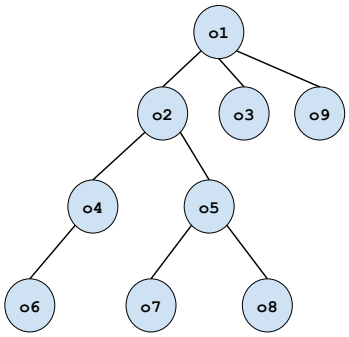
CS2103 B-term 2021 (Whitehill) — Final Examination

Rules

1. The exam is Thursday, December 16, from 4:00pm-5:30pm. You must upload your solutions to Canvas by 5:35pm. (That said, do not panic if you end up submitting a minute late due to an uploading error; we will be reasonable.) If you have an official academic accommodation to receive more time on the exam, then you may submit your exam at the adjusted deadline. For example, academic accommodations of 1.5x the exam time mean that, instead of finishing by 5:30pm (90min), you may finish by 6:15pm (135min) — and then upload your solution at most 5 minutes later.
2. The maximum score is 50 points.
3. You are allowed to pre-download all the CS 2103 (2021) lectures slides from Lectures 1-25 and use them during the exam. You may also consult freely the official Java documentation from Oracle (https://docs.oracle.com/en/java/javase/11/docs/api/index.html). You are **not** allowed to use any other web content during the exam. During the exam you are also **not** allowed to communicate with anyone about the exam content except with the course staff.
4. You are allowed to use written materials, either on paper or on your computer, *that you wrote yourself ahead of time*. For example, you may write code to practice implementing and/or using all the data structures we have covered in class, and refer to them during the exam. You may *not*, however, use any materials that are neither the lecture slides nor something you wrote yourself.
5. The Zip file for this exam contains 3 files: **`FamilyTree.java`**, **`DoublyLinkedList.java`**, and **`Polynomial.java`**.
6. To submit your exam, you should Zip all your **`.java`** files and upload it to Canvas, just like with a programming project. Exams will be graded through a combination of automatic unit tests and manual review. Hence, it is important that your code at least compile! (That said, even if your code does not compile, you can still get a decent score if the basic logic is correct.)
7. I will be available on Slack (#final) and on email to answer questions. **You are required to join this channel**, as it will be used to share late-breaking announcements.

Problem 1 — Finding Common Ancestors — 12 points

The figure to the right shows a family tree. Consider each node and its set of **ancestors**: We define the set of ancestors of a node *n* to consist of *n* itself, *n*'s parent, *n*'s parent's parent, and so on. For instance, the set of ancestors of node `o5` contains `o5`, `o2`, and `o1`. The **lowest common ancestor** *x* between two nodes *n* and *m* is the node *x* that is an ancestor of both *n* and *m*, such that no descendent (a child, a child's child, and so on) of *x* is also an ancestor of both *n* and *m*. For instance, the lowest common ancestor between `o6` and `o7` is `o2`. Notice that, although `o1` is also a common ancestor of `o6` and `o7`, it is not the lowest common ancestor. For another example, the lowest common ancestor between `o3` and `o3` is just `o3` itself (since the set of ancestors of *n* includes *n* itself). Finally, the lowest common ancestor of `o6` and `o2` is just `o2`.



Your task: Implement the `findLowestCommonAncestor (FamilyTreeNode node1, FamilyTreeNode node2)` method in the class above, which returns a `FamilyTreeNode` object representing the lowest common ancestor of `node1` and `node2`. (Remember that, even though `FamilyTreeNode` is a private static inner class of `FamilyTree`, its private instance variables are still accessible to the outer class.) Your method must work correctly on *any* family tree, not just the one shown above. You may assume that: (1) the `_parent` node of the root is set to `null`; (2) neither `node1` nor `node2` is `null`; (3) a node's `_children` is never `null`, but may be empty (if the node has no children) and (4) `node1` and `node2` come from the same family tree, and hence a lowest common ancestor always exists. **Hint**: import and use `HashSet` (or `HashMap`, if you prefer) to keep track of which nodes you've seen when iterating up toward the root.

No particular time-cost is required for this problem. This problem will be graded manually, and we will give partial credit when possible.

```
public class FamilyTree {
  private static class FamilyTreeNode {
    private String _name;
    private FamilyTreeNode _parent;
    private Collection<FamilyTreeNode> _children;
  }

  private FamilyTreeNode _root;

  FamilyTreeNode findLowestCommonAncestor
    (FamilyTreeNode node1, FamilyTreeNode node2) {
    // TODO: IMPLEMENT ME
  }
}
```

Problem 2 — Iterators — 18 points

The `DoublyLinkedList<T>` class below implements a doubly-linked list using 2 **dummy nodes** — one for the head and one for the tail. Even when the list is empty, the dummy nodes still exist. The dummy nodes are never used to store data that the user adds to the list; rather, they help to simplify the implementation of the linked list. The `DoublyLinkedList` class is `Iterable`, meaning that it offers an `Iterator` that can be used to iterate through the list, e.g.:

```
DoublyLinkedList<String> list = new DoublyLinkedList<String>();
list.add("a");
list.add("b");
list.add("c");
Iterator<String> iterator = list.iterator();
iterator.next();  // Returns "a"
iterator.next();  // Returns "b"
iterator.remove();// Removes "b" and modifies list
iterator.next();  // Returns "c"
```

Your job: Finish implementing the `DoublyLinkedListIterator` (non-static) inner class shown below so that the iterator operates correctly. Notice that the outer class does **not** implement a `remove()` method. **Requirements**:

1. All three methods of your iterator must operate in ***O*(1)** time — in particular, the `next()` method should **not** "start from scratch" from the head every time it is called.
2. You may **not** modify the **outer class** in any way; however, you are free to modify the **inner class** (`DoublyLinkedListIterator`) as you wish.
3. On this exam, the `DoublyLinkedListIterator.remove` method is **required**. In particular, this method should remove from the list the last element returned by this iterator. All of the existing code written in `DoublyLinkedList` must operate correctly in conjunction with the code you write.

Grading: This problem will be graded manually, and we will give partial credit when possible. **4 points** for construction of the `Iterator`, **4 points** for `hasNext()`, **4 points** for `next()`, and **6 points** for `remove()`.

```
class DoublyLinkedList<T> implements Iterable<T> {
      private static class Node<T> {
            Node<T> _next, _prev;
            T _data;
      }

      private Node<T> _head = new Node<T>(), _tail = new Node<T>();
      private int _numElements = 0;

      public DoublyLinkedList () {
            _head._next = _tail;
            _tail._prev = _head;
      }

      // Returns the number of elements in the list
      public int size () {
            return _numElements;
      }

      // Adds an element to the tail of the list
      public void add (T elem) {
            final Node<T> node = new Node<T>();
            node._data = elem;
            node._prev = _tail._prev;
```

```
            node._next = _tail;
            _tail._prev._next = node;
            _tail._prev = node;
            _numElements++;
      }

      // Returns the element at the specified index
      public T get (int index) {
            if (index >= _numElements || index < 0) {
                  throw new IllegalArgumentException("Invalid index!");
            }
            Node<T> cursor = _head._next;
            for (int i = 0; i < index; i++) {
                  cursor = cursor._next;
            }
            return cursor._data;
      }

      // Returns an iterator
      public Iterator<T> iterator () {
            // TODO: implement me
      }

      // *** WRITE YOUR SOLUTION IN THE INNER-CLASS BELOW ***
      private class DoublyLinkedListIterator implements Iterator<T> {
            // TODO: add any instance variables/methods you want

            // Returns true if the iteration has more elements
            public boolean hasNext () {
                  // TODO: implement me
            }

            // Returns the next element in the iteration. You do NOT
            // need to throw NoSuchElementException if next() is called
            // and there are no more elements to return.
            public T next () {
                  // TODO: implement me
            }

            // Removes from the underlying collection the last element
            // returned by this iterator. The user of the iterator
            // is required to call next() before calling remove().
            public void remove () {
                  // TODO: implement me
            }
      }
}
```

Problem 3 — Symbolic Calculator — 20 points

Besides graphing calculators and GUI-based expression editors, another use-case of manipulating mathematical expressions in code is to create a **symbolic calculator**. (Commercial examples of symbolic calculator software include Mathematica and Maple.) As an example, suppose the user types in the following expression: $2x + 3x$. Clearly, this expression can be simplified to $5x$; importantly, this simplification holds true for *any* value of x. As another example, $(x - 2x^2)(3x^3 - 4x^4)$ can be simplified to $3x^4 - 4x^5 - 6x^5 + 8x^6 = 3x^4 - 10x^5 + 8x^6$. The fact that mathematical expressions are being manipulated in ways that do *not* depend on the particular value of the variable is what makes this a "symbolic" calculator.

In this problem, you will implement methods for a symbolic calculator that operates on polynomials. For the purposes of this problem, a **polynomial** (in some variable x) is defined as a mathematical expression of the form $c_0 + c_1x^1 + c_2x^2 \ldots + c_nx^n$, where n is an integer and each **coefficient** c_i (where $i = 0, \ldots, n$) is a real number (represented as a `double` in Java). The **degree** of the polynomial is the highest power of x whose coefficient is non-zero. For example, the polynomial $3.2 + x^2 - x^4$ is a polynomial where c_0=3.2, c_1=0, c_2=1, c_3=0, and c_4=-1; the degree is 4. We can represent a polynomial using a Java class, `Polynomial`, that takes a `double[]` in its constructor whose elements contain c_0, c_1, …, c_n. In particular, in the constructor `Polynomial (double[] c)`, each coefficient c_i is given by `c[i]`, for `0 <= i < c.length`. To instantiate a `Polynomial` for the previous example, we could write:

```
new Polynomial(new double[] { 3.2, 0, 1, 0, -1 });
```

(Note: It would also be correct to instantiate it as `new Polynomial(new double[] { 3.2, 0, 1, 0, -1, 0, 0 });`; however, the degree would still be 4.)

In this problem, your task is to implement two of the key operations in a symbolic calculator — **add** and **multiply** — that operate on polynomials in a single variable x. Each of these methods should take another `Polynomial` as a parameter and return a `Polynomial` as the result. To implement these methods, recall the basic rules:

$ax^i + bx^i = (a+b)x^i$

$ax^i * bx^j = (a*b)x^{i+j}$

for any real numbers a, b, and any integers i, j. In practice, this means that, when adding two polynomials, the degree of the resulting polynomial is (at most) the *max* of the degrees of the two input polynomials; when multiplying two polynomials, the degree of the resulting polynomials is the *sum* of the degrees of the two input polynomials. You also need to provide a way of converting a polynomial into a `String` representation. Finally, you need to implement methods that returns the polynomial's degree and to return the coefficient c_i for any specified i.

Examples:

```
Polynomial p1 = new Polynomial(new double[] { 3.2, 0, 1, 0, -1 });
Polynomial p2 = new Polynomial(new double[] { 2 });
Polynomial p3 = new Polynomial(new double[] { 0, 1 });
Polynomial p1TimesP2 = p1.multiply(p2);
Polynomial p1TimesP3 = p1.multiply(p3);
Polynomial p1PlusP2 = p1.add(p2);
System.out.println(p1TimesP2);  // Should be: "6.4 + 2.0*x^2 - 2.0*x^4"
System.out.println(p1TimesP3);  // Should be: "3.2*x^1 + 1.0*x^3 - 1.0*x^5"
System.out.println(p1PlusP2);  // Should be: "5.2 + 1.0x^2 - 1.0x^4″
```

Requirements:

- `add`: Given two input `Polynomial` objects, return a `Polynomial` representing their sum. **(5 points)**
- `multiply`: Given two input `Polynomial` objects, return a `Polynomial` representing their product. **(6 points)**.
- `getDegree`: Return the degree of the `Polynomial`, i.e., the highest-valued integer n such that the associated coefficient c_n in the `Polynomial` is non-zero. **(2 points)**

- **`getCoefficient`**: Given an index *i*, return the coefficient c_i. If *i* is less than 0 or greater than the degree, then return 0. **(1 points)**
- **`toString`**: Create a **`String`** representation of the **`Polynomial`**: if the polynomial is $c_0 + c_1x^1 + \ldots + c_nx^n$, then the **`String`** should be of the form **`"`**c_0 ± c_1**`*x^1`** ± c_2**`*x^2`** ± … ± c_n**`*x^n"`**, where each ± should be either **`+`** or **`-`** (depending on whether the subsequent coefficient is positive or negative); … should be replaced by all the terms in the middle; and each c_i should be replaced by the appropriate **`double`** value. Terms whose coefficient c_i=0 should be **omitted**. Spaces between terms are **optional**. For instance, the polynomial $3.2 + x^2 - x^4$ can be represented as the String **`"3.2+1.0*x^2-1.0*x^4"`** or by **`"3.2 + 1.0*x^2 - 1.0*x^4"`**. To get full credit, you must follow these guidelines exactly; however, we will certainly give partial credit when appropriate. **(6 points)**

No particular time-cost is required for these methods. **Important note**: you may **only** import classes from **`java.util.*`** (and even they are not required).

This problem will be graded with a combination of automatic tests and manual inspection; we will give partial credit when possible.

Some starter code is shown below:

```
public class Polynomial {
  // TODO: add whatever instance variables you need

  // You can assume that c.length > 0
  public Polynomial (double[] c) {
    // TODO: implement me
  }

  // Helper method -- use if you want to
  private static int max (int x, int y) {
    return x > y ? x : y;
  }

  public Polynomial add (Polynomial other) {
    // TODO: implement me
  }

  public Polynomial multiply (Polynomial other) {
    // TODO: implement me
  }

  public int getDegree () {
    // TODO: implement me
  }

  public double getCoefficient (int i) {
    // TODO: implement me
  }

  public String toString () {
    // TODO: implement me
  }
}
```