Homework #6
due Monday, March 4, 10:00 PM

In this assignment we continue our investigation of implementing a collection where each element is associated with a function, or what will be called a "tag." The new ADT is *generic*; it works where the element type is any Java type. This will give you practice implement a generic class where you do not know what the element type is. We will use a doubly-linked list with a dummy node for the data structure.

Use this link to accept the assignment:

`https://classroom.github.com/a/Zw5llIO3`

1 Concerning the generic interface `TagCollection`

The `TagCollection` interface is intended to generalize the PartCollection ADT to work with any collection of elements where each is tagged with a string. It functions very similarly except that it works with any type, not just "parts" and it doesn't require that these elements are non-null.

It has the same methods as `Collection` and adds the following:

add(E, String) The regular `add` method will not be supported because it adds without a tag. So, we overload `add` to take a tag as well. The tag must not be null. But the element can be null (unlike with `addPart` in previous assignments).

get(int) Return the element at the 0-based index given. (This method is not always constant-time.)

get(int, String) Return the element from among those with the given tag (if null, it can be any element) according to the given index.

iterator(String) Return an iterator that goes over just the ones that have the given tag.

In Homework #3, the `PartCollection` class had similar methods. These new ones generalize those. We do *not* have the equivalent of `removePart`, which arguably wasn't designed very well anyway.

2 Concerning the generic class `LinkedTagCollection`

The `LinkedTagCollection` class is a *generic* class, which means it takes an element type (`E`) in angle brackets. Then all uses of the type must specify what the element type should be. Omitting the angle brackets leaves a "raw" type, a feature permitted for legacy code (written before 2005!). Your code may not use any raw types.

As before, we have a private "static" nested class called "`Node`" but this time it will be generic since it needs to hold different kinds of data depending on what the sequence needs

it to do. Where we had a "function," we now have a "tag," which is still a `String`; instead of a "part," we will have "data," an element of the generic type. You should choose a different name (`T`) for the type parameter so that it is clear that this is a different declaration than the type parameter `E`. You need to declare at the least a constructor that takes two parameters: a data value and a string (for the tag).

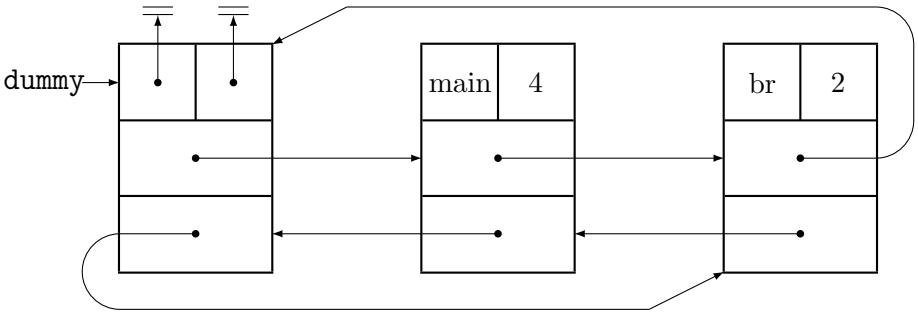
Inside the outer class, you can use type parameter `E` everywhere where an explicit type was used. And inside the `Node` class, you can use `T`. But when we *use* the `Node` class, we will need to provide a type, for example `String` or `E` (since `E` is a legal type inside the outer class).

2.1 Data Structure

The data structure to use for this week is a *cyclic doubly-linked list with a dummy node.*

The list contains a *dummy* node even when it represents the empty sequence. The main class keeps track of the dummy node with a field. The dummy node has an illegal tag value, null. At the start, when the collection is empty, the `next` and `prev` field of the dummy node will point back to the dummy node. After elements are added, the dummy node's `next` pointer refers to the first node, and the `prev` field points to the last node.

The main class also has fields for the size and the version (for fail-fast semantics), which are not shown in the following diagram showing an example of the situation where there are two elements in our tag collection:



Using a dummy node means that our data structure will not need to use null pointers ever, which makes the implementation simpler.

For the data structure, we use a single "dummy" node in a cyclic doubly-linked list. There will always be a dummy node, and it will always be linked in a cycle from the itself to itself. We don't need "head" or "tail" pointers (we can access either from the dummy node).

2.2 The Invariant

You need to check the data structure in `wellFormed` as usual, and check the following things:

- The dummy is not null.
- The dummy's data and tag fields are both null.

- The node are linked in a cycle beginning and ending with the dummy;
- The "prev" links of a node always point back to the node whose "next" field pointed to them;
- None of the links in any of the nodes (including the dummy) are null;
- None of the tags in any of the non-dummy nodes are null;
- The "size" field correctly counts the number of non-dummy nodes in the cyclic list.

You do not need to check these in this order, but rather need to be careful not to be caught in a (bad) cycle, if one exists.

You may use a variant of Floyd's tortoise and hare algorithm to check against bad cycles. It's a variant, because the hare should never go past the dummy node. Our solution doesn't do this because it suffices to check the "prev" links; a bad cycle will always involve a node with a bad "prev" link.

Make sure that your code in `wellFormed` does not *assume* the links aren't null; it *checks* that they are not null. Then your code in public methods can indeed assume there are no null pointers.

2.3 The iterator data structure

We will use a new approach with the iterator this time: we keep *two* pointers into the list: `cur` and `next`. The `next` pointer should be the node where the next element to be returned by the iterator lives; it is the dummy if there are no more elements. If the `cur` pointer is not equal to `next`, it should be the node which will be removed if the iterator removes the current element. If nothing can be removed, then it should be equal to `next`.

As in Homework #3, the iterator will also keep the tag (formerly known as "function") for which it is iterating elements for, and a copy of the collection's version for implementing fail-fast semantics. Thus, the iterator has four fields in total. The invariant is up to you to figure out, althoguh we have tests to help you check your ideas. Reviewing Homework #3 can also help.

The `Spy` class is written assuming you have particular fields with particular names; you should use any reported errors in the `Spy` class as indications of where you need to correct the declaration of the data structure of the main class or the iterator.

2.4 The methods

The class has all the Collection methods we saw in Homework #3 as well as the new ones in `TagCollection`. The class should also provide a `clone` method which will need to perform a deep-clone of the data structure.

Our ADT tests use a new Employee ADT so that the collection will tag employees with roles in the organization. The example roles are whimsical and not meant to be serious.

Random testing is accomplished by instantiating your class with the `Part` class and then comparing with the reference implementation of the PartCollection from Homework #3, except that nulls are permitted now.

3 Files

The repository includes the following files:

src/TestLinkedTagCollection.java Functionality tests.

src/TestInvariant.java The invariant checker tests.

src/TestEfficiency.java Efficiency tests; each test should take no more than a second at most.

src/edu/uwm/cs351/TagCollection.java Skeleton file for interface.

src/edu/uwm/cs351/LinkedTagCollection.java Skeleton file for ADT implementation.

So this project has the full complement of internal, functional, efficiency and random testing but no locked tests.

4 What You Need to Do

We recommend that you perform the following steps:

1. Implement the `private static` nested `Node` class (which must be generic with parameter `T`) and implement the data structure. In particular, it should have a constructor that takes a value of the generic type and a string (for the tag). Make sure that the `Spy` class has no compiler errors.

2. Fix any remaining compiler errors by implementing and extending, and overriding methods (giving the reason, usually "required" except for `clone`). When overriding methods, it is recommended that you use Eclipse's "Override/Implement" Menu option under the "Source" menu; it makes a stub method with the correct parameter types, assuming that you extended and implemented the correct things (never "raw"!).

3. **IGNORE**: Unlock the tests. This will help you test whether you understand *what* the methods are supposed to do.

4. Complete `wellFormed` in the main class and check with the invariant tests `testA`n through `testH`n.

5. Complete `wellFormed` in the nested iterator class and check with the remaining invariant tests.

6. Complete the required methods in the main class other than `clone`. Check with the first few tests of `TestLinkedTagCollection` (`test0`n).

7. Complete the iterator methods except for `remove` and check with the next three sets of tests: `test1`n, `test2`n and `test3`n.

8. Complete iterator removal, and check with the next two sets of tests: `test4`n and `test5`n.

9. Implement fail-fast semantics if you haven't earlier, and check with `test6`n tests.

10. At this point, you should finally complete the `clone` method and pass the clone tests: `test7`n.

11. Now, go to the `TagCollection`and complete the "default" implementation of `get` with two parameters. This implementation cannot use the data structure; it must use other public methods of the class. Test with the last set of tests: `test8`n.

12. Once you are passing all the tests in `TestLinkedTagCollection`, it is time to check with random testing to find any obscure bugs.

13. Once your class is functionally correct, you should test with efficiency tests. If you have to override anything for efficiency, check that you still pass all the functional tests.

The data structure should permit your code to have very few special cases (except in the invariant checker). The desired simplicity will form a major aspect of the grading of this assignment.make

5 Objectives

In doing this homework, it is intended that the student will learn

- to understand and implement generic classes,
- to understand and implement a cyclic doubly-linked list with a dummy node,
- to write loops over a cyclic list with a dummy node, and
- to implement a default implementation in an interface.