1. Binary Tree Traversal

Consider the following tree:

```
                    +---+
                    | 4 |
                    +---+
                  /       \
                /           \
          +---+               +---+
          | 1 |               | 9 |
          +---+               +---+
         /     \             /
        /       \           /
     +---+     +---+     +---+
     | 6 |     | 0 |     | 2 |
     +---+     +---+     +---+
    /               \
   /                 \
+---+               +---+
| 3 |               | 8 |
+---+               +---+
     \                   \
      \                   \
     +---+               +---+
     | 7 |               | 5 |
     +---+               +---+
```

Fill in each of the traversals below :

- Pre-order: 4 1 6 3 7 0 8 5 9 2
- In-order: 3 7 6 1 0 8 5 4 2 9
- Post-order: 7 3 6 5 8 0 1 2 9 4

2. Binary Search Tree

Draw a picture below of the binary search tree that would result from inserting the following words into an empty binary search tree in the following order: Facebook, Accenture, Nintendo, Expedia, Amazon, Microsoft, Zillow, Google. Assume the search tree uses alphabetical ordering to compare words.

```
                  +------------+
                  |  Facebook  |
                  +------------+
               /                   \
            /                         \
         /                               \
+------------+                      +------------+
|  Accenture |                      |  Nintendo  |
+------------+                      +------------+
             \                      /              \
              \                    /                \
               \                  /                  \
         +------------+    +------------+       +------------+
         |  Expedia   |    | Microsoft  |       |   Zillow   |
         +------------+    +------------+       +------------+
         /                      /
        /                      /
       /                      /
+------------+         +------------+
|  Amazon    |         |   Google   |
+------------+         +------------+
```

3. Inheritance/Polymorphism Mystery

Consider the following classes:

```
public class Clock extends Bear {
    public void method3() {
        System.out.println("Clock 3");
    }
}

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

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

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

    public void method3() {
        System.out.println("Bear 3");
        super.method3();
    }
}

public class Can {
    public void method2() {
        System.out.println("Can 2");
        method3();
    }

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

and that the following variables are defined:

```
Object var1 = new Bear();
Can var2 = new Can();
Can var3 = new Lamp();
Bear var4 = new Clock();
Object var5 = new Can();
Can var6 = new Clock();
```

In the table below, indicate in the right-hand column the output produced by the statement in the left-hand column. If the statement produces more than one line of output, indicate the line breaks with slashes as in "a/b/c" to indicate three lines of output with "a" followed by "b" followed by "c". If the statement causes an error, fill in the right-hand column with either the phrase "compiler error" or "runtime error" to indicate when the error would be detected .

Statement	**Output**
`var1.method2();`	`compiler error`
`var2.method2();`	`Can 2/Can 3`
`var3.method2();`	`Can 2/Lamp 3`
`var4.method2();`	`Can 2/Clock 3`
`var5.method2();`	`compiler error`
`var1.method3();`	`compiler error`
`var2.method3();`	`Can 3`
`var3.method3();`	`Lamp 3`
`var6.method3();`	`Clock 3`
`((Lamp)var6).method1();`	`runtime error`
`((Can)var1).method1();`	`compiler error`
`((Can)var1).method2();`	`Can 2/Bear 3/Can 3`
`((Bear)var1).method3();`	`Bear 3/Can 3`
`((Clock)var1).method1();`	`runtime error`
`((Clock)var4).method2();`	`Can 2/Clock 3`

4. Recursive Backtracking Programming

Write a method **`printNumbers2`** that takes two integers `n1` and `n2` as parameters and uses recursive backtracking to print out all possible permutations of `n1` 2's and `n2` 5's such that there are no more than two 2's or two 5's in a row.

For example, a call of `printNumbers2(3, 2)` should produce the output

```
22525
22552
25225
25252
25522
52252
52522
```

Notice, for example, that `22255` is not printed because it has three 2's in a row.

As a hint, Section 14 handout had a problem called `printNumbers` that asked you to print all possible permutations of two 2's and two 5's. The solution for `printNumbers` was as follows

```
public static void printNumbers() {
    explore(2, 2, "");
}

private static void explore(int twos, int fives, String number) {
    if (twos == 0 && fives == 0) {
        System.out.println(number);
    } else if (twos >= 0 && fives >= 0) {
        explore(twos - 1, fives, number + "2");
        explore(twos, fives - 1, number + "5");
    }
}
```

You should only have to make a few modifications to the above solution, however, you should write out `printNumbers2` and its private helper method in its entirety.

One possible solution

```
public static void printNumbers(int twos, int fives) {
   explore(twos, fives, "");
}

private static void explore(int twos, int fives, String number) {
   if (twos == 0 && fives == 0) {
      System.out.println(number);
   } else if (twos >= 0 && fives >= 0) {
      if (!number.endsWith("22")) {
         explore(twos - 1, fives, number + "2");
      }
      if (!number.endsWith("55")) {
         explore(twos, fives - 1, number + "5");
      }
   }
}
```

5. Collections Programming

Write a method called `recordDate` that information about a date between two people. For each person, the map records an ordered list of people that person has dated. For example, the map might record these entries for two people

```
Michael => [Ashley, Samantha, Joshua, Brittany, Amanda, Amanda]
Amanda  => [Michael, Daniel, Michael]
```

The dates are listed in reverse order. The list for Michael indicates that he most recently dated Ashley and before that Samantha and before that Joshua, and so on. Notice that he has dated Amanda twice. The list for Amanda indicates that she most recently dated Michael and before that Daniel and before that Michael. All names are stored as string values.

The method takes three parameters: the map, the name of the first person, and the name of the second person. It should record the date for each person and should return what date number this is (1 for a first date, 2 for a second date, and so on). Given the entries above, if we make this call:

```
int n = recordDate(dates, "Michael", "Amanda");
```

The method would record the new date at the front of each list:

```
Michael => [Amanda, Ashley, Samantha, Joshua, Brittany, Amanda, Amanda]
Amanda  => [Michael, Michael, Daniel, Michael]
```

The method would return the value 3 indicating that this is the third date for this pair of people. When someone is first added to the map, you should construct a `LinkedList` object (we use `LinkedList` instead of `ArrayList` because it has fast insertion at the front of the list).

One possible solution

```
public static int recordDate(Map<String, List<String>> dates,
                             String name1, String name2) {
   if (!dates.containsKey(name1)) {
      dates.put(name1, new LinkedList<String>());
   }
   if (!dates.containsKey(name2)) {
      dates.put(name2, new LinkedList<String>());
   }
   dates.get(name1).add(0, name2);
   dates.get(name2).add(0, name1);
   int n = 0;
   for (String s : dates.get(name1)) {
      if (s.equals(name2)) {
         n++;
      }
   }
   return n;
}
```

6. Comparable

Define a class **`Donation`** that represents donations made to organizations. Each `Donation` object keeps track of an amount, the organization the donation was made to and a `boolean` to indicate whether or not the donation was tax-deductible. Your class must have the following public methods:

Member	Description
`public` **`Donation`**`(organization, amount, isDeductible)`	constructs a donation object with the given organization, amount and tax-deductible status
`public String` **`toString`**`()`	returns a String representation of the donation

Your constructor should throw an `IllegalArgumentException` if the amount passed to it is negative or 0.

The `toString` method returns a `String` composed of a dollar sign ($), followed by the amount donated, followed by a colon and the name of the organization. If the donation is tax-deductible, an asterisk (*) is added to the beginning of the `String`. You must exactly reproduce the format of the examples given below.

Make `Donation` objects comparable to each other using the `Comparable<E>` interface. `Donation` objects that are tax-deductible are considered "less" than donations that are not tax-deductible. In other words, all tax-deductible donations go before donations that are not tax-deductible. Then, they are sorted by amount in ascending order, breaking ties by organization in ascending alphabetical order. For example, if the following objects are declared:

```
Donation uw1 = new Donation("University of Washington", 600.75, true);
Donation uw2 = new Donation("University of Washington", 40, true);
Donation sj = new Donation("Snap Judgment", 30, false);
Donation tal = new Donation("This American Life", 40, true);
Donation mc = new Donation("Microphone Check", 99.99, false);
```

Printing them in sorted order would result in the following output:

```
* $40.0: This American Life
* $40.0: University of Washington
* $600.75: University of Washington
$30.0: Snap Judgment
$99.99: Microphone Check
```

(You may also write on the next page.)

6. Comparable (writing space)

One possible solution

```
public class Donation implements Comparable<Donation> {
   private String org;
   private double amount;
   private boolean deductible;

   public Donation(String org, double amount, boolean deductible) {
      if (amount <= 0) {
         throw new IllegalArgumentException();
      }
      this.org = org;
      this.amount = amount;
      this.deductible = deductible;
   }

   public int compareTo(Donation other) {
      if (deductible && !other.deductible) {
         return -1;
      } else if (!deductible && other.deductible) {
         return 1;
      } else if (amount != other.amount) {
         if (amount < other.amount) {
            return -1;
         } else {
            return 1;
         }
      } else {
         return org.compareTo(other.org);
      }
   }

   public String toString() {
      String prefix = "";
      if (deductible) {
         prefix += "* ";
      }
      return prefix + "$" + amount + ": " + org;
   }
}
```

7. Binary Tree Programming

Write a method called `printLevel` that takes an integer n as a parameter and that prints the values at level n from left to right. By definition the overall root is at level 1, its children are at level 2, and so on. The table below shows the result of calling this method on an `IntTree` variable t storing the following tree.

```
         +----+
         | 12 |
         +----+
       /        \
    +----+      +----+
    | 19 |      | 93 |
    +----+      +----+
   /      \           \
+----+   +----+      +----+
| 11 |   | 14 |      | 15 |
+----+   +----+      +----+
```

call	output
`t.printLevel(1)`	nodes at level 1 = 12
`t.printLevel(2)`	nodes at level 2 = 19 93
`t.printLevel(3)`	nodes at level 3 = 11 14 15
`t.printLevel(42)`	nodes at level 42 =

Notice that if there are no levels at the level (eg level 42), your method should produce no output after the equals sign. You must exactly reproduce the format of this output. Your method should throw an `IllegalArgumentException` if passed a value for level that is less than 1.

You are writing a public method for a binary tree class defined as follows:

```
public class IntTreeNode {
    public int data;          // data stored in this node
    public IntTreeNode left;  // reference to left subtree
    public IntTreeNode right; // reference to right subtree

    <constructors>
}

public class IntTree {
    private IntTreeNode overallRoot;

    <methods>
}
```

You are writing a method that will become part of the `IntTree` class. You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class. You may not construct any extra data structures to solve this problem.

(Write your answer on the next page.)

7. Binary Tree Programming (writing space)

One possible solution

```
public void printLevel(int target) {
   if (target < 1) {
      throw new IllegalArgumentException();
   }
   System.out.print("nodes at level " + target + " =");
   printLevel(overallRoot, target, 1);
   System.out.println();
}

private void printLevel(IntTreeNode root, int target, int level) {
   if (root != null)
      if (level == target)
         System.out.print(" " + root.data);
      else {
         printLevel(root.left, target, level + 1);
         printLevel(root.right, target, level + 1);
      }
}
```

8. Binary Tree Programming

Write a method `mirror` that converts a binary tree of integers to its mirror image. For example, if a variable called t stores a reference to the binary tree and you make a call of `t.mirror()` then the links of the tree should be rearranged so that after the call, t stores the mirror image of what it stored before the call. Below is a specific example of how a tree would change:

Before call

```
           +---+
           | 9 |
           +---+
          /     \
         /       \
     +---+       +---+
     | 6 |       | 14|
     +---+       +---+
    /     \           \
   /       \           \
+---+     +---+       +---+
| 9 |     | 2 |       | 11|
+---+     +---+       +---+
         /     \
        /       \
    +---+      +---+
    | 4 |      | 1 |
    +---+      +---+
```

After call

```
           +---+
           | 9 |
           +---+
          /     \
         /       \
     +---+       +---+
     | 14|       | 6 |
     +---+       +---+
    /           /     \
   /           /       \
+---+       +---+     +---+
| 11|       | 2 |     | 9 |
+---+       +---+     +---+
           /     \
          /       \
       +---+      +---+
       | 1 |      | 4 |
       +---+      +---+
```

After the call is made, the tree stores the structure that you would see if you were to hold the original tree's diagram up to a mirror. More precisely, the overall root (if it exists) remains in the same place in the mirror image and every other node is moved so that it's new path from the overall root is composed of the old path from the overall root with left and right links exchanged. For example, the node that stores 4 in the example above has the path (left, right, left) in the original tree. In the mirror image, it has path (right, left, right). The node that stores 1 has the path (left, right, right) in the original tree. In the mirror image it has path (right, left, left).

You are writing a public method for a binary tree class defined as follows:

```
public class IntTreeNode {
    public int data;          // data stored in this node
    public IntTreeNode left;  // reference to left subtree
    public IntTreeNode right; // reference to right subtree

    // post: constructs an IntTreeNode with the given data and links
    public IntTreeNode(int data, IntTreeNode left, IntTreeNode right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}

public class IntTree {
    private IntTreeNode overallRoot;

    <methods>
}
```

You are writing a method that will become part of the IntTree class. You may define private helper methods to solve this problem, but otherwise you may not call any other methods of the class. You may not construct any new nodes, and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). You also may not change any data fields of the nodes. You MUST solve this problem by rearranging the links of the tree.

8. Binary Tree Programming (writing space)

One possible solution

```
public void mirror() {
   mirror(overallRoot);
}

private void mirror(IntTreeNode root) {
   if (root != null) {
      IntTreeNode temp = root.left;
      root.left = root.right;
      root.right = temp;
      mirror(root.left);
      mirror(root.right);
   }
}
```

9. Linked List Programming

Write a method of the `LinkedIntList` class called `reOrder` that assumes the list starts by being sorted by absolute value and reorders the list so that it is sorted by value. The resulting list should contain all the values from the original list in sorted order from smallest to greatest. For example, if the variable list stores the following:

```
list = [0, -3, 9, 12, -13, -22, 43, 54]
```

The call of `list.reOrder();` would change the list to store the following values:

```
list = [-22, -13, -3, 0, 9, 12, 43, 54]
```

You are writing a public method for a linked list class defined as follows:

```
public class ListNode {
    public int data;       // data stored in this node
    public ListNode next;  // link to next node in the list

    <constructors>
}

public class LinkedIntList {
    private ListNode front;

    <methods>
}
```

You are writing a method that will become part of the `LinkedIntList` class. You may define 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`, but you may not construct any new nodes, and you may not use any auxiliary data structure to solve this problem (no array, `ArrayList`, stack, queue, String, etc). You also may not change any data fields of the nodes. You MUST solve this problem by rearranging the links of the lists. Your solution must run in O(n) time where n is the length of the list. As in the examples above, assume that the list starts sorted by absolute value.

9. Linked List Programming (writing space)

One possible solution

```
public void reorder() {
   if (front != null) {
      ListNode negative = null;
      ListNode current = front;
      while (current.next != null) {
         if (current.next.data < 0) {
            ListNode toMove = current.next;
            current.next = current.next.next;
            toMove.next = negative;
            negative = toMove;
         } else {
            current = current.next;
         }
      }
      if (negative != null) {
         current = negative;
         while (current != null && current.next != null) {
            current = current.next;
         }
         current.next = front;
         front = negative;
      }
   }
}
```