1. Concepts

Part A - True/False

Say whether the statements below are true or false. You may abbreviate true as T and false as F.

Statement	**True/False**
James likes drawing pictures.	T
James is good at drawing pictures.	T or F is ok
The `compareTo` method of `Comparable` must return -1, 0, or 1.	F
When testing, you should run the code first to see the output, then check if it looks right.	T or F is ok
In Java, you can `extend` multiple classes, but `implement` only one interface.	F
Run time happens before compile time.	F
`super()` can only be called as the first line of a method.	T or F is ok

Consider the following code:
`public static void m1() {`
`String x = "";`
`m2(x);`
`System.out.println(x.length());  // (***)`
`}`
`public static void m2(String y) { /* unknown code here */ }`
True or False: if the print statement on the line marked (***) runs, it is guaranteed to print 0.
	T

ArrayIntList.get(index) and LinkedIntList.get(index) are in the same complexity class.	F
Modifying a LinkedIntList requires writing to the `front` field or to a `next` field.	T
In a recursive method, there is at least one line of code representing the base case.	F
An iterative implementation will always have a faster complexity class than a recursive one.	F
It is impossible to traverse a binary tree without using recursion.	F
It is impossible to modify a binary tree without using x=change(x).	F
When a recursive method calls itself, a new call frame is pushed on the call stack.	T
When one non-recursive method calls another non-recursive method, no call frame is pushed on the call stack.	F
In the worst case, searching a data structure for an element always requires looking at all the data in the data structure.	F
Using a binary tree is always faster than using an array.	F

Part B - Run-time Analysis

Analyze the worst-case running time of each method below. Choose the *most accurate* (fastest) correct running time, if more than one choice is correct. Unless otherwise stated, *n* is the length of the input data structure.

Code	**O(1)**	**O(log(n))**	**O(*n*)**	**O(n^2)**

`public void m(int[] data) {`
`  for (int i = data.length - 1; i > 0; i -= 2) {`
`    System.out.println(data[i]);`
`  }`
`}`
			X	

`public void m(int[] data) {`
`  for (int i = 0; i < 2147483647; i++) {`
`    System.out.println(data[i]);`
`  }`
`}`
	X			

`public void m(int[] data) {`
`  int n = data.length;`
`  for (double x = 0; x < n; x += 1.0 / n) {`
`    System.out.println(data[(int)x]);`
`  }`
`}`
				X

`public void m(int[] data) {`
`  for (int i = data.length - 1; i > 0; i /= 2) {`
`    System.out.println(data[i]);`
`  }`
`}`
		X		

Searching for a value in an array.			X	
Searching for a value in a sorted array.		X		
Searching for a value in a linked list.			X	
Searching for a value in a sorted linked list.			X	
Searching for a value in a binary tree.			X	
Searching for a value in a binary search tree.			X	
Searching for a value in a balanced binary search tree.		X		
Searching for a value in a hash table.	X		X	

(For the hash table question, either O(1) or O(n) is accepted.)

2. Inheritance and Polymorphism

Consider the following code:

```
public interface Food {
  public String name();
}

public abstract class Fruit
                implements Food {
  public String name() {
    return fruitName() + " fruit";
  }
  public abstract String fruitName();
}
```

```
public class Apple extends Fruit {
  public String fruitName() {
    return "Apple";
  }
  public boolean isRed() { return true; }
}
public class Orange extends Fruit {
  public String fruitName() {
    return "Orange";
  }
  public String name() { return "James"; }
  public int slices() { return 8; }
}
```

Mark the appropriate option in each row below, according to whether the code will have a run-time error, a compile-time error, or no error. If the code has no error, also say what it will print, if anything. If the code has any kind of error, you do not need to say what the output is.

Each row is separate. Do not consider the code of previous rows when looking at the next row.

CE = Compile-time Error, RE = Run-time Error, NE = No Error

Code	CE	RE	NE	Output
`Food x = new Apple();` `System.out.println(x.name());`			X	Apple fruit
`Fruit x = new Apple();` `System.out.println(x.name());`			X	Apple fruit
`Apple x = new Apple();` `System.out.println(x.name());`			X	Apple fruit
`Food x = new Apple();` `x = new Orange();`			X	(none)
`Apple x = new Apple();` `x = new Orange();`	X			
`Apple x = new Apple();` `x = (Orange)new Orange();`	X			
`Apple x = new Apple();` `x = (Apple)new Orange();`	X			

Code	CE	RE	NE	Output
`Apple x = new Apple();` `System.out.println(x.slices());`	X			
`Apple x = new Apple();` `Orange y = (Orange) x;`	X			
`Fruit x = new Apple();` `Orange y = (Orange) x;`		X		

`Food x = new Apple();`
`x = new Orange();`
`Orange y = (Orange) x;`
			X	(none)

`Food x = new Orange();` `System.out.println(x.name());`			X	James
`Fruit x = new Orange();` `System.out.println(x.name());`			X	James
`Orange x = new Orange();` `System.out.println(x.name());`			X	James
`Food x = new Apple();` `System.out.println(x.fruitName());`	X			
`Fruit x = new Apple();` `System.out.println(x.fruitName());`			X	Apple
`Orange x = new Apple();` `System.out.println(x.fruitName());`	X			
`Food[] foods = { new Apple(), new Orange() };` `System.out.println(foods[0].isRed());`	X			
`Food[] foods = { new Apple(), new Orange() };` `System.out.println(foods[1].isRed());`	X			

`Food[] foods = { new Apple(), new Orange() };`
`Apple x = foods[0];`
`System.out.println(x.isRed());`
	X			

`Food[] foods = { new Apple(), new Orange() };`
`Orange y = (Orange)foods[0];`
`System.out.println(y.slices());`
		X		

`Food[] foods = { new Apple(), new Orange() };`
`Orange y = (Orange)foods[1];`
`System.out.println(y.slices());`
			X	8

3. Data structure design

Part A - Initial design and implementation

You are asked to create a new data structure called `IntQueue` that represents a queue of integers. It is not important to be super familiar with queues to do well on this problem. A queue is a list where you can only add to the end of the list and you can only remove from the beginning of the list. It is similar to how people wait in line at the coffee shop. New customers are added to the end of the line, while the customer at the front of the line is served next by the worker.

Suppose we have the following starter code for implementing the IntQueue using an ArrayIntList.

```
public class IntQueue {
    private IntList list;  // Stores the queue elements in front-to-back order.

    public IntQueue() {
        this.list = new ArrayIntList();
    }
}
```

Fill in the methods below to finish the implementation. **You may assume that all the methods available on Java Lists (shown on the reference sheet) are available on ArrayIntLists and IntLists.** (Since there is ambiguity between remove(index) and remove(value), you can name them removeIndex and removeValue.)

```
// Add the value to the back of the queue.
public void addBack(int value) {
    this.list.add(value);
}
```

```
// Remove and return the value at front of queue. Assume queue is non-empty.
public int removeFront() {
    return this.list.removeIndex(0);
}
```

What is the running time for each method? Mark an X in the appropriate column for each row.

Operation	O(1)	O(log(n))	O(n)	O(n^2)
`addBack`	X		X	
`removeFront`			X	

(For addBack, either O(1) or O(n) is accepted because of resizing.)

Part B: Alternate designs

Your senior engineer has an idea to speed up the IntQueue. They want to store the values internally in the opposite order of how you did it on the previous page. Then, they want to change the implementations of the two methods **`addBack`** and **`removeFront`** so that clients will observe the same behavior.

Describe how we can adjust **`addBack`** and **`removeFront`** so that clients will observe the same behavior as before. (You don't need to write code. Just explain at a high level in 1-2 sentences.)

Since the list is stored in the opposite order, change addBack to add to the front of the list and change removeFront to remove from the end of the list. The client will not know the difference because they can think of the queue as being stored in order.

How will this change affect the runtime of the two IntQueue operations? Mark an X in the appropriate box.

Operation	**O(1)**	**O(log(n))**	**O(*n*)**	**O(n^2)**
`addBack`			X	
`removeFront`	X			

Your senior engineer's boss has a fancy title containing the word "architect". You're not so sure what buildings have to do with queues, but the architect has a different idea for how to speed up the original IntQueue. They claim that if you replace the ArrayIntList with a LinkedIntList, then there will be a way to make both operations faster, either by storing the values in front-to-back order, or in the opposite order.

Is the architect correct in saying that using a LinkedIntList will allow the IntQueue to implement both **`addBack`** and **`removeFront`** to have faster complexity classes than your original ArrayIntList-based implementation? Why or why not?

No. No matter which order we store the items in in a linked list, one of the two operations will be O(n), so both operations cannot be faster than the original.

The following sub-question is just for fun and will not affect your grade:

Do you think it's possible to implement IntQueue using an ArrayIntList or LinkedIntList, or by modifying one of those classes, so that both **`addBack`** and **`removeFront`** run in O(1) time? If so, how? If not, why not?

NOT GRADED

Yes, we can modify the LinkedIntList to also have a reference to the last node in the list. Then we can implement adding to the end of a linked list in O(1) and both queue operations can be implemented in O(1) using this modified linked list. (This is part of why Java's LinkedList can be used as a queue with O(1) operations.)

Another way to do it is to modify ArrayIntList to be a "ring buffer" which supports O(1) add/remove from both ends.

4. Linked Lists

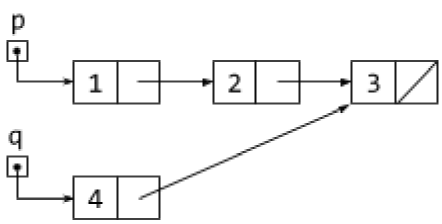
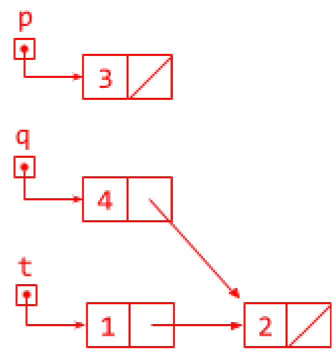
Part A - Tracing

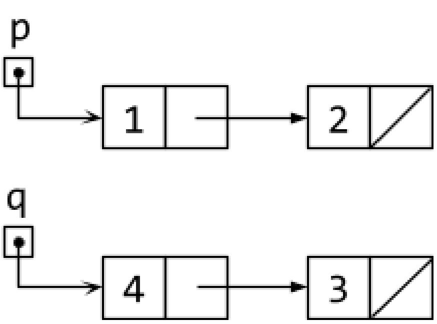
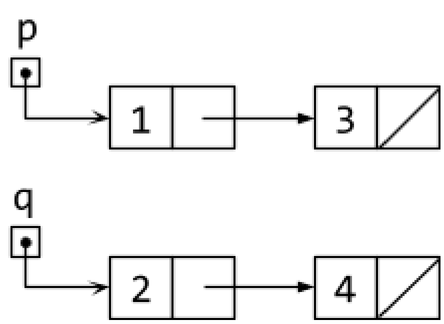
Complete the following table, filling in either the before picture, the code, or the after picture. You should not create any new `ListNodes` or modify any `.data` fields, and **there should be only one instance of each node with a specific value.**

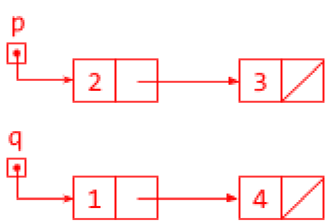
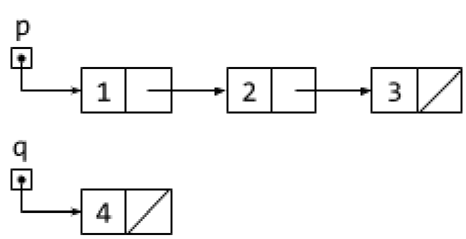
Your drawings must show all temporary references created by the code.

Your `ListNode` diagram format doesn't have to match that of the problem so long as it is clear what you intend. In your code, you may use as many temporary references as you'd like to accomplish your goal.

Before	Code	After

	`q.next = p.next;`
	`ListNode t = p;`
	`p = p.next.next;`
	`t.next.next = null;`
		

	`ListNode t = p.next;`
	`p.next = q.next;`
	`t.next = q;`
	`q = t;`
	`q.next.next = null;`
		

	`q.next.next = p;`
	`p = q;`
	`q = q.next;`
	`p.next = p.next.next;`
	`q.next = null;`
	This code takes the first node of q and puts it at the beginning of p. Then truncates q to just second node.
		

Part B - Debugging

Consider adding this method of the LinkedIntList class:

```
public void mystery(int x) {
    ListNode node = front;
    while (node != null) {
        if (node.next == null) {
            node.next = new ListNode(x);
        }
        node = node.next;
    }
}
```

What do you think the author of this code was trying to do? What does the code actually do?

The author was probably trying to add a node to the end of the list.

One problem is the code loops forever. Another problem is it is missing a front case. (Mentioning one problem is enough.)

Fix the code so that it does what the author intended. **Add at most 6 lines. Do not change or remove any lines.** You can either write out the code again or explain where to make the changes. Do not use recursion.

```
public void mystery(int x) {
    if (front == null) {
        front = new ListNode(x);
    } else {
        ListNode node = front;
        while (node != null) {
            if (node.next == null) {
                node.next = new ListNode(x);
                node = node.next;
            }
            node = node.next;
        }
    }
}
```

Either fixing the front case or the infinite loop (or both) is accepted.

5. Recursive Backtracking

Write a method findSentences with the following specification:

```
// Prints out all sentences consisting of some combination of the given words
// subject to the following rules:
//   - Each sentence is printed on its own line.
//   - A sentence is made up of words separated by a single space (" ").
//   - There is a period (".") at the end of the sentence.
//   - Each word is used at most once. (A word is allowed not to be used at all.)
//   - The total number of characters in the sentence (including spaces and periods)
//     is between minLength (inclusive) and maxLength (inclusive).
public void findSentences(List<String> words, int minLength, int maxLength)
```

For example, consider the following client code:

```
List<String> words = new ArrayList<String>();
words.add("hi");
words.add("world");
words.add("thee");
words.add("Rasmussen");

findSentences(words, 10, 13);
```

A correct implementation will print the following lines. (The lines are allowed to be printed in any order.)

```
hi Rasmussen.
world thee.
thee world.
Rasmussen.
Rasmussen hi.
```

Note:

- "hi world." is not printed because it is too short. (It has 9 characters including spaces and periods, which is less than the minimum of 10 in this example.)
- "hi thee world." is not printed because it is too long. (It has 14 total characters including spaces and periods, which is longer than the maximum of 13 in this example.)
- "Rasmussen." is printed because it has exactly 10 characters (including the period), and 10 is the minimum length (inclusive) in this example.
- "world thee." is printed because it has exactly 11 characters (including the space and period), and 11 is between 10 (inclusive) and 13 (inclusive).
- "hi Rasmussen." is printed because it has exactly 13 characters (including the space and period), and 13 is the maximum length (inclusive).

Write your solution on the next page. Follow these additional rules:

- To get an S or better, your code must be primarily recursive.
  (But using iteration within the recursion *is* allowed.)
- To get an E, you must prune "dead ends" in the search tree when the sentence becomes too long.

```java
public void findSentences(List<String> words, int minLength, int maxLength) {
   findSentences(words, minLength, maxLength, "");
}

private void findSentences(
    List<String> words, int minLength,
    int maxLength, String soFar
) {
    if (soFar.length() <= maxLength) {  // prune dead ends
        String sentence = soFar + ".";
        if (sentence.length() >= minLength &&
            sentence.length() <= maxLength) {
            System.out.println(sentence);
        }
        for (int i = 0; i < words.size(); i++) {
            String next = soFar;
            if (!soFar.equals("")) {
                next += " ";
            }
            String word = words.remove(i);  // choose
            next += word;
            // explore
            findSentences(words, minLength, maxLength, next);
            words.add(i, word);  // unchoose
        }
    }
}
```

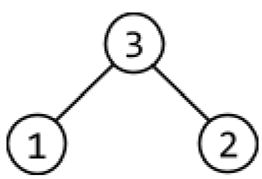
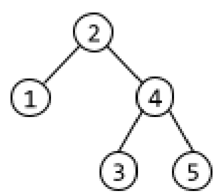
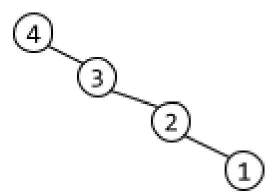
6. Binary Trees

Part A - Tracing

Consider the following code for a method inside the IntTree class:

```
public boolean mystery() {
    return mysteryHelper(overallRoot, Integer.MIN_VALUE, Integer.MAX_VALUE);
}
private boolean mysteryHelper(IntTreeNode currentRoot, int lo, int hi) {
    if (currentRoot == null) {
        return true;
    } else {
        return lo <= currentRoot.data &&
            currentRoot.data <= hi &&
            mysteryHelper(currentRoot.left, lo, currentRoot.data) &&
            mysteryHelper(currentRoot.right, currentRoot.data, hi);
    }
}
```

The constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE` are the smallest and largest possible integers representable by an int in Java. For any int x, the comparison `x >= Integer.MIN_VALUE` returns true.
Say what the method returns on the following trees:

Input tree			
Return value	**F**	**T**	**F**

Explain what the mystery method computes.

The method returns true if the tree is a binary search tree and false otherwise.

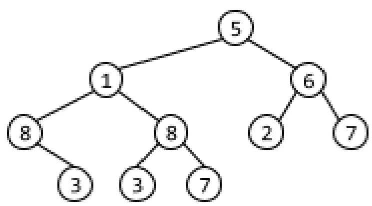
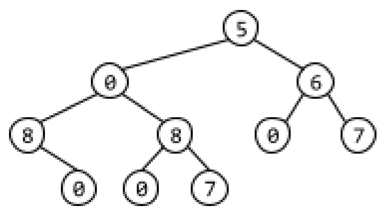
Part B - Modification

If x is a node in a binary tree, define the *depth* of x to be the number of edges on the path from the overall root to x. In the example "Before" tree on the left below, the nodes with data 8 have depth 2, and the node with data 6 has depth 1. The root node (the node with data 5 in the example below) has depth 0.

Write a method of the `IntTree` class with the following specification:

```
// Finds any nodes whose data is equal to that node's depth and replaces each such
// node with a new node that stores 0 as its data instead.
public void zeroDepthNodes()
```

For example, when this method is called on the "Before" tree, it changes it into the "After" tree.

Before	After	Notes
		Notice the four nodes that became 0. They each had a depth equal to their data. All other nodes were left unchanged.

Write your code below. You must use recursion. Remember that the data field is `final`.

```
public void zeroDepthNodes() {
    // x = change(x)
    this.overallRoot = zeroDepthNodes(this.overallRoot, 0);
}
private IntTreeNode zeroDepthNodes(
    IntTreeNode currentRoot, int depth
) {
    if (currentRoot != null) {
        if (currentRoot.data == depth) {
            currentRoot = new IntTreeNode(0,
              currentRoot.left, currentRoot.right);
        }
        currentRoot.left =   // x = change(x)
            zeroDepthNodes(currentRoot.left, depth+1);
        currentRoot.right =  // x = change(x)
            zeroDepthNodes(currentRoot.right, depth+1);
    }
    return currentRoot;
}
```