

Name:

Midterm

CS 121
Software Engineering
Fall 2021

October 20, 2021

Instructions

This exam contains 10 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Score	Max
1		20
2		20
3		30
4		30
Total		100

Question 1. Short Answer (20 points).

a. (5 points) Java includes both `Integer` and `int`. Briefly explain the difference, and explain why `Integer` is sometimes needed.

b. (5 points) Briefly explain the difference between *checked* and *unchecked* exceptions in Java.

c. (5 points) List three potential benefits of choosing a good modular structure for a program.

d. (5 points) List three potential disadvantages of using *reflection*.

Question 2. Maps with Lists (20 points). In this problem, you will write part of `ListMap`, a class that implements the `Map` interface using a linked list of pairs, similarly to class `LL` from lecture. Here is the beginning of the class, to get you started. Throughout this problem, assume there are no duplicate keys in the map.

```
class ListMap<K, V> implements Map<K, V> {
  class Cell {
    K key; V value; Cell next;
    Cell(K key, V value, Cell next) { this.key = key; this.value = value; this.next = next; }
  }
  Cell head; // null if empty
  ...
}
```

a. (4 points) Write a method `V get(Object key)` that returns the value corresponding to `key`, or `null` if there is no such value. Use `equals` to compare keys.

b. (5 points) Write a method `V put(K key, V value)` that adds a mapping from `key` to `value` to the map and returns `null` if there was no previous mapping for `key`. Otherwise, update the previous mapping for `key` to map to `value` and return the previous mapping.

c. (5 points) Write a method `List<V> values()` that returns a list of all the values in the map. A value should appear in the result as many times as it appears in the map. The order of values is undefined.

d. (6 points) Write a method `boolean equals(ListMap<K,V> o)` that returns true if and only if `o` contains the same key–value pairs as this, *in any order*. *Note: Java's Map class doesn't include this kind of deep equality check...but perhaps it should!*

Question 3. Design Patterns (30 points). In this problem, we will consider the *Specification Pattern*, in which we use objects to represent predicates. In particular, we will write specs for student transcripts, which we will represent as a map from CS course numbers (**Integers**) to grade points (**Doubles**). For example, here is code that creates a transcript `t` in which a student got an A- in CS 105 and an A in CS 121:

```
Map<Integer, Double> t = new HashMap<Integer, Double>();
t.put(105, 3.7); // A- in CS 105
t.put(121, 4.0); // A in CS 121
```

A *spec* is an object with a `boolean` `check` method that tests whether a transcript has some property of interest:

```
interface Spec { boolean check(Map<Integer, Double> t); }
```

For example, here is a spec that returns true if and only if the given transcript includes at least a B- in CS 121.

```
class BMinus121 implements Spec {
    boolean check(Map<Integer, Double> t) {
        if (!t.containsKey(121)) { return false; }
        return t.get(121) >= 2.7; // B- or above in 121
    }
}
```

a. (4 points) Using `BMinus121`, write code that prints `true` if transcript `sample` has a grade of at least B- in CS 121, and prints `false` otherwise.

```
HashMap<Integer, Double> sample = ...; // some transcript
```

b. (8 points) Write a spec `CMinusGPA` that returns true if and only if the grade point average of a transcript is at least a C- (2.7). Raise an exception (any exception) if the transcript is empty. *Hint: You may want to use method `Collection<V> values()` of `HashMap`, which returns the values of a hash map.*

c. (8 points) Write a spec `CAnd` such that, if `s1` and `s2` are specs, then `new CAnd(s1, s2)` returns `true` if and only if both `s1` and `s2` return `true`. For example, `new CAnd(new BMinus(), new CMinusGpa())` returns `true` if and only if the transcript includes at least a B- in CS 121 and has an average GPA of C- or better.

d. (6 points) Suppose that we similarly have written classes `CNot` and `COr`, such that `new CNot(s)` returns the negation of spec `s`, and `new COr(s1, s2)` returns the disjunction of specs `s1` and `s2`. Writing all those news is annoying. Write a class `S` with methods `Spec not(Spec s)`, `Spec and(Spec s1, s2)`, and `Spec or(Spec s1, s2)` such that calling `S.not(s)` returns the negation of `s` and analogously for the other methods.

e. (4 points) Class `S` from part d is an example of what other design pattern that we saw in class? Explain your answer briefly.

Question 4. Graphs and Iterators (30 points). Recall the `ListGraph` class from Project 1:

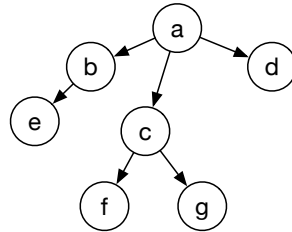
```
public class ListGraph implements Graph {
    private HashMap<String, LinkedList<String>> nodes = new HashMap<>();
    public List<String> succ(String n) { ... }
    public List<String> pred(String n) { ... }
    ...
}
```

a. (4 points) Write a method `Iterator<String> iterator()` that returns an iterator over all nodes of the `ListGraph`. Below is the `Iterator` interface, for reference. *Hint: You may want to use the method `Set<K> keySet()` of `HashMap`. A `Set` is a standard Java Collection, with the usual methods; as such, you can assume there exist any typical methods that would exist for a `Set` or a `Collection`.*

```
interface Iterator <E> {
    boolean hasNext();
    E next();
}
```

b. (6 points) Write a method `List<String> roots()` that returns a list of the *roots* of the graph, i.e., all nodes that have no predecessor.

c. (10 points) Write a method `Iterator<String> bfs()` that returns an `Iterator` whose `next` method returns all the nodes in the graph in *breadth-first search order*. (You also need to define the `hasNext` method.) This means after a node is visited, its children are visited next, then its childrens' children, etc. For example, for the following graph, the nodes are visited in alphabetical order (a, b, c, ...).



Hints:

- Your `Iterator` should maintain a `List` of nodes to visit next.
- Initially, the list contains the roots of the graph (call `roots` from part b). There is no specified ordering among the roots.
- The next node to visit is the first element of the list (which could be retrieved with `remove(0)`).
- After visiting a node, add its children to the *end* of the list (with `add(E)` or `addAll(List<E>)`). The children may be added in any order.
- You may assume there are no cycles in the graph, and your iterator may visit the same node multiple times if there are multiple paths to a node from a root.

d. (10 points) Write a method `Iterator<String> bfs(List<ListGraph> gs)` that returns an `Iterator` that visits all the nodes of all of the graphs in `gs` in breadth-first order, as follows:

- The graphs in `gs` should be visited from first to last, in order.
- Use an iterator to keep track of where you are among the list of graphs.
- Use `bfs` from part c to visit the nodes of each individual graph.
- When `next` is called, there are two cases: Either get the next node from the current graph; or iterate to the next graph and get its first node.
- The logic for `hasNext` is similar.
- You may assume all of the graphs include at least one node.