Programming Assignment 4

Assigned: Nov. 6
Due: Nov. 29

In this assignment you will implement four versions of a priority queue, and two drivers to test them. The drivers both simulate putting "jobs" of specified priority onto the queue, and then taking them off and "execute" them. The first driver takes commands from input, like the drivers in the first three programming assignments. The second driver will experimentally test the behavior of the queues by generated large numbers of random jobs and interspercing job creation and execution randomly.

Basic idea

The objective of this assignment is to compare the running time and quality of four 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 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 four priority queue implementations are:

1. A stack, implemented as an array. When a new job comes in, it is put at the top of the stack; and the job at the top of the stack is always chosen to execute next. That is, the priorities are ignored altogther. This will run very quickly but will have poor quality (defined below).
2. An array of four stacks, each stack implemented as an array. The first stack has the jobs for which $0.0 \leq \text{priority} \leq 0.25$; the second stack has the jobs for which $0.25 < \text{priority} \leq 0.5$; the third stack has the jobs for which $0.5 < \text{priority} \leq 0.75$; the fourth stack has the jobs for which $0.75 < \text{priority} \leq 1.0$. When a job comes in, it is put on the head of the appropriate stack. When a job is to be executed, it is taken off the head of the highest priority non-empty stack. This will run a few times slower than (1) and give higher quality than (1), though not optimal.
3. A headed linked list. Add a new job at the front. To find the highest priority job do a linear search through the list. This is optimal quality, but very slow.
4. A Max heap, implemented as an array. This should run a few times slower than (2) and will give optimal quality.

The heap and the linked list will always execute jobs of equal quality, but if there are two jobs of equal priority, then they may make different choices.

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. Define the quality of a queue on a collection as as the average of the individual delay costs of the jobs. (Low numbers are considered good.) "Time" here is measured in terms of an imaginary clock that ticks once each time a job is created or executed.

Commands from input

The first driver, called `TestQ`, takes commands from standard input or an input text file, like the first three assignments. Again, there is one command per line, and you may assume that the input is correctly formatted. There are three commands:

```
J   <priority>: Create a job with the specified priority. The priority
                is a floating point number
E             : Pop a job off the queue and execute it
F             : Repeatedly pop jobs off the queue and execute them
                until the queue is empty. Exit the program
```

The driver should run all four queues, in parallel, and execute each command on all four of them.

The form of the output is shown below.

You may assume that at most 50 commands will be executed in this driver, so you can use array size 50 for implementations (1), (2), and (3).

Example Input and Output

```
Time Input     Output
0    J 0.9
               Creating J0 priority 0.9
1    J 0.4
               Creating J1 priority 0.4
2    E
               Stack: Executing J1. Delay cost 0.4
               Four stacks: Executing J0. Delay cost 1.8
               Linked list: Executing J0. Delay cost 1.8
               Heap: Executing J0. Delay cost 1.8

3    J 0.95
               Creating J3 priority 0.95
4    J 0.1.
               Creating J4 priority 0.1
5    J 0.6
               Creating J5 priority 0.6
6    J 0.8
               Creating J6 priority 0.8
7.   J 0.3
               Creating J7 priority 0.3
8.   F
               Stack: Executing J7. Delay cost 0.3
               Four stacks: Executing J6. Delay cost 1.6
               Linked list: Executing J3. Delay cost 4.75
               Heap: Executing J3. Delay cost 4.75
```

```
9
               Stack: Executing J6. Delay cost 2.4
               Four stacks: Executing J3. Delay cost 5.7
               Linked list: Executing J6. Delay cost 2.4
               Heap: Executing J6. Delay cost 2.4
10
               Stack: Executing J5. Delay cost 3.0
               Four stacks: Executing J5. Delay cost 3.0
               Linked list: Executing J5. Delay cost 3.0
               Heap: Executing J5. Delay cost 3.0
11
               Stack: Executing J4. Delay cost 0.7
               Four stacks: Executing J7. Delay cost 1.2
               Linked list: Executing J1. Delay cost 4.0
               Heap: Executing J1. Delay cost 4.0
12
               Stack: Executing J3. Delay cost 8.55
               Four stacks: Executing J1. Delay cost 4.4
               Linked list: Executing J7. Delay cost 1.5
               Heap: Executing J7. Delay cost 1.5

13
               Stack: Executing J0. Delay cost 11.7
               Four stacks: Executing J4. Delay cost 0.9
               Linked list: Executing J4. Delay cost 0.9
               Heap: Executing J4. Delay cost 0.9

               Quality: Stack: 3.864   Four stacks: 2.657
                        Linked List: 2.621    Heap: 2.621
```

The states of the four data structures at the start of time 8 is as follows.

Stack (top to bottom): J7, J6, J5, J4, J3, J0

Four stacks:
S[1]: J4
S[2]: J7, J1
S[3]: J5,
S[4]: J6, J3

Linked list: J7, J6, J5, J4, J3, J1
Heap: J3, J6, J7, J1, J5, J4

Class definition

Define the class `Job` as having two data fields: `priority` which is a `double`, and `creationTime` which is an `int`. Feel free, of course, to write getters, setters, and constructors.

Define the abstract class `MyQueue` with three abstract methods: `void add(Job j)`; `boolean empty()`; and `Job pop()`. Write four classes that extend `MyQueue`: `MyStack`, `FourStacks`, `MyLinkedList`, and `MyHeap`. For each of these, implement the three methods appropriately.

Experimentation

The second driver, called `ExperimentQ`, will experimentally test the quality and running times of the four queues with large numbers of jobs.

The command line should contain two arguments: The number of commands to execute, and the type of queue to run (1 for stack, 2 for four stacks, 3 for linked list, 4 for max heap). For example, if you execute the run line command `java ExperimentQ 100 3` , that will generate 100 random commands, as described below, and run them on a linked list.

The output is the overall quality and the running time. (1 line of output for the entire execution).

For queue implementations (1) and (4), use an array of size $\max(50, n/2)$, where n is the number of commands. For implementation (2), each stack can be implemented using an array of size $\max(50, n/8)$. It is immensely unlikely, though not absolutely impossible, that these sizes will be overrun.

Randomization

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

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

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

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

Generating random jobs

The program will then go through three 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. In stage 3, it simply executes all the jobs remaining in the queue, without creating any. 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.

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.

Experiments

For each of the four 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, or until Java runs out of memory. 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

Upload to the classes site the source code plus a text file describing the results of the experimentation.

Pseudo-code for the driver

So here's the whole pseudo-code for `ExperimentQ`

```
main {
   read in n and qIndex from the command line;
   MyQueue queue
   switch (qindex) {
      case 1: queue = new MyStack(n);
      case 2: queue = new FourStacks();
      case 3: queue = new MyLinkedList();
      case 4: queue = new MyHeap(n);
     }
   jobCount = 0;
   totalDelay = 0.0;
   p = 0.8;
   time = 0;
   long startTime = currentTime();
   for (time = 0; time < n; time++) {
       flip a coin that comes up heads with probability p;
       if (heads) {
           priority = random priority between 0.0 and 1.0;
           jobCount++;
           queue.add(new Job(time, priority));
           }
        else if (the queue is non-empty) {
          nextJob = queue.pop();
          totalDelay +=
            (time - nextJob.creationTime) * nextJob.priority;
          }
      if (i == n/2) p = 0.2;     // switch from stage 1 to stage 2
     }  // end for loop
    while (! queue.empty()) {    // stage 3
          nextJob = queue.pop();
          totalDelay +=
            (time - nextJob.creationTime) * nextJob.priority;
          time++
         }
  print("Running time = " + (currentTime() - startTime) +
        "Quality = " + totalDelay/numJobs);
 }
```