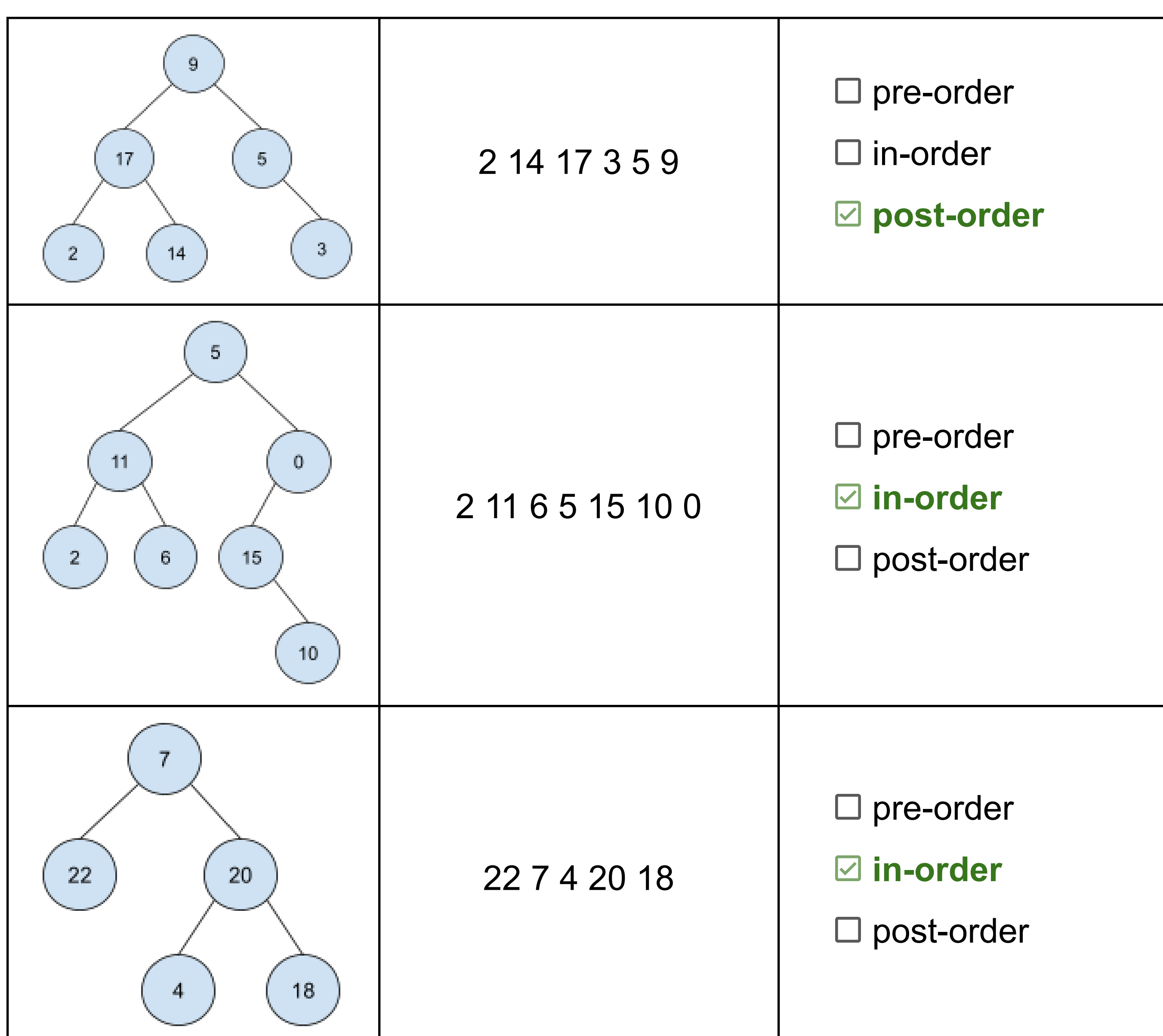
CSE 123 Autumn 2023 Practice Final Exam #1 Key

1. Comprehension

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

- ☐ Abstract classes cannot contain method implementations.
- ☑ **Linked lists are a better choice than array lists when values will frequently be added to the middle of the list.**
- ☑ **Recursive methods can contain multiple base cases.**
- ☐ Some problems can *only* be solved recursively.

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



Tree	Traversal	Type
(root 9; 17 with children 2, 14; 5 with right child 3)	2 14 17 3 5 9	☐ pre-order ☐ in-order ☑ **post-order**
(root 5; 11 with children 2, 6; 0 with left child 15; 15 with right child 10)	2 11 6 5 15 10 0	☐ pre-order ☑ **in-order** ☐ post-order
(root 7; 22; 20 with children 4, 18)	22 7 4 20 18	☐ pre-order ☑ **in-order** ☐ post-order

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

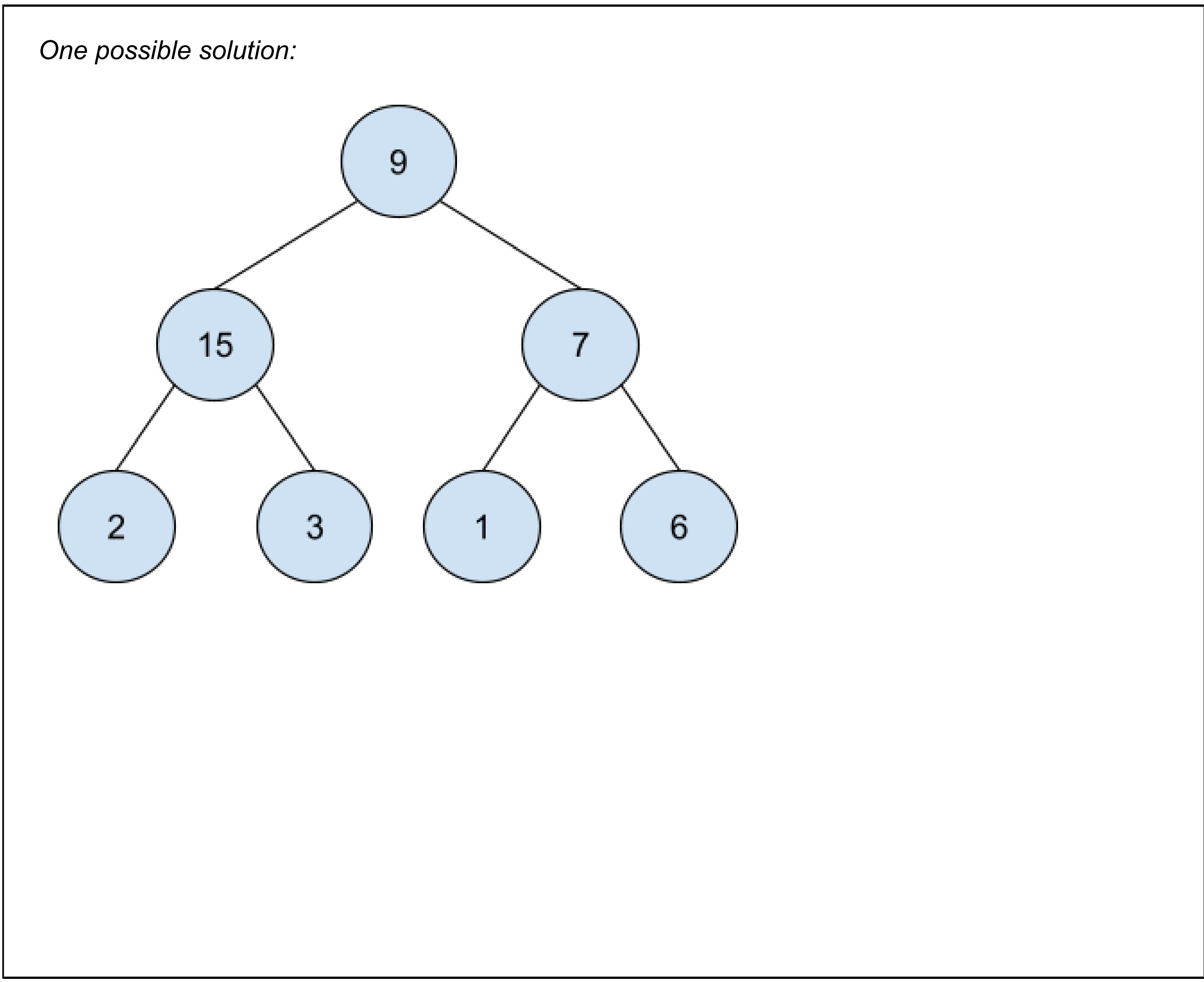
```
public int mystery(int sum) {
        return mystery(overallRoot, sum, 0);
}

private int mystery(IntTreeNode root, int sum, int curr) {
        if (root.left == null && root.right == null) {
                if (root.data + curr >= sum) {
                        return 1;
                } else {
                        return 0;
                }
        }

        return mystery(root.left, sum, curr + root.data) +
                mystery(root.right, sum, curr + root.data);
}
```

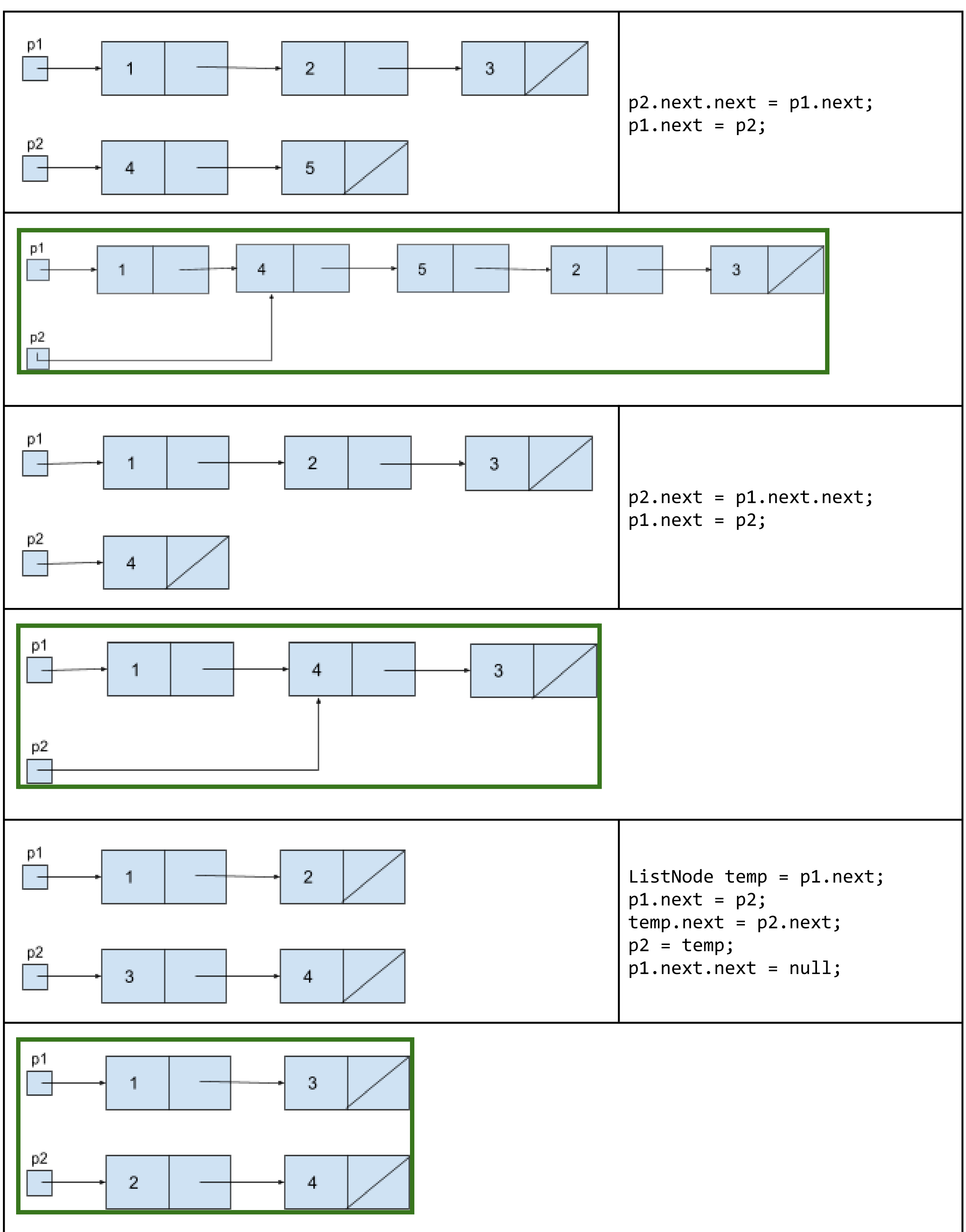
Draw a binary tree such that, if it were stored in the variable `tree`, the call `tree.mystery(20)` would return 3.

One possible solution:



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.



Part B: Consider the following classes:

```
public class Animal {
    public void method1() {
        System.out.println("Animal 1");
    }

    public void method2() {
        method1();
        System.out.println("Animal 2");
    }
}

public class Cat extends Animal {
    public void method1() {
        System.out.println("Cat 1");
    }

    public void method3() {
        method2();
        System.out.println("Cat 3");
    }
}
```

```
public class Dog extends Animal {
  public void method1() {
       System.out.println("Dog 1");
  }

  public void method3() {
       System.out.println("Dog 3");
       method2();
  }
}

public class Husky extends Dog {
  public void method2() {
       super.method2();
       System.out.println("Husky 2");
  }
}
```

Assume the following variables have been defined:

```
Dog var1 = new Husky();
Cat var2 = new Cat();
Animal var3 = new Animal();
Animal var4 = new Cat();
```

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.method2();`	**Dog 1** **Animal 2** **Husky 2**
`var2.method2();`	**Cat 1** **Animal 2**
`var3.method1();`	**Animal 1**
`var4.method3();`	**error**

Part C: Consider the following method:

```
public static void mystery(int x) {
    if (x < 10) {
        System.out.print(x);
    } else {
        int y = x % 10;
        System.out.print(y + "-");
        mystery(x / 10);
        System.out.print("-" + y);
    }
}
```

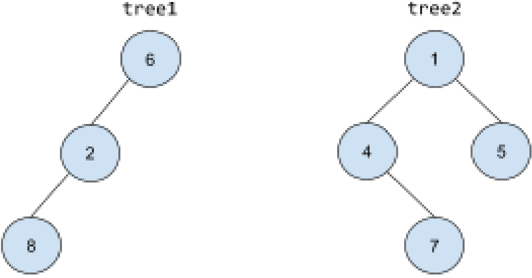
For each of the following statements, indicate what the output would be.

`mystery(2)`	2
`mystery(861)`	1-6-8-6-1
`mystery(4039)`	9-3-0-4-0-3-9

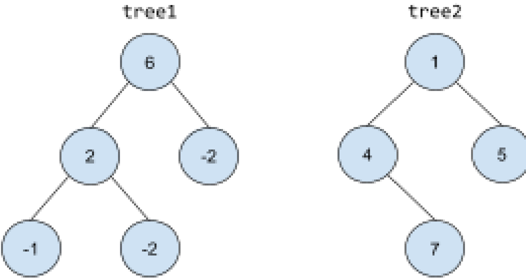
3. Binary Tree Debugging

Consider a method in the `IntTree` class called **`markDiffs`** that takes a single `IntTree` parameter, `other`, and modifies this tree to indicate nodes that exist in one of this tree or `other` but not both. If a node exists in this tree but not in `other`, the node in this tree should be replaced with a node with the value -1. If a node exists in `other` but not in this tree, a corresponding node with the value -2 should be added to this tree. If the node exists in both trees or neither tree, leave it unchanged in this tree.

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

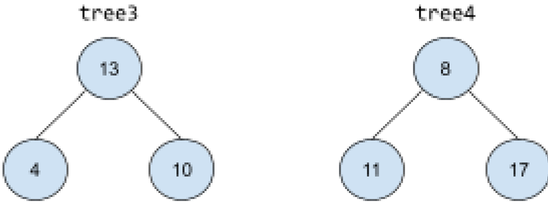


The method call `tree1.markDiffs(tree2)` would result the following trees:



Notice that the node that exists only in `tree1` (8) has been replaced with a -1, and that nodes with the value -2 have been added to `tree1` in the places of nodes that exist only in `tree2` (5 and 7). Nodes that exist in both trees (6 and 2 in `tree1`, 1 and 4 in `tree2`) are unchanged in `tree1`, and `tree2` is entirely unchanged.

Now, suppose that the variables `tree3` and `tree4` contain references to the following trees:



The method call `tree3.markDiffs(tree4)` would not make any changes to either tree because each node in `tree3` has a corresponding node in `tree4` and vice versa.

(continued on next page…)

Below is a buggy implementation of `markDiffs` that does not work as intended. Given the original example trees above, `tree3.markDiffs(tree4)` produces the correct results, but `tree1.markDiffs(tree2)` results in a `NullPointerException` on line 10.

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 markDiffs(IntTree other) {
2        overallRoot = markDiffs(this.overallRoot, other.overallRoot);
3    }
4
5    private IntTreeNode markDiffs(IntTreeNode root1, IntTreeNode root2) {
6        if (root1 != null && root2 != null) {
7            root1.left = markDiffs(root1.left, root2.left);
8            root1.right = markDiffs(root1.right, root2.right);
9        } else if (root1 == null && root2 != null) {
10           root1 = new IntTreeNode(-2, markDiffs(null, root2.left),
11                                       markDiffs(null, root2.right));
12       } else if (root1 != null && root2 == null) {
13           root1 = new IntTreeNode(-1, markDiffs(root1.left, null),
14                                       markDiffs(root1.right, null));
15       }
16       return root1;
17   }
```

4. Inheritance Programming

Consider the following class:

```
public class IceCream {
        private Map<String, Integer> flavors;
        private int scoops;

        public IceCream() {
                this.flavors = new HashMap<>();
        }

        public void addScoops(String flavor, int scoops) {
                if (!flavors.containsKey(flavor)) {
                        flavors.put(flavor, 0);
                }
                flavors.put(flavor, flavors.get(flavor) + scoops);
                this.scoops += scoops;
        }

        public int getScoops() {
                return scoops;
        }

        public String toString() {
                if (scoops == 0) {
                        return "No ice cream :(";
                }
                return scoops + " total scoops of " + flavors.keySet();
        }
}
```

Write a new class called **`Sundae`** that represents an ice cream sundae. `Sundae` should extend `IceCream` but differ in the following ways:

- `A Sundae` can include toppings (represented by strings). Each `Sundae` starts with no toppings.
- `Sundae` has a method `addTopping(String topping)` to add a topping.
  - A `Sundae` either includes a topping or does not– there are no amounts, and toppings cannot be included multiple times. Adding a topping that is already present has no effect.
- `Sundae` has a method `hasTopping(String topping)` that returns `true` if the sundae includes the specified topping and `false` otherwise.
- If a `Sundae` has any toppings, the string representation of the `Sundae` should include the list of toppings (in any order).
  - The rest of the string representation is the same as any other `IceCream`. For example, `"4 total scoops of [chocolate, strawberry] with [hot fudge, sprinkles]"`
- `Sundae` implements the `Comparable` interface; `Sundaes` are compared first by number of scoops (fewer scoops is "less than" more scoops) then by number of toppings (fewer toppings is "less than" more toppings)

To earn an E on this problem, your `Sundae` class must not duplicate any code from the `IceCream` 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 Sundae extends IceCream implements Comparable<Sundae> {
    private Set<String> toppings;

    public Sundae() {
        this.toppings = new HashSet<>();
    }

    public void addTopping(String topping) {
        toppings.add(topping);
    }

    public boolean hasTopping(String topping) {
        return toppings.contains(topping);
    }

    public String toString() {
        String result = super.toString();
        if (!toppings.isEmpty()) {
            result += " with " + toppings;
        }
        return result;
    }

    public int compareTo(Sundae other) {
        if (this.getScoops() != other.getScoops()) {
            return this.getScoops() - other.getScoops();
        } else {
            return this.toppings.size() - other.toppings.size();
        }
    }
}
```

5. Recursive Programming

Write a *recursive* method called **`drawMarbles`** that takes two parameters, a list of strings, `marbles`, representing marble colors and an integer, `num`, and computes all possible ways to draw `num` marbles from the list of possibilities (without replacement). Your method should print out each possible sequence of draws.

For example, suppose the variables `colors` contained a reference to the following list:

```
["red", "blue", "green"]
```

Then, the call `drawMarbles(colors, 3)` would produce the following output:

```
[red, blue, green]
[red, green, blue]
[blue, red, green]
[blue, green, red]
[green, red, blue]
[green, blue, red]
```

Notice that order matters in the possibilities– `[red, blue, green]` and `[blue, green, red]` are considered different.

As another example, if the variable colors contained the following list:

```
["red", "blue", "green", "yellow"]
```

Then the call `drawMarbles(colors, 2)` would produce the following output:

```
[red, blue]
[red, green]
[red, yellow]
[blue, red]
[blue, green]
[blue, yellow]
[green, red]
[green, blue]
[green, yellow]
[yellow, red]
[yellow, blue]
[yellow, green]
```

You may assume that `num` is less than or equal to the number of elements in `marbles`, and that each element of `marbles` is unique.

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 drawMarbles(List<String> marbles, int target) {
     drawMarbles(marbles, target, new ArrayList<String>());
}

private static void drawMarbles(List<String> marbles, int target, List<String> lst) {
     if (lst.size() == target) {
          System.out.println(lst);
     } else {
          for (int i = 0; i < marbles.size(); i++) {
               String curr = marbles.get(i);
               marbles.remove(i);
               lst.add(curr);
               drawMarbles(marbles, target, lst);
               lst.remove(curr);
               marbles.add(i, curr);
          }
     }
}
```

6. Linked List Programming

Write a method called **`removeDups`** to be added to the `LinkedIntList` class (see the reference sheet). This method should reduce all sequences of the same value in the list to a single instance of that value. For example, suppose the variable `list` contains a reference to the following list:

```
[3, 0, 0, 2, 8, 8]
```

After the call `list.removeDups()` executes, `list` would contain:

```
[3, 0, 2, 8]
```

As another example, suppose the variable `list2` contains the following list:

```
[6, 6, 1, 1, 1, 9, 7, 7, 9, 4]
```

After the call `list.removeDups()` executes, `list` would contain:

```
[6, 1, 9, 7, 9, 4]
```

Notice that if the list contains non-consecutive duplicates, they do *not* need to be collapsed

To earn an E on this problem, you must not create any new objects of any kind, though you may create new variables or references.

Write your solution on the next page.

Write your solution to problem #6 here:

One possible solution:

```
public void removeDups() {
      ListNode curr = front;
      while (curr != null && curr.next != null) {
          if (curr.data == curr.next.data) {
              curr.next = curr.next.next;
              size--;
          } else {
              curr = curr.next;
          }
      }
  }
```

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.)