Recursive Data Structures: Trees

-- Binary Heaps --

UNIVERSITY OF VIRGINIA
DATA SCIENCE INSTITUTE

Priority Queue

- A priority queue (PQ) is an ADT; it's like a regular linear data structure but each element comes with a "priority" associated with it. In a PQ, an element with "highest priority" is served before any element with lower priority"

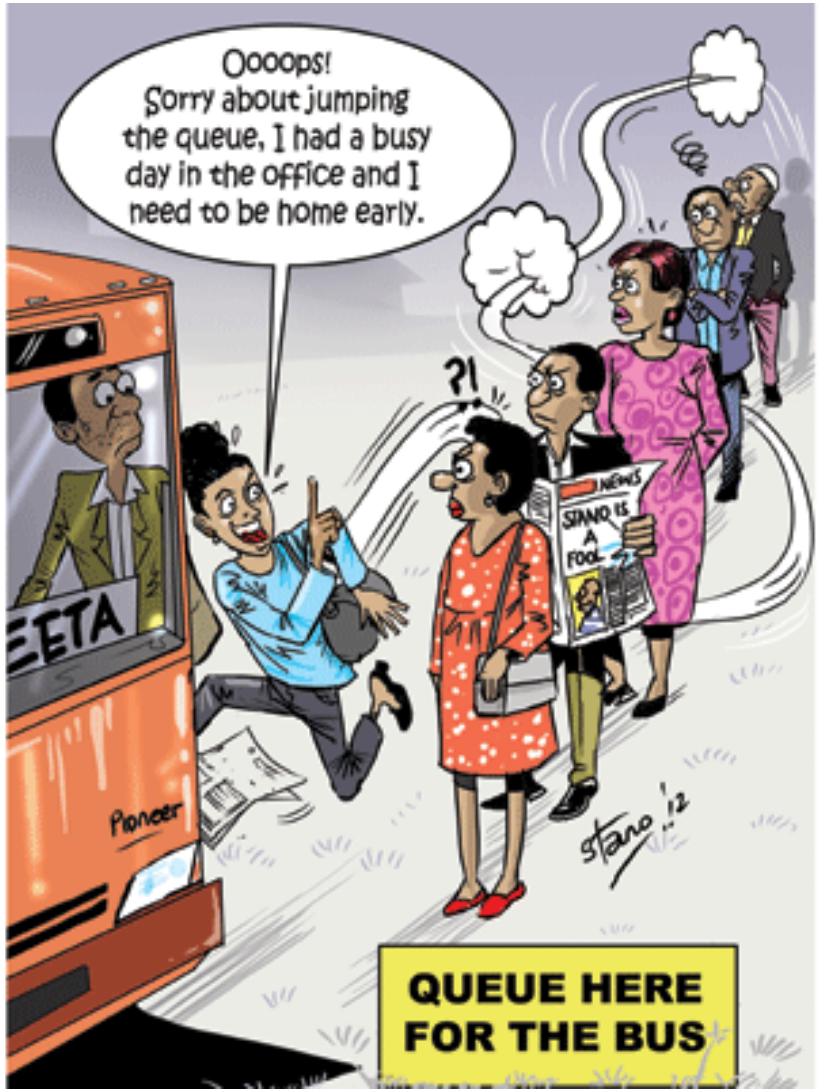
Priority Queue (PQ) - Definition

- A priority queue (PQ) is an Abstract Data Type (**ADT**)
- It is like a regular data structure, but where additionally each element has a "priority" associated with it
- The most important element deserving priority can be Min or Max depending on the nature of application
- In a PQ, an element with “high priority” is served before an element with “low priority”
- If two elements have the same priority, they are served according to their *order in the queue*

A Queue is always based on priority

- We have known about queue from childhood.
- A **queue** has '**FIFO**' structure (unlike a **stack**: "**LIFO**")
- In any busy place, people can be in a queue waiting to be served. It's always the first person to be served, with very rare exceptions such as: a customer with a full grocery cart might give way to someone buying one carton of milk; or, a person standing in queue to get a taxi will happily allow an elderly or a disabled person to go ahead of him
- There are very few such exceptions, otherwise it's always first come first served

Why Jump The Queue?



Why jump the Queue?

- There are many situations where it's morally and functionally acceptable, rather essential, to jump the normal queue and set a new sequence of priorities that does **not** follow FIFO pattern

1. In an **emergency room**, a person with unstoppable bleeding, a massive heart attack, or a lady waiting for delivery will be given higher priority and immediate attention
2. People who are **flying at 9 AM** will be given higher priority for check-in than the passengers who are taking off at 10 AM, irrespective of when they came to the airport or where they are standing

Why break the normal priority of a Queue?

- **Operating systems**: Scheduling jobs, allowing shorter jobs to be processed ahead of longer ones in a ***pre-emptive shortest job first scheduling technique***. This improves average response time (not raw performance)
- There is a very large number of situations where people, tasks or events are tagged with a priority key for processing
- The element with the topmost priority can be ‘max’ or ‘min’. A student with the highest grade ‘max’ could have top priority or a worker with the lowest execution time ‘min’ could have the highest priority or a job with shortest burst time ‘min’ could have highest priority

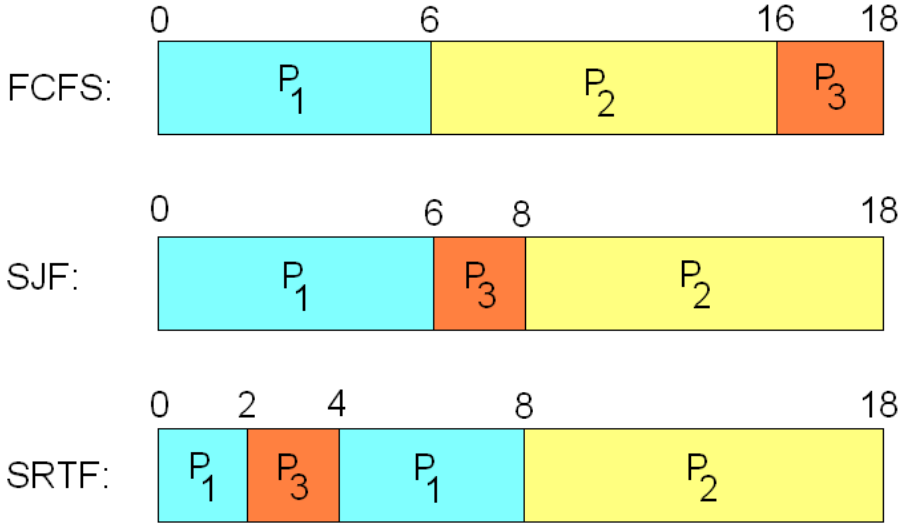
Example: Operating System Scheduling

Examples of Scheduling Types

FCFS = First Come First Served
(non pre-emptive)

SJF = Shortest Job First
(non pre-emptive)

SRTF = ***pre-emptive*** version of SJF
("shortest remaining time first")
Average response time is improved



Suppose that a system has three processes, with the following table of arrival times and burst times:

process number	arrival time	burst time
1	0	6
2	0	10
3	2	2

Abstract Data Type

- So… what is an abstract data type?

- An **abstract data type** (**ADT**) is a computational model for data structures that have similarity in behavior
- An abstract data type is defined indirectly, only by the **operations that may be performed** on it and by mathematical constraints on the effects of those operations for performance efficiency

Abstract Data Types (ADTs)

- ADTs are purely theoretical entities used to
  - Simplify the description of abstract algorithms
  - Classify and evaluate data structures

- An ADT may be implemented by specific **data structures**
- Often implemented as modules
- The module's interface simplifies the implementation and coding for the user

Priority Queue Operations

- **Priority Queue**: An abstract data type whose basic operations are:
  1. **Construct** a priority queue from a set of items
  2. **Insert** new elements
  3. **Remove** the largest or smallest element
  4. **Peek(return)** an element

- Because of its simplicity of implementation and versatility, the Priority queue has *many applications*

Basic properties of a Priority Queue

- Supports four basic operations: construct, insert (add), remove (delete), peek (return)
- Elements in the queue consist of a value and a priority key : element (k, v)
- Priorities are not unique: one or more elements may have ***same*** priority
- Elements are entered in sequence but removed according to priority
- There are several ways to implement a priority queue, however the **heap data structure** is the preferred option

Priority Queue Implementation

- **Array implementation** of priority queue
- **List-Based** priority queue
- Implementing priority queue using **heaps**

- It is to be emphasized that while we can use a number of data structures including the three given above, the core interface of priority queue remains the same, allowing a small set of standard operations for the user

PQ: Array implementation

- There are two options: use a **sorted array** or an **unordered array**
- In either of the two cases, deleteMin or deleteMax operations are carried out on the element of least or maximum priority (depending on the application)
- In an *unsorted array implementation* new elements are inserted at the **end**
- In a *sorted array implementation*, new elements are inserted **at a correctly ordered position** i.e., element with priority ‘B’ will be inserted between elements of priority ‘A’ and ‘C’

PQ: Array implementation

- Here is a step by step demonstration:

OPERATION	UNSORTED	SORTED
Insert X	X	X
Insert Z	XZ	XZ
Insert Y	XZY	XYZ
delMax (Z)	XY	XY
Insert B	XYB	BXY
Insert K	XYBK	BKXY
delMax (Y)	XBK	BKX

PQ: Array implementation

- In an **unsorted list** the new elements are placed (inserted) at the end of the array, requiring very little time. However, removing a priority item requires searching for that item, resulting in extra time

- In a **sorted list** the search and removal of a priority key is very fast because the array is sorted and the desired element is easily found (first element). However, insertion of an new item takes additional time because the new element must be placed in its correct ordered position

PQ: Array implementation

- Time complexity of sorted / unsorted arrays

	Insert	Delete
Unsorted array	O(1)	O(n)
Sorted array	O(n)	O(1)

- Once again… *Trade-offs!*
- As we can see, the **unsorted** array is efficient for insertion because the new element is added at the end of the array without any processing or ordering
- The **sorted** array is more efficient for deletion because finding an item to be deleted from a sorted list is faster

PQ: Anything better than arrays and lists?

- We briefly discussed PQ implementations using **arrays** and **lists**. In all cases, average time complexity for two key operations (insert, remove) is **O(n)**
- While these implementations are simple and give a good introduction to PQ, they are not efficient
- A much more efficient implementation of a PQ uses a **Binary Heap**
- A binary heap is a special version of **binary tree** with special structure and heap-order properties
- Before starting the discussion of implementing PQ using binary heaps, let's have brief introduction of **trees**, **heaps** and **binary heaps**

Binary Heaps

- A very interesting data structure that has a binary tree as its underlying structure (hence the name “binary heap”) It has many uses, one of which is to implement a data structure called a priority queue

Heaps (“binary heaps”)

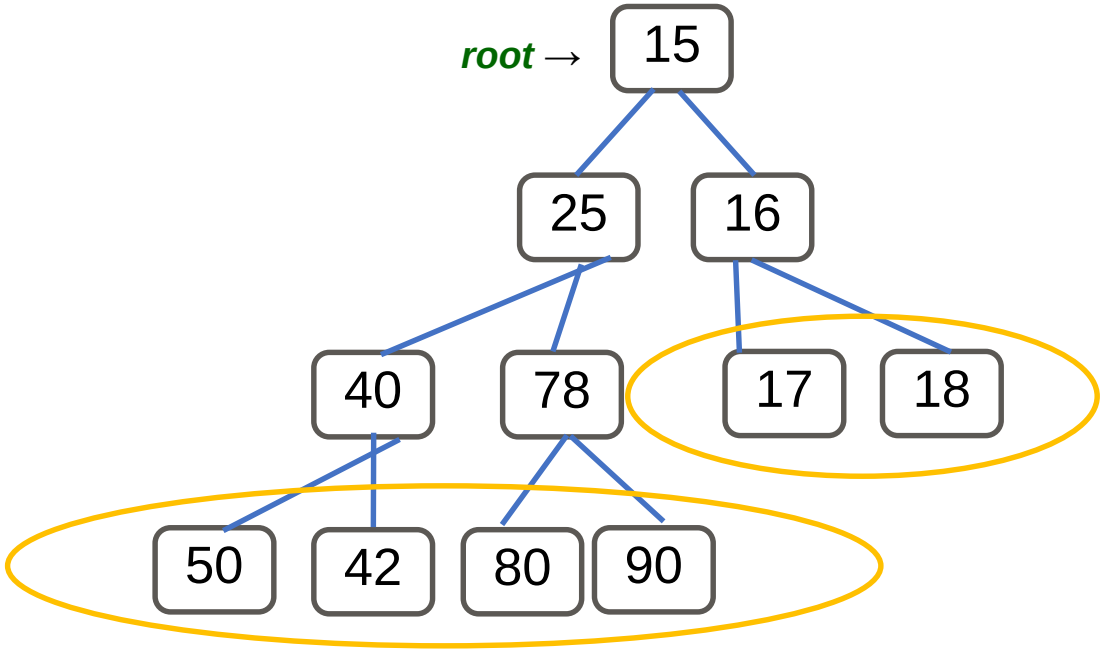
- The **heap** data structure is an example of a *balanced* ***binary tree***
- Useful in solving three types of problems:
  - Finding the **min or max** value within a collection
  - **Sorting** numerical values into ascending or descending order
  - Implementing another important data structure called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



Level i-3 (level 1)

Level i-2 (level 2)

Level i-1 (level 3)

Level i (level 4)

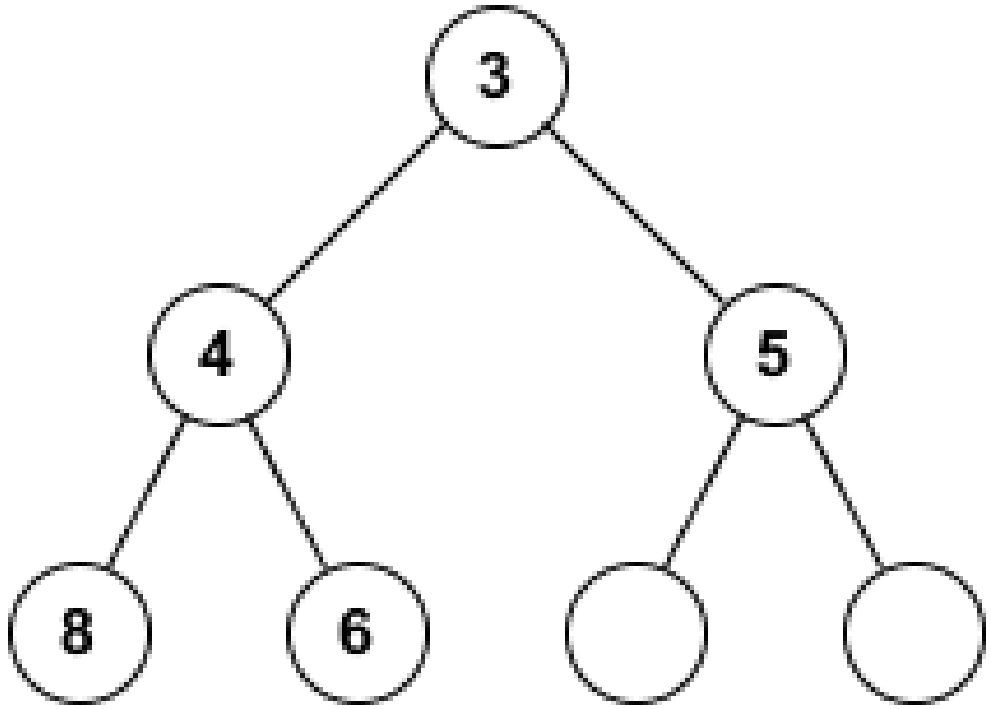
Minheap vs **Max**heap

- 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.

Binary Heap

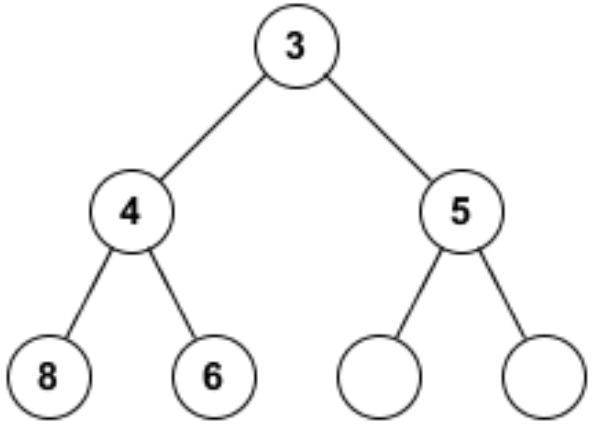
- The following binary tree is an example of a ‘min’ binary heap because the *minimum value is on the top*
- This tree representing a min heap has 5 keys with values
- Node 5 has its child nodes empty



Binary Heap

- Currently, each node has just a number only – the priority rating of that key. This is done for *simplicity* (you'll see this throughout the examples)
- In reality, each node has a **pair** of items (key, element)
- Scenario: hospital emergency room. Let's assume the pairs are:
- (3, Ann)
- (4, Joe)
- (5, Sue)
- (6, Bob)
- (8, Steve)

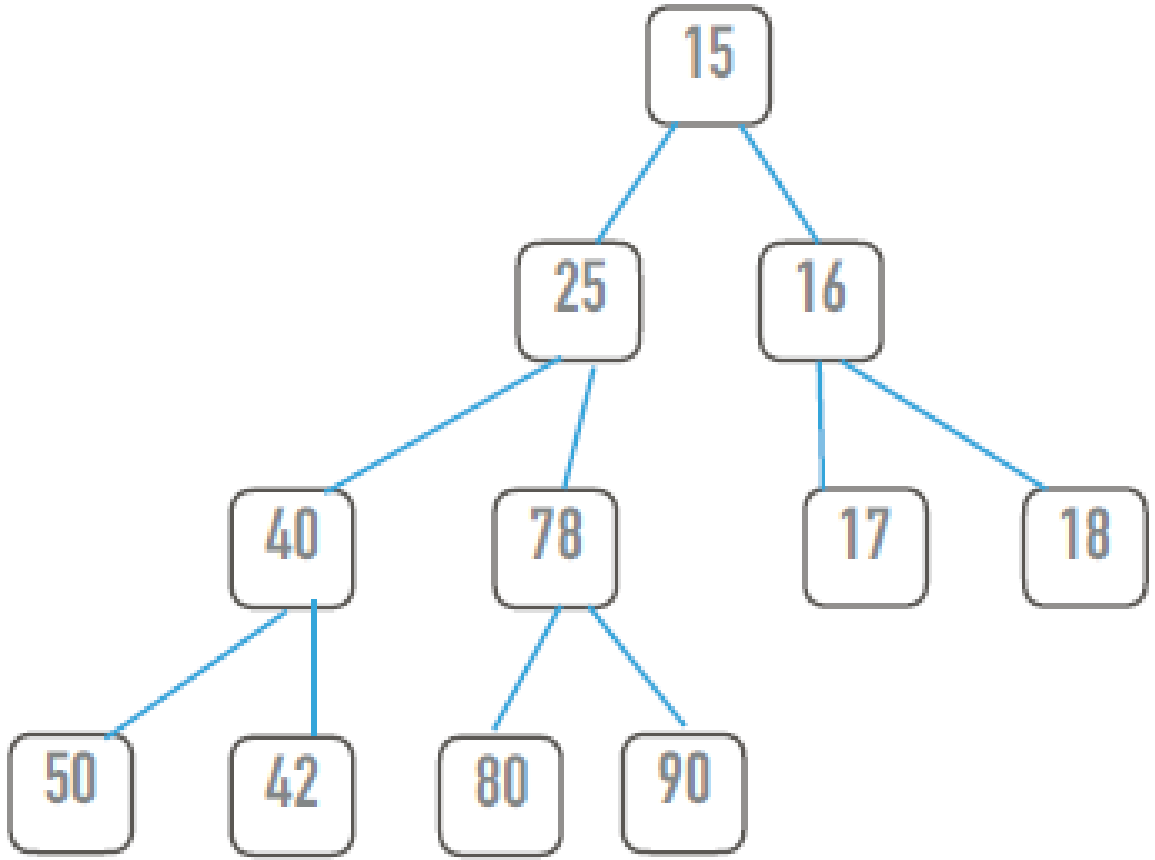
As our heap shows. From the 5 people who entered the emergency area, **Ann** will be taken first. In "PQ lingo", we say that the first key to be removed is '3'



Minheap

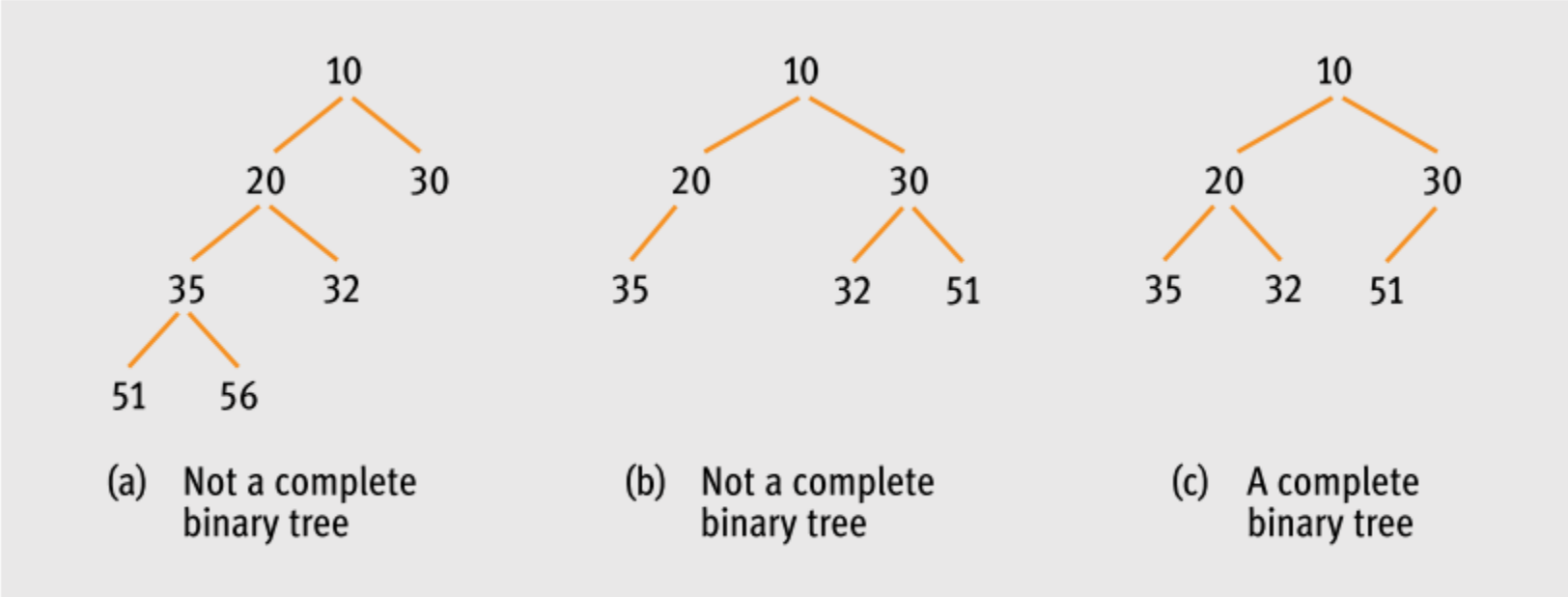
- This is a **min heap**
- The **smallest** value is the **root** of the tree
- All nodes are **smaller** than ALL its descendants

- 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?



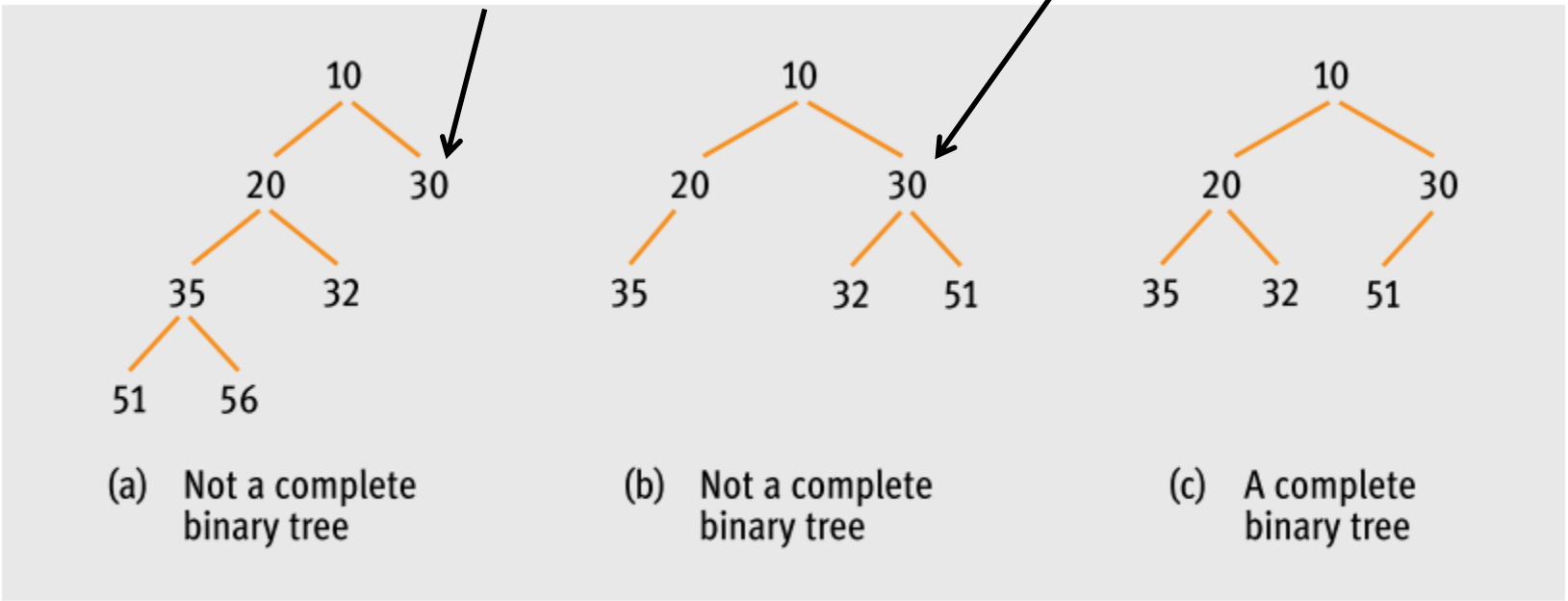
[FIGURE 7-29] Examples of valid and invalid complete binary trees

(complete except for the 'last' level)

Why First Two Are Invalid?

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

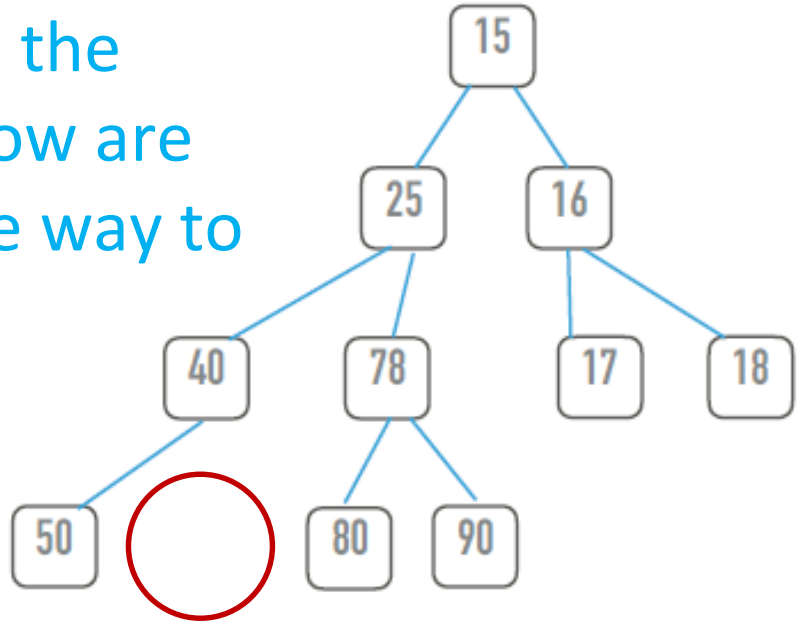
Leaves on level i are NOT as far to the left as possible

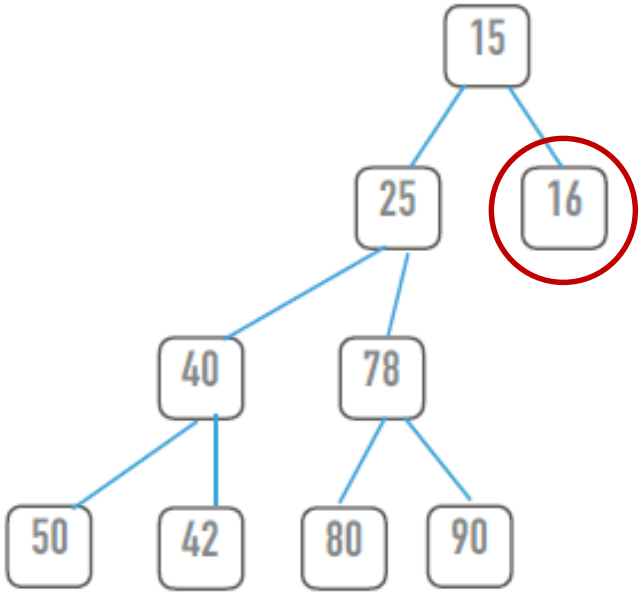


[FIGURE 7-29] Examples of valid and invalid complete binary trees

Examples of Invalid Heaps

Nodes on the bottom row are not 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

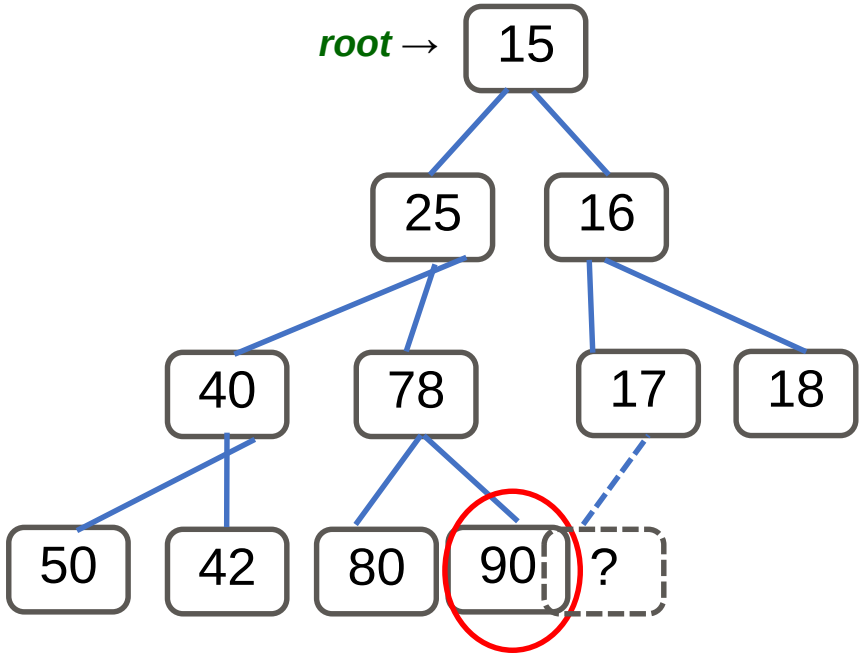
• all leaves are on the *lowest two levels*

• **nodes are added on the *lowest level, from left to right***

• **nodes are removed** (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)* ← ← ← ←

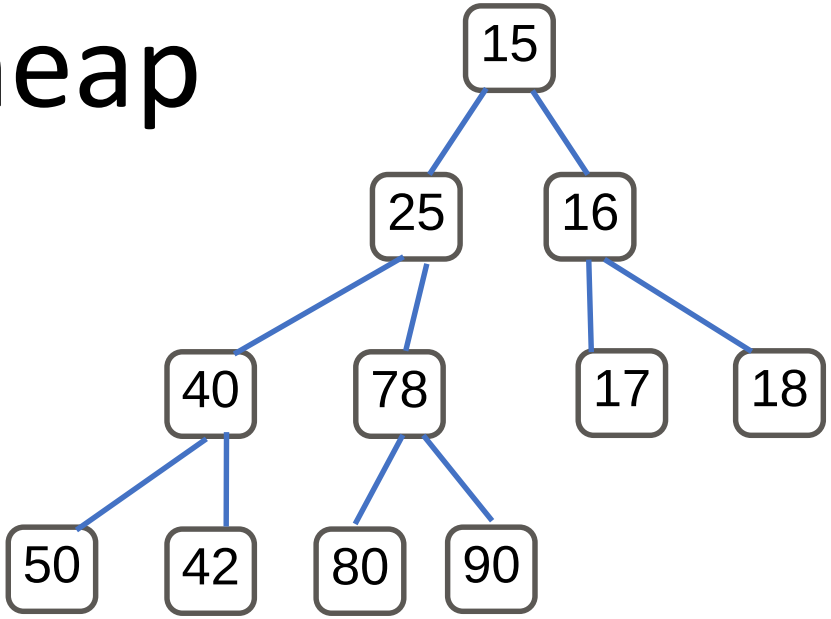
Binary Heap

- 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

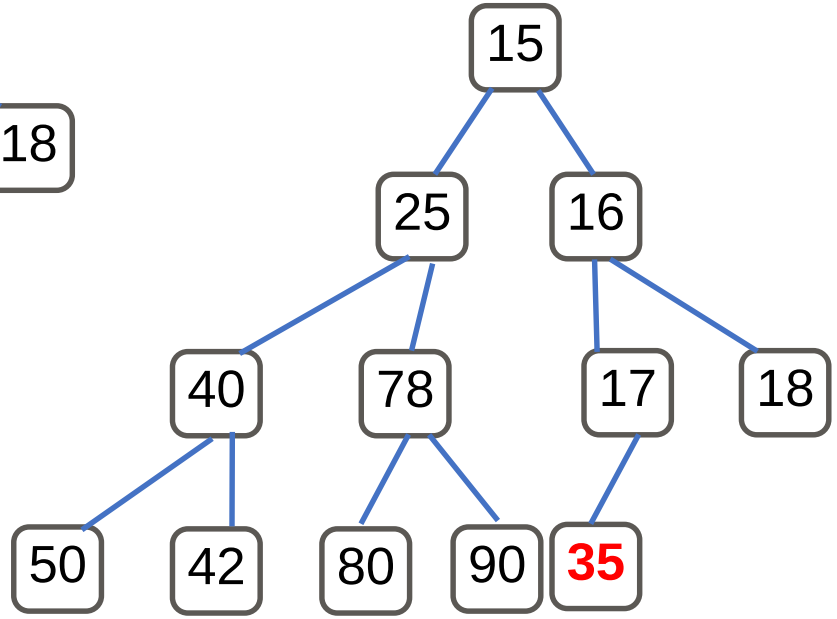
Inserting a node into a Heap

- 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

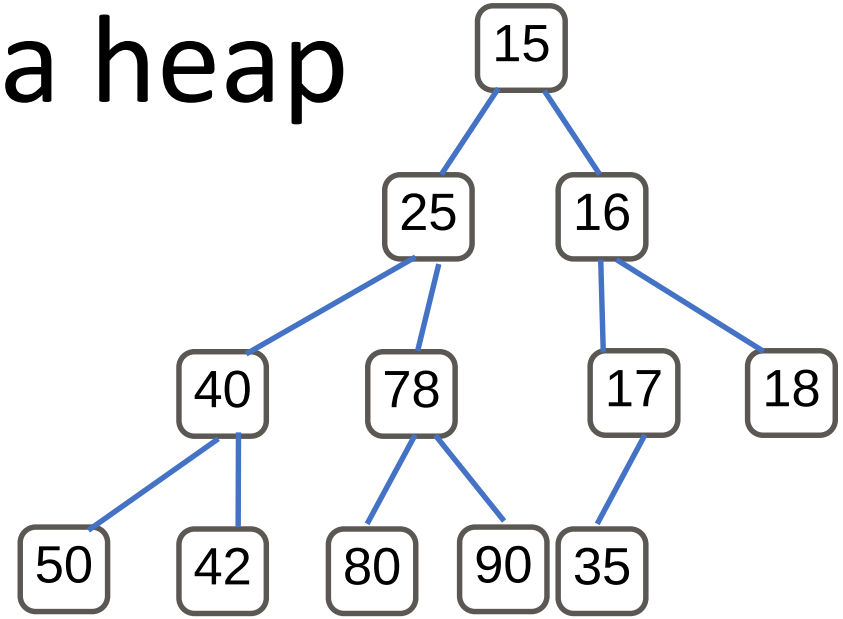


Let's add the value 35

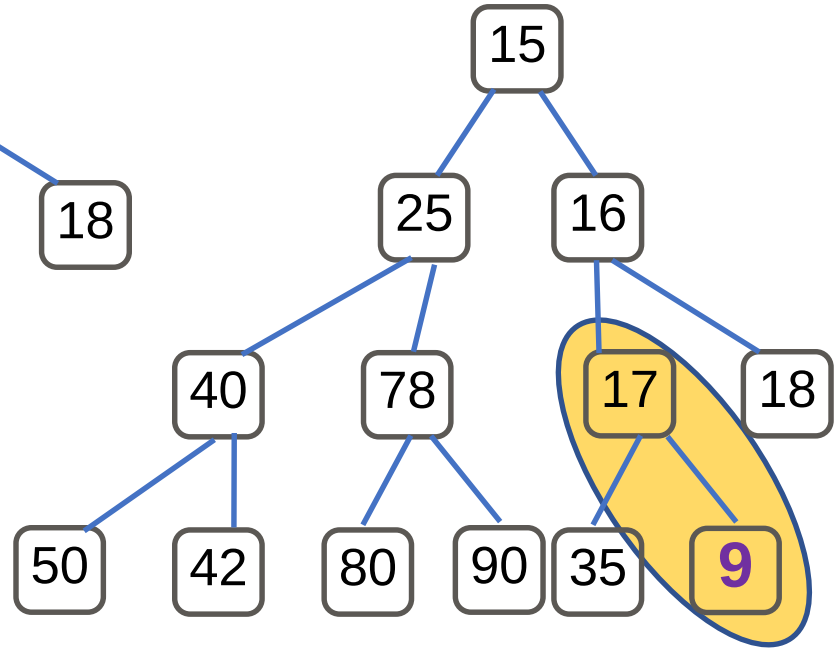


The heap properties are satisfied, nothing to re-arrange.

Adding to a heap



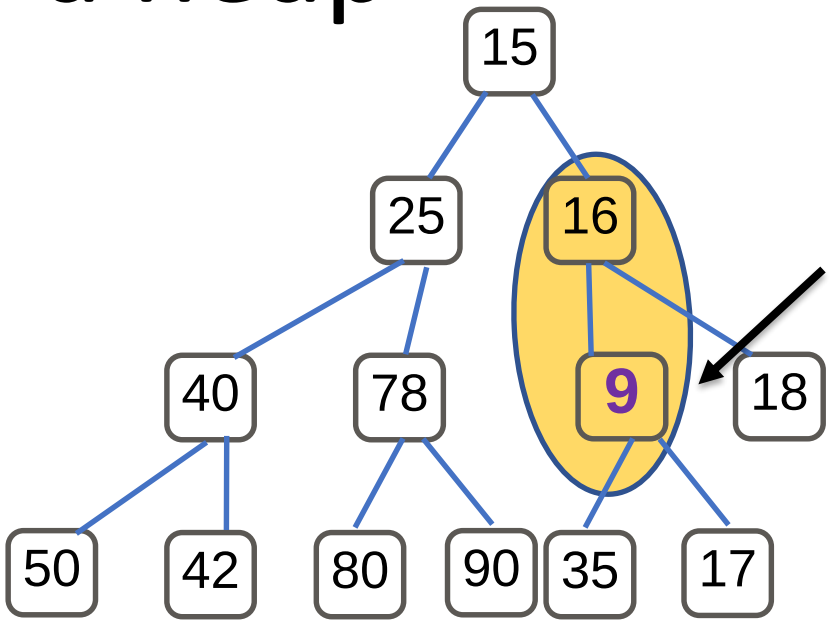
Let's add the value **9**



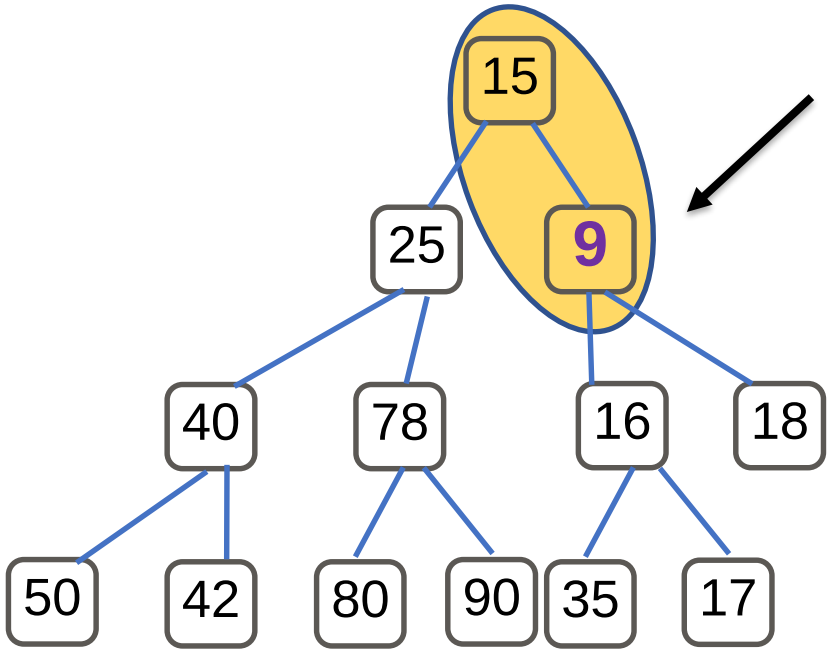
swap upwards

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

Adding to a heap



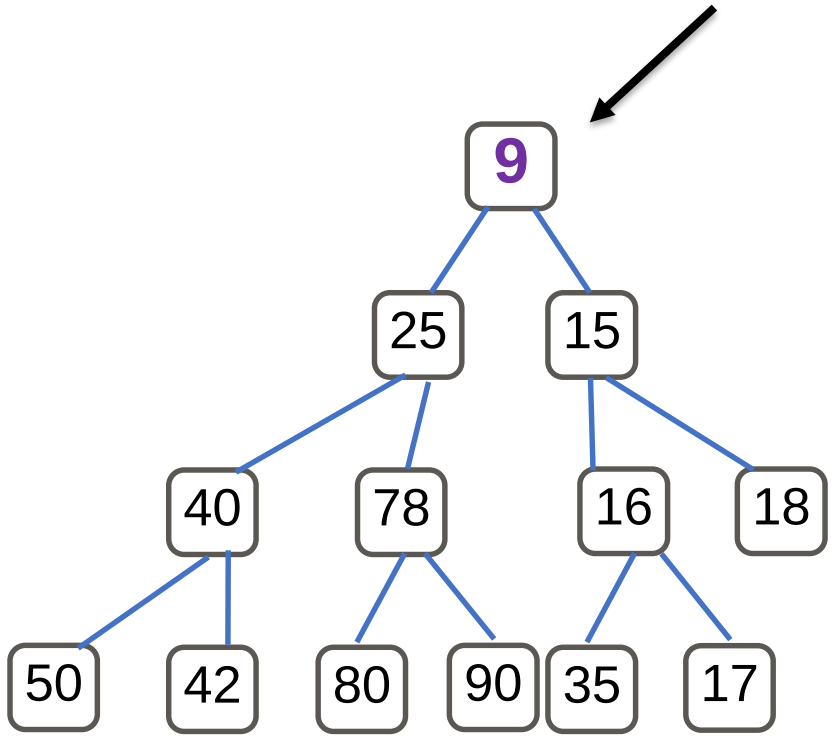
16 cannot be a parent to 9
(16 > 9)



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!

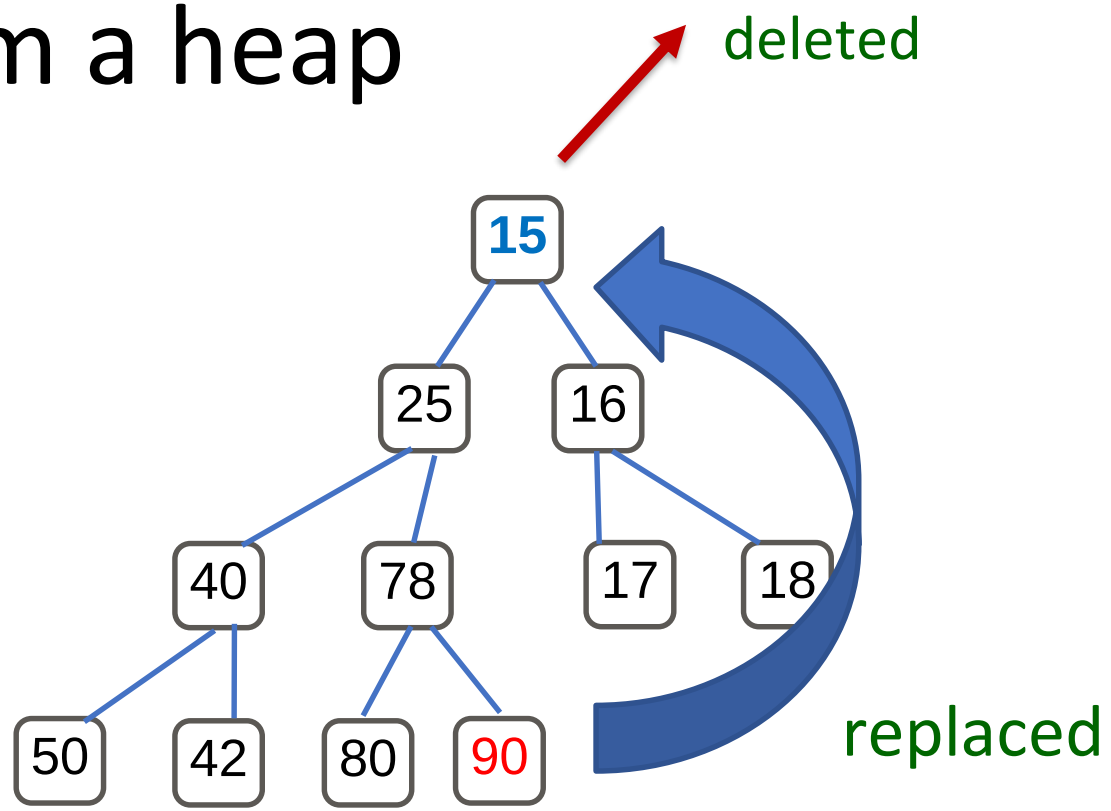


Deleting a node from a Heap

- 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)



Remove() method: remove root!

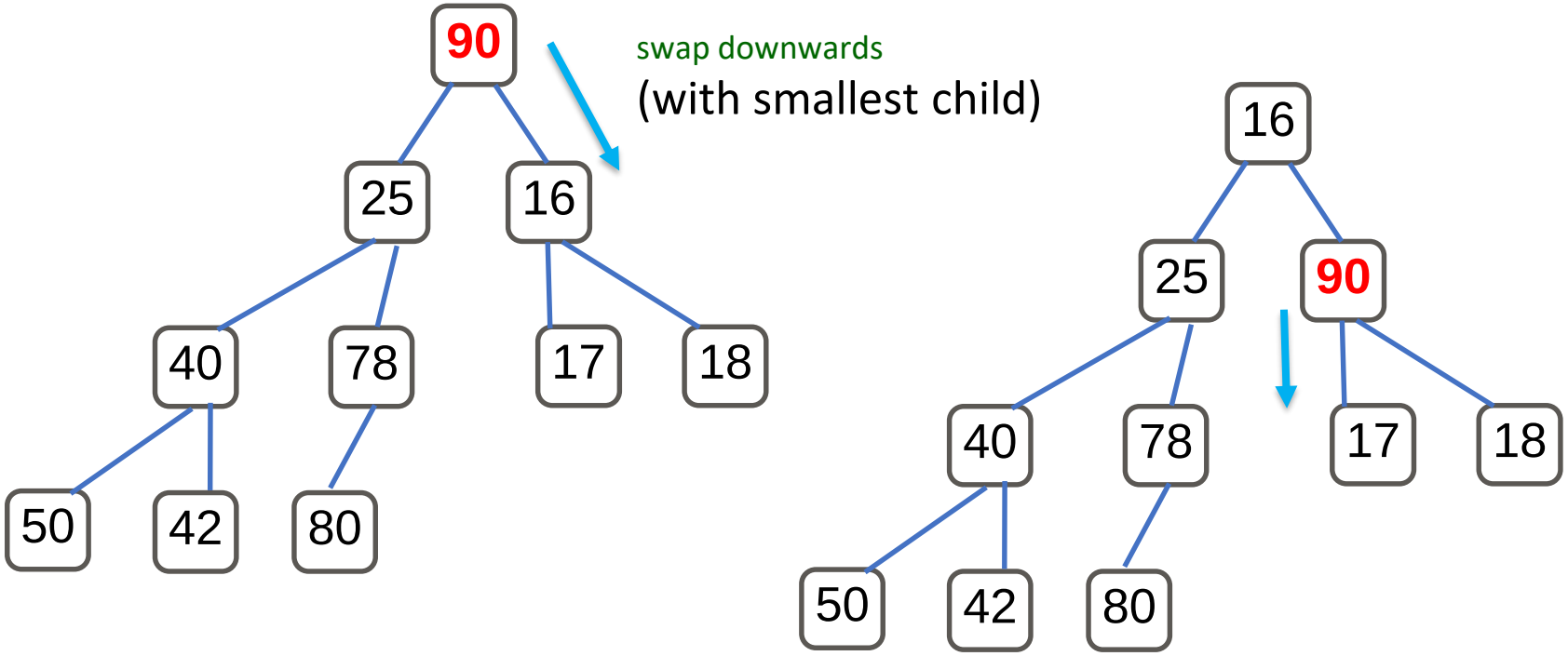
- Remember, when calling `remove()` you are NOT specifying which element to remove

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

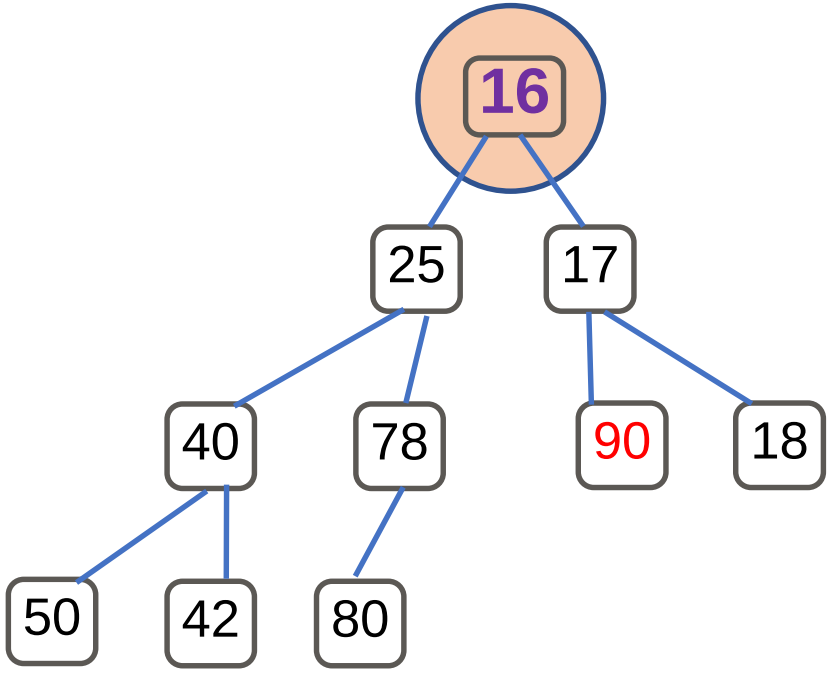
- So that the tree (heap) doesn't remain without a root node, replace it with **last node on the right** at the bottom level of the heap

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!*

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))**

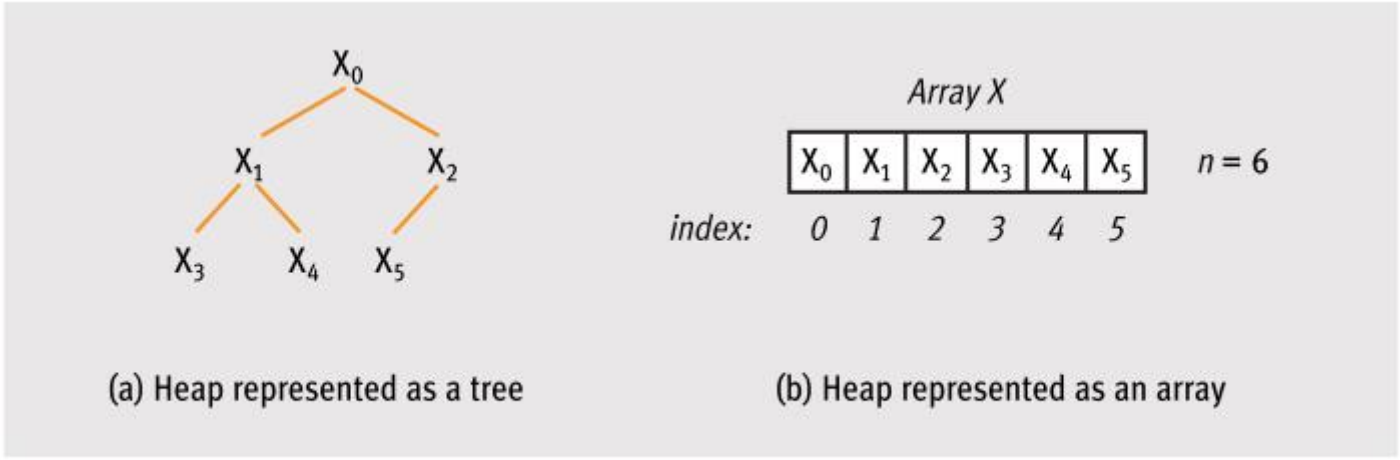
Additional Note About Time Complexity

- Given a priority queue of *n* elements, implemented using a binary heap, we know its time complexity will be:
- **Insertion** of a new element: O(log n)
- **Deletion** of the top priority element: O(log n)

- In a situation where data is available from a third party and we want to build a brand new priority queue it will take: **O(n log n)** time [Why? *n* * time to insert!]
- This is a *significant improvement* over the implementations that have been discussed earlier (array or list)

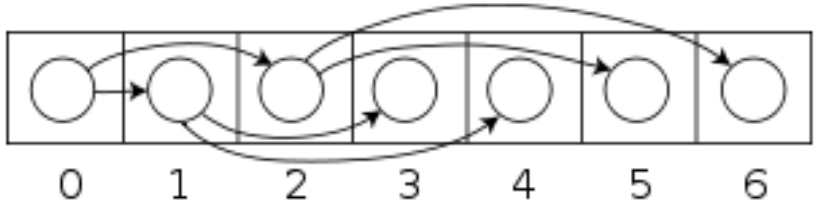
Heaps ~ 1-D Arrays

- 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**



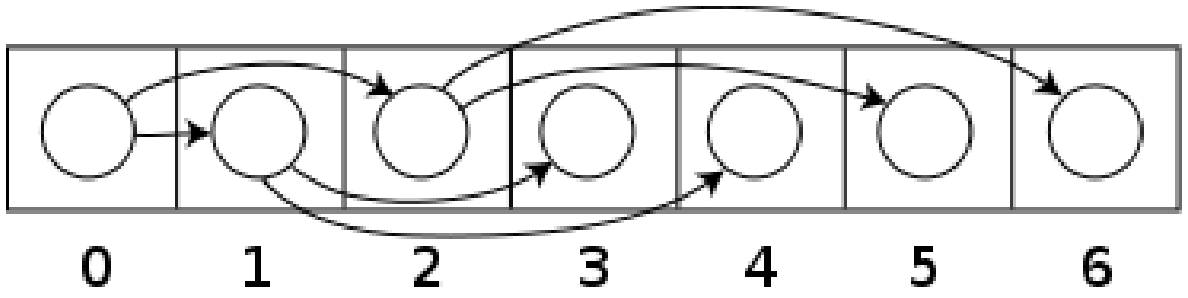
[FIGURE 7-31] A tree and a one-dimensional array representation of a heap

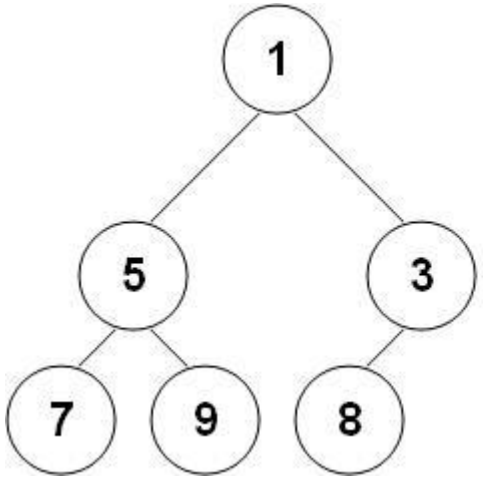
Heaps ~ 1-D Arrays



- 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)` = **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

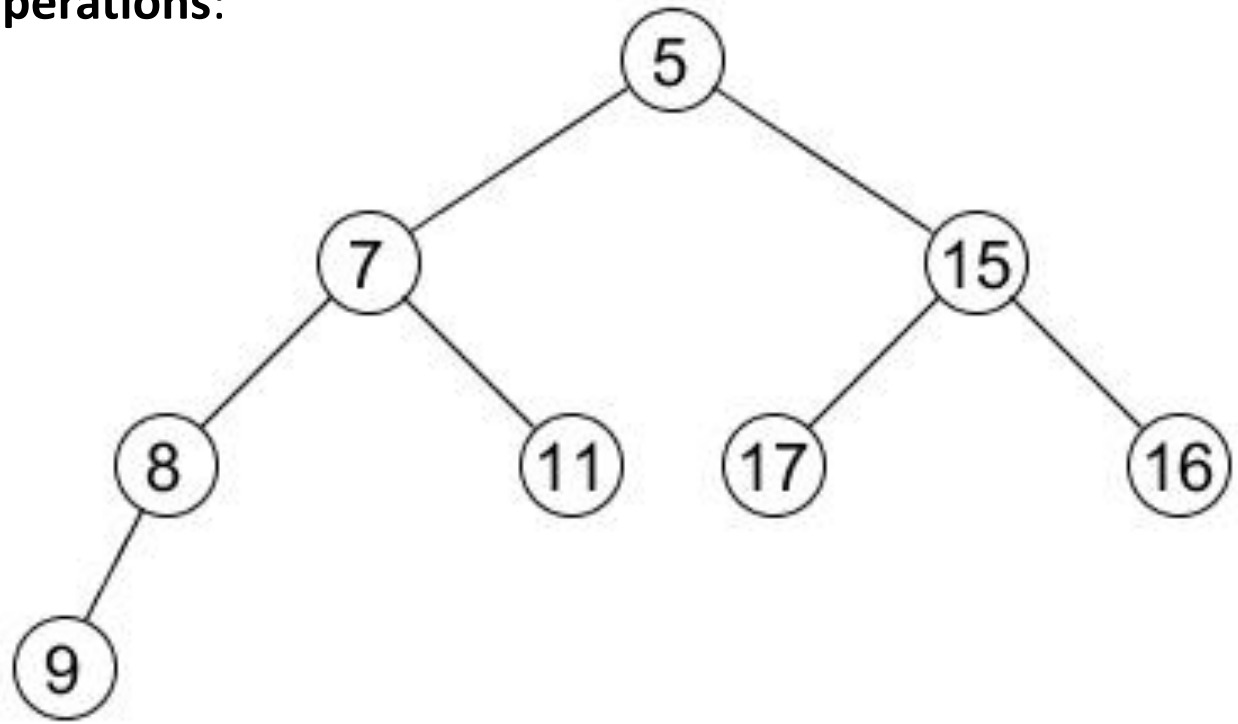
Activity – Binary Heaps

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

- What to submit: (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

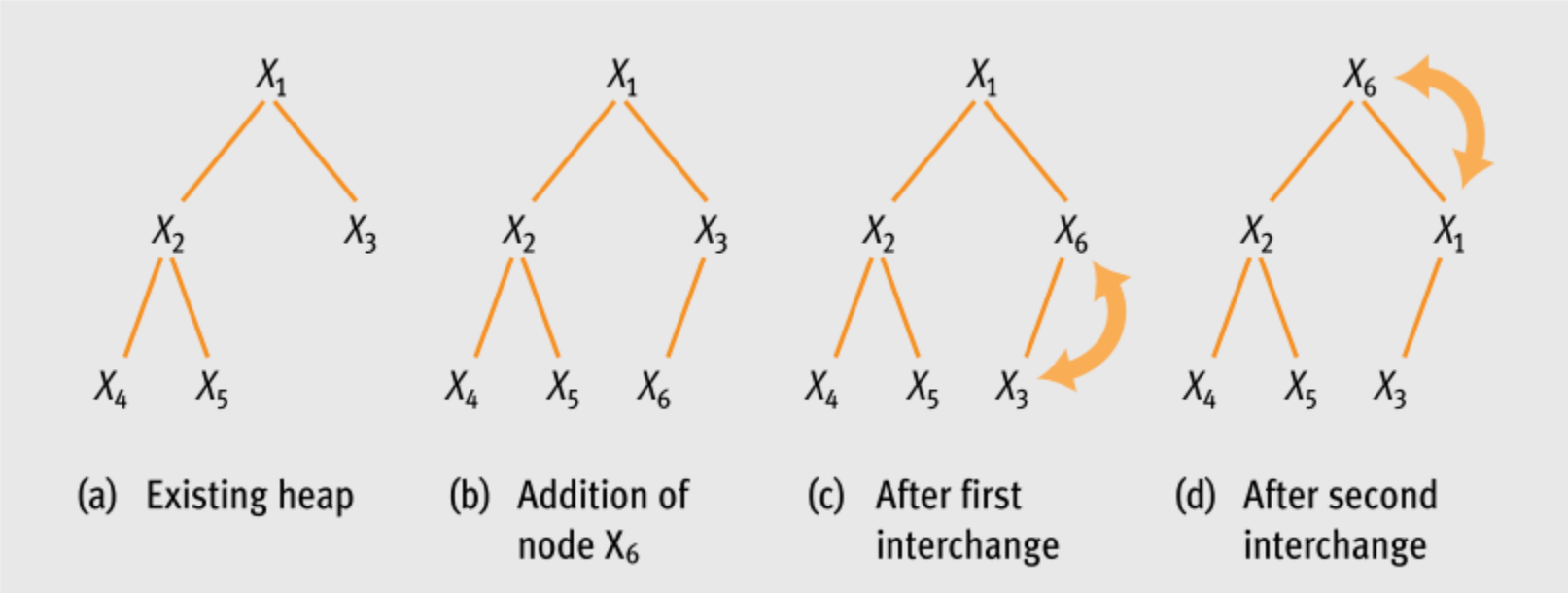
Use the (min) **Heap** given and perform the following **operations**:



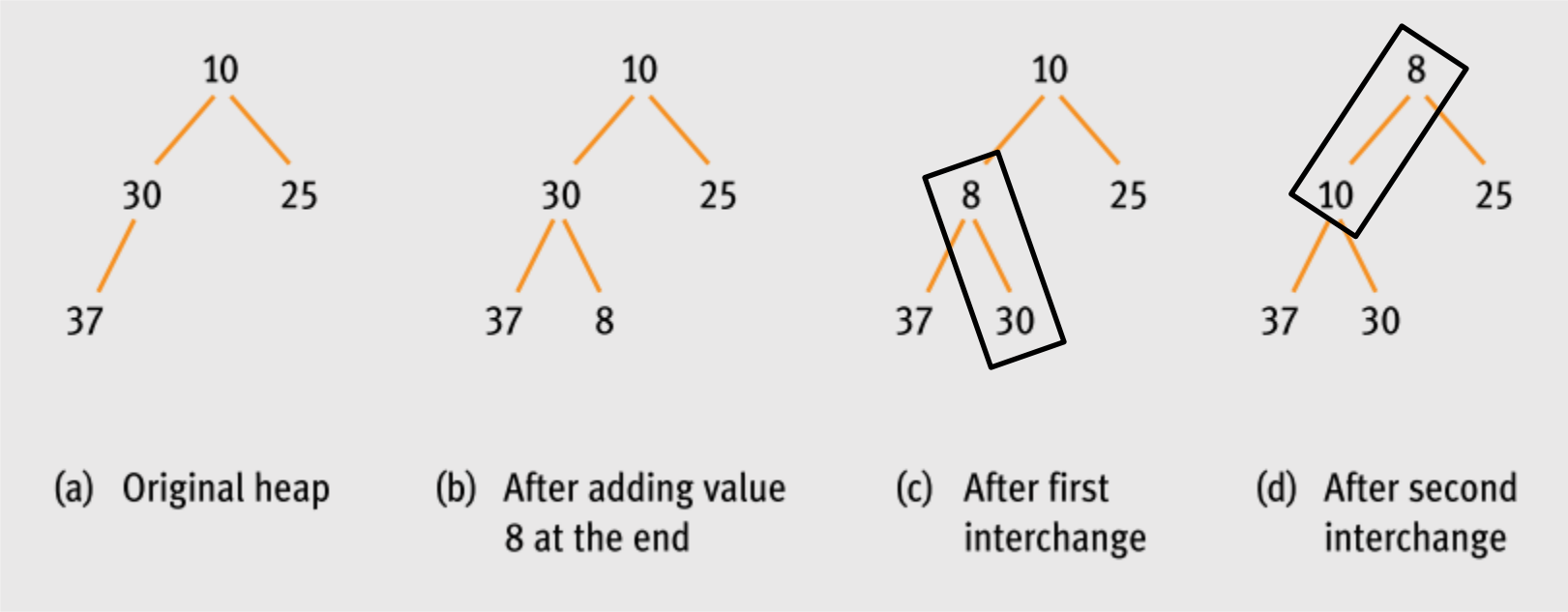
`remove()`
`remove()`
→ List the **in-order traversal** of the tree at this point.
`add(12)`
`add(18)`
`add(6)`
→ List the **in-order traversal** of the tree at this point.

Additional Slides

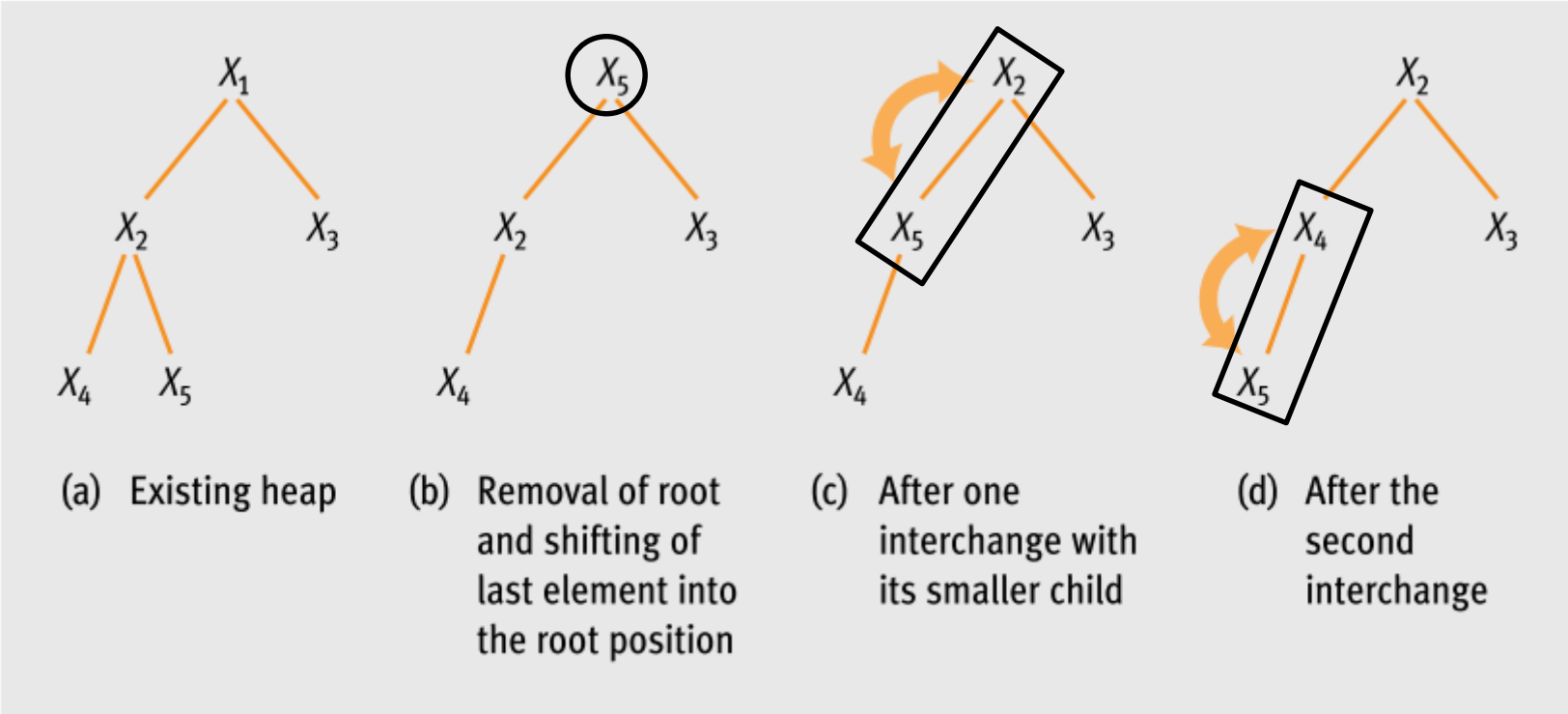
Inserting a node into a Heap



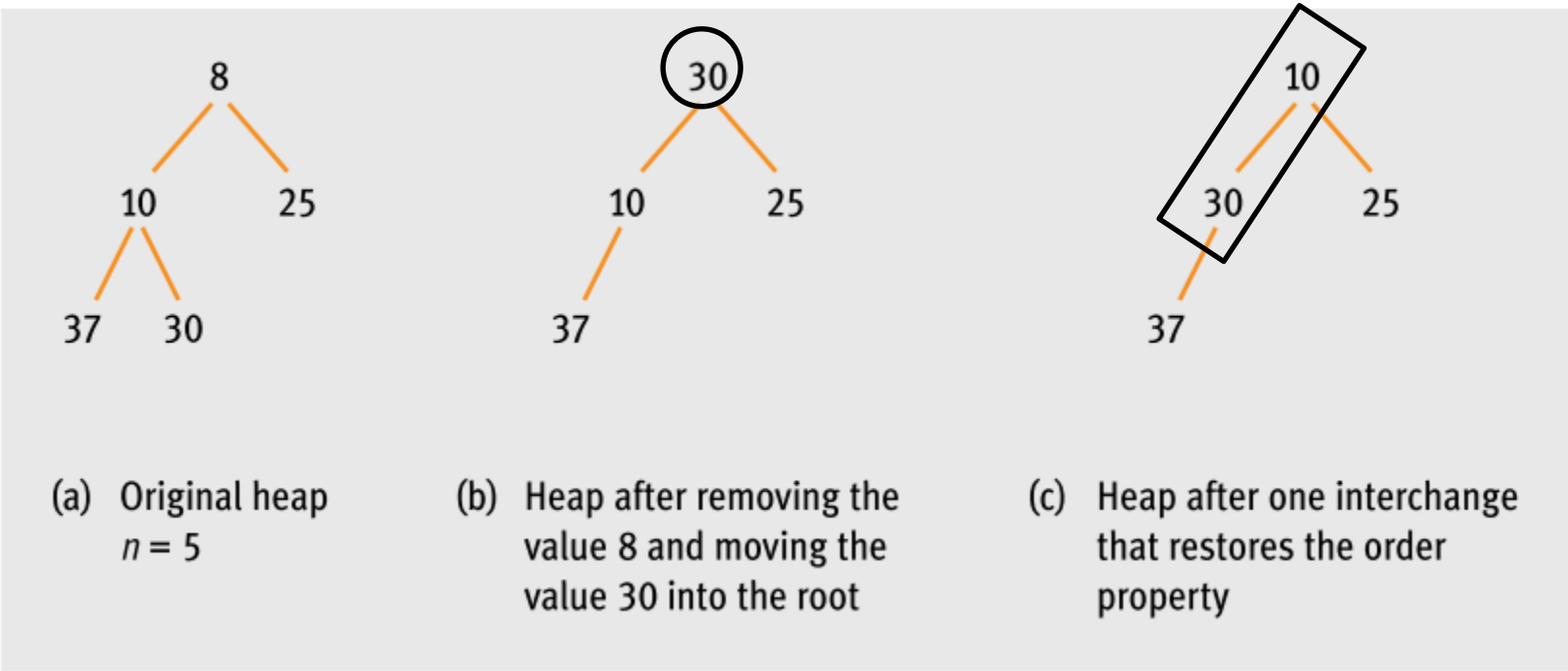
Inserting a node into a Heap



Deleting a node from a Heap



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

- *Note*: *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; }`

PQ: Additional Notes

- Two properties (insert / delete) have been discussed so far that are generally considered essential to define a priority queue. However, individual programmers may provide **additional commands**, including the following:

Scenario: hospital emergency room

- **decreaseKey**: condition of a person may worsen while waiting, his priority status will be improved
- **increaseKey**: increase the priority value and bring the element down on the binary tree
- **Remove**: An item may have to be removed. For example, the patient left the emergency area

PQ: Additional Notes

- **Create**: This is the priority queue's class constructor. It will create a new empty priority queue
- **findMin**() returns the item with the minimum key value, leaving item in the heap
- **isEmpty**() returns true if the heap is empty, false otherwise
- **size**() returns the number of items in the heap