PriorityQueue

A priority queue in java is implemented using a data structure called a heap. This lecture will not explain how a heap works but you can read up on that structure on your own time. Instead, we will explain the methods a priority queue contains and the order runtime of each method.

The purpose of the priority queue is to provide quick access to the smallest element in the queue based on some sorting. It works similar to a regular queue except additions with lower priority values are handled first.

For the most part in a priority queue you will be working with four main methods:

```
q.add(newValue);
q.poll();
q.peek();
q.size();
```

Add

The add method for a priority queue behaves in $O(\log n)$ runtime. The n in this runtime is the number of elements in the priority queue at the time of addition. Here log means log base 2. If you plug 1,000,000 in to the log function of your calculator, you will see this is not a large number of steps. This makes priority queues a nice choice for speeding up problems where you need to know either the largest or smallest element you would like to process next.

The purpose of this method is to add an element to the container. Note that the runtime of $O(\log n)$ is worse than the runtime $O(1)$. Essentially we are paying for the fast lookup of the smallest element by making adding elements slower. Sometimes this is a worthy tradeoff.

Poll

This method takes the smallest element in the queue based on each objects compareTo method and removes it from the queue. The method call takes $O(\log n)$ time as well as the structure must rearrange itself to figure out the next smallest element.

Peek

This method allows you access to the smallest element in the queue without removing it. It works the same as ArrayDeque's peek method and even runs in $O(1)$ time!

Example 1

If you want to try implementing this problem, you can find the problem description here on codeforces:

http://codeforces.com/contest/523/problem/D

In this problem we have **m** processors and **n** tasks, you will assign the first available processor to the first task as they appear. For each task, you want to print out the time the task will be completed.

Solution

```
// Read in the number of tasks and the number of processors
int n = in.nextInt();
int m = in.nextInt();

// First add all processors in the time they are available
PriorityQueue<Long> q = new PriorityQueue<>();
for (int i=0; i<m; i++)
   q.add(0L);

// For each task determine the next available process and print
// when the task will be completed.
for (int i=0; i<n; i++)
{
   int timeArrived = in.nextInt();
   int taskDuration = in.nextInt();
   long nextAvailable = Math.max(timeArrived, q.poll());
   long completion = taskDuration + nextAvailable;
   System.out.println(completion);
   q.add(completion);
}
```

Example 2

In this problem you want to read in a list of values. After reading in each value print the median of the list so far. There can be up to 10^5 values.

How can we use PriorityQueues to efficiently solve this problem?

Sets

Before learning about the set data structure, it is prudent to explore the concept of sets in mathematics. This way you understand the mathematical collection from which this data structure originates.

In mathematics, sets are unordered collections without duplicates. Objects that are contained inside sets are called elements of that set.

When writing out sets mathematically we can use the following notation (called the roster method):

$$\{apple, orange, peaches, grapes\}$$
$$\{1, 2, 3, 4, 5\}$$

Notice that the two following sets are equal under our definition:

$$\{1, 3, 2\} = \{2, 1, 3\}$$

Two sets A and B are equal if the elements in A are all contained in set B and all elements of set B are contained in set A. Also notice that each element may be contained only once in the set. We cannot have the following set:

$$\{orange, orange\}$$

There are some other useful definitions to know about sets. A set A is called a subset of set B if all elements in set A are also in set B. We use the following notation to denote subset:

$$A \subseteq B$$

An alternative definition of set equality is $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

Another piece of notation from set theory is the element of symbol. We use this to denote when one element is contained in a set. So the following is a true statement:

$$apple \in \{apple, peach, orange\}$$

Sets also have a few handy operations:

Set Union

Given two sets A and B, the union of those sets is defined as the set containing all elements in either A or B. Union is denoted $A \cup B$.

$$\{apple, orange, peach\} \cup \{apple, tomato, carrot\} = \{apple, orange, peach, tomato, carrot\}$$

Set Intersection

Given two sets A and B, the intersection of those sets is defined as the set containing all elements in both A and B. Intersection is denoted $A \cap B$.

$$\{apple, orange, peach\} \cap \{apple, tomato, carrot\} = \{apple\}$$

Set Difference

Given two sets A and B, the set difference of those sets is defined as the set containing all elements in A but not in B. Set difference is denoted $A - B$.

$$\{apple, orange, peach\} - \{apple, tomato, carrot\} = \{orange, peach\}$$

Set theory goes much deeper than this but a brief introduction to these concepts will greatly aid in your understanding of not only the data structures below but more advanced concepts based off Set Theory such as Inclusion-Exclusion, the Disjoint Sets data structure, and brute force concepts enumerating sets.

TreeSet

TreeSets are useful is representing ordered sets. Ordered sets are subject to some natural ordering. Each object you add to a TreeSet must implement the Comparable interface including the compareTo method.

Useful methods

```
ts.add(val);     // Adds the specified value to the TreeSet
ts.remove(val); // Removes the specified value from the TreeSet
ts.contains(val); // Returns if the specified value is in the TreeSet
ts.first();     // Returns the smallest element in the TreeSet
ts.last();      // Returns the largest element in the TreeSet
ts.pollFirst(); // Returns and removes the smallest element
ts.pollLast();  // Returns and removes the largest element
ts.lower(val);  // Returns the first element strictly lower than val
ts.higher(val); // Returns the first element strictly higher than val

ts.floor(val);     // Returns the first value ≤ val
ts.ceiling(val);   // Returns the first value ≥ val

// Keeps all elements shared between ts and other collection
ts.retainAll(otherCollection);

// Remove all elements shared between ts and other collection
ts.removeAll(otherCollection);

// Checks if ts contains all elements of other collection
ts.containsAll(otherCollection);
```

The methods {**add**, **remove**, **pollFirst**, **pollLast**, **lower**, **higher**, **floor**, **ceiling**} all behave in $O(\log n)$ time. The methods **first** and **last** run in $O(1)$ time.

Now that you have learned some of the TreeSet methods, can you figure out how to implement the operations union, intersection and set difference easily? What about properties like subset or element of?

Let's say you have a set of numbers, can you determine the closest number in the set to some query number quickly using the methods above?