CSCI 3901 Assignment 2

Due date: 11:59pm Wednesday, September 25, 2019 in Brightspace

Problem 1

Goal

Get practice in writing test cases.

Background

One of the gains from cloud-based software deployment is that we can leverage the interactions of many people to improve future interactions. We can crowdsource ideas or learn from past behaviours.

One example of these improvements is in recommender systems. Recommender systems look at your past behaviour, compare your behaviour with that of other similar individuals, and use that group experience to make suggestions to you that seem in-line with your interests. It's a way of personalizing your experience without gathering lots of data about you.

One success of recommender systems is in online shopping. Based on your past purchases, online vendors will recommend new products that may also interest you.

Problem

Write test cases for the following code.

We want to test a class that recommends products for you to purchase. More specifically, the class will read a set of past purchase records from a file to load context. The past purchases have one purchase record per line. Each purchase record will have a set of tab-separated products that were part of that one purchase.

Given that context, a user can invoke method *recommend*() with a list of products (their shopping cart) and will be returned with a list of up to 5 suggested products to add to the shopping cart.

The *recommend*() method will operate by looking for all past purchase records and will identify those records that are supersets of the provided shopping cart, which means that all products in the current shopping cart are also in the purchase record. From that smaller list of purchase records, the method will extract the additional products that were purchased and will tally the frequency of each of these products. The method will then return the 5 most frequent products that it finds as its recommendations, sorted in decreasing order of frequency.

Inputs

An input data file for your class will have one line for each purchase record. Each line will contain a set of tab-separated product names (case invariant) that are part of the purchase. Example of past purchase records:

```
milk	bread	orange_juice	eggs
flour	baking_soda	olive_oil		eggs	butter	apples
orange_juice	bread	eggs	raspberries	strawberries	blueberries
candy_bar
```

The recommend() method will have an ArrayList as a parameter, which is a list a list of strings that are product names currently in the shopping cart.

Sample request

Suppose that the current shopping cart contains orange_juice and eggs. The recommend() method will find that 2 rows also contain those (rows 1 and 3). From those rows, it will find that bread appears twice and each of milk, raspberries, blueberries, and strawberries occur once. The list returned is then bread, blueberries, milk, raspberries, strawberries (with bread first because it has a higher frequency of occurrence).

Notes

- You are <u>not</u> asked to write code to solve this problem.
- Only write test cases for this problem.
- <u>Do not provide</u> test data or expected outputs for your test cases.
- I am looking for distinct test cases. Do not duplicate conditions across cases.
- Ensure that your test case description is short but also clear on what is being tested. Cases where we can't tell what is being tested will be discarded.

Marking scheme

- List of test cases, assessed based on completeness of coverage for the problem and distinctness of the cases – 5 marks

Problem 2

Goal
Implement a data structure from basic objects.

Background
Balanced binary search trees can be tricky to code and expensive to maintain. However, we don’t always need a perfectly balanced tree. We are often content with an approximation to the tree or a heuristic structure that mimics the balanced binary tree.

In this assignment, you will implement a data structure that looks like a set of linked lists but that mimics some behaviour of a balanced binary tree.

Problem
Write a class called “ListHierarchy” that accepts data values and then lets you search the list to see if a value is in the list. The key part of the class is in the data structure that it uses to store the data.

The data structure will have a random behavior to it. For testing purposes, the constructor for the class will accept an object of type “coin” to use for all the random parts. You will be supplied with different implementations for the “coin” object: one implementation will use randomness and the other will let you specify the coin flips yourself so that you can have deterministic debugging.

Data structure
The data structure is a hierarchy of sorted linked lists. Each higher level of the hierarchy contains a subset of the values stored in the level directly below it. The lowest level of the hierarchy contains all of the elements as a sorted linked list.

Figure 1 shows a structure with 8 values in it. These values are in the lowest linked list. Four of the values (date, grape, orange, and tea) were randomly selected to appear in the next level up. Of those four, two of them (date and orange) were randomly selected to appear in another higher level, creating 3 levels.

Add: To add a value, we insert the value into the lowest list, keeping the sorted order. Once inserted, we randomly choose whether or not the new item should go into the list above; call the coin class to get back a random 0 or 1 value. 0 means we do not put the new value into the upper list and 1 means that we do put it into the upper list. When we add the value to a higher list, we keep a connection between the nodes with the value in the two lists.

If we add the value to the upper list then we get another random value to see if we add it to the next higher list. This random selection process continues until you are in the topmost list or until you have a choice to no longer promote the value to a higher list. In the topmost list, you still get a random value about moving the value to a higher list; if you do move it up then you add a new level to the hierarchy, add the value, and then stop.

Search: Given a value v to search for, we begin the search in the highest level (the one with 2 entries in Figure 1). Suppose that we are searching for “mango” in Figure 1. We would search the top list for “mango”, find that it is not there, and find that it would go between the values of “date” and “orange”. We would then follow the link from “date” to the next lower level and continue the search in that next lower level, starting at the “date” node to find that “mango” isn’t there, but it would appear between “grape” and “orange”. Again, follow the link from “grape” in one level to the next lower level and continue the search starting at the “grape” node. That continuation shows that “mango” is not there, but it would fit between “lettuce” and “orange”. Since we are at the lowest level, we would conclude that “mango” is not in the list.

Given the search description, it is useful for each level to have a sentinel node at the start of each list so you always have some earlier node to reference.

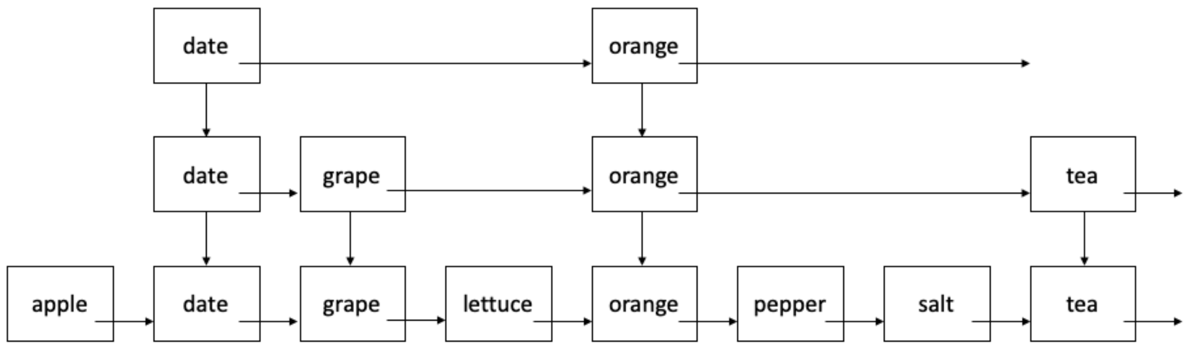

Figure 1 Sample data structure with 8 values in it.

Methods for the ListHierarchy class:

- boolean ListHierarchy(Coin flip) – constructor for the method that accepts an object of type Coin to give random values (store “flip” as an attribute of the class for future use). Return true if the object is ready to use at the end of the method call and false if there is some error.
- boolean add(String key) – add a key to the data structure. Return true if the key can be found in the list at the end of the method (so return true if the key is already in the list too). Return false if there was an error.
- boolean find(String key) – search the data structure for the key value. Return true if the key value is in the data structure. Return false if there is an error or if the key is not in the data structure.

Inputs

All the inputs will be determined by the parameters used in calling your methods.

Assumptions

You may assume that

- no string to store will be more than 15 characters.

Constraints

- You may <u>not</u> use any data structures from the Java Collection Framework, including ArrayLists.
- If in doubt for testing, I will be running your program on bluenose.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- I suggest that you begin by implementing a single sorted linked list; you won't need any random element for that. Get that working. Next, implement a single additional level to the hierarchy and get that working. Only after you have done this should you think of making the hierarchy grow dynamically.
- When it comes time to use levels of the hierarchy, start with a Coin object where you control whether or not it returns a 0 or a 1. That control will make it easier for you to test.
- Although Figure 1 shows arrows in one direction, you may find it more convenient to have references go in both directions, so a doubly-linked list at each level.
- It may be convenient, as a mental model, to think of the links between levels also as a doubly-connected linked list.
- You might want to implement a print() method that prints your data structure, to help with debugging.

Marking scheme

- Documentation (internal and external) – 3 marks
- Program organization, clarity, modularity, style – 3 marks
- Ability to act as a single level linked list (the bottom level) – 6 marks
- Ability to have two levels of hierarchy that operate correctly with add and find – 3 marks
- Ability to have as many levels as we want – 3 marks

Test cases

- Provide a null coin to the constructor
- Provide a valid coin to the constructor

Add

- Add an item to an empty list
- Add an item to the start of a list with 1 entry
- Add an item to the end of a list with 1 entry
- Add an item to the start of a list with many entries

- Add an item to the middle of a list with many entries
- Add an item to the end of a list with many entries
- Add an item and use a coin that keeps the entry in the bottom level
- Add an item that is promoted to the next level up in the hierarchy
- Add an item that is promoted to the topmost level of the hierarchy and stops
- Add an item that ultimately results in a new hierarchy level in being created
- Add an item that is already in the data structure

Find

- Search for a key in an empty data structure
- Search for a key (found) in a structure with 1 element
- Search for a key in a structure with many elements and the key is the first element of the whole list
- Search for a key in a structure with many elements and the key is a middle element of the whole list
- Search for a key in a structure with many elements and the key is the last element of the whole list
- Search for a key that is in the topmost hierarchy
- Search for a key that is first appears in a middle-level of the hierarchy
- Search for a key that is only in the lowest level of the hierarchy
- Search for a key that is not in the structure when the structure has 1 element
- Search for a key that is not in the structure when the structure has many elements and the search key comes before all entries in the structure
- Search for a key that is not in the structure when the structure has many elements and the search key would normally be in the middle of the structure
- Search for a key that is not in the structure when the structure has many elements and the search key comes after all entries in the structure