Heaps: Read Chapters 6.1-6.5 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.

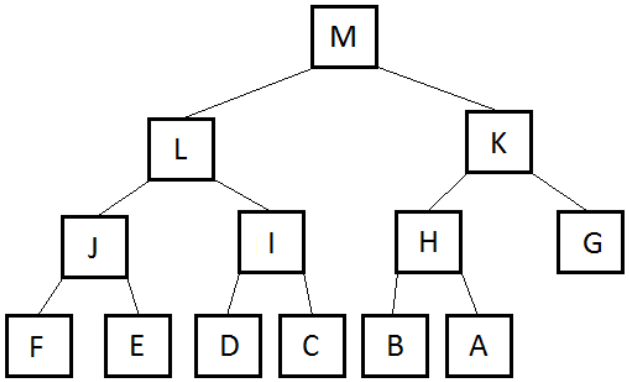
1. BINARY TREE HEAPS

Binary trees can be organized as binary search trees, as you know. They can also be organized as heaps. (These heaps have absolutely nothing to do with the memory region you call the heap.)

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 lab 3 and lab 4 arrays.

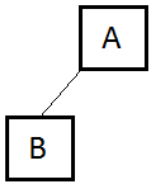


The reason we need to enforce completeness on the heap's binary tree is that when we write algorithms that use heaps, we end up really using an array; the binary tree itself is usually imaginary.

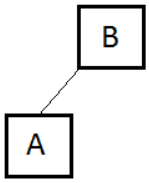
Like binary search trees, heaps also have a greater than/less than rule. For a maxheap, the usual case, a heap node's data is always greater than or equal to the data in its left and right children. And this holds true recursively for all the nodes in the heap. A minheap, by contrast, keeps the smallest item at the top of the heap. But we'll only talk about maxheaps here. So for us, a maxheap is a heap.

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

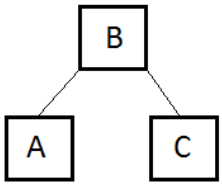
To insert into a non-empty heap, we insert at the first open spot in the complete tree:



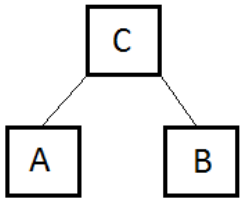
But now our new entry might be breaking the heap rules. Is it bigger than its parent? If so, we have to swap it with the parent. (Assume that this swap is easy to do, for a moment. We'll come back to this.)



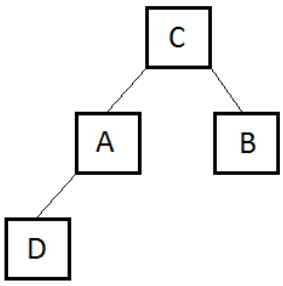
Now we'll add a new entry at the next open spot in the heap:



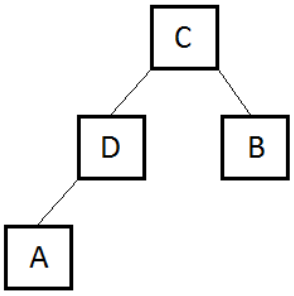
And we'll compare this entry to its parent. It's bigger, so we swap it with its parent…



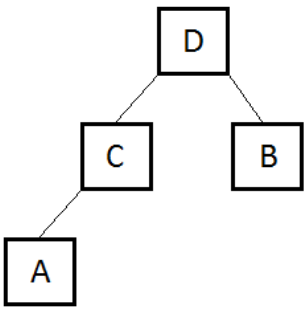
Now, we can add another entry…



Swap with the parent:



Are we done? No. Our swapped number is now out of order with its new parent. So we swap it with its parent again….



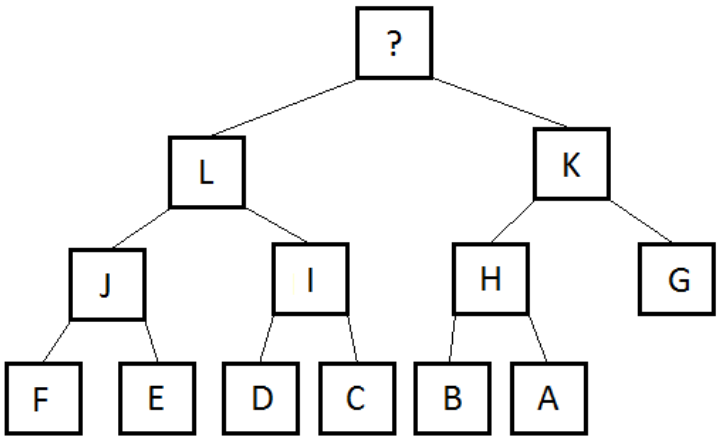
Eventually, all the swaps terminate (at the root node, in the worst case). Most of them are likely to stop earlier than the root, but swapping up to the root is the worst case cost.

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 is minimized; for a tree with n nodes, the depth is log(n) + 1 at most. That's how we turn an O(n^2) problem into an O(n*log(n)) problem.

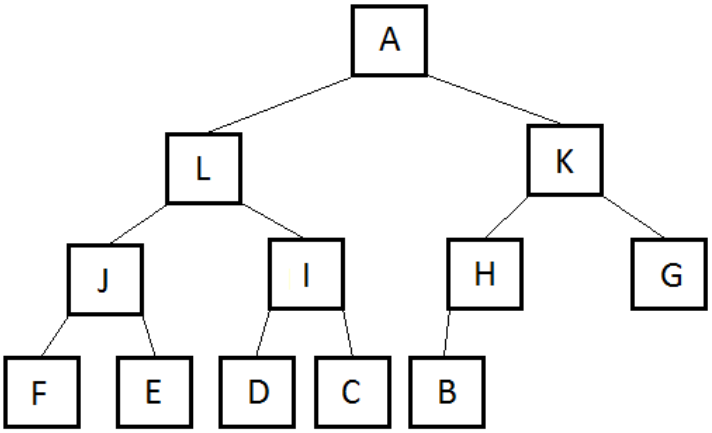
So we can add any number to a heap and then swap it up to its rightful spot in log(n) time, where n is the number of items in the heap. Adding n items to the heap is thus an O(n*log(n)) process.

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.

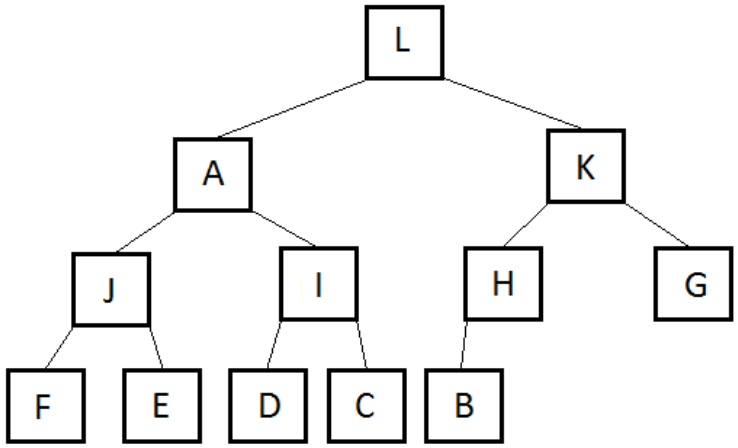
Removing the root definitely messes up our heap, though.



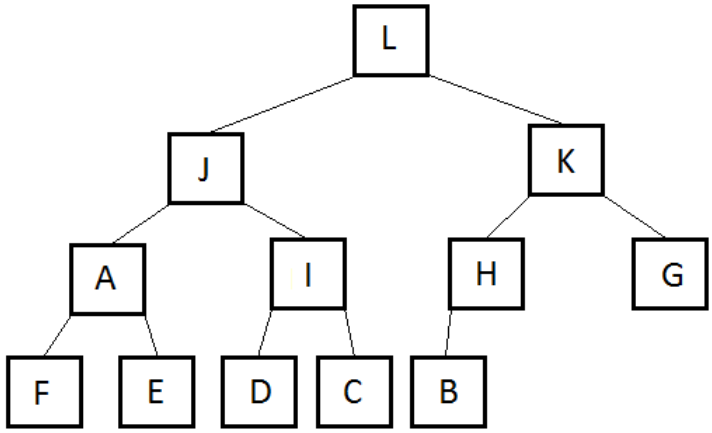
The only entry we can move up to the root here is the last entry in the bottom row. But that gives us a tree that's definitely not a heap:



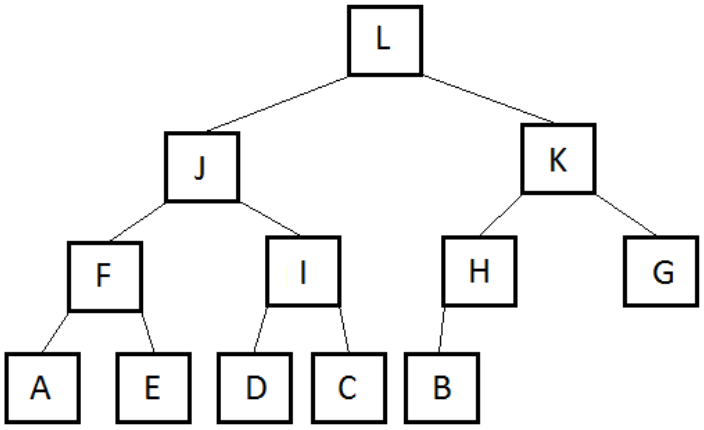
We can fix the root by swapping the root data with one of its children here. It's important to pick the correct child, though. The root has to be bigger than both the kids. So we have to swap the root with its bigger child to preserve heapishness.



Again we swap:

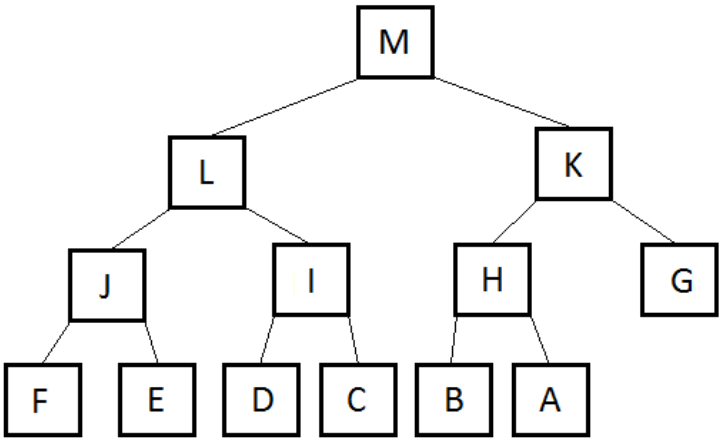


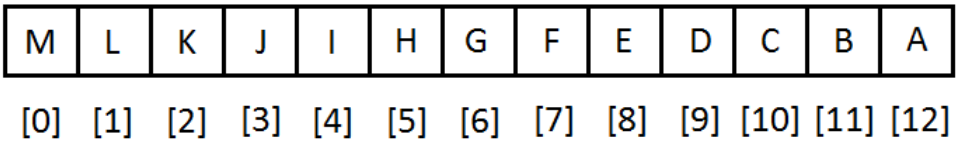
And so on, all the way down. Again, the depth of the tree is the maximum number of swaps we have to do, if we are unlucky each time. This is another O(n log n) process. At the end, the heap's restored.



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:





The trick here is to write 3 functions to find parent and child nodes for the array version, so it can act like a complete binary tree. These guys do integer division and round 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 3. Et cetera. 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.

We sort, then, by making a heap from all of our numbers. We remove the root element, and put it at the end of the array. We then move a child's data to the root and reheapify, giving us a new root element that is the next biggest entry in the array. Repeating this logic n - 1 times gives us an empty heap and a sorted array, in n log n time.