CSE143 Final
Autumn 2019

Name of Student: _____

Section (e.g., AA):_____ Student Number: _____

The exam is divided into eight questions with the following points:

#	Problem Area	Points	Score
1	Binary Tree Traversal	6	_____
2	Binary Search Tree	4	_____
3	Inheritance/Polymorphism	10	_____
4	Comparable	20	_____
5	Collections	10	_____
6	Binary Tree Programming	10	_____
7	Binary Tree Programming	20	_____
8	LinkedList Programming	20	_____
EC	Extra Credit	+0.5	_____
	Total	100	_____

This is a closed-book/closed-note exam. Space is provided for your answers. There is a "cheat sheet. You are not allowed to access any of your own papers during the exam.

The exam is not, in general, graded on style and you do not need to include comments. For the Collections questions, however, you are expected to use generics properly and to declare variables using interfaces when possible. Some problems may specify additional restrictions. You are not allowed to use programming constructs like break, continue, or returning from a void method on this exam.

Do not abbreviate code, such as "ditto" marks or dot-dot-dot ... marks. For the inheritance problem, you may abbreviate “compiler error” as CE and “runtime error” as RE.

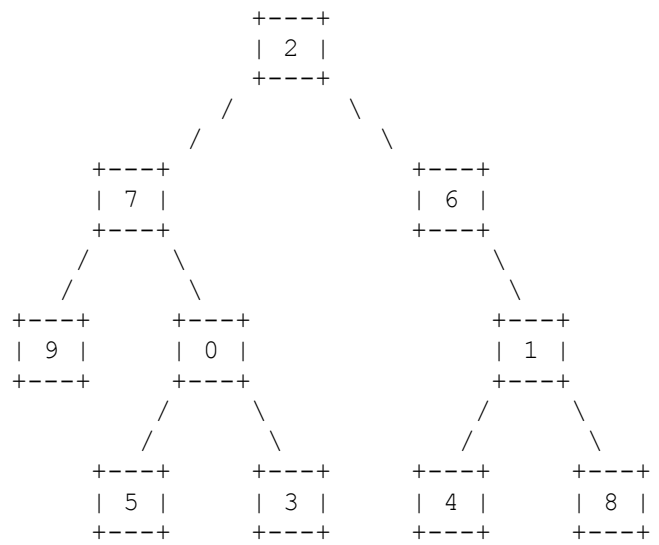
You are NOT to use any electronic devices while taking the test, including calculators. Anyone caught using an electronic device will receive a 10 point penalty.

Do not begin work on this exam until instructed to do so. Any student who starts early or who continues to work after time is called will receive a 10 point penalty. If you finish the exam early, please hand your exam to the instructor and exit quietly through the front door.

You are allowed to ask for scratch paper after the exam starts to use as additional space when writing answers, but you must indicate on the original page for the problem that part of the answer is on scratch paper. Scratch paper must be stapled to the end of your exam after you finish the test. Failure to do so may result in your work on scratch paper not being graded.

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

1. **Binary Tree Traversals, 6 points**: Consider the following tree:



Fill in each of the traversals below:

Preorder traversal _____

Inorder traversal _____

Postorder traversal _____

2. **Binary Search Tree, 4 points**: Draw a picture below of the binary search tree that would result from inserting the following words into an empty binary search tree in the following order:

Ravioli, Ziti, Rigatoni, Maccheroni, Penne, Conchiglie, Tortellini

Assume the search tree uses alphabetical ordering to compare words.

3. **Inheritance/Polymorphism, 10 points**: Assuming that the following classes have been defined:

```
public class Coffee {
    public void method1() {
        System.out.println("Coffee 1");
        method2();
    }

    public void method2() {
        System.out.println("Coffee 2");
    }
}

public class Drip extends Coffee {
    public void method2() {
        System.out.println("Drip 2");
    }

    public void method3() {
        System.out.println("Drip 3");
    }
}

public class Latte extends Coffee {
    public void method1() {
        System.out.println("Latte 1");
    }

    public void method2() {
        System.out.println("Latte 2");
        method1();
    }
}

public class Mocha extends Latte {
    public void method1() {
        super.method1();
        System.out.println("Mocha 1");
    }

    public void method3() {
        System.out.println("Mocha 3");
    }
}
```

And assuming the following variables have been defined:

```
Coffee var1 = new Drip();
Coffee var2 = new Mocha();
Drip var3 = new Drip();
Latte var4 = new Latte();
Latte var5 = new Mocha();
Object var6 = new Latte();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a/b/c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with either the phrase "compiler error" or "runtime error" to indicate when the error would be detected; you may use the abbreviations "CE" and "RE" respectively.

Statement	Output
`var1.method1();`	_____
`var2.method1();`	_____
`var3.method1();`	_____
`var4.method1();`	_____
`var5.method1();`	_____
`var6.method1();`	_____
`var1.method2();`	_____
`var2.method2();`	_____
`var3.method2();`	_____
`var4.method2();`	_____
`var5.method2();`	_____
`var6.method2();`	_____
`var3.method3();`	_____
`var5.method3();`	_____
`((Drip) var1).method3();`	_____
`((Mocha) var5).method3();`	_____
`((Coffee) var3).method3();`	_____
`((Drip) var2).method3();`	_____
`((Latte) var2).method2();`	_____
`((Mocha) var6).method2();`	_____

4. **Comparable, 20 points**: Define a class called **MovieRating** that represents a movie and the ratings it receives. A MovieRating starts with no ratings, but ratings can be added in the form of 0 to 5 stars (inclusive). The MovieRating class has the following methods:

MovieRating(String name)	Constructs a new MovieRating for a movie with the given name
void addRating(int rating)	Adds the given rating (0 to 5 stars) for this movie
int getVotes(int rating)	Returns the number of times the given rating (0 to 5 stars) has been given for this movie
double averageRating()	Returns the average star rating for this movie. Returns 0.0 if there are no ratings at all. An example below shows how this number is computed
String toString()	Returns a string representation of this MovieRating

For the methods above, you can assume the rating parameters are valid: a number between 0 and 5 inclusive. Your class will need to record the number of times each possible rating has been given for this movie (e.g. number of times a 0-star review has been given, the number of 1-star reviews, etc.). To do this, your class must use an fixed-length array to store the number of times each possible star-rating has been given. This array should store the data such getVotes(...) is has O(1) run-time.

If the MovieRating has no ratings, the toString should return the string with the following format (where <name> is a placeholder for the movie's name)

"<name>: No ratings yet!"

If ratings have been added, it should instead return a string with this format

"<name>: <average rating> (<num ratings> reviews)"

For example, if the following lines are executed:

```
MovieRating movie0 = new MovieRating("Trolls");
MovieRating movie1 = new MovieRating("Mad Max");
movie1.addRating(5);
movie1.addRating(4);
movie1.addRating(5);
```

Then the following calls to toString would return:

```
movie0.toString();    "Trolls: No ratings yet!"
movie1.toString();    "Mad Max: 4.66 (3 ratings)"
```

Notice the average rating for movie1 is 4.66 since there is one 4-star review and two 5-star reviews: (4 + 5 + 5) / 3 = 4.66. This is the same as the return value for averageRating for this movie. The averageRating for movie0 should be 0.0. Your class does not need to worry about rounding this output or the result of averageRating.

In addition, the MovieRating class should implement the Comparable<E> interface. MovieRating objects are first compared by their average rating, where the MovieRating with a lower average rating should be consider less-than the other. Ties are broken by comparing the number of 5-star reviews, where a movie with fewer 5-star reviews than the other is considered less-than. Then ties should be broken the same way with 4-star reviews, then 3-star, all the way down to (and including) 0-star reviews. If there is still a tie after comparing all possible star values, MovieRatings should be compared by the movie's name, where alphabetically earlier names should be considered greater-than.

This problem is generally not graded on style, but you must make your fields private, use an array as specified above, and avoid redundancy in the compareTo method with respect to comparisons based on various star ratings.

This page is left blank so you have extra space on #4

5. **Collections Programming, 10 points:** Write a method called twoFlightsAway that computes which cities are exactly two flights away from other cities. Your method should take as a parameter a map whose keys are names of cities and whose values are sets of the names of the cities that can be reached by one flight from the city represented by the key. For each city in the map, you should generate a set of all that cities that can reached by taking exactly 2 flights using the algorithm described below.

For example, suppose your method were passed a map called flights with the entries below on the left. Then the result of calling twoFlightsAway(flights) should return a new map with the following entries on the right.

```
       flights                                    twoFlightsAway(flights)
Cairo   => [Ottawa, Rome]                  Cairo   => [Seattle, Tokyo]
Ottawa  => [Cairo, Seattle, Tokyo]         Ottawa  => [Rome]
Pyrus   => []                              Rome    => [Ottawa]
Rome    => [Cairo]                         Seattle => [Cairo]
Seattle => [Ottawa, Tokyo]                 Tokyo   => [Cairo]
Tokyo   => [Ottawa, Seattle]
```

Cairo was listed as having Seattle and Tokyo as two-flights-away cities in the result, because both Seattle and Tokyo have direct flights from Ottawa, which has a direct flight to Cairo. Notice that Tokyo is not in the result for Seattle. Even though there is a way to take two flights from Seattle to Tokyo (through Ottawa), there is already a direct flight from Seattle to Tokyo which makes them not two-flights-away. A city cannot be considered two-flights-away from itself.

More generally, the result set for city C1 should be the set of all cities that can be flown to from the set of all cities that can be flown to from C1 (a verbose way of describing all cities that are two flights away). This result set should not include C1 itself or any cities that can be flown directly from C1. In other words, city C3 should be in the result for city C1 if there is at least one city C2 that can be reached directly from C1 and C3, but there is no direct flight from C1 to C3 and C1 and C3 aren't the same city.

If a city has no two-flights-away cities, it should not have an entry in the returned map (hence Pyrus not being in the result). The keys in your result map and values the sets of cities should appear in sorted order.

You may assume flights are bidirectional and the map has valid data. This means if there is an entry in the map indicating a flight from C1 to C2, there will be a corresponding entry in the map for a flight from C2 to C1. You may assume that the map and its entries are not null. You may not modify the map given to you.

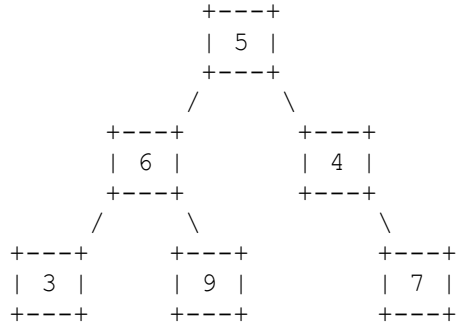
You should use space on the next page to write your answer.

This page is left blank so you have extra space on #5

6. **Binary Tree Programming, 10 points:** Write a method of the IntTree class called **randomPathSum** that takes a random path from the top of the tree down and returns the sum of the data in the nodes on that path. To choose the random path, we will use a simple strategy that starts at the root and randomly choose which nodes to include in the path based on a random number generator as it traverses the tree. It's possible for the chosen path to be the empty path with no nodes, in which case the sum is 0.

Starting with the root node, you should repeat the following procedure for each node visited. You should randomly generate a number between 0 and 2 inclusive. If the number is 1, you should include the root node's data and then continue deciding the path with the nodes to the left of the root node. If the number is 2, it should do the same as the 1 case except, continue making the path to the right of the root node. If the number is 0, the root node should not be included and the path ends. These numbers should be generated with equal probability. The sum of a path with no nodes (e.g. a path that ends before choosing nodes or if the tree is empty) is 0.

For example, suppose that a variable t stores a reference to the following tree and we call t.randomPathSum()



Below, we will describe one possible execution of the program (since it is random).
* We start at the overall root 5. Say our random number generator returns 1 for this first node. In this case, we should include the node with 5 in the path and then continue choosing a path to the left of the 5.
* Say then our random number generator returns a 2. This means we should include the 6 in the path and then continue to the right.
* Say then the random number generator returns a 0. That means we should not include the 9 in the path and our process should stop.
* In this case, the method should then return 11 since that is the sum of the numbers chosen in the path (5 and 6)

You are writing a public method for a binary tree class defined as follows:

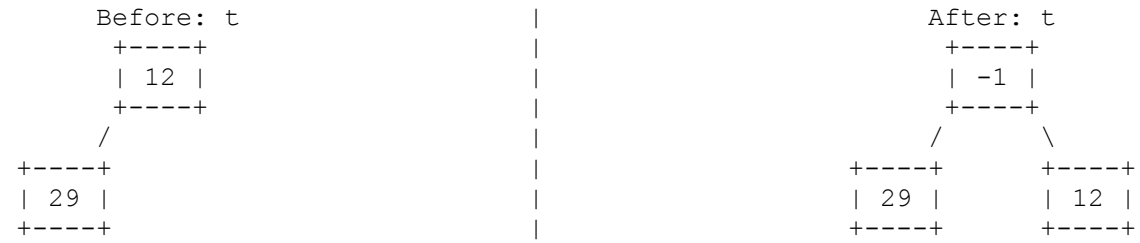
```
public class IntTree {
    private IntTreeNode overallRoot;
    <methods>
    private static class IntTreeNode {
        public int data;          // data stored in this node
        public IntTreeNode left;  // reference to left subtree
        public IntTreeNode right; // reference to right subtree
        <constructors>
    }
}
```

You are writing a method that will become part of the IntTree class.

Your solution must meet the following restrictions:
* You may define private helper methods to solve this problem, but otherwise you may not assume that any particular methods are available.
* Your method must not use any loops and must use recursion.
* Your method should be efficient in the sense that it only explores nodes that the random number generator indicates should be included in the path. This means if the generator says 1, you should only go left and if it returns 0, you should stop traversing the tree.

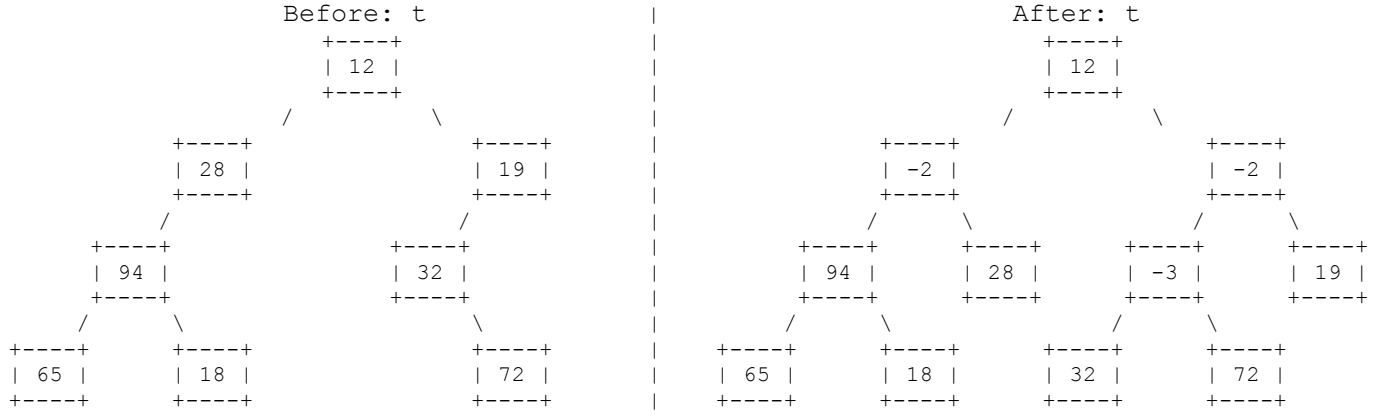
This page is left blank so you have extra space on #6

7. **Binary Tree Programming, 20 points:** Write a method of the IntTree class called **makeFull** that turns a binary tree of integers into a full binary tree. A binary tree is full if every node has either 0 or 2 children. Your method should produce a full binary tree by replacing each node that has one child with a new node that has the old node as a leaf where there used to be an empty tree. The new node should store a value that indicates the level of the tree (-1 for the first level of the tree, -2 for the second level of the tree, and so on). For example, if a tree called t stores the following tree on the left, after a call to t.makeFull() it should then store the tree on the right:



Notice that the node storing 12 that used to be at the top of the tree is now a leaf where there used to be an empty tree. In its place at the top of the tree is a new node that stores the value -1 to indicate that it was added at level 1 of the tree. Your method should perform this operation at every level of the tree.

As another example, if t had instead stored the tree on the left, after the call to t.makeFull(), it would now store the tree on the right:



Notice that two nodes were added at level 2, and one at level 3.

You are writing a public method for a binary tree class defined in problem 6.

Your solution must meet the following restrictions:

* You may define private helper methods to solve this problem, but otherwise you may not assume that any particular methods are available.
* You are not allowed to change the data fields of the existing nodes in the tree.
* You are not allowed to construct new nodes or additional data structures outside of the new nodes to store the negative level (notice the 3-argument IntTreeNode constructor that can be used to do so).
* Your solution must run in O(n) time where n is the number of nodes in the tree.

This page is left blank so you have extra space on #7

8. **LinkedList Programming, 20 points:** Write a method for the LinkedIntList class called **expand** that treats the list as a series of count/value pairs and modifies the list to be the expansion of those count/value pairs.

For example, given the following list named list has been defined with the following values (pairs underlined for emphasis):

```
list = [3, 8, 4, 2, 1, 7, 5, 1]
```

This list indicates that the result should have 3 occurrences of 8 followed by 4 occurrences of 2 followed by 1 occurrence of 7 followed by 5 occurrences of 1. Then after a call of list.expand() the list should contain the following values:

```
[8, 8, 8, 2, 2, 2, 2, 7, 1, 1, 1, 1, 1]
```

Notice that the values in the resulting list appear in the same order as the original list. You may assume that the list will contain an even number of elements and the list will not contain any values less than or equal to 0.

You are writing a public method for the LinkedIntList class defined as follows:

```
public class LinkedIntList {
    private ListNode front;
    <methods>

    private static class ListNode {
        public int data;       // data stored in this node
        public ListNode next;  // link to next node in the list

        <constructors>
    }
}
```

Your solution should follow the following restrictions:

* You may define private helper methods to solve this problem, but otherwise you may not assume that any particular methods are available.
* You also may not change any data fields of the nodes already in the list. You MUST solve this problem by constructing new nodes for the expanded values and rearranging the links of the lists.
* You may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc).
* Your solution must run in O(n) time where n is the length of the resulting list.

Grading: About half the points in this problem are partial credit that is awarded for having the code that handles the requisite components to solve the problem. The remaining points are considered "external correctness" where certain points are awarded based on if the written solution correctly works with lists of various lengths (length 0, length 2, length4, etc.). This means even if you don't know how to fully solve the problem, you can still get a partial credit for attempting to write the necessary code and getting it to work in small cases. However, in order to receive full credit, you must verify that the code actually works on all input types.

This page is left blank so you have extra space on #8

Extra Credit: You and your best friends, your TA and your favorite topic from 143, decide you want to do something fun after the exam today! Draw a picture or write a short-story describing what the three of you decide to do today. Note that artistic ability is not required to earn this point; any work that demonstrates at least one minute of effort and is not inappropriate, offensive, or disrespectful will be given credit of 0.5 points.

CSE143 Cheat Sheet

*(DO NOT WRITE ANY WORK YOU WANT GRADED ON THIS SHEET. IT WILL **NOT** BE GRADED)*

Linked Lists

Below is an example of a method that could be added to the LinkedIntList class to compute the sum of the list:

```
public int sum() {
    int sum = 0;
    ListNode current = front;
    while (current != null) {
        sum += current.data;
        current = current.next;
    }
    return sum;
}
```

`Math` Methods

mathematical operations

`Math.abs(`*value*`)`	absolute value
`Math.min(`*v1*`, `*v2*`)`	smaller of two values
`Math.max(`*v1*`, `*v2*`)`	larger of two values
`Math.random()`	random value between 0.0 and 1.0
`Math.round(`*value*`)`	nearest whole number
`Math.pow(`*b*`, `*e*`)`	b to the e power
`Math.signum(v)`	signum of v

`Iterator<E>` Methods

(An object that lets you examine the contents of any collection)

`hasNext()`	returns `true` if there are more elements to be read from collection
`next()`	reads and returns the next element from the collection
`remove()`	removes the last element returned by `next` from the collection

`List<E>` Methods

(An ordered sequence of values)

`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value at given index, shifting subsequent values right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`isEmpty()`	returns `true` if the list's size is 0
`contains(`**value**`)`	returns `true` if the given value is found somewhere in this list
`remove(`**value**`)`	finds and removes the given value from this list if it is present
`iterator()`	returns an object used to examine the contents of the list

`Set<E>` Methods

(A fast-searchable set of unique values)

`add(`**value**`)`	adds the given value to the set
`contains(`**value**`)`	returns `true` if the given value is found in the set
`remove(`**value**`)`	removes the given value from the set if it is present
`clear()`	removes all elements of the set
`size()`	returns the number of elements in the set
`isEmpty()`	returns `true` if the set's size is 0
`iterator()`	returns an object used to examine the contents of the set

*(DO NOT WRITE ANY WORK YOU WANT GRADED ON THIS SHEET. IT WILL **NOT** BE GRADED)*

`Map<K, V>` Methods

(A fast mapping between a set of keys and a set of values)

`put(`**`key, value`**`)`	adds a mapping from the given key to the given value
`get(`**`key`**`)`	returns the value mapped to the given key (`null` if none)
`containsKey(`**`key`**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**`key`**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`keySet()`	returns a `Set` of all keys in the map
`values()`	returns a `Collection` of all values in the map
`putAll(`**`map`**`)`	adds all key/value pairs from the given map to this map

`String` Methods

(An object for storing a sequence of characters)

`charAt(`**`i`**`)`	the character in this String at a given index
`compareTo(`**`str`**`)`	compares this String to the given String based on alphabetical order, returning an int as defined by the `Comparable<E>` interface
`contains(`**`str`**`)`	`true` if this String contains the other's characters inside it
`endsWith(`**`str`**`)`	`true` if this String ends with the other's characters
`equals(`**`str`**`)`	`true` if this String is the same as *str*
`equalsIgnoreCase(`**`str`**`)`	`true` if this String is the same as *str*, ignoring capitalization
`indexOf(`**`str`**`)`	first index in this String where given String begins (-1 if not found)
`lastIndexOf(`**`str`**`)`	last index in this String where given String begins (-1 if not found)
`length()`	number of characters in this String
`isEmpty()`	`true` if this String is the empty string
`startsWith(`**`str`**`)`	`true` if this String begins with the other's characters
`substring(`**`i`**`)`	characters in this String from index *i* (inclusive) to the end
`substring(`**`i, j`**`)`	characters in this String from index *i* (inclusive) to *j* (exclusive)
`toLowerCase(),toUpperCase()`	a new String with all lowercase or uppercase letters

`Random` Methods

(An object for creating random numbers)

`Random()`	constructs a new `Random` object
`nextInt(`**`bound`**`)`	returns a random int between 0 (inclusive) and `bound` (exclusive)
`nextDouble()`	returns a random double between 0.0 (inclusive) and 1.0 (exclusive)

Collections Implementations

`List<E>`	`ArrayList<E>` and `LinkedList<E>`
`Set<E>`	`HashSet<E>` and `TreeSet<E>` (values ordered)
`Map<K, V>`	`HashMap<K, V>` and `TreeMap<K, V>` (keys ordered)