CS161L: Implementation of Algorithms

Friday, April 25, 2014

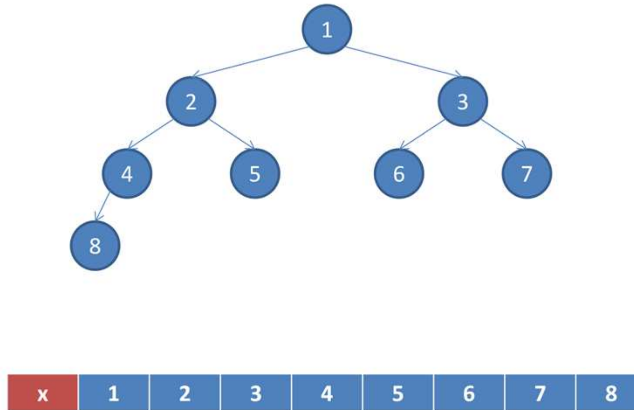
So on the survey, someone mentioned they wanted to work on heaps, and someone else mentioned they wanted to work on balanced binary search trees. According to the 161 schedule, heaps were last week, hashing is this week, and binary search trees are next week. However, balanced binary search trees are actually kind of a pain to implement, especially if we try to make it fit just in this class's time, so what we're gonna do instead is work on heaps this week to do something that balanced binary search trees are good for. Next week we'll work on a hashing problem.

Min Heaps

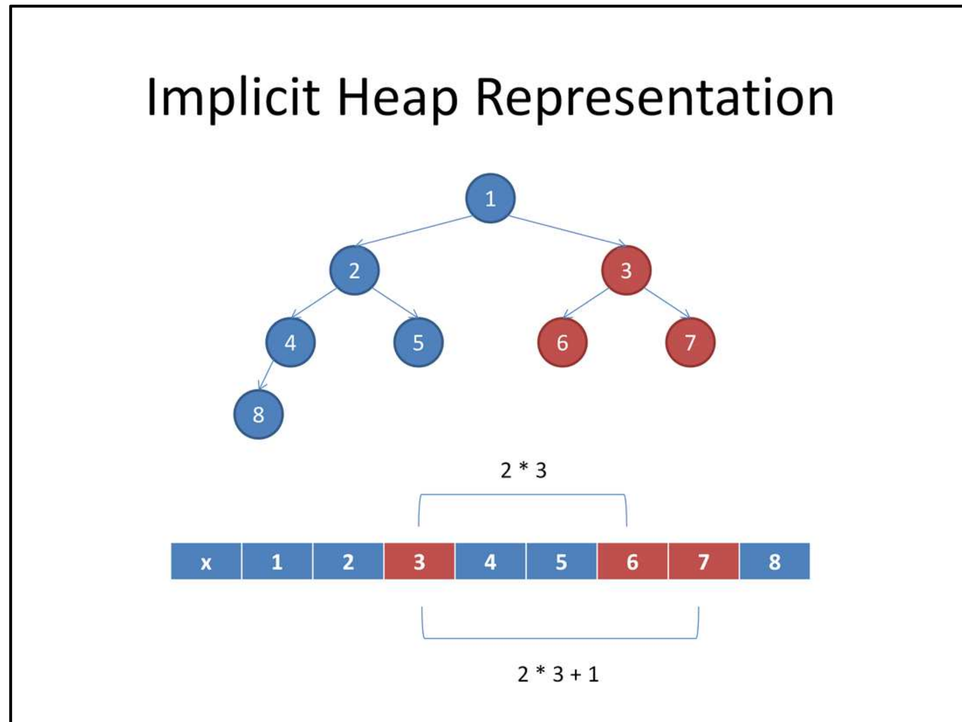
- Warning: 161 slides are all for MAX Heaps
  - Switch < and >
- Why min instead of max?
  - Prim's, Dijkstra's
  - priority convention (P0 bugs worse than P1)

First let's start with a regular heap, like the one discussed in 161. Well, not exactly the one in 161; those slides are all for MAX heaps, but the first problem for today is to implement a MIN heap. You can still use the 161 slides to help you out; all you need to do is remember to switch the direction of all the comparisons of heap elements. Now on the survey someone asked whether we could have more consistency between this class and the main 161 class. In general, I'd like to present the material in basically the same way, but I do want to flip a convention if it is better suited for the purposes of this class. So why are we swapping min and max for this assignment? Well, it turns out that the classical algorithms that you'll learn in this course that need a priority queue, such as Prim's algorithm or Dijkstra's algorithm, need a min priority queue, not a max priority queue, in order to work. The convention of lower priority numbers coming first is also fairly common in industry; for example, if you've ever worked with a bug tracking or feature request system, you might have noticed that P0 things are more important than P1 things, which are more important than P2 things. So in general, you're going to find that min priority queues are more common than max priority queues, so we're going to code up min heaps.

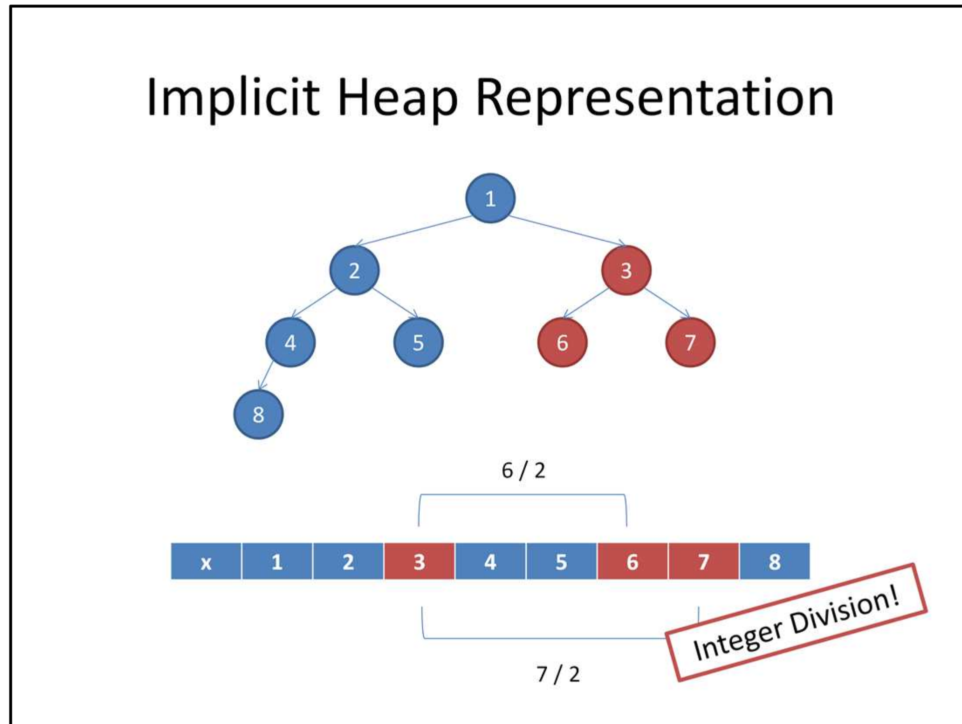
Implicit Heap Representation



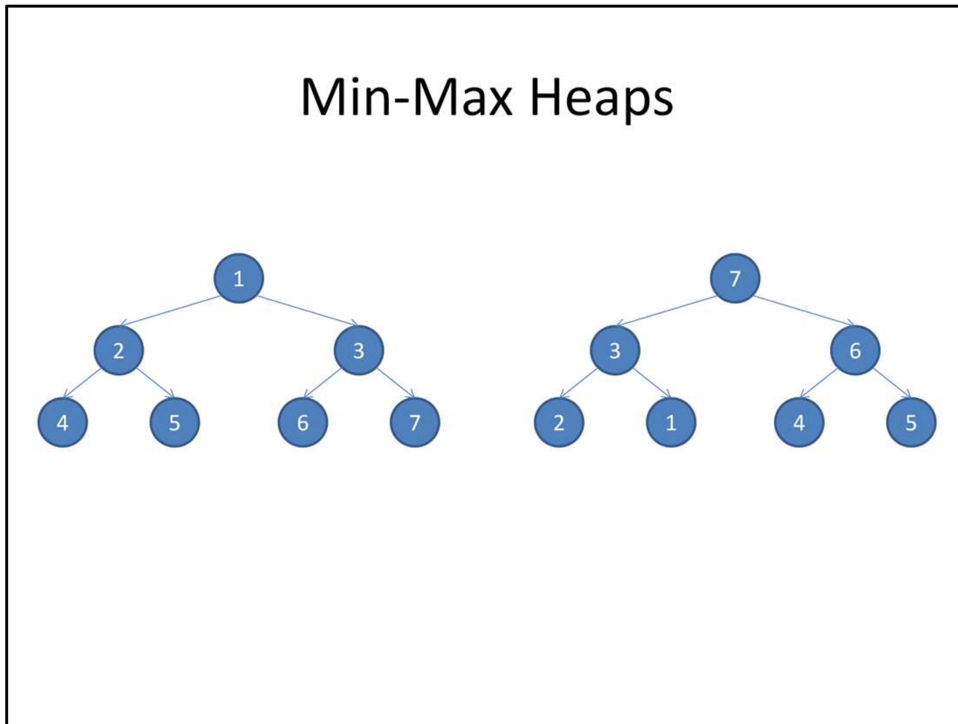
As mentioned in the main lecture, while it's easier to think of heaps as tree structures with pointers, when we actually implement them, we just use a single array, with the parent and child relationships being implied from the array indices. We'll see this trick in a couple different forms before the end of the class, so keep an eye out for that. Anyway, heaps are an odd beast in that they really do depend on one-indexed arrays, even though the languages we usually code them in have zero-indexed arrays. What this means is when you make a heap with a capacity of n, you're going to need to allocate an array of size n+1 and leave the first element blank.



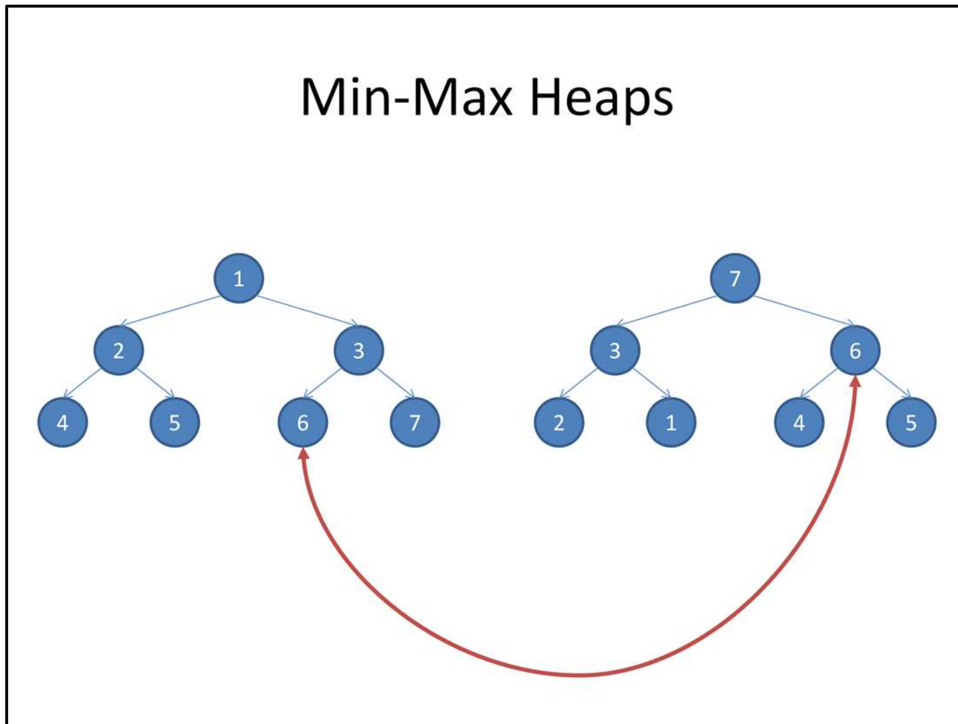
As a reminder, to access your children, you multiply your current index by 2 and add 0 for your left child, or 1 for your right child. A side note: As was pointed out on homework 2 problem 1, multiplication is actually a somewhat expensive operation, and multiplying by 2 is the same as performing a left bit shift of 1, which is a cheap operation (on par with addition). However, bit shifts make for difficult code to read, especially since for some reason they're really low in the order of operations, which would be a tradeoff of sorts EXCEPT for the fact that modern compilers can recognize when you're just multiplying by 2 and turn it into a bit shift operation at compile time anyway. So keep the multiplication for readability's sake, since you're not actually taking a performance hit at all.



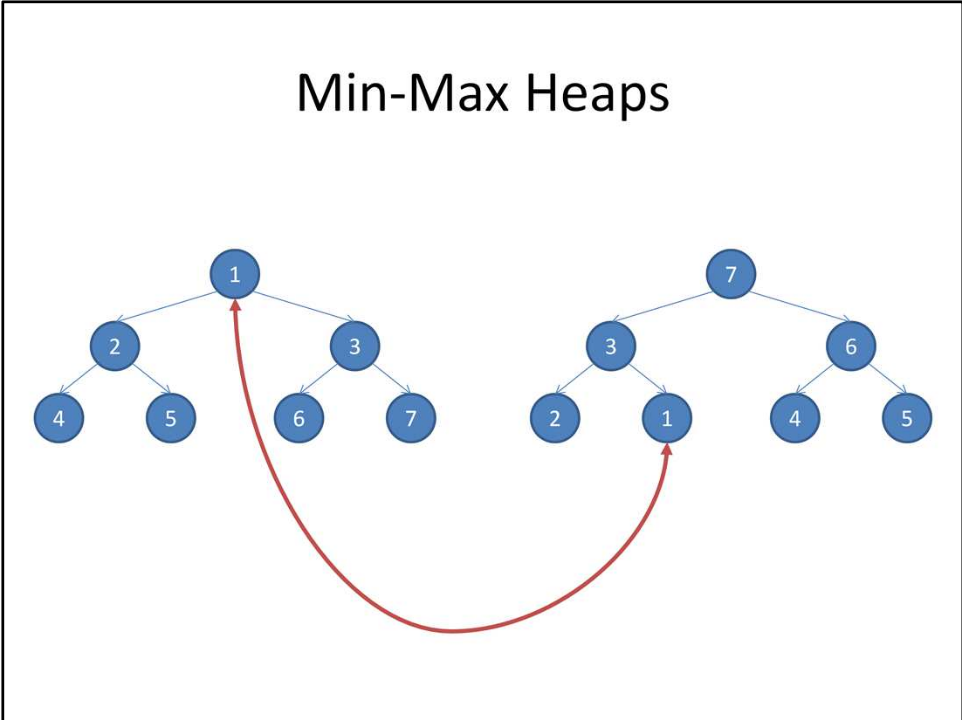
In the other direction, to find your parent, you divide your index by 2 and round down. Notice that you don't actually need to use a floor function; instead, you can simply rely on integer division to automatically round down for you. Again, this is also possible with a bit shift, but again, the compiler will take care of that for you. Now, there are extensive slides from the main course that explain what you need to do to insert a number into a heap, and how to extract the minimum element from the heap, so I'll refer you all to those slides.



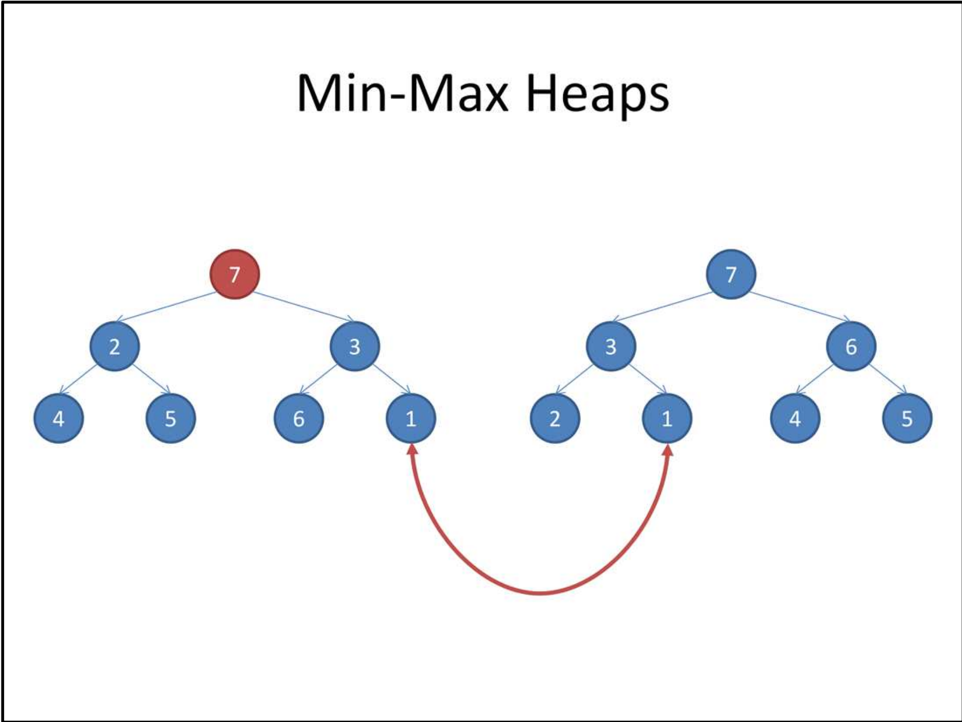
Let's move on to the main problem for today. A min heap lets you insert a number, and it lets you find and take out the smallest number you have, both in log time. A max heap lets you insert a number, and it lets you find and take out the LARGEST number you have, both in log time. Is it possible to do both, that is, to choose to take out either the smallest number OR the largest number, both in log time? Well, one way you could do this is with a balanced binary search tree, since finding the first and last elements in the tree each take log time. But can we do this with heaps, and avoid the huge constant factors associated with balanced binary search trees? What if we kept BOTH a min-heap and a max-heap? Insertion is easy; we just have to insert the number into both heaps. Extraction is harder; when we extract the min, it's easy to figure out what to do with the min-heap, but we need to also delete it from the max-heap, which means we have to find it in the max-heap first. Same goes for the other direction.



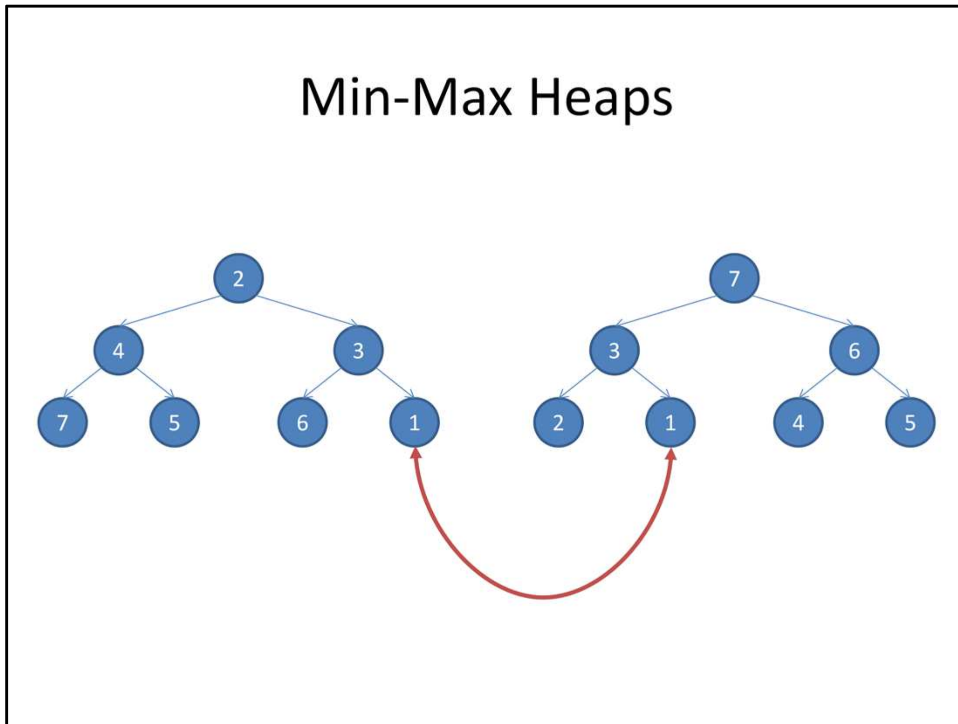
In order to make sure that we can quickly find the element in the other heap, we need to maintain a pair of pointers that links the two copies of the same element in the two heaps to each other. Now I'm drawing the heaps and these links as arrows to make them easier to visualize, but later on we'll see that we're still going to be using arrays, not pointers. This is important, since we want to take advantage of the performance boost we get from using flat arrays.



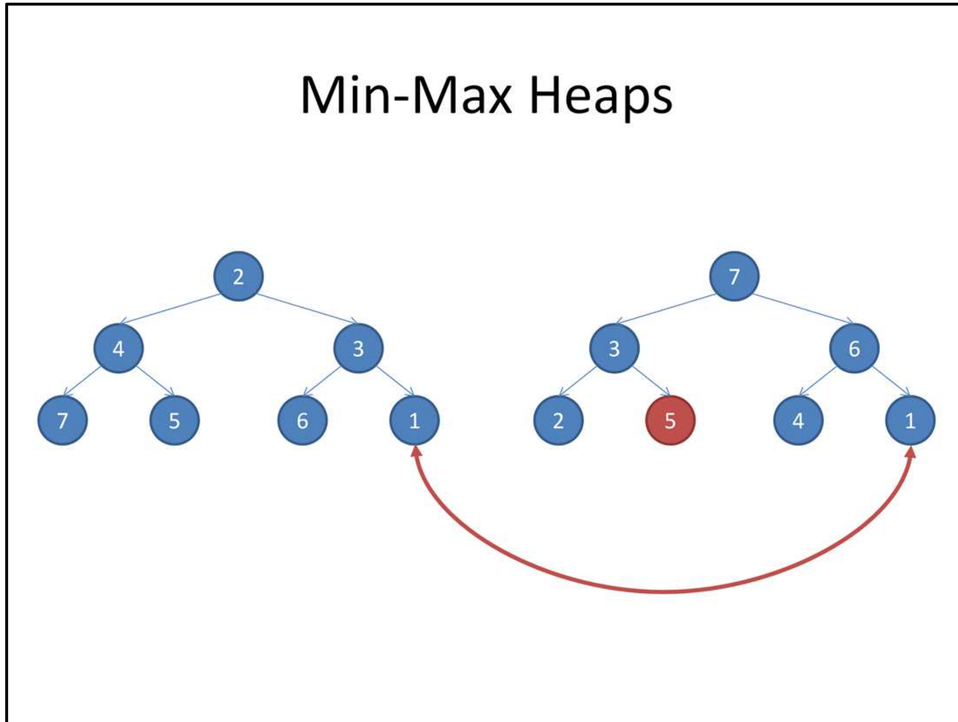
So how does this work? Well, let's walk through extract-min. The first part of the process works the same as for a single heap;



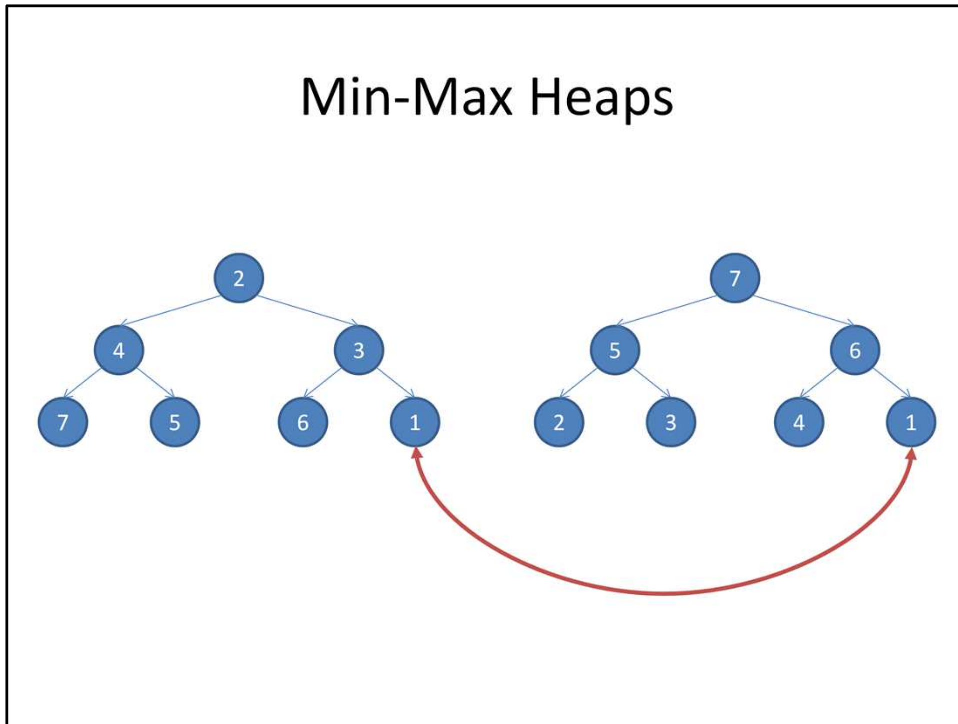
we swap the root of the min-heap with the last element in the min-heap,



and then we fix the min-heap up. If we had just one heap, we'd cut off and report back the last element now. In a min-max heap, though, we also need to remove this 1 from the max-heap. To do that,



we follow the pointer to find it in the max-heap, and then swap it with the last element in the max-heap. Now we're going to need to fix the max-heap by swapping that red element into the right place. Now remember that for the min-heap, the red element started at the top of the heap and bubbled down. To fix the max-heap, though, we're going to start at the bottom and bubble up. Why is it that we don't have to worry about bubbling down at all? (Because we swapped the red element with the smallest element in the max-heap, which can't have any descendants that are larger than it, so the red element, which is no smaller, also can't have any descendants that are larger than it.)



Once that's fixed, we can just chop the last element off of both heaps, and then we're done. So to summarize, to extract the minimum element, we do a regular extraction on the min-heap, which involves bubbling down to fix the heap, and then we follow the pointer to do a special extraction on the max-heap, which involves bubbling up to fix the heap. Similarly, to extract the maximum element, we extract from the max-heap, bubbling down, and then follow the pointer to extract from the min-heap, bubbling up.

Java and Pointers

- Actual pointers don't exist; use array indices
- Recommended: 4 arrays minHeap, maxHeap, minPtr, maxPtr
  - minPtr[i] = j means the element at position i in minHeap matches the element at position j in maxHeap
  - Check your swap function!

So all this time I've been talking about pointers, but this is a Java class, and Java doesn't have actual pointers you can manipulate. But I also said earlier that we wanted to use arrays anyway. For the individual heaps themselves, you already know how to treat them as an array. For the links between the heaps, I'm going to recommend storing an array that holds the indices of the elements in the other array. For example, if you have an array of pointers from the min heap to the max heap called minPtr, then minPtr of i being j means that the element at position i in the min heap matches the element at position j in the max heap. To make this easier to code, you should make sure your swap function correctly updates all of the relevant pointers in both heaps, even if you're just swapping elements in one heap. For this week only, I've provided you with starter code that handles the I/O, and also suggests a way of decomposing such a swap function. If you follow this suggestion, then your extraction code should look really similar to the extraction code you write for the regular min-heap.