CSE 123 Spring 2025 Practice Final Exam

Name of Student: ____

Section (e.g., AA):____________________ Student UW **Number** :______________________________

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.
- If you require scratch paper, raise your hand and we will bring some to you.
- **We will only scan your exam packet**. We will not scan scratch paper or your notes sheet. We have included a blank page in the exam packet. If you run out of space while answering a question, write your answer on that blank page and clearly indicate that we should look for your answer there.
- 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.
- 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. 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.

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]);`
`  }`
`}`
			X	

`public void m(int[] data) {`
`  for (int i = 0; i < 2147483647; i++) {`
`    System.out.println(data[i]);`
`  }`
`}`
	X			

`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]);`
`  }`
`}`
				X

`public void m(int[] data) {`
`  for (int i = data.length - 1; i > 0; i /= 2) {`
`    System.out.println(data[i]);`
`  }`
`}`
		X		

Searching for a value in an array.			X	
Searching for a value in a linked list.			X	
Searching for a value in a binary tree.			X	
Searching for a value in a binary search tree.			X	
Searching for a value in a balanced binary search tree.		X		

2. Inheritance and Polymorphism

Consider the following classes:

```
public interface GardenTool {
    public boolean isWorking();
    public void operate();
}

public abstract class HandTool
  implements GardenTool {
    public boolean isWorking() {
        return true;
    }
}

public class Rake extends HandTool {
    public void operate() {
        System.out.println(“Tidying leaves.”);
    }
    public void steppedOn() {
        System.out.println("Whack!");
    }
}

public class Trowel extends HandTool {
    public operate() {
        System.out.println(“Digging...”);
    }
}
```

```
public abstract class PowerTool
  implements GardenTool {
    private double powerRemaining;
    private double powerPerUse;

    public PowerTool(double powerPerUse) {
        this.powerRemaining = 1.0;
        this.powerPerUse = powerPerUse;
    }

    public boolean isWorking() {
        return powerRemaining > 0;
    }

    public void usePower() {
        this.powerRemaining -= powerPerUse;
    }

    public void recharge() {
        this.powerRemaining = 1.0;
    }
}
```

Part A: Fill in the blanks in the `Lawnmower` class on the next page so that the client code shown below will produce the indicated output.

```
Lawnmower lm1 = new Lawnmower(0.2, 20);
Lawnmower lm2 = new Lawnmower(0.2, 50);
Lawnmower lm3 = new Lawnmower(0.5, 50);

for (int i = 0; i < 3; i++) {
    lm1.operate();                     // prints Vroom
    lm2.operate();                     // prints Vroom
    lm3.operate();                     // prints Vroom
}
System.out.println(lm1.isWorking());   // prints false because lm1 has no more capacity
System.out.println(lm2.isWorking());   // prints true
System.out.println(lm3.isWorking());   // prints false because lm3 has no more power
```

(continued on next page...)

```
public class Lawnmower extends PowerTool {
    // capacity in bag for more grass
    private int capacity;
    private int maxCapacity;

    public Lawnmower(double powerPerUse,
                     int capacity) {
        super(powerPerUse);
        this.maxCapacity = capacity;
        this.capacity = capacity;
    }
    public boolean isWorking() {
      return super.isWorking() && capacity > 0;
    }

    public void operate() {
        if (this.isWorking()) {
            this.usePower();
            this.capacity -= 10;
        }
        System.out.println("Vroom");
    }

    public void empty() {
        this.capacity = maxCapacity;
        System.out.println("Bag emptied!");
    }
}
```

Part B: 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

`HandTool x = new Rake();`
`x.operate();`
			X	Tidying Leaves.

`PowerTool x = new LawnMower(0.1, 40);`
`System.out.println(x.isWorking());`
			X	true

`PowerTool x = new Trowel();`
`x.operate();`
	X			

`HandTool x = new LawnMower(0.5, 100);`
`x.empty();`
	X			

`HandTool x = new Trowel();`
`System.out.println(x.isWorking());`
			X	true

`PowerTool x = new LawnMower(0.33, 10);`
`((LawnMower) x).operate();`
			X	Vroom

`Trowel x = new Trowel();`
`((Rake) x).operate();`
	X			

`HandTool x = new Trowel();`
`((Rake) x).steppedOn();`
		X		

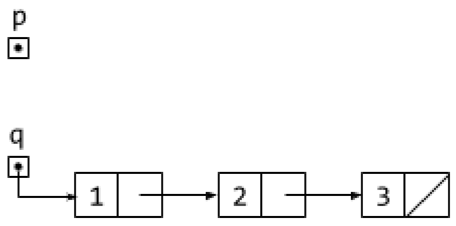
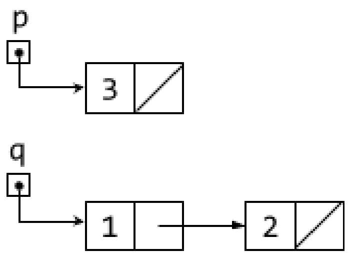
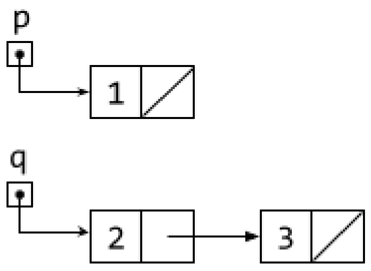
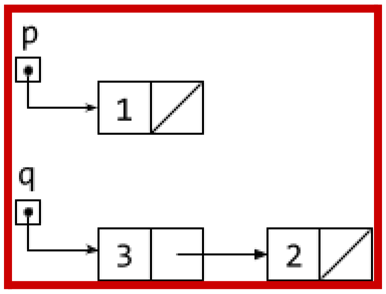
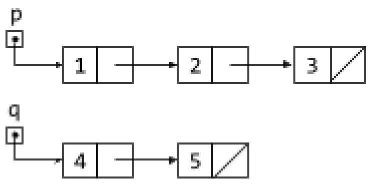
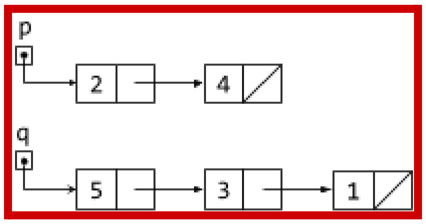
3. Linked Lists

Part A: Reference Semantics

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

Part B: Linked List Implementation
Implement the method `set(int index, int element)`, which changes the value at the given index to be the given element.

For example, suppose the variable `list1` contains a reference to the following list:

```
[4, 8, 15, 16, 23, 42]
```

Then after the call `list1.set(2, 10)` executes, `list1` should contain a reference to this list:

```
[4, 8, 10, 16, 23, 42]
```

Then, After the call `list1.set(0, 10)` executes, `list1` should contain a reference to this list:

```
[10, 8, 10, 16, 23, 42]
```

Provide an implementation of `set` for the `LinkedIntList` class. Remember that the `data` field of `ListNode` is final! Assume that the only field that the `LinkedIntList` class has is `front`. Assume that the parameter `index` is always in bounds. Your implementation may be either recursive or iterative, whichever is your preference!

```
public void set(int index, int element){
    front = set(index, element, front);
}

private ListNode set(int index, int element, ListNode curr){
    if(index == 0){
        return new ListNode(element, curr.next);
    }
    curr.next = set(index-1, element, curr.next);
}

//iterative

public void set(int index, int element){
    if(index == 0){
        front = new ListNode(element, front.next);
    } else{
        ListNode curr = front;
        while(index > 1){
            curr = curr.next;
            index -= 1;
        }
        curr.next = new ListNode(element, curr.next.next);
    }
}
```

4. Recursion Debugging

Consider a method class called **`printSeq(List<String> list, int n)`** that prints all sequences of strings in `list` that are of length `n`. For example, suppose the contents of **`list`** are:

```
list = [“A”, “B”, “C”]
```

Then, after a call to **`printSeq(list, 2)`** is made, the following 6 lines should be printed:

```
[A, B]
[A, C]
[B, A]
[B, C]
[C, A]
[C, B]
```

If the length of **`list`** is less than **`n`**, the code should throw an **`IllegalArgumentException`**.

Consider the following incorrect implementation of **`printSeq`**:

```
public static void printSeq(List<String> strs, int n){
    if (strs.size() < n) {
        throw new IllegalArgumentException();
    }
    printSeq(strs, n, new ArrayList<String>());
}


private static void printSeq(List<String> strs, int n, List<String> curr){
    if (curr.size() == n) {
        System.out.println(curr);
    } else {
        for (int i = 0; i < strs.size(); i++) {
            String s = strs.remove(i);
            curr.add(s);
            printSeq(strs, n, curr);
            curr.remove(curr.size()-1);
            strs.add(s);
        }
    }
}

```

(Continued on following page)

Part A: When reviewing this implementation, you discover that the code contains a bug that is causing it to not work as intended. Specifically, you find that some sequences are not being printed, and others are being printed multiple times! You decide that you want to write a test that exposes the incorrect behavior. Provide contents for `list` and n, then identify one sequence that should be printed but is not, as well as one sequence that is printed more than once.

`list =` ["A", "B"]   `n =` 1

```
printSeq(list, n);
```

Sequence that should be printed, but is not:

["B"]

Sequence that is printed more than once:

["A"]

Part B: You discover that the bug actually only requires a change to line 18! Fill in the following solution with the fix that would make the solution work on the test case above.

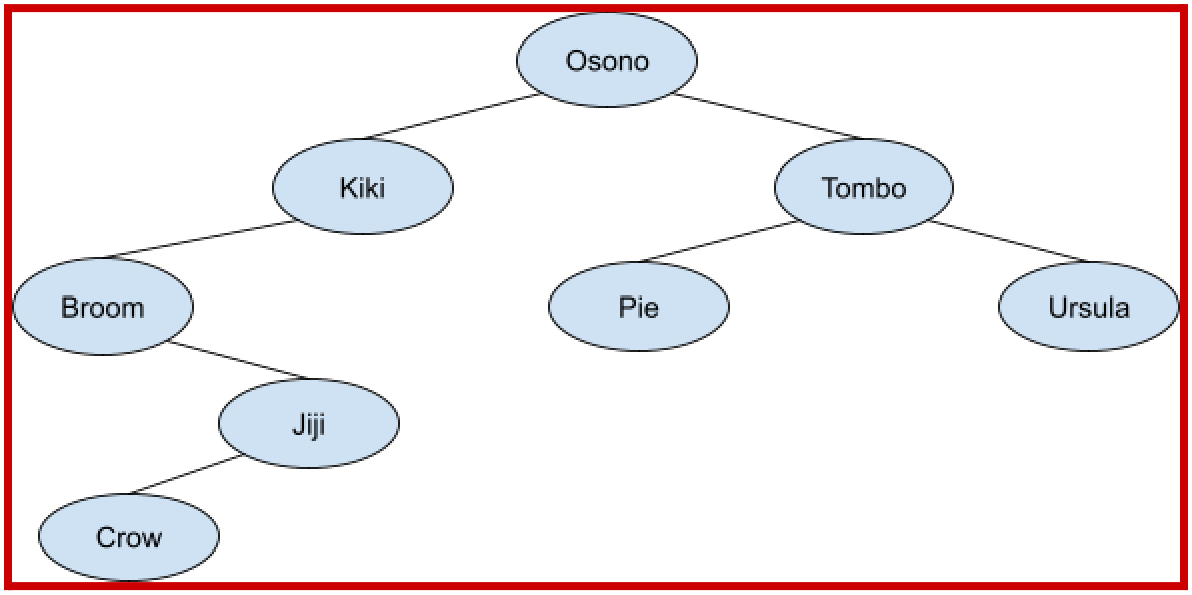
```
1  public static void printSeq(List<String> names, int n) {
2      if (strs.size() < n) {
3          throw new IllegalArgumentException();
4      }
5      printSeq(names, n, new ArrayList<String>());
6  }
7
8
9  private static void printSeq(List<String> strs, int n, List<String> curr) {
10     if (curr.size() == n) {
11         System.out.println(curr);
12     } else {
13         for (int i = 0; i < strs.size(); i++) {
14             String s = strs.remove(i);
15             curr.add(s);
16             printSeq(strs, n, curr);
17             curr.remove(curr.size()-1);
18             ~~strs.add(s);~~

19             strs.add(i,s);
20         }
21     }
22 }
```

5. Binary (Search) Tree Comprehension

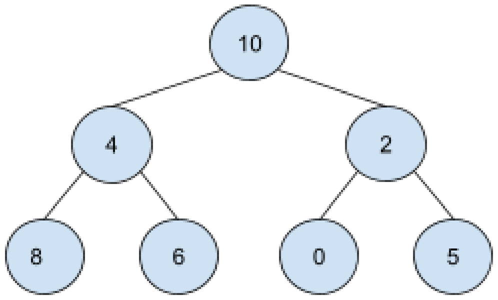
Part A: BST Insertion

Draw a binary search tree that would result if these elements were added to an empty tree in the following order, where the nodes are ordered alphabetically:

```
Osono, Kiki, Broom, Tombo, Pie, Ursula, Jiji, Crow
```



Part B: **Traversals**
For this question you will reference the binary tree below:



We will consider the contents of the call stack while printing the tree's elements using each of a pre-order, in-order, and post-order traversal. Specifically, in this problem you will identify the contents of the call stack at the point <u>when the value 4 is printed</u> by completing the table below.

For each row, we give an element of the tree. Each column represents a traversal type. To complete the table you need to identify whether for that traversal (at the time 4 is printed):

A. A stackframe in which `currentRoot` holds that element has **not yet been put on** the call stack
B. A stackframe in which `currentRoot` holds that element is **currently on** the call stack
C. A stackframe in which `currentRoot` holds that element has been **removed from** the call stack

Write the letter of the correct choice in the corresponding cell.

Element	Pre-Order	In-Order	Post-Order
10	**B**	**B**	**B**
8	**A**	**C**	**C**
6	**A**	**A**	**C**
2	**A**	**A**	**A**

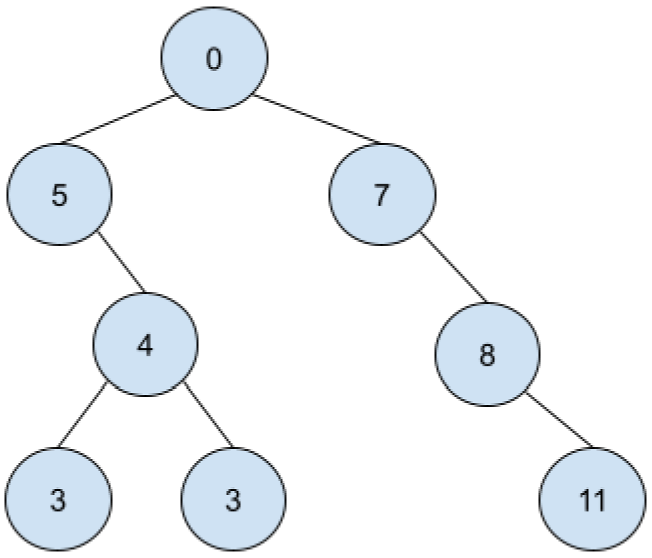
Part C: Recursive Method Tracing
Consider the following method in the `IntTree` class:

```
public int mystery(int n) {
    return mystery(overallRoot, n);
}

private int mystery(IntTreeNode curr, int n) {
    if (curr == null) {
        return 0;
    } else if (n == 0) {
        return 1;
    } else {
        return mystery(curr.left, n - 1) + mystery(curr.right, n - 1);
    }
}
```

Draw a binary tree such that, if it were stored in the variable `tree`, the call `tree.mystery(3)` would return 3.

Any tree with exactly three nodes at depth 3 (where the root is at depth 0).

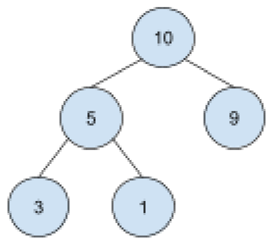
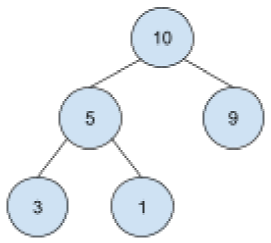
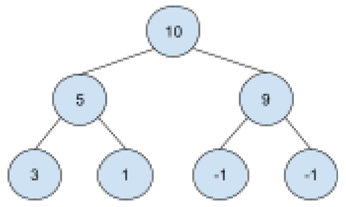
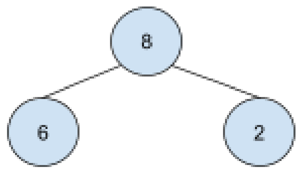
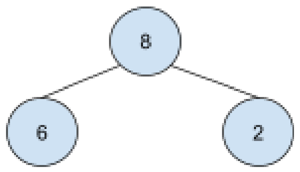
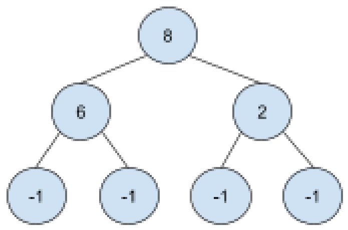
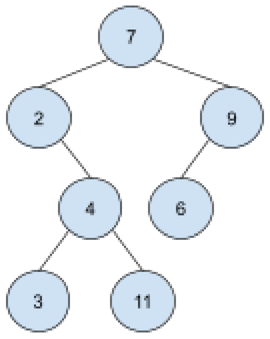
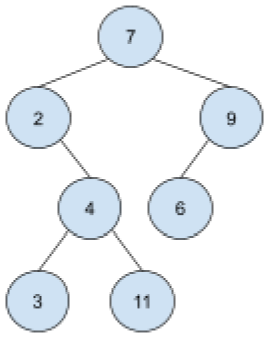
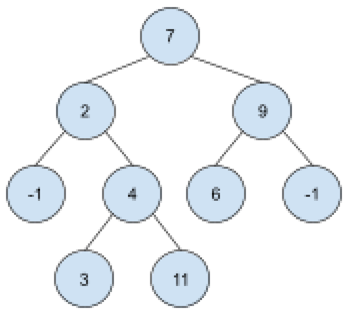
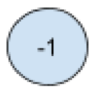
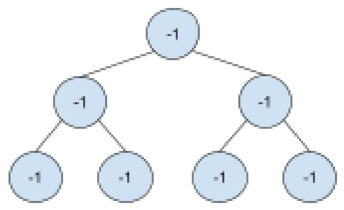
One example:



6. Binary Tree Programming

Write a method called **`fillToDepth`** to be added to the `IntTree` class (see the reference sheet). This method should take a single integer parameter, `depth`, and should modify a tree to be full up to the specified depth (that is, so that all nodes at a depth less than `depth` have exactly two children). All nodes added should contain the value -1. The *depth* of a node in a binary tree is the distance from the root to that node. So, for example, the root has a depth of 0, its children have a depth of 1, those nodes children have a depth of 2, and so on.

The following table shows the results of some example calls to `fillToDepth`.

Original Tree	**`tree.fillToDepth(0)`**	**`tree.fillToDepth(2)`**
		
		
		
empty tree		

Notice that in some cases the tree is not modified (if the tree is already full to the specified depth). Notice also that in some cases, multiple layers of new nodes may be needed. You may assume that `depth` is greater than or equal to zero.

Write your solution on the next page.

Write your solution to problem #6 here:

Two possible solutions:

```
public void fillToDepth(int depth) {
    overallRoot = fillToDepth(depth, overallRoot, 0);
}

private IntTreeNode fillToDepth(int depth, IntTreeNode root, int currDepth) {
    if (currDepth <= depth) {
        if (root == null) {
            root = new IntTreeNode(-1);
        }

        root.left = fillToDepth(depth, root.left, currDepth + 1);
        root.right = fillToDepth(depth, root.right, currDepth + 1);
    }
    return root;
}
```