Circular Linked Lists

Preamble

This assignment begins our discussions of data structures. In this assignment, you will implement a data structure called a *circular linked list*. Read the whole handout before starting. Near the end, we give important instructions on 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 A3, 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 and generics.
- 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. We hope you do. Both members of the group should do what is required on the CMS to form a group well before the assignment due date. Both must do something to form the group: one proposes, the other accepts.

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, from this semester or a similar assignment from a previous semester of CS2110, 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.

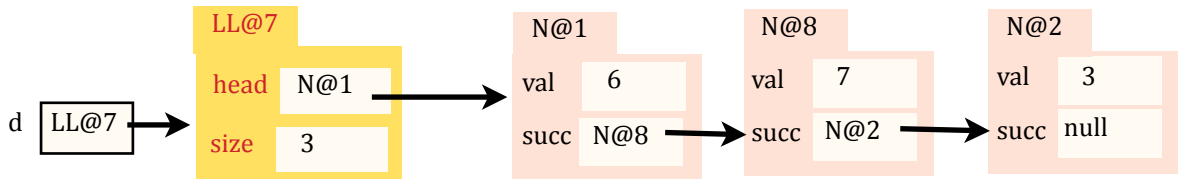
Getting help

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. If you find yourself spending more than an hour one issue, not making any progress, STOP and get help. Some pondering and thinking with no progress is helpful, but too much of it just wastes your time. A little in-person help can do wonders. See the course webpage for contact information. The Piazza A3 FAQs note contains a suggested timetable for completing A3.

Singly linked lists

A list L is just a sequence of values, for example, L = [6, 7, 3]. We write the individual values as L[0], L[1], L[2], … no matter how the list is implemented. So, here, L[0] is 6, L[1] is 7, and L[2] is 3.

The diagram below represents the list L = [6, 7, 3] in a *linked list*. The leftmost object, LL@7, is called the *header*. It contains two values: the size of the list, 3, and a pointer to the first *element* of the list. Each of the other three objects, of class `N` (for *Node*) contains a value of the list and a pointer to the successor node of the list (or **`null`** if there are no more nodes in the list). This data structure is call a *singly linked list*, or just *linked list*.



Many data structures can be used to implement a list. Different programs use lists in different ways. When writing a program, we choose data structures that optimize the program in some way, for example, making the most frequently used operations as fast as possible. For example, maintaining a list of size in the range 0..100 in an array `b` has the advantage that *any* element `b[i]` can be referenced in constant time. But an array has disadvantages: (1) The maximum size of the array has to be determined when it is first created, (2) A variable is needed that contains the number of values currently in the list, and (3) Inserting or removing a value at the beginning takes time proportional to the size of the list.

A singly linked list 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 just a few operations, bounded above by some constant: Create a new node for the value and change a few pointers. On the other hand, to reference element number `i` of the list takes time proportional to `i`—one has to sequence through all the nodes `0..i-1` to find it.

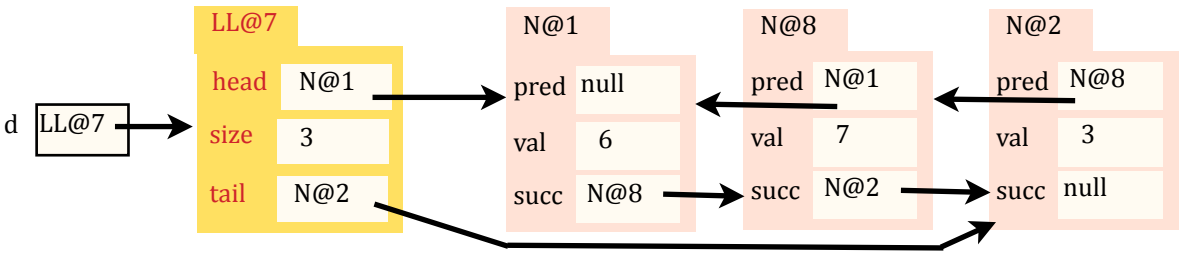
Exercise

At this point, you will gain more understanding by doing the following to construct a linked list that represents the sequence [4, 6, 7, 3]. Do what follows, don't just read it. (1) Copy the linked list diagram shown at the bottom of the previous page. (2) Below that diagram, draw a new object of class `N`, with 4 in field `val` and **null** in field `succ`. (3) Change field `succ` of the new node to point to node N@1 and change field `head` to point to the new node. (4) Finally, change field `d.size` to 4. The diagram now represents the list [4, 6, 7, 3].

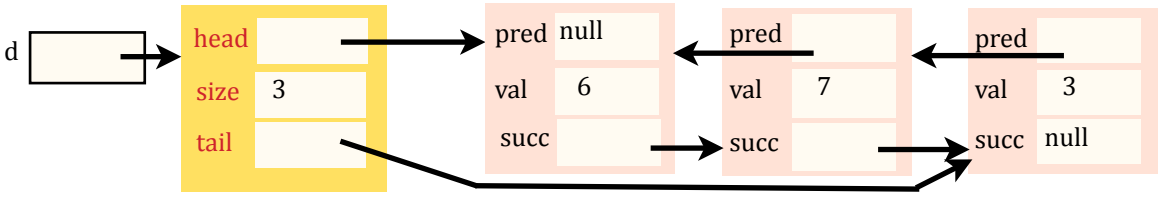
Doubly linked lists

A singly linked list has field `head` in the header and field `succ` in each node, as shown above. A *doubly linked list* has, in addition, a field `tail` in the header and a field `pred` in each node, as shown below. In the diagram below, one can traverse the list of values in reverse: first `d.tail.val`, then `d.tail.pred.val`, then `d.tail.pred.pred.val`. This doubly linked lists represents the same sequence [6, 7, 3] as the singly linked list given above, but the data structure lets us easily enumerate the values in reverse, [3, 7, 6], as well as forward.

The advantage of a doubly linked list over a singly linked list is that, given a variable `n` that contains something like N@8, one can get to `n`'s predecessor and successor in constant time. For example, removing node `n` from the list can be done in constant time, but in a singly linked list, the time may depend on the length of the list (why?).

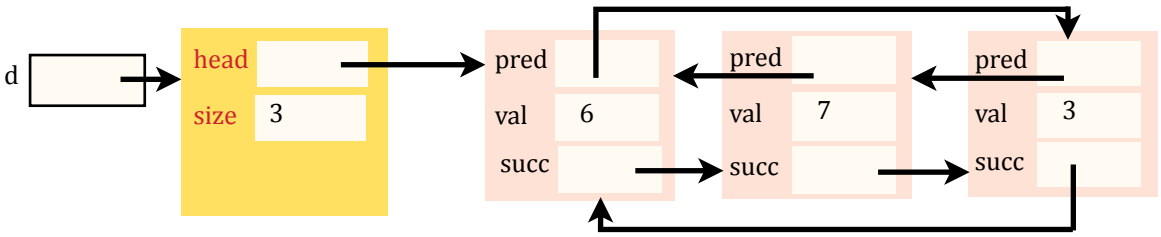


We often write such linked lists without the tabs on the objects and even without names in the pointer fields, as shown below. No useable information is lost, since the arrows take the place of the object pointer-names.



A doubly linked list allows the following operations to be executed in constant time —using just a few assignments and perhaps if-statements: append a value and prepend a value (i.e. insert an element at the beginning of the list). In an array implementation of such a list, prepending would take time proportional to the length of the list in the worst case.

Finally, below is a *circular doubly linked list*. This is like the doubly linked list shown above except that: (1) the successor of the last node is the first node, (2) the predecessor of the first node is the last node, and (3) field `tail` has been removed, since it is not needed.



This data structure can be used when it doesn't matter which value in the list is first; field head could point to any node, and it would be the same (circular) list of values. Here's an example of the use of a circular linked list. Suppose a game is being played with n players. First player 0 goes, then player 1, …, then player n-1, then player 0 again, etc. Keep the players in a linked list and just continue move from one to the next.

This assignment

We provide a skeleton for class `CLL`. It contains many methods, some already written. The ones you have to write are annotated with //TODO comments and are numbered to indicate the order in which they must be written. Please do not delete these comments. The class also contains a definition of class `Node` (an *inner class*; see below). Since you now know about exception handling, each method that you have to write has a body that says the following, and you have to replace it by the necessary code;

thrown new UnsupportedOperationException();

You must also develop a JUnit test class, `CLLTest`, that thoroughly tests the methods you write. We give *important* directions on writing and testing/debugging below.

Generics

The definition of `CLL` has `CLL<E>` in its header. Here, `E` is a "type parameter". To declare a variable `v` that can contain (a pointer to) a linked list whose values are of type `Integer`, use:

```
CLL<Integer> v;    // (replace Integer by any class-type you wish)
```

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

```
new CLL<String>()
```

We will talk more thoroughly about generic types later in the course.

Access modifiers and testing

Java has four different access modifiers: **public**, **private**, **protected**, and package (you can't place "package" in your program; it is not a keyword; it's what you get if you *don't* put an access modifier). This table shows you what each means by showing where the item with that modifier can be referenced (Y means Yes; N means No).

modifier	**class**	**package**	**subclass**	**world**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
package	Y	Y	N	N
private	Y	N	N	N

Some methods in `CLL` are **public** because we want users to be able to use them. Others are helper methods, which should remain unknown and inaccessible to users. So, we could make them **private**. But that makes it very hard to test the helper methods in a JUnit testing class! How can we call a method in order to test it if we can't access it? The same with fields.

Instead, we give such components access modifier "package" (meaning they have no explicit access modifier). Then, since the JUnit testing class and `CLL` are in the same package `LinkedList`, the JUnit testing class can reference the helper methods. We may comment such package components like this in order to remind you that the access modifier is package:

```
/* package */ Node head;  // a node of a linked list (null if empty)
```

When writing a realistic class, when done, you could create a jar file (we'll see what that is later) containing your class (still in a package), and the user would use this. The user would not be able to put a class in the same package, so the user would not be able to get at the helper methods. Isn't that neat?

We have organized the class so that you won't need to reference things with access modifier "package" in the JUnit testing class.. But *we* will reference them, in our grading program.

Inner classes

Just as one can declare a variable (field) and a method in a class, one can declare another class. Class `Node` is declared within class `CLL`. It is called an *inner class*. We'll talk more about inner classes later.

The methods of `CLL` *can* and *must* reference the fields of the inner class directly. In `CLL`, d*on't use getter/setter methods; use the fields directly*!

What to do for this assignment

1. Start a project a3 (or another name) in Eclipse, download file `CLL` from the course website or the course Piazza, and put the file into a3, in package `LinkedList`. The A3 FAQ note on the Piazza will give help in this. Create a new JUnit testing class. Name it `CLLTest`, which is what Eclipse will try to name it, and put it in package `LinkedList`. Inner class `CLL.Node` is complete; do not change it. Write the methods indicated in class `CLL`, testing each thoroughly as suggested below. We provide extensive comments to help you out.

On the first line of file `CLL`, replace *nnnn* by your netids and *hh* and *mm* by the hours and minutes you spent on this assignment. If you are doing the project alone, replace only the first *nnnn*. Please do all this carefully. If the minutes is 0, replace *mm* by 0. We wrote 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. We waste time. Also, please take a few minutes to tell us what you thought of this assignment.

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

Grading: 65 points will be based on functional correctness and 5 points on efficiency of function `getNode`, as specified in the comments within `getNode`. 30 points will be based on testing: we will look carefully at class `CLLTest`. If you don't test a method properly, points might be deducted in two places: (1) the method might not be correct and (2) the method might not be tested properly.

Further guidelines and instructions

Some methods that you have to write have an extra comment in the body, giving more instructions and hints on how to write it. Follow these instructions carefully.

Writing methods: You have to be *extremely* careful to write methods that change the linked list correctly. It is best to draw the list before the change and draw it 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` 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. This will probably means implementing the method with two cases using an if-statement.

Methodology on testing: Write and test one group of methods at a time! Writing 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.

Determining what test cases to use: In testing a method, you must do two things (1) make sure that each statement of the method is exercised in at least one test case. For example, if the method has an if-statement, at least two test cases are required. (2) Ensure that "corner cases" or extreme cases are tested. In the context of circular linked lists, an empty list and a non-empty list may be extreme cases, depending on what is done to the list. When finished with a method, to make sure there are enough test cases, look carefully at the code, based on the above points (1) and (2).

How to test: Determining how to test a method that changes the linked 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` and `succ` fields? 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 must must*** *do the following*. In class `CLL`, first write the constructor and function `toStringRev`, as best as you can, following comments in the body of the methods. Do not put in a JUnit testing procedure for `toStringRev` because it will be tested when testing method `append`, just as getters were tested in testing the first constructor in A1.

After completing the constructor and `toStringRev()`, test that they work properly using the this method:

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

Now write function `append`. Testing it will fully test `toStringRev`. You are testing `append and toStringRev` together. Each call on `append` will be followed by 3 `assertEquals` calls, similar to those in `testConstructor`. Here's the testing for the first call on `append`.

```
@Test
public void testAppend() {
    CLL<Integer> c= new CLL<Integer>();
    c.append(5);
    assertEquals("[5]", c.toString());
    assertEquals("[5]", c.toStringRev());
    assertEquals(1, c.size());
}
```

Because of the way `toString` and `toStringRev` are written, this tests all fields. The first call on `assertEquals` tests field `head` and all the `succ` fields. The second one tests field `head` and all the `pred` fields. The last

one tests field `size`. **To test a call that changes the linked list in any way, *use three similar* `assertEquals` *statements to test everything***! Thus, you don't have to think about which fields to test; you automatically test all of them.

You *must* test all methods that change the circular linked list in this fashion. If you don't, lots of points will be deducted.

Would you have thought of using `toString()` and `toStringRev()` and `size()` like this? It is useful to spend time thinking not only about writing the code but also about how to simplify testing.

The pinned Piazza note A3 FAQ note contains more information about testing: (1) Ensure that you know that the value returned by a function must be tested. (2) Show you how to test that exceptions are thrown properly.