Linked Lists

Preamble

This assignment introduces you to the beginning of our discussions on data structures. In this assignment, you will implement a data structure called a doubly linked list. Please read the whole handout before starting. The end contains important hints 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 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 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.

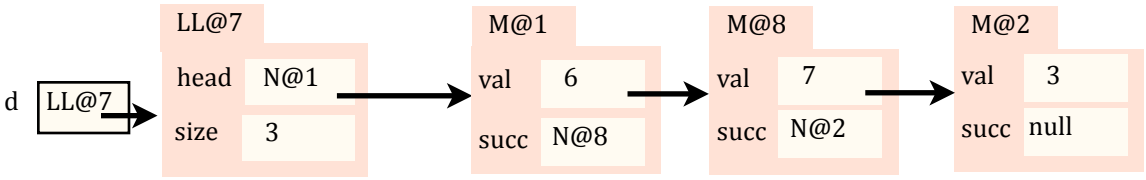
With the exception of your CMS-registered group partner, you may not look at anyone else's code, in any form, or show your code to anyone else (except the course staff), in any form. You may not show or give your code to another student in the class.

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. A little in-person help can do wonders. See the course homepage for contact information.

Singly linked lists

The diagram on the left below, represents the list of values [6, 7, 3]. 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 a class `M`; 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. This data structure is call a *singly linked list*, or just *linked list*.



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 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, and (2) Inserting or removing values at the beginning takes time proportional to the size of the list.

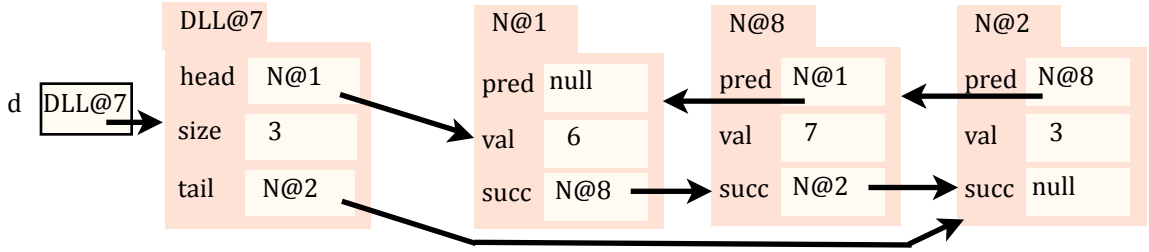
A singly linked list has these advantages: (1) The list can be any size, and (2) Inserting (or removing) a value at the beginning can be done in *constant* time. It takes just a few operations, bounded above by some constant: create a new object and change a few pointers. On the other hand, to reference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

Doubly linked lists

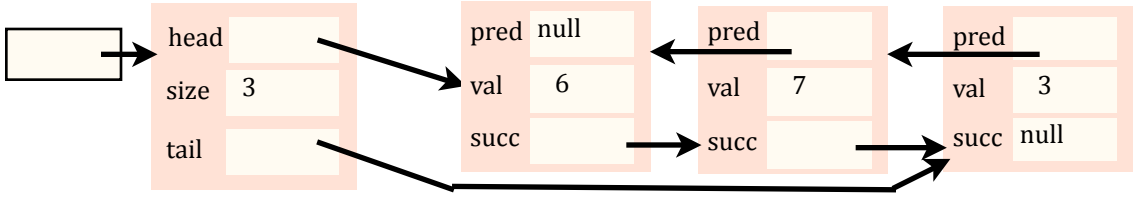
A singly linked list has field `head` in the header and field `succ` in each node, as shown above. A *doubly linked list* has, in addition, a field `tail` in the header and a field `pred` in each node, as shown below. In the diagram below, one can traverse the list of values in reverse: first `d.tail.val`, 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 major advantage of a doubly linked list over a singly linked list is that, given a node `e` (containing something like N@8), one can get to `e`'s predecessor and successor in constant time. For example, removing node `e` 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?).

You will be implementing a doubly linked list using the representation below. The header will be of class `DLinkedList` (abbreviated `DLL` below), and nodes will objects of class `Node` (abbreviated `N` 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.



The doubly linked list allows the following operations to be executed in "constant time" —just a few assignments and perhaps an if-statement are necessary: prepend a value (insert an element at the beginning of the list), append a value, insert a value before or after a given element, and delete a value. 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

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

Generics

The definition of the doubly linked list class has `DLinkedList<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`:

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

We will introduce you to generic types more thoroughly later in the course.

Inner classes

Class `Node` is declared as a public component of class `DLinkedList`. It is called an *inner class*. Its fields and some of its methods are private, so you cannot reference them outside class `DLinkedList`, e.g. in a JUnit testing class. But the methods in `DLinkedList` itself *can* and *should* refer to the components of `Node`, even though some are private, because `Node` is a component of `DLinkedList`. 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 `DLinkedList`'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:

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

What to do for this assignment

1. Start a project a3 in Eclipse, download file DLinkedList.java from the CMS, and put that file into a3. Insert into a3 a new JUnit test class (menu item **File -> New -> JUnit Test Case**) named DLinkedListTester.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 `DLinkedList.java`, testing each one thoroughly in the JUnit test class.
2. On the first line of the file DLinkedListTester.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: we will look carefully at class `DLinkedListTester`. If you don't test a method properly, points might 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 have 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; 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 `DLinkedList.java`, FIRST write functions `toString` and `toStringReverse`, as best you can. In writing them, *do not use field* `size`. Instead, use only fields `head` and `tail` 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() {
   DLinkedList<Integer> b= new DLinkedList<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`. And, testing each of the other methods will require only calls to `assertEquals` like those above.

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.

You should, of course, test that `append` returns the correct node, since append is a function. In addition to the three assertEquals statement as shown above, use something like this::

```
…
DLinkedList<Integer>.Node n= b.append(6);
… three assertEquals, as above …
assertEquals(n, b.getTail());
```