COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides from “Algorithms and Data Structures”
by Douglas Wilhelm Harder

Priority Queues

Background

We have discussed Abstract Lists with explicit linear orders
- Arrays, linked lists

We saw three cases which restricted the operations:
- Stacks, queues, deques

Following this, we looked at binary search trees for storing implicit linear orders:
- Run times were generally $\Theta(\ln(n))$

We will now look at a restriction on an implicit linear ordering:
- Priority queues

Priority Queue

3 jobs have been submitted to a printer in the order A, B, C.

Sizes: Job A – 100 pages
Job B – 10 pages
Job C -- 1 page

Average waiting time with FIFO service:

(100+110+111) / 3 = 107 time units

Average waiting time for shortest-job-first service:

(1+11+111) / 3 = 41 time units

Need to have a queue which does insert and delete item with *the highest priority*

Priority Queue

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 new objects in the order of arrival

Lexicographical Priority

Priority may 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)

Implementations

Linked Lists: (array is slightly different but almost the same)

-Insert - $\Theta(1)$

-Find the minimum - $\Theta(n)$

-Remove - $\Theta(n)$

Binary search trees (like AVL tree):

-Insert - $\Theta(\log(n))$

-Find the minimum - $\Theta(\log(n))$

-Remove - $\Theta(\log(n))$ (but it does many other operations)

Binary Heap

We are going to use a new data structure *Binary Heap*

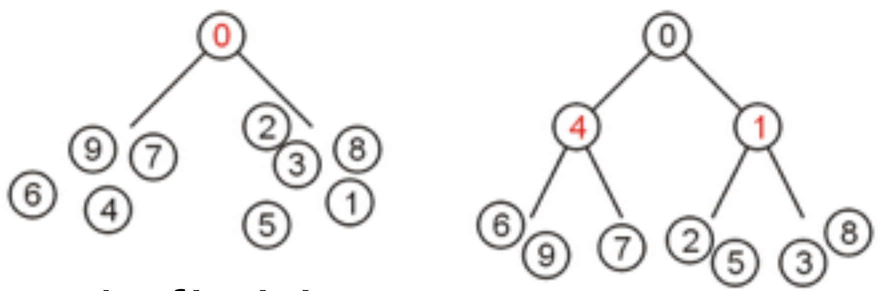
- A binary tree in which each node has a higher priority than its children
- Min-Heap: the smaller the higher priority
- Max-Heap: the larger the higher priority

Binary Heap

Definition

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

- The key (element, comparable item) 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
- The value at each node is less than or equal to that of all its descendants.

Definition

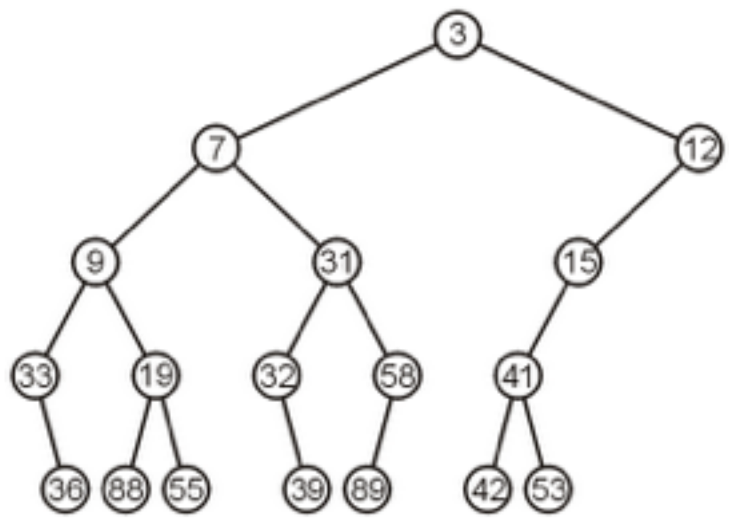
Important:

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

Failing to understand this is the greatest mistake students make about heaps

Example

This is a binary min-heap:



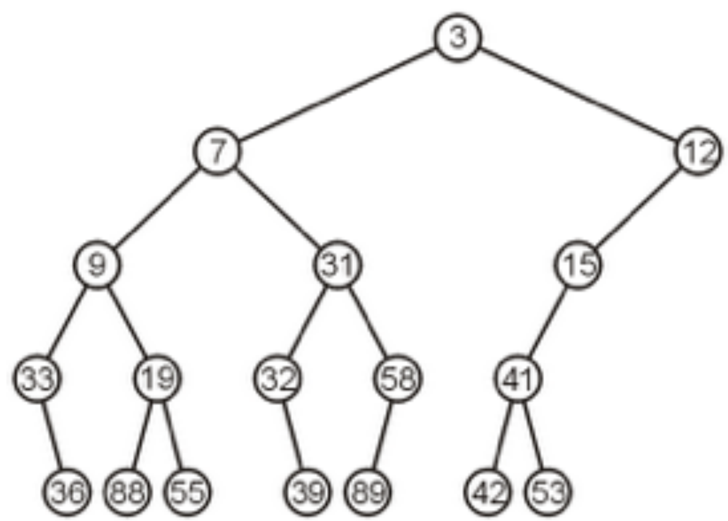
Operations

We will consider three operations:

- Top
- Pop
- Push

Example

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



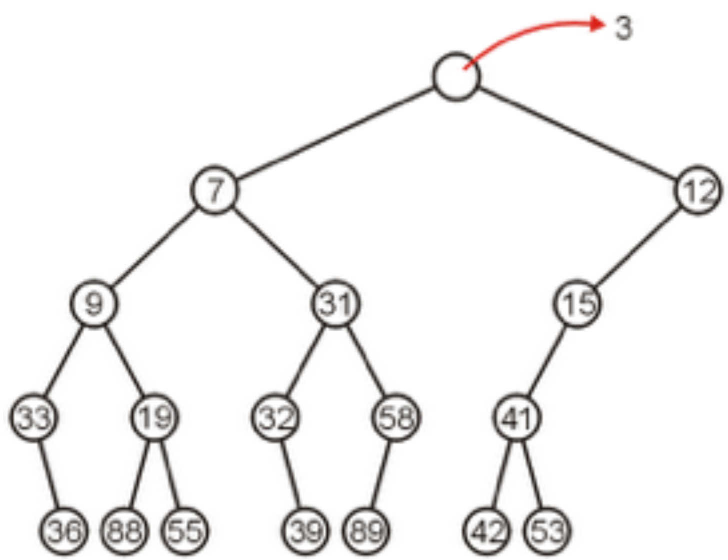
Pop

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

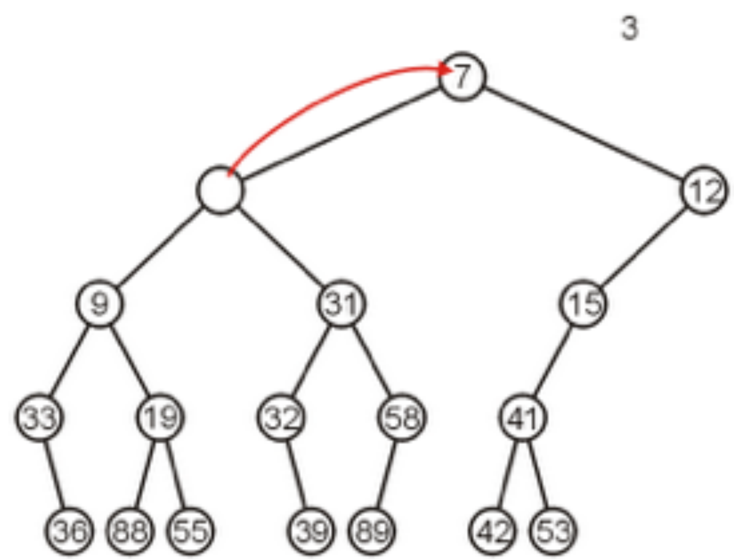
Pop

Using our example, we remove 3:



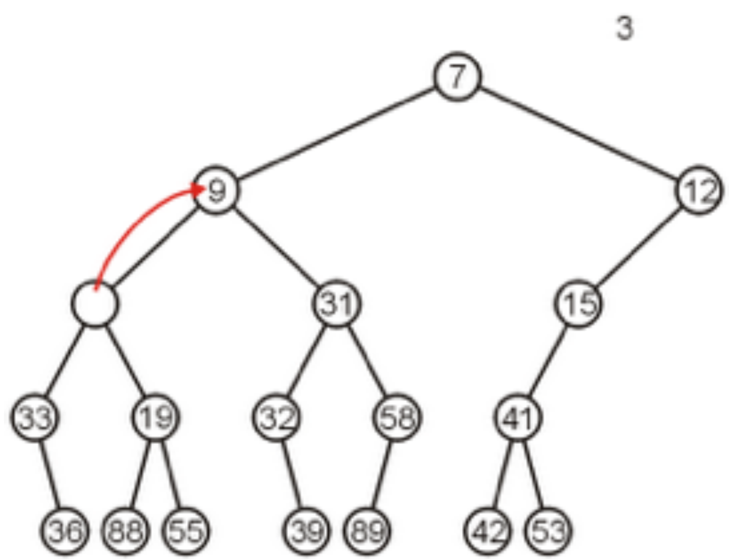
Pop

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



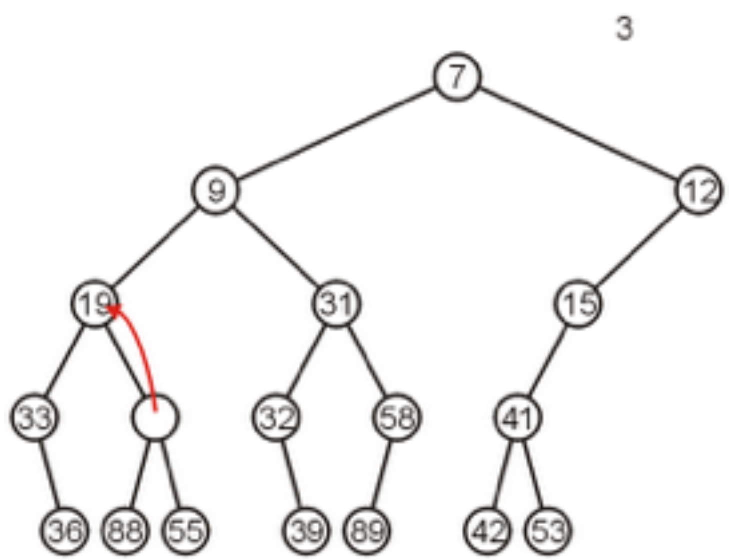
Pop

In the left sub-tree, we promote 9:



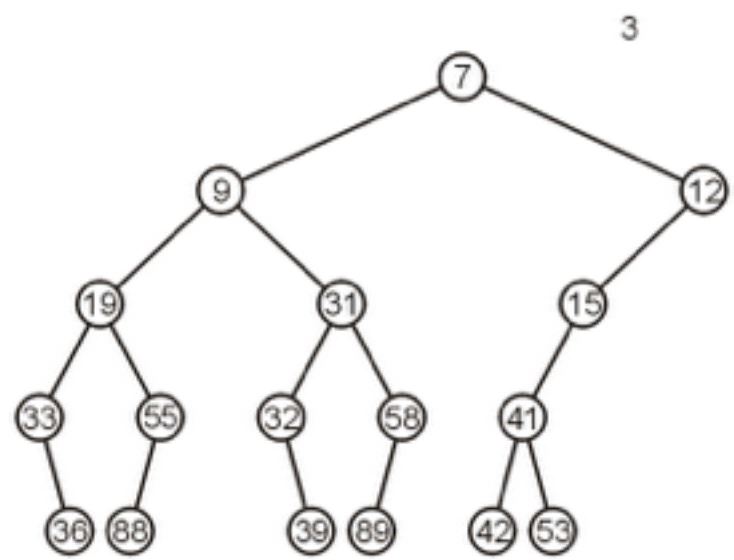
Pop

Recursively, we promote 19:



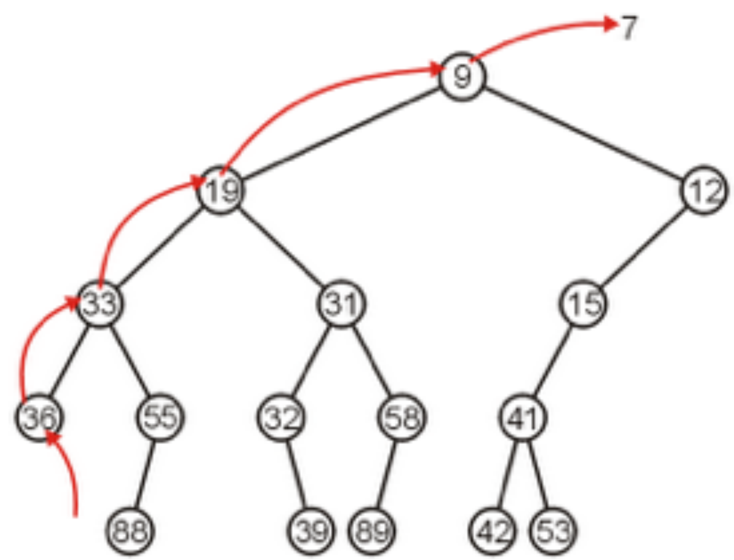
Pop

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



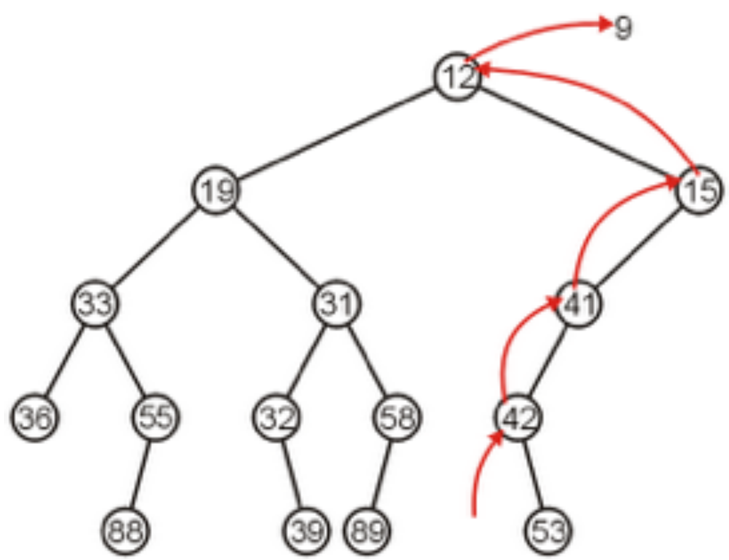
Pop

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)

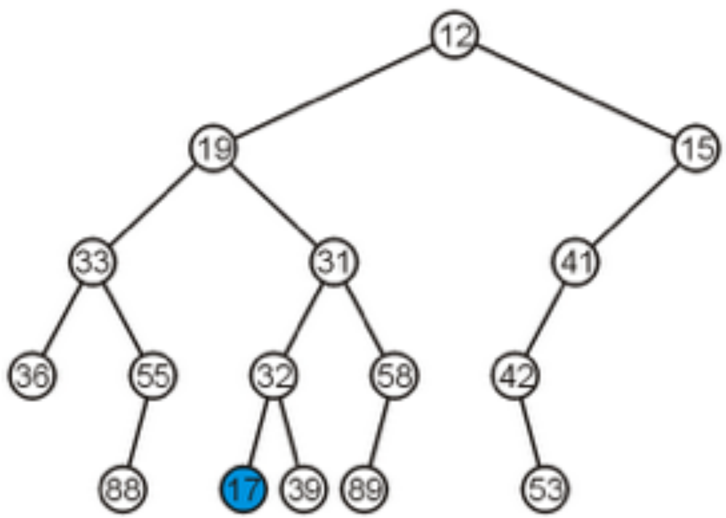
We will use the first approach with binary heaps

- Other heaps use the second

Push

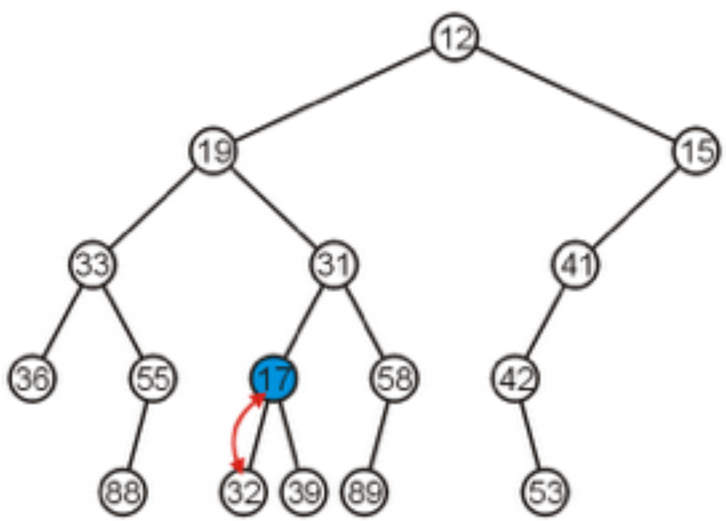
Inserting 17 into the last heap

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



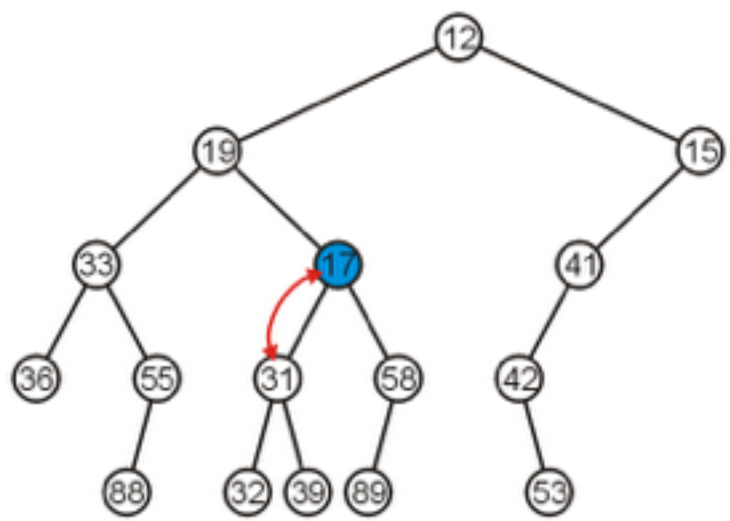
Push

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



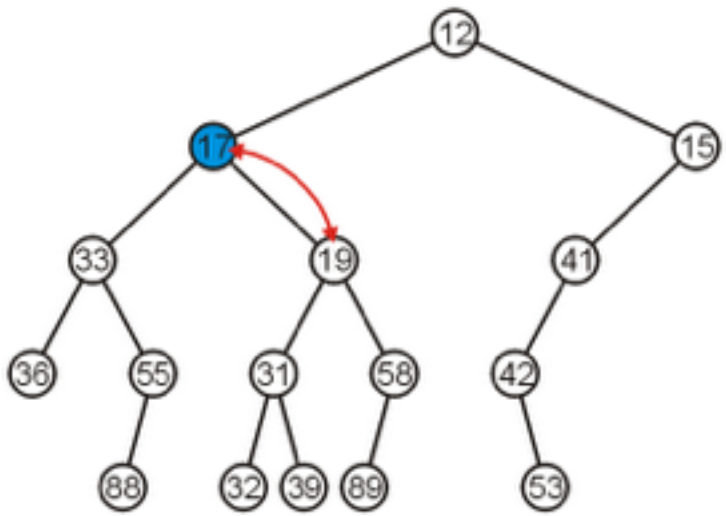
Push

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



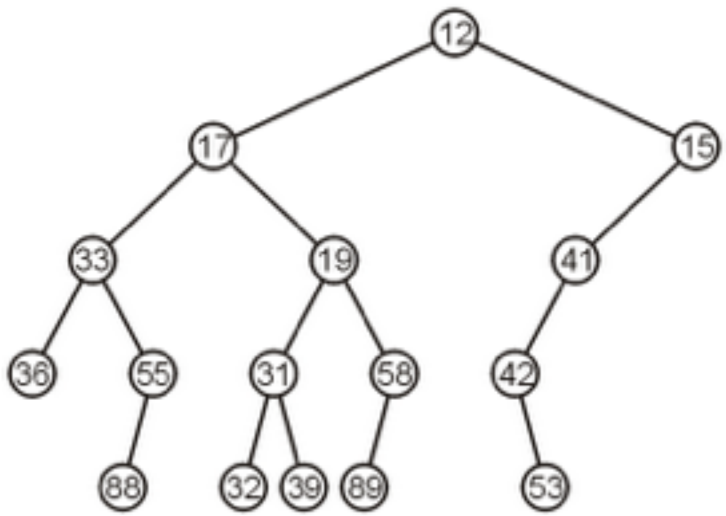
Push

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 discussed about different possible shapes and *balanced* trees,

We looked at:

– AVL Trees

How can we determine where to insert in binary heap so that it is kept balanced?

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

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

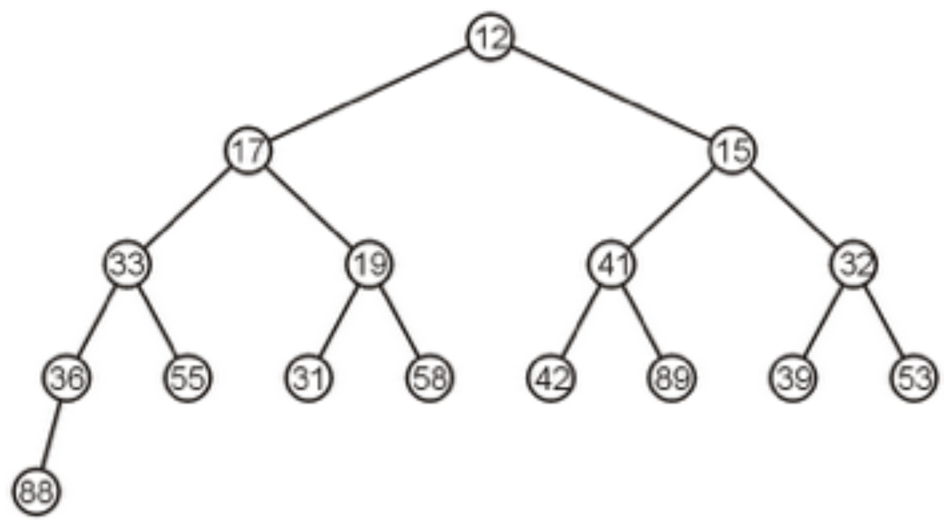
Complete Trees

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

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

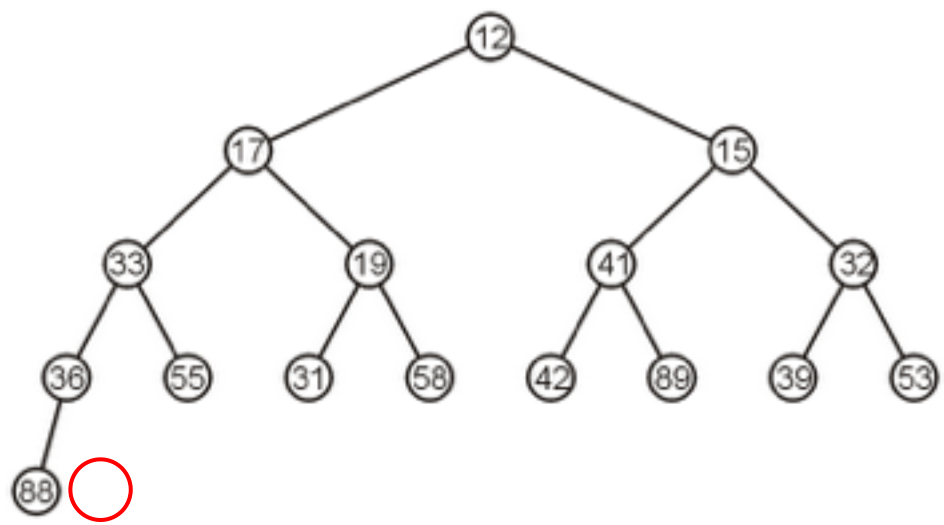
Complete Trees

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



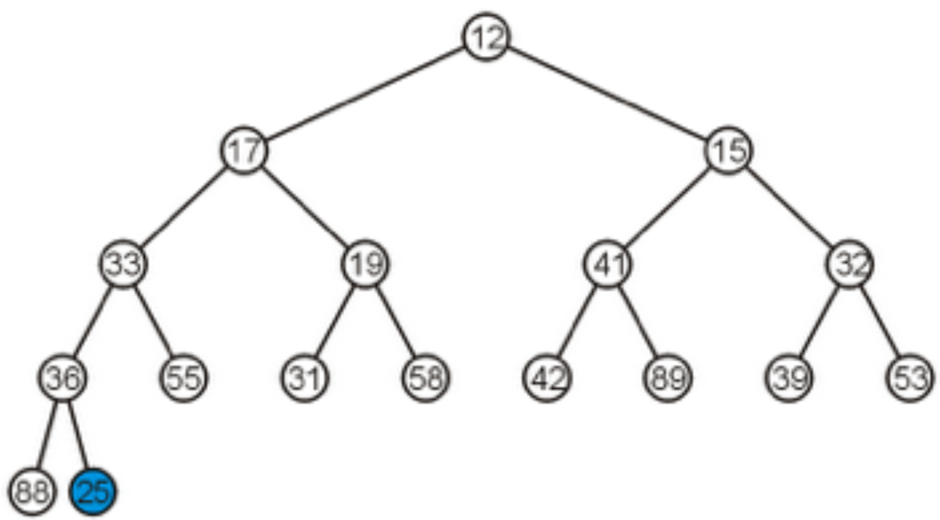
Complete Trees: Push

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



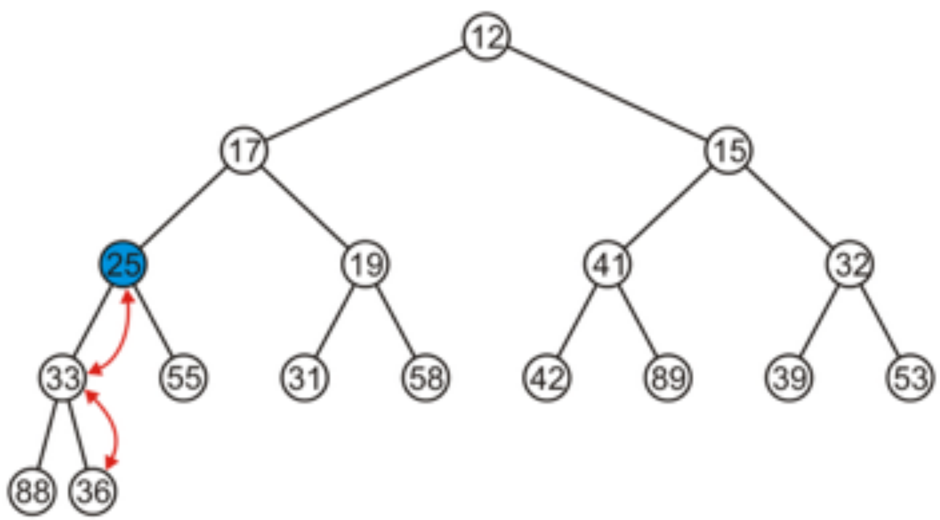
Complete Trees: Push

For example, push 25:



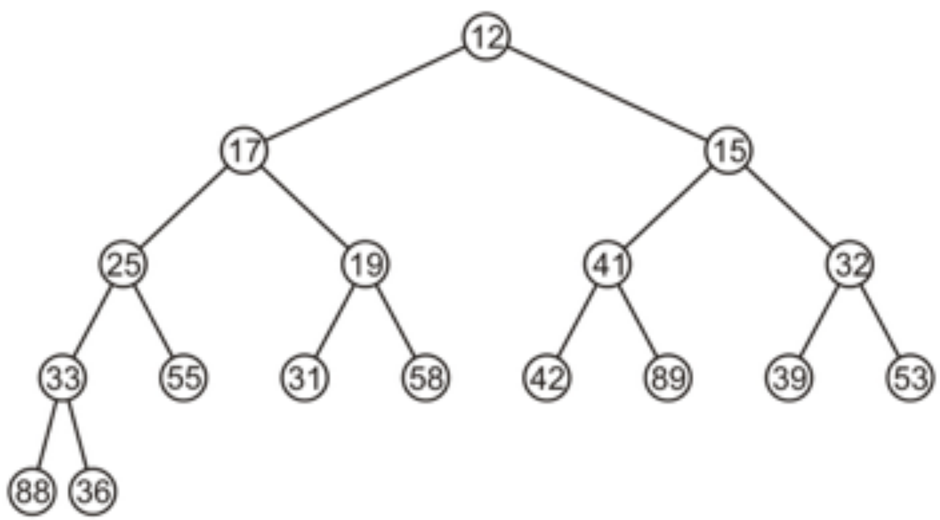
Complete Trees: Push

- We have to percolate 25 up into its appropriate location
  - The resulting heap is still a complete tree



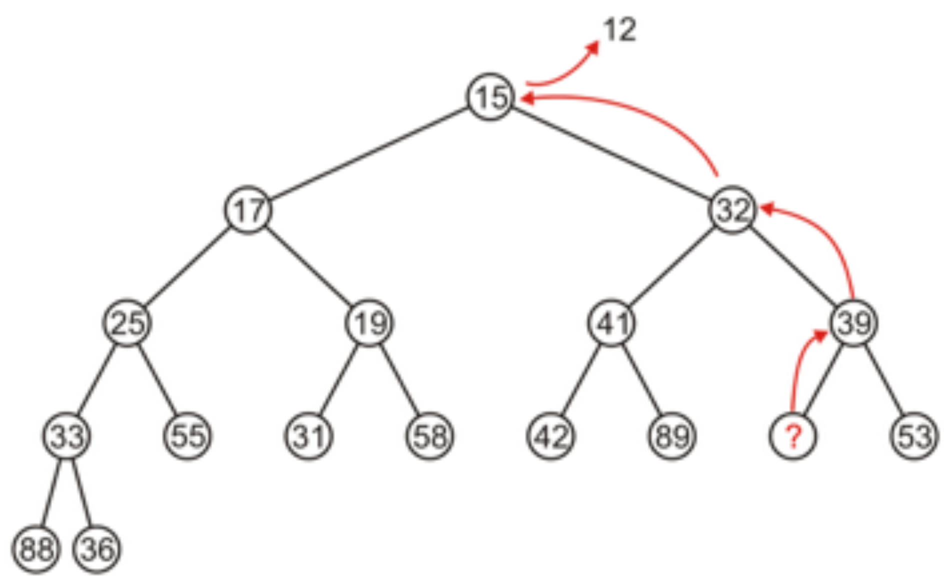
Complete Trees: Pop

Suppose we want to pop, it should be the top entry: 12



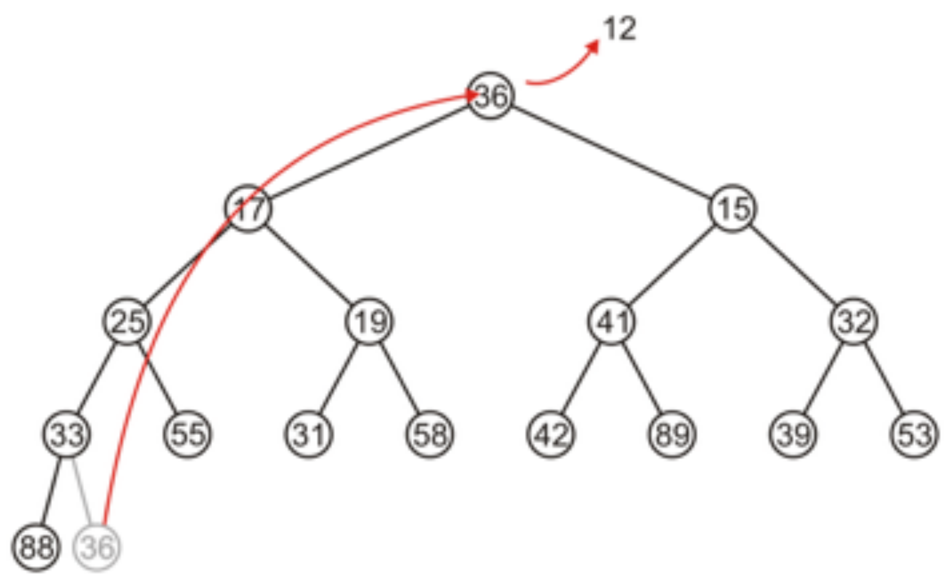
Complete Trees: Pop

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



Complete Trees: Pop

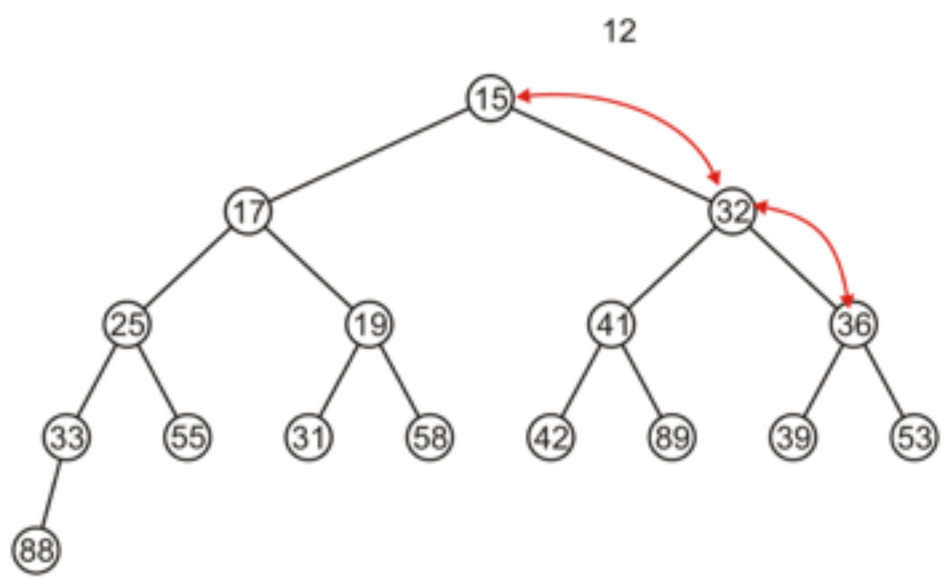
Alternatively, copy the last entry in the heap to the root



Complete Trees: Pop

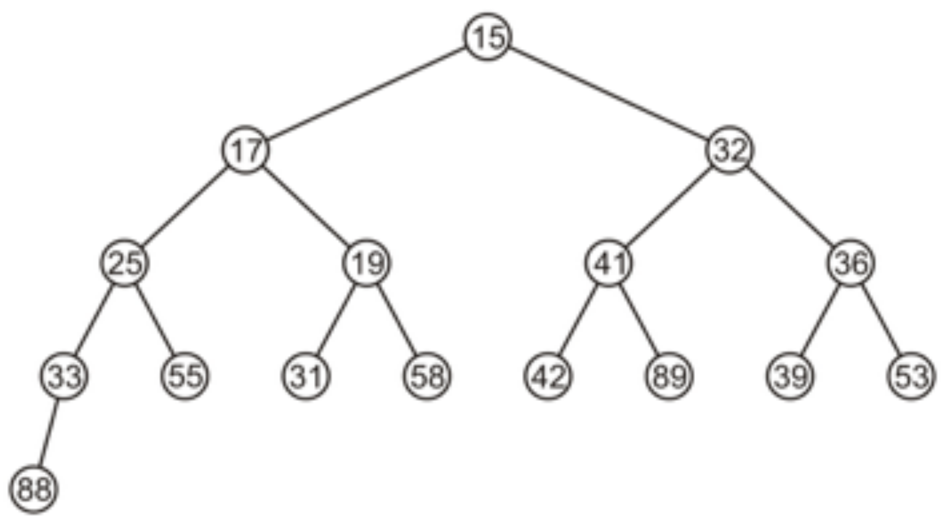
Now, percolate 36 down swapping it with the smallest of its children

- We halt when both children are larger



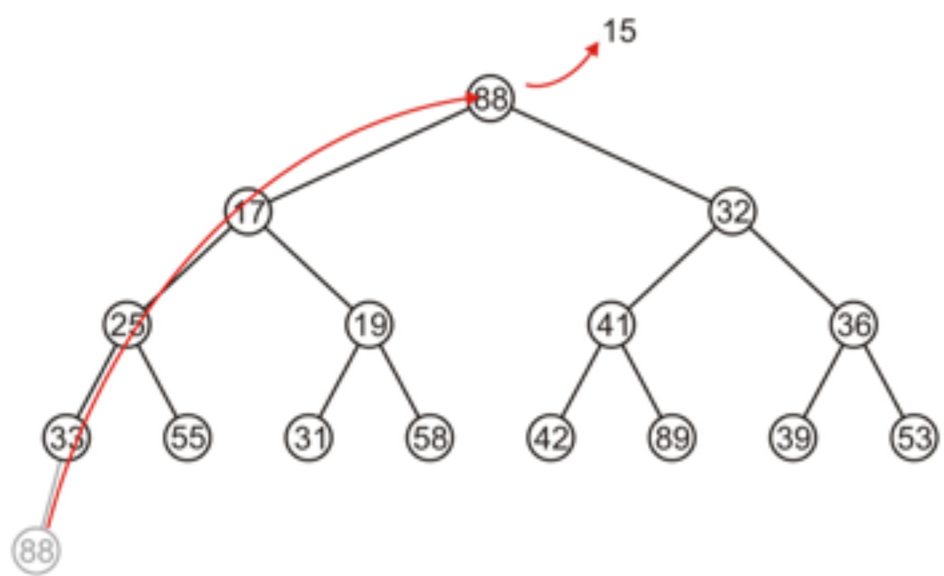
Complete Trees: Pop

The resulting tree is now still a complete tree:



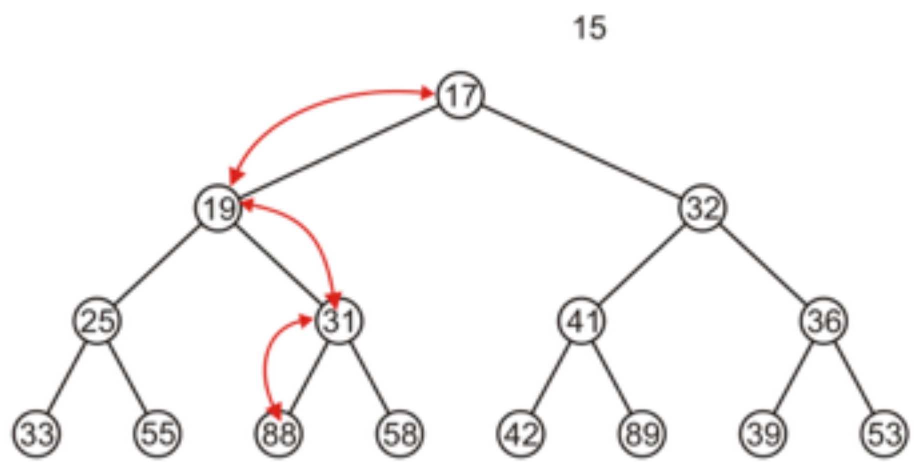
Complete Trees: Pop

Again, we want to pop, remove the 15, copy up the last entry: 88 to the root



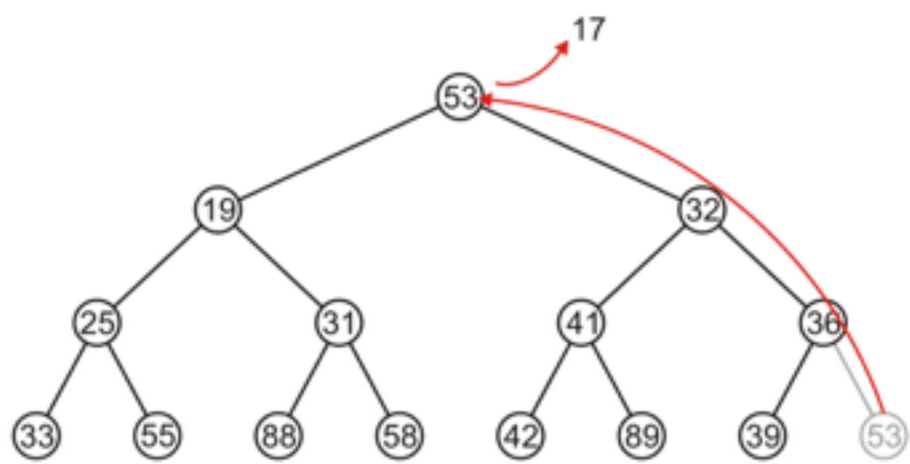
Complete Trees: Pop

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



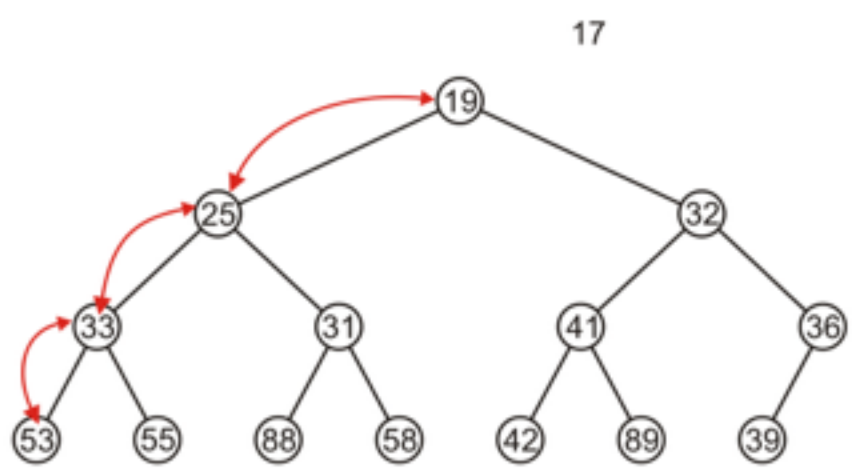
Complete Trees: Pop

In next pop, 17 is piped and 53 is moved to the root



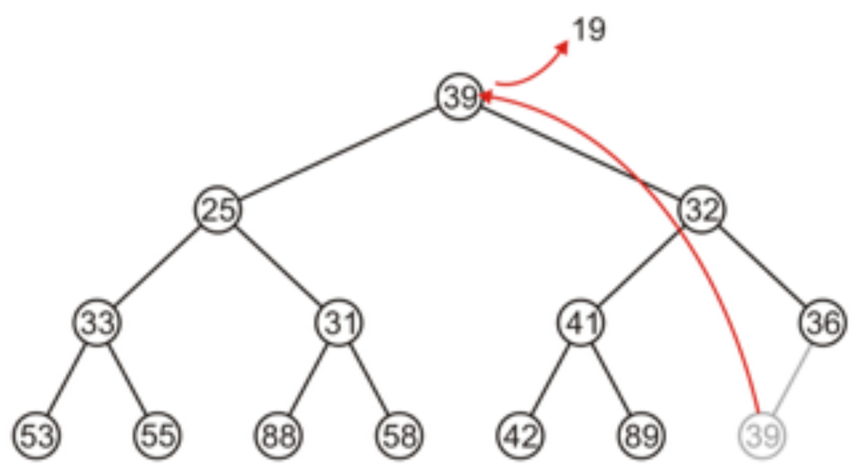
Complete Trees: Pop

And percolated down, again to the deepest level



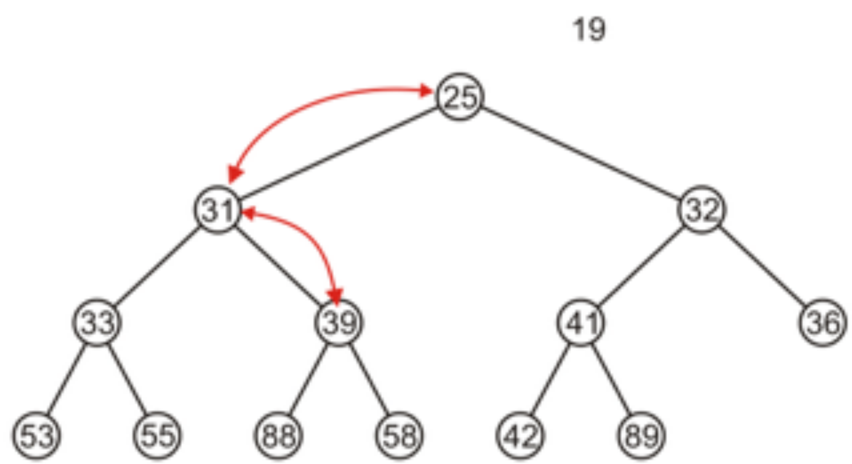
Complete Trees: Pop

Popping 19 copies up 39



Complete Trees: Pop

Which is then percolated down to the second deepest level



Complete Tree

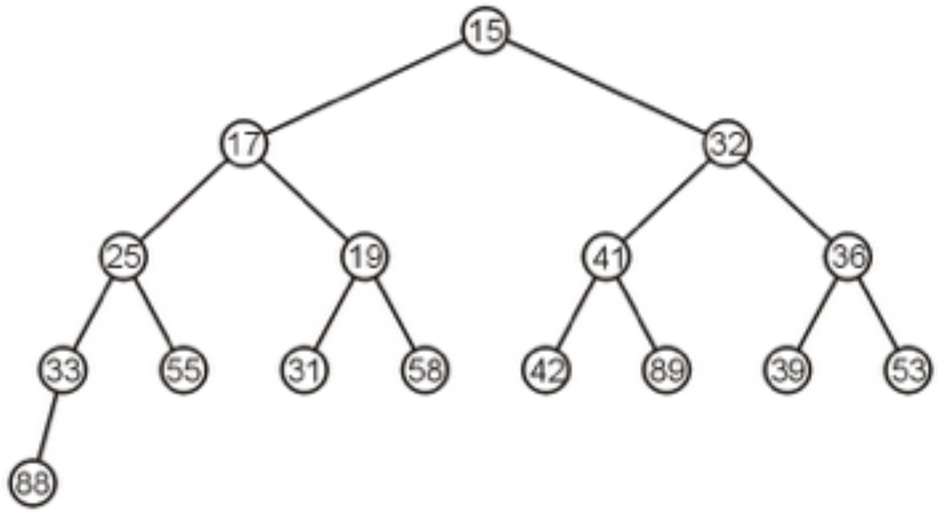
Therefore, we can maintain the complete-tree shape of a heap

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

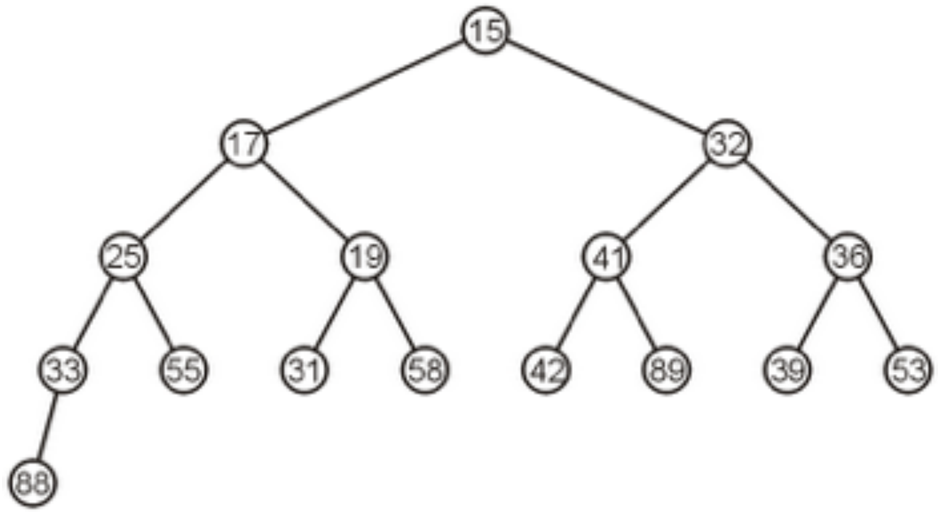
For the heap



a breadth-first traversal yields:

Array Implementation

For the heap

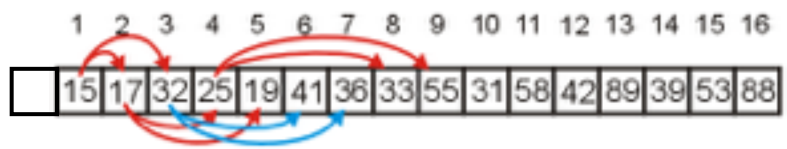


a breadth-first traversal yields:

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

Array Implementation

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

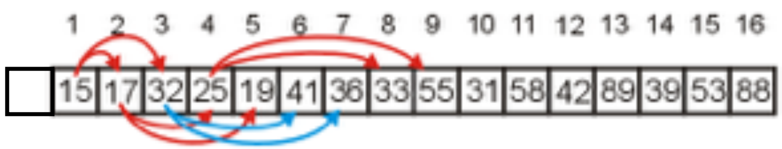


Given the entry at index k, it follows that:

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

Array Implementation

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 $2k + 1$

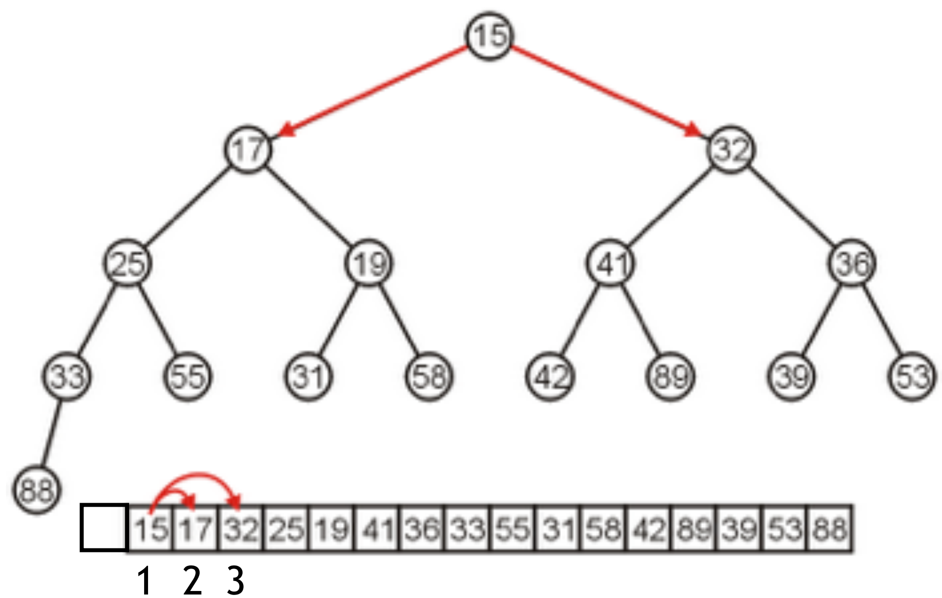
Bitwise Operators

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

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

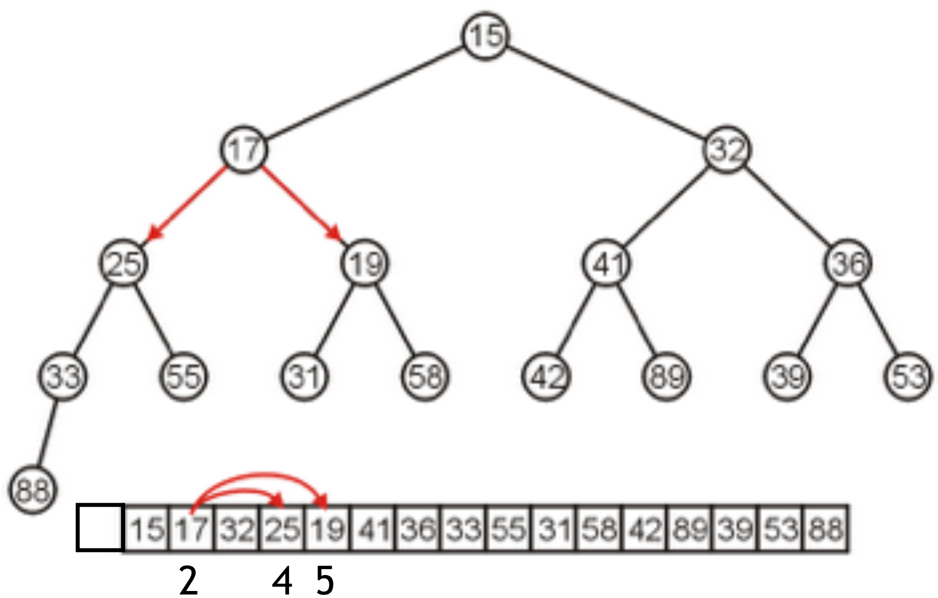
Array Implementation

The children of 15 are 17 and 32:



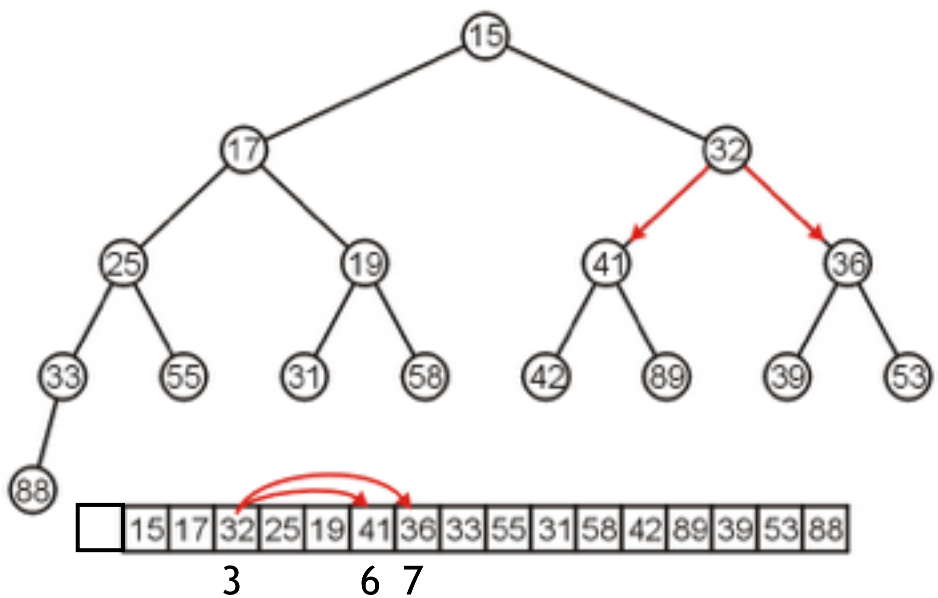
Array Implementation

The children of 17 are 25 and 19:



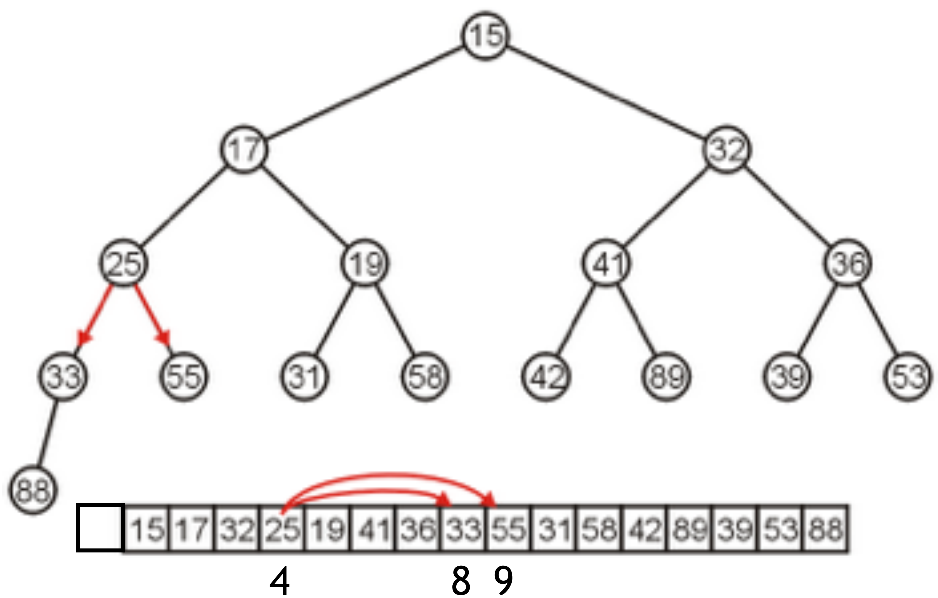
Array Implementation

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`**

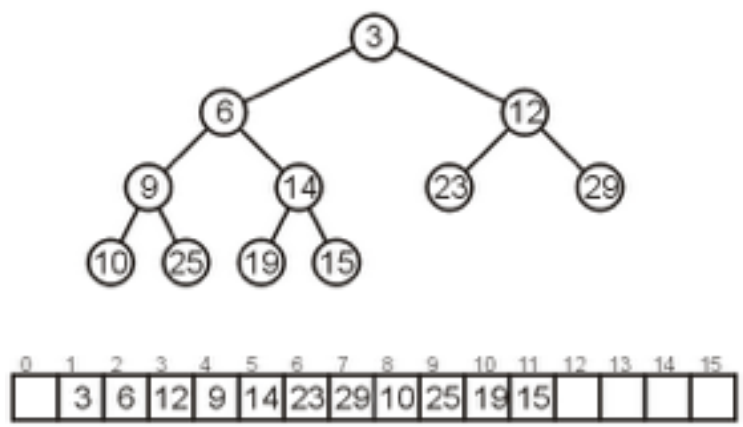
We compare the item at location **`posn`** with the item at **`posn/2`**

If they are out of order

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

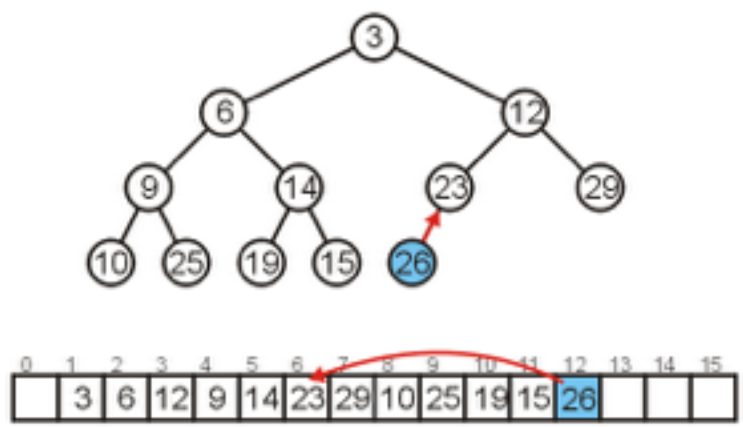
Array Implementation

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



Array Implementation: Push

Inserting 26 requires no changes



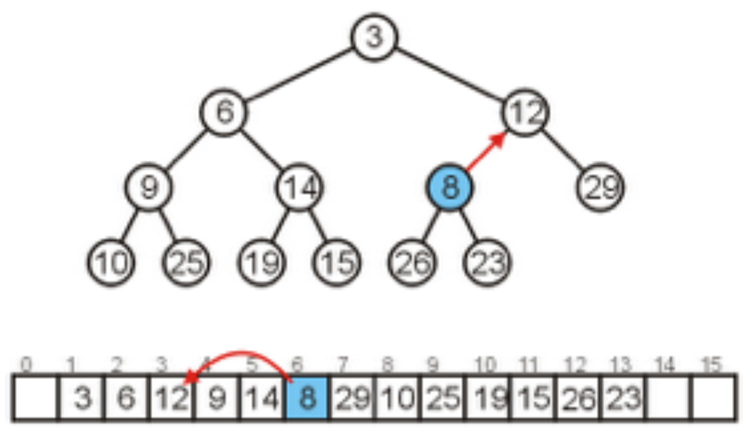
Array Implementation: Push

Inserting 8 requires a few percolations:

- Swap 8 and 23

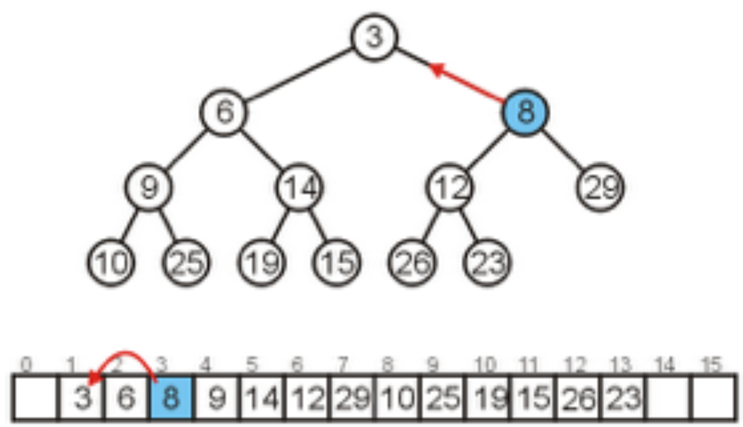
Array Implementation: Push

Swap 8 and 12



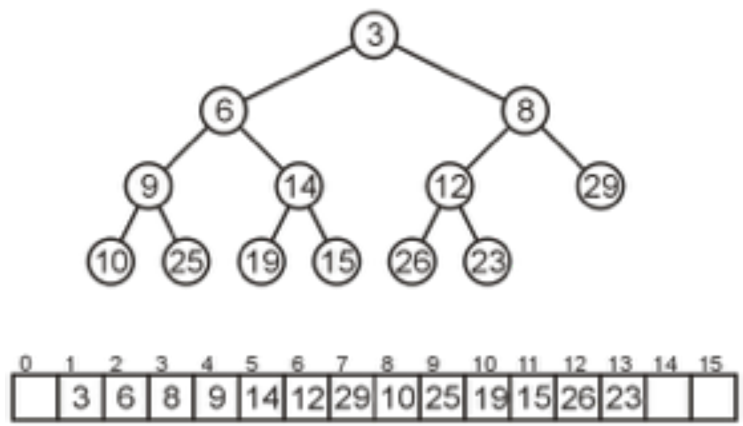
Array Implementation: Push

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



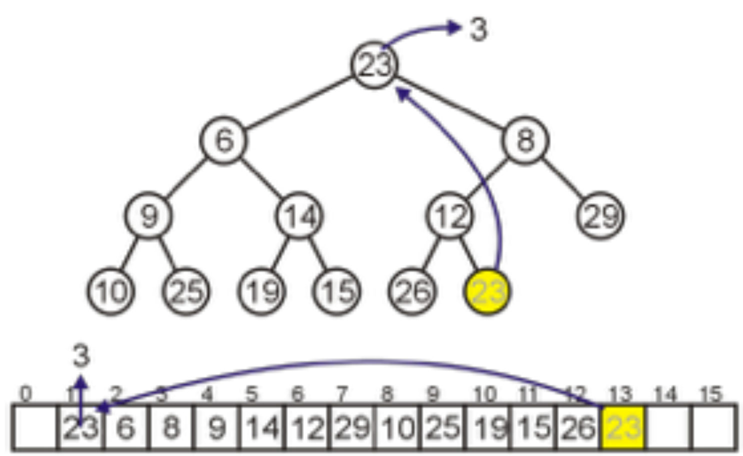
Array Implementation: Pop

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



Array Implementation: Pop

Copy the last object, 23, to the root

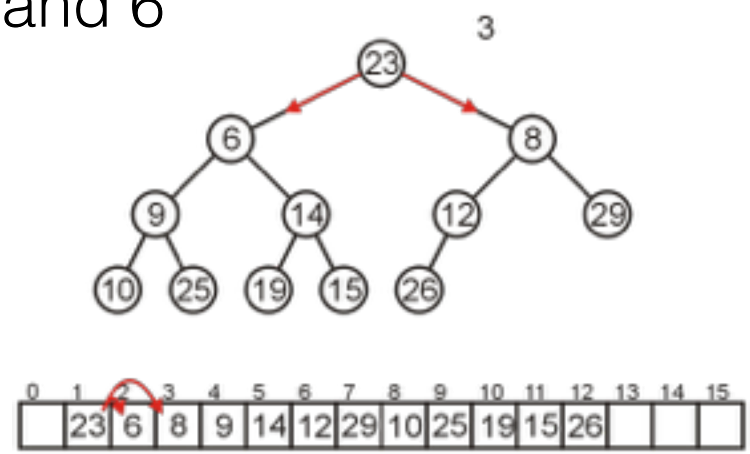


Array Implementation: Pop

Now percolate down

Compare Node 1 with its children: Nodes 2 and 3 (choose the smaller one)

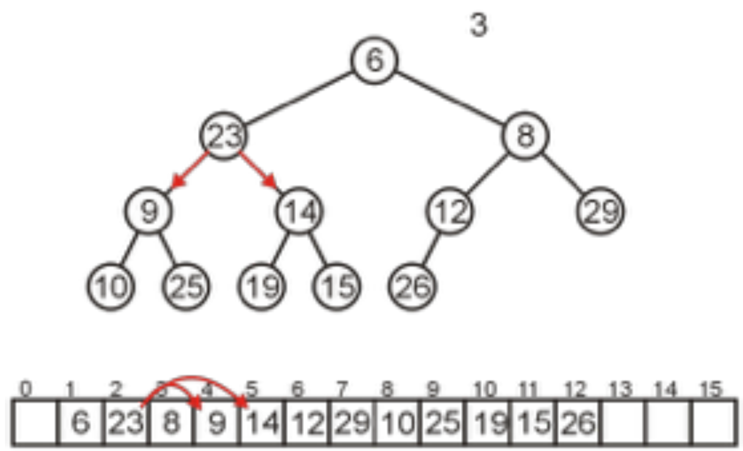
– Swap 23 and 6



Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

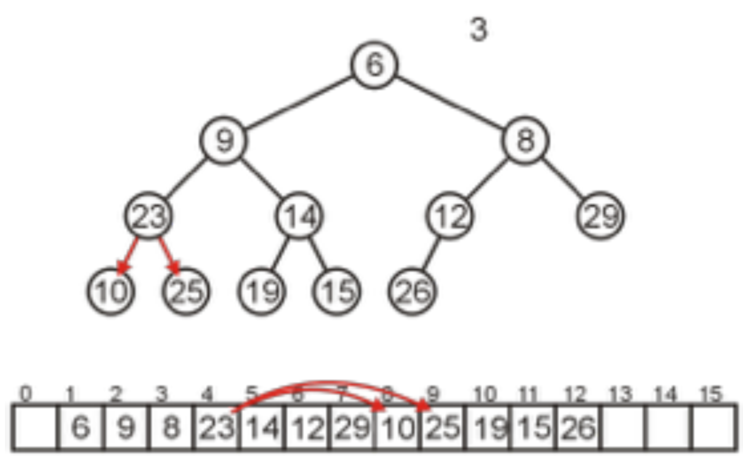
– Swap 23 and 9



Array Implementation: Pop

Compare Node 4 with its children: Nodes 8 and 9

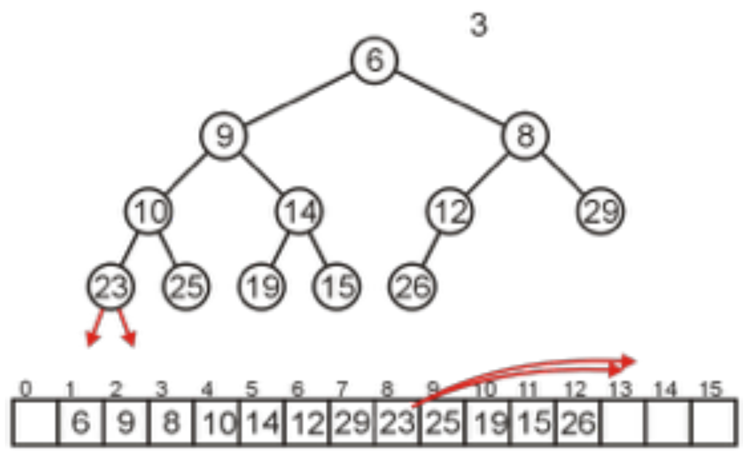
- Swap 23 and 10



Array Implementation: Pop

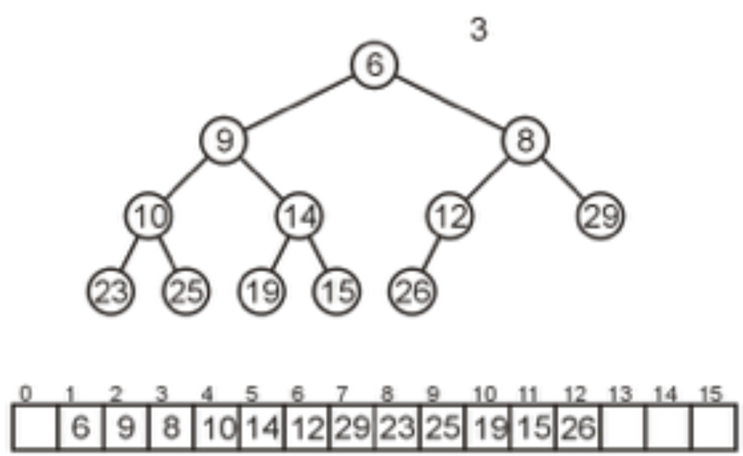
The children of Node 8 are beyond the end of the array:

– Stop



Array Implementation: Pop

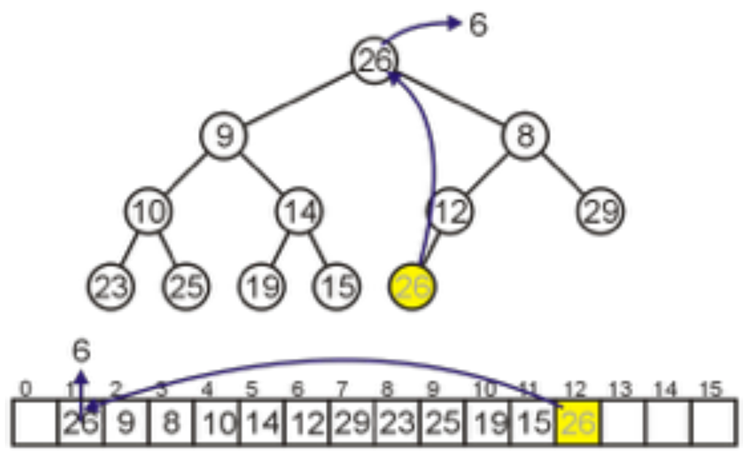
The result is a binary min-heap



Array Implementation: Pop

Popping or dequeuing the minimum again:

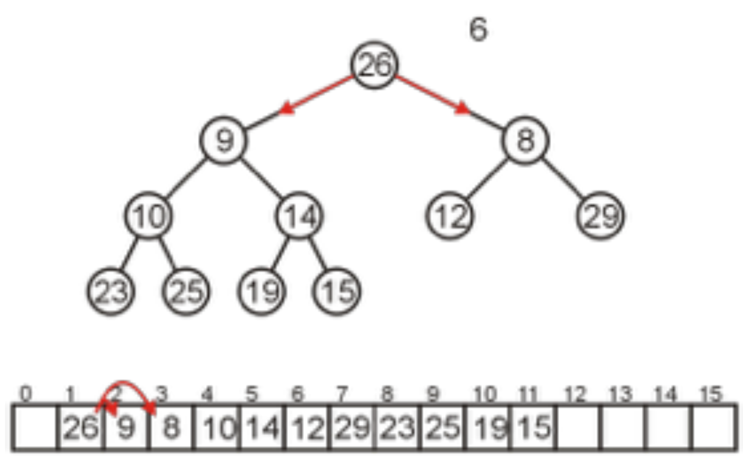
– Copy 26 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

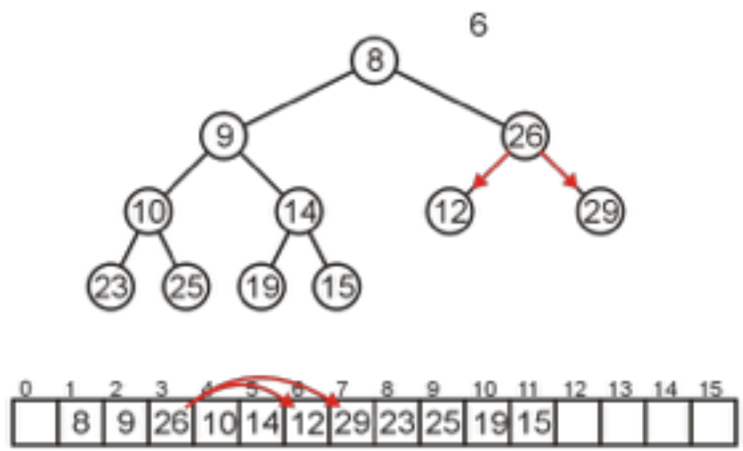
- Swap 26 and 8



Array Implementation: Pop

Compare Node 3 with its children: Nodes 6 and 7

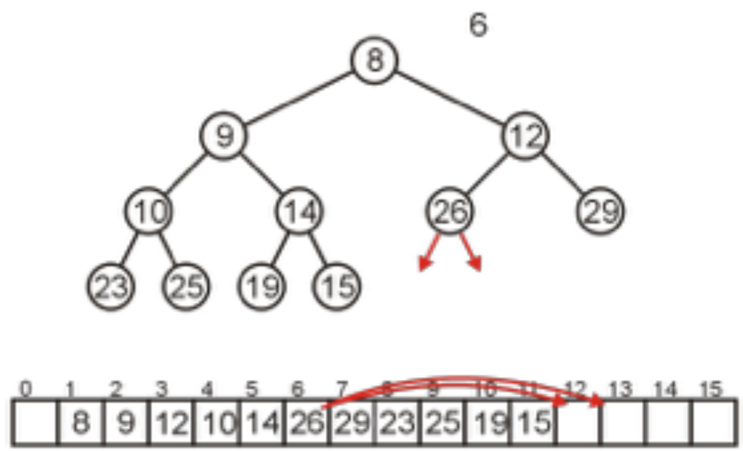
- Swap 26 and 12



Array Implementation: Pop

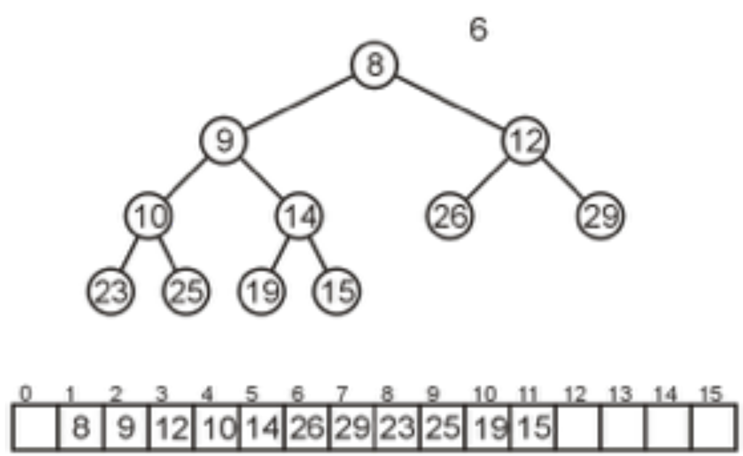
The children of Node 6, Nodes 12 and 13 are unoccupied

– Currently, `count == 11`



Array Implementation: Pop

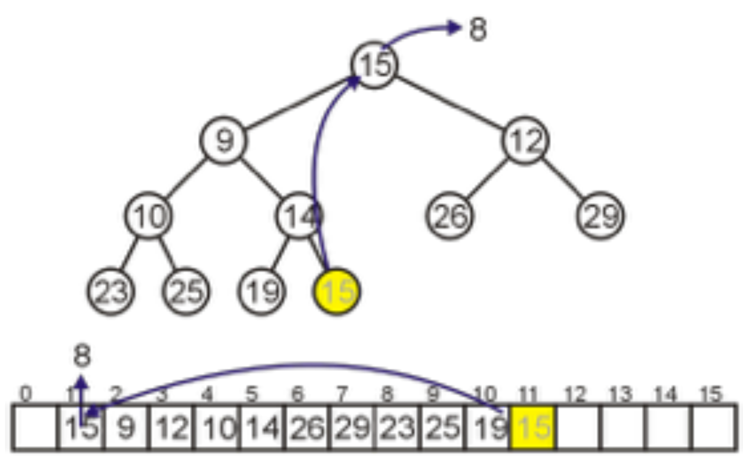
The result is a min-heap



Array Implementation: Pop

Dequeuing the minimum a third time:

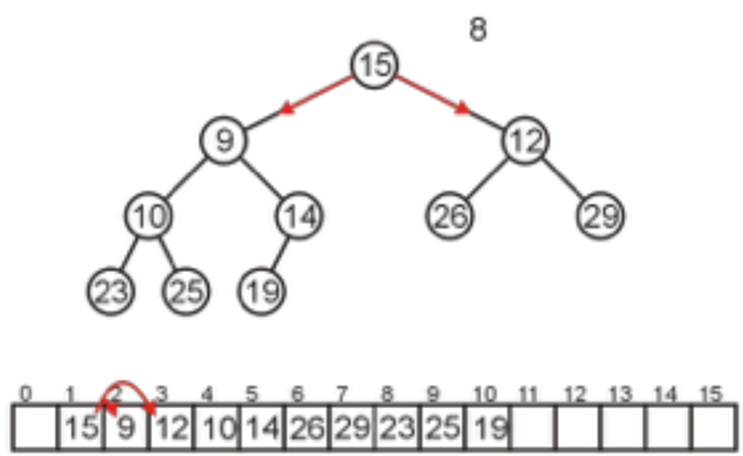
– Copy 15 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

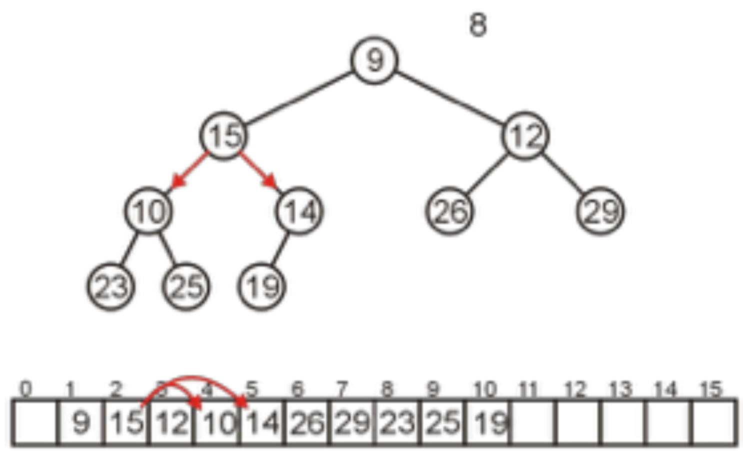
– Swap 15 and 9



Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

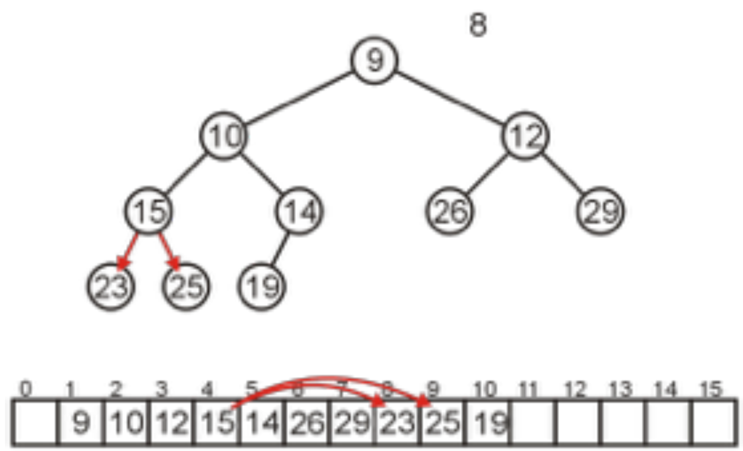
– Swap 15 and 10



Array Implementation: Pop

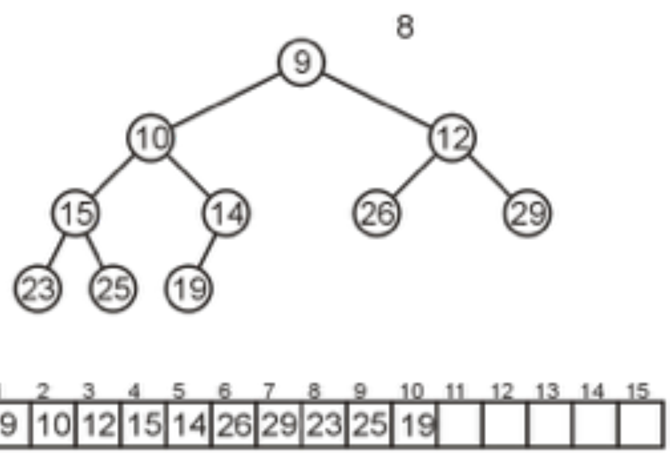
Compare Node 4 with its children: Nodes 8 and 9

– 15 < 23 and 15 < 25 so stop



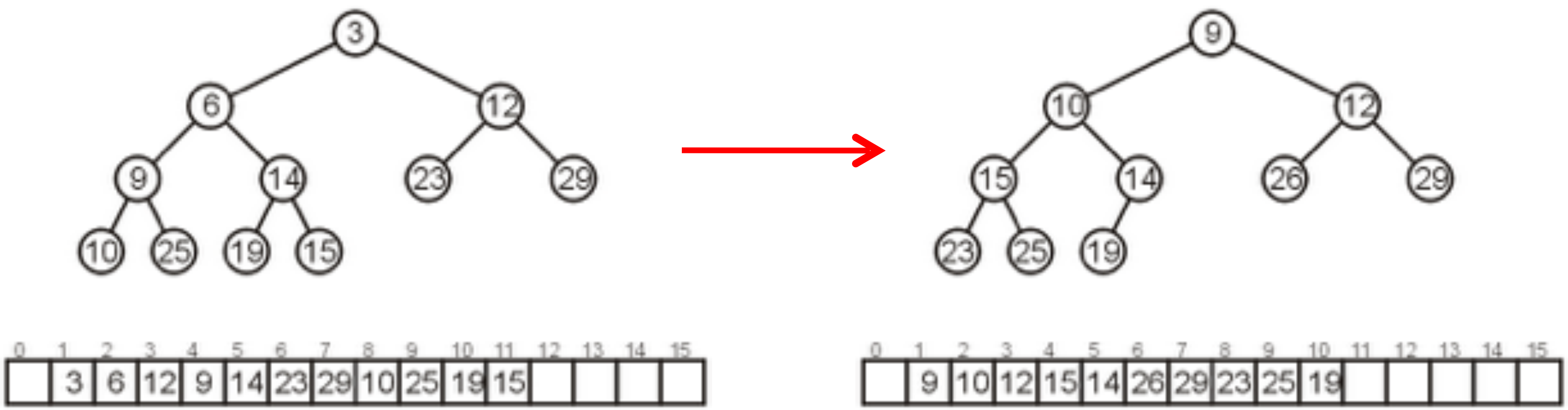
Array Implementation: Pop

The result is a properly formed binary min-heap



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, then the run time will be $\Theta(\ln(n))$

If we insert an object 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

To find the average time complexity of insertion, we need to find the average height of nodes in the binary heap tree (complete tree)

The tree has 1 node at height h, 2 nodes at height h-1, 4 nodes at height h -2, etc

1 (2^0)	h
2 (2^1)	h - 1
4 (2^2)	h - 2
8 (2^3)	h - 3
…….	
2^h	0

Theorem

For a perfect binary tree of height h containing $N = 2^{h+1} - 1$ nodes,

the sum S of the heights of the nodes is
$S = 2^{h+1} - 1 - (h + 1) = N - h - 2$

Run-time Analysis

To find the average time complexity of insertion, we need to find the average height of nodes in the binary heap tree (complete tree)

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

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

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

Run-time Analysis

There are other heaps with better run-time characteristics, but:

- Leftist, skew, binomial and Fibonacci heaps all use a node-based implementation requiring $\Theta(n)$ additional memory

Run-time Analysis

Analyzed the run time:

- Top $\Theta(1)$
- Push $\Theta(1)$ average, $\mathbf{O}(\ln(n))$ worst case
- Pop $\mathrm{O}(\ln(n))$
- Arbitrary remove $\mathrm{O}(n)$
- Merge two heaps (size n) $\mathbf{O}(n)$

Other Heap Operations

1. DecreaseKey(p,d)

   increase the priority of element p in the heap with a positive value d.

   percolate up.

2. IncreaseKey(p,d)

   decrease the priority of element p in the heap with a positive value d.

   percolate down.

Other Heap Operations

3. BuildHeap

input N elements

Trivial solution: place them into an empty heap through successive inserts. The worst case running time is $O(n*\log(n))$.

Build Heap - $\mathbf{O}(n)$

Given an array of elements to be inserted in the heap,

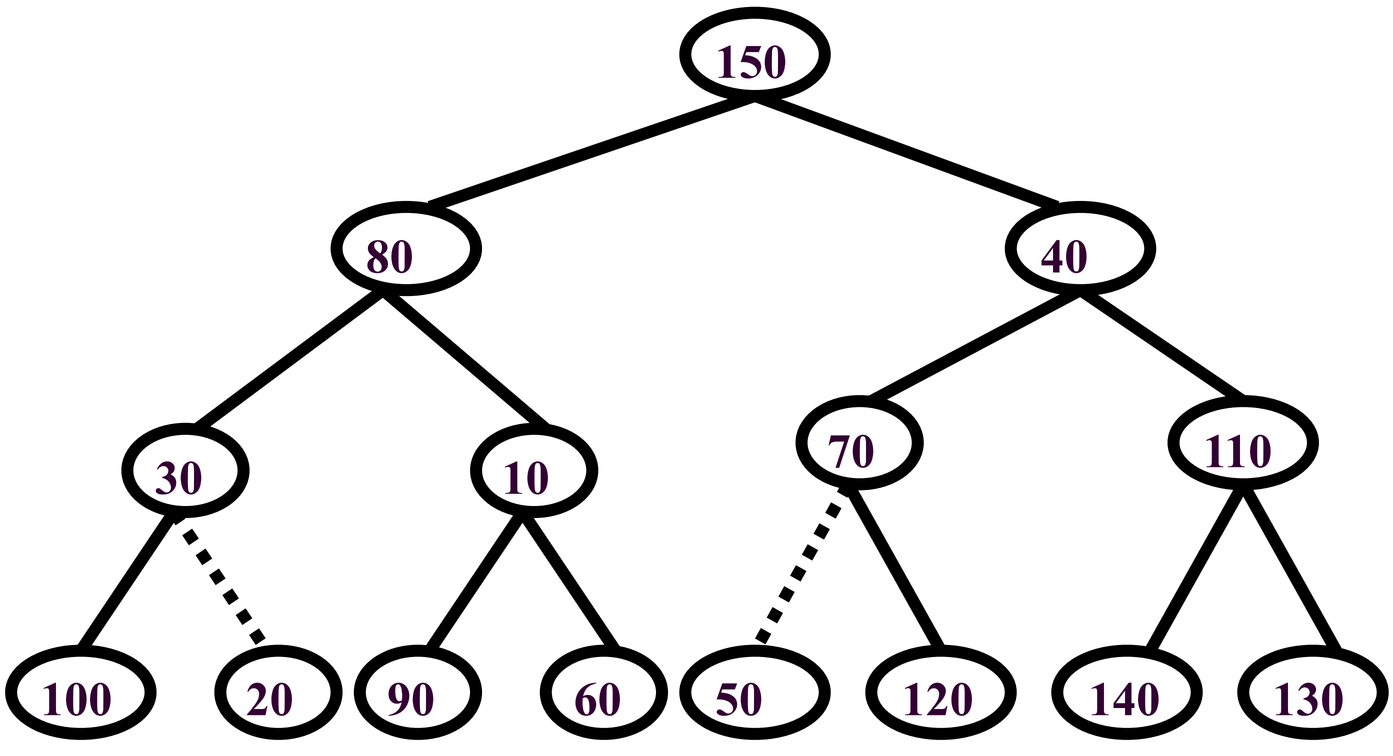
✓treat the array as a heap with order property violated,

✓and then do operations to fix the order property.

Example:

150 80 40 30 10 70 110 100 20 90 60 50 120 140 130

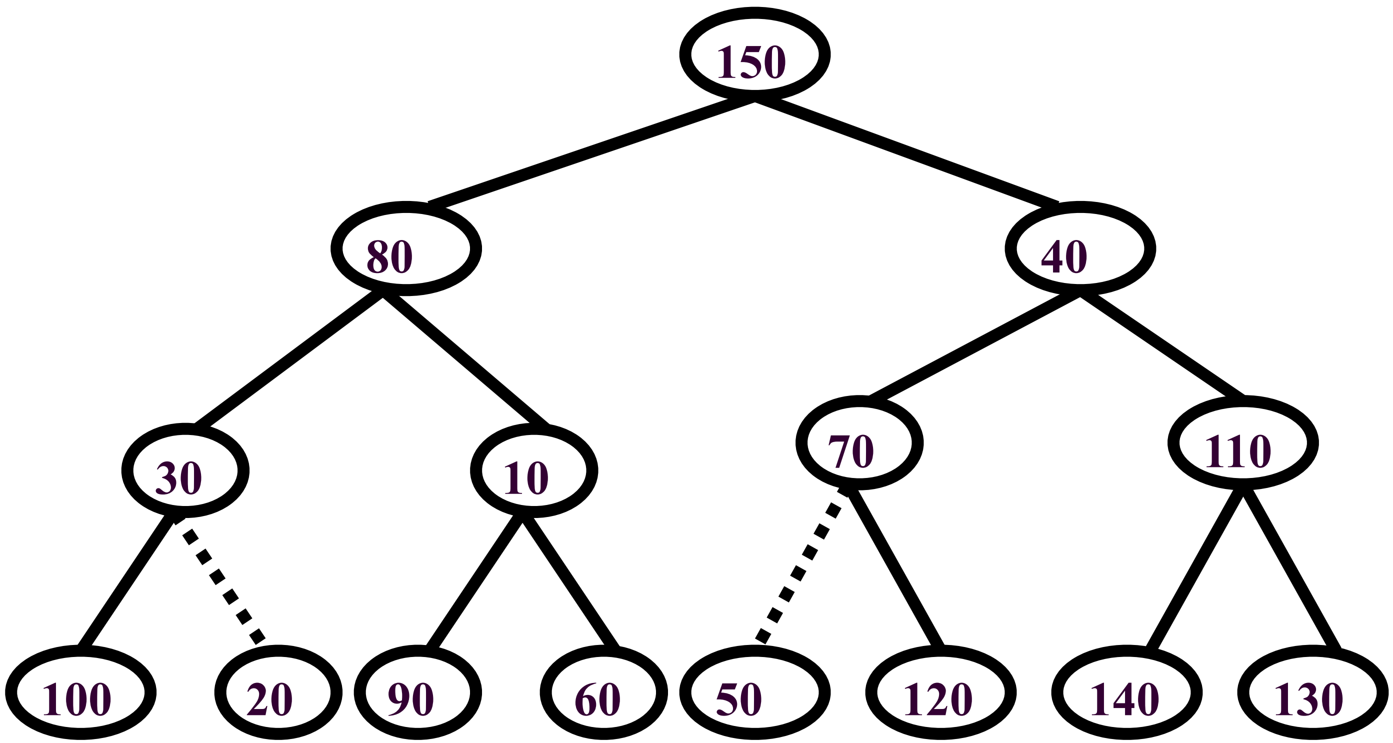
the numbers in array create a tree like this:



Example:

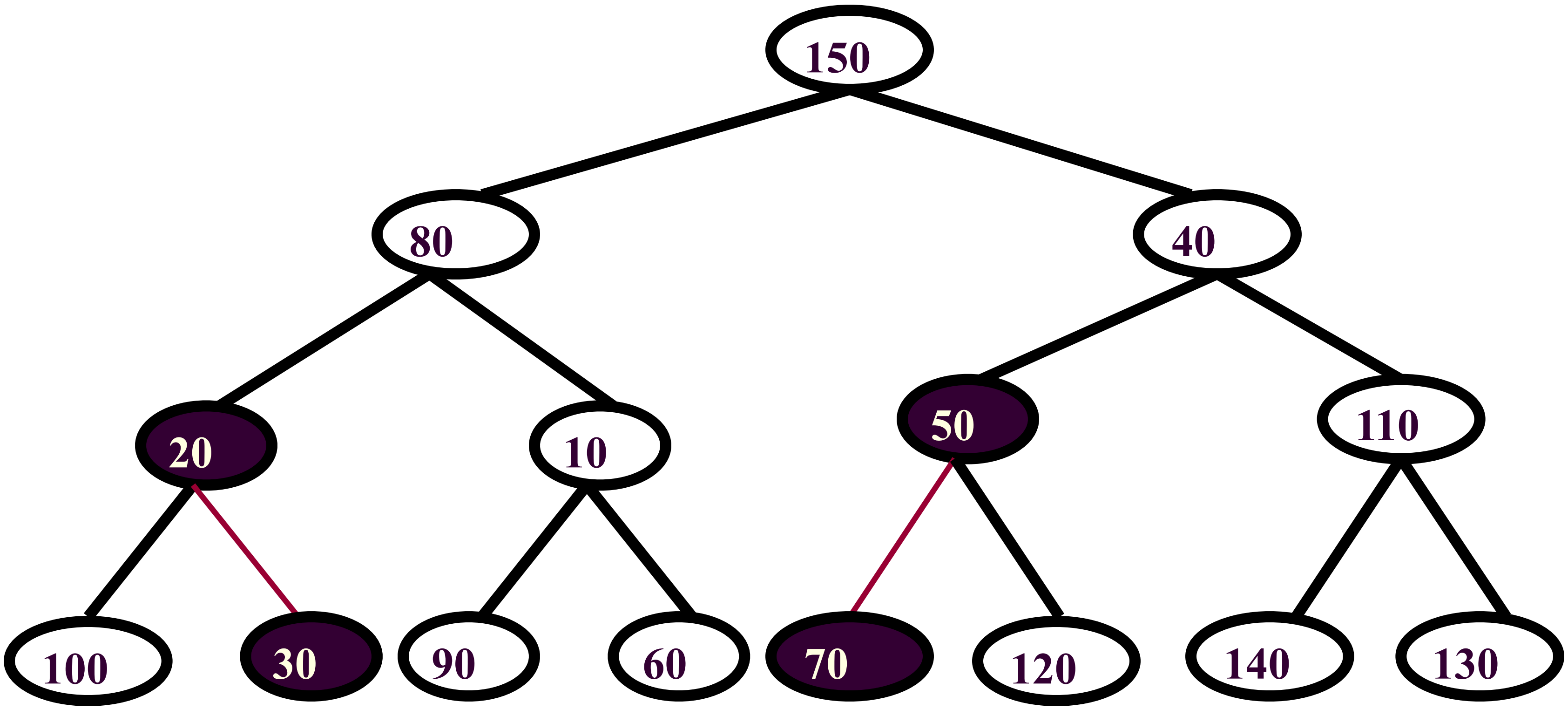
150 80 40 30 10 70 110 100 20 90 60 50 120 140 130

Check if the order of nodes in height 1 are fine
(they are smaller than their children)



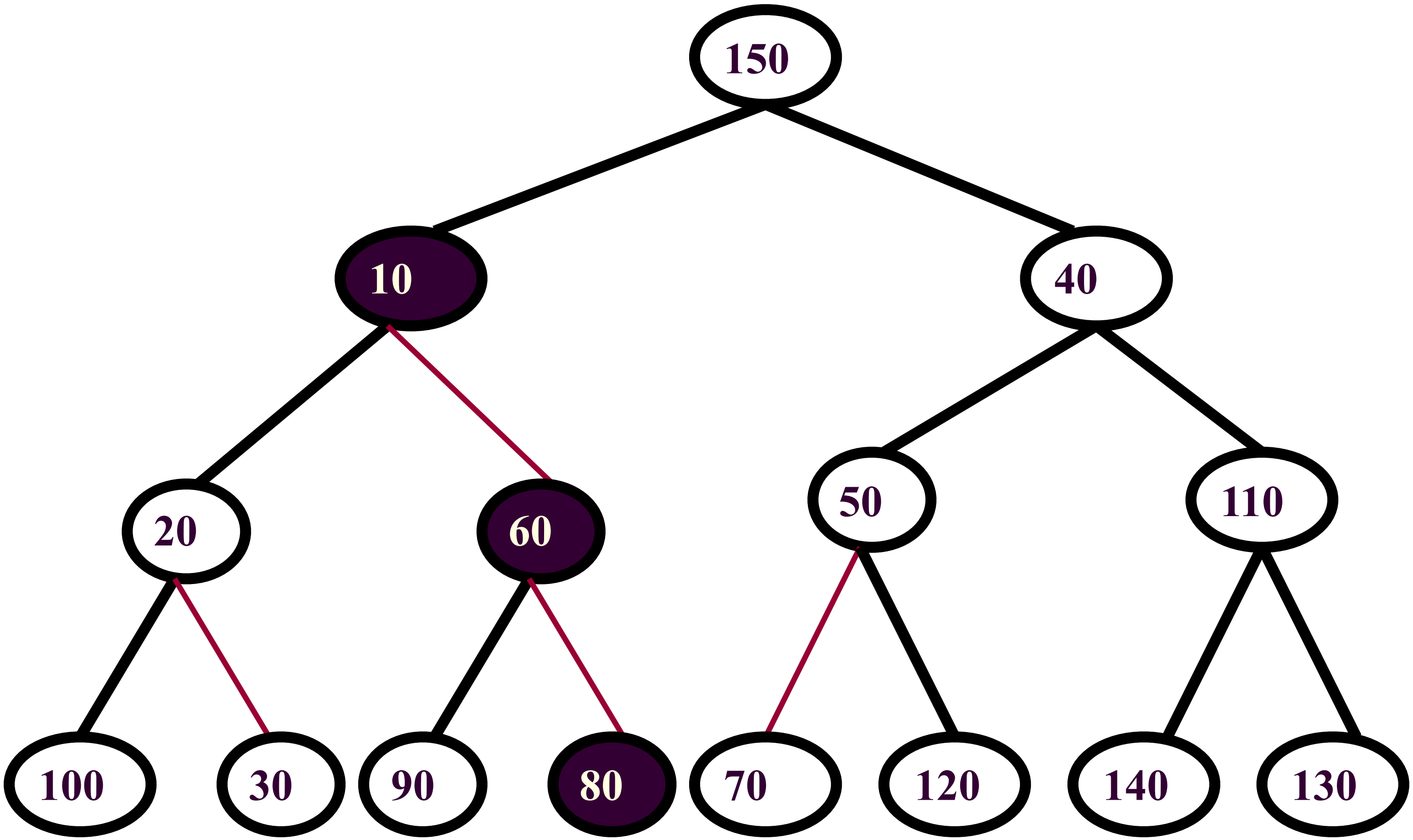
Example (cont)

After processing height 1



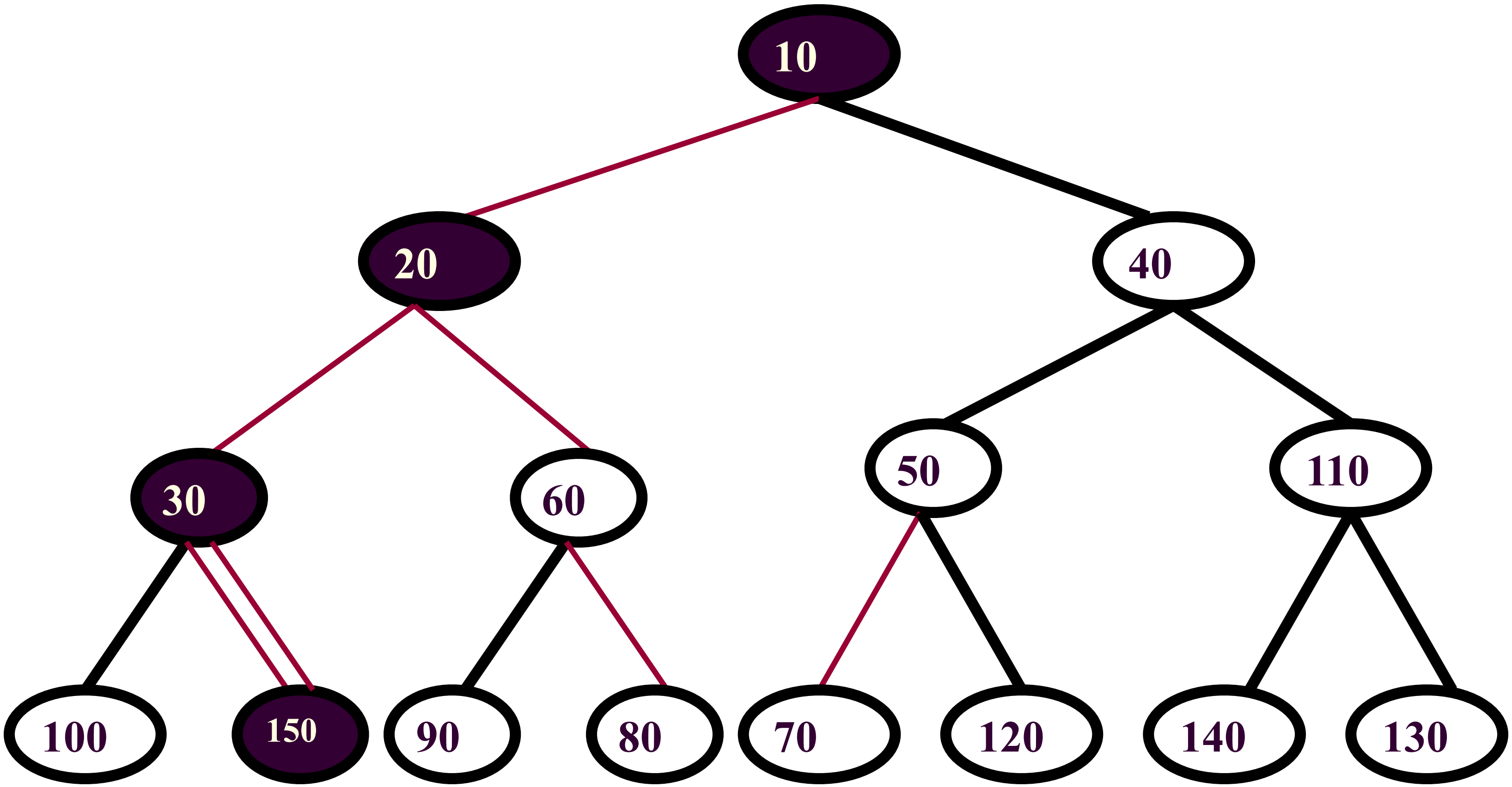
Example (cont)

After processing height 2



Example (cont)

After processing height 3



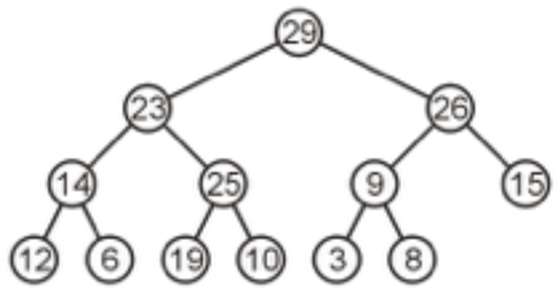
Theorem

For a perfect binary tree of height h containing $N = 2^{h+1} - 1$ nodes,

the sum S of the heights of the nodes is
$S = 2^{h+1} - 1 - (h + 1) = O(N)$

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



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		

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

References

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

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

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