CS 2420: Binary Heap

Dr. Tsung-Wei (TW) Huang

Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

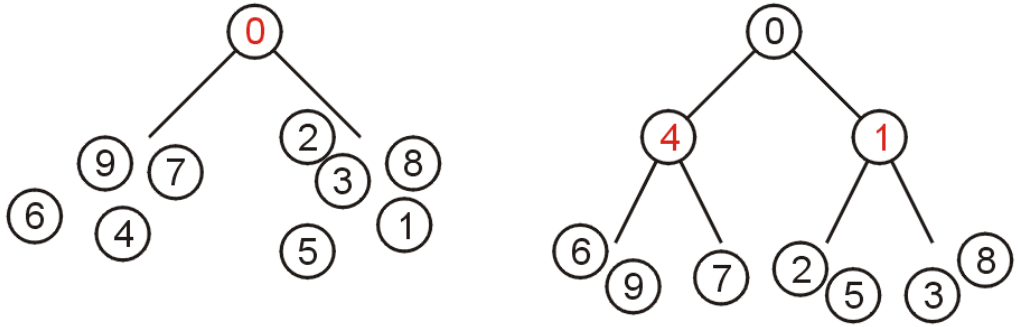
Outline

- 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

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

Definition

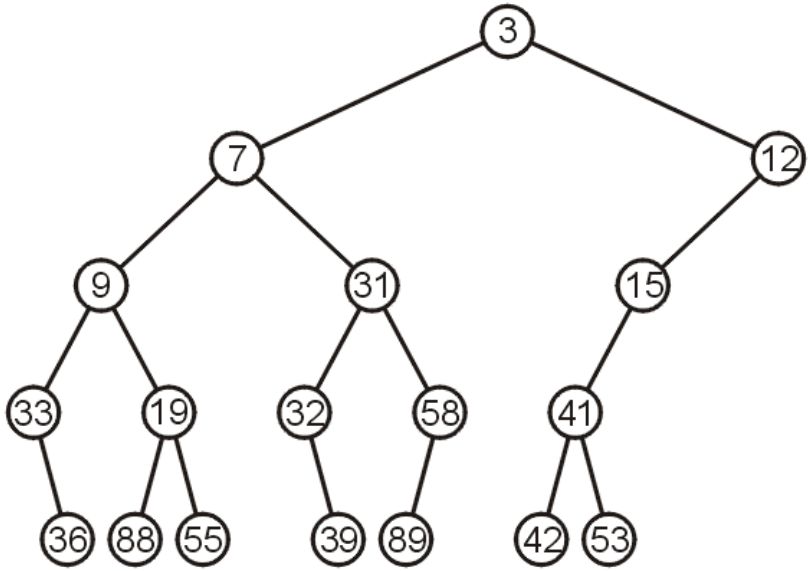
Important:

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

Failing to understand this is the greatest mistake a student makes

Example

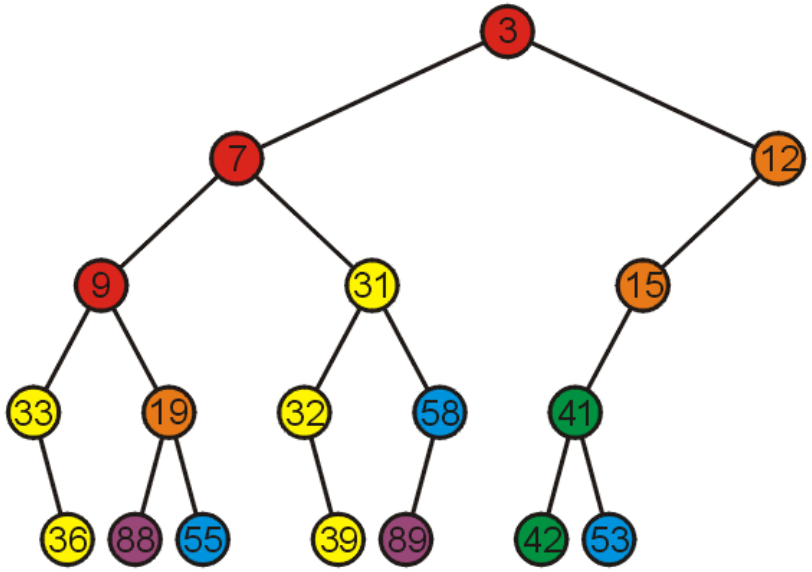
This is a binary min-heap:



Example

Adding colour, we observe

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



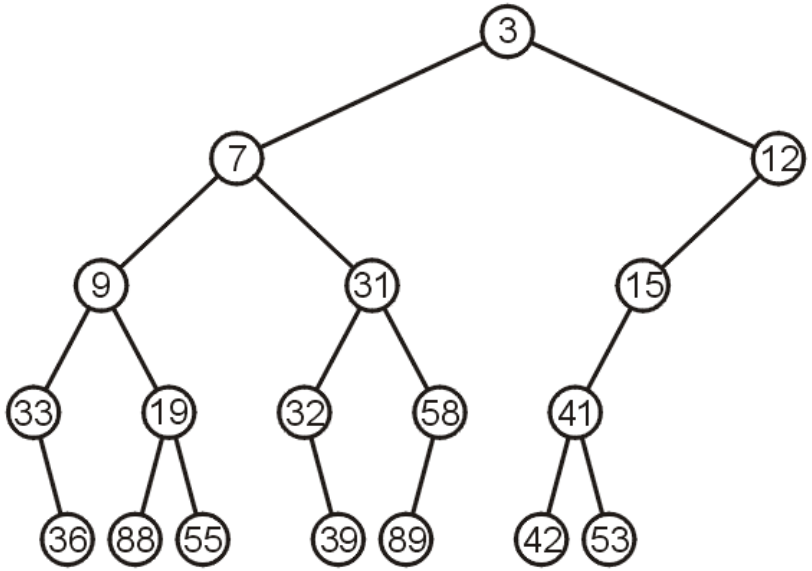
Operations

We will consider three operations:

- Top
- Pop
- Push

Example

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



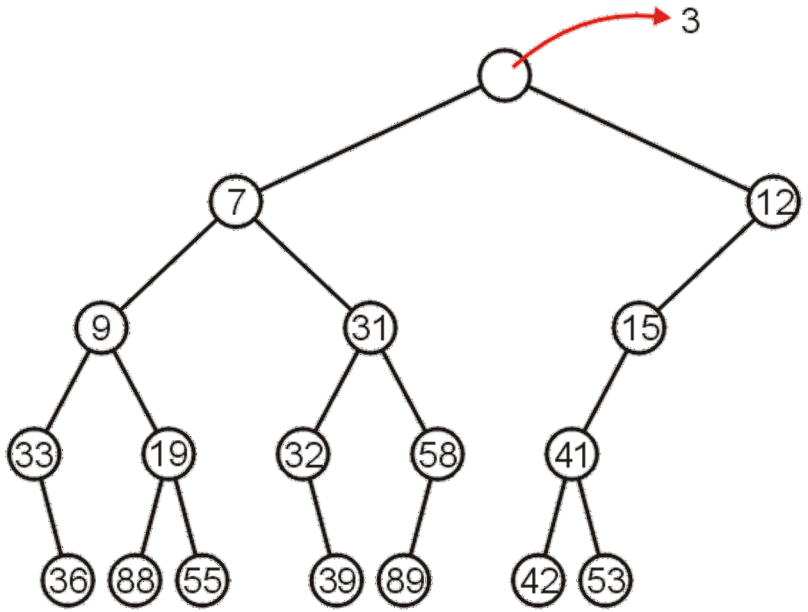
Pop

To remove the minimum object:

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

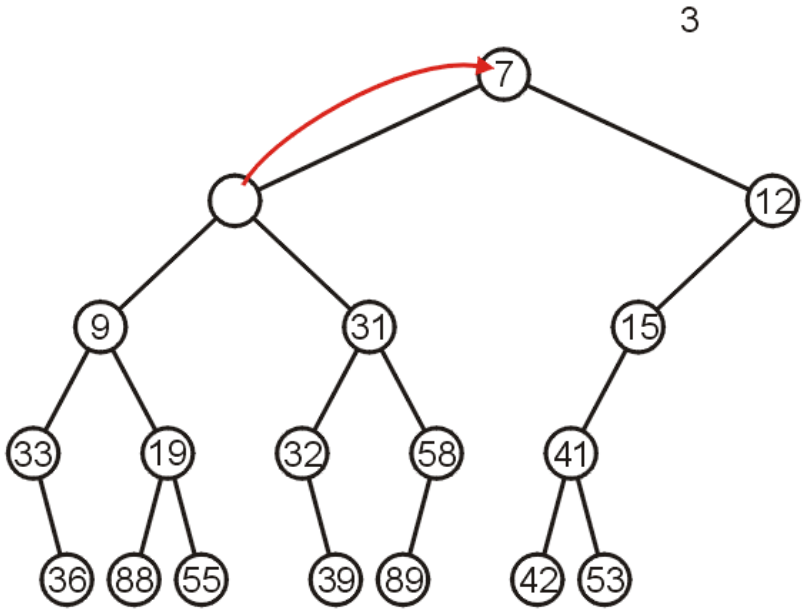
Pop

Using our example, we remove 3:



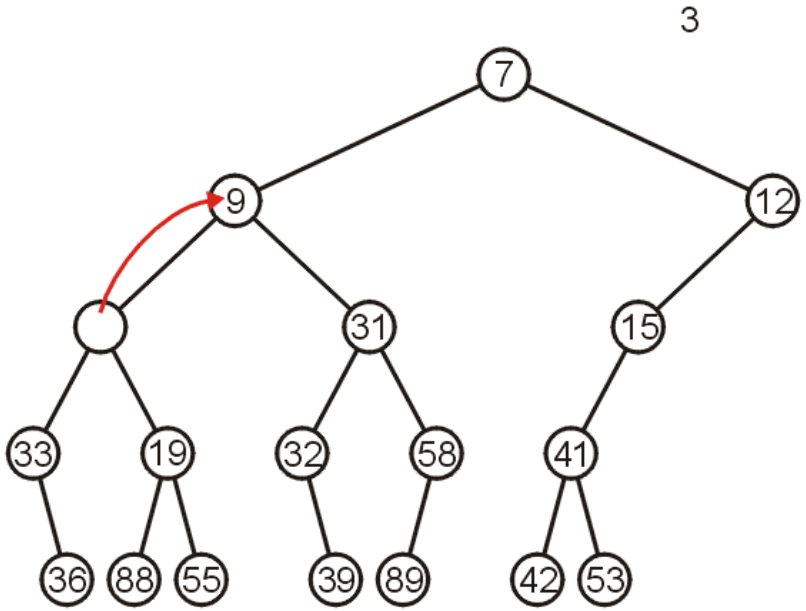
Pop

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



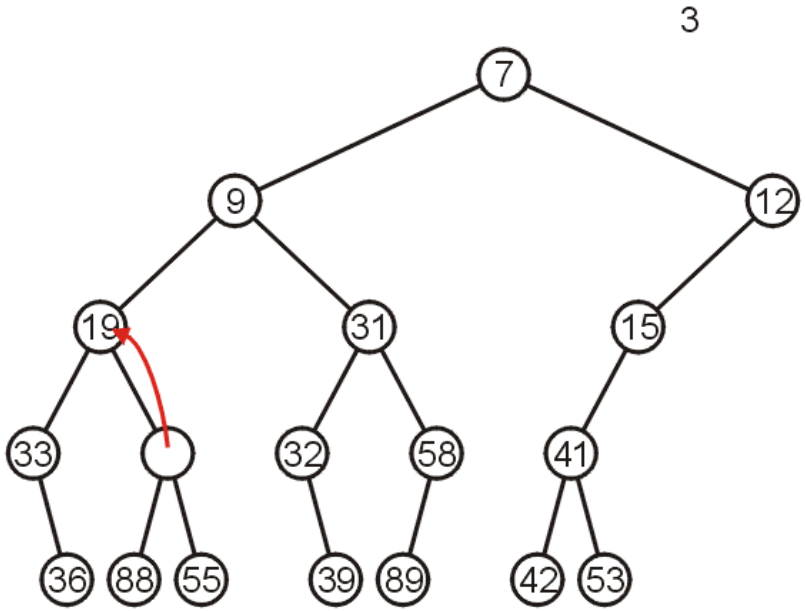
Pop

In the left sub-tree, we promote 9:



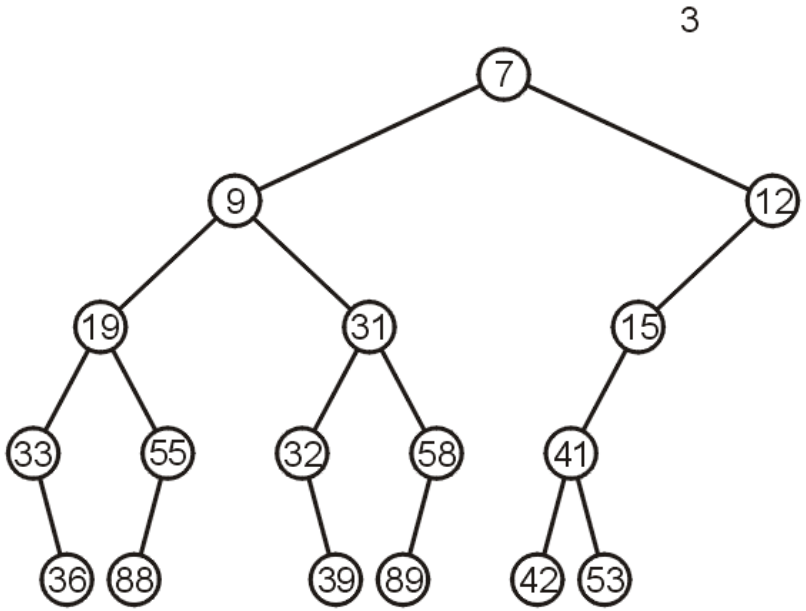
Pop

Recursively, we promote 19:



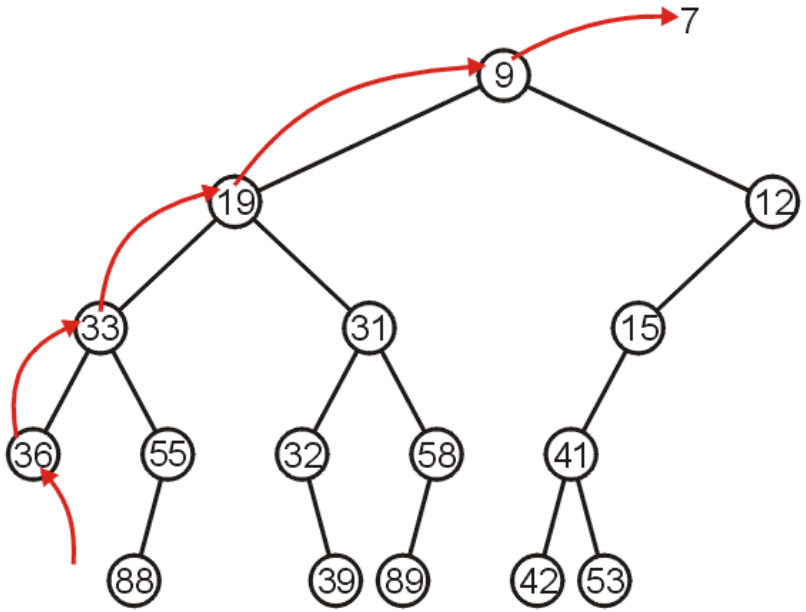
Pop

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



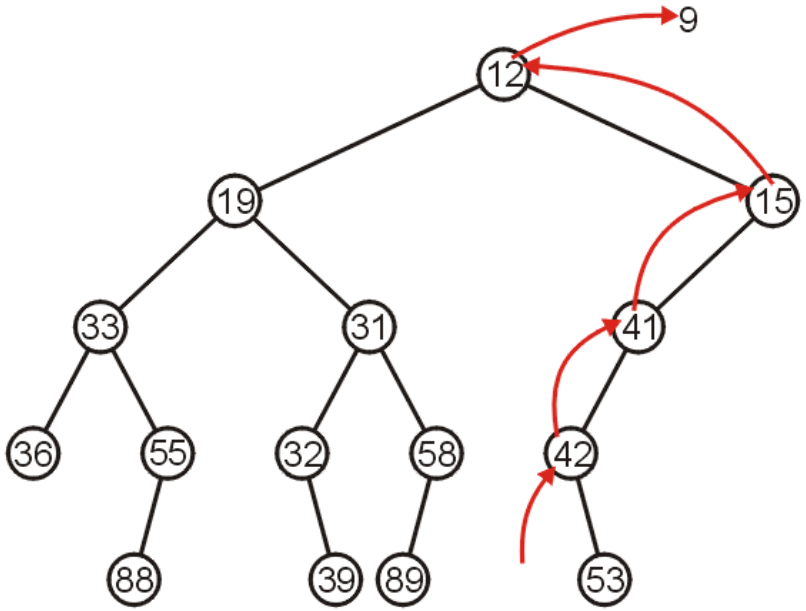
Pop

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)

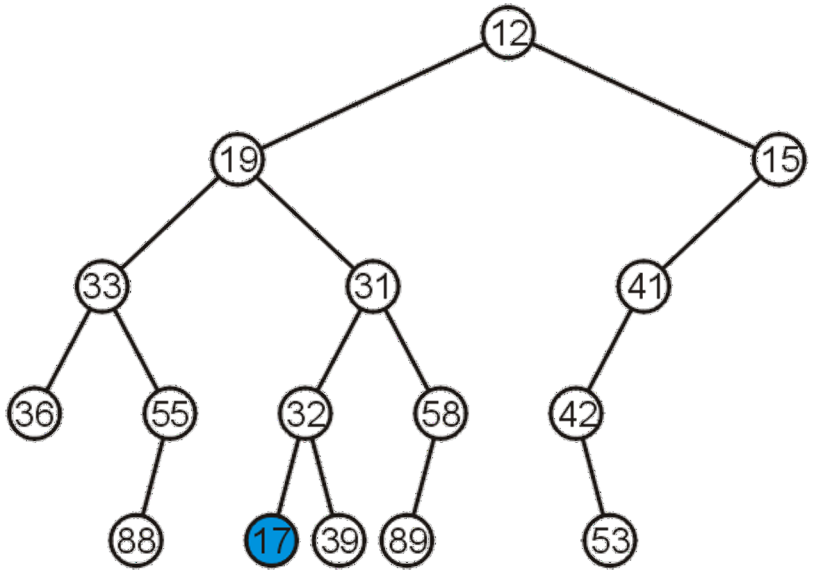
We will use the first approach with binary heaps

- Other heaps use the second

Push

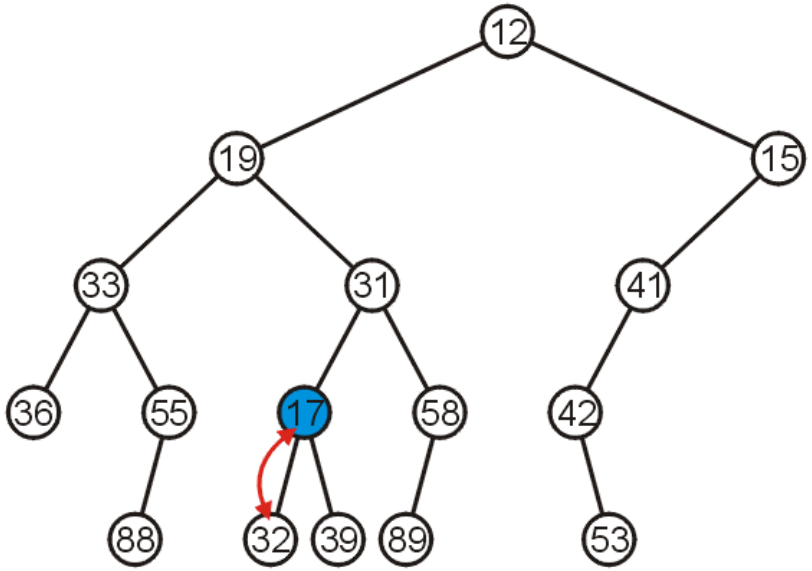
Inserting 17 into the last heap

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



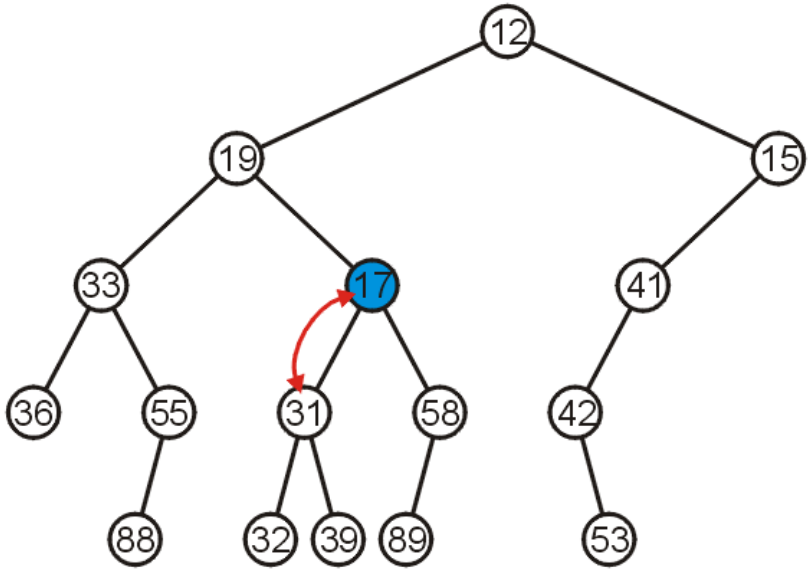
Push

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



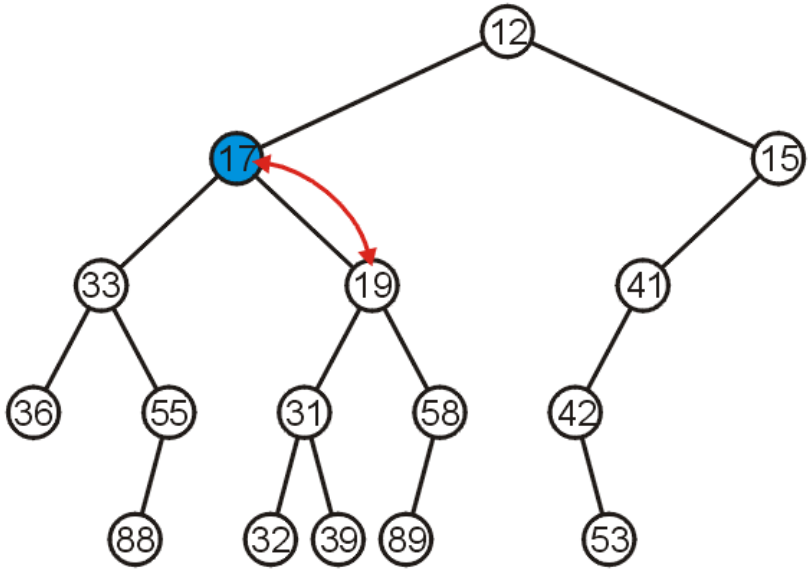
Push

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



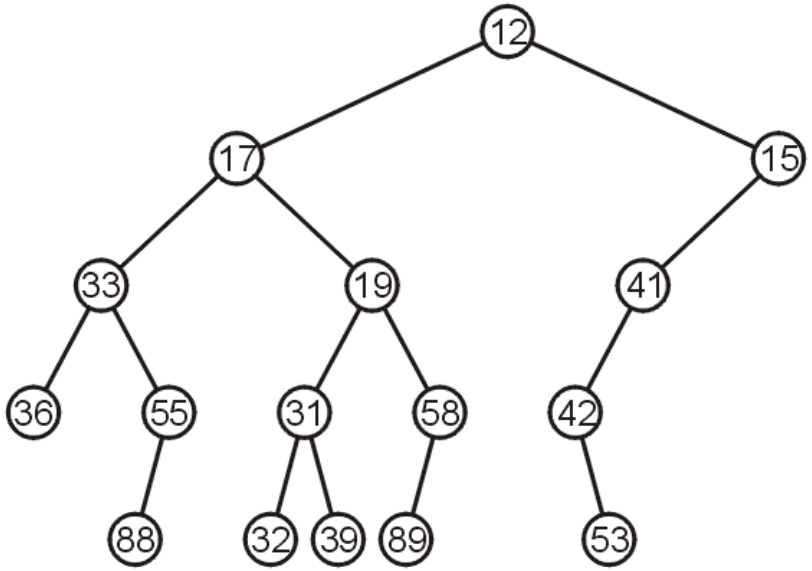
Push

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

Implementations

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

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

Complete Trees

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

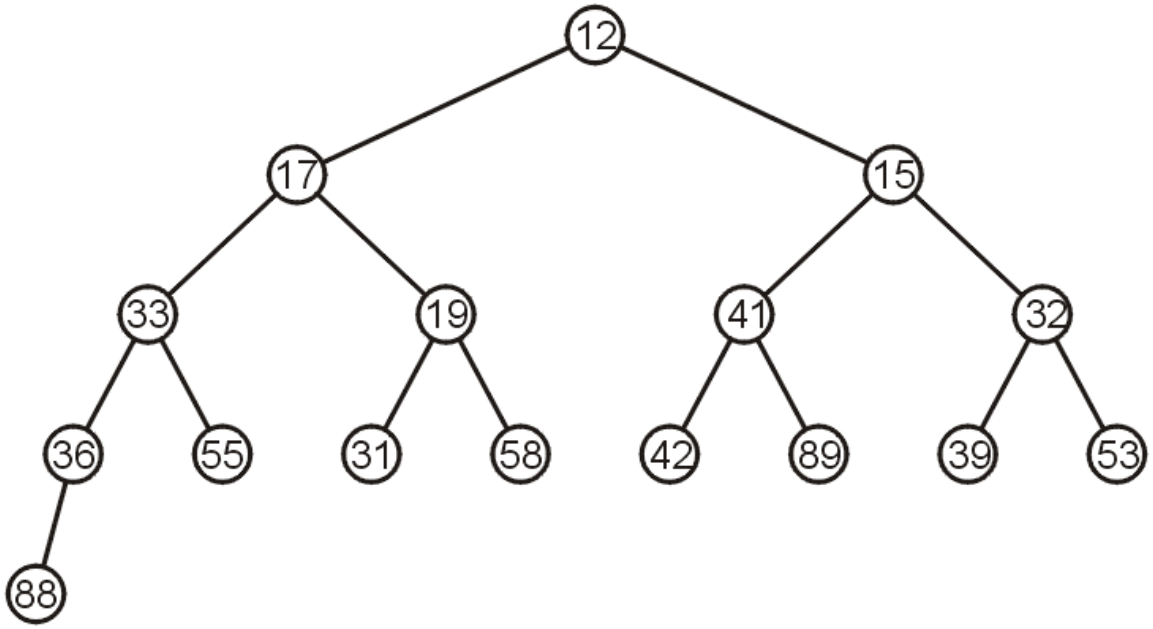
We have already seen

- 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!

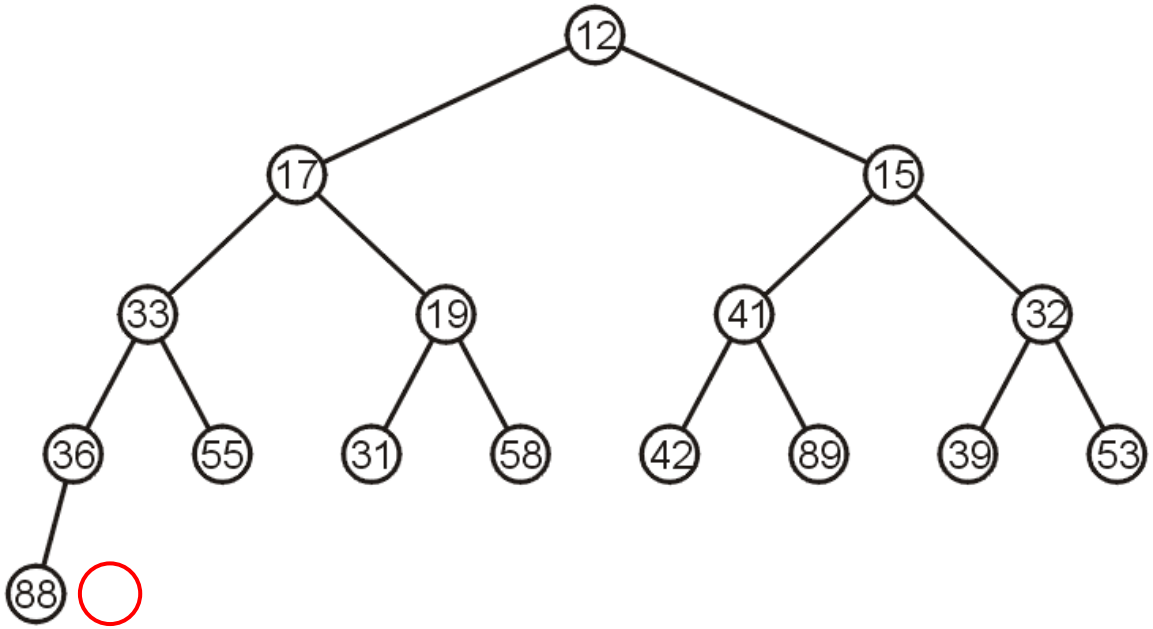
Complete Trees

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



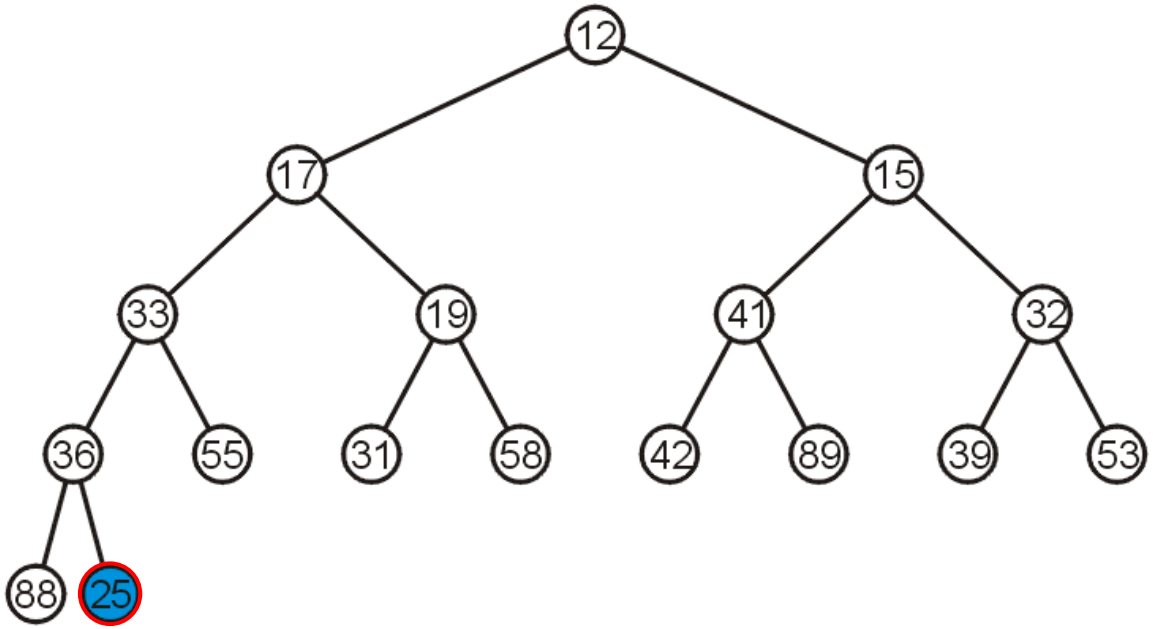
Complete Trees: Push

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

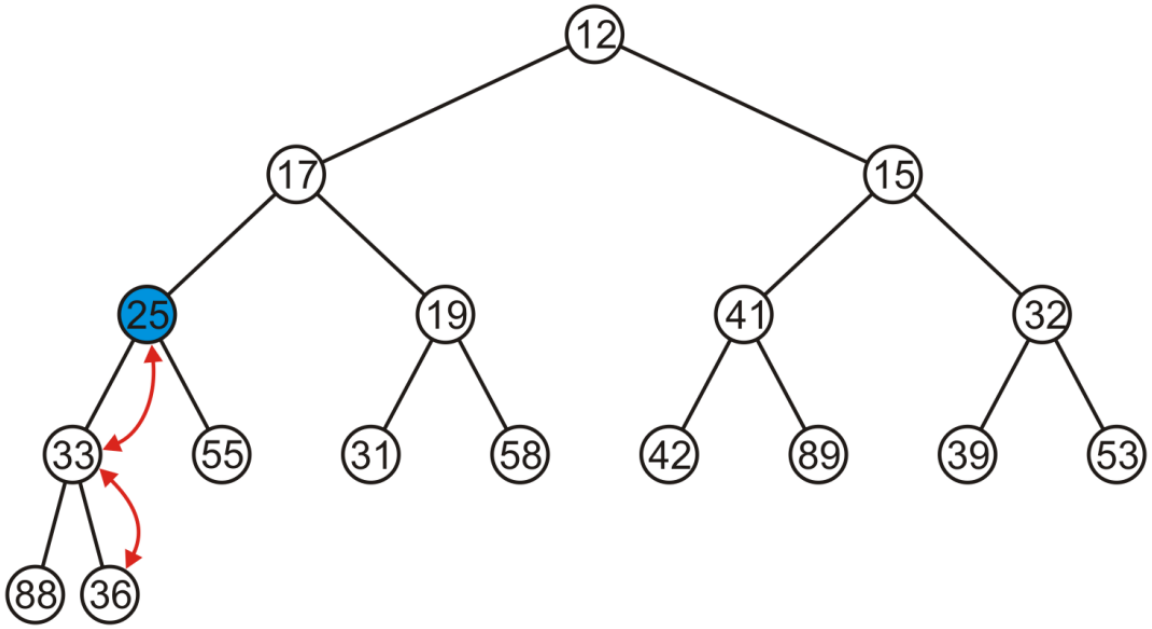
For example, push 25:



Complete Trees: Push

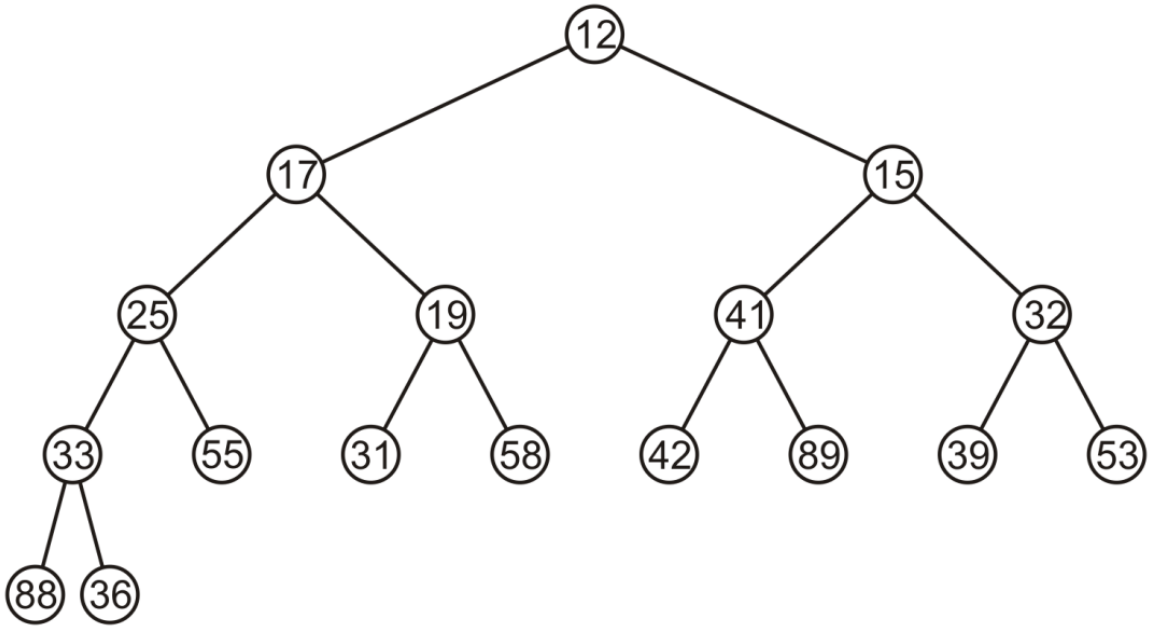
We have to percolate 25 up into its appropriate location

- The resulting heap is still a complete tree



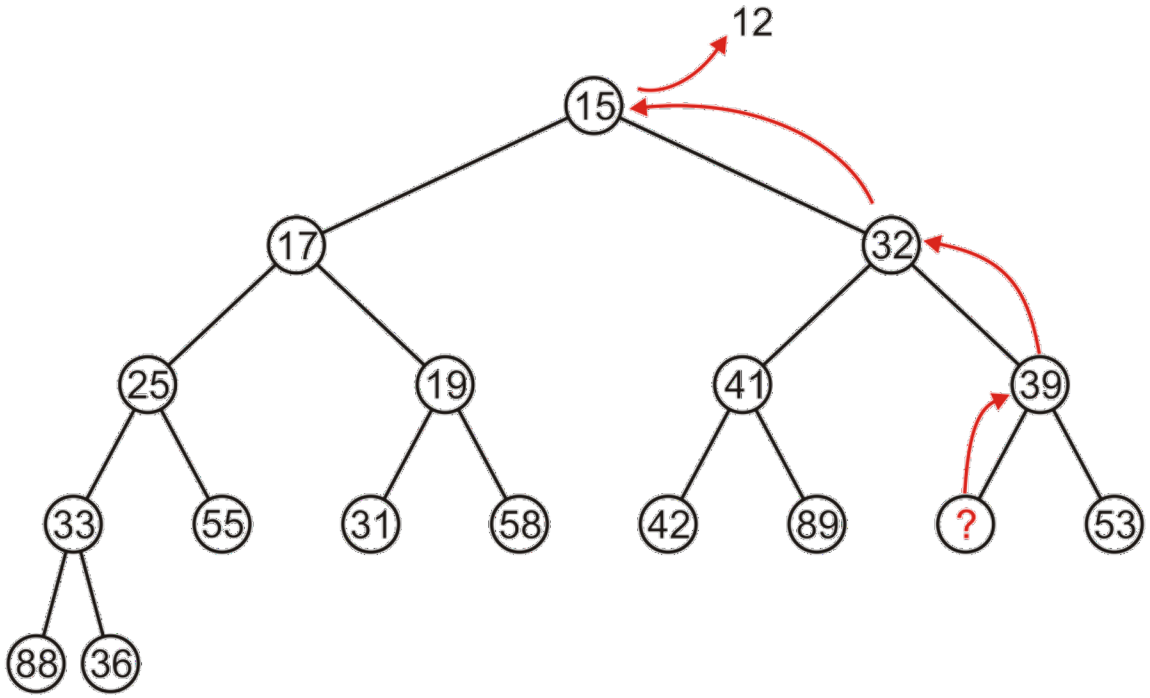
Complete Trees: Pop

Suppose we want to pop the top entry: 12



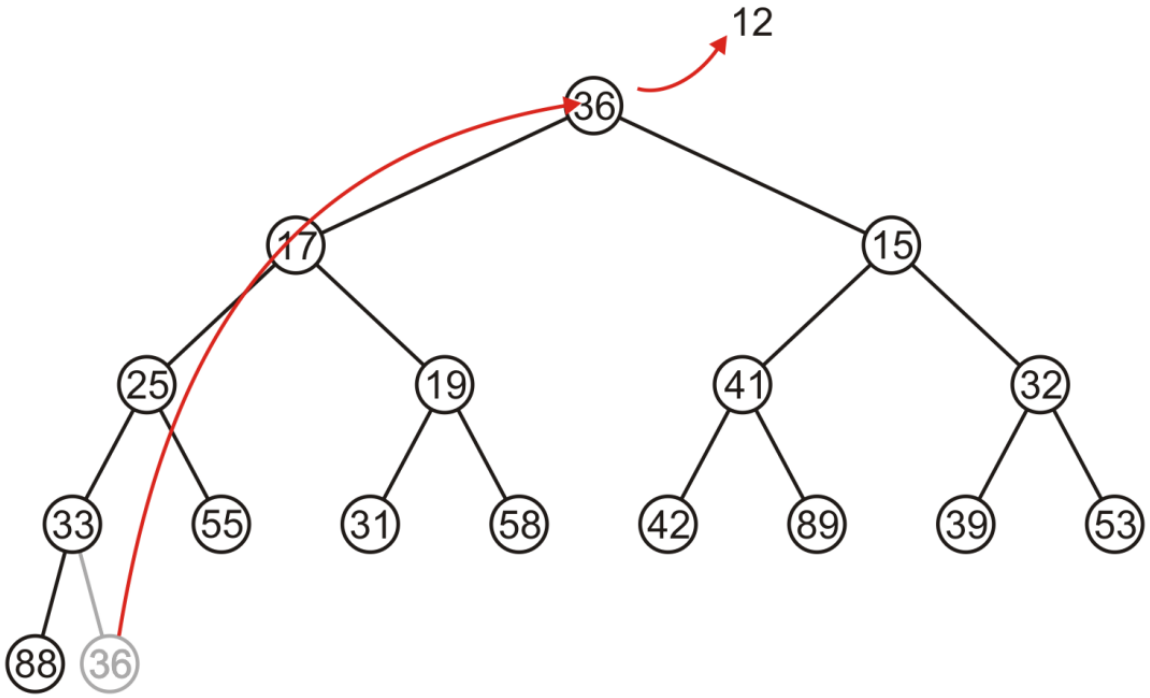
Complete Trees: Pop

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



Complete Trees: Pop

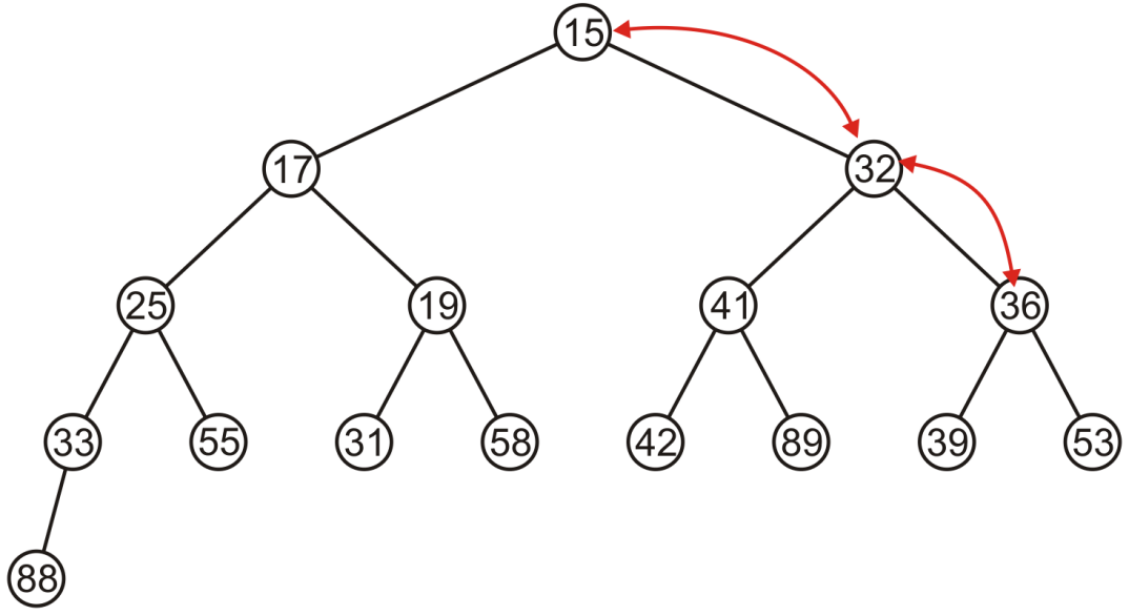
Alternatively, copy the last entry in the heap to the root



Complete Trees: Pop

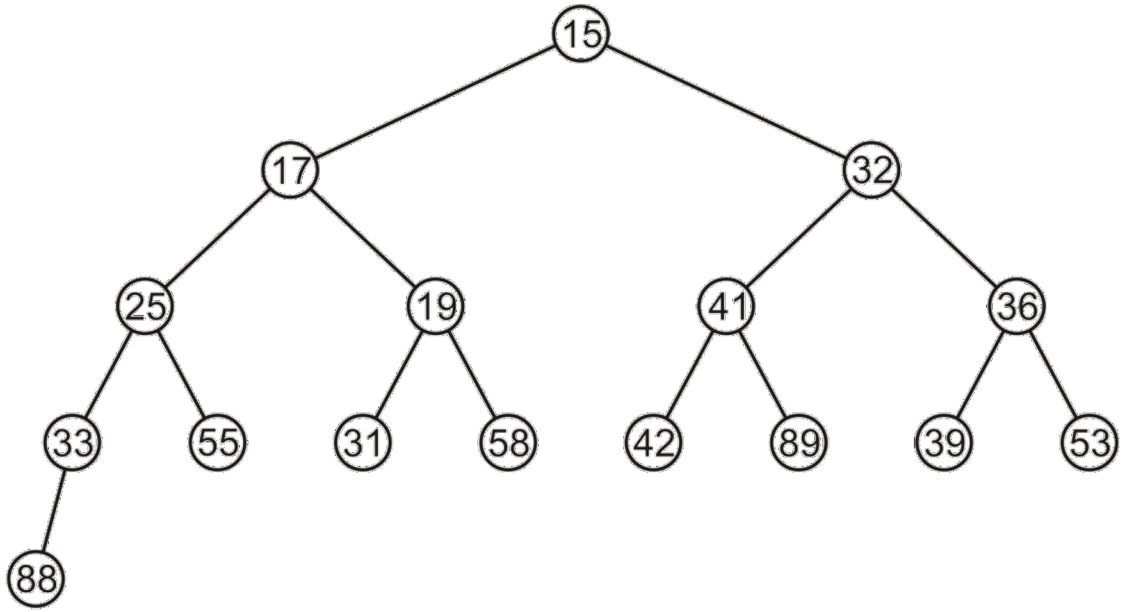
Now, percolate 36 down swapping it with the smallest of its children

- We halt when both children are larger 12



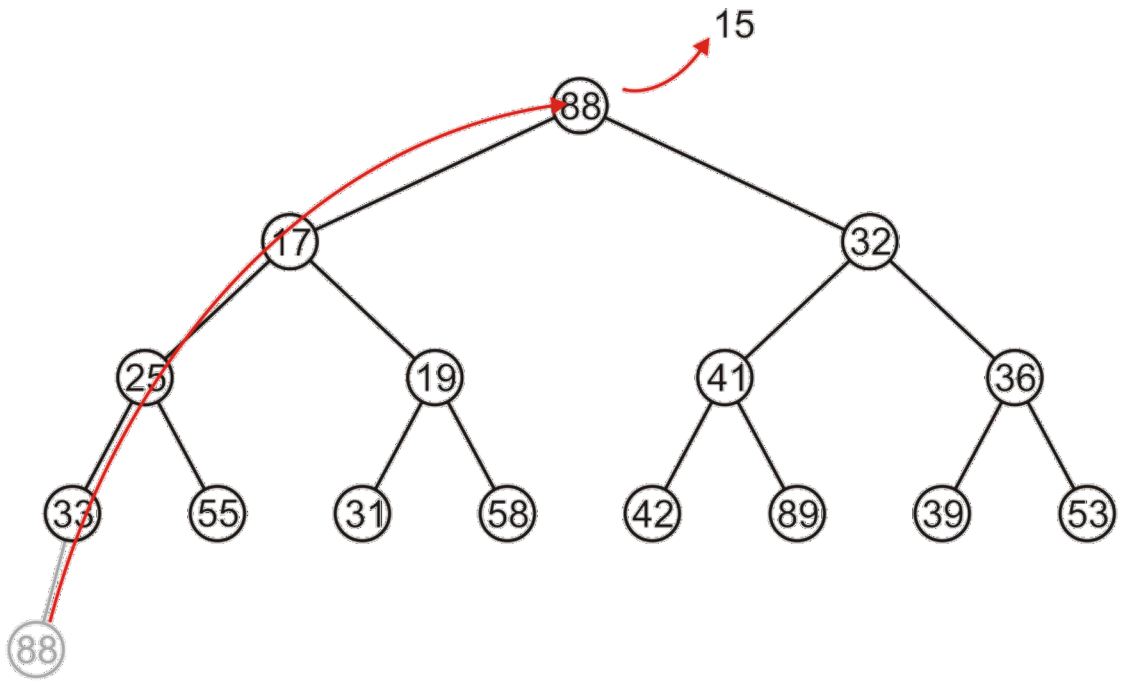
Complete Trees: Pop

The resulting tree is now still a complete tree:



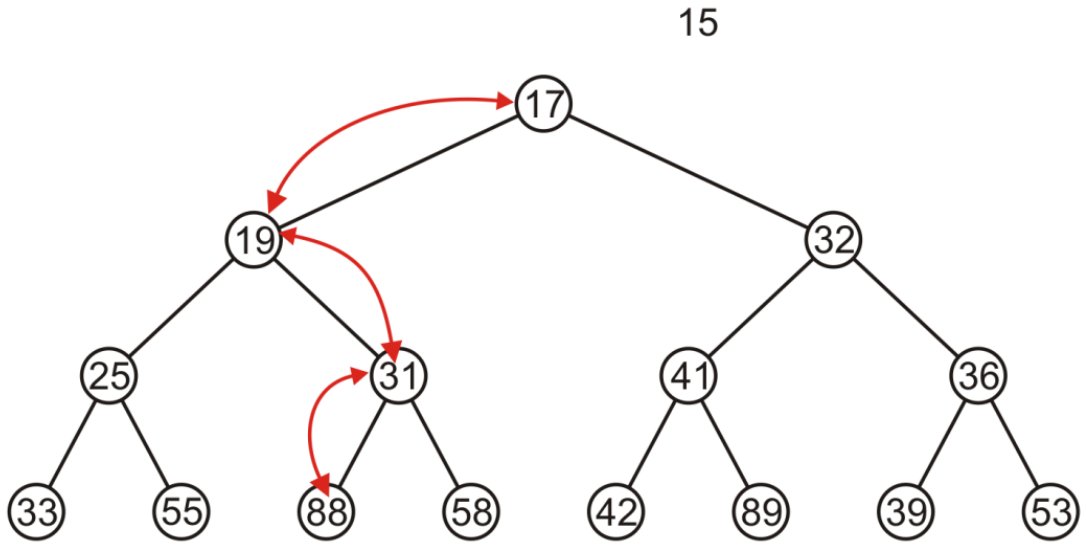
Complete Trees: Pop

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



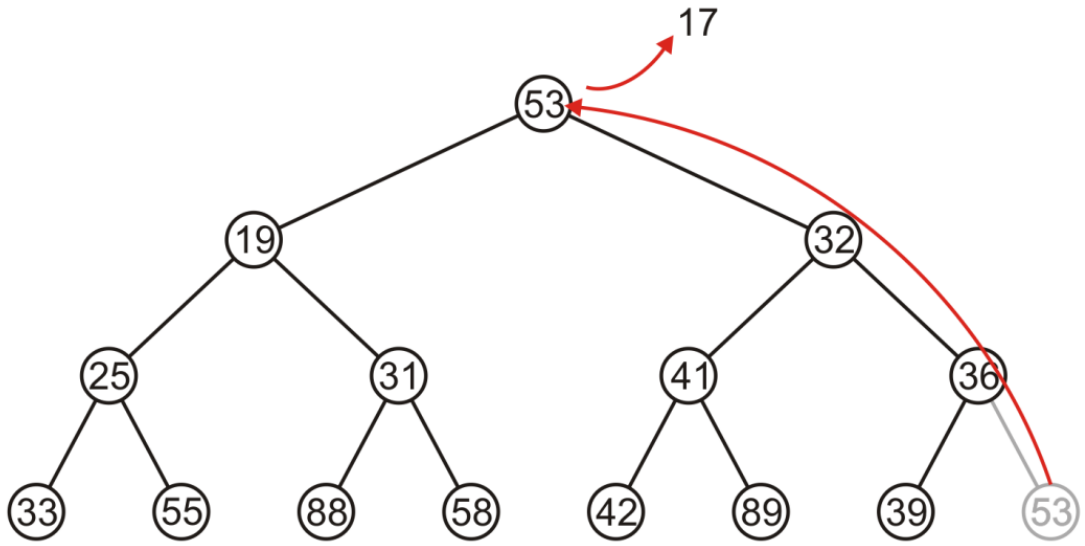
Complete Trees: Pop

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



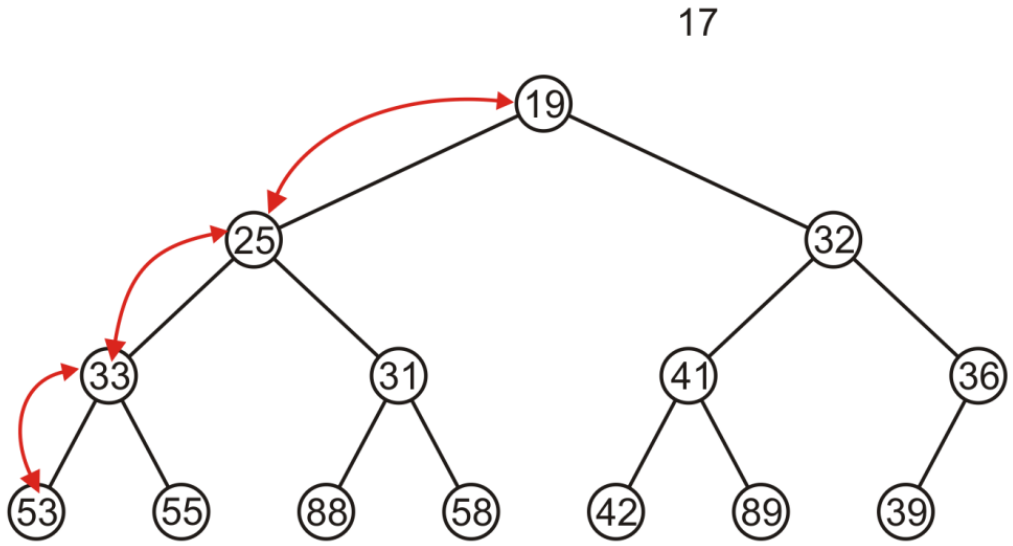
Complete Trees: Pop

In popping 17, 53 is moved to the top



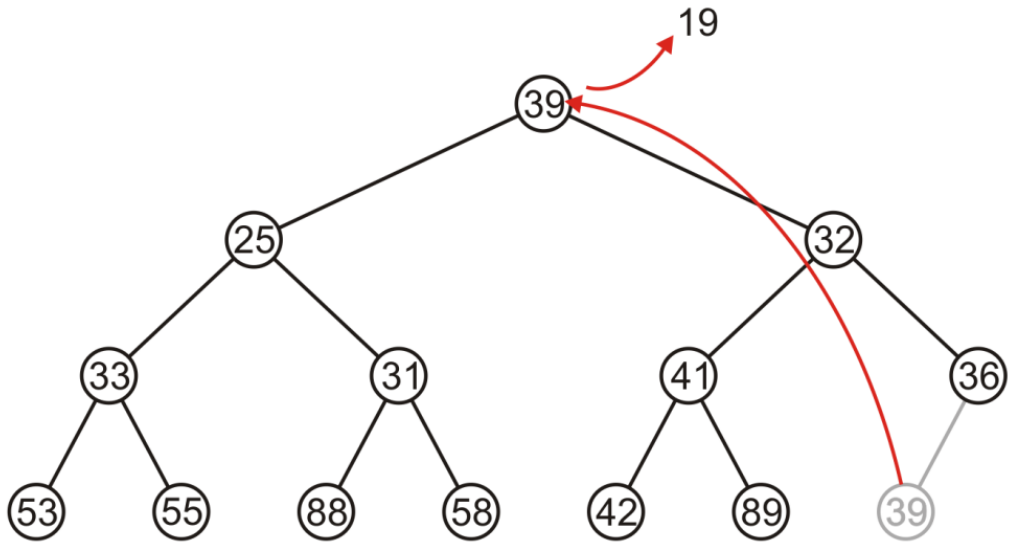
Complete Trees: Pop

And percolated down, again to the deepest level



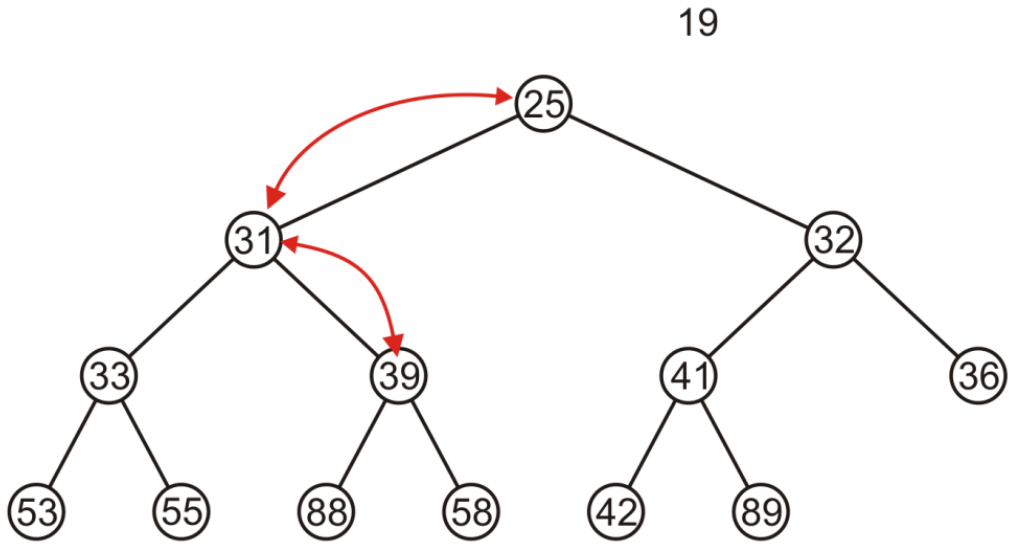
Complete Trees: Pop

Popping 19 copies up 39



Complete Trees: Pop

Which is then percolated down to the second deepest level



Complete Tree

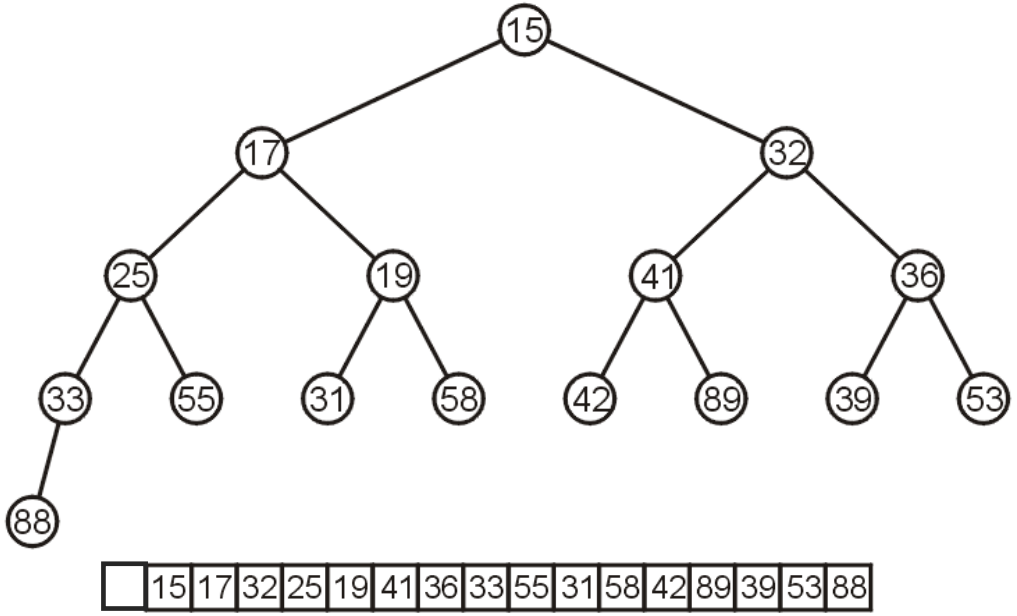
Therefore, we can maintain the complete-tree shape of a heap

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

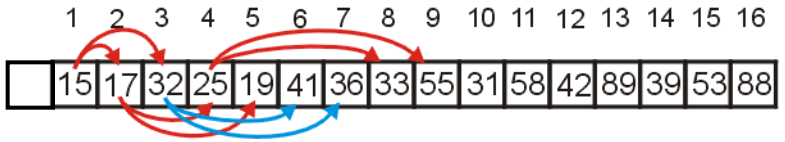
For the heap



a breadth-first traversal yields:

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 $k/2$
- the children are at $2k$ and $2k+1$

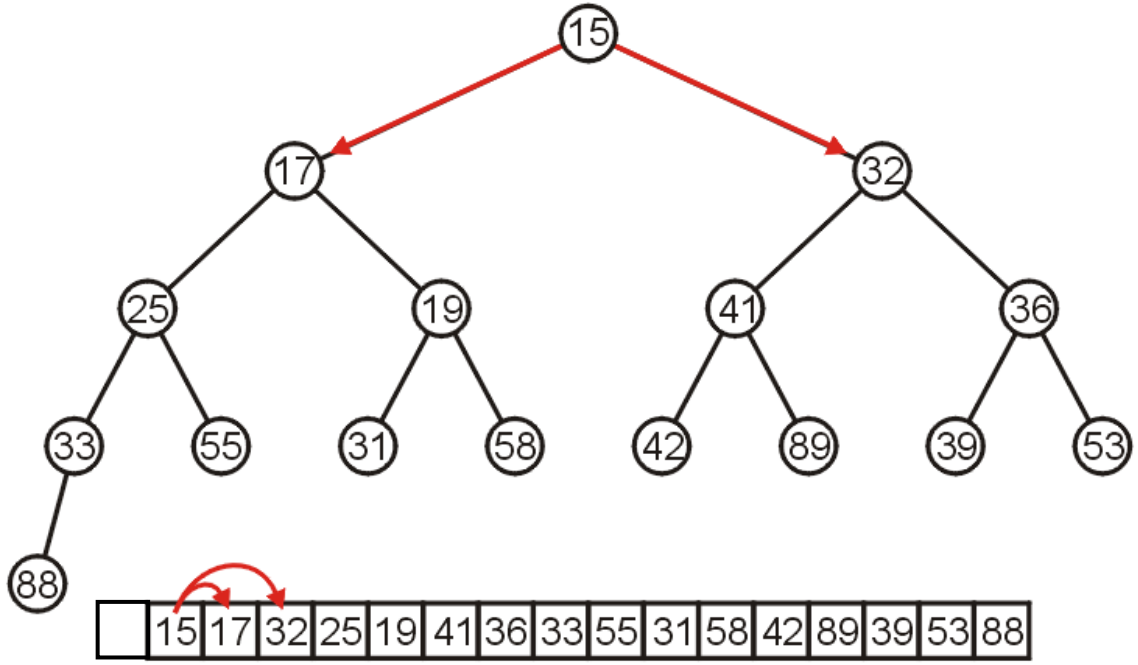
```
parent = k >> 1;

left_child = k << 1;
right_child = left_child | 1;
```

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

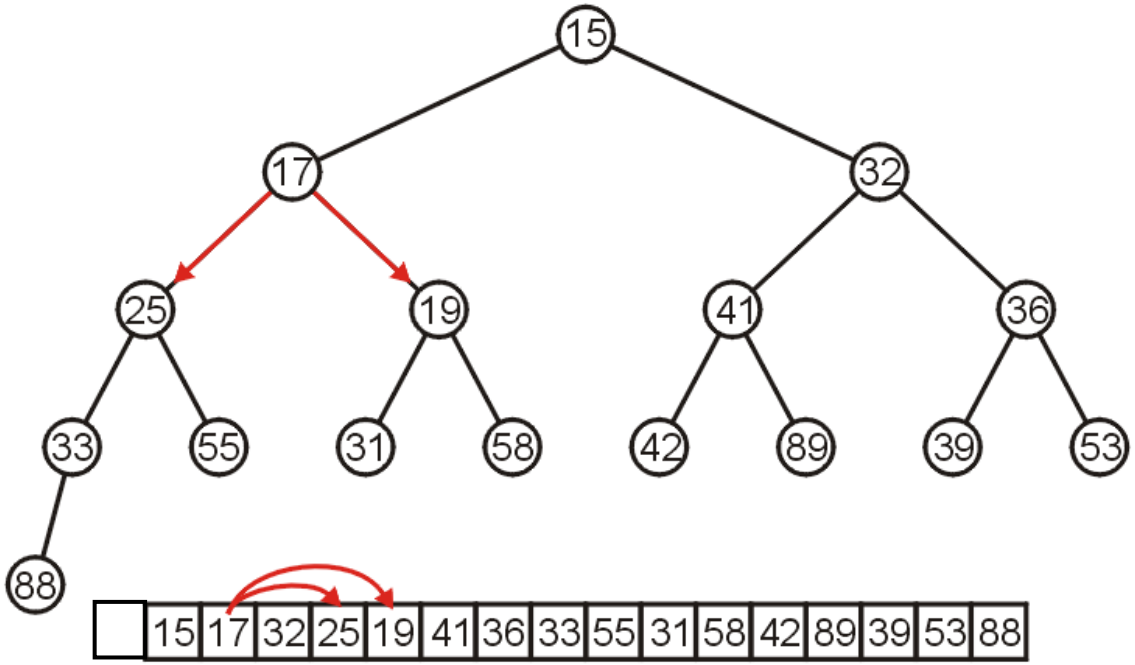
Array Implementation

The children of 15 are 17 and 32:



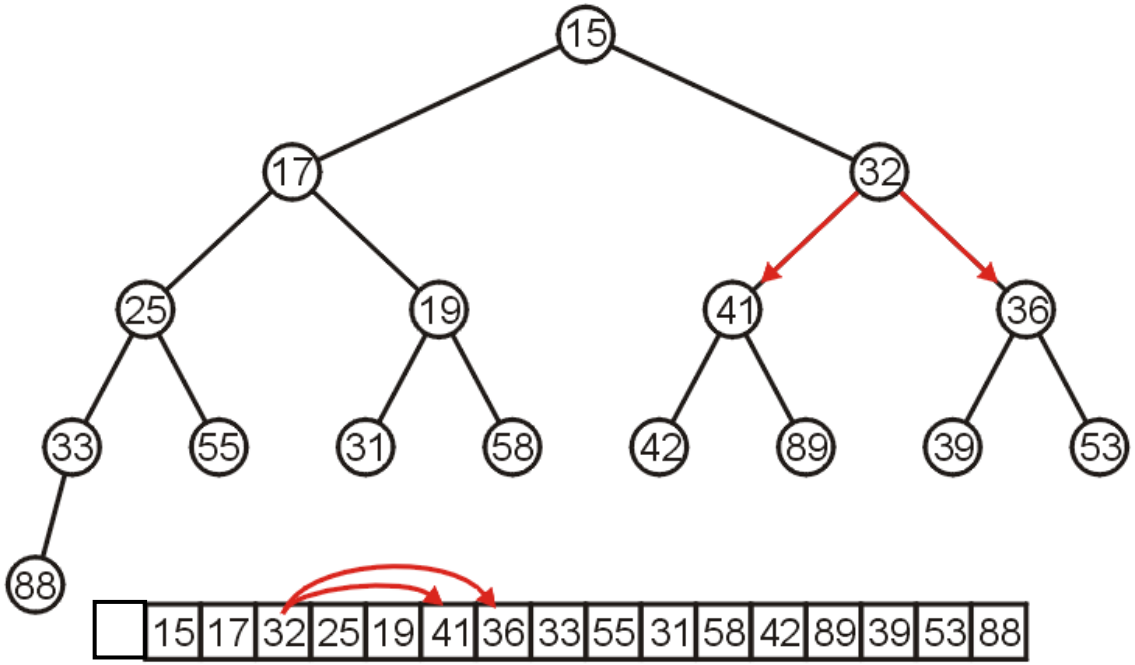
Array Implementation

The children of 17 are 25 and 19:



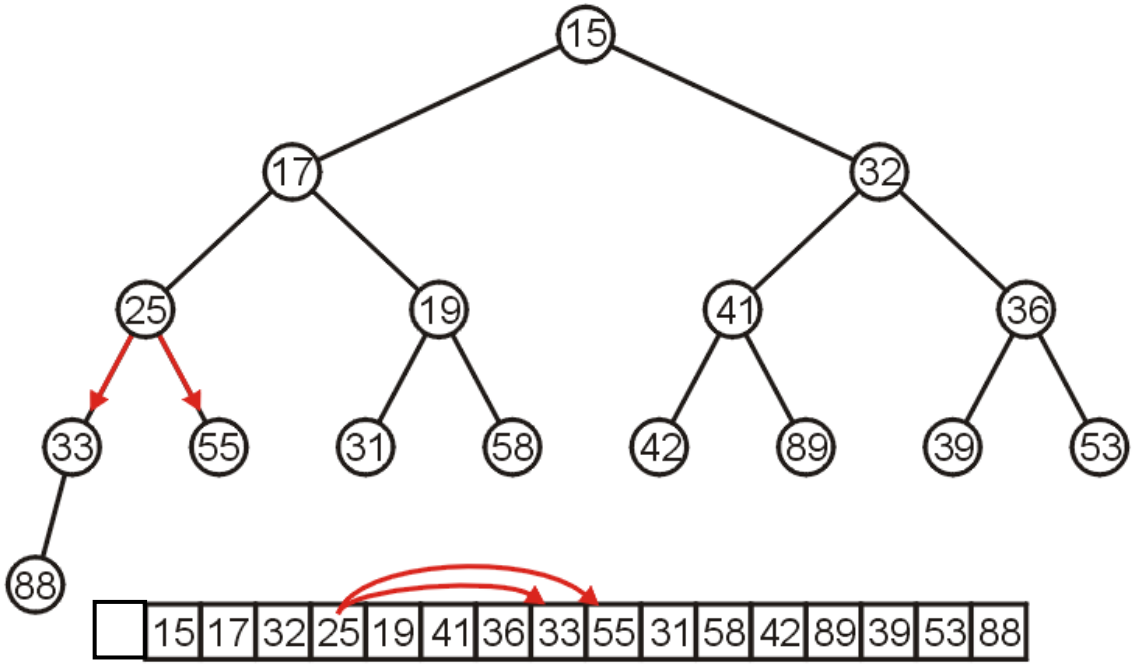
Array Implementation

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`**

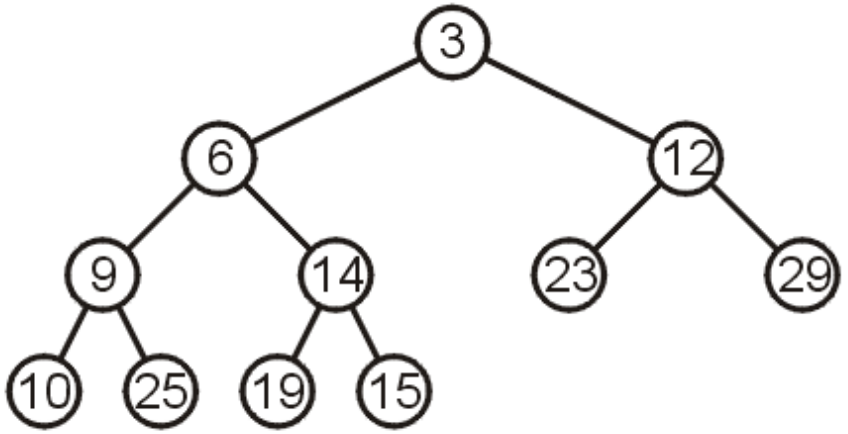
We compare the item at location **`posn`** with the item at **`posn/2`**

If they are out of order

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

Array Implementation

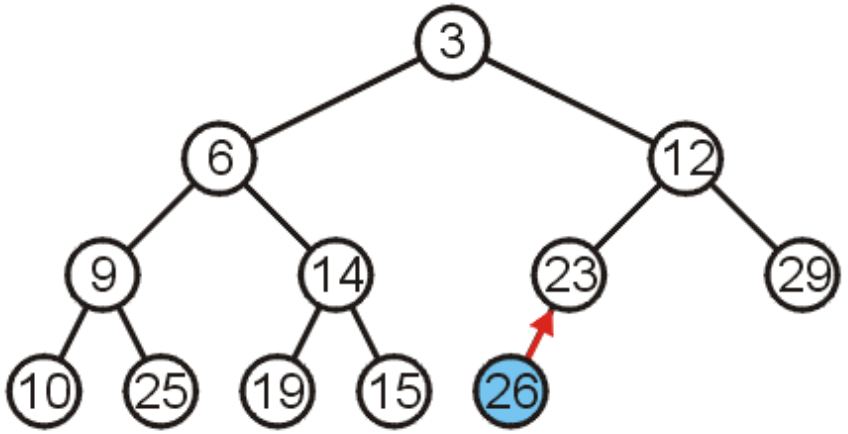
Consider the following heap, both as a tree and in its array representation

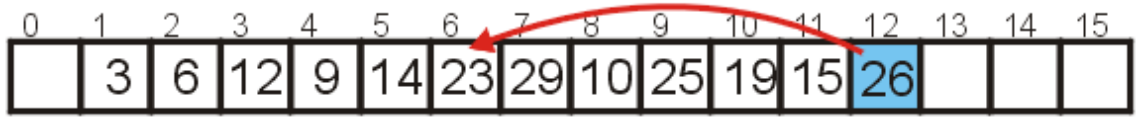


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	12	9	14	23	29	10	25	19	15				

Array Implementation: Push

Inserting 26 requires no changes





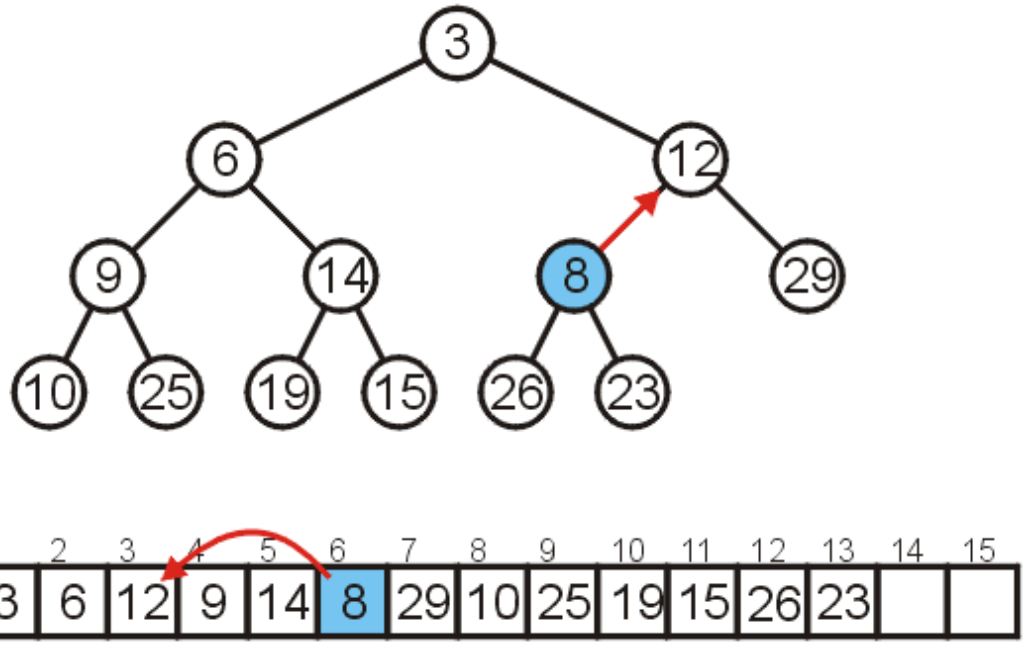
Array Implementation: Push

Inserting 8 requires a few percolations:

- Swap 8 and 23

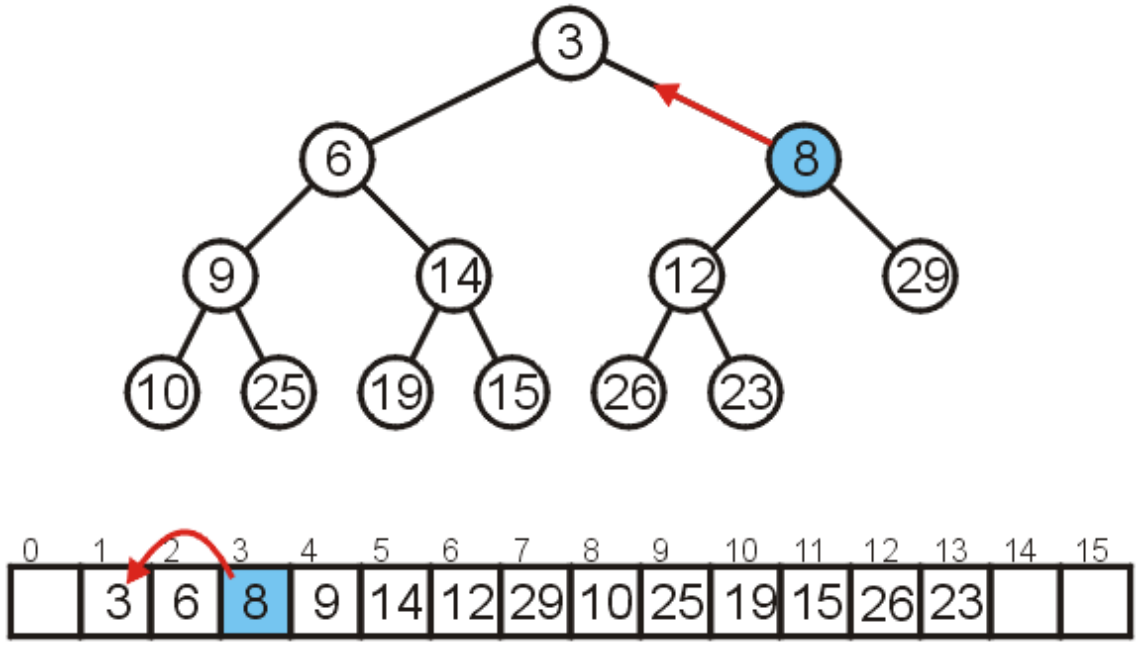
Array Implementation: Push

Swap 8 and 12



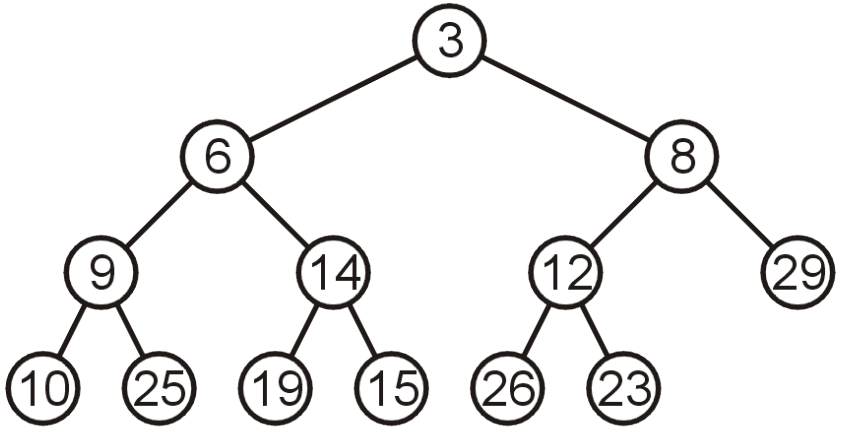
Array Implementation: Push

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



Array Implementation: Pop

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

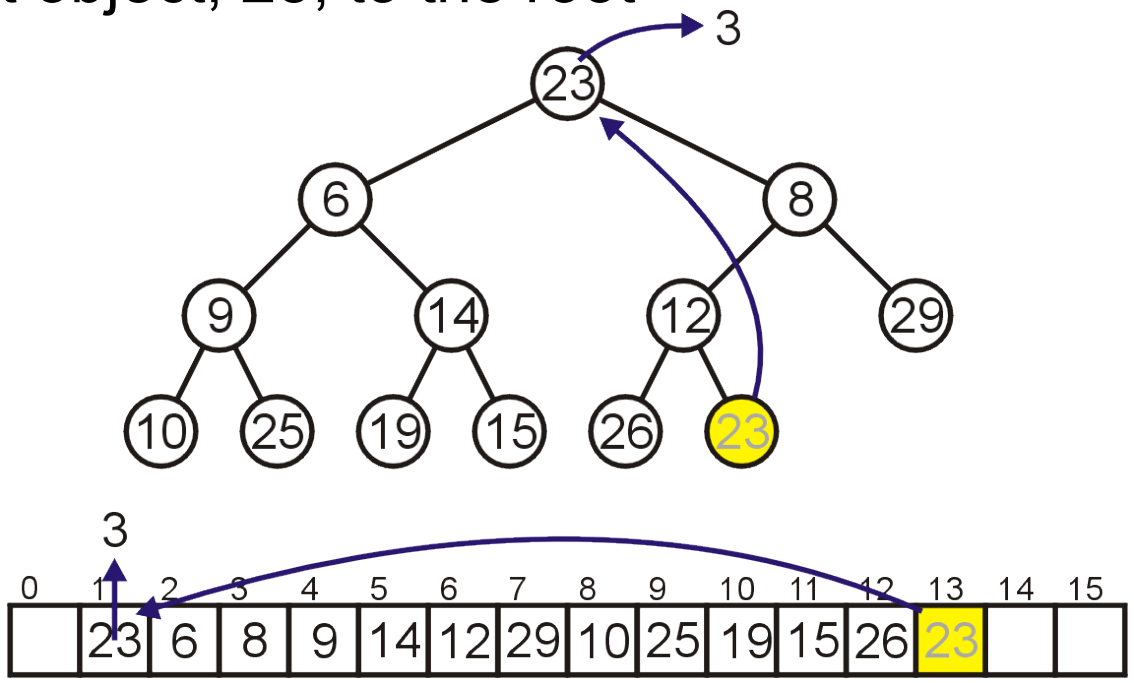


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	6	8	9	14	12	29	10	25	19	15	26	23		

Array Implementation: Pop

Instead, consider this strategy:

- Copy the last object, 23, to the root

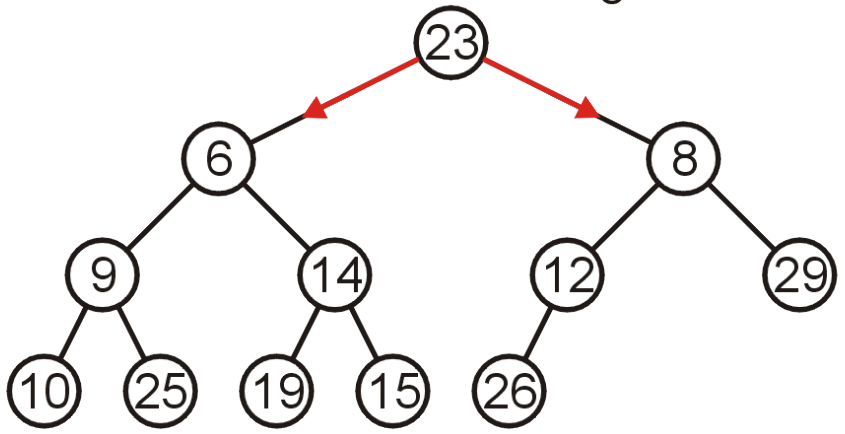


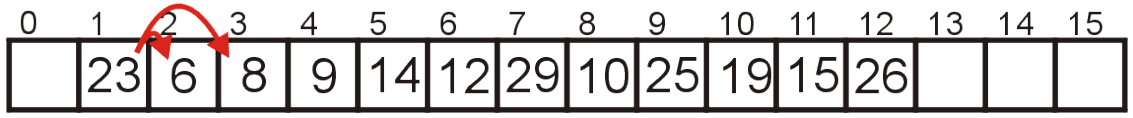
Array Implementation: Pop

Now percolate down

Compare Node 1 with its children: Nodes 2 and 3

- Swap 23 and 6

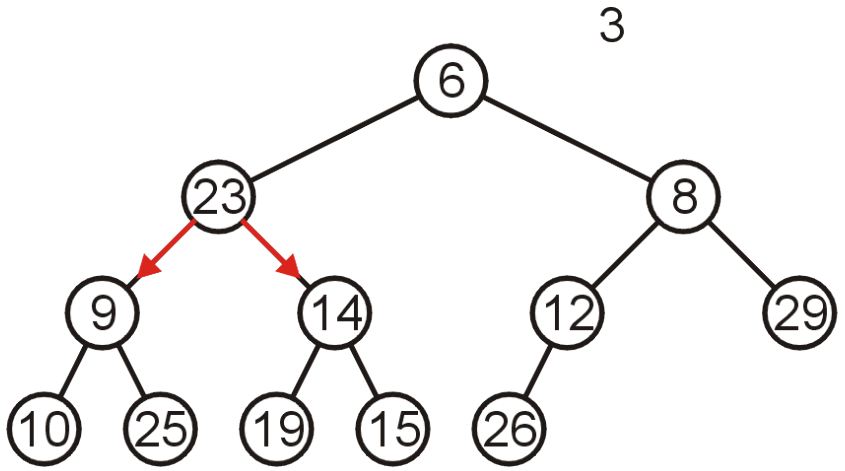


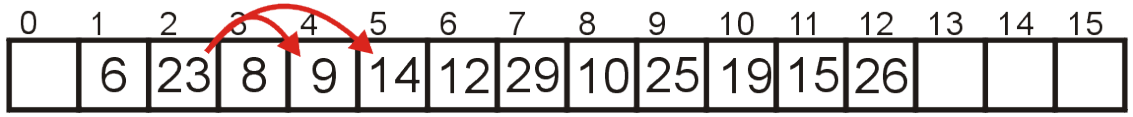


Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

- Swap 23 and 9

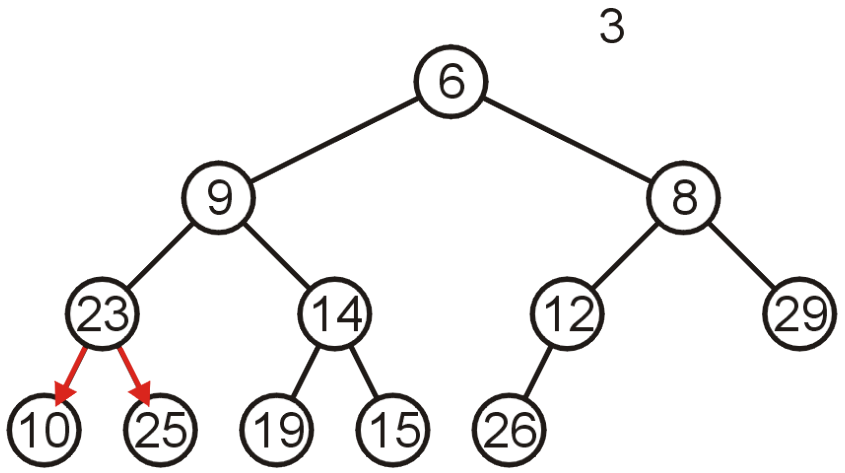


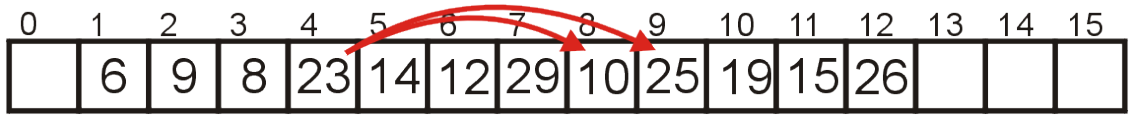


Array Implementation: Pop

Compare Node 4 with its children: Nodes 8 and 9

- Swap 23 and 10

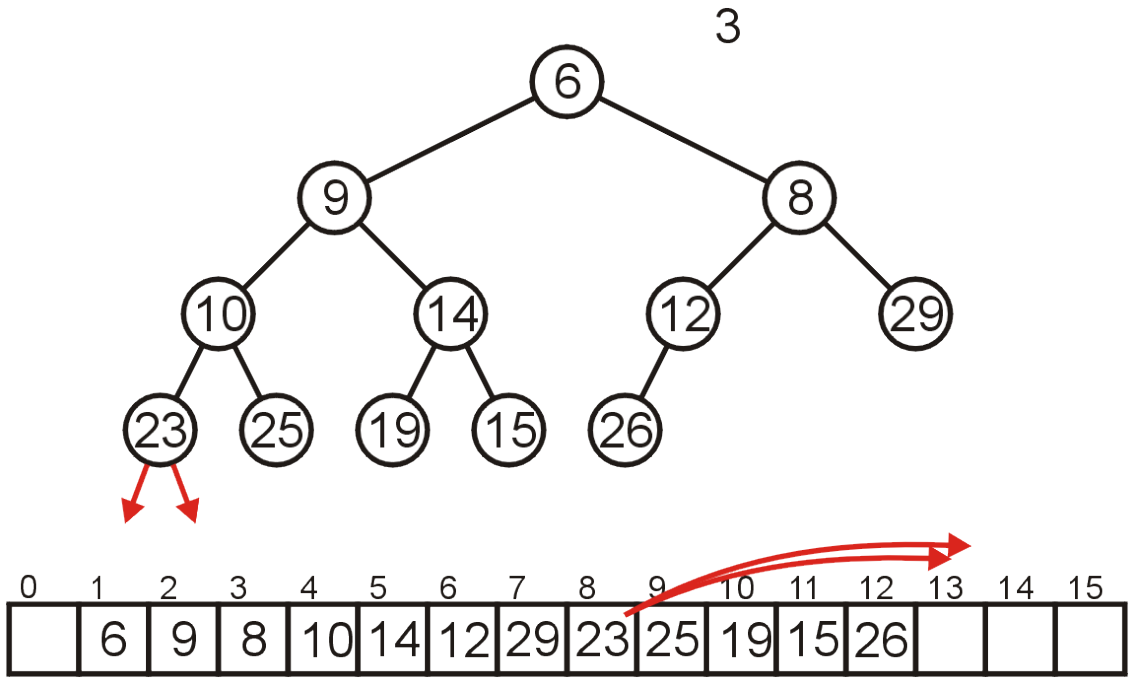




Array Implementation: Pop

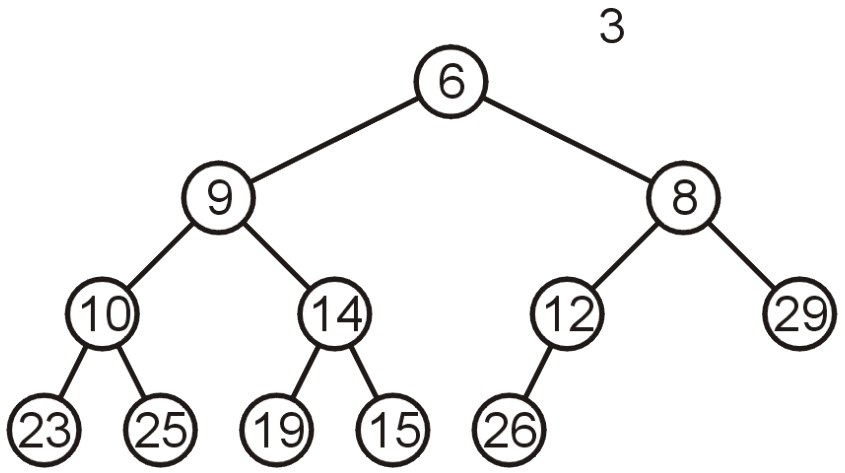
The children of Node 8 are beyond the end of the array:

- Stop



Array Implementation: Pop

The result is a binary min-heap

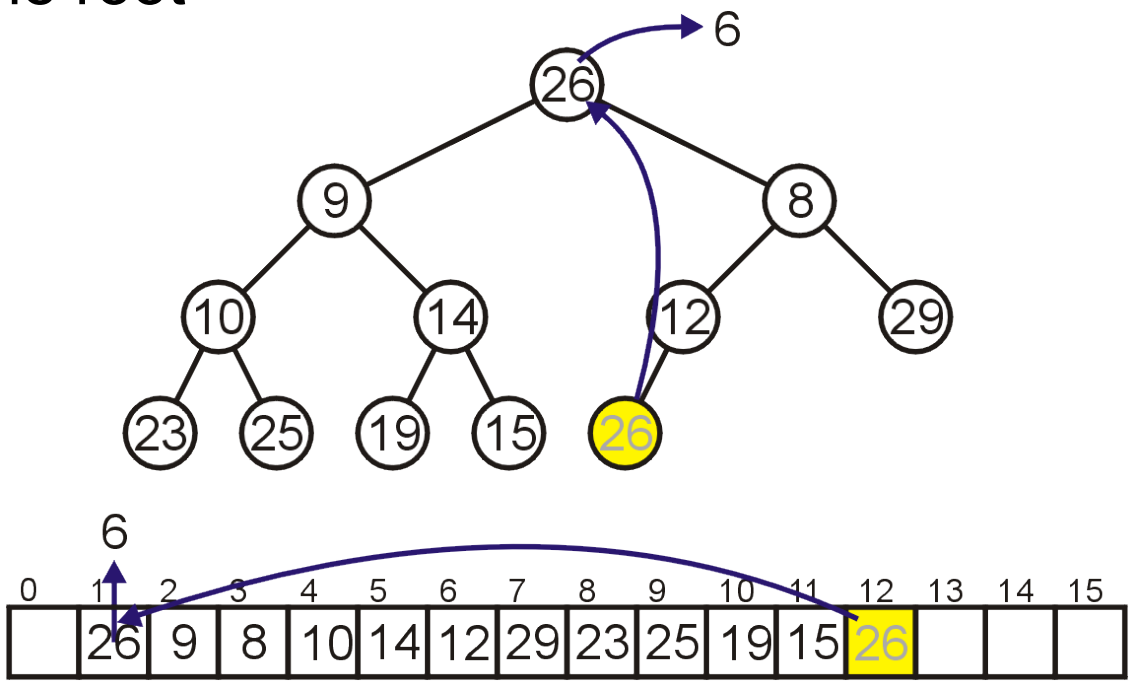


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	6	9	8	10	14	12	29	23	25	19	15	26			

Array Implementation: Pop

Dequeuing the minimum again:

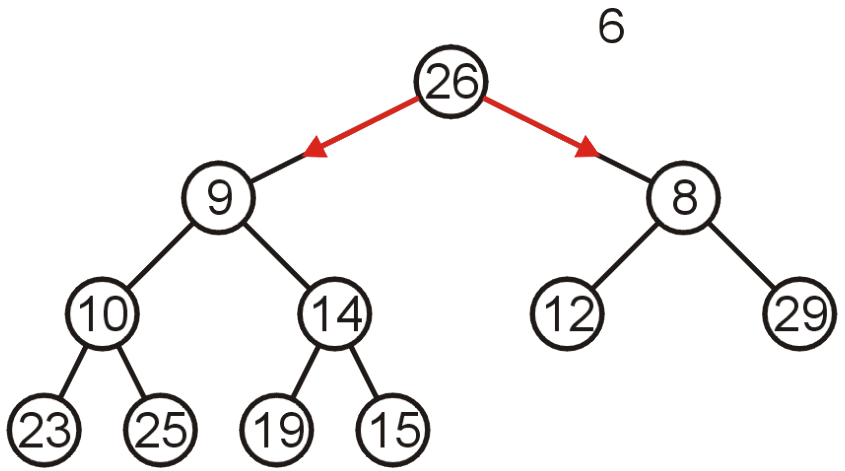
- Copy 26 to the root

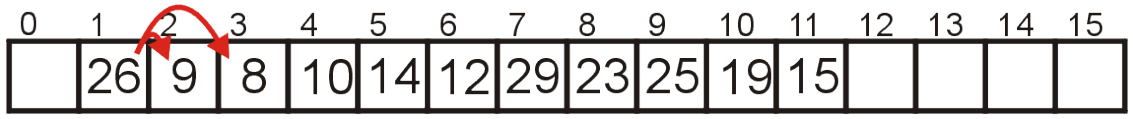


Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

- Swap 26 and 8

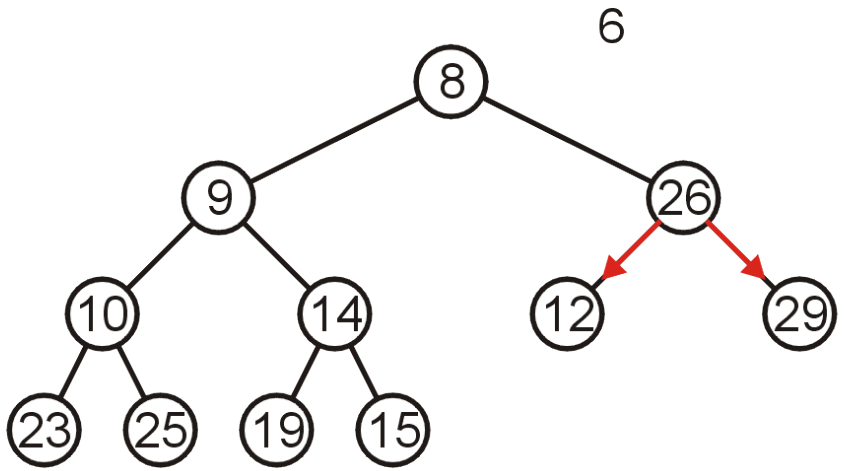


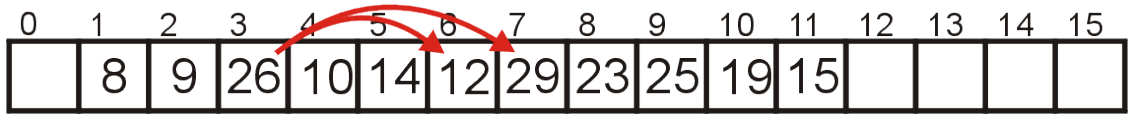


Array Implementation: Pop

Compare Node 3 with its children: Nodes 6 and 7

- Swap 26 and 12

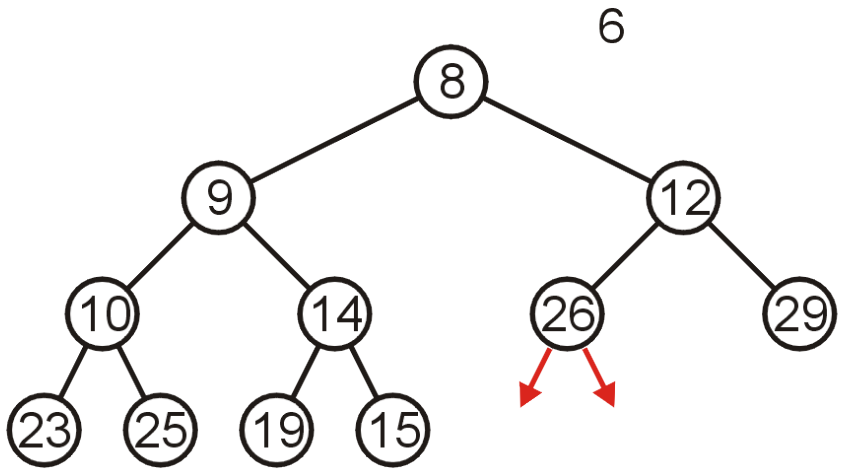


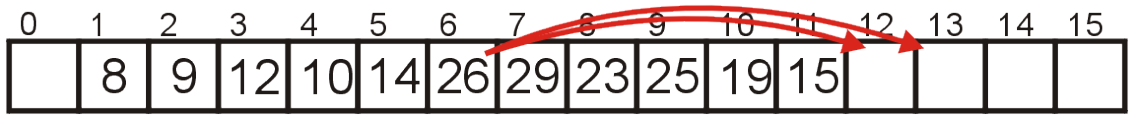


Array Implementation: Pop

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

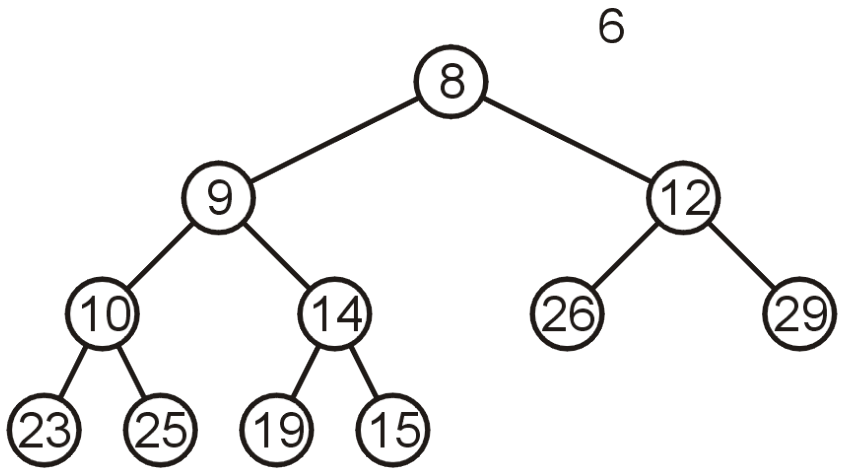
- Currently, `count == 11`





Array Implementation: Pop

The result is a min-heap

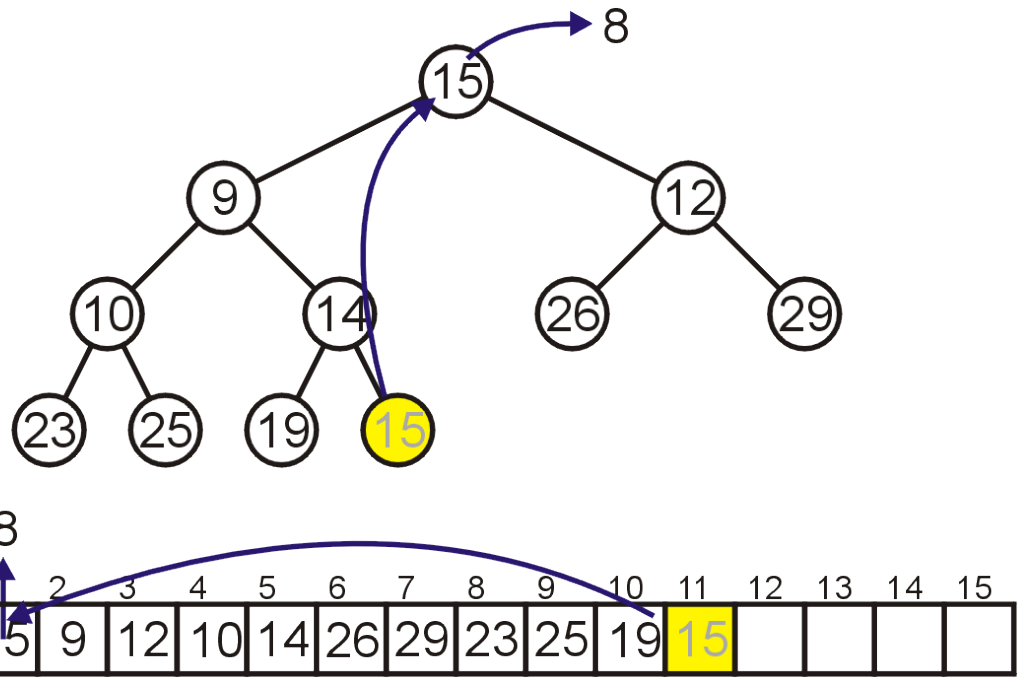


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	8	9	12	10	14	26	29	23	25	19	15				

Array Implementation: Pop

Dequeuing the minimum a third time:

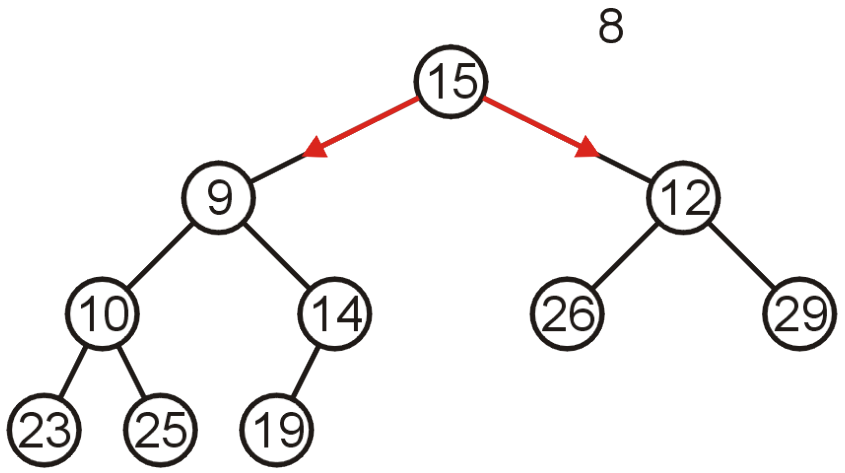
- Copy 15 to the root

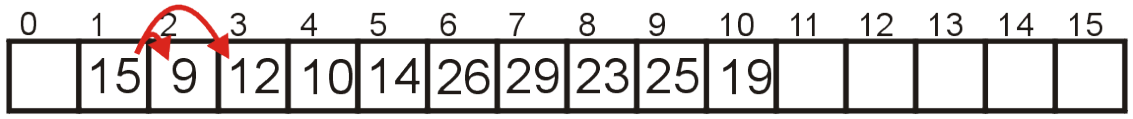


Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

- Swap 15 and 9

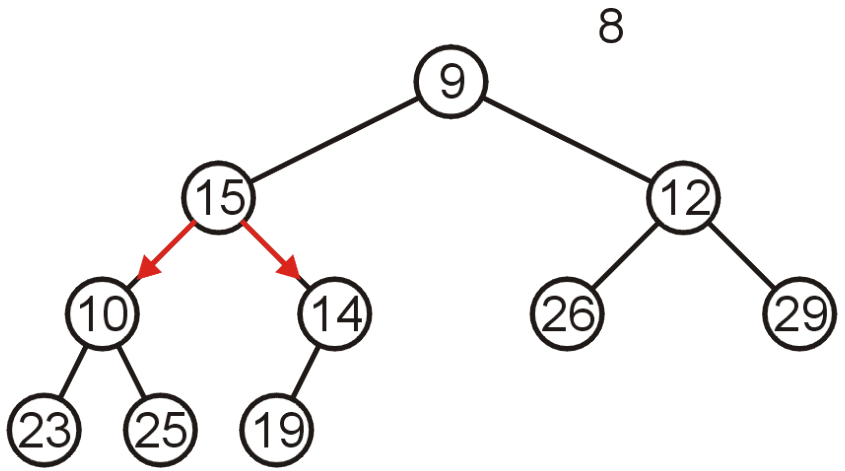


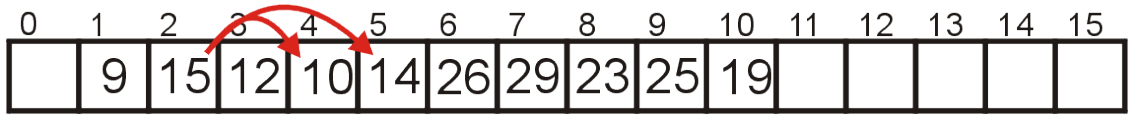


Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

- Swap 15 and 10

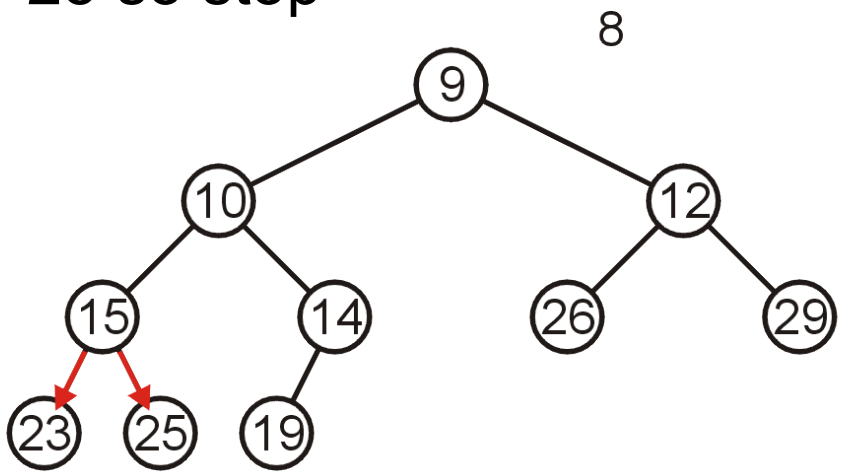


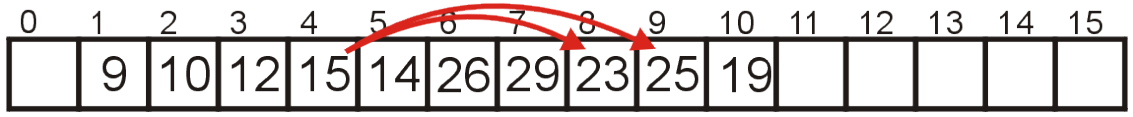


Array Implementation: Pop

Compare Node 4 with its children: Nodes 8 and 9

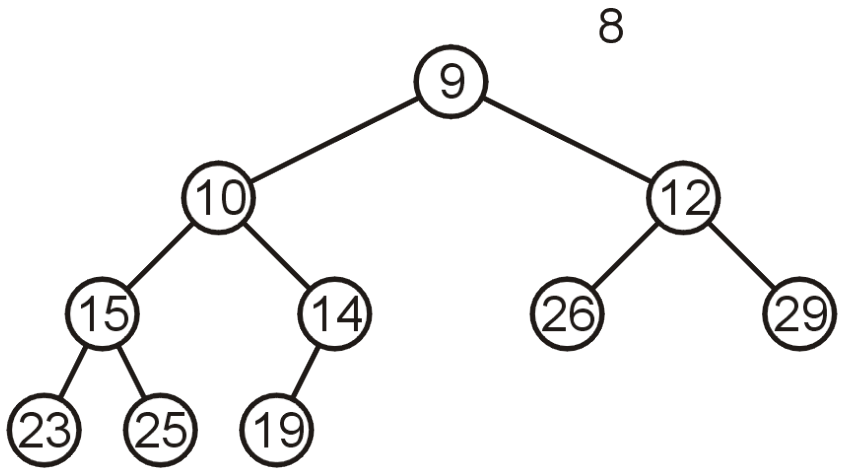
- 15 < 23 and 15 < 25 so stop





Array Implementation: Pop

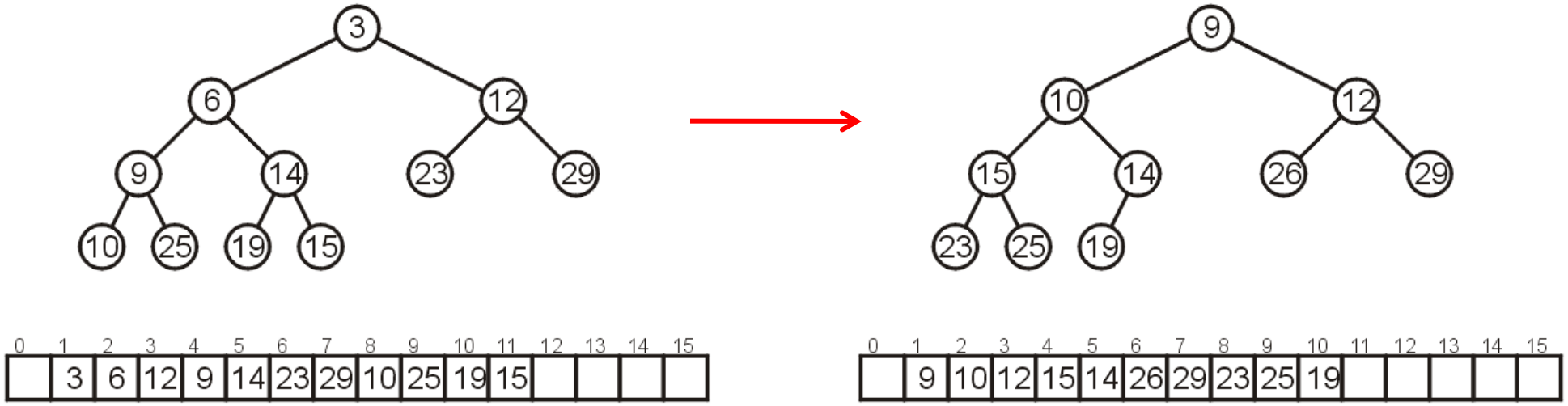
The result is a properly formed binary min-heap



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					

Array Implementation: Pop

After all our modifications, the final heap is



Run-time Analysis

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?

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 $\mathbf{O}(\ln(n))$? 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

$$\frac{1}{n}\sum_{k=0}^{h}(h-k)2^k = \frac{2^{h+1}-h-2}{n}$$

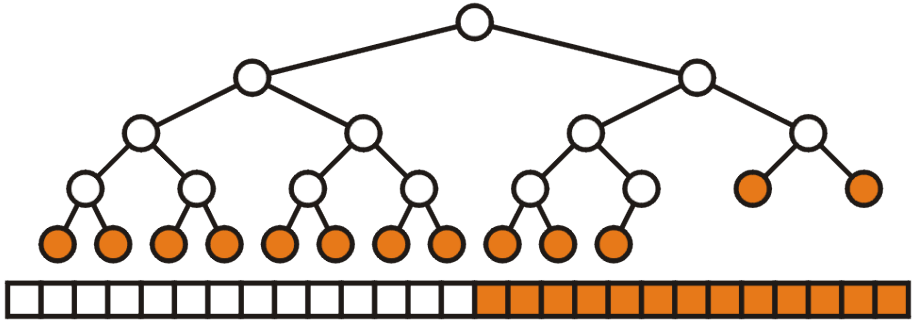
$$= \frac{n-h-1}{n} = \Theta(1)$$

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: $\mathbf{O}(n)$

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



Run-time Analysis

Thus, our grid of run times is given by:

	front	arbitrary	back
insert	$\mathbf{O}(\ln(n))^*$	$\mathbf{O}(1)$	$\mathbf{O}(1)$
access	$\mathbf{O}(1)$	$\mathbf{O}(n)$	$\mathbf{O}(n)$
delete	$\mathbf{O}(\ln(n))$	$\mathbf{O}(n)$	$\mathbf{O}(n)$

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 $\mathbf{O}(\ln(n))$
- If the objects are coming in order of priority, use a regular queue with swapping
- Merging two binary heaps of size n is a $\mathbf{\Theta}(n)$ operation

Run-time Analysis

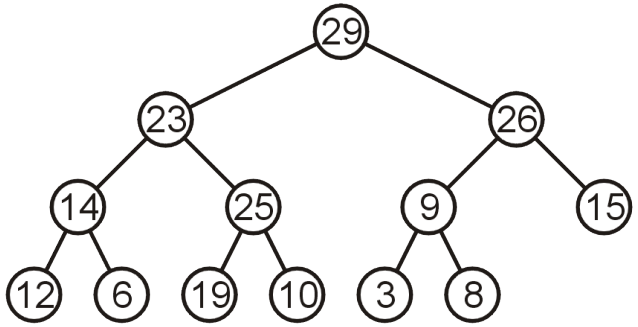
Other heaps have better run-time characteristics

- 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 $\Theta(1)$

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

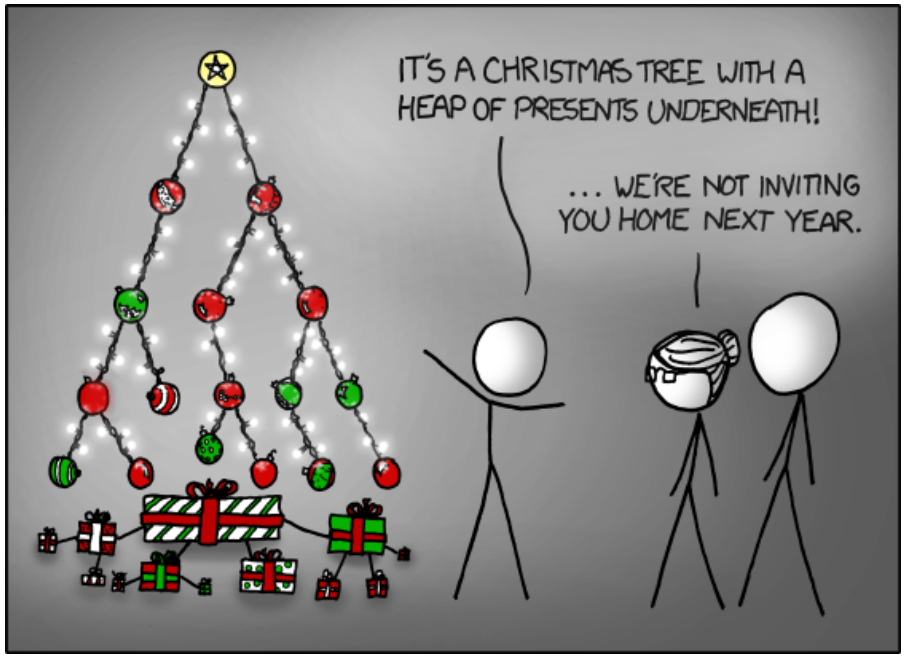
For example, the same data as before stored as a max-heap yields



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		

Example

Here we have a max-heap of presents under a red-green tree:

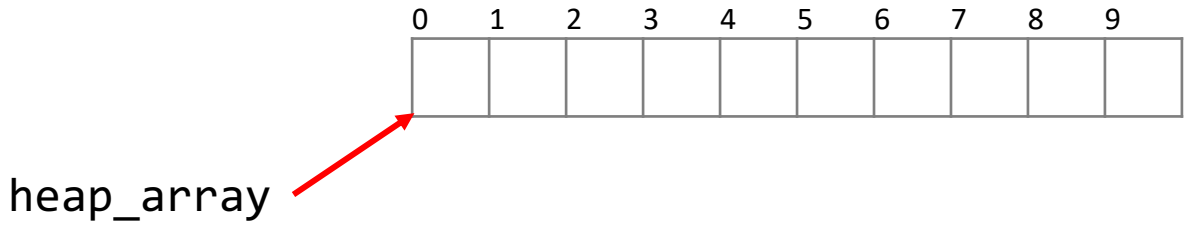


http://xkcd.com/835/

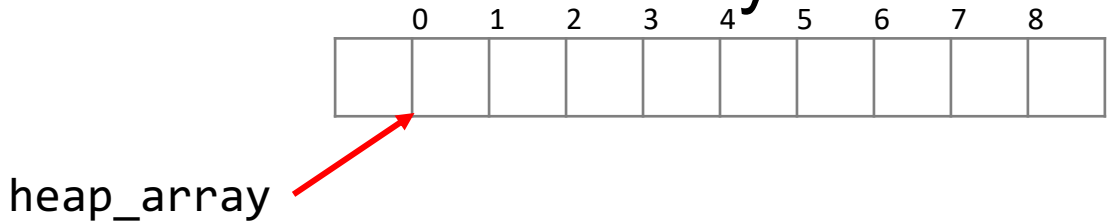
Memory allocation and pointer arithmetic

Do we really have to allocate one additional memory location for a binary tree-as-heap?

```
Type *heap_array = new Type[capacity() + 1];
```



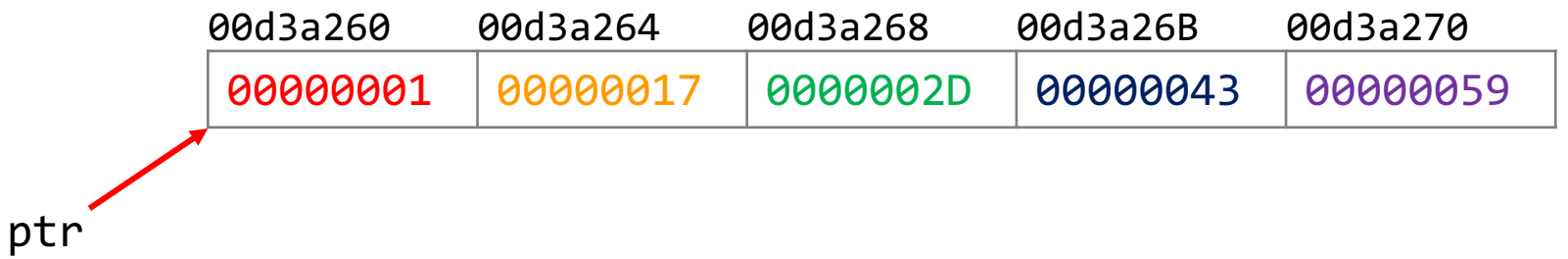
Could we not just allocate one less memory and point to the previous location in memory



Memory allocation and pointer arithmetic

To do this, we must understand pointer arithmetic:

```
int *ptr = new int[5] = {1, 23, 45, 67, 89};
std::cout << ptr << std::endl;
std::cout << *ptr << std::endl;
```



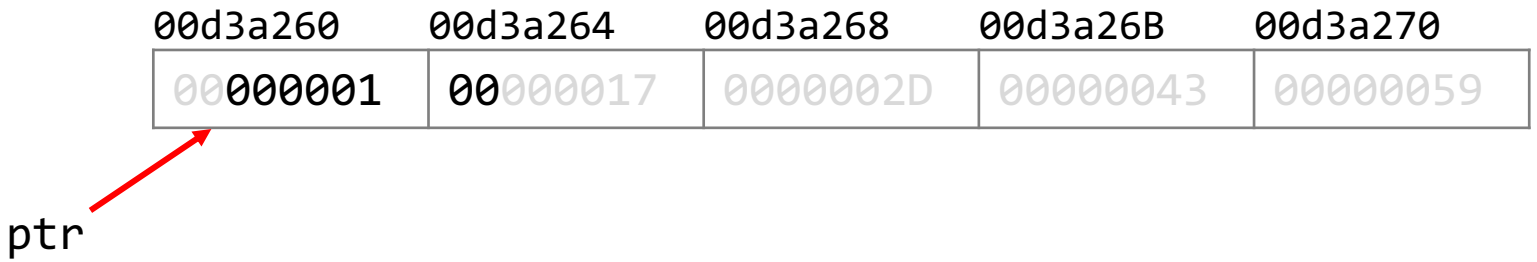
What is the output of?

```
std::cout << (ptr + 1) << std::endl;
std::cout << *(ptr + 1) << std::endl;
```

Memory allocation and pointer arithmetic

Just adding one to the address would be, in almost all cases, useless

- Assuming big endian, this would have a value 256
- If this was little endian, it would be even more bizarre…



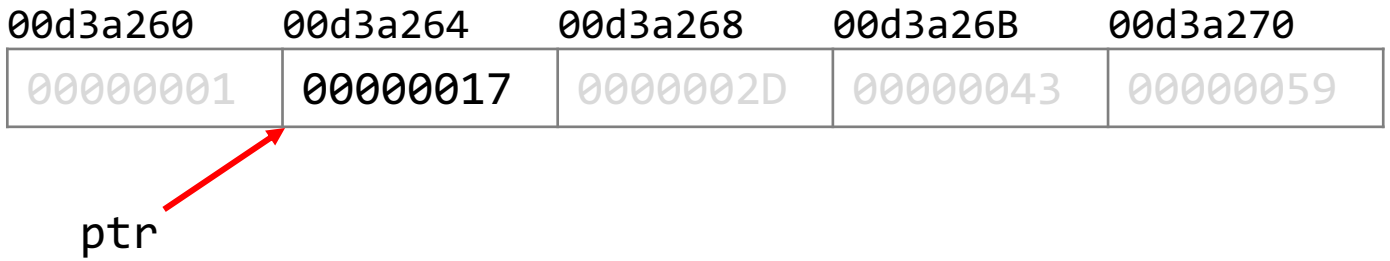
What is the output of?

```
std::cout << (ptr + 1) << std::endl;
std::cout << *(ptr + 1) << std::endl;
```

Memory allocation and pointer arithmetic

Instead, C and C++ add as many bytes as the size of the object being pointed to

- In the cases of `int`, `sizeof( int ) == 4` on most 32-bit machines
- The output is 23



What is the output of?

```
std::cout << (ptr + 1) << std::endl;
std::cout << *(ptr + 1) << std::endl;
```

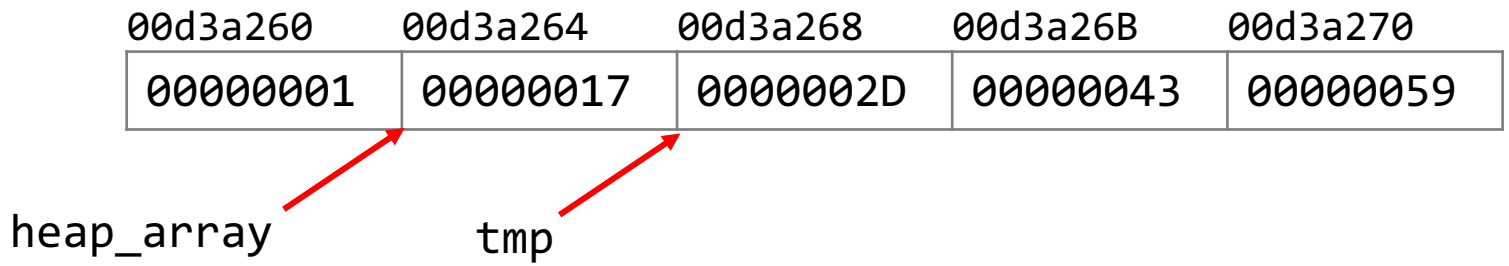
Memory allocation and pointer arithmetic

Essentially, these two statements are identical:

```
std::cout << ptr[i] << std::endl;
std::cout << *(ptr + i) << std::endl;
```

Now you can do the following:

```
Type *tmp = new Type[capacity()];
Type *heap_array = tmp - 1;
```



Now, `heap_array[1];` and `tmp[0];` both point to the same memory location

Memory allocation and pointer arithmetic

Issues:

- Never access or modify the contents of `heap_array[0]`
- When you deallocate memory, you must point to the original address returned by `new`:

```
delete [] (heap_array + 1);
```

- Pointer arithmetic is not for the faint of heart but it is fun; for example:

```
int array[N];
int *ptr = array;

// Print all the entries of the array
for ( int i = 0; i < N; ++i ) {
    std::cout << *(ptr++) << std::endl;
}
```

Priority Queues

Now, does using a heap ensure that that object in the heap which:

- 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 (colour indicates order):

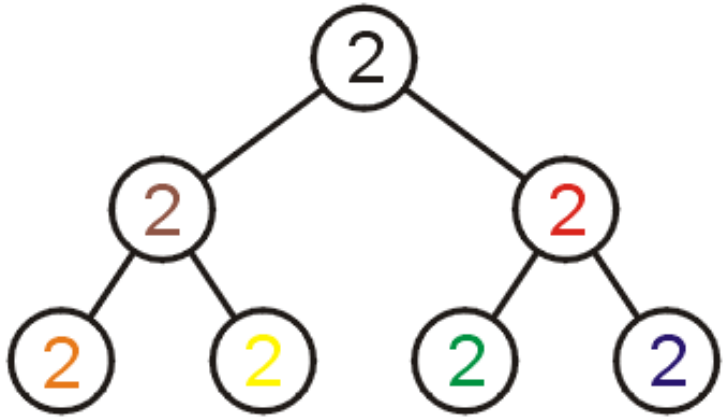
2, 2, 2, 2, 2, 2, 2

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:

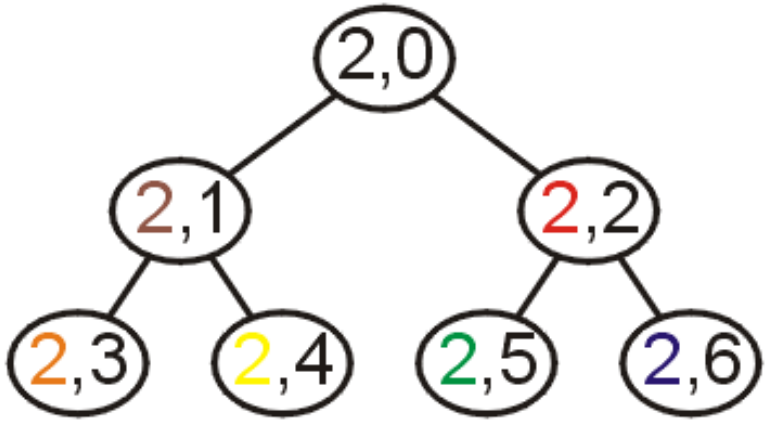
- Come up with an algorithm which removes all seven objects in the original order

Lexicographical Ordering

A better solution is to modify the priority:

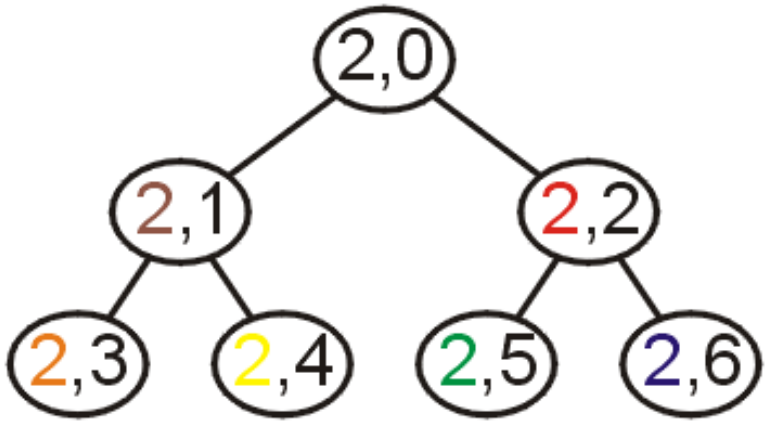
- Track the number of insertions with a counter k (initially 0)
- For each insertion with priority n, create a hybrid priority (n, k) where:

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



Priority Queues

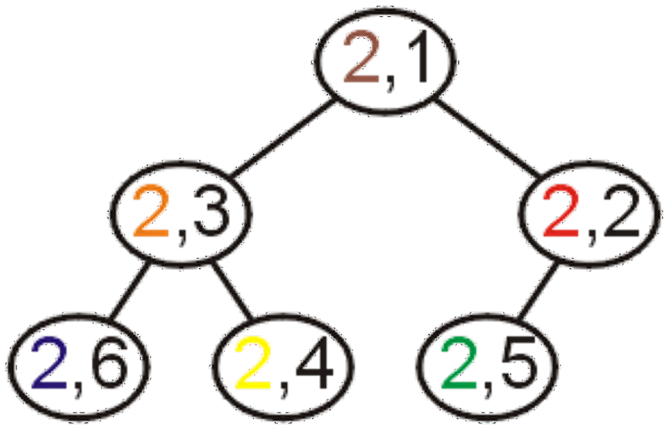
Removing the objects would be in the following order:



Priority Queues

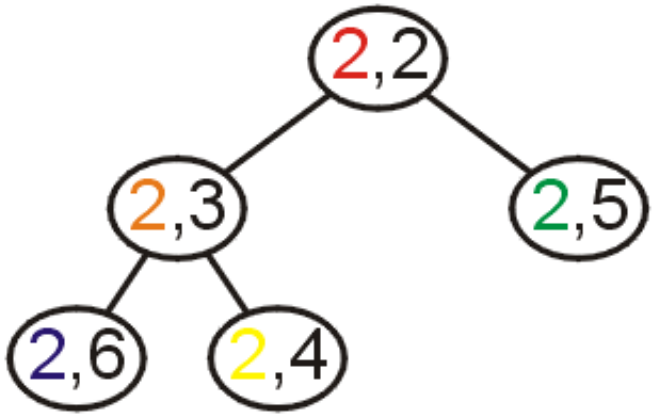
Popped: 2

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



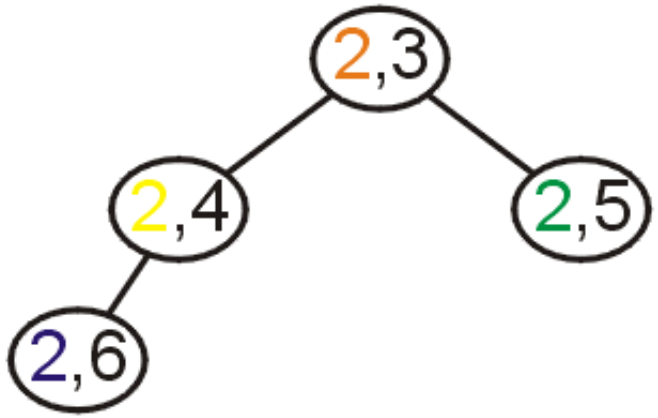
Priority Queues

Removing the objects would be in the following order:



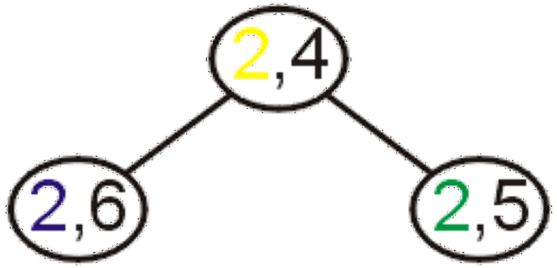
Priority Queues

Removing the objects would be in the following order:



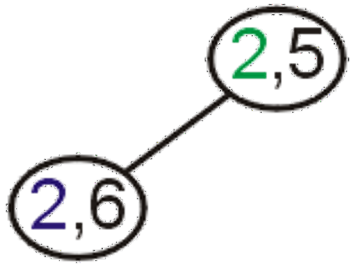
Priority Queues

Removing the objects would be in the following order:



Priority Queues

Removing the objects would be in the following order:



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
- The use of a lexicographical ordering