Prioritizing: Priority Queues & Heaps

Priority Queues

Priority Queues

- Instead of FIFO — “Best Out”
- Typically, low priority numbers are removed first from PQs
  - think of “I’m number 1!” — certainly better than being 2 or 3…
  - I think Nelly did a song about this…
  - https://www.youtube.com/watch?v=Vt-96_byt68

Priority Queues: Why?

- There are *hundreds* of applications of priority queues.
  - high-priority print jobs
  - various priority levels of jobs running on a time sharing system
  - picking which philosopher can pick up forks (sharing/fairness) :)
  - finding shortest paths
  - sorting
  - etc.

Priority Queues: MinPriorityQueue.java

- Simple interface for a minimum priority queue.
- Operations:
  - isEmpty() — determine whether PQ is empty
  - insert() — insert item into PQ
  - minimum() — return smallest item (leave it in the PQ)
  - extractMin() — return smallest item and remove from PQ

Priority Queues: Notes

- Max priority queue works in a similar way but everything is done with respect to items with max. priority rather than min. priority.
- Java has the java.util.PriorityQueue class which is based on *heaps* (later). The operations are named a little different in some cases:
- Operations:
  - isEmpty —> isEmpty
  - insert —> add
  - minimum —> peek
  - extractMin —> remove
- Comparable vs. Comparator interfaces (see book: pg. 363-364)
  - natural ordering vs. arbitrary/custom ordering

ArrayList Implementations

Unsorted ArrayList Implementation

- The simplest way to implement a min-priority queue: ArrayList whose elements may appear in any order
- See: **ArrayListMinPriorityQueue.java**
- isEmpty() — returns boolean indicating whether size is zero
  - $\Theta(1)$
- insert() — add at end
  - $\Theta(1)$ (amortized time)
- minimum() & extractMin() — use indexOfMinimum() to find/extract
  - $\Theta(n)$
- **Disadvantage**: minimum & extractMin() take $\Theta(n)$ time…

5	9	1	6	10	2	4	3	8	7

Sorted ArrayList Implementation

- We can get improvements if we keep list sorted
- See: **SortedArrayListMinPriorityQueue.java**
  - NOTE: sorted in decreasing order so min item is last
- isEmpty() — returns boolean indicating whether size is zero
  - $\Theta(1)$
- insert() — move backwards through list and look for spot to insert
  - $O(n)$
- minimum() & extractMin() — min located at last item
  - $\Theta(1)$
- How can we improve this?!

5	9	1	6	10	2	4	3	8	7

Heaps

But first, a joke

So a String asks an int on a date.

The int is like, "Hell no."

So the String asks why.

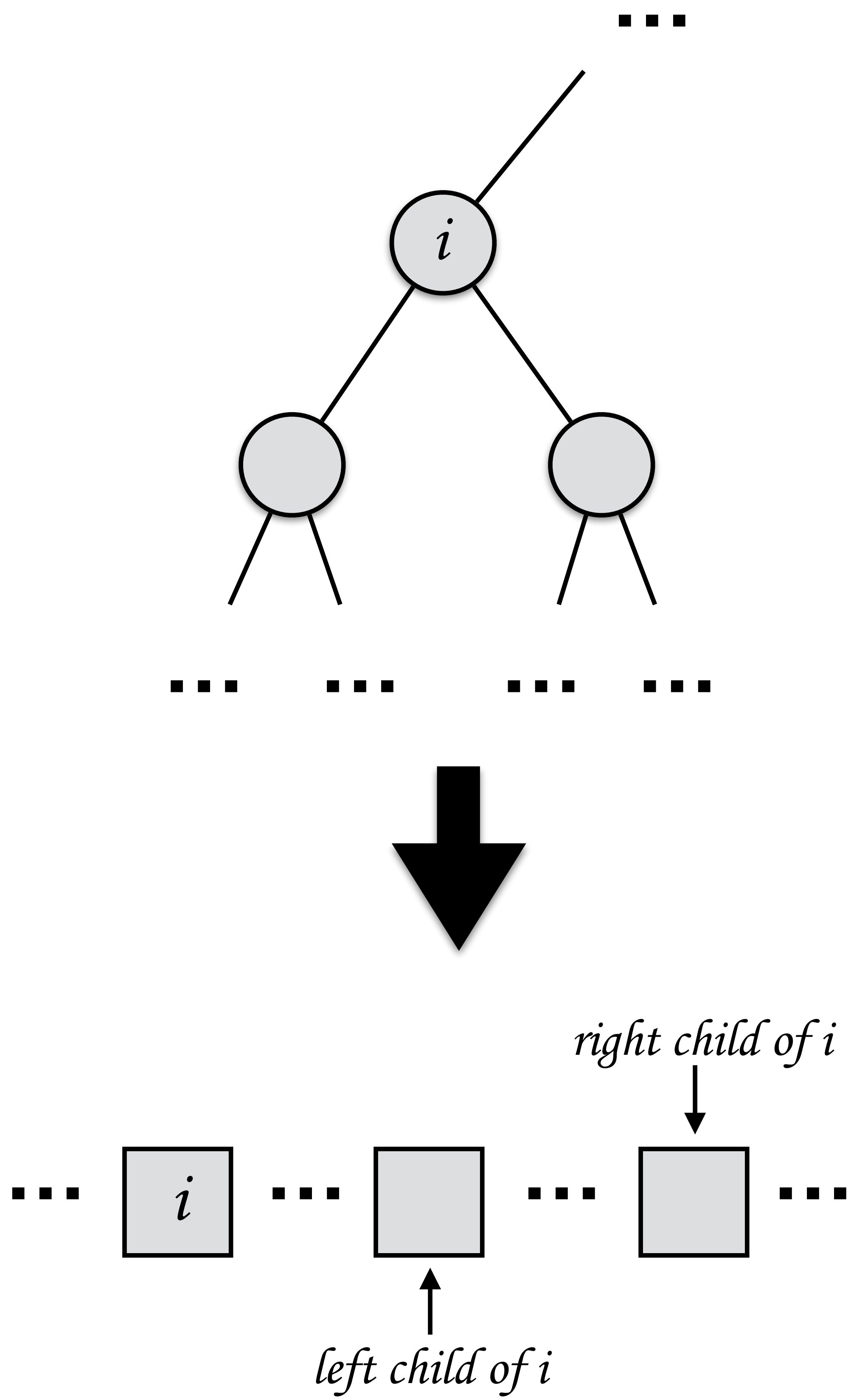
The int says

"You're not my type."

- Anonymous CS 10 Student

Heaps

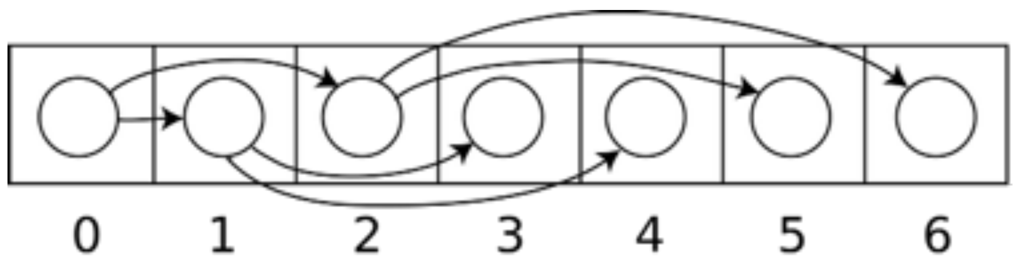
- Node above *i* is the parent
- Nodes below *i* are the children of
- Connections between nodes: edges
- A binary heap is stored an *array* and can be viewed as a "nearly complete" binary tree
- Shape Property:
  Fill the tree from the root down toward the leaves, level by level, not starting a new level until we "complete" (fill) the previous level.
  (i.e., no empty holes in the array!)
- Representation as an array
  - makes storage compact
  - accessing parent/children can be done with simple index arithmetic



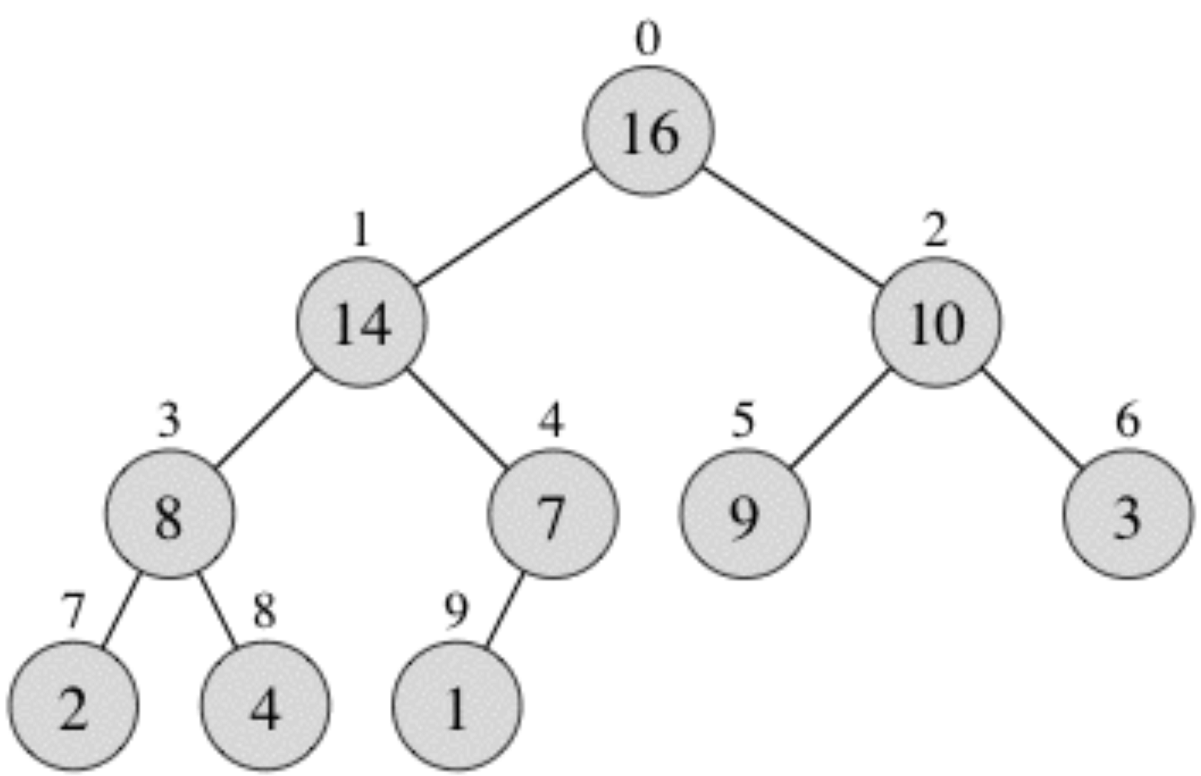
Heaps: Example (Max-Heap)

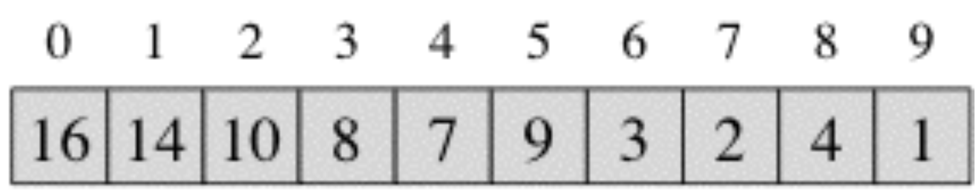
It's easy to compute the array index of a node's *parent*, *left child*, or *right child*, given the array index *i* of the node:

- Parent is at index ***(i-1)/2*** (using integer division).
- Left child is at index ***2*i + 1***.
- Right child is at index ***2*i + 2***.



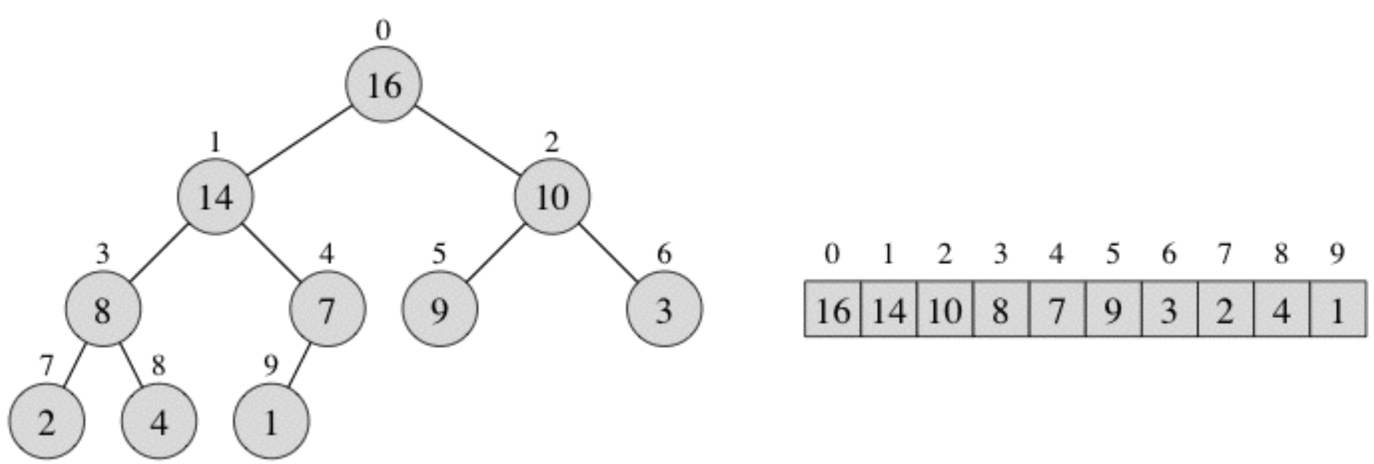
public domain: wikipedia.org





Max-Heap

- 2 types of heaps: max-heap and min-heap
- In a **max-heap**, nodes satisfy the **max-heap property:**
  For every node i other than the root, the value in the parent of node i is greater than or equal to the value of node i.
- I.e., node *i* is **at most** the value in its parent.
- The largest value in a max-heap must be the root
- Since a subtree consists of a node and all of the nodes below it, the largest value of the subtree must be the root of that subtree.
- The example (previous slide) we looked at before is a max-heap.



Min-Heap

- In a **min-heap**, nodes satisfy the **min-heap property:**
  For every node i other than the root, the value in the parent of node i is less than or equal to the value of node i.
- I.e., node *i* is **at least** the value in its parent.
- The smallest value in a min-heap must be the root

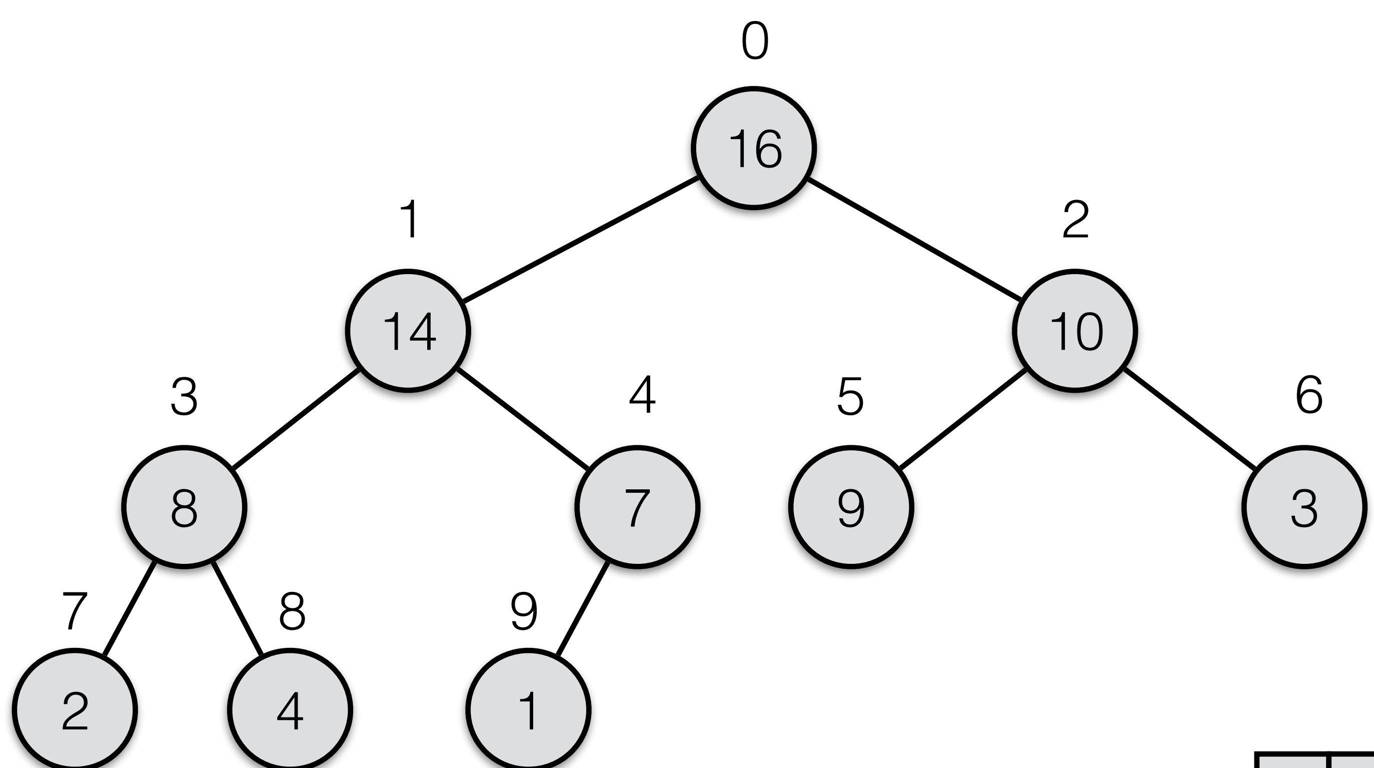
Height of a Heap

- The ***height*** of a heap is the # of edges on the longest path from the root down to a leaf.
- Recall that a leaf is a node with no children.
- The height is the greatest integer less than or equal to **lg n**.
- Formal proof of this result on pg. 371 of the textbook.

Heaps: Inserting

Heaps: Inserting

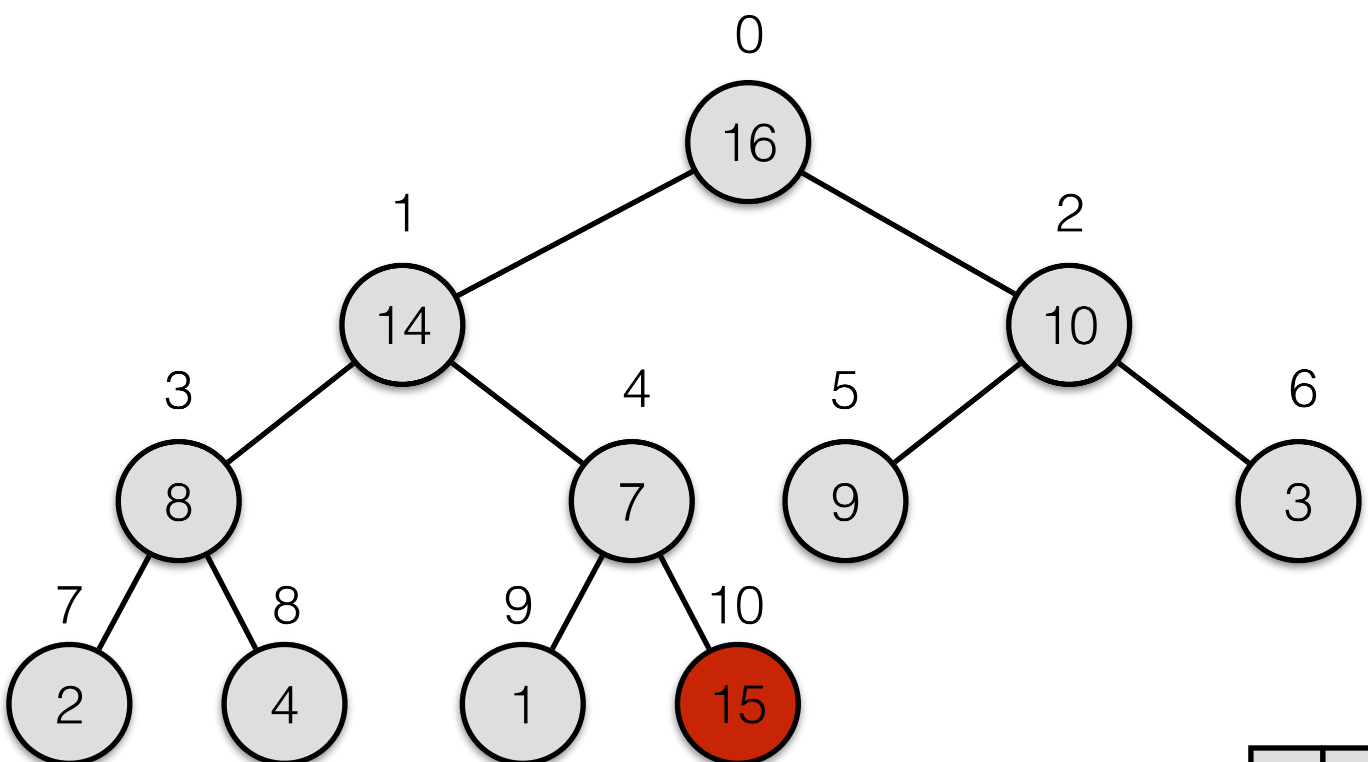
- Suppose we want to insert **15** into the heap shown below
- Q: Where should it go? Why?



16	14	10	8	7	9	3	2	4	1

Heaps: Inserting

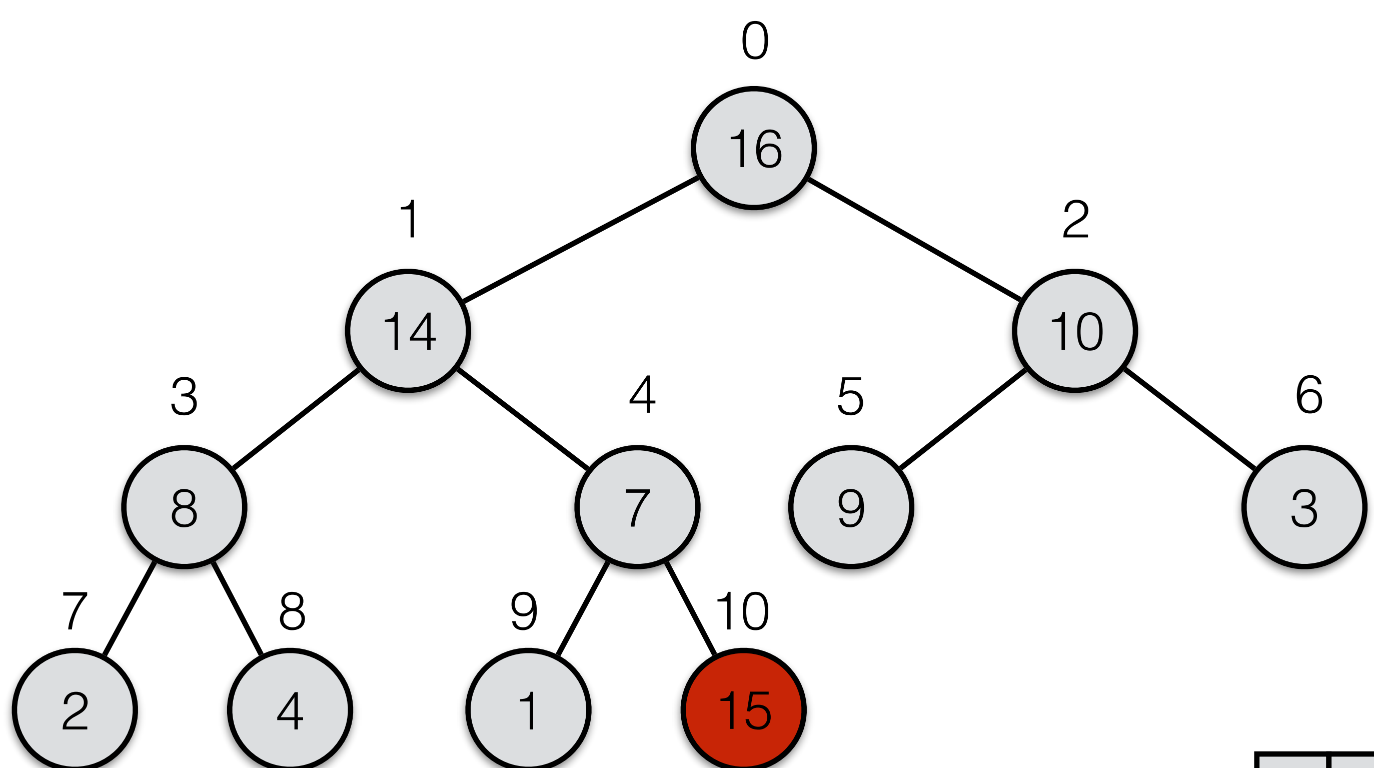
- The **shape property** is satisfied by putting the new node in the next open position of the binary heap — this really is the last position in the array since our binary heap is *actually* stored as an array.



16	14	10	8	7	9	3	2	4	1	15

Heaps: Inserting

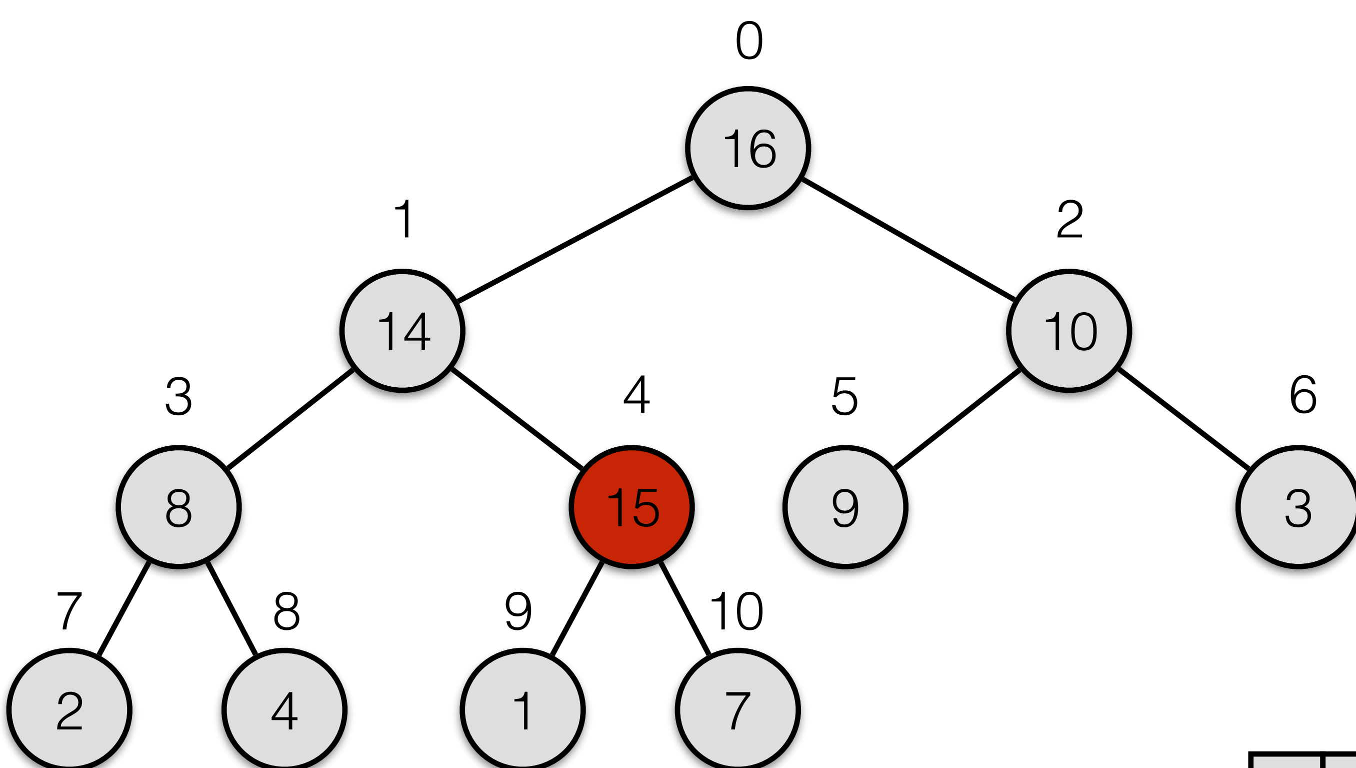
- Q: Is the **heap property** satisfied?
- Everything is fine with the possible exception that the newly inserted item might be greater than its parent.
- Easy fix: if new item is less than parent —> swap!



16	14	10	8	7	9	3	2	4	1	15

Heaps: Inserting

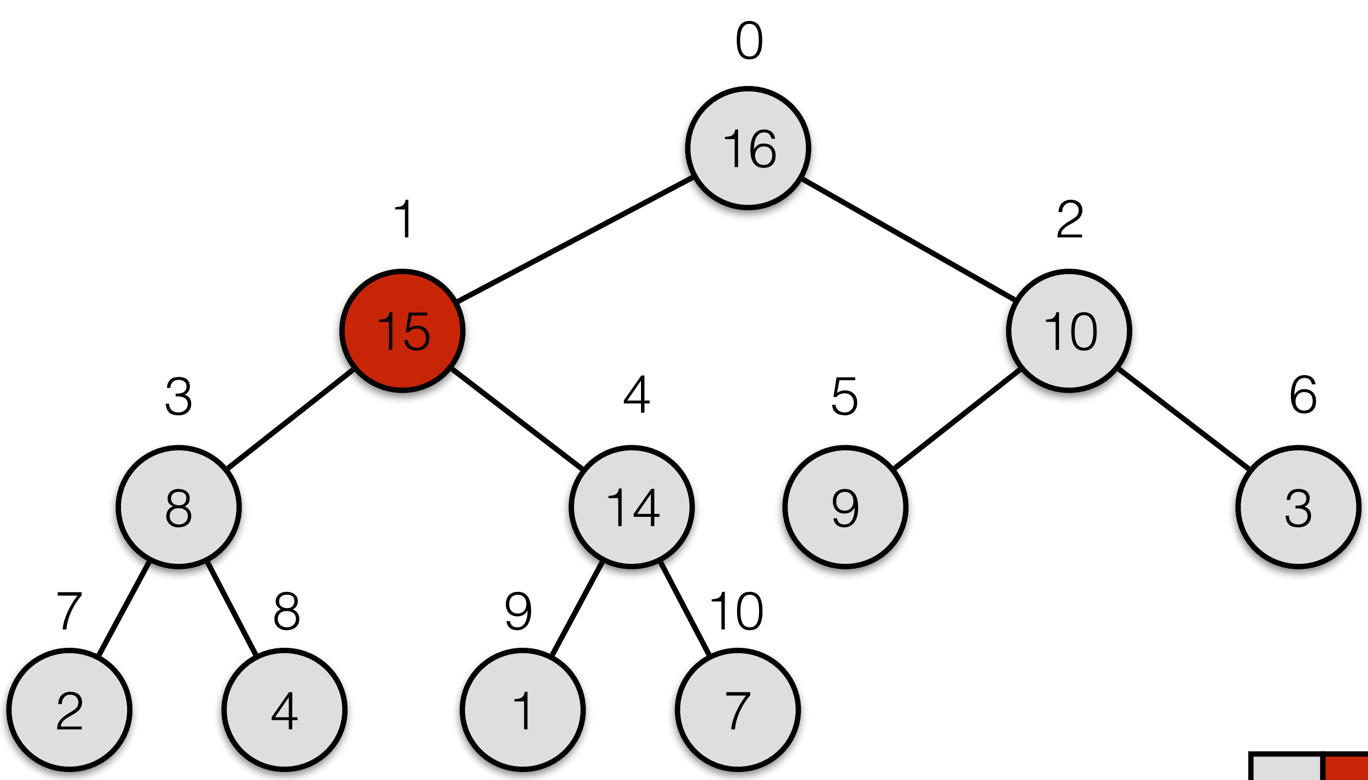
- Again, their may be an issue with the new item and its parent, so we have to keep working up and swapping as needed…



16	14	10	8	15	9	3	2	4	1	7

Heaps: Inserting

- Keep going until the newly inserted item is less than or equal to its parent or reaches the root.



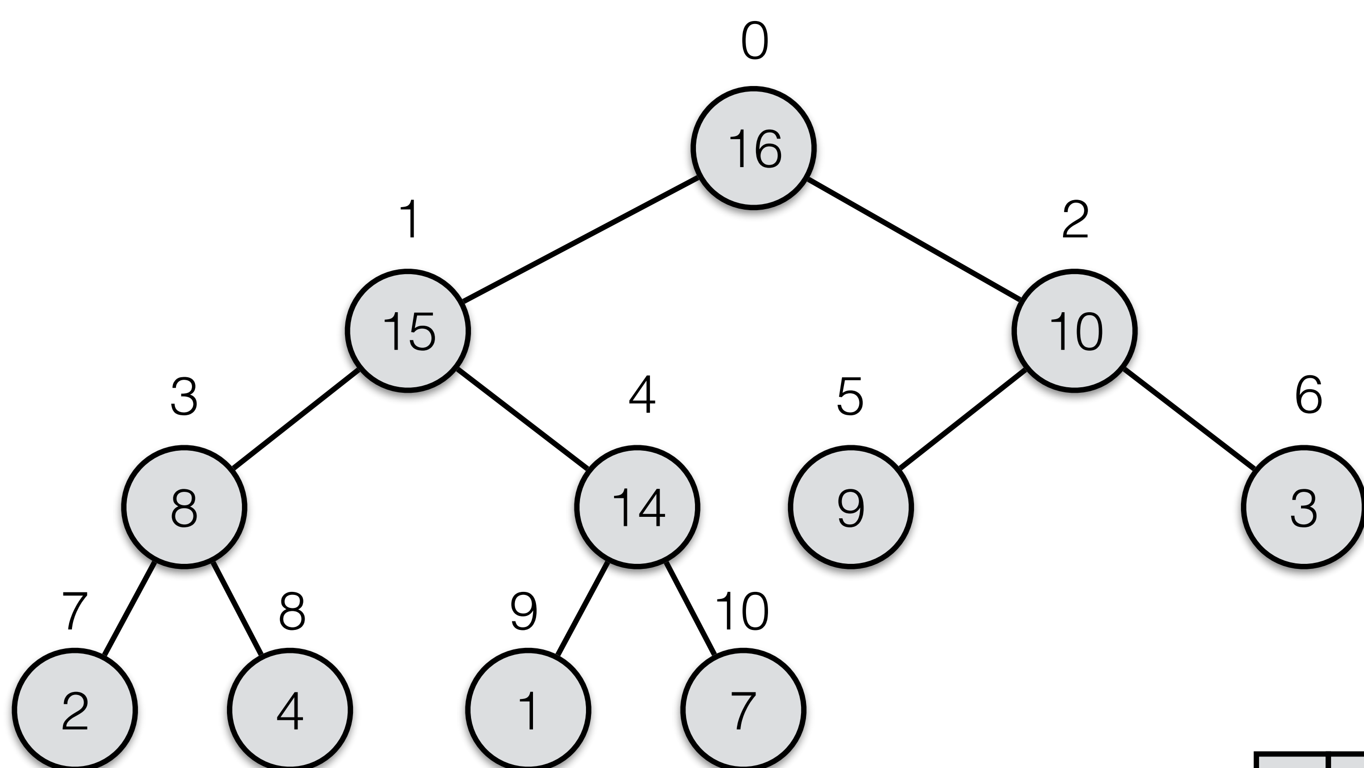
Done!

16	15	10	8	14	9	3	2	4	1	7

Heaps: Deleting

Heaps: Deleting

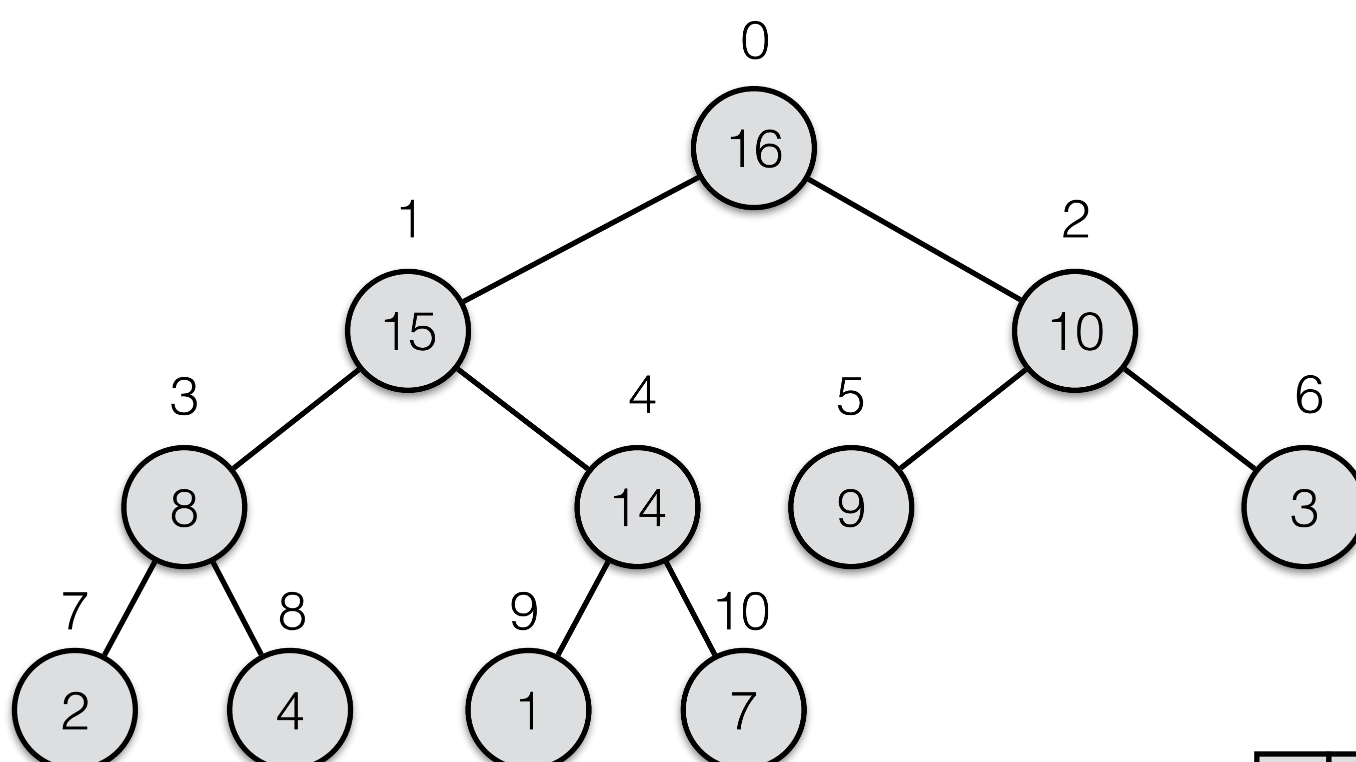
- Suppose we want to delete the max now
- We know the max is at the root (max-heap) — i.e., index 0 of array
- Simply removing item at index 0 leaves a hole, which is not allowed…
- Also, the heap has one fewer item, so the rightmost leaf at the bottom needs to disappear.



16	15	10	8	14	9	3	2	4	1	7

Heaps: Deleting

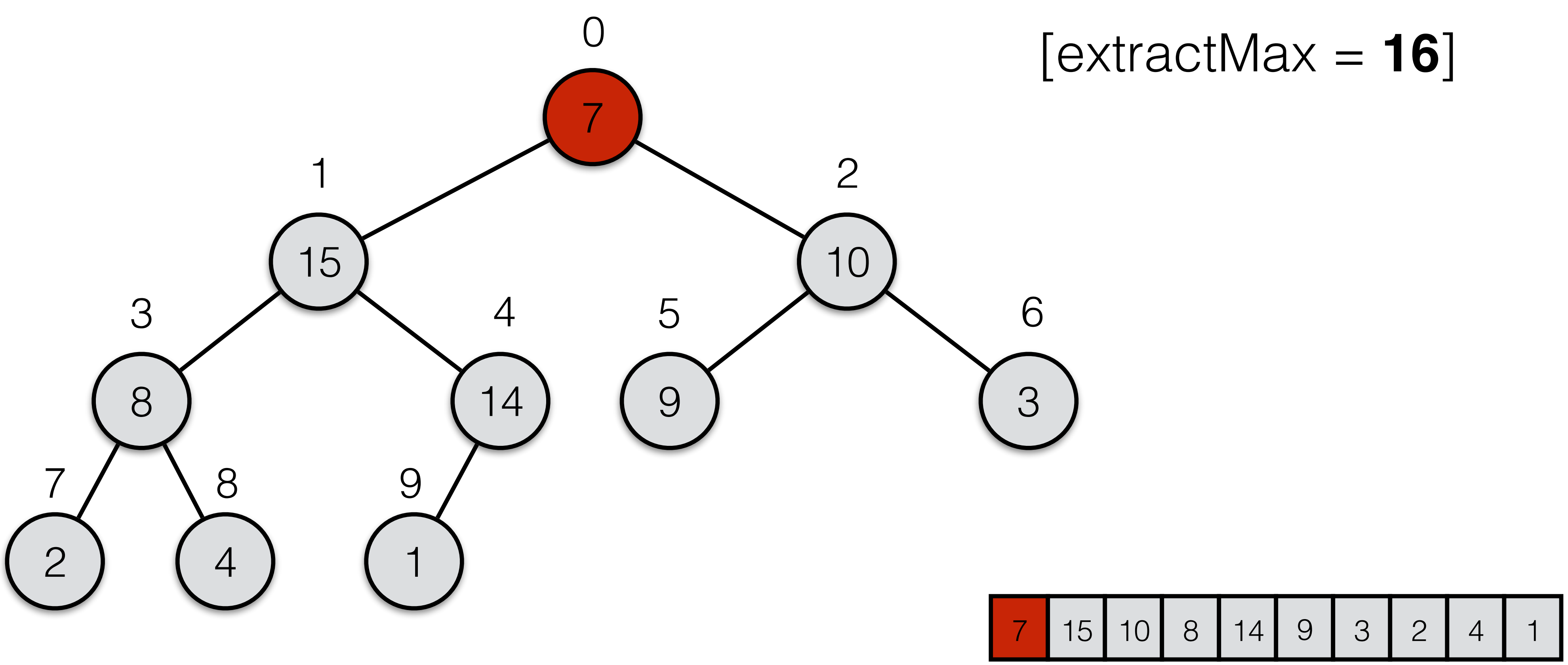
- Q: How to delete while maintaining the *shape* property?



16	15	10	8	14	9	3	2	4	1	7

Heaps: Deleting

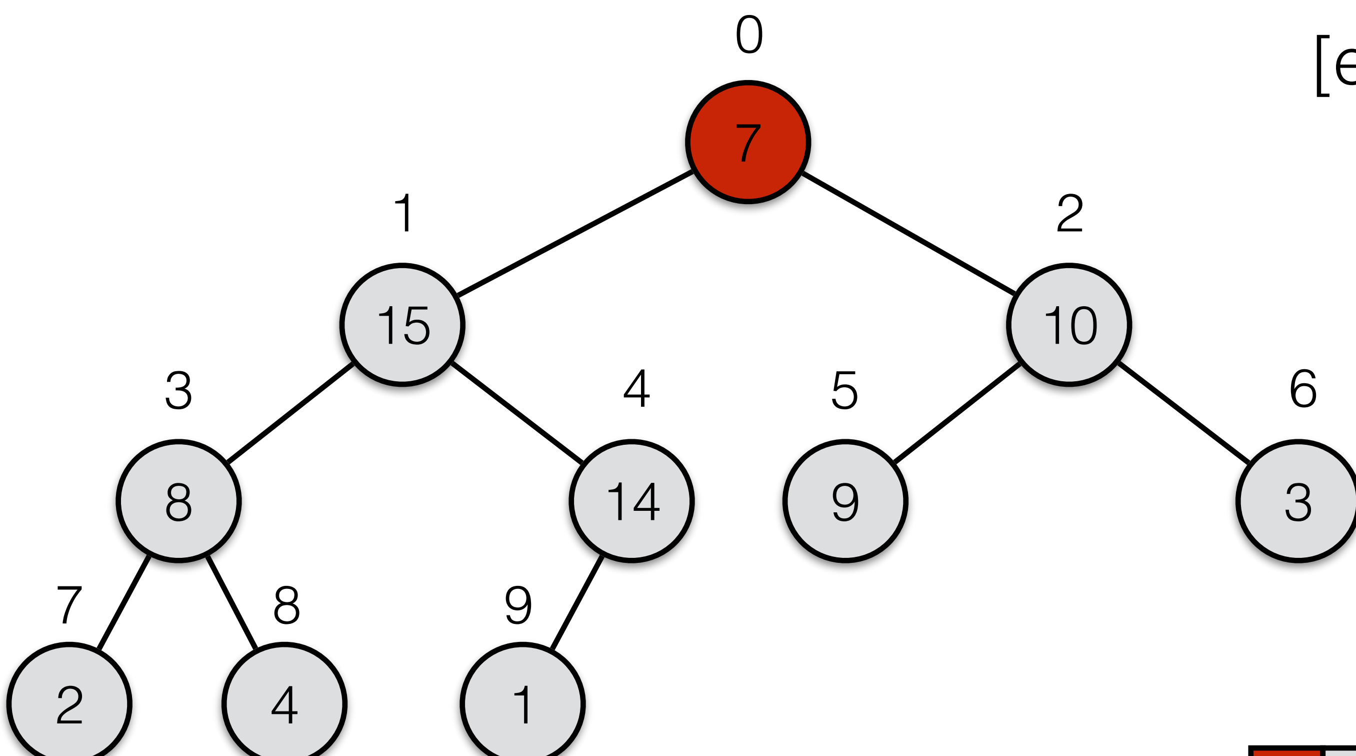
- Solution: extract the max, then move the bottom rightmost leaf to the root and decrement the size of the occupied portion of the array.
- Addresses previous concerns with **shape property**.



Heaps: Deleting

- Q: What about the **heap property**?
- Left and right subtrees are both valid heaps…
- But the root might be smaller than one or both of its children…
- Q: What to do?!

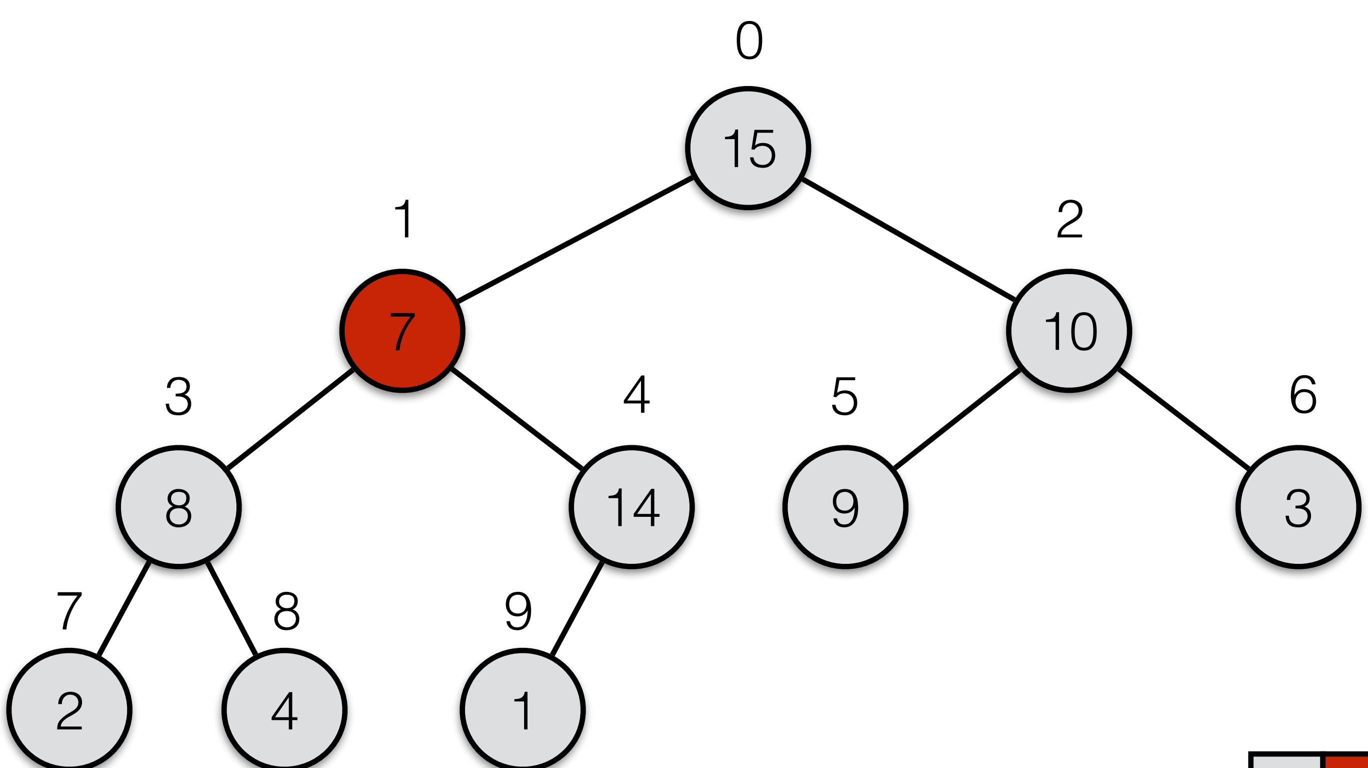
[extractMax = **16**]



7	15	10	8	14	9	3	2	4	1

Heaps: Deleting

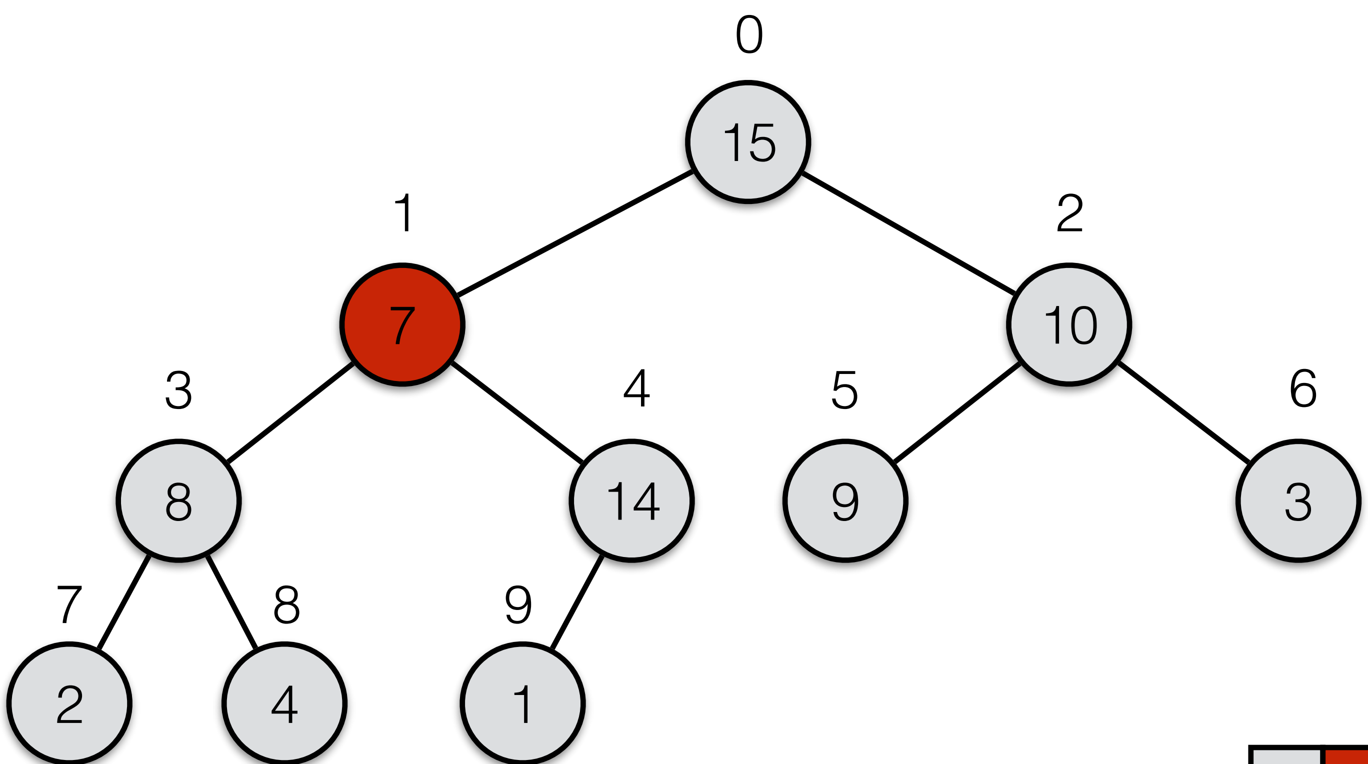
- Solution: swap the root with the **larger** child.
- Larger child is bigger than (1) the current root, (2) everything in its subtree, *and* (3) the smaller child of the current root (and thus everything in its subtree).



15	7	10	8	14	9	3	2	4	1

Heaps: Deleting

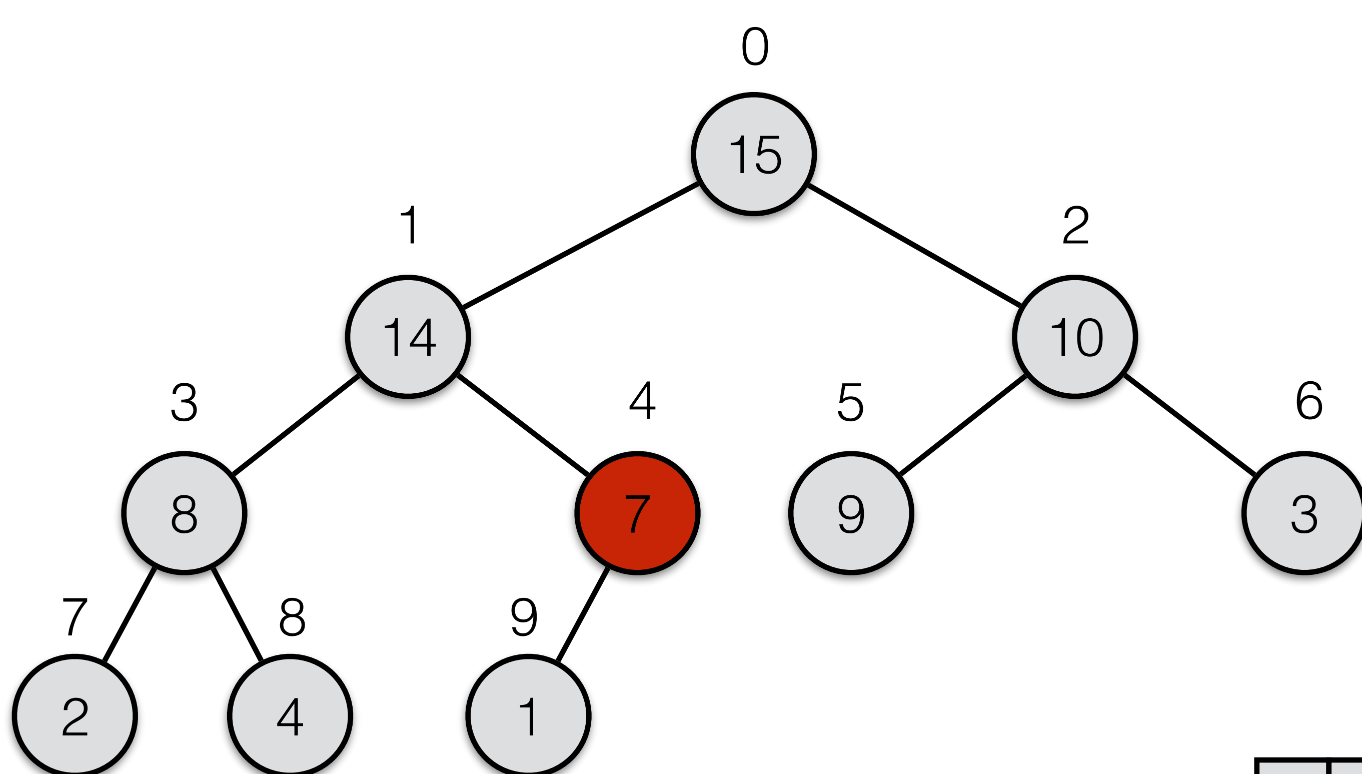
- Now we’ve moved the problem down (i.e., newly swapped item may violate heap property in the new subtree)!



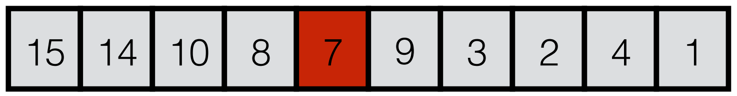
15	7	10	8	14	9	3	2	4	1

Heaps: Deleting

- But that is okay and easy to fix — it is just the same problem we had before. Repeat the process of moving the item down until (1) it is a leaf, or (2) it is larger than its child or children.



Done!



Heapsort

[See: Heapsort.java & Heapsort.ppt]

Heapsort

- In addition to supporting priority queues, heaps are also the basis for a sorting algorithm known as **heapsort**.
- Running time: $O(n \lg n)$
- Sorts in place!
  - no additional space for copy items (as mergesort does)
  - or for a stack of recursive calls (as quicksort and mergesort)
  - similar to selection sort
- Heapsort has two major phases:
  - build a heap
  - pick out max (root), swap with "last item", and restore heap property.
- Notice: as we remove the current max, it goes into a position in the array which is no longer included in the heap

Build a Heap

- Q: How to build a heap starting with an unordered array?
- **Okay Solution**: Repeated inserts:
  - first item is a valid heap
  - insert 2nd item, then 3rd, then 4th, etc.
  - after inserting the last item, we have a valid heap.
  - this works and leads to *O(n lg n)* heapsort.
- **Better Solution**: use code to restore heap property after deletion
  - no need to implement separate “insert” code — use **maxHeapify()**
  - maxHeapify() takes three params
    - **a** — the array
    - **i** and **lastLeaf** — indicies into the array
  - valid heap is in subarray a[i … lastLeaf]
  - assume: heap property holds everywhere except possibly node i and its children
  - after maxHeapify() runs, it has restored the heap property everywhere

Build a Heap (cont.)

- How it works:
  - compute indices of left and right children (if they exist)
    - child exists if its index is <= lastLeaf
  - determine largest among the current node and its children
  - check to see if a swap is necessary (i.e., if one of the children was larger)

```
if (i == largest)
    done!
else
    swap and recursively call maxHeapify()
```

  - # of recursive calls is at most O(lg n) — i.e., the height of the tree
- Start at the bottom and work up!
- Notice: calling maxHeapify() on leaves changes nothing so we can skip those…
- **SEE: slides 1-17 in Heapsort.ppt**
- See: class notes for more detailed (and accurate/rigorous) discussion of runtime analysis.

Build a Heap (cont.)

See: **Heapsort.java**

- buildMaxHeap()
- maxHeapify()
- leftChild()
- rightChild()
- swap()

Sorting a Heap (After Building)

- **SEE: remaining slides (18+) in Heapsort.ppt**
- After building the heap, the largest element is the root
- Swap largest item w/ item in last position currently occupied by the last leaf in the heap
- After the swap, that last position is no longer part of the valid heap
- The swap may have caused violation of heap property — call maxHeapify() on root to restore
- Repeat until the only remaining item in the heap is the root — done!

<u>Runtime analysis</u>

- maxHeapify takes O(lg n) time — could run *n* times for a total of O(n lg n) time.
- Adding in the O(n lg n) time to build the heap gives a total sorting time of O(n lg n).

Sorting a Heap (After Building)

See: **Heapsort.java**

- sort()
- heapsort()