Simple Data Structures, Iterators, and JUnit

1 An Introduction to Data Structures

There are a few introductory data structures that we will go over which will serve as a base for many of the concepts that we go over later on in this book.
The first data structure that we'll briefly discuss is an ***array***. The array is a built in data structure in Java that is a fixed-size collections of elements of the same type. To declare an array we do as follows:

```
1 int arr = new arr[8]
```

The above array will be have 8 available spots that can contain the int type. In order to instantiate one item in the array, we do as follows.

```
1 arr[3] = 5;
```

This will make the 4th item, the one at the 3rd index, to be 3- note that the first element in an array starts at the index 0. Attempting to access an item that has not yet been instantiated yet will result in a *NullPointerException* and trying to access an item in the array whose index is not in the array would result in an *ArrayOutOfBoundsException*. Below is an example of what would result in a *NullPointerException* followed by an *ArrayOutOfBoundsException*.

```
1 int i = arr[0];
2 arr[8] = 10;
```

Say that we feel like adding many elements to a sequence, but we don't want to resize our array over and over. To fix this we may wanted to use a linked data structure. In this type of data structure, each item would have a *first* and *rest* variable. The first would be the actual element stored, and the rest would be the rest of the list. The typical API for these is as follows.

```
1 void addFirst(int x);
2 int getFirst();
3 int size();
4 void addLast(int x);
5 int getLast();
```

Note that the above example is tailored towards ints and it can be made to hold any type using generics. To have pointers to various nodes, we could use a private helper class within our Linked Data Structure class- once again, this is tailored to ints for simplicity, but we can make it hold generics.

```
1 private class Node{
2       int item;
3       Node next;
4       public Node(int i, Node n){
5             item = i;
6             next = n;
7       }
8 }
```

A problem that one can realize while coding is the *addLast* method is that you will need to have a special case to deal with. With all these special cases, we tend to get unwieldy code that just is not practical. To deal with such a problem, we will briefly discuss the concept of the ***sentinel node***. Basically, what we will do is create a "dummy" node that is instantiated whenever the Linked data structure is . This dummy node would be exactly like any other node, with the special case that it will never be null. This allows us to make much fewer checks when creating methods.

Though the *addLast* method has been simplified, we can recognize that anytime we would want to add the the end or retrieve anything using the naive method (traversing the list all the way to the end) could

take a lot of time, the length of the list to be precise. Well to fix this problem, we can optimize our Linked Data Structure by adding a "backwards pointer". This will point to the previous element. Thus, in our finalize approach, the first node would have a previous pointer that points to the end and the last node would have a next pointer that points to the front. This makes it so all of our methods (addFirst, addLast, getFirst, getLast, remove()) would all be very fast.

2 Iterators and Iterables

When we are given some sort of *collection*, that is some sort of list or set that has various items, we may want to be able go over all of them. An *iterable* is a series of elements that can be iterated over while an *iterator* is a class that manages the iteration over the iterable. To make a class an Iterable, simply implement the Iterable interface.

```
1 public interface Iterable<T>{
2     Iterator<T> iterator();
3 }
```

The method *iterator* returns a class that implements the iterator interface. Frequently, iterators are implemented as private classes within iterables.

```
1 public interface Iterator<T>{
2     boolean hasNext();
3     T next();
4     default void remove();
5 }
```

The method *remove* does not need to be implemented, to do so, you will need to reverse the iterator's previous *Next* call. The *hasNext* method checks if the iterable has another element. In this method, nothing, the state of the iterator should not change as it can have a negative consequence if you call *next* again. The *next* method returns the item of the next element in the iterable and moves the pointer to the next element.

3 JUnit Tests

Frequently, in the real world, you will not get autograders. In order to make sure that our programs work, we can use JUnit. Using JUnit requires us to import the package as follows

```
1 import org.junit.Test;
2 import static org.junit.Assert.*;
```

After doing this, we will need to declare certain methods as test methods, we can do that by writing *@test* above every test that we want to have some sort of test. While creating tests, it is very important that you separate your tests. Do not have one large test that checks everything at once, otherwise you will not be able to properly debug. What I suggest to do is make tests for the basic functionality of your program followed by more complicated tests that test two methods together. Oftentimes, test classes are as long or longer than classes that have your actual code.

Common tests in Junit involve the following:

```
1 assertEquals(expected, actual);
2 assertArrayEquals(expectedarray, actualarray);
3 assertTrue(value);
4 assertFalse(value);
```