ECE430.217 Data Structures

Binary Heaps

Textbook: Weiss Chapter 6.3

Byoungyoung Lee
https://compsec.snu.ac.kr
byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

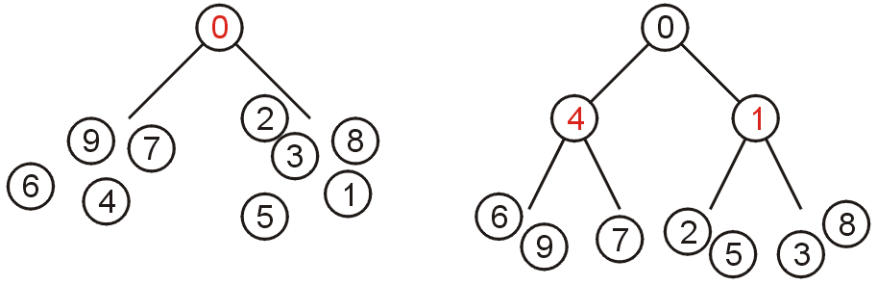
In this topic, we will:

- Define a binary min-heap
- Look at some examples
- Operations on heaps:
  - Top
  - Pop
  - Push
- An array representation of heaps
- Define a binary max-heap
- Using binary heaps as priority queues

Definition: Min-Heap

A non-empty binary tree is a min-heap if

- The key of the root is less than or equal to all the keys in both sub-trees
- Both of the sub-trees (if any) are also binary min-heaps

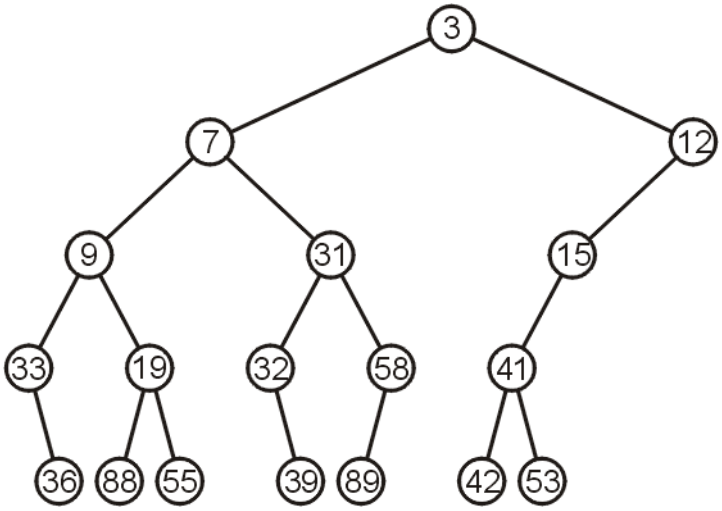


From this definition:

- A single node is a min-heap
- All keys in either sub-tree are greater than the root key

Example

This is a binary min-heap:



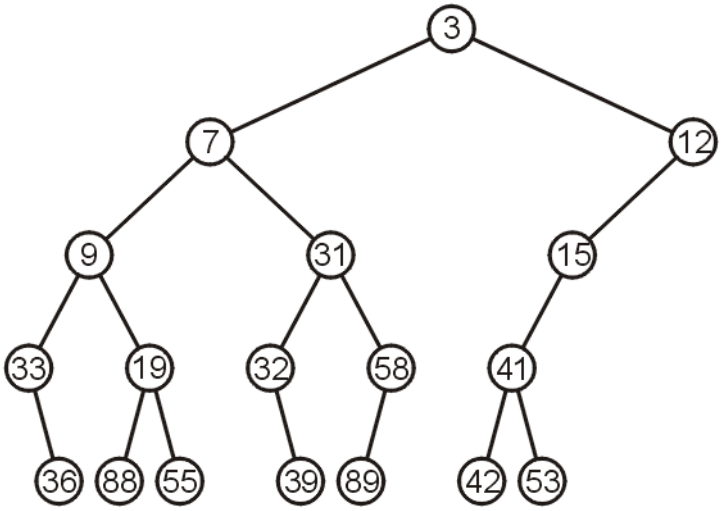
Operations

We will consider three operations:

- Top
- Pop
- Push

Top

We can find the top object in $\Theta(1)$ time: 3



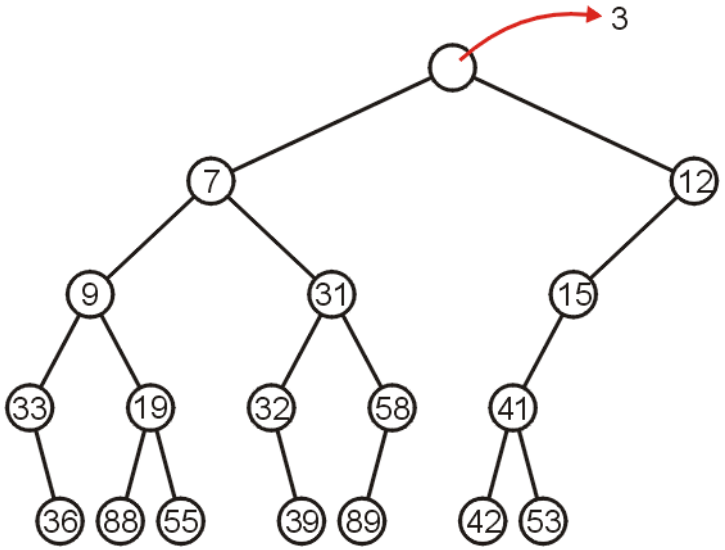
Pop

To remove the minimum object:

- 1) Remove the root node
- 2) Promote the node of the sub-tree which has the least value
- 3) Recurs down the sub-tree from which we promoted the least value

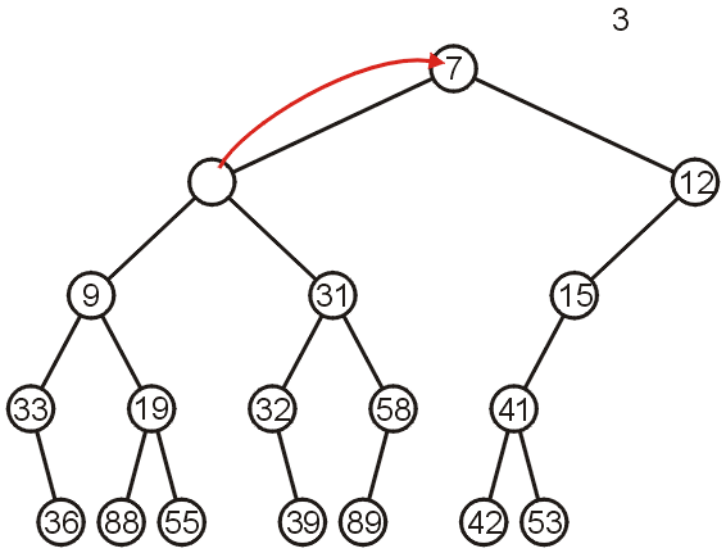
Pop: 3

Using our example, we remove 3:



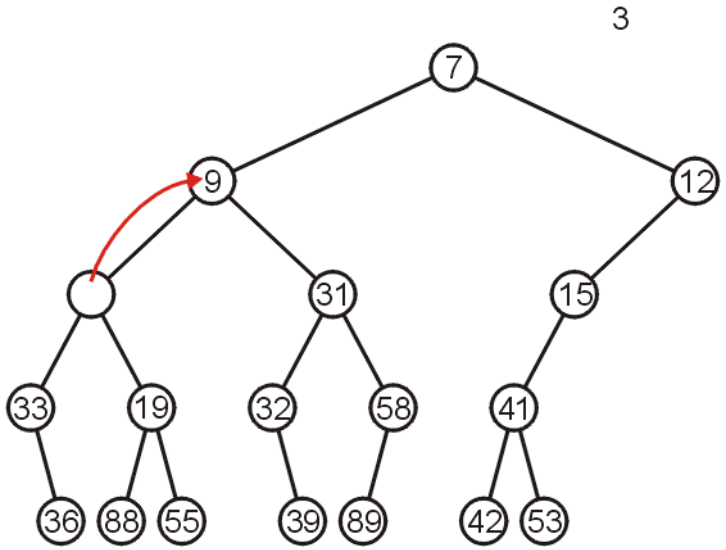
Pop: 3

We promote 7 (the minimum of 7 and 12) to the root:



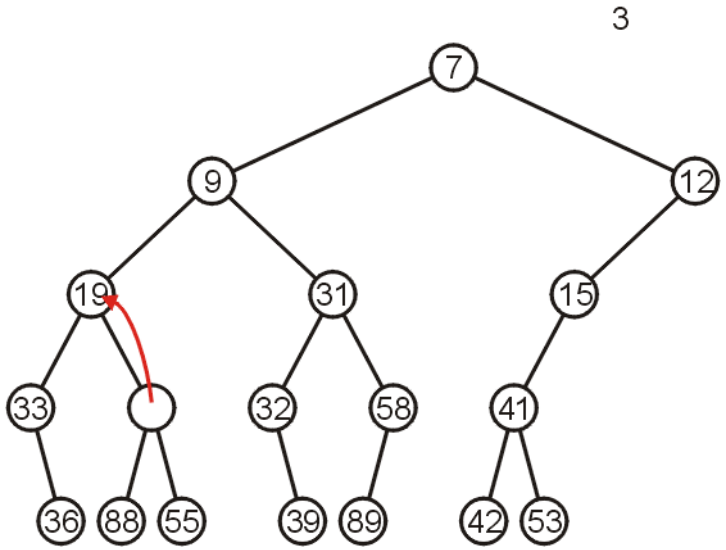
Pop: 3

In the left sub-tree, we promote 9:



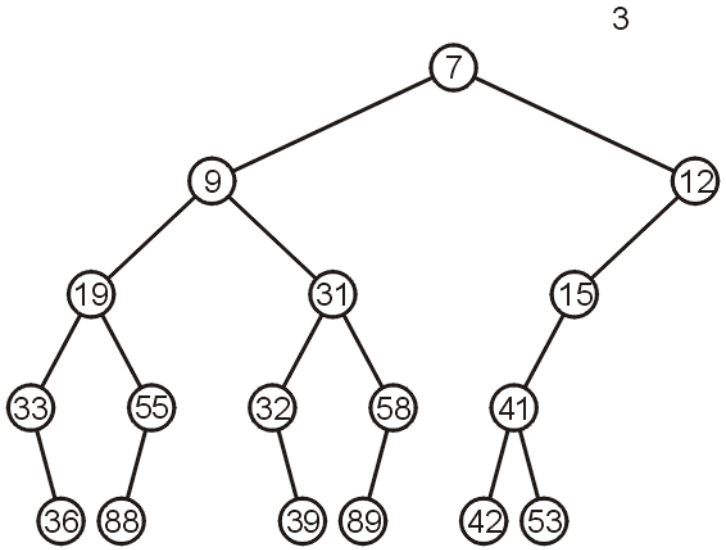
Pop: 3

Recursively, we promote 19:



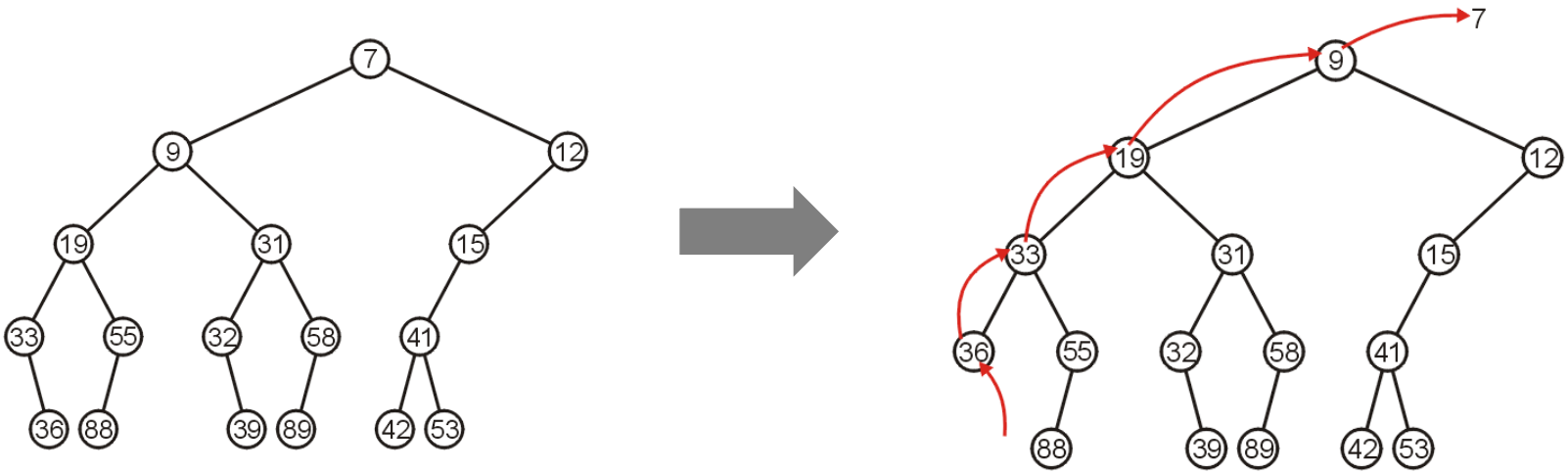
Pop: 3

Finally, 55 is a leaf node, so we promote it and delete the leaf



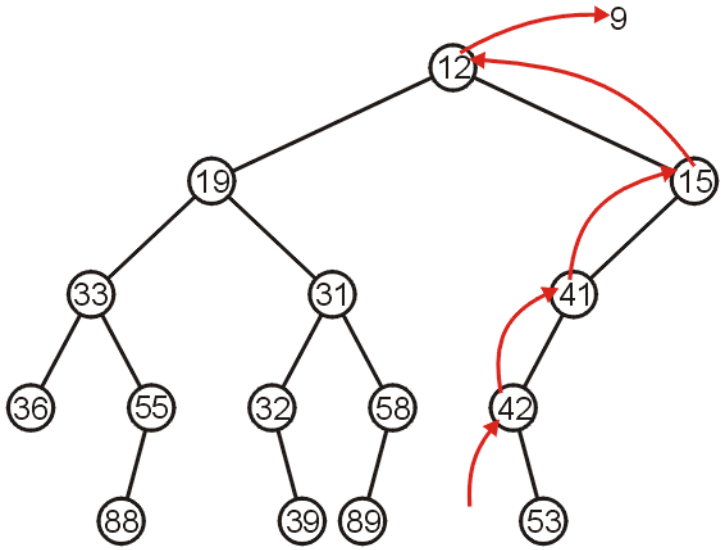
Pop: 7

Repeating this operation again, we can remove 7:



Pop: 9

If we remove 9, we must now promote from the right sub-tree:



Push

Inserting into a heap may be done either:

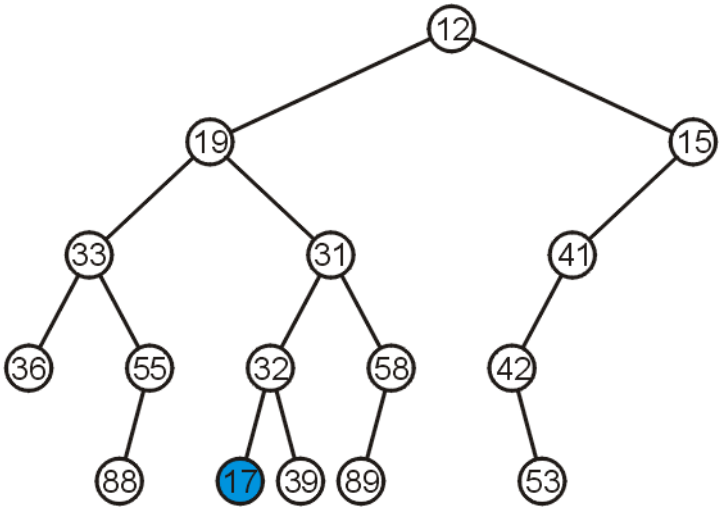
- **Bottom-up:** At a leaf (move it up if it is smaller than the parent)
- **Top-down:** At the root (insert the larger object into one of the subtrees)

We will use **the bottom-up approach** with binary heaps

Push: 17

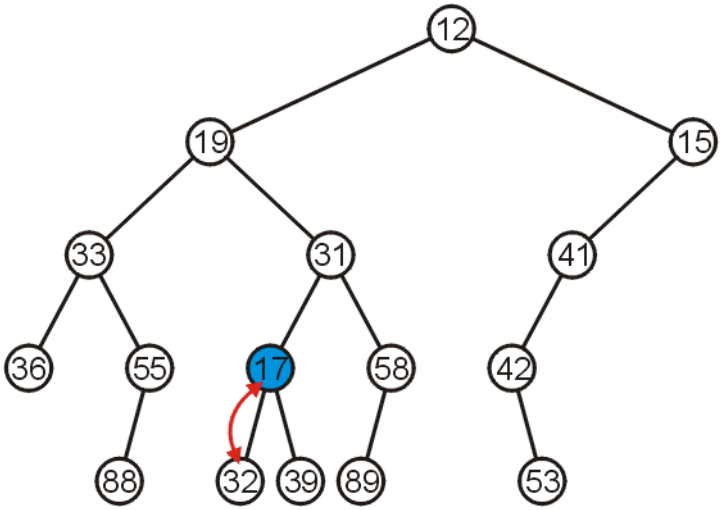
Inserting 17 into the last heap

– Select an arbitrary node to insert a new leaf node:



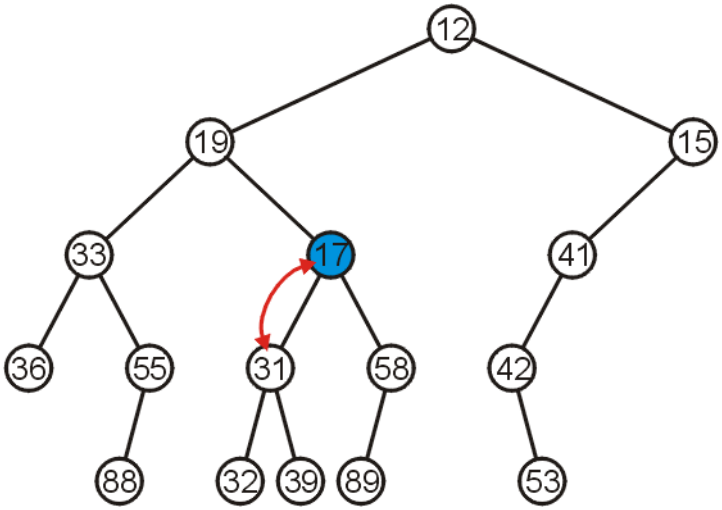
Push: 17

The node 17 is less than the node 32, so we swap them



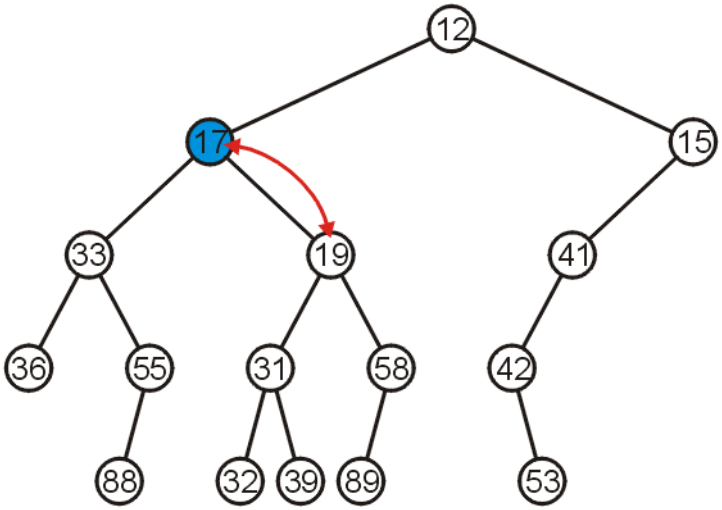
Push: 17

The node 17 is less than the node 31; swap them



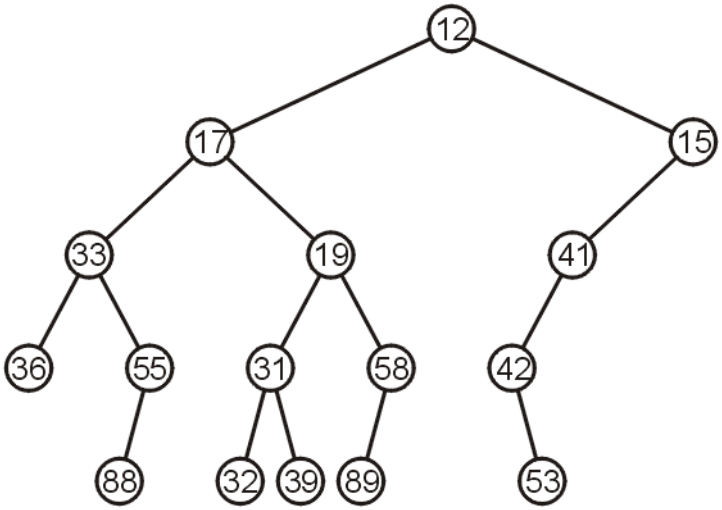
Push: 17

The node 17 is less than the node 19; swap them



Push: 17

The node 17 is greater than 12 so we are finished



Push Observation: One-way Percolation up/down

Observation: Both the left and right subtrees of 19 were greater than 19, thus we are guaranteed that we don't have to send the new node down

This process is called ***percolation up***, that is, the lighter (smaller) objects move up from the bottom of the min-heap

Keeping Balance of Binary Heap

With binary search trees, we introduced the concept of *balance*

- AVL Trees
- B-Trees
- Red-black Trees

How do we maintain the balance of binary heap?

Easy Solution: Complete Tree

To keep the balance, we maintain the shape of complete tree structure

We have already seen

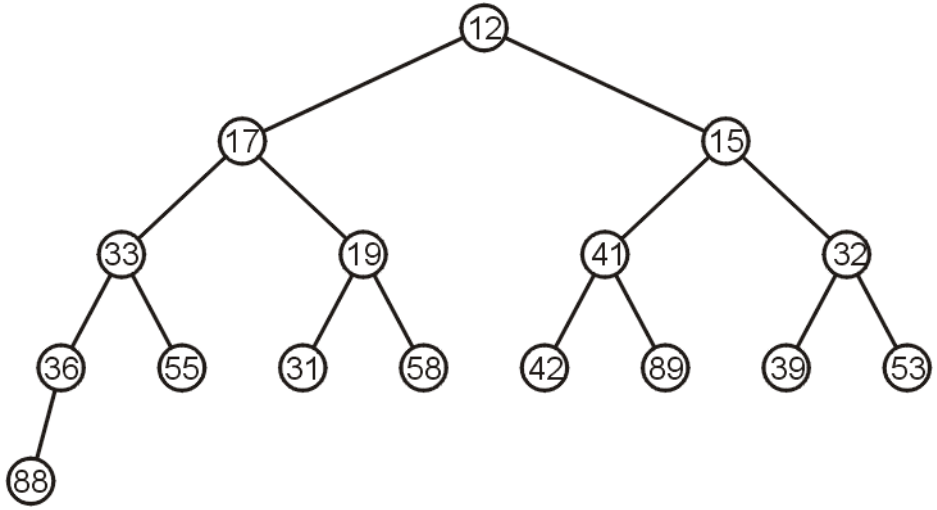
- It is easy to store a complete tree as an array

If we can store a heap of size n as an array of size $\Theta(n)$, this would be great!

We now need to think about how to support push and pop.

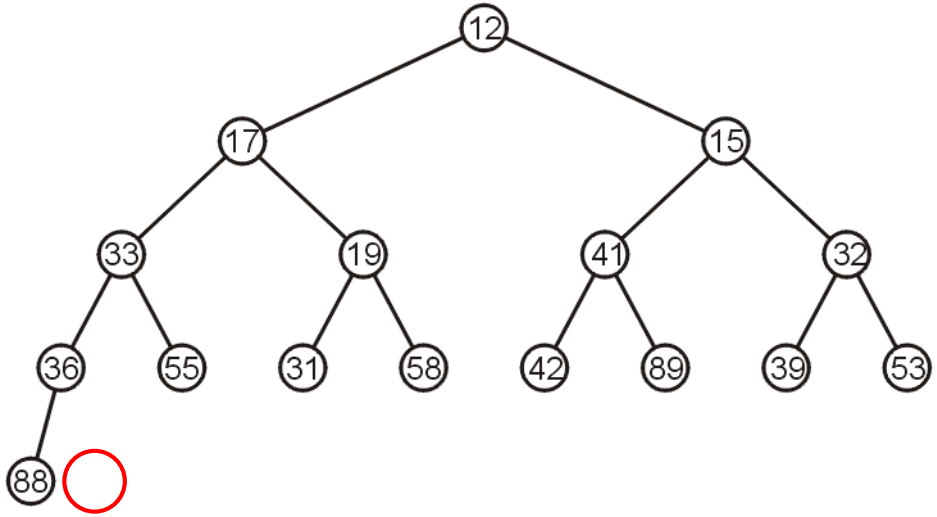
Complete Trees

For example, the previous min heap may be represented as the following complete tree:



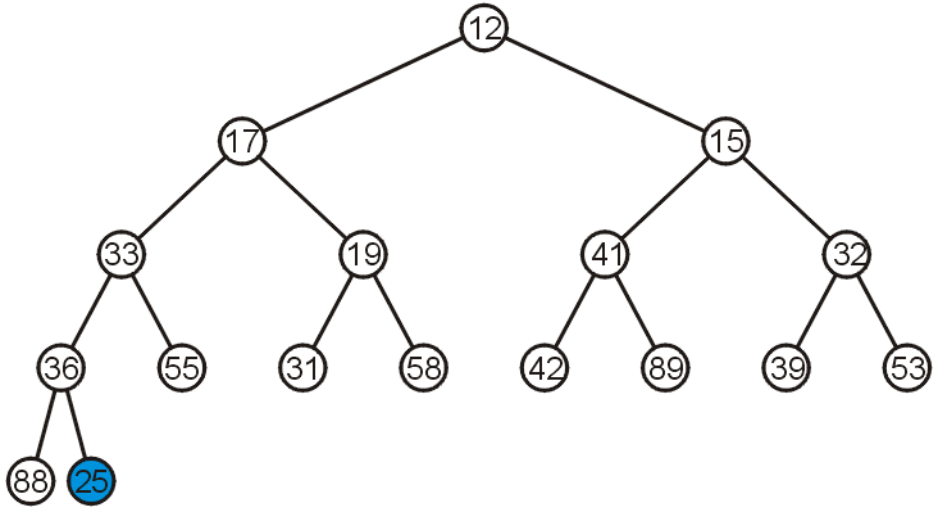
Complete Trees: Push

If we insert into a complete tree, we only need to place the new node as a leaf node in the appropriate location and percolate up



Complete Trees: Push 25

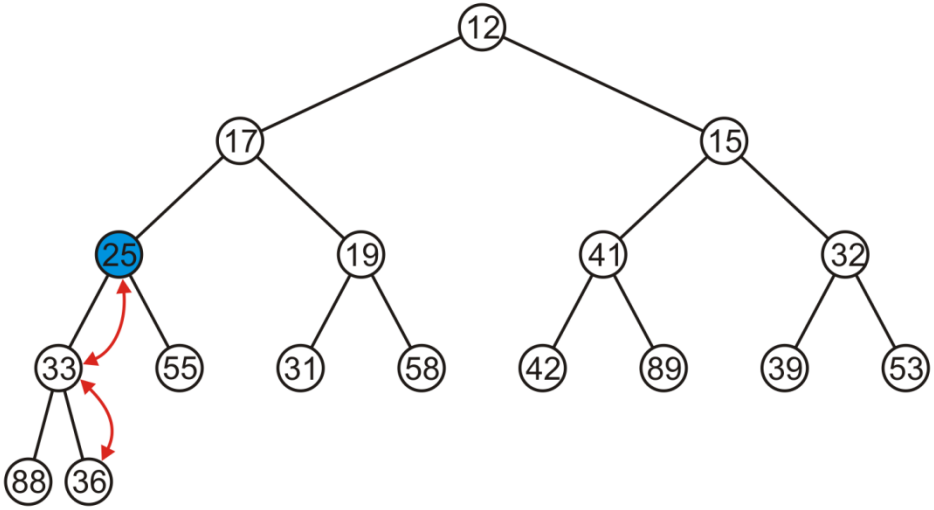
For example, push 25:



Complete Trees: Push 25

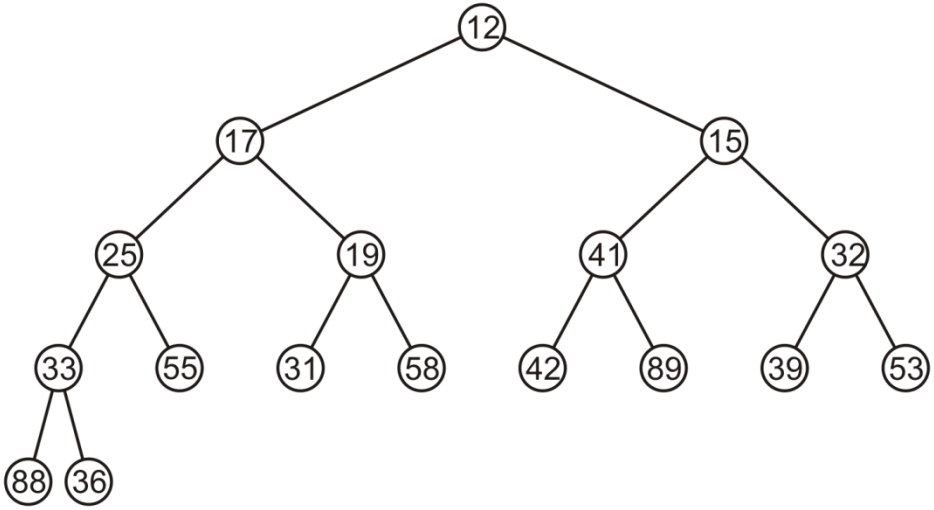
We have to percolate 25 up into its appropriate location

- The resulting heap is still a complete tree



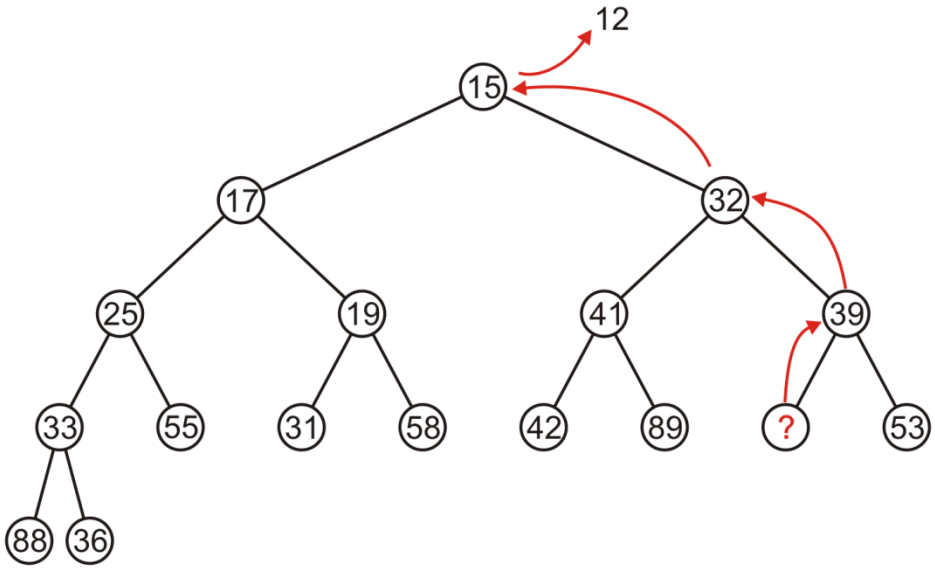
Complete Trees: Pop 12

Suppose we want to pop the top entry: 12



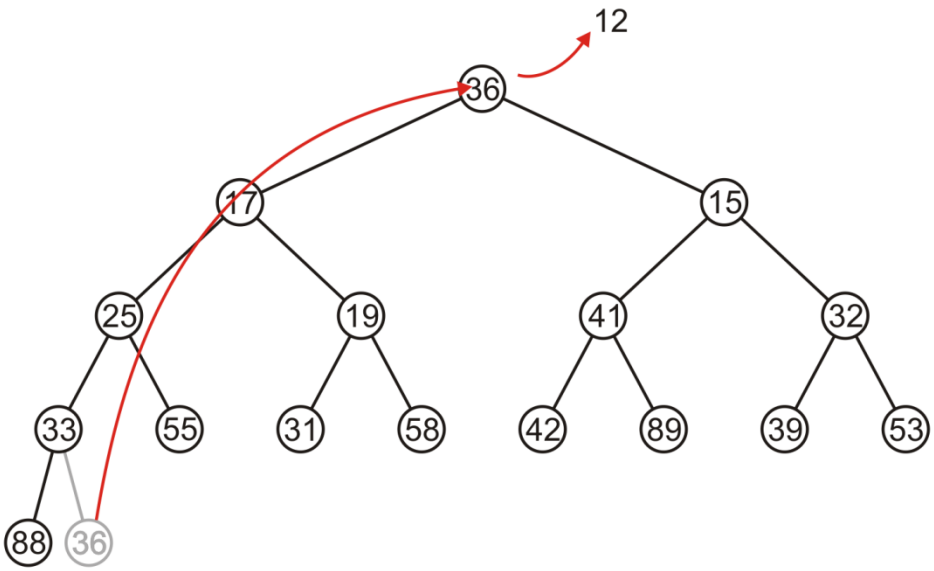
Complete Trees: Pop 12

Percolating up creates a hole leading to a non-complete tree



Complete Trees: Pop 12

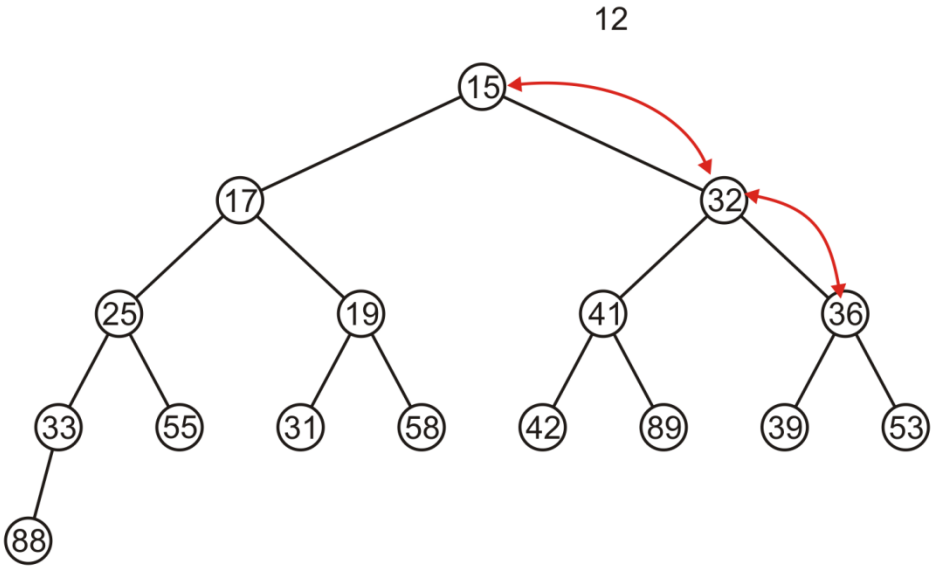
Alternatively, copy the last entry in the heap to the root



Complete Trees: Pop 12

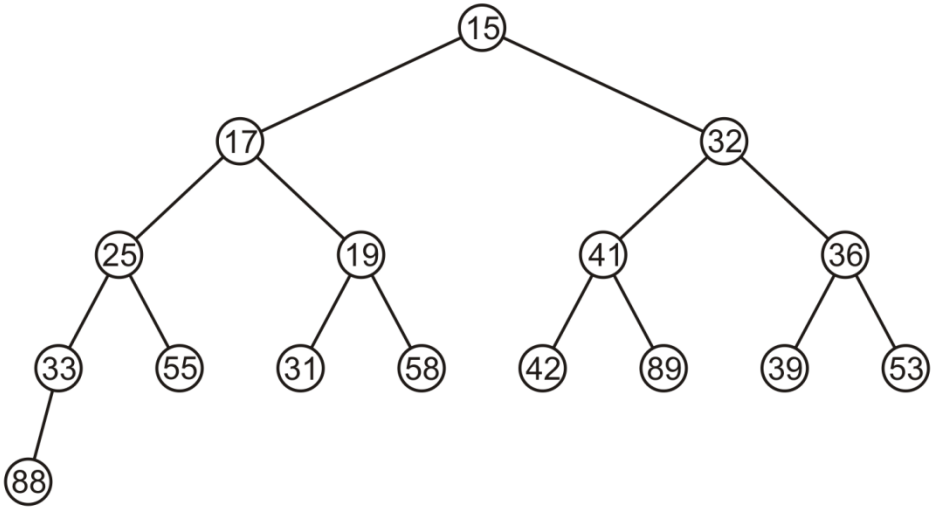
Now, percolate 36 down swapping it with the smallest of its children

- We halt when both children are larger



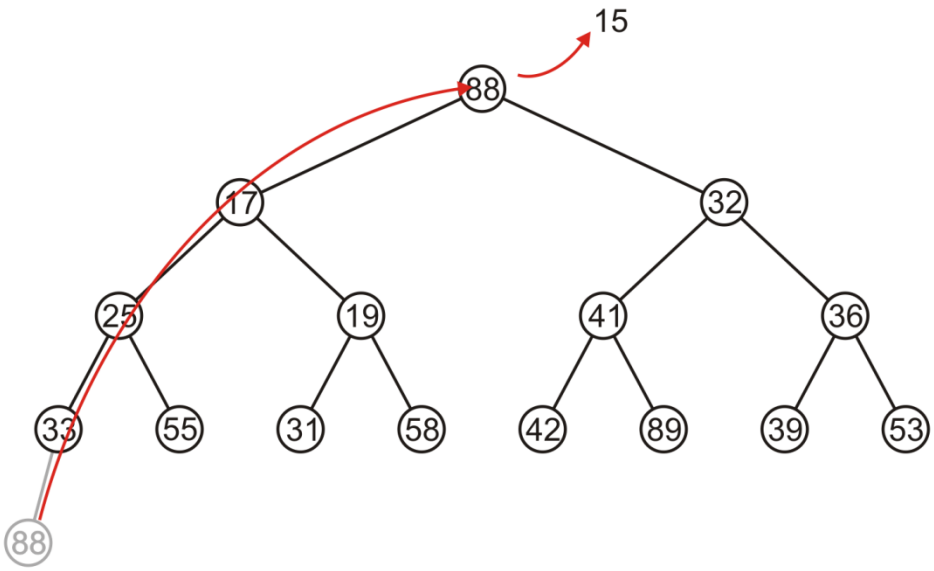
Complete Trees: Pop 12

The resulting tree is now still a complete tree:



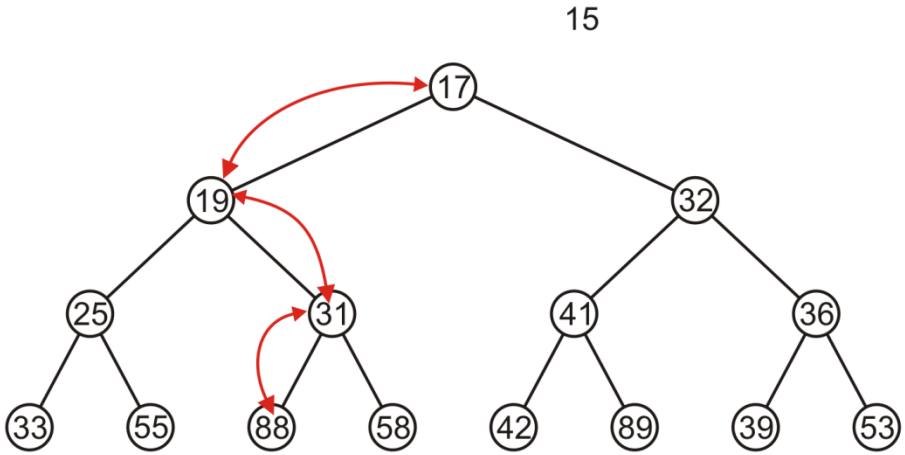
Complete Trees: Pop 15

Again, popping 15, copy up the last entry: 88



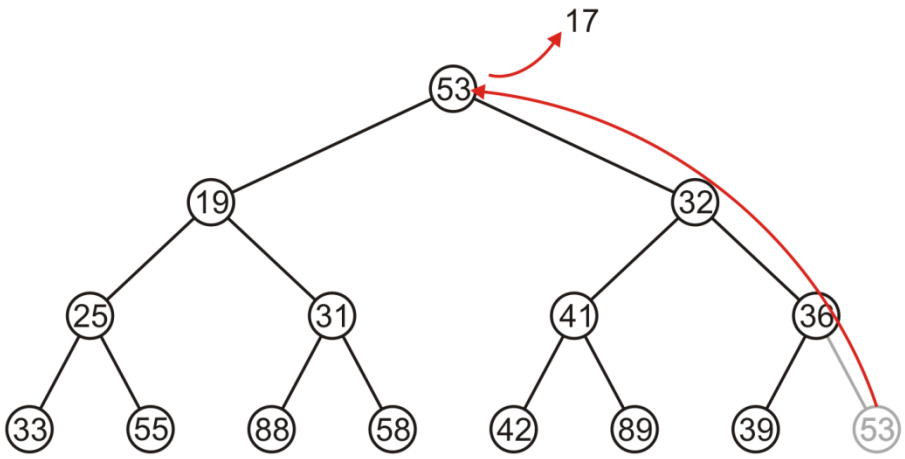
Complete Trees: Pop 15

This time, it gets percolated down to the point where it has no children



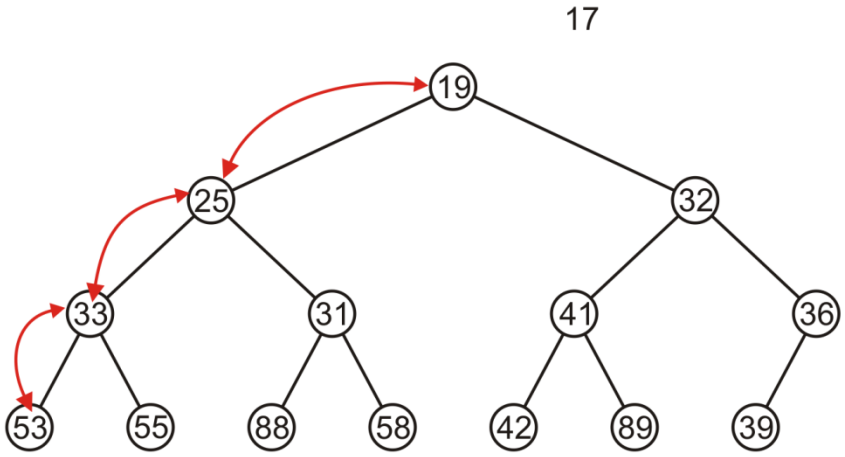
Complete Trees: Pop 17

In popping 17, 53 is moved to the top



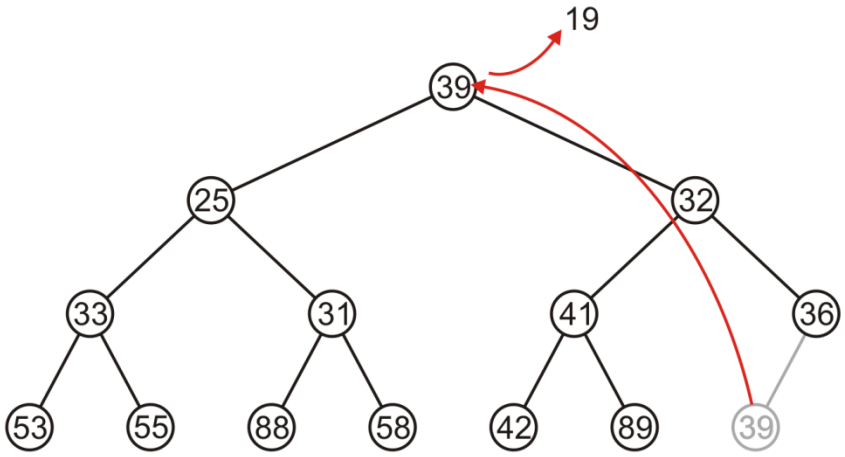
Complete Trees: Pop 17

And percolated down, again to the deepest level



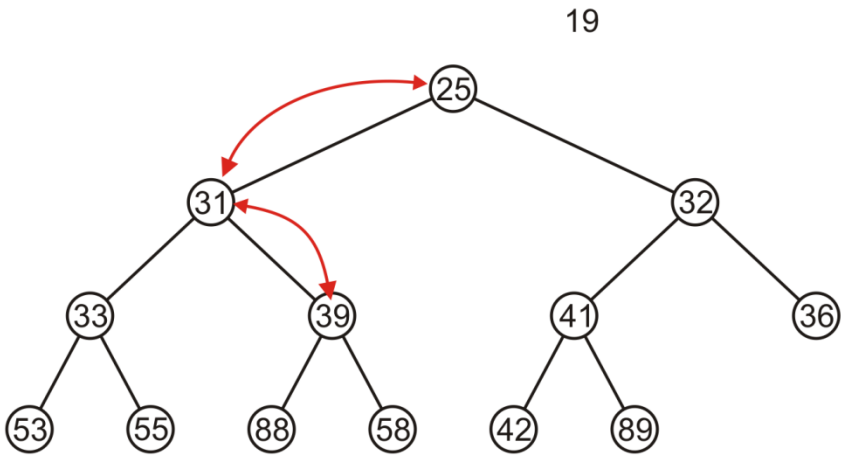
Complete Trees: Pop 19

Popping 19 copies up 39



Complete Trees: Pop 19

Which is then percolated down to the second deepest level



Run-time Analysis

Assume min heap, balanced using the layout of complete tree.

Accessing the **top** object is $\Theta(1)$

Popping the top object is $\mathbf{O}(\ln(n))$

- We copy something that is already in the lowest depth—it will likely be moved back to the lowest depth

How about **push** and **remove**?

Run-time Analysis: Push

Recall our insertion works bottom-up (percolation up)

Worst case: If we are inserting an object less than the root (at the front), then the run time will be $\mathbf{O}(\ln(n))$

Best case: If we insert an object greater than any object (at the back), then the run time will be $\mathbf{O}(1)$

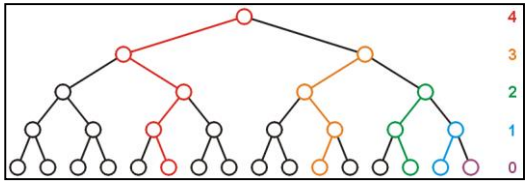
Average Case? This is tricky to answer

- Will it be $\mathbf{O}(\ln(n))$?

Run-time Analysis: Push

- **Assumption**
  - Previously inserted n values were drawn from a distribution $\mathbf{U}$
  - To be inserted value x is also drawn from the same distribution $\mathbf{U}$

- **Analysis**
  - $n/2$ nodes are at height h (the leaves)
    - At the ½ probability, x is less than $n/2$ nodes
    - At the ½ probability, we need at least one percolation up
  - $n/4$ nodes are at height h-1
    - At the ¼ probability, x is less than $n/4$ nodes
    - At the ¼ probability, we need at least two percolation up
  - …
  - 1 node is at height 0 (the root)



So the expected number of percolation up is

$$\frac{1}{2}\cdot 1+\frac{1}{4}\cdot 2+\frac{1}{8}\cdot 3+\cdots=\sum_{k=1}^{\infty}\frac{k}{2^k}=2$$

Therefore, we have **an average run time of $O(1)$**

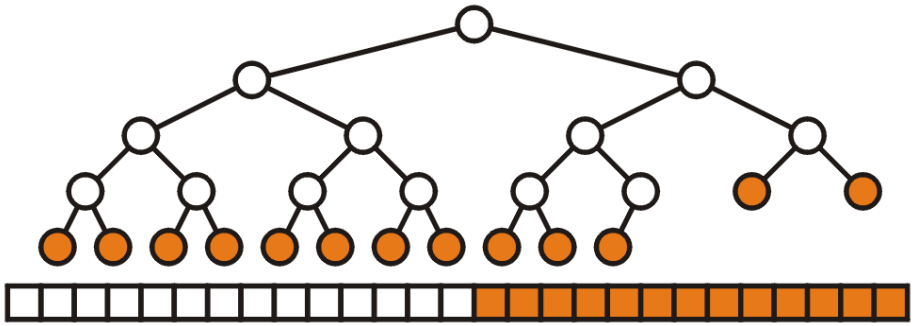
Run-time Analysis: Remove

To remove an arbitrary priority, it takes $\mathbf{O}(n)$

- All entries must be checked to remove an arbitrary one

To remove the largest priority, it still takes $\mathbf{O}(n)$

- All leaf nodes must be checked, and there are approximately $n/2$ leaf nodes:



Run-time Analysis

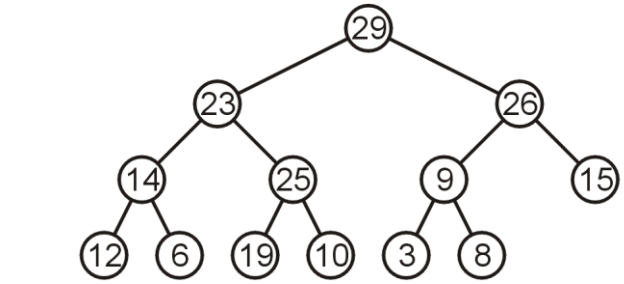
To summarize, the time complexity would be:

	Average	Worst
Top (Find min)	$\mathbf{O}(1)$	$\mathbf{O}(1)$
Pop (Delete min)	$\mathbf{O}(\ln(n))$	$\mathbf{O}(\ln(n))$
Insert	$\mathbf{O}(1)$	$\mathbf{O}(\ln(n))$

Binary Max Heaps

A binary max-heap is identical to a binary min-heap except that the parent is always larger than either of the children

For example, the same data as before stored as a max-heap yields



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	29	23	26	14	25	9	15	12	6	19	10	3	8		

Other Heaps

Other heaps have its own unique run-time characteristics

- Leftist, skew, binomial and Fibonacci heaps all use a node-based implementation requiring $\Theta(n)$ additional memory
- For Fibonacci heaps, the run-time of all operations (including merging two Fibonacci heaps) except pop are $\Theta(1)$

Summary

In this talk, we have:

- Discussed binary heaps
- Looked at an implementation using arrays
- Analyzed the run time:
  - Head $\Theta(1)$
  - Push $\Theta(1)$ average
  - Pop $O(\ln(n))$
- Discussed implementing priority queues using binary heaps

References

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §7.2.3, p.144.

[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990, §7.1-3, p.140-7.

[3] Weiss, *Data Structures and Algorithm Analysis in C++, 3rd Ed.*, Addison Wesley, §6.3, p.215-25.