CS61BL Summer 2013 Final Exam

Sample Solutions + Common Mistakes

Question 0A:

Students lost 0.5 points (capped at 1) for each of the following:

- Not circling login
- Not circling section TA / time
- Not writing the chosen random number on all pages of the exam

Strangely enough there were far more students who forgot to circle their login or section TA / time than students who did not write their random number on all of the exam pages.

Question 0B:

Students received a point for completing the survey from August 12 and 13 labs before 2pm PDT on August 18.

Question 1:

Sample Solution:

If the value is now greater than its parent, bubble it up. If the value is now less than one of its children, bubble it down.

Rubric

You lost 1 point for each of the following: missing parent comparison, missing comparison with children, and missing bubble direction.

Common Mistakes:

There weren't really common mistakes on this question, for the most part it was straightforward.

The most common mistake was simply not reading/answering the question. This included trying to remove/insert nodes (rather than just changing the value) or assuming nothing would change because we didn't give you code for the method. Otherwise, a majority of students did not have trouble on this question.

Question 2:

Sample Solution:

No such sequence exists. After the third insertion, the AVL tree is guaranteed to height-balance itself via a single rotation if it is not already balanced. A height-balanced 3-node AVL tree has a height balance factor of 0 at every node. Adding the 4th node will cause one of the leaves to have a height balance factor of 1 between its left and right branches. Since no node has a height balance factor of 2 (or more) after 4 node insertions, adding the 4th node will never cause a rotation.

Rubric

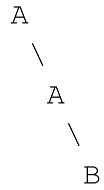
-0.5	Correct answer but slightly incorrect justification
-1	Correct answer but doesn't adequately explain why tree will be balanced
-2	Correct answer but incorrect justification

Common Mistakes:

The most common mistake was not providing enough justification for why the 4th insertion will never cause a rotation. A few students confused 2-3 trees with AVL trees or rotated a balanced 4-node AVL tree at the root node (which effectively did nothing for the height balance of the tree). Others did not rotate the AVL tree after the 3rd insertion when they should have, then rotated the tree after the 4th insertion, incorrectly claiming that this was a valid AVL tree rotation. Note that only saying "the AVL tree will be balanced during the 4th insertion" with no explanation of why you know it will be balanced lost you a point.

Question 3A:

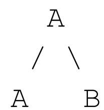
Sample Solution:



- Because the inorder and preorder lists are the same, `pos` will always equal 0, and no node will ever have a left branch (the second `if` body of the helper will never be reached).
- The recursive calls were: `helper([A, B], [A, B])` followed by `helper([B], [B])`

Common Mistakes:

Many students provided the following incorrect tree:



Some students falsely claimed that the tree they provided was the only tree that corresponded to a preorder and inorder of [A, A, B]. By far the most common mistake was when a student simply drew a tree without providing evidence that the student had looked at the code.

Question 3B:

Sample Solution:

- Before line 1: `(preorder == null || inorder == null)`, "Cannot process null lists"
- Before line 8 (also accepted 6 or 1): `(preorder.size() != inorder.size())`, "Preorder and inorder lists don't have the same number of elements"
- Before line 9: `(if(!rootItem.equals(inorder.get(0))))`, "Preorder and inorder lists don't have the same elements."
- Before line 12: `(pos == -1)`, "Root of preorder tree is not found in inorder"

The last item could have been covered by a call to `.contains()`.

Instead of doing separate checks at lines 12 and 9, it's also acceptable to check:

`inorder.contains(myRoot)`, after line 6 or 7.

Rubric
You received 2.5 for each correct error/fix from among the above list, and 1 for each flawed error/fix (mostly involving wrong fix location). You received credit for at most one of a set of overlapping checks; see below.

Common Mistakes:
The most common mistake was neglecting to check that the arguments (inorder and preorder, as Lists) were not null. You *must* do this first, otherwise calls to get() or size() will crash the code when an argument is null.

Another common mistake was not checking that the elements matched in the base case, that is, checking if preorder and inorder contained the same Object when creating leaves.

Many students tried to check for things we didn't care about. An example is trying to avoid having null elements in the tree. There's nothing that says a user cannot store "null" in a tree, so we do not need to check for this. Other extraneous/unwanted checks included trying to sort preorder/inorder (nothing in the problem mentions that tree elements are comparable), or attempts to see if the lists contained the same items (this usually fails when the lists have duplicate elements).

One more kind of error was the inclusion of multiple comparisons of the sizes of preorder and inorder (possibly also involving pos or checks for empty lists). You were only given credit for one such comparison; others are redundant. If the sizes are equal in the constructor, they will also be equal through the helper since both the arguments to each recursive call either have length pos or length size–pos+1. Multiple checks for null arguments also earned credit for only one.

Question 4A:

Sample Solution:

Here are the most common solutions we saw to this problem, and the ones we used as a general outline for grading. Method 5 was the correct algorithm, although there were some variations of it that did get full credit.

** **Method 1** **

(no preprocessing of allElements)

1) Look up and store the position of each toSort element in allElements via linear search.
2) Sort according to those positions

Running time = T * A + T log T = O(T * A)

** **Method 2** **

(no preprocessing of allElements)

1) Allocate an array addrs of A elements.
2) For each element x of toSort, find its position p in allElements and set addrs[p] to x.
3) Then compress addrs

Running time = O(T * A + A) = O(T * A)

** **Method 3** **

(no preprocessing of allElements)

1) Hash the toSort list in a HashSet
2) Do gets from allElements in order, checking each to see if it's in the toSort hash table

Running time = O(T + A)

** **Method 4** **

(preprocessing step)

1) Associate each allElements element with its position in a key-value pair
2) Sort pairs alphabeticaly by allElements entry

(processing step)

1) Look up each toSort element's position in that list
2) Sort by that value

Running time = T log A + T log T = O(T log A)

** **Method 5** **

(preprocessing step)

1) Use a HashMap to hash each element of allElements to associate it with its position

(processing step)

1) Sort toSort using the associated allElements position as the comparison value

Running time = O(T log T)

****

Here was the point distribution for each of the methods:
2/8 for Method 1 (or similar implementations)
2/8 for Method 2 (or similar implementations)
4/8 for Method 3 (or similar implementations)
6/8 for Method 4 (or similar implementations)
8/8 for Method 5 (or similar implementations)

There were some exceptions to this on a case-by-case basis, but in general we wanted to see that you hashed (String, Integer) values into a HashMap (corresponding to the strings in allElements and their indices) and then used these indices to correctly sort toSort. Alternative full-credit solutions involved using radix sort, or hashing every possible subset/permutation of allElements and then accessing the correct ordering in constant time in part B.

Common Mistakes:
Common mistakes were solutions where you hashed the strings of allElements into a hashmap with the string as key and index as value, but then used an inferior sorting method in part B (ie, using insertion sort instead of using quicksort) in order to correctly order toSort.

Many people had algorithms that accessed or modified toSort for part A, which was not allowed - we wanted you to *preprocess allElements*, meaning you could only use the allElements data for this step. The fact that using toSort was not allowed was written on the “Errata” screen at the start of the final. Because of this, if you accessed toSort during part A we just treated it as though you were doing all of the work in the post-processing phase, which of course means your algorithm is not optimal, as we wanted the runtime of part B to be as fast as possible.

Question 4B:
This question was graded with part A, so please look at the above comments for this solution.

Question 4C:
The correct answer for this question was O(T log T). That is, you had to run quicksort (or heapsort or mergesort) on the toSort list using the values you stored in the hashmap from part A, which takes T log T time. Technically heapsort in this case takes T + T log T time, but in big-Oh notation this is simply O(T log T).

Points for this question depended on the algorithm that you used for part A and part B. If you used Methods 1 or 2, or a similar implementation, then **correctly** analyzing your implementation would earn you 2 points, and 0 or 1 otherwise. If you used methods 3, 4 or 5 and **correctly** analyzed the runtime of Part B, then you earned 4 points. If you used methods 3, 4 or 5 and had a flawed or severely flawed analysis of your runtime, then you earned 2 or 1 points, respectively. Please note that there were a variety of answers and many needed to be graded case-by-case, so if your grade seems to differ from the above rubric this is why.

Common Mistakes:

Most of the mistakes here came from people having incorrect implementations for part A and B. As most students recognized, storing all of the values in allElements into a HashMap<String, Integer> took O(A) time for part A. Then for part B, you need to sort the values in toSort by associating each element in toSort with the index value you get from using part A's hashmap. Once you have associated each element in toSort with it's allElements index value, you can use quicksort or mergesort or heapsort to sort the values in O(T log T) time.

Question 5:

Sample Solution:

```
public void processLeftMove(Tray current)
      // Make sure it's possible to move a block left.
      if(current.space.column() == 2){
            return;
      }

      // Construct the tray that represents moving a block left,
      // and check whether we've seen it already.
      Tray tray = new Tray(current);
      tray.previous = current;
      // Block moving left means space moves right.
      // space's position can be changed in place.
      tray.space
          = new Position(current.space.row(),
                         current.space.column()+1);
      tray.myBlocks[current.space.row()][current.space.column()]
            = tray.myBlocks[tray.space.row()][tray.space.column()];
      tray.myBlocks[tray.space.row()][tray.space.column()] = 0;
      if(savedTrays.contains(tray)){
            return;
      }
      // It's a tray we haven't seen before;
      // remember it and include it in the iteration.
      savedTrays.add(tray);
      nextTrays.put(tray);
}
```

Another approach was to work with Strings rather than the myBlocks and space instance variables, and then pass the relevant String to the appropriate Tray constructor.

Rubric

You received 2 points for correct inclusion of each of the following operations, with deductions for incorrect or overly vague steps.

- checking for a block at the edge of the tray;
- creating a new tray (and not thereafter modifying current);
- setting the new tray's previous link correctly;
- moving the block (updating the destination position in myBlocks);
- moving the block (replacing the original position with 0);
- updating the position of the space (tray.space);
- putting the tray into savedTrays if appropriate;
- putting the tray into the fringe (nextTrays) if appropriate.

Common Mistakes:
The hardest part about this problem was just remembering all the things you had to do: create a new tray, change two of the values in myBlocks, set previous and space's position, and add it to both savedTrays and nextTrays (if not already in savedTrays). For example, many many students overlooked the update of the previous link.

You must not modify the current tray in this method. The way the fringe works is that you store in it copies of different trays that you can select from when deciding where to search next. All this method was supposed to do was add to the fringe, so all you needed to do was create a new tray, modify that a little bit, and add it to the fringe.

A common mistake was not to realize there was no need to iterate over the 2D array to find the empty space because that information was contained in the space variable.

Another common mistake was to assume that 2D array myBlocks contained Block objects when in reality it was an integer array. This was a little tricky because they aren't stored the same as the space variable.

Yet another mistake was to assume that TrayIterator would provide the next move for the tray.

Many students lost points for giving too high-level a description. The problem statement specified essentially one pseudocode statement for each Java statement in processLeftMove. You had to be explicitly accessing or assigning to Tray instance variables (previous, space, myBlocks) to get any points for moving the block and updating space and previous. For example, saying that "swap the blank space with the block on its left" received at most partial credit for moving the block and no credit for updating the space.

Finally, there was no need to throw exceptions within this method or exit early. You have to consider the context that this method is in: all it is doing is adding next possible moves to the fringe. If there are no more possible left moves it doesn't mean you found an error or anything; that's just how the puzzle works.