CSCI 3353 Object Oriented Design
Homework Assignment 7
Due Monday, March 26

This assignment is a continuation of HW 6. The goal is to write a class named *SortedCollection*, which uses a mergesort algorithm to efficiently sort a given collection. Here is the basic idea:

We say that a collection is *sorted* if its iterator returns elements in sorted order. We can think of an unsorted iterator as a sequence of sorted segments, called *runs*. For example, suppose that an iterator generates the following sequence of integers:

```
1, 4, 7, 6, 2, 5, 10, 3, 9, 8
```

This sequence contains five runs: [1,4,7], [6], [2, 5, 10], [3, 9], and [8]. Note that a new run begins whenever a value is less than the previous value.

Given an unsorted collection c, define the *split-merge* operation as follows.

1. Create two empty collections c1 and c2.
2. Traverse c's iterator, distributing its runs evenly between c1 and c2. That is, add the elements in the odd numbered runs get added to c1, and the elements in the even numbered runs get added to c2. (This is the *split* phase.)
3. Create a merged iterator from the iterators of c1 and c2. Then create a merged collection from the merged iterator, as in HW6.(This is the *merge* phase.)

The output of a split-merge operation will be a collection having about half as many runs as the input collection. You can then repeat the operation until you get a collection with just one run; this is the desired sorted collection.

If you want this sorting algorithm to be efficient, you need to be careful about how you merge c1 and c2. In particular, it is important to merge the iterators run by run. That is, the first run of c1 gets merged with the first run of c2, then the second runs of each iterator are merged, and so on, until one of the iterators is exhausted. At that point, the remaining elements of the other iterator are added to the end of the merged iterator.

For an example, consider the above sequence of integers. The split phase will create the two collections having the following values (the runs are separated from each other):

```
c1 = 1 4 7   2 5 10   8
c2 = 6   3 9
```

The merge phase will then merge the corresponding runs of c1 and c2, resulting in a collection having the following three runs:

```
1 4 6 7   2 3 5 9 10    8
```

Repeating the split-merge operation, the split phase produces the following collections:

```
c1:  1 4 6 7 8
c2:  2 3 4 9 10
```

Each collection has a single run (note that the 8 happened to "join" the first run of c1). When they are merged, the resulting collection will be sorted.

With this information as background, we are now ready to tackle the steps needed to write the code. I have divided the task into four parts.

1. In Homework 6 you wrote a class *MergedIterator* that merged sorted iterators. Here is my code for the class (which you can download):

```
public class MergedIterator<T> implements Iterator<T> {
   private LookAheadIterator<T> i1, i2;
   private Comparator<T> comp;

   public MergedIterator(Iterator<T> iter1, Iterator<T> iter2,
                         Comparator<T> comp) {
      i1 = new LookAheadIterator<T>(iter1);
      i2 = new LookAheadIterator<T>(iter2);
      this.comp = comp;
   }

   public boolean hasNext() {
      return i1.hasNext() || i2.hasNext();
   }
```

```
    public T next() {
       T result;
       if (!i1.hasNext())
          result = i2.next();
       else if (!i2.hasNext())
          result = i1.next();
       else if (comp.compare(i1.peek(), i2.peek()) < 0)
          result = i1.next();
       else
          result = i2.next();
       return result;
    }
}
```

This code uses a yet-to-be-written class named *LookAheadIterator*. A *LookAheadIterator* object is an iterator that has an additional method *peek*; this method lets you look at next element without actually moving to the next element. Compare my code to your code. You should discover that using the look-ahead iterator makes my code much simpler and easier to understand.

In this problem, I want you to write the code for the class *LookAheadIterator*. This class should be defined like this:

```
public class LookAheadIterator<T> implements Iterator<T> {
     ...
     public LookAheadIterator(Iterator<T> iter) { ... }

     public boolean hasNext() { ... }
     public T next() { ... }
     public T peek() { ... }
}
```

Note that it has *hasNext* and *next* methods, just like any iterator. It also has the method *peek*. The way to implement *peek* is to have the class cache the next element, using the same ideas you used in HW6.

Also note that the constructor for *LookAheadIterator* takes an iterator as an argument. This iterator is the source of the look-ahead iterator's elements. In particular, the look-ahead iterator will behave exactly the same as its component iterator. The only difference is that it also implements *peek*.

2. My *MergedIterator* class assumes that its component iterators are sorted. In this problem, I want you to modify the code so that it merges unsorted iterators by merging their corresponding runs.

The modifications are not difficult to make, but do require thought. The *next* method will need to keep track of the previously seen element. Each time it has to choose the next element from its two iterators, it should avoid choosing an element that is less than the previous element. (This allows the current run to finish before starting the next one.) If the next element of both component iterators are less than the previous element, then the current run has ended and a new run can begin. Do some examples on paper to see exactly what needs to happen.

3. Write the class *SortedCollection*, which should look something like this:

```
public class SortedCollection<T> extends AbstractCollection<T> {
   private Collection<T> result;

   public SortedCollection(Collection<T> c, Comparator<T> comp) {
      result = sort(c, comp);
   }
   ...
}
```

The private method *sort* will create a sorted collection from its input, using the split-merge operation algorithm defined at the beginning of the assignment. Its merge phase will create a *MergedIterator* object, as in problem 2, and then a *MergedCollection* object from that (as in HW 6). The *sort* method will then repeat the split-merge operation until the collection is sorted. The easiest way to know when you are finished is to check the collections after the split phase. If the collection c2 is empty, then c1 must contain all of the elements in sorted order.

Note that the sorting is being done in the constructor, and the sorted collection is being saved in the variable *result*. The other methods of *SortedCollection* can simply reference the *result* collection; no further sorting is needed.

4. Finally, you need to write a test class named *HW7Test*. I have started it for you. Here is the code, which you can download.

```
public class HW7Test {
   public static void main(String[] args) throws IOException {
      Collection<String> dict = readFromFile("dictionary.txt");
      Collection<String> reversedict = reverse(dict);

      // predicate p1 denotes words longer than 20 characters
      printWords(reversedict, p1);
      System.out.println();

      // predicate p2 denotes words beginning with "chori".
      printWords(reversedict, p2);
   }
}
```

Your job is to complete this code by writing appropriate code for the methods *readFromFile*, *reverse*, and *printWords*, as well as the predicates *p1* and *p2*.

The method *readFromFile* should use a scanner to read each word from the specified text file, and add them to a collection. In this case, the file is "dictionary.txt", which you should download from the course website.

The method *reverse* creates a *SortedCollection* object that sorts the specified collection in reverse order.

The method *printWords* prints the words that satisfy the specified predicate.

HINT: If your *HW7Test* class takes more than a few seconds to run, then your revised *MergedIterator* class is most likely not merging runs correctly. Go back and test it on some smaller example collections.

When you are done, create a zip file containing the files *LookAheadIterator.java*, *MergedIterator.java*, *MergedCollection.java*, *SortedCollection.java*, and *HW7Test.java*, and submit it to Canvas. Make sure that all of your java files are in the package *hw7*. The *MergedCollection.java* file can (and probably should) be the version you wrote for HW6. I want you to submit it so that we can run your code as is.

If you are uncomfortable using your *MergedCollection* code from Chapter 6, then let me know and I will email you my version.