Binary Heap

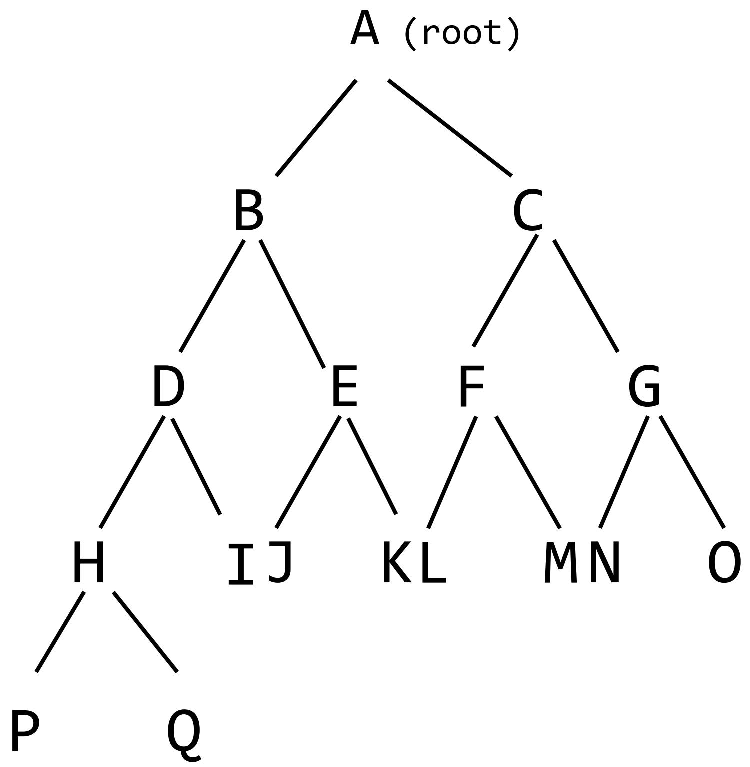
Binary Heap

- Binary heap is a commonly used data structure.
  - It is used in a priority queue to find the largest (or smallest) element.
  - It can be used to find the k^{th} largest(or smallest) element.
    - Note: there is a faster algorithm for finding k^{th} largest(or smallest) element (2nd course)
  - It is also used in heap sort algorithm.

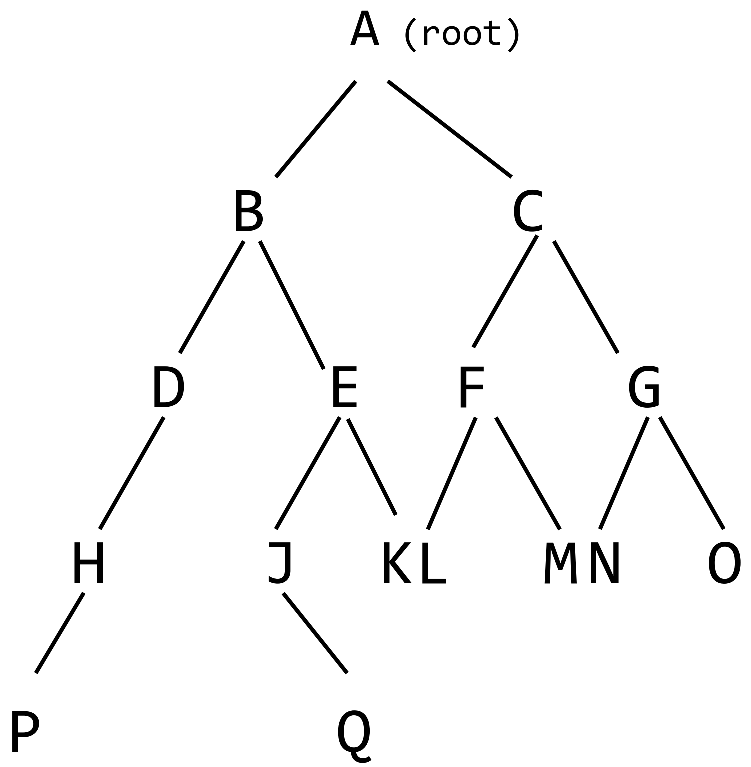
- What does a binary heap look like:
  - Binary heap is a *complete* binary tree (See next slide).
  - It satisfies the *heap ordering* property, which we will look at very soon.

Complete binary tree

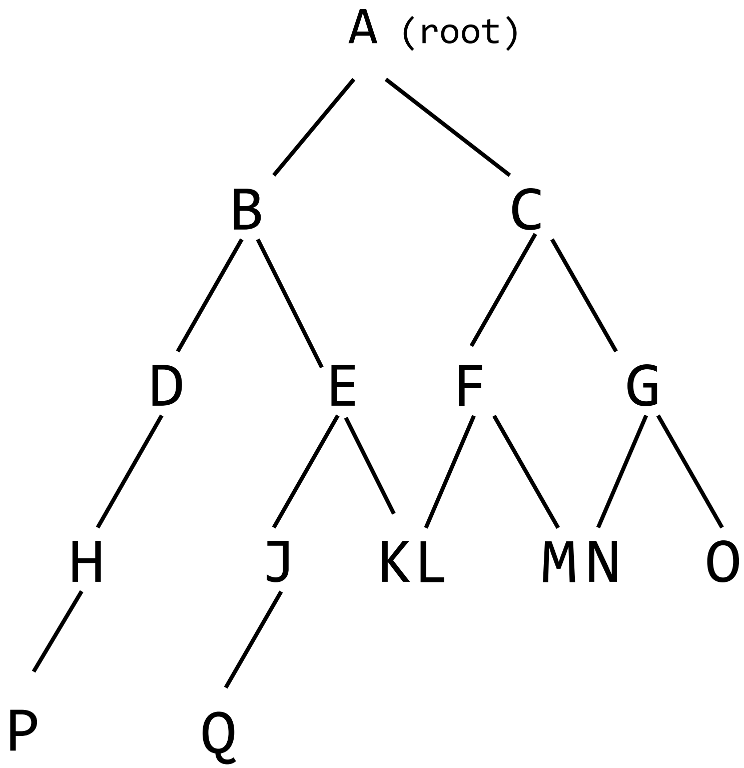
- A binary tree is *complete* if
  - all its nodes have two children,
  - with the **exception** of the deepest level,
    - which would be filled from left to right.
- In other words, any empty spots are only on the right side of the bottom level.
- All levels above the bottom level are fully occupied.
- A complete tree with N nodes will have height $\log_2 N$



Complete tree



Not complete



Not complete

Heap ordering property

- Binary heap satisfies the heap ordering property, which can be one of:

  1. *Min* heap:

  - Value of each node is >= value of its parent node.
  - So, given a node and its two children, the parent node value <= children node value.
  - Extending this property to the whole tree (i.e., the whole binary heap):
    - it means the **smallest** value in the binary heap is at the **root** of the heap
  - OR (next slide)

Heap ordering property

- Binary heap satisfies the heap ordering property, which can be one of:

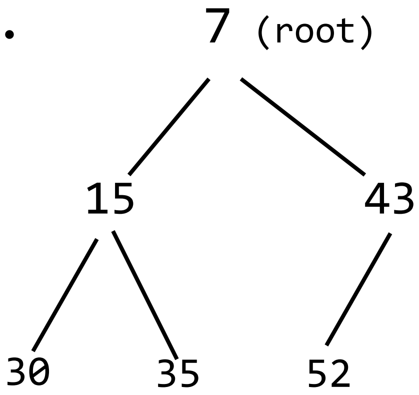
2. *Max* heap:

  - Value of each node is <= value of its parent node.
  - So, given a node and its two children, the parent node value >= children node value.
  - Extending this property to the whole tree (i.e., the whole binary heap):
    - it means the **biggest** value in the binary heap is at the **root** of the heap.

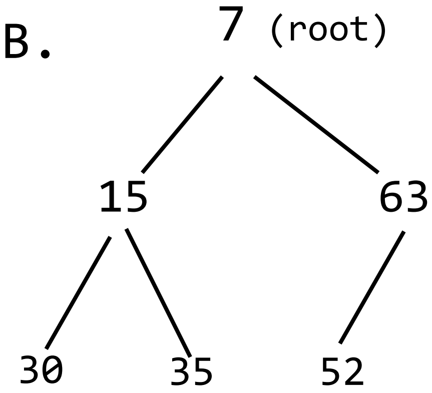
Min- heap

- Which of the following are min-heaps?

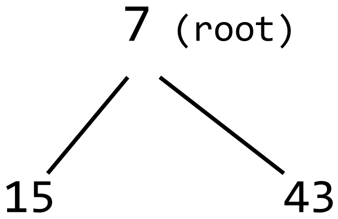
A.



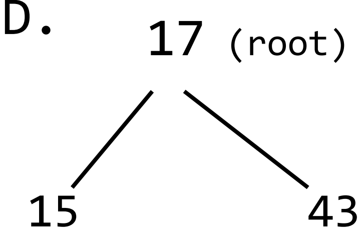
B.



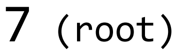
C.



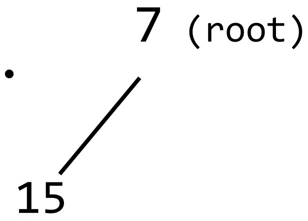
D.



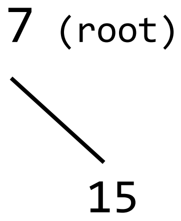
E.



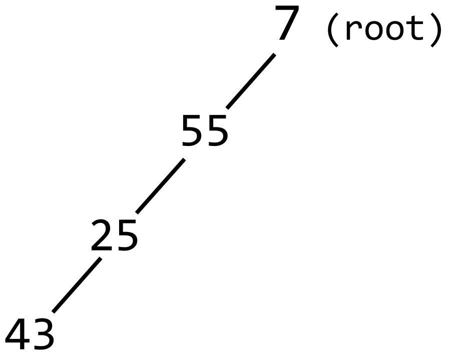
F.



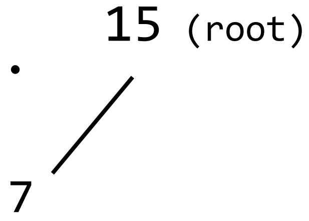
F2.



G.



I.



7 (root)

We just said this is a min heap.

Could it be a max heap?

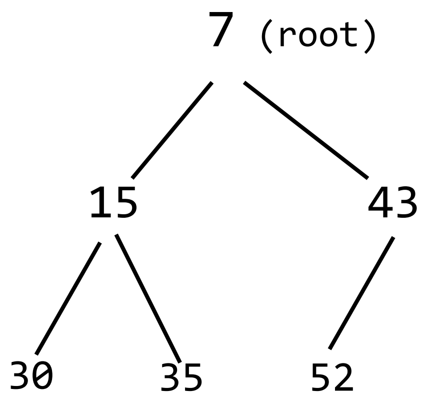
Binary heap representation

- So far, how have we represented binary trees?
  - What data structure have we used?

- Now, a binary heap is:
  - a complete tree
  - it means that there are no holes or gaps in the tree
    - except in the bottom level.

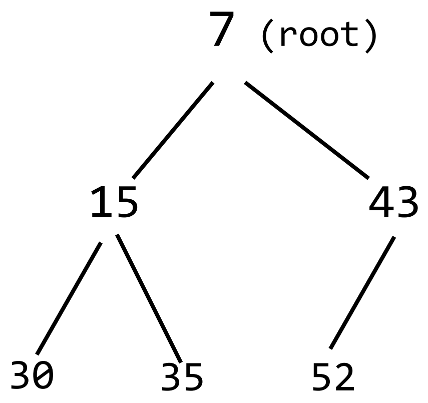
- As a result, we can easily represent a binary heap in an array.

0	1	2	3	4	5	6	7
	7	15	43	30	35	52	



0	1	2	3	4	5	6	7
	7	15	43	30	35	52	

Skip index 0 for ease of implementation.



Root is at index 1.

Left child of root at index 2

Right child of root at index 3

Left child of 15 is at index 4, and its right child is at index 5

In general, for i > 0:

Left child of a node at index i is at index : (2 * i) and right child is at index (2 * i) + 1

LeftChildIndex(i) = 2 * i

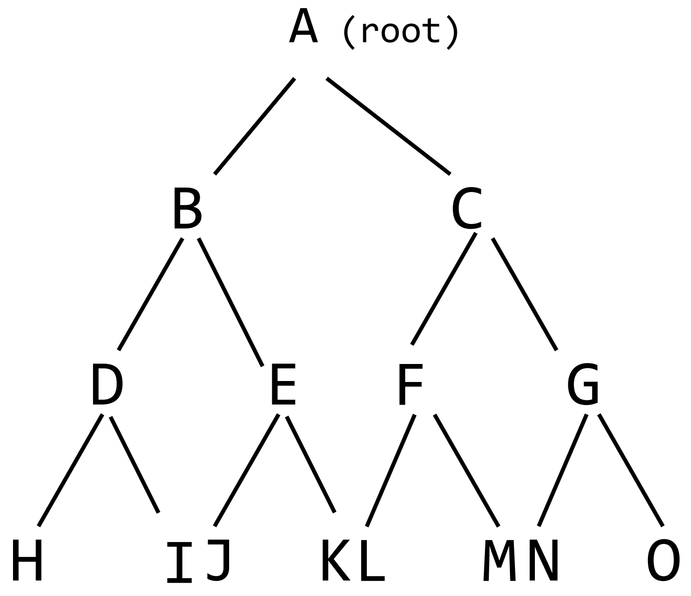
RightChildIndex(i) = (2 * i) + 1

ParentIndex(i) = i / 2 ← Integer division

Try these formulae on some indices in the heap above.

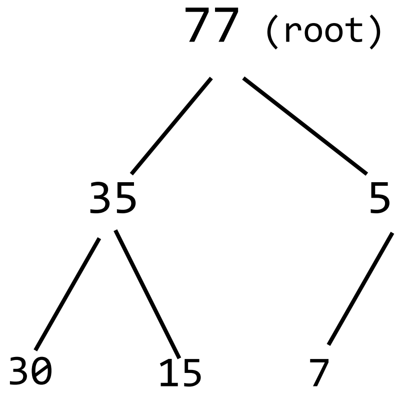
- A complete tree with N nodes will have height $\log_2 N$

- Now, in the binary heap that we see here, notice that the leaf nodes are nodes N/2 + 1 to N
    - Assuming root node is at index 1



Creating a min-heap

- Lets say we have the structure on the right
- It is a complete tree. So the structure is correct.
  - The only gap is that 52 has only a left child. That is ok.
- But the order is not correct for it to be a min heap.



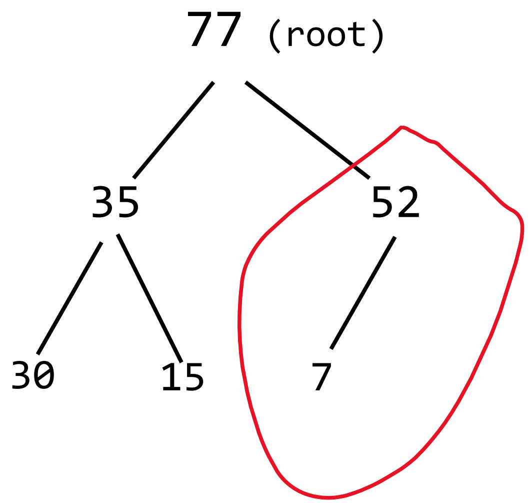
- In order to turn this into a min-heap, we **start** at the **leaf** level and then **go up**.
  - Now, when we look at the leafs (like 7, 15 and 30), they are already a valid heap in themselves.
  - So, there is **nothing to be done for the leaf nodes**… that’s great news.
    - In a full complete binary tree, 50% of the nodes are at leaf level.
    - Saves us a lot of work.

Creating a min-heap

So, if nothing to be done at the leaf level (bottom level), lets start with one level up.

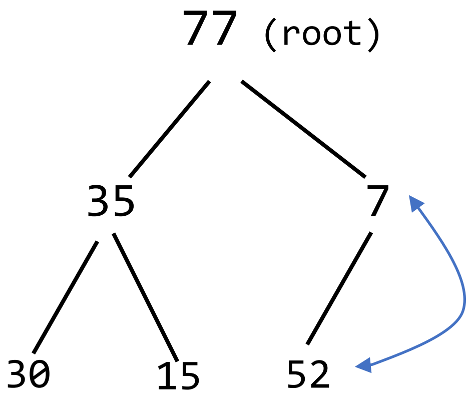
We will start heapify process with the sub tree rooted at 52, i.e., **parent** of the **last** value.

- **Note**: In heapify, we are ONLY concerned with the sub tree of the element we are starting at.



52 is the parent of 7, and so we need to swap the two.

We will get:



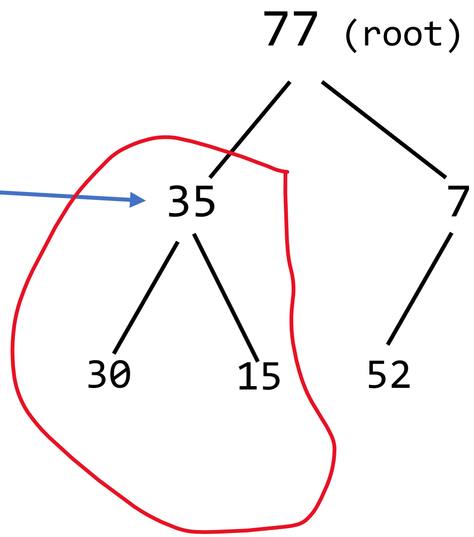
The sub tree rooted at 7 is now a binary heap. That is the ONLY subtree we worry about in this step.

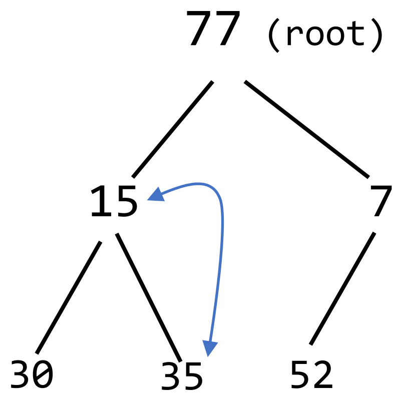
We move on to 35. (next slide)

The sub tree rooted at 35 does not satisfy the order property.

We take the **smaller** of its two children (30 and 15) and swap.

So, we swap 35 with 15, to get:





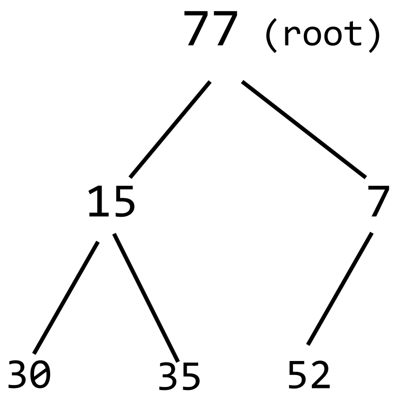
Now, the subtree at 15 is a binary heap.

That is the <u>ONLY subtree</u> we worry about in this step.

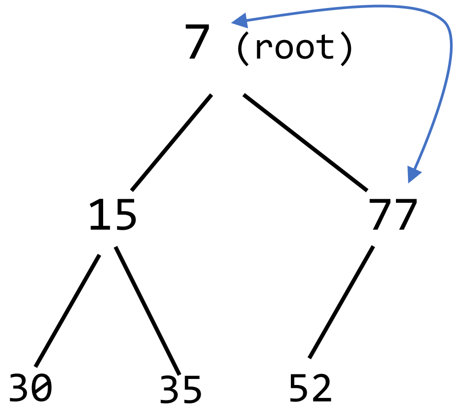
Next we look at 77 (next slide).

We now look at 77, which in this case happens to be the root.

- Note that in this step, we will work on the subtree rooted at 77.

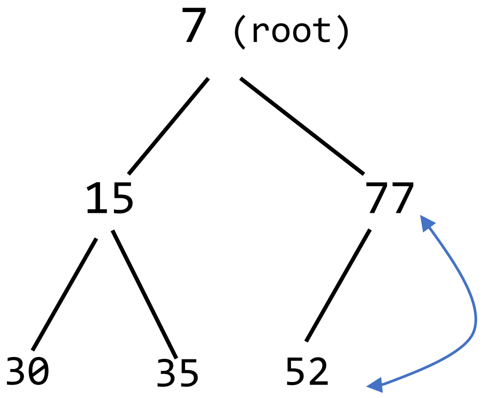


Comparing 77 with the smaller of its two children, we need to swap it with 7. However, notice that now the subtree rooted at new position of 77 is no longer a min heap, as it is > its child (52).



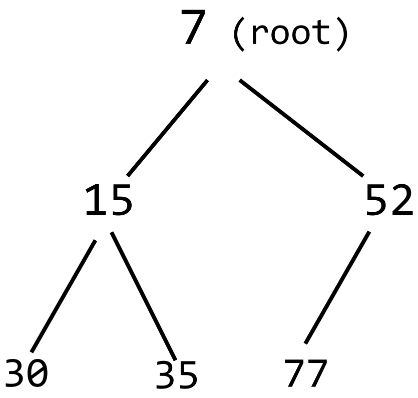
So the heapify process continues, and we now look at sub tree rooted at 77.

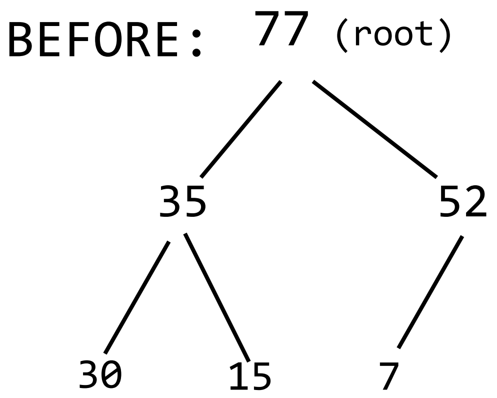
We need to swap 77 with 52 (77’s only child).



We get the following, which is a min heap.

The heapify process is now complete, because in this last step we handled the root, and so we heapified the whole tree.





Big O of heapify process

- When we heapify, the **maximum distance an element may move is the height of the tree**
  - e.g. the root element may move all the way down to leaf.
  - The height is log N.
- There are a total of N elements in the heap.
- So what is the complexity of the heapify process?

Big O of heapify process

- So one would think that the complexity of the heapify process should be O(N log N).
- Turns out it is actually O (N)… which is great.

- Will not go into the proof of this here, but remember that in the heapify process, we didn't have to do anything with nodes in the bottom (leaf) level… that is 50% of the nodes.
- And then the nodes at level above leaf level (the 2nd last level) would move at most 1 level.
  - This is N/4 nodes.

- The maximum ***potential*** movement of log N is only true for the **one root node**.
  - For all other nodes, it is less than log N

Big O of heapify process

- So, total *potential* moves would be something like:
  - (N/2 * 0) + (N/4 * 1) + (N/8 * 2) + (N/16 * 3) + … + (1 * log N)
    - N/4 nodes move down a max of 1 level
    - N/8 nodes move down a max of 2 level, and so on.
- This series converges to a sum of N.

Big O of heapify process (with an example)

- Lets take a look at an example.

- Lets say we have a heap of, say, 127 elements (For simplicity, I am assuming a full tree).
    - How many levels is that (height of tree)?
    - How many nodes (or heaps) are at the leaf level?
        - These are heaps of size 1
    - How many nodes (or heaps) are at the level immediately above the leaf?
        - These are heaps of size 3 (at the most)

- On next slide, let’s calculate
        - the number of *actual* comparisons and
        - the number of *potential* moves

Big O of heapify process (with an example)

- Again, we have a heap of 127 elements (For simplicity, I am assuming a full tree).

- Number of nodes (or heaps) at leaf level: 64
    - Nothing to be done here.

- Number of nodes (or heaps) at leaf - 1 level: 32
    - Number of comparisons: 2 for each heap, so: 64
    - Number of potential moves: 1 for each heap, so: 32

- Number of nodes (or heaps) at leaf - 2 level: 16
    - Number of comparisons: 2 for each heap, so: 32
    - Number of potential moves: 1 for each heap, so: 16

- Total comparisons = (32 * 2) + (16 * 2) + (8 * 2) + (4 * 2) + (2 * 2) + (1 * 2) = 126 (that's N)
- Total *potential* moves = (32 * 1) + (16 * 2) + (8 * 3) + (4 * 4) + (2 * 5) + (1 * 6) = 120 (that's N)

Insert

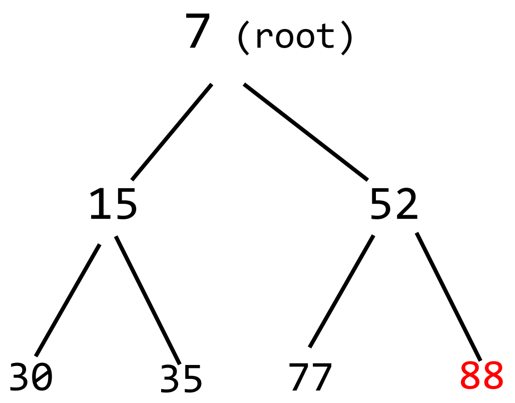
- Lets talk about inserting an element into a binary heap (min heap).

- We take the new element and add it to **end** of the binary heap
    - That’s the only place we have space.
    - Remember, binary heap is completely full, except in the leaf level, where the only gaps are on the right end.

- Now, with the new element inserted at the end, we know:
    - the structural property is still maintained
    - but the order property *may* not.
    - So we need to heapify … and this needs to happen in **the up direction**
        - we will see what we mean by the up direction.

- Lets start with the min heap on the right

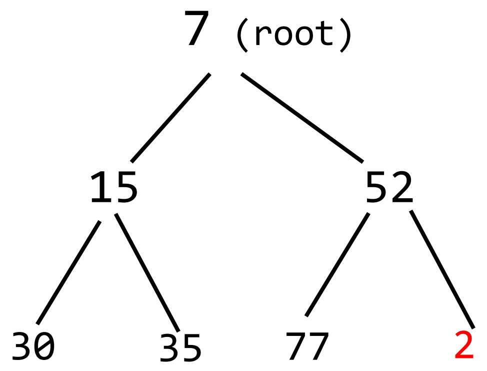
- And we will insert a new element, 88, at the end.



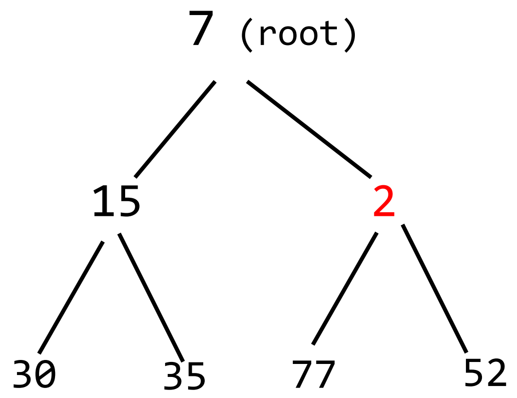
- After insertion, we now *HeapifyUp*
  - Compare 88 with its parent, i.e., 52.
  - We notice that the order property is still maintained, because the new node added is > its parent (it's a min heap).
  - So there is nothing more to be done here… **we got lucky**.

  - Next slide, lets look at another example

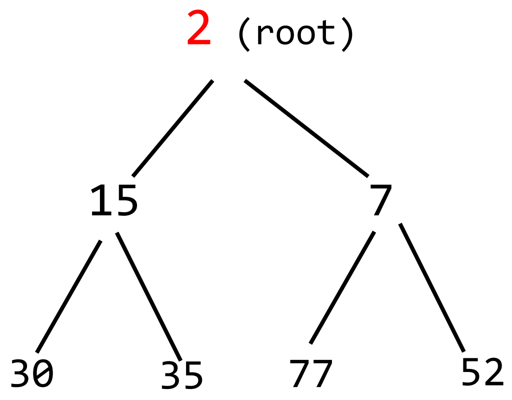
- Lets take another example of insertion.
- We will insert 2 instead



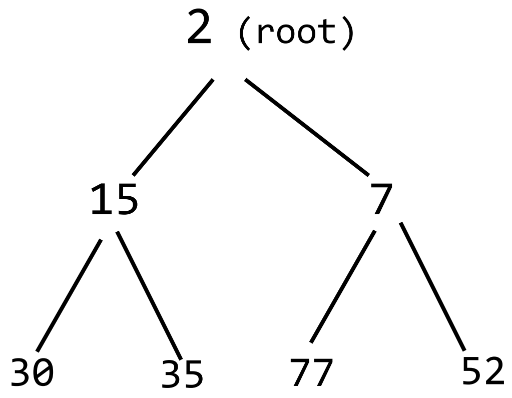
- After insertion, we now *HeapifyUp*
  - Compare 2 with its parent, i.e., 52.
  - Swap the nodes, since 2 < 52



  - Now compare 2 with its parent, 7
  - We need to swap again, to get the final heap



- So our final min heap after insertion looks like this.



- This was a case where the newly inserted element travelled up to become the new root.
    - So it had to move up the height of the tree.

- So, the worst case run time of an insert is O (log N)

GetMin

- GetMin operation is the fastest in a min heap because it just has to return the root.
- Its complexity is O (1)
- In a max heap, GetMax is the corresponding operation, and that just returns the root, and is, as expected, O(1)

ExtractMin (or DeleteMin)

- In a min heap, the smallest element is at the root, as we saw.

- In order to remove the minimum element (root), we replace it with the **last** element in the heap.
    - This means we decrease the heap size by 1.

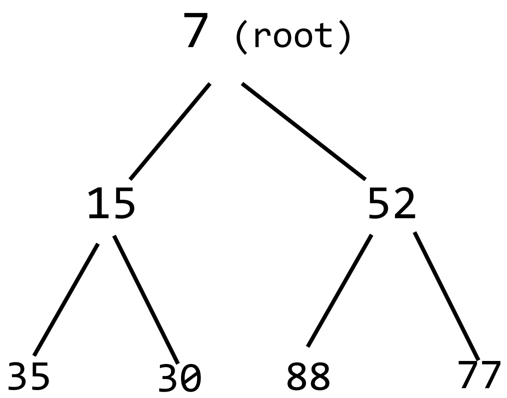
- Now, the structure property is still ok (because we removed the last element).

- For fixing the order property, we *HeapifyDown*.
    - This process is similar to *HeapifyUp* that we just saw.

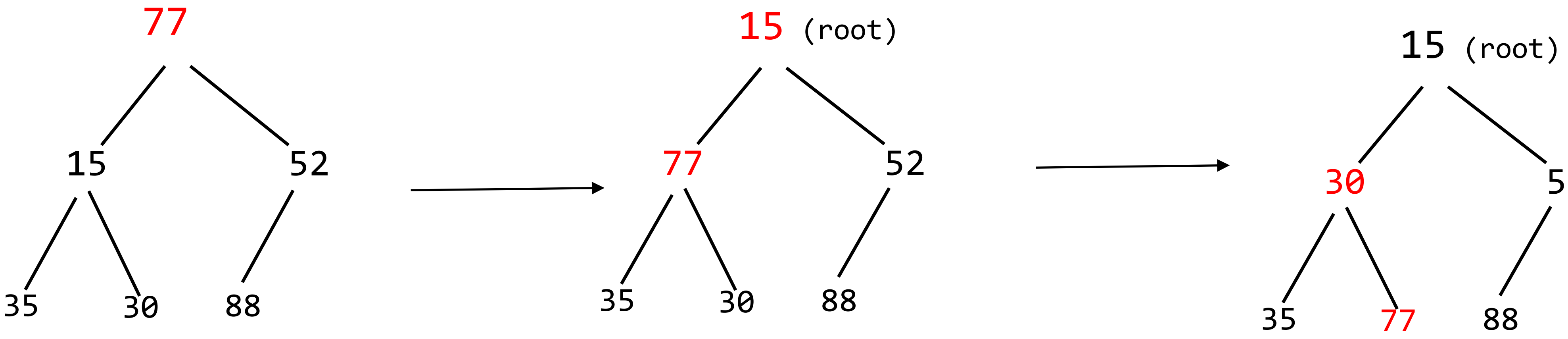
- Worst case complexity of ExtractMin is O (log N)

Doing ExtractMin repeatedly

- Lets keep calling ExtractMin on the min heap here.

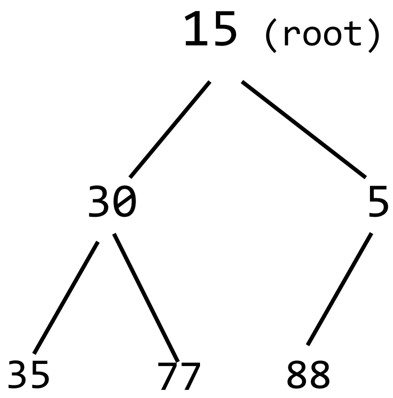


- Here's what we would get:
- 7 would the first element extracted.
- We would put the last element(77) in its place
- **Heapify down** would look like:

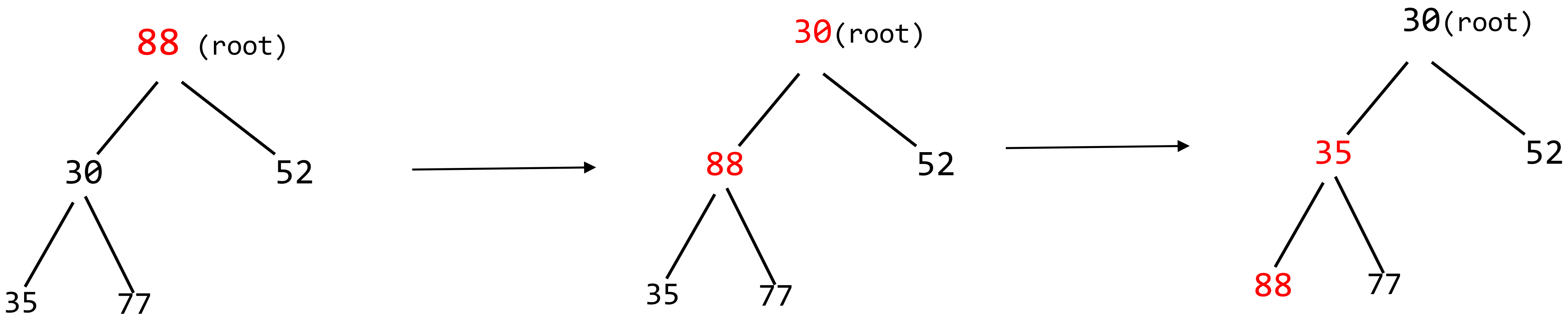


Doing ExtractMin repeatedly

- Call ExtractMin again, we extract 15.
  - So far, we have: 7, 15

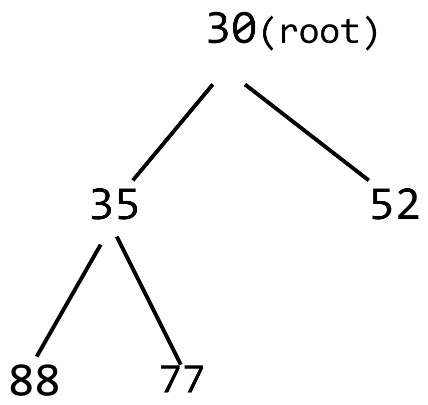


- **Heapify down** would look like:

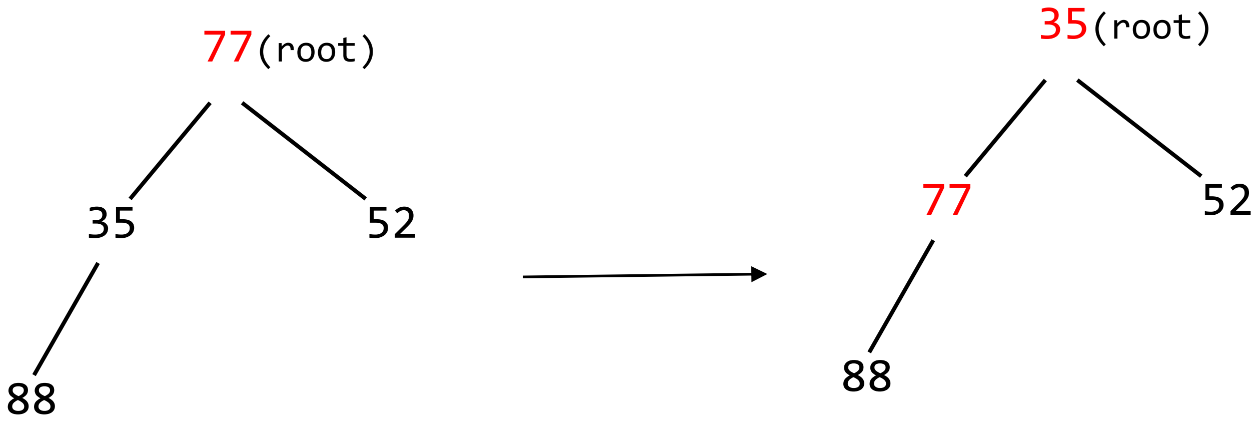


Doing ExtractMin repeatedly

- Call ExtractMin again, we extract 30.
  - So far, we have: 7, 15, 30



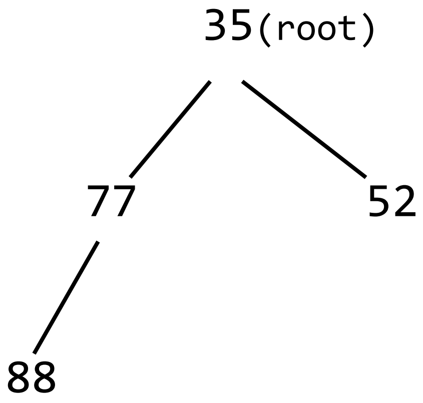
- **Heapify down** would look like:

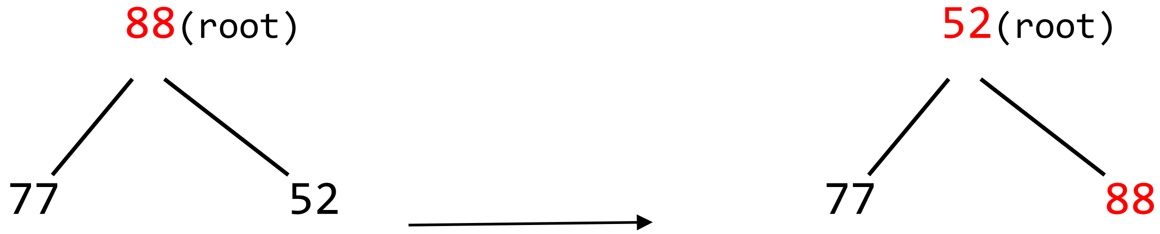


Doing ExtractMin repeatedly

- Call ExtractMin again, we extract 35.
  - So far, we have: 7, 15, 30, 35

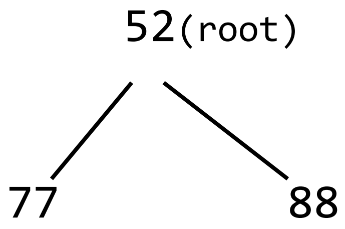
- **Heapify down** would look like:



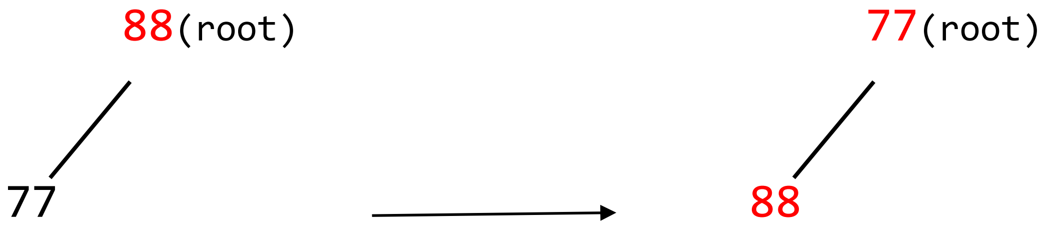


Doing ExtractMin repeatedly

- Call ExtractMin again, we extract 52.
  - So far, we have: 7, 15, 30, 35, 52

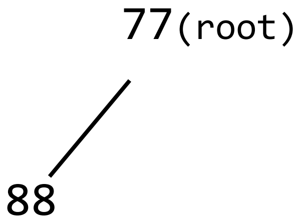


- **Heapify down** would look like:



Doing ExtractMin repeatedly

- Call ExtractMin again, we extract 77.
  - So far, we have: 7, 15, 30, 35, 52, 77



- **Heapify down** would look like:

88(root)

Doing ExtractMin repeatedly

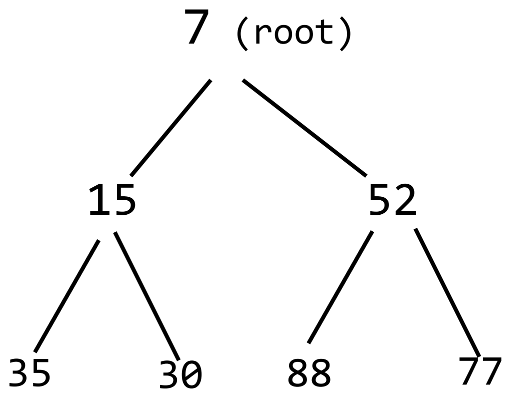
- Call ExtractMin again, we extract 88.
    - So far, we have: 7, 15, 30, 35, 52, 77, 88

- Now we extracted all elements from the min heap.

- What do you notice about the sequence of numbers we got by repeated calls to ExtractMin?

Sorting

- The sequence that we obtained is sorted
  - 7, 15, 30, 35, 52, 77, 88



- So here’s what we did:

1. Built a min heap
2. Then called ExtractMin repeatedly to get the numbers out in a sorted order.
3. Sort order is non-decreasing.
   - Technically correct way of saying increasing.
   - We say non-decreasing because two or more elements could be the same value.

Sort Big O

The steps we did for sorting the numbers are:

- Built a min heap.
  - Big O is O (N)

- Then called ExtractMin repeatedly to get the numbers out in a sorted order.
  - Big O for ExtractMin is O (log N)
  - We called ExtractMin N times (once for each element in the min heap)
  - So, Big O for this whole loop is O (N log N)

- Overall complexity is:
  - Complexity of BuildHeap + complexity of ExtractMin loop
  - O (N + N log N)
  - Which is O (N log N)

- This is **heap sort.**

Priority Queue

- **Priority queue i**s a data structure that lets you pull out the highest priority item from a queue.
    - For example, can be used by the operating system to schedule a task on a core / CPU.

- This should ideally be a fast operation, and a binary heap is a great data structure to use here.

- As we saw, GetMin is an O(1) operation, and ExtractMin is O log N.
- And insertion in a priority queue would also be O log N.

- Note that a min or a max binary heap can be used for a priority queue.
- This is because a lower value could imply a higher priority
    - Higher priority doesn’t have to be a higher value.
    - Its whatever the convention would be in a given software system.

LAB

1. Insert the following values in the given order to a binary min-heap

   5, 10, 4, 77, 43, 15, 22, 61

Draw the final resulting min heap.

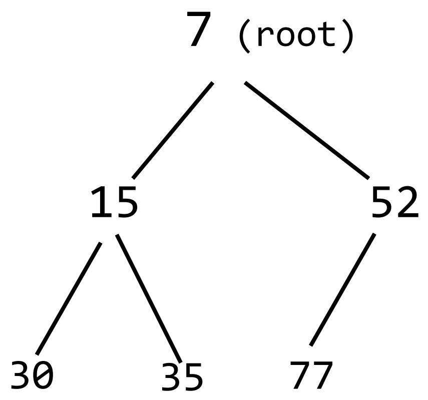
2. Now extract the min element from this heap **two times**, and draw the final resulting min heap.

3. Now add the following values to the resultant heap from step above. Draw the final resulting min heap
   - 100, -1, -50

4. You are given a binary heap (which as we know, is a complete binary tree) that contains a total of 127 elements.
   - How many leaf elements are there?
   - How many elements in the level just above the leaf level?

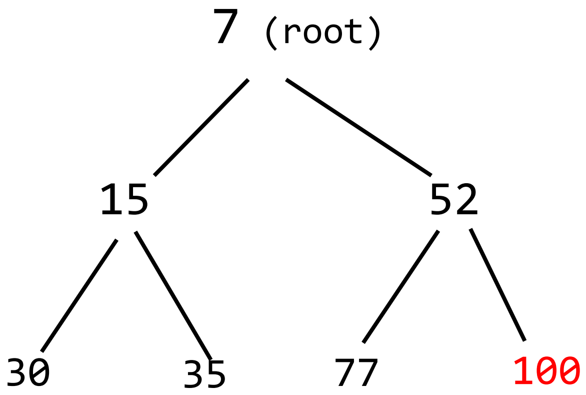
5. Code up below functions for a binary heap:
   - GetParentIndex(int index)
   - GetLeftChildIndex(int index)
   - GetRightChildIndex(int index)

Code example



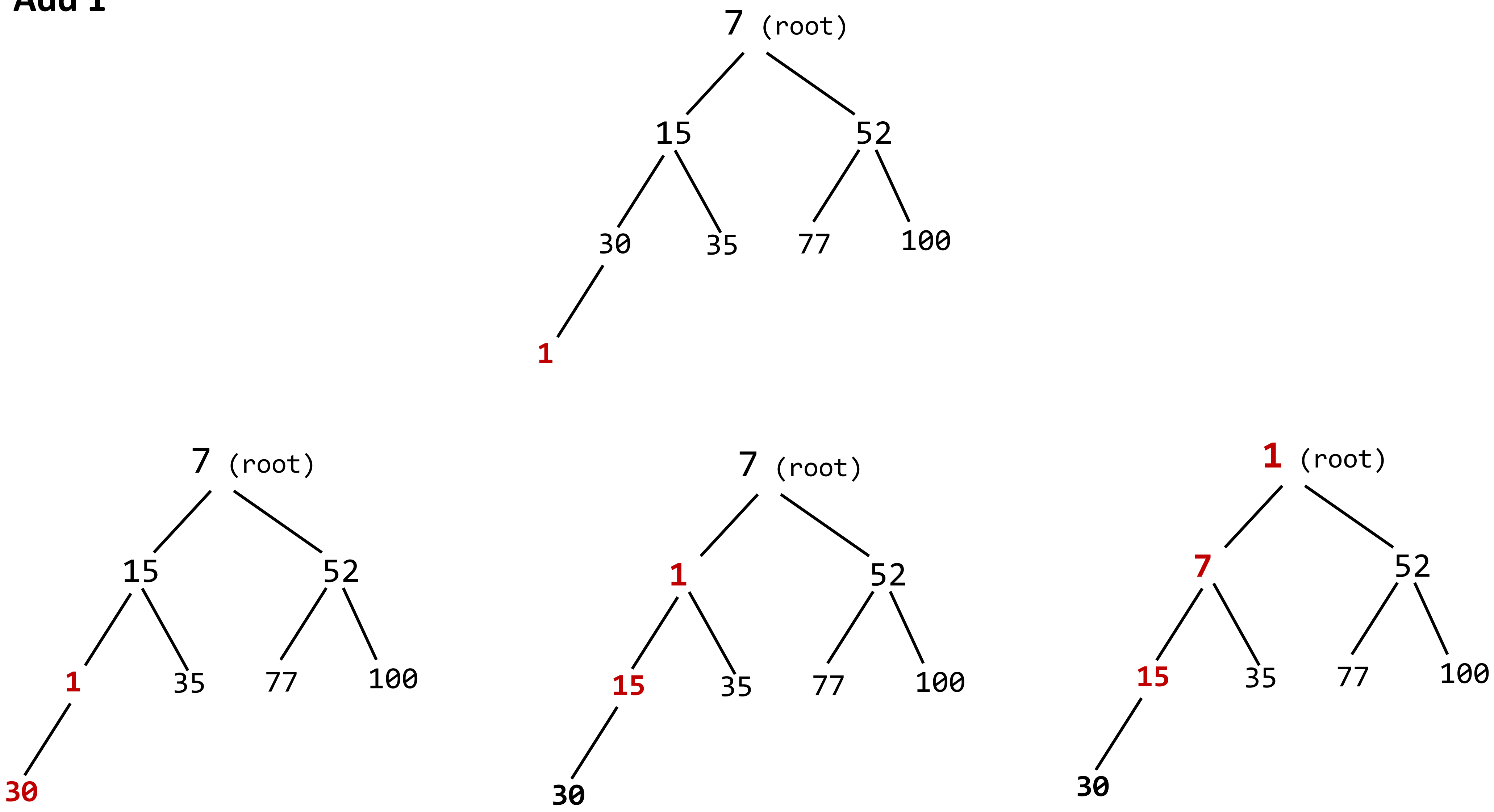
- This is the code example in the .cs file (we will step thru this in class)
- The min heap we use is from class example earlier where we built the heap.

- In the code, after building the heap, we do the following
    - Add 100



Code example

- **Add 1**



Code example

- Then we do a bunch of DeleteMins … See the existing slides in the class example for diagrams for that.

Home exercise (optional, NOT to be submitted)

You should first design/write up your solution. Then code it up, if u want to, and if u have time.

Design a priority queue that enables u to query it for **min as well as max**.

This should support the following operations on it:

- O(log N) time complexity:
    1. Delete maximum.
    2. Delete minimum.
    3. Insert an element.

- Constant time complexity:
    1. Get maximum
    2. Get minimum

- Hint for the min max priority queue exercise on the previous slide:
  - U can use two binary heaps.