Homework 3: Imperative
Due: 5:00 PM, Feb 22, 2020

Contents

1 Overview of Generic/Parameterized Types 2

2 Double the Fun with Doubly-Linked Lists 3
   2.1 Implementing Core Methods 3
   2.2 Adding an Iterator 4
   2.3 Testing 5

3 SeatMap 6
   3.1 The Mystery Continues... 6
   3.2 Introduction to ArrayLists 7
   3.3 Implementing SeatMap 8

Objectives

By the end of this homework, you will be able to:

- Implement doubly-linked lists from soup to nuts!
- Determine when to use a for loop vs. a while loop

How to Hand In

For this (and all) homework assignments, you should hand in answers for all questions. For this homework specifically, this entails answering the Double the Fun with Doubly-Linked Lists and Seatmap questions.

In order to hand in your solutions to these problems, they must have the appropriate package headers.

Begin by copying the source code (*/course/cs0180/src/hw03/src/*.java) from the course directory to your own personal directory. That is, copy the following files from */course/cs0180/src/hw03/src/* to */course/cs0180/workspace/javaproject/src/hw03/src:

- SinglyLinkedList.java containing public class SinglyLinkedList
- AbsList.java containing public abstract class AbsList<T>
You will also need to copy the following files for testing purposes to your own personal directory. That is, copy the following files from /course/cs0180/sol/hw03/sol/* .java to ~/course/cs0180/workspace/javaproject/sol/hw03/sol:

- IList . java containing public interface IList < T >
- ConstructorTest . java containing public class ConstructorTest

Do not alter these files, except for IList . java !

After completing this assignment, the following solution files should be in your workspace /javaproject/sol/hw03/sol directory, all as part of the hw03 . sol package:

- Double the Fun with Doubly-Linked Lists
  - IList . java containing public interface IList < T > extends Iterable < T >
  - DoublyLinkedList . java containing public class DoublyLinkedList < T >, which extends AbsList < T >.
  - ConstructorTest . java containing public class ConstructorTest
  - A file DoublyLinkedListTest . java containing public class DoublyLinkedListTest, which thoroughly tests your doubly-linked list implementation.

- SeatMap
  - SeatMap . java, containing public class SeatMap, which implements the methods bookSeats, findBlock1, and findBlock2.
  - SeatMapTest . java, containing public class SeatMapTest, which thoroughly tests your SeatMap implementation

To hand in your files, submit them to Gradescope. Once you have handed in your homework, you should receive an email, more or less immediately, confirming that fact. If you don’t receive this email, try handing in again, or ask the TAs what went wrong.

Note: If your code does not work with the autograder you will be penalized. If there are any complications please contact the TA’s via hours or Campuswire. If the gradescope autograder fails, try this before posting on Campuswire: run hw03_compatibility from your personal cs0180 directory - or the directory that contains your workspace folder. This will tell you what methods that the autograder expects from you, and which of these methods you might be missing.

1 Overview of Generic/Parameterized Types

When you used Java’s LinkedList class in Lab, you supplied the type of the list elements in angle brackets, as in LinkedList < Integer >. For this assignment, we want you to create lists that can similarly take contents of any type, not just the ints that we have been doing in class. Fortunately, this doesn’t involve a lot of work.

Just as you provide a type in angle brackets to use LinkedLists, you can annotate a class or interface with a type parameter (such as T), as in the following example:
```java
class Node<T> {
    T item;
    Node next;

    Node(T item, Node next) {
        this.item = item;
        this.next = next;
    }
}
```

To create an object of this class, you would just supply the type, as you do when using the built in linked lists. For example, `Node<Integer>`.

The notes from Friday will be extended (if not there already) to show this for our in-class example. It will also show how you can nest the Node class inside the List class to share the List class' type parameter. You will see the `<T>` annotation in the starter files for this assignment.

## 2 Double the Fun with Doubly-Linked Lists

As a computer scientist, you will often be presented with a problem where it is easy to determine what functionality you require from a piece of code you are writing, but you do not yet know the best implementation. This is a good situation for an interface. That way, you can decide on one implementation now, and later you can switch it out for a newer and better one without impacting your code base.

In class, we’ve been implementing variations on an `IList` interface (one functional, one with the `LinkedList` class). For this assignment, we change the implementation again to be a *doubly-linked list*. These are lists in which each node refers to both the next node *and the previous node*, as shown in Figure 2. Can you imagine ways in which this implementation might have advantages over a singly-linked list (which we have seen in lecture)?

### Invariants

A doubly-linked list maintains the following invariants:

1. When the linked list is empty,
   - `start` is null
   - `end` is null

2. When the linked list is nonempty,
   - `start` refers to the first node of the list
   - `end` refers to the last node of the list
   - if the list contains exactly one node, then both `start`'s and `end`'s `next` and `prev` fields are null
   - if the list contains more than one node, then only `end.next` and `start.prev` are null; no other node’s `next` or `prev` fields are null
2.1 Implementing Core Methods

The goal of this part of the assignment is to augment the IList interface and provide a DoublyLinkedList class to include get, remove, reverse, removeFirst, and removeLast methods:

- The get method takes as input an index, say i, and returns the ith item in the list. You should assume 0-indexing where the first item in the list has index 0. If there is an index that is too large, or if the list is empty, you should throw an IllegalArgumentException with the string “index out of bounds”.

- The remove method takes as input an item, the first instance of which it proceeds to remove from the list. It returns a boolean indicating whether or not the remove operation was successful: i.e., true if the item was found and removed, or false if not.

- The reverse method reverses the order of the items in the list that it is called on. It should take O(n) time and O(1) space (it should not create any new lists), where n is the number of items in the list.

- The removeFirst method removes the first item in the list and returns it. If there are no items in the list, it throws a RuntimeException with the string “tried to remove from empty list”.

- The removeLast method removes the last item in the list and returns it. If there are no items in the list, it throws a RuntimeException with the string “tried to remove from empty list”.

Task: Augment the given IList interface to include these five methods.

Task: Write a doubly-linked list class, DoublyLinkedList. Your doubly-linked list class should implement your augmented IList interface (both the methods we initially included in the interface, plus the ones that you have added). Furthermore, your DoublyLinkedList should extend the AbsList abstract class provided in the support code.

2.2 Adding an Iterator

Adding an iterator to the DoublyLinkedList class is useful for enabling people to write for-loops over the data structure. It is also useful for implementing toString and equals methods (the AbsList class we provided sets that up for you, once you provide the iterator.

Task: Modify your IList interface so that it extends Iterable. Doing this is a key step in enabling Java’s built-in for-loops to work on your DoublyLinkedList implementation.

Task: Write an iterator for your DoublyLinkedList.

Hint: Model your solution after the one we did in class on Monday (Feb 10). If you need to report an error, throw a RuntimeException with the string “iterator error”.
@Override
public Iterator<T> iterator() {
    return new Iterator<T>() {
        @Override
        boolean hasNext() {
            // TODO
        }

        @Override
        public T next() {
            // TODO
        }
    }
}

2.3 Testing

As usual, you must exhaustively test your DoublyLinkedList methods! However, because we are working with mutable data, you should create a setup method in your test class, like you did in Lab 3. A setup method takes in nothing and returns an object that you will perform tests on; in this case, a DoublyLinkedList. You know the contents of this list, because the method makes a list, exactly as you want, and returns it. This allows you to safely perform tests. We use setup methods because we want our tests to pass regardless of what order the test methods are run in. If we were running tests on a DoublyLinkedList that was a global variable in our test class, switching the order of our testAdd and testRemove methods would be disastrous. But if we create a new DoublyLinkedList in each test method, we won’t run into any issues! Here’s an example of what a setup method could look like:

```java
/**
 * A setup method for a basic list
 * @return a basic linked list
 */
public IList<Integer> setupBasicList() {
    IList<Integer> dll = new DoublyLinkedList<Integer>();
    dll.addFirst(1);
    dll.addFirst(5);
    dll.addFirst(10);
    return dll // A doubly linked list with elements 10,5,1, in that order
}

// A method to test get
public void testGet(Tester t) {
    IList<Integer> testList = setupBasicList();
    t.checkExpect(testList.get(0), 10);
    ...
}
```

The contents of this list are up to you, but you should make several setup methods so that you can test methods such as remove and reverse. Remember to check edge cases!
Starting with this homework, you must test your exceptions. Here’s how to do that:

```java
// say you have a runtime exception somewhere in your code:

public class ErrorObject {
    public void alwaysFail(int i, String s, boolean t) {
        throw new RuntimeException("informative error message");
    }
}

// and then somewhere in your testing class, you should have this
checkException():
t.checkException(new RuntimeException("informative error message"),
    new ErrorObject(), "alwaysFail", 1, "hello", false);
```

In general, the syntax for checkException is this:

```java
t.checkException(<new object of exception type>, <new object of class that calls the method you want to test>, <method name, as a String>, <method parameters, in order>);
```

You should test your exceptions for both problems in this homework.

**Task:** Write a class to exhaustively test your `DoublyLinkedList` methods, including those methods found in the iterator (except the methods given to you in source code). Be sure to use at least one setup method to create `DoublyLinkedLists` for testing.

**Just for Fun:** Can you figure out how to implement reverse in constant time? Consider trading off space for time. (you don’t have to turn in anything for this question)

3 SeatMap

3.1 The Mystery Continues...

After poring over code book after code book from the SciLi, you finally cracked the mysterious message from the SignMeUp code — it leads you to an AYCE sushi place in Boston. Your heart beats faster as you walk up to the reception — this could be what leads you to find the perpetrator of all the CIT crimes. When you ask for a table, though, the receptionist looks shocked and asks you if you’re the resident CIT detective. They hand you a paper bag and tell you that this was left for you by a mysterious customer earlier in the day. You thank them and walk out of the restaurant. On the street, you open the bag. Inside is Maceo’s untitled lamp bear™ plush — you knew it! This is confirmation that all of these crimes are connected.

Your mind races and out of the corner of your eye, you spot a figure in a lumpy, oversized trench coat and sunglasses looking around furtively and trying to sneak away from the sushi place. You nonchalantly walk up to them and to your surprise, you find that it’s your CS 18 TA Estelle! Could she be the one that left all these messages? You confront her and she’s surprised to see you. She sheepishly admits that she loves Salmon Rolls more than anything in the world, but she can’t let anyone find out. She swears you to secrecy and to prove her innocence, she opens her trench coat to reveal a mountain of bags containing nothing but Salmon Rolls. You ask her how she plans to get
back to campus with all of her contraband and she blanches. You need to get back to Providence, but you can’t just go through the normal channels. You need a way to book enough train seats to accommodate Estelle’s precious Salmon Rolls. She can’t just leave them sitting around the train for anyone to see though — the seats have to be consecutive. She tasks you with designing a SeatMap class to do just that.

For this segment, rather than use Java’s LinkedList class, we’re going to use a variation on lists called the ArrayList. Think of array lists as Linked Lists with the ability to quickly hop to a specific item in the list (we’ll see how this works next week). ArrayLists are nice when you have data (such as seats) that can be indexed or referenced by numeric values. Calendars are a good example: they are organized around consecutive numeric dates, and we often want to access information for a specific date.

### 3.2 Introduction to ArrayLists

You import ArrayLists into your file with the following line:

```java
import java.util.ArrayList;
```

The documentation (linked here) for ArrayList summarizes the available methods. For this assignment, you may only use the methods `add`, `get`, `set`, `clear`, `size` and `remove`. Note that there are two methods for `add`, one that adds an element to the end of the ArrayList and one that ends an element at a particular index of the ArrayList. Feel free to use either. You may assume all of the methods you will be using have constant runtime.

The following code shows an example of working with ArrayLists: it creates an ArrayList with 31 entries (for a calendar month), where each day stores the flavor to be featured at an ice cream store. It sets up each day to default to Vanilla, then updates (sets) the flavors for a couple of specific days. Finally, it counts how many days will still feature Vanilla.

```java
import java.util.ArrayList;

public class ArrayListExample {
    // A sample array list of ice cream flavor of the day

    public static void main(String[] args) {
        // create a arraylist with 31 days
        // they will be indexed as 0 .. 30
        ArrayList<String> flavorOfDay = new ArrayList<String>(31);

        // all flavors default to Vanilla
        for(int i = 0; i < 31; i++) {
            flavorOfDay.add(i, "Vanilla");
        }

        // update the flavors on certain days
        flavorOfDay.set(1, "Pumpkin");
        flavorOfDay.set(14, "Hazelnut");
        flavorOfDay.set(22, "Bubble Gum");

        // count how many days are still at the default
```
The sample uses a new kind of for-loop to set the default – one that iterates through a sequence of
numbers rather than through the items in a list. For setting the default flavors, it is easiest to loop
through the dates, and add “Vanilla” in for each date. Contrast this to the other for-loop style used
at the bottom of the sample for counting the number of times “Vanilla” is in the list.

For this exercise, you can use the numeric style loop if it is easiest to loop through the index numbers
into your array, rather than the contents. If your computation only needs the contents, however,
you should use the style of for-loop that you worked on in lab.

### 3.3 Implementing SeatMap

The constructor for the SeatMap class should take in one parameter, an integer specifying the
number of seats in the row as a whole. The class itself should contain the following:

- A single ArrayList of booleans, which represents the seat availability, where false represents
  a taken seat, and true represents an empty seat. Upon constructing a SeatMap object, every
  index in this ArrayList should be true (i.e. the row of seats should be entirely empty).

- A findBlock method, which takes a seat number from which to start searching and the
  number of (consecutive) seats that someone wants to find. The method returns either (a) a
  seat number from which there are the requested number of consecutive seats available, or (b)
  throws a RuntimeException (with string “no seats”) if there are no blocks of open seats at
  the requested size.

  We will have you implement this method two different ways (see below).

- The bookSeats method, which takes the starting seat number and the number of consecutive
  seats to reserve and sets all seats to reserve to false in the array. This method should assume
  that the desired number of seats is available starting from the starting seat (i.e., that someone
  ran findBlock before calling bookSeats).

  You must write this method with the parameters in the order as shown below:

  ```java
  public void bookSeats(int startSeat, int numSeats)
  ```

**Task:** Create the SeatMap.java class with the appropriate constructor, and implement the
bookSeats method.

Now, let’s look at two (of many!) different ways to implement findBlock. For each of the findBlock
methods, the first parameter must be the starting seat (int), and the second parameter must be the
number of seats (int). Otherwise you will fail our tests (and get a low grade). Don’t worry about (or check for) bad inputs (like a negative start seat). Just focus on handling valid inputs.

**Task:** In your SeatMap class, create the `findBlock1` method, which uses two for loops (one inside the other) to find the requested blocks of seats. The outer loop will iterate across each seat, while the inner loop will iterate starting from a given seat, checking each consecutive seat to see if it’s open. For this part, don’t use `break` (if you happen to know it), and don’t worry about efficiency. The parameter ordering for `findBlock1` and `findBlock2` should be the following:

```java
public int findBlock1(int startSeat, int numSeats)
```

**Note:** For both `findBlock1` and `findBlock2` you should book the seats that you found, using `bookSeats` (if you found seats). After the seats have been booked, the method should return the seat number from which you started booking seats.

You successfully complete `findBlock1`, but you have some reservations. In particular, you feel that this is inefficient. Despite the fact that the inner loop might find an occupied seat, you continue to iterate, which is wasteful. You decide to step it up, and make an even faster version.

**Task:** Create a helper method `checkSeats`, which takes the seat to start checking from and the number of seats to look for. It then returns a boolean indicating whether the requested number of seats is available from the given start seat. This method should return as soon as an occupied seat is found (i.e., as soon as you know that you can’t find a block of the appropriate size starting from a given seat). Note while this function could be written using recursion or a for loop, write it using a while loop.

You must write this method with the parameters in the order as shown below:

```java
public boolean checkSeats(int startSeat, int numSeats)
```

**Task:** Create a method `findBlock2`, which adapts the implementation of `findBlock1` to replace the inner loop with your newly created `checkSeats` method.

**Note:** The parameter ordering for this method should be the same as `findBlock1`.

**Note:** (does not need to be turned in) There is another speedup you could make to your program, this time with the outer loop. This outer loop still checks seats that can’t work as it checks every seat. However, since you know the locations of filled seats, you can use this information to skip checking some seats. For example, if you just checked a seat and it didn’t work, you can be sure that the seat next to it also won’t work, as it is either occupied, or it is unoccupied, but is also eventually going to be broken up by the later occupied seat which caused the previous seat to fail.

**Task:** Test all of your methods carefully. Additionally, since we are working with mutable data, be sure to create a setup method to use between your tests, the same way you did for `DoublyLinkedList` testing.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)