CSE 123 Autumn 2025 Final Exam

Name of Student: _____

Section (e.g., AA):_____ Student UW **Email** :_____@uw.edu

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 be reported as academic misconduct to the university.
- 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 be reported as academic misconduct to the university.
- 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. Loose scratch paper with answers on it is likely to be lost and not graded.
- 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:

- There are six problems. 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: 

1. Inheritance and Polymorphism

Part A: Implementation

Consider the following classes:

```
public interface SpecialPrinter {
   public void printSpecial(String s);
}

public class SimplePrinter
                  implements SpecialPrinter {
    public void printSpecial(String s) {
        System.out.println(s);
    }
}

public class UpperPrinter
                   extends SimplePrinter {
    public void printSpecial(String s) {
        super.printSpecial(s.toUpperCase());
    }
}
```

```
public abstract class MultiPrinter
                      implements SpecialPrinter {
    private int times;
    public MultiPrinter(int times) {
        this.times = times;
    }
    public void multiPrint(String s) {
        for (int i = 0; i < times; i++) {
            printSpecial(s);
        }
    }
}

public class DoublePrinter extends MultiPrinter {
    public DoublePrinter() {
        super(2);
    }
    public void printSpecial(String s) {
        System.out.print(s);
    }
}
```

Fill in the blanks below to complete the `MultiUpperPrinter` class so that the client code shown below will successfully compile and run to produce the indicated output. Completing the class requires extending another (possibly abstract) class, implementing a constructor, and implementing/overriding a method.

```
MultiPrinter mp1 = new MultiUpperPrinter(1);
MultiPrinter mp2 = new MultiUpperPrinter(2);
MultiPrinter mp3 = new MultiUpperPrinter(3);
mp1.multiPrint("cAmEl"); // prints "CAMEL"
mp2.multiPrint("cAmEl"); // prints "CAMEL" twice, one per line
mp3.multiPrint("cAmEl"); // prints "CAMEL" three times, one per line
// In total there are 6 lines printed, each line being the string "CAMEL"
```

```
public class MultiUpperPrinter extends ____________________________ { // class to extend

    public MultiUpperPrinter(int times) {
                                                                      // body of constructor

    }
    public void ____________________________________________________ { // method signature

                                                                      // method body









    }
}
```

Part B: Tracing

Mark the appropriate option in each row below, according to whether the code will have a run-time error, a compile-time error, or no error. If the code has no error, also say what it will print, if anything. If the code has any kind of error, you do not need to say what the output is.

Each row is separate. Do not consider the code of previous rows when looking at the next row.

CE = Compile-time Error, RE = Run-time Error, NE = No Error

Code	CE	RE	NE	Output
`SpecialPrinter x = new SimplePrinter();` `x.printSpecial("james");`				
`SpecialPrinter x = new UpperPrinter();` `x.printSpecial("james");`				
`SpecialPrinter x = new DoublePrinter();` `x.printSpecial("james");`				
`MultiPrinter x = new DoublePrinter();` `x.printSpecial("james");`				
`MultiPrinter x = new SimplePrinter();` `x.printSpecial("james");`				
`MultiPrinter x = new MultiPrinter(3);` `x.multiPrint("james");`				

`SpecialPrinter x = new DoublePrinter();`
`UpperPrinter u = (UpperPrinter) x;`
`u.printSpecial("james");`				

`DoublePrinter x = new DoublePrinter();`
`MultiPrinter m = (MultiPrinter) x;`
`m.multiPrint("james");`				

`DoublePrinter x = new DoublePrinter();`
`SpecialPrinter s = (SpecialPrinter) x;`
`s.multiPrint("james");`				

2. Reference Semantics

Part A: Tracing

Consider the following code:

```
List<List<Integer>> x = new ArrayList<>();
x.add(new ArrayList<>());
x.add(new ArrayList<>());
x.add(x.get(0));
x.get(0).add(17);
x.get(1).add(42);
```

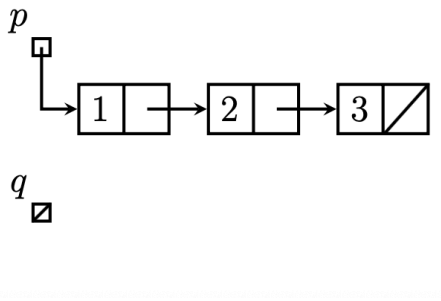
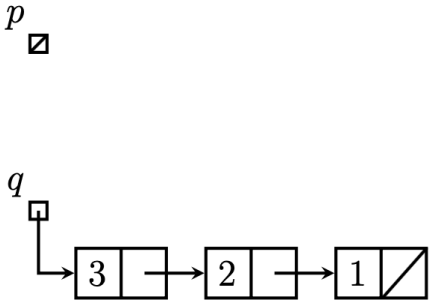
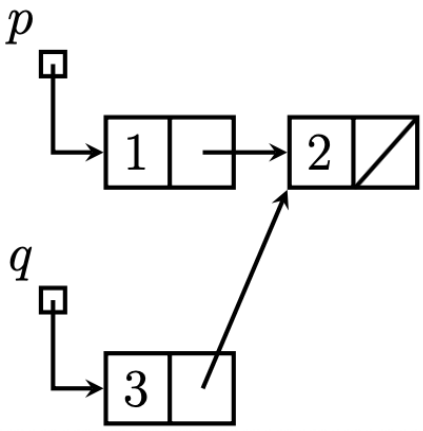
Draw a picture of the state of memory that shows how the different objects reachable through x are related. Use arrows to depict which references refer to which objects. The exact format is up to you. You do *not* need to show details about how the lists are implemented. Show us what you know about reference semantics!

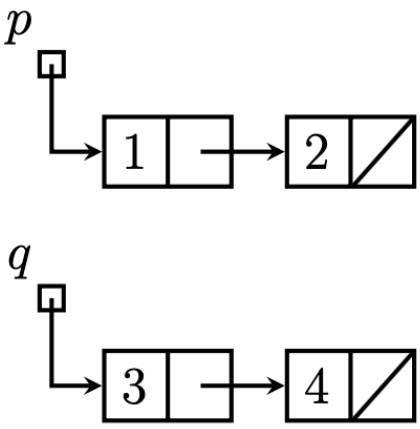
For each of the following code snippets, mark the appropriate box to indicate whether the code will throw a compile-time error (CE), a runtime error (RE), or no error (NE). If there is no error, indicate the resulting value.

Code	**CE**	**RE**	**NE**	**Resulting Value**
x.size()				
x.get(0).size()				
x.get(1).size()				
x.get(0).get(0)				
x.get(1).get(0)				
x.get(2).get(0)				

Part B: Linked List Manipulation

Complete the following table, filling in either the before picture, the code, or the after picture. You should not create any new ListNodes or modify any .data fields

Before	Code	After
		
	`p.next = new ListNode(4);`	

	`p.next.next.next = q;`
	`q = p.next.next;`
	`p.next.next = null;`
		

3. Iterative Programming on Linked Lists

Write a method `replaceIndexRange(start, end, newValue)` to be added to the LinkedIntList class that replaces elements at indices from `start` to `end` (inclusive) with `newValue`. If `start` or `end` are not valid indices into the list (between 0 (inclusive) and the size of the list (exclusive)), or if `end` is strictly less than `start`, then throw an `IllegalArgumentException`.

Original List	**Call**	**Resulting List**
[6, 4, 9, 1, 2, 8, 7]	`replaceIndexRange(2, 4, 0)`	[6, 4, 0, 0, 0, 8, 7]
[6, 4, 9, 1, 2, 8, 7]	`replaceIndexRange(0, 2, 0)`	[0, 0, 0, 1, 2, 8, 7]
[6, 4, 9, 1, 2, 8, 7]	`replaceIndexRange(4, 6, 0)`	[6, 4, 9, 1, 0, 0, 0]

Note that in the first example, `start` is 2, `end` is 4, and `newValue` is 0. So the elements at indices 2, 3, and 4 were replaced by 0s. Your implementation should follow these additional rules:

- Do not use recursion. Do not call any other methods except size(). Do not create helper methods.

```
public void replaceIndexRange(int start, int end, int newValue) {









}
```

4. Runtime Analysis

Analyze the worst-case running time of each method below. Choose the *most accurate* (fastest) correct running time, if more than one choice is correct. Unless otherwise stated, *n* is the length of the input data structure. None of the methods provided throw any exceptions. Assume no resizing is necessary.

Code	**O(1)**	**O(log(n))**	**O(*n*)**	**O(n^2)**
Setting the value at the root of a binary tree to a different value.				
Setting the value at index 5 of a linked list to a different value.				
Setting the value at index 5 of an array list to a different value.				
Adding a value to the beginning of an array list.				
Adding a value to the end of an array list.				
Adding a value to the beginning of a linked list.				
Adding a value to the end of a linked list.				
Adding a value to a binary search tree.				
Adding a value to a balanced binary search tree.				

Running the method `m2` below (which invokes `m1`).
 	
`public void m1(int[] data) {`
`  for (int i = 0; i < data.length; i++) {`
`  System.out.println(data[i]);`
`  }`
`}`
`public void m2(int[] data) {`
`  for (int i = 0; i < data.length; i++) {`
`  m1(data);`
`  }`
`}`				

5. Recursion

This problem is about a method called `arrangeWithRepeats` that takes a set of strings called `elements` and an integer `numRepeats` and prints all ways to arrange the elements into a single string such that each element appears exactly `numRepeats` times in each arrangement. For example, consider the following client code.

```
Set<String> elements = new HashSet<>();
elements.add("a");
elements.add("b");
arrangeWithRepeats(elements, 2);
```

This code will print the following output. (The lines can be printed in any order.)

```
aabb
abab
abba
baab
baba
bbaa
```

These are all the ways to create a string with exactly 2 occurrences of "a" and exactly 2 occurrences of "b". Note that "aba", for example, is not printed, because it only uses "b" once. Also note that no separator is included between the elements in the output string.

Fill in the blanks in the partial solution on the next page to implement `arrangeWithRepeats` and its private helper method.

The helper takes an argument `usesLeft`, which is a map from strings to integers that keeps track of how many times each element still needs to be used. You should write code that initializes usesLeft to map every element to `numRepeats`. You also need to figure out how to detect your base case and make the recursive calls.

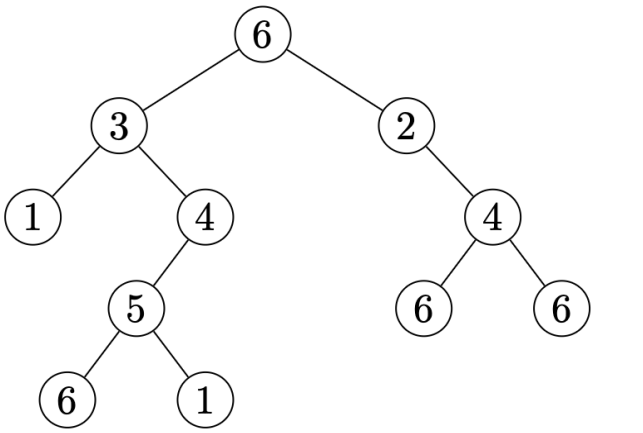
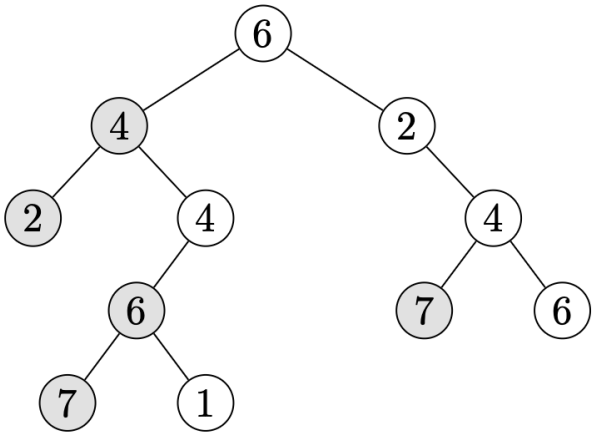
You may assume that `elements` is not `null`, does not contain `null`, and does not contain the empty string, and that `numRepeats` is at least 1. Your code should work for any arguments subject to these assumptions.

Do not remove or change any existing lines of code. You may fill in explicitly underlined blanks and add code in provided whitespace. Some correct solutions may leave some empty spaces blank.

```
public void arrangeWithRepeats(Set<String> elements, int numRepeats) {
    Map<String, Integer> usesLeft = new HashMap<>();




    arrangeWithRepeats(usesLeft ____________________________________________);
}

private void arrangeWithRepeats(Map<String,Integer> usesLeft ____________________) {









    if (_____________________________________________) {          // base case




    } else {
        for (String chosen : usesLeft.keySet()) {
            int oldVal = usesLeft.get(chosen);
            if (oldVal > 0) {


                                                                                  // choose
                arrangeWithRepeats(usesLeft ___________________________________);  // explore
                                                                                  // unchoose




            }
        }
    }
}
```

6. Binary Trees

Write a method `incrementLefties` to be added to the `IntTree` class that increments the value of every node in the tree that is the left child of its parent. (Since the `data` field is final, you will have to increment the value by replacing the node with a new node with updated value.) For example, calling `incrementLefties` on the "before" tree will change it to the "after" tree.

Before	**After**
	

In this diagram, the nodes that were replaced (had their values incremented because they are the left child of their parent) are highlighted in gray. Note that the overall root was not incremented because it is not the left child of any node.

Your implementation must be recursive. No loops are allowed for this problem.

Hint: You will likely need to define and call a helper method.

Hint: You may find it useful to have your helper method take a boolean argument indicating whether the current node is the left child of its parent.

Write your answer on the next page.

```
public void incrementLefties() {
```

7. Art (optional - no credit)

The CSE 123 TAs are a very hard-working group, dedicating a lot of time to serving the students of CSE 123 alongside their own schoolwork. However, every once in a while, they do find time for fun and relaxation. In the space below, draw your TA as you envision them spending their free time. There is no credit for this work, but your TAs are looking forward to seeing your work. 🙂 (Note that artistic ability is *not* required– even stick figures or scribbles will bring a smile to your TA's face.)

CSE 123 Final 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 `"{1=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
`compareTo(`**`str`**`)`	Returns a negative number if this comes lexicographically (alphabetically) before other, 0 if they're the same, positive if this comes lexicographically after other.

`JUnit` Methods

`assertEquals(`**`expected, actual`**`)`	Tests that expected equals actual (using .equals)
`assertNotEquals(`**`expected, actual`**`)`	Tests that expected doesn't equal actual (using .equals)
`assertSame(`**`expected, actual`**`)`	Tests that expected equals actual (using ==)
`assertNotSame(`**`expected, actual`**`)`	Tests that expected doesn't equal actual (using ==)
`assertTrue(`**`actual`**`)`	Tests that actual is true
`assertFalse(`**`actual`**`)`	Tests that actual is false

Abstract Class Syntax

```
public abstract class AbstractClass{
    // an abstract class can contain fields
    private type name;

    // an abstract class can contain constructors
    public AbstractClass(...) {
        // initialize the object
    }

    public abstract returnType abstractMethod(...);

    public returnType implementedMethod(...) {
        ...
    }
}
```

Inheritance Syntax

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

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

    public void method() {
       // do something
    }

    public abstract void abstractMethod();
}
```

ArrayIntList Class

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

    public static final int DEFAULT_CAPACITY = 10;
    public ArrayIntList() {...}
    public void add(int value) {...}
    public int get(int index) {...}
    public String toString() {...}
    public int indexOf(int value) {...}
    public boolean contains(int value) {...}
    public void add(int index, int value) {...}
    public void remove(int index) {...}
    public void set(int index, int value) {...}
    public int size() {...}
}
```

LinkedIntList Class

```
public class LinkedIntList extends AbstractIntList {
    private ListNode front;

    public LinkedIntList() {...}
    public LinkedIntList(int[] nums) {...}
    public void add(int index, int value) {...}
    public int remove(int targetIndex) {...}
    public int size() {...}
    public int get(int index) {...}

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

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

IntTree Class

```
public class IntTree {
    private IntTreeNode overallRoot;

    public IntTree() {...}
    public IntTree(int[] arr) {...}
    public boolean contains(int value) {...}
    public String toString() {...}
    public void replace(int toReplace, int newValue) {...}

    private static class IntTreeNode {
        public final int data;
        public IntTreeNode left;
        public IntTreeNode right;

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

This page intentionally left blank