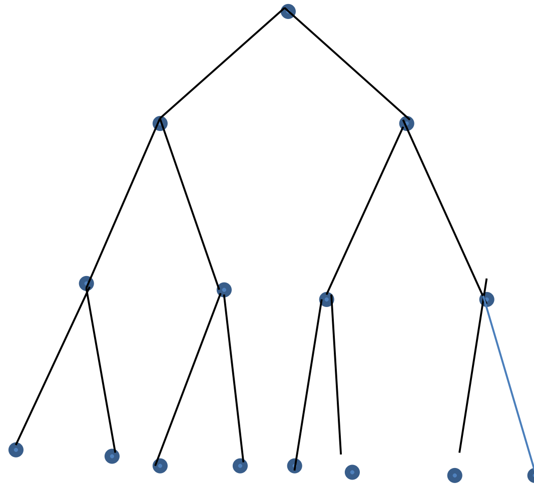
Priority Queues and Binary Heaps

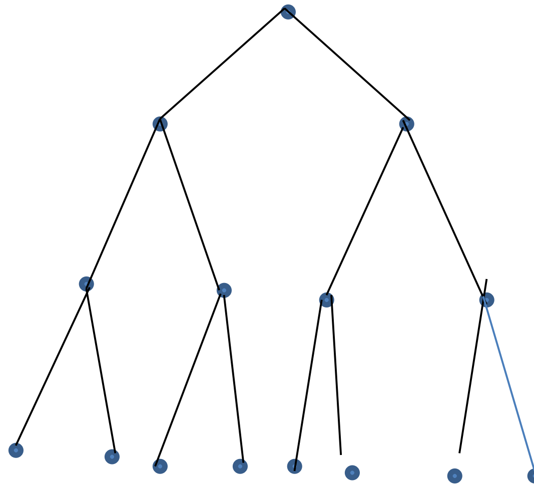
See Chapter 21 of the text, pages 807-839.

It is time to do some analysis of these operations. First, how tall are heaps?

An easy induction shows that a full binary tree with height H has $2^{H+1}-1$ nodes. Here is a full tree with height 3:



Note that it has $15 = 2^4-1$ nodes.



The bottom row of such a tree has 2^H nodes on it -- roughly half of the nodes are leaves. A heap of height h has anywhere from one of these leaves to all of them. This means that a heap of height H has between 2^H and $2^{H+1}-1$ nodes.

So $2^H <= N < 2^{H+1}$

If we take base-2 logarithms of everything this says

$H <= \log(N) < H+1$

So the height of a heap with N nodes is essentially log(N).

We insert a node into a heap by putting it as a leaf and letting it bubble up towards the root; the number of steps is bounded by the height of the tree, so this is a O(log(n)) operation. We remove the smallest value by replacing it with a leaf and letting it percolate down. Again the number of steps is bounded by the height of the tree, and deleteMin() is also a O(log(n)) operation.

Here is something surprising and cool. We can turn an array into a heap in linear time! We start at the leaves and work our way up. Of course, there is nothing to do at the leaves, they are already heaps. When we get to a node we will have already turned its leaves into heaps, so all that we need to do is to percolate the value at the node downward:

```
void buildHeap( ) {
        for (int i = size/2; i > 0; i--)
                percolateDown(i);
}
```

At each node we call percolateDown( ), so buildHeap( ) is bounded by the sum of the heights of all of the nodes in the tree.

Theorem: In a full binary tree of height H (containing $N = 2^{H+1}-1$ nodes) the sum of the heights of all nodes is N-H-1.
Proof: Weiss gives (p. 821) an edge-coloring proof. Here is a more analytical one. There are 2^H leaves of such a tree; these have height 0. The row above this has 2^{H-1} nodes of height 1. Above this there are 2^{H-2} nodes of height 2, and so forth.

The sum of the heights of all of the nodes is

$$S = 1 \cdot 2^{H-1} + 2 \cdot 2^{H-2} + 3 \cdot 2^{H-3} + 4 \cdot 2^{H-4} + 5 \cdot 2^{H-5} + \ldots + H \cdot 2^{0}$$

If we double this we get a similar sum:

$$2S = 1 \cdot 2^{H} + 2 \cdot 2^{H-1} + 3 \cdot 2^{H-2} + 4 \cdot 2^{H-3} + 5 \cdot 2^{H-4} + \ldots + H \cdot 2^{1}$$

Now subtract these:

$$2S - S = 2^{H} + 2^{H-1} + 2^{H-2} + 2^{H-3} + \ldots\ldots + 2^{1} - H \cdot 2^{0}$$

The left side is just S, the right side is a geometric sequence that we know how to sum:

$$S = 2^{H+1} - 2 - H$$

$$= N-1-H$$

A heap is not necessarily a full tree but it is a full tree of height H-1 with some additional leaves of depth H; we can derive a similar O(N) bound for any heap.

This means that we can construct a heap out of any array of n elements in time O(n).

While we are talking about heaps there is one more important application of them. One of the sweetest sorting algorithms is built on our percolateDown( ) method. This is called *HeapSort.* It sorts an array of size n in time O(n*log(n)) and uses no additional storage.

HeapSort requires a few changes in the way we think of heaps. For one thing, it uses maxHeaps, where the maximum element rather than the minimum element is stored in each root. For another, it indexes the heap starting at 0, so the children of the node at index i are at indices 2*i+1 and 2*i+2.

The idea behind HeapSort is really simple. We first make the array into a maxHeap, which only takes time O(n). We then go into a loop that pulls off the top element and puts it at the end of the array, and reduce the size of the heap by 1 so we don't consider this element part of the heap any more. We switch a leaf with the root and let this percolate down, rebuilding the array. The percolate operation takes time
O (log(n)) and we do it n times, so this is O(n*log(n)).

Since this is a slightly different heap construction, I'll rename the percolate method to percDown( ). It takes 3 arguments: the array, the index at which to start percolating, and the current size of the heap.

Here is the HeapSort algorithm in terms of percDown( ):

```
public static <E extends Comparable <? super E>> void HeapSort( E[] a ) {
        // build the heap
        for (int i = a.length/2 -1; i >= 0; i-- )
                percDown(a, i, a.length);

        //sort
        for (int i = a.length-1; i > 0; i--) {
                swap(a, 0, i);  // put the max of heap ai position i
                                // and the last leaf at the root
                percDown(a, 0, i);
        }
}
```

The next slide has the code for the new percDown( ), which works with maxHeaps. This is a line-by-line translation of percolateDown(), reversing the inequalities and starting the indexing at 0 rather than 1:

```
public static <E extends Comparable <? super E>> void percDown(E[] a,int hole,int size) {
        E value = a[hole];
        while (2*hole+1 <size) {
                int bigChild;
                int child1 = 2*hole+1;
                int child2 = 2*hole+2;
                if (child1 == size-1)
                        bigChild = size-1;
                else {
                        if (a[child1].compareTo(a[child2]) > 0)
                                bigChild = child1;
                        else
                                bigChild = child2;
                }
                if (value.compareTo(a[bigChild]) > 0)
                        break;
                else {
                        a[hole] = a[bigChild];
                        hole = bigChild;
                }
        }
        a[hole] = value;
}
```