Lab 5: Linked List

Due **10:00 pm Tues**, Mar 11, 2008

1 Assignment

This assignment has two parts. You must complete both to receive full credit, although the first part is worth substantially more. To submit your solution, use the commands:

```
tar –cvf link3d.tar *.java
turnin –c 136 link3d.tar
```

The tar command combines all of your java files into a single file called a "tarball" (which is a Tape ARchive). This allows you to submit multiple files.

You can download the starter files from the course web page. They are marked "TODO" wherever you must add code. To search a set of files for the string TODO, use the Unix command:

```
grep TODO *.java
```

1.1 LinkedList

Implement the LinkedList<Value> class specified below with all public methods **exactly** as printed here. You may add additional public methods if desired. You may add any private methods and classes that you wish and may extend the documentation as needed to explain the behavior and assumption of the classes. You may use any code from lecture, previous lab solutions, or the textbook, but are responsible for ensuring that that code is correct, well-designed, fully documented, and properly attributed.

Your LinkedList class **must** implement addFirst, addLast, add, and size in constant time and space. The sort method **must** be implemented by either the merge sort or insertion sort algorithm. You must actually sort linked list nodes—you may not convert the list to an array. All other methods should be reasonably efficient. You may not use any java.util classes (although you'll have to use java.util.Iterator and java.util.Comparator *interfaces*) or structure5 classes except for Assert. When in doubt, see the Java LinkedList documentation for clarification of what the methods should do.

Comment every public method with a tight asymptotic time bound. Compile only LinkedList.java and Test.java when you are testing using the command: java LinkedList.java Test.java.

```
import java.util.Iterator;
import java.util.Comparator;

/** A linked list with efficient access to the last  element. */
public class LinkedList<Value> implements java.lang.Iterable<Value> {

/** Appends the specified element to the end of this list.*/
    public void add(Value v);

    /** Inserts the specified element at the specified position in
this list. */
    public void add(int i, Value v);

    /**  Inserts the given element at the beginning of this list.*/
    public void addFirst(Value v);

    /** Appends the given element to the end of this list. */
    public void addLast(Value v);

    /** Removes all of the elements from this list. */
    public void clear();

    /** Returns the element at position i in this list. */
    public Value get(int i);

    /** Returns the first element in this list.*/
    public Value getFirst();

    /** Returns the last element in this list.*/
    public Value getLast();

    /** Returns the index in this list of the first occurrence of
       the specified element, or -1 if the List does not contain
       this element.*/
    public int indexOf(Value v);

/** Returns true if this value is in the list */
    public boolean contains(Value v);

    /** Replaces the element at the specified position in this list
     * with the specified element and returns the old value. */
    public Value set(int i, Value v);

    /**  Removes the element at the specified position in this list
     *  and returns the old value of that element. */
    public Value remove(int i);

    /** Removes and returns the first element from this list. */
    public Value removeFirst();

    /** Removes and returns the last element from this list. */
    public Value removeLast();

    /** Number of elements in the list */
    public int size();

    /** Returns true if there are no elements in this list.*/
    public boolean isEmpty();

    /** Sorts the elements of this list, ranked by the comparator. */
    public void sort(java.util.Comparator<Value> comparator);

    /** Returns an iterator over the elements in this list (in proper
sequence). */
    public Iterator<Value> iterator();

    /** See last week's lab for the format. */
    public String toString();
}
```

1.2 Radix Sort

Copy your Array class from last week[1] into your new directory and make the following changes to it:

1. Extend the class declaration with:
   implents java.lang.Iterable

2. Add and implement the following methods:
   a. public java.util.Iterator<Value> iterator()
   b. public void radixSort(
      int numBuckets,
      java.util.Comparator<Value> comparator,
      Evaluator<Value> evaluator)

The evaluator returns a floating point number for each value that gives the same sort ranking as the comparator. The radixSort method is an O(n) expected time sort for arrays of size n. It operates by constructing a second Array of LinkedLists, which are called "buckets". Element i of the original array is dumped into the bucket whose index is:

```
b = Math.min(numBuckets – 1, (int)((numBuckets * (evaluator.evaluate(get(i)) – min) / (max – min)));
```

Each bucket is initially null; the linked list for that bucket is only allocated when the first element is ready to be added to it. When all elements have been distributed into the appropriate buckets, clear the original array and iterate through the bucket array. Each non-null bucket must be sorted (using LinkedList.sort) and its contents should then be added to the end of the original array. When complete, the original array will be sorted.

Don't forget to test and debug your radixSort and iterator methods. Finally, take your code out for a spin! When you've implemented everything, the commands:

```
javac *.java
java View cube.xml
```

Will load a 3D file and render it, sorting the polygons according to your radix sort. If you've implemented radix sort correctly you'll see a colored cube spinning. If you have a bug you will likely see something other than a cube, or what looks like a cube with a face missing so that you can see inside.

When you've got the cube working, try some of the other .xml files from the web page for this week…

2 Evaluation

Linked List	
Correctness	**10**
Readability	**10**
Design	**10**
Efficiency	**10**
Array iterator and radixSort	
Correctness	**5**
Readability	**5**
Total	**50**

[1] You may, with permission and attribution, use someone else's implementation if yours wasn't working correctly.

3 Advice

I've provided Test.java, which contains some tests that you can use to aid in debugging. Your code may pass all of these tests and still be incorrect, however. I recommend extending my tests with your own, although this is not required. Use lots of assertions in your code to help detect problems. Comment out any tests that you don't want to run while debugging.

My complete solution for LinkedList.java was about 315 lines of code, including assertions, comments, and whitespace. I chose to implement a circular linked list with a merge sort. The merge sort alone was about 80 lines of code and used many helpers.

I used a helper method getNode(int i) that returned the i^{th} node in my list to implement many methods. I often called it with (i – 1) as an argument to get the parent node of the one I really wanted.

When computing the bucket index in the radix sort, you should remove repeated computations like (max – min) from the loop. You can also turn the divide into a multiply, which is much faster.

Write some tests for your radixSort method. If it works correctly, it should generate the same results as Array.sort. Note that I've provided a TriangleComparator, TriangleEvaluator, IntegerComparator, and IntegerEvaluator.

Weren't you glad last week that your Date class was so well documented and tested? You'll be glad this week that your Array class was in such good shape, too. And maybe you'll want that LinkedList class to be efficient, well tested, and documented, just in case you need it in a future lab…

Here's code outlining the radixSort implementation, with "…" where you need to insert something:

```
/** Radix sorts all triangles from smallest to largest evaluator value */
public void radixSort(int numBuckets, java.util.Comparator<Value> comparator, Evaluator<Value> evaluator) {
        // Compute min and max evaluator range
        float min = Float.POSITIVE_INFINITY;
        float max = Float.NEGATIVE_INFINITY;
        for (Value v : this) {
            float e = evaluator.evaluate(v);
            min =  ... ;
            max =  ... ;
        }

        Array<LinkedList<Value>> bucket = new Array<LinkedList<Value>>();
        bucket.setSize(numBuckets);

        for (Value v : this) {
            // Compute the bucket index:
            int b =   ...  ;

            // Insert into bucket.get(b), allocating a list for it if that bucket does not
            // yet exist...
            LinkedList<Triangle> list = bucket.get(b);
            ...
        }

        // Sort all buckets and dump the results back into the original array
        clear();
        for (LinkedList<Value> list : bucket) {
            if (list != null) {
                list.sort(comparator);
                for (Value t : list) {
                    add(t);
                }
            }
        }
}
```