CSE 123 Summer 2024 Practice Final 1 Answer Key

Name of Student: _____

Section (e.g., AA):_____ Student Number (7 digits):_____

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

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 stream/functional 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.
- 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. Be sure you 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: 

This page intentionally left blank

Nothing written on this page will be graded

Inheritance and Comparable (A)

Take a look at the given classes.

```
public class Alarm {
    private boolean onOff;
    private int time; // in minutes

    public Alarm() {
        onOff = false;
        time = 0;
    }

    public void turnOnOff() {
        this.onOff = !this.onOff;
    }

    // sets an alarm "time"
    // minutes into the future
    public void setAlarm(int time) {
        this.time = time;
    }
}
```

```
public class BuildingAlarm extends Alarm {}

public class FireAlarm extends BuildingAlarm {}

public class PhoneAlarm extends Alarm {}
```

Based on these classes, these methods will be added. For each method, state whether it's overriding, overloading, or neither.

Class Added To	Method	Type
BuildingAlarm	`public void sendAlert() { System.out.println("Alert sent!"); }`	Neither
FireAlarm	`public void sendAlert() { System.out.println("Fire trucks!"); }`	Overriding
PhoneAlarm	`public void sleep() { setAlarm(5); }`	Neither

What's a possible `compareTo` method that could be written here? Explain the return statement and what is going to be compared (2-3 sentences).

We can compare on the basis of time since the method setAlarm is setting the alarm "time" amount in the future. This way if time < other.time, we would return a negative number, time == other.time would be 0 as it's tied, and time > other.time would be positive.

LinkedList (A)

Write a method `removeRange` that accepts a starting and ending index as parameters and removes the elements at those indexes (inclusive) from the list. For example, if a variable list stores the following values:

[8, 13, 17, 4, 9, 12, 98, 41, 7, 23, 0, 92]
And the following call is made:

```
listRange.removeRange(3, 8);
```

Then the values between index 3 and index 8 (the value 4 and the value 7) are removed, leaving the following list:

[8, 13, 17, 23, 0, 92]
You should throw an `IllegalArgumentException` if either of the positions is negative. Otherwise you may assume that the positions represent a legal range of the list (0 <= start index <= end index < size of list).

Assume that you are adding this method to the `LinkedIntList` class as defined below:

```
public class LinkedIntList {
    private ListNode front;   // null for an empty list
    ...
}
```

```
public void removeRange(int left, int right) {
    while (front != null && front.data >= left && front.data <= right) {
        front = front.next;
    }

    ListNode curr = front;
    while (curr != null && curr.next != null) {
        if (curr.next.data >= left && curr.next.data <= right) {
            curr.next = curr.next.next;
        } else {
            curr = curr.next;
        }
    }
}
```

Binary Trees (A)

Part A

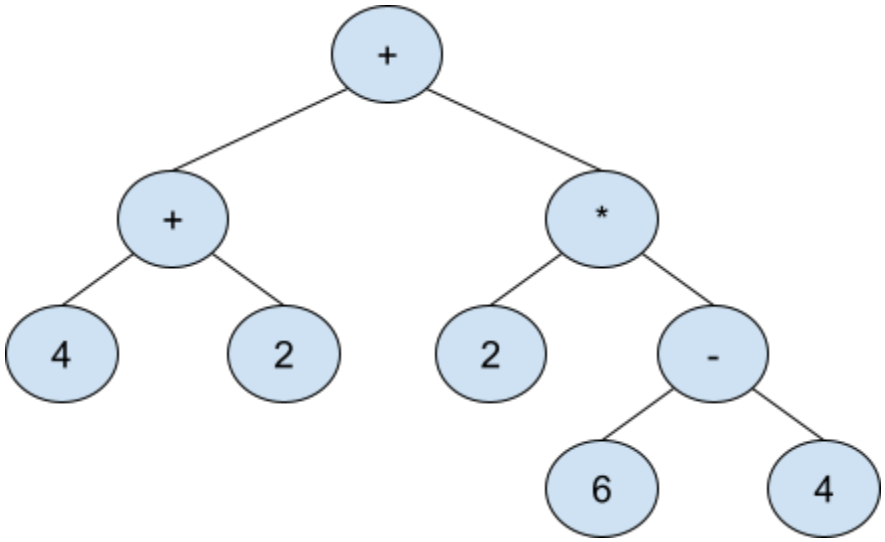
What is the recursive definition of a Binary Tree?

Empty Tree / Node with left and right subtree

Part B

Write a method call `calculate(Node root)` that takes in a binary tree and returns the final result from all the operations. Every intermediate node in the tree will contain an “operand” and all the leaf nodes will contain numbers. The only operations possible are +, -, *, and /.

For example, the mathematical equation (4 + 2) + (2 * (6 - 4)) can be represented by the tree:



Running the method `calculate()` with this tree would return the integer 10. Assume that the given tree will have at least 3 nodes (enough for one operation) and will always be in correct format (mathematically correct)

Write the method `calculate(Node root)`:

```
public static int calculate(Node root) {
    if (root.left == null && root.right == null) {
        return root.data;
    } else {
        String operand = root.operand;

        if (operand.equals("+")) {
            return calculate(root.left) + calculate(root.right);
        } else if (operand.equals("-")) {
            return calculate(root.left) - calculate(root.right);
        } else if (operand.equals("*")) {
            return calculate(root.left) * calculate(root.right);
        } else {
            return calculate(root.left) / calculate(root.right);
        }
    }
}
```

CSE 123 Quiz/Exam Reference Sheet

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

Methods Found in ALL collections (`List`, `Set`, `Map`)

`clear()`	Removes all elements of the collection
`equals(`**`collection`**`)`	Returns `true` if the given other collection contains the same elements
`isEmpty()`	Returns `true` if the collection has no elements
`size()`	Returns the number of elements in a collection
`toString()`	Returns a string representation such as `"[10, -2, 43]"`

Methods Found in both `List` and `Set` (ArrayList, LinkedList, HashSet, TreeSet)

`add(`**`value`**`)`	Adds value to collection (appends at end of list)
`addAll(`**`collection`**`)`	Adds all the values in the given collection to this one
`contains(`**`value`**`)`	Returns `true` if the given value is found somewhere in this collection
`iterator()`	Returns an Iterator object to traverse the collection's elements
`remove(`**`value`**`)`	Finds and removes the given value from this collection
`removeAll(`**`collection`**`)`	Removes any elements found in the given collection from this one
`retainAll(`**`collection`**`)`	Removes any elements *not* found in the given collection from this one

`List<Type>` Methods

`add(`**`index, value`**`)`	Inserts given value at given index, shifting subsequent values right
`indexOf(`**`value`**`)`	Returns first index where given value is found in list (-1 if not found)
`get(`**`index`**`)`	Returns the value at given index
`lastIndexOf(`**`value`**`)`	Returns last index where given value is found in list (-1 if not found)
`remove(`**`index`**`)`	Removes/returns value at given index, shifting subsequent values left
`set(`**`index, value`**`)`	Replaces value at given index with given value

`Map<KeyType, ValueType>` Methods

`containsKey(`**`key`**`)`	`true` if the map contains a mapping for the given key
`get(`**`key`**`)`	The value mapped to the given key (`null` if none)
`keySet()`	Returns a `Set` of all keys in the map
`put(`**`key, value`**`)`	Adds a mapping from the given key to the given value
`putAll(`**`map`**`)`	Adds all key/value pairs from the given map to this map
`remove(`**`key`**`)`	Removes any existing mapping for the given key
`toString()`	Returns a string such as `"{a=90, d=60, c=70}"`
`values()`	Returns a `Collection` of all values in the map

`Math` Methods

`abs(`**`x`**`)`	Returns the absolute value of `x`
`max(`**`x, y`**`)`	Returns the larger of `x` and `y`
`min(`**`x, y`**`)`	Returns the smaller of `x` and `y`
`pow(`**`x, y`**`)`	Returns the value of `x` to the `y` power
`random()`	Returns a random number between `0.0` and `1.0`
`round(`**`x`**`)`	Returns `x` rounded to the nearest integer

`String` Methods

`charAt(`**i**`)`	Returns the character in this String at a given index
`contains(`**str**`)`	Returns `true` if this String contains the other's characters inside it
`endsWith(`**str**`)`	Returns `true` if this String ends with the other's characters
`equals(`**str**`)`	Returns `true` if this String is the same as *str*
`equalsIgnoreCase(`**str**`)`	Returns `true` if this String is the same as *str*, ignoring capitalization
`indexOf(`**str**`)`	Returns the first index in this String where *str* begins (-1 if not found)
`lastIndexOf(`**str**`)`	Returns the last index in this String where *str* begins (-1 if not found)
`length()`	Returns the number of characters in this String
`isEmpty()`	Returns `true` if this String is the empty string
`startsWith(`**str**`)`	Returns `true` if this String begins with the other's characters
`substring(`**i, j**`)`	Returns the characters in this String from index *i* (inclusive) to *j* (exclusive)
`substring(`**i**`)`	Returns the characters in this String from index *i* (inclusive) to the end
`toLowerCase()`	Returns a new String with all this String's letters changed to lowercase
`toUpperCase()`	Returns a new String with all this String's letters changed to uppercase

Inheritance Syntax

```
public class Example extends BaseClass {
    private type field;
    public Example() {
       field = something;
    }
    public void method() {
       // do something
    }
}

public interface InterfaceExample {
    public void method();
}
```

```
public abstract class AbstractExample {
    private type field;

    public void method() {
       // do something
    }

    public abstract void abstractMethod();
}
```

ArrayIntList

```
public class ArrayIntList {
    private int[] elementData;
    private int size;
}
```

LinkedIntList

```
public class LinkedIntList {
    private ListNode front;

    private static class ListNode {
        public int data;
        public ListNode next;

        public ListNode(int data) {
            this(data, null);
        }

        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```