CSE143 Final
Autumn 2018

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	15	_____
5	Collections	15	_____
6	Binary Tree Programming	10	_____
7	Binary Tree Programming	20	_____
8	LinkedList Programming	20	_____
	Total	100	_____

This is a closed-book/closed-note exam. Space is provided for your answers. There is a "cheat sheet" at the end that you can use as scratch paper. 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. You are not allowed to use programming constructs like break, continue, or returning from a void method on this exam. Do not use constructs from Java 8.

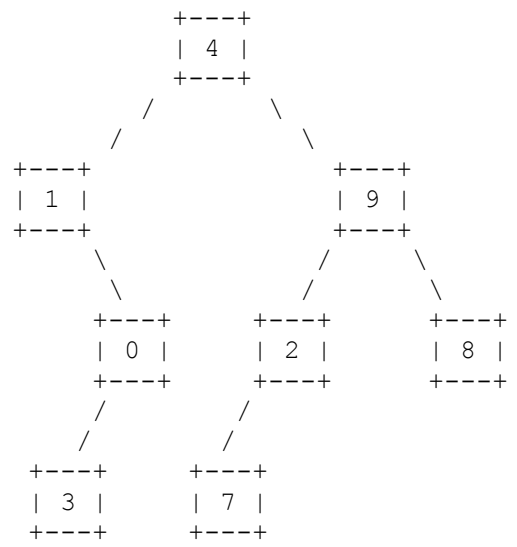
Do not abbreviate code, such as "ditto" marks or dot-dot-dot ... marks. For the inheritance problem, you may abbreviate “compiler error” as CE and “runtime error” as RE.

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.

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:

Hilda, David, Johanna, Frida, Woodman, Alfur, Twig

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 Pupper extends Doggo {
    public void method2() {
        System.out.println("Pupper 2");
    }

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

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

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

public class Pet {
    public void method1() {
        System.out.println("Pet 1");
        method3();
    }

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

public class Doggo extends Pet {
    public void method3() {
        System.out.println("Doggo 3");
        super.method3();
    }
}
```

And assuming the following variables have been defined:

```
Pet var1 = new Doggo();
Pupper var2 = new Pupper();
Pet var3 = new Cat();
Pet var4 = new Pupper();
Pet var5 = new Pet();
Object var6 = new Doggo();
```

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();`	_____
`var1.method3();`	_____
`var2.method3();`	_____
`var3.method3();`	_____
`((Pupper) var6).method1();`	_____
`((Doggo) var3).method2();`	_____
`((Pupper) var4).method2();`	_____
`((Pet) var3).method2();`	_____
`((Cat) var3).method2();`	_____
`((Pupper) var1).method1();`	_____
`((Doggo) var4).method3();`	_____
`((Pet) var6).method3();`	_____

4. **Comparable, 15 points**: Define a class PokemonTrainer that stores information about a Pokemon trainer and how many battles they have won. Each PokemonTrainer object keeps track of a trainer's name, number of badges, total battles played, and number of battles won.

The class has the following public methods:

PokemonTrainer(String name, int badges)	constructs a PokemonTrainer object with the given name and number of badges. When a PokemonTrainer object is constructed, it has played zero battles.
getBadges()	returns the number of badges the trainer has earned
getBattlePercent()	returns a real number representing the exact percent of battles this trainer has won. If the trainer has won all of their battles, should return 100.0, if the trainer has won none of their battles, should return 0.0. If the trainer has not played any battles yet, should return 0.0
battle(boolean won)	records a battle for this trainer. Passed true if the trainer won the battle, false otherwise
toString()	returns a String with name, number of badges, and percent of wins (or "no battles" if none). The battle percentage should be truncated after the decimal, so if getBattlePercent() returned 73.835, then toString would report a battle percent of 73%

Below is example client code using PokemonTrainer objects:

```
PokemonTrainer trainer1 = new PokemonTrainer("Sam", 1);
PokemonTrainer trainer2 = new PokemonTrainer("Anika", 6);

trainer2.getBadges();          // returns 6
trainer1.battle(true);         // records a battle win for trainer1
trainer1.battle(false);        // records a battle loss for trainer1
trainer1.getBattlePercent();   // returns 50.0
trainer2.getBattlePercent();   // returns 0.0
trainer1.battle(false);        // records a battle loss for trainer1
trainer1.getBattlePercent();   // returns 33.3333333333333333
trainer1.toString();           // returns "Sam has 1 badge(s) and a 33% win rate"
trainer2.toString();           // returns "Anika has 6 badge(s) and no battles"
```

The PokemonTrainer class should be comparable to other PokemonTrainer objects and should implement the Comparable interface. Trainers that have a higher battle percentage should be considered "less" than other trainers so that they appear at the beginning of a sorted list. You should use the complete value of the battle percentage rather than a truncated value. Trainers that have the same battle percentage should be ordered by the number of battles played, with trainers who have battled more often considered "less" than trainers that have battled less frequently. If there is still a tie, the trainers should be sorted alphabetically by name.

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

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

5. **Collections Programming, 15 points:** Write a method called numPlacesTraveled that takes a List describing a trip someone took to some place and returns a map indicating the number of unique places each person has visited. The input will be a list of strings in the format "<name>:<location>". The map you are to return should map people's names to the number of unique locations they have traveled to.

For example, if a variable called trips contained the list

```
["Erik:Reno", "Porter:Mexico", "Cherie:Vancouver", "Erik:Mexico", "Erik:LasVegas",
 "Cherie:LosAngeles", "Porter:NewYork", "Cherie:Vancouver", "Yuma:LosAngeles",
 "Erik:Reno"]
```

In this example, we see that Erik has taken trips to Reno, Mexico, Las Vegas, and then Reno again, while Yuma has only been to Los Angeles. Suppose the following call is made:

```
numPlacesTraveled(trips)
```

Given this call, the following map would be returned

```
{"Cherie"=2, "Yuma"=1, "Porter"=2, "Erik"=3}
```

Notice that the value for the key "Erik" is 3 because Erik has been to 3 unique places of the ones he has traveled to. The value for the key "Yuma" is 1 because Yuma has only been to one place.

The map you return should prioritize constant-time access for the values in the map. If the given list is empty, your method should throw an IllegalArgumentException.

Your method should run faster than O(n^2) time where n is the number of elements in the input list. Your method should not modify the provided list. To solve this problem, you will need to create auxiliary data structures; you should think about what data structures would be most appropriate for this problem.

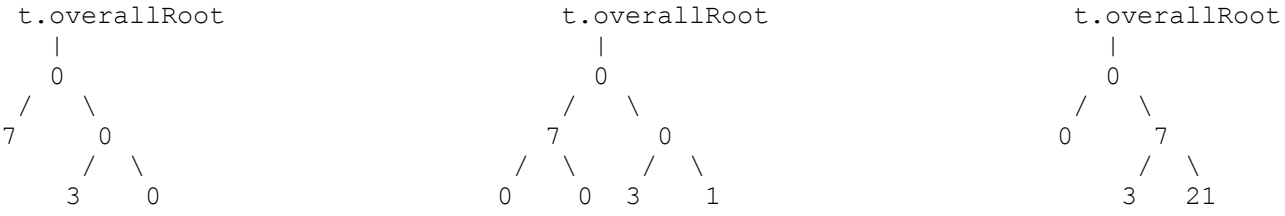
You may assume that the list and none of the strings in the list are null. You may also assume that the strings are in the proper format "<name>:<location>" with exactly one instance of ":" in the string. You should treat the names and locations in a case-sensitive manner and use the values in the strings as given without having to modify them.

You can 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 hasZeroPath that returns true if there is a path from the root of the tree to a leaf consisting only of elements with zero value. An empty tree is considered to have a 0-length path and should therefore return true.

Examples



hasZeroPath() returns true hasZeroPath() returns false hasZeroPath() returns true

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

```
public 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>
}

public class IntTree {
    private IntTreeNode overallRoot;

    <methods>
}
```

You are writing a method that will become part of the IntTree class. You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class. You may not construct any extra data structures to solve this problem.

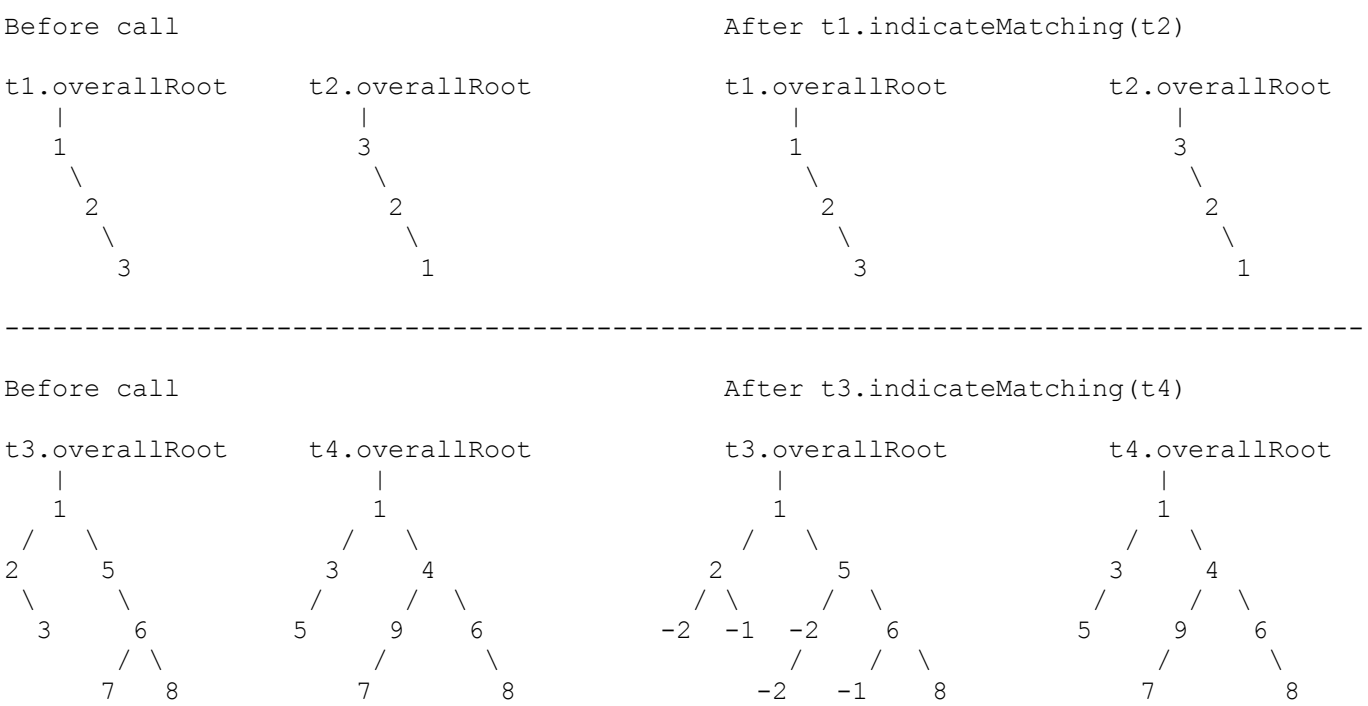
Space is provided on the next page.

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

7. **Binary Tree Programming, 20 points:** Write an IntTree method indicateMatching which compares the nodes of this IntTree to the nodes of a second IntTree. For each node in the trees, change this tree in the following way:

* If a node exists in this tree but not the other tree, replace it with -1.
* If a node exists in the other tree but not this tree, replace it with -2.
* Otherwise, leave the node unchanged.

Here are two sample calls:



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

```
public class IntTreeNode {
    public int data;          // data stored in this node
    public IntTreeNode left;  // reference to left subtree
    public IntTreeNode right; // reference to right subtree

    public IntTreeNode(int data) { ... }
    public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) { ... }
}

public class IntTree {
    private IntTreeNode overallRoot;

    <methods>
}
```

You are writing a method that will become part of the IntTree class. You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class. You may not construct any extra data structures to solve this problem. Your solution should run in O(n) time, where n is the number of nodes in the trees and should create as few IntTreeNodes as possible. Your solution should not modify the data of any existing nodes.

Space is provided on the next page.

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

8. **LinkedList Programming, 20 points:** Write a method of the LinkedIntList class called mergeWith that takes a LinkedIntList as a parameter and that moves the values from the second list into the first list so as to preserve sorted order assuming that the two lists are in sorted (nondecreasing) order initially. The resulting list should contain the values from both lists in sorted order and the list passed as a parameter should be empty after the call. For example, if the variables list1 and list2 store the following:

```
list1 = [-3, 0, 9, 12, 43, 54], list2 = [9, 9, 15, 98]
```

and you make the following call:

```
list1.mergeWith(list2);
```

then the lists should store the following values after the call:

```
list1 = [-3, 0, 9, 9, 9, 12, 15, 43, 54, 98], list2 = []
```

Notice that list2 is empty after the call and that the values that were in list2 have been moved into list1 so as to preserve sorted order. If the call instead had been list2.mergeWith(list1) then the result would be:

```
list1 = [], list2 = [-3, 0, 9, 9, 9, 12, 15, 43, 54, 98]
```

Either list might be empty, as in:

```
list1 = [5, 7, 7, 12, 15], list2 = []
```

in which case the call list1.mergeWith(list2) would leave the lists unchanged while list2.mergeWith(list1) would leave you in this state:

```
list1 = [], list2 = [5, 7, 7, 12, 15]
```

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

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

    <constructors>
}

public class LinkedIntList {
    private ListNode front;

    <methods>
}
```

Both lists are of type LinkedIntList. You may define private helper methods to solve this problem, but otherwise you may not assume that any particular methods are available. You are allowed to define your own variables of type ListNode, but you may not construct any new nodes, and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). You also may not change any data fields of the nodes. You MUST solve this problem by rearranging the links of the lists. Your solution must run in O(n) time where n is the length of the merged list. As in the examples above, assume that both lists are in sorted order.

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

CSE143 Cheat Sheet

Linked Lists (16.2)

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 (3.2)

mathematical operations

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

`Iterator<E>` Methods (11.1)

(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 (10.1)

(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 (11.2)

(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

`Map<K, V>` Methods (11.3)

(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 (3.3)

(An object for storing a sequence of characters)

`charAt(`**`i`**`)`	the character in this String at a given index
`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

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)