Topics

• Exam I stuff, II stuff: polymorphism, exceptions, lists, complexity, and recursion.

• Trees
  – Understand all the tree terminologies: node, edge, height, level, complete, balanced, perfect.
  – Be familiar with the underlying implementations of Binary Trees, BSTs, and trees from homework problems (e.g., Expression Trees, Huffman Trees).
  – Understand the BST properties.
  – Understand how add/search/delete/traversal works in a BST.
  – Be able to write recursive methods, including those that manipulate known data structures. For instance, think size(), height(), and remove().
  – Be familiar with the best/worst-case BST height, and how they are derived.
  – Be able to justify the time complexities of commonly used BST methods.

• Heaps and Priority Queues
  – Understand the Min-Heap (and by extension, Max-Heap) properties.
  – Know how to identify parent/children nodes in a heap.
  – Be familiar with the “percolate” algorithms, and when they’re used.
  – Understand how the heap’s balance is maintained after adding/removing nodes.
  – Be able to justify the time complexities of commonly used heap methods.
  – Understand how a priority queue is implemented using a heap.

• Sets and Maps
  – Know the Set and Map interfaces.
  – Know the Set operations, and though we didn’t implement Sets, think about how you might.
  – Maps: Understand the differences between open-addressing and chaining implementations.
  – Understand the space-time tradeoff and its impact on Maps’ performance.
  – Know the situations in which you would use or avoid using a Map.

• Sorting Algorithms
  – I won’t ask you to blindly reproduce the algorithm on an exam, but you should understand the mechanisms behind: insertion, bubble/shaker, odd-even, merge, heap, and quick sort.
  – Know under which circumstances would trigger sorting algorithm’s best or worst case.
  – Know the best and worst-case time complexity of each.
Practice Problems

1. **[BST]** Draw BST that would result if these elements were added to an empty tree in this order: Kirk, Spock, Scotty, McCoy, Chekov, Uhura, Sulu, Khaan!

```
1. Kirk
  /   \
3. Chekov       Spock
  /     /   \
5. Khaan!     Scotty      Uhura
 /     /     /   \
7. McCoy       Sulu
```

2. **[BST]** Write the elements of your tree above in the order they would be visited by each kind of traversal: pre-order, post-order, and in-order.

   **Sol:**
   - Pre-order: Kirk, Chekov, Khaan!, Spock, Scotty, McCoy, Uhura, Sulu
   - Post-order: Khaan!, Chekov, McCoy, Scotty, Sulu, Uhura, Spock, Kirk
   - In-order: Chekov, Khaan!, Kirk, McCoy, Scotty, Spock, Sulu, Uhura

3. **[BST Programming]** Write a method `public static BinarySearchTree<Integer> buildBST(int[] list)` that accepts a sorted array of comparable objects of type E, and constructs a balanced BST containing those objects. The tree should be constructed so that for every node, either the left/right subtrees have the same number of nodes, or the left subtree has one more node than the right.

   Notice that when it is not possible to have left and right subtrees of equal size, the extra value always ends up in the left subtree, as in the overall tree which has 5 nodes in the left subtree and 4 in the right. For full credit, your solution must run in $O(n)$ time, where $n$ is the number of elements in the array. You may assume that the values in the array appear in sorted order. You may define private helper methods to solve this problem. You also may not alter the array that you are passed.

   **Sol:** See code file.
4. **[BST Programming]** Write a method in the BinarySearchTree class (get it from lab solution — code would be provided to you if this question appeared on the exam), public int treeSum(), that returns the sum of all the nodes. If the Nodes do not store integers, you should throw a NumberFormatException. You can create helper methods, but you cannot use an instance variable.

```java
BinarySearchTree<Integer> bst1 = new BinarySearchTree<>();
bst1.add(5);
bst1.add(17);
bst1.add(23);
System.out.println(bst1.treeSum());
> 45

BinarySearchTree<String> bst2 = new BinarySearchTree<>();
bst2.add("Riley");
bst2.add("Keith");
System.out.println(bst2.treeSum());
> java.langNumberFormatException: Nodes do not store ints!
at BinarySearchTree.treeSum(BinarySearchTree.java:225)
at BSTProblems.main(BSTProblems.java:39)
```

**Sol:** See code file. Inside BinarySearchTree class.

5. **[Map Programming]** Write a method called sumStrings takes a Map whose keys are strings and whose values are doubles and that returns a Map that associates each double-value with the sum of the lengths of the strings it is associated with in the input Map. For instance, suppose we have a Map called data has the following associations:

```
[a=4.2, apple=1.0, be=2.6, bear=7.9, carpet=5.7, cat=4.2, dog=3.5, specialty=7.9, student=4.2, umbrella=5.13]
```

Then the call sumStrings(data) should return the following Map:

```
[1.0=5, 5.13=8, 3.5=3, 4.2=11, 5.7=6, 7.9=13, 2.6=2]
```

Notice that the value 1.0 maps to 5 in the result because it was associated with a string of length 5 in the original ("apple"). Notice that 4.2 maps to 11 because it was associated with three strings ("a", "cat", "student") whose lengths add up to 11. Your method should construct the new Map, but should otherwise not construct any new data structures. It should also not modify the map passed as a parameter and it should be reasonably efficient.

**Sol:** See code file.

6. **[Sorting]** For this problem on the exam, I would give you the implementation of bubbleSort. Modify the bubbleSort algorithm so that it implements shakerSort. The Big-O complexity should not change.

**Sol:** See code file.
7. **[Sorting]** Consider the quicksort algorithm. What is the pivot? Why is it important to choose a good one? Describe why the algorithm fails when the input is mostly sorted.

   **Sol:** During each pass of quicksort, the pivot is the element that gets sorted into place. Quicksort does this by calling `partition()`, which swaps items less-than the pivot to its left, and all other items to its right. It’s important to pick a good pivot because it directly affects quicksort’s performance. This is because quicksort is called recursively for the sublists preceding and succeeding the pivot, so if the sizes of these sublists are imbalanced, then the larger sublist would consume more work.

   In the worst case, the pivot is always selected as the first or final element in the unsorted sublist. For instance, if the pivot happens to be the smallest or largest item, then one of the sublists will be empty (and require no work), while the other list only reduced by one element.

8. **[Sorting]** Here is an array which has just been partitioned by the first step of quicksort: 3, 0, 2, 4, 5, 8, 7, 6, 9. Which of these elements could be the pivot? (There may be more than one possibility!)

   **Sol:** 4 and 5 could both serve as pivots because numbers on either side of them are all less on the left, and greater than or even to, on the right. On the exam, I’d expect both to be identified.

9. **[Sorting]** Given that quicksort’s worst-case complexity is worse than mergesort, why is quicksort so commonly used in practice?

    **Sol:** It’s all about space. Quicksort can be run without auxiliary storage, but mergesort requires another list just to hold the merged list (and at least in our implementation, two other lists to hold the copies of the halves). Moreover, we showed in lecture that randomly choosing a pivot element is a good approach to avoiding the worst-case.