Linked Lists

Preamble

This assignment is the beginning of our discussions on data structures. In this assignment, you will implement a data structure called a *circular doubly linked list*. Please read the whole handout before starting. The end contains important guidelines to be used in testing.

At the end of this handout, we tell you what and how to submit. We will ask you for the time spent in doing this assignment, so *please keep track of the time you spend on it*. We will report the minimum, average, and maximum.

Learning objectives

- Learn about and master the complexities of linked lists.
- Learn a little about inner classes.
- Learn and practice a sound methodology in writing and debugging a small but intricate program.

Collaboration policy and academic integrity

You may do this assignment with one other person. Both members of the group should get on the CMS and do what is required to form a group well before the assignment due date. Both must do something to form the group: one proposes, the other accepts. The grouping is *not* automatically copied over from A1; you must explicitly group.

People in a group must *work together*. It is against the rules for one person to do some programming on this assignment without the other person sitting nearby and helping. Take turns "driving" —using the keyboard and mouse.

With the exception of your CMS-registered group partner, you may not look at anyone else's code, in any form, or show your code to anyone else (except the course staff), in any form. You may not show or give your code to another student in the class.

Getting help

If you don't know where to start, if you don't understand testing, if you are lost, etc., please SEE SOMEONE IMMEDIATELY —an instructor, a TA, a consultant. Do not wait. A little in-person help can do wonders. See the course homepage for contact information.

Linked lists

The diagram on the left below, represents the list of values $[2, 7, 4]$. The leftmost box is an object called the *header*. It contains two values: the size of the list, 3, and a pointer to the first *node* of the list. Each of the other three boxes is an object of class `Node`; it contains the value of an element of the list and a pointer to the next node of the list —the last node has pointer **null** since there are no more nodes in the list. We call the whole thing a *linked list*.

In the diagram on the right below, the **null** pointer in the last node has been replaced by a pointer to the first node. We now have a *circular linked list*.
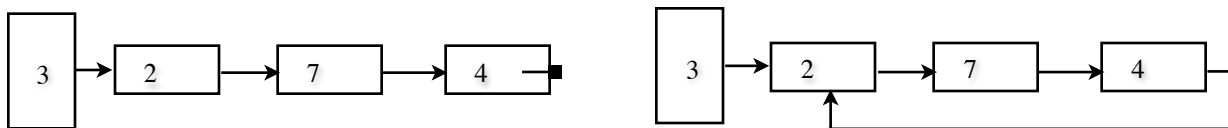
Starting from the header node, one can progress forward and process each node in turn in some fashion. But suppose we have a pointer to the node containing 4 and we want to obtain a pointer to its predecessor —the node containing 7. This requires starting from the header and working up the linked list until it is found, which can be time consuming if the list is long. If finding the predecessor is a common operation, we may want to use nodes that con-

tain pointers to both the successor and the predecessor. The data structure used in this assignment does this, as we now explain.

The middle diagram below represents the list of values [2, 7, 4]. The box with the value 3 is given by an object of class CLL (for CircularLinkedList), shown to the left. Field size gives the number of values in the list. Field head contains a pointer to the object that contains the first value in the list, 2. This is shown with an arrow in the middle diagram. Each value of the list is given by an object of class N, (for Node) shown to the right. Field val contains the value. Field succ, for *successor*, contains a pointer to the next value in the list. It is shown in the middle diagram by the lower arrow leaving box. Field pred, for *predecessor*, contains a pointer to the preceding value in the list. It is shown in the middle diagram by the upper arrow leaving the box. This is a *circular doubly linked list with header*.
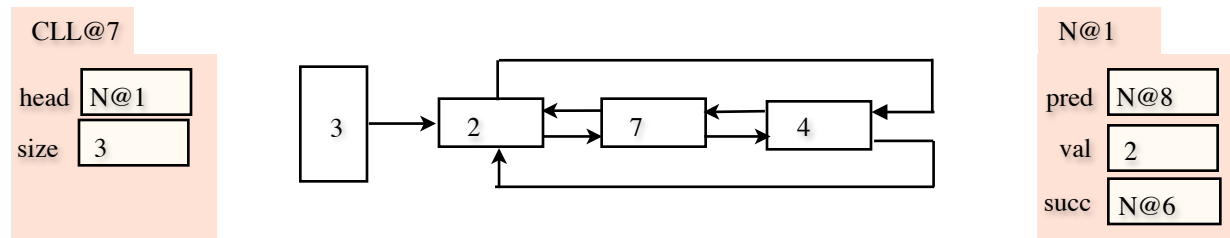
Use of linked lists

One chooses a data structure that optimizes a program in some way, making the most frequently used operations as fast as possible. For example, maintaining a list in an array has disadvantages: (1) The size of the array has to be determined when the array is first created, and (2) Inserting or removing values at the beginning takes time proportional to the size of the list. On the other hand, an array b has the advantage that *any* element, say number i, can be referenced in constant time, using (typically) b[i].

A linked list of any form has these advantages: (1) The list can be any size, and (2) Inserting or removing a value at the beginning can be done in *constant* time —it takes a few operations, bounded above by some constant; create a new object and change a few pointers. On the other hand, if one has only a pointer to the header, to reference element i of the list takes time proportional to i —one has to sequence through all the nodes 0..i-1 to find it.

Circular linked lists are useful for representing lists that are naturally circular. One example is the list of corners of a polygon. Another example is the set of processes on a computer; each is to be given some CPU execution time, in round-robin order, over and over again. In this application, the header could be continually set to its successor, so that the header always points to the next node to process.

A *doubly linked list* typically has a header with three fields: head and size as above and tail, which points to the last node. Each node contains fields pred, val, and succ, as above, but the first node has pred = **null** and the last node has succ = **null**. Thus, it is not circular.

But we can use the representation given in the diagram above and think of it as a non-circular doubly linked list. If h is the head node, instead of using h.tail to reference the last node, use h.pred. And, if p contains a pointer to a node, to test whether it is the last node, use p.succ == h. instead of p.succ == **null**.

This assignment

This assignment gives you a skeleton for class CircularLinkedList<E> (where E is any type). The class also contains a definition of Node (it is an *inner class*; see below). You have to complete 7 methods, as indicated in the skeleton. You must also develop a JUnit test class, called CircularLinkedListTester, that thoroughly tests the methods you write. We give directions on writing and testing/debugging below.

Generics

The definition of the linked list class has `CircularLinkedList<E>` in its header. To declare a variable `v` of that class, use the following to create a linked list whose values are of type `Integer`:

   `CircularLinkedList<Integer> v;`   // (replace `Integer` by any type you wish)

Similarly, create an object whose list-values will be of type `String` using the new-expression:

       new `CircularLinkedList<String>()`

`CircularLinkedList<String>` is a generic type, because of the phrase `.<String>`. We will introduce you to generic types thoroughly later in the course.

Inner classes

Class `Node` is declared as a public component of class `CircularLinkedList`. It is called an *inner class*. Its fields and some of its methods are private, so you cannot reference them outside class `CircularLinkedList`, e.g. in a JUnit testing class. But the methods in `CircularLinkedList` itself *can* and *should* refer to the components of `Node`, even though some are private, because `Node` is a component of `CircularLinkedList`. Thus, inner classes provide a useful way to allow one class but not others to reference the components of a class. We will discuss inner classes in depth in a later recitation.

The constructor in class `Node` is private. The only way to get an object of class `Node` is to use one of `CircularLinkedList`'s functions. For example, in the JUnit testing class, to obtain the first element of doubly linked list `b` of `Integers` and store it in variable `node`, use:

       `CircularLinkedList<Integer>.Node node= b.getFirst();`

What to do for this assignment

1. Start a project a2 in Eclipse, download file CircularLinkedList.java from the CMS, and put that file into a2. Insert into a2 a new JUnit test class (menu item **File -> New -> JUnit Test Case**) named CircularLinkedListTester.java. Note that inner class `Node` is complete; you do not have to and should not change it. Write the 7 methods indicated in class `CircularLinkedList`, in the order given in that class, testing as described below.

2. On the first line of the file CircularLinkedListTester.java, replace *hh* and *mm* by the hours and minutes you spent on this assignment in the comment on the first line. Please do this carefully. If the minutes is 0, replace *mm* by 0. We write a program to extract these times, and when you don't actually replace *hh* and *mm* but instead write in free form, that causes us trouble. In the second comment in the class, write your name and netid and tell us what you thought about this assignment .

3. Submit the assignment (both classes) on the CMS before the end of the day on the due date.

Grading: Each of the 7 methods you write is worth 10 points. The *testing* of each is worth 4-5 points. If you don't test a method properly, points may be deducted in two places: the method might not be correct and it was not tested properly.

Further guidelines and instructions

Note that some methods that you have to write has an extra comment in the body, giving more instructions and hints on how to write it. Follow these instructions carefully. Also, in writing methods 4..7, writing them in terms of calls on previously written methods may save you time.

Writing a method that changes the list: Five of the methods you write change the list in some way. These methods are short, but you have to be extremely careful to write them correctly. It is best to draw the linked list before the change and draw it again after the change; note which variables have to be changed; and then write the code. Not doing this is sure to cause you trouble.

Be careful with a method like append(val) because a single picture does not tell the whole story. Here, two cases must be considered: the list is empty and it is not empty. So *two* sets of before-and-after diagrams should be drawn.

Methodology on testing: Write and test one method at a time! Writing them all and then testing will waste your time, for if you have not fully understood what is required, you will make the same mistakes many times. Good programmers write and test incrementally, gaining more and more confidence as each method is completed and tested.

What to test and how to test it: Determining how to test a method that changes the list can be time consuming and error prone. For example: after inserting 6 before 8 in list $(2, 7, 8, 5)$, you must be sure that the list is now $(2, 7, 6, 8, 5)$. What fields of what objects need testing? What pred fields and what succ fields need testing? How can you be sure you didn't change something that shouldn't be changed?

To remove the need to think about this issue and to test *all* fields automatically, you *must* do the following. In class CircularLinkedList.java, FIRST write functions toString and toStringReverse, as best you can. In writing them, *do not use field* size. Instead, use only field head in the header class and the pred and succ fields of nodes. Do not put in testing procedures for these two functions in the JUnit testing class, because they will be tested automatically when testing the other methods, just as getters are tested when testing a constructor.

For example, after completing toString and toStringReverse, you can test that they work properly on the empty list using this method:

```
@Test
public void testConstructor() {
    CircularLinkedList<Integer> b= new CircularLinkedList<Integer>();
    assertEquals("[]", b.toString());
    assertEquals("[]", b.toStringReverse());
    assertEquals(0, b.size());
}
```

Testing the next procedure, append, will fully test toString and toStringReverse. Each call on append will be followed by 3 assertEquals calls, similar to those in testConstructor.

Would you have thought of using toStringReverse like this? It is useful to spend time thinking not only about writing the code but also about how to simplify testing.