Linked Lists

Preamble

This assignment begins our discussions of data structures. In this assignment, you will implement a data structure called a *doubly 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 doubly 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.

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.

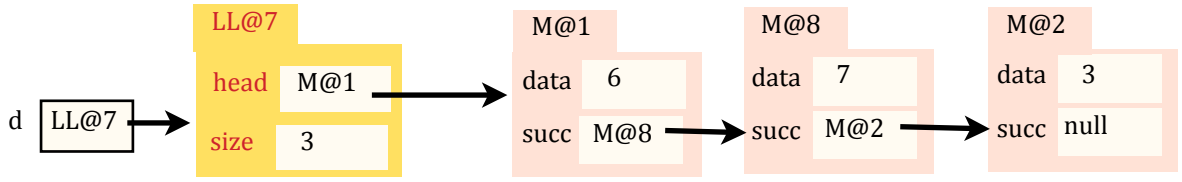
With the exception of your CMS-registered group partner, you may not look at anyone else's code, in any form, or show your code to anyone else (except the course staff), in any form. You may not show or give your code to another student in the class.

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. If you find yourself spending more than an hour or two on 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.

Singly linked lists

The diagram below represents the list of values [6, 7, 3]. 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 *node* of the list. Each of the other three objects, of class `M`, 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*. For simplicity in discussions, for *any* list d, we may use the notation d[0] for its first element, d[1] for its second, and so on. Below, d[0] = 6. But remember, this is not Java notation for lists.



Many data structures can be used to implement a list, Different programs use lists in different ways. 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 b has the advantage that *any* element, say number `i`, can be referenced in constant time, using (typically) `b[i]`. But maintaining a list in an array has disadvantages: (1) The size of the array has to be determined when the array is first created, (2) One needs a variable that contains the number of values in the list, and (3) Inserting or removing values 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 object and change a few pointers. On the other hand, 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.

Exercise

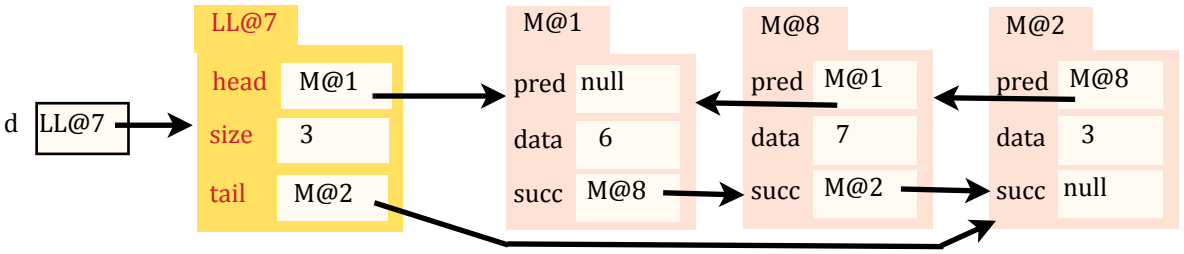
At this point, you will gain more understanding by doing the following, to construct a linked list that represents the sequence [4, 6, 7, 3]. Do what follows, don't just read it. (1) Copy the linked list diagram shown near the bottom of the previous page. (2) Below that diagram, draw a new object of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

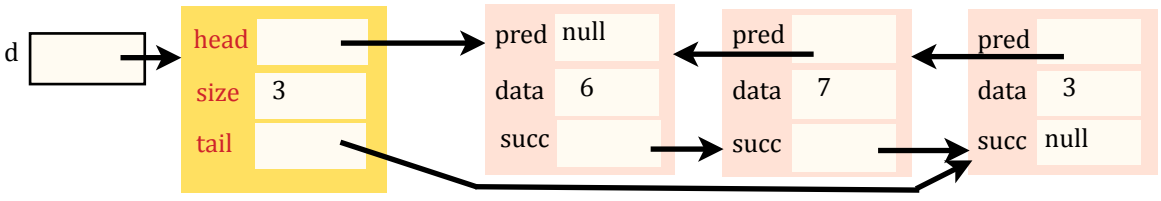
A singly linked list has field `head` in the header and field `succ` in each node, as shown above. A *doubly linked list* has, in addition, a field `tail` in the header and a field `pred` in each node, as shown below. In the diagram below, one can traverse the list of values in reverse: first `d.tail.data`, then `d.tail.pred.data`, then `d.tail.pred.pred.data`. 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 major advantage of a doubly linked list over a singly linked list is that, given a node `n` (containing something like M@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?).

In A3, you will implement a doubly linked list using the representation below. The header will be of class `DLinkedList` (abbreviated `LL` below), and nodes will be objects of class `Node` (given by `M` below), Study this diagram carefully. All further work rests on understanding this data structure.



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— to append a value to the list, prepend a value (insert an element at the beginning of the list), insert a value before or after a given element, and delete a value. *It will be your job to implement these operations in constant* time. In an array implementation of such a list, most of these operations could take time proportional to the length of the list in the worst case.

This assignment

We give you a skeleton for class `DLinkedList<E>` (where `E` is any class-type). It extends class `java.util.AbstractList<E>`. The release code contains many methods, some already written. The ones you have to write are annotated with a //TODO comment, and they indicate the order in which the methods must be written. Please don't delete these comments. The class also contains a definition of `Node` (it is an *inner class*; see below).

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

Generics

The definition of `DLinkedList` has `DLinkedList<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 elements are of type `Integer`, use:

```
DLinkedList<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 DLinkedList<String>()
```

We will introduce you to generic types more thoroughly 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's what you get if you *don't* put an access modifier). This table shows you what each means:

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 of the methods in `DLinkedList` are **public** because we want users to be able to use them. But other methods 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?

Instead, we give the helper methods access modifier package (meaning they have no explicit access modifier). Then, since the JUnit testing class and `DLinkedList` are in the same package, the default package, the JUnit testing class can reference the helper methods.

When writing a realistic class that you want others to use, you would put the class in a real package, not the default package, along with your testing class. When done, you could create a jar file (we'll see that 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?

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 `DLinkedList`. It is called an *inner class*. We'll talk more about inner classes later.

The methods of `DLinkedList` *can* and *should* reference the fields of the inner class directly. There is no need for getter and setter methods for these fields.

In the JUnit testing class, you will need to declare local variables of class `Node`. Here is an example of how to do it, where b is a variable of type `DLinkedList<Integer>` .

```
DLinkedList<Integer>.Node node= b.getHead();
```

What to do for this assignment

1. Start a project a3 (or another name) in Eclipse, download file `DLinkedList.java` from the course website or the Piazza, and put the file files into a3, i**n the default package**. Create a new JUnit testing class. Name it `DLinkedListTest`, which is what Eclipse will try to name it. Inner class `Node` is complete; you do not have to and should not change it. Write the 13 methods indicated in class `DLinkedList`, testing each thoroughly as suggested below.

Some of the methods are helper methods, which is where the core functionality should be. The rest are public methods, which should be made rather small by using the helper methods. We provide extensive comments to help you out.

On the first line of file `DLinkedList.java`, 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. 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 `DLinkedListTest`. So, overall, if you don't test a method properly, points might be deducted in three places: (1) the method might not be correct, (2) the method might not use the best way to get to a node, and (3) 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 a helper method: Five of the methods are helper methods. You have to be *extremely* careful to write them correctly. It is best to draw the linked list. If the method changes the list, draw the list before and 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. Thus, 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 doubly linked lists, an empty list and a list may be extreme cases, depending what is done to the list. When finished with a method, to determine test cases, look carefully at the code, based on the above points (1) and (2).

What to test and how to test it: 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 `DLinkedList`, FIRST write the constructor, function size, and function `toStringRev`, as best you can. In writing `toStringRev`, *do not use field* `size`. Instead, use only fields `tail` in class `DLinkedList` and the `pred` and `data` fields of nodes. Do not put in JUnit testing procedures for `toStringRev`, because it will be tested when testing method `append`, just as getters were tested in testing a constructor in A1.

For example, after completing the constructor, `size`, and `toStringRev`, you can test that they work properly on the empty list using this method:

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

Testing function `append` will fully test `toStringRev`. You are testing those two functions together. Each call on `append` will be followed by 3 `assertEquals` calls, similar to those in `testConstructor`, followed by a test that the returned value is correct.

```
@Test
public void testAppend() {
    DLinkedList<String> ll= new DLinkedList<String>();
    DLinkedList<String>.Node n= ll.append("Ross");
    assertEquals("[Ross]", ll.toString());
    assertEquals("[Ross]", ll.toStringRev());
    assertEquals(1, ll.size());
    assertEquals(ll.getTail(), n);
}
```

Many of your tests of your functions will have similar assertEquals calls. That way, you don't have to think about what fields to test: you test them all.

Once you have implemented size, getNode, and get, the implementation for methods equals inherited from AbstractList will work as well, so you can test using that in addition to using toString and toStringRev.

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