CSE 123 Winter 2025 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.
- 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.

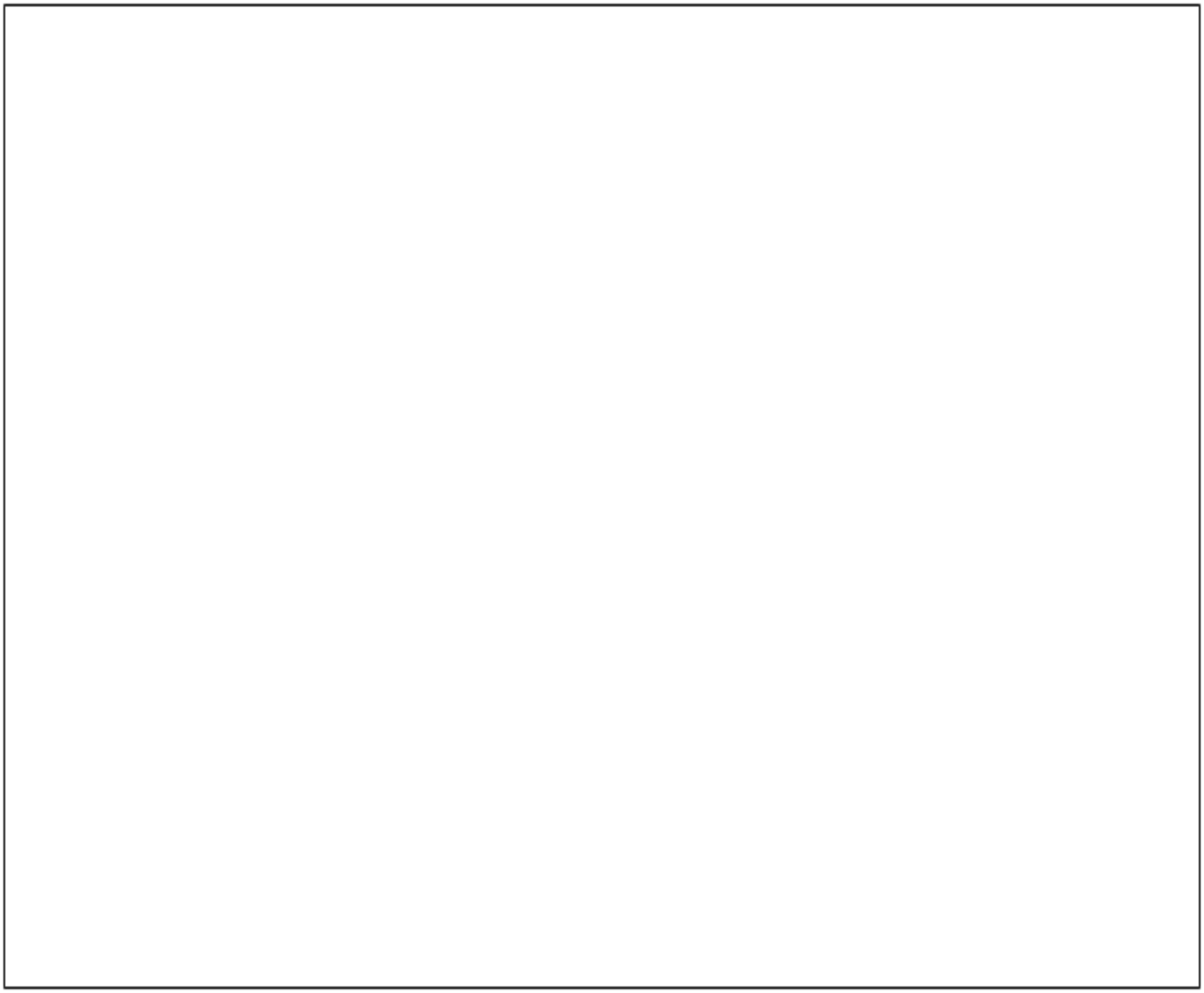
Initial here to indicate you have read and agreed to these rules: 

This page intentionally left blank
Nothing written on this page will be graded

1. Comprehension

Part A: 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
```



Based on the tree you drew above, write the sequence of elements that would be visited by each kind of traversal:

Pre-order: _____

In-order: _____

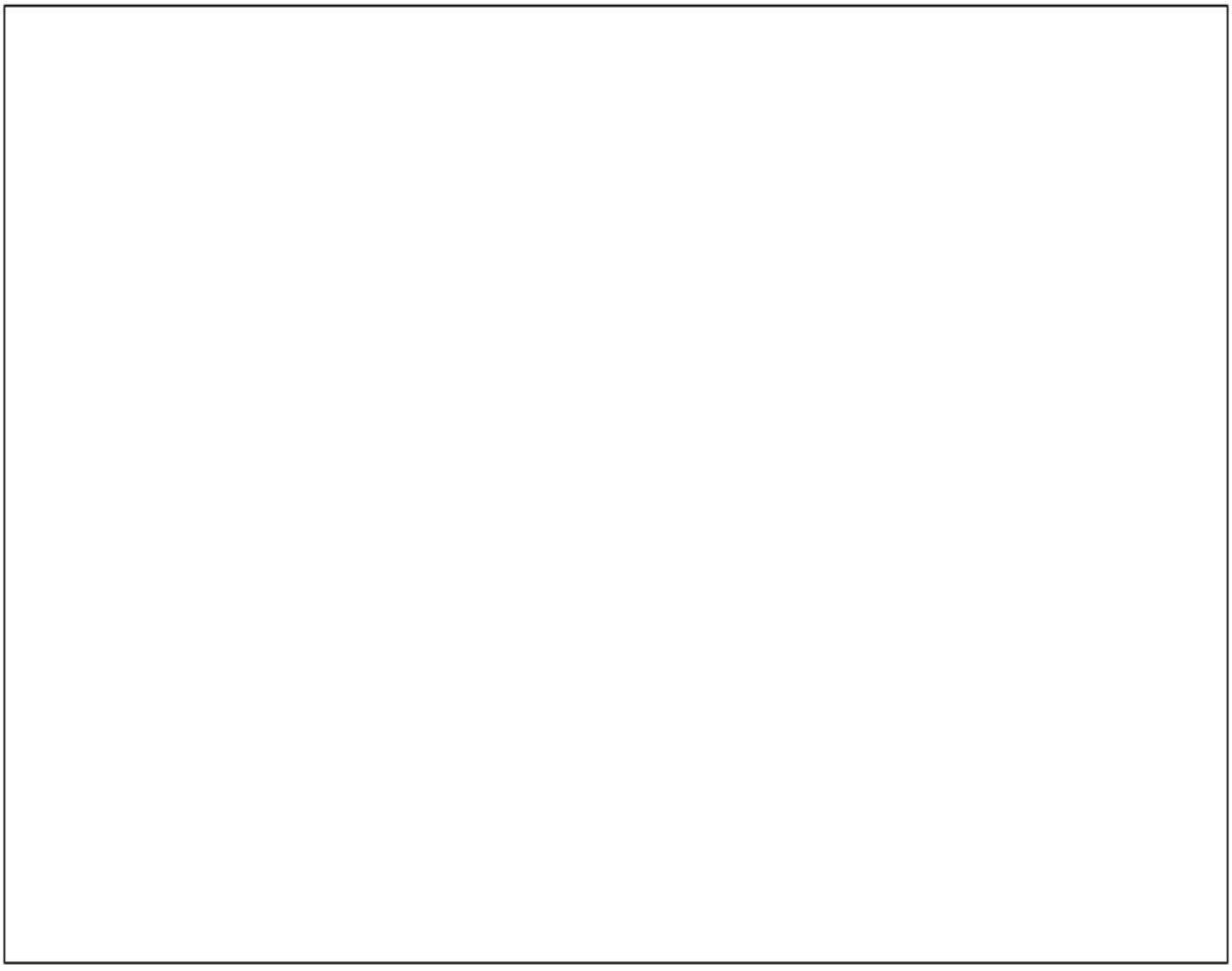
Post-order: _____

Part B: 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.



Part C:

Write a method that takes an `int[]` as a parameter that runs in ***O(1)*** time. (The exact functionality of the method is up to you.)

Write a method that takes an `int[]` as a parameter that runs in ***O(n)*** time where **n** is the length of the parameter array. (The exact functionality of the method is up to you.)

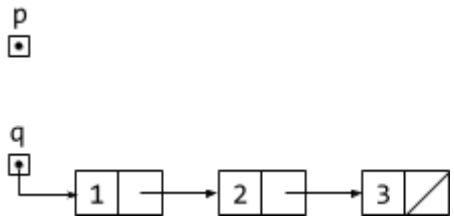
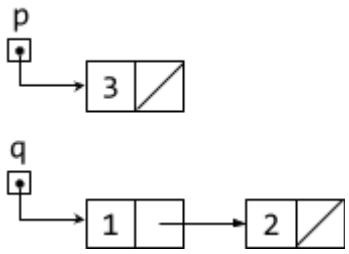
Write a method that takes an `int[]` as a parameter that runs in ***O(n^2)*** time where **n** is the length of the parameter array. (The exact functionality of the method is up to you.)

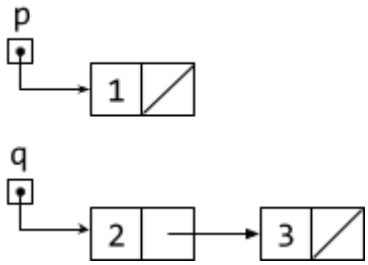
2. Code Tracing

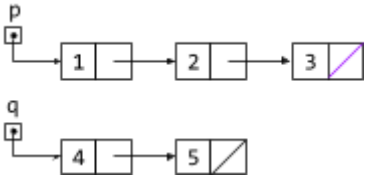
Part A: In the following table, the “Before” column shows a diagram of some linked nodes, the “Code” column specifies some code to be applied to the nodes in the before diagram, and the “After” column shows a diagram of the nodes after the code has been applied.

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

	`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: Consider the following method:

```
public void mystery(int n) {
    if (n <= 1)
        System.out.print(n);
    else {
        mystery(n / 2);
        System.out.print(", " + n);
    }
}
```

For each of the following statements, indicate what the output would be.

`mystery(8)`

`mystery(40)`

`mystery(25)`

3. 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 _______________ {
    // capacity in bag for more grass
    private int capacity;
    private int maxCapacity;

    public Lawnmower(double powerPerUse,
                     int capacity) {
        _________________________;
        this.maxCapacity = capacity;
        this.capacity = capacity;
    }
    public boolean isWorking() {
        return _____________________________;    }
    }
```

```
    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();`				
`PowerTool x = new LawnMower(0.1, 40);` `System.out.println(x.isWorking());`				
`PowerTool x = new Trowel();` `x.operate();`				
`HandTool x = new LawnMower(0.5, 100);` `x.empty();`				
`HandTool x = new Trowel();` `System.out.println(x.isWorking());`				
`PowerTool x = new LawnMower(0.33, 10);` `((LawnMower) x).operate();`				
`Trowel x = new Trowel();` `((Rake) x).operate();`				
`HandTool x = new Trowel();` `((Rake) x).steppedOn();`				

4. Linked List Debugging

Consider a method in the `LinkedIntList` class called **`limitTo`** that takes two integer parameters, min and max, and ensures all values in the linked list are between `min` and `max` (inclusive) by removing all elements that are outside that range.

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

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

Then after the call `list1.limitTo(10, 20)` executes, `list1` would contain a reference to this list:

```
[15, 16]
```

Similarly, suppose the variable `list2` contains a reference to the following list:

```
[8, 6, 7, 5, 3, 0, 9]
```

Then after the call `list2.limitTo(3, 6)` executes, `list2` would contain a reference to this list:

```
[6, 5, 3]
```

As a third example, suppose the variable `list3` contains a reference to the following list:

```
[10, 5, 4, 23, 9, 36, 0, 4]
```

Then after the call `list3.limitTo(100, 200)` executes, `list3` would contain a reference to this list:

```
[]
```

On the next page is a buggy implementation of `limitTo` that does not work as intended. In this implementation, given the original `list1`, `list2`, and `list3` above, `list1.limitTo(10, 20)` would result in `list1` containing `[15, 16, 42]`; `list2.limitTo(3, 6)` would result in `list2` containing `[6, 5, 3, 9]`; however `list3.limitTo(100, 200)` would work correctly.

(continued on next page...)

Part A: Provide a list and corresponding call to `limitTo` *other than the three given above* that would trigger the bug, along with the result that would be produced by the buggy implementation and the expected result if the method were implemented correctly.

Input list:	
Method call:	
Buggy result:	
Correct result:	

Part B: Annotate (write on) the code below to indicate how you would fix the bug. You may add (using arrows to indicate where to insert), remove (by crossing out), or modify (with a combination) any code you choose. Be sure to clearly indicate where you would add/remove/change code in addition to what changes you would make. (Incorrect or incomplete attempted fixes in the correct place may still be eligible for an S.)

```
1       public void limitTo(int min, int max) {
2           while (front != null && (front.data < min || front.data > max)) {
3               front = front.next;
4           }
5
6           ListNode curr = front;
7           while (curr != null && curr.next != null) {
8               if (curr.next.data < min || curr.next.data > max) {
9                   curr.next = curr.next.next;
10              }
11              curr = curr.next;
12          }
13      }
```

5. Recursive Backtracking

Write a *recursive* method called **`rollNoDuplicates`** that takes two integer parameters, `sides` and `rolls`, and computes and prints out all the ways to roll a die a certain number of times *without* rolling the same number twice. Your method should simulate rolling a die that can show the numbers 1 through `sides`, and should simulate rolling `rolls` times.

For example, the call `rollNoDuplicates(4, 3)` would produce the following output:

```
[1, 2, 3]
[1, 2, 4]
[1, 3, 2]
[1, 3, 4]
[1, 4, 2]
[1, 4, 3]
[2, 1, 3]
[2, 1, 4]
[2, 3, 1]
[2, 3, 4]
[2, 4, 1]
[2, 4, 3]
[3, 1, 2]
[3, 1, 4]
[3, 2, 1]
[3, 2, 4]
[3, 4, 1]
[3, 4, 2]
[4, 1, 2]
[4, 1, 3]
[4, 2, 1]
[4, 2, 3]
[4, 3, 1]
[4, 3, 2]
```

Notice that the order in which the values are rolled matters– rolling [1, 2, 3] is different from rolling [1, 3, 2] or rolling [3, 2, 1]. You may assume that `sides` and `rolls` are both greater than 0.

You may print the possibilities in any order, but to earn an E, you must print all possibilities and you must not print any possibility more than once. To earn a grade other than N, your method **must** be implemented recursively, though you may also use loops as part of your recursive algorithm.

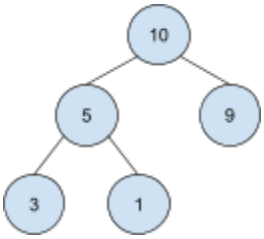
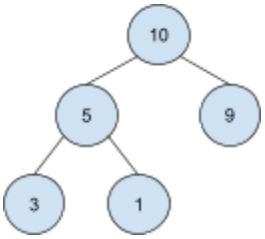
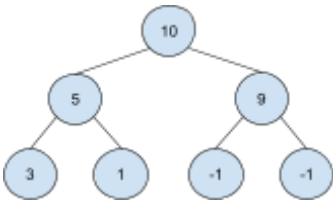
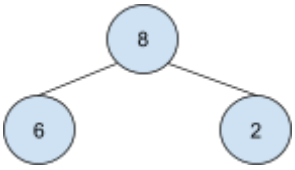
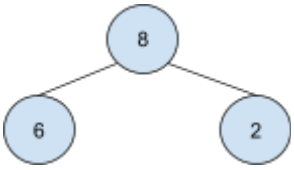
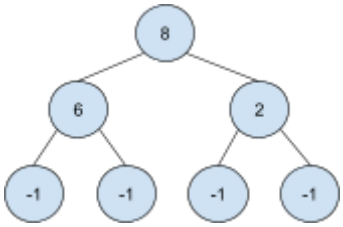
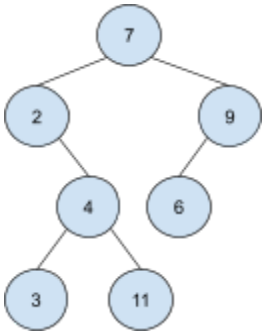
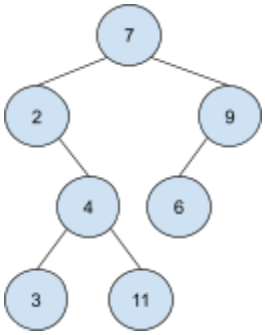
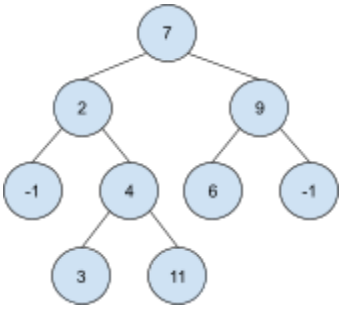
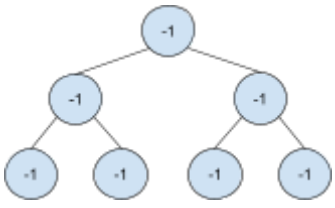
Write your solution on the next page.

Write your solution to problem #5 here:

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 or equal to `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:

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.)