UC Berkeley – Computer Science
CS61B: Data Structures

Midterm #1, Spring 2021

This test has 7 questions across 18 pages worth a total of 1280 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use unlimited written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. **Write the statement out below in the blank provided and sign your name. You may do this before the exam begins.**

"I have neither given nor received any assistance in the taking of this exam."

____

____

Signature: ___________________________

#	Points	#	Points
1	240	5	220
2	120	6	110
3	120	7	160
4	310	8	
		TOTAL	1280

Name: ______________________________

SID: _______________________________

GitHub Account # : sp21-s_____

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones you are comfortable with first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.**
- Not all information provided in a problem may be useful, and **you may not need all lines.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs in the exam, we'll announce a fix. **Unless we specifically give you the option, the correct answer is not 'does not compile.'**
- ○ indicates that only one circle should be filled in.
- □ indicates that more than one box may be filled in.
- For answers which involve filling in a ○ or □, please fill in the shape completely.

Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹_______________☺].
After exam: [☹_______________☺].

1. Cal Central.

a) (30 points). Let's start with a short helper method that will be useful in later questions. Fill in public static boolean contains(int[] a, int x) so it returns true if and only if the integer array a contains the integer x. Your solution may use at most **9 lines**. (Clarification: you can assume a will never be null.)

```
public class Utils {
    public static boolean contains(int[] a, int x) {
        for (int i=0; i < a.length; i += 1) {
                if (x == a[i]) {
                        return true;
                }
        }
        return false;
      }
}
```

For the remainder of this problem we'll be trying to complete the CalStudent and CourseCatalog classes given below. Quick note: You might observe that these classes are poorly designed (public instance variables and a bad choice of data structure). We'll address this in part d of this problem.

```
public class CourseCatalog {
    public int[] courses;

    public CourseCatalog() {
        courses = new int[0];
    }

    public void offerNewCourse(int courseid) {
        /* ... */
    }
}

public class CalStudent {
    public String name;
    public int[] courses;
    public int numOfCourses;

    public CalStudent(String name) {
        this.name = name;
        this.courses = new int[4];
        numOfCourses = 0;
    }

    public void addCourse(int courseid, CourseCatlog cc) {
        /* ... */
    }
}
```

The behavior we are interested in is as follows:

- Courses at Cal are referred to by their integer courseid.
- To add a course for CalStudent x , we call the x.addCourse(int courseid, CourseCatalog cc) method. If the course obeys the three restrictions below, the course is added to the first available spot in x's length 4 courses array. If it does not obey the restrictions, nothing happens.
    1. Students may only take courses that are available in the course catalog.
    2. Students may only be in at most 4 courses at the same time.
    3. Students cannot add the same class more than once.
- To offer a new course to a CourseCatalog called cc, we call the cc.offerNewCourse(int courseid) method. If the course is not already in the catalog, it is added. If it is already in the catalog, nothing happens.

Here is an example usage of these classes:

```
CourseCatalog cc = new CourseCatalog(); -> initially empty
CalStudent omar = new CalStudent();
omar.addCourse(22076, cc); // nothing added, invalid course
cc.offerNewCourse(22076);  // 22076 added to course catalog
cc.offerNewCourse(22076);  // nothing added, 22076 already exists
cc.offerNewCourse(24890);  // 24890 added to course catalog
omar.addCourse(22076, cc); // omar.courses: [22076, 0, 0, 0]

omar.addCourse(22076, cc); // omar.courses: [22076, 0, 0, 0] (no change)
omar.addCourse(24890, cc); // omar.courses: [22076, 24890, 0, 0]

cc.offerNewCourse(22078);  // 22078 added to course catalog
cc.offerNewCourse(33150);  // 33150 added to course catalog
cc.offerNewCourse(28108);  // 28108 added to course catalog

omar.addCourse(22078, cc); // omar.courses: [22076, 24890, 22078, 0]
omar.addCourse(33150, cc); // omar.courses: [22076, 24890, 22078, 33150]

omar.addCourse(28108, cc); // [22076, 24890, 22078, 33150] (no change)
```

b) (120 Points) First, let's implement the void offerNewCourse(int courseid) method. You may use your answer to part a (Utils.contains) as if it is correct. Do not worry about efficiency.

```
public class CourseCatalog {
    public int[] courses;

    public CourseCatalog() {
        courses = new int[0];
    }

    public void offerNewCourse(int courseid) {
        if (!Utils.contains(courses, courseid)) {
            int[] newArr = new int[courses.length + 1];
            for (int i = 0; i < courses.length; i++) {
                newArr[i] = courses[i];
            }
            newArr[courses.length] = courseid;
            courses = newArr;
        }
    }
}
```

c) (60 Points) Next, lets implement the void addCourse(int courseid) method. You may use your answer to part a (Utils.contains) as if it is correct, and you may assume that the CourseCatalog has been correctly created. Do not worry about efficiency. Remember that the && operator is the equivalent of "and" in Python or the equivalent of the & operator in MATLAB.

```
public class CalStudent {
    public String name;
    public int[] courses;
    public int numOfCourses;

    public CalStudent(String name) {
        this.name = name;
        this.courses = new int[4];
        numOfCourses = 0;
    }

    public void addCourse(int courseid, CourseCatalog cc) {
        if (numOfCourses < 4 && !Utils.contains(courses, courseid) &&
Utils.contains(cc.courses, courseid) {
            this.courses[numOfCourses] = courseid;
            numOfCourses += 1;
        }
    }
}
```

d) (30 Points) The design of the classes above is poor. One issue is that the classes expose their instance variables as public. Instead, access to these variables should be private and the classes should provide methods that do semantically useful tasks, e.g. the CourseCatalog might have a boolean courseNumberExists(int i) method.

Another significant issue is the poor choice of data structure, specifically int[] in both classes. The resulting code is awkward and overly verbose. As we said in the first lecture "Good programmers care about the data structures [in their code]." One better choice of data structure would have been a List. Give one reason why a List is better.

List Advantage:

- The size of a List is flexible, so we don't need to create an entirely new array whenever we add a courseid in offerNewCourse( ).
- A list keeps track of its own size, so we won't have to use a separate numOfCourses variable to track the number of courses added.

Note that later in 61B, we will learn about the Set, which is an even better data structure for this task than a List.

2. WWJD (120 points). Consider the code below:

```
1: public class Egg {
2:     public String owner;
3:     public static String staticOwner;
4:
5:     public Egg(String name) {
6:         this.owner = name;
7:         staticOwner = name;
8:     }
9:
10:    public void weirdExchange(String borrower, Egg g2) {
11:        String lender = this.owner;
12:        this.owner = borrower;
13:        g2.owner = lender;
14:        g2 = new Egg("Omar");
15:        System.out.println("g2 owner: " + g2.owner);
16:    }
17:
18:    public static void main(String[] args) {
19:        Egg g1 = new Egg("Itai");
20:        Egg g2 = new Egg("Connor");
21:
22:        String p1 = "Linda";
23:        g1.weirdExchange(p1, g2);
24:        System.out.println("g1 owner: " + g1.owner);
25:        System.out.println("g2 owner: " + g2.owner);
26:        System.out.println("Egg owner: " + Egg.staticOwner);
27:    }
28:}
```

What will be printed when lines 23, 24, 25, and 26 execute?

For each line, check the bubble for the correct output or for "Error" if the line causes an error of any kind. If a line causes an error, ignore it and still consider the output of lines below it.

Output printed when line 23 executed:	○ `g2 owner: Itai` ○ `g2 owner: Connor` ○ `g2 owner: Linda` [✓ `g2 owner: Omar`] ○ `Error`
Output printed when line 24 executed:	○ `g1 owner: Itai` ○ `g1 owner: Connor` [✓ `g1 owner: Linda`] ○ `g1 owner: Omar` ○ `Error`
Output printed when line 25 executed:	[✓ `g2 owner: Itai`] ○ `g2 owner: Connor` ○ `g2 owner: Linda` ○ `g2 owner: Omar` ○ `Error`
Output printed when line 26 executed:	○ `Egg owner: Itai` ○ `Egg owner: Connor` ○ `Egg owner: Linda` [✓ `Egg owner: Omar`] ○ `Error`

3. WWJDMS (120 points). Suppose we define the classes below:

```
public interface Instrument {
    default public String getName() {
        return "instrument";
    }
}

public class Guitar implements Instrument {
    public String getName() {
        return "guitar";
    }
}

public interface Music {
    default public void play(Instrument i) {
        System.out.println("music on a " + i.getName());
    }
}

public class Rock implements Music {
    public void play(Instrument i) {
        System.out.println("rock on a " + i.getName());
    }
    public void play(Guitar g) {
        System.out.println("RPG");
    }
}

public class Jota implements Music {
    public void play(Guitar g) {
        System.out.println("JPG");
    }
}
```

For each of the problems below, assume we have declared variables i and g as follows:

```
Instrument i = new Guitar();
Guitar g = new Guitar();
```

a) (30 Points) What will be the output of the two calls to play below? If the first line causes an error, ignore it and still consider the output of the second line.

```
Rock r = new Rock();
r.play(i);
r.play(g);
```

`r.play(i);`	○ `music on a instrument` ○ `rock on a instrument` [✓ ○ `rock on a guitar`] ○ `music on a guitar` ○ `RPG` ○ `Error`
`r.play(g);`	○ `music on a instrument` ○ `rock on a instrument` ○ `rock on a guitar` ○ `music on a guitar` [✓ ○ `RPG`] ○ `Error`

b) (30 Points) What will be the output of the two calls to play below? If the first line causes an error, ignore it and still consider the output of the second line.

```
Music mr = new Rock();
mr.play(i);
mr.play(g);
```

`mr.play(i);`	○ `music on a instrument` ○ `music on a guitar` ○ `rock on a instrument` [✓ ○ `rock on a guitar`] ○ `RPG` ○ `Error`
`mr.play(g);`	○ `music on a instrument` ○ `music on a guitar` ○ `rock on a instrument` [✓ ○ `rock on a guitar`] ○ `RPG` ○ `Error`

c) (30 Points) What will be the output of the two calls to play below? If the first line causes an error, ignore it and still consider the output of the second line.

```
Jota j = new Jota();
j.play(i);
j.play(g);
```

`j.play(i);`	○ `music on a instrument` [✓ ○ `music on a guitar`] ○ `JPG` ○ `Error`
`j.play(g);`	○ `music on a instrument` ○ `music on a guitar` [✓ ○ `JPG`] ○ `Error`

d) (30 Points) What will be the output of the two calls to play below? If the first line causes an error, ignore it and still consider the output of the second line.

```
Music mj = new Jota();
mj.play(i);
mj.play(g);
```

`mj.play(i);`	○ `music on a instrument` [✓ ○ `music on a guitar`] ○ `JPG` ○ `Error`
`mj.play(g);`	○ `music on a instrument` [✓ ○ `music on a guitar`] ○ `JPG` ○ `Error`

4. BLList. (310 points) Before starting this problem, be aware that **for all parts of this problem, you may assume the earlier parts were done correctly and can be used later, e.g. feel free to use getNode from part c on remove from part e.**

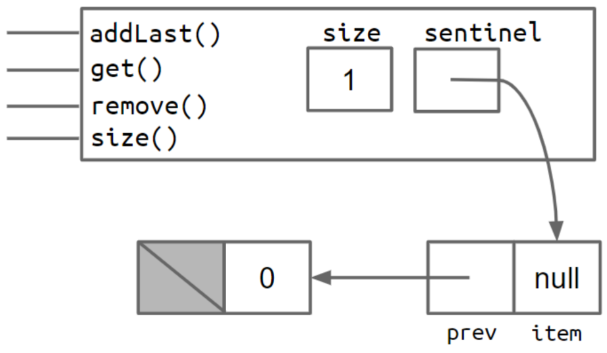
Suppose we have the BLList class defined below as follows. The addLast and get methods behave exactly like you'd expect from a list, i.e. if we call addLast(5) then addLast(10), we'll end up with [5, 10], i.e. get(0) will return 5 and get(1) will return 10.

```
public class BLList<T> {
    public void addLast(T item) // adds an item to the end of the list
    public T get(int i)         // gets the ith item
    public void remove(int i)   // removes the ith item
    public int size()           // returns the number of items
}
```

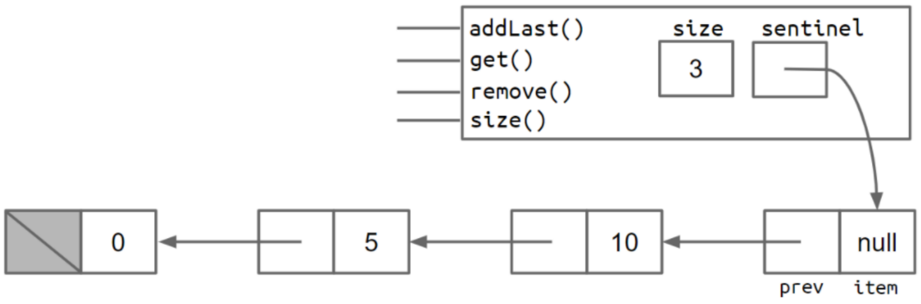
a) (40 Points) Fill in the JUnit test below to test that addLast and get work correctly. Your list should include at least 3 elements. You may use a maximum of **12 lines**. (Clarification: Assume any call to get has an in-bounds index.)

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

public class TestBLList {
      @Test
      public void testAddLastAndGet() {
          BLList<Integer> b = new BLList<>();
          b.addLast(1);
          b.addLast(2);
          b.addLast(3);
          assertEquals(1, b.get(0));
          assertEquals(2, b.get(1));
          assertEquals(3, b.get(2));

      }
}
```

Let's now start to implement the BLList class. BLList stands for "backward linked list".
A BLList consists of a sentinel node which points backwards at the last item in the list. For example, if we create a BLList then call addLast(0), the resulting list looks like:



If we then call addLast(5), then addLast(10), the list will now look like the picture below, i.e. get(0) would return 0, get(1) would return 5, and get(2) would return 10, i.e. we can think of the list as [0, 5, 10].



b) (60 Points) Fill in the addLast function below.

```
public class BLList<T> implements MT1List<T> {
    private class Node {
        private T item;
        private Node prev;

        public Node(T i, Node p) {
            item = i;
            prev = p;
        }
    }

    private Node sentinel;
    private int size;

    public BLList() {
        sentinel = new Node(null, null);
        size = 0;
    }

    public void addLast(T item) {
        sentinel.prev = new Node(item, sentinel.prev);
        size += 1;
    }
    ...
}
```

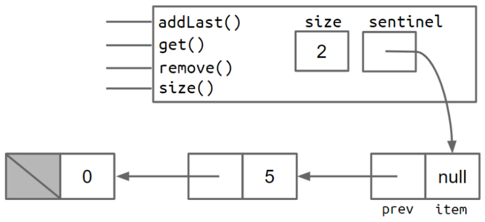
c) (80 Points) Next we'll write a useful private helper method called getNode(int i) that uses **recursion** to find the ith node in the BLList. For example, for the picture above, getNode(2) would return the node containing 10. If i is equal to the number of items in the list, then getNode should return the sentinel node. For example, for the picture above, getNode(3) would return the sentinel. The behavior is not defined for other values of i -- that is, for the diagram above, for i values that are not 0, 1, 2, or 3, we don't care what value your code returns or even if it errors. **Your code must not use any loops, i.e. no for or while.**

```
public class BLList<T> implements MT1List<T> {
    ...
    private Node sentinel;
    private int size;
    ...

    private Node getNode(int i) {
        return getNode(size() – i, sentinel);
    }

    private Node getNode(int i, Node p) {
        if (i == 0) {
            return p;
        }
        return getNode(i - 1, p.prev);
    }
}
```

d) (50 Points) Instead we could have written an iterative version of getNode(int i). In this part of the problem, write getNode(int i) which does not utilize recursion. It should behave exactly as in part c. You may use a maximum of **11 lines.**

```
public class BLList<T> implements MT1List<T> {
    ...
    private Node sentinel;
    private int size;
    ...

    private Node getNode(int i) {
      int numNodesToHop = size() – i;
      Node p = sentinel;

      while (numNodesToHop > 0) {
            p = p.prev;
            numNodesToHop -= 1;
      }

      return p;
    }
}
```

Now let's write remove(int i). Like the methods above, the first item on the list is item 0. For example, if we call remove(2) on the size 3 BLList above, we'd end up with:



e) (80 Points) Fill in the code below. You may use getNode(int i) even if your answers to part c and part d were wrong. The remove method should do nothing if it is called with a index that is out of bounds. (i.e. remove(3) should have no effect on a BLList with 3 elements in it, i.e. no effect on a BLList of size 3, i.e. the sentinel should not be removed). You may use a maximum of **7 lines.** Recall the || operator is the equivalent of "or" in Python or the equivalent of the | operator in MATLAB.

```
public class BLList<T> implements MT1List<T> {
    ...
    private Node sentinel;
    private int size;
    ...

    public void remove(int i) {
        if (i < 0 || i >= size()) {
          return;
        }
        Node parentOfNodeToRemove = getNode(i + 1);
        parentOfNodeToRemove.prev = parentOfNodeToRemove.prev.prev;
        size -= 1;

    }
}
```

Since get(int i) and size() are trivial, we won't have you write them. However, you're welcome to use these methods in any of the problem above if it seems like they're useful.

5. MT1List (220 Points).

Suppose we have the MT1List interface defined as follows:

```
public interface MT1List<T>; {
    public void addLast(T item);    // adds an item to the end of the list
    public T get(int i);            // gets the ith item
    public void remove(int i);      // removes the ith item
    public int size();              // returns the number of items
    public MT1List<T>; createList(); // returns a new empty MT1List&lt;T&gt;
}
```

a) (50 Points) Suppose we add a default public MT1List<T> getEvens() method which returns a new list with all the even-indexed items. For example, suppose we have an MT1List X: ["ape", "banana", "cow", "dog"]. Then X.getEvens() will return a new MT1List: ["ape", "cow"]. Fill in the code below. You may use a maximum of **10 lines**. X should not be changed in any way by the call to X.getEvens().

```
public interface MT1List<T> {
    ...
    default public MT1List<T> getEvens() {
         MT1List<T> returnList = createList();
         for (int i = 0; i < size(); i += 1) {
                if (i % 2 == 0) {
                       returnList.addLast(get(i));
                }
        }
        return returnList;
    }
}
```

Alternate solution:

```
public interface MT1List<T> {
    ...
    default public MT1List<T> getEvens() {
         MT1List<T> returnList = createList();
         for (int i = 0; i < size(); i += 2) {
                returnList.addLast(get(i));
        }
        return returnList;
    }
}
```

b) (80 Points) Suppose we add another default method: default public MT1List<T> remove(int start, int end). This method removes all items in the list between indices start and end, inclusive, and returns a list with all of the removed items.

For example, if we have a MT1List called X: ["ape", "banana", "cow", "dog", "elephant"], then after a call to X.remove(1, 3), X will be ["ape", "elephant"], and the return value of the function will be a new MT1List ["banana", "cow", "dog"]. Fill in the method below. You may use a maximum of **10 lines**, and you may assume that the arguments are valid (start is less than end, and both are valid indices).

```
public interface MT1List<T> {
  ...
  default public MT1List<T> remove(int start, int end) {
      MT1List<T> returnList = createList();
      int numToRemove = end – start + 1;
      for (int i = 0; i < numToRemove; i += 1) {
            returnList.addLast(get(start));
            remove(start);
      }
      return returnList;
    }
}
```

Alternate solution:

```
public interface MT1List<T> {
  ...
  default public MT1List<T> remove(int start, int end) {
      if (start > end) {
            return createList();
      }
      MT1List<T> partial = remove(start, end – 1);
      partial.addLast(get(start));
      remove(start);
      return partial;
    }
}
```

c) (90 Points) Lastly, suppose we add another default method: default public void reverse(). This method reverses the list and returns nothing. For example, if we have a MT1List called X: ["ape", "banana", "cow", "dog", "elephant"], then call X.reverse(), then X will become: ["elephant", "dog", "cow", "banana", "ape"]. You may not use the new keyword for this problem, nor may you call createList. You may use a maximum of **10 lines.**

```
public interface MT1List<T> {
      ...
    default public void reverse() {
      int N = size();
      for (int i = N -2; i >= 0; i -= 1) {
            T x = get(i);
            remove(i);
            addLast(x);
      }
    }
}
```

Alternate solution:

```
public interface MT1List<T> {
      ...
    default public void reverse() {
      if (size() > 0) {
            T item = get(0);
            remove(0);
            reverse();
            addLast(item);
      }
    }
}
```

Note: If you want to do so, you can use any of these three default methods as helper methods for any of the others. However, our solutions are completely independent and do not use each other.

6. AListBug (110 points).

The AListMax100 below is a partial implementation of an array list that can store at most 100 items. The implementation of the AListMax100 class has a serious bug that happens even when the list still has plenty of space left. The bug is **not that the AList can run out of space! The bug is also not an off-by-one error!**

```
1: public class AListMax100<Item> {
2:     private Item[] items;
3:     private static int size;
4:
5:     /** Creates an empty list. */
6:     public AListMax100() {
7:         items = (Item[]) new Object[100];
8:         size = 0;
9:     }
10:
11:    /** Inserts X into the back of the list if there is room. */
12:    public void addLast(Item x) {
13:        if (size >= 100) {
14:            return;
15:        }
16:        items[size] = x;
17:        size = size + 1;
18:    }
19:
20:    /** Returns the item from the back of the list or null if empty. */
21:    public Item getLast() {
22:        if (size == 0) {
23:            return null;
24:        }
25:        return items[size - 1];
26:    }
27:
29:    /** Returns the number of items in the list. */
30:    public int size() {
31:        return size;
32:    }
33:
34:    // other methods not shown
35:}
```

a) (15 Points) What single line number could we change so that the bug is fixed?

Line number: **3**

b) (15 Points) Describe how we would change the line of code to fix the bug.

Description of the bug fix: **Remove the “static” keyword in front of size**

c) (80 Points) Below, write a JUnit test that fails because of the bug. You may use a maximum of **10 lines.**

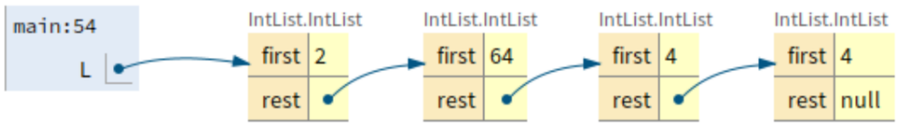
```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestAListMax100 {
      @Test
      public void test() {

            AListMax100<Integer> L = new AListMax100<>();
            L.addLast(5);
            L.addLast(10);

            AListMax100<Integer> L2 = new AListMax100<>();
            L2.addLast(5);

            assertEquals(2, L.size());

      }
}
```

d) PNH (0 points). In what country would you expect to dance a Jota?

Answer: **Spain.**

7. Tilt (160 points). For this problem, we will be re-implementing the tilt method from 2048! However, this time around, we will represent a row of the board as a LinkedList.

For example, if one row of the 2048 board is [2, 64, 4, 4], then we will represent this with the IntList below:



Below, write the tiltRowLeft method so that the given IntList (defined in the reference sheet) is transformed into the result of a tilt operation towards the left. For this problem we assume you know the basic rules for tilt operations from project 0. However, if you do not, they are repeated at the bottom of this page. For simplicity on the exam:

1. You may assume that there are no empty tiles in the IntList given to you.
2. If there are merge operations, you do not need to create empty tiles at the end of the list.

For example, your code should pass the following JUnit tests:

```
@Test
    public void testNoMerge() {
        IntList L = IntList.of(2, 4, 2, 4);
        tiltRowLeft(L);
        IntList expected = IntList.of(2, 4, 2, 4);
        assertEquals(expected, L);
    }

    @Test
    public void testOneMerge() {
        IntList L = IntList.of(2, 2, 8, 4);
        tiltRowLeft(L);
        // Note that the operation below results in an
        // IntList of length 3! There are no 0 or null tiles!
        IntList expected = IntList.of(4, 8, 4);
        assertEquals(expected, L);
    }

    @Test
    public void testTwoMerges() {
        IntList L = IntList.of(2, 2, 4, 4);
        tiltRowLeft(L);
        IntList expected = IntList.of(4, 8);
        assertEquals(expected, L);
    }

    @Test
    public void testTrickyCase() {
        IntList L = IntList.of(2, 2, 4, 8);
        tiltRowLeft(L);
        IntList expected = IntList.of(4, 4, 8);
        assertEquals(expected, L);
    }
    @Test
    public void testMoreThanFour() {
        IntList L = IntList.of(2, 2, 4, 2, 8, 8, 4, 2, 2);
        tiltRowLeft(L);
        IntList expected = IntList.of(4, 4, 2, 16, 4, 4);
        assertEquals(expected, L);
    }

public static void tiltRowLeft(IntList tiles) {
    IntList p = tiles;
    while (p.rest != null) {
        if (p.first == p.rest.first) {
            p.first += p.first;
            p.rest = p.rest.rest;
            if (p.rest == null) { // Using this is optional.
                break;        // If you don't want to break,
            }                 // just put "false" in the blank.
        }
        p = p.rest;
    }
}
```

Alternate solution:

```
public static void tiltRowLeft(IntList tiles) {

    IntList p = tiles;
    while (p != null && p.rest != null) {
        if (p.first == p.rest.first) {
            p.first += p.first;
            p.rest = p.rest.rest;
            if (false) { // Using this is optional.
                break;        // If you don't want to break,
            }                 // just put "false" in the blank.
        }
        p = p.rest;
    }
}
```

Rules from the official project 0 page:

1. When moving tiles left, two tiles of the same value *merge* into one tile containing double the initial number.
2. A tile that is the result of a merge will not merge again on that tilt. For example, if we have 64 -> 2 -> 2 -> 4, and we move the tiles to the left, we should end up with 64 -> 4 -> 4, not 64 -> 8. This is because the leftmost 4 was already part of a merge so should not merge again.
3. When three adjacent tiles in the direction of motion have the same number, then the leading two tiles in the direction of motion merge, and the trailing tile does not. For example, if we have 32 -> 2 -> 2 -> 2 and move tiles left, we should end up with 32 -> 4 -> 2 not 32 -> 2 -> 4.

As a corollary of these rules, if there are four adjacent tiles with the same number in the direction of motion, they form two merged tiles. For example, if we have 4 -> 4 -> 4 -> 4, then if we move to the left, we end up with 8 -> 8. This is because the leading two tiles will be merged as a result of rule 3, then the trailing two tiles will be merged, but because of rule 2 these merged tiles (8 in our example) will not merge themselves on that tilt.