CS 61B Homework 8
Due 4pm Friday, November 17, 2006

This homework will give you practice implementing sorting algorithms, and will illustrate how sorting linked lists can be different from sorting arrays. This is an individual assignment; you may not share code with other students.

Copy the Homework 8 directory by doing the following, starting from your home directory. Don't forget the "-r" switch in the cp command.

```
cp -r $master/hw/hw8 .
```

Your job is to implement two sorting algorithms for linked lists. The data structure you will use is a catenable queue. "Catenable" means that two queues can be concatenated into a single queue efficiently--in O(1) time. The LinkedQueue data structure is implemented as a singly-linked list with a tail pointer, much like the one you worked with in Lab 3.

The LinkedQueue class (in the list package) has the following methods.

```
public LinkedQueue();
public int size();
public boolean isEmpty();
public void enqueue(Object item);
public Object dequeue() throws QueueEmptyException;
public Object front() throws QueueEmptyException;
public String toString();
public Object nth(int n);
public void append(LinkedQueue q);
```

The second-last method, nth(), returns item n in the queue (with the assumption that items are numbered from 1) without removing it. The last method, append(), concatenates the contents of q onto the end of this queue (in constant time), and leaves q empty.

You will implement mergesort and quicksort in the file ListSorts.java. In Parts I and II below, assume that the input LinkedQueue (to be sorted) contains only Comparable items, so that casting items to Comparable is safe. All comparisons should be done using the compareTo method. Your code should be work with any Comparable objects, not just the Integer objects used by the test code. (In other words, do not cast queue items to Integers.)

The dequeue() and front() methods can throw QueueEmptyExceptions; make sure that these exceptions are always caught. (If your code is bug-free, such an exception will never occur, so handle caught exceptions by simply printing an error message.) Do not add a "throws" clause to any method prototype that doesn't already have one.

Part I (4 points)
---
Implement mergesort on LinkedQueues using the following three steps.

(1) Write a method called makeQueueOfQueues() that takes a LinkedQueue as input and returns a queue of queues in which each queue contains one item from the input queue. For example, if the input queue is [ 3 5 2 ], this method should return the queue [ [ 3 ] [ 5 ] [ 2 ] ].

```
public static LinkedQueue makeQueueOfQueues(LinkedQueue q);
```

(2) Write a method called mergeSortedQueues() that merges two sorted queues into one. See the comments in ListSorts.java for details.

```
public static LinkedQueue mergeSortedQueues(LinkedQueue q1, LinkedQueue q2);
```

(3) Implement mergeSort(), which sorts a LinkedQueue q as follows. First, convert q into a queue of queues using makeQueueOfQueues(). Repeatedly do the following: remove two items from the queue of queues, merge them using mergeSortedQueues(), and enqueue the resulting queue in the queue of queues. When there is only one queue left on the queue of queues, move its items back to q (preferably using the fast append() method).

```
public static void mergeSort(LinkedQueue q);
```

Part II (4 points)
---
Implement quicksort on LinkedQueues using the following two steps.

(1) Implement a partition() method that partitions a queue into three separate queues containing items less than, equal to, or greater than a pivot item. Again, see the comments for details.

```
public static void partition(LinkedQueue qIn, Comparable pivot,
                             LinkedQueue qSmall, LinkedQueue qEquals,
                             LinkedQueue qLarge);
```

(2) Implement quickSort(), which sorts a LinkedQueue q as follows. Choose a random integer between 1 and q.size(). Find the corresponding item using the nth() method, and use the item as the pivot in a call to partition(). Recursively quickSort() the smaller and larger partitions, and then put all of the items back into q in sorted order by using the append() method.

```
public static void quickSort(LinkedQueue q);
```

There is test code for both mergesort and quicksort; run "java ListSorts". The test code generates different random arrays each time you run it.

Part III (1 point)
---
Uncomment the timing code in the main() method, then run your sorting routines on larger lists. By changing the SORTSIZE constant, you may change the size of the queues you sort. What are the running times (in milliseconds) for sorting lists of sizes 10, 100, 1000, and 10,000? Also try 100,000 if your code can do it. Put your answers in the GRADER file.

Part IV (1 point)
---
A sort is _stable_ if items with equal keys always come out in the same order they went in. If you sort the database records [ 3:hex 7:boo 3:spa ] by number, and the output is [ 3:spa 3:hex 7:boo ], then the sort is not stable because two records with the same key (3) traded places.

Is your mergesort always stable? Explain why or why not.
Is your quicksort always stable? Explain why or why not.

Give a sentence or two in your explanations that describe where in your code or in the data structures the stability of the sort is decided--that is, whether or not equal items are kept in their original order at critical parts of the sorting process. Put your answers in the GRADER file.

```
Submitting your solution
------------------------
Change (cd) to your hw8 directory, which should contain GRADER and
ListSorts.java.  (Your entire implementation should be in ListSorts.java.)
You're not allowed to change any other files, so you can't submit them.
Include your name, login, and answers to Parts III and IV in GRADER.  Make sure
your code compiles and your tests run correctly on the _lab_ machines just
before you submit.

From your hw8 directory, type "submit hw8".  You may submit as often as you
like.  Only the last version you submit before the deadline will be graded.
```