(Unofficial) CSE 143 Final Test-Taking TA Tips – Summer 2018 **o(^‿^)o**

General Tips:

- Practice early so you don't cram the day before! A clear head is essential to doing well on a CSE 143 exam, as there will be a lot of logic involved.
- Practice-It, section handouts, and past final exams posted on the course website are perhaps the best resources for studying. Make sure to write your solutions on paper first to mimic the setting of the exam (give yourself 120 minutes for a printed practice exam, without notes). It's good to practice testing your code by hand (on paper) since you won't have Practice-It to tell you 20/20 on the exam.
- As you practice, write down your 2-3 most common mistakes for each category. This will **really help** as you'll prepare yourself to avoid what you'll most commonly make a mistake on!
- Which problems are you avoiding, cringing at the thought of returning to? Do those. You don't want to get to the test without feeling any more confident about them.
- It can be helpful to have go-to strategies to have when you're stuck. Having a go-to (even if it's not end-all be-all) can help you feel more comfortable and get you started in the right direction without wasting too much time.
    - One common strategy includes drawing additional example inputs and considering what the return/result should be (case analysis - try simple inputs and complex ones!). By understanding what the method does, you'll have a more intuitive grasp on the overall idea and may start seeing some more patterns.
    - Active reading with pens (circling, underlining, highlighting, etc.) can help you catch/remember important details, such as any important ordering, exceptions to throw, specified parameters and returns, data structures/methods not allowed, empty cases, etc.
- Give yourself space between statements to account for any last-minute fixes/special cases.
- Make sure to review the exams information page! There are a number of points to keep in mind when studying. The exam cheat sheet will also be useful to familiarize yourself with.
- Review your midterm - while there will be different problems, many topics carry-over into the exam (e.g. recursion, collections, working with ListNodes in linked list programming, etc.)
- Writing something is better than nothing. Most programming problems have some form of partial credit so even if your solution isn't complete, you may receive points for parts of your method that would also appear in a correct solution.
- Relax. Unlike the midterm, you don't have to go your absolute fastest to finish all the problems in time. You may even get a chance to check some of your answers. Use this time effectively, but also use it to keep a clearer mind.

Common in-the-moment mistakes:

- Forgetting proper interfaces (e.g. using `ArrayList` instead of the `List` interface)
- Mixing up size/length methods and whether to include parentheses :
    - `String word : word.length()` - *method*
    - `List<Integer> names: names.size()` - *method*
    - `int[] numbers : numbers.length` - *field*
- Forgetting to declare a variable before using it
- Forgetting to actually return (where relevant) what you've worked so hard to solve
- Reversing order of things (especially common with binary tree programming)
- Modifying data in binary tree or linked list problems. For these problems you're only allowed to modify a data structure by changing the next, left, or right fields in the node, as well as by setting the front/overallRoot fields.

Tips for Different Problem Types:

Binary Tree Traversal

- Use the sailboat method. Imagine each node is an island and the links between them as barriers you can't sail through. As you start moving down the left of the overall root node, sail around the tree, print nodes as you see them. In a preorder traversal, you print the node when you sail past the left side, inorder the bottom side, and postorder the right side.

Binary Search Tree

- Remember the alphabet! Some students find it helpful to write out the sequence for a quick visual reference.
- Double-check your work when you're done. This problem is easy points, and you don't want to lose any on a small mistake.

Binary Tree Programming

- Remember public/private pairs!
  - If you're traversing a tree, remember to start with your overall root, and keep of your current node as a parameter in the private method.
- Leave some space in your private method header in case you find you need extra data for each recursive call, such as levels, currentSum, etc. - sometimes it's easier to determine these after you draw/write your solution a bit and find you're missing the information
- Common questions for the easier binary tree programming question are traversals, calculating some information about the tree
  - Sometimes these ask to calculate different properties of a tree - parameters may be needed in the private method to figure out level/pathlength/etc.
- Common harder binary tree programming questions require changing a tree(s) in some way (x=change(x)), and/or build new ones
  - Remember, for x=change(x) problems, we want to take a node parameter, return a node, and store the return nodes.
  - Recall that to change a tree, we need either to i) change the overall root or ii) change one of the `.left` or `.right` fields on a node. This means you will probably have to call `this.root = method(this.root,` [other necessary parameters]`)` in the public method and some combination of `x=change(x)` to assign left/right node children in the private method.
- Make sure to review the difference between x=change(x) problems and basic traversal methods. A good tip is to review the section handouts and/or Practice-It problems involving binary tree programming, and start with identifying whether each problem is a traversal, construction, or x=change(x) problem.
- If you get stuck on a binary tree problem, try to consider smaller cases - what should happen for a single node with 0, 1, or 2 children? Start your solution focusing on this "neighborhood", and then check that the you've covered all of the cases correctly/have necessary parameters for the public/private method pair. Working "inside-out" this way can sometimes help get unstuck on binary tree problems.
- If you're really stuck because the problem seems structurally impossible, perhaps consider different types of traversals. Most binary tree problems for these tests are pre-order, but occasionally there will be a post-order problem as well. Be prepared for that, because it can save a lot of time to realize this immediately.
- Practice binary tree problems!

Collections Programming

- Make sure you're familiar with Map/Set/List methods on the exam reference sheet - if you forget the syntax of the return or parameters, use the sheet!
- Make sure to review how loops work with Iterators, how to add/remove with Iterators, the difference between TreeSets vs. HashSets, how to use a Map with a List or Set as its value type, etc. - any of these types of problems may be included. Recall that an Iterator is just an object that lets us traverse elements of a Collection, and access/remove elements while looking at one at a time.
- For Maps, make sure you're familiar with keys and values, and be able to handle the case of reversing keys to values, counting number of unique values, etc. Refer to Section 9 for different problem types!
- Remember to identify whether you're using a TreeMap or HashMap. If the keys must be sorted, you have to use a TreeMap but if the key type doesn't implement comparable, e.g. Point, you must use HashMap. Same goes with Sets.
- For problems using Maps, compare the differences between adding to a Map that has a data structure as its value type (e.g. Set<Integer>) and one that doesn't (e.g. Integer)
    - A common pattern with nested data structures / maps: a special case for the first time you put something in the map. ex: if (!containsKey) then put the key with a new ArrayList, TreeSet, etc. See problems like `groupExchanges` from section.
- It can be tricky to correctly identify the types of the parameters and returns if you're not careful. Be sure to find that info in the problem statement and circle it before you begin. Otherwise, you may be solving a different problem.
- Remember that Sets don't have indices, so usually you'll want to use a for-each loop (or Iterator if you're modifying the Set)

Inheritance

- Refer to the Inheritance section handout (Section 16) for review of compiler vs. runtime errors. You'll need to be able to know when and why these occur!
- Many students find it most efficient to first draw the inheritance tree and then draw and fill in table of classes/methods - refer to Erika's example walkthrough in Inheritance/Polymorphism lecture if needed!
- Be careful not to evaluate method calls in the method table, since you'll be working with different variables with different object types.
- Review super calls if you're still unsure about how to evaluate them (these lecture notes provide a great overview and motivation for the different cases in inheritance and polymorphism!)
- There are a good number of practice problems from section - we recommend going through at least 3 (keeping in mind prioritization of harder programming problems), otherwise it can be time-consuming on the exam if you don't develop the muscle memory to approach this problem efficiently

Comparable Class

- Remember you'll have to **create a whole class here.** It's easy points to identify the right fields and methods before you do the compareTo method. toString() will also be a likely method, so make sure you can convert certain types to a requested format (ie. currency to exactly two decimals)
- Remember, EVERY comparable class needs to implement `Comparable<class I'm writing>` and include the `compareTo(class I'm writing other)`. Here's the example format for `Office`:

```
public class Office implements Comparable<Office> {
    private int numCouches;
    private String officeOwner;
    private List<String> furniture;
    …
    // constructors, other methods, etc…

    public int compareTo(Office other) {
       // return < 0 if this object is less than other
       //        > 0 if this object is greater than other
       //          0 (equal)
       // If you're comparing two doubles, remember that don't want
       // to lose precision from returning double1 - double2, so
       // it's a good idea to check manually like you would
       // for other non-integer types.
    }
}
```

- Don't forget to make fields private!!
- Familiarize yourself with the ways various data types are compared. Floating point values are compared differently than integers, and sometimes you may have to call another class's compareTo method.
- Underline any special cases (ie. a list of furniture that is in sorted order (alphabetical), etc.)
- Try a couple of these (there are 8 on the section handout) – remember, it is easy to forget small details when writing classes on paper. And they're pretty fun.

LinkedIntList Programming

- Draw pictures! Drawing arrows representing the traversal of a ListNode current variable and reassignment of nodes, labeling them with the corresponding step number, and then translating this to actual statements will really help ensure you don't lose parts of your Linked List or miss cases like having more than one node in a row to remove.
- Remember, multiple nodes may point to the same node, but one node can only point to one other node! Remembering this will help you avoid losing part of your list.
- Use descriptive variable names to help you know where you're at
- Remember special cases (front, middle, end, empty case). Save yourself some writing by thinking carefully about exactly which cases need to be handled in special ways.
- For some problems, the middle case is the same as the front case with "front" replaced with "current.next". If you write one of these first, keeping this in mind can help with writing the other. This can also serve as a sanity check for afterward.
- Run through your code after you're done - chances are you may find yourself losing nodes, getting a NullPointerException, forgetting an empty / single-node case, etc.

- Practice, practice, practice! LinkedIntList problems are usually the more difficult and time-consuming problems during the exam for students.