RandomQueue

2015-02-13, rev. 720f78e

Build a data structure that supports insertion, deletion of a uniformly random element, and iteration in random order. (This is basically exercises [SW] 1.3.35 and 1.3.36.) Besides the algorithmic content this exercise tests several aspects of good programming practice (in particular, abstract data types, generics, and the iterator design pattern).

Description

Write the class RandomQueue with the following API:

```
public class RandomQueue<Item> implements Iterable<Item>

public RandomQueue() // create an empty random queue
public boolean isEmpty() // is it empty?
public int size() // return the number of elements
public void enqueue(Item item) // add an item
public Item sample() // return (but do not remove) a random item
public Item dequeue() // remove and return a random item
public Iterator<Item> iterator() // return an iterator over the items in random order
```

Throw a `RuntimeException` if the client attempts to dequeue or sample from an empty randomized queue.

Deliverables

1. `RandomQueue.java` and a report
2. the outpout of `java RandomQueue`, as a text file.

There is a code skeleton for this exercise on the last page.

Requirements

Note that "random" does not mean "arbitrary". It means "uniformly and independently at random". In particular, when there are N items in the queue then each must have a chance of exactly $1/N$ to be returned by `sample()` or `dequeue()`.

All operations must take constant amortised time (basically, just like the dynamic array implementation of Stack). The exception is `iterator()`, which takes linear time in N. The iterator operations `hasNext()` and `next()` take constant time. The `RandomQueue` object and the `Iterator` object take linear space in N.

The code skeleton in the `src` directory contains a client method that examines several aspects of your implementation.[1]

[1] It is very, very difficult to write systematic test suites for randomized computation. We're just goofing around, but it's better than nothing.

Remarks

If you don't know what an iterator is, read up on it [SW, pp. 138]. Note that it is perfectly legal to have two (or a thousand) iterators over the same `RandomQueue`. Each iterator should use its own random order. Do not implement a `remove()` method in the iterator.

[SW] 1.3.35 contains a useful hint for this exercise.

This exercise can (and should) be solved without importing any other Java classes than `java.util.Iterator`. In particular, you should not base your solution on Java's Collection package. (None of the classes in that package would be of any help anyway, so you'd be wasting your time.)

If you're a good little trooper, try to "avoid loitering" (in the course book's terminology) by freeing unused references.

Code skeleton

```
import java.util.Iterator;
public class RandomQueue<Item> implements Iterable<Item>
{
  // Your code goes here.
  // Mine takes ca. 60 lines, my longest method has 5 lines.

  // The main method below tests your implementation. Do not change it.
  public static void main(String args[])
  {
    // Build a queue containing the Integers 1,2,...,6:
    RandomQueue<Integer> Q= new RandomQueue<Integer>();
    for (int i = 1; i < 7; ++i) Q.enqueue(i); // autoboxing! cool!

    // Print 30 die rolls to standard output
    StdOut.print("Some die rolls: ");
    for (int i = 1; i < 30; ++i) StdOut.print(Q.sample() +" ");
    StdOut.println();

    // Let's be more serious: do they really behave like die rolls?
    int[] rolls= new int [10000];
    for (int i = 0; i < 10000; ++i)
      rolls[i] = Q.sample(); // autounboxing! Also cool!
    StdOut.printf("Mean (should be around 3.5): %5.4f\n", StdStats.mean(rolls));
    StdOut.printf("Standard deviation (should be around 1.7): %5.4f\n",
      StdStats.stddev(rolls));

    // Now remove 3 random values
    StdOut.printf("Removing %d %d %d\n", Q.dequeue(), Q.dequeue(), Q.dequeue());
    // Add 7,8,9
    for (int i = 7; i < 10; ++i) Q.enqueue(i);
    // Empty the queue in random order
    while (!Q.isEmpty()) StdOut.print(Q.dequeue() +" ");
    StdOut.println();

    // Let's look at the iterator. First, we make a queue of colours:
    RandomQueue<String> C= new RandomQueue<String>();
    C.enqueue("red"); C.enqueue("blue"); C.enqueue("green"); C.enqueue("yellow");

    Iterator I= C.iterator();
    Iterator J= C.iterator();

    StdOut.print("Two colours from first shuffle: "+I.next()+" "+I.next()+" ");

    StdOut.print("\nEntire second shuffle: ");
    while (J.hasNext()) StdOut.print(J.next()+" ");

    StdOut.print("\nRemaining two colours from first shuffle: "+I.next()+" "+I.next());
  }
}
```