CIS 291 – Data Structures in C++ - Spring 2005
Week 6 Lab Exercise
Week 6 Lab Exercise due: Tuesday, February 22nd, END of lab

Purpose: thinking and code-reading related to sorting algorithms

INDIVIDUAL-investigation, TEAM-verification exercise:

Answer the following questions on a piece of paper individually. Then, compare and discuss your answers with at least one other class member. Then, write your name on the "Next:" list to get your work checked over. As always, to receive credit for this lab exercise, your work must be completed by the end of the lab period.

NOTE that this time, you will lose points for incorrect answers --- I want you to be especially careful in your checking with one another, and to really discuss any differences until you have resolved them.

Remember the cool site demonstrating different sorts that was demonstrated in lecture? (with thanks to Karen Burgess for pointing it out):
http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html

As you remember, on this page, if you click on a picture, you see the corresponding sort in action. If you click on the sort's name, you see this implementation's source code. It is in Java, but the essential sort-logic should still be quite readable for a person who knows C++.

(a) Observe the **bubble sort** demonstration; look at its source code. Can this particular implementation ever stop "early"? If so, how does it tell/determine if it can stop "early"? (be specific!)

(b) Observe the **selection sort** demonstration; look at its source code. Is it selecting and swapping the maximum value from each pass, or the minimum value from each pass?

(c) Observe the **insertion sort** demonstration; look at its source code. In this implementation, will it always assign the current value-to-be-inserted-next into its proper place in the sorted-portion of the array, even if it is already there (as in the case of an already-sorted array)?

(d) Note that the **double storage merge sort** here is the rough equivalent to what we discussed in lecture (using a second array to help with the merging); there is also an **in-place merge sort** that performs the merging within the original array (it shifts up the other elements when a merged element needs to go "earlier".) Observe each of these demonstrations, and look at their source code.

Which of these two is faster? Why?

Both of these have the same "base" cases. For what array size(s) are there no recursion? (be careful!)

(continued on next page!)

(e) There are four different versions of **quick sort** here **(quick sort, quick sort with bubble sort, enhanced quick sort**, and **fast quick sort).** Observe each of their demonstrations.

The first three use the same method to select their **pivot**. Looking at their source code --- how do these select their pivot (to partition around)?

In a quickie attempt to explain the partitioning algorithm (since it was NOT described in lecture), partitioning is accomplished by moving from the "left" end until a value is found greater than the pivot, and moving from the "right" end until a value is found less than the pivot, and swapping these two elements. This is then repeated until the two meet in the middle, and that's where the pivot is placed (since everything smaller than the pivot has been swapped to the right of this, and everything larger has been swapped to the left of this). Note that all 4 of these implementations accomplish partitioning in this fashion.

Looking at the first, **quick sort**: for what array size(s) is there no recursion? (what array sizes are "base" cases?) (again, be careful!)
Looking at the second, **quick sort with bubble sort**: for what array size(s) is bubble sort called instead of quick sort?

(f) There is even a version of **radix sort** here. Observe its demonstration; look at its source code. Notice that there are actually 3 separate links here, one to the sort's source code, one to the LinkedQueue class, and one to the class for the linked list node.

Consider the actual **sort** method here. What arguments does it require?

It happens to use a **LinkedQueue** class. We have discussed queues; go ahead and look at this queue implementation, and list the operations/methods provided for this particular queue implementation.

To receive credit for this lab exercise, the above must be completed by the end of the lab period.