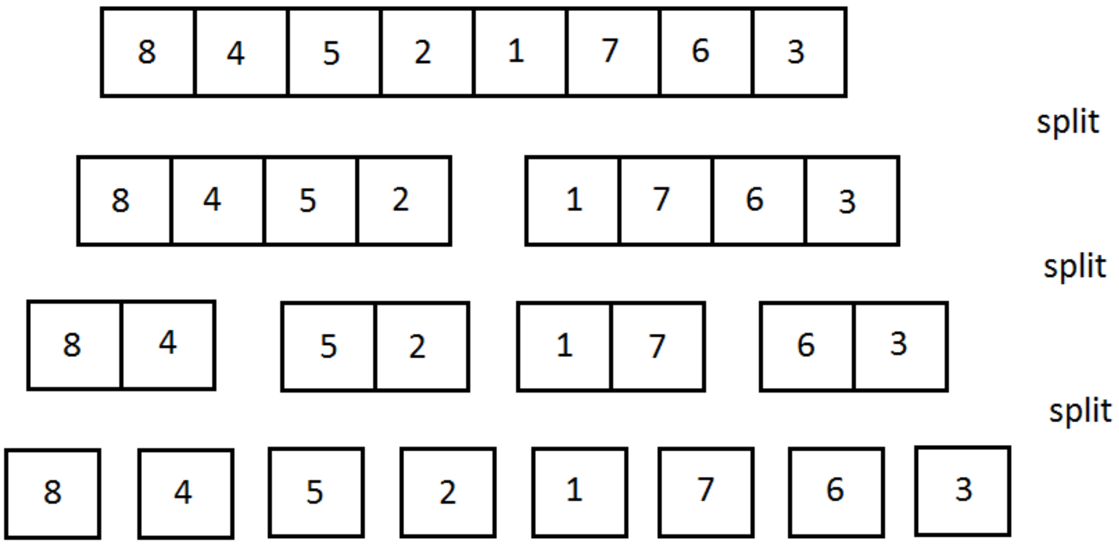
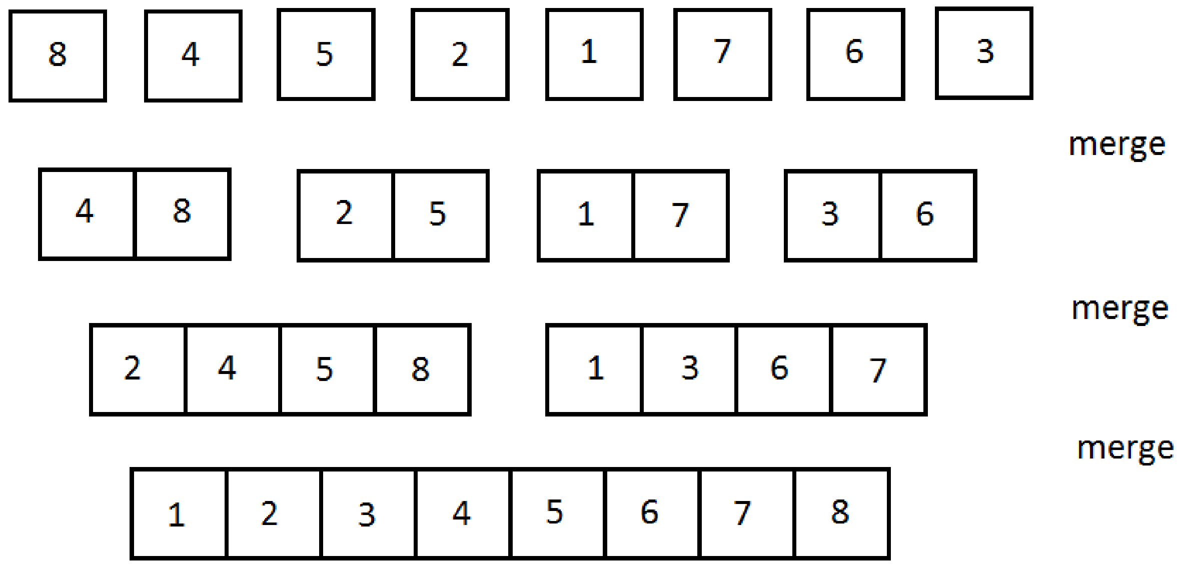
O(n log n) sorting algorithms

1. Mergesort.

Mergesort takes a very simple approach to sorting, in which it keeps splitting its array in half until the array has only one item. A single-item array is, by definition, sorted; nothing is out of order there. This step is pretty easy to program.



When all the arrays are of size one, we begin merging them into size-2 arrays that are sorted. This is the part of the program that takes a little more work to write.



Mergesort splits a size-n array in half log2(n) times. It then merges each array; merging 2 arrays of size n/2 to get one array of size n takes n comparisons. Just as there were log2(n) splits, there are log2(n) merges. The total time for mergesort is n log2(n). This is true in the best case and the worst case (unlike insertion sort or bubble sort).

The downside of mergesort is that each of the little merges requires us to make an extra array to hold the merged data. We copy this array back at each step to the main array and write the merges into the second array. The overhead of this extra array is pretty expensive, for large n.

HEAP SORT

Heaps: Read pp. 585-606 in the text.

Before heapsort, we must discuss the kind of binary tree that is a heap. Like binary trees, heaps depend on the order in which we add items. Unlike binary trees, heaps remain complete, which keeps them balanced.

1. BINARY TREE HEAPS

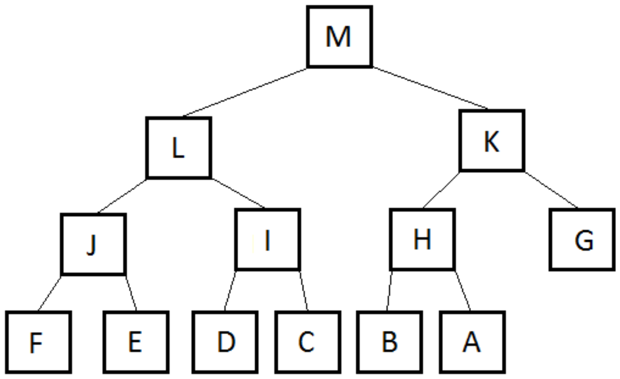
Binary trees can be organized as binary search trees, as you know. They can also be organized as heaps, using different rules.

Rule 1. Heaps (unlike binary search trees) must be complete trees. This means that:

All rows except the last one must be completely filled, and
The last row must be filled in from left to right with no gaps, like your ArrayBag's itemArray.

Rule 2. In addition, the data at every node of a heap must be greater than or equal to the data in each of its children. (By extension, this means that the largest item is at the root. Make sure you understand why.)

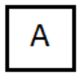
Here's an example of a heap: for this lecture, we assume that A < B and so on, which is pretty much the rule for alphabetical order.



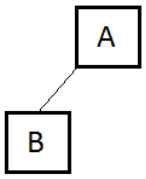
The reason we need to enforce completeness on the heap's binary tree is that when we write algorithms that use heaps, we end up actually using an array; the binary tree itself is usually imaginary. Like binary search trees, heaps also have a greater than/less than rule. For a maxheap, the usual case, a heap node's data is always greater than or equal to the data in its left and right children. And this holds true recursively for all the nodes in the heap. A minheap, by contrast, keeps the smallest item at the top of the heap. But we'll only talk about maxheaps here. So for us, a maxheap is a heap, and its biggest item is always at the root of the binary tree for the heap.

1a. INSERTING ITEMS INTO HEAPS

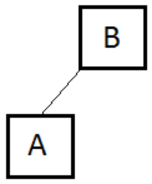
To insert into an empty heap, we just make a single node at the root. Booyah!



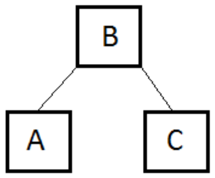
To insert into a non-empty heap, we insert the new item at the first open spot in the complete tree:



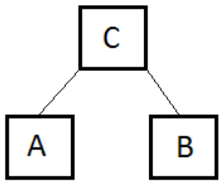
But now our new item might be breaking the heap rules. Is it bigger than its parent? If so, we have to swap it with the parent. (Assume that this swap is easy to do, for a moment. We'll come back to this.)



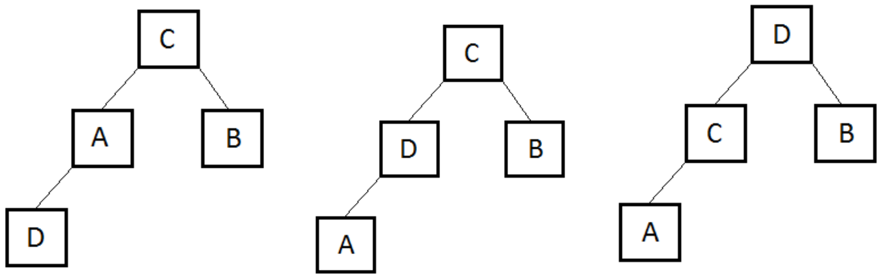
Add a third item to the heap in the next open row:



Now this third item is out of order with its parent, so we swap:



Each time we add a new item, we start it out in the next open spot in the lowest row, and then we swap that new item with its parent as many times as we need to for the heap to be valid.



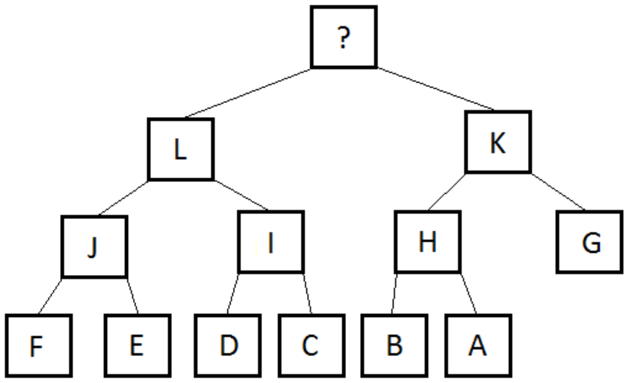
So what's the cost of all those darn swaps? Is it expensive? It's not so bad. It's limited by the height of the tree. If you remember how messy binary search trees could get, then maybe that restriction does not impress you yet. But remember that heaps are complete trees—they can't sprawl out into degenerate list-like structures. So the depth (or height) is minimized; for a tree with n nodes, the depth (or height) is log(n) + 1 at most. Most swaps of a new item are likely to stop earlier than the root, but (as we said) swapping up to the root is the worst case cost.

That's how we turn an O(n^2) problem into an O(n log(n)) problem. If we can add any number to a heap of size n and then swap it up to its rightful spot in log(n) time, then adding n items to an empty heap must be an O(n log(n)) process.

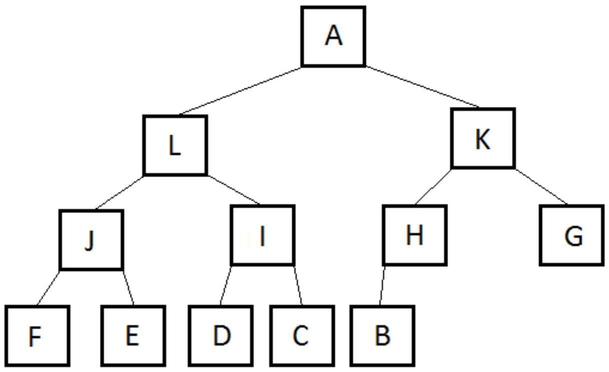
1b. REMOVING ITEMS FROM HEAPS

What about removing an item from a heap? We only do this when we want the biggest item. The biggest item is the one at the root, of course. Remember the recursive rule for heap comparisons.

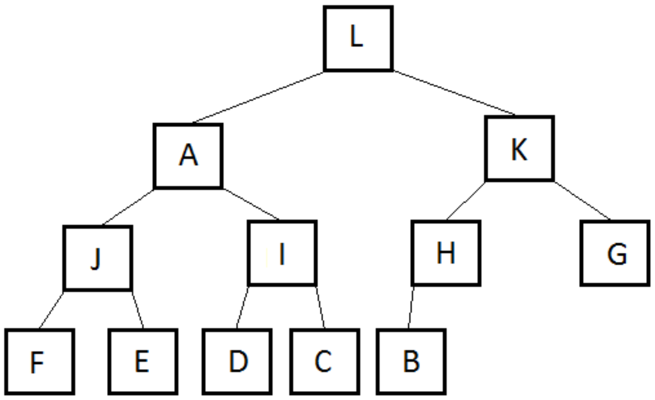
Removing the root definitely messes up our heap, though.



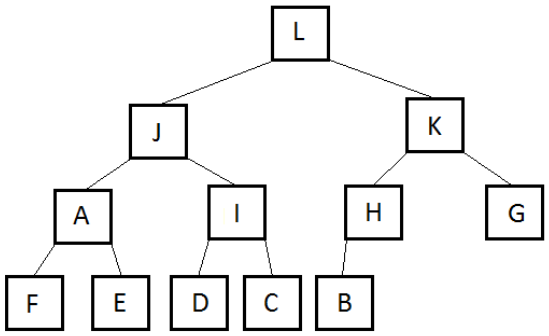
To start the repair, we move the child in the rightmost side of the bottom row up to the root. But that messes up the heap again. Now the root is smaller than both its children.

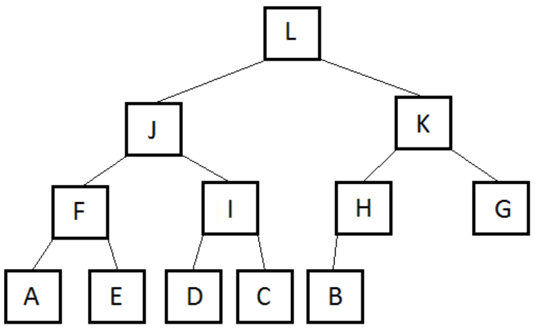


We can fix the root by swapping the root data with one of its children, here. It's important to pick the correct child, though. The root has to be bigger than both the kids. So we have to swap the root with its bigger child to preserve heapiness.



As before, we have to keep swapping along the tree until we have a valid heap again. In this example, two more swaps are still needed to restore the heap's order:

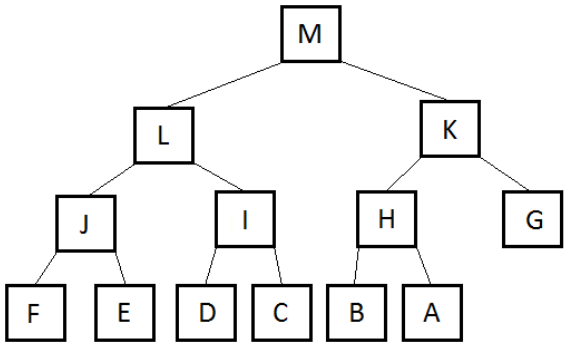




Again, the depth of the tree is the number of swaps we have to do, in the worst case (if we are unlucky each time with the swapping).

2. HEAPS AS ARRAYS

The reason we need to enforce completeness on the heap's binary tree is that when we write algorithms that use heaps, we end up really using an array; the binary tree itself is usually imaginary. See how this works:



Here is the corresponding array, filled in by writing the rows of the tree in from left to right:

M	L	K	J	I	H	G	F	E	D	C	B	A
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

The trick here is to write 3 functions for the array, so it can act like a complete binary tree. These guys do integer division (remember, that rounds down…)

```
unsigned int parent_index(unsigned int child_index)          { return (child_index - 1) / 2; }
unsigned int left_child_index(unsigned int parent_index)     { return 2*parent_index + 1; }
unsigned int right_child_index(unsigned int parent_index)    { return 2*parent_index + 2; }
```

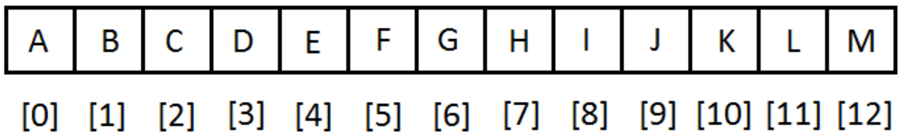
Now we can use the array items as heap node items. The root is at 0. The root's left child would be at 1. The root's right child would be at 2. The root's left child's left child would be at 3. The root's left child's right child would be at 4. Note that:

1. Reheapification upward relies on stopping the child-to-root swapping at the root index, 0.

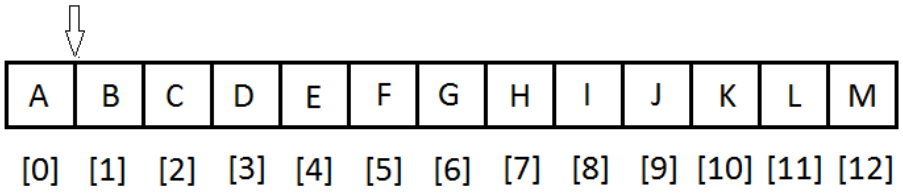
2. Reheapification downward relies on finding the larger of the left and right children (assuming you have them) and swapping with the larger child.

3. Each of these is limited by the size of the heap.

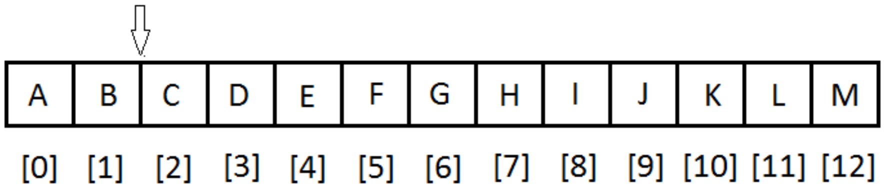
To sort arrays, you build a heap in array format in the first part of the array, adding and swapping as we go. Here is the starting array:



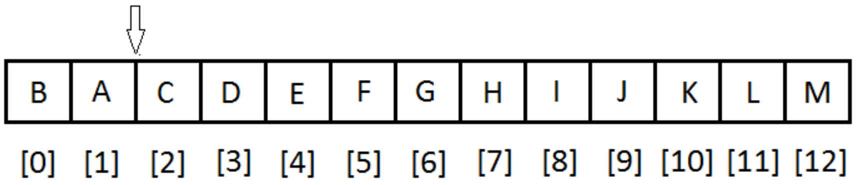
We define the first element as a tiny heap (containing just A).



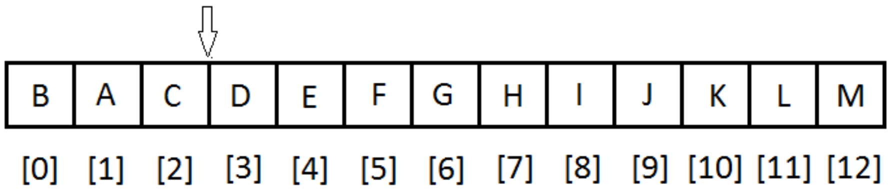
Next, we add the next item in the array, B to the heap, at the bottom row. Note that the arrow has moved:



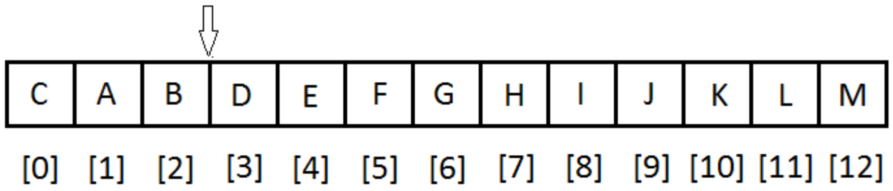
Now the heap part of the array is out of order, so we swap as in 1b:



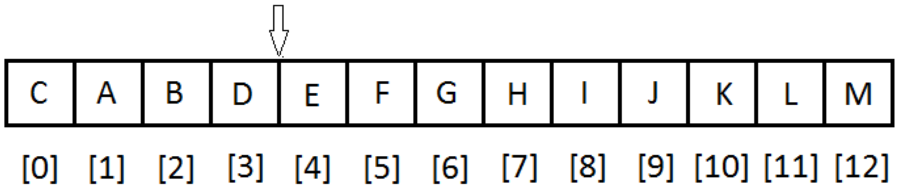
Next we expand the heap to include C:



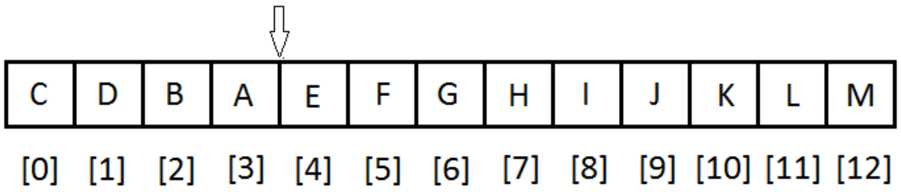
Next we swap C up to the root,



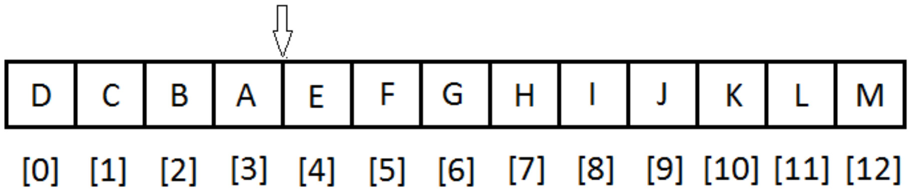
Next we add D to the heap,



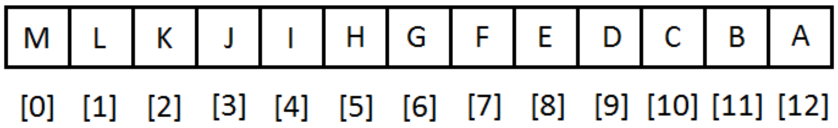
which gives us two swaps, the first between A and D,



and the second between D and C, putting D at the root:

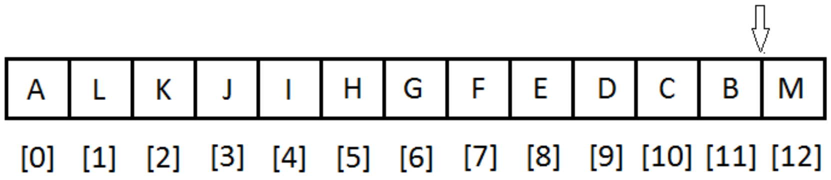


To remove from a complete heap, like the array here,



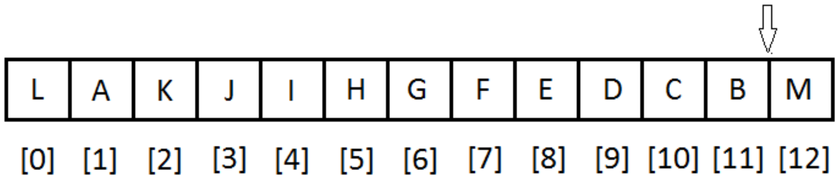
M	L	K	J	I	H	G	F	E	D	C	B	A
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

we move the first element (the root item of the heap) to the end of the array to start a sorted sub-array at the end of the array, and we move the last element from the last row of the heap to the root. This makes a bad heap, as in part 1b:

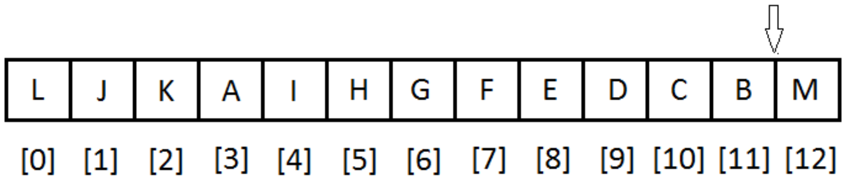


A	L	K	J	I	H	G	F	E	D	C	B	M
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

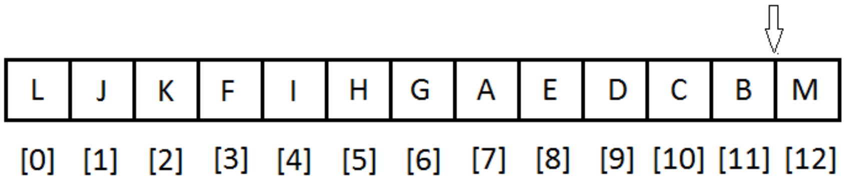
Then swapping down, in steps, gives us:



L	A	K	J	I	H	G	F	E	D	C	B	M
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]



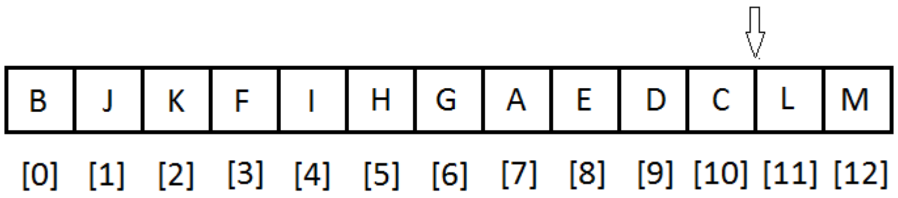
L	J	K	A	I	H	G	F	E	D	C	B	M
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

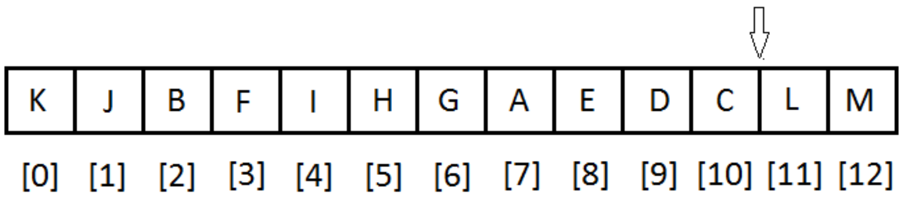


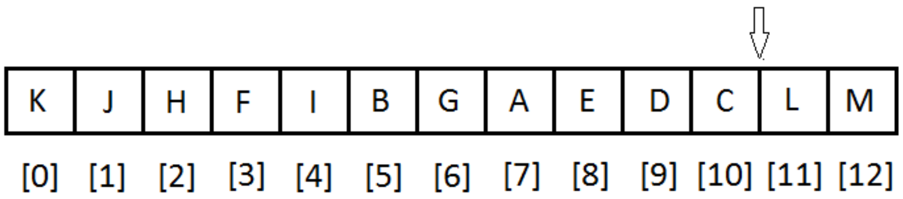
L	J	K	F	I	H	G	A	E	D	C	B	M
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]

Finally, we have a proper heap once again.

Similar steps put the L into place next and reorganize the heap.







Now, we have 2 numbers in place and can re-shuffle the heap again. Each pass where we remove the root element and put it into the right spot at the end of the array costs us log n swaps, as before. Overall, doing this n times means we have an algorithm with time O(n log n).

Heapsort is consistently O(n log n) in the worst case—as you can see, it would do better on a backwards sorted array (approaching O(n)), but at worst, it's O(n log n), which is reliable and good. Heapsort also sorts in place, unlike mergesort, and thus doesn't need extra memory to hold array copies. It's a little weird, because it imagines a binary tree but doesn't actually use a single tree node; all of that tree behavior is done by the array. But it's a powerful sorting technique that never has to rely on luck.