Linked Lists

Preamble

This assignment begins our discussion 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 A4, 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 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. If you are going to work together, then, as soon as possible —and certainly before you submit the assignment— get on the course CMS and form a group. Both people must do something before the group is formed: 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 for this assignment or in any similar assignment in a previous version of CS2110, 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., 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 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 it to you. Your first task is to read the two-page entry "Linked lists" in JavaHyperText. You will learn about singly linked lists, 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 be using linked lists a lot in the rest of this course!

This assignment

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

Generics

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

```
DList<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 DList<String>()
```

You have seen a bit about generic types in lecture and recitation. We 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*.

Inner classes

Class `Node` is declared as a component of class `DList`. It is called an *inner class*. Its fields are private, so they cannot be referenced outside the class —e*xcept that the methods in* `DList` *can and should* reference the fields of `Node`, even if they are private, because `Node` is a component of `DList`. 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*.

The class and its methods have default access modifier *package*, so they can be referenced in any class in the same package, like a JUnit testing class. In a JUnit testing class in the same package, to obtain the (pointer to the) first node of doubly linked list `d` of `Integer`s and store it in variable `node`, use:

```
DList<Integer>.Node node= d.head();
```

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 and does not depend on the size of the data involved.

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

Thus, execution of the loop executes or evaluates at most 4`n` + 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* `s`.

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

What to do for this assignment

1. Getting Eclipse set up for the project is similar to what you did for assignment A3. Start a project a4 (or another name) in Eclipse, create package `linkedlist`, and put file `DList.java` into the package. Create a new JUnit test class in the package (menu item **File -> New -> JUnit Test Case**) with name `DListTest.java`. Use Jupiter (JUnit 5). Write the five methods in class `DList.java` that are marked with //TODO comments, testing each thoroughly in class `DListTest` before moving on to the next one. Please do not remove the //TODO comments. Inner class `Node` is complete; do not change it. *We tell you later about how to test*.

Test each method thoroughly. Write a separate testing procedure for each method. (The first two methods are tested together, so write only one testing procedure for the two of them.)

Lines 11..15 (roughly) of class `DList` contain a static field `timeSpent`. Replace its value (-1) by the time you spent on A4. Do it carefully. We use these values to show you the average and maximum times spent on this assignment.

On lines 3..8 (roughly), put your name and netid (both names and net ids if grouped). Also, please tell us what you thought of this assignment in this comment. We will make your comments anonymously available to the whole class.

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

Grading: The correctness of the five methods you write is worth 62. The *testing* of each is worth 4-5 points: we will look carefully at class `DListTest`. 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. More points may be deducted for egregious errors. We also deduct 3 points if you don't give us the time spent, put your names and net ids, or tell us what you thought of this assignment.

Further guidelines and instructions

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.

Check Preconditions with assert statements? On A1, we required you to check Preconditions of methods using the assert statement in all cases. From now on, it is up to you to determine whether to check a Precondition with an assert statement or not. Good programmers often do it, but not always, because it helps them detect errors sooner. We don't always do it and you don't either.

But note this: The specs of some of the methods talk about time, for example, saying that a method takes constant time. If inserting an assert statement would mean the methods takes more than constant time, like linear time, don't put in the assert statement! Be careful.

Writing a method that changes the list: Three 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 `append(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. Therefore, *two* sets of before-and-after diagrams should be drawn. This will probably mean that the method uses an if-statement.

Methodology on testing: Write and test one group of methods at a time! Writing *all* methods 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: Read the pdf file in the JavaHyperText 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 after 8 in list [2, 7, 8, 5], you must be sure that the list is now [2, 7, 8, 6, 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 `DList`, FIRST write function `toStringR` as best you can. In writing it, *do not use field* `size`, `head`, and the `succ` fields. Instead, use only fields `tail` in class `DList` and the `pred` and `val` fields of nodes. Look at how we wrote `toString` —that can help you. Do not put in a JUnit testing procedure for `toStringR`, because it will be tested when testing procedure `append`, just as getters were tested in testing a constructor in assignment A1.

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

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

Now write procedure `prepend`. Testing `prepend` will also test `toStringR`. You are testing those two method together. Each call on `prepend` will be followed by THREE `assertEquals` calls, similar to those in `testConstructor` (we suggest copying and pasting):

```
@Test
public void testPrependAndToStringR() {
    DList<String> dl= new DList<String>();
    dl.prepend("2110");
    assertEquals("[2110]", dl.toString());
    assertEquals("[2110]", dl.toStringR());
    assertEquals(1, dl.size());
}
```

The call `dl.toString()` tests field `head`, all fields `succ`, and all fields `val`. The call `dl.toStringR()` tests field `tail`, all fields `pred`, and all fields `val`. The call `dl.size()` tests field `size`. Thus, *all* fields are tested.

You **must** test *every* method call that changes the linked list with three such `assertEquals` calls, one calling `toString()`, one calling `toStringR()`, and one calling `size()`.

We'll say it again because some people do not read the last paragraph carefully: *Every* test case that tests a method that changes the linked list must use three such `assertEquals` calls.

That way, you don't have to think about what fields to test; you test them the all. And it's easy to do: Just copy those three calls from elsewhere, paste them in, and edit them.

If you do NOT test each method call that changes a linked list in this manner, many points will be deducted.

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