Programming Assignment 4

Assigned: April 17
Due: May 8

The grader for this assignment will be Monish Vachhani. Email: monish.k.vachhani@gmail.com

In this assignment you will implement three versions of a priority queue and test their comparative running times and quality over a large data set.

Basic idea

The objective of this assignment is to compare the running time and quality of three different implementations of a priority queue. A driver will run the priority queue by putting "jobs" on the queue and requesting that they be "executed". The quality of a queue is determined by the extent to which it executed high priority jobs quickly.

The priority of a job is a number randomly chosen between 0.0 and 1.0. Large numbers correspond to high priority.

If the queue is empty, then the request to "execute the next job" is a no-op.

Priority Queue Implementation

The three priority queue implementations are:

1. A stack, implemented as a linked list. When a new job comes in, it is put at the front of the list; and the job at the front of the list is always chosen to execute next. That is, the priorities are ignored altogether. This will run very quickly but will have poor quality (defined below).

2. A linked list. To find the highest priority do a linear search through the list. This is optimal quality, but very slow.

3. A Max heap, implemented as an array. This will be reasonably fast and optimal quality.

You should either use the Java library interface Queue or (probably easier) define your own interface MyQueue, with a method for adding an element, a method that pops the highest priority element, and a method that checks whether the queue is empty.

Quality

The quality of a queue on a set of jobs is measured as follows: Define the “delay cost” of a particular job as its priority times the length of time that it waits in the queue, and then define the total cost of a collection of jobs as the average of their individual delay costs. Low values of delay cost is considered high quality. ”Wait time” here is considered in terms of the number of job operations; creating and executing a job both count as 1 tick of the clock.

Jobs

To calculate this, implement a job as an object with two data fields: the priority, which is a Double, and the creation time, which is an integer.
To generate a random priority, use the Java Random package. This is done as follows:

1. Import the random class: `import java.util.Random;`

2. Create a pseudo-random number generator with a long integer as seed. E.g. `Random generator = new Random(100000000)`. The particular choice of seed doesn't matter.

   The advantage of specifying a seed for the random number generator, rather than using the default generator `new Random()`, is that each time you run the code you get the same sequence of random numbers. This is very helpful for debugging; otherwise, you get unrepeatable bugs — bugs that appear with one sequence, but you can’t figure out, because you can’t get that sequence back again. In production code, you generally do better to use the default generator, which gives a different sequence each time.

3. To get a random integer between 0 and \(n - 1\), call `generator.nextInt(n)`. E.g. to randomly get 0 or 1, call `generator.nextInt(2)`.

4. To get a random real number between 0.0 and 1.0 call `generator.nextDouble();`

5. If you want to simulate flipping a coin that has probability \(p\) of coming up heads, you can call

   ```java
   if (generate.nextDouble() < p) then heads else tails
   ```

   Alternative, if \(p = n/d\) is a fraction where \(n \leq d\), you can call

   ```java
   if (generate.nextInt(d) < n) then heads else tails
   ```

**Generating random jobs**

You will read a value \(n\) from input for the number of jobs to be generated. The program will then go through two stages. In stage 1, you will generate \(n/2\) random operations. With probability 0.8, this is to create a job of random priority, and with probability 0.2 it is to execute the next job. Stage 2 is the same, but reversing the two probabilities. Thus, in stage 1, the queue fills up rapidly, as jobs are created much more rapidly than they are executed; in stage 2, the queue empties, as jobs are executed much more rapidly than they are created. However, at the end of \(n\) operations, the queue may or may not be empty.

**Execution time**

To measure execution times in java, use the system call `System.currentTimeMillis()`. This returns the amount of CPU time used in millisecond. Note that this is returns as a `long`. If you call this before and after executing code and substract, you will get the CPU time used by the code in milliseconds.

**Input and output**

The input consists of two integers read from standard input or the command line. The first is \(n\), the number of queue operations to execute. The second is the index (1, 2, or 3) of the queue implementation, as numbered above.

The output is the total delay and the total run time, in milliseconds.
Pseudo-code for the driver

So here's the whole pseudo-code for the driver:

```java
public class Job {
    public int creationTime;
    public double priority;
}

main {
    read in n and qIndex;
    MyQueue queue = new empty queue of type qIndex;
    totalDelay = 0.0;
    p = 0.8;
    long startTime = currentTime();
    for (i = 0; i < n; i++) {
        flip a coin that comes up heads with probability p;
        if (heads) {
            priority = random priority between 0.0 and 1.0;
            queue.addQueue(new Job(i, priority));
        } else if (the queue is non-empty) {
            nextJob = queue.popBestJob();
            totalDelay +=
                (i - nextJob.creationTime) * nextJob.priority;
        }
        if (i == n/2) p = 0.2; // switch from stage 1 to stage 2
    } // end for loop
    print("Running time = " + (currentTime() - startTime) +
        ". Average delay = " + totalDelay/n);
}
```

Experimentation

For each of the three queue implementations, experiment to see how the running time varies with n. Start with n = 100 and try successively doubling until the running time is a minute. Record the running time and quality at each value.

Of course, while you are still writing and debugging the code, you should use very small values of n.

Submission

Email to the grader the source code for your assignment plus a text file showing the results of the experimentation.
Honors problem

For the honors problem, you will additionally implement a fourth version of a priority queue. This should be somewhat faster than (3) with a small cost in quality.

The data structure here is analogous to a bucket sort. Specifically: Choose a fixed value $\text{numBuckets}$. Create an array $b$ of size $\text{numBuckets}$. In $b[i]$ put a linked list of the jobs with priority between $i/64$ and $(i+1)/64$. Equivalently, a job of priority $p$ gets put into the bucket with index $\lfloor p \times 64 \rfloor$. When a new job comes in, put it at the head of the corresponding linked list. To find the highest priority job, find the highest non-empty bucket and pop the first element off that list.

So that you can quickly find the first non-empty bucket, construct an additional data structure $\text{activeBuckets}$, which is a maxHeap of the non-empty buckets. Thus, the values here are ints between 0 and $\text{numBuckets} - 1$. This is used as follows:

- To get the highest priority non-empty index in $b$, look at, but do not pop, the maximal element in $\text{activeBuckets}$.
- If you execute the last job remaining in $b[i]$, then $i$ must be the maximal element in $\text{activeBuckets}$. Execute $\text{activeBuckets}.\text{deleteMax}()$.
- If you create a job that goes in bucket $j$ and $b[j]$ is currently empty, then add $j$ to $\text{activeBuckets}$.

A good value for $\text{numBuckets}$ might be 64. The cost in quality will be small — about 1% — so, what with the random variation, you would have to have $n$ greater than 10,000 or so to detect it systematically. (These numbers are from a back-of-the-envelope estimate, and could easily be off by a factor of 2 or 4.)