Binary Heap and Priority Queue

Outline

- Priority Queue
- Examples of Priority Queue
- Implementation details of Priority Queue
- Binary Heap
  - Min Heap
  - Max Heap

Priority Queue

With queues

- The order may be summarized by *first in, first out*

If each object is associated with a priority, we may wish to pop that object which has highest priority

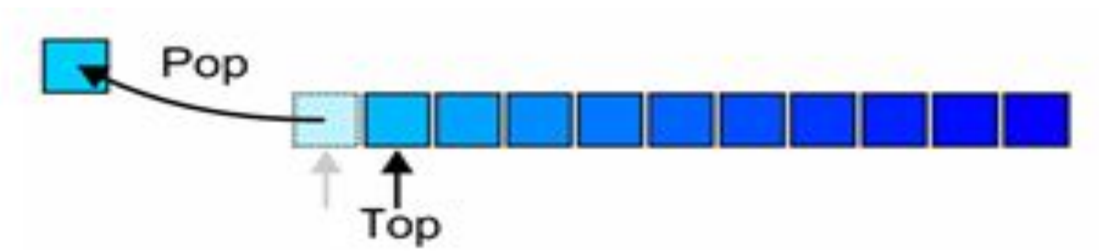
With each pushed object, we will associate a nonnegative integer (0, 1, 2, ...) where:

- The value 0 has the *highest* priority, and
- The higher the number, the lower the priority

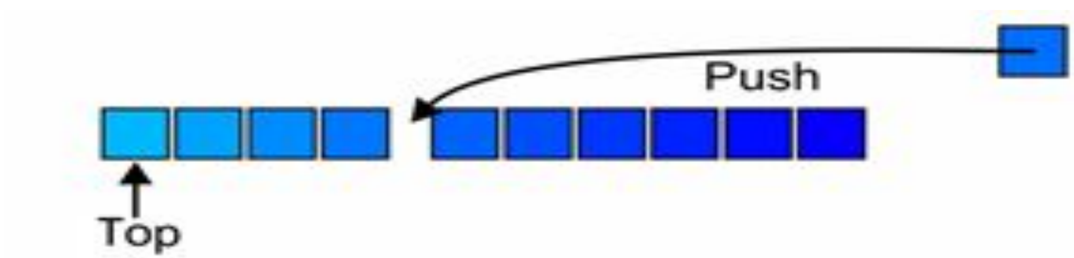
Operations

The top of a priority queue is the object with highest priority

Popping from a priority queue removes the current highest priority object:



Push places a new object into the appropriate place



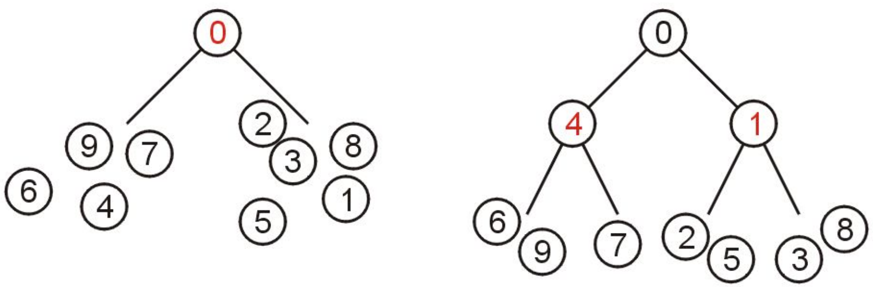
Heap

- Heap is a tree with the highest priority at the root.
- We will look at binary heaps
- Numerous other heaps exists:
  - D-ary heaps
  - Leftlist heaps
  - Skew heaps
  - Binomial heaps
  - Fibonacci heaps
  - Bi-parental heaps

Heap

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

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

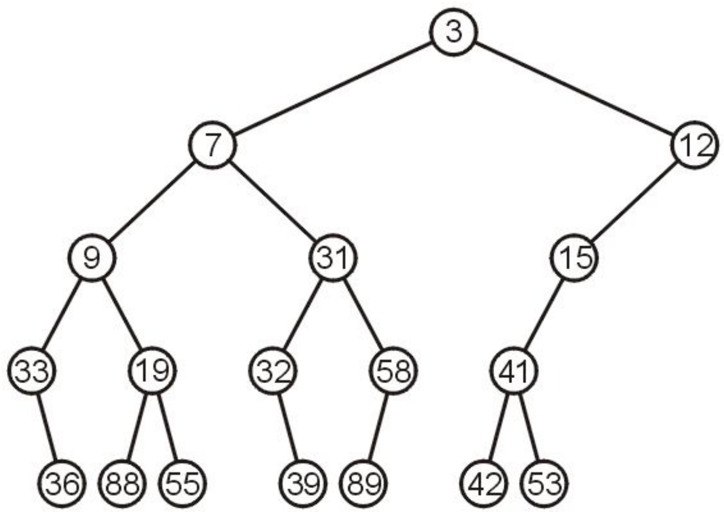


From this definition:

- 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:



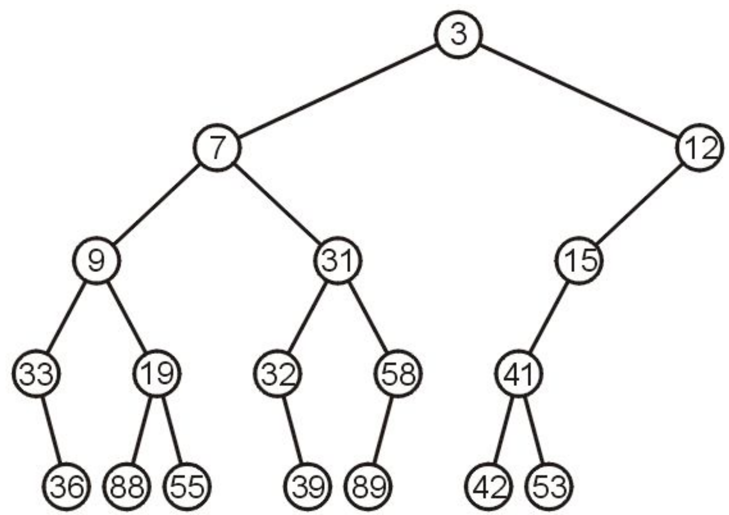
Operations on Heap

We will consider three operations:

- Top
- Pop
- Push

Example

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



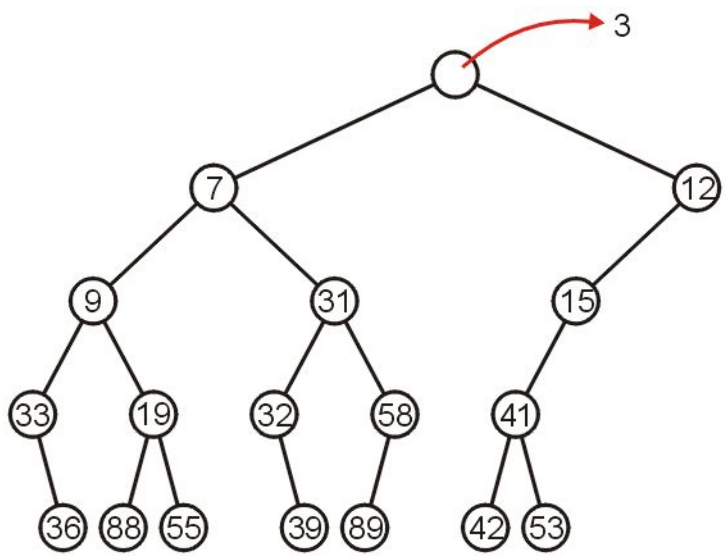
Pop

To remove the minimum object:

- Promote the node of the sub-tree which has the least value
- Recurs down the sub-tree from which we promoted the least value

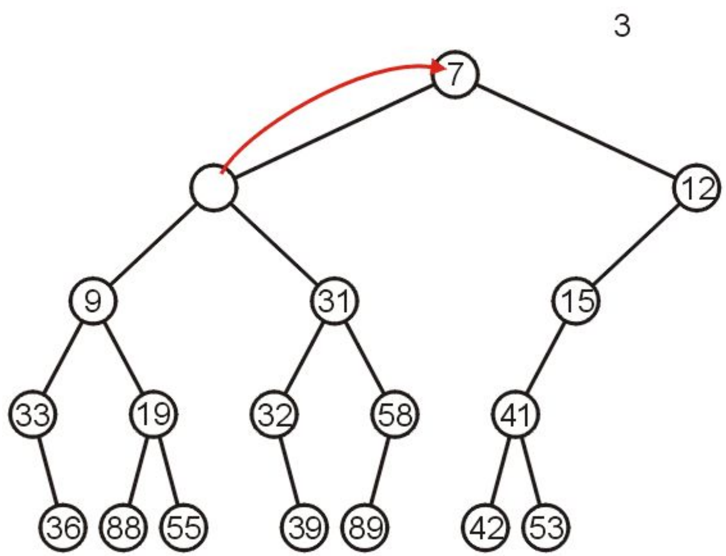
Pop

Using our example, we remove 3:



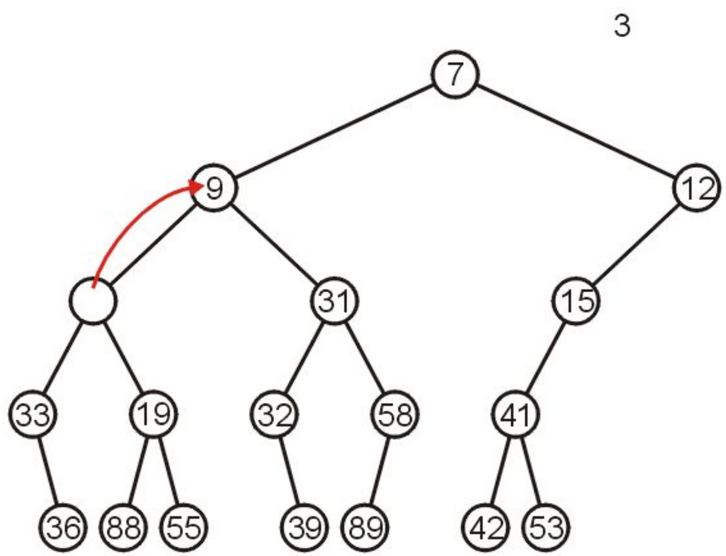
Pop

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



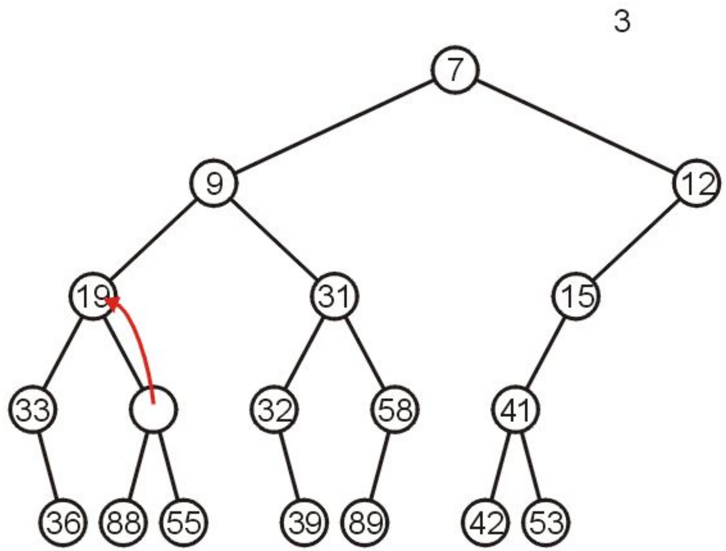
Pop

In the left sub-tree, we promote 9:



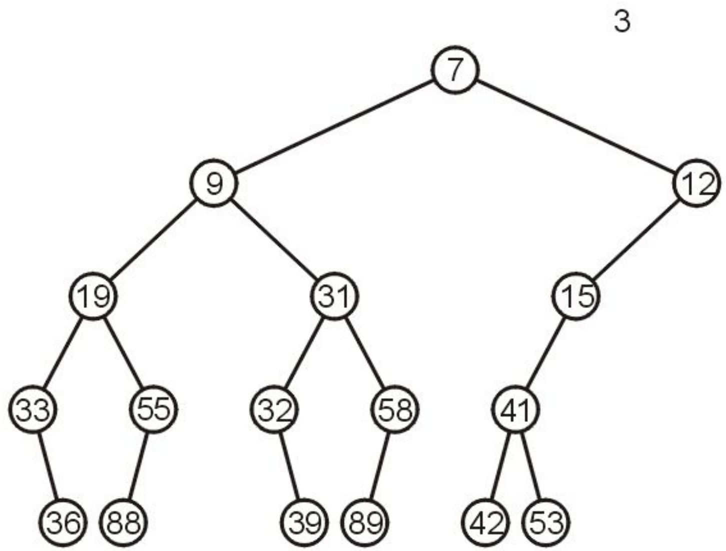
Pop

Recursively, we promote 19:



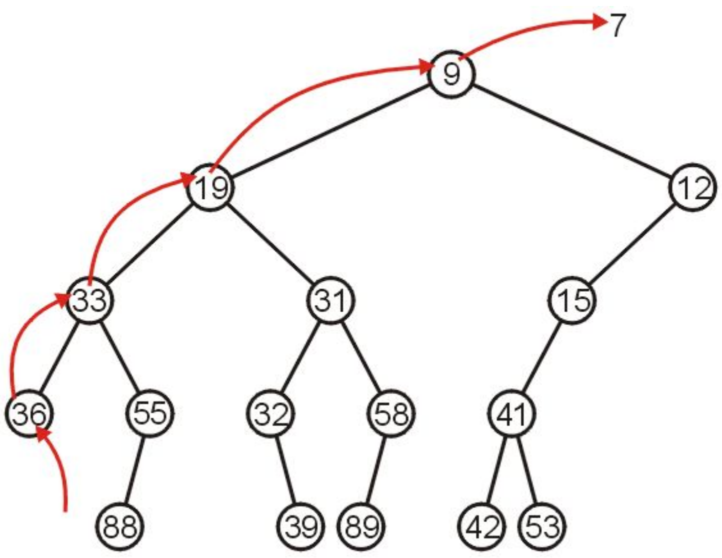
Pop

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



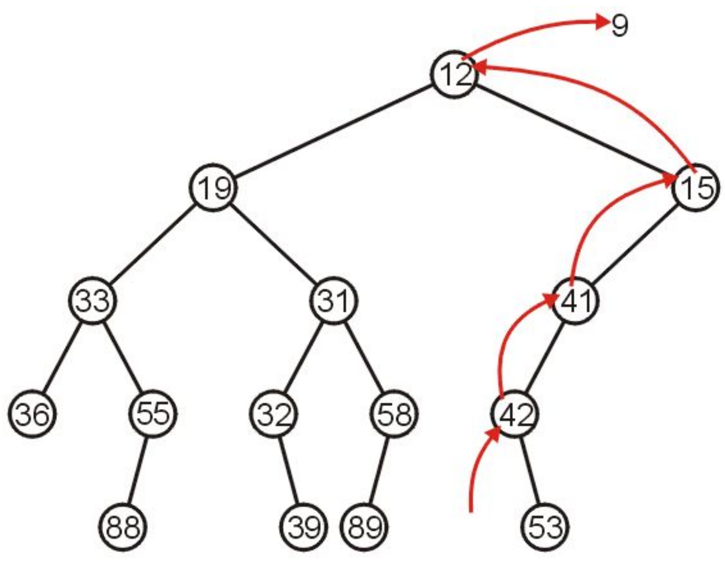
Pop

Repeating this operation again, we can remove 7:



Pop

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



Push

Inserting into a heap may be done either:

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

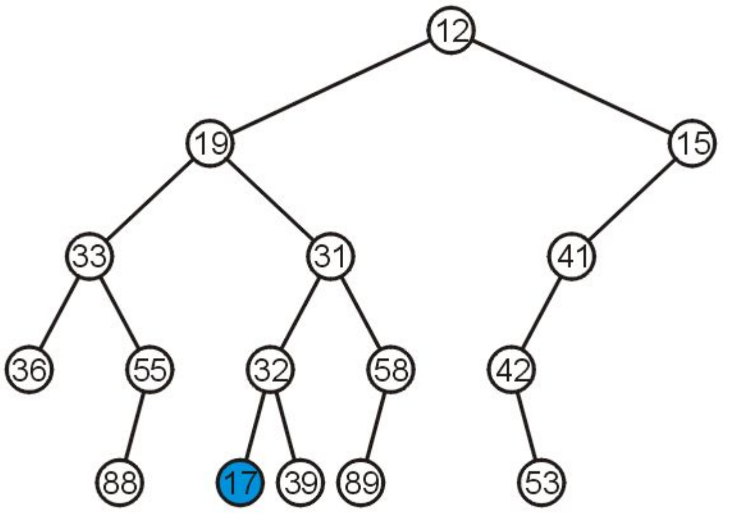
We will use the first approach with binary heaps

- Other heaps use the second

Push

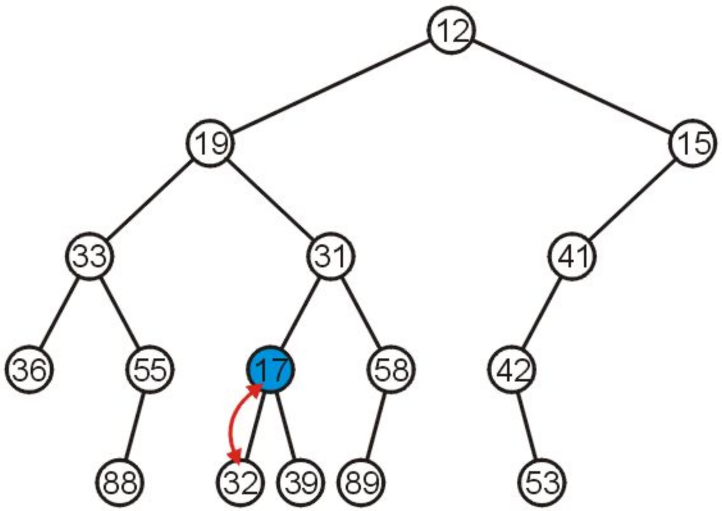
Inserting 17 into the last heap

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



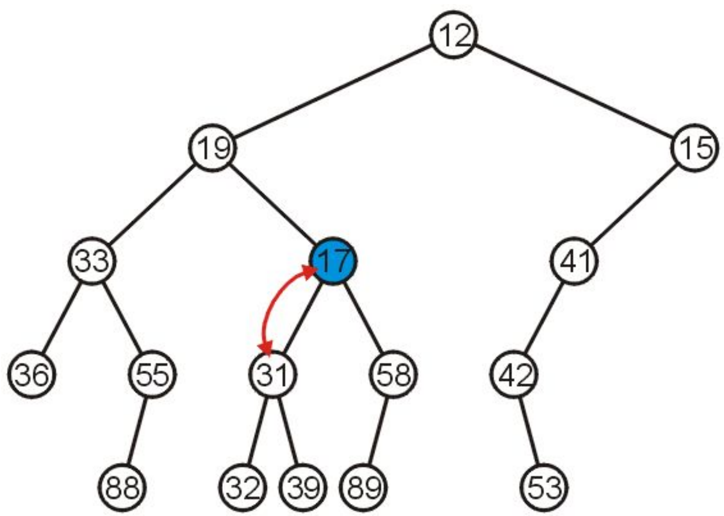
Push

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



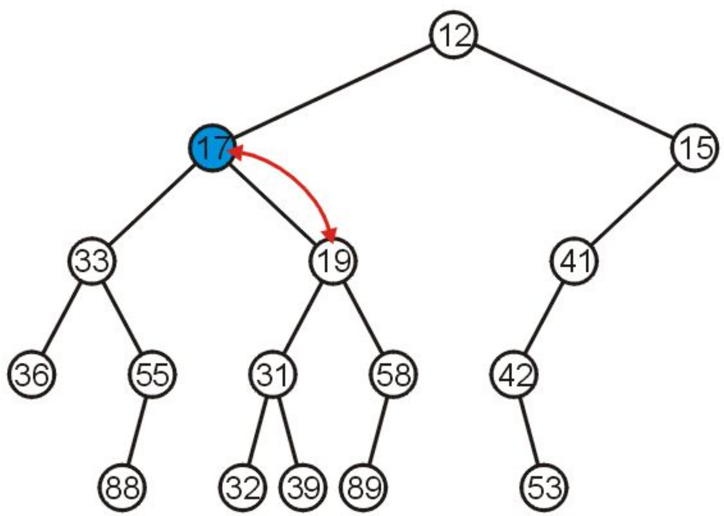
Push

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



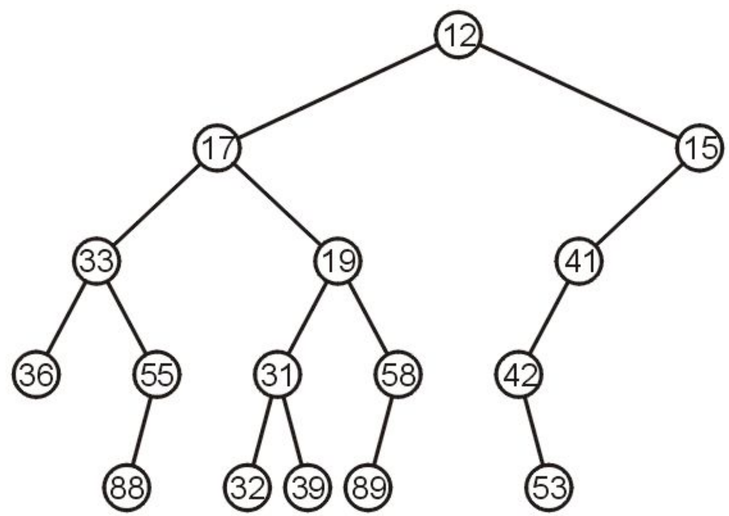
Push

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



Push

The node 17 is greater than 12 so we are finished



Push

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

Implementation Details

By using complete binary trees, we will be able to maintain, with minimal effort, the complete tree structure

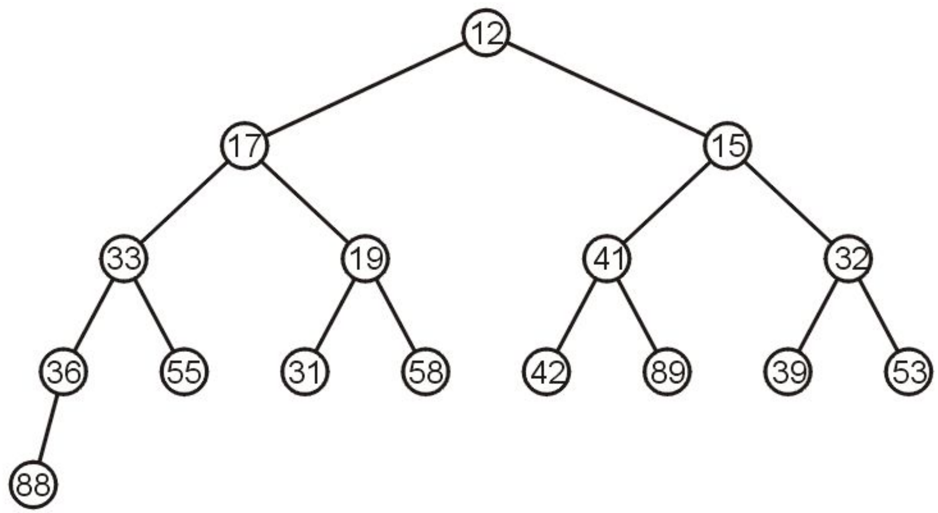
We have already seen

- 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!

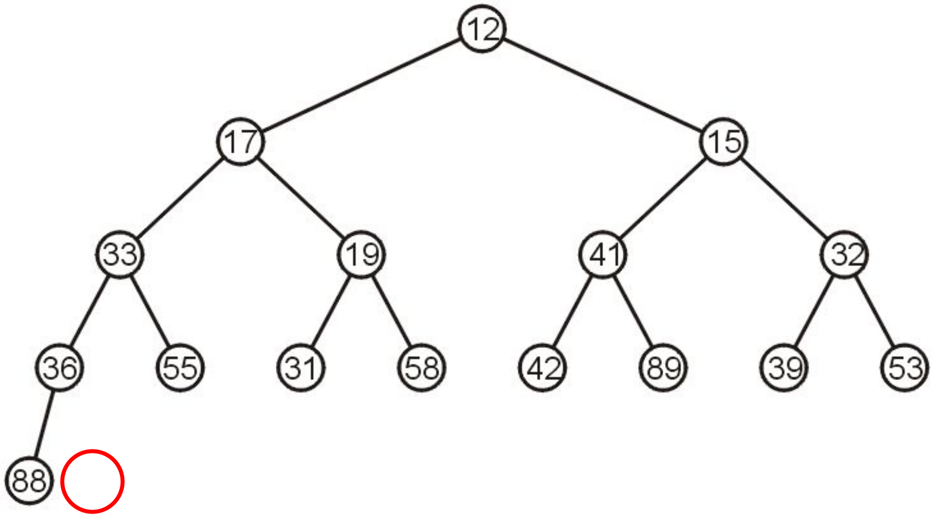
Example

For example, the previous heap may be represented as the following (non-unique!) complete tree:



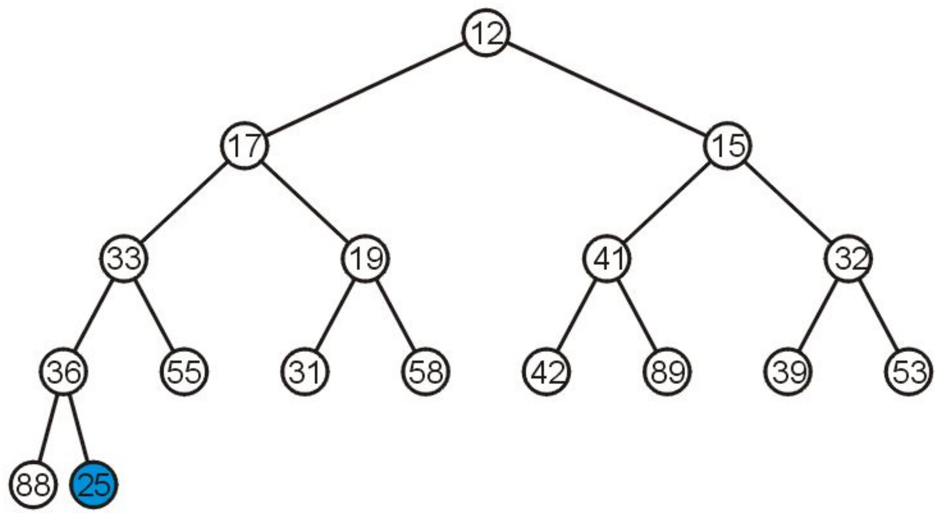
Complete Trees: Push

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



Complete Trees: Push

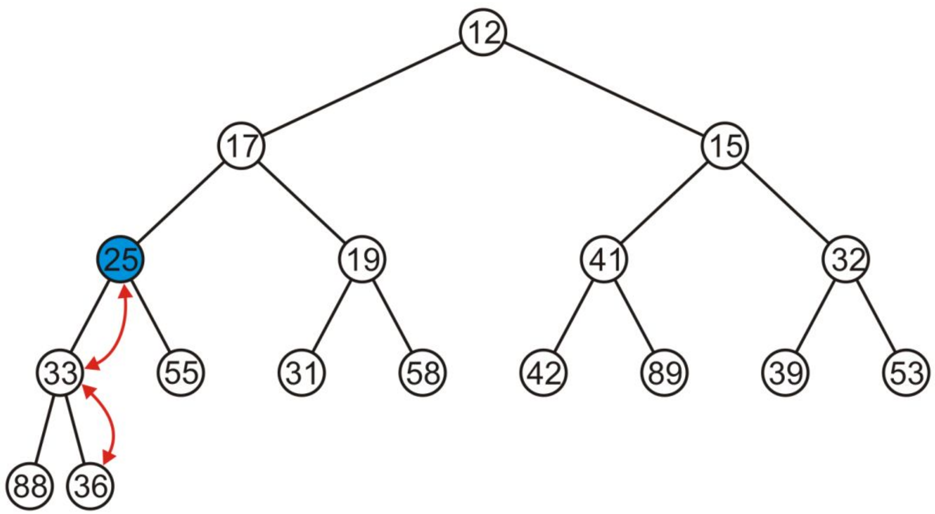
For example, push 25:



Complete Trees: Push

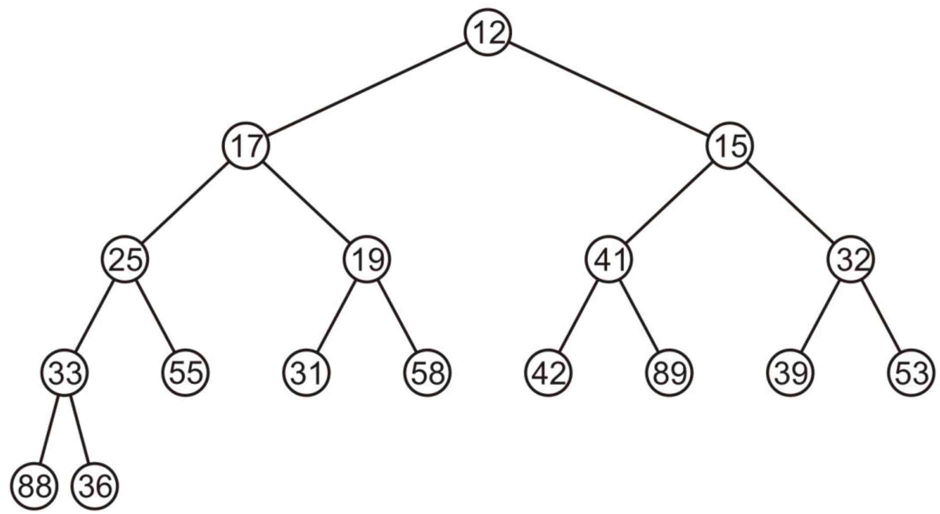
We have to percolate 25 up into its appropriate location

- The resulting heap is still a complete tree



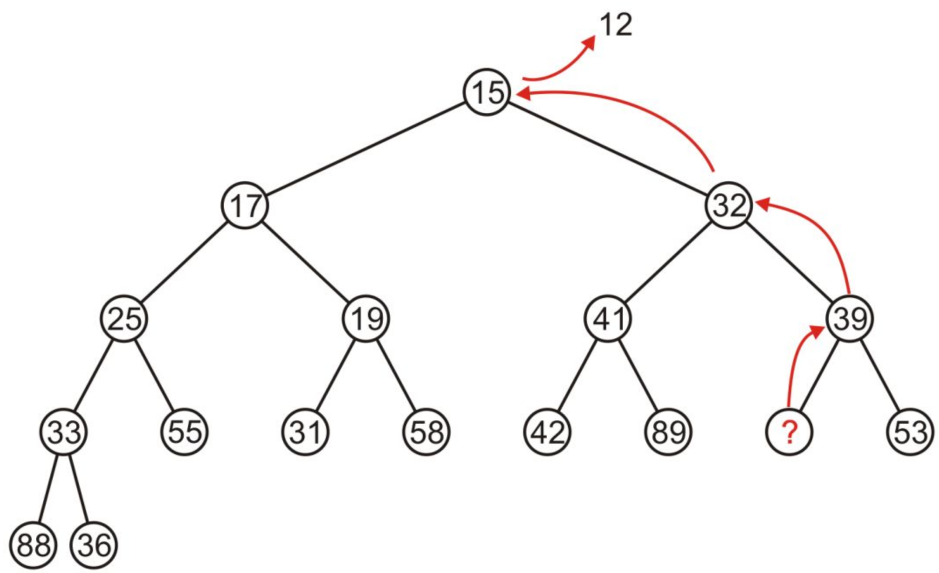
Complete Trees: Pop

Suppose we want to pop the top entry: 12



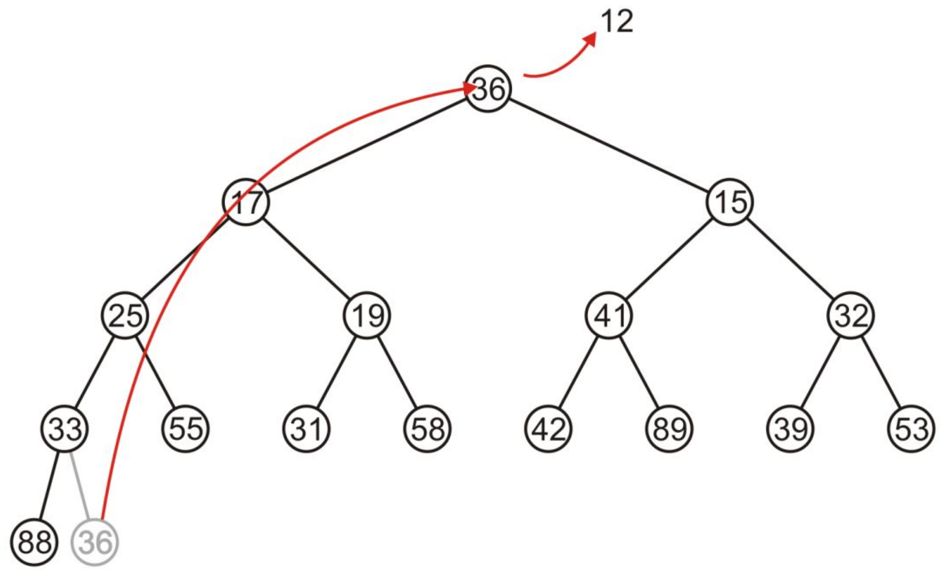
Complete Trees: Pop

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



Complete Trees: Pop

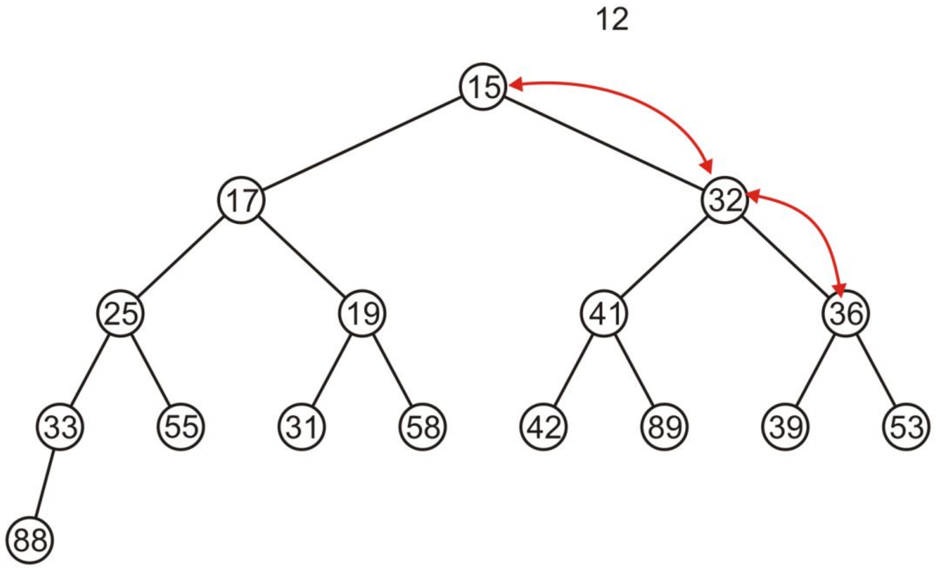
Alternatively, copy the last entry in the heap to the root



Complete Trees: Pop

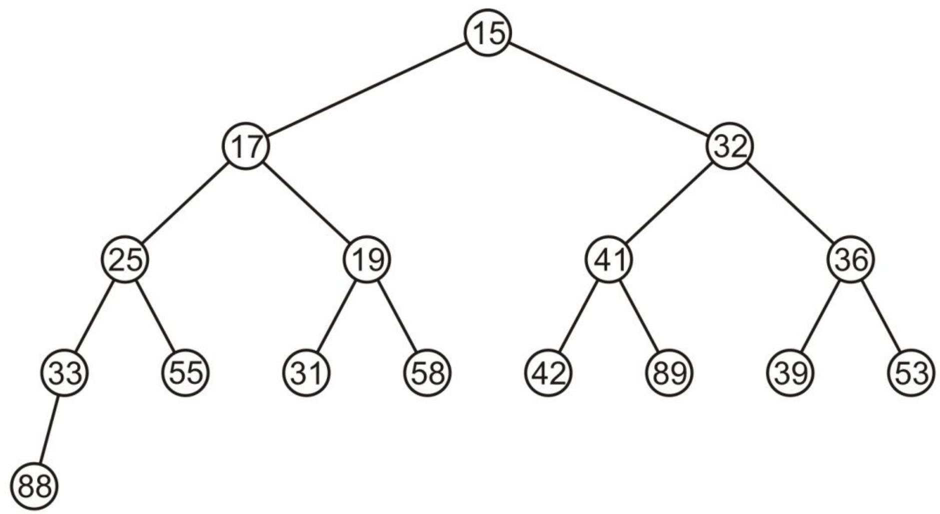
Now, percolate 36 down swapping it with the smallest of its children

– We halt when both children are larger



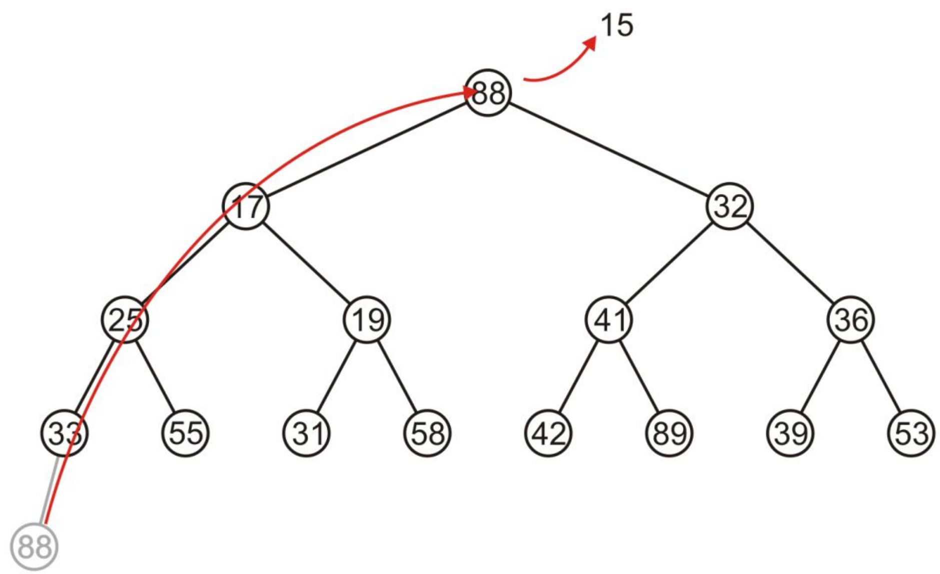
Complete Trees: Pop

The resulting tree is now still a complete tree:



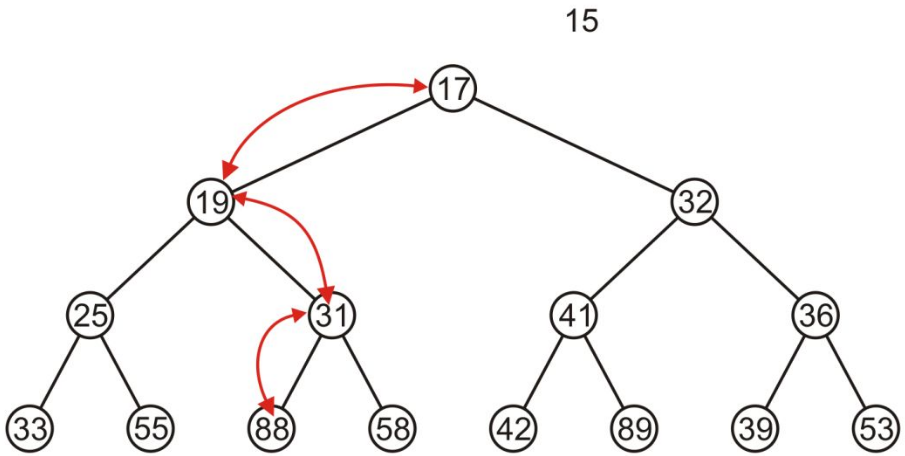
Complete Trees: Pop

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



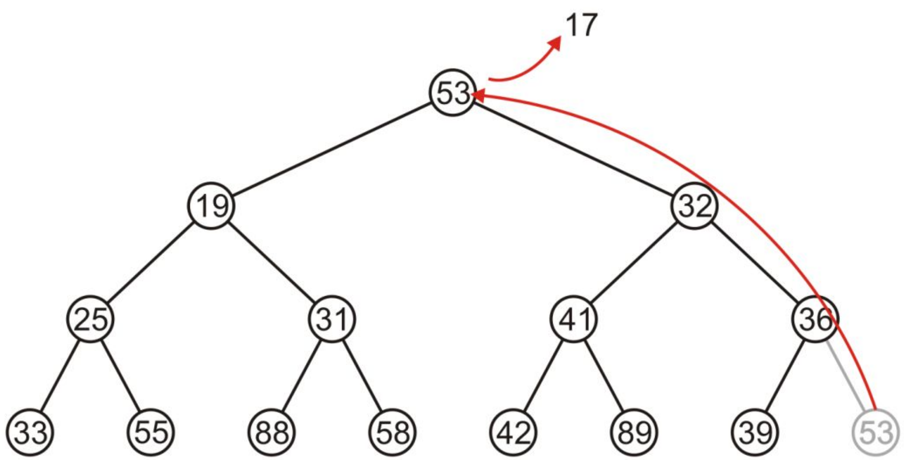
Complete Trees: Pop

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



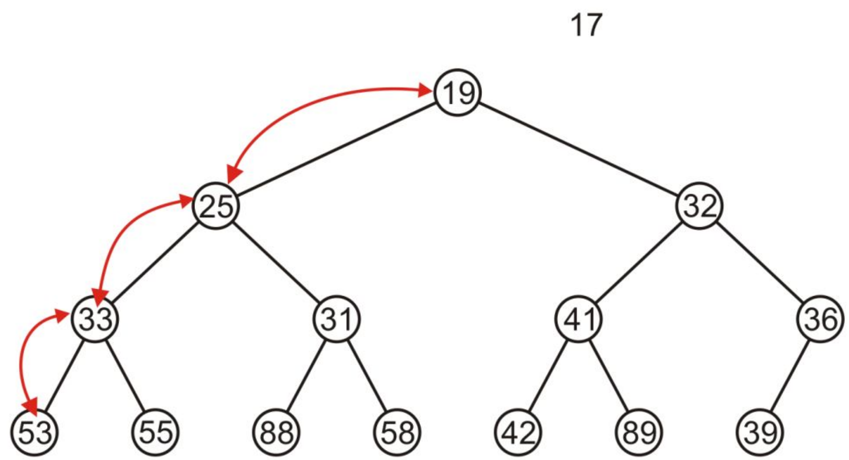
Complete Trees: Pop

In popping 17, 53 is moved to the top



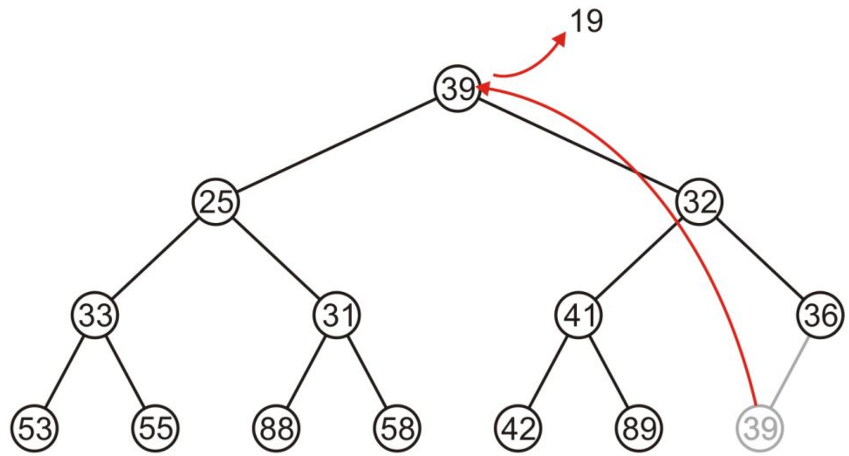
Complete Trees: Pop

And percolated down, again to the deepest level



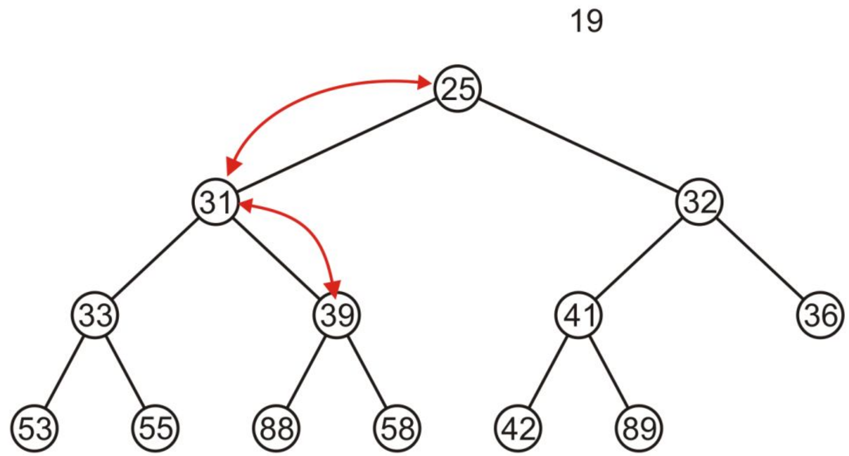
Complete Trees: Pop

Popping 19 copies up 39



Complete Trees: Pop

Which is then percolated down to the second deepest level



Complete Tree

Therefore, we can maintain the complete-tree shape of a heap

We may store a complete tree using an array:

- A complete tree is filled in breadth-first traversal order
- The array is filled using breadth-first traversal

Run-time Analysis

Accessing the top object is $\mathbf{\Theta}(1)$

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