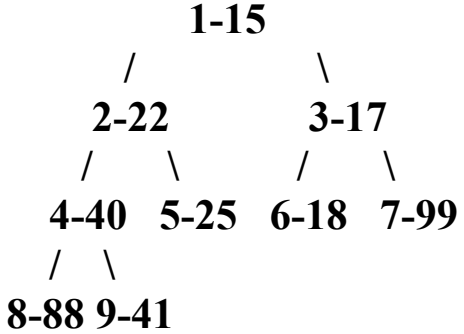
The Binary Heap

A binary heap looks similar to a binary search tree, but has a different property/invariant that each node in the tree satisfies. In a binary heap, all the values stored in the subtree rooted at a node are greater than or equal to the value stored at the node.

We can use a binary heap for a couple of things: maintaining a priority queue, and performing a heapsort.

To maintain a priority queue, we need efficient findMin and deleteMin operations. A priority queue is a queue where you always extract the item with the highest priority next. Our implementation will achieve O(log n) time for the deleteMin and Insert operations. (In a normal queue with just insert items and delete them.)

When we maintain a binary heap, we will do so as a complete binary tree. (A tree where all levels are filled in completely but the last and the last has nodes from left to right.) Interestingly enough, we can even store a binary heap in an array instead of an actual binary tree. Basically, the children of node i are nodes 2i and 2i+1. Consider the picture below, where the first item is the node number and the second is the value stored in the node:



Heap Operation: Insert

To do an insertion, consider an existing heap. Since we want to keep the heap balanced, we must insert into the following spot in the heap. (This would be the next array location if you are storing the heap in an array.)

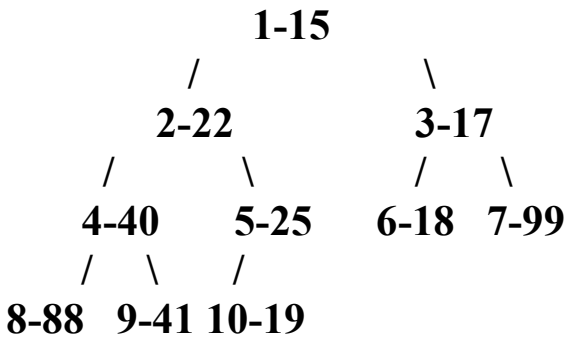
However, the problem is in all likelihood, if the insertion is done in this location, the heap property will not be maintained. Thus, you must do the following "Percolate Up" procedure:

If the parent of the newly inserted node is greater than the inserted value, swap the two of them. This is a single "Percolate Up" step. Now, continue this process until the inserted node 's parent stores a number lower than it.

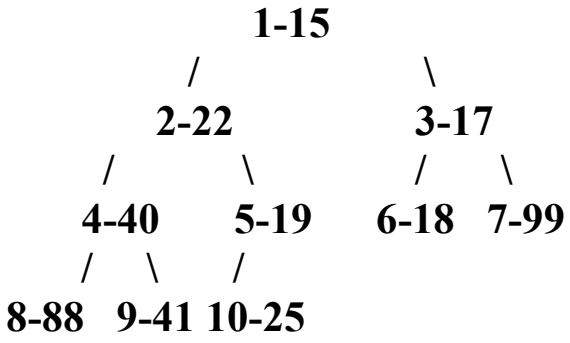
One way I like to think about this is consider a new employee to a company. Naturally, he/she will be placed at the bottom of the totem pole, ie. the last slot in the heap. But then, when he/she reports to their boss, they may recognize that the person ordering them around is less capable then them. Thus, rightfully so, they swap spots with their immediate boss (the parent node.) Why doesn't the new employee have to look at his "co-worker" who used to have the same boss? Continue this until the new employee finds a boss that is indeed his/her superior.

Since the height of the tree is O(lg n), this is an O(lg n) operation.

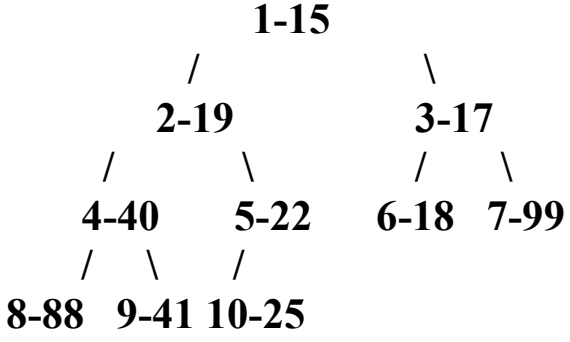
Consider inserting 19 into the heap previously shown, first place it in index 10, as the left child of 25:



Now, we can just compare 19 with 25, its parent. (In the array representation, we just look at index i/2, where i is the newly inserted index.) Since 19 is smaller, we swap:



Now, we just repeat…compare 19 with its parent, 22, and since 19 < 22, we swap again:



Now, comparing 19 to its parent 15, we see 19 is in a valid spot.

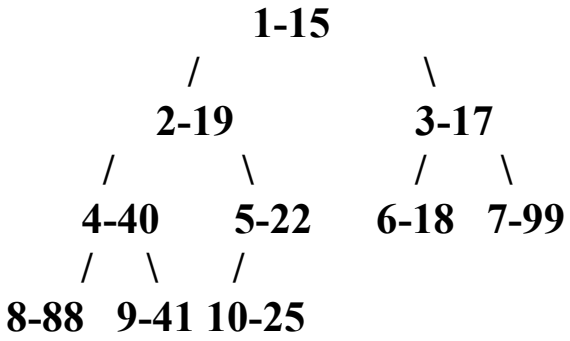
Heap Operation: deleteMin

The first part of a deleteMin operation is quite easy. All you have to do is return the value stored in the root. BUT, after you find this value, you must also, fix up the heap. This means deleting the "last" node of the heap and finding a new spot for it in the tree.

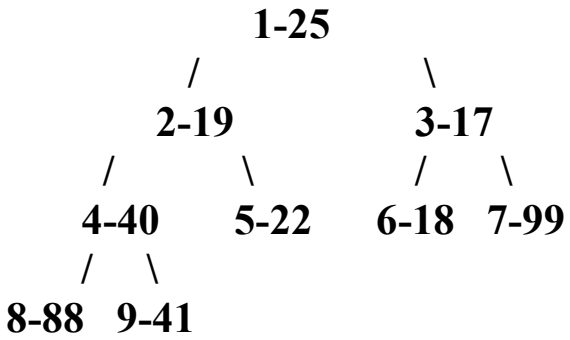
Temporarily place this "last" node in the vacated root of the tree. But, this is almost definitely not going to be the correct location for this value. Chances are one of its two children will be storing a value lower than it. If so, swap this value with the minimum of the two child values. This is a single "Percolate Down" step. Continue these steps until this "last" node has children both with larger values than it.

Here's how I like to look at this operation: Imagine that the CEO of the company as retired. So some hotshot upstart thinks they can take their place. But, quickly, the two most senior officers realize what has happened and try to rectify the situation. The higher ranking of the two takes the CEO position, relegating the hotshot upstart to their position. But soon again, someone realizes what has happened, once again demoting the hotshot upstart. This continues until the upstart has found a position in the company that he rightfully deserves.

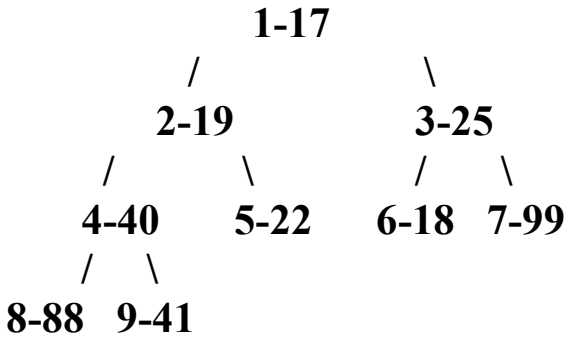
Consider deleting the minimum of the previous heap:



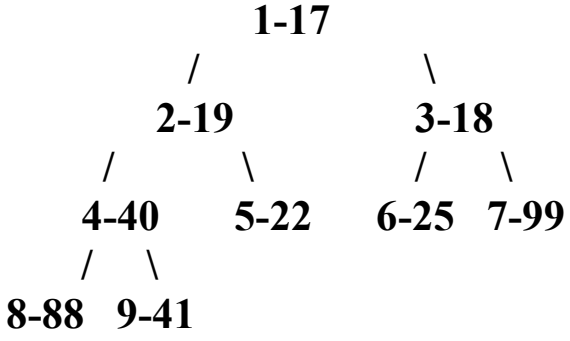
We store 15 to return, and then we must copy 25 into its position (index 1):



Now, we must compare 25 to both children, 19 and 17. Since 17 is smallest, it "wins." This means that we swap 17 and 25:



Now, we repeat: compare 25 to its new children 18 and 99, and we see that 18 is the smallest. So, we swap 25 and 18:



Now, there are no more children of 25, so we are done. It's also possible that this node that percolated down would have, at some step, been the smallest between itself and its two children (or possibly one child). In this case, the node has found its proper spot and we can return the minimum and complete the operation.

Heap Operation: Bottom-Up Heap Construction

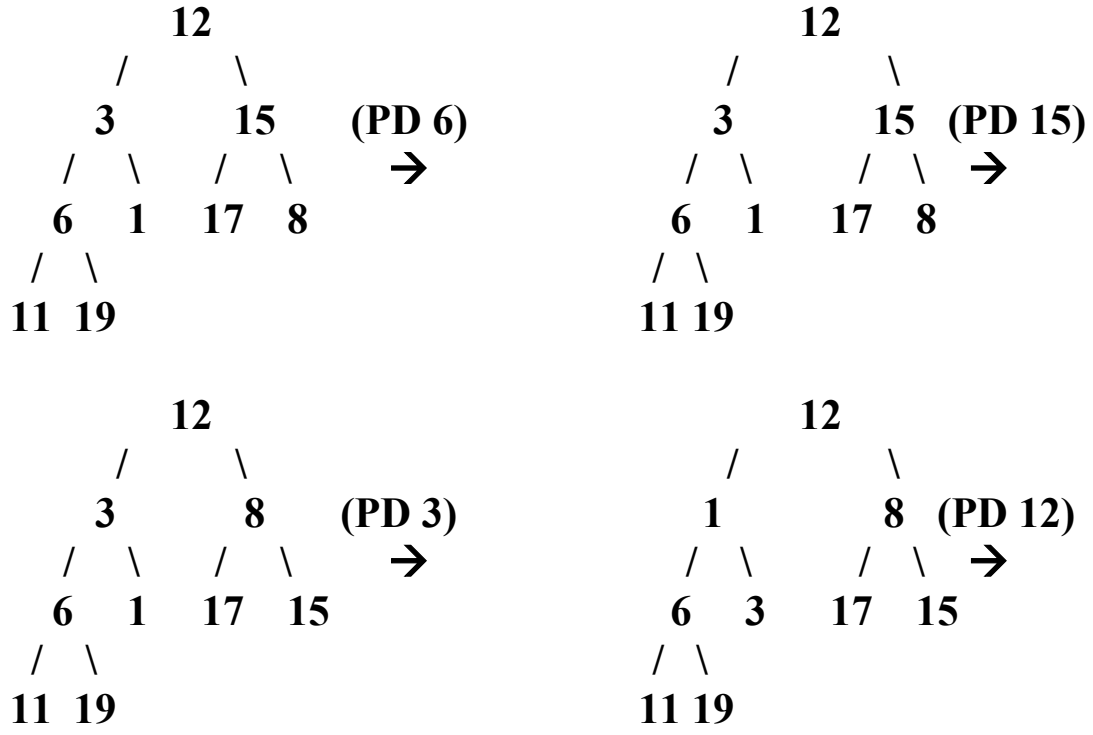
We will show how to construct a heap out of unsorted elements.

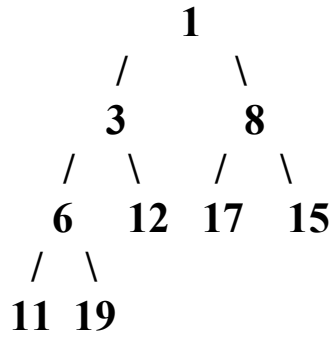
The basic idea is as follows:

1) Place all the unsorted elements in a complete binary tree.

2) Go through the nodes of the tree in backwards order (of how they are stored in an array implementation of a heap), running Percolate Down on each of these nodes. (Skip over all leaf nodes.) As this is done, one invariant we see is that each subtree below any node for which Percolate Down has already executed is a heap. Thus, when we run Percolate Down on the root at the end of this algorithm, the whole tree is one heap.

Here is a small example (Percolate Down is abbreviated PD):





In this last percolate down, first 12 swapped with 1, and then since 12 was greater than 3, it was also swapped with 3. (Note, we swap with 3 since 3 < 6.)

Can you see why we can NOT go through the nodes in forward order? Give an example where doing so produces a tree that is not a heap.

Now, let's analyze the running time of the algorithm to create a heap:

In a heap with n nodes, we end up running the "Percolate Down" on n/2 of those nodes. A very simple analysis would note that the maximum number of steps any of the Percolate Down's would take is O(log n). Thus, an upper bound for the running time of this Make-Heap function is O(nlogn).

However, a more careful analysis will yield a tighter upper bound for running time.

For the sake of simplifying the mathematics, let's assume we are dealing with a value of n such that $n=2^k - 1$, for some positive integer k. (This is the number of nodes in any complete binary tree of height k-1.)

In this tree, there are 2^{k-1} leaf nodes and 2^{k-2} parents of those leaf nodes. For each of these 2^{k-2} nodes, the Percolate Down can only traverse one step of the tree. For the 2^{k-3} nodes above those nodes, the Percolate Down can only traverse two steps of the tree. This argument continues until the last node, which can only traverse k-1 steps of the tree.

Let's make a chart with these numbers:

Number of Nodes	Depth of Percolate Down
2^{k-2}	1
2^{k-3}	2
2^{k-4}	3
...	...
2	k-2
1	k-1

Using this chart, we can write a summation that represents the run-time of the Make Heap function:

$$\sum_{i=0}^{k-2} 2^i (k-1-i)$$

Written-out, we have

$1(k-1) + 2(k-2) + 4(k-3) + ... + 2^{k-2}(1)$.

Let this sum be S. Now, multiply each term by 2 to get 2S:

$2S = \quad 2(k-1) + 4(k-2) + 8(k-3) + ... + 2^{k-2}(2) + 2^{k-1}(2)$

Subtract S from this:

$2S = \quad 2(k-1) + 4(k-2) + 8(k-3) + ... + 2^{k-2}(2) + 2^{k-1}(2)$
$S = 1(k-1)+2(k-2)+ 4(k-3) +8(k-4) + ... \quad +2^{k-2}(1)$

Subtracting, we get:

$S = -(k-1) + 2 + 4 + 8 + 16 + ... + 2^{k-2} + 2^{k-1}$.
$S = -k + 1 + 2 + 4 + 8 + 16 + ... + 2^{k-2} + 2^{k-1}$.

The latter terms in this sum form a geometric sequence with the sum $2^k - 1$. So, we have:

$S = -k + 2^k - 1$
$S = -k + n$, since we set $n=2^k - 1$ at the beginning.
$S = n - k = O(n)$.

Thus, Make-Heap runs in O(n) time.

Heap Sort

Now that we have determined how to execute several operations on a heap, we can use these to sort values using a heap. Here is the idea:

1) Insert all items into a heap
2) Extract the minimum item n times in a row, storing the values sequentially in an array.

Since each inserting and extraction take O(lg n) time, this sort works in O(n lg n) time. Let's trace through an example: