CS61B, Fall 2015 **Test #1 Solution** **P. N. Hilfinger**

Reference Material.

```
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 = 0. */
    public IntList() { this(0, null); }

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

    /** Non-destructively returns a new IntList containing all my
     *  values that have indices >= START and < END. Undefined if
     *  any of the required items are non-existent or I is null.
     *  The result shares no objects with me. */
     public static IntList sublist(IntList L, int start, int end) {
           // 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
    }

}
```

1. [2 points] `logSwap` is a static method of the `ArrayDiddle` class that takes as arguments an array of `int`s and an index into that array. It takes the value at the specified index, v, and swaps it with whatever value, h, is at half that index (rounded down, as for Java integer division), if $v < h$. If the swap occurs, it then repeats the entire process, starting at the new (halved) index of v. If `logSwap` receives a starting index that is out of the array's bounds, `logSwap` throws an `IllegalArgumentException`.

Fill in the blanks below to finish implement `logSwap`. Put at most one semicolon per line. You need not use all the blanks.

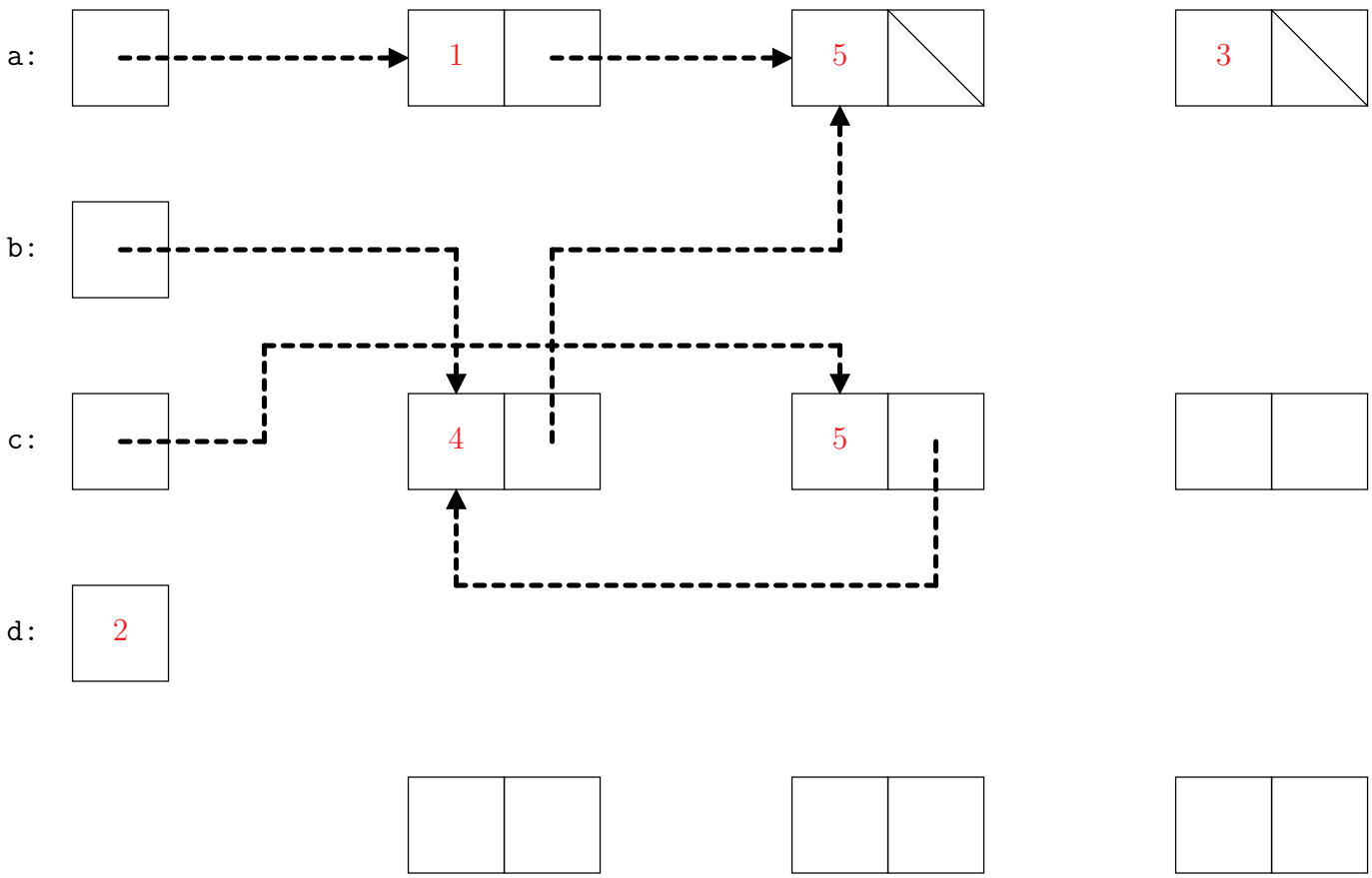
```
public class ArrayDiddle {
    /** Swap the value at ARR[INDEX] down the array (towards index 0),
     *  starting with the element halfway between INDEX and 0 and
     *  continuing to reduce the destination index by 1/2 as long as the
     *  value there is strictly smaller. Throws
     *  IllegalArgumentException if INDEX is out of bounds.
     *
     *  For example, if A initially contains { 7, 6, 5, 4, 3, 2, 1 },
     *  then after logSwap(A, 6), it contains { 1, 7, 5, 6, 3, 2, 4 }.
     *  If B initially contains { 7, 1, 5, 4, 3, 2, 1 }, then after
     *  logSwap(B, 5) it contains { 7, 1, 2, 4, 3, 5, 1 }.
     */
    public static void logSwap (int[] arr, int index) {


        if (index < 0 || index >= arr.length) {
             throw new IllegalArgumentException();
        }
        while (arr[index] < arr[index / 2]) {
            // There is no need to test index, since the while
            // condition will always become false eventually.
            int t = arr[index];
            arr[index] = arr[index / 2]; arr[index / 2] = t;
        }


    }
}
```

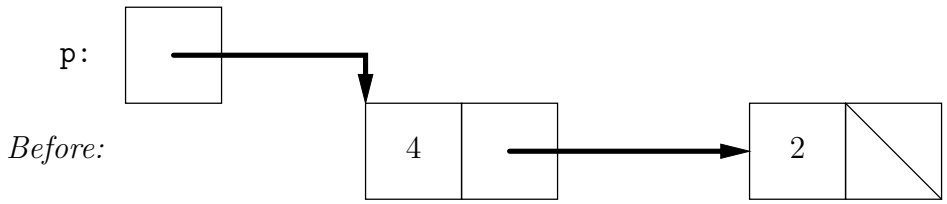
2. [2 points] Fill in the box-and-pointer diagram below to show the state of the program after executing the following code segment. You may not need to use all the boxes drawn. Show only the final state.

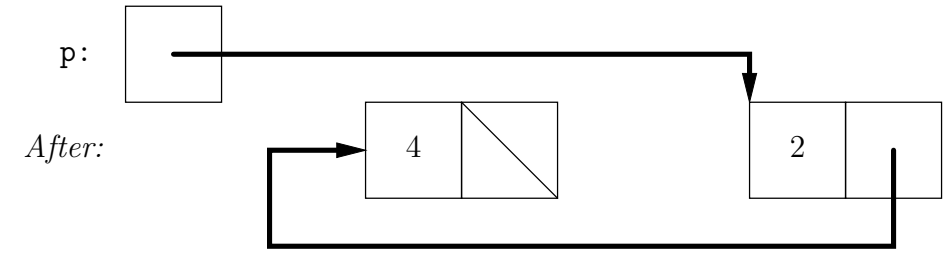
```
IntList a = IntList.list(1, 2, 3);
IntList b = new IntList(4, a.tail);
b.tail.head = 5;
IntList c = b.tail.tail;
a.tail.tail = null;
c.tail = new IntList(5, b);
int d = a.tail.head;
d = 2;
```



3. [2 points] The *Before* diagram below shows a state of some `IntList` objects and one local variable. In the spaces provided, write the Java statements that transform it into the *After* diagram, subject to the following restrictions:

- Do not modify the `.head` variables in any `IntList` object.
- Do not introduce any new variables.






```
p.tail.tail = p;
p = p.tail;
p.tail.tail = null;
```

4. [2 points] The following problems all involve three distinct classes, `A`, `B`, and `C`.
In the following solutions, let $\prec$ mean "is a subtype of."

a. Suppose that the following code compiles and runs without any exceptions. What can you say about the relationship between the classes `A`, `B`, and `C`?

```
A x1 = new C();
B x2 = x1;
B x3 = (B) x1;
B x4 = (C) x1;
```

$C \prec A \prec B$.

b. Now suppose the code in part (a) fails to compile, but if we remove the second line it compiles and runs without any exceptions. What can you say about the relationship between the classes `A`, `B`, and `C`?

$C \prec B \prec A$.

c. Now suppose that the code in (a) fails to compile, and when we remove the second and fourth lines it compiles, but causes a `ClassCastException` at runtime. What can you say about the relationship between the classes `A`, `B`, and `C`?

$C \prec A$, $B \prec A$, but $C \not\prec B$.

5. [2 points] Define the *k-partial sums* of a list L for an integer $k > 0$ to be the list of length k whose i^{th} element is the sum of the elements of L whose positions in L are of the form $i + nk$ for some nonnegative integer n. If L has length less than k then the extra partial sums of the resulting list are just 0. For instance, if L is `[1, 2, 3, 4, 5, 6, 7]` the 3-partial sums of L are `[12, 7, 9]` (because $1+4+7 = 12$, $2+5 = 7$, and $3+6 = 9$) and if L is `[1, 2, 3]` then the 5-partial sums are `[1, 2, 3, 0, 0]`.

The following functions compute partial sums of `L` by taking each element of `L` in turn and adding it to the appropriate element of the result list (rather than computing each element of the result list completely in turn). Fill in the missing lines so that `partialSums(L, k)` returns an `IntList` that represents the `k`-partial sums of `L`. Do not add any semicolons.

```
/** Returns an IntList representing the K-partial sums of L. Assumes K>0. */
public static IntList partialSums(IntList L, int k) {
    IntList last = circularList(k);
    IntList cur = last.tail;
    while (L != null) {
        cur.head += L.head;
        cur = cur.tail;
        L = L.tail;
    }
    IntList rtn = last.tail;
    last.tail = null;
    return rtn;
}
/** Returns an IntList of length K filled with zeroes where the last node
  * links back to the first node (i.e. the list returned is circular). */
public static IntList circularList(int k) {
    IntList last = new IntList();
    IntList first = last;
    k -= 1;
    while (k > 0) {
        first = new IntList(0, first);
        k -= 1;
    }
    last.tail = first;
    return first;
}
```

6. [2 points] `Sorting` (below) is a utility class whose exported method merges multiple sorted lists into one sorted list. Fill in the blanks to return the correct result.

```
import java.util.ArrayList;
public class Sorting {
    /* Assuming each IntList in LISTS is sorted in ascending order, return
     * a List of all their values sorted in ascending order. Destructive:
     * May modify LISTS. For example, if LISTS contains IntLists
     * containing [1, 5], [3], and [2, 4], the returned List should contain
     * [1, 2, 3, 4, 5].  */
    public static List<Integer> mergeSortedLists(IntList[] lists) {
        ArrayList<Integer> toReturn = new ArrayList<>();
        int smallestElemIndex = indexOfListWithSmallestElement(lists);
        while (smallestElemIndex != -1) {
            toReturn.add(lists[smallestElemIndex].head);
            lists[smallestElemIndex] = lists[smallestElemIndex].tail;
            smallestElemIndex = indexOfListWithSmallestElement(lists);
        }
        return toReturn;
    }

    /* Return the index in LISTS of a non-empty list whose head is smallest.
     * Returns -1 if all elements of LISTS are empty. */
    private static int indexOfListWithSmallestElement(IntList[] lists) {
        int index;
        index = -1;
        for (int i = 0; i < lists.length; i += 1) {
            IntList L = lists[i];
            if (L != null && (index == -1 || L.head < lists[index].head))
                index = i;


        }
        return index;
    }
}
```

7. [1 point] What common thing (having nothing to do with animals) do the words *apenstaartje* (Dutch for "monkey's tail"), *snabel* (Danish for "elephant's trunk"), *kissanhnta* (Finnish for "cat's tail"), *klammeraffe* (German for "spider monkey"), *παπάκι* (Greek for "little duck"), *kukac* (Hungarian for "worm"), *grisehalehka* (Norwegian for "pig's tail"), and *sobachka* (Russian for "little dog") all refer to?

The "commercial at sign" (@).

8. [1 point] Assume that `a` and `b` are initially non-negative integers less than 8. What is the value of `y` after the following code executes?

```
int x, y, z;
x = a; y = b;
z = x ^ y;  x = (x & y) << 1; y = z;
z = x ^ y;  x = (x & y) << 1; y = z;
z = x ^ y;  x = (x & y) << 1; y = z;
z = x ^ y;  x = (x & y) << 1; y = z;
```

The xor (`^`) operation performs bitwise addition without (binary) carries, while the and (`&`) operation computes carries, ignoring carries from previous columns. So the operation adds `y` first to `a` and subsequently to the carries from the previous operation (appropriately shifted over 1 bit position). Doing this four times allows carries to propagate from the units column all the way left. Thus the final value of `y` is `a+b`.

9. [1 point] A certain type of bracketed list consists of words composed of one or more lower-case letters alternating with unsigned decimal numerals and separated by commas, as in these examples:

```
[]
[wolf, 12, badger, 11]
[dog,10, cat]
```

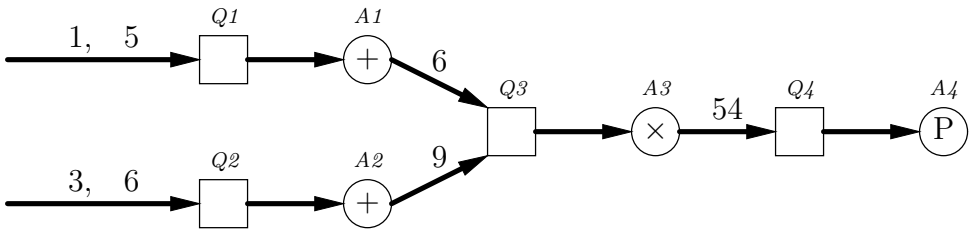
As illustrated, commas may be followed (but not preceded) by blanks, and the list may have odd length (ending in a word with no following numeral.) There are no spaces around the '`[]`' braces. Write a Java `Pattern` that matches such lists.

```
\[(([a-z]+, *\d+, *)*[a-z]+(, *[0-9]+)?)?\]
```

Or in Java String form:

```
\\[(([a-z]+, *\\d+, *)*[a-z]+(, *[0-9]+)?)?\\]
```

10. [3 points] A simple computational network is composed of *value queues* (represented by squares in the example below) and *computation nodes* (circles in the example below). Both value queues and computation nodes have inputs and outputs. Integer values come into each value queue, which accumulates them until the computation node on its output indicates there are enough values for its computation. At that point, the value queue sends its accumulated values to its computation node, which computes a value that is sent to another value queue. In the example below, '+' nodes wait for two inputs and compute their sum; '×' nodes wait for two inputs and compute their product; and 'P' nodes wait for one input and print it (producing nothing):



We'll represent value queues with the class `ValueQueue` and describe computations with an interface called `Computation`. Here is `ValueQueue`:

```
import java.util.ArrayList;

public class ValueQueue {
    public void attach(Computation sink) { _sink = sink; }

    public void accept(Integer value) {
        _queue.add(value);
        if (_sink.enabled(_queue.size())) {
            _sink.consume(_queue);  _queue.clear();
        }
    }

    private Computation _sink;
    private ArrayList<Integer> _queue = new ArrayList<>();
}
```

The following code sets up the network and inputs pictured in the diagram.

```
ValueQueue Q1 = new ValueQueue(), Q2 = new ValueQueue(),
           Q3 = new ValueQueue(), Q4 = new ValueQueue();
Computation A1 = new Adder(Q3), A2 = new Adder(Q3),
            A3 = new Multiplier(Q4), A4 = new Printer();
Q1.attach(A1); Q2.attach(A2); Q3.attach(A3); Q4.attach(A4);
Q1.accept(1); Q2.accept(3); Q2.accept(6); Q1.accept(5);
```

Working backwards from the code, fill in suitable definitions for the `Computation` interface and the `Adder` class.

```
public interface Computation {
     boolean enabled(int size);
     void consume(List<Integer> values);
}
public class Adder implements Computation {

    public Adder(ValueQueue out) {
        _out = out;
    }

    public boolean enabled(int size) {
        return size == 2;
    }

    public void consume(List<Integer> values) {
        _out.accept(values.remove(0) + values.remove(0));
    }

    private ValueQueue _out;
}
```