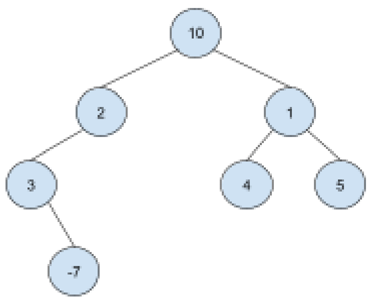
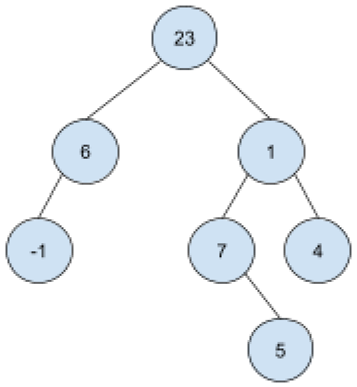
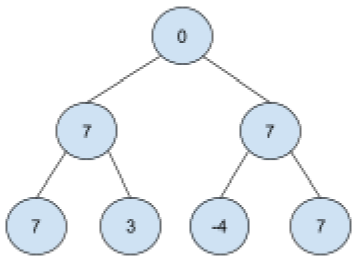
CSE 123 Autumn 2024 Review Session Practice Exam

1. Comprehension

Part A: (Select all that apply) Which of these statements are true about runtime?

- ☐ The runtime of `size()` in `ArrayIntList` (from class) is *O*(n)
- ☐ The time complexity for a method with two for-loops placed side by side (not nested) is greater than the time complexity for a method with one for-loop.
- ☐ A function with a runtime of $O(n^2)$ grows faster than a function with a runtime of *O*(n) as the input size increases.
- ☐ If `method2` has an *O*(n) runtime, and `method1` calls `method2` n times, then the runtime of `method1` is $O(n^n)$
- ☐ In Big-Oh notation, only the dominating term matters for complexity because lower-order terms become insignificant for very large input sizes.

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

	3 -7 2 10 4 1 5	☐ pre-order ☐ in-order ☐ post-order
	23 6 -1 1 7 5 4	☐ pre-order ☐ in-order ☐ post-order
	7 3 7 -4 7 7 0	☐ pre-order ☐ in-order ☐ post-order

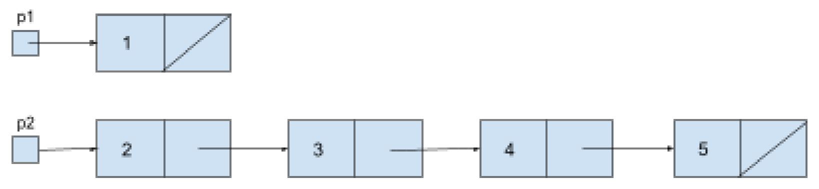
Part C: Consider the following method in the `IntTree` class:

```
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);
}
```

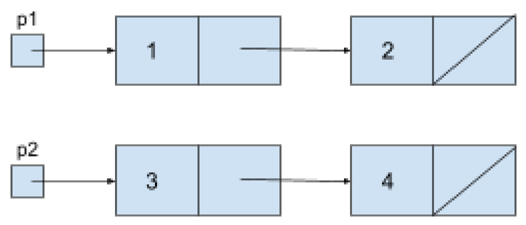
Draw a binary tree **with at least 3 nodes** such that, if it were stored in the variable `tree`, the call `tree.mystery()` would return 10.

2. Code Tracing

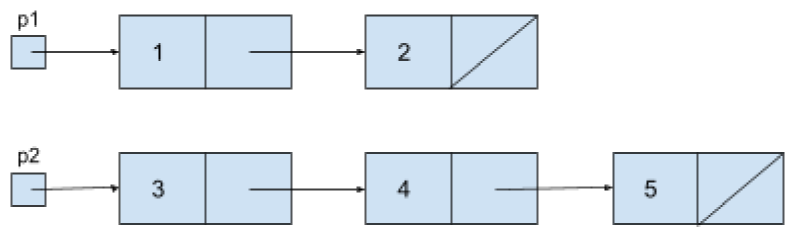
Part A: For each of the following, draw the linked lists that are produced by starting with the lists shown on the right and executing the code provided. You only need to draw the final lists, not any intermediate steps. You do not need to draw any variables created in the code, only the references p1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



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



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



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

Part B: Consider the following classes:

```
public class Tree {
    public void fullName() {
        genus();
        System.out.println("Tree");
    }

    public void genus() {
        System.out.println("Unknown");
    }
}
```

```
public class Fir extends Tree {
    public void genus() {
        System.out.println("Abies");
    }
}
```

```
public class GrandFir extends Fir {
    public void fullName() {
        genus();
        System.out.println("Grandis");
    }

    public void describe() {
        super.fullName();
        System.out.println("Flat needles");
    }
}
```

```
public class PacificSilverFir extends Fir {
    public void fullName() {
        System.out.println("Pacific Silver Fir");
    }

    public void describe() {
        fullName();
        System.out.println("Beautiful");
    }
}
```

Assume the following variables have been defined:

```
    Tree var1 = new Tree();
    Fir var2 = new Fir();
    Tree var3 = new GrandFir();
    PacificSilverFir var4 = new PacificSilverFir();
```

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.fullName();`	
`var2.fullName();`	
`var3.describe();`	
`var4.describe();`	

Part C: Consider the following method:

```
public static void mystery(int n) {
    if (n % 2 == 0) {
        System.out.print(2);
        mystery(n / 2);
    } else if (n % 3 == 0) {
        System.out.print(3);
        mystery(n / 3);
    } else if (n % 5 == 0) {
        System.out.print(5);
        mystery(n / 5);
    } else if (n >= 1) {
        System.out.print(n);
    }
}
```

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

`mystery(5)`

`mystery(150)`

`mystery(132)`

3. 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`**:

```java
public static void printSeq(List<String> names, int n){
    if (names.size() < n) {
        throw new IllegalArgumentException();
    }
    printSeq(names, 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. You decide that you want to write a test that exposes the incorrect behavior. Provide contents for `list` and n, then write the output that the code above will produce.

```
list = [                    ]          n = [   ]

printSeq(list, n);
```

Output:

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 (names.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             _____________________________________________;
20         }
21     }
22 }
```

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 `SweetenedDrink` that represents a beverage containing coffee. `SweetenedDrink` should extend `Beverage` but differ in the following ways:

- `SweetenedDrink` constructor should take in two values: sweetener content (in mg) as an integer, and a size of the drink, a double
- `SweetenedDrink` has a `getSweetener()` method that returns the sweetener content of the drink
- `SweetenedDrink` has an `isSweetened()` method that returns `true` if the drink contains at least 10mg of sweeteners and `false` otherwise
- If a `SweetenedDrink` is sweetened (contains at least 10mg of sweetener), the string representation of the drink ends with `"(sweetened)"`
    - The rest of the string representation is the same as any other `Beverage`. For example, `"8 oz beverage (sweetened)"`
- `SweetenedDrink` implements the `Comparable` interface; `SweetenedDrinks` are compared first by size (smaller drinks are “less than” bigger drinks) then by sweetener content (less sweetener is “less than” more sweetener)

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

Write your solution on the next page.

Write your solution to problem #4 here:

5. Linked List Programming

Write a method called **`squash(int target)`** to be added to the `LinkedIntList` class (see the reference sheet). This method takes a single integer parameter, `target`, and modifies the list such that the first occurrence of a node with value `target` is squashed with the node after it, combining the values of both nodes.

For example, if the original contents of the linked list were:

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

Then after running **`linkedList.squash(2)`**, the list's contents would be:

```
linkedList = [1, 6, 2, 2, 3]
```

Notice that calling squash would only squash the ***first*** occurrence of a node with the value `target`.

If no node with the value `target` exists within the list, the list should remain unmodified. If a node with the value `target` is found, but there is no node following it, then the list should remain unmodified. Take our original linked list again:

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

Then after running **`linkedList.squash(3)`**, the list should be unmodified.

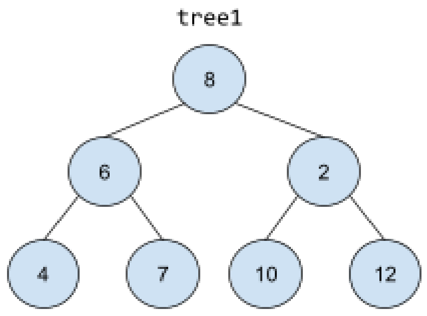
Your implementation for **`squash`** may be recursive or iterative — your choice! You may use private helper methods to solve this problem, but otherwise, you may not assume that any particular methods are available. You are allowed to define your own variables of type `ListNode`, and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). Recall that the `data` field in the `ListNode` class is final, and so node values cannot be changed. You MUST solve this problem by constructing new nodes for the expanded values and rearranging the links of the lists. Your solution must run in `O(n)` time where n is the length of the list. Write your implementation to **`squash(int target)`** on the next page.

Write your solution to problem #5 here:

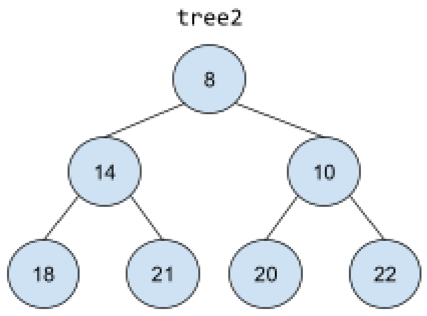
6. Binary Tree Programming

Write a method called **`pathSumCopy()`** to be added to the `IntTree` class (see the reference sheet). This method creates and returns a copy of the tree except all the node values of the new tree contain the path sum of the other tree. The path sum of a node is defined as the sum of all the values from the root up to and including that node.

For example, suppose we have the following tree:



Then, after calling **`IntTree tree2 = tree1.pathSumCopy();`** `tree2` should look like:



In this case, notice that `tree2` has the same exact structure as `tree1`, but each node contains the path sum of the corresponding node in the other tree.
You are writing a method that will become part of the IntTree class. You may use private helper methods to solve this problem, but otherwise you may not call any other methods of the class. `tree1` and `tree2` should not share any references

Write your solution on the next page.

Write your solution to problem #6 here:

CSE 123 Quiz/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 `"{a=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.

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();
}
```

```
public interface InterfaceExample {
    public void method();
}
```

ArrayIntList

```
public class ArrayIntList {
    private int[] elementData;
    private int size;
}
```

LinkedIntList

```
public class LinkedIntList {
    private ListNode front;

    private static class ListNode {
        public final int data;
        public ListNode next;

        public ListNode(int data) {
            this(data, null);
        }

        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }}}
```

`IntTree` Class

```
public class IntTree {
    private IntTreeNode overallRoot;

    private static class IntTreeNode {
        public final int data;
        public IntTreeNode left;
        public IntTreeNode right;

        public IntTreeNode(int data) {
            this(data, null, null);
        }

        public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }
}
```