CSE 123 Autumn 2024 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.
- 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. Concepts

Part A - True/False

Say whether the statements below are true or false. You may abbreviate true as T and false as F.

Statement	**True/False**
James likes drawing pictures.	
James is good at drawing pictures.	
The `compareTo` method of `Comparable` must return -1, 0, or 1.	
When testing, you should run the code first to see the output, then check if it looks right.	
In Java, you can `extend` multiple classes, but `implement` only one interface.	
Run time happens before compile time.	
`super``()` can only be called as the first line of a method.	

Consider the following code:
`public static void m1() {`
`  String x = "";`
`  m2(x);`
`  System.out.println(x.length());  // (***)`
`}`
`public static void m2(String y) { /* unknown code here */ }`
True or False: if the print statement on the line marked (***) runs, it is guaranteed to print 0.	

ArrayIntList.get(index) and LinkedIntList.get(index) are in the same complexity class.	
Modifying a LinkedIntList requires writing to the `front` field or to a `next` field.	
In a recursive method, there is at least one line of code representing the base case.	
An iterative implementation will always have a faster complexity class than a recursive one.	
It is impossible to traverse a binary tree without using recursion.	
It is impossible to modify a binary tree without using x=change(x).	
When a recursive method calls itself, a new call frame is pushed on the call stack.	
When one non-recursive method calls another non-recursive method, no call frame is pushed on the call stack.	
In the worst case, searching a data structure for an element always requires looking at all the data in the data structure.	
Using a binary tree is always faster than using an array.	

Part B - Run-time 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.

Code	**O(1)**	**O(log(n))**	**O(*n*)**	**O(*n*^{2})**
`public void m(int[] data) { for (int i = data.length - 1; i > 0; i -= 2) { System.out.println(data[i]); } }`				
`public void m(int[] data) { for (int i = 0; i < 2147483647; i++) { System.out.println(data[i]); } }`				
`public void m(int[] data) { int n = data.length; for (double x = 0; x < n; x += 1.0 / n) { System.out.println(data[(int)x]); } }`				
`public void m(int[] data) { for (int i = data.length - 1; i > 0; i /= 2) { System.out.println(data[i]); } }`				
Searching for a value in an array.				
Searching for a value in a sorted array.				
Searching for a value in a linked list.				
Searching for a value in a sorted linked list.				
Searching for a value in a binary tree.				
Searching for a value in a binary search tree.				
Searching for a value in a balanced binary search tree.				
Searching for a value in a hash table.				

2. Inheritance and Polymorphism

Consider the following code:

```
public interface Food {
  public String name();
}

public abstract class Fruit
                implements Food {
  public String name() {
    return fruitName() + " fruit";
  }
  public abstract String fruitName();
}
```

```
public class Apple extends Fruit {
  public String fruitName() {
    return "Apple";
  }
  public boolean isRed() { return true; }
}
public class Orange extends Fruit {
  public String fruitName() {
    return "Orange";
  }
  public String name() { return "James"; }
  public int slices() { return 8; }
}
```

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**

`Food x = new Apple();`
`System.out.println(x.name());`				

`Fruit x = new Apple();`
`System.out.println(x.name());`				

`Apple x = new Apple();`
`System.out.println(x.name());`				

`Food x = new Apple();`
`x = new Orange();`				

`Apple x = new Apple();`
`x = new Orange();`				

`Apple x = new Apple();`
`x = (Orange)new Orange();`				

`Apple x = new Apple();`				

x = (Apple)**new** Orange();				

Code	CE	RE	NE	Output

Apple x = **new** Apple();
System.out.println(x.slices());				

Apple x = **new** Apple();
Orange y = (Orange) x;				

Fruit x = **new** Apple();
Orange y = (Orange) x;				

Food x = **new** Apple();
x = **new** Orange();
Orange y = (Orange) x;				

Food x = **new** Orange();
System.out.println(x.name());				

Fruit x = **new** Orange();
System.out.println(x.name());				

Orange x = **new** Orange();
System.out.println(x.name());				

Food x = **new** Apple();
System.out.println(x.fruitName());				

Fruit x = **new** Apple();
System.out.println(x.fruitName());				

Orange x = **new** Apple();
System.out.println(x.fruitName());				

Food[] foods = **{ new** Apple(), **new** Orange() };
System.out.println(foods[0].isRed());				

Food[] foods = **{ new** Apple(), **new** Orange() };
System.out.println(foods[1].isRed());				

Food[] foods = **{ new** Apple(), **new** Orange() };
Apple x = foods[0];
System.out.println(x.isRed());				

Food[] foods = **{ new** Apple(), **new** Orange() };
Orange y = (Orange)foods[0];
System.out.println(y.slices());				

`Food[] foods = { new Apple(), new Orange() };`
`Orange y = (Orange)foods[1];`
`System.out.println(y.slices());`				

3. Data structure design

Part A - Initial design and implementation

You are asked to create a new data structure called `IntQueue` that represents a queue of integers. It is not important to be super familiar with queues to do well on this problem. A queue is a list where you can only add to the end of the list and you can only remove from the beginning of the list. It is similar to how people wait in line at the coffee shop. New customers are added to the end of the line, while the customer at the front of the line is served next by the worker.

Suppose we have the following starter code for implementing the IntQueue using an ArrayIntList.

```
public class IntQueue {
    private IntList list;  // Stores the queue elements in front-to-back order.

    public IntQueue() {
        this.list = new ArrayIntList();
    }
}
```

Fill in the methods below to finish the implementation. **You may assume that all the methods available on Java Lists (shown on the reference sheet) are available on ArrayIntLists and IntLists.** (Since there is ambiguity between remove(index) and remove(value), you can name them removeIndex and removeValue.)

```
// Add the value to the back of the queue.
public void addBack(int value) {






}
```

```
// Remove and return the value at front of queue. Assume queue is non-empty.
public int removeFront() {






}
```

What is the running time for each method? Mark an X in the appropriate column for each row.

Operation	**O(1)**	**O(log(n))**	**O(*n*)**	**O(n^2)**
`addBack`				
`removeFront`				

Part B: Alternate designs

Your senior engineer has an idea to speed up the IntQueue. They want to store the values internally in the opposite order of how you did it on the previous page. Then, they want to change the implementations of the two methods `addBack` and `removeFront` so that clients will observe the same behavior.

Describe how we can adjust `addBack` and `removeFront` so that clients will observe the same behavior as before. (You don't need to write code. Just explain at a high level in 1-2 sentences.)

How will this change affect the runtime of the two IntQueue operations? Mark an X in the appropriate box.

Operation	**O(1)**	**O(log(n))**	**O(*n*)**	**O(n^2)**
`addBack`				
`removeFront`				

Your senior engineer's boss has a fancy title containing the word "architect". You're not so sure what buildings have to do with queues, but the architect has a different idea for how to speed up the original IntQueue. They claim that if you replace the ArrayIntList with a LinkedIntList, then there will be a way to make both operations faster, either by storing the values in front-to-back order, or in the opposite order.

Is the architect correct in saying that using a LinkedIntList will allow the IntQueue to implement both `addBack` and `removeFront` to have faster complexity classes than your original ArrayIntList-based implementation? Why or why not?

The following sub-question is just for fun and will not affect your grade:

Do you think it's possible to implement IntQueue using an ArrayIntList or LinkedIntList, or by modifying one of those classes, so that both `addBack` and `removeFront` run in O(1) time? If so, how? If not, why not?

4. Linked Lists

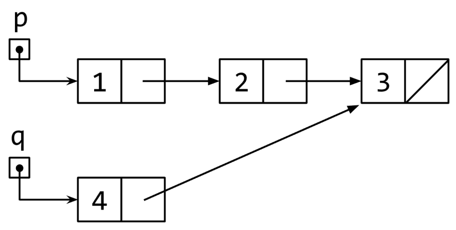
Part A - Tracing and Reference Semantics

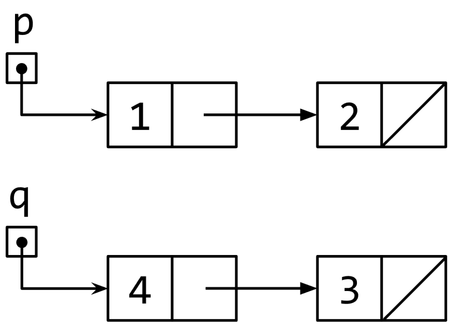
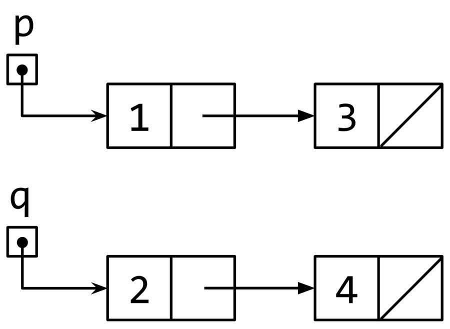
Complete the following table, filling in either the before picture, the code, or the after picture. You should not create any new `ListNodes` or modify any `.data` fields, and **there should be only one instance of each node with a specific value.**

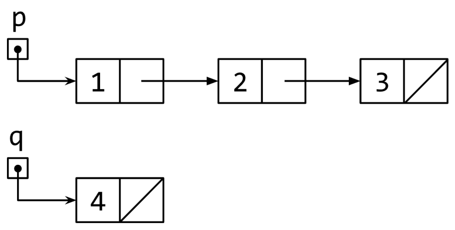
Your drawings must show all temporary references created by the code.

Your `ListNode` diagram format doesn't have to match that of the problem so long as it is clear what you intend. In code that you write, you may use as many temporary references as you'd like to accomplish your goal, even if the final picture does not show any temporary references.

Before	Code	After

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

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

Part B - Debugging

Consider adding this method of the LinkedIntList class:

```
public void mystery(int x) {
    ListNode node = front;
    while (node != null) {
        if (node.next == null) {
            node.next = new ListNode(x);
        }
        node = node.next;
    }
}
```

What do you think the author of this code was trying to do? What does the code actually do?

Fix the code so that it does what the author intended. **Add at most 6 lines. Do not change or remove any lines.** You can either write out the code again or explain where to make the changes. Do not use recursion.

5. Recursive Backtracking

Write a method findSentences with the following specification:

```
// Prints out all sentences consisting of some combination of the given words
// subject to the following rules:
//   - Each sentence is printed on its own line.
//   - A sentence is made up of words separated by a single space (" ").
//   - There is a period (".") at the end of the sentence.
//   - Each word is used at most once. (A word is allowed not to be used at all.)
//   - The total number of characters in the sentence (including spaces and periods)
//     is between minLength (inclusive) and maxLength (inclusive).
public void findSentences(List<String> words, int minLength, int maxLength)
```

For example, consider the following client code:

```
List<String> words = new ArrayList<String>();
words.add("hi");
words.add("world");
words.add("thee");
words.add("Rasmussen");

findSentences(words, 10, 13);
```

A correct implementation will print the following lines. (The lines are allowed to be printed in any order.)

```
hi Rasmussen.
world thee.
thee world.
Rasmussen.
Rasmussen hi.
```

Note:

- "hi world." is not printed because it is too short. (It has 9 characters including spaces and periods, which is less than the minimum of 10 in this example.)
- "hi thee world." is not printed because it is too long. (It has 14 total characters including spaces and periods, which is longer than the maximum of 13 in this example.)
- "Rasmussen." is printed because it has exactly 10 characters (including the period), and 10 is the minimum length (inclusive) in this example.
- "world thee." is printed because it has exactly 11 characters (including the space and period), and 11 is between 10 (inclusive) and 13 (inclusive).
- "hi Rasmussen." is printed because it has exactly 13 characters (including the space and period), and 13 is the maximum length (inclusive).

Write your solution on the next page. Follow these additional rules:

- To get an S or better, your code must be primarily recursive.
  (But using iteration within the recursion *is* allowed.)
- To get an E, you must prune "dead ends" in the search tree when the sentence becomes too long.

```
public void findSentences(List<String> words, int minLength, int maxLength) {
```

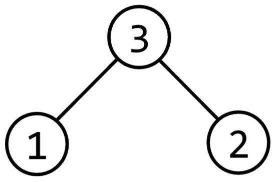
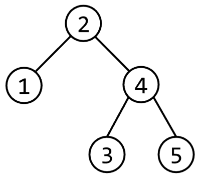
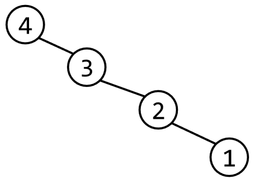
6. Binary Trees

Part A - Tracing

Consider the following code for a method inside the IntTree class:

```
public boolean mystery() {
    return mysteryHelper(overallRoot, Integer.MIN_VALUE, Integer.MAX_VALUE);
}
private boolean mysteryHelper(IntTreeNode currentRoot, int lo, int hi) {
    if (currentRoot == null) {
        return true;
    } else {
        return lo <= currentRoot.data &&
            currentRoot.data <= hi &&
            mysteryHelper(currentRoot.left, lo, currentRoot.data) &&
            mysteryHelper(currentRoot.right, currentRoot.data, hi);
    }
}
```

The constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE` are the smallest and largest possible integers representable by an int in Java. For any int x, the comparison `x >= Integer.MIN_VALUE` returns true.
Say what the method returns on the following trees:

Input tree			
Return value			

Explain what the mystery method computes.

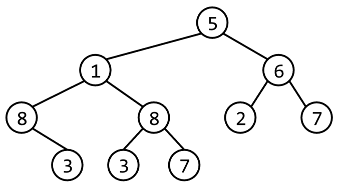
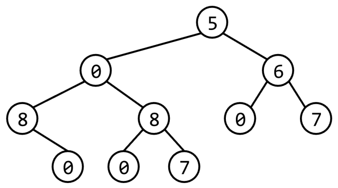
Part B - Modification

If x is a node in a binary tree, define the *depth* of x to be the number of edges on the path from the overall root to x. In the example "Before" tree on the left below, the nodes with data 8 have depth 2, and the node with data 6 has depth 1. The root node (the node with data 5 in the example below) has depth 0.

Write a method of the `IntTree` class with the following specification:

```
// Finds any nodes whose data is equal to that node's depth and replaces each such
// node with a new node that stores 0 as its data instead.
public void zeroDepthNodes()
```

For example, when this method is called on the "Before" tree, it changes it into the "After" tree.

Before	After	Notes
		Notice the four nodes that became 0. They each had a depth equal to their data. All other nodes were left unchanged.

Write your code below. You must use recursion. Remember that the data field is `final`.

```
public void zeroDepthNodes() {
```

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

`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.

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();
}
```

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

IntList and ArrayIntList

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

LinkedIntList

```
public class LinkedIntList implements IntList {
    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;
                                                }
                                            }
                                        }
```

`IntTree` Class

```
public class IntTree {
    private IntTreeNode overallRoot;

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

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

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