UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS212 - Data structures and algorithms

Practical 5 Specifications: Hashmap

Release date: 15-04-2024 at 06:00
Due date: 19-04-2024 at 23:59
Total marks: 190

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually; no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of the built-in Java data structures. Doing so will result in a zero mark. You may only use native 1-dimensional arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile, you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experiences a runtime error, you will be awarded a zero mark. Runtime errors are considered unsafe programming.
- Read the entire specification before you start coding.
- **Ensure your code compiles with Java 8**
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent), and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers to avoid any misunderstanding.** Also note that the OOP principle of code reuse does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

On completion of this practical, you will have gained experience with the following:

- Implementing an algorithm to generate prime numbers.
- Implementing a hashmap.

4 Introduction

Prime numbers are natural numbers whose only factors are 1 and the number itself. These are often used in mathematical and computer science applications, and as such an algorithm is needed to generate these numbers. An easy way to accomplish this, is to use the Sieve of Eratosthenes, which is an algorithm created by Greek mathematician Eratosthenes of Cyrene, around 200BC.

Hashmaps are data structures with a constant lookup time. Meaning that even when a lot of elements are added to the data structure, we can still find an element in constant lookup time. This is achieved by using a hash function. This is a function that takes in an input and returns a number which is then used to insert the element into the hashmap.

5 Tasks

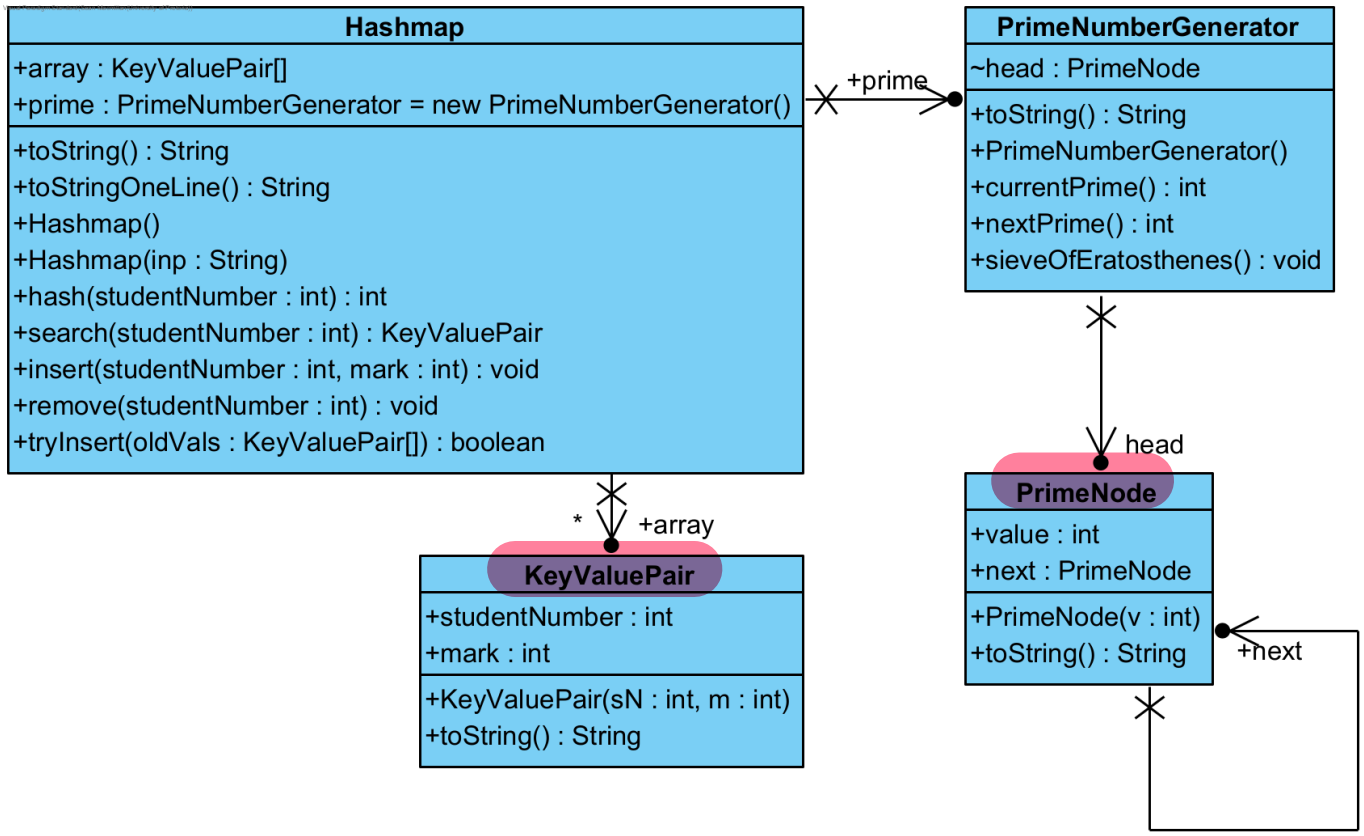

Figure 1: UML

5.1 PrimeNode

- This class is given to you.
- This class is used as a linked list node for the PrimeNumberGenerator.
- Don't change the toString function, as this is used on Fitchfork.
- Feel free to add extra functions or members.

5.2 KeyValuePair

- This class is given to you.
- This class is used as the array type for the hashmap. It contains a student number and a mark.
- Don't change the toString as this is used on Fitchfork.
- Feel free to add extra functions or members.

5.3 PrimeNumberGenerator

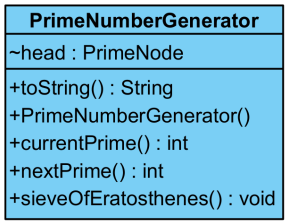

Figure 2: PrimeNumberGenerator UML

- Members
  - head: PrimeNode
    - This is the head of the PrimeNumberGenerator linked list.
    - **This can never be null, thus, the list can never be empty**.
- Functions
  - toString(): String
    - This function is given to you.
    - Don't change it, as it is used on Fitchfork.
  - PrimeNumberGenerator()
    - This is the constructor.
    - It initialises the list, so that the head node contains the value 2.
  - currentPrime(): int
    - This returns the value of the head, without changing anything.
  - nextPrime(): int
    - This should move the list along, such that the next prime number is now at the head. After the head is updated, return the value of the new head.
    - Note that we are actually moving the head. Therefore, all of the old prime numbers are lost.
    - If the head is the only element in the list, then call `sieveOfEratosthenes()` **before** moving the head along.

– sieveOfEratosthenes(): void

* This function adds prime numbers to the linked list.
* Implement the following psuedocode

```
1. Create an array of booleans with size (head.value * 2 + 1), and
   call it notPrime. This will be initialised to an array filled
   with false.
2. Create an int variable called jump and initialize it to 2.
3. While jump is smaller than the number of elements in the notPrime
   array:
    3.a. Create an int variable called counter and initialize it to
       jump + jump.
    3.b. While counter is smaller than the number of elements in the
       notPrime array:
        3.b.i Set notPrime[counter] to true.
        3.b.ii Increment counter by jump.
    3.c Increment jump by 1.
4. notPrime is now an array of booleans where all indexes that store
   false, are prime numbers. Note that even though 0 and 1 are
   false, they are not prime.
5. Update the linked list, by adding all of the found primes in the
   notPrime array, to the linked list. Be careful not to add prime
   numbers which were already in the list (and have been deleted by
   moving the head).
```

5.4 Hashmap

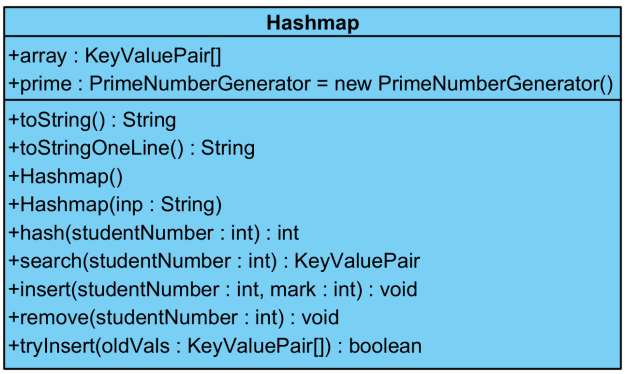

Figure 3: Hashmap UML

- Members
  - array: KeyValuePair[]
    * This is an array of KeyValuePairs, which will store the hashmap data.
    * Empty spots will be indicated by null.
  - prime: PrimeNumberGenerator
    * This is the object used for prime numbers that are needed for the implementation of the hashmap.

- Functions
  - toString(): String
    - This function is given to you.
    - Do not change this as it is used on Fitchfork.
  - toStringOneLine(): String
    - This function is given to you.
    - Do not change this as it is used on Fitchfork.
  - Hashmap()
    - This is the hashmap constructor.
    - Set the array to an empty array of size 1.
  - Hashmap(inp: String)
    - This is another constructor. This one is used to reconstrct hashmaps.
    - The input parameter has the same format as `toStringOneLine()`. Read through this function to see the format and then reverse engineer that function for this constructor.
    - **You may assume that the input is valid and don't need to check the format.** It is possible that hashmaps are created which have students in the wrong place. This is intended, which might lead to weird results when calling `search()` and `remove()`.
    - You may assume the following:
      - The input is a String which can be returned by the `toStringOneLine()`.
      - The array will have a size of at least 1.
      - Each student will have a student number and a mark, and the format can be seen in the KeyValuePair `toString()` function.
    - **Note:** Using this constructor can create arrays with different sizes than normal, and also start the prime numbers at a different place.
      - You should look at how many items are in the input, and then set the array variable to hold the same number of elements.
      - If the number at the start is not a prime number, then your prime number object should be set to the first prime number larger than the value in the String. Also, remember that the smallest prime number is 2.
  - hash(studentNumber: int): int
    - This returns the hash value of the passed-in student number.
    - This should implement a version of Horner's rule.
    - In the normal Horner's rule, we used the ASCII value of the character. For this version, since we work with student numbers, each character should be read as an int. Example: If the student number is 22222222, then each character is read as 2, and not 50.
    - In the Lecture slides, 37 was used as the prime number. This should be replaced by the current prime from the prime member variable.
    - After looping through the input, remember to make sure that the answer is not negative. In the lectures this was done by adding the table size, but this won't work for our application. Instead, call absolute value on the value.
    - Before returning the answer, perform a modulus operation using the array size, to ensure it is a valid index.

- search(studentNumber: int): KeyValuePair
  - This searches for the passed-in student number in the hashmap.
  - Use the insertion rules, to see where the student number can be stored.
  - If the student number was not found, return null, otherwise return the KeyValuePair.
  - *Hint: Because we have deletion, there is more than one possible spot where the student number can be stored. Therefore, do not stop searching at the first null.*
- insert(studentNumber: int, mark: int): void
  - If the passed-in student number is already in the hashmap, then just change the mark of that student to the passed-in mark.
  - Calculate the hash of the passed-in student number. If that index is empty in the hashmap, then add the new student there.
  - If the index is not empty, the collision resolution works as follows:
    - A version of quadratic probing is used, where the following offsets are checked:

```
abs(hash + 1 * currentPrime) % array length,
abs(hash - 1 * currentPrime) % array length,
abs(hash + 4 * currentPimre) % array length,
abs(hash - 4 * currentPrime) % array length,
abs(hash + 9 * currentPrime) % array length,
abs(hash - 9 * currentPrime) % array length
```

    - If all of theses indices are full, then the table needs to be resized. The old elements should first be moved to a temporary array. Then double the size of the hashmap's array and call `nextPrime()` on the prime object. Loop linearly through the old values, inserting them into the new array. After all of the old values have been inserted, then insert the passed-in student's mark and student number.
    - *Note that the above mentioned algorithm might lead to scenarios where the old values can't fit in. In this case, the array's size will be doubled more than once, and the prime will move on more than once.*
- remove(studentNumber: int): void
  - Remove the passed-in student number from the hashmap, by setting the correct index to null.
  - You should not change any of the other KeyValuePairs.

6 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, approximately 10% of the assignment marks will be assigned to your testing skills. To do this, you will need to submit a testing main (inside the Main.java file) that will be used to test an Instructor-provided solution. You may add any helper functions to the Main.java file to aid your testing. In order to determine the coverage of your testing the jacoco tool. The following set of commands will be used to run jacoco:

```
javac *.java
rm -Rf cov
mkdir ./cov
java -javaagent:jacocoagent.jar=excludes=org.jacoco.*,destfile=./cov/output.exec
    -cp ./ Main
mv *.class ./cov
java -jar ./jacococli.jar report ./cov/output.exec --classfiles ./cov --html
    ./cov/report
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

and we will scale this ratio according to the size of the class.
The mark you will receive for the testing coverage task is determined using table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the function stipulated in this specification will be considered to determine your mark. Remember that your main will be testing the instructor-provided code and as such it can only be assumed that the functions outlined in this specification are defined and implemented.

7 Upload checklist

The following files should be in the root of your archive

- Main.java
- Any textfiles needed by your Main
- KeyValuePair.java
- PrimeNode.java
- PrimeNumberGenerator.java
- Hashmap.java

8 Allowed libraries

- java.lang.Math

9 Submission

You need to submit your source files on the FitchFork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above-mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. Your code must be able to be compiled with the Java 8 standard.

For this practical, you will have 5 upload opportunities and your best mark will be your final mark. Upload your archive to the appropriate slot on the FitchFork website well before the deadline. **No late submissions will be accepted!**