CSE 123 Practice Final Exam Key

Section (e.g., AA):____________________ Student Number: ________________________

Do not turn the page until you are instructed to do so.

Name of Student: ***KEY***

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 result in a penalty.
- 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 result in a penalty.
- 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 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:

- Each problem will receive a single E/S/N grade.
  - On problems 1 through 3, earning an E requires answering all parts correctly and earning an S requires answering almost all parts correctly.
  - On problems 4 through 6, earning an E requires an implementation that meets all stated requirements and behaves exactly correctly in *all* cases. Earning an S requires an implementation that meets all stated requirements and behaves exactly correctly in *most* cases or behaves nearly correctly in *all* cases.
- 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.

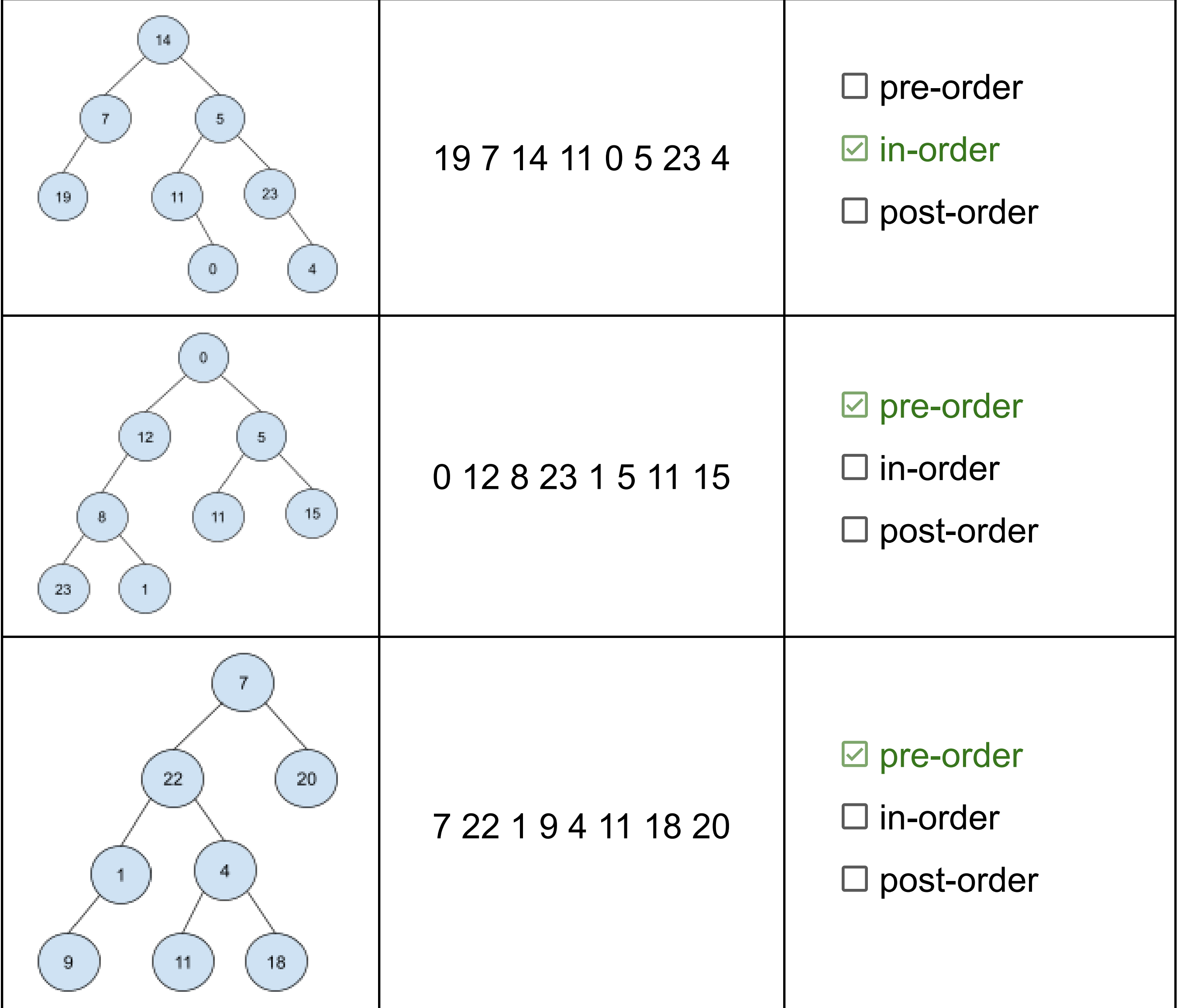
Advice:

- Read all questions carefully. Be sure you understand the question *before* you begin your answer.
- The questions are not necessarily in order of difficulty. Feel free to skip around. Be sure you are able to 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. Code Comprehension

Part A: For each of the following binary trees, indicate which type of traversal is shown: pre-order, in-order, or post-order.



Tree	Traversal	Type
14; 7, 5; 19, 11, 23; 0, 4	19 7 14 11 0 5 23 4	☐ pre-order ☑ in-order ☐ post-order
0; 12, 5; 8, 11, 15; 23, 1	0 12 8 23 1 5 11 15	☑ pre-order ☐ in-order ☐ post-order
7; 22, 20; 1, 4; 9, 11, 18	7 22 1 9 4 11 18 20	☑ pre-order ☐ in-order ☐ post-order

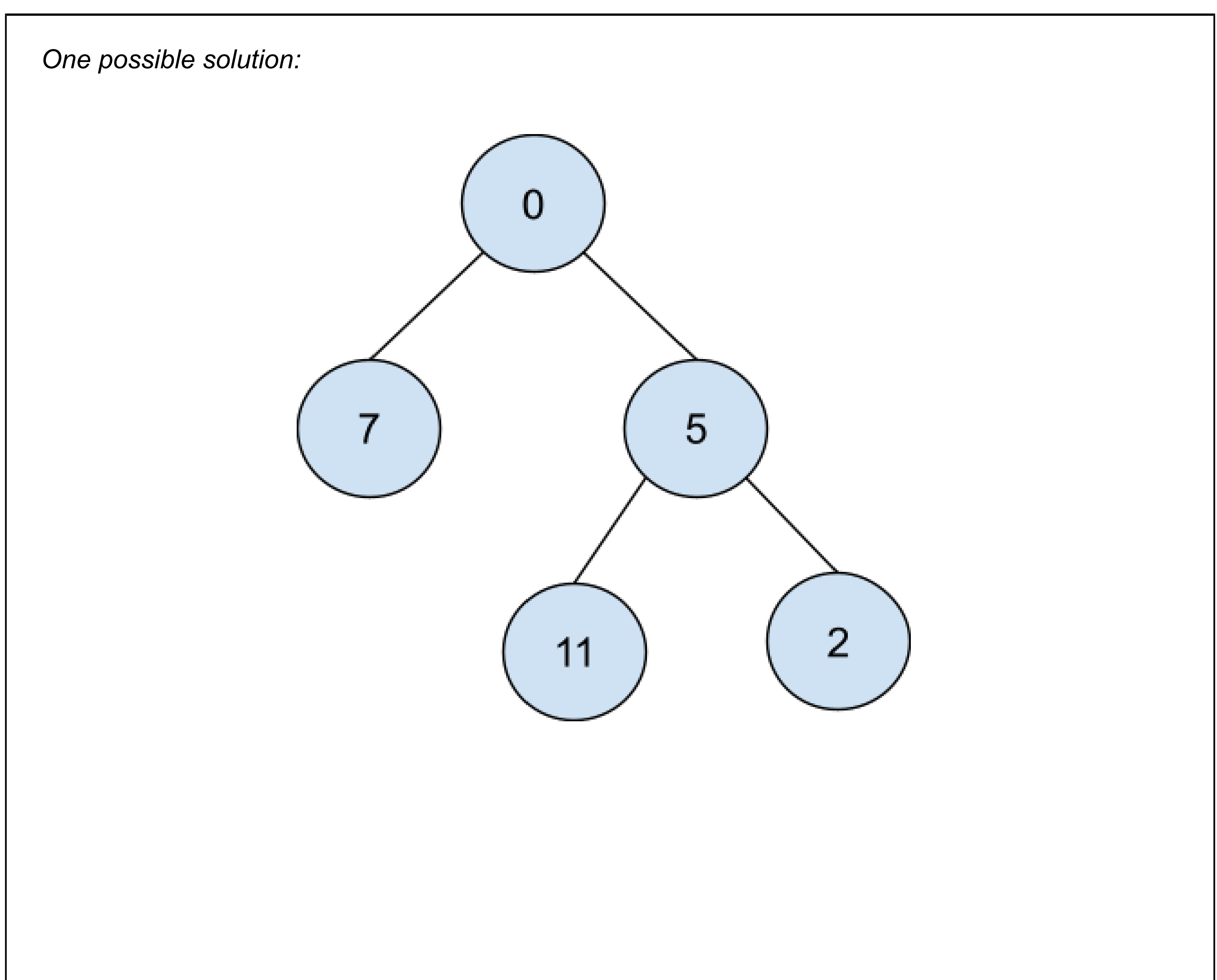
Part B: Which of these statements are true? (Select all that apply.)

- ☑ Abstract classes cannot be instantiated.
- ☐ Each class can extend only one base class and implement only one interface.
- ☑ A class can call its base class's method using the `super` keyword.
- ☑ If `ClassC` extends `ClassB` and `ClassB` extends `ClassA`, then an instance of `ClassC` can be passed as a parameter to a method that expects an instance of `ClassA`.

Part C: Consider the following method in the `IntTree` class:

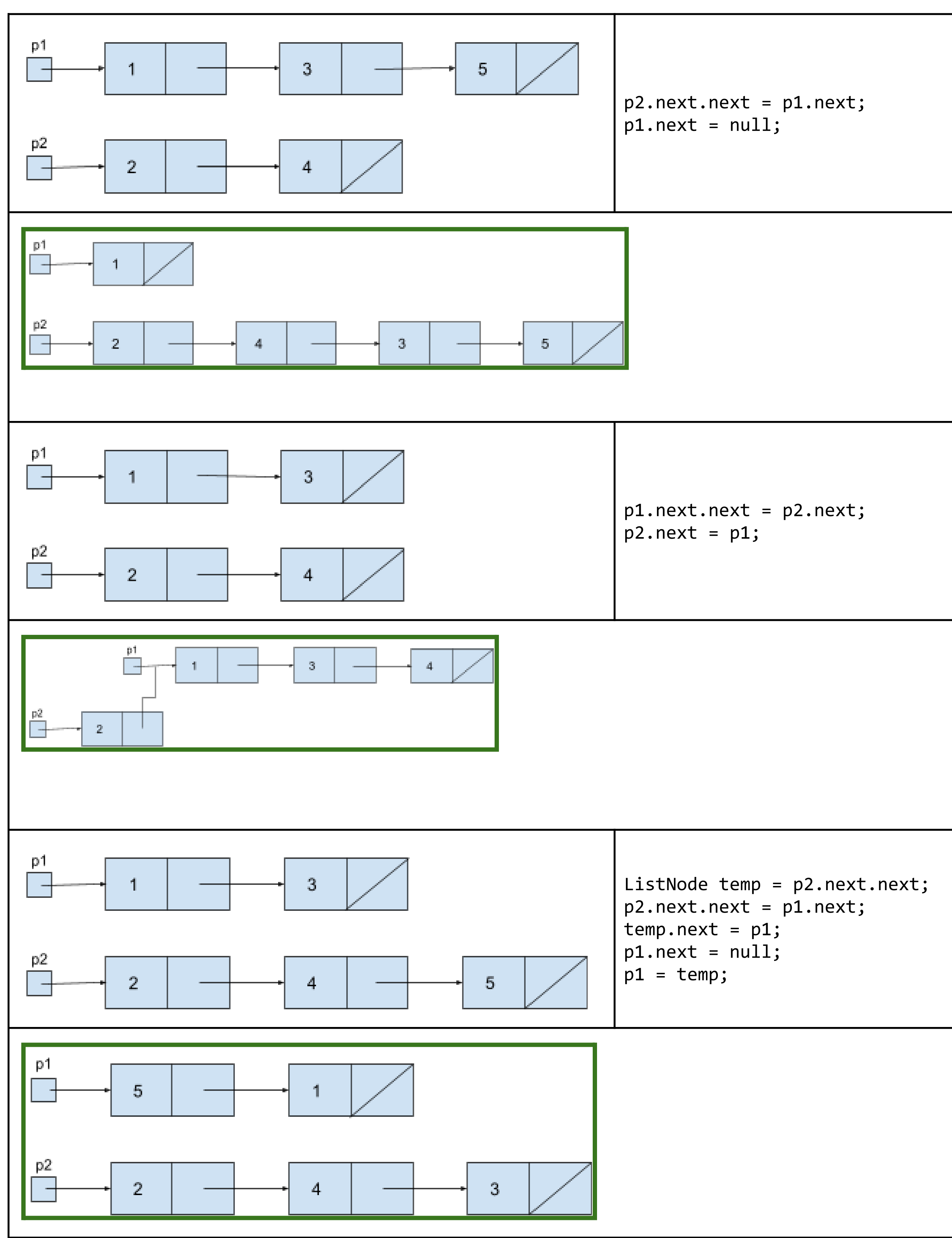
```
public int mystery(int limit) {
      return mystery(limit, overallRoot);
}

private int mystery(int limit, IntTreeNode root) {
      if (root == null) {
            return 0;
      }

      int result = 0;
      if (root.data <= limit) {
            result = 1;
      }

      result += mystery(limit, root.left);
      result += mystery(limit, root.right);
      return result;
}
```

Draw a binary tree such that, if it were stored in the variable `tree`, the call `tree.mystery(10)` would return 4.



2. Code Tracing

Part A: For each of the following, draw the linked lists that result from starting with the lists shown on the right and executing the code provided.



Part B: Consider the following classes:

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

For each of the following statements, Indicate what the output would be. If the statement would result in an error, write “error” instead. (You may use a slash to indicate line breaks. For example, “line1/line2” indicates two lines of output: “line1” and “line2.”)

`var1.method1();`	Shape 1 / Rect 3 / Shape 3
`var2.method3();`	Square 3
`var3.method1();`	Shape 1 / Circle 3
`var4.method2();`	Error

Part C: Consider the following method:

```
public static void mystery(int n, int m) {
    if (n > 0 && m > 0) {
        mystery(n / 10, m / 10);
    }

    if (n % 10 == m % 10) {
        System.out.print("=");
    } else {
        System.out.print("!");
    }
}
```

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

`mystery(0, 90)`	=
`mystery(12, 22)`	=!=
`mystery(582, 1522)`	!=!=

3. Linked List Debugging

Consider a method in the LinkedIntList class called mergeWith that takes a LinkedIntList as a parameter and moves the values in the parameter into this list. Assuming both this list and the parameter list are in sorted order, after mergeWith executes, this list will contain the values from both lists in sorted order and the parameter list will be empty.

For example, suppose the variables `list1` and `list2` contain the following lists:

```
list1 = [-3 2 5 9 12]
list2 = [0 1 8 9 22]
```

After the call `list1.mergeWith(list2)` executes, the variables `list1` and `list2` would contain:

```
list1 = [-3 0 1 2 5 8 9 9 12 22]
list2 = []
```

On the other hand, if the lists were in their original state as above, then after the call `list2.mergeWith(list1)` executes, the variables `list1` and `list2` would contain:

```
list1 = []
list2 = [-3 0 1 2 5 8 9 9 12 22]
```

Consider the following buggy implementation of `mergeWith`:

```
1     public void mergeWith(LinkedIntList other) {
2         if (front == null) {
3             front = other.front;
4         } else if (other.front != null) {
5             ListNode curr = front;
6             while (curr.next != null && other.front != null) {
7                 if (curr.next.data > other.front.data) {
8                     ListNode temp = curr.next;
9                     curr.next = other.front;
10                    other.front = other.front.next;
11                    curr.next.next = temp;
12                }
13                curr = curr.next;
14            }
15
16            if (other.front != null) {
17                curr.next = other.front;
18            }
19        }
20        other.front = null;
21    }
```

This implementation contains a bug that is causing it to not work as intended. For example, given `list1` and `list2` as above, `list1.mergeWith(list2)` would produce the correct results, but `list2.mergeWith(list1)` would result in the following:

```
list1 = []
list2 = [0 -3 1 2 5 8 9 9 12 22]
```

(continued on next page…)

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 earn an S.)

```
1      public void mergeWith(LinkedIntList other) {
2          if (front == null) {
3              front = other.front;
4          } else if (other.front != null) {
5              if (other.front.data < front.data) {
6                  ListNode temp = front;
7                  front = other.front;
8                  other.front = other.front.next;
9                  front.next = temp;
10             }
11             ListNode curr = front;
12             while (curr.next != null && other.front != null) {
13                 if (curr.next.data > other.front.data) {
14                     ListNode temp = curr.next;
15                     curr.next = other.front;
16                     other.front = other.front.next;
17                     curr.next.next = temp;
18                 }
19                 curr = curr.next;
20             }
21
22             if (other.front != null) {
23                 curr.next = other.front;
24             }
25         }
26         other.front = null;
27     }
```

4. Inheritance Programming

Consider the following class:

```
public class Beverage {
    private double size;

    public Beverage(double size) {
        this.size = size;
    }

    public String toString() {
        return getSize() + "oz beverage";
    }

    public double getSize() {
        return size;
    }
}
```

Write a new class called `CoffeeDrink` that represents a beverage containing coffee. `CoffeeDrink` should extend `Beverage` but differ in the following ways:

- `CoffeeDrink` has a caffeine content (in mg) specified in the constructor as an integer
- `CoffeeDrink` has a `getCaffeine()` method that returns the caffeine content of the drink
- `CoffeeDrink` has an `isCaffeinated()` method that returns `true` if the drink contains at least 10mg of caffeine and `false` otherwise
- If a `CoffeeDrink` is *not* caffeinated, the string representation of the drink ends with `(decaf)`
    - The rest of the string representation is the same as any other `Beverage`
- `CoffeeDrink` implements the `Comparable` interface; `CoffeeDrinks` are compared first by size (smaller drinks are “less than” bigger drinks) then by caffeine content (less caffeine is “less than” more caffeine)

To earn an E on this problem, your `CoffeeDrink` class must not duplicate any code from the `Beverage` class.

Write your solution on the next page.

Write your solution to problem #4 here:

One possible solution:

```
public class CoffeeDrink extends Beverage implements Comparable<CoffeeDrink> {
    private int caffeine;

    public CoffeeDrink(double size, int caffeine) {
        super(size);
        this.caffeine = caffeine;
    }

    public boolean isCaffeinated() {
        return caffeine >= 10;
    }

    public int getCaffeine() { return caffeine; }

    public String toString() {
        String result = super.toString();
        if (!isCaffeinated()) {
            result += " (decaf)";
        }
        return result;
    }

    public int compareTo(CoffeeDrink other) {
        if (this.getSize() > other.getSize()) {
            return 1;
        } else if (this.getSize() < other.getSize()) {
            return -1;
        } else {
            return this.getCaffeine() - other.getCaffeine();
        }
    }
}
```

5. Recursive Programming

Write a *recursive* method called `waysToClimb` that takes two integer parameters, `steps` and `max`, and computes all possible ways to climb a set of `steps` by taking strides of at least 1 and at most `max` stairs at once. Your method should print out each possible way of climbing the stairs and return the total number of ways that are found.

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

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

This call would return the value `4`, since there are four possible options. Notice that order matters in the possibilities– `[1, 2]` and `[2, 1]` are considered different. Options should only consist of strides of at least 1 step– do not include 0 or negative steps.

As another example, the call `waysToClimb(5, 2)` would produce the following output:

```
[1, 1, 1, 1, 1]
[1, 1, 1, 2]
[1, 1, 2, 1]
[1, 2, 1, 1]
[1, 2, 2]
[2, 1, 1, 1]
[2, 1, 2]
[2, 2, 1]
```

This call would return the value `8`.

You may assume that both `steps` and `max` are 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 if you like.

Write your solution on the next page.

Write your solution to problem #5 here:

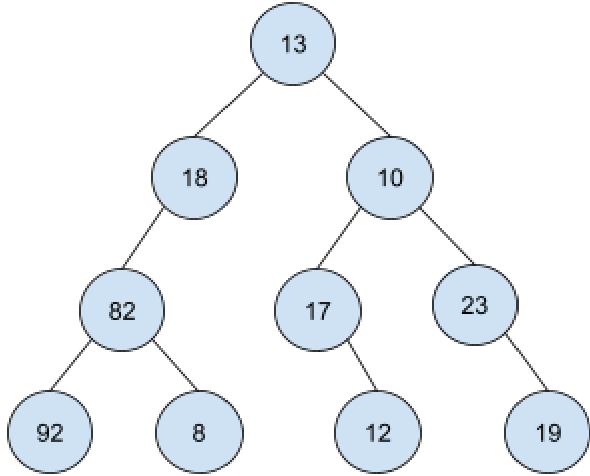
One possible solution:

```
public static int waysToClimb(int steps, int max) {
    return waysToClimb(steps, max, 0, new Stack<Integer>());
}

private static int waysToClimb(int steps, int max, int curr, Stack<Integer> plan) {
    if (curr == steps) {
        System.out.println(plan);
        return 1;
    }

    int total = 0;
    for (int i = 1; i <= max; i++) {
        if (curr + i <= steps) {
            plan.push(i);
            total += waysToClimb(steps, max, curr + i, plan);
            plan.pop();
        }
    }
    return total;
}
```

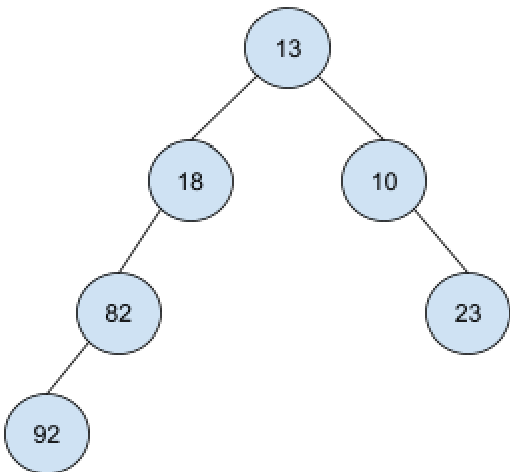
6. Binary Tree Programming

Write a method called **`limitLeaves`** to be added to the `IntTree` class (see the reference sheet). This method takes a single integer parameter, `min`, and modifies the tree so that no leaf node contains a value that is less than `min`.

For example, suppose that the variable `tree` contains a reference to the following tree:



The method call `tree.limitLeaves(20)` would result in `tree` referring to the following tree:



Notice that the leaf nodes containing the values 8, 12, and 19 were removed because they are less than 20. Notice also that removing the node containing 12 created a new leaf node containing 17, which was also removed.

Your method should not create any `IntTreeNode` objects or modify the `data` field of any existing `IntTreeNode` objects; you should implement your method only by removing nodes from the existing tree. You should not remove any nodes that are not necessary to produce a tree with no leaves less than `min`. Specifically, you should only remove leaf nodes, or nodes that become leaf nodes after removing other nodes.

Write your solution on the next page.

Write your solution to problem #6 here:

One possible solution:

```
public void limitLeaves(int min) {
    overallRoot = limitLeaves(min, overallRoot);
}

private IntTreeNode limitLeaves(int min, IntTreeNode root) {
    if (root == null) {
        return null;
    }

    root.left = limitLeaves(min, root.left);
    root.right = limitLeaves(min, root.right);
    if (root.left == null && root.right == null && root.data < min) {
        root = null;
    }
    return root;
}
```