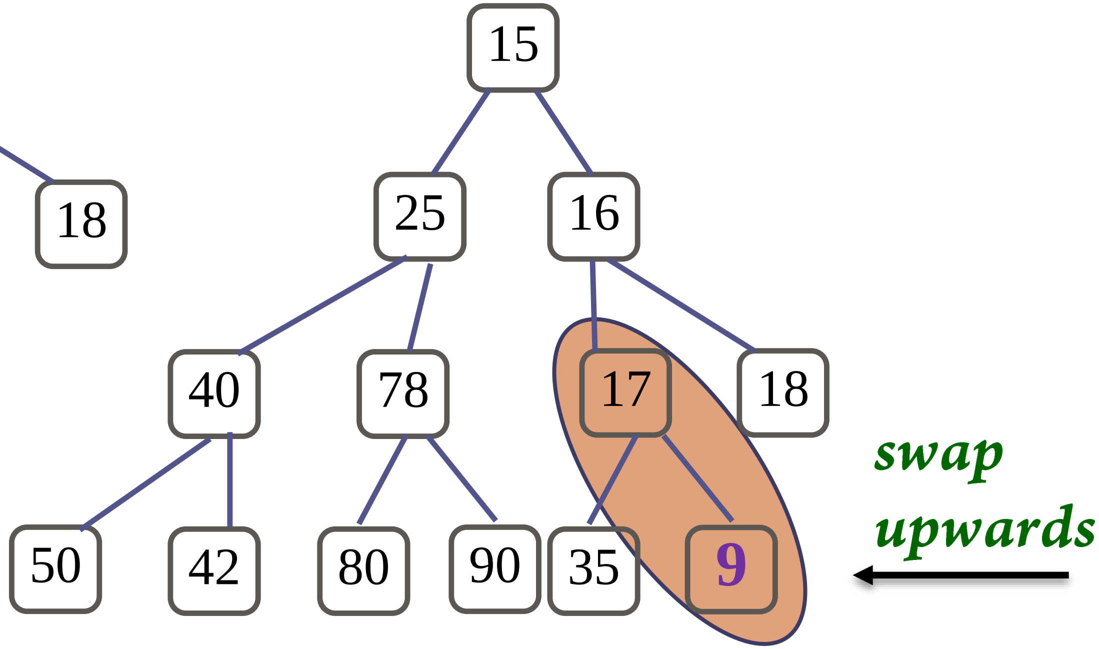


The heap properties are satisfied, nothing to re-arrange.

Adding to a heap

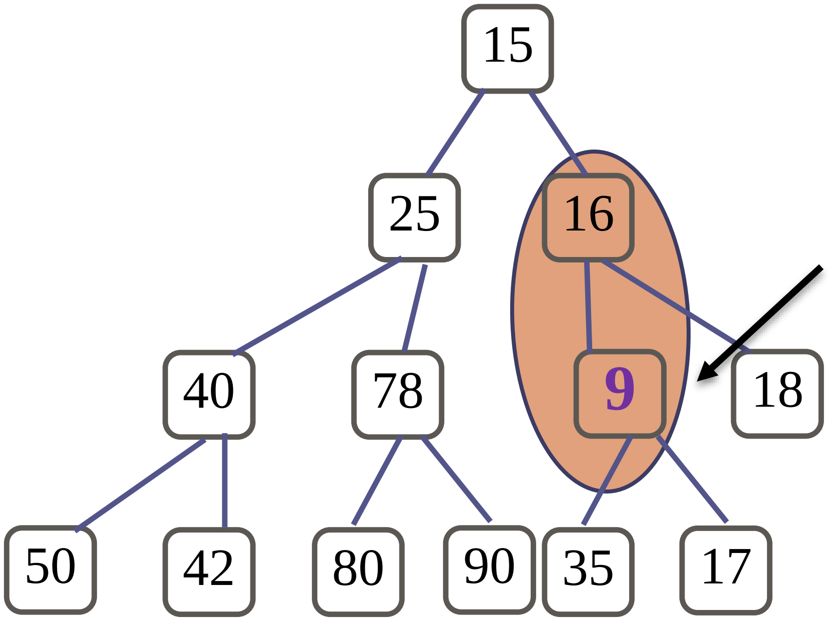


Let's add the value **9**

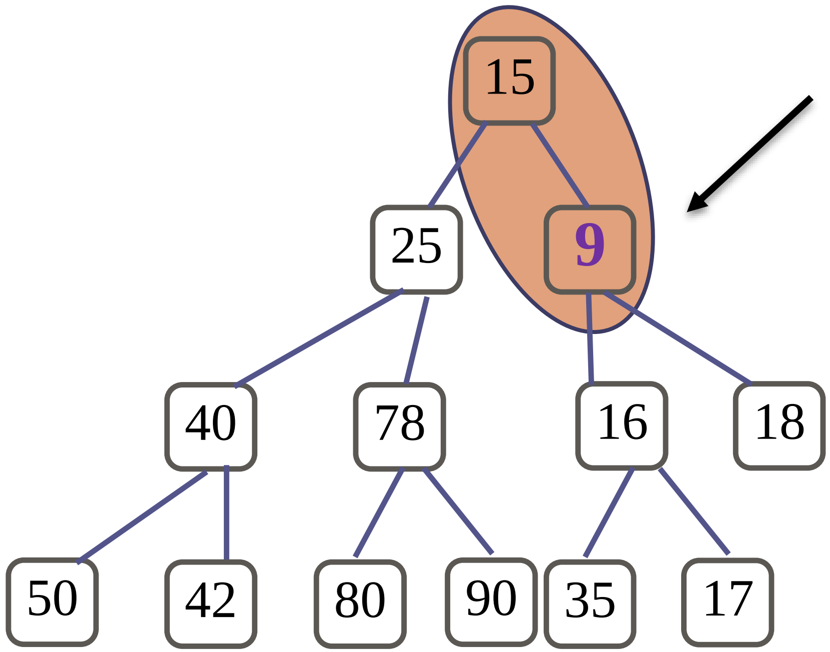


17 cannot be a parent to 9, as 9 is less than 17

Adding to a heap



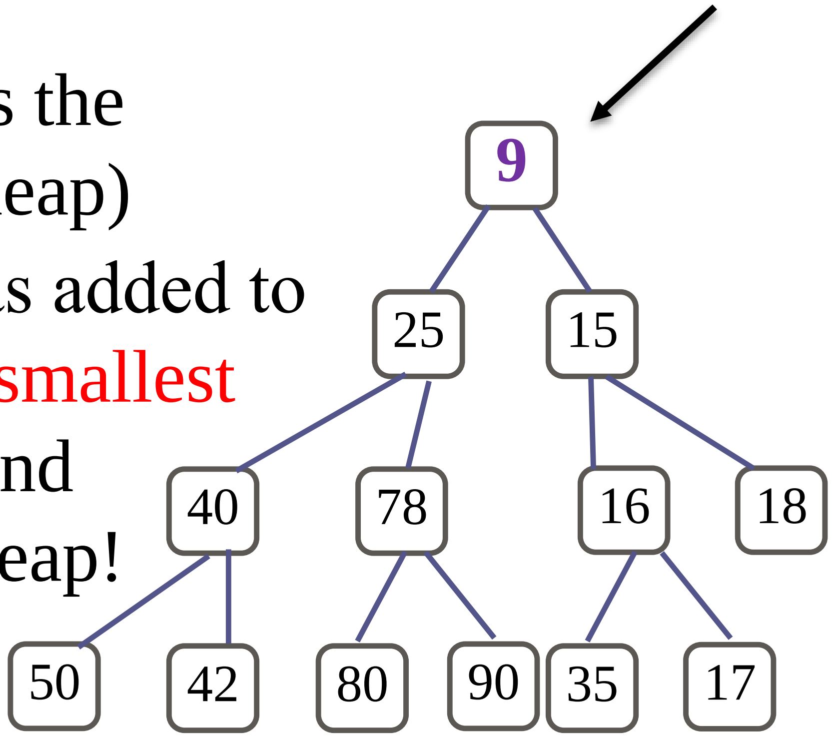
16 cannot be a parent to 9
(16 > 9)



15 cannot be a parent to 9

Adding to a heap

- The **min value** is always the **root** element (in a min heap)
- In this case, since ‘9’ was added to the heap, and it was the smallest item, it *rose to the top*, and became the root of the heap!



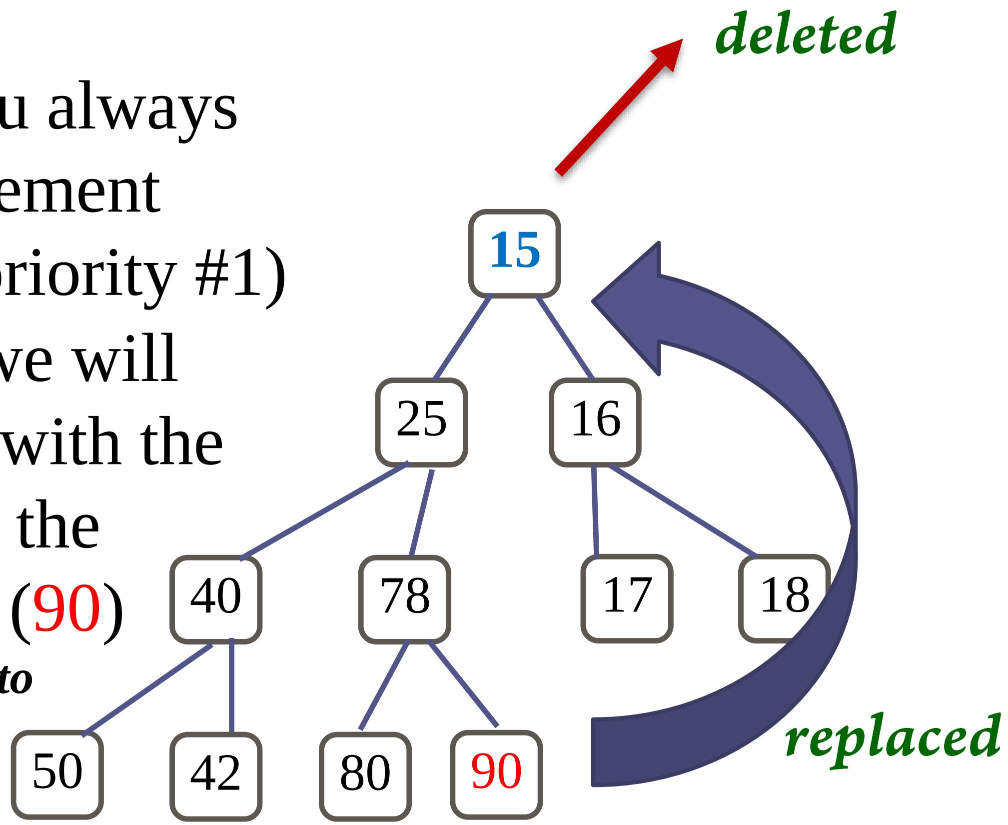
Deleting a node from a Heap

- Replace the **root** of the heap with the ***last element on the last level*** – *maintaining the* **shape property**
- Compare the new root with its children; if they are in the correct order, stop
- If not, **swap** the element with one of its children and return to the previous step. (Swap with its *smaller* child in a min-heap and its *larger* child in a max-heap – *maintaining the* **order property**)
- In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree. **Time complexity:** O(log n)
- [When retrieving the *smallest element*, we delete the root node]

Removing an element from a heap

- For a **priority queue**, you always remove the least value element (highest priority - think priority #1)
- In this heap, **15** is least, we will **remove** it and **replace** it with the **last node on the right** at the **bottom level** of the heap (90)

  Note: no other node is appropriate to initially replace 15!
- When retrieving the *smallest element*, we delete the root node (*min heap*)



Remove() method: remove root!

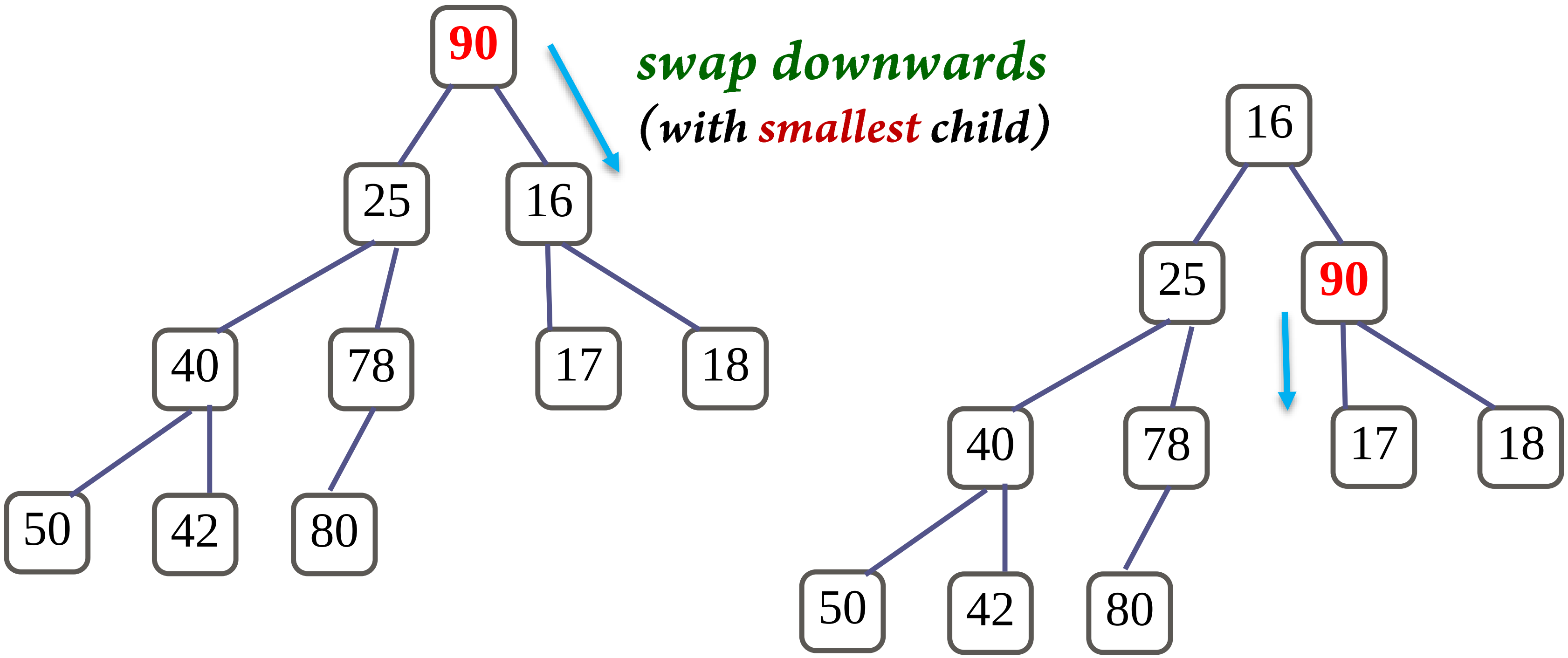
- Remember, when calling `remove()` you are NOT specifying which element to remove

- The `remove()` method **always** removes at ***the root*** of the binary heap *(nothing else!)*

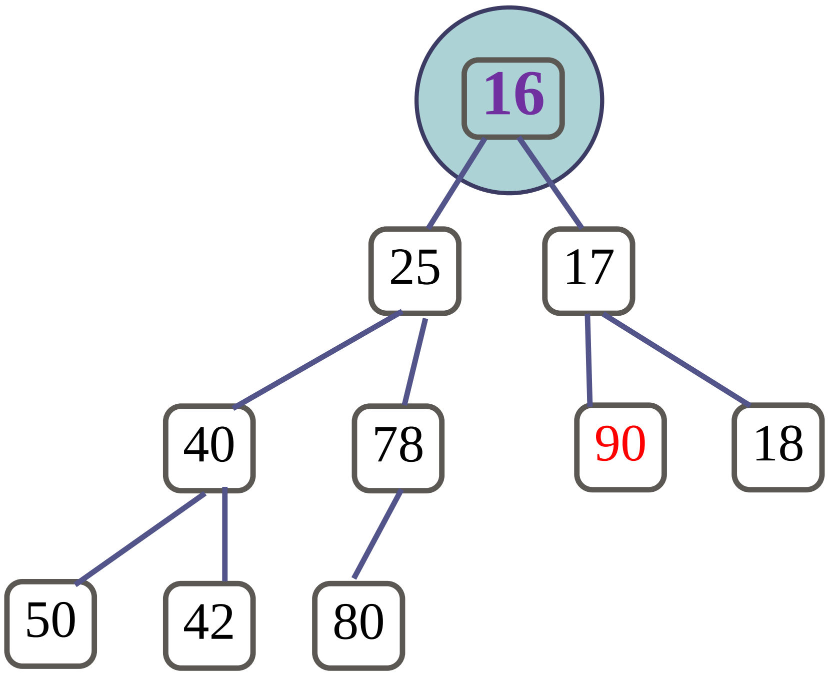
- So that the tree (heap) doesn't remain without a root node, replace it with **last node on the right** at the bottom level of the heap

Removing an element from a heap

- Maintain **order** property … (to preserve the heap!)

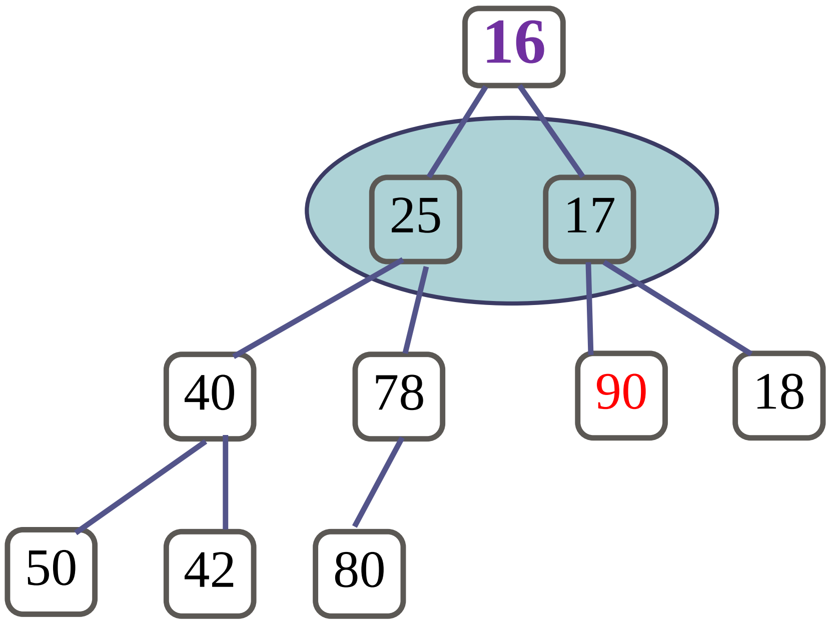


Removing an element from a heap



- Note how this rearrangement results in ***the next smallest element*** (16) positioned at the root (after original root was removed)?
- Also the tree is *balanced*!

Removing an element from a heap



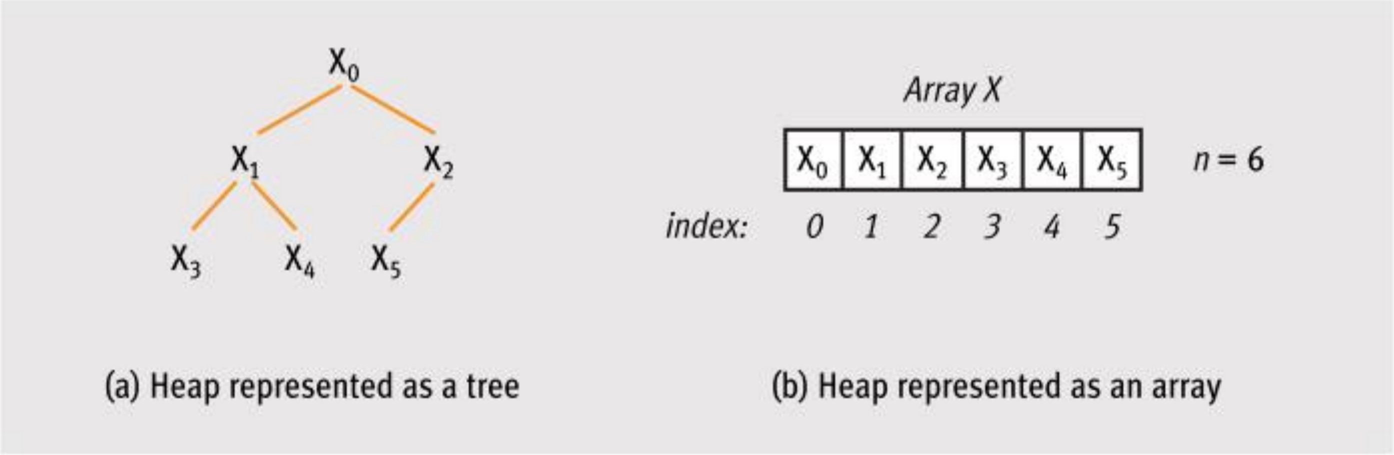
Notice: In a binary heap, after the root node, the next <u>two</u> smallest values are NOT always going to be the immediate children of the root node! (17 is but 18 isn't)

Why keep a balanced Tree?

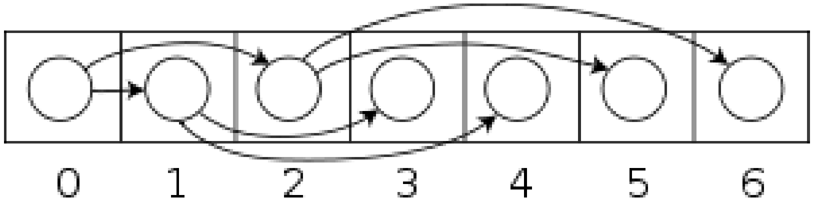
- Height of the tree determines **the time required for adding and removing elements** - keeping the tree balanced **maximizes performance**
- Adding an element requires 1 step for every level of height
- A tree of **height h** contains **$2^{h-1} <= n < 2^h$ elements**
- or: h-1 <= **$\log_2(n)$** < h
- Therefore: adding and removing elements is **O(log(n))**

Heaps ~ 1-D Arrays

- We can store the elements of our heap in a one-dimensional array in strict left-to-right, **level order** ***("breadth-first traversal")***
- That is, we store all of the nodes on level *i* from left to right before storing the nodes on level *i* + 1. This one-dimensional array representation of a heap is called a **heapform**



[FIGURE 7-31] A tree and a one-dimensional array representation of a heap



Heaps ~ 1-D Arrays

- We do not need pointers in this array-based representation because the parent, children, and siblings of a given node must be placed into array locations that can be determined with some simple calculations
- For a node stored at array location i, $0 \leq i < n$, where n is the total number of nodes in the heap…

`Parent(i)` $= \textbf{int}\ ((i-1)/2)$ if $(i > 0)$, else i has no parent

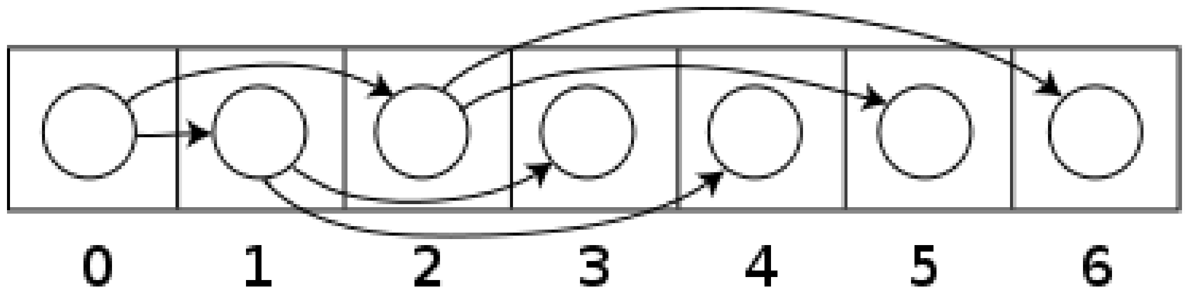
`LeftChild(`i`)` $= 2i + 1$ if $(2i+1) < n$ else i has no left child

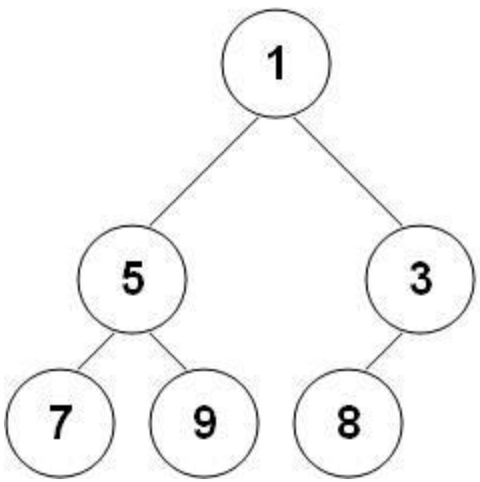
`RightChild` $(i) = 2i + 2$ if $(2i+2) < n$ else i has no right child

`Sibling (i)` $=$

if odd(i) then $i + 1$ if $i < n$ else i has no sibling

if even (i) then $i - 1$ if $i > 0$ else i has no sibling





Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

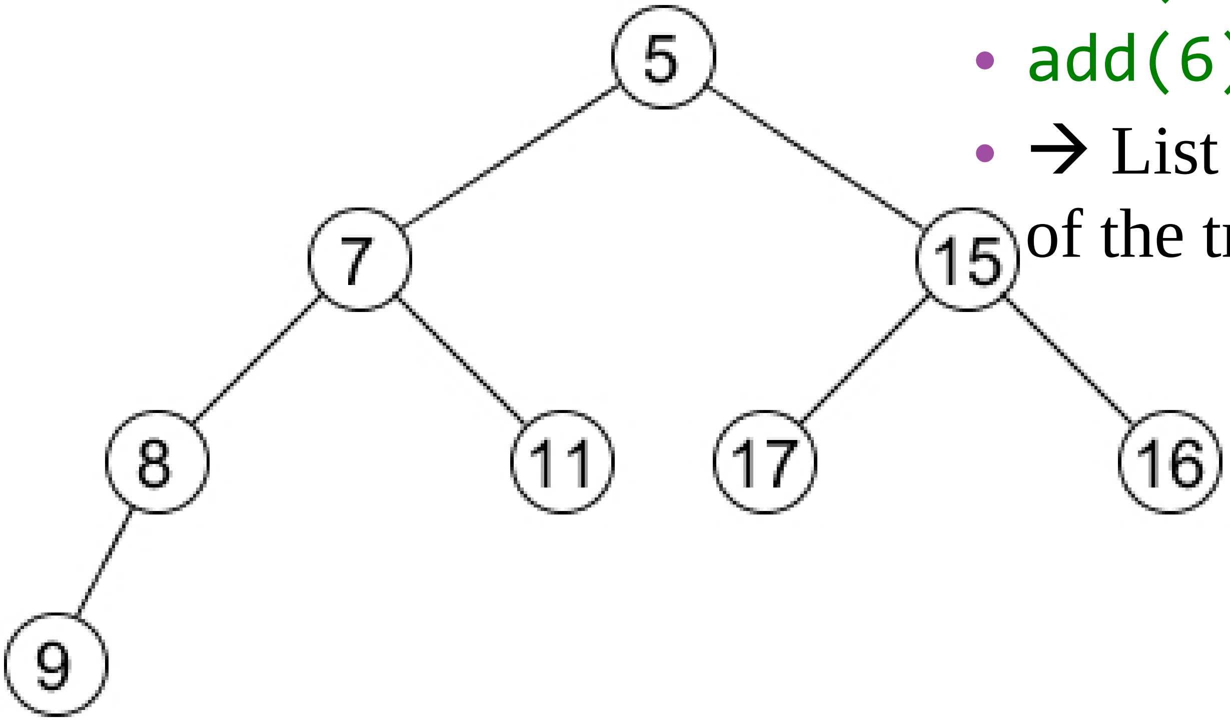
In-Class Activity – Binary Heaps

- You may work in pairs on this activity
- EVERYONE must submit individually

- Submit on **Collab** (see next slide for details)
  1. → Submit the **first** **in-order traversal** after performing the two `remove()` operations
  2. → Submit the **second** **in-order traversal** after performing the three `add()` operations

Heaps

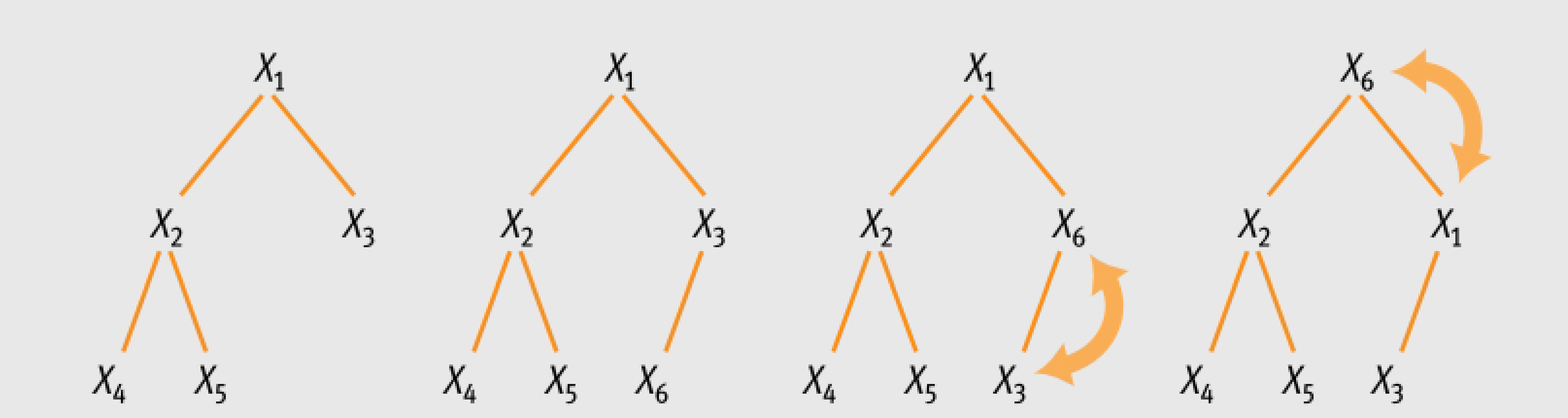
Use the (min) **Heap** given and perform the following **operations**:



- `remove()`
- `remove()`
- → List the **in-order traversal** of the tree at this point.
- `add(12)`
- `add(18)`
- `add(6)`
- → List the **in-order traversal** of the tree at this point.

Additional Slides

Inserting a node into a Heap



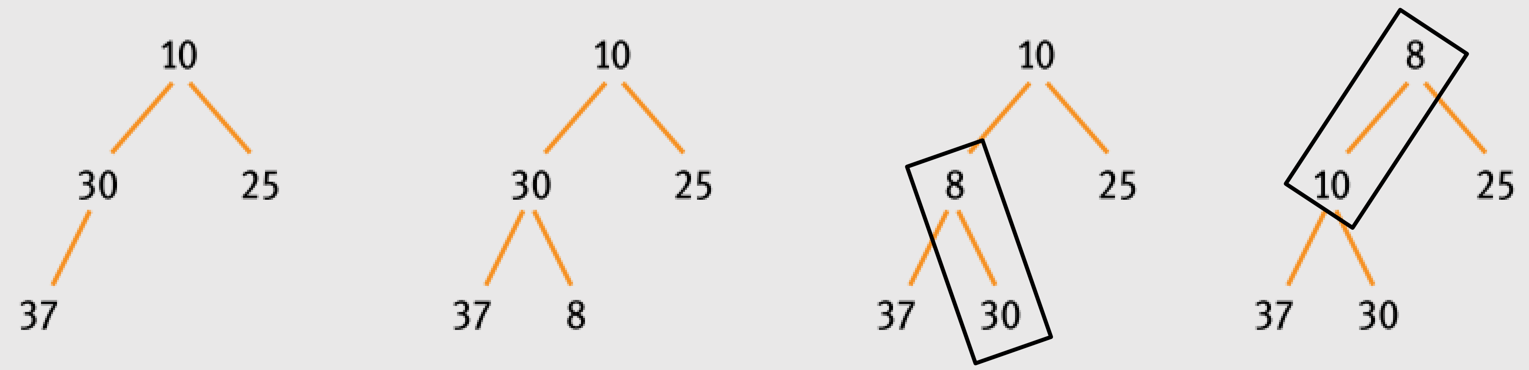
(a) Existing heap

(b) Addition of node X_6

(c) After first interchange

(d) After second interchange

Inserting a node into a Heap



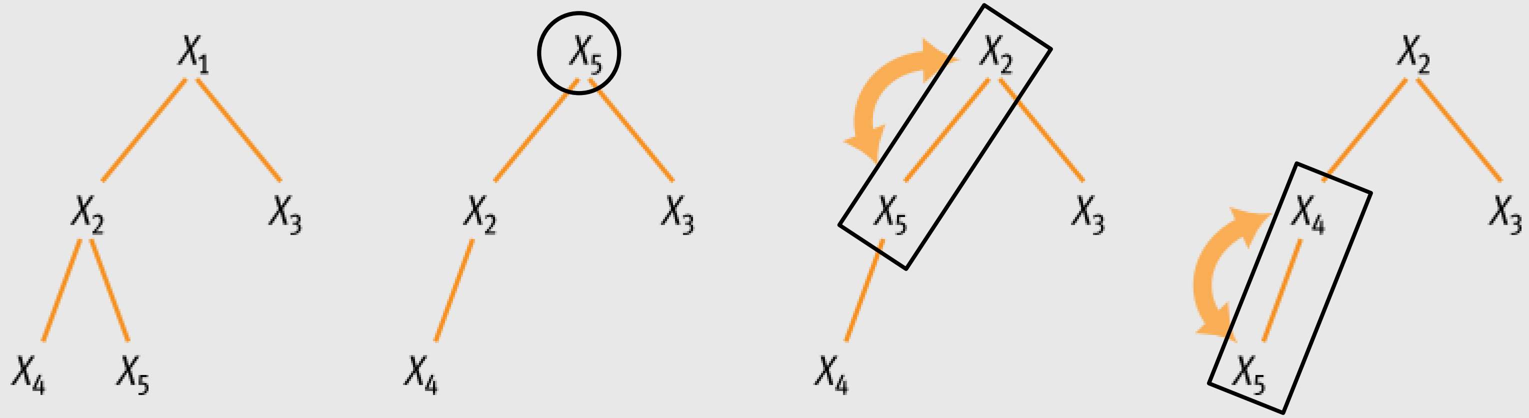
(a) Original heap

(b) After adding value 8 at the end

(c) After first interchange

(d) After second interchange

Deleting a node from a Heap



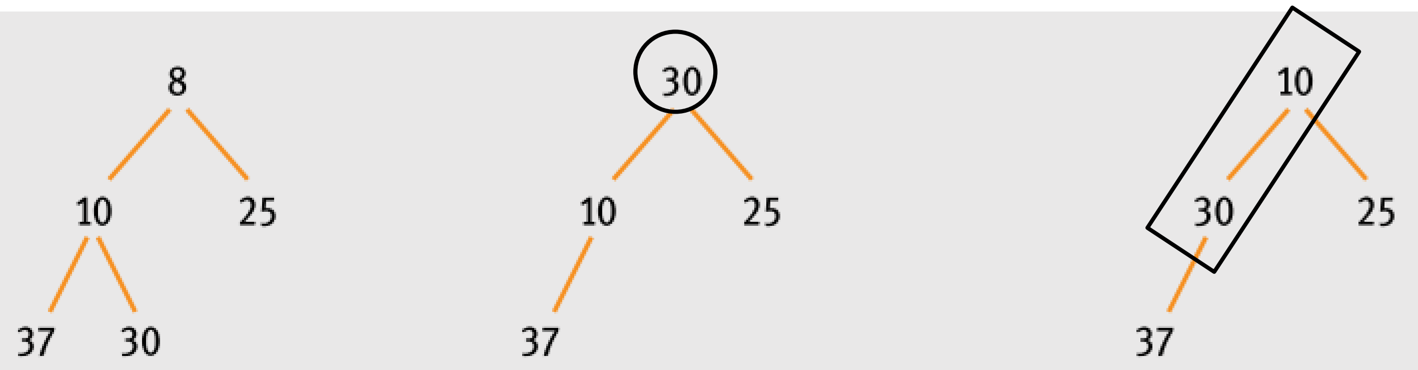
(a) Existing heap

(b) Removal of root and shifting of last element into the root position

(c) After one interchange with its smaller child

(d) After the second interchange

Deleting a node from a Heap



(a) Original heap
$n = 5$

(b) Heap after removing the value 8 and moving the value 30 into the root

(c) Heap after one interchange that restores the order property

- *<u>Note</u>: next* ***smallest*** *element is now at the root!*

Implementing a Heap in an Array

- Several methods can be implemented without recursion.
  For a heap with a **starting index of 1:**
  ```
  int getParent ( i )     { return i / 2; }
  int getLeftChild ( i )  { return 2i; }
  int getRightChild ( i ) { return 2i + 1; }
  int getSibling ( i )    { if i is even and i < n: i+1,
                             else if i is odd and i > 2: i-1; }
  ```
- For a heap with a **starting index of 0:**
  ```
  int getParent ( i )     { return (i-1) / 2; }
  int getLeftChild ( i )  { return 2i + 1; }
  int getRightChild ( i ) { return 2i + 2; }
  int getSibling ( i )    { if i is odd and i < n-1: i+1,
                             else if i is even and i > 1: i-1; }
  ```