Priority Queue Data Structures

261217 Data Structures for Computer Engineers

Patiwet Wuttisarnwattana, Ph.D.

patiwet@eng.cmu.ac.th

Computer Engineering, Chiang Mai University

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

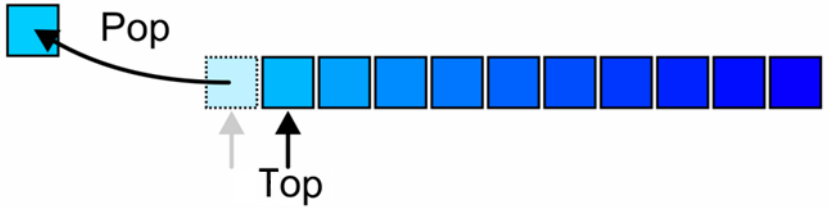
With each pushed object, we will associate a nonnegative integer (0, 1, 2, ...) where:

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

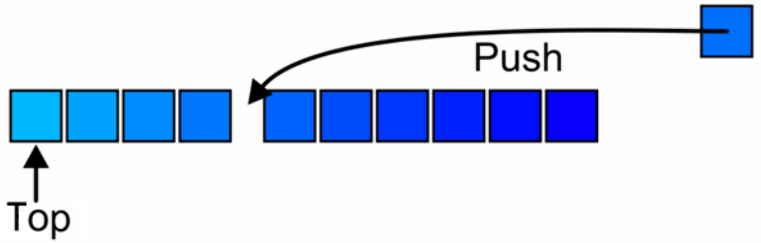
Operations

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



Lexicographic Priority

Priority may also depend on multiple variables:

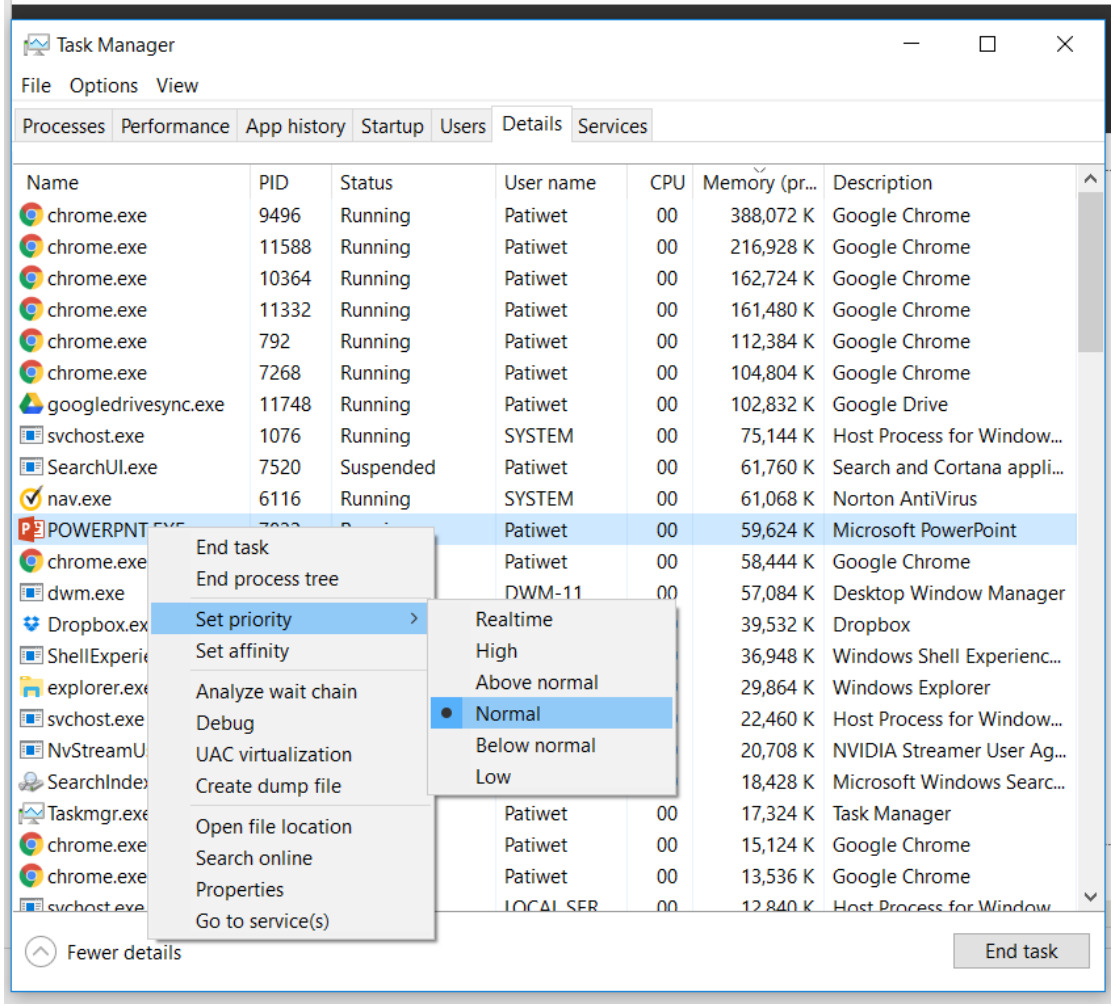
- Two values specify a priority: (a, b)
- A pair (a, b) has higher priority than (c, d) if:
  - $a < c$, or
  - $a = c$ and $b < d$

For example,

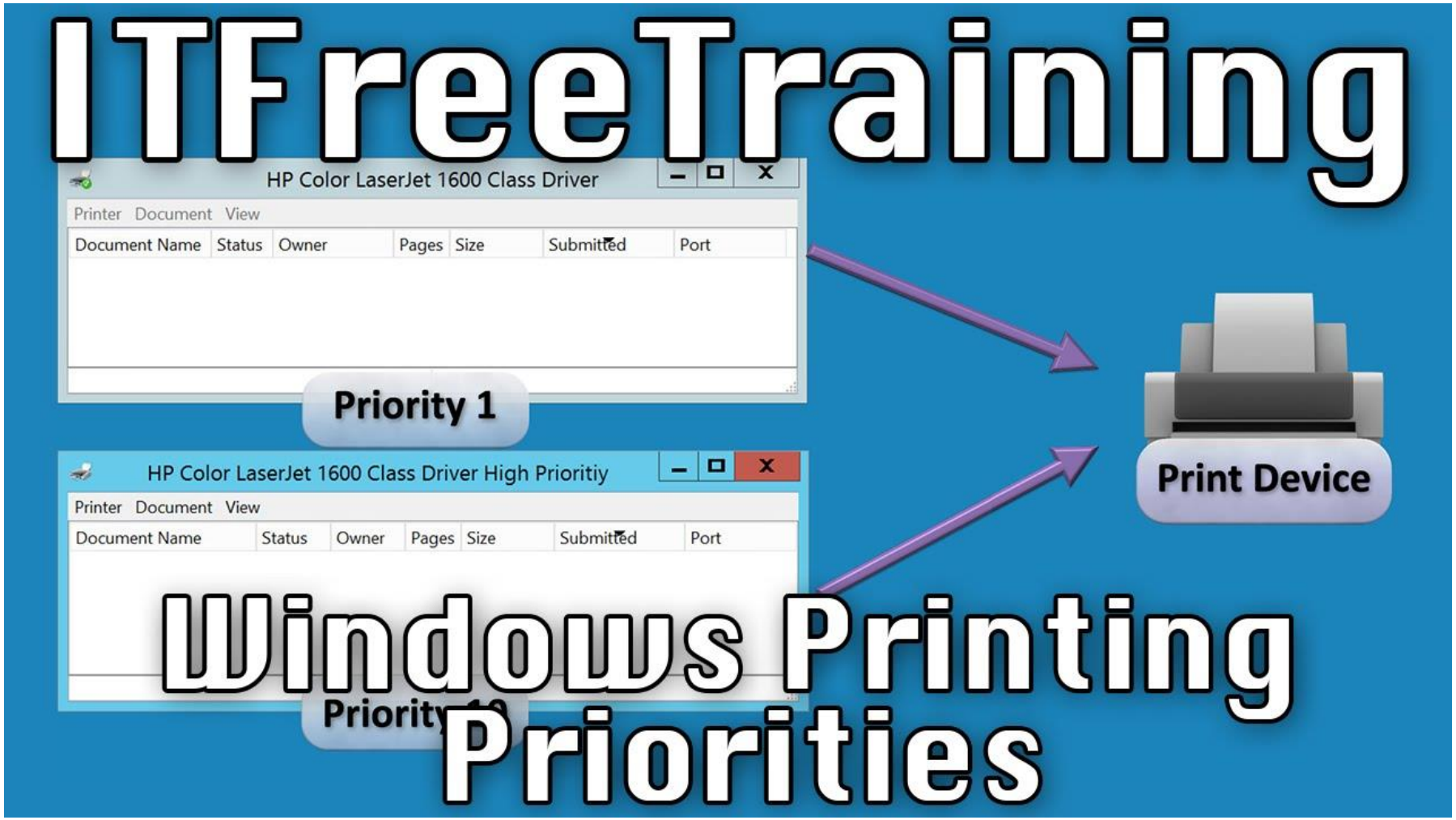
- (5, 19), (13, 1), (13, 24), and (15, 0) all have *higher* priority than (15, 7)
- Slave Game
  - (9, ♣) vs (10, ♠) who wins?
  - (10, ♣) vs (10, ♠) who wins?

Process Priority in Windows

The priority of processes in Windows may be set in the *Windows Task Manager*



Printing Priority



Stock Market

PTT		351.00	+4.00(+1.15%)
Volume	Bid	Offer	Volume
229,800	350.00	351.00	17,700
129,300	349.00	352.00	733,200
75,800	348.00	353.00	36,700
113,800	347.00	354.00	190,000
80,400	346.00	355.00	104,700

All buyers want to buy things at the lower price; but the buyers who buy at the highest price win

Sellers want to sell things at the higher price; but the sellers who sell at the cheapest price win

Implementations

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

We will look at two implementations using data structures we already know:

- Multiple queues—one for each priority
- An AVL tree

The next topic will be a more appropriate data structure: the heap

Multiple Queues

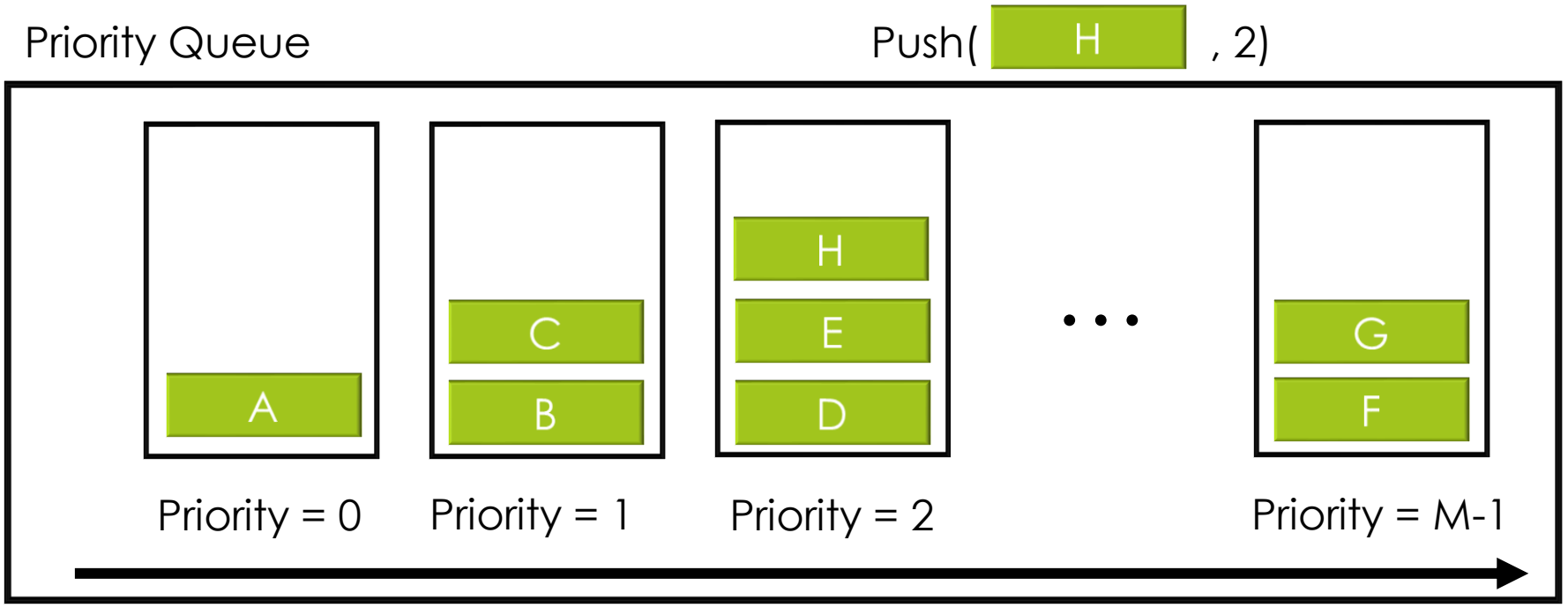
Assume there is a fixed number of priorities, say M

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

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 empty queue with highest priority



Multiple Queues

The run times are reasonable:

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

Unfortunately:

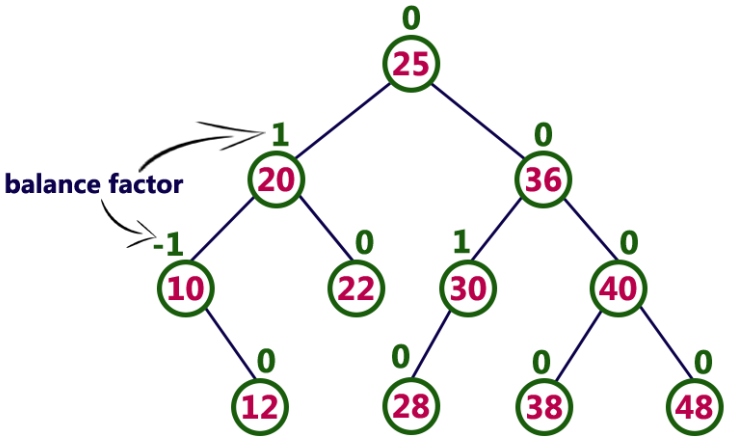
- It restricts the range of priorities
- The memory requirement is $\Theta(M + n)$

AVL Trees

We could simply insert the objects into an AVL tree where the order is given by the stated priority:

- Insertion is $\Theta(\ln(n))$ `void insert( Node);`
- Top is $\Theta(\ln(n))$ `Node findMin();`
- Remove is $\Theta(\ln(n))$ `void delete( findMin() );`

There is significant overhead for maintaining both the tree and the corresponding balance



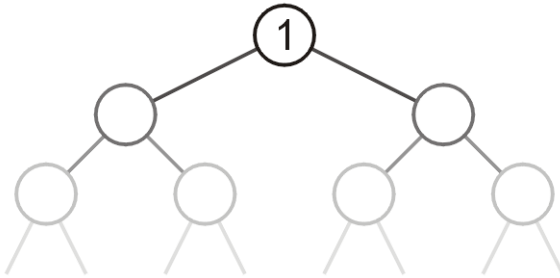
Heaps

Can we do better?

- That is, can we reduce some (or all) of the operations down to $\Theta(1)$?

The next topic defines a *heap*

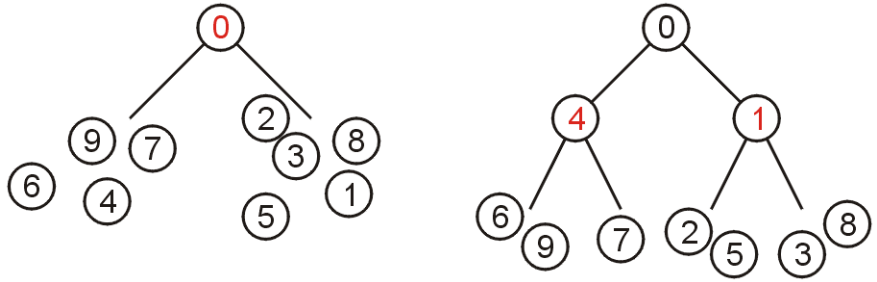
- A tree with the top object at the root
- We will look at binary heaps
- Numerous other heaps exists:
  - d-ary heaps
  - Leftist heaps
  - Skew heaps
  - Binomial heaps
  - Fibonacci heaps
  - Bi-parental heaps



Min-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

Min-Heap

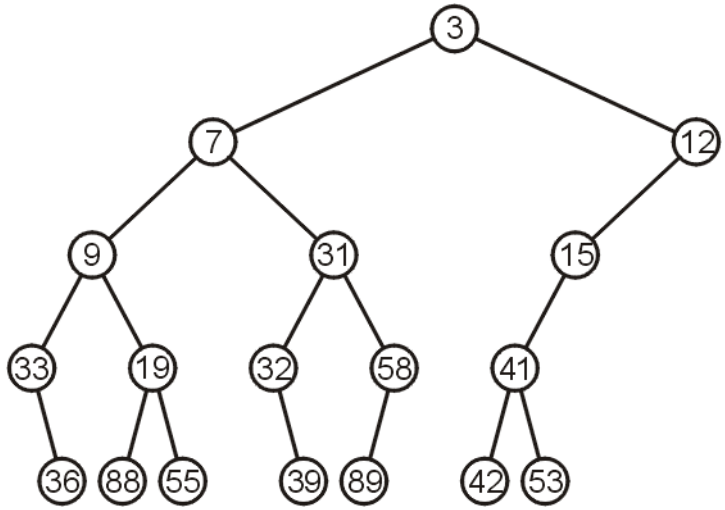
Important:

THERE IS NO OTHER RELATIONSHIP BETWEEN THE ELEMENTS IN THE TWO SUBTREES

Failing to understand this is the greatest mistake a student makes

Example

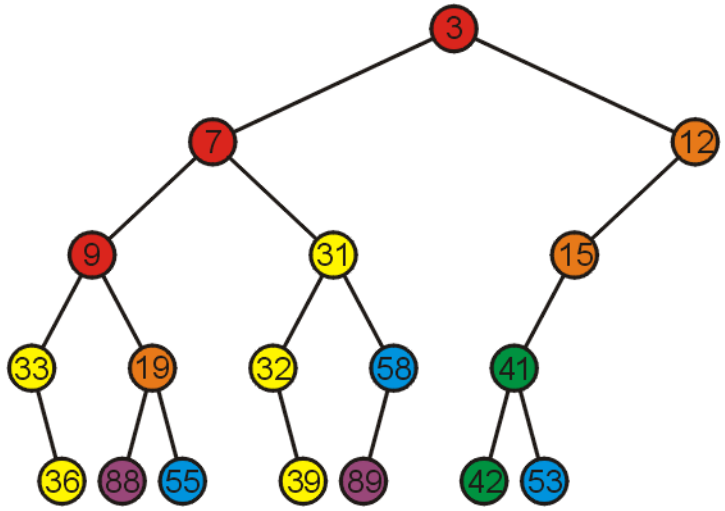
This is a binary min-heap:



Example

Adding colour, we observe

- The left subtree has the smallest (7) and the largest (89) objects
- No relationship between items with similar priority



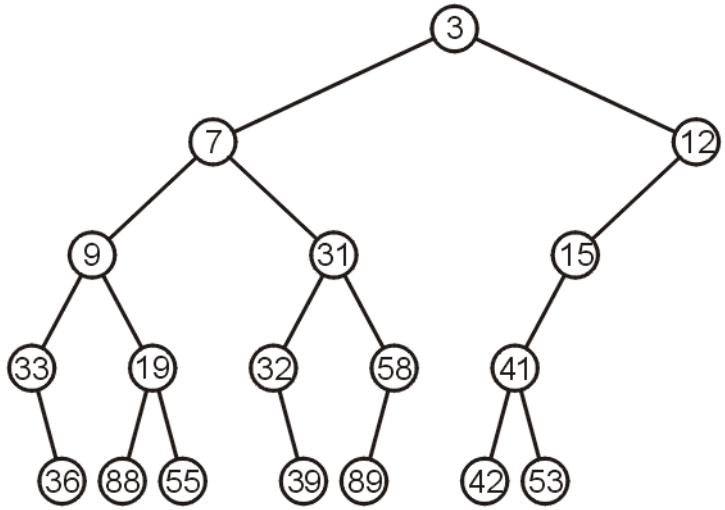
Priority Queue Operations

We will consider three operations:

- Top
- Pop (Dequeue)
- Push (Enqueue)

Example

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



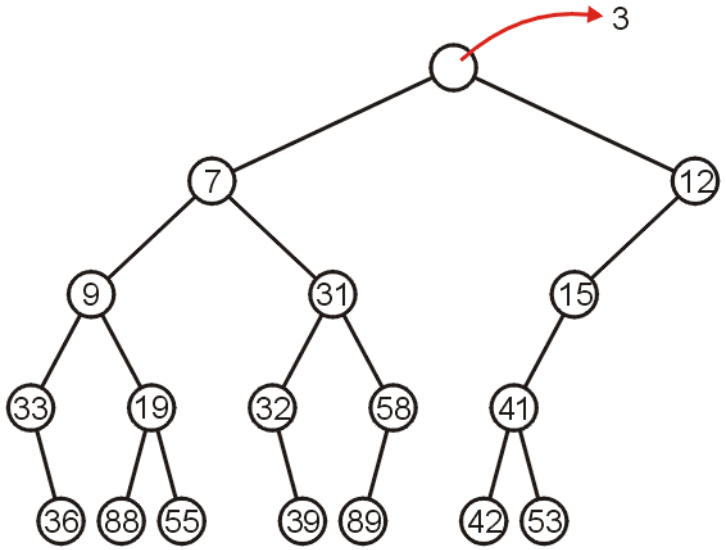
Pop

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

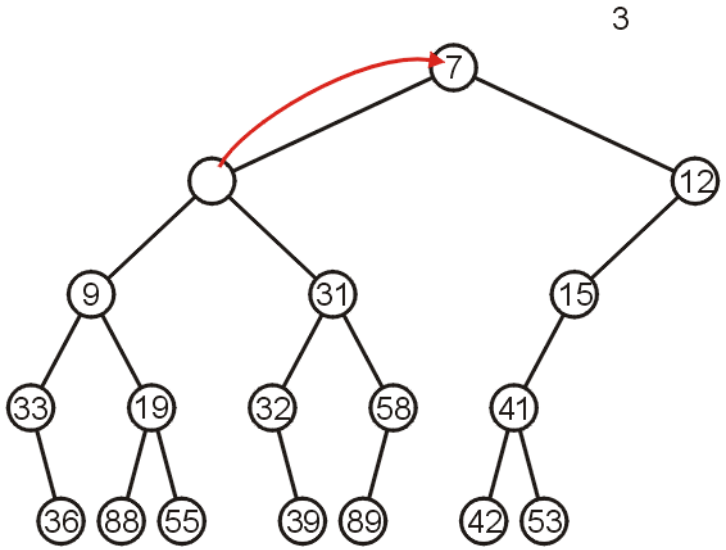
Pop

Using our example, we remove 3:



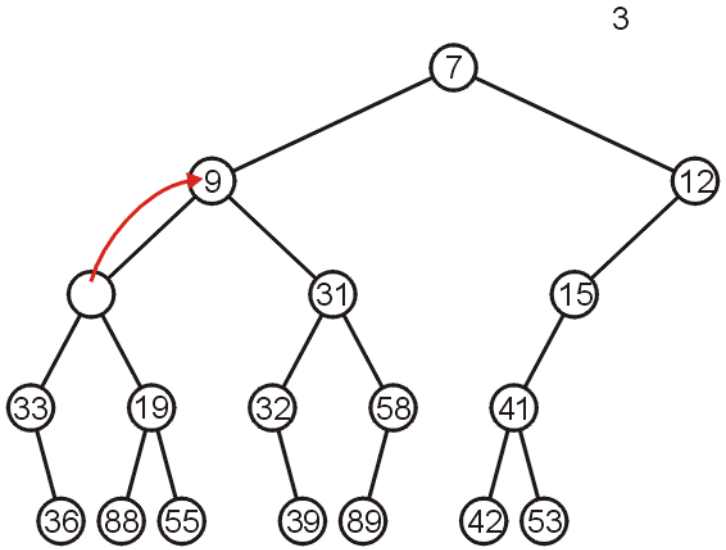
Pop

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



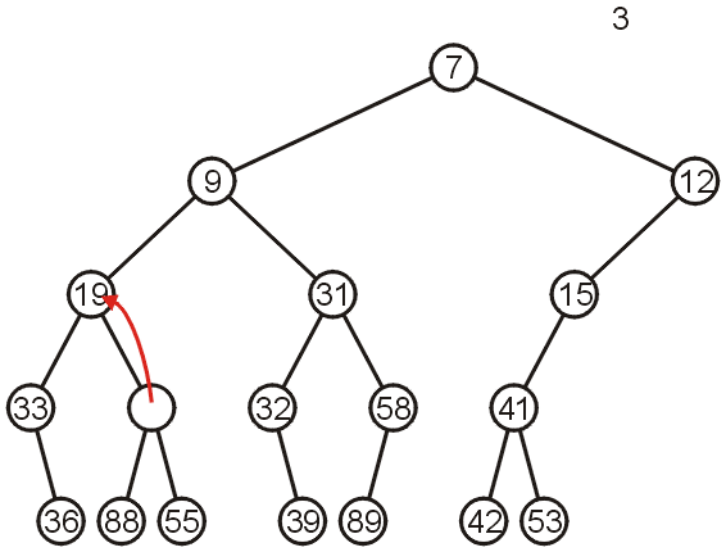
Pop

In the left sub-tree, we promote 9:



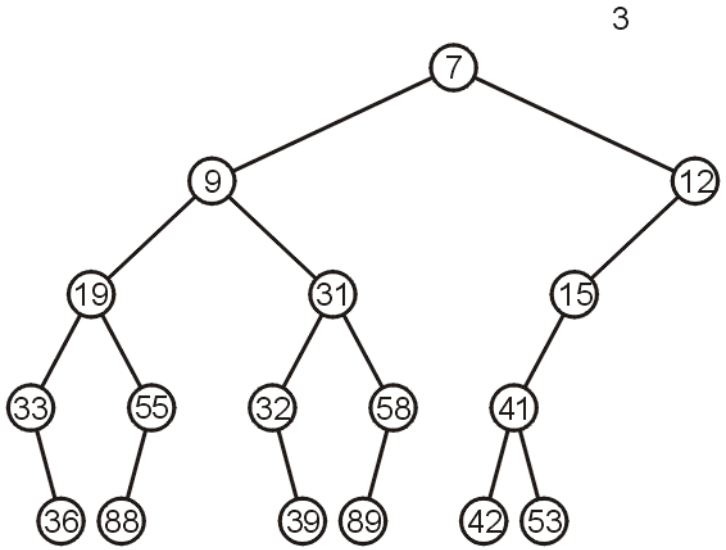
Pop

Recursively, we promote 19:



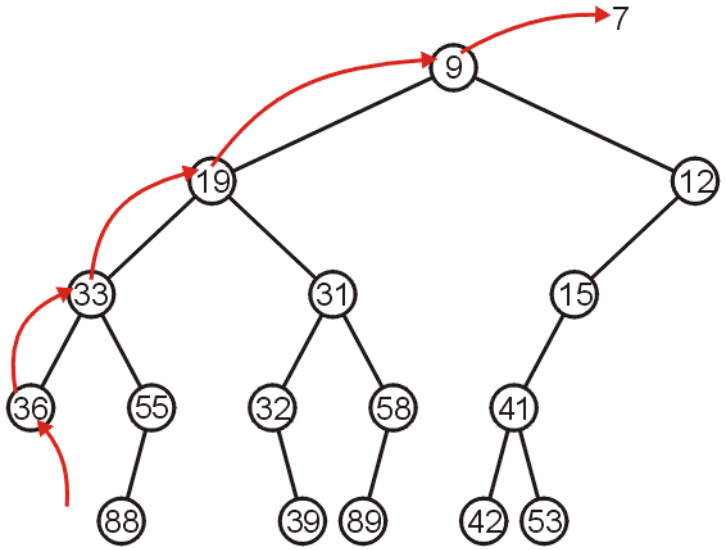
Pop

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



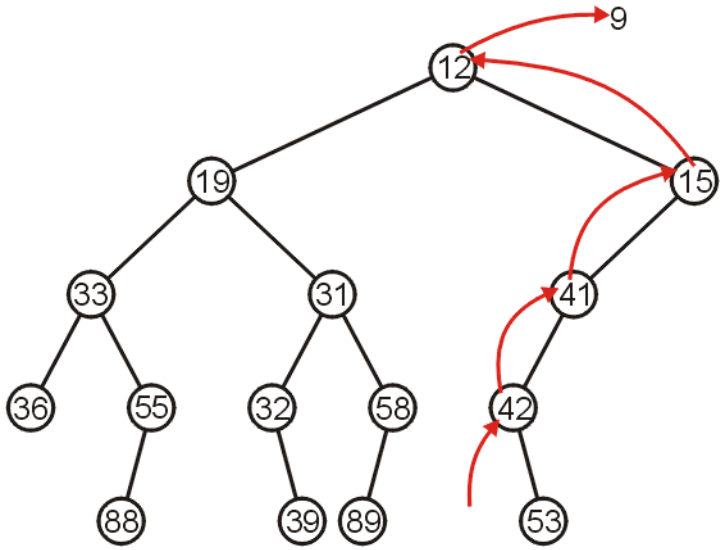
Pop

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)

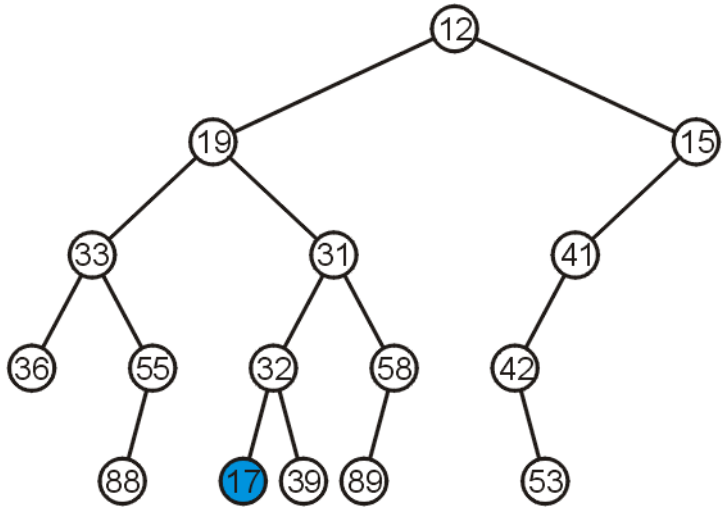
We will use the **first** approach with binary heaps

- Other heaps use the second

Push

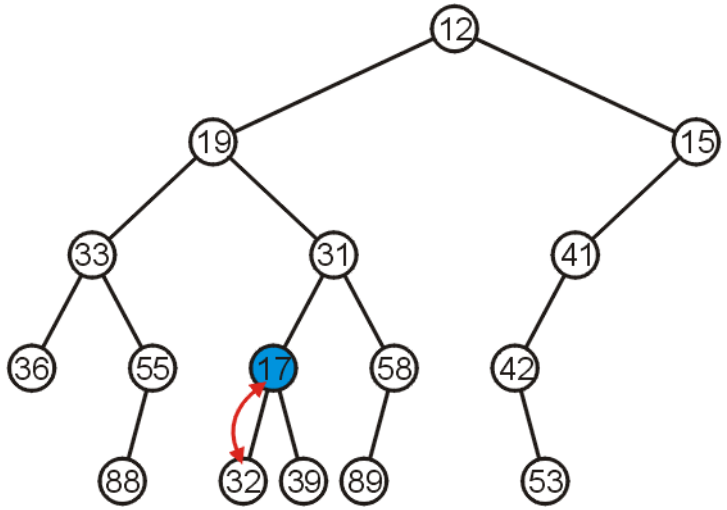
Inserting 17 into the last heap

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



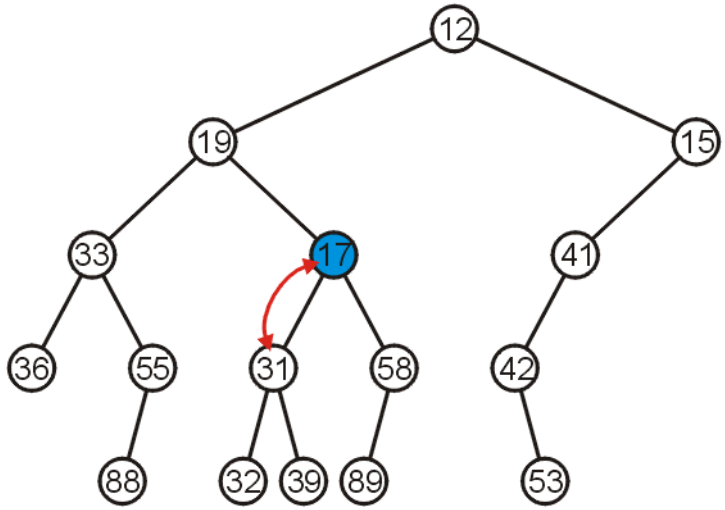
Push

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



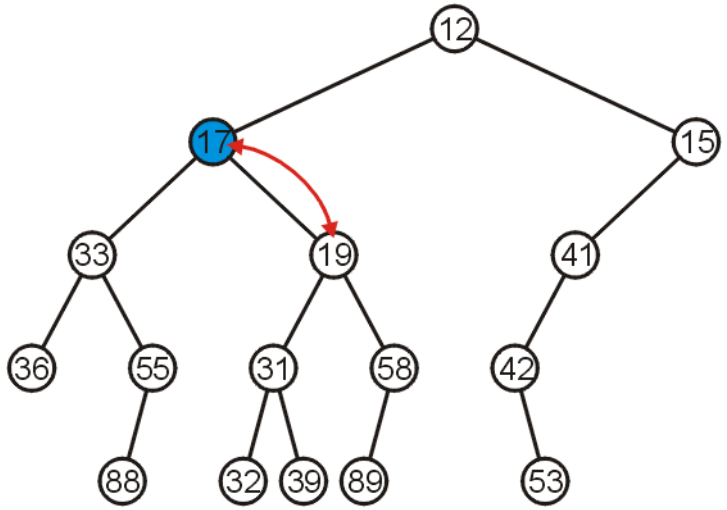
Push

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



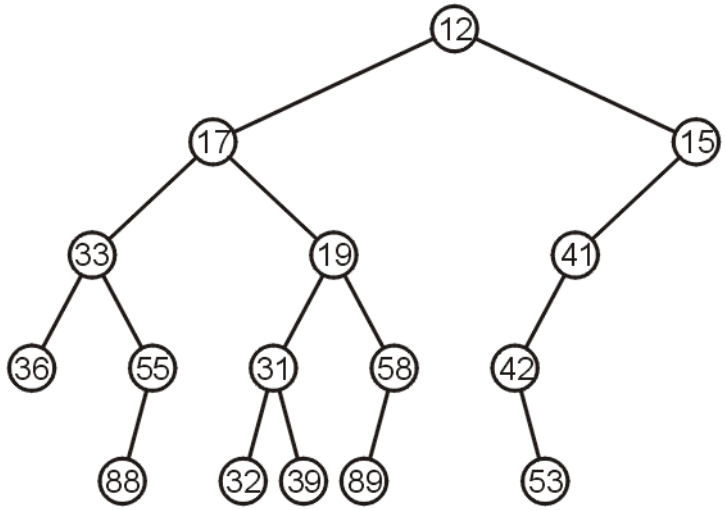
Push

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: The new node always goes up but not goes down

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

Implementation

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

From this, we looked at:

- AVL Trees
- B-Trees
- Red-black Trees (not course material)

How can we determine where to insert so as to keep balance?

Implementations

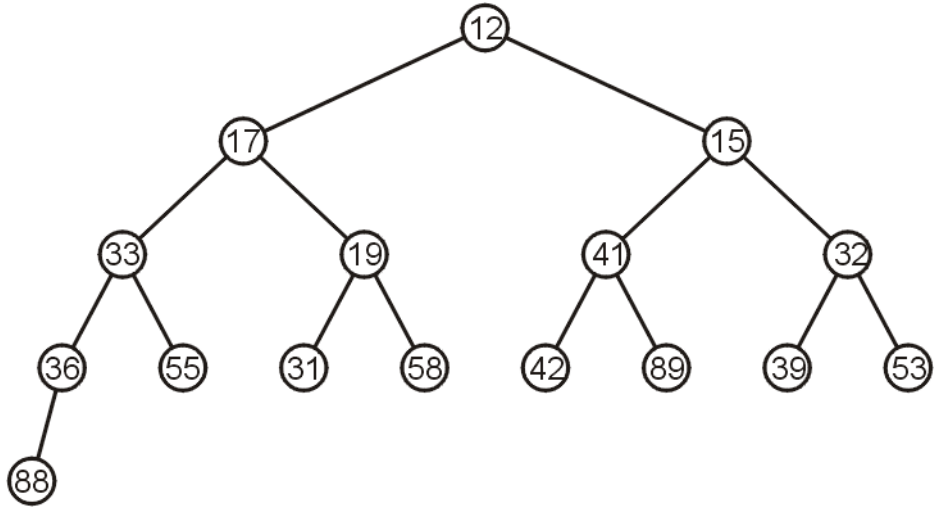
There are multiple means of keeping balance with binary heaps:

- Complete binary trees
- Leftist heaps
- Skew heaps

We will look at using complete binary trees

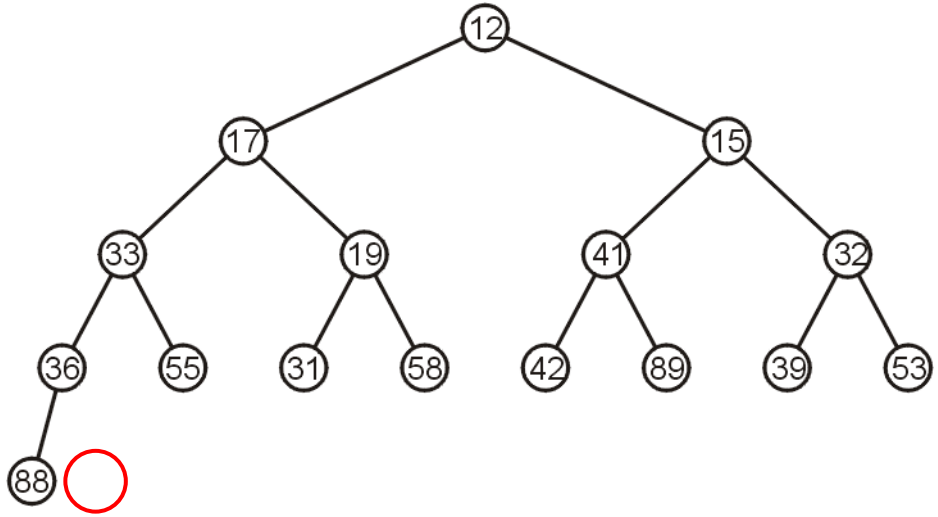
Complete Trees

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



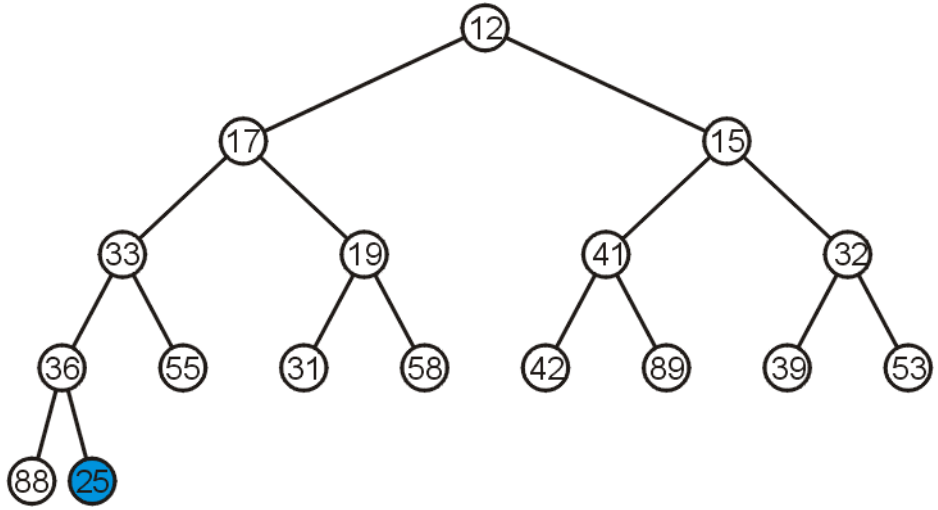
Complete Trees: Push

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

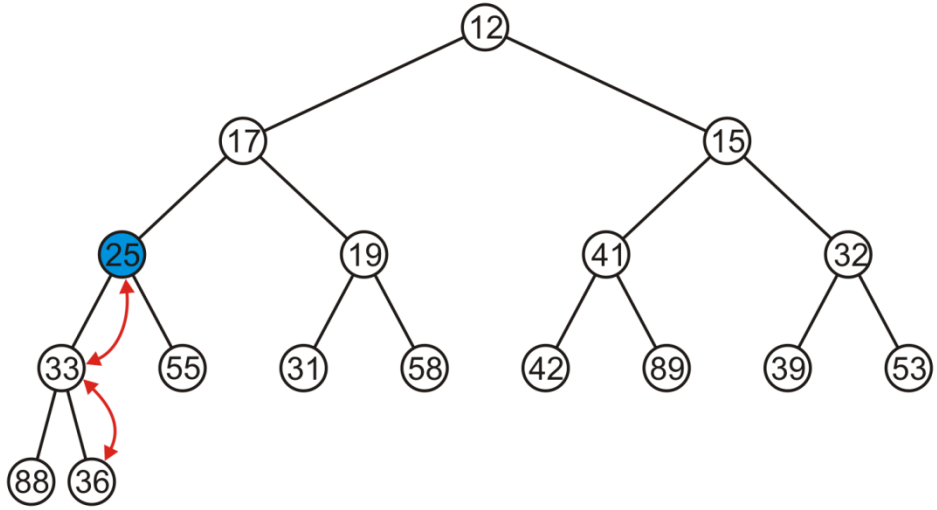
For example, push 25:



Complete Trees: Push

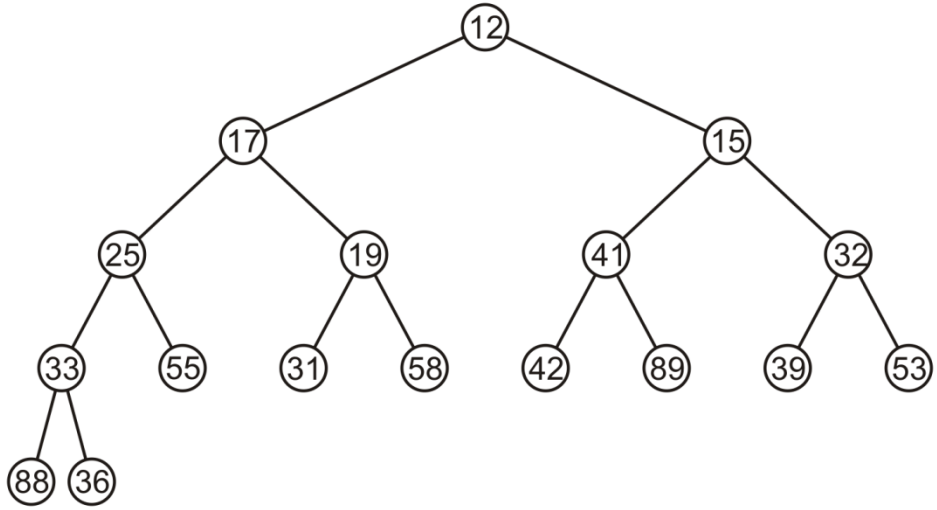
We have to percolate 25 up into its appropriate location

– The resulting heap is still a complete tree



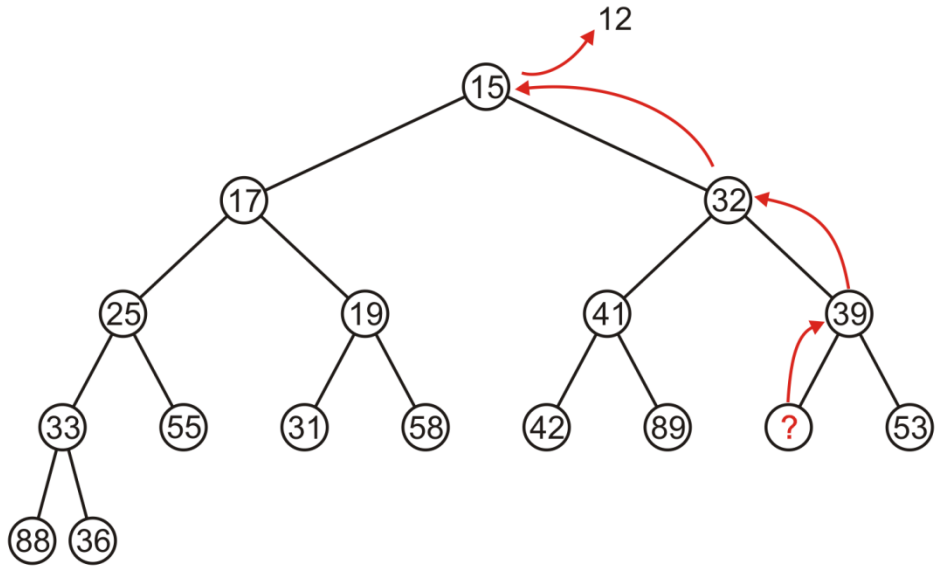
Complete Trees: Pop

Suppose we want to pop the top entry: 12



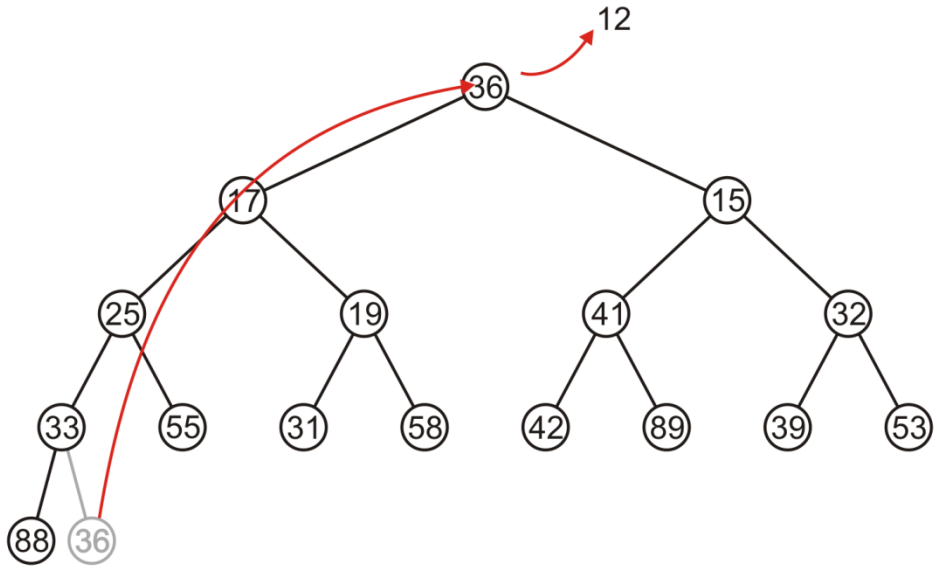
Complete Trees: Pop

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



Complete Trees: Pop

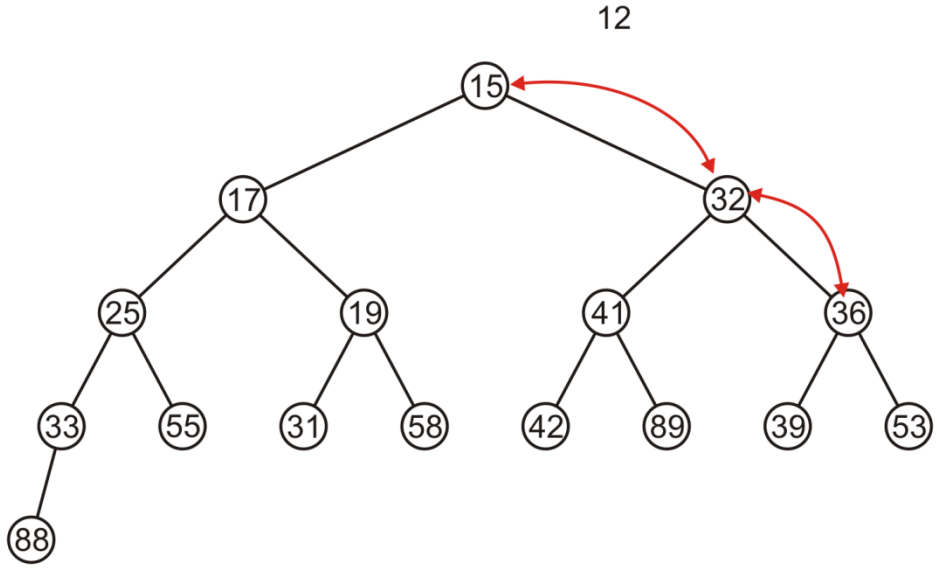
Alternatively, copy the last entry in the heap to the root



Complete Trees: Pop

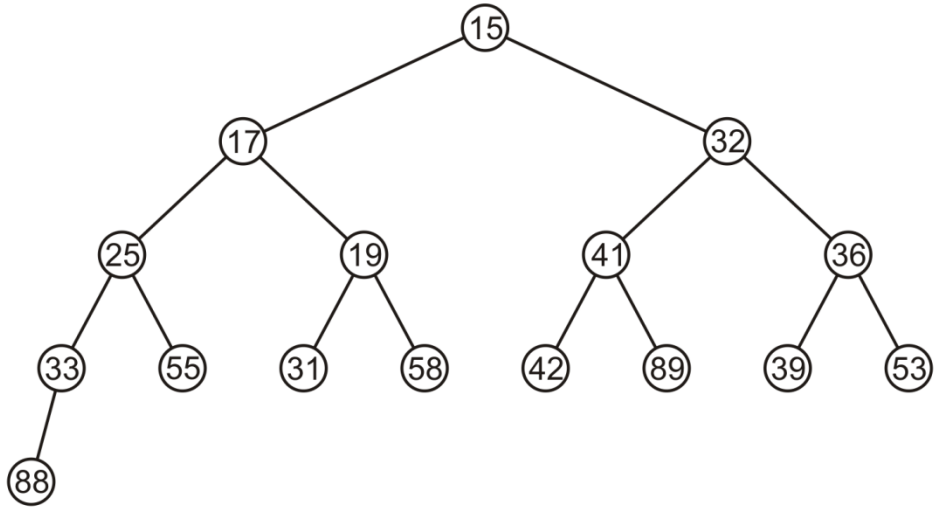
Now, percolate 36 down swapping it with the smallest of its children

– We halt when both children are larger



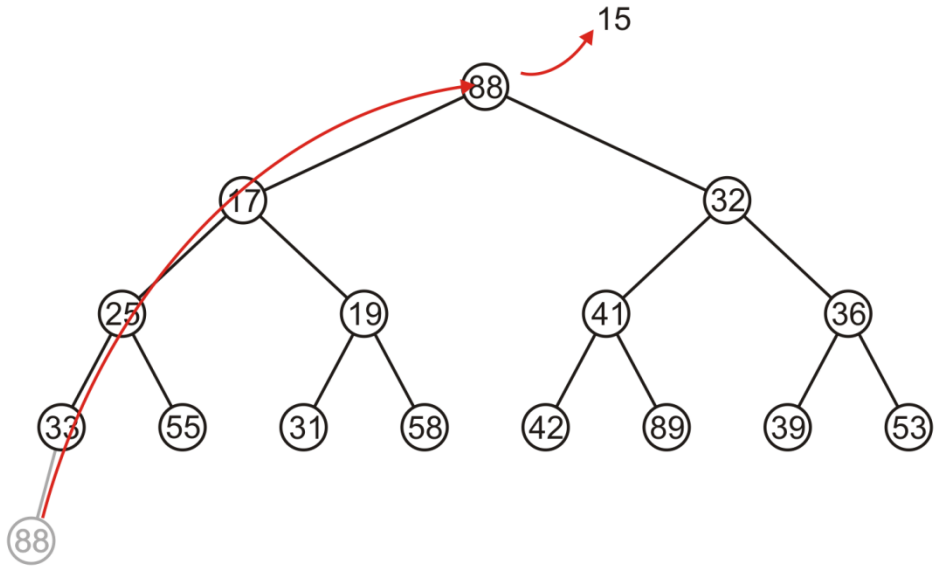
Complete Trees: Pop

The resulting tree is now still a complete tree:



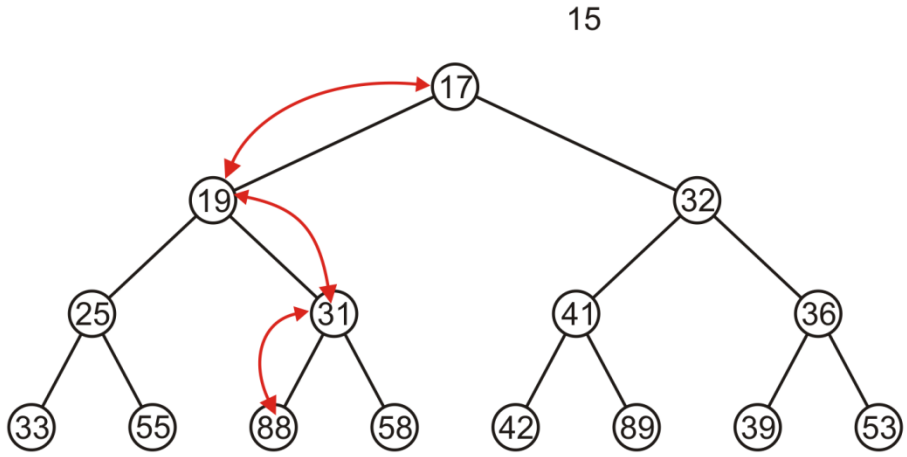
Complete Trees: Pop

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



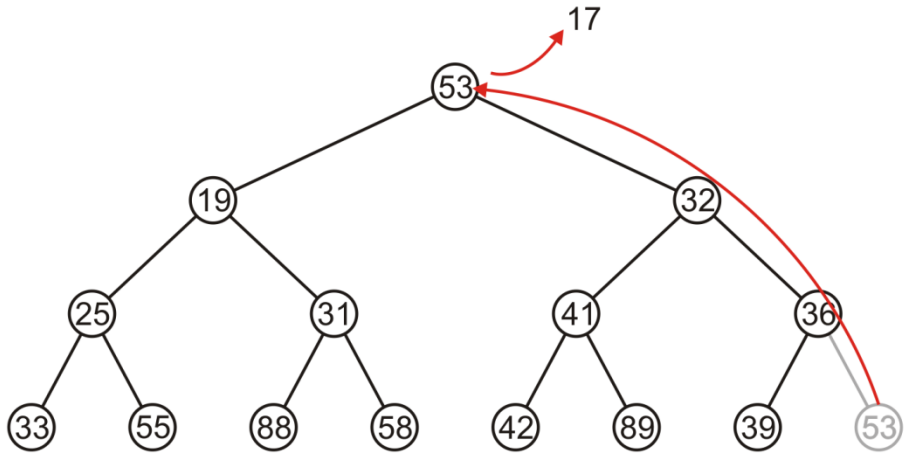
Complete Trees: Pop

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



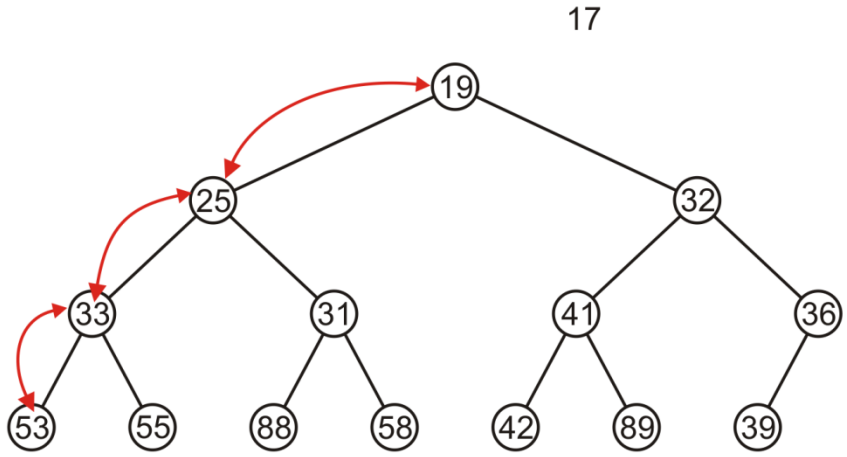
Complete Trees: Pop

In popping 17, 53 is moved to the top



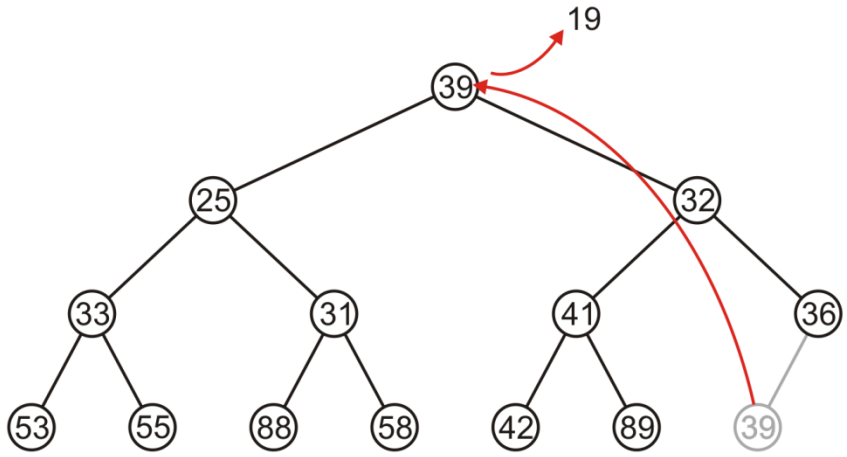
Complete Trees: Pop

And percolated down, again to the deepest level



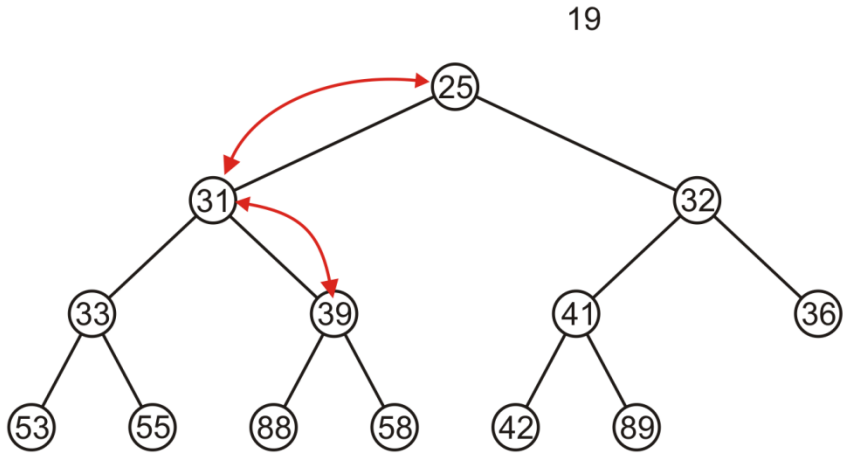
Complete Trees: Pop

Popping 19 copies up 39



Complete Trees: Pop

Which is then percolated down to the second deepest level



Complete Tree

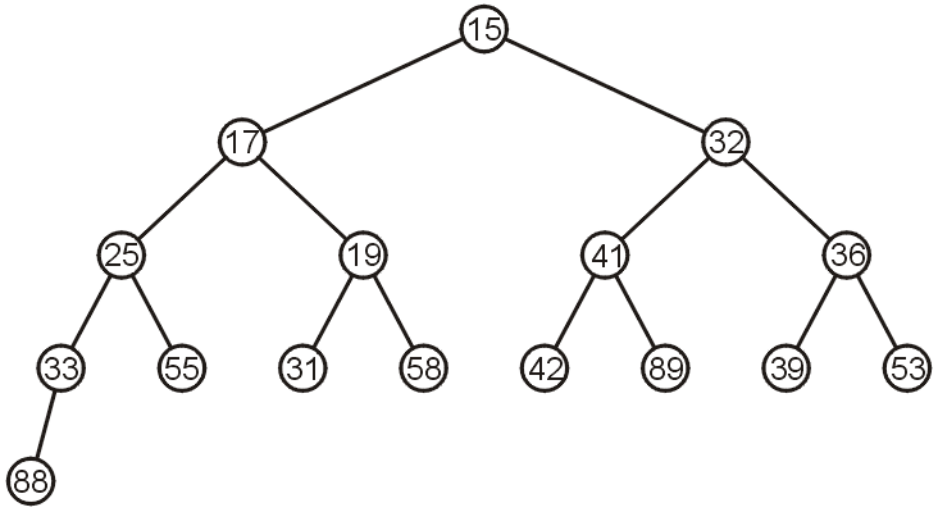
Therefore, we can maintain the complete-tree shape of a heap

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

Array Implementation

For the heap

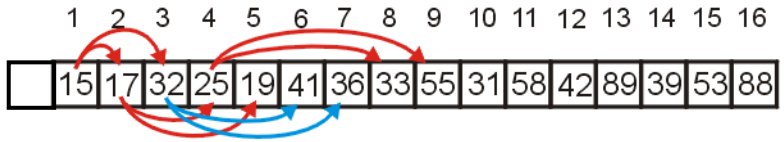


a breadth-first traversal yields:

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

Array Implementation

Recall that If we associate an index–starting at 1–with each entry in the breadth-first traversal, we get:



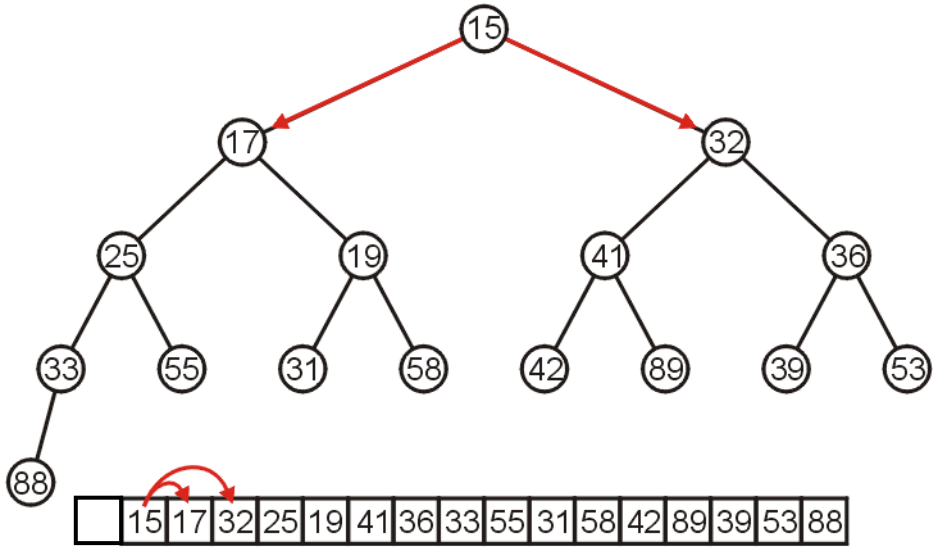
Given the entry at index k, it follows that:

- The parent of node is a floor($k/2$)
- the children are at $2k$ and $2k + 1$

Cost (trivial): start array at position 1 instead of position 0

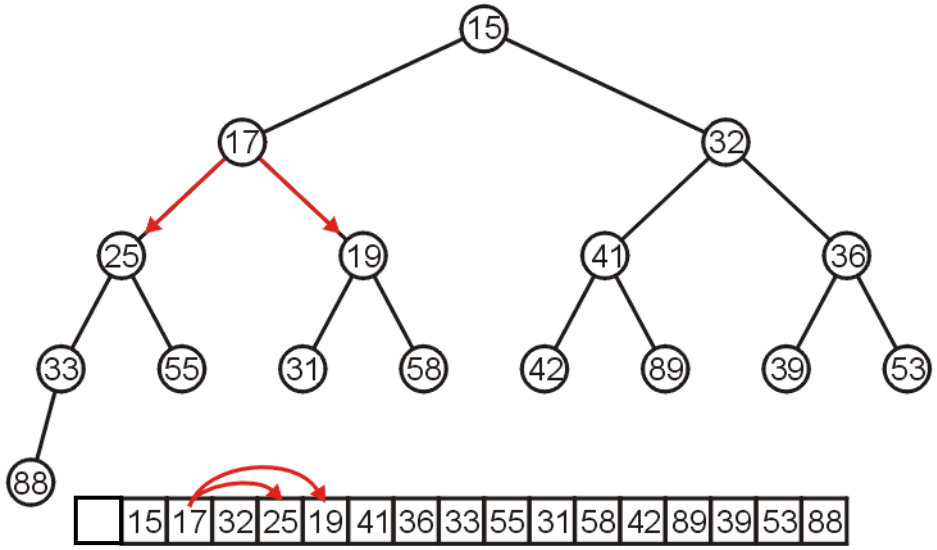
Array Implementation

The children of 15 are 17 and 32:



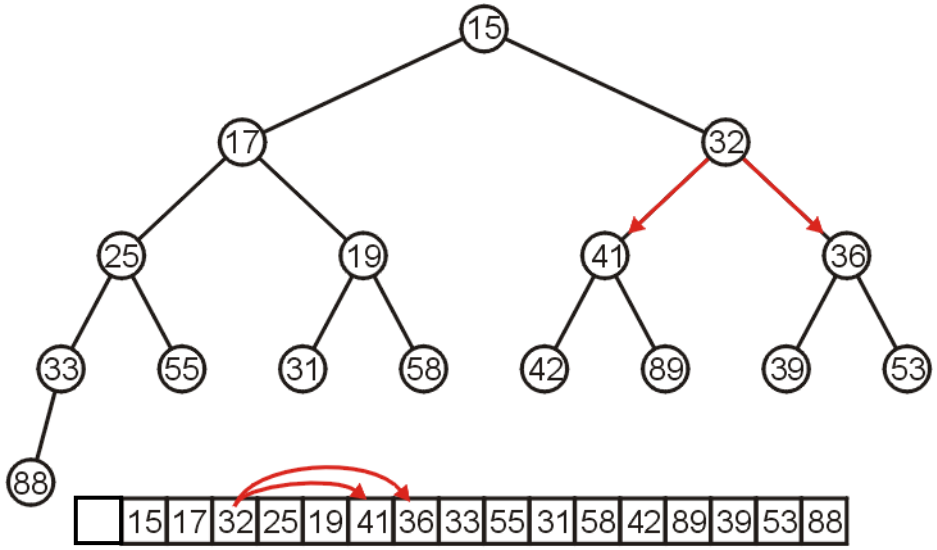
Array Implementation

The children of 17 are 25 and 19:



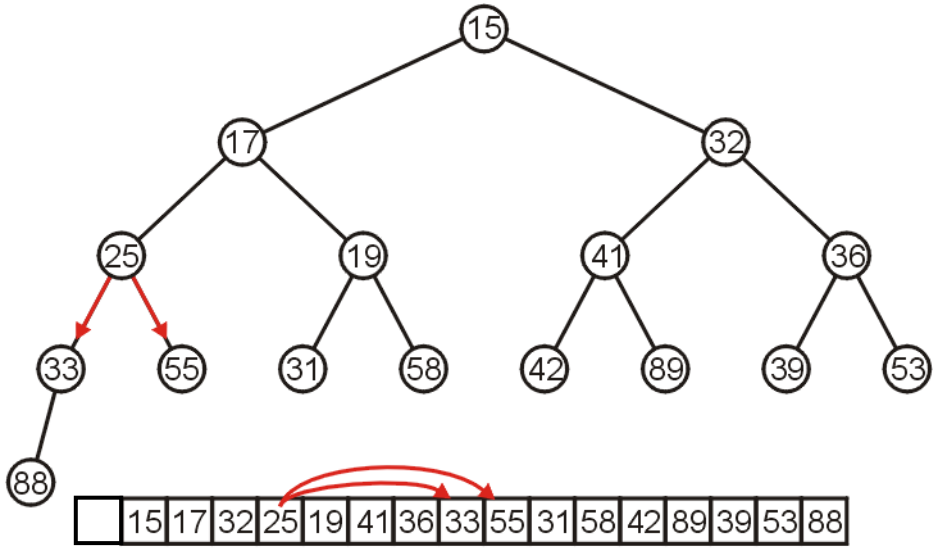
Array Implementation

The children of 32 are 41 and 36:



Array Implementation

The children of 25 are 33 and 55:



Array Implementation

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

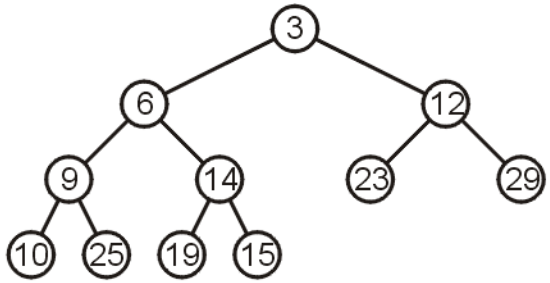
We compare the item at location **`posn`** with the item at **`posn/2`**

If they are out of order

- Swap them, set **`posn /= 2`** and repeat

Array Implementation

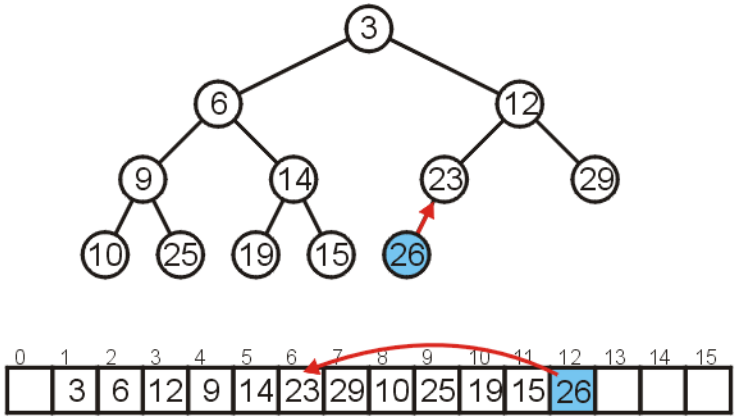
Consider the following heap, both as a tree and in its array representation



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

Array Implementation: Push

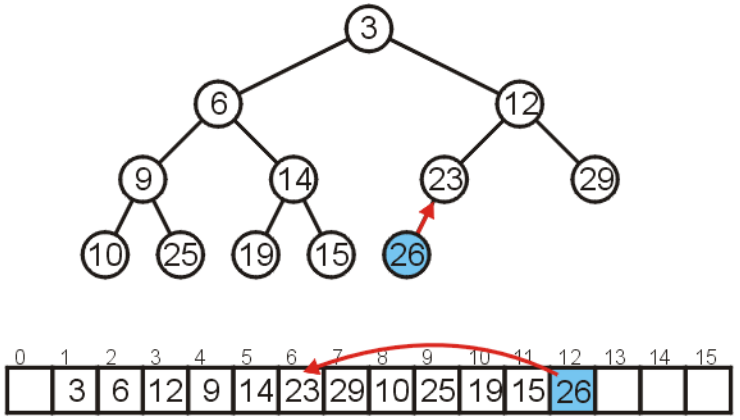
Inserting 26 requires no changes



Array Implementation: Push

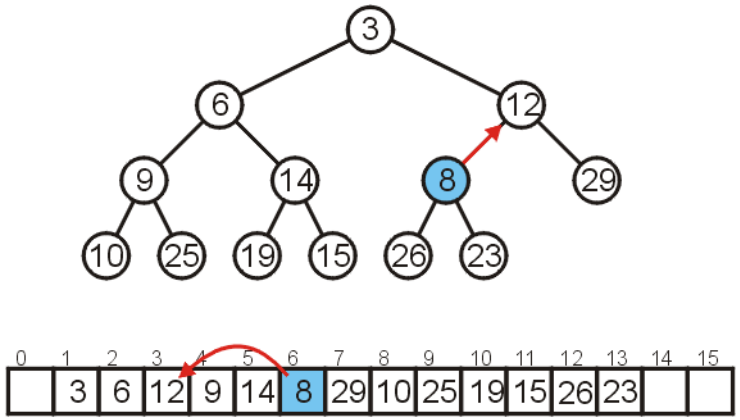
Inserting 8 requires a few percolations:

- Swap 8 and 23



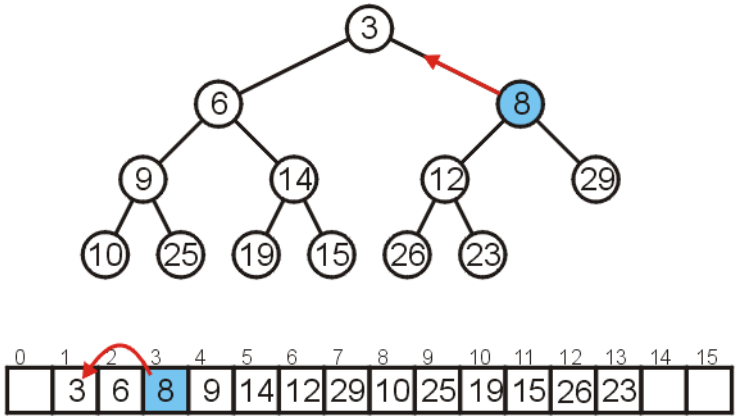
Array Implementation: Push

Swap 8 and 12



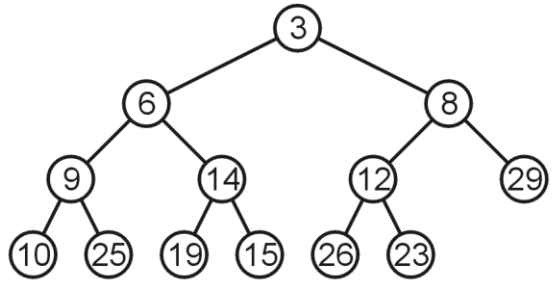
Array Implementation: Push

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



Array Implementation: Pop

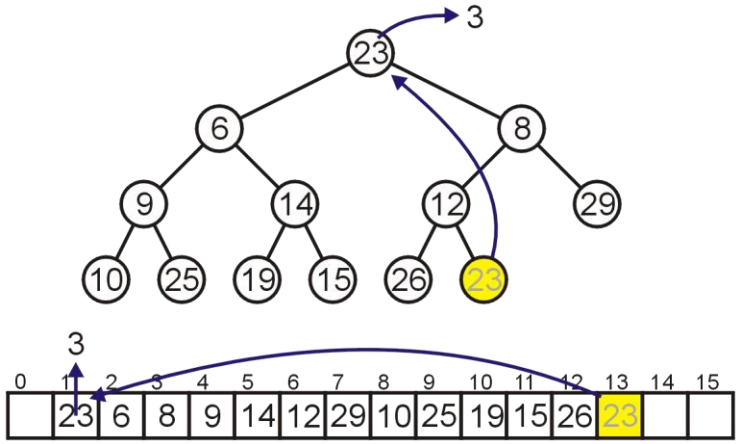
As before, popping the top has us move the last entry to the top



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

Array Implementation: Pop

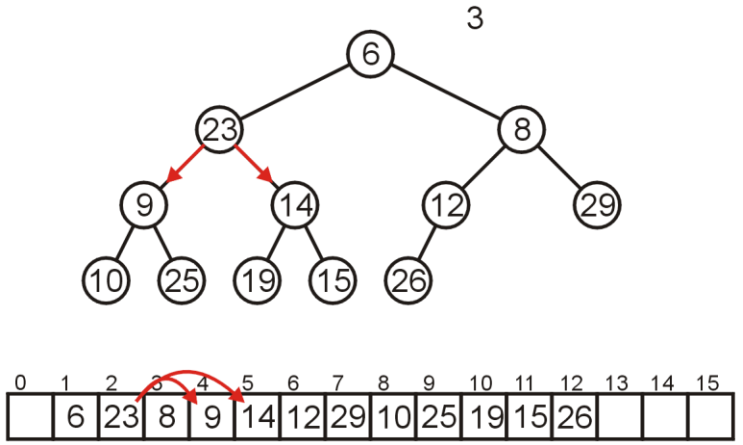
Move the last object, 23, to the root



Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

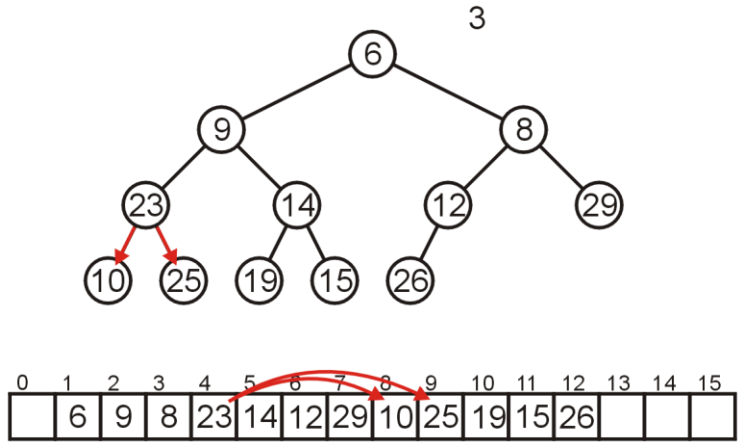
– Swap 23 and 9



Array Implementation: Pop

Compare Node 4 with its children: Nodes 8 and 9

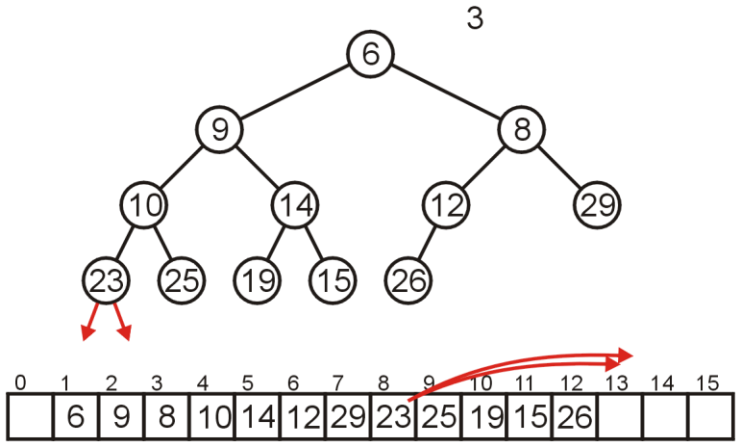
– Swap 23 and 10



Array Implementation: Pop

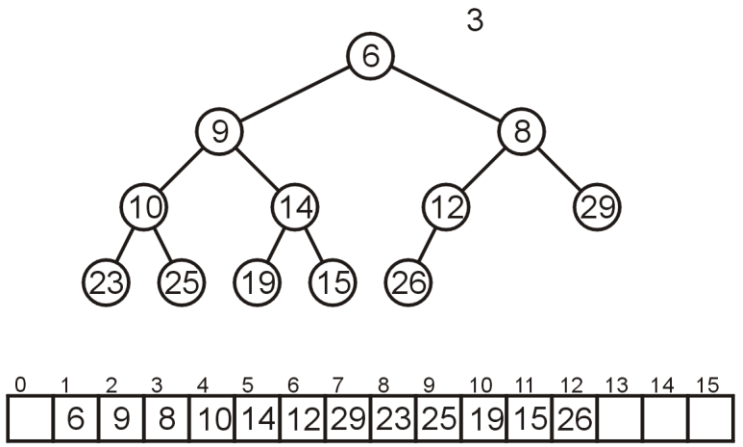
The children of Node 8 are beyond the end of the array:

- Stop



Array Implementation: Pop

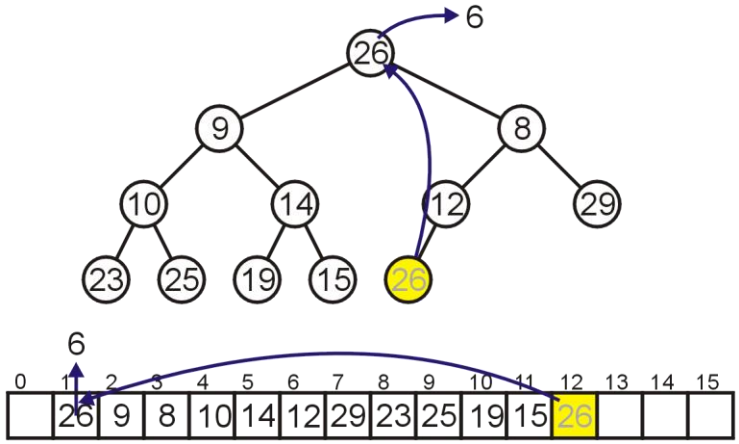
The result is a binary min-heap



Array Implementation: Pop

Dequeuing the minimum again:

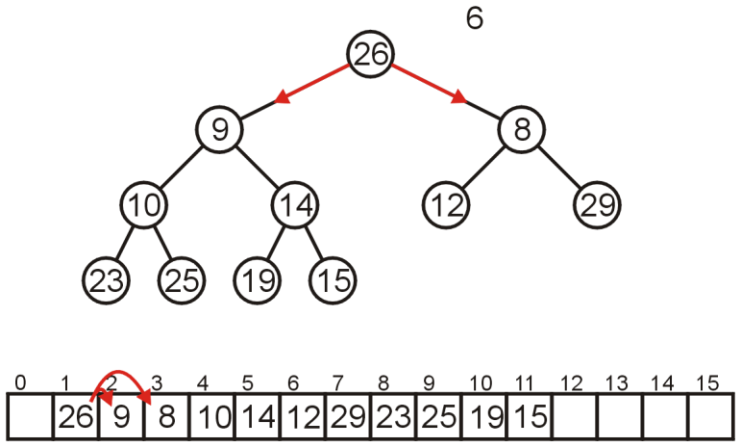
– Move 26 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

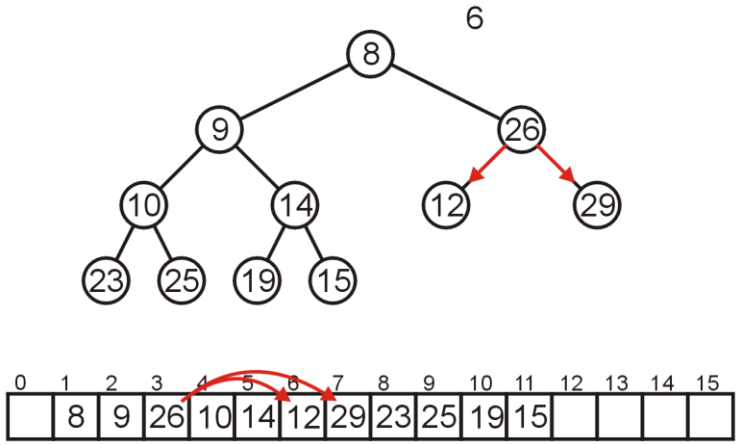
– Swap 26 and 8



Array Implementation: Pop

Compare Node 3 with its children: Nodes 6 and 7

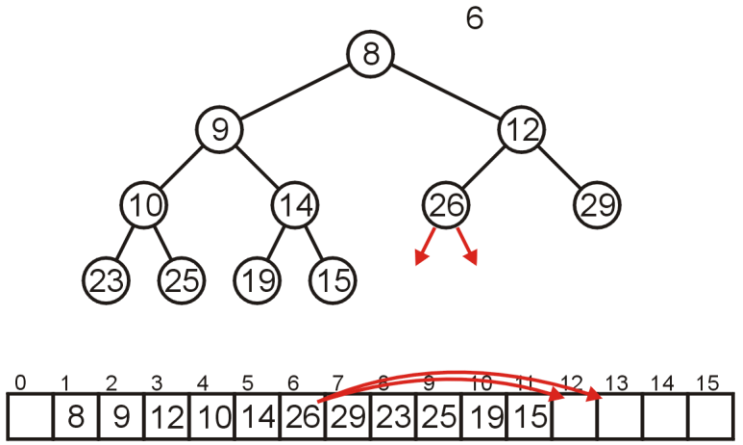
– Swap 26 and 12



Array Implementation: Pop

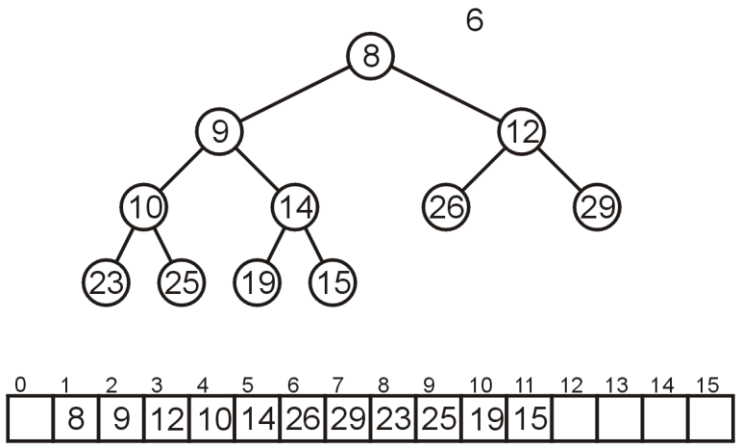
The children of Node 6, Nodes 12 and 13 are unoccupied

- Currently, `count == 11`



Array Implementation: Pop

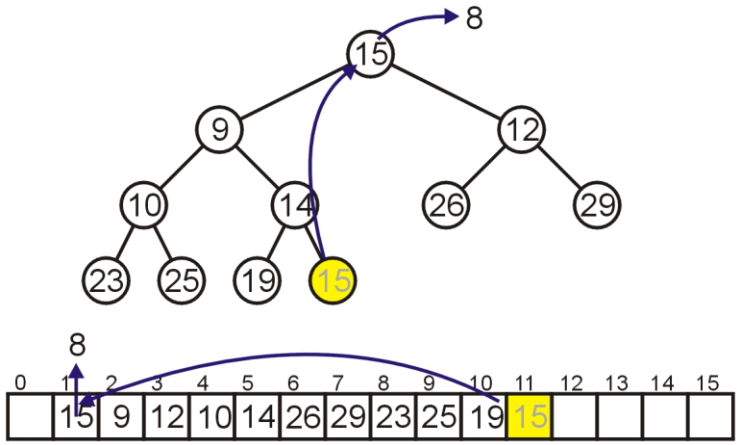
The result is a min-heap



Array Implementation: Pop

Dequeue the minimum a third time:

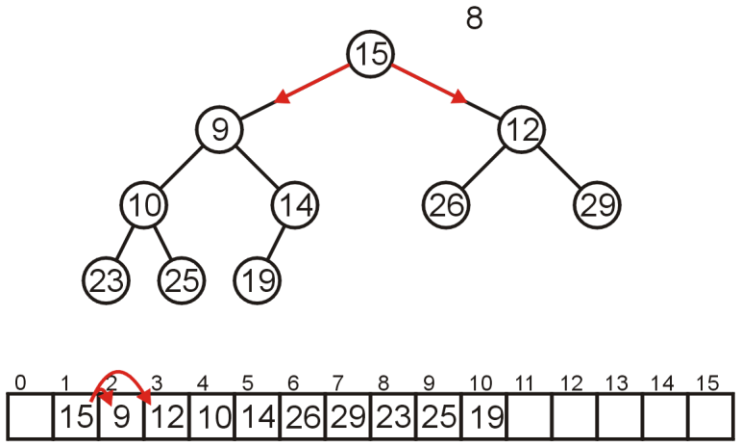
– Move 15 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

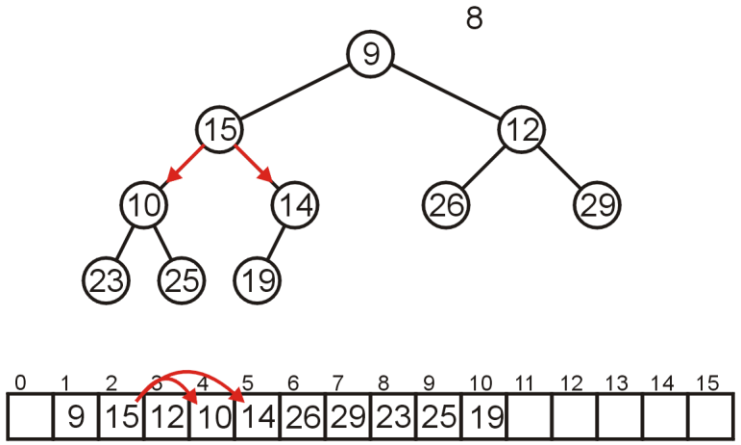
- Swap 15 and 9



Array Implementation: Pop

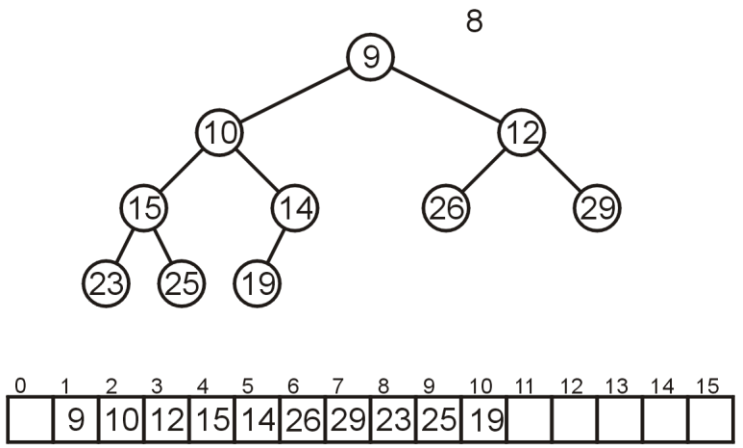
Compare Node 2 with its children: Nodes 4 and 5

– Swap 15 and 10



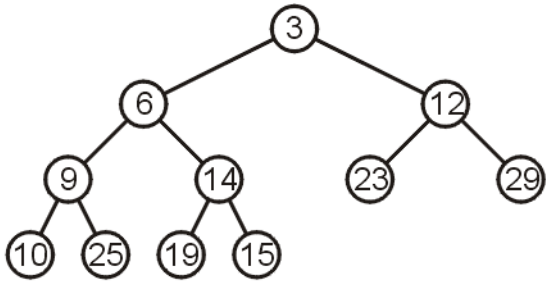
Array Implementation: Pop

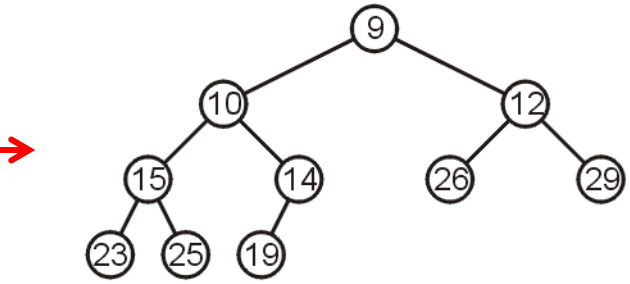
The result is a properly formed binary min-heap



Array Implementation: Pop

After all our modifications, the final heap is

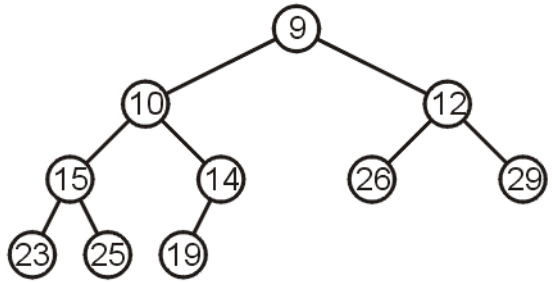


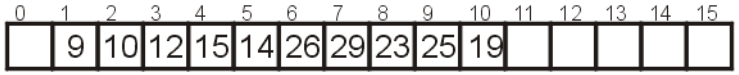


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	12	9	14	23	29	10	25	19	15				

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

Runtime Analysis





Push = O(log n)

Pop = O(log n)

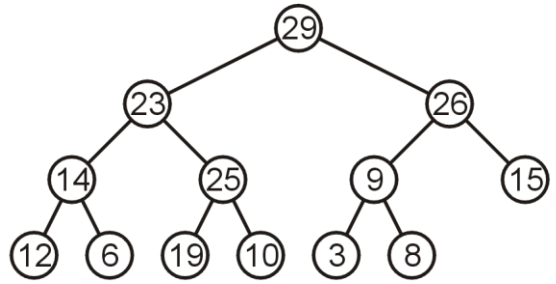
Top = O(1)

Merge two heaps?

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		

Priority Queues

Now, does using a heap ensure that that next popping object:

- has the highest priority, and
- of that highest priority, has been in the heap the longest

Consider inserting seven objects, all of the same priority (color indicates order):

2, 2, 2, 2, 2, 2, 2

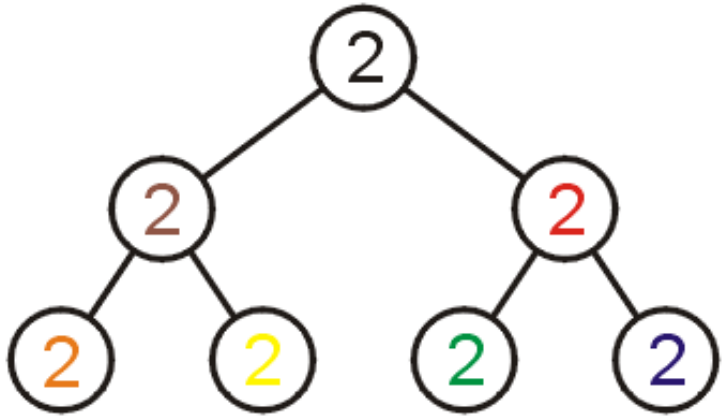
PTT		351.00		+4.00(+1.15%)
	Volume	Bid	Offer	Volume
	229,800	350.00	351.00	17,700
	129,300	349.00	352.00	733,200
	75,800	348.00	353.00	36,700
	113,800	347.00	354.00	190,000
	80,400	346.00	355.00	104,700

Priority Queues

Whatever algorithm we use for promoting must ensure that the first object remains in the root position

- Thus, we must use an insertion technique where we only percolate up if the priority is lower

The result:



Challenge:

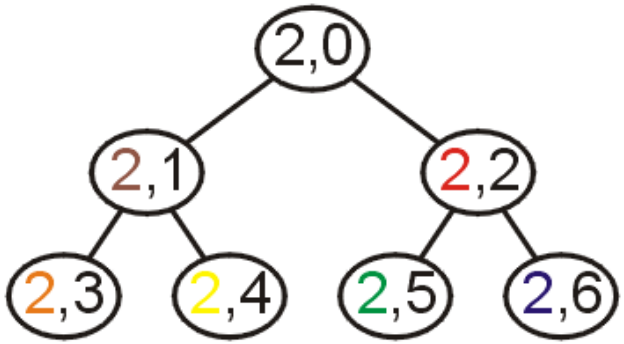
- Come up with an algorithm which removes all seven objects in the original order

Lexicographical Ordering

A better solution is to modify the priority:

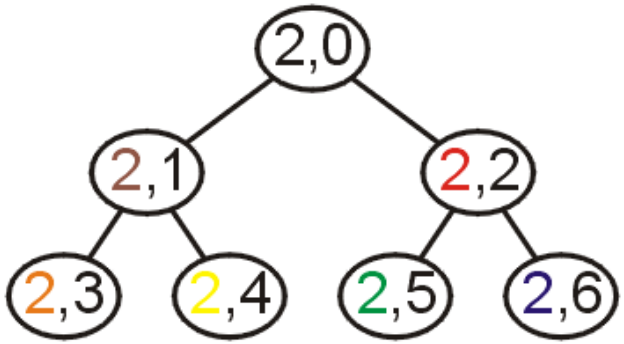
- Track the number of insertions with a counter k (initially 0)
- For each insertion with priority n, create a hybrid priority (n, k) where:

  $(n_1, k_1) < (n_2, k_2)$ if $n_1 < n_2$ or ($n_1 = n_2$ and $k_1 < k_2$)



Priority Queues

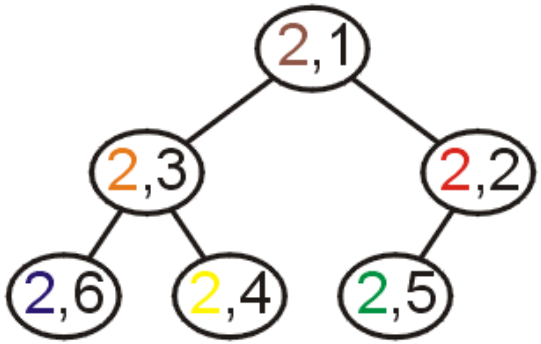
Removing the objects would be in the following order:



Priority Queues

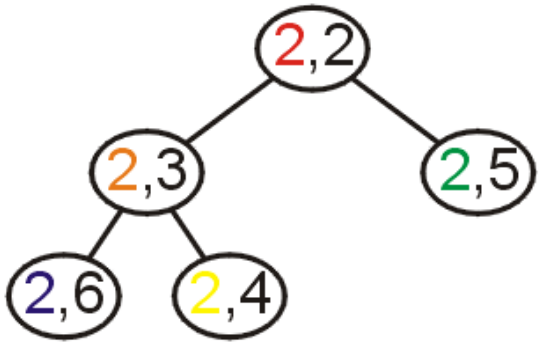
Popped: 2

– First, (2,1) < (2, 2) and (2, 3) < (2, 4)



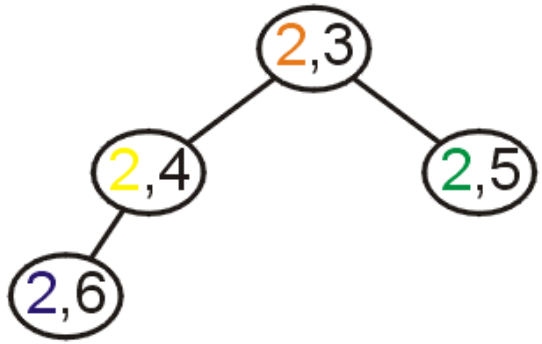
Priority Queues

Removing the objects would be in the following order:



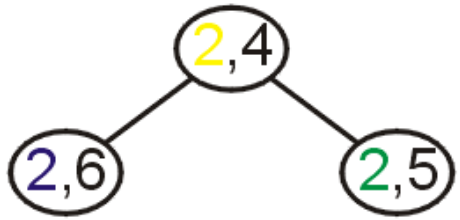
Priority Queues

Removing the objects would be in the following order:



Priority Queues

Removing the objects would be in the following order:



Priority Queues

Removing the objects would be in the following order: