

# CS231 Fall 2013 Lecture 23

Stephanie R Taylor

October 28, 2013

## 1 Administrative Topics

- I will return the quizzes on Wednesday.

## 2 Priority Queues

A priority queue (PQ) is an interface with the following operations:

1. get the size of an existing PQ
2. add a new item in a PQ
3. remove an item from a PQ

What does the PQ interface look like in Java?

```
public interface PriorityQueue {  
    public int size();  
    public void add(...);  
    public ... remove();  
}
```

This set of signatures is not quite a complete definition of the PQ, since all container classes have these operations. For example, a set or even queue or stack have all these operations. In addition for a PQ we need:

- every item in a PQ must have a priority that is comparable to the priorities of the other items. (whaddya mean comparable? Well, integers are comparable, fruits are not.)
- The item removed by the remove operation is the one with the highest priority. Note how this differs from stacks, which remove the last one added, and queues, which remove the first one added.

Who cares?

- What do you do when you have some time to study? Which do you give the highest priority to? [CS of course]
- deadline-driven scheduling of HW or projects. You always work on (and remove) the HW or project with the nearest deadline.
- operating system with many tasks to perform.
- sorting (we'll see this Wed)

But first we need to understand what “priority” means. If you had an apple and an orange in your queue, which one has higher priority? Unfortunately, they are not comparable objects. We need comparable objects. Heres how well do it. All objects to be added to the PQ must have a compareTo method

```
public int compareTo(Object o)
```

that returns  $x > 0$  if **this** has higher priority than o,  $x = 0$  if they have the same priority, and some  $x < 0$  if o has higher priority than **this**.

More precisely, all objects added to a PQ must implement the Comparable interface:

```
public interface Comparable {
    public int compareTo(Object o);
}
```

If all elements in the PQ have such a method, then we can compare the elements to each other to find out which element has the highest priority and so gets to be removed next.

Here is more precisely what a PQ looks like:

```

public interface PriorityQueue {
    public int size();
    public void add(Comparable c);
    public Comparable remove();
}

```

Performance analysis of PQ implementations In order for PQSort to be a good sorting algorithm, we need to have fast efficient add and remove methods. Lets try to implement our PQ class so that they are both fast.

- (a) Implementation of PQs using an unsorted linked list:  
Adding is  $O(1)$  (we can just add to one end of the list). Removing is  $O(n)$  because we need to examine every element to determine which one has the highest priority.
- (b) Implementation of PQs using a sorted linked list:  
Adding requires a list traversal, which takes time proportional to the number of elements. On average, it takes  $n/2$  traversal steps, although it is easy to construct a worst case where every add takes  $n$ . Remove is constant time.
- (c) Implementation of PQs using an unsorted array:  
Actually we need more details, such as whether we are storing the items in the first part of the array and how often we have to copy. But, in general, adding is constant time, except that every once in a while we have to copy the whole array. Removing takes  $O(n)$  time.
- (d) Implementation of PQs using a sorted array:  
Adding is  $O(n)$  because we will likely need to insert into the middle of the array, and will have to shift the rest of the elements over, to make room. Removing is constant time.

### 3 Comparators

Variation: using a separate class to provide the comparisons The idea of expecting each of the objects in the PQ to have a compareTo method so that

the PQ can dump the responsibility of determining who has highest priority onto those objects is a real cool idea.

However, there are a couple of quite serious problems with this. Lets suppose there are two types of objects that want to be in a PQ:

1. An Apple class, each object of which has an integer diameter instance variable and the `compareTo` method compares its diameter to the other Apples diameter and the larger diameter one is the one with higher priority.
2. An Orange class, each object of which has a color instance variable and the `compareTo` method compares its color to the other Oranges color and the brighter one is the one with higher priority.

What if you add both an Apple and an Orange to a PQ? Whats going to happen when you tell the PQ to remove the highest priority item? [Have them look through the code and tell you. It will crash when the typecasting is attempted inside the Apple or Oranges `compareTo` method].

Heres another more serious problem. Suppose there are only one type of object to be included in the PQ, such as a Person object. It is easy to add a `compareTo` method to the Person class so that this class implements the Comparable interface: just compare two Persons last names and give the one that comes first alphabetically the higher priority. But what if you want one PQ to give the Persons priority based on the alphabetical ordering of the last names and another PQ to give the Persons priority based on the persons age? Since Person can have only one `compareTo` method, they cannot be compared in two different ways! Suggestions? [Have them guess.]

It is not generic enough to have each object have a `compareTo` method since different applications might want the same objects to be compared in different ways. So instead, separate the comparison from the objects. That is, the objects dont compare themselves, something else does. Does this sound like an improvement? At first glance, it sounds like we are regressing back to the case where the PQ has to determine the priority of the objects and so we are less generic than before. That is if the PQ does the comparisons, then the PQ is good only for priority queues that want to use that particular comparison.

Solution: Have an object, lets call it a Comparator, that is completely separate from the PQ and from the data objects, that is responsible for the comparisons. Once again we have an interface:

```
interface Comparator
{
    //returns 1 if o1 has higher priority,
    //returns 0 if both have the same priority
    //returns -1 if o2 has higher priority.
    int compare(Object o1, Object o2);
}
```

A comparator object is any object that has a method that compares any two objects and returns 1 if o1 has higher priority, 0 if they have the same priority, and -1 if o1 has lower priority.

A PQ uses a Comparator instance variable where it stores the Comparator object and a setComparator() method for giving the instance variable a value. The PQs remove() method asks the comparator to do the comparing whenever it needs to compare two objects to find the highest priority item. The neat thing is that you can change the Comparator when you want to stop comparing Person via their last names and start comparing them via their ages. Neat, huh!!!!

What does a Comparator do if it finds an item in the PQ that is not of a known or expected type? Suggestions? If a Comparator finds an object of an unexpected type, it can just always give it the lowest priority so that that object must wait at the end of the queue. Another completely different approach might have the Comparator return a special value indicating uncomparable.