Koç University
COMP202
Data Structures and Algorithms
Assignment 4

Instructor: Barış Akgün
Due Date: May 3 2019, 23:59
Submission Trough: Blackboard

This programming assignment will test your knowledge and your implementation abilities of what you have learned in the Maps and Hashmaps parts of the class.

This homework must be completed individually. Discussion about algorithms and data structures are allowed but group work is not. Any academic dishonesty, which includes taking someone else's code or having another person doing the assignment for you will not be tolerated. **By submitting your assignment, you agree to abide by the Koç University codes of conduct.**

Description

This assignment will have two parts. The first part of the assignment requires you to implement hashmaps and related data structures. Specifically, you are asked to implement hashmaps with two different collision checking approaches, a hashmap based counter and a hashset. The second part will require you to to solve an applied problem, calculating the co-occurence of words, using the data structures you have implemented. The files you are given include comments to help you out. However, do not hesitate the contact us if anything is unclear.

Part I: Implementing the Data Structures

Hashmaps

This assignment includes two hashmap versions with different collision handling approaches. One of them uses open addressing, specifically double hashing, and the other uses separate chaining, specifically linked-lists. Both of the approaches use multiply-add-divide (MAD) compression. These hashmaps implement the map interface and extend an abstract hashmap class. This abstract class has some methods and fields, such as the parameters for the compression function, that are common to both implementations. The design of these hashmaps rely on a hashentry class which represent the key-value pairs.

The files provided to you have comments that will help you with your implementation. You can find the files related to hashmap part of the assignment below. Bold ones are the ones you need to modify and they are placed under the *code* folder, with the rest under *given*.

- *iMap.java*: This file defines a simple map interface. Even though you do not need to modify this, make sure to read the comments in detail. It is pretty straight forward. This is same interface from the binary search tree assignment.
- *AbstractHashMap.java*: This file includes common fields and methods for the hashmaps. The trivial methods are handled for you. The compression function parameters and the method that updates them based on the underlying array capacity are also given to you in this file. You might need to look at this class from time to time while implementing the hashmaps.
- *HashEntry.java*: This file includes the `HashEntry` class that describes a *Key-Value* pair for a hashmap. Certain useful methods are implemented for you. This is practically the same as the entry class from the binary search tree assignment. The skeleton of the hashmaps are designed such that the containers hold this type.

- ***HashMapDH.java***: This file has the base code for the hashmap that uses open addressing for collision handling with double hashing, namely the `HashMapDH` class. It extends the `AbstractHashMap` class and implements the `iMap` interface. Feel free to add more fields and methods. You are not allowed to use existing Java data structures apart from implementing the `keySet` method. The specific requirements are given as comments just before the class definition.

- ***HashMapSC.java***: This file has the base code for the hashmap that uses separate chaining for collision handling with linked lists, namely the `HashMapSC` class. It extends the `AbstractHashMap` class and implements the `iMap` interface. Feel free to add more fields and methods. You are not allowed to use existing Java data structures apart from the secondary containers and implementing the `keySet` method. The specific requirements are given as comments just before the class definition.

Counter

A counter is an abstract data type that keeps track of how many times equivalent keys are added. It can be considered as a string-integer map with a several modifications. Its main operations are increment (analogous to put) and getCount (analogous to get). Some implementations allow for decrementing the count and/or removing the key altogether which we are not going to need.

In this part of the assignment, you are to implement a counter by wrapping a hashmap. The counter will hold an internal hashmap (instead of extending it) which can be supplied from the outside. The class is setup such that you are only going to fill two methods, rest are handled for you. Make sure you understand the requirements of the increment and the getCount methods from the comments!

The only file that you need to work on for this part is the ***HashCounter.java*** file under the *code* folder.

Set

A set is an abstract data type that holds unique keys without any particular order. It can be considered as an implementation of the mathematical concept of a finite set. Its main operations are put, remove and contains which should be self-explanatory. Sets can be implemented by binary search trees or as hashsets. Set data structure does not involve key-value pairs but only involve keys. During the class, most of our examples for maps only had a key instead of a key-value pair so this part should not pose any difficulty.

In this part of the assignment, you are to implement a set using hashing ideas. You can directly use one of your hashmap implementations and hide the value or you can implement a hashset from scratch. The former is easier, the latter will end up cleaner and faster. The files in this part of the assignment are below. Bold ones are the ones you need to modify and they are placed under the *code* folder, with the rest under *given*.

- *iSet.java*: This file defines a simple set interface. Even though you do not need to modify this, make sure you read the comments in detail. It has some differences with the map interface but it is still pretty straight forward.

- ***HashSet.java***: This file contains the fairly empty definition of the `HashSet` class which extends the set interface. You are free to design and implement it however you want. However, you are not allowed to use existing Java data structures apart from the secondary containers and implementing the `keySet` method.

Part II: Word Co-Occurrence Calculation

In fields related to computational linguistics, it is common to assume that semantically related terms appear together wihin a context. These terms are usually taken as words, which define the concept of *word co-occurrence.* The context can differ, from single sentences to entire documents.

In this part of the assignment you are going to count the number of times two words appear together within a certain distance of each other in a single sentence given some text. You are going to form a *word co-occurrence matrix.* This is a square matrix where row and column indices correspond to words and the specific entry in a given coordinate correspond to number of times that these two words occur together.

Suppose we are given the text “To be or not to be. That is the quetion.” and are asked to compute the co-occurence matrix. The context is given as 2 word neighbor distance in a sentence:

1. The first step we are going to take is about finding the unique words (this does not need to be a separate step but helps the example). The set of unique words is {to,be,or,not,that,is,the,question}. It is common practice to convert all to lower case.

2. The second step is to separate the sentences which is trivial. The list of sentences is [to be or not to be, that is the question].

3. The third step is to iterate over these sentences. For each word, we find its neighbors within the given distance and jointly count them. For example:

   - For the first “to” in the first sentence, its neighbors are “be” and “or”. We set the entries that correspond to to-be and to-or to 1
   - For the first “be”, the neighbors are “to”, “or” and “not”. We set the entries that correspond to be-to, be-or and be-not to 1
   - For “or”, the neighbors are “to”, “be”, “not” and “be”. The entries should be set accordingly.
   - Skipping to the second “to”, the neighbors are “or”, “not” and “be”. We have seen to-be and to-or before so we incement them to 2 and set to-not to 1.

Note that we do not count any neighbors outside the sentence. If we do it for all the words and sentences we get the following word co-occurence matrix:

	to	be	or	not	that	is	the	question
to	0	2	2	1	0	0	0	0
be	2	0	1	2	0	0	0	0
or	2	1	0	1	0	0	0	0
not	1	2	1	0	0	0	0	0
that	0	0	0	0	0	1	1	0
is	0	0	0	0	1	0	1	1
the	0	0	0	0	1	1	0	1
question	0	0	0	0	0	1	1	0

Note that the matrix is symmetric. Co-occurrence matrices in general do not need to be symmetric but our definition of the context made it so.

For certain problems, it is desirable to ignore certain extremely common words such as the English articles “a”, “an”and “the”. Another, potentially harmful, appraoch is to ignore words that are shorther than two characters since they maybe one of the common words.

For other problems, there is only a set of keywords that we care about and want to calculate the co-occurence matrix just based on those, ignoring all the words.

The file ***HashBasedWordCo.java***, has the skeleton of the code to implement this part. This class takes an entire text as a string and calculates the co-occurence matrix. The matrix is represented by a hashmap of string - hashcounter of string. A lot of support code such as loading files, pre-processing text and splitting the sentences are done for you. You should go over them. You need to complete two methods; (1) finding the unique words and (2) calculating the matrix. You may use anything you have implemented in this assignment but not allowed to use any other Java data structures apart from the given sentence lists. In fact, the class uses your implementation of hashmaps, hashcounters and hashsets.

Testing

The *Test.java* has been updated and the file *TestHashmap.java* has been added to give your code a more thorough testing.

Warnings

All the methods that take a key as input should return null if the key is null. If the method return type is a primitive, it should return -1 if numeric and false if boolean. There are not any other options for this assignment.

Read all the comments, they are helpful. Let us know if they are confusing, contradictory or unclear.

You are not allowed to use any default Java data structures other than to implement the `keySet` methods and for the secondary containers of the `HashMapSC` class. The *HashBasedWordCO.java* file has lists and arraylists for limited parts which you can use but do not add your own.

Grading

Your assignment will be graded through an autograder. Make sure to implement the code as instructed, use the same variable and method names. A version of the autograder is released to you. Our version will more or less be similar, potentially including more tests. Run the main program in the *Test.java* to get the autograder output and your grade.

In case the autograder fails or gives you 0 when you think you should get more credit, do not panic. Let us know. We can go over everything even after your submission. Make sure that your code compiles!

Submission

You are going to submit a compressed archive through the blackboard site. The file should extract to a folder with your student ID without the leading zeros. This folder should only contain files that were in **boldface** in the previous section. Other files, which you should not have modified anyways, will be deleted. Other files will be deleted and/or overwritten.

Important: Make sure to download your submission to make sure it is not corrupted and it has your latest code. You are only going to be graded by your blackboard submission.

Submission Instructions

- You are going to submit a compressed archive through the blackboard site. The file can have *zip*, *tar*, *rar*, *tar.gz* or *7z* format.
- This compressed file should extract to a folder with your student identification number with the two leading zeros removed which should have 5 digits. Multiple folders (apart from operating system ones such as MACOSX or DS Store) greatly slows us down and as such will result in penalties
- Code that does not compile will be penalized and may receive no credits.
- Do not trust the way that your operating system extracts your file. They will mostly put the contents inside a folder with the same name as the compressed file. We are going to call a program (based on your file extension) from the command line. The safest way is to put everyting inside a folder with your ID, then compress the folder and give it your ID as its name.
- One advice is after creating the compressed file, move it to your desktop and extract it. Then check if all the above criteria is met.
- Once you are sure about your assignment and the compressed file, submit it through Blackboard.
- After you submit your code, download it and check if it the one you intended to submit.
- **DO NOT SUBMIT CODE THAT DOES NOT TERMINATE OR THAT BLOWS UP THE MEMORY.**

Let us know if you need any help with setting up your compressed file. This is very important. We will put all of your compressed files into a folder and run multiple scripts to extract, cleanup and grade your codes. If you do not follow the above instructions, then scripts might fail. This will lead you to get a lower grade than what the autograder suggests.