CS 2100: Data Structures & Algorithms 1

Priority Queues / Binary Heaps

Binary Heap Operations (Insert; DeleteMin; findMin)

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- The University updated the mask policy. As per my Request on Mar 28, 2022 (see Collab), I would greatly appreciate if you would do me a kind favor by **continuing to wear your masks** in CS 2100 (Ridley G008). I know it is a lot to ask, and it is **voluntary**, but I appreciate your understanding.
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Binary Heap Operations

Methods: insert, deleteMin, findMin)

Binary Heap Operations

- **push(T data)**: Add data to priority queue
  - Uses percolate up
  - Sometimes priority is given OR data is comparable
  - *(term comes from Java documentation)*
- **poll()**: Remove next priority item
  - Uses percolate down
  - *(term comes from Java documentation)*
- **peek()**: just look at the **root** node

Binary Heap: push(T data)

- Basic Idea:
  - Put data at "next" leaf position
  - Repeatedly exchange node with its parent if needed
- Example Implementation:
  - Assume that the heap is represented by a **vector**, and `heap_size` is the number of *heap elements* inserted into the vector (it is NOT the capacity of the vector)

```
push(data) {
    // a vector push_back will resize as necessary
    heap.insertAtEnd(data);
    heap_size+=1

    // move it up into the right position
    percolateUp(heap_size);
}
```

[pseudo-code] Implementation: Heap = vector

```
public void push(T data) {
    heap.add(data);
    percolateUp(this.size());
}
```

Implementation: Heap = ArrayList

Binary Heap: percolateUp(int index)

- Place a new item in the right-most position in the bottom-most level (without introducing gaps). We have appeased the SHAPE property. Now let's look at the order property (`percolateUp(index)` method to place the item in the right spot.)

```
percolateUp(int index) {
    if(index <= 1) return;

    int pIndex = Floor(index/2);

    if(heap[index] < heap[pIndex]) {
      swap(index, pIndex);
      percolateUp(pIndex);
    }
}
```

- Note, **`pIndex`** is for storing the index of the parent node so we can **compare** its value with the new item's value (to determine if a swap is needed).

Complexity:
Inserting (push) a node into a Binary 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** ***(worst case):*** **O(log *n*)**

Complexity (another view): Inserting (push) a node into a Binary Heap

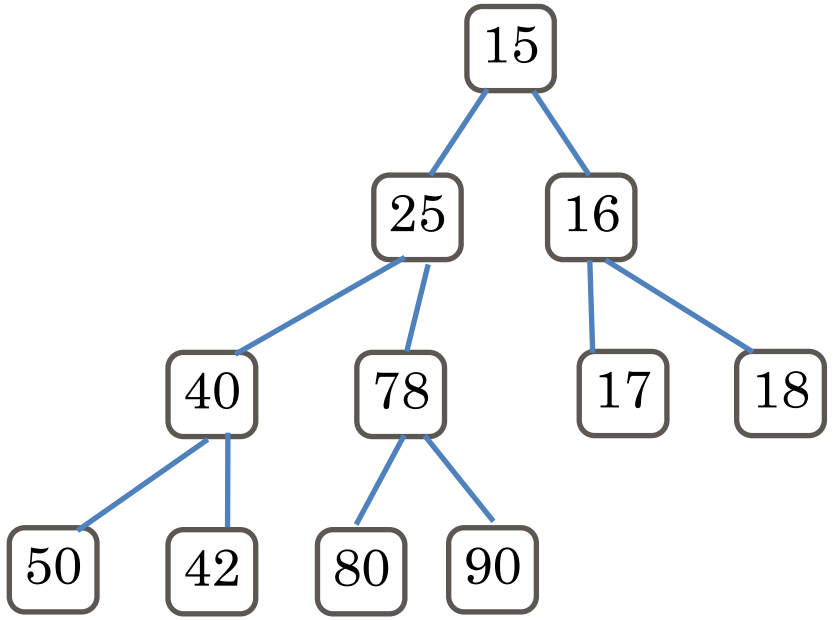
- Expected running time:
  - In a binary tree, the number of elements per level doubles (1, 2, 4, 8, …)
  - *How far do elements actually move up the Binary Heap?*
    - **Half** of the nodes are leaves, so half of the inserts perform 1 check (and remain at the bottom / leaf level)
    - **A quarter** of the nodes are one level above the leaves, so ¼ of the inserts will perform 2 checks (first check causing a swap, and second check requiring no swap) moving up two levels
    - **One eighth** of the nodes are two levels above the leaves, so 1/8 of the inserts will perform 3 checks (swap/swap/no swap) moving up three levels
    - **One sixteenth** … will require moving up four levels
    - Expected running time: $\frac{1}{2} * 1 + \frac{1}{4} * 2 + \frac{1}{8} * 3 + \cdots = \text{Sum}[\left(\frac{1}{2^i}\right) * i] = 2$
- **Time complexity *(in practice)*:** Weighted average of sum approaches 2 = **Constant time!**

Complexity (another view) for Inserting (push) & Discussion

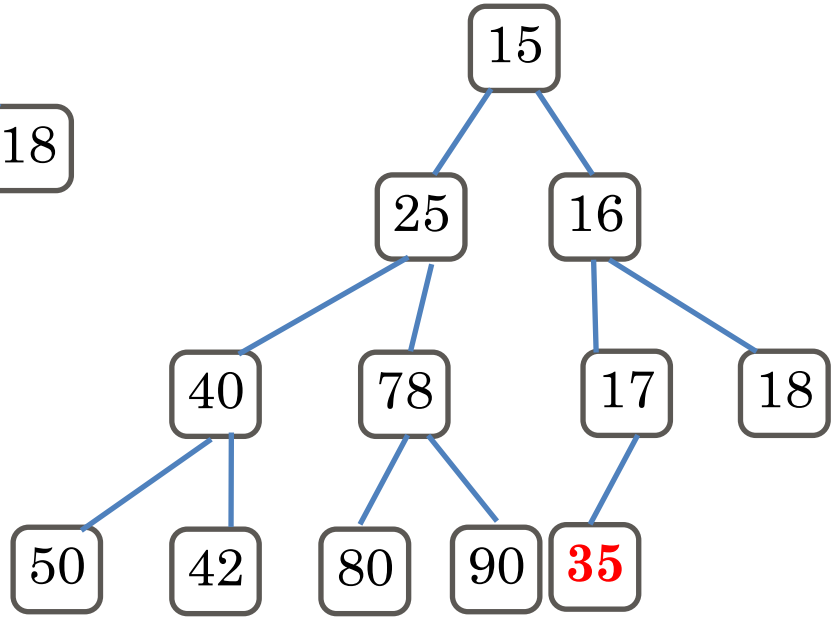
- The worst-case scenario is **O(log n)**
- However, in practice it is **constant** – approximately 2 checks per insert!

- Discussion:
  - *For your analysis assignment don't be surprised that your insert/push method behaves like it is **constant time** even though technically its complexity is O(log n)* ☺
  - *What might the **worst-case** scenario be? Adding in descending order (large -> small) every new element is the smallest element in the heap! (travels all the way to the top every time). In reality, adding the smallest element in the heap is **a rare occurrence**.*

Adding to a Binary Heap (push and percolateUp)

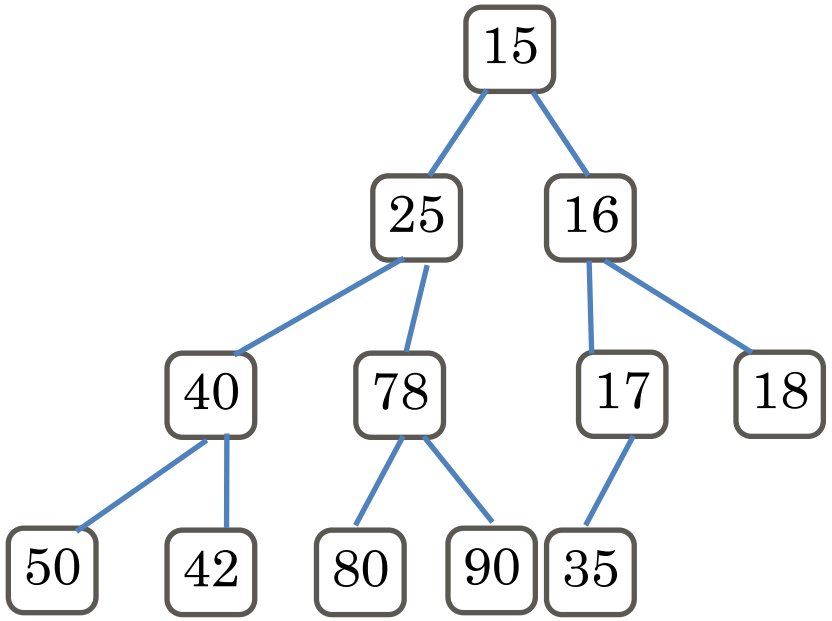


Let's push **35** into the Binary Heap

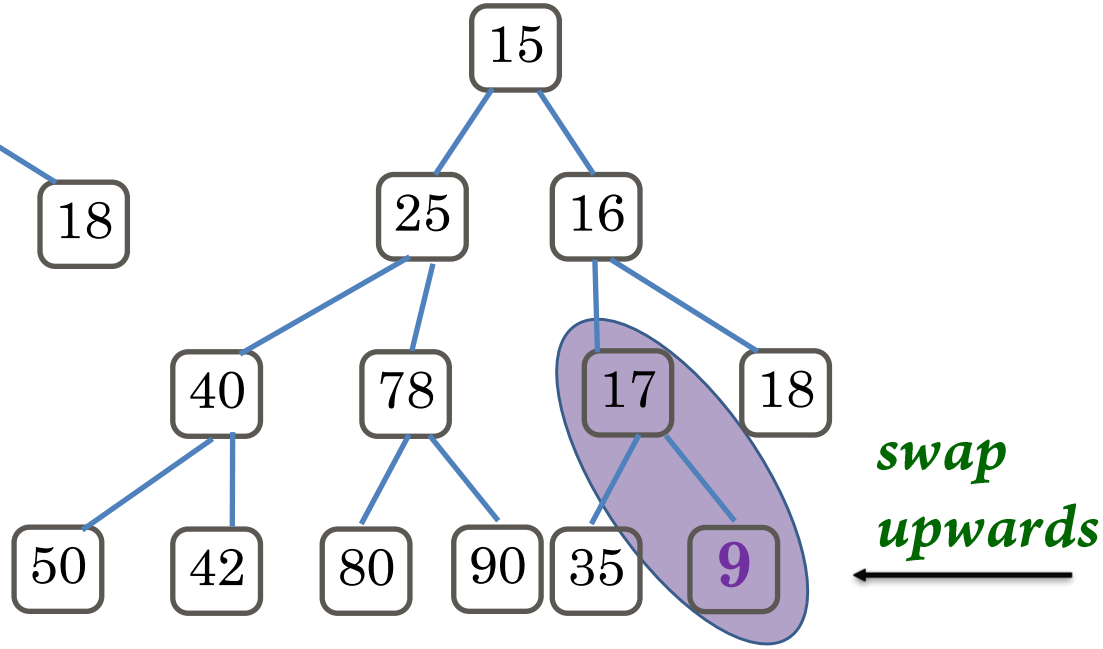


The heap properties are satisfied, nothing to re-arrange.

Adding to a Binary Heap (push and percolateUp)



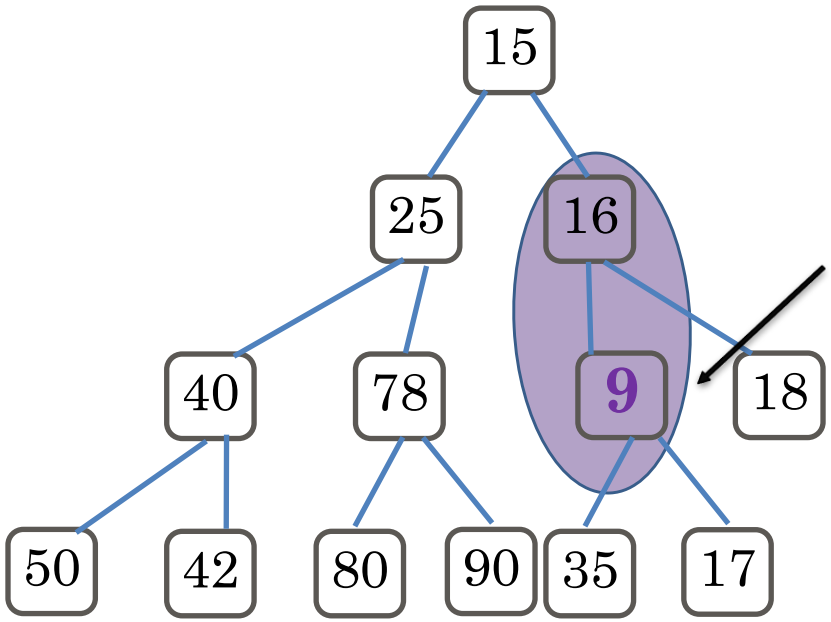
Let's push **9** into the Binary Heap



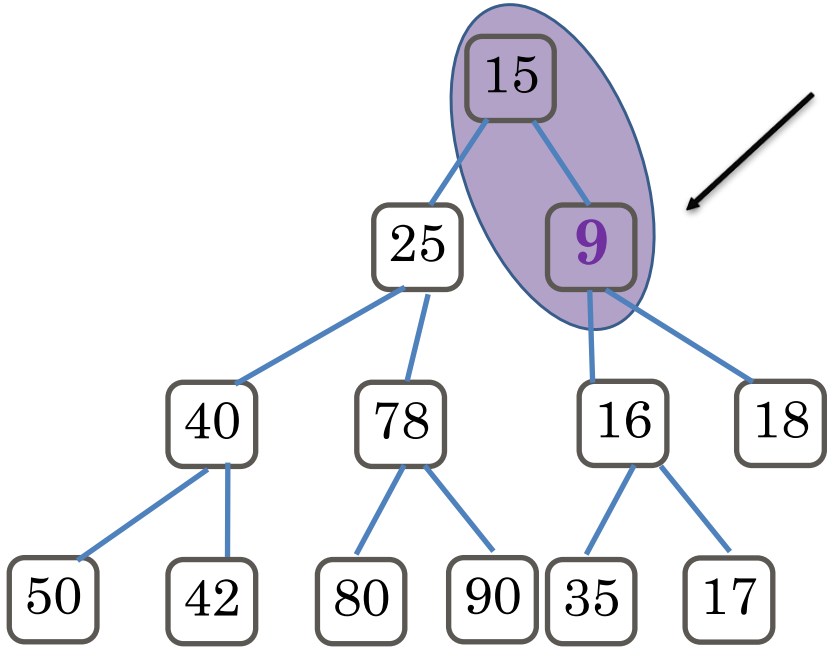
swap upwards

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

Adding to a Binary Heap (push and percolateUp)



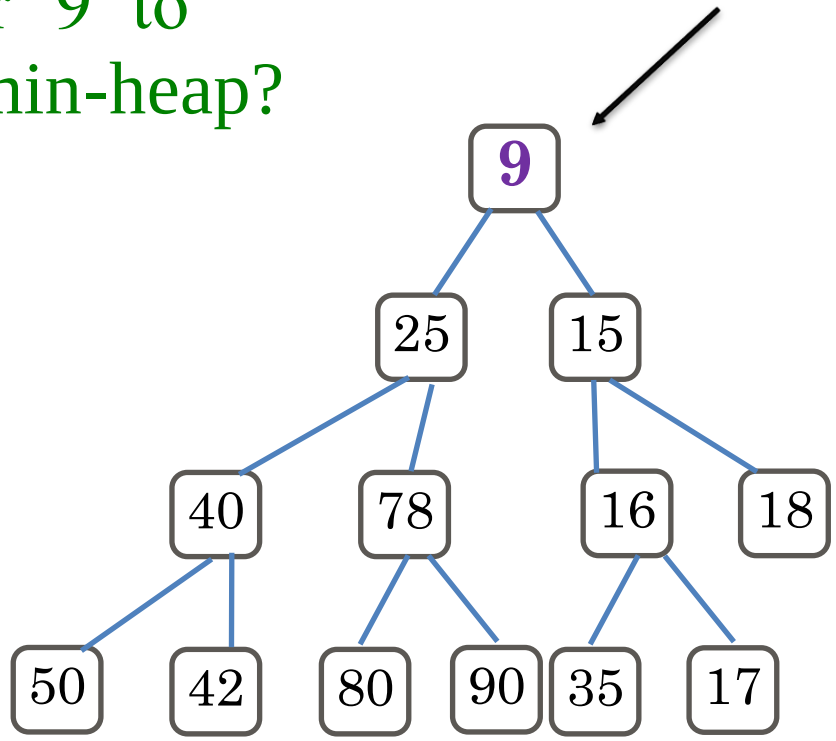
16 cannot be a parent to 9
(16 > 9)



15 cannot be a parent to 9

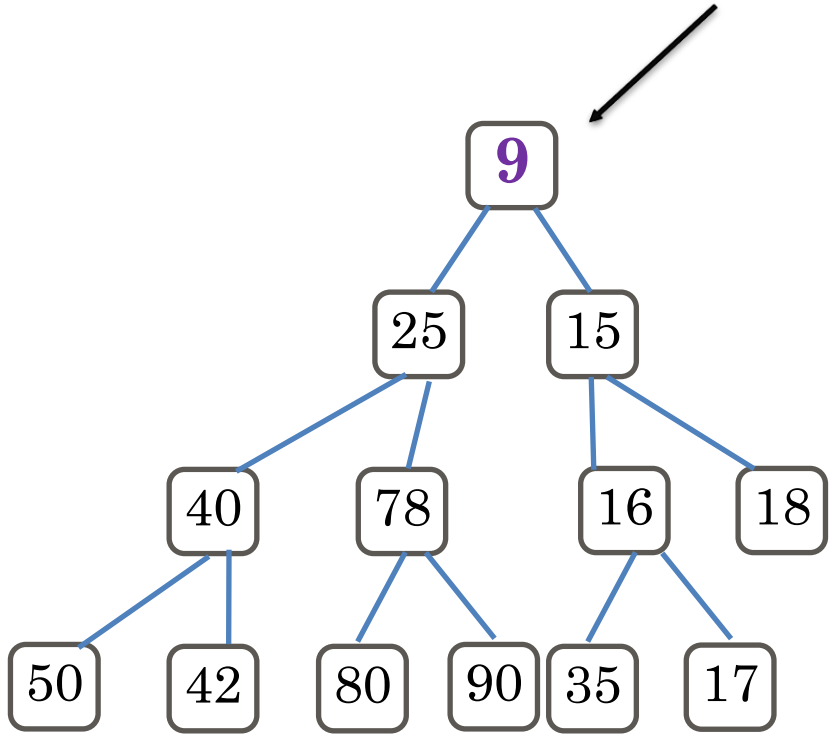
Adding to a Binary Heap (push and percolateUp)

- What does it mean for '9' to rise to the top of the min-heap?



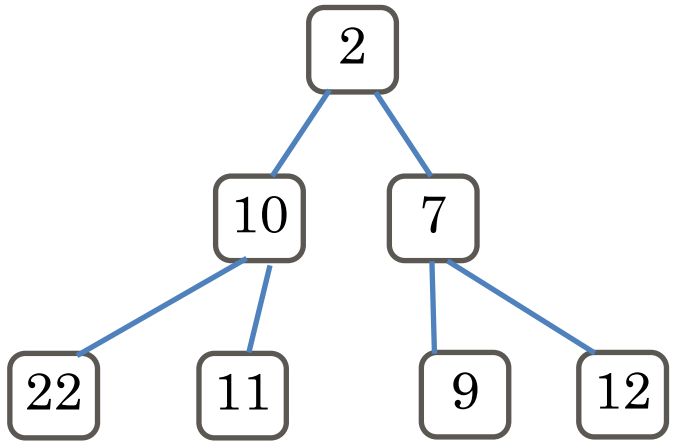
Adding to a Binary Heap (push and percolateUp)

- 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!
  (Generally, a rare occurrence)



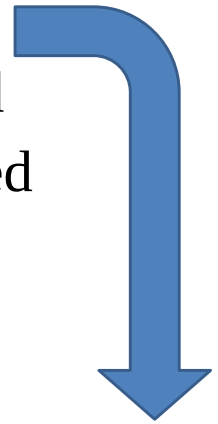
Exercise For You (push and percolateUp)

Let's push **5** into the following Binary Heap

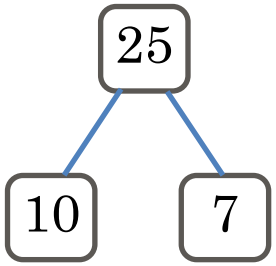


Binary Heap: poll()

- Basic Idea:
  1. Remove root (that is always the min!)
  2. Put "last" leaf node at root (maintains shape property)
  3. Find smallest child (why?)
  4. Swap node with smallest child if needed
  5. Repeat steps 3 & 4 until no swaps needed



Consider this min-heap. 25 needs percolating. What do you do? Which child node do you swap with?



Swap with the SMALLEST child!

What happens if we swap with the 10?

Binary Heap: poll() – Pseudo-code

- Remove from the **root**
- Decrement size
- Replace the root with the **last element** (size, not capacity); **delete** that leaf
- `percolateDown(index)` as needed

```
T poll() {
    // make sure the heap is not empty
    if ( heap_size == 0 )
        return null;

    int ret = heap[1];
    heap_size--;
    heap[1] = heap[heap_size];
    heap.removeLast();
    percolateDown(1);
    return ret;
}
```

Complexity: Deleting (poll) a node from a Binary Heap

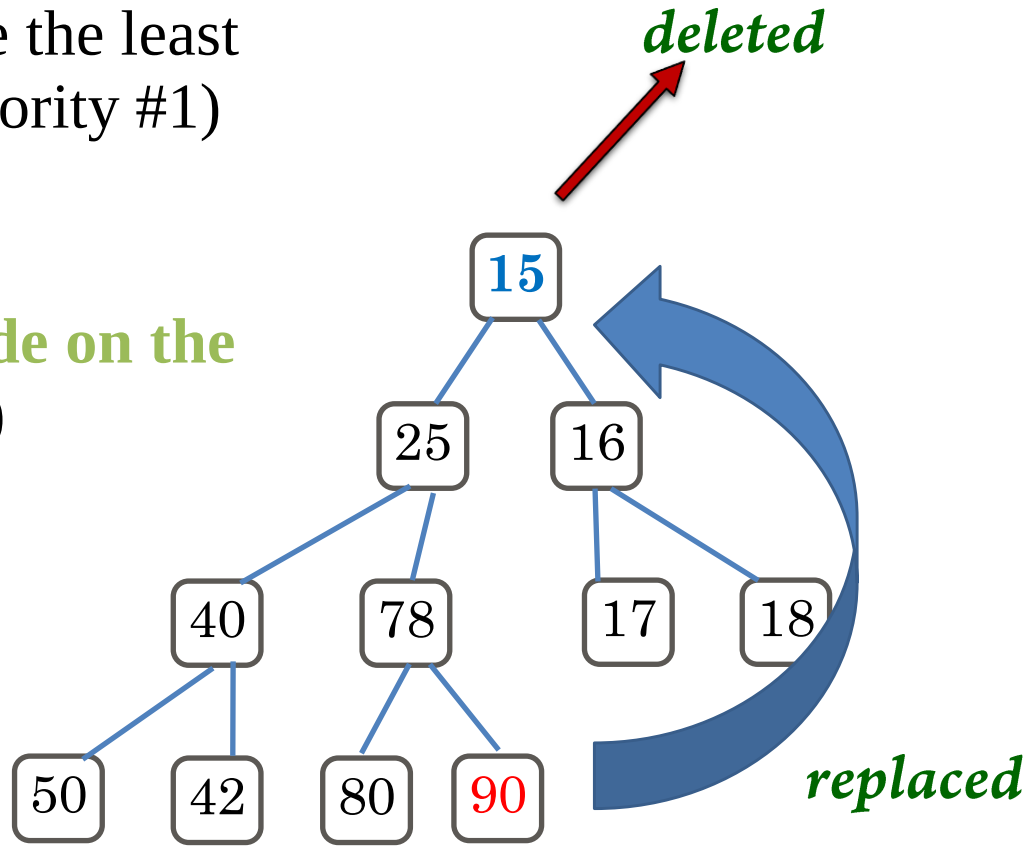
- Replace the **root** of the heap with the ***last element on the last level*** – *maintaining the* **shape property**
- Compare the new root with its children; if they are in the correct order, stop
- If not, **swap** the element with one of its children and return to the previous step. (Swap with 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]

Deleting (poll) a node from a Binary 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*)



poll() method: remove root!

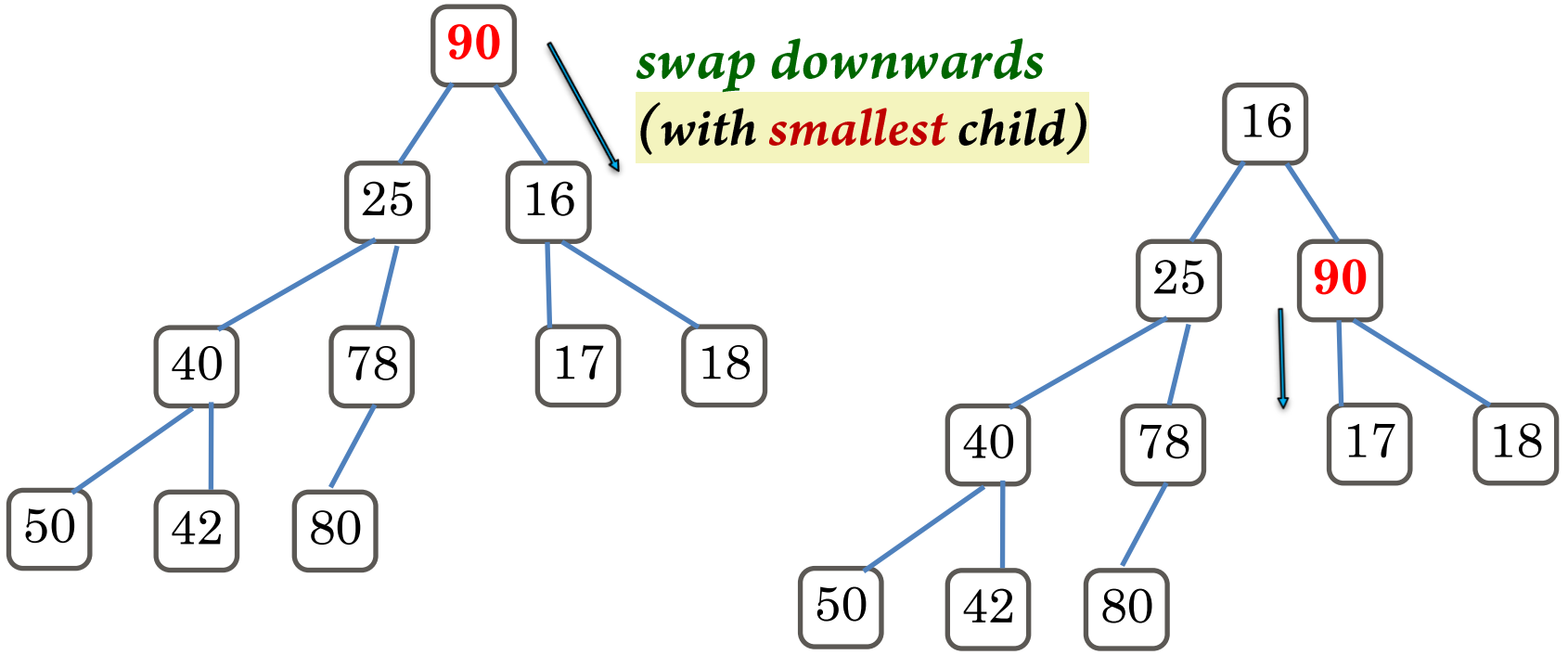
- Remember, when calling **`poll()`** you are NOT specifying which element to remove

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

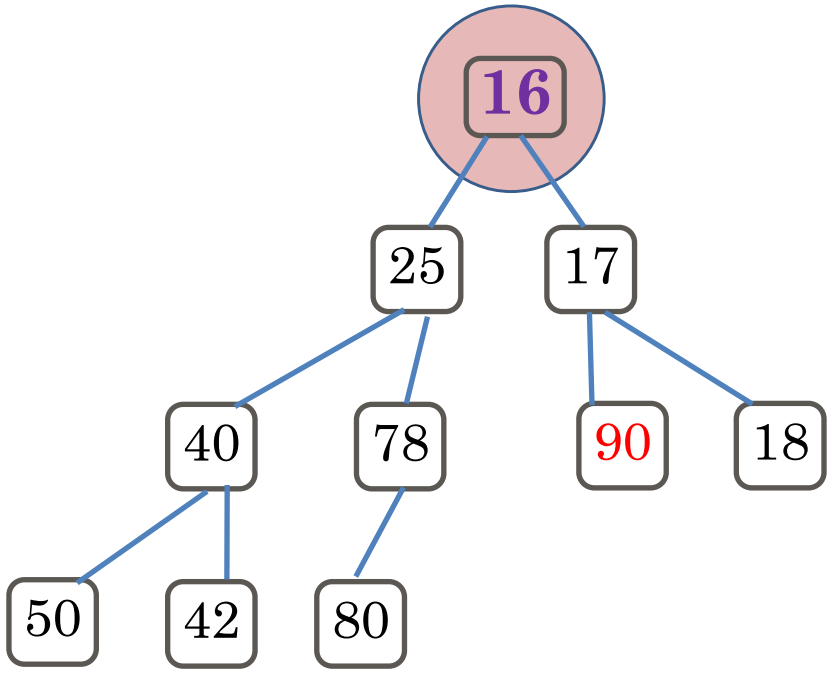
- So that the tree (heap) doesn't remain without a root node, replace it with **last node on the right** at the bottom level of the heap

Removing an element from a Binary Heap (poll and percolateDown)

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

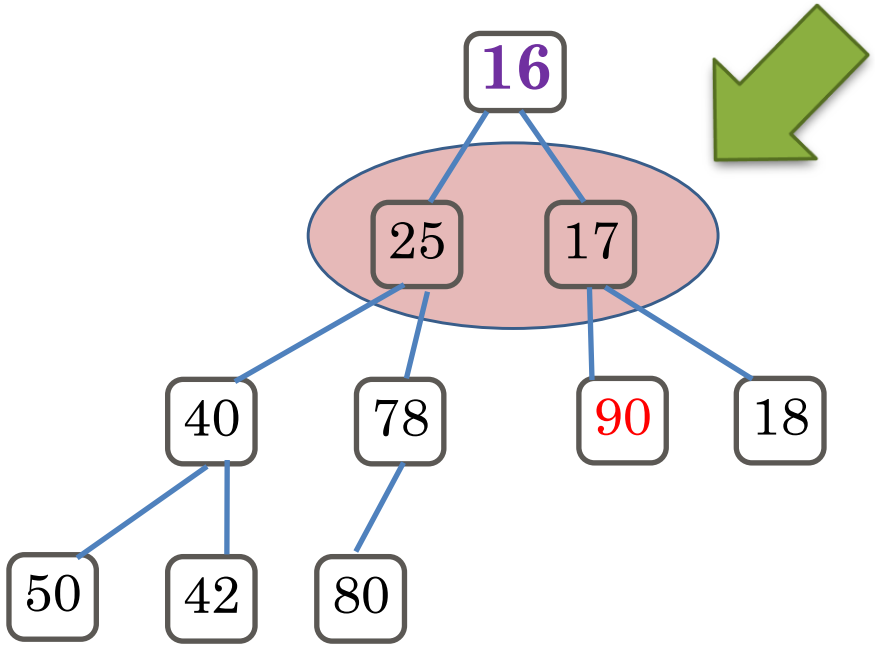


Removing an element from a Binary Heap (poll and percolateDown)



- Notice 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 Binary Heap (poll and percolateDown)



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

Other Possible Heap Operations

- `decreaseKey(processID, amount)`: find, "*raise*" the priority of a process, percolate up
- `increaseKey(processID, amount)`: find, "*lower*" the priority of a process, percolate down
- `remove(processID)`: find, remove a process, move to top, then delete.

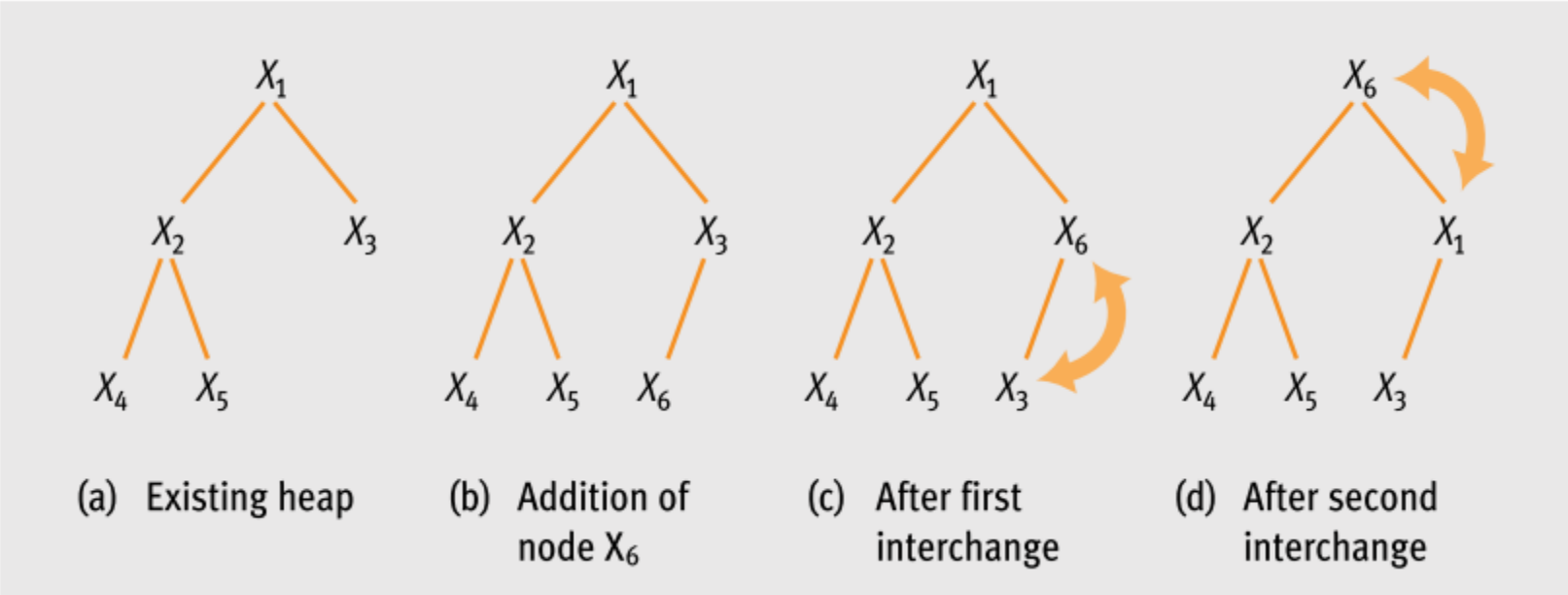
- Worst-case running time for all: **Θ(n)**, because for all these methods `find()` operation is required. *(In a priority queue, no natural way to find an element quickly)*
- What about `FindMax`?
  - *Create a maxHeap instead! Or remove everything until you remove the last element! Or in the vector, linearly search for the largest element in the lower half of the vector (usually where you'll find the largest element – not near the root/top)*
- `ExpandHeap`: when heap fills, copy into new space.

Heaps (Summary & Complexity)

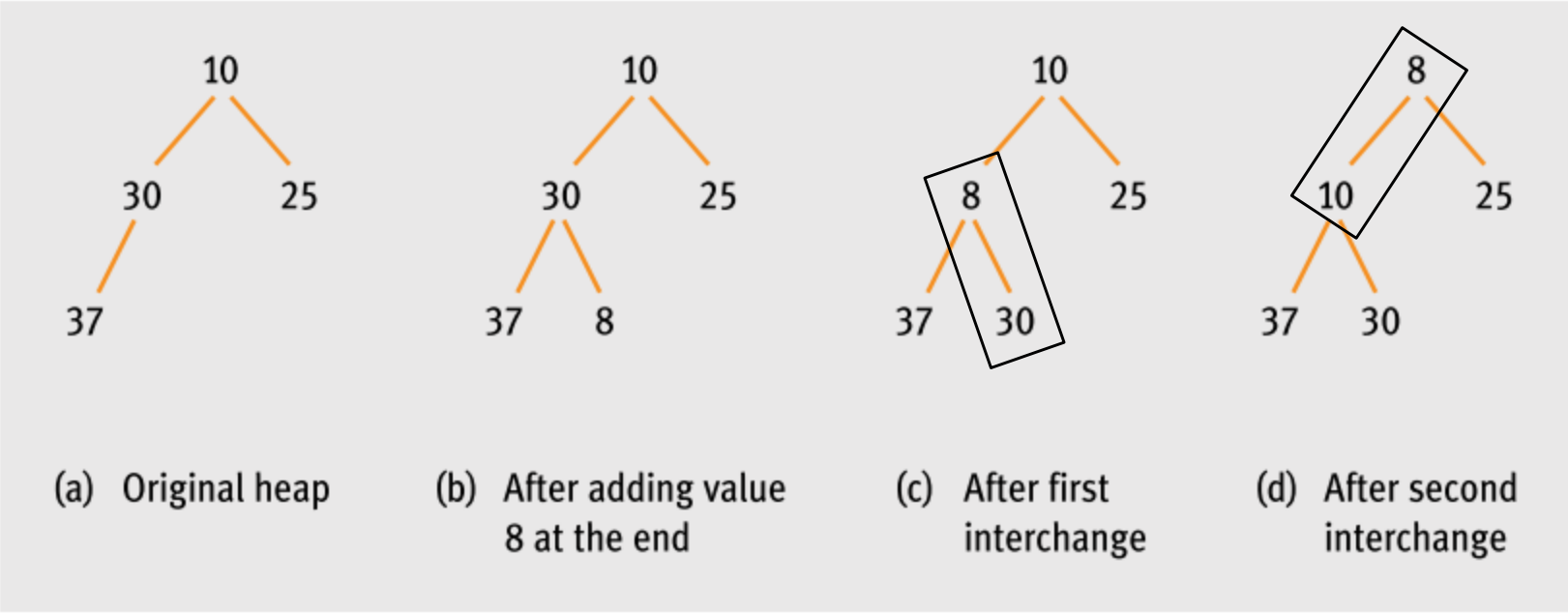
- **push**: percolate up; Θ(log n) time worst case,
  but constant expected time
  - Less likely to be the smallest value, so average work is 2 checks = constant time
- **poll**: percolate down; Θ(log n) time worst case;
  also logarithmic expected time
  - Due to replacing the root with the last element in the vector/list, it is going to be a large value, so you will percolate the value back down the heap taking log(n) time
- **peek**: Θ(1) time
  - Just looking at the root value, takes constant time to read the element at index position 1 heapVector[1] ➔ where the root is stored

Additional Slides

Inserting a node into a Heap



Inserting a node into a Heap



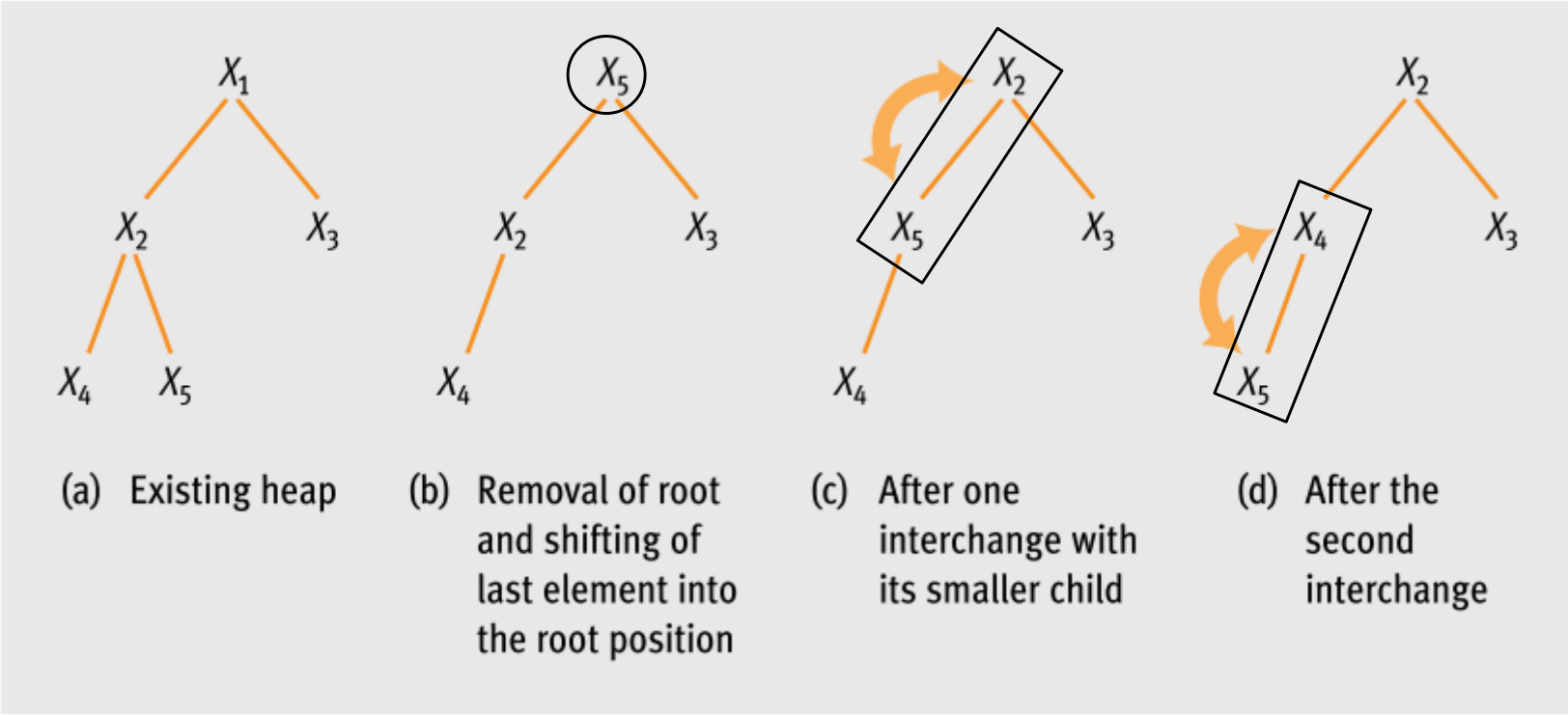
(a) Original heap

(b) After adding value 8 at the end

(c) After first interchange

(d) After second interchange

Deleting a node from a Heap



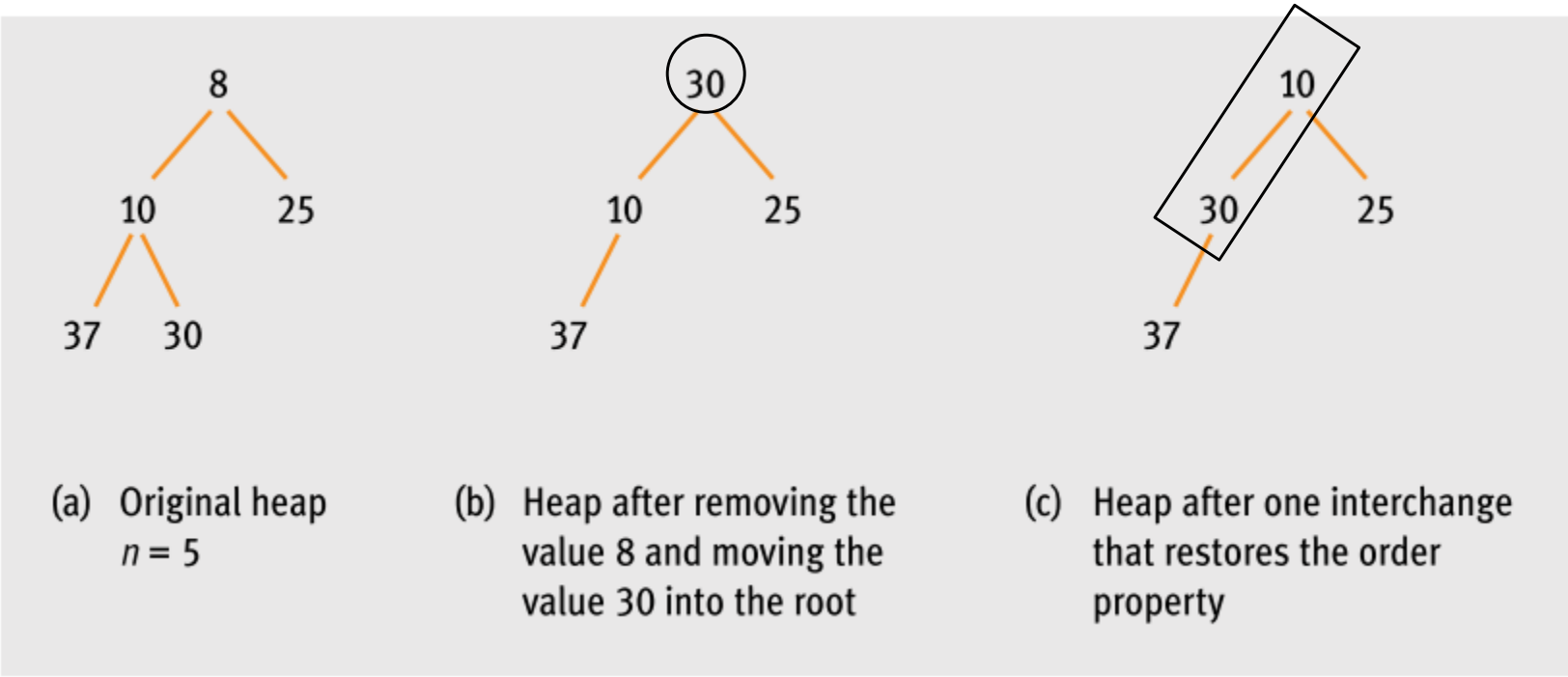
(a) Existing heap

(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

- *Note*: *next* ***smallest*** *element is now at the root!*