

CS101 Data Structure

Heaps and Priority Queues

Textbook Ch 6

Outline

- Priority queue
- Binary heap
- Heapsort

Definition

Queues

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

Priority queues

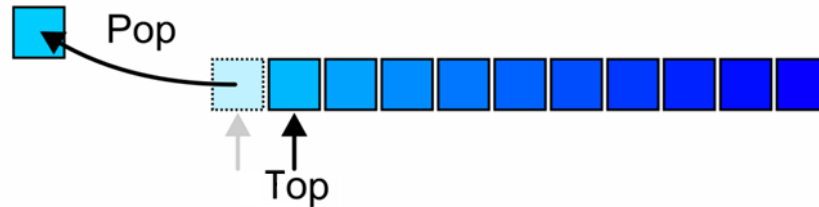
- Each object is associated with a priority
 - The value 0 has the *highest* priority, and
 - The higher the number, the lower the priority
- We pop the object which has the highest priority

Operations

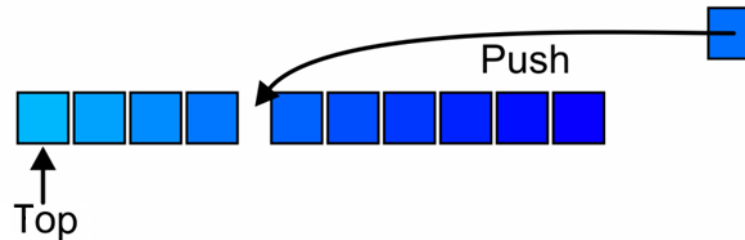
The top of a priority queue is the object with highest priority



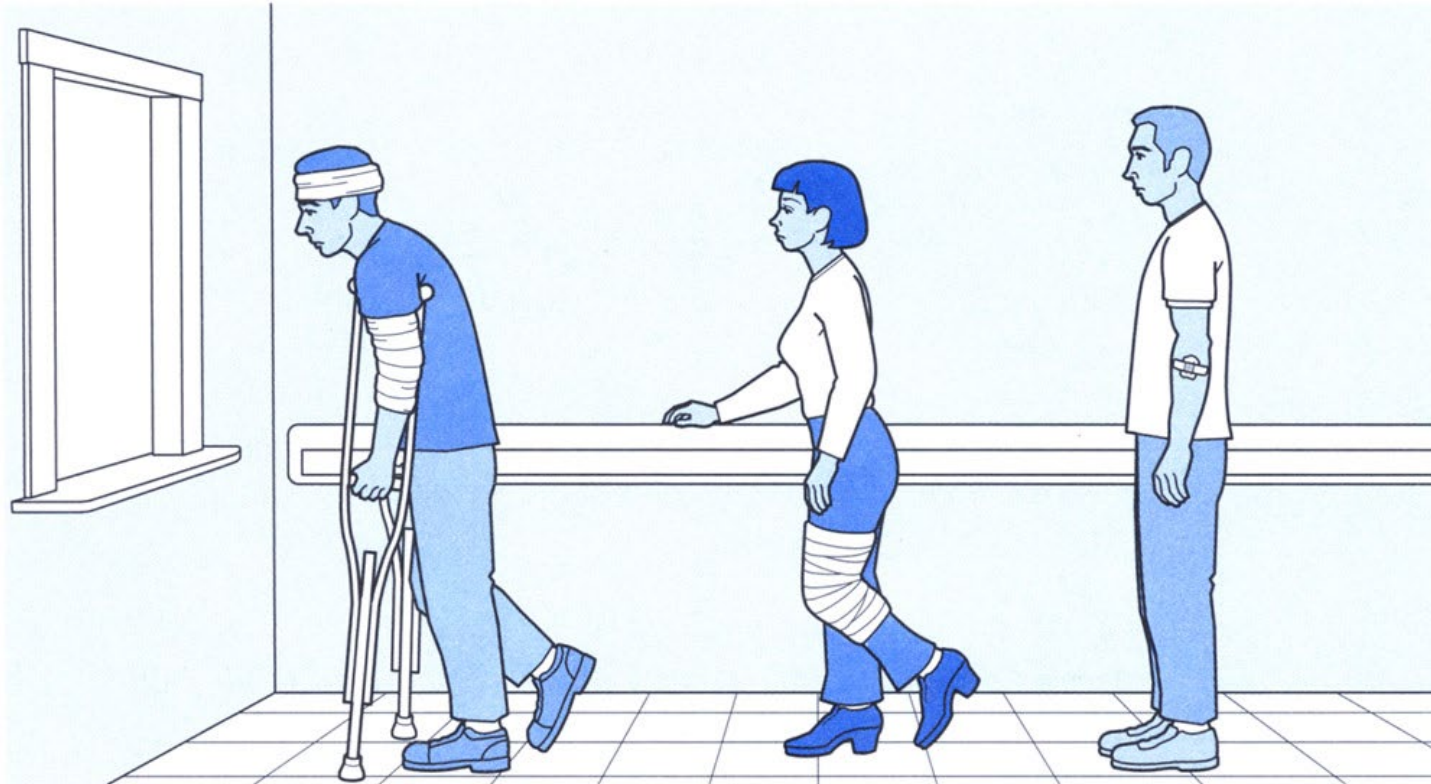
Popping from a priority queue removes the current highest priority object:



Push places a new object into the appropriate place



Application



Application

Process priority in operation systems

- In Unix, you may set the priority of a process, e.g.,

```
% nice +15 ./a.out
```

reduces the priority of the execution of the routine `a.out` by 15

Implementations

Our goal is to make the run time of each operation as close to $\Theta(1)$ as possible

We will look at an implementation using a data structure we already know:

- **Multiple queues** — one for each priority

Then we will introduce a more appropriate data structure: *heap*

Multiple Queues

Assume there is a fixed number of priorities, say M

- Create an array of M queues
- Push a new object onto the queue corresponding to the priority
- Top and pop find the first non-empty queue with highest priority

Multiple Queues

The run times are reasonable:

- Push is $\Theta(1)$
- Top and pop are both $\mathbf{O}(M)$

Problems:

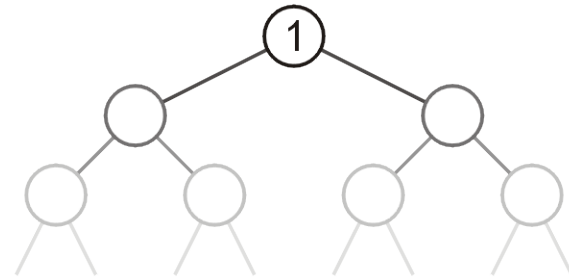
- It restricts the range of priorities
- The memory requirement is $\Theta(M + n)$

Heaps

Can we do better?

We need a *heap*

- A tree with the top object at the root
- We will look at **binary heaps**
- Numerous other heaps exists:
 - d -ary heaps
 - Leftist heaps
 - Skew heaps
 - Binomial heaps
 - Fibonacci heaps
 - Bi-parental heaps



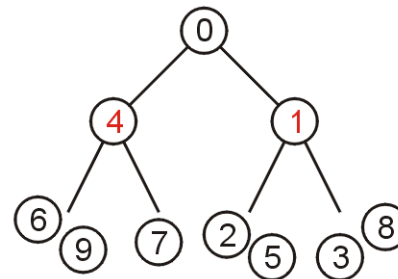
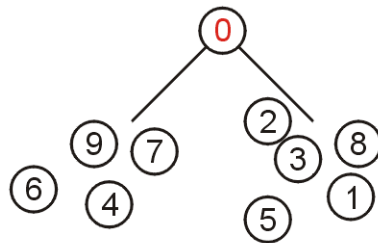
Outline

- Priority queue
- Binary heap
- Heapsort

Definition

A non-empty tree is a min-heap if

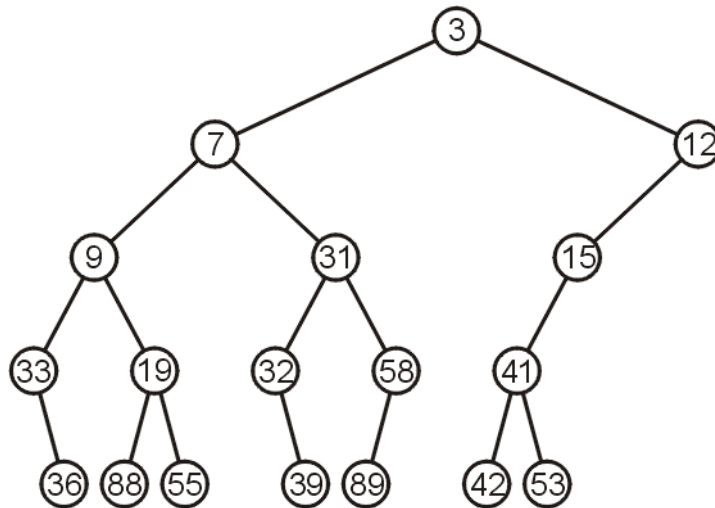
- The key associated with the root is less than or equal to the keys associated with the sub-trees (if any)
- The sub-trees (if any) are also min-heaps



There is no other relationship between the elements in the subtrees!

Example

This is a (*naïve*) binary min-heap:



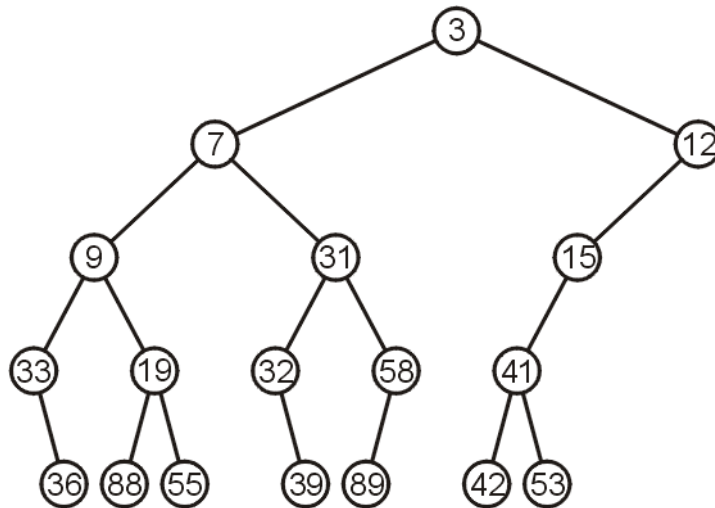
Operations

We will consider three operations:

- Top
- Pop
- Push

Example

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



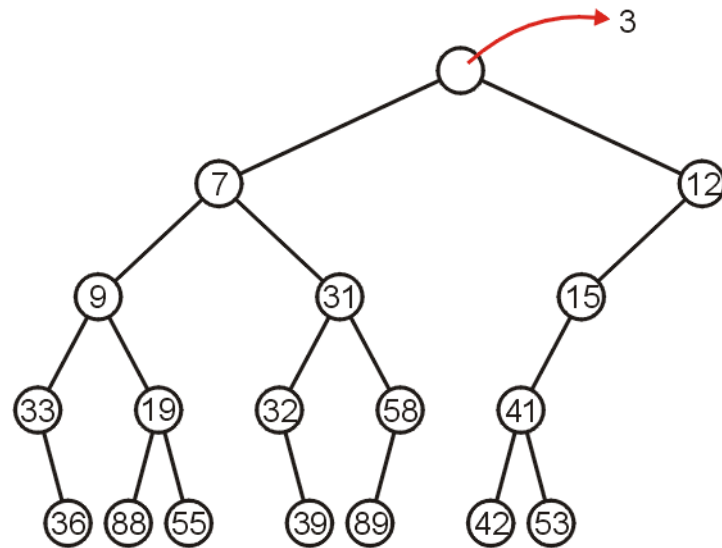
Pop

To remove the minimum object:

- Promote the node of the sub-tree which has the least value
- Recursively process the sub-tree from which we promoted the least value

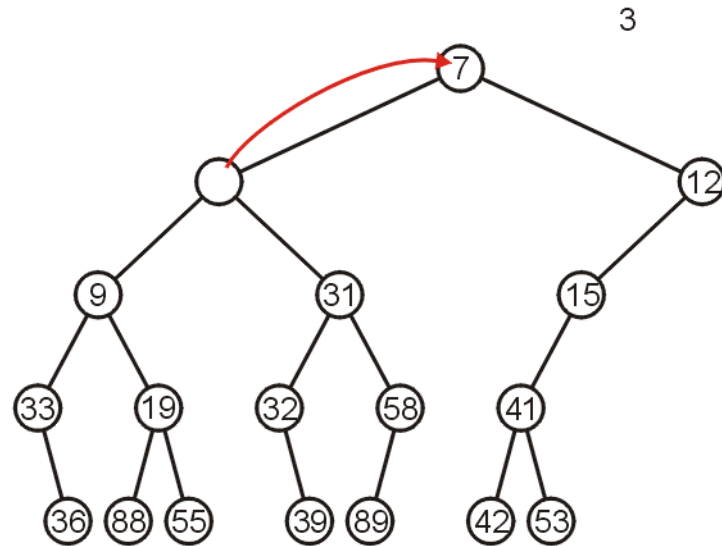
Pop

Using our example, we remove 3:



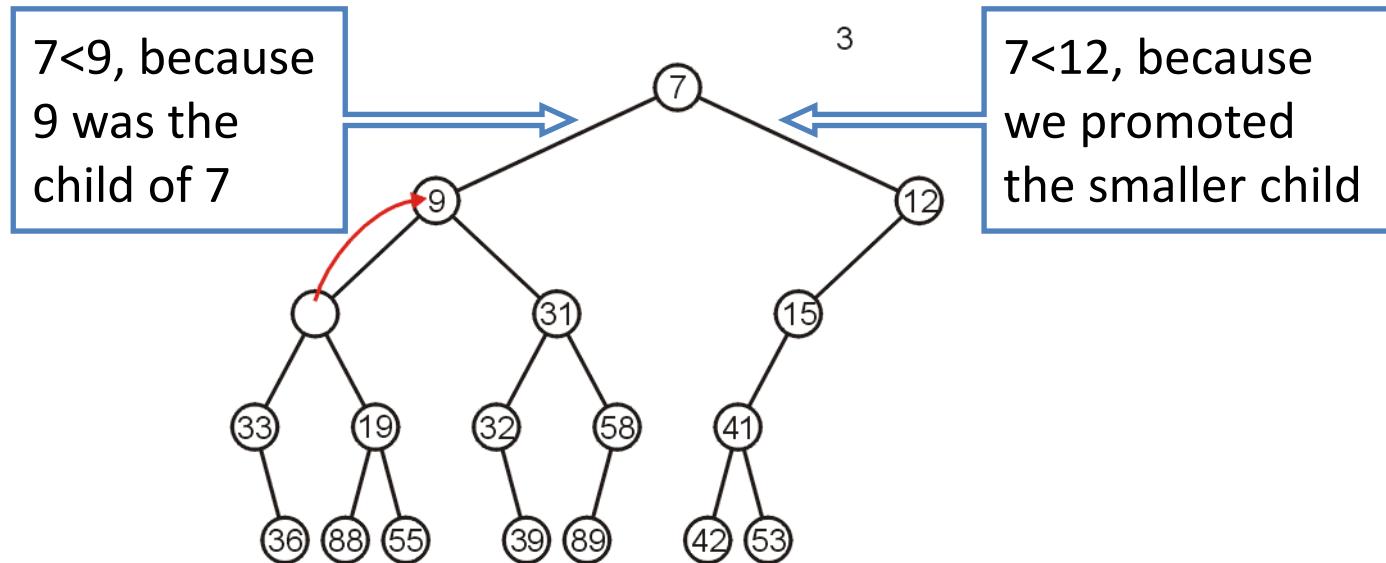
Pop

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



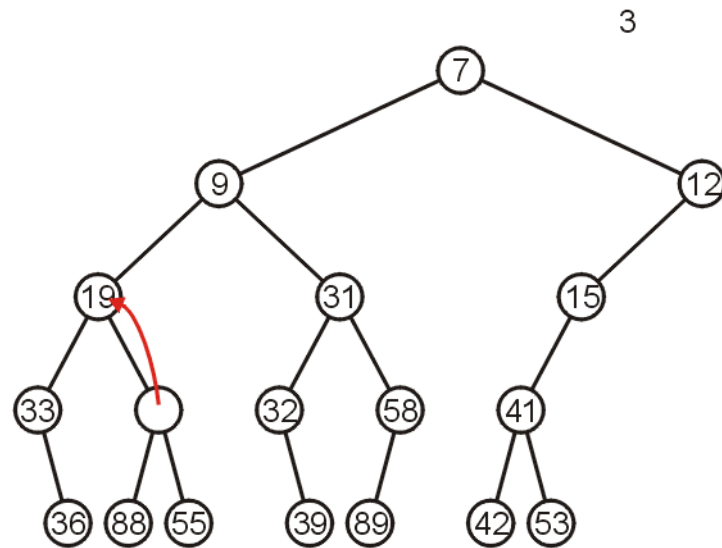
Pop

In the left sub-tree, we promote 9:



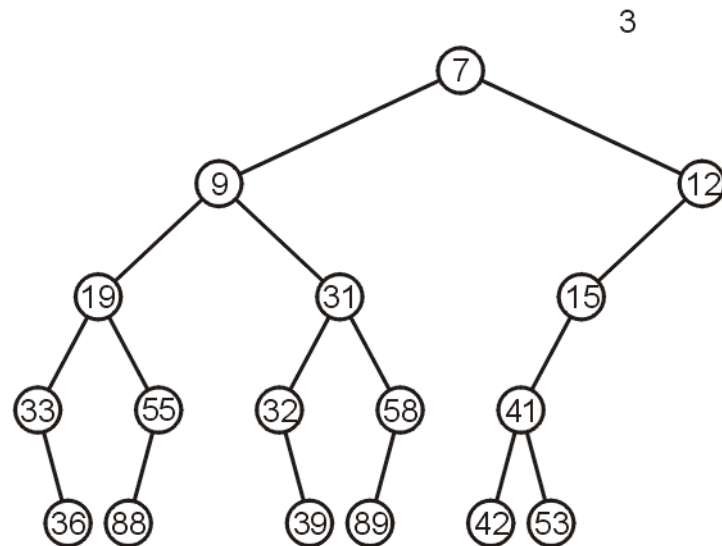
Pop

Recursively, we promote 19:



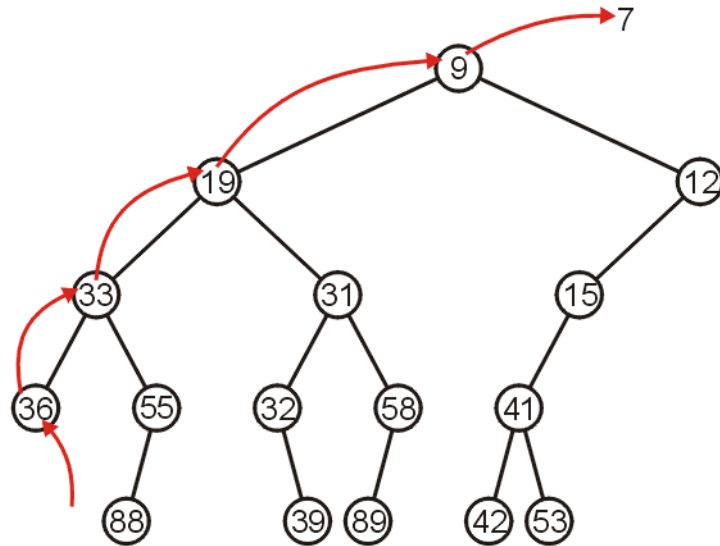
Pop

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



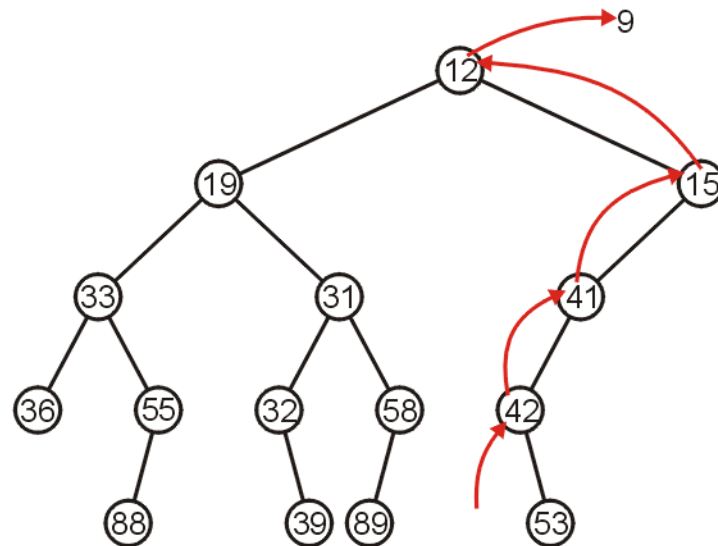
Pop

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)

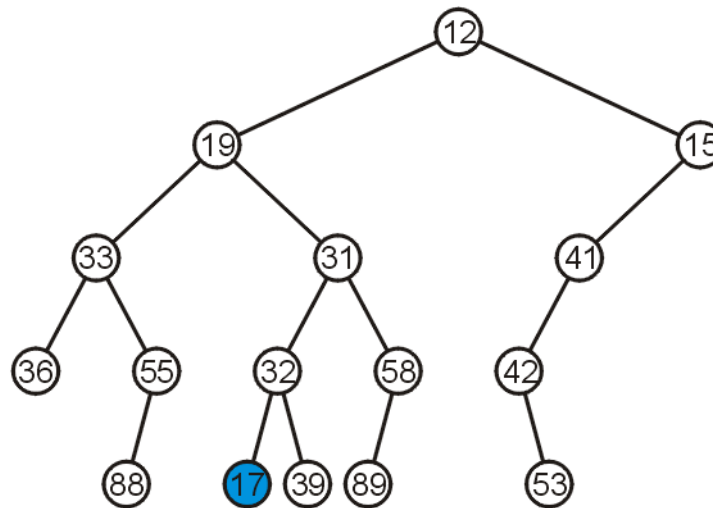
We will use the first approach with binary heaps

- Other heaps use the second

Push

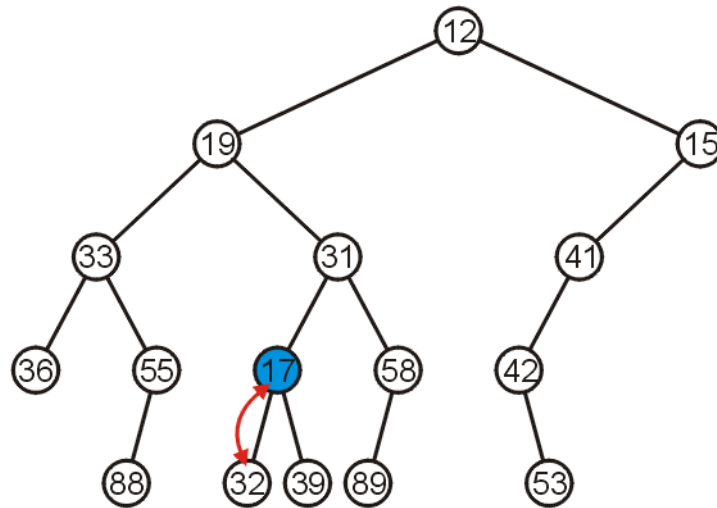
Inserting 17 into the last heap

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



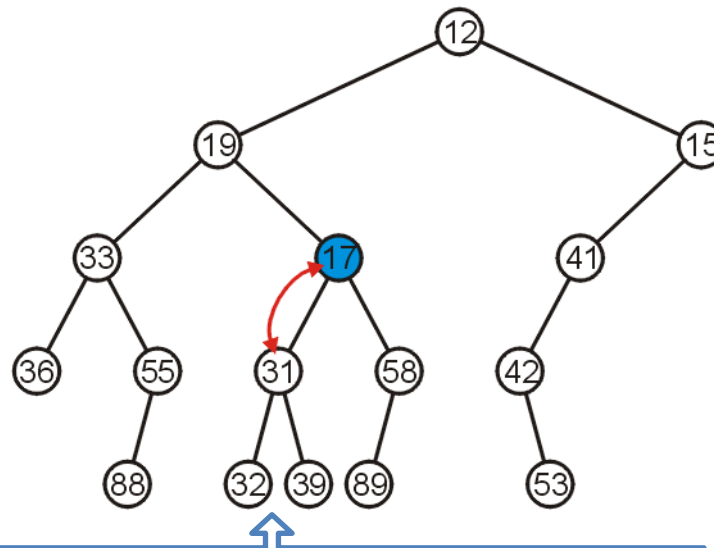
Push

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



Push

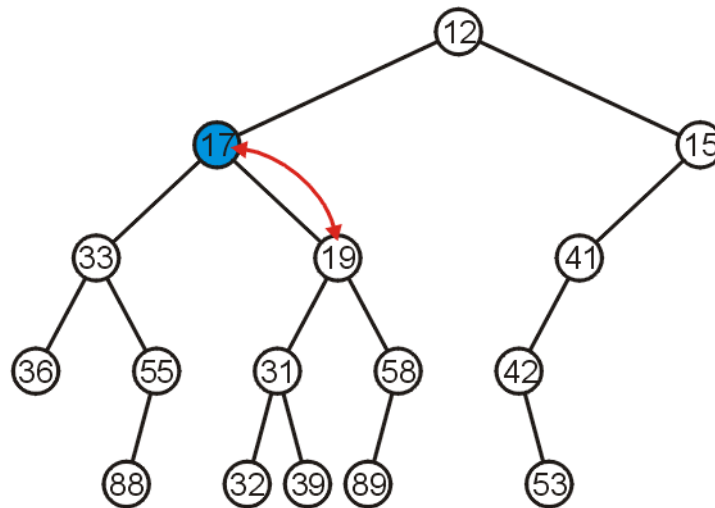
The node 17 is less than the node 31; swap them



31 is smaller than 32 and 39 because
31 was the ancestor of 32 and 39

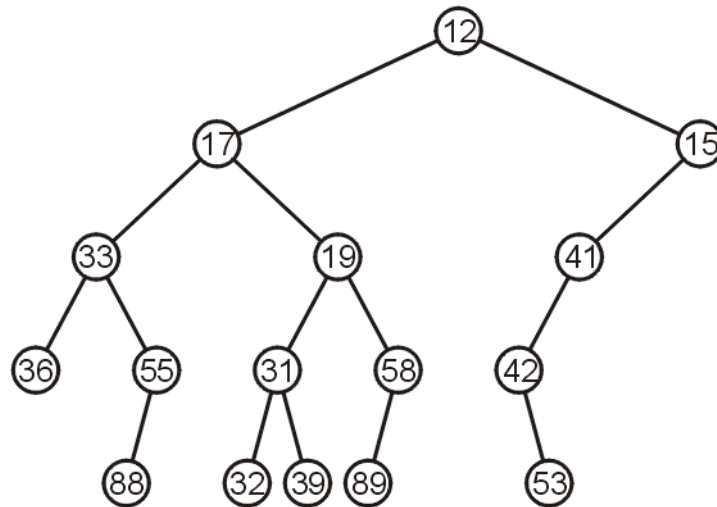
Push

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



Push

The node 17 is greater than 12 so we are finished

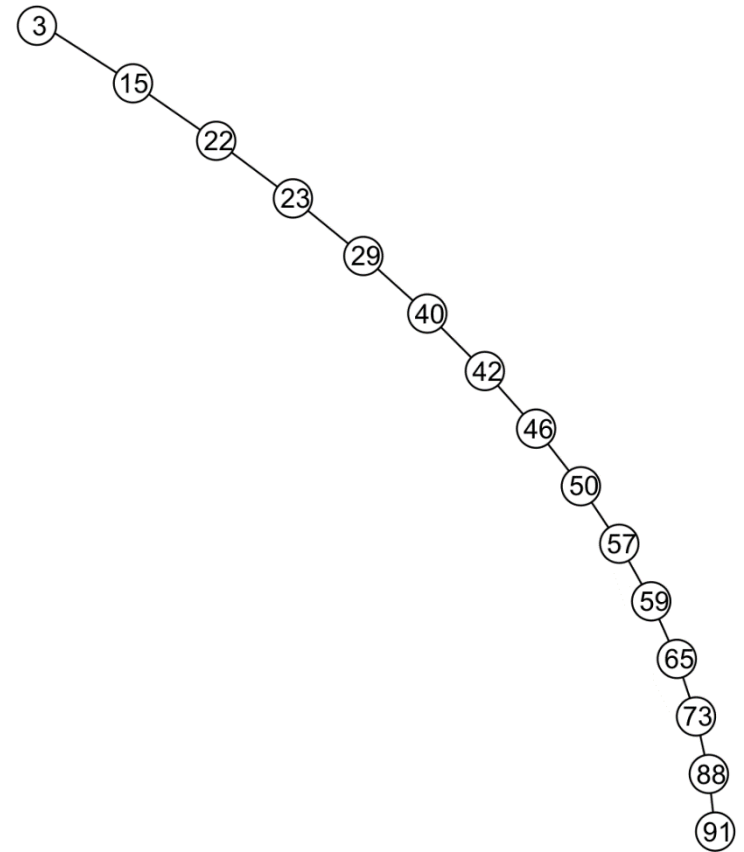


Push

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

Time Complexity

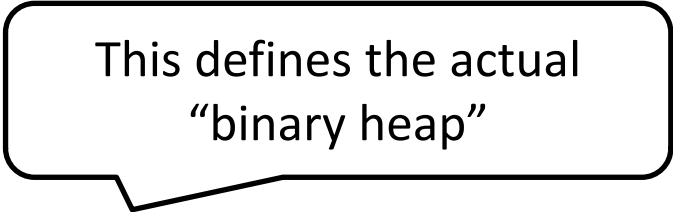
- Time complexity of pop and push?
 - $O(n)$
 - Worst case: the binary tree is highly unbalanced
- Can we do better?
 - Keep balance of the binary tree



Balance

There are multiple means of keeping balance with binary heaps:

- Complete binary trees
- Leftist heaps
- Skew heaps



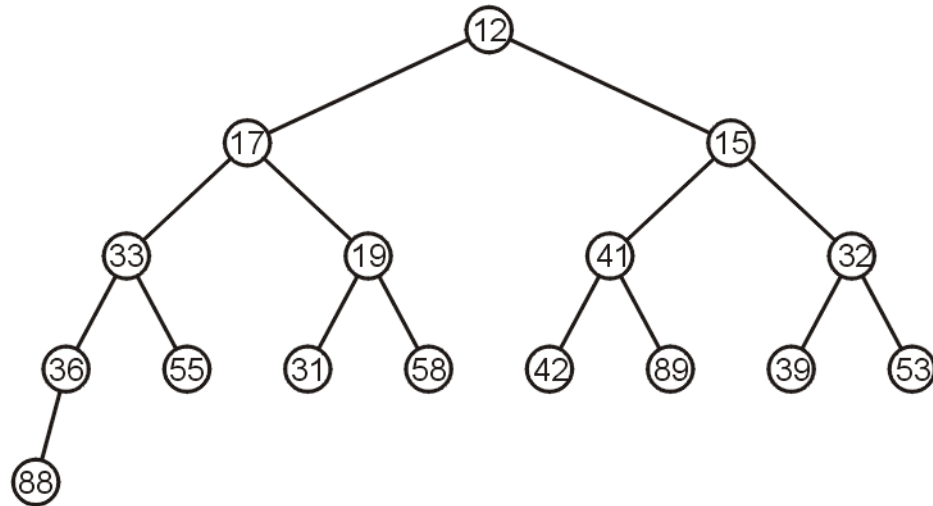
This defines the actual
“binary heap”

We will look at using **complete binary trees**

- It has optimal memory characteristics but sub-optimal run-time characteristics

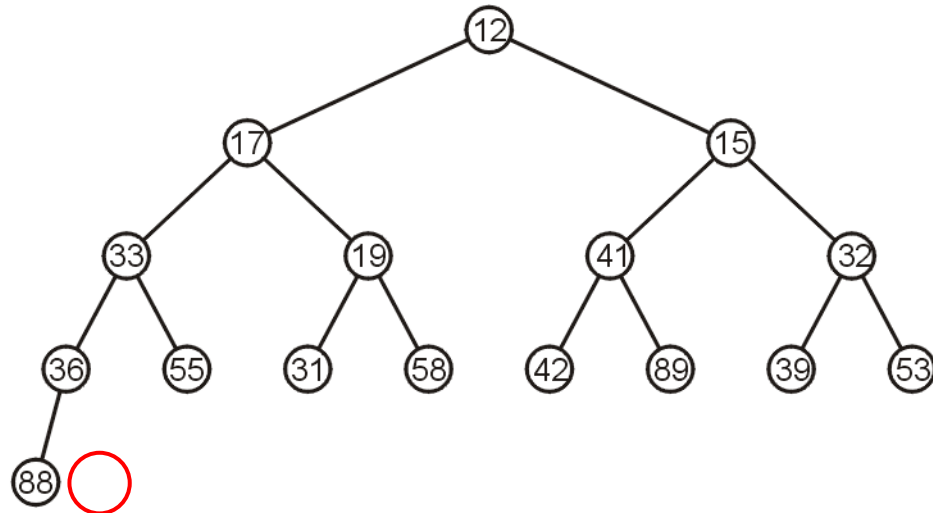
Complete Trees

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



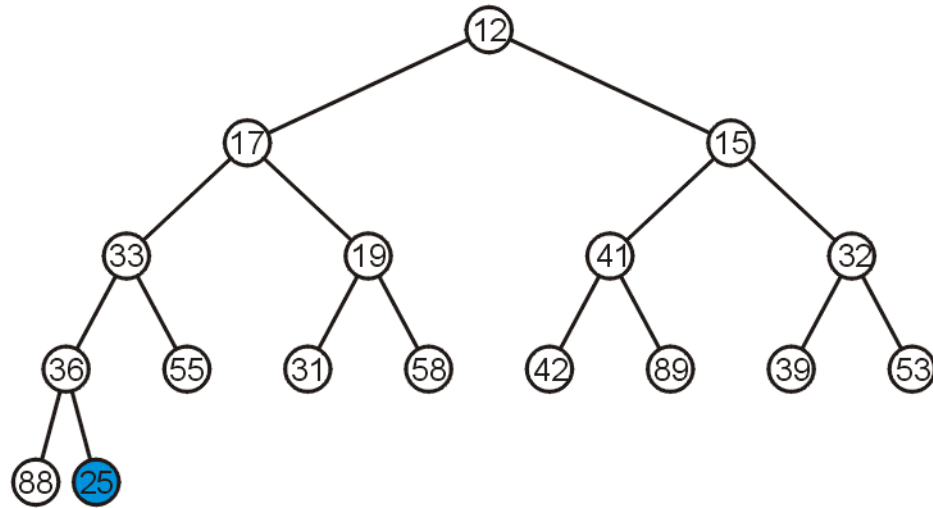
Complete Trees: Push

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

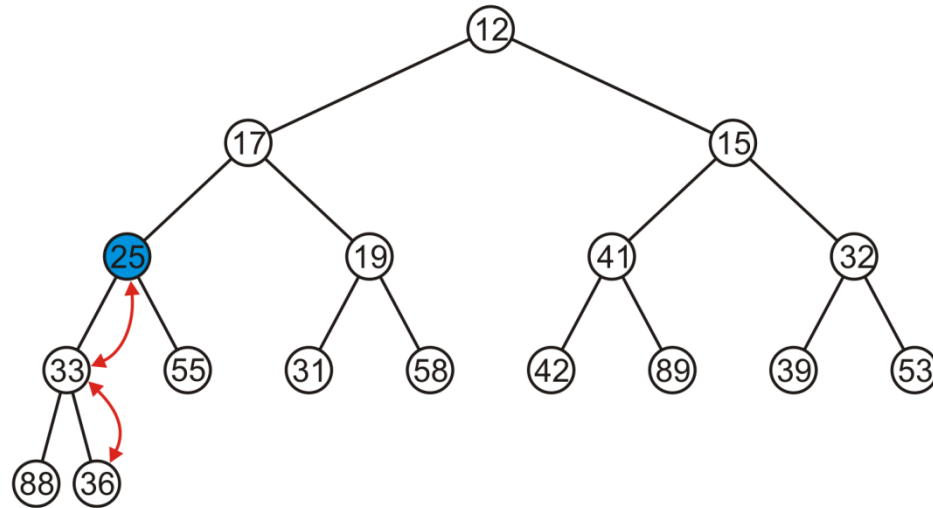
For example, push 25:



Complete Trees: Push

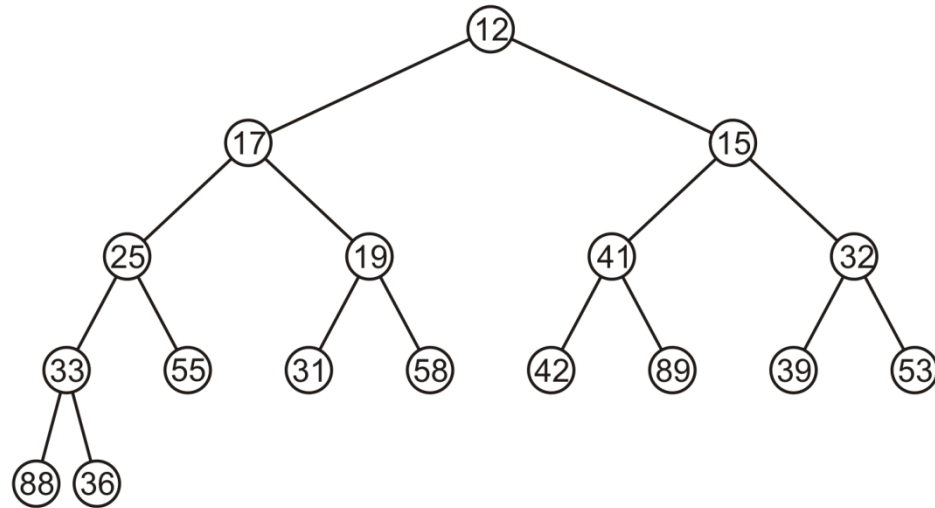
We have to percolate 25 up into its appropriate location

- The resulting heap is still a complete tree



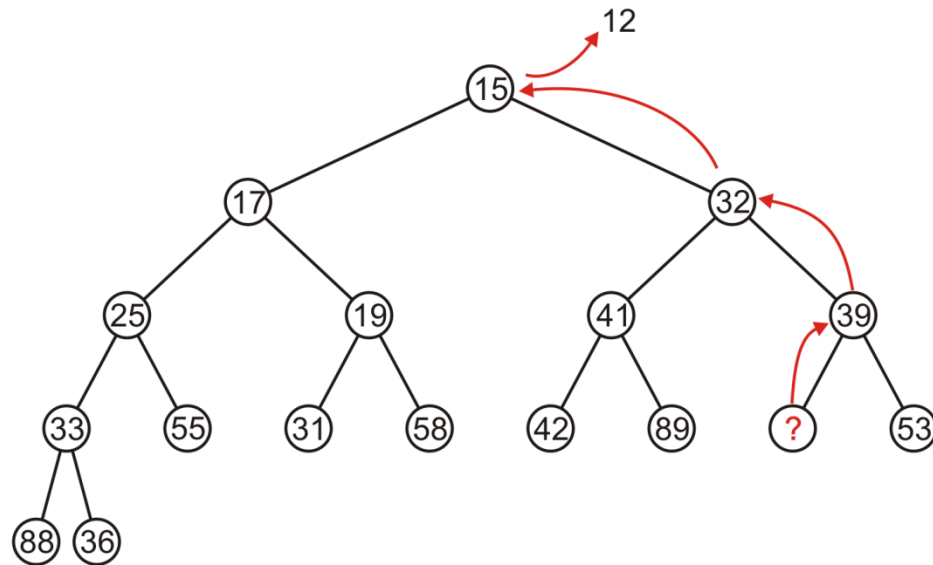
Complete Trees: Pop

Suppose we want to pop the top entry: 12



Complete Trees: Pop

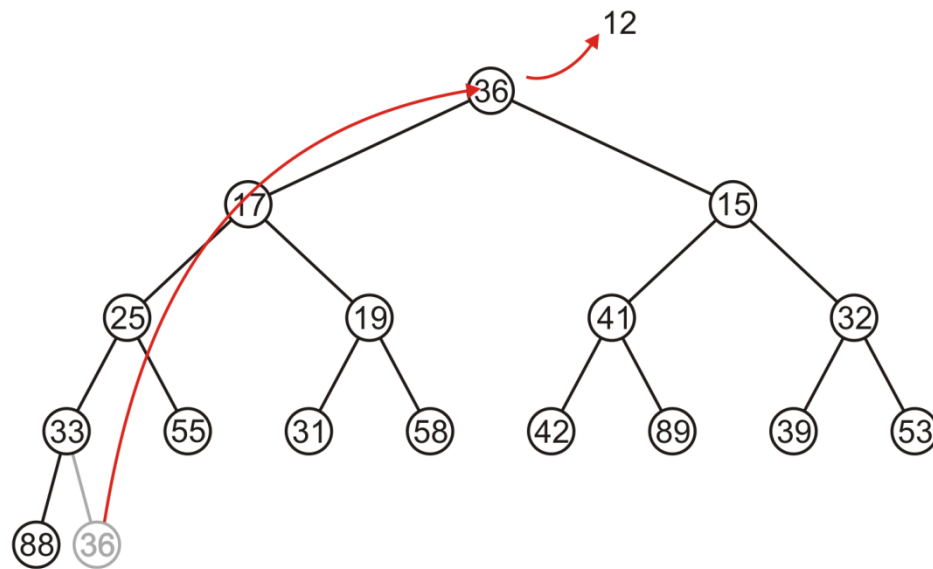
Percolating up creates a hole leading to a non-complete tree



What's wrong?

Complete Trees: Pop

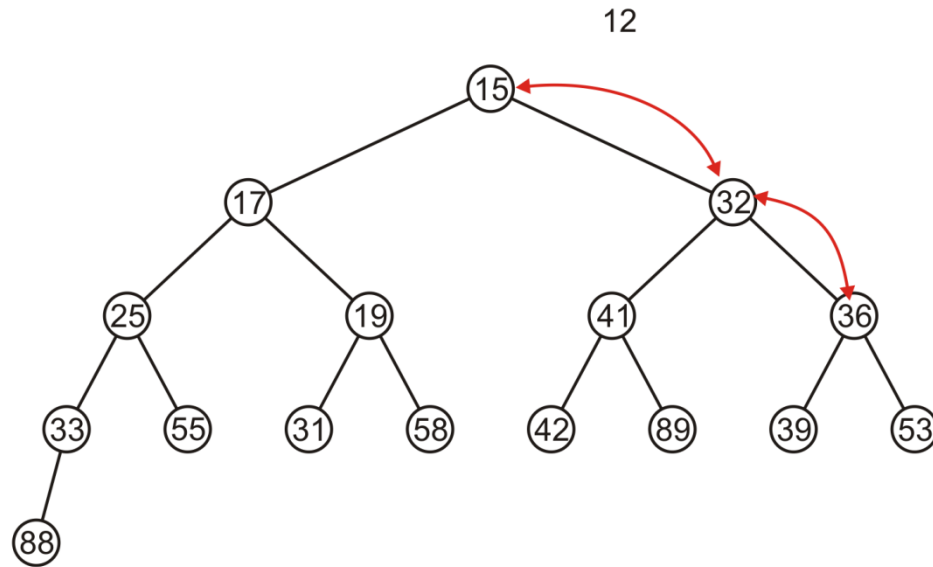
Instead, copy the last entry in the heap to the root



Complete Trees: Pop

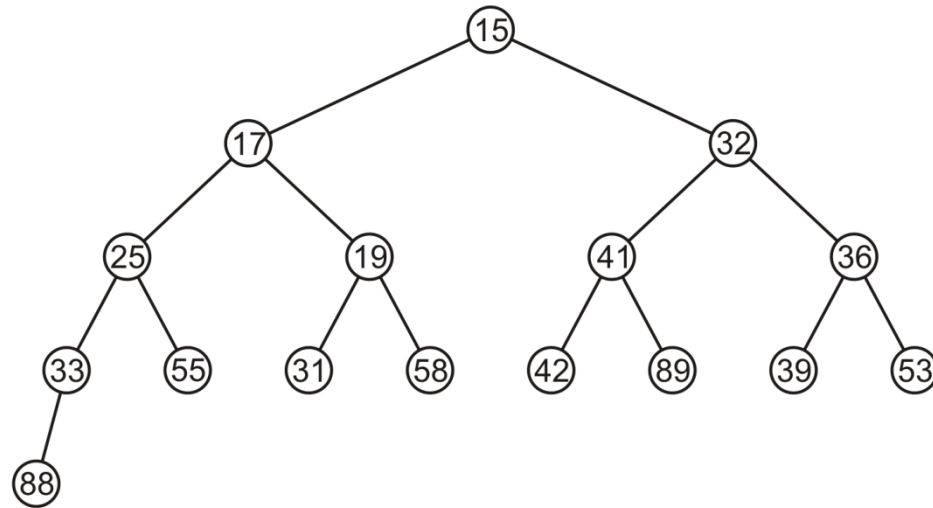
Now, percolate 36 down swapping it with the smallest of its children

- We halt when both children are larger



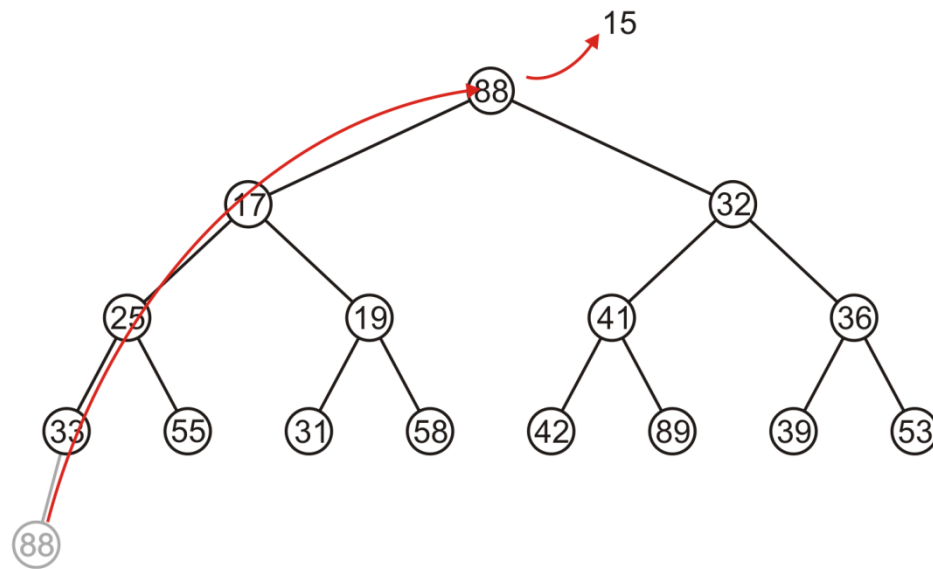
Complete Trees: Pop

The resulting tree is now still a complete tree:



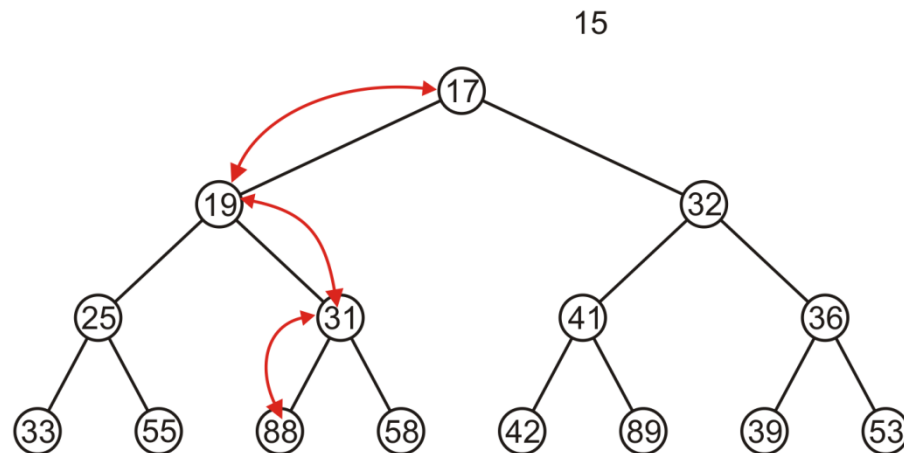
Complete Trees: Pop

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 Tree

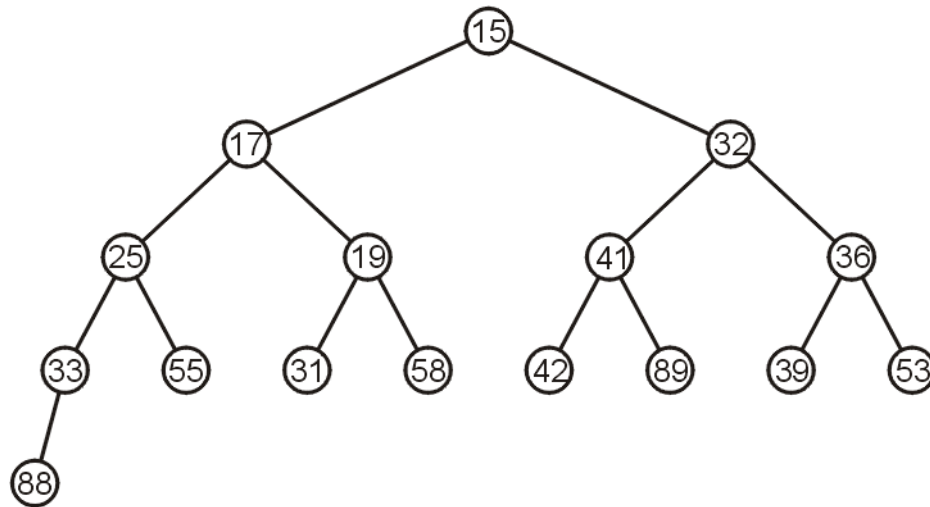
Therefore, we can maintain the complete-tree shape of a heap

We may store a complete tree using an array:

- The array is filled using breadth-first traversal on the tree

Array Implementation

For the heap

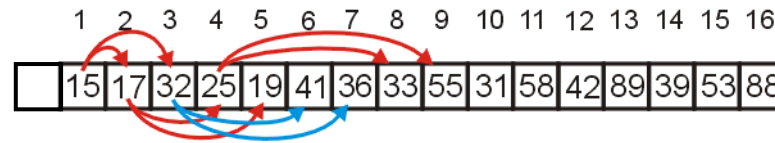


a breadth-first traversal yields:

	15	17	32	25	19	41	36	33	55	31	58	42	89	39	53	88
--	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Array Implementation

We start at index 1 when filling the array.



Given the entry at index k , it follows that:

- The parent of node is a $k/2$
 - the children are at $2k$ and $2k + 1$
- ```
parent = k >> 1;
left_child = k << 1;
right_child = left_child | 1;
```

# Array Implementation

If the heap-as-array has **N** entries, then the next empty node in the corresponding complete tree is at location  **$L = N + 1$**

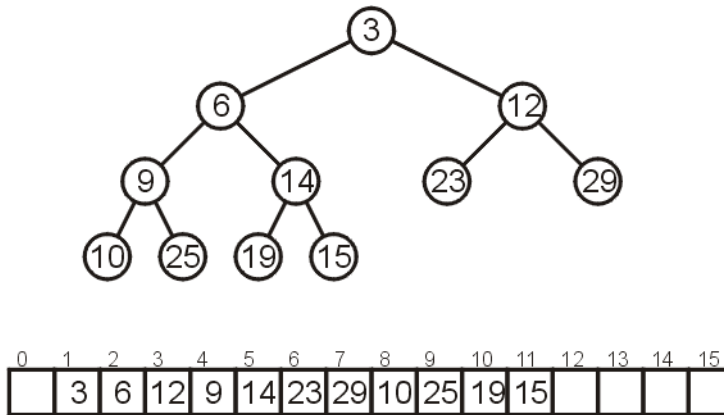
We compare the item at location **L** with the item at  **$L/2$**

If they are out of order

- Swap them
- Set  **$L/= 2$**  and repeat

# Array Implementation

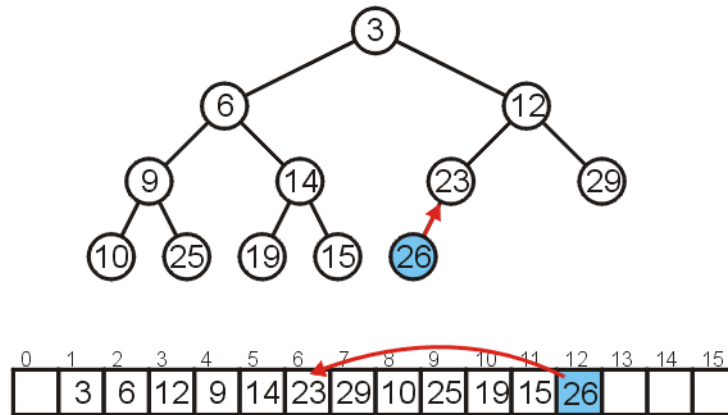
Consider the following heap, both as a tree and in its array representation





# Array Implementation: Push

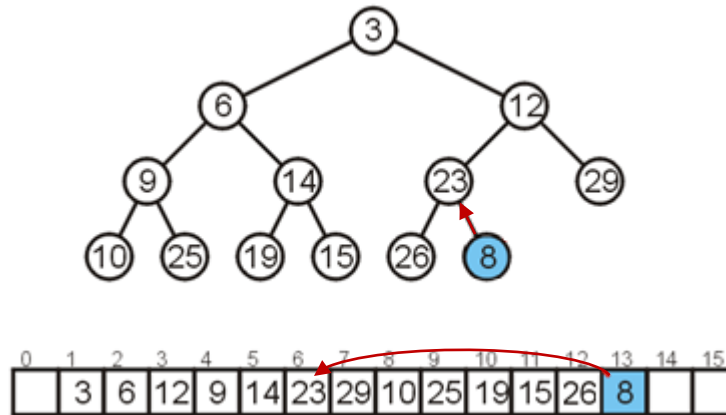
Inserting 26 requires no changes



# Array Implementation: Push

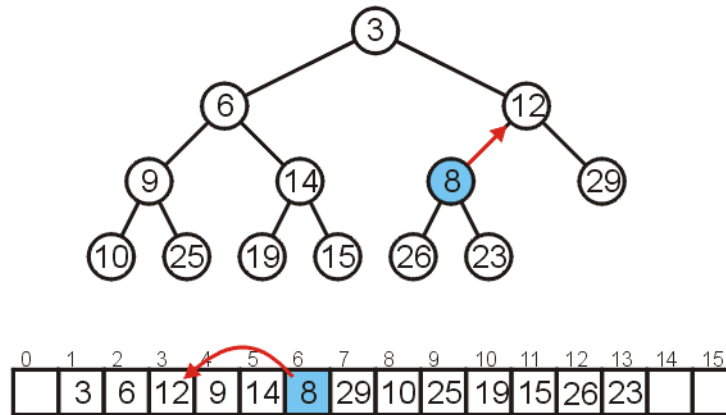
Inserting 8 requires a few percolations:

- Swap 8 and 23



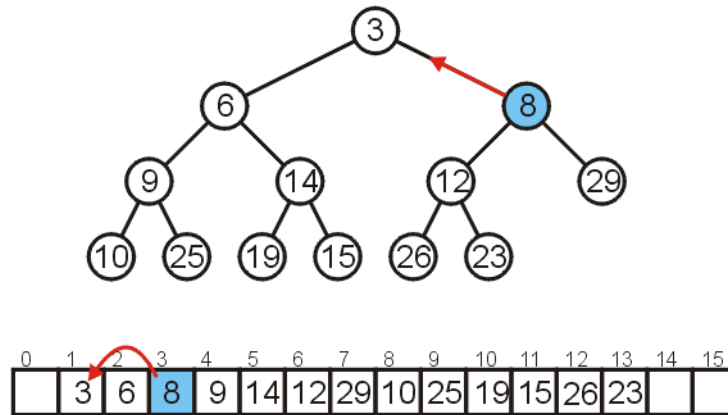
# Array Implementation: Push

Swap 8 and 12



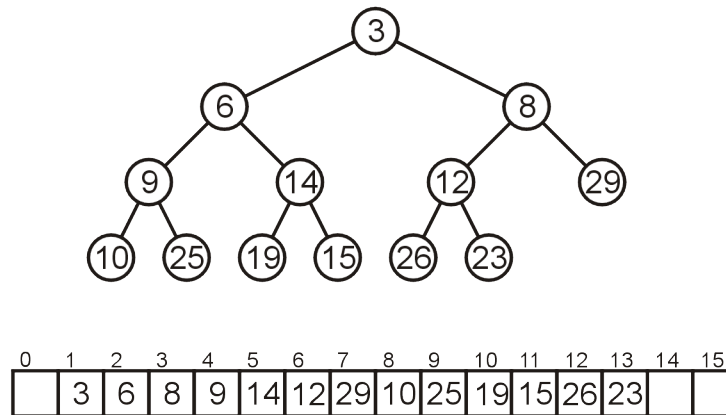
# Array Implementation: Push

At this point, it is greater than its parent, so we are finished



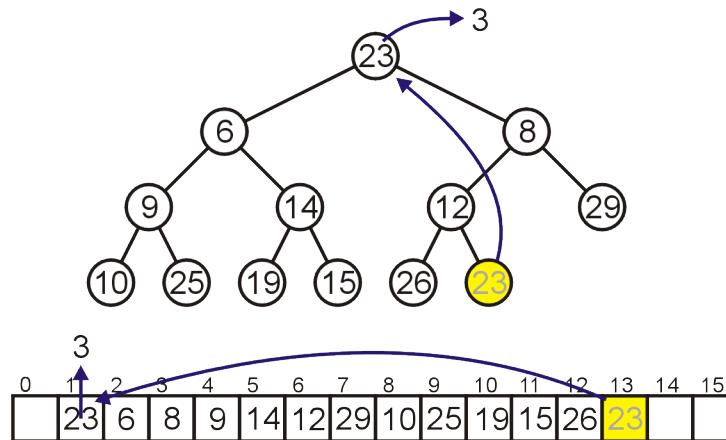
# Array Implementation: Pop

As before, popping the top has us copy the last entry to the top



# Array Implementation: Pop

As before, popping the top has us copy the last entry to the top

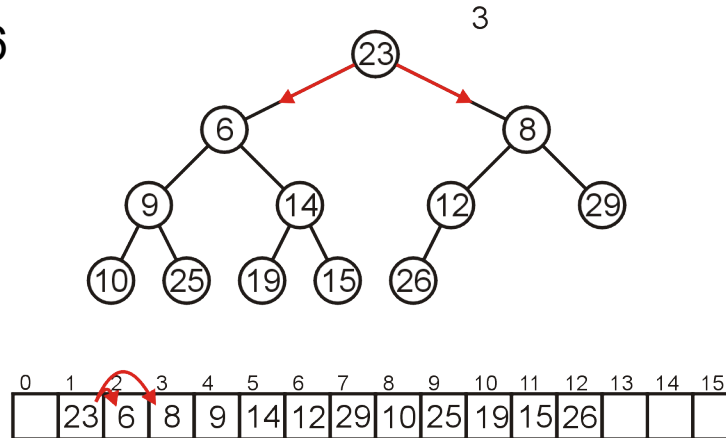


# Array Implementation: Pop

Now percolate down

Compare Node 1 with its children: Nodes 2 and 3

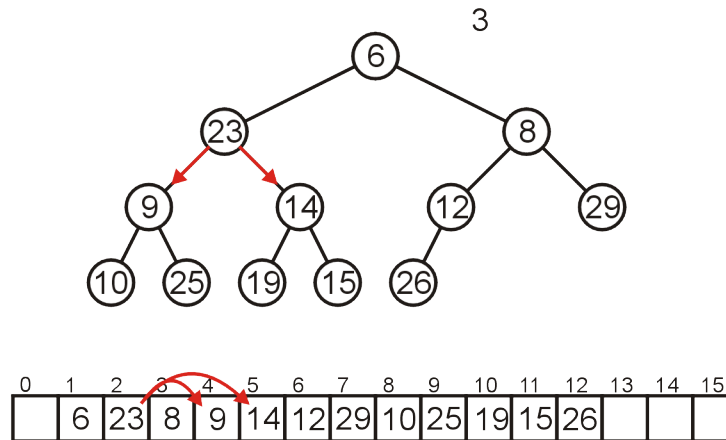
- Swap 23 and 6



# Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

- Swap 23 and 9

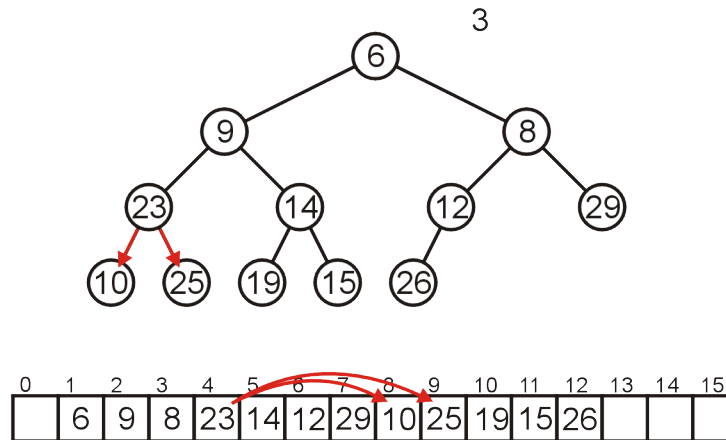




# Array Implementation: Pop

Compare Node 4 with its children: Nodes 8 and 9

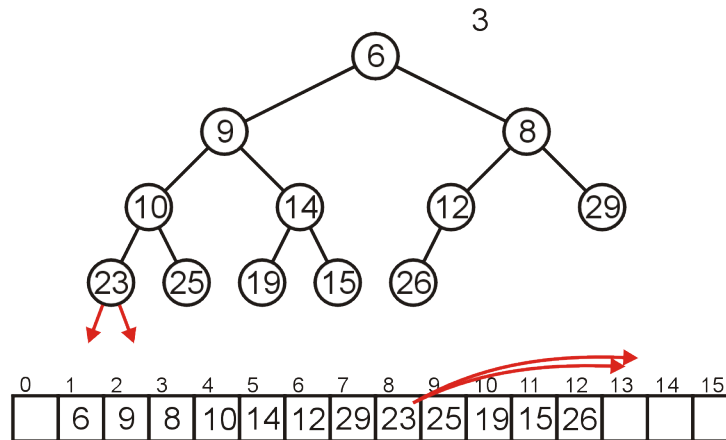
- Swap 23 and 10



# Array Implementation: Pop

The children of Node 8 are beyond the end of the array:

- Stop



# Run-time Analysis

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

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

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

Pushing an object is also  $O(\ln(n))$

- If we insert an object less than the root, it will be moved up to the top

Space complexity  $O(n)$

*So binary heap is a better implementation of priority queue*

# Run-time Analysis

If we are inserting an object less than the root (at the front), then the run time will be  $\Theta(\ln(n))$

If we insert at the back (greater than any object) then the run time will be  $\Theta(1)$

How about an arbitrary insertion?

- It will be  $O(\ln(n))$ ? Could the average be less?

# Run-time Analysis

With each percolation, it will move an object past half of the remaining entries in the tree

- Therefore, after one percolation, it will probably be past half of the entries, and therefore *on average* will require no more percolations

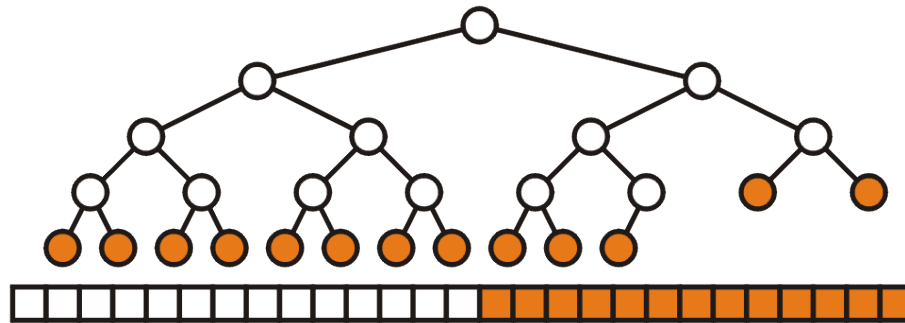
$$\begin{aligned}\frac{1}{n} \sum_{k=0}^h (h-k)2^k &= \frac{2^{h+1} - h - 2}{n} \\ &= \frac{n - h - 1}{n} = \Theta(1)\end{aligned}$$

Therefore, we have an average run time of  $\Theta(1)$

# Run-time Analysis

An arbitrary removal requires that all entries in the heap be checked:  
 $O(n)$

A removal of the largest object in the heap still requires all leaf nodes to be checked – there are approximately  $n/2$  leaf nodes:  $O(n)$



# Run-time Analysis

Thus, our grid of run times is given by:

|        | front         | arbitrary | back   |
|--------|---------------|-----------|--------|
| insert | $O(\ln(n))^*$ | $O(1)$    | $O(1)$ |
| access | $O(1)$        | $O(n)$    | $O(n)$ |
| delete | $O(\ln(n))$   | $O(n)$    | $O(n)$ |

# Run-time Analysis

Some observations:

- Continuously inserting at the front of the heap (*i.e.*, the new object being pushed is less than everything in the heap) causes the run-time to drop to  $O(\ln(n))$
- If the objects are coming in order of priority, use a regular queue with swapping
- Merging two binary heaps of size  $n$  is a  $\Theta(n)$  operation



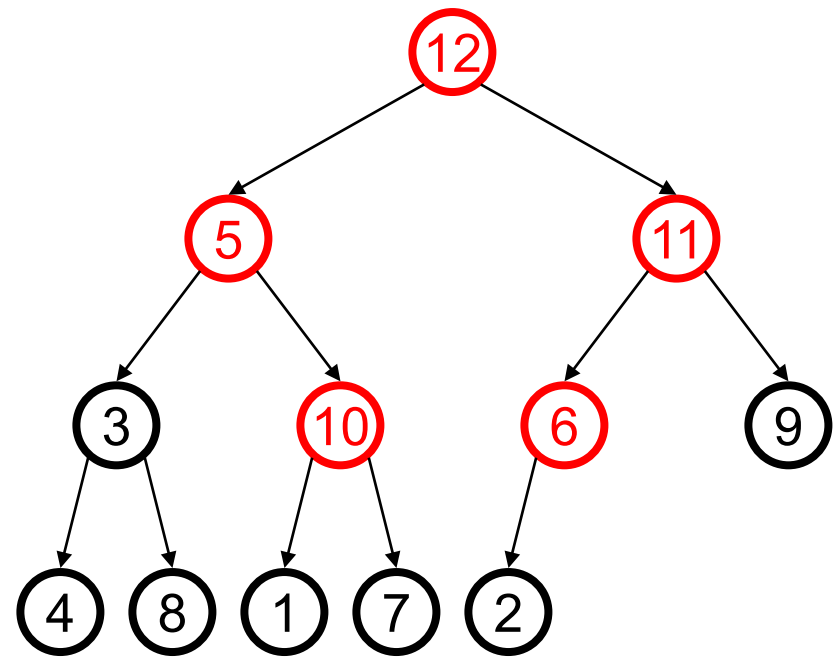
# Build Heap

- Task: Given a set of  $n$  keys, build a heap all at once
- Approach 1
  - Repeatedly perform **push**
- Complexity
  - $O(n \ln(n))$

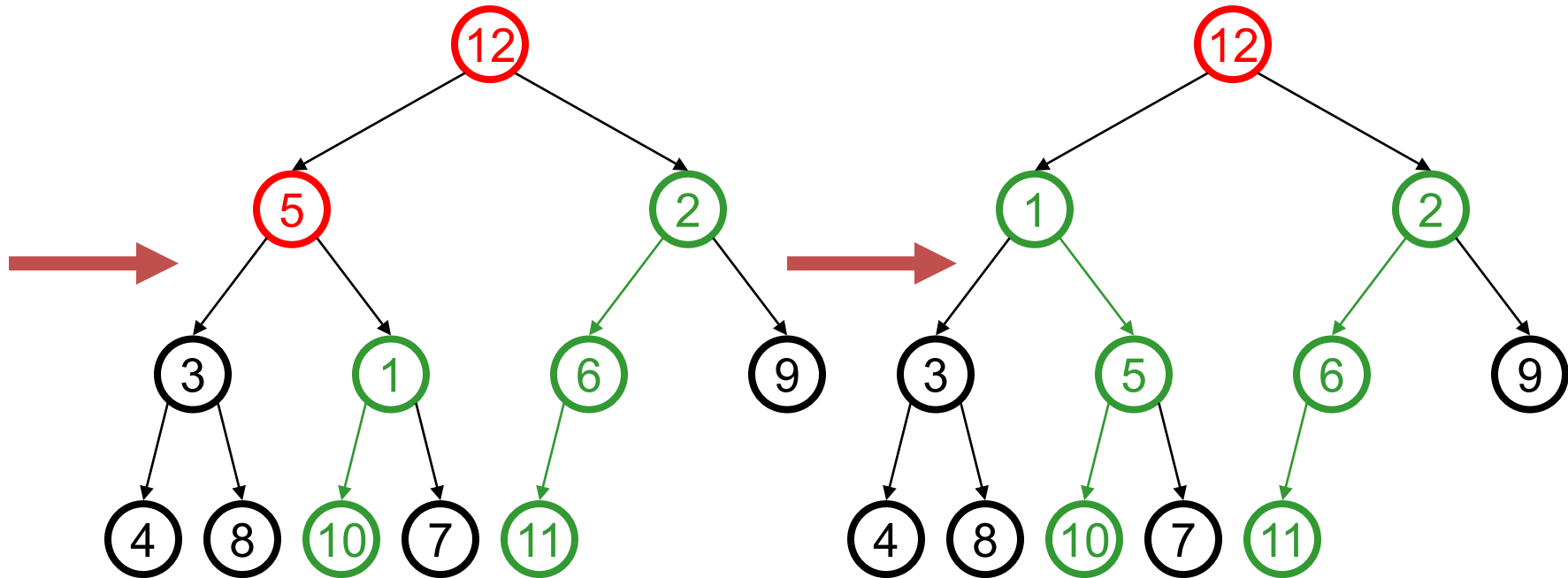
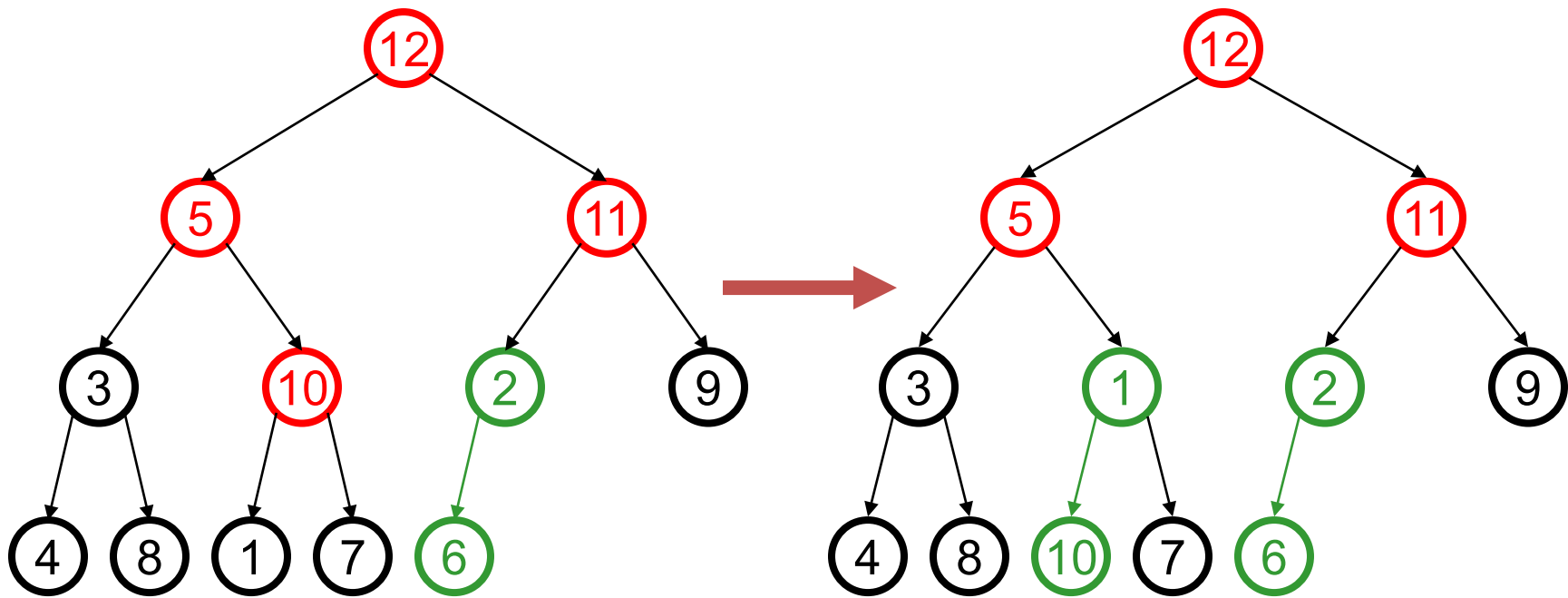
# Floyd's Method

Put the keys in a binary tree and fix the heap property!

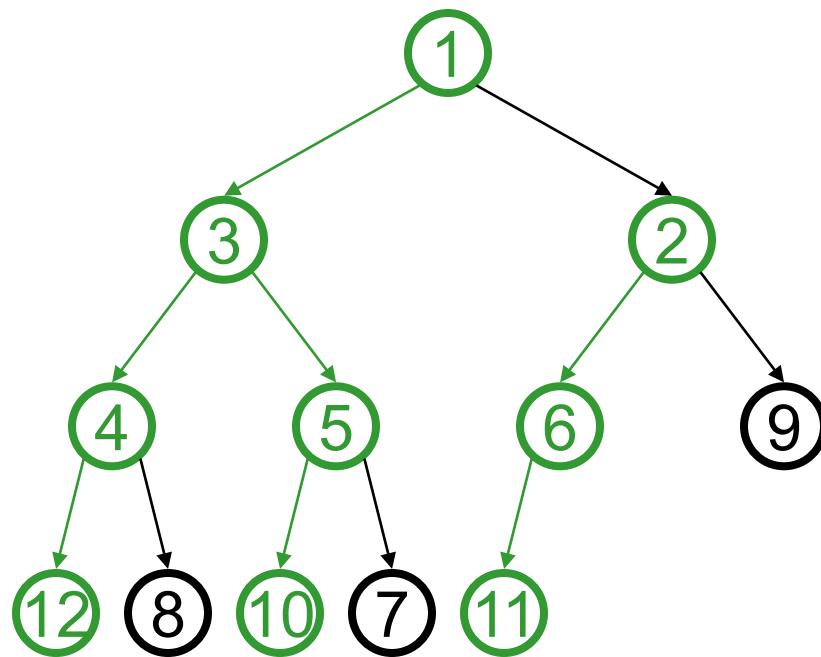
```
buildHeap() {
 for (i=size/2; i>0; i--)
 percolateDown(i);
}
```



|    |   |    |   |    |   |   |   |   |   |   |   |
|----|---|----|---|----|---|---|---|---|---|---|---|
| 12 | 5 | 11 | 3 | 10 | 6 | 9 | 4 | 8 | 1 | 7 | 2 |
|----|---|----|---|----|---|---|---|---|---|---|---|



Finally...



# Complexity of Build Heap

- No percolation for the leaf nodes ( $n/2$  nodes)
- At most  $n/4$  nodes percolate down 1 level  
at most  $n/8$  nodes percolate down 2 levels  
at most  $n/16$  nodes percolate down 3 levels  
...

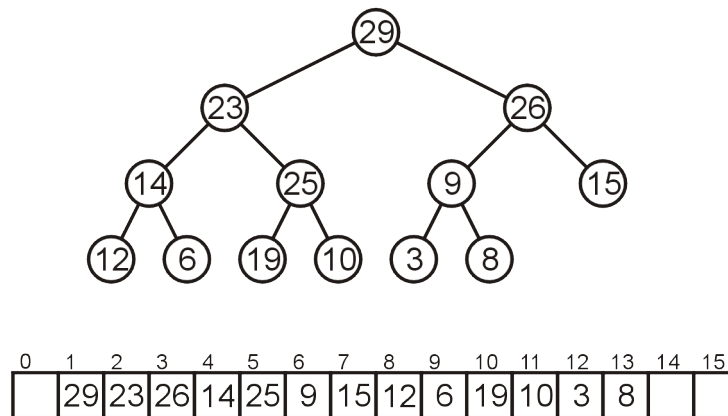
$$1\frac{n}{4} + 2\frac{n}{8} + 3\frac{n}{16} + \dots = \sum_{i=1}^{\log n} i \frac{n}{2^{i+2}} = \frac{n}{4} \sum_{i=1}^{\log n} \frac{i}{2^i} = \frac{n}{4} 2 = \frac{n}{2}$$

$$\Theta(n)$$

# 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



# Outline

- Priority queue
- Binary heap
- Heapsort

# Heapsort

- Sorting
  - take a list of objects  $(a_0, a_1, \dots, a_{n-1})$
  - return a reordering  $(a'_0, a'_1, \dots, a'_{n-1})$  such that  $a'_0 \leq a'_1 \leq \dots \leq a'_{n-1}$
- Heapsort
  - Place the objects into a heap
    - $O(n)$  time
  - Repeatedly popping the top object until the heap is empty
    - $O(n \ln(n))$  time
  - Time complexity:  $O(n \ln(n))$



# In-place Implementation

Problem:

- This solution requires additional memory: a min-heap of size  $n$
- This requires  $\Theta(n)$  memory

If the unsorted objects are stored in an array, is it possible to perform a heap sort **in place**, that is, require at most  $\Theta(1)$  memory (a few extra variables)?

# In-place Implementation

Instead of implementing a min-heap, consider a max-heap:

- The maximum element is at the top of the heap

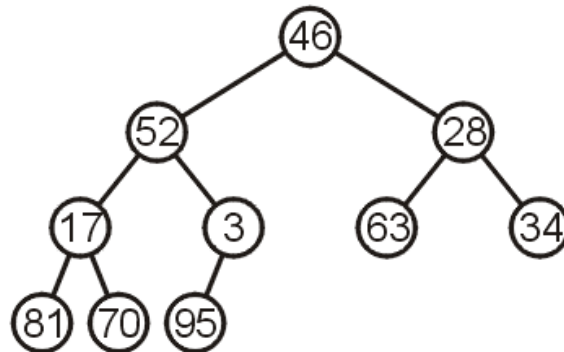
We then repeatedly pop the top object and move it to the end of the array.

# In-place Implementation

Now, consider this unsorted array:

|    |    |    |    |   |    |    |    |    |    |
|----|----|----|----|---|----|----|----|----|----|
| 46 | 52 | 28 | 17 | 3 | 63 | 34 | 81 | 70 | 95 |
|----|----|----|----|---|----|----|----|----|----|

This array represents the following complete tree:

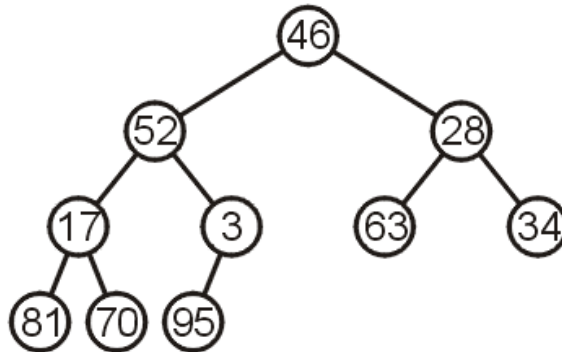


# In-place Implementation

Now, consider this unsorted array:

|    |    |    |    |   |    |    |    |    |    |
|----|----|----|----|---|----|----|----|----|----|
| 46 | 52 | 28 | 17 | 3 | 63 | 34 | 81 | 70 | 95 |
|----|----|----|----|---|----|----|----|----|----|

Because we start at 0 (instead of 1 as in array storage of complete trees), we need different formulas for finding the children and parent



Children

$$2*k + 1 \quad 2*k + 2$$

Parent

$$(k + 1)/2 - 1$$

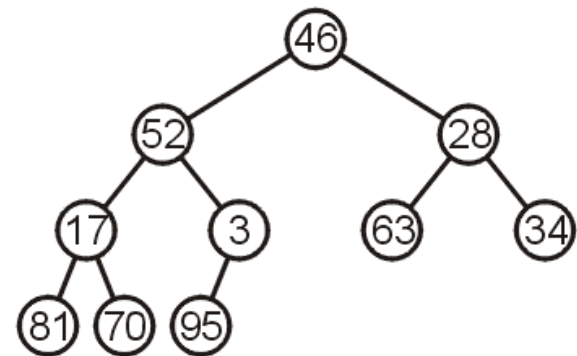
# Example Heap Sort

First, we must convert the unordered array with  $n = 10$  elements into a max-heap

|    |    |    |    |   |    |    |    |    |    |
|----|----|----|----|---|----|----|----|----|----|
| 46 | 52 | 28 | 17 | 3 | 63 | 34 | 81 | 70 | 95 |
|----|----|----|----|---|----|----|----|----|----|

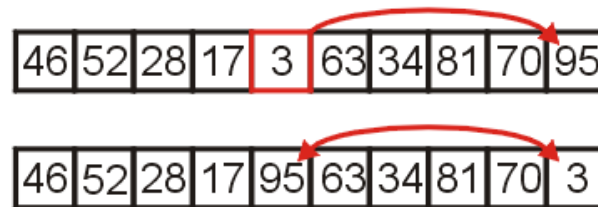
None of the leaf nodes need to be percolated down, and the last non-leaf node is in position  $n/2-1$

Thus, we start with position  $10/2-1 = 4$



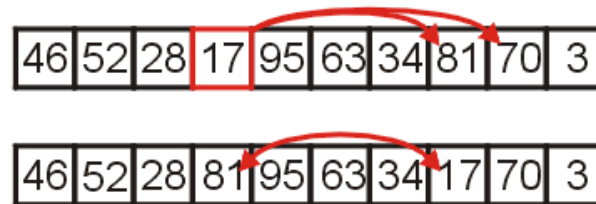
# Example Heap Sort

We compare 3 with its child and swap them



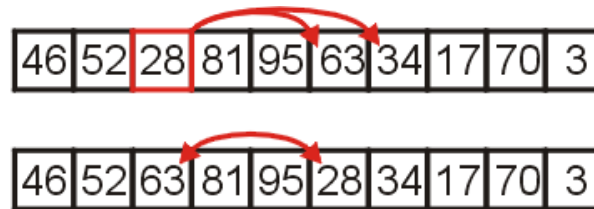
# Example Heap Sort

We compare 17 with its two children and swap it with the maximum child (81)



# Example Heap Sort

We compare 28 with its two children, 63 and 34, and swap it with the largest child





# Example Heap Sort

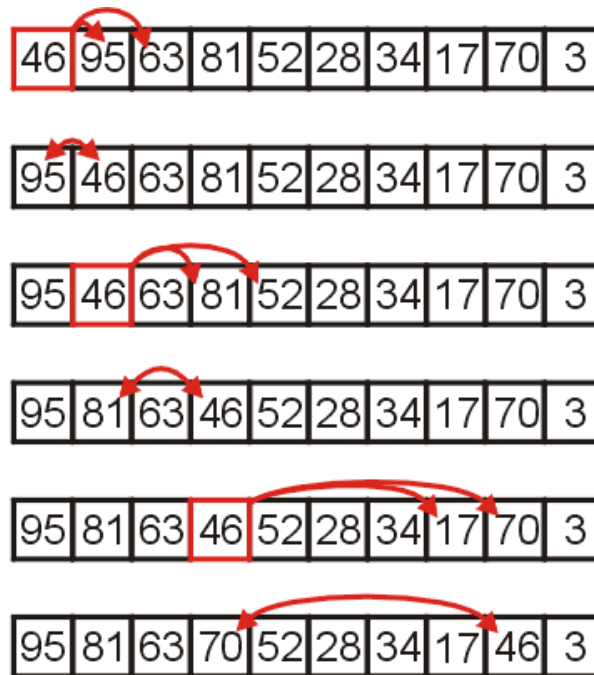
We compare 52 with its children, swap it with the largest

- Recursing, no further swaps are needed



# Example Heap Sort

Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70



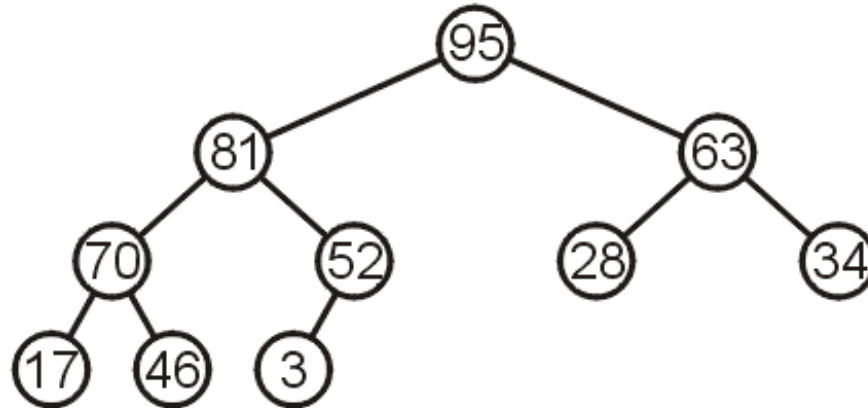
# Heap Sort Example

We have now converted the unsorted array

|    |    |    |    |   |    |    |    |    |    |
|----|----|----|----|---|----|----|----|----|----|
| 46 | 52 | 28 | 17 | 3 | 63 | 34 | 81 | 70 | 95 |
|----|----|----|----|---|----|----|----|----|----|

into a max-heap:

|    |    |    |    |    |    |    |    |    |   |
|----|----|----|----|----|----|----|----|----|---|
| 95 | 81 | 63 | 70 | 52 | 28 | 34 | 17 | 46 | 3 |
|----|----|----|----|----|----|----|----|----|---|

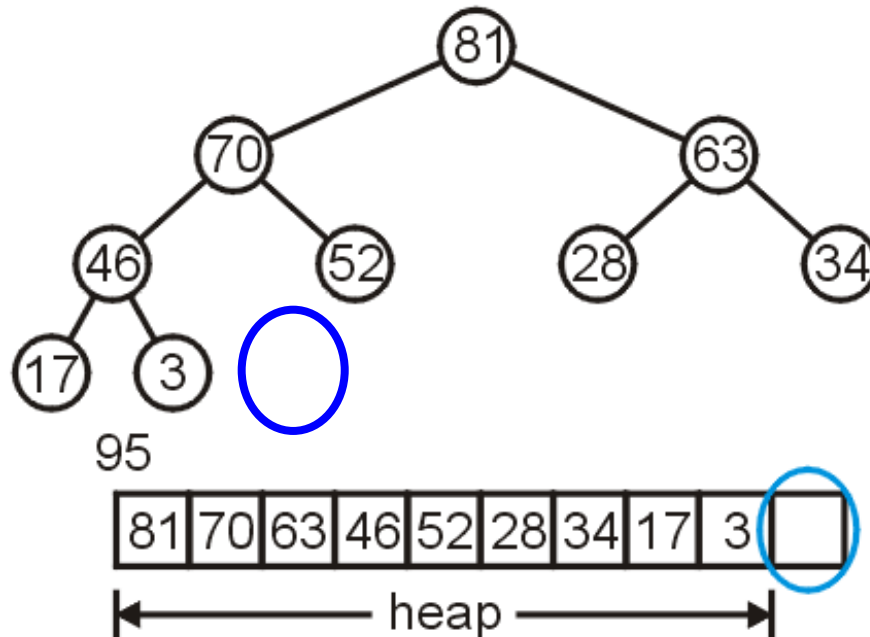


# Heap Sort Example

We pop the maximum element of this heap

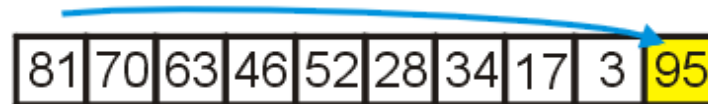


This leaves a gap at the back of the array:

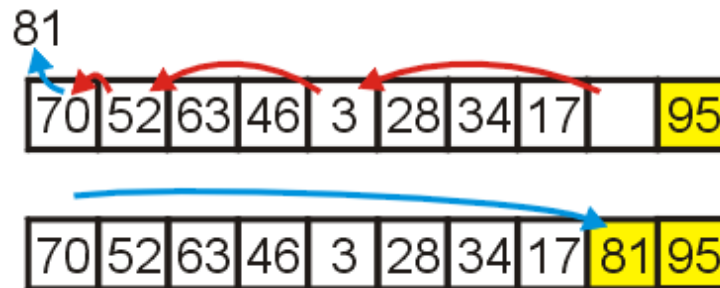


# Heap Sort Example

This is the last entry in the array, so why not fill it with the largest element?



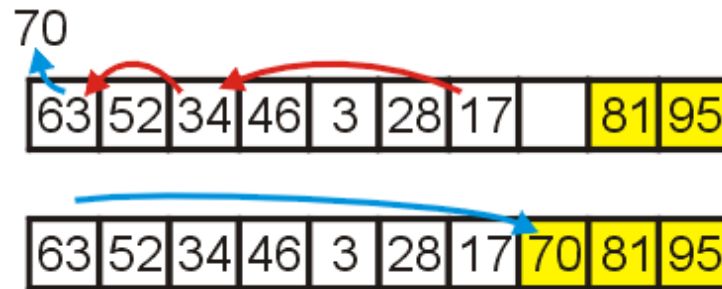
Repeat this process: pop the maximum element, and then insert it at the end of the array:



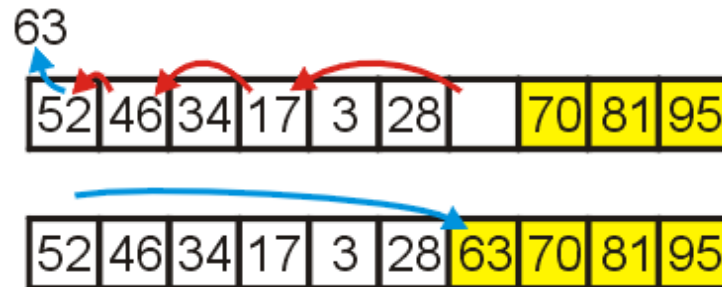
# Heap Sort Example

Repeat this process

- Pop and append 70

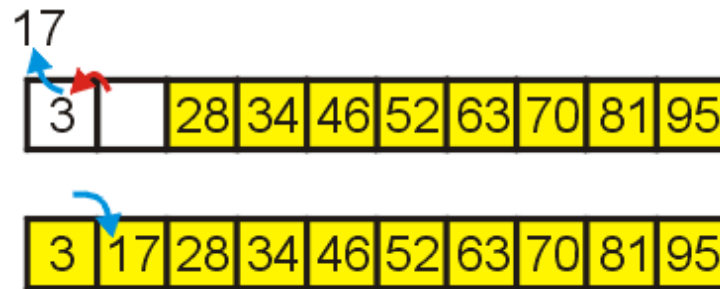


- Pop and append 63



# Heap Sort Example

Finally, we can pop 17, insert it into the 2<sup>nd</sup> location, and the resulting array is sorted



# Summary

- Priority queue
  - pop the object with the highest priority
- Binary heap
  - Operations
    - Top  $\Theta(1)$
    - Push  $O(\ln(n))$
    - Pop  $O(\ln(n))$
    - Build  $\Theta(n)$
  - Implementation using arrays
- Heapsort
  - Time:  $O(n \ln(n))$
  - Space:  $O(1)$