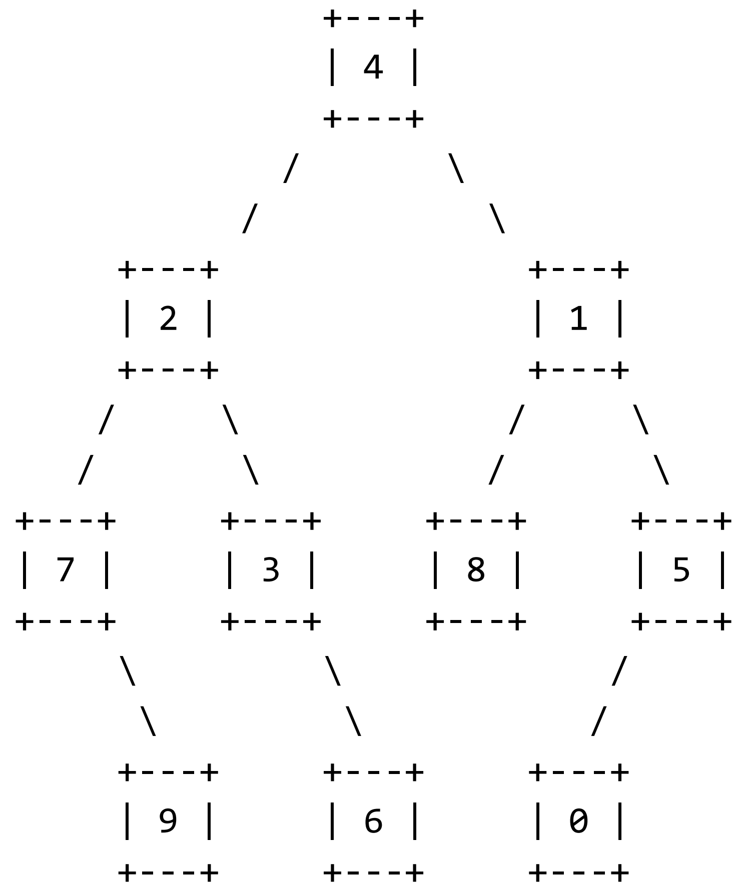
1. Code Comprehension

Part A: For the following binary tree, write the preorder, inorder, and postorder traversal.



Preorder: 4, 2, 7, 9, 3, 6, 1, 8, 5, 0

Inorder: 7, 9, 2, 3, 6, 4, 8, 1, 0, 5

Postorder: 9, 7, 6, 3, 2, 8, 0, 5, 1, 4

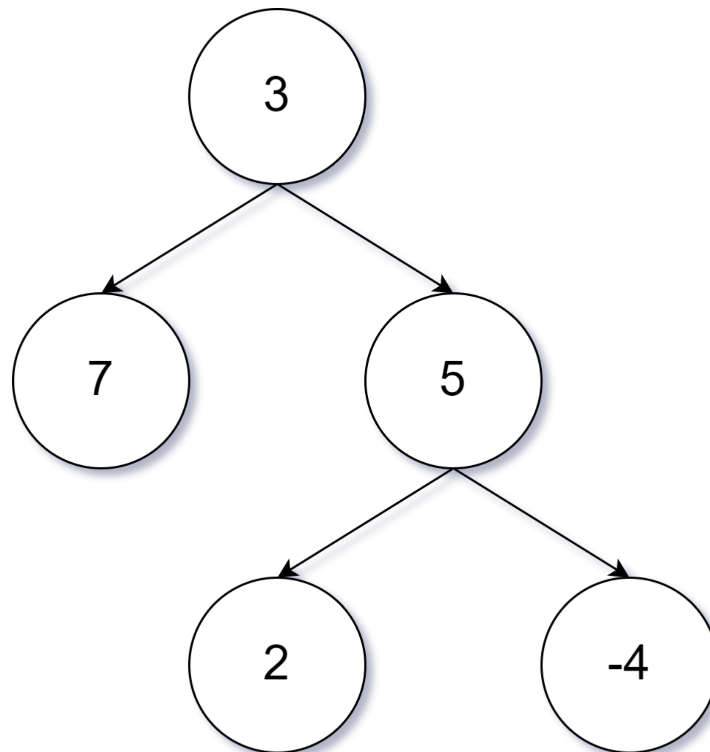
Part B: (Select all that apply) Which of these statements are true about inheritance?

- [ ] Inheritance allows a subclass to access all private properties and methods of its parent class.
- [x] Calling a method using `super` will go to the parent class one level above the current subclass.
- [ ] To indicate that class A is a subclass of B, we write `public class B extends A`
- [ ] Parent classes can call the methods of their subclasses.
- [x] If class A is a subclass of B, then the following statement is legal:
  `B a = new A();`

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

```
1     public List<String> method() {
2         List<String> result = new ArrayList<>();
3         methodHelper(overallRoot, result, "");
4         return result;
5     }
6
7     private void methodHelper(TreeNode root, List<String> result, String s) {
8         if (root != null) {
9             s += root.data;
10            if (root.left == null && root.right == null) {
11                result.add(s);
12            } else {
13                s += ", ";
14                methodHelper(root.left, result, s);
15                methodHelper(root.right, result, s);
16            }
17        }
18    }
19
```

Provide a tree that, if called by the above method, would result in a list of size 3.



2. Code Tracing

Part A: For each of the following, write the code necessary to convert the following sequences of ListNode objects:

Before: `list -> [5] -> [4] -> [3] /`	**After:** `list -> [4] -> [5] -> [3] /`

```
ListNode temp = list.next;
list.next = list.next.next;
temp.next = list;
list = temp;
```

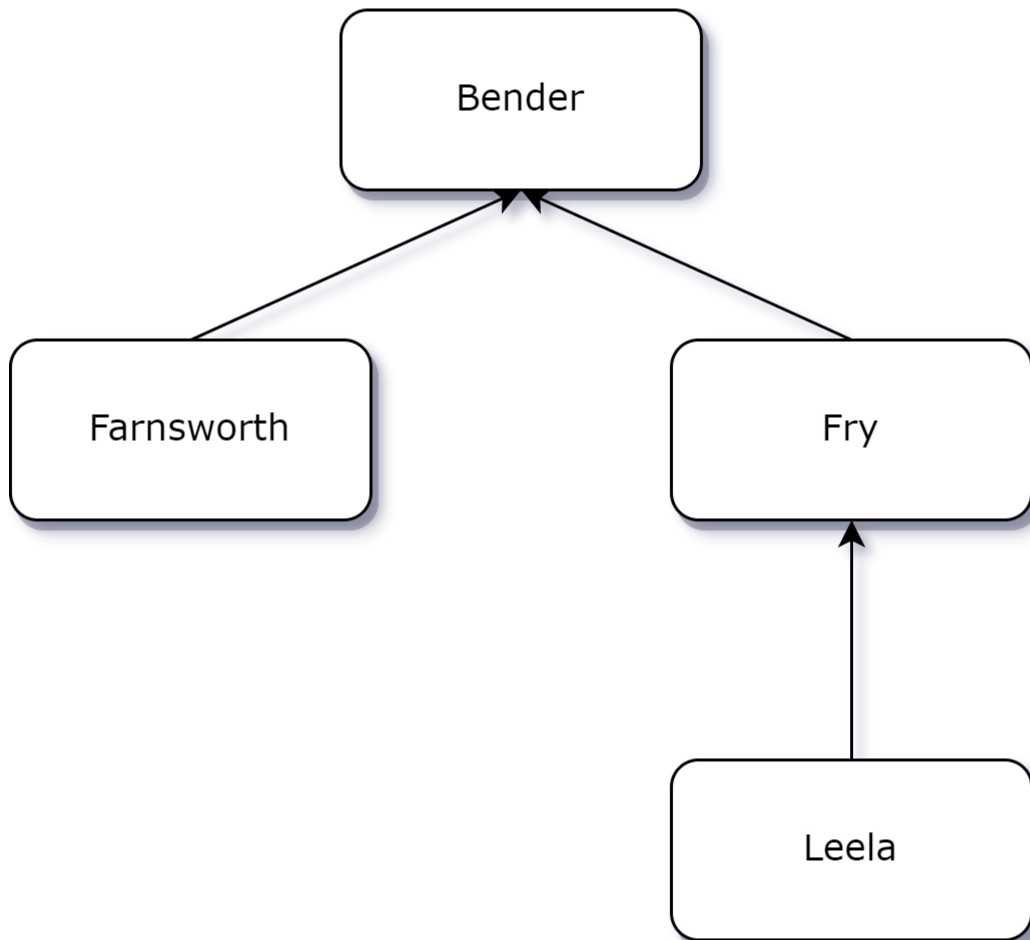
Before:
`list  -> [1] -> [2] /`
`list2 -> [3] -> [4] /`
	After:
	`list  -> [4] -> [1] /`
	`list2 -> [2] -> [3] /`

```
list.next.next = list2;
list2.next.next = list;
list = list2.next;
list2 = list.next.next;
list.next.next = null;
list2.next.next = null;
```

Part B: Consider the following classes:

```
public class Leela extends Fry {
    public void method1() {
        System.out.print("Leela1 ");
    }

    public void method2() {
        System.out.print("Leela2 ");
        super.method2();
    }
}

public class Farnsworth extends Bender {
    public void method1() {
        System.out.print("Farnsworth1 ");
    }

    public String toString() {
        return "Good news everyone!";
    }
}
```

```
public class Fry extends Bender {
    public void method2() {
        System.out.print("Fry2 ");
        super.method2();
    }
}

public class Bender {
    public void method1() {
        System.out.print("Bender1 ");
    }

    public void method2() {
        System.out.print("Bender2 ");
        method1();
    }

    public String toString() {
        return "We're doomed!";
    }
}
```



Given the classes above, what output is produced by the following code?

```
Bender[] rodriguez = {new Leela(), new Bender(), new Farnsworth(), new Fry()};
for (int i = 0; i < rodriguez.length; i++) {
    rodriguez[i].method2();
    System.out.println();
    System.out.println(rodriguez[i]);
    rodriguez[i].method1();
    System.out.println();

}
```

Indicate what the output of each of the following statements would be.

`elements[0]: Leela`	`Leela2 Fry2 Bender2 Leela1`
	`We're doomed!`
	`Leela1`

`elements[1]: Bender`	`Bender2 Bender1`
	`We're doomed!`
	`Bender1`

`elements[2]: Farnsworth`	`Bender2 Farnsworth1`
	`Good news everyone!`
	`Farnsworth1`

`elements[3]: Fry`	`Fry2 Bender2 Bender1`
	`We're doomed!`
	`Bender1`

3. Linked List Debugging

Consider a method in the `LinkedIntList` class called **`removeFromEnd`** that takes an integer as a parameter and removes the nth node from the end of the list.

For example, suppose the variable list1 contains the following list:

```
list1 = [1, 8, 9, 3, 12]
```

After the call `list1.removeFromEnd(2)` executes, the variables `list1` would contain the following list:

```
list1 = [1, 8, 9, 12]
```

Consider the following buggy implementation of **`removeFromEnd`**:

```
1      public void removeFromEnd(int n) {
2          ListNode curr = front;
3          for (int i = 0; i < n; i++) {
4              curr = curr.next;
5          }
6
7          if (curr == null) {
8              front = front.next;
9          } else {
10             ListNode temp = front;
11             while (curr != null) {
12                 curr = curr.next;
13                 temp = temp.next;
14             }
15             temp.next = temp.next.next;
16         }
17     }
```

(continued on next page…)

This implementation contains a *single* bug that is causing it to not work as intended.

Part A: Identify the single line of code that contains the bug. Write your answer in the box to the right as a single number.

11

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. However, the fix should not require a lot of work.

```
1       public void removeFromEnd(int n) {
2           ListNode curr = front;
3           for (int i = 0; i < n; i++) {
4               curr = curr.next;
5           }
6
7           if (curr == null) {
8               front = front.next;
9           } else {
10              ListNode temp = front;
11              while (curr.next != null) { // Not stopping one early
12                  curr = curr.next;
13                  temp = temp.next;
14              }
15              temp.next = temp.next.next;
16          }
17      }
```

4. Inheritance Programming

You have been asked to extend a pre-existing class `Student` that represents a college student. A student has a name, a year (such as 1 for freshman and 4 for senior), and a set of courses that he/she is taking. The `Student` class is as follows:

```
public class Student {
	private String name;
	private int year;
	private Set<String> courses;

	public Student(String name, int year) {
		this.name = name;
		this.year = year;
		this.courses = new HashSet<>();
	}

	public void addCourse(String name) {
		courses.add(name);
	}

	public void dropAll() {
		courses.clear();
	}

	public int getCourseCount() {
		return courses.size();
	}

	public String getName() {
		return name;
	}

	public int getYear() {
		return year;
	}
}
```

You are to define a new class called **`GradStudent`** that extends `Student` through inheritance. A **`GradStudent`** should behave like a `Student` except for the following differences:

- A grad student keeps track of a research advisor, which is a professor working with the student.
- Grad students are considered to be 4 years further ahead than typical students. So, for example, a grad student in year 1 of grad school is really in year 5 of school overall.
- Grad students can enroll in a maximum of 3 courses at a time. If a grad student tries to add additional courses beyond 3, the course is not added to the student's set of courses.
- If grad students work too much, they become "burnt out." A burnt-out student is one who is in his/her 5th or higher year of grad school (9th or higher year of school overall) or one who is taking 3 courses.

You should provide the same methods as the superclass, as well as the following new behavior.

Constructor/Method	**Description**
`public GradStudent(String name, int year, String advisor)`	Constructs a graduate student with the given name, given year (where 1 is the first year of grad school, 5th year of school overall; etc.), and research advisor
`public String getAdvisor()`	Returns this grad student's research advisor
`public boolean isBurntOut()`	Returns true if grad student is "burnt out" (in at least the 5th year of grad school, and/or taking 3 courses)

Write your solution to problem #4 here:

```
public class GradStudent extends Student {
    private String advisor;

    public GradStudent(String name, int year, String advisor) {
        super(name, year + 4);
        this.advisor = advisor;
    }

    public void addCourse(String name) {
        if (super.getCourseCount() < 3) {
            super.addCourse(name);
        }
    }

    public String getAdvisor() {
        return this.advisor;
    }

    public boolean isBurntOut() {
        return super.getCourseCount() >= 3 || super.getYear() >= 9;
    }
}
```

5. Recursive Programming

Write a recursive method called **`groupChars`** that takes a string as a parameter and that returns a new string obtained by inserting parentheses and brackets so as to group the characters. For strings with only one or two characters, the characters should be surrounded by square brackets. For example, **`groupChars("in")`** should return "[in]" and **`groupChars("a")`** should return "[a]". For strings with more than 2 characters, parentheses should be inserted that surround and separate individual characters with the center-most characters surrounded by square brackets. The table below includes more examples.

Call	Return
`groupChars(“the”)`	`"(t[h]e)"`
`groupChars(“rain”)`	`"(r[ai]n)"`
`groupChars(“in”)`	`"[in]"`
`groupChars("Spain")`	`"(S(p[a]i)n)"`
`groupChars("falls")`	`"(f(a[l]l)s)"`
`groupChars("mainly")`	`"(m(a[in]l)y)"`
`groupChars("recursively!")`	`"(r(e(c(u(r[si]v)e)l)y)!)"`
`groupChars("")`	`"*"`

Notice that the method might be passed an empty string, in which case it returns a string composed of a single asterisk. You are not allowed to construct any structured objects other than `Strings` (no array, `List`, `Scanner`, etc.) and you may not use any loops to solve this problem; you must use recursion.

Write your solution to problem #5 here:

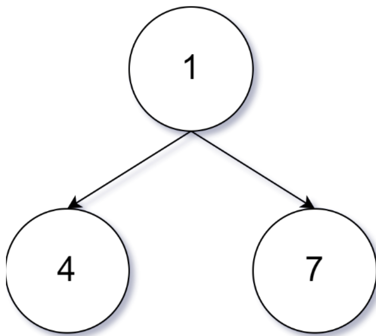
```
public String groupChars(String s) {
     if (s.length() == 0) {
         return "*";
     } else if (s.length() < 3) {
         return "[" + s + "]";
     } else {
         int last = s.length() - 1;
         return "(" + s.charAt(0) + groupChars(s.substring(1, last)) +
             s.charAt(last) + ")";
     }
 }
```

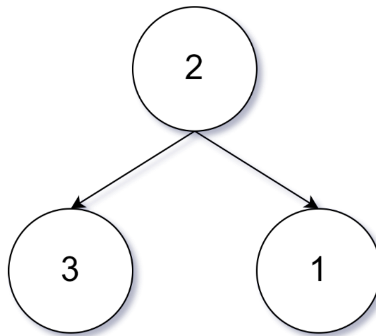
6. Binary Tree Programming

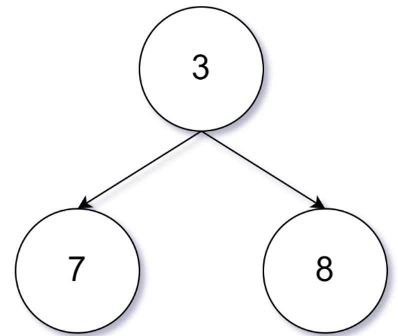
Write a method called **add** that takes as a parameter a reference to a second binary tree and that adds the values in the second tree to this tree. If the method is called as follows:

```
tree1.add(tree2);
```

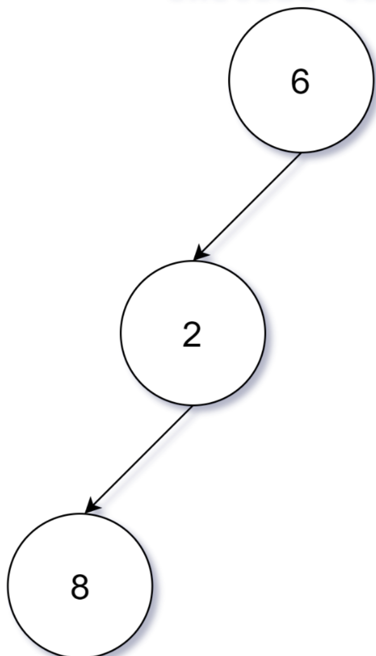
it should add all values in `tree2` to the corresponding nodes in `tree1`. In other words, the value stored at the root of `tree2` should be added to the value stored at the root of `tree1` and the values in `tree2`'s left and right subtrees should be added to the corresponding positions in `tree1`'s left and right subtrees. The values in `tree2` should not be changed by your method.

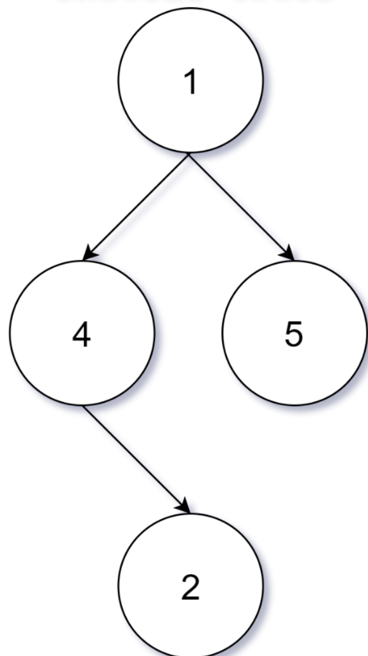


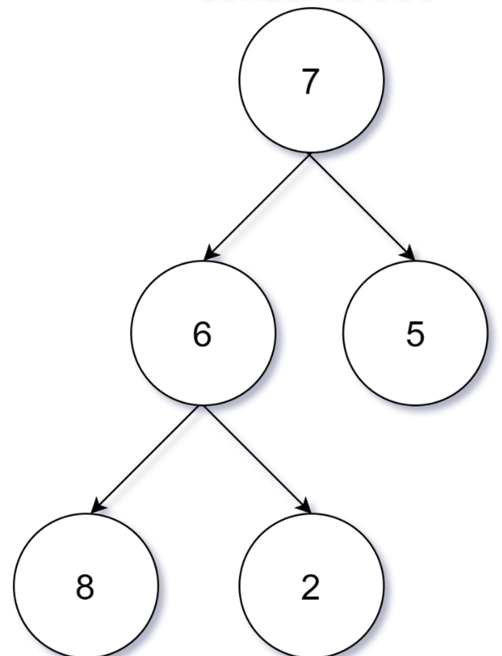




If `tree1` has a node that has no corresponding node in `tree2`, then that node is unchanged. For example, if `tree2` is empty, `tree1` is not changed at all. It is also possible that `tree2` will have one or more nodes that have no corresponding node in `tree1`. For each such node, create a new node in `tree1` in the corresponding position with the value stored in `tree2`'s node. For example:







Write your solution to problem #6 here:

```
public void add(IntTree other) {
     overallRoot = add(overallRoot, other.overallRoot);
 }

public IntTreeNode add(IntTreeNode root1, IntTreeNode root2) {
     if (root2 != null) {
           if (root1 == null) {
                 root1 = new IntTreeNode(0);
           }
           root1.data += root2.data;
           root1.left = add(root1.left, root2.left);
           root2.left = add(root1.right, root2.right);
     }
     return root1;
}
```

This page intentionally left blank for scratch work