RandomQueue

Wed Jan 30 23:16:34 2019 +0100, rev. 4bb5ece

> 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.

Note that in python there is no notion of public, the types are slightly different and there are no generic types, so your RandomQueue will be a collection of Objects of arbitrary type. For the sake of avoiding the confusion of different names in the API, the names of the methods are java-style (PascalCase/camelCase) and thus deviate from the python standard of using an underscore to separate two-word that make up a method name.

Deliverables

1. your implementation `RandomQueue` (`.java` or `.py`)
2. a report — there is a skeleton in the `bads-labs` repository
3. the output of your implementation, as a text file.

There are two code skeletons (java and python) 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 to create an iterator. The iterator operations `hasNext()` and `next()` take constant time. The `RandomQueue` object and the `Iterator` object take linear space in N.

Your implementation should not use any of the data structures in `java.util.*` or use the power of the built-in list in python. In other words, it should be based on a non-resizable array.

The code skeleton in the `src` directory (in the git-repository or the linked zip-file) contains a client method that examines several aspects of your implementation.[1]

Tests similar to the ones in the skeleton are implemented on codeJudge. Your implementation must at least pass these tests.

[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]. (or `https://docs.python.org/3/library/stdtypes.html#typeiter`, perhaps you want to use the `yield` construct.) 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.

In the python version, we recommend the pattern alluded to in the code skeleton.

[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 much 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.

Similarly, in python, it is wise to restrict yourself to only use the libraries `algs4.stdlib.stdrandom` (or `random` if you prefer that) (and `algs4.stdlib.stdstats` for the testing functionality of the skeleton).

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());
  }
}
```

In python (output to stderr does not disturb the codeJudge tests):

```
from algs4.stdlib.stdrandom import uniformInt,shuffle
from algs4.stdlib.stdstats import mean,stddev
#from algs4.stdlib.stdio import eprint
import sys
def eprint(*args, **kwargs):
    print(*args, file=sys.stderr, **kwargs)

class RandomQueue:
# the code replaces the following Nones;
    def __init__(self):
        None
    ...
    def __iter__(self):
        # your code here:
        # create a list mine of the objects in the intended order; the following iterates over mine
        for x in mine:
            yield x

# This  "main method" tests your implementation. Do not change it.
if __name__ == '__main__':
    Q = RandomQueue()
    # build a randomQueue with 1,2,..,6
    for i in range(1,7):
        Q.enqueue(i)

    # print 30 die rolls
    eprint( ' '.join([str(Q.sample()) for i in range(30) ] ) )

    # Let's be more serious: do they really behave like die rolls?
    rolls = [ Q.sample() for i in range(1000) ]
    eprint("Mean (should be around 3.5): {:5.4f}".format(mean(rolls)))
    eprint("Standard deviation (should be around 1.7): {:5.4f}".format(stddev(rolls)))
    # removing 3 random values
    eprint( "Removing {}".format(' '.join( [str(Q.dequeue()) for i in range(3) ] ) ) )
    #Add 7,8,9
    for i in range(7,10):
        Q.enqueue(i);
    # Empty the queue in random order
    while not Q.isEmpty():
        eprint(Q.dequeue(),end=' ');
    eprint()

    # Let s look at the iterator. First, we make a queue of colours:
    C= RandomQueue()
    C.enqueue("red"); C.enqueue("blue"); C.enqueue("green"); C.enqueue("yellow");
    I = iter(C)
    J = iter(C)
    eprint("Two colours from first shuffle: {} {}".format(next(I),next(I)))
    eprint("Entire second shuffle: {}".format(' '.join([i for i in J])));
    eprint("Remaining two colours from first shuffle: {} {}".format(next(I),next(I)))
```