CSE 123 Spring 2024 Practice Final Exam #2

1. Code Comprehension

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

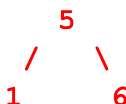
- ☐ In Java, it is possible to inherit from multiple superclasses (i.e. `public class A extends B, C`).
- ☑ **It is impossible for a subclass to directly access fields / methods declared `private` within a superclass.**
- ☑ **Overriding is when a subclass implements its own version of a superclass method.**
- ☐ Overloading is when a subclass implements its own version of a superclass method.
- ☑ **Inheritance supports both polymorphism and code reuse**

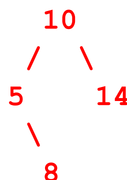
Part B: Consider that we added the following method in the `IntSearchTree` class. Currently, its comment does not perfectly match its behavior:

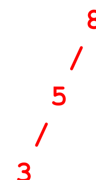
```
1       // Finds and returns the largest value that's smaller than the provided 'x'
2       // within this binary search tree. If there is no such value, returns -1
3       public int largestSmallerValue(int x) {
4           return largestSmallerValue(overallRoot, x);
5       }
6
7       private int largestSmallerValue(IntTreeNode root, int x) {
8           if (root == null) {
9               return -1;
10          } else if (root.left == null && root.right == null) {
11              return root.data;
12          } else if (root.data <= x) {
13              return largestSmallerValue(root.right, x);
14          } else if (root.data > x) {
15              return largestSmallerValue(root.left, x);
16          }
17      }
```

Draw a **binary search tree** containing only positive integers such that, if it were stored in the variable `tree`, the call `tree.largestSmallerValue(5)` would return a value that does not comply with the comment (i.e., it returns an integer that is not the largest value smaller than 5 within the search tree).

A single node tree that contains a value > 5 *OR*
Any valid BST that contains the value 5 and
a non-null subtree on the right of 5 *OR*
a null subtree on the right of 5 and a non-null subtree on the left of 5
Some examples:







Part C: Consider the following scenario:

In preparation for your free time over Summer break, you and a friend decided to start working on an online platform in which users can review their favorite restaurants. After a quick discussion of how a restaurant should be represented, you settled on the following fields:

```
1  private String name;
2  private String cuisineType;
3  private Set<Review> reviews;
4
5  // Returns the average review rating for this restaurant
6  public double getAverageRating()
```

On your new platform, you want to be able to provide users an ordered collection of restaurants that they should consider visiting based on the information you have stored, favoring less-visited restaurants. Being a near 123 graduate, you recognized the potential of using the **`Comparable`** interface to define an ordering between restaurants (where the first element after sorting would be considered the restaurant a user should most consider visiting). You tell this to your collaborator who suggests the following implementation for the compareTo method:

```
1  public int compareTo(Restaurant other) {
2      if (this.getAverageRating() > other.getAverageRating()) {
3          return -1;
4      } else if (this.getAverageRating() < other.getAverageRating()) {
5          return 1;
6      } else {
7          if (other.reviews.size() == this.reviews.size()) {
8              return this.cuisineType.compareTo(other.cuisineType);
9          }
10         return other.reviews.size() - this.reviews.size();
11     }
12 }
```

Write an appropriate method comment for the above `compareTo` implementation.

Compares two Restaurants first by their average rating (higher rating comes first) then by total number of reviews (more reviews comes first) and then by the type of cuisine (alphabetically).

Describe a potential concern a client of this platform 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 a _client_, not the implementer.

Any reasonable answer is ok here. Including:

1. **Positive feedback loop in which higher reviewed / visited restaurants are prioritized (meaning less reviewed restaurants will be forgotten and not given a chance)**
2. **Breaking ties by cuisineType (not really related to how order should be determined)**
3. **User-determined comparison is going to negatively impact restaurants differently (bias of users could translate into bias of our recommendations)**

2. Code Tracing

Part A: For each of the following, draw a reference diagram of the LinkedNodes referenced by **q** after the provided code snippet has been executed. An example has been provided in red below. Only draw the reference q and the nodes it connects to. Do **not** draw nodes that are garbage collected, references for variables created within the provided code, nodes that are referenced exclusively by p, or any intermediate steps.

<table>
<tr><td>Before:
q -> [1] -> [2] /</td><td>Code:
q.next = null;</td></tr>
<tr><td colspan="2">After:
q -> [1] /</td></tr>
<tr><td>Before:
p -> [5] -> [4] -> [3] /
q -> [2] -> [1] /</td><td>Code:
p.next.next.next = q;
q = p;
p = p.next.next;
q.next.next = null;</td></tr>
<tr><td colspan="2">After:
q -> [5] -> [4] /</td></tr>
<tr><td>Before:
p -> [1] -> [2] /
q -> [3] -> [4] /
r -> [5] -> [6] /</td><td>Code:
ListNode temp = q.next;
q.next = r;
r.next.next = p;
p.next.next = temp;
temp = p;
p = q.next.next.next.next.next;
temp.next = null;</td></tr>
<tr><td colspan="2">After:
q -> [3] -> [5] -> [6] -> [1] /</td></tr>
</table>

Part B: Consider the following classes:

```
public class Fridge extends Appliance {
    private double temp;

    public Fridge(double temp) {
        this.temp = temp;
    }

    public String toString() {
        return this.temp + " - " +
               super.toString();
    }
}

public class Timer extends Appliance {
    public String toString() {
        if (this.isOn())
            return "BEEP!";
        else
            return super.toString();
    }
}

public class Clock extends Timer {
    public String toString() {
        return "00:00 - " + super.toString();
    }
}
```

```
public class Appliance {
    private boolean on;

    public Appliance() {
        this.on = false;
    }

    public boolean isOn() {
        return this.on;
    }

    public void switch() {
        this.on = !this.on;
    }

    public String toString() {
        if (this.on)
            return "On";
        else
            return "Off";
    }
}
```

Using the classes above, fill in the **Freezer** class such that the following output is printed to the console:

```
Freezer f1 = new Freezer(-0.5, true);
Freezer f2 = new Freezer(-0.8, false);

System.out.println(f1);
f2.switch();
System.out.println(f2);
f1.makeIce();
f2.makeIce();
```

Console:

```
-0.5 - Off

-0.8 - On
Making ice
Can't make ice
```

```
public class Freezer extends ____________Fridge_______________ {

    private boolean ice;

    public Freezer(double temp, boolean ice) {

        super(temp)_________________________;

        this.ice = ice______________________;

    }

    public void makeIce() {
        if (ice)
            System.out.println("Making ice")____;
        else
            System.out.println("Can't make ice");
    }

}
```

Part C: Given the following array, indicate what the output of each of the following statements would be, or fill in the box for the appropriate error thrown.

```
Appliance[] elements = { new Appliance(), new Timer(), new Clock(),
                         new Fridge(35.6), new Freezer(0.1, false) };
```

Note that you should be **either** filling in the appropriate error thrown **or** writing out the console output. You should not be doing both. If there is an output, the number of lines will match the number of answer spaces provided.

Code	Error / Output		
	Compiler Error	Runtime Error	Output
`System.out.println(elements[0]);`	☐	☐	Off

`System.out.println(elements[1]);`
`elements[1].switch();`
`System.out.println(elements[1]);`
	☐	☐	Off
			BEEP!

`System.out.println(elements[2]);`
`elements[2].switch();`
`System.out.println(elements[2]);`
	☐	☐	00:00 - Off
			00:00 - BEEP!

`elements[2].makeIce();`	X	☐	
`((Freezer)elements[2]).makeIce();` `System.out.println(elements[2]);`	☐	X	
`System.out.println(elements[3]);`	☐	☐	35.6 - Off
`elements[4].makeIce();`	X	☐	
`((Freezer)elements[4]).makeIce();`	☐	☐	Can’t make ice

3. Linked List Debugging

Consider a method in the **LinkedIntList** class called **surroundWith(int x, int y)** that surrounds all instances of **x** with two **y** values. For example, suppose the variable **list** contains the following:

```
list = [0, 1, 2, 1]
```

Then, after a call to **list.surroundWith(1, 4)** is made, the list would then store the elements:

```
list = [0, 4, 1, 4, 2, 4, 1, 4]
```

If the list is empty or **x** doesn't appear in the list at all, then the list should not be changed by your method. You must preserve the original order of the elements of the list.

Consider the following implementation of **surroundWith**:

```
1    public class LinkedIntList {
2        private ListNode head;
3        private int size;
4
5        [...]
6
7        public void surroundWith(int x, int y) {
8            ListNode curr = front;
9            if (front != null && front.data == x) {
10               front.next = new ListNode(y, front.next);
11               front = new ListNode(y, front);
12               curr = curr.next;
13               size += 2;
14           }
15           while (curr != null && curr.next != null) {
16               if (curr.next.data == x) {
17                   curr.next.next = new ListNode(y, curr.next.next);
18                   curr.next = new ListNode(y, curr.next);
19                   size += 2;
20               }
21               curr = curr.next;
22           }
23       }
24   }
```

(Continued on following page)

Part A: When reviewing this implementation, you discover that the implementation 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. Fill in the following boxes with values that would reveal the bug when executed.

```
list = [0, 14, [ 1 ], 8]
list.surroundWith(1, [ any ])
```

Assuming the method was implemented correctly, what should the expected output of the variable **`list`** be after the method call from above?

```
list = [0, 4, 7, [ above ], [ 1 ], [ above ], 8]
```

If you ran the implementation above on this specific case, what type of issue would you encounter?

Incorrect output	NullPointerException	Infinite Loop
[ ]	[ ]	[X]

Part B: You discover that the bug actually only requires a single line to change! Fill in the following solution with the fix that would make the solution work on the test case above.

```
1  public void surroundWith(int x, int y) {
2      ListNode curr = front;
3      if (front != null && front.data == x) {
4          front.next = new ListNode(y, front.next);
5          front = new ListNode(y, front);
6          curr = curr.next;
7          size += 2;
8      }
9      while (curr != null && curr.next != null) {
10         if (curr.next.data == x) {
11             curr.next.next = new ListNode(y, curr.next.next);
12             curr.next = new ListNode(y, curr.next);
13             curr = curr.next.next;______________________;
14             size += 2;
15         }
16         curr = curr.next;
17     }
18 }
```

4. Programming Translation

Below is an iterative solution to **`primeFactors(int n)`**. Your goal is to translate it into a recursive solution. Prime factorization of a number is finding the prime numbers that multiply together to result in that number. Consider the following examples:

Function call	Printed Output
primeFactors(1)	
primeFactors(4)	2*2*
primeFactors(17)	17*
primeFactors(24)	2*2*2*3*
primeFactors(315)	3*3*5*7*

The following is the iterative solution to **`primeFactors(int n)`**. Consider how the given examples above fit with the given solution:

```
1    // this function prints out all the prime factors of the input n
2    public static void primeFactors(int n) {
3        int factor = 2;
4        while (factor <= n) {
5            if (n % factor == 0) {
6                System.out.print(factor + "*");
7                n /= factor;
8            } else {
9                factor++;
10           }
11       }
12   }
```

Write the recursive version of this problem **`primeFactors(int n)`** that prints out all the prime factors of the integer n. You may define private helper methods to solve this problem. You may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc).

Write your solution to problem #4 here:

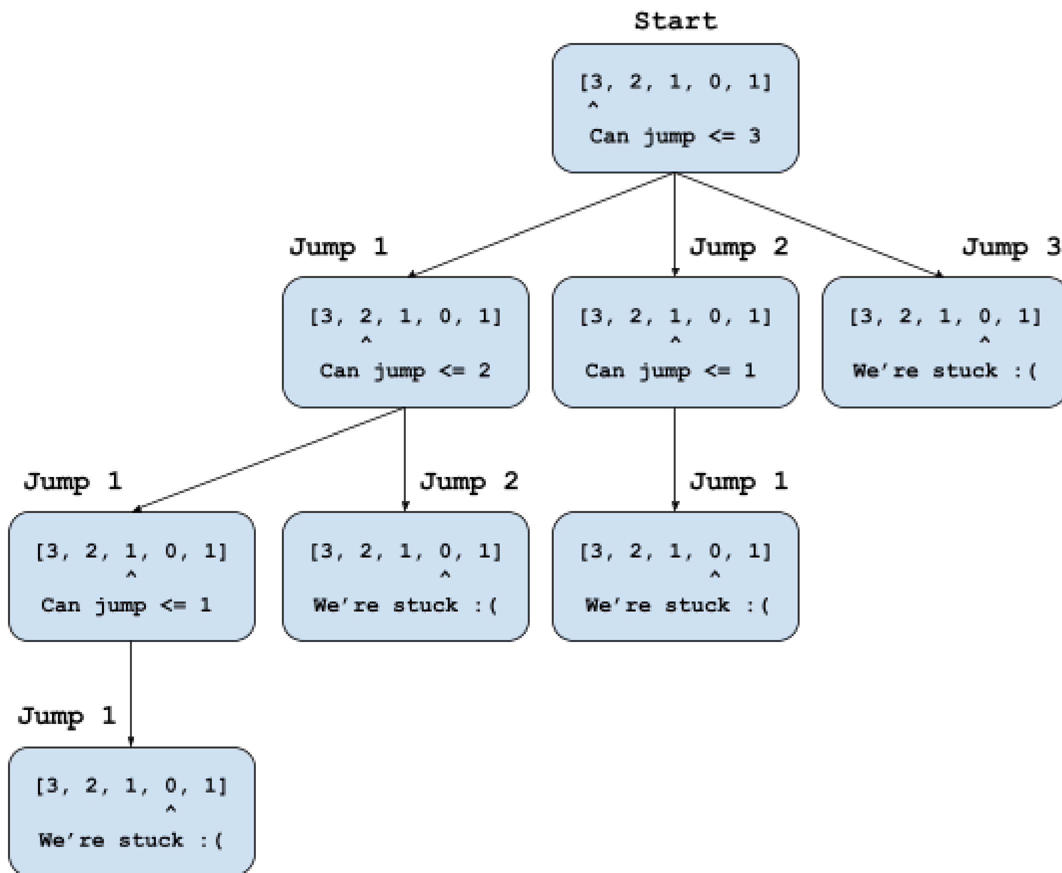
```
// One possible solution:
public static void primeFactors(int n) {
    primeFactors(n, 2);
}

private static void primeFactors(int n, int factor) {
    if (factor <= n) {
        if (n % factor == 0) {
            System.out.print(factor + "*");
            primeFactors(n / factor, factor);
        } else {
            primeFactors(n, factor + 1);
        }
    }
}
```

5. Recursive Programming

Write an implementation for the method **canLeapfrog(int[] arr)** which returns whether or not it's possible to win a game of "leapfrog" on the provided array. In the game, each value in the array represents the maximum number of spots you can jump forwards from your current position. The game is considered winnable if you can jump past the very last element of the array. An example of a winnable game `[1, 2, 0, 1]` in which **canLeapfrog** should return **true** is as follows:

Start:	**Choice 1:**	**Choice 2:**	**Choice 3:**
`[1, 2, 0, 1]`	`[1, 2, 0, 1]`	`[1, 2, 0, 1]`	`[1, 2, 0, 1]`
`^` (under 1st element)	`^` (under 2nd element)	`^` (under 4th element)	`^` (past the last element)
Can jump <= 1 Jump 1 time	Can jump <= 2 Jump 2 times	Can jump <= 1 Jump 1 time	We won!

However, `[3, 2, 1, 0, 1]` would not be considered winnable. No matter what decisions are made at each point, it's impossible to jump past the 0 value and thus **canLeapfrog** should return **false**. Below is a diagram of all the possible options chosen at each point to demonstrate that the game is not winnable in this case.



Your implementation for **canLeapfrog** must be recursive. You may define private helper methods to solve this problem. You may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). Write your implementation to **canLeapfrog(int[] arr)** on the following page.

Write your solution to problem #5 here:

```
// One possible recursive solution
public static boolean canLeapfrog(int[] arr) {
    return canLeapfrog(arr, 0);
}

private static boolean canLeapfrog(int[] arr, int i) {
    if (i >= arr.length) {
        return true;
    } else {
        for (int j = 1; j <= arr[i]; j++) {
            if (canLeapfrog(arr, i + j)) {
                return true;
            }
        }
        return false;
    }
}
```

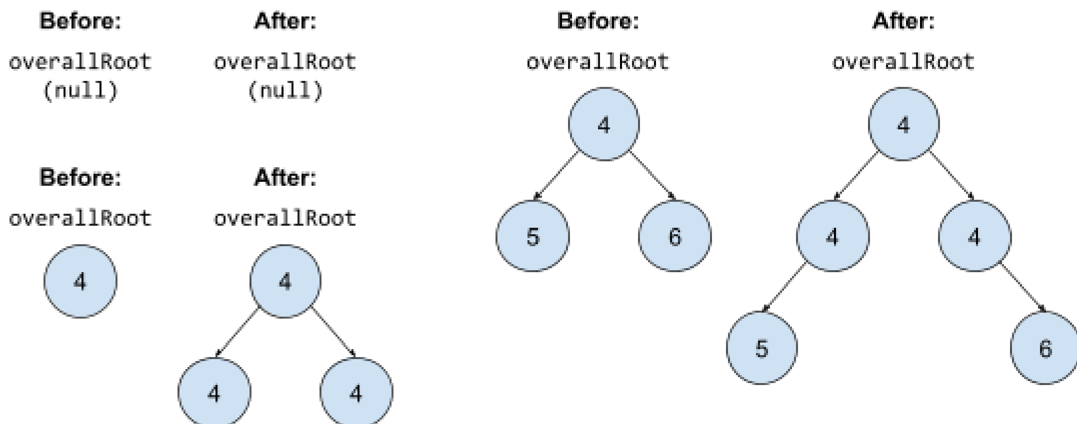
6. Binary Tree Programming

Write an implementation for **expand(int x)**, a method within **IntTree** class defined below:

```
1    public class IntTree {
2        private IntTreeNode overallRoot;
3
4        [...]
5
6        private static class IntTreeNode {
7            public final int data;
8            public IntTreeNode left;
9            public IntTreeNode right;
10
11           public IntTreeNode(int data) {
12               this(data, null, null);
13           }
14
15           public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {
16               this.data = data;
17               this.left = left;
18               this.right = right;
19           }
20       }
21   }
```

expand should append two new children nodes to each node storing **x** within the original tree. These new children nodes should also store **x** and any previous children should become grandchildren of the original node (`.left` should become the new left child's left child and `.right` should become the new right child's right child). Below are diagrams demonstrating the expected changes after calling **expand(4)** on 3 different starting trees:

expand(4)



You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the **IntTree** class. You may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). Note that the `data` field within `IntTreeNode` is final - you are unable to change the data field of an existing node.

Write your solution to problem #6 here:

```
// One possible solution
public void expand(int x) {
    overallRoot = expand(overallRoot, x);
}

private IntTreeNode expand(IntTreeNode root, int x) {
    if (root != null) {
       // Post-order traversal matters! Otherwise we expand the new
       // children infinitely
       root.left = expand(root.left, x);
       root.right = expand(root.right, x);

       if (root.data == x) {
           root.left = new IntTreeNode(x, root.left, null);
           root.right = new IntTreeNode(x, null, root.right);
       }
   }
   return root;
}
```

This page intentionally left blank for scratch work