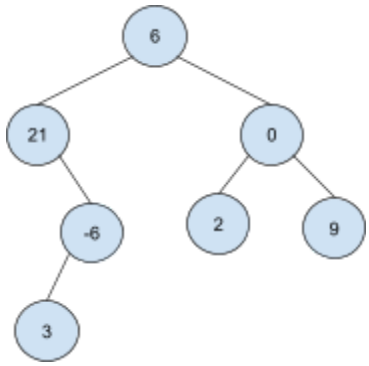
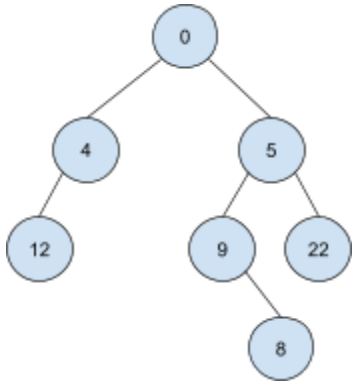
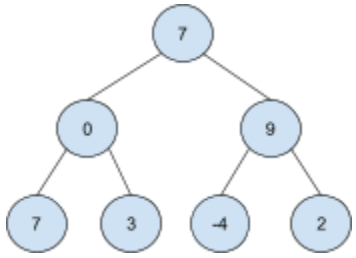
CSE 123 Winter 2024 Practice Final Exam #1 Key

1. Comprehension

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

	6 21 -6 3 0 2 9	☑ pre-order ☐ in-order ☐ post-order
	12 4 0 9 8 5 22	☐ pre-order ☑ in-order ☐ post-order
	7 0 3 7 -4 9 2	☐ pre-order ☑ in-order ☐ post-order

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

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

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

    if (root.left == null && root.right == null) {
        return root.data;
    }

    return mystery(root.left) + mystery(root.right);
}
```

Draw a binary tree such **with at least 3 nodes** that, if it were stored in the variable `tree`, the call `tree.mystery()` would return 25.

Any tree with at least 3 nodes whose leaf nodes sum up to exactly 25.

Part C: Hearing that you have recently completed CSE 123, the instructor for one of your other classes comes to you for help. They would like to build a system to keep track of their TAs and how they are supporting students. You have a few conversations with your instructor and settle on the following basic design for a `TA` class and an interface to represent message board posts:

```
public class TA {
    private String name;
    private int quartersExperience;
    private List<MBPost> posts;
}
```

```
public interface MBPost {
      public String poster;
      public String content;
      public boolean isAnswer;
}
```

At the end of the quarter, the instructor asks you to extend the system to help them decide which of their TAs to rehire for next quarter. You decide to use the `Comparable` interface to provide a way for the instructor to sort the `TA` objects in their system, and implement the following method:

```
public int compareTo(TA other) {
      if (this.posts.size() != other.posts.size()) {
            return other.posts.size() - this.posts().size();
      } else if (this.quartersExperience != other.quartersExperience) {
            return other.quartersExperience - this.quartersExperience;
      } else {
            return this.name.compareTo(other.name);
      }
}
```

I. Write an appropriate method comment for the above `compareTo` implementation. You should follow similar guidelines for comments as you did on CSE 123 assignments.

One possible response:

Compares two TAs, first by number of message board posts (decreasing), then by quarters of experience (decreasing), then by name (lexicographically). Returns a negative number if this TA is “less than” (i.e. “better”) than the other TA.

II. Describe a potential concern your instructor might have with its design, and explain a high-level change you could make to address that concern. Your critique should be from the perspective of the ***client*** (your instructor), not the implementer (you).

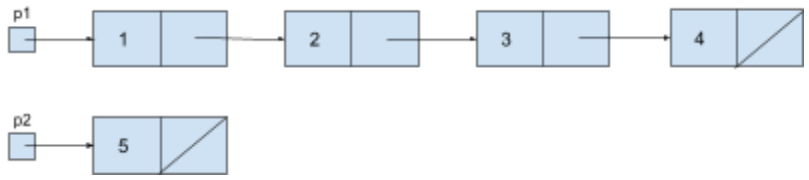
Some possible concerns:

- Ordering by message board posts, especially without regard to usefulness, encourages bad behavior.
- Ordering by descending experience will only prefer rehiring experienced TAs, which might lead to an issue when those TAs all leave.
- Ordering by name is arbitrary.

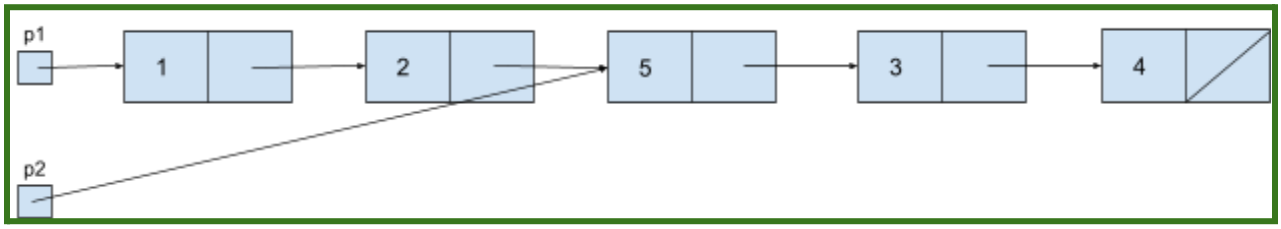
Changes should describe some alternate implementation of `compareTo`.

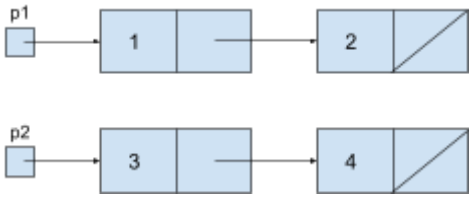
2. Code Tracing

Part A: For each of the following, draw the linked lists that are produced by starting with the lists shown on the right and executing the code provided. You only need to draw the final lists, not any intermediate steps. You do not need to draw any variables created in the code, only the references `p1` and `p2` in the original diagram.

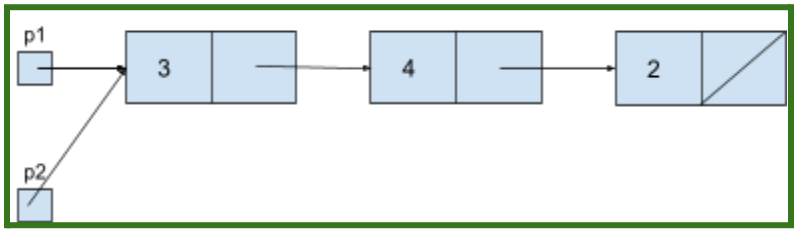


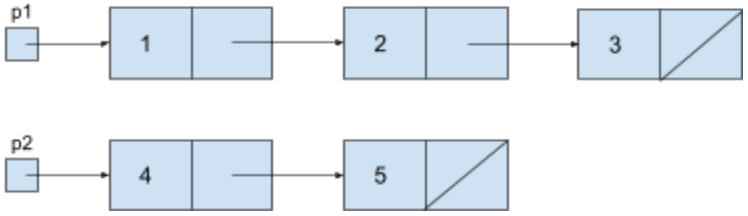

```
p2.next = p1.next.next;
p1.next.next = p2;
```



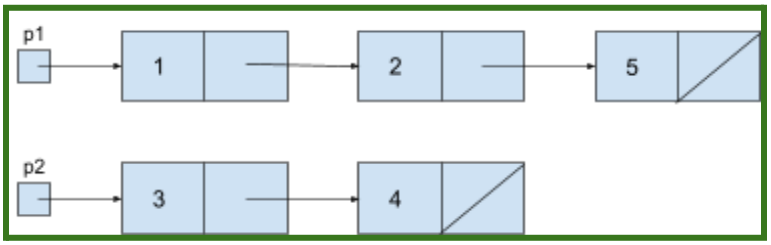



```
p2.next.next = p1.next;
p1 = p2;
```






```
ListNode temp = p1.next.next;
p1.next.next = p2.next;
temp.next = p2;
p2 = temp;
p2.next.next = null;
```



Part B: Consider the following classes:

```
class Fruit {
    public void method1() {
        System.out.println("Fruit 1");
        method2();
```

```
    }

    public void method2() {
        System.out.println("Fruit 2");
    }
}

class Apple extends Fruit {
    public void method1() {
        method2();
        System.out.println("Apple 1");
    }

    public void method3() {
        System.out.println("Apple 3");
    }
}
class CosmicCrisp extends Apple {
    public void method3() {
        super.method1();
        System.out.println("Cosmic Crisp
3");
    }
}

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

    public void method3() {
        method1();
        System.out.println("Honeycrisp 3");
    }
}
```

Assume the following variables have been defined:

```
Apple var1 = new Apple();
Fruit var2 = new Honeycrisp();
Apple var3 = new CosmicCrisp();
Fruit var4 = new Fruit();
```

For each of the following statements, Indicate what the output would be. If the statement would result in an error (either a compiler error or an exception), 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();`	`Fruit 2` `Apple 1`
`var2.method2();`	`Honeycrisp 2`

`var3.method3();`	`Fruit 2`
	`Apple 1`
	`Cosmic Crisp 3`

`var4.method3();`	`error`

Part C: Consider the following method:

```
public static void mystery(String str1, String str2) {
    if (str1.length() == 0 || str2.length() == 0) {
        System.out.print("!");
    } else {
        if (str1.charAt(0) == str2.charAt(0)) {
            System.out.print(str1.charAt(0));
        } else {
            System.out.print(".");
       }
       mystery(str1.substring(1), str2.substring(1));
    }
}
```

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

`mystery("cat", "cat")`	`cat!`
`mystery("bird", "burn")`	`b.r.!`
`mystery("dog", "dollar")`	`do.!`

3. Linked List Debugging

Consider a method in the `LinkedIntList` class called `combinePairs` that replaces each pair of nodes in the linked list with a new node whose value is the sum of the values of the two replaced nodes. If the list contains an odd number of nodes, the last node of the list should remain.

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

```
[1, 2, 3, 4, 5, 6]
```

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

```
[3, 7, 11]
```

Notice that each pair of nodes (1 and 2, 3 and 4, 5 and 6) has been replaced by a single node containing their sum (3, 7, and 11 respectively).

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

```
[10, 15, 20, 25, 30, 35, 40]
```

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

```
[25, 45, 65, 40]
```

As in the previous example, each pair of nodes has been replaced, but the final node (the 40) has been left unchanged at the end of the list.

On the next page is a buggy implementation of `combinePairs` that does not work as intended. In this implementation, given the original `list1` and `list2` above, `list1.combinePairs()` would result in `list1` containing `[1, 5, 9, 6]`; and `list2.combinePairs()` would result in `list2` containing `[10, 35, 55, 75]`.

Part A: Provide a different example list 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:	*Any list of size greater than 1*
Buggy result:	*Skips first element and merges elements 2 and 3, 4 and 5, etc.*
Correct result:	*Merges elements 1 and 2, 3 and 4, etc.*

(continued on next page…)

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 combinePairs() {
2       if (front != null) {
3           if (front.next ! null) {
4               front = new ListNode(front.data + front.next.data, front.next.next);
5               size--;
6           }
7
8           ListNode prev = front;
9           ListNode curr = front.next;
10          while (curr != null && curr.next != null) {
11              ListNode replace = new ListNode(curr.data + curr.next.data, curr.next.next);
12              prev.next = replace;
13              prev = replace;
14              curr = prev.next;
15              size--;
16          }
17      }
18  }
```

4. Inheritance Programming

Consider the following class:

```
public class Consultant {
    private String name;
    private Map<String, Double> proposals;
    private List<String> contractsWon;

    public Consultant(String name) {
        this.name = name;
        this.proposals = new HashMap<String, Double>();
        this.contractsWon = new ArrayList<String>();
    }

    public void writeProposal(String title, double value) {
        proposals.put(title, value);
    }

    public int getNumProposals() {
        return proposals.keySet().size();
    }

    public int getNumContractsWon() {
        return contractsWon.size();
    }

    public double getTotalValueWon() {
        double total = 0;
        for (String contract : contractsWon) {
            total += proposals.get(contract);
        }
        return total;
    }

    public boolean getsBonus() {
        return (double)getNumContractsWon() / getNumProposals() > 0.5;
    }
}
```

Write a new class called **`SeniorConsultant`** that represents a more senior consultant who can supervise others and whose success is partially based on that of their employees. `SeniorConsultant` should extend `Consultant` but differ in the following ways:

- A `SeniorConsultant` has a list of employees, who are themselves `Consultants`. Each `SeniorConsultant` starts with no employees.
- `SeniorConsultant` has a method `addEmployee(Consultant employee)` to add an employee.
- A `SeniorConsultant` gets a bonus (that is, the `getsBonus()` method returns `true`) if either they meet the requirements for a regular `Consultant` *or* they and their employees have won contracts totalling at least $100,000.

To earn an E on this problem, your `SeniorConsultant` class must not duplicate any code from the `Consultant` class and must not include any unnecessary overrides.

Write your solution on the next page.

Write your solution to problem #4 here:

One possible solution:

```
public class SeniorConsultant extends Consultant {
    private List<Consultant> employees;

    public SeniorConsultant(String name) {
        super(name);
        this.employees = new ArrayList<Consultant>();
    }

    public void addEmployee(Consultant employee) {
        employees.add(employee);
    }

    public boolean getsBonus() {
        double total = getTotalValueWon();
        for (Consultant employee : employees) {
            total += employee.getTotalValueWon();
        }

        return total >= 100000 || super.getsBonus();
    }
}
```

5. Recursive Programming

Write a *recursive* method called **`makePhrases`** that takes two parameters: a list of strings named `words`, and an integer named `length`. Your method should print out all possible phrases of *exactly* `length` characters made up of words from `words` separated by spaces.

For example, suppose the variable words contained a reference to the following list:

```
["i", "am", "a", "cat", "with", "toys"]
```

Then the call `makePhrases(words, 3)` would produce the following output:

```
i a
a i
cat
```

Notice that the spaces between words are included in the character count. For example, the phrase `"i a"` has length 3. Notice also that order matters: `"i a"` and `"a i"` are considered separate phrases.

Similarly, the call `makePhrases(words, 6)` would produce the following output:

```
i am a
i a am
i with
i toys
am i a
am a i
am cat
a i am
a am i
a with
a toys
cat am
with i
with a
toys i
toys a
```

Again, notice that the spaces between words are included in the character count and that order matters.

You may assume that `words` is not null and that `length` is greater than or equal to 0. It is acceptable (but not required) for your method to print an extra space at the end of each phrase, however this space should *not* be included in the character count– only spaces between words are included. (That is, you may print `"i a "` instead of `"i a"`, but `"i a "` should still be considered to have length 3.)

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:

One possible solution:

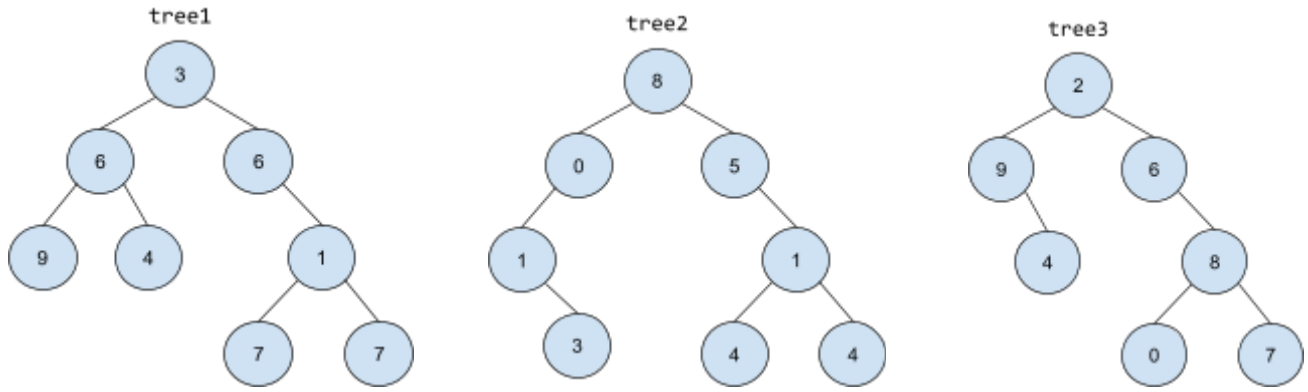
```
public static void makePhrase(List<String> words, int length) {
    makePhrase(words, length, "");
}

private static void makePhrase(List<String> words, int length, String phrase) {
    if (phrase.length() == length + 1) {
        System.out.println(phrase);
    } else {
        for (int i = 0; i < words.size(); i++) {
            String curr = words.remove(i);
            if (phrase.length() + curr.length() + 1 <= length + 1) {
                makePhrase(words, length, phrase + " " + curr);
            }
            words.add(i, curr);
        }
    }
}
```

6. Binary Tree Programming

Write a method called **`countTwins`** to be added to the `IntTree` class (see the reference sheet). This method should return the number of pairs of “twins" in the tree– that is, pairs of nodes that store the same value and have the same parent. Or, put another way, the method should return the number of nodes that have two children with the same value.

For example, suppose the variables `tree1`, `tree2`, and `tree3` contain references to the following trees:



In these cases, the call `tree1.countTwins()` would return 2– the nodes containing 3 and 1 each have twins as their children. The call `tree2.countTwins()` would return 1 for the node containing 4. (In `tree2`, the nodes containing 1 are *not* twins because they do not share the same parent.) The call `tree3.countTwins()` would return 0.

Write your solution on the next page.

Write your solution to problem #6 here:

One possible solution:

```
public int countTwins() {
    return countTwins(overallRoot);
}

private int countTwins(IntTreeNode root) {
    if (root == null) {
        return 0;
    }

    int total = countTwins(root.left) + countTwins(root.right);
    if (root.left != null && root.right != null &&
        root.left.data == root.right.data) {
        total++;
    }

    return total;
}
```