UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B **P. N. Hilfinger**
Fall 2018

Test #1

READ THIS PAGE FIRST. *Please do not discuss this exam with people who haven't taken it.* Your exam should contain 7 problems on 15 pages. Officially, it is worth 17 points (out of a total of 200).

This is an open-book test. You have 110 minutes to complete it. You may consult any books, notes, or other non-responsive objects available to you. You may use any program text supplied in lectures, problem sets, or solutions. Please write your answers in the spaces provided in the test. Make sure to put your name, login, and TA in the space provided below. Put your login and initials *clearly* on each page of this test and on any additional sheets of paper you use for your answers.

Be warned: my tests are known to cause panic. Fortunately, this reputation is entirely unjustified. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious. Should you feel an attack of anxiety coming on, feel free to jump up and run around the outside of the building once or twice.

Your name: _____ Login: _____

Login of person to your Left: _____ Right: _____

Discussion TA: _____

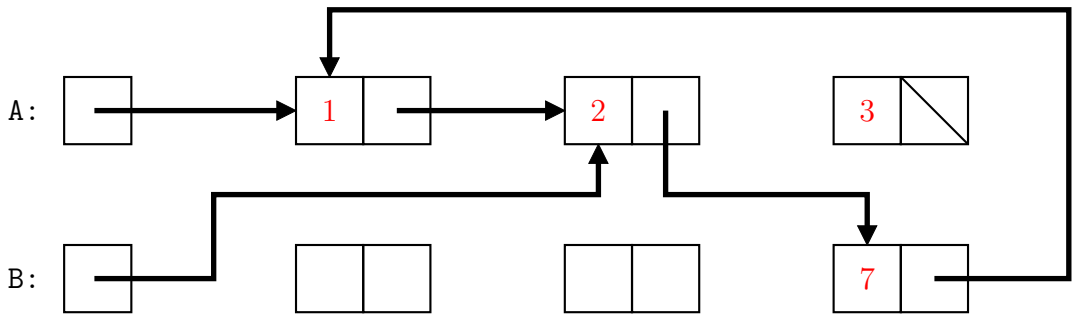
Reference Material.

```
/* arraycopy(FROM_ARR, FROM_INDEX, TO_ARR, TO_INDEX, LENGTH) */
import static java.lang.System.arraycopy;

public class IntList {
    /** First element of list. */
    public int head;
    /** Remaining elements of list. */
    public IntList tail;

    /** A List with head HEAD0 and tail TAIL0. */
    public IntList(int head0, IntList tail0)
    { head = head0; tail = tail0; }

    /** A List with null tail, and head = HEAD0. */
    public IntList(int head0) { this(head0, null); }

    /** Returns a new IntList containing the ints in ARGS. */
    public static IntList list(Integer ... args) {
        // Implementation not shown
    }

    /** Returns true iff L (together with the items reachable from it) is an
     *  IntList with the same items as this list in the same order. */
    @Override
    public boolean equals(Object L) {
        // Implementation not shown
    }

    /** Return the length of the non-circular list headed by L. */
    public int size() {
        // Implementation not shown
    }

}
```

1. [2 points]

(a) [1 point] Fill in the box and pointer diagram to show the variables and objects created and their contents after executing the given snippet, using the empty boxes provided. After the code executes, some objects may be unreachable from any named pointer variable (may be "garbage" to use the technical term); ***show them anyway.*** The double boxes represent `IntList` objects (see definition on page 2), with the left box containing the `head` field. You may not need all the boxes provided.

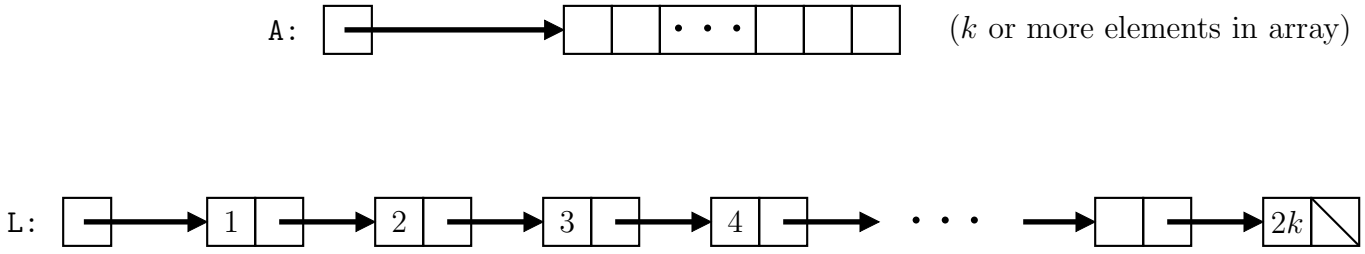
```
A = IntList.list(1, 2, 3);
B = A.tail;
B.tail = new IntList(7, A.tail.tail);
A.tail.tail.tail = A;
```



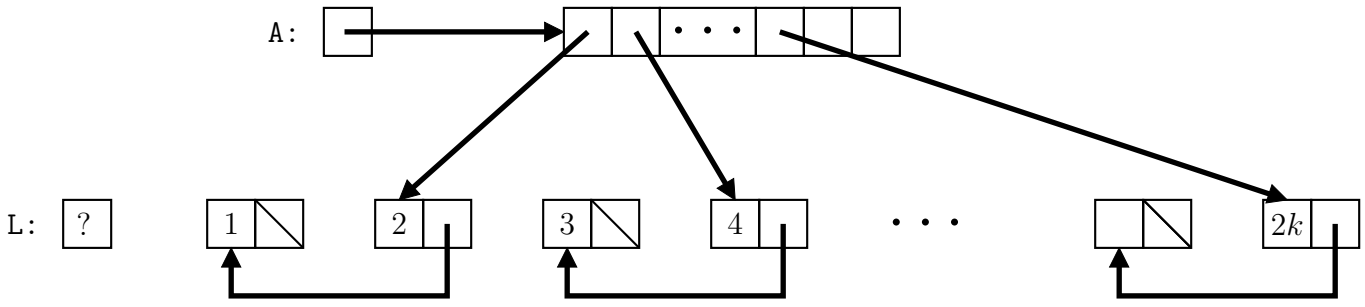
(b) [1 point] For this one, fill in the program so that it converts the *Before* diagram to the *After* diagram, given the variables `A` and `L`. Here, the group of boxes at the top is an array of `IntList`s. **Do not** create any additional `IntList` objects, and **do not** change the `head` of any `IntList` object. There is no variable k available in the program, but you can see that there is a relationship between the `head`s of the list objects and the indices of the array elements that point to them. Not all elements of the array need to be filled (that is, $k \leq$ `A.length`). The final value of `L` is unspecified.

```
while (L != null) {
    IntList next = L.tail.tail;
    A[L.head / 2] = L.tail;
    L.tail.tail = L;
    L.tail = null;
    L = next;
}
```

Before:



After:



2. [2 points] For each of the following programs, write the output produced by running the code in the blanks following the print calls.

(a) Be sure to read the code carefully; it is not entirely straightforward.

```
public class Account {
    private String name;
    private static int amount;

    public Account(String name) { this.name = name; amount = 0; }
    public Account(String name, int amount)
    { this.name = name; this.amount = amount; }

    public void deposit(int amount) { this.amount += amount; }
    public boolean withdraw(int amount) {
        if (amount > this.amount) {
            return false;
        } else {
            this.amount -= amount; return true;
        }
    }
    public int getAmount() { return this.amount; }
    public String getName() { return this.name; }

    public static void main(String[] args) {
        Account aliceBank = new Account("Alice", 200);
        Account billBank = new Account("Bill", 25);

        System.out.println(aliceBank.withdraw(200));

        System.out.println(aliceBank.getAmount() == 0);

        System.out.println(aliceBank.withdraw(1));
        billBank.deposit(61);
        System.out.println(billBank.getName() + " " + billBank.getAmount() + " "
                           + aliceBank.getName() + " " + aliceBank.getAmount());
    }
}
```

Output

false

false

true

Bill 85 Alice 85

(b)

```
public class ArrayStack {
    private int[] stack;

    public ArrayStack(int[] elems) {
        stack = elems;
    }

    public int pop() {
        int top = stack[0];
        shiftLeft(stack);
        return top;
    }

    public static void shiftLeft(int[] arr) {
        for (int i = 1; i < arr.length; i += 1)
            arr[i - 1] = arr[i];
    }

    public static void main(String[] args) {
        int[] elems = new int[] {1, 3, 5, 7, 9};
        ArrayStack stk = new ArrayStack(elems);
        elems[0] = stk.pop();

        System.out.println(stk.pop());

        System.out.println(elems[0]);
        shiftLeft(elems);
        stk.shiftLeft(elems);

        System.out.println(elems[0] == elems[4]);
    }
}
```

Output

1

5

true

EXTRA PAGE

3. [4 points] For this problem, we'll represent a sequence of `int`s as an `int` array whose first element contains the length of a sequence of `int`s that is stored at index 1, 2, ... in the same array. There may be additional array elements after that, but they are not used and their contents do not matter. For example, the 5-element sequence

```
[ 1, 3, 3, 2, 17 ]
```

could be represented as the array

```
{ 5, 1, 3, 3, 2, 17, 0, 1, 0 }
```

with 3 extra elements (shown underlined here). The function `riffle` takes an array of such sequences (which would therefore be a 2-D `int` array) and riffles them together into a sequence by taking the first item of the first sequence, the first item of the second sequence, etc., up to the first item of the last sequence, and then the second item of the first sequence, the second item of the second, etc. Sequences that have run out of elements are skipped when their turn comes. For example, the three sequences

```
[ 1, 3, 3, 2, 17 ], [ 3, -1, 9 ], [ 4, 8, 12, 13 ]
```

could be represented by the array

```
int[][] A =
 { { 5, 1, 3, 3, 2, 17, 0, 1 }, { 3, 3, -1, 9 }, { 4, 4, 8, 12, 13, 21 } };
```

As you can see, the first and third arrays have extra elements at the end (underlined) that are unused. After executing

```
int[] B = new int[16];
riffle(A, B);
```

the array `B` should contain

```
{ 12, 1, 3, 4, 3, -1, 8, 3, 9, 12, 2, 13, 17, 0, 0, 0 }
```

representing the 12-element sequence

```
[ 1, 3, 4, 3, -1, 8, 3, 9, 12, 2, 13, 17 ]
```

You may assume that the output array is at least the necessary size. Fill in `riffle` on the next page to meet this specification.

Observation: Since a 2D array of `int`s, `arr2d`, is actually a 1D array of 1D arrays of `int`s, the effect of the Java loop

```
for (int[] arr1d : arr2d) {
    ...
}
```

is to set `arr1d` to point to each row of `arr2d` in turn.

```
static void riffle(int[][] mlist, int[] result) {
    int k;

    result[0] = 0;
    k = 1;
    while (true) {
        int len0 = result[0];
        for (int[] row : mlist) {
            if (k <= row[0]) {
                result[0] += 1;
                result[result[0]] = row[k];
            }
        }
        if (len0 == result[0]) {
            break;
        }
        k += 1;
    }
    return result;
}
```

4. [1 point] Termites are members of the suborder Isoptera, and are now considered part of the order Blattodea. What type of insect comprises the majority of species in this order?

Cockroaches.

5. [4 points] The following methods are intended to produce the same results by two different means: recursive and iterative. Fill them in to obey their comments. For example, is `S` and `R` are `IntList`s containing

`S: [ 2, 6, 9, 15, 25, 50 ],` `R: [ -3, 2, 7, 15, 29, 60 ]`

then `demerge(S, R)` and `demerge1(S, R)` would both return

`[ 6, 9, 25, 50 ].`

(a)

```
/** Return a list of all items in SOURCE that are not in REMOVE.
 *  Assumes that the items in SOURCE and REMOVE are both sorted in
 *  ascending order (no duplicates).  Does not disturb the original
 *  data. */
static IntList demerge(IntList source, IntList remove) {
    if (source == null) {
        return null;
    } else if (remove == null) {
        return source;
    } else if (source.head == remove.head) {
        return demerge(source.tail, remove.tail);
    } else if (source.head < remove.head) {
        return new IntList(source.head, demerge(source.tail, remove));
    } else {
        return demerge(source, remove.tail);
    }
}
```

(b)

```
/** Return a list of all items in SOURCE that are not in REMOVE.
 *  Assumes that the items in SOURCE and REMOVE are both sorted in
 *  ascending order (no duplicates).  Does not disturb the original
 *  data. */
static IntList demergei(IntList source, IntList remove) {
    IntList result, last;

    result = last = null;
    while (source != null) {
        if (remove == null || source.head < remove.head) {
            IntList item = new IntList(source.head);
            if (last == null) {
                result = item;
            } else {
                last.tail = item;
            }
            last = item;
            source = source.tail;
        } else if (source.head == remove.head) {
            remove = remove.tail;
            source = source.tail;
        } else {
            remove = remove.tail;
        }
    }
    return result;
}
```

6. [2 points] Suppose we have the following classes (in separate files):

```
public class Truck {
    public void crumple() { }                    // A
    public static void collide(Truck t) { destroy(t); }
    public void destroy(Truck t) { }             // B
}

public class BigTruck extends Truck {
    public void honk() {  }                      // C
    public void crumple() { honk(); }            // D
}

public class MonsterTruck extends BigTruck {
    public void destroy(Truck t) {               // E
        super.destroy(t);
        t.crumple();
    }
    public void honk() {}                        // F
}

public class Pickup extends Truck {
}
```

Place X's in the table on the next page to indicate, for each of the indicated lines, whether it would cause a compilation error (CE), runtime error (RE), or would execute one or more of the methods marked A–G above. That is, if your answer is not CE or RE, put X for *ALL* the methods that are executed. If a line causes an error (compilation or runtime), assume that line is removed (i.e. never executed or compiled) when supplying answers for the lines following it.

```
Truck colossus = new MonsterTruck();
Truck godzilla = new BigTruck();
Truck peewee = new Pickup();
Truck inferno = new Truck();
MonsterTruck thunder = new MonsterTruck();
BigTruck heavyweight = new BigTruck();
```

Code	CE	RE	A	B	C	D	E	F
`godzilla.honk();`	X							
`((BigTruck) colossus).honk();`								X
`((BigTruck) peewee).honk();`		X						
`((MonsterTruck) colossus).honk();`								X
`thunder.collide(heavyweight);`				X	X	X	X	
`thunder.collide(colossus);`				X		X	X	X
`thunder.collide(peewee);`			X	X			X	
`thunder.collide(inferno);`			X	X			X	
`peewee.collide(thunder);`				X				
`inferno.collide(thunder);`				X				

7. [3 points] We have a utility method that is intended to give the effect of a **for** loop:

```
class Util {
    static void forit(int start, Pred test, Proc body) {
        while (test.apply(start)) {
              body.apply(start);
              start += 1;
        }
    }
}
```

The types `Pred` and `Proc` are interfaces. A certain client of `Util.forit` wants to use it to sum the elements of an integer array:

```
class Client {
    /** Return the sum of the elements of A. */
    static int sum(int[] A) {
        Body body = new Body(A);
        Util.forit(0, new EndTest(A), body);
        return body.result();
        // Observation: forit does not need result.
    }
}
```

Fill in the blanks below and the definitions on the next page to make the `sum` method work. Not all blanks need be used.

```
interface Pred {

    boolean apply(int x);
}

interface Proc {

    void apply(int x);
}
```

```
class EndTest implements Pred {

    private int[] arr;

    EndTest(int[] A) {

        arr = A;
    }

    public int apply(int x) {
        return x < arr.length;
    }
}

class Body implements Proc {

    private int[] A;
    private int accum;

    Body(int[] arr) {

        A = arr;
        accum = 0;
    }

    public void apply(int x) {

        accum += A[x];
    }

    int result() {

        return accum;
    }
}
```