# 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*

- Practice learning something by reading about it.
- Learn about and master the complexities of doubly linked lists.
- Learn a little about inner classes.
- Learn a little about generics.
- 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. This all applies also to code from similar assignments in previous semesters of 2110.

### *Getting help*

If you don't know where to start, if you don't understand testing, if you are lost, etc., SEE SOMEONE IMMEDIATELY —an instructor, a TA, a consultant. 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 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.

## Learning about linked lists

Part of this assignment is for you to learn about an interesting data structure by *reading* about it —without an instructor explaining everything it to you. Your first task it to read the two-page entry "Linked lists" in the Java Hypertext. You will learn about (1) Singly linked lists, (2) doubly linked lists, and circular linked lists —all with and without headers. You won't see much about applications. Take our word for it: You will see many applications of linked lists in the rest of this course!

## This assignment

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

## Generics

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

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

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

```
new DLList<String>()
```

We will introduce you to generic types more thoroughly later in the course. For now, read the pdf files in the first two lines of the JavaHyperText entry for "generic" and look at slides of 15 Feb. lecture.

## Inner classes

Class `Node` is declared as a public component of class `DLList`. It is called an *inner class*. Its fields and some of its methods are private, so you cannot reference them outside class `DLList`, e.g. in a JUnit testing class. But the methods in `DLList` *can and should* refer to the fields of `Node`, even though they are not public, because `Node` is a component of `DLList`. Thus, inner classes provide a useful way to allow one class but not others to reference the components of the inner class. We will discuss inner classes in depth in a later recitation. For now, read the pdf file linked to in the first line of the JavaHyperText entry for "inner class" and look at slides for 15 Feb. lecture.

The constructor in class `Node` has access modifier *package* . So classes in the same package can use the constructor. For example, in the JUnit testing class, to obtain the first node of doubly linked list `b` of `Integer`s and store it in variable `node`, use:

```
DLList<Integer>.Node node= b.first();
```

## Describing the time a method takes

This is your first look at estimating the time an algorithm takes.

Consider storing the number of 'e's in a `String s`:

```
int n= 0;
for (int k= 0; k < s.length(); k= k+1) {
    if (s.charAt(k) == 'e') n= n+1;
}
```

The repetend consists of an if-statement. At each iteration, it either (1) evaluates the boolean expression, finds it true, and executes the assignment or (2) evaluates the boolean expression and finds it false. We say that execution of the *if-statement takes constant time*, because it does at most two things (evaluate an expression, execute an assignment) and the time it takes for either of those things is always the same.

The time to execute the for-loop obviously depends on how long `String s` is. If s contains `n` characters, the repetend is executed `n` times. In detail, what is executed or evaluated during execution of the for-loop?

- n= 0; k= 0; // once
- k < s.length() // n+1 times
- k= k+1; // n times
- s.charAt(k) == 'e' // n times
- n= n+1; // at most n times

So we see that execution of this loop executes or evaluates at most 4n +3 things, each of which takes constant time. We say that *it takes time proportional to n*, the length of String s, or *it takes time linear in the length of the string*.

These terms *constant time* and *linear time* will be used in discussing this assignment.

## What to do for this assignment

1. Start a project a3 (or another name) in Eclipse. Select directory `src` in the project and use menu item **File -> New -> Package** to create a package named `LinkedList`. Put file `DLList.java` into the package —you can do this by dragging the file on top of `LinkedList`. Insert into package `LinkedList` a new JUnit test class (menu item **File -> New -> JUnit Test Case**) named `DLListTest.java`. Write the 6 methods indicated in class `DLList.java`, testing each thoroughly, before moving on to the next one, in the JUnit test class. Inner class `Node` is complete; do not change it.

Test each method thoroughly. It's best to write a separate testing procedure for each one. We tell you later about how to do the testing. Note that if a method is supposed to throw an exception in certain cases, then you must test that it is thrown properly. Look at JavaHyperText entry "JUnit testing" for information on how to do this.

2. On the first line of file `DLList.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.

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

**Grading**: The correctness of the 6 methods you write is worth 62. The *testing* of each is worth 4-5 points: we will look carefully at class `DLListTest`. If you don't test a method properly, points might be deducted in two places: (1) the method might not be correct and (2) it was not tested properly.

## Further guidelines and instructions

Note that some methods that you will write have an extra comment in the body, giving more instructions and hints on how to write it. Follow these instructions carefully. Also, writing some methods 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 what it looks like 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 `prepend(v)` 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 mean a method that uses 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**: Please read the pdf file found in the JavaHyperText at entry "testing", especially the last part of that pdf file. It is important. Also, refer to the notes on the recitation on testing.

**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 `prev` and `next` 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 `DLList`, FIRST write function `gnirtSot` as best you can. In writing it, *do not use field* `size`. Instead, use only fields `last` in class `DLList` and the `prev` and `val` fields of nodes. You can look at how we wrote `toString` —that can help you. Do not put in JUnit a testing procedure for `gnirtSot`, because it will be tested when testing procedure `prepend`, just as getters were tested in testing a constructor in A1.

For example, after completing `gnirtSot`, you can test that it works properly on the empty list using this method:

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

Now write procedure `prepend`. Testing `prepend` will fully test `gnirtSot`. You are testing those two method together. Each call on `prepend` will be followed by 3 `assertEquals` calls, similar to those in `testConstructor`:

```
@Test
 public void testAppend() {
   DLList<String> ll= new DLList<String>();
   ll.prepend("Sampson");
   assertEquals("[Sampson]", ll.toString());
   assertEquals("[Sampson]", ll.gnirtSot());
   assertEquals(1, ll.size());
}
```

The call `ll.toString()` tests field `first`, all fields `next`, and all fields `val`. The call `ll.gnirtSot()` tests field `last`, all fields `prev`, and all fields `val`. (Remember, `gnirtSot` is `toString` in reverse.) The call on ll.size() tests field size. Thus, *all* fields are tested.

You **must** test *all* methods that change the linked list with three such `assertEquals` calls. That way, you don't have to think about what fields to test; you test them the all.

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