CSSE230 Exam 2 Winter 2020–21 Name: ________________________ Section: 01 02 03

You may use any printed or written materials that you brought with you, code that you wrote yourself or with a partner before the exam, anything on the CSSE230 Moodle and web sites, and the Java API documentation (on your laptop or at oracle.com).

You may not otherwise search the internet or communicate with any person other than your instructor or a student assistant; no communication devices allowed. You may not communicate with anyone else about this exam until at least 5:00 PM today.

You must actually get these problems working on your computer. All of the credit for each will be for code that actually works and for its efficiency and elegance. Use parameters and return values, not instance variables (fields), to pass information between methods. Circumventing the purpose of a problem (e.g., writing a tree method by copying all contents of a tree to an array and operating on that array instead) may result in no credit even if unit tests pass.
For elegance, aim for simple code, for example, using NULL_NODE to eliminate special cases and doing your recursion in the inner BinaryNode class.

For each problem, we will start with the score from the unit tests, and compute efficiency and elegance scores manually. Note that you will lose unit-test points if (for example) your code is hard-coded to work on our specific unit tests.

Turn-in checklist:

1. _____ I did not and will not give or receive help on this exam. Signature: ___________________________

2. _____ I saved my work and committed my solution to my personal git repository.

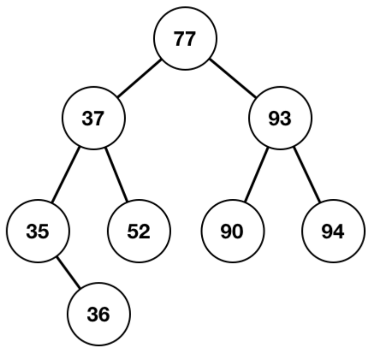
3. _____ I verified in Eclipse that my solution was committed.

4. _____ I am about to hand this test paper, with my name and section number on it, to my instructor.

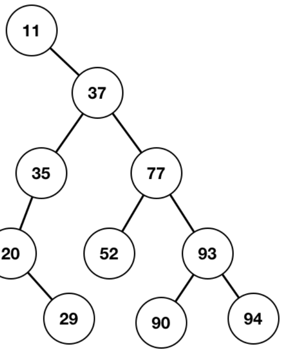
Problem	**Possible**	**Score**	**Comments**
1	16		
2 unit tests	15		
2 efficiency	5		
3	20		
Elegance	4		
Total	**60**		

Do all work in the `BinarySearchTree` class in the **Exam2** project from your repository.
The following trees are used in some test cases and may help you plan your solutions.

Tree *t1*



Tree *t2*

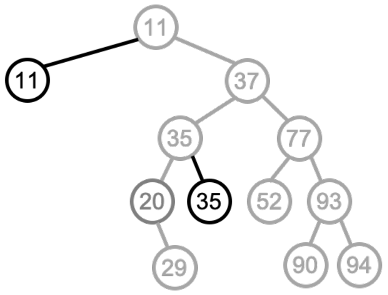


Note: we use BSTs of `ints` rather than a BSTs of generic type `T` on this exam, to simplify your code.
All methods that we specify for you to write are **BinarySearchTree** instance methods, but for elegance you will likely want to write recursive helper methods in the **BinaryNode** class. You shouldn't modify the existing BST fields or completed methods, and it would be inelegant to add a field that doesn't have relevance outside a method call, but it is OK to write a custom class whose objects only exist within a method call.

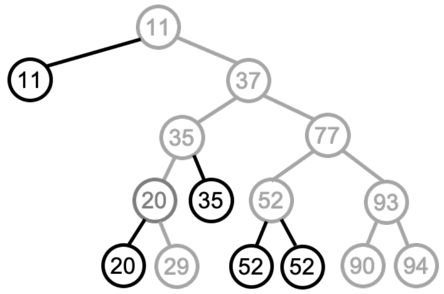
1. (16 points) **int countSiblingDiffGreaterThan(int threshold)**
   If a BST of integers has a node with two children, then the right child's value will always be larger than the left child's. Call the positive difference between them the "sibling difference". This method will return the number of nodes with a sibling difference strictly greater than the given **threshold** value. On t1 above given input 10, the method returns 2, since 93 – 37 = 46 > 10 and 52 – 35 = 17 > 10, so 2 nodes. 94 – 90 = 4 ≤ 10 so it is not counted, and there are no other nodes in t1 with two children. The unit tests have more examples.

2. (20 points = 15 unit tests + 5 efficiency) **ArrayList<Integer> buildSearchPathList(int item)**
   This method returns an ArrayList of all integers on the path from the root to the item. For example, in t1 and for item = 52, it returns [77, 37, 52]. We will also check your code for efficiency: it should run in O(height(T)) time. Finally, if the given item isn't in the tree, it returns an empty list. (Test-taking hint: if it's not immediately clear how to handle that, you may choose to move on to #3 and return to it later.)

3. (20 points) **void sproutToDepth(int maxDepth)**. Given an argument **maxDepth** ≥ –1, "sprout" the tree—by which we mean, insert a new Node wherever there is currently a NULL_NODE—at all locations where that new Node would have depth at most **maxDepth**. Each new Node's data should equal the value in the node's parent from the original tree. (If the tree was initially empty, a sprouted Node should have data 0.)
   Note: this means that the resulting tree will no longer be a BST, since it will have repeated values.
   For example: (old nodes grey; sprouted nodes black)



Result of `t2.sproutToDepth(3)`



Result of `t2.sproutToDepth(4)`