CSE 123 Autumn 2025 Practice 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. Comprehension

Part A - True/False

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

Clearly indicate your answer. Sneaky answers such as will be considered automatically incorrect.

Statement	**True/False**
In Big-Oh notation, only the dominating term matters for complexity because lower-order terms become insignificant for very large input sizes.	
Methods declared as `abstract` in an abstract class have a concrete implementation.	
In Java, you can `extend` multiple classes, but `implement` only one interface.	
Run time happens before compile time.	
`super`(<*methodName*>) is used to invoke the specified method with the name *methodName* of the superclass.	
You should always return -1, 0, or 1 from `compareTo` and not any other integers.	
If a subclass does not override a public superclass method, then it is an error to call that method on that subclass.	
An abstract class can have constructors	
An abstract class can be instantiated.	
The number of operations for a method with a runtime of $O(n^2)$ grows faster than the number of operations for a method with a runtime of $O(n)$ as the input size increases.	
Constructors are inherited	
Subclasses can directly access private fields of the superclass by calling `super.`<*fieldName*>	
You do not need to use the `@Override` when overriding methods from the superclass	
It is impossible to modify a binary tree without using x = change(x).	
Every iterative solution also has a recursive solution.	
Every recursive method needs a base case.	
You cannot call an abstract method while in an abstract class.	
Similar to interfaces, you must use the `implements` keyword with abstract classes.	

Part B - Runtime Analysis

Analyze the worst-case running time of each operation 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.

Operation	**O(1)**	**O(*n*)**	**O(*n*²)**

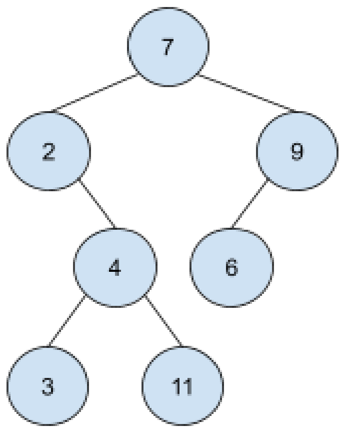
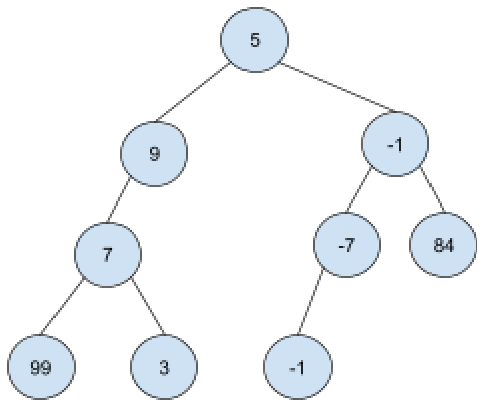
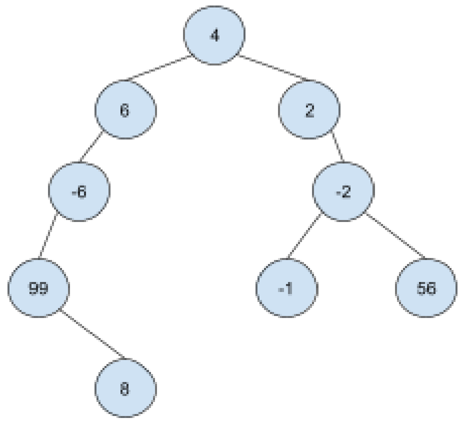
Running the method `m2` below (which invokes `m1`).
 	
`public void m1(int[] data) {`
`    for (int i = data.length - 1; i >= 0; i--) {`
`        System.out.println(data[i]);`
`    }`
`    for (int i = 0; i < data.length; i++) {`
`        System.out.println(data[i]);`
`    }`
`}`
`public void m2(int[] data) {`
`    for (int i = 0; i < data.length; i++) {`
`        m1(data);`
`    }`
`}`			

Running the method `m1` below.
 	
`public void m1(int[] data) {`
`    for (int i = 0; i < 50; i++) {`
`        for (int j = 0; j < i; j++) {`
`            System.out.println(data[j]);`
`        }`
`    }`
`}`			

Running the method `m1` below.
 	
`public void m1(int[] data) {`
`    for (int i = 0; i < data.length; i++) {`
`        for (int j = 0; j < i; j++) {`
`            if (i + j == 4) {`
`                i = data.length;`
`            }`
`            System.out.println(data[j])`
`        }`
`    }`
`}`			

Part C - Binary Tree Traversals

For each of the following binary trees, indicate which type of traversal is shown: pre-order, in-order, post-order, or none of these.

	2 3 4 11 7 9 6	☐ pre-order ☐ in-order ☐ post-order ☐ none
	99 3 7 9 -1 -7 84 -1 5	☐ pre-order ☐ in-order ☐ post-order ☐ none
	99 8 -6 6 4 2 -1 -2 56	☐ pre-order ☐ in-order ☐ post-order ☐ none

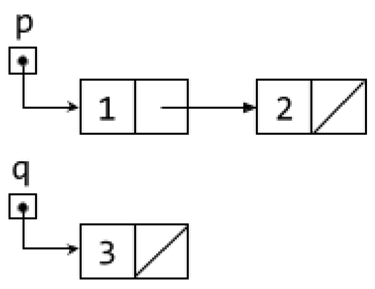
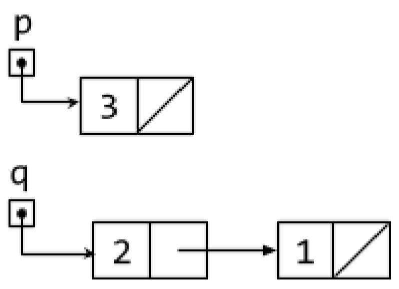
2. Code Tracing

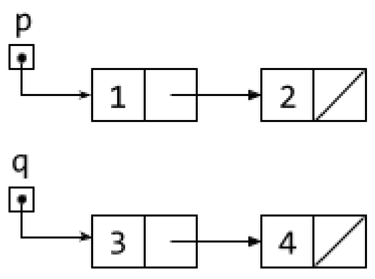
Part A: Linked Node Manipulation

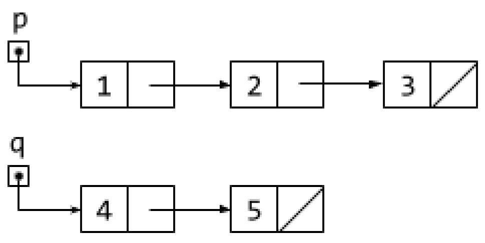
In the following table, the “Before” column shows a diagram of some linked nodes, the “Code” column specifies some code to be applied to the nodes in the before diagram, and the “After” column shows a diagram of the nodes after the code has been applied.

Complete the table, filling in either the before picture, the code, or the after picture. You should not create any new `ListNode` objects or modify any `.data` fields, and **there should be only one instance of each node with a specific value.** The after picture does not need to show any temporary references that were created.

Your `ListNode` diagram format doesn’t have to match that of the problem, so long as it is clear what you intend. In your code, you may use as many temporary references as you’d like to accomplish your goal, but you may *not* create any new `ListNode` objects.

Before	Code	After
		

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

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

Part B: Inheritance Tracing

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

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

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

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

```
public class Rectangle extends Shape {
    public void method3() {
        System.out.println("Rect 3");
        super.method3();
    }
}

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

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

Assume the following variables have been defined:

```
Shape var1 = new Rectangle();
Shape var2 = new Square();
Circle var3 = new Circle();
Rectangle var4 = new Square();
```

Now consider the following code. For each line, place an X next to the correct outcome. If the code runs without error, also indicate what output is printed by filling in the blank. If it produces no output, write "none". If the output includes multiple lines, you may indicate line breaks with a slash character "/" in the output.

`var1.method1();`		Compile-time error
		Run-time error
		Runs without error and prints: ______________________

`Square square1 = (Square) var2;`
`square1.method1();`
		Compile-time error

		Run-time error
		Runs without error and prints: ______________________

`var3.method1();`		Compile-time error
		Run-time error
		Runs without error and prints: ______________________

`var4.method2();`		Compile-time error
		Run-time error
		Runs without error and prints: ______________________

Part C: Recursive Tracing

Consider the following method:

```
public static void mystery(int n, List<Integer> list) {
    if (list.isEmpty()) {
        System.out.print(n + " ");
    } else if (list.size() > n) {
        System.out.print(list.remove(n) + " ");
    } else {
        int num = list.remove(0) / 2;
        mystery(num, list);

        list.add(num);
        mystery(n - 1, list);
    }
}
```

For each of the following statements, indicate what the output would be. For the sake of simplicity, we will represent lists using array notation.

`mystery(1, [10])`	
`mystery(2, [4, 20])`	

}

3. Data Structure Design

You are asked to create a new data structure called PriorityIntQueue that represents a priority queue of integers. A priority queue is like a queue (i.e., only allows adding and removing in first-in, first-out or "FIFO" order), but each item is added with a “priority,” and when an item is removed, the highest priority item is removed.

Since we are working with integers, we will consider the **smallest** integer as the highest priority item. For example, consider the following code snippet:

```
PriorityIntQueue pq = new PriorityIntQueue();
pq.add(1);
pq.add(3);
pq.add(2);
pq.remove();  // returns 1
pq.remove();  // returns 2
pq.remove();  // returns 3
```

Now, consider the following incomplete implementation of the `PriorityIntQueue`:

```
public class PriorityIntQueue {
     private IntList list; // the list of data used to store the priority queue

     public PriorityIntQueue() {
          this.list = // TODO: Which implementation should we use?
     }

     // adds an element to the queue
     public void add(int element) {
          // TODO: Implement add
     }

     // removes and returns the smallest integer
     public int remove() {
          if (this.list.isEmpty()) {
               throw new NoSuchElementException();
          }

          list.remove(0);
     }
}
```

Implement a working `add(int element)` so that when `remove()` is called, the smallest value is removed.

```
public void add(int element) {
```

What is the runtime of `remove()` if an `ArrayIntList` is used? What is the runtime if a `LinkedIntList` is used?

Likewise, what is the runtime of your `add(int element)` implementation if an `ArrayIntList` is used? What is the runtime if a `LinkedIntList` is used?

	ArrayIntList	LinkedIntList
remove()		
add(int element)		

Is one implementation more advantageous than the other? Support your answer by using the runtimes of each implementation. Try to limit your answer to at most five sentences.

4. LinkedList Programming

Consider a method in the `LinkedIntList` (see the reference sheet) class called `pairUp(int value, int target)` that adds a given value in front of all target numbers in the list. For example, if the original contents of the list were:

`list = [1, 7, 1, 1]`

Then, after running `list.pairUp(0, 1)`, the list's contents would be:

`list = [0, 1, 7, 0, 1, 0, 1]`

Part A: `LinkedList` with recursion

Below is a buggy recursive solution to the problem:

```
public void pairUp(int value, int target) {
    pairUp(this.front, value, target);
}
private void pairUp(ListNode curr, int value, int target) {
    if (curr != null) {
        pairUp(curr.next, value, target);
        if (curr.data == target) {
            ListNode curr = new ListNode(value, curr);
        }
    }
}
```

You decide that changing the return type from `void` to `ListNode` can help us solve the problem. Implement `pairUp` with the provided method designs below. You may not use any loops to solve this problem; you must use recursion.

```
public void pairUp(int value, int target) {

}
private ListNode pairUp(ListNode curr, int value, int target) {
```

Part B: `LinkedList` with iteration

Now implement it iteratively. You should not use any recursion.

```
public void pairUp(int value, int target) {
```

5. Recursive Programming

Write a **recursive** method `stutter` that accepts a `Stack` of integers as a parameter and replaces every value in the stack with two occurrences of that value. For example, suppose a stack stores these values:

```
bottom [3, 7, 1, 14, 9] top
```

Then the stack should store the following values after the method terminates:

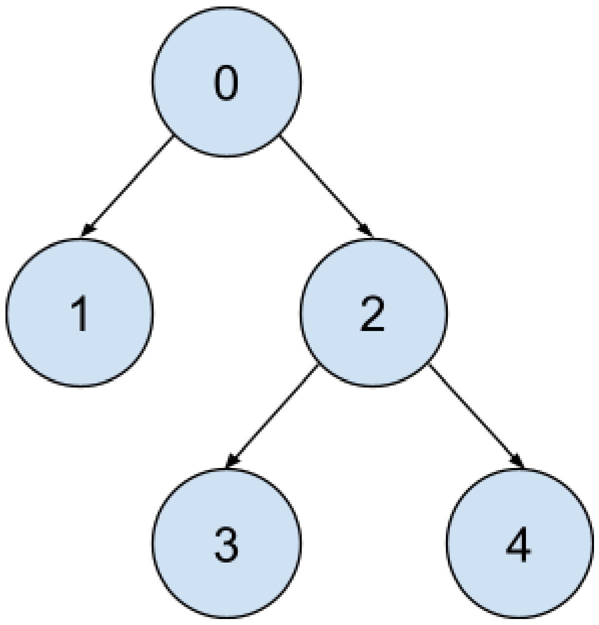
```
bottom [3, 3, 7, 7, 1, 1, 14, 14, 9, 9] top
```

Notice that you must preserve the original order. In the original stack, the 9 was at the top and would have been popped first. In the new stack, the two 9s would be the first values popped from the stack. Also, you must not use any auxiliary data structures to solve this problem. If the original stack is empty, the result should be empty as well. You may not use any loops to solve this problem; you must use recursion.

```java
public static void stutter(Stack<Integer> stack) {
```

6. Binary Tree Programming

Write a method called `isDescendant(int descendant, int ancestor)` to be added to the `IntTree` class (see the reference sheet). This method returns true if a node with data `descendant` is a descendant of a node with the data `ancestor`. A descendant is defined as a node reachable by repeated proceeding from parent to child. Note that a node can be considered its own descendant. Consider the following example:



A call to `isDescendant(4, 1)` or `isDescendant(4, 3)` would return false, since there is no way to reach 4 by going from parent to child. However, a call to `isDescendant(4, 0)`, `isDescendant(4, 2)`, and `isDescendant(4, 4)` would each return true.

You may assume that all values in the tree are **unique**. You may **not** assume that a node with data `descendant` and a node with data `ancestor` are contained within the tree.

You are writing a method that will become part of the `IntTree` class. You are free to create as many private helper methods to solve the problem, but you may not call any other method in the `IntTree` class. Lastly, you may not use any auxiliary data structures to solve this problem (e.g., no arrays, `Lists`, `Stacks`, `Queues`, `Strings`, etc.).

```
public boolean isDescendant(int descendant, int ancestor) {
```

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) {...}
    }
}
```