CSE 123 Practice Final Exam Key

Section (e.g., AA):____________________ Student Number: _________________________

Do not turn the page until you are instructed to do so.

Name of Student: ***KEY***

Rules/Guidelines:

- You must not begin working before time begins, and you must stop working **promptly** when time is called. Any modifications to your exam (writing *or* erasing) before time begins or after time is called will result in a penalty.
- You are allowed one page of notes, no larger than 8.5 x 11 inches. You may not access any other resources or use any electronic devices (including calculators, phones, or smart watches, among others) during the exam. Using unauthorized resources or devices will result in a penalty.
- In general, you are limited to Java concepts or syntax covered in class. You may not use `break`, `continue`, a `return` from a `void` method, `try/catch`, or Java 8 features.
- You are limited to the standard Java classes and methods listed on the provided reference sheet. You do not need to write import statements.
- If you abandon one answer and write another, ***clearly cross out*** the answer(s) you do not want graded and ***draw a circle or box*** around the answer you do want graded. When in doubt, we will grade the answer that appears in the space indicated, and the first such answer if there is more than one.
- If you require scratch paper, raise your hand and we will bring some to you.
- If you write an answer on scratch paper, please ***write your name and clearly label*** which question you are answering on the scratch paper, and ***clearly indicate*** on the question page that your answer is on scratch paper. Staple all scratch paper you want graded to the ***end*** of the exam before turning in.
- Answers must be written as proper Java code. Pseudocode or comments will not be graded.
- The exam is not graded on code quality. You are not required to include comments.
- You are also allowed to abbreviate "System.out.print" and "System.out.println" as "S.o.p" and "S.o.pln" respectively. You may **NOT** use any other abbreviations.

Grading:

- Each problem will receive a single E/S/N grade.
  - On problems 1 through 3, earning an E requires answering all parts correctly and earning an S requires answering almost all parts correctly.
  - On problems 4 through 6, earning an E requires an implementation that meets all stated requirements and behaves exactly correctly in *all* cases. Earning an S requires an implementation that meets all stated requirements and behaves exactly correctly in *most* cases or behaves nearly correctly in *all* cases.
- Minor syntax errors will be ignored as long as it is unambiguous what was intended (e.g. forgetting a semicolon, misspelling a variable name where there is only one close option). Major syntax errors, or errors where it is unclear what was intended, may have an impact on your grade.

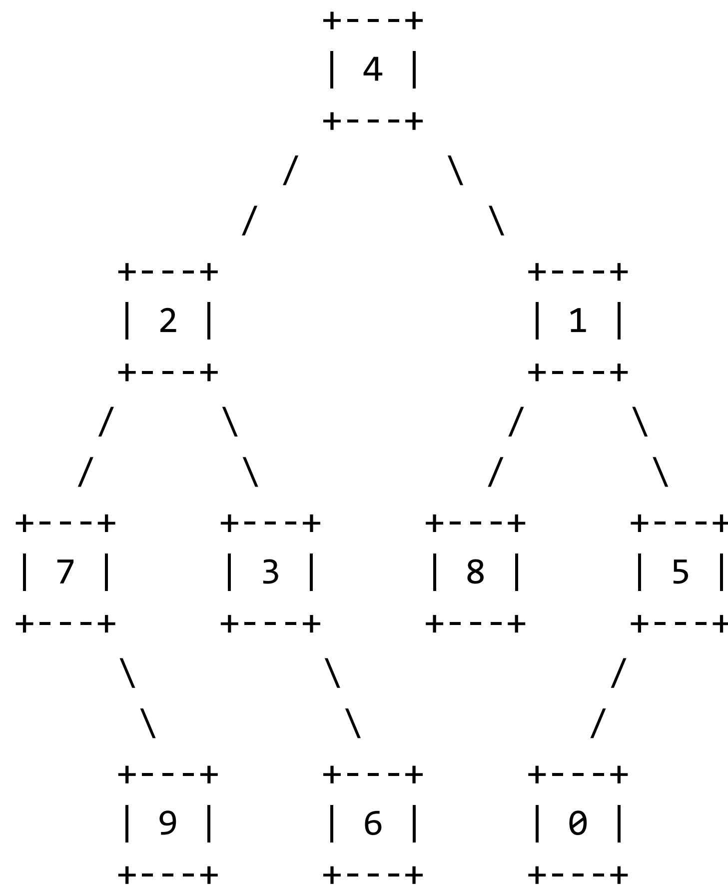
Advice:

- Read all questions carefully. Be sure you understand the question *before* you begin your answer.
- The questions are not necessarily in order of difficulty. Feel free to skip around. Be sure you are able to at least attempt every question.
- Write clearly and legibly. We cannot award credit for answers we cannot read.
- If you have questions, raise your hand to ask. The worst that can happen is we will say "I can't answer that."
- Ask questions as soon as you have them. Do not wait until you have several questions.

Initial here to indicate you have read and agreed to these rules:

1. Code Comprehension

Part A: For the following binary tree, write the preorder, inorder, and postorder traversal.



Preorder: 4 2 7 9 3 6 1 8 5 0

Inorder: 7 9 2 3 6 4 8 1 0 5

Postorder: 9 7 6 3 2 8 0 5 1 4

Part B: (Select all that apply) Which of these statements are true about inheritance?

- ☐ Inheritance allows a subclass to access all private properties and methods of its parent class.
- ☑ Calling a method using `super` will go to the parent class one level above the current subclass.
- ☐ To indicate that class A is a subclass of B, we write `public class B extends A`
- ☐ Parent classes can call the methods of their subclasses.
- ☑ If class A is a subclass of B, then the following statement is legal:
  `B a = new A();`

Part C: Consider the following method in the `IntTree` class:

```
1      public List<String> method() {
2           List<String> result = new ArrayList<>();
3           methodHelper(overallRoot, result, "");
4           return result;
5       }
6
7       private void methodHelper(TreeNode root, List<String> result, String s) {
8           if (root != null) {
9               s += root.data;
10              if (root.left == null && root.right == null) {
11                  result.add(s);
12              } else {
13                  s += ", ";
14                  methodHelper(root.left, result, s);
15                  methodHelper(root.right, result, s);
16              }
17          }
18      }
19
```

Provide a tree that, if called by the above method, would result in a list of size 3.

Any tree with 3 leaf nodes

2. Code Tracing

Part A: For each of the following, write the code necessary to convert the following sequences of ListNode objects:

Before: `list -> [5] -> [4] -> [3] /`	**After:** `list -> [4] -> [5] -> [3] /`

```
ListNode temp = list.next;
list.next = list.next.next;
temp.next = list;
list = temp;
```

Before:
`list  -> [1] -> [2] /`
`list2 -> [3] -> [4] /`
	After:
	`list  -> [4] -> [1] /`
	`list2 -> [2] -> [3] /`

```
list.next.next = list2;
list2.next.next = list;
list = list2.next;
list2 = list.next.next;
list.next.next = null;
list2.next.next = null;
```

Part B: Consider the following classes:

```
public class Wanda extends Hulk {
    public void method1() {
        System.out.print("Wanda1 ");
    }

    public void method2() {
        System.out.print("Wanda2 ");
        super.method2();
    }
}

public class Superman extends Thor {
    public void method1() {
        System.out.print("Superman1 ");
    }

    public String toString() {
        return “Good news everyone!”;
    }
}
```

```
public class Hulk extends Thor {
    public void method2() {
        System.out.print("Hulk2 ");
        super.method2();
    }
}

public class Thor {
    public void method1() {
        System.out.print("Thor1 ");
    }

    public void method2() {
        System.out.print("Thor2 ");
        method1();
    }

    public String toString() {
        return “We’re doomed!”;
    }
}
```

(continued on next page…)

Given the classes above, what output is produced by the following code?

```
Thor[] superheros = {new Wanda(), new Thor(), new Superman(), new Hulk()};
for (int i = 0; i < superheros.length; i++) {
    superheros[i].method2();
    System.out.println();
    System.out.println(superheros[i]);
    superheros[i].method1();
    System.out.println();

}
```

Indicate what the output of each of the following statements would be.

superheros[0]: Wanda	Wanda2 Hulk2 Thor2 Wanda1 We’re doomed! Wanda1
superheros[1]: Thor	Thor2 Thor1 We’re doomed! Thor1
superheros[2]: Superman	Thor2 Superman1 Good news everyone! Superman1
superheros[3]: Hulk	Hulk2 Thor2 Thor1 We’re doomed! Thor1

3. Linked List Debugging

Consider a method in the `LinkedIntList` class called **`removeFromEnd`** that takes an integer as a parameter and removes the nth node from the end of the list. You may assume n is less than or equal to the size of the list.

For example, suppose the variable list1 contains the following list:

```
list1 = [1, 8, 9, 3, 12]
```

After the call `list1.removeFromEnd(2)` executes, the variables `list1` would contain the following list:

```
list1 = [1, 8, 9, 12]
```

Consider the following buggy implementation of **`removeFromEnd`**:

```
1      public void removeFromEnd(int n) {
2          ListNode curr = front;
3          for (int i = 0; i < n; i++) {
4              curr = curr.next;
5          }
6
7          if (curr == null) {
8              front = front.next;
9          } else {
10             ListNode temp = front;
11             while (curr != null) {
12                 curr = curr.next;
13                 temp = temp.next;
14             }
15             temp.next = temp.next.next;
16         }
17     }
```

(continued on next page…)

This implementation contains a *single* bug that is causing it to not work as intended.

Part A: Identify the single line of code that contains the bug. Write your answer in the box to the right as a single number.

11

Part B: Annotate (write on) the code below to indicate how you would fix the bug. You may add (using arrows to indicate where to insert), remove (by crossing out), or modify (with a combination) any code you choose. However, the fix should not require a lot of work.

```
1      public void removeFromEnd(int n) {
2          ListNode curr = front;
3          for (int i = 0; i < n; i++) {
4              curr = curr.next;
5          }
6
7          if (curr == null) {
8              front = front.next;
9          } else {
10             ListNode temp = front;
11             while (curr.next != null) { // Not stopping one early
12                 curr = curr.next;
13                 temp = temp.next;
14             }
15             temp.next = temp.next.next;
16         }
17     }
```

4. Inheritance Programming

You have been asked to extend a pre-existing class `Student` that represents a college student. A student has a name, a year (such as 1 for freshman and 4 for senior), and a set of courses that the student is taking. The `Student` class is as follows:

```
public class Student {
      private String name;
      private int year;
      private Set<String> courses;

      public Student(String name, int year) {
            this.name = name;
            this.year = year;
            this.courses = new HashSet<>();
      }

      public void addCourse(String name) {
            courses.add(name);
      }

      public void dropAll() {
            courses.clear();
      }

      public int getCourseCount() {
            return courses.size();
      }

      public String getName() {
            return name;
      }

      public int getYear() {
            return year;
      }
}
```

You are to define a new class called **`GradStudent`** that extends `Student` through inheritance. A **`GradStudent`** should behave like a `Student` except for the following differences:

- A grad student keeps track of a research advisor, which is a professor working with the student.
- Grad students are considered to be 4 years further ahead than typical students. So, for example, a grad student in year 1 of grad school is really in year 5 of school overall.
- Grad students can enroll in a maximum of 3 courses at a time. If a grad student tries to add additional courses beyond 3, the course is not added to the student's set of courses.
- If grad students work too much, they become "burnt out." A burnt-out student is one who is in their 5th or higher year of grad school (9th or higher year of school overall) or one who is taking 3 courses.

You should provide the same methods as the superclass, as well as the following new behavior.

Constructor/Method	**Description**
`public GradStudent(String name, int year, String advisor)`	Constructs a graduate student with the given name, given year (where 1 is the first year of grad school, 5th year of school overall; etc.), and research advisor
`public String getAdvisor()`	Returns this grad student's research advisor
`public boolean isBurntOut()`	Returns true if grad student is "burnt out" (in at least the 5th year of grad school, and/or taking 3 courses)

`GradStudent` should also implement the `Comparable` interface; `GradStudents` are sorted first by their year in school (younger students come before older students) then by the number of classes they are currently taking (less classes come before more classes). Those that are in the same year of school and are currently taking the same amount of classes are sorted alphabetically by their name.

Write your solution to problem #4 here:

One possible solution:

```
public class GradStudent extends Student implements Comparable<GradStudent> {
      private String advisor;

      public GradStudent(String name, int year, String advisor) {
            super(name, year + 4);
            this.advisor = advisor
      }

      public void addCourse(String name) {
            if (super.getCourseCount() < 3) {
                  super.addCourse(name);
            }
      }

      public String getAdvisor() {
            return this.advisor;
      }

      public boolean isBurntOut() {
            return super.getCourseCount() >= 3 || super.getYear() >= 9;
      }

      public int compareTo(GradStudent other) {
            if (this.getYear() != other.getYear()) {
                  return this.getYear() - other.getYear();
            } else if (this.getCourseCount() != other.getCourseCount()) {
                  return this.getCourseCount() - other.getCourseCount();
            } else {
                  return this.getName().compareTo(other.getName());
            }
      }

}
```

5. Recursive Programming

Write a recursive method called **`partitionable`** that accepts a list of integers as a parameter and discovers whether the list can be partitioned into two sub-lists of equal sum. Your method should return **`true`** if the given list can be partitioned equally, and **`false`** if not. THe table below includes various possible values for variable name **`list`** and the value that would be returned by the call of **`partitionable(list)`**:

List	Return
`[]`	`true`
`[42]`	`false`
`[1,2,3]`	`true`
`[1,2,3,4,6]`	`true`
`[2,1,8,3]`	`false`
`[8,8]`	`true`
`[-3,14,3,8]`	`true`
`[-4,5,7,2,9]`	`false`

For example, the list **`[1,2,3]`** can be split into **`[1,2]`** and **`[3]`**, both of which have a sum of 3. The list **`[1,2,3,4,6]`** can be split into **`[1,3,4]`** and **`[2,6]`**, both of which have a sum of 8. For the list **`[2,1,8,3]`**, there is no way to split the list into two sub-lists with equal sums.

You are allowed to modify the list passed in as the parameter as you compute the answer, as long as you restore it to its original form by the time the overall call is finished. You may assume that the list passed in is not **`null`**, but it might be empty. Do not use any loops in solving this problem, it must be solved recursively.

Write your solution to problem #5 here:

One possible solution:

```
public boolean partitionable(List<Integer> list) {
      return partitionableHelper(list, 0, 0);
}

private boolean partitionableHelper(List<Integer> rest, int sum1, int sum2) {
      if (rest.isEmpty()) {
            return sum1 == sum2;
      } else {
            int n = rest.remove(0);
            boolean result = partitionableHelper(rest, sum1 + n, sum2) ||
                             partitionableHelper(rest, sum1, sum2 + n);
            rest.add(0, n);
            return result;
      }
}
```

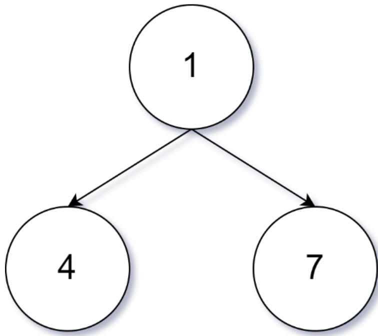
6. Binary Tree Programming

Write a method called **add** that takes as a parameter a reference to a second binary tree and that adds the values in the second tree to this tree. If the method is called as follows:

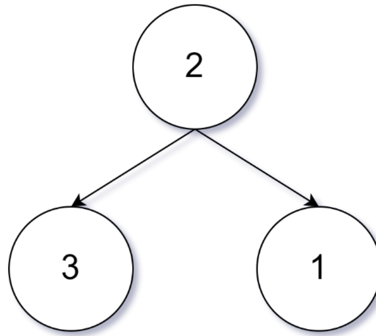
```
tree1.add(tree2);
```

it should add all values in `tree2` to the corresponding nodes in `tree1`. In other words, the value stored at the root of `tree2` should be added to the value stored at the root of `tree1` and the values in `tree2`'s left and right subtrees should be added to the corresponding positions in `tree1`'s left and right subtrees. The values in `tree2` should not be changed by your method.

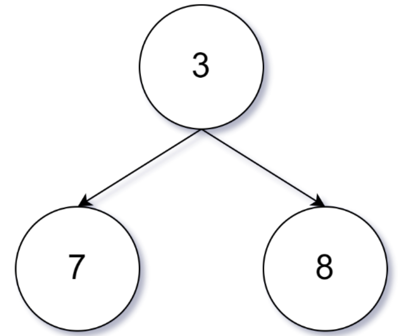
initial tree1



initial tree2

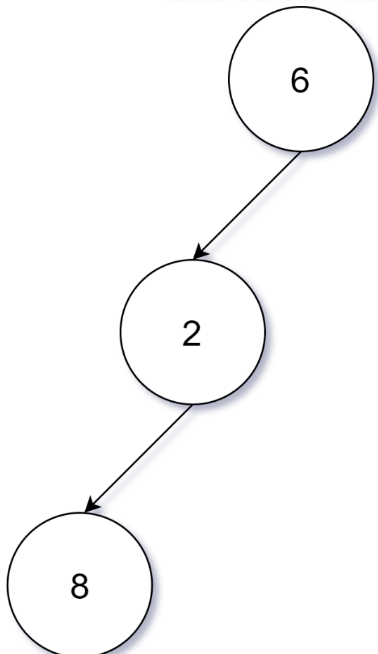


final tree1

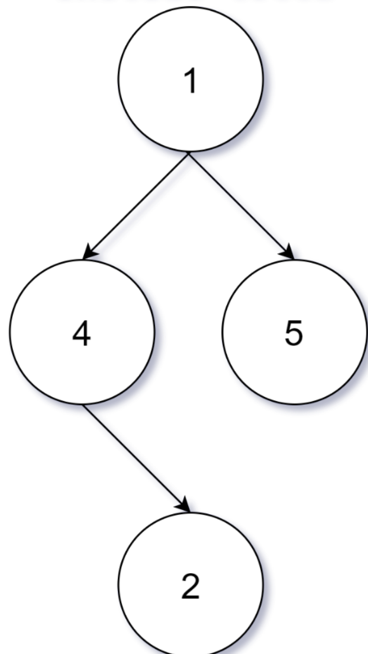


If `tree1` has a node that has no corresponding node in `tree2`, then that node is unchanged. For example, if `tree2` is empty, `tree1` is not changed at all. It is also possible that `tree2` will have one or more nodes that have no corresponding node in `tree1`. For each such node, create a new node in `tree1` in the corresponding position with the value stored in `tree2`'s node. For example:

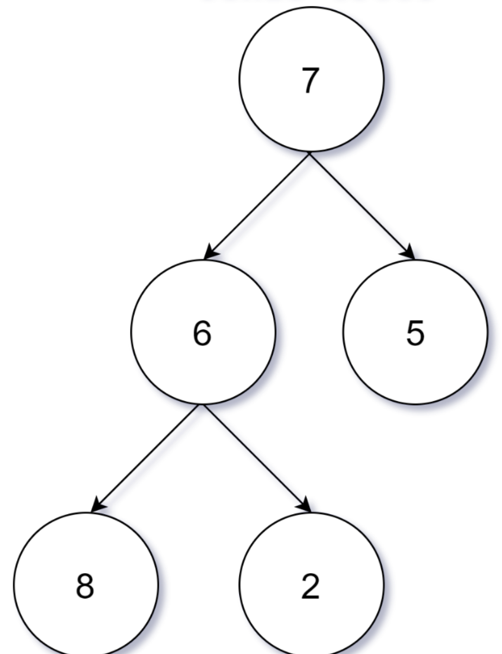
initial tree1



initial tree2



final tree1



Write your solution to problem #6 here:

One possible solution:

```
public void add(IntTree other) {
      overallRoot = add(this.overallRoot, other.overallRoot);
}

private IntTreeNode add(IntTreeNode root, IntTreeNode otherRoot) {
      if (otherRoot != null) {
            if (root == null) {
                  root = new IntTreeNode(otherRoot.data);
            } else {
                  root += otherRoot.data;
            }
            root.left = add(root.left, otherRoot.left);
            root.right = add(root.right, otherRoot.right);
      }
      return root;
}
```

This page intentionally left blank for scratch work