**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.

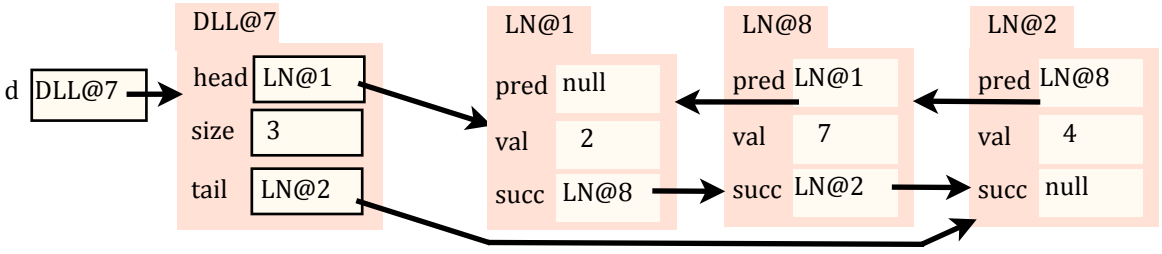
With the exception of your CMS-registered group partner, you may not look at anyone else's code, in any form, or show your code to anyone else (except the course staff), in any form. You may not show or give your code to another student in the class.

### *Getting help*

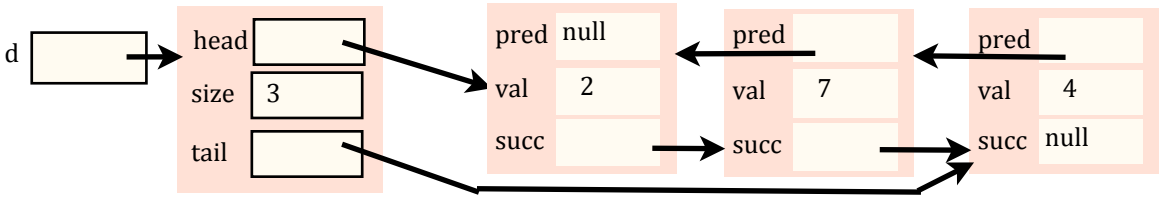
If you don't know where to start, if you don't understand testing, if you are lost, etc., please SEE SOMEONE IMMEDIATELY —an instructor, a TA, a consultant. Do not wait. A little in-person help can do wonders. See the course homepage for contact information.

## Doubly linked lists

A *doubly linked list* is list (or sequence) of values implemented in a data structure composed of two kinds of objects: it contains one *header*, of class `DoublyLinkedList` (abbreviated `DLL` in the diagram below), and objects of class `ListNode` (abbreviated `LN`), which contain the values in the list. The DLL object contains pointers to the head (first element) and tail (last element) of the list. Each `ListNode` contains a value of the list as well as a pointer `pred` to its predecessor in the list (**`null`** if none) and a pointer `succ` to its successor in the list (**`null`** if none). This diagram represents the list (2, 7, 4):




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.




If field `pred` is removed from class `ListNode` and `tail` from `DoublyLinkedList`, the data structure is called a (singly) *linked list with header*.

One can enumerate the values in the above list by sequencing through the nodes, beginning at `ListNode d.head` and using field `succ` to get to each next element. Value **`null`** in `succ` indicates the end of the list. Similarly, one can enumerate the elements in reverse order by beginning at `d.tail` and using field `pred`. Thus, a doubly linked list provides a lot of flexibility.

Moreover, this data structure 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 an 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.

On the other hand, finding the value at a position `i` (e.g. value number 4 of the list) takes time proportional to `i`, but looking at array element `b[i]` takes constant time. Each data structure has its advantages and disadvantages.

This assignment gives you a skeleton for class `DoublyLinkedList<E>` (where `E` is any type). The class also contains a definition of `ListNode` (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 `DoublyLinkedListTester`, that thoroughly tests the methods you write. We give some hints on writing and testing/debugging below.

**Generics**

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

```
DoublyLinkedList<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 DoublyLinkedList<String>()
```

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

**Inner classes**

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

```
DoublyLinkedList<Integer>.ListNode node= b.getHead();
```

## What to do for this assignment

1. Start a project a2 in Eclipse, download file DoublyLinkedList.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 DoublyLinkedListTester.java. Note that inner class `ListNode` is complete; you do not have to and should not change it. Write the 6 methods indicated in class `DoublyLinkedList.java`, and perhaps the optional one, testing each one thoroughly in the JUnit test class.
2. Write the hours *hh* and minutes *mm* you spent on this assignment in the comment on the first line of `DoublyLinkedList`. Please also 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 6 methods you write is worth 10 points. The *testing* of each is worth 6-7 points: we will look carefully at class `DoublyLinkedListTester`. 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.

## Hints

**Hint on 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 `insertBefore(val,node)` (which inserts value `val` in the list before `ListNode node`) because a single picture does not tell the whole story. Here, two cases must be considered: `node` is the first `ListNode` of the list and `node` is *not* the first. So *two* sets of before-and-after diagrams should be drawn.

**Hint 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.

**Hint on testing:** 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? How can you be sure you didn't change something that shouldn't be changed?

Here is a simple way to remove the need to think about this issue and to test *all* fields automatically. In class `DoublyLinkedList.java`, write function `toStringReverse` as well as function `toString`. Then, for each function that changes the list, you need three `assertEquals` statements: (1) test the size of the list, (2) test `toString` [which tests field `head` and all the `succ` fields], and (3) test `toStringReverse` [which tests field `tail` and all the `pred` fields]. After inserting 6 before 8 in list (2, 7, 8, 5), the size should be 5, `toString` should yield "(2, 7, 6, 8, 5)", and `toStringReverse` should yield "(5, 8, 6, 7, 2)".

Using this idea for testing means that the first methods to write are `toString()` and `toStringReverse()`. Then, testing the first few methods that change the list will also be testing `toString` and `toStringReverse`. That's part of the game: you test these two functions as you test the first few other methods that you write.

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.