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

CS61B **P. N. Hilfinger**
Fall 2011

Test #3 (with corrections)

READ THIS PAGE FIRST. *Please do not discuss this exam with people who haven't taken it.* Your exam should contain 6 problems on 11 pages. Officially, it is worth 20 points (out of a total of 200).

This is an open-book test. You have fifty minutes to complete it. You may consult any books, notes, or other inanimate 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 lab section 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: __________

1. __________/6

2. __________/5 Login of person to your Left: __________ Right: __________

3. __________/2 Discussion section number or time: __________

4. __________/ Discussion TA: __________

5. __________/2 Lab section number or time: __________

6. __________/5 Lab TA: __________

TOT __________/20

1. [6 points] Provide simple and tight asymptotic bounds for the running times of the following methods (including any methods they call) as a function of the value of N, the length of `A`. If possible, give a single Θ bound that always describes the running time, regardless of the input. Otherwise give an upper and a lower bound. Give brief justifications of your answers.

a. [1 point]

```
static List<Integer> reverse(List<Integer> A) {
    ArrayList<Integer> R = new ArrayList<Integer>();
    for (Integer x : A) {
        R.add(0, x);
    }
    return R;
}
```

b. [1 point]

```
static List<Integer> reverse(List<Integer> A) {
    ArrayList<Integer> R = new ArrayList<Integer>();
    for (int i = A.length - 1; i >= 0; i -= 1) {
        R.add(A.get(i));
    }
    return R;
}
```

c. [1 point]

```
/** True iff X is among elements #L through #U of A,
 *  assuming A is sorted. */
static boolean contains(ArrayList<Integer> A, int L, int U, int x) {
    if (L > U)
         return false;
    int m = (L + U) / 2;
    if (A.get(m) < x)
         return contains(A, m + 1, U, x);
    else if (A.get(m) > x)
         return contains(A, L, m - 1, x);
    else
         return true;
}
```

Initial call: `contains(someList, 0, someList.size()-1, someValue).`

d. [1 point]

```
/** True iff X is among elements #L through #U of A,
 *  assuming A is sorted. */
static boolean contains(LinkedList<Integer> A, int L, int U, int x) {
    if (L > U)
         return false;
    int m = (L + U) / 2;
    if (A.get(m) < x)
         return contains(A, m + 1, U, x);
    else if (A.get(m) > x)
         return contains(A, L, m - 1, x);
    else
         return true;
}
```

Initial call: `contains(someList, 0, someList.size()-1, someValue).`

e. [2 [points]

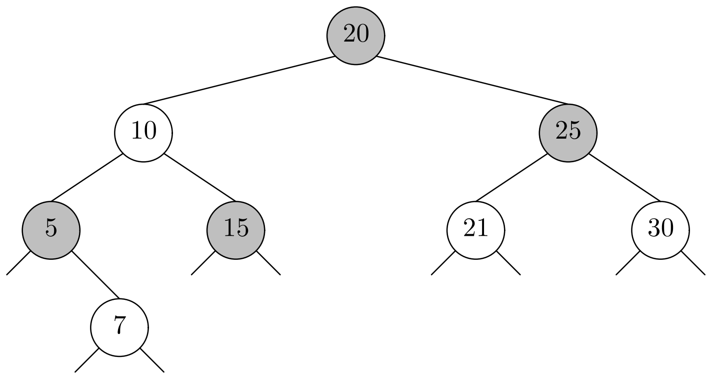
```
static void munge(int[] A, Random R) {
   for (int i = 1; i < A.length; i += 1) {
       if (A[i-1] > A[i])
           return;
   }

   for (int i = 0; i < 8; i += 1) {
       k = R.nextInt(A.length); /* 0 <= k < A.length */
       int t = A[k]; A[k] = A[0]; A[0] = t;
   }

   insertionSort(A);
}
```

2. [5 points]

a. Convert the following red-black tree into the corresponding 2-4 tree (shaded nodes are black).



b. Show a (2,4) tree of minimum height that contains 15 keys (we're not interested in specific key values).

c. Show a (2,4) tree of maximum height that contains 15 keys.

d. Show a minimum-height red-black tree containing 15 keys.

e. Show a maximum-height red-black tree containing 15 keys.

3. [2 points] A certain proposed implementation of a set type uses an open-address hash table implementation (with an ArrayList to hold the values), and does not allow null values in the set. It has the following implementation for its `remove` method:

```
/** Remove OBJ != null from the table.  Returns true if OBJ was in
 *  the table, and otherwise false. */
public boolean remove(Object obj) {
    if (obj == null)
        throw new IllegalArgumentException("null value in set");
    int hash = obj.hashCode() % _values.size();
    if (hash < 0)
        hash = -hash;
    int i;
    i = hash;
    do {
        if (_values.get(i) == null)
            break;
        if (obj.equals(_values.get(i))) {
            _values.put(i, null);
            return true;
        }
        i = (i + 1) % _values.size();
    } while (i != hash);
    return false;
}
```

A simple unit test for this method seems to show that it works. But although the hash-table implementation throws no exceptions for valid input (in this or other methods), there seems to be a problem, which you suspect might be due to `remove`. What is wrong, and what test would reveal it?

4. [1 point] What is the source of the following lines?

La fleur que tu m'avais jetée,
Dans ma prison m'était restée,
Flétrie et sèche, cette fleur
Gardait toujours sa douce odeur. . . .

5. [2 points] The following fragment of a program compiles, but has what are almost certainly two bugs. Indicate clearly how to fix them.

```
public class Distributor {

    /** A new Distributor with N outlets. */
    public Distributor(int N) {
        ArrayList<Outlet> _outlets = new ArrayList<Outlet>();
        for (int i = 0; i < N; i += 1) {
            _outlets.add(new Outlet(i));
        }
    }

    /** Current number of outlets. */
    public int size() {
        return _outlets.size();
    }

    /** Turn off outlet #K. */
    public void close(int k) {
        _outlets.get(k).close();
    }

    /** Remove all closed outlets. */
    public void demolish() {
        for (Iterator<Outlet> it = _outlets.iterator(); it.hasNext(); ) {
            Outlet out = it.next();
            if (out.isClosed())
                _outlets.remove(out);
        }
    }

    private ArrayList<Outlet> _outlets;
}
```

6. [5 points] *Lazy* data structures resemble their non-lazy (or *strict*) counterparts except that they compute the results of accessors (e.g., `.get` on a list) only when they are needed the first time, saving the result. Any subsequent calls to an accessor with the same arguments then simply returns the previously computed value. Thus, the data structure grows only in response to need. In CS61A, streams had this property.

The type `LazyTree`, below, is one such structure. A `LazyTree` is a binary tree node that has operations `getLabel`, `getLeft`, and `getRight`, which provide its label and its left and right children, as for an ordinary binary tree. To create a `LazyTree` node, one supplies a `TreeFormer` argument, which returns the node label and subtrees upon demand. However, a given `TreeFormer` is to be consulted at most three times during the lifetime of its tree node: once for the label, once for the left child, and once for the right. If the program asks a `LazyTree` for the label or a subtree from a node more than once, the second and subsequent requests return the previously computed value without reconsulting the `TreeFormer`.

```
/** Supplies the label (of generic type T) and tree formers for the
 *  left and right children of a tree node. */
public abstract class TreeFormer<T> {
    public abstract T label();
    public abstract LazyTree<T> left();
    public abstract LazyTree<T> right();
};

/** A binary tree whose contents are constructed as needed
 *  from TreeFormers.  The empty tree is represented by null. */
public class LazyTree<T> {
     public LazyTree(TreeFormer<T> former) {
         _former = former;
     }

     T getLabel() {
         if (!_haveLabel) { _haveLabel = true; _label = _former.label(); }
         return _label;
     }

     T getLeft() {
         if (!haveLeft) { _haveLeft = true; _left = _former.left(); }
         return _left;
     }

     T getRight() {
         if (!_haveRight) { _haveRight = true; _right = _former.right(); }
         return _right;
     }

     private TreeFormer<T> _former;
     private boolean _haveLeft, _haveRight, _haveLabel;
     private T _label;
     private LazyTree<T> _left, _right;
}
```

a. [3 points] Fill in the function below to obey its comment. Feel free to add any additional classes you need.

```
/** Returns a lazy tree that contains elements L through U of a sorted
 *  array A as a balanced binary search tree. For example, if A is
 *  {1, 2, 3, 4, 5, 6, 7, 8}, the tree resulting from toBST(A, 0, 7) is
 *              4
 *            /   \
 *          2       6
 *         / \     / \
 *        1   3   5   7
 *                     \
 *                      8
 *  As illustrated here, when the subtrees are uneven, the extra element goes
 *  on the right.
 */
LazyTree<String> toBST(String[] A, int L, int U) {
    // FILL IN








}
```

b. [2 points] Suppose I wanted to test another implementation of `LazyTree` by checking the properties that its nodes consult their `TreeFormer`s at most once to get their labels and children and do not consult their `TreeFormer`s for any labels or children that the client does not request (by calls to the `getLeft`, `getRight`, and `getLabel` methods). Describe in sufficient detail JUnit tests to check these properties.