CMSC 132 Project #2 Fall 2012

Date due: Thursday, October 18, 10:00:00 p.m.

1 Introduction

In this project you will implement both an unsorted or unordered list class, meaning that elements will be inserted into a list of this type without considering their values, as well as a sorted list class, meaning that elements of these lists will always be stored in order. In particular the unordered list will be a superclass, and the sorted list will be a subclass of it. The sorted list can inherit and use some methods from the unordered list without modification, while those that need to be rewritten or adapted will be overridden by the sorted list as discussed below. This situation may not be completely realistic– it's arguable whether having a sorted list class as a subclass of an unsorted list is really a good design or not– but this is largely just for the purpose of getting more practice with inheritance.

Both lists will be lists of generic types, so any kind of object can be stored in them, and primitive types can be stored using their respective wrapper classes. The sorted list will use a comparator in order to be able to keep its elements in order, but even the unsorted list will use a comparator, which is needed when searching for elements (finding elements also has to be done in other operations as well). Lists may store class objects that have multiple fields, and each list object will be created with a comparator that allows it to compare the desired field or fields of whatever types of objects it is storing.

Note that a few concepts about linked lists needed for this project may not yet have been covered in lecture as of the date it's being assigned, but will be explained soon even if not. However, most of the material has already been discussed, and in any case everything necessary is covered in Chapter 2 of the text.

By the way, this project doesn't have a GUI and it doesn't involve a a game; its purpose is just to get necessary practice with linked lists, which are extremely important in programming. It also happens to include a number of other topics that have recently been covered.

Check out the project starting code using the procedure given in the first homework assignment. The project to check out is named `proj2`.

2 More about the lists

The first line of the `List` class definition reads as follows:

```
public class List<T> implements Iterable<T>, Comparable<List<T>>
```

As mentioned above, the class to be implemented is a generic class, so we could create a list storing integers (using the class `Integer`), or a list storing `String`s, or a list storing any type of class objects. The class implements the `Iterable` interface, meaning that it must have an `iterator()` method that returns an `Iterator`, and it can be used with the enhanced `for` loop. Using the same generic type parameter `<T>` means the iterator that is returned will be an iterator over the same type of elements that the list is storing. Lastly, the list will implement the `Comparable` interface, meaning that it must contain a `compareTo()` method, which will compare an element of this class with another `List<T>` object (another list with the same element type).

The `List` class uses a `Node` inner class. You may **not** modify this class. (You may also not create any other node classes inside the list classes to use instead.) When we run your code on the submit server we'll use our compiled class file for this inner class, so if you modify it your code will compile and run for you, but fail to work when submitted. You may use the default constructor for `Node`, which initializes both if its fields to `null`.

The `List` and `SortedList` classes described below are in the `list` package. You need to implement all of the methods described below for the `List` superclass, but Section 3.3 discusses which methods of the `SortedList` class you will need to write. There is also an inner class `ListIterator` that you have to implement in the `List` class, and in the `SortedList` class if necessary.

Note that the project requirements below say that you **may not** use any of Java's collection classes (such as `ArrayList`, `LinkedList`, etc.) in implementing either the `List` or `SortedList` classes, since the purpose of the project is for you to write linked list code yourself.

There are different variations that can be used for linked lists, that are mentioned in the text; some will be mentioned in class as well. You may use any of these if you like, as long as you use the provided `Node` class and don't violate any of the requirements in Section 4.

Some of the methods should throw exceptions in certain situations, as described below. In addition, if any method has a reference parameter that is `null` it must throw a `NullPointerException`, without changing anything in its current object or in its parameters. (Note that you may not really have to do anything special to cause this to happen.) In both of these cases it doesn't matter what string or message the exception is constructed with, or even if it has a message string at all.

3 Description of classes to be written

3.1 The `List<T>` class methods

3.1.1 `List(Comparator<T> comparator)`

The `List` constructor is passed a comparator, so notice that a comparator class must be written in order be able to instantiate a `List` object. The comparator will have a `compare()` method that will compare two objects of type `T`, the same type that the list was created to hold, and will return a negative integer if the value of its first parameter (of type `T`) is less than that of its second parameter, 0 if their values are the same, and a positive value if the value of its first parameter is greater than that of its second parameter. The comparator must be used any time that elements of the type stored in the list have to be compared, for example in performing a lookup or insertion operation. This means that the comparator has to be stored by the constructor so that the other list methods can use it later. Other than that, your constructor should appropriately initialize the new list being created so it contains no elements.

Note that whoever is creating a `List` will have to write a comparator, that compares whatever type of elements they want to store in the list, and pass it into this constructor when creating the list. If a `List` storing `Integers` is desired, they will have to write a comparator that compares `Integers`. If a `List` storing some type of complex class objects is needed they will have to write a comparator that compares those objects in whatever way is needed.

3.1.2 `List(List<T> otherList)`

This copy constructor will create a new `List` object whose contents will be exactly the same as its parameter object's. After this method returns, the newly–created object and the parameter object `otherList` must be completely **independent**, so that subsequently changing either one of them will have no effect on the other one. However, notice that when we say "a new `List` object" we mean that the newly–created list should not share any nodes with the list `otherList`, but it will not actually be a full deep copy– the actual data elements that the `data` field in each node points to will in fact be aliased. This is because the reference to the data field in each node is declared using a generic type parameter, and as you will recall from lecture it's not possible in a generic class to instantiate an object of a generic type parameter, due to the way that generics are implemented in Java. Therefore, after making a copy of a list using this method, it should be possible to add or remove elements from one list without affecting the other list's elements, but if mutable objects are being stored in lists, and after this copy constructor is called a method is called on an object in either the original list or the copied list that changes its state, it will be changed in both lists.

3.1.3 `List<T> add(T newElt)`

This method should **append** `newElt` to the **end** of its current object list. Since a list implemented as a linked list essentially has no limit to its capacity (other than the number of nodes that can be stored in memory), this method should always succeed (unless the program encounters an error due to running out of memory, which in normal usage should not occur in any tests your project has to work for). Since a `List` is unordered we want each element added to just be **appended** to the end of a list, without respect to its value or to the values of any other elements already present in the list. In particular a `List` may contain duplicate copies of values, and have values in any order. The method should return a reference to its current object, to allow chained method calls.

3.1.4 `T get(int index) throws IndexOutOfBoundsException`

This method should return the element (meaning a reference to it) that is at position `index` in its current object list. The first element of a list (meaning the one stored at the head of the list) is considered to be at position 0, the one in the following node is considered to be at position 1, etc. If the value of `index` is out of bounds in any way for the list, meaning that there is no element at the position indicated by `index` in the list for any reason, the method should throw an `IndexOutOfBoundsException` (one of Java's library exceptions).

Note that since the method returns a reference to an element in the list, its caller will be able to call methods upon that object if desired, perhaps even modifying its value if it has set methods or other methods that change its state.

3.1.5 `T lookup(T element)`

This method should return a reference to the **first** element, in its current object list, if there is one, that compares identically to its parameter `element`. If there is no such element the method should just return `null`. "Compares identically" means using the comparator that the list was created with.

Note that as for the `get()` method, if the method returns a non–`null` reference the caller will then have a reference to an element in the list that it can can call methods upon, perhaps modifying it if it is a mutable object.

3.1.6 `int size()`

This method should return the number of elements stored in its current object list. An empty list contains no elements.

3.1.7 `boolean isEmpty()`

This method should return `true` if its current object list contains no elements, and `false` otherwise.

3.1.8 `List<T> delete(T element) throws NoSuchElementException`

This method should remove the element from its current object list that compares identically (using the list's comparator) to its parameter `element`, if there is one. If there is more than one such element the **earliest** one in the list (closest to the head) is the one that should be removed. If there is no element in the list that matches `element` the method should throw a `NoSuchElementException` without modifying the current object list. Unless an exception is thrown the method should return a reference to its current object, to allow chained method calls.

3.1.9 `void replace(T oldElt, T newElt) throws NoSuchElementException`

This method should replace the element that matches `oldElt` in its current object list, if it is present, with the element `newElt`. In doing so it should not modify the order of the elements in the list. If `oldElt` is present more than once in the list only the first occurrence should be replaced. If `oldElt` is not present in the list the method should throw a `NoSuchElementException` without modifying the current object list.

3.1.10 `T getLargest() throws NoSuchElementException`

This method should return a reference to the largest element (the one that compares larger than any other) in its current object list. If there is more than one such element it doesn't matter which one a reference is returned to. If there are no elements in the list the method should throw a `NoSuchElementException`.

3.1.11 `T getSmallest() throws NoSuchElementException`

This method should return a reference to the smallest element (the one that compares smaller than any other) in its current object list. If there is more than one such element it doesn't matter which one a reference is returned to. If there are no elements in the list the method should throw a `NoSuchElementException`.

3.1.12 `String toString()`

This method should return a string representation of the current object list. If the list has no values an empty string should be returned. Otherwise the string should be formed by calling `toString()` on each element in the list in order, starting from the head of the list, and joining the results with a single blank space in between each pair. (Every type in Java has some `toString()` method that can be called on it; if it does not override it a class will inherit the `Object toString()` method.) No blank space should appear in the string before the first element's string representation, or after the last element's string representation. (**Note**: this method **should not** add a blank space after element's string representation and then remove the last one; it should instead just not add a blank space after the last element's string representation in the string to be returned.)

Note that if a list is storing strings or class objects then calling `toString()` on them might result in strings that contain embedded blank spaces, so there may be more blank spaces in the returned string than your method will add, but your method should only add a single blank space between the string representations of elements.

3.1.13 `void clear()`

This method must remove all of the elements from its current object list, so it becomes an empty list, just as it was when the constructor had first been called on it. Note that it should still be a valid list, and it should be perfectly fine to add new elements after removing all of the existing elements from a list.

3.1.14 `int compareTo(List<T> otherList)`

As mentioned earlier the `List` class implements the `Comparable` interface, so the `compareTo()` method must be written in it. This method should have the following effect:

- If the current object list and the parameter list contain all the same elements, in the same order, and only the same elements (where "same elements" means they compare identically using the list's comparator) the method should return 0.
- If the current object list and the parameter list do not have elements that compare identically, and the first nonmatching element of the current object list compares as less than the corresponding element of the parameter list, the method should return a negative integer (any negative integer you like).
- If the current object list and the parameter list do not have elements that compare identically, and the first nonmatching element of the current object list compares as greater than the corresponding element of the parameter list, the method should return a positive integer (whichever one you like).
- If the current object list and the parameter list have elements that all compare identically except one list is longer, or has additional trailing elements that the other one does not contain, then the method should return a negative integer if the parameter list is longer and a positive integer if the current object list is longer.

Here are a few examples, assuming lists are storing characters (shown separated by blanks, in the form of a string that your `toString()` method would return for them) in the order shown in the table below:

if the current object list contains the elements	and the parameter list contains the element	the method should return
`c a t`	`c a t`	0
`c a t`	`d o g g y`	a negative integer
`c a t`	`c a u t e r i z e`	a negative integer
`c a t`	`c a t a s t r o p h e`	a negative integer

3.1.15 `ListIterator<T> iterator()`

This method should return an iterator of type `ListIterator`, described below, capable of iterating through the elements in its current object list.

3.2 The `ListIterator<E> implements Iterator<E>` class

Like the `Node` class, the `ListIterator` class is an inner class in the `List` class, and it must be written because the `List` class implements the `Iterable` interface.

3.2.1 `boolean hasNext()`

Returns `true` if the list that the iterator is iterating over has more elements that have not yet been returned by `next()`, and `false` otherwise.

3.2.2 `E next() throws NoSuchElementException`

Returns the next element of the list that the iterator is iterating over, with the first list element (closest to the list's head) being returned the first time that it is called on a new iterator. Note that to agree with the Java API documentation an iterator's `next` method must, if called when the iterator has no more elements, throw a `NoSuchElementException`, so this is what your method must do.

3.2.3 `void remove()`

The effect of `remove()` is not described because you don't have to implement it.

3.3 The `SortedList<T>` class

The first line of the `SortedList` class is:

```
public class SortedList<T> extends List<T>
```

Note that since a `SortedList` is also a `List`, it implements the `Iterable` and `Comparable` interfaces even though they are not explicitly mentioned.

As explained above, a `SortedList` should store elements just like a list, but in nondecreasing order based upon the comparator that the list is created with. Some of the methods of the `List` class must be overridden in the `SortedList` subclass, for example the `add()` method must be written so as to insert the element being added **not** necessarily at the end of the list, but instead in the proper position so the elements of the list are in nondecreasing order based upon the list's comparator. You will need to examine the `List` class methods you wrote and determine which of them will work as is for a sorted list, and not reimplement or override those in the `SortedList` class (except as described below), because using existing code rather than reimplementing it unnecessarily makes a program shorter and reduces the opportunity for errors to be introduced.

However, you also need to override in the `SortedList` class any methods that could be written more efficiently for a sorted list, given the fact that its elements are known to always be in sorted order. For example, searching for an element in an unordered list requires examining all of the elements, while for a sorted list on average it's only necessary to examine half of the elements, because once an element greater than (or equal to, of course) the one being searched for is seen, the search can stop.

So to summarize: any operations in the `List` class that will work for a `SortedList` as well, and cannot be implemented more efficiently for a sorted list, should **not** be unnecessarily reimplemented or overridden in the `SortedList` class; you should figure out which methods these are. But any methods that **must** be rewritten to ensure the ordered property of a sorted list, **or** which may be more efficient for a sorted list, **must** be overridden in the `SortedList` class. You should also figure out which these are.

Note that it is not necessary to store any additional data in a sorted list compared to an unordered list; the components used to represent lists can be identical. It's only necessary to modify or reimplement some of the list methods.

As in the `List` class, duplicate elements may be added to a `SortedList` (except they will all be adjacent in the list). Note that the sorted property of a `SortedList` must always be maintained. In particular, where the `List` class version of the `replace()` method is supposed to maintain the list elements based upon the order that they were inserted, the `SortedList` version of `replace()` must ensure that the list is **still in sorted order** after performing a replacement.

Note lastly that, as mentioned in Section 3.1.4 and Section 3.1.5, the user of your class may call methods on a reference to an object in a `SortedList` that was returned by the `get()` or `lookup()` methods. If they modify the data of the object such that the list's elements are no longer in order your methods have no way to detect that, so it's their error and their problem, not yours (i.e., you don't have to try to detect or prevent it if they want to be a loser and clobber the ordered property of their sorted list).

Keep in mind that a subclass constructor probably has to explicitly make a proper call to the corresponding superclass constructor (unless the superclass has a default constructor) , and other subclass methods may want to call the corresponding superclass method– or other superclass methods– in order to accomplish their job.

Important: by saying that the elements of a sorted list must always be stored in nondecreasing (sorted) order, we **do not** mean that your list should add an element somewhere and then sort or rearrange the list elements. (If you do this you will lose significant credit when your project is graded.) Instead, each element must be added in whatever position in the list will preserve or maintain the sorted property **without** having to do any subsequent rearrangement.

4 Project requirements and related issues

1. You may add any **private or protected** fields and **private or protected** helper methods to the `List` class, but you may not add anything with public or package access to it.

   You may add whatever **private** helper methods you find necessary in the `SortedList` class, but you may not add any public or package methods. You may **not** add any fields at all to this class.

   You may **not** add any other new classes to the project.

   You **must** use the `Node` class given, without modification, to implement both lists.

2. You **may not use any library collection classes** anywhere in writing your `List` or `SortedList` classes (if desired your student tests may use library classes). Using any type of collection classes in the list classes (such as `LinkedList`, `ArrayList`, etc.) will result in receiving **no credit** for the project.

   The **only** library classes that can be used are:

   - The interfaces mentioned (`Comparator`, `Iterable`, and `Iterator`) may of course be used.
   - The exception classes mentioned above (`NullPointerException`, `IndexOutOfBoundsException` and `NoSuchElementException`) may be used.
   - `String`s may be declared and string concatenation may be performed (and of course the `toString()` method may be called on the objects in lists in the `toString()` method).

   No other library classes may be used, and no other methods of the `String` class may be used.

3. You should try to avoid code duplication in implementing the project. If you find that multiple methods contain the same or similar code, turn it into a (private or protected) helper method.

4. Grades for this project will be based on:

public tests	45 points
secret tests	35 points
student tests	10 points
source code and programming style	10 points

5. You must write thorough tests for each method **as soon as you write it**, and ensure that it works correctly before going on. As before, your student tests **must** be in the Java source file `StudentTests.java`, which is in the `tests` package.

   Note that some of the secret tests may store different types of elements in lists than are used in the public tests. If you want to create lists with other element types in your student tests (which would be a good idea), you will have to write a comparator that compares elements of those classes, and pass it into the appropriate list constructor.

6. If you have a bug in your code that you can't figure out, and need to come to office hours, you **must** have written thorough student tests of all your methods, and be able to show the ones that illustrate the problem, otherwise we will not be able to assist you until you have written them as described. You must also be prepared to show the TAs how you tried debugging your code using Eclipse's debugger, and what results you found.

7. The course project grading policy document describes how projects will be graded, and in particular the criteria that will be used for grading your programming style. Be sure to read this information **carefully**.

8. The project grading policy handout describes the late policy for projects and homeworks. **Do not wait** until the last minute to finish and submit your project. It is strongly suggested that you finish and submit the project **at least** one day before the deadline, to allow time to reread the assignment and ensure you have not missed anything that could cause you to lose credit. If you have, this will give you time to correct it. This will also give you time to get any last–minute unexpected problems fixed.

9. As the project grading policy discusses in more detail, **do not use the submit server's web submission procedure to submit a jarfile or zip archive**. If you have trouble submitting assignments using the procedure described in the first homework you should get the problem resolved so you can use the procedure to submit directly from Eclipse.

10. After you submit you must **log into the submit server** and verify that your submission compiled and worked correctly there!

11. Recall that the course project grading policy says that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. See the project grading policy for full details.

5 Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.

The academic integrity requirements also apply to any student tests for programming assignments, which must be **your own original work**. Copying the public tests and turning them in as your student tests would be plagiarism, and sharing student tests in any way is prohibited.