Computer Science CS136 (Fall 2021)
Duane Bailey & Aaron Williams
Laboratory 3
Lists with Dummy Nodes

Objective. To gain experience implementing List-like objects.

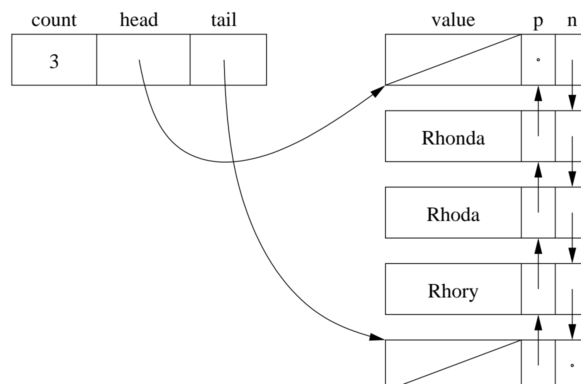
Discussion. Anyone attempting to understand the workings of a doubly linked list understands that it is potentially difficult to keep track of the references. One of the problems with writing code associated with linked structures is that there are frequently *boundary cases*. These are special cases that must be handled carefully because the "common" path through the code makes an assumption that does not hold in the special case.

Take, for example, the addFirst method for DoublyLinkedLists:

```
public void addFirst(E value)
// pre: value is not null
// post: adds element to head of list
{
    // construct a new element, making it head
    head = new DoublyLinkedNode<>(value, head, null);
    // fix tail, if necessary
    if (tail == null) tail = head;
    count++;
}
```

The presence of the if statement suggests that sometimes the code must reassign the value of the tail reference. Indeed, if the list is empty, the first element must give an initial non-null value to tail. Keeping track of the various special cases associated with a structure can be very time consuming and error-prone.

One way that the complexity of the code can be reduced is to introduce *dummy nodes*. Usually, there is one dummy node associated with each external reference associated with the structure. In the DoublyLinkedList, for example, we have two references (head and tail); both will refer to a dedicated dummy node:



The use of dummy nodes.

These nodes appear to the code to be normal elements of the list. In fact, they do not hold any useful data. They are completely hidden by the abstraction of the data structure. They are *transparent*.

Because most of the boundary cases are associated with maintaining the correct values of external references and because these external references are now "hidden" behind their respective dummy nodes, most of the method code is simplified. This comes at some cost: the dummy nodes take a small amount of space, and they must be explicitly stepped over if we work at either end of the list. On the other hand, the total amount of code to be written is likely to be reduced, and the running time of many methods decreases if the special condition testing would have been expensive.

This Week's Tasks. In this lab we will extend the `DoublyLinkedList<E>`, building a new class, `LinkedList<E>`, that makes use of two dummy nodes: one at the head of the list, and one at the end.

You should begin taking a copy of the `LinkedList.java` starter file. This file simply declares `LinkedList<E>` to be an extension of the `structure5` package's `DoublyLinkedList<E>` class. The code associated with each of the existing methods is similar to the code from `DoublyLinkedList`. You should replace that code with working code that makes use of two dummy nodes:

1. First, recall that the three-parameter constructor for `DoublyLinkedNodes` takes a value and two references—the nodes that are to be `next` and `previous` to this new node. That constructor will also update the `next` and `previous` nodes to point to the newly constructed node. You may find it useful to use the one-parameter constructor, which builds a node with `null` `next` and `previous` references.

2. Replace the constructor for the `LinkedList`. Instead of constructing `head` and `tail` references that are `null`, you should construct two dummy nodes; one node is referred to by `head` and the other by `tail`. These dummy nodes should point to each other in the natural way. Because these dummy nodes replace the `null` references of the `DoublyLinkedList` class, we will not see any need for `null` values in the rest of the code. Amen.

3. Check and make necessary modifications to `size`, `isEmpty`, and `clear`.

4. Now, construct two important `protected` methods. The method `insertAfter` takes a `value` and a reference to a node, `previous`. It inserts a new node with the value `value` that directly follows `previous`. It should be declared `protected` because we are not interested in making it a formal feature of the class. The other method, `remove`, is given a reference to a node. It should unlink the node from the linked list and return the value stored in the node. You should, of course, assume that the node removed is not one of the dummy nodes. These methods should be simple with no `if` statements.

5. Using `insertAfter` and `remove`, replace the code for `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, and `removeLast`. These methods should be very simple (perhaps two lines each), with no `if` statements.

6. Next, replace the code for the indexed versions of methods `add`, `remove`, `get`, and `set`. Each of these should make use of only methods you have already written. They should work without any special `if` statements. For `add` and `remove`, think carefully about how `count` is updated.

7. Finally, replace the versions of methods `indexOf`, `lastIndexOf`, and `contains` (which can be written using `indexOf`), and the `remove` method that takes an object. Each of these searches for the location

of a value in the list and then performs an action. You will find that each of these methods is simplified, without mentioning the `null` reference.

Thought Questions. When we make a significant change in design, like we have in this lab, it is useful to reflect on what was gained or lost. The following questions—ideal for an exam—probe a little deeper into the design of `LinkedList`. We'll have an opportunity to discuss these next week.

1. The three-parameter constructor for `DoublyLinkedNodes` makes use of two `if` statements. Suppose that you replace the calls to this constructor with the one-parameter constructor and manually use `setNext` and `setPrevious` to set the appropriate references. The `if` statements disappear. Why?
2. The `contains` method can be written making use of the `indexOf` method, but not the other way around. Why?
3. Notice that we could have replaced the method `insertAfter` with a similar method, `insertBefore`. This method inserts a new value *before* the indicated node. Some changes would have to be made to your code. There does not appear, however, to be a choice between versions of `remove`. Why is this the case? (Hint: Do you ever pass a dummy node to `remove`?)
4. Even though we don't need to have the special cases in, for example, the indexed version of `add`, it is desirable to handle one or more cases in a special way. What are the cases, and why is it desirable?
5. Which file is bigger: your final result source or the original?

Submitting Your Work. To get credit for this week's lab make sure that you've added, committed, and pushed the files `LinkedList.java` and `honorcode.txt` (containing your signature). Recall that `git status` and `git push` help to verify that everything is up-to-date.

⋆