Priority Queues and Binary Heaps

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

Question 1: What would you do if you had a very large unsorted collection of data values and you wanted to find the smallest value in the collection?

Answer 1: You have to look at all of the values, so there is really nothing you could do better that to walk through the whole collection comparing each value to the smallest value you have seen so far.

This just takes n comparisons.

Question 2: Now suppose that instead of the smallest value you need the 10 smallest values. What would you do?

Answer 2: You might keep an ArrayList of the 10 smallest values you have seen so far. Each time you see a new datum you have to find where it goes in this list.

This might tale 10*n comparisons.

Question 3: Now suppose that you need a bunch of the smallest values, but you aren't sure in advance how many you will need. And you might have to add in some new data after you have removed some. What do you do?

We can do better than just using lists. In time O(n) we can turn an ArrayList into a *Heap*. In time O(log(n)) we can remove the smallest element from the Heap (and restore it to being a heap. So to find the k smallest values takes time O(n) at the start and O(k*log(n)) to find the small values. Inserts also take time O(log(n)).

A Heap is a particular instance of our next data structure, the *priority queue*.

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.

We will assume the data has a Comparator: if the data has type T, then there is a function

```
int compare(T x, T y)
```

that returns -1 if x is “smaller” than y, 1 if x is “larger” than y, and 0 if they are “equal”

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 by -1.
C. Put the data in an array and sort it.
D. Put the data in an array and reverse-sort it.

Answer A: Flip the comparator.

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 8. 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

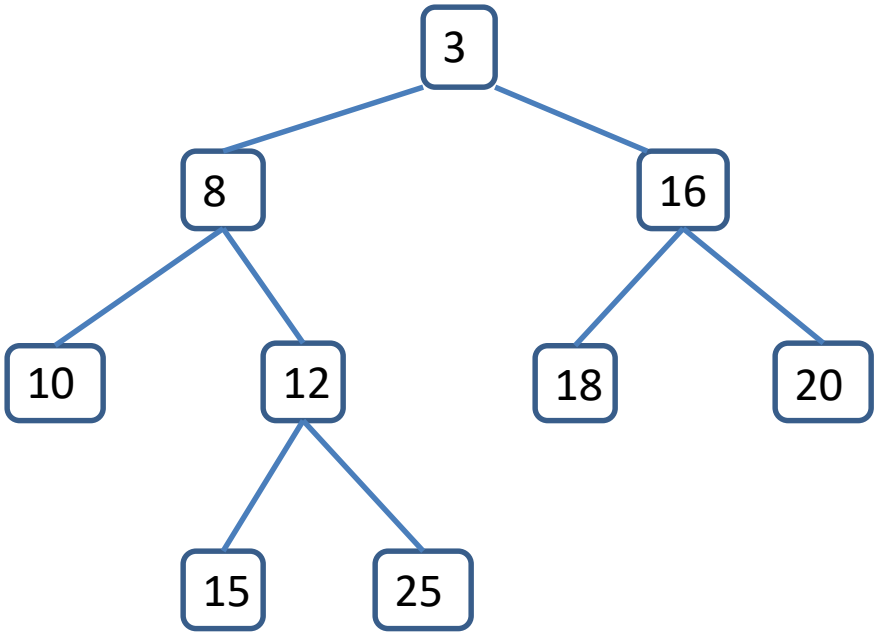
void setComparator(Comparator<E> cmp): installs a new comparator and reorders the queue.

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.