Priority Queues and Binary Heaps

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

A *priority queue* is a queue-like data structure that assumes data is comparable in some way (or at least has some field on which you can base comparisons). You can only see or remove the smallest value in a priority queue.

Clicker Question: What do I do if I want a priority queue based on the largest rather than the smallest value?

A. Use a comparator that flips its values: if x < y have compare(x,y) return +1 rather than -1.
B. Multiply all of the values in the tree by -1.
C. Put the data in an array and sort it.
D. Put the data in an array and reverse-sort it.

There is varying terminology for priority queues. Here are the Java names for the standard operations. These differ from the names our text uses; the text we used to use for 151 had an even different set of names. The following are what we will use in Lab 7. As usual, this assumes that E is the base type of the structure.

int **size**( ): returns the number of items currently in the queue
boolean **offer**(E x) : inserts element x into the queue
E **peek**( ): returns the smallest element in the queue without changing the queue, or returns null if the queue is empty.
E **poll**( ): removes from the queue the smallest element in it and returns this element, or null if the queue is empty
void **clear**( ): removes all of the elements from the queue
Iterator<E> **iterator**(): returns an iterator for the queue
Comparator<? super E> **comparator**(): returns the comparator used for ordering the queue

We will add to these

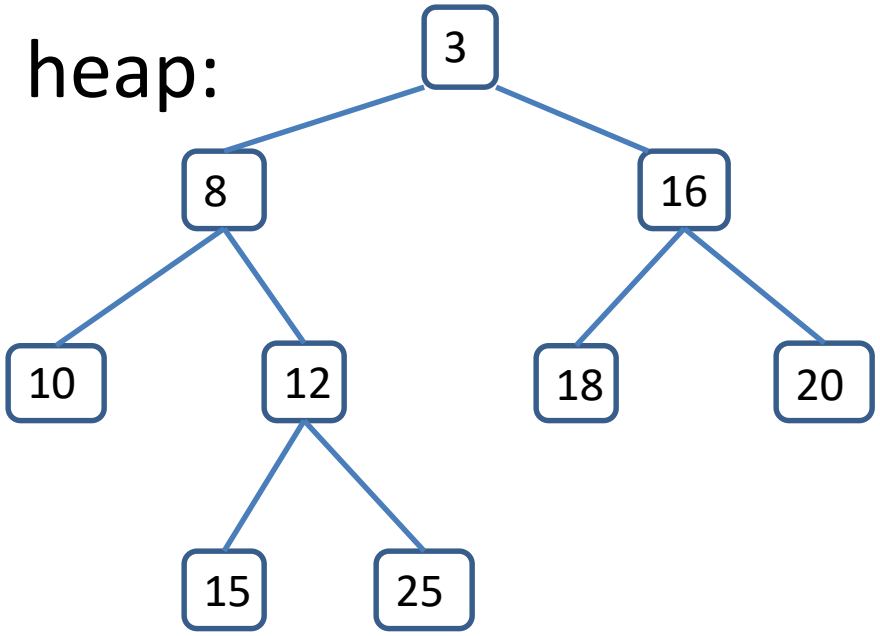
void setComparator(Comparator<E> cmp): installs a new comparator and reorders the queue.

It should not be surprising that priority queues are important. In many situations we do not need a complete ordering of our data; we just need to know what comes next.

For example suppose you are making a to-do list where some tasks are more important than others. Put the tasks in a priority queue organized by importance. The **offer** method adds a job to the queue. The **peek** method lets you see whatever is currently the most important job. When you are ready to do a job the **poll** method gives you the most important job and removes it from the queue.

Priority queues are often implemented in terms of Binary Heaps. A *heap* is a tree with the property that the value in each node is less than or equal to the values of its children.

Here is a picture of a heap:

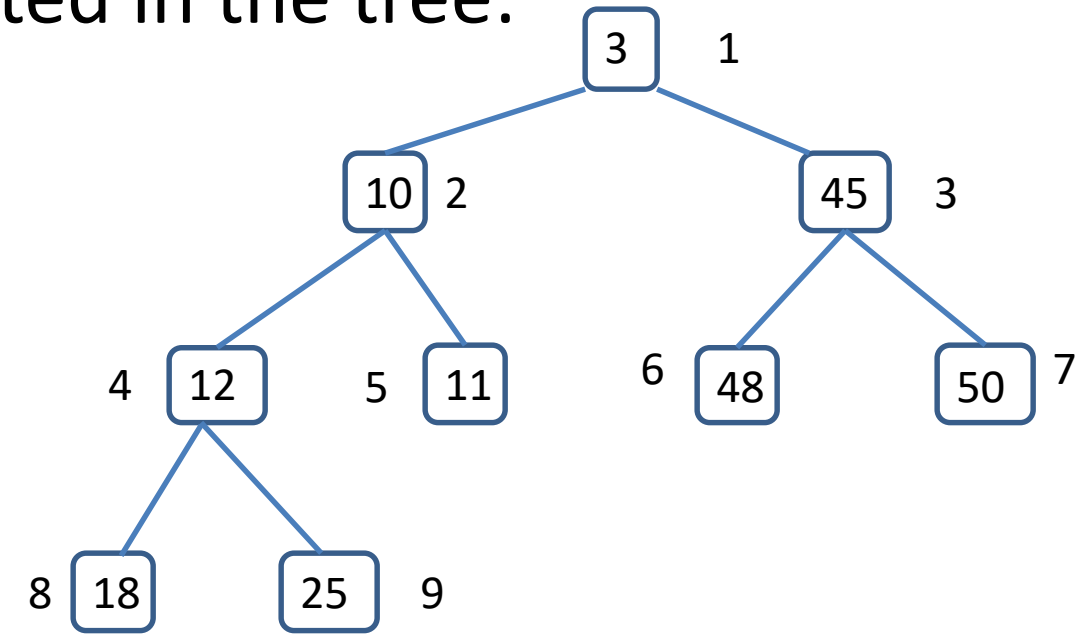


If we changed the 12 to a 6 it would no longer be a heap because this node would have a value less than its parent.

Note that in a heap the smallest node must be at the root. If the smallest value had a parent, it would violate the heap property because it would be a child with smaller value than its parent.

Binary heaps are often implemented in arrays, using a convenient indexing system for trees. We put the root at index 1. The two children of the node at index n are at indices 2*n and 2*n+1. Alternatively, the parent of the node at index i is at index i/2. If the tree is *complete*, meaning that every level except the bottom is completely filled and the bottom level has entries filled from left to right, then there are no gaps in the array.

Here is a picture of a complete heap and its corresponding array. The index of each node is also indicated in the tree:

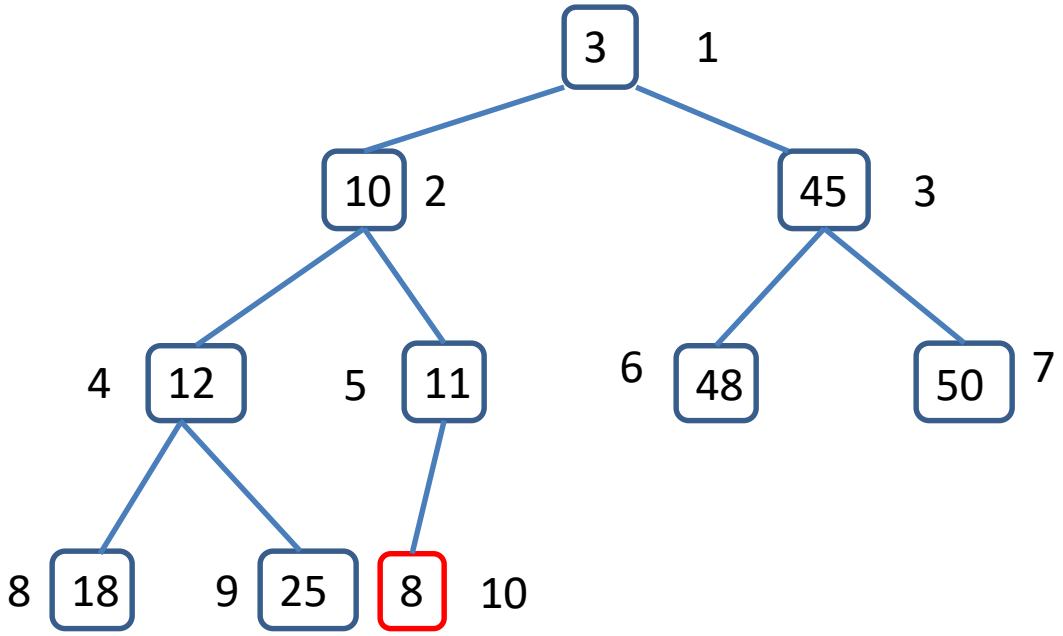


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	3	**10**	**45**	**12**	**11**	**48**	**50**	**18**	**25**	

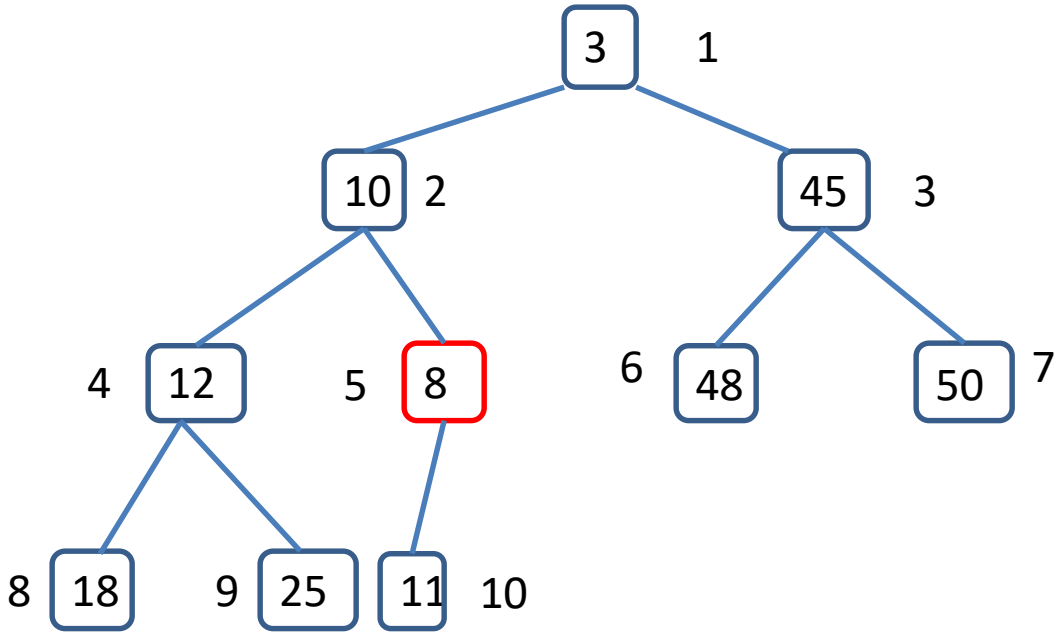
Note that a complete heap with N nodes uses entries 1 to N of an array of size N+1.

To insert an element into the heap we start by placing it at the next available spot. If the heap has n elements indexed from 1 to n, we put the new element at index n+1. If it has value greater than its parent node (index (n+1)/2), we are done. If not, we interchange it with its parent node and try again. The new value percolates up the tree until the heap property is satisfied.

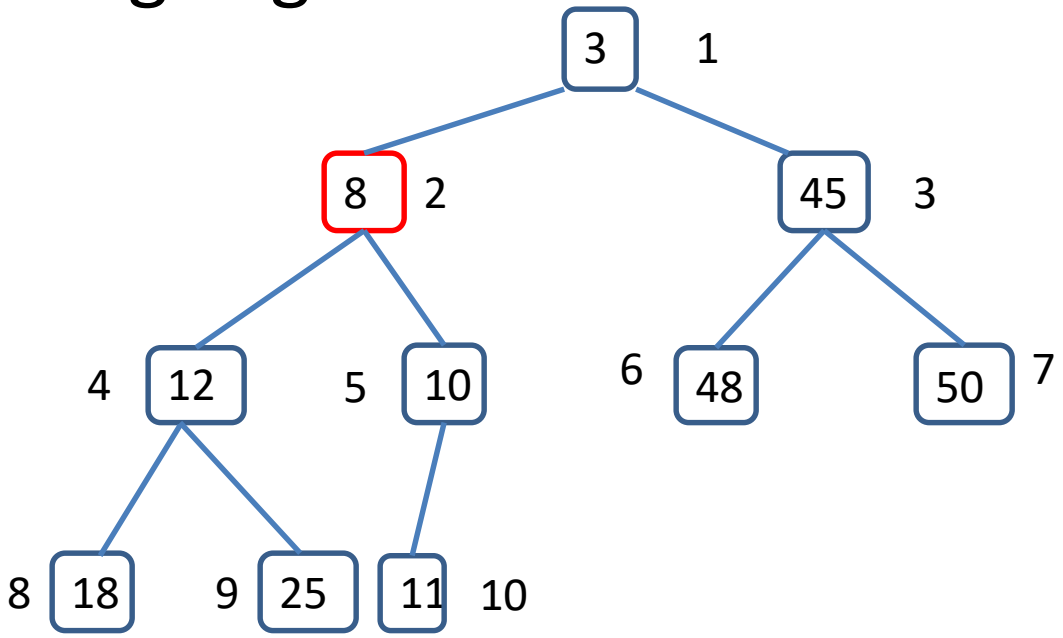
Here is an example. We add value 8 to the heap we just displayed. First we insert it as the next leaf:



The value 8 is less than that of its parent node so we interchange it with its parent:

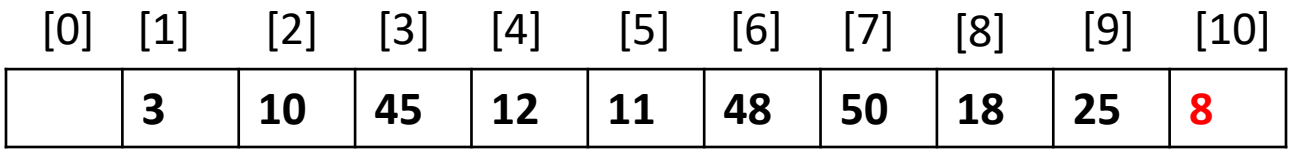


Its value is still less than its parent's so we interchange again:

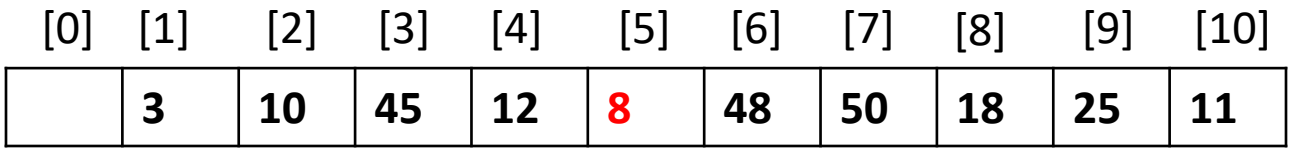


Our node now satisfies the heap property, so we stop interchanging and the entire tree is again a heap.

Here is this process in terms of the underlying array. We start by adding value 8 to the end:



The value we just inserted into index 10 is less than that of its parent at index 5, so we switch these values:



Our value at index 5 is still less than its parent at index 2 so we switch those:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	3	**8**	**45**	**12**	**10**	**48**	**50**	**18**	**25**	**11**

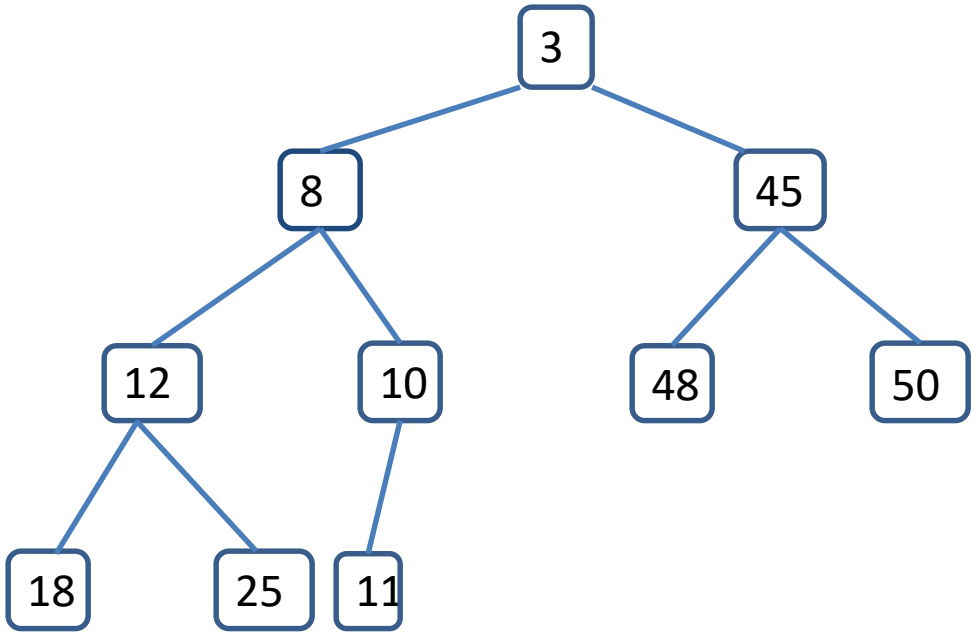
This now satisfies the heap property.

Here is code for this:

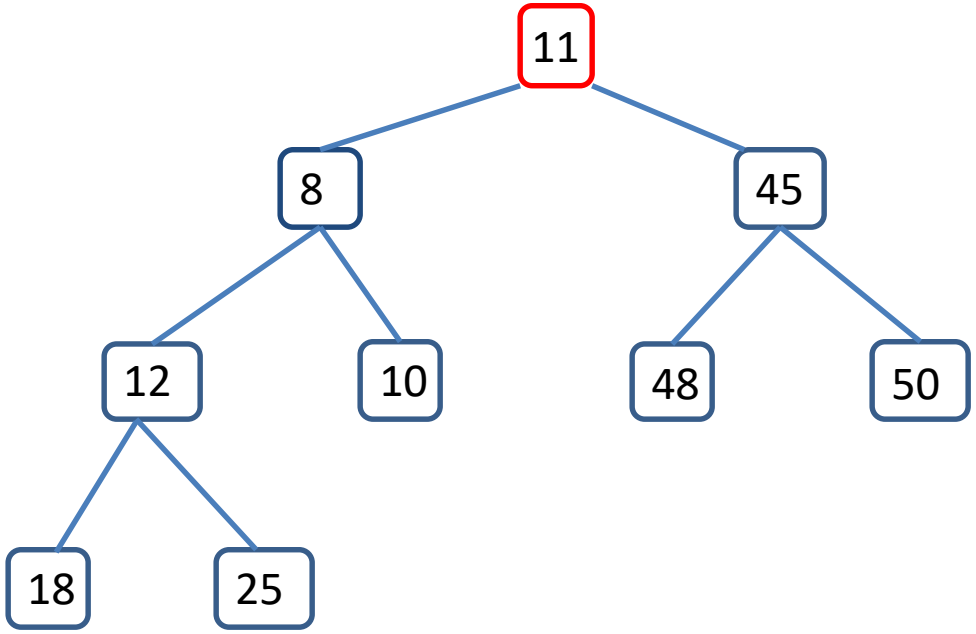
```
boolean add(E x) {
    if (size == CAPACITY)
            return false;

    int hole = size+1; // where we might put x
    size += 1;
    nodes[0] = x; // guarantees we will stop when hole=1
    while (compare(x, nodes[hole/2])< 0 ) {
            nodes[hole] = nodes[hole/2];
            hole = hole/2;
    }
    nodes[hole] = x;
    return true;
}
```

It is easy to find the smallest element; this is the first element of the array, or the root of the tree. We need an operation that removes the smallest element. In order to maintain a complete tree we must ultimately remove the last leaf, or last element of the array. We make a hole at the root and pass it down through the tree until we find where we can insert the value of the last leaf. We call this process "percolating down" the tree.

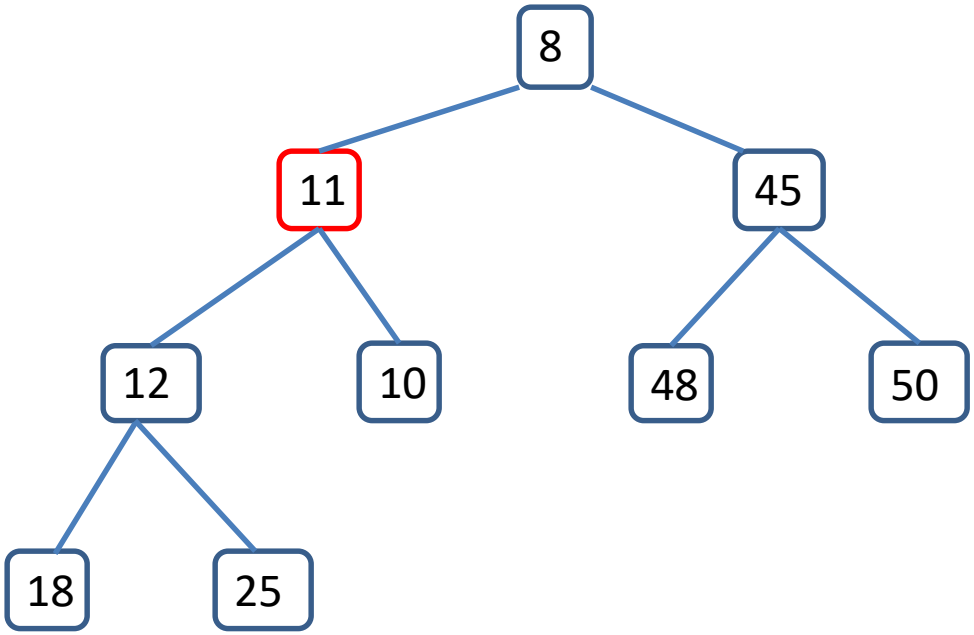
Graphically it looks like this. We start with a heap and want to remove the root.



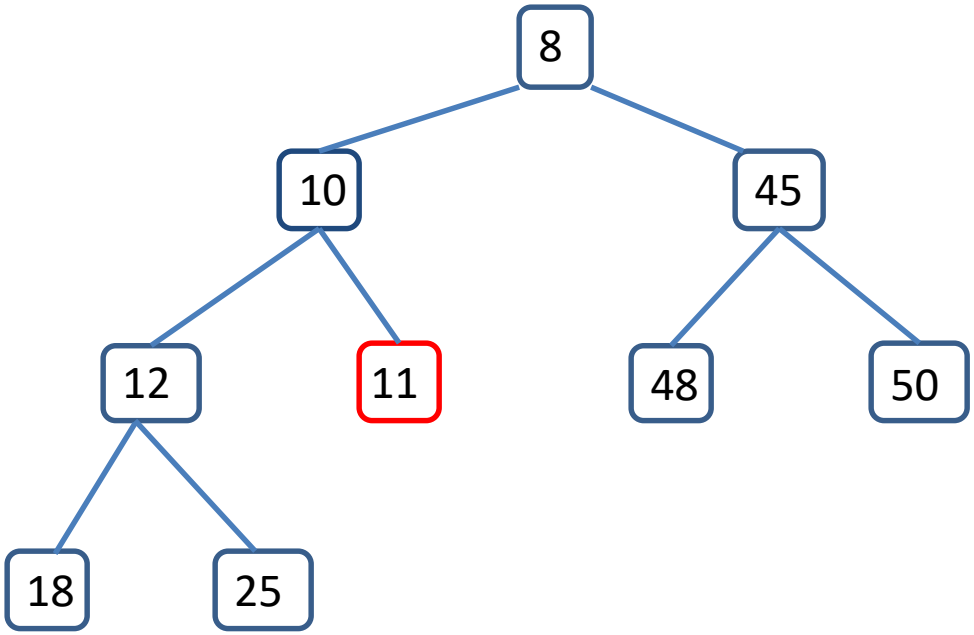
We make a hole at the root and put the value of the last leaf there, deleting the last leaf node.



The smaller of the two children of the hole has value 8; this is smaller than the value in our hole so we switch these items, moving the hole down:



Again the smaller of the two children of the hole is smaller than our leaf, so we move this smaller child into the hole:



This time the hole has no child so we are done.

We will divide the code for this into two steps. Here is the deleteMin() method:

```
E deleteMin() {
        E smallest = nodes[1];
        nodes[1] = nodes[size];
        size -= 1;
        percolateDown(1);
        return smallest;
}
```

This saves the value at the root, puts the last leaf into the hole at the root, and calls percolateDown() to pass the hole down through the tree. We separate out percolateDown() because it has other uses as well.

```
void percolateDown( int hole) {
    E value = nodes[hole]; // value being passed down
    while (2*hole <= size) {
        int smallChild; // index of smaller child
        if (2*hole == size)
            smallChild = size;
        else {
            if (compare( nodes[2*hole], nodes[2*hole+1]) < 0)
                smallChild = 2*hole;
            else
                smallChild = 2*hole+1;
        }
        if (compare(value, nodes[smallChild]) < 0)
            break;
        else {
            nodes[hole] = nodes[smallChild];
            hole = smallChild;
        }
    }
    nodes[hole] = value;
}
```

Here is the really cool part. 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);
}
```

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. The bottom row of such a tree has 2^{H} nodes on it -- roughly half of the nodes are leaves. A complete tree has anywhere from one of these nodes to all of them. This means that a complete tree of height H has between 2^{H} and 2^{H+1}-1 nodes. As a result, the height is the logarithm of the number of nodes in the tree.

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.

The surprising one is buildHeap(). For this we visit each internal node, starting with the lowest and working upwards. At each of these nodes 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 has 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 percolateUp( ) method. This is called *HeapSort.* It sorts and array 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;
}
```