CS61BL Summer 2013 Midterm 2

Sample Solutions + Common Mistakes

Question 0:
Each of the following cost you .5 on this problem: you earned some credit on a problem and did not put your five-digit on the page, you did not adequately identify your lab section, or you failed to put the names of your neighbors on the exam. The reason for this apparent harshness is that exams can get misplaced or come unstapled, and we want to make sure that every page is identifiable. We also need to know where you will expect to get your exam returned. Finally, we occasionally need to verify where students were sitting in the classroom while the exam was being administered.

Question 1A:
Sample Solution (Version A):

```
public class TrackedQueue extends Queue {
    private int myMaxSize;

    // technically, this constructor is unnecessary
    // but you didn’t lose points if you implemented it correctly
    public TrackedQueue() {
        super();
        myMaxSize = 0;
    }

    public void enqueue(Object obj) {
        super.enqueue(obj);
        if (size() > myMaxSize) {
            myMaxSize = size();
        }
    }

    public int maxSizeSoFar() {
        return myMaxSize;
    }
}
```

Common Mistakes:
There were two main issues that students frequently encountered. The first was not realizing that inheritance would do most of the work for you. That is, since you extend from the Stack class, you don’t need to define a peek(), pop(), or size() method - the only thing you need to override is the push() method. For the other test version, you extend from a Queue, which means you do not need to override the peek(), dequeue() or size() methods, only the enqueue() method.

The other main mistake was not realizing that you have to compare myMaxSize (or whatever int variable you made) to super.size() in the push() method. You only want to increment myMaxSize if (myMaxSize<super.size()), because otherwise you are keeping track of *the total number of calls to push/enqueue,* instead of the largest count that we’ve seen *so far*. This distinction caused a number of students to lose points.

Question 1B:
Sample Solution (Version A):
(For version B replace “Queue” with “Stack” and “enqueue” with “push”)

```
Queue myQ = new TrackedQueue();
System.out.println(myQ.maxSizeSoFar());
// Compile-time error: can’t find maxSizeSoFar in Queue

TrackedQueue myQ = new Queue();
System.out.println(myQ.maxSizeSoFar());
// Compile-time error: static type is a subclass of the dynamic type

Queue myQ = new TrackedQueue();
myQ.enqueue(“MIKE”);
// Both TrackedQueue.enqueue() and Queue.enqueue() are called
```

Common Mistakes:
For the first part, the most common mistake was getting it completely wrong by saying that there’s no error. The error is in the second line -- remember that Java checks the static class during compile-time, and the dynamic class during run-time. Since myQ is of static type Queue and Queues don’t have maxSizeSoFar(), there’s a compile-time error.

For the second part, the most common mistake was being confused about run-time error vs compile time error. Or rather, they detected the run-time error in the second line if the first line was allowed to compile. However, the first line doesn’t compile because saying that a Queue “is-a” TrackedQueue is false—not all Queues are TrackedQueues. Java will complain about “incompatible types”.

In the third part, most people didn’t realize that both TrackedQueue’s enqueue() and Queue’s enqueue() are called. First, TrackedQueue’s enqueue() method is called, which, if you implemented 1a correctly makes a call to super.enqueue().

Question 2:
Sample Solution (Version A):
A, Enqueue: linear time; need to free up a slot by shifting all elements by 1 slot.
A, Dequeue: constant time; simply call ArrayList.remove(N-1).

Sample Solution (Version B):
A, Enqueue: constant time; simply call ArrayList.add() at the back of the queue
A, Dequeue: linear time; remove the first element, then shift all other elements by 1 slot.

Sample Solution (Both Versions):
B, Enqueue: constant time; adding to the front of a linked list takes constant time
B, Dequeue: linear time; need to traverse the entire list to access the next-to-last node

Common Mistakes:
The most common mistake was B, dequeue - not realizing that since you don't have previous pointers, you can't access the next-to-last node without traversing the entire list in linear time.

Another common mistake was not realizing that removing or adding something to the beginning of an arrayList (at index 0) requires a shift of all the items in the ArrayList, which is linear. However, adding to the end or removing the end of an ArrayList is in constant time because you can access ArrayLists by index (ArrayList is implemented as an Object array). To access the last item we can do something along the lines of arrayList.get(size() - 1). It is NOT necessary to traverse the whole arraylist to find the last index.

The last common mistake was not reading the implementations correctly, particularly for B-enqueue... Many people were trying to enqueue the new node to the back of the list even though according to the implementation the back of the queue is the front of the list (so you want to add to the front of the list).

Question 3A: (3B on Exam B)

Sample Solution:

Changing the hashCode() function causes every value to have the same hash code (10). This means they all hash to the same bucket, and a very large chain will form. This will cause adding new keys, or looking up values, to be very slow, as we now need to essentially look through a LinkedList. This loses any benefits the HashMap would've given us.

Common Mistakes:

Most points lost here were for not being specific in answers. For example, stating that there would be many collisions (or many items in the same bucket), but not explaining that it causes the HashMap to act like a LinkedList or run very slowly.

Question 3B: (3A on Exam B)

Sample Solution:

When a key is added to the HashMap, HashMap first computes the hashCode of the key to determine its bucket. Then, within that bucket, it uses .equals() to determine if the new key is already in the HashMap... and overwrites it if there is. This means in each bucket, there will be at most one key/value pair. contains() and get() will also act oddly, returning true (or a value) so long as there's something in the correct bucket, regardless of whether or not the keys are equal.

Common Mistakes:

To get full credit, you must have explicitly mentioned the erroneous behavior of adding two different keys that hashed to the same bucket, even if you correctly identified the behavior of contains() and/or get().

The most common mistake was assuming that objects were added to the HashMap normally, and that calls to get() would return the first value from the appropriate bucket. This has the same effect as overwriting the keys, but technically is not what's happening behind the scenes.

Another common mistake was assuming that hashCode() had changed. In most cases, there is an invariant which says your HashMap will not function correctly unless equal objects have the same hash code. In this case, we were not working with hashCode(), all we did was change equals(). Changing one does not automatically affect the other.

A less common mistake was assuming that only one object could be stored in the HashMap at a time. This is incorrect, as each bucket can have a key/value pair. This misconception either stems from assuming we changed hashCode(), or assuming HashMap checks ALL buckets for equal keys before adding new values.

Question 4A (Both Versions):

```
/*     Sample Solution 1     */
public void isOK() throws IllegalStateException {
    Iterator<Amoeba> iter = allAmoebas();
    HashSet<String> names = new HashSet<String>();

    while (iter.hasNext()) {
        Amoeba a = iter.next();
        if (names.contains(a.myName)) {
            throw new IllegalStateException ("duplicate name = " + a.myName);
        }
        names.add(a.myName);

        // or if (!names.add(a.myName)) { throw ... }
    }
}
```

```
/*     Sample Solution 2     */
HashSet<String> allNames = new HashSet<String> ( );

public void isOK ( ) throws IllegalStateException {
    helper (myRoot);
}

private void helper (Amoeba current) {
    if (current == null) {
        return;
    }

    if (allNames.contains (current.myName)) {
        throw new IllegalStateException ("duplicate name: " + current.myName);
    }
    allNames.add (current.myName);

    Iterator<Amoeba> iter = current.myChildren.iterator ( );
    while (iter.hasNext ( )) {
        helper (iter.next ( ));
    }
}
```

```
/*     Sample Solution 3     */
/*     Note: This solution would earn only 2.5 out of 5, since it runs
       in time proportional to N-squared.     */
public void isOK ( ) {
    AmoebaIterator iter1 = allAmoebas ( );
    while (iter1.hasNext ( )) {
        Amoeba a1 = iter1.next ( );

        AmoebaIterator iter2 = allAmoebas ( );
        while (iter2.hasNext ( )) {
            Amoeba a2 = iter2.next ( );
            if (a1 != a2) {
                if (a1.myName.equals (a2.myName)) {
                    throw new IllegalStateException ("duplicate name :" +
                        a1.myName);
                }
            }
        }
    }
}
```

Common Mistakes:
One of the most common errors was attempting to hash Amoeba objects, which doesn't work since we have not overridden Amoeba's hashCode. Instead, you should hash their names directly, because String does have an overridden hashCode.

Some students still had troubles instantiating an iterator or differentiating between the AmoebaFamily and AmoebaClasses. The method you have to write is in AmoebaFamily, and so you can call the method allAmoebas() which returns an iterator over all Amoebas in the tree, which greatly simplifies the problem. You can also just construct a new AmoebaIterator() since AmoebaIterator is a public class inside AmoebaFamily. You cannot create an iterator over all Amoebas in the tree from a single Amoeba; however, you can create an iterator only over its immediate children by taking advantage of ArrayList's iterator, which is what you needed to do if you attempted a recursive solution.

For students that tried recursion instead of iteration, a common mistake was to not pass your HashSet of seen names through calls. If you don't pass it in then you won't be able to check against the names you've seen in other calls. An alternative solution to this problem is to make the HashSet an instance variable of AmoebaFamily, so all your recursive calls can access it (provided the recursive calls are not static).

The heaviest deductions were for slow solutions (using an ArrayList instead of a HashSet, or doing nested iterations and not saving the names at all). These slow solutions run in O(n^2) time rather than O(n), which was unacceptable.

Some students also didn’t realize that the instructions to use a “hash table” meant they could use a HashSet or a HashMap. “hash table” is just a generic term that can refer to either. HashSet suited this problem better, but some students made a HashMap work and weren’t deducted for it.

Question 4B:
Version A:
Correct Answer: Wilma, Hilary, Homer
Partial Credit (Reversed Depth-First, or pushed in wrong order): Homer, Hilary, Wilma

Version B:
Correct Answer: Marge, Bart, you
Partial Credit (Reversed Depth-First, or pushed in wrong order): You, Bart, Marge

Common Mistakes:
Some answers saw that we were using a stack, and assuming the order based on that... Amoebas were pushed onto the stack left to right, which causes the traversal to go down the right side of the tree first.

Some students confused breadth first with other traversals, or had no clear pattern to how they traversed the tree.

Putting down the same Amoeba multiple times (or naming an Amoeba we did not ask for) were considered completely wrong.

Question 5:

```
/* Sample solution */
public boolean sameComputation (BinaryTree expr) {
      return helper (myRoot, expr.myRoot);
}

private boolean helper(TreeNode root1, TreeNode root2) {
      if (root1 == null) {
            return root2 == null;
      }
      if (root1.myOper == root2.myOper) {
            if (root1.myOper == '+') || root1.myOper == '*') {
                  return (helper (root1.myLeft, root2.myLeft)
                          && helper (root1.myRight, root2.myRight)
                      || (helper (root1.myLeft, root2.myRight)
                          && helper(root1.myRight, root2.myLeft));
            } else { // else not necessary
                  return (helper (root1.myLeft, root2.myLeft)
                       && helper (root1.myRight, root2.myRight));
            }
      } else {
            return false;
      }
}
```

Common Mistakes:
One common mistake was to avoid using helper methods. Unless you knew how to compare the trees iteratively using stacks, it simplified the problem to go straight into using a helper (with two TreeNode parameters) to solve it recursively. There were also many occurrences of confusion between TreeNodes and BinaryTrees (earning a deduction of at least 2).

In terms of the recursive solution, a common mistake was to not carefully think about what base cases were needed. Based on how students carved out their answers, not all the base cases were the same for all solutions. Solutions flagged as "bad base cases" in the rubric included missing base cases along with comparing TreeNode references (e.g. myRoot) rather than variables and operators (e.g. myItem).

A few students didn't properly handle the cases for + and * carefully, which would result in all cases being meshed together. And surprisingly many solutions failed to combine the results of the recursive calls, for instance, as follows:

```
      helper (root1.myLeft, root2.myLeft);
      helper (root1.myRight, root2.myRight);
```

Question 6:

Sample Solution (version A):

```
p.myRest = soFar;
soFar = p;
p = temp;
```

Sample Solution (version B):

```
soFar = p;
p = p.myRest; // or p = soFar.myRest;
soFar.myRest = temp;
```

Common Mistakes:

The provided for-loop hinted that students should have used iteration instead of recursion (let n be the number of nodes in the List; a recursive call to reverseHelper would have resulted in anywhere from n^2 to n^3 to infinity calls to reverseHelper). Students who used recursion or created additional ListNode objects received no credit. Some students also attempted to change nodes' myFirst values and/or assign a ListNode to another node's myFirst, which resulted in incorrect solutions.