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: Hashing

Release date: 17-03-2025 at 06:00
Due date: 21-03-2025 at 23:59
Total marks: 169

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 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.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Please note that there is a late deadline which is 1 hour after the initial deadline, but you will lose 20% of your mark.

2 Overview

In this Practical you will be dealing with storage of patient records for a regional hospital that receives large volumes of patients and must store patient records for them all, each patient record is identified by a unique 12 digit code

This unique identifier is of the format XXXX-XXXX-XXXX where:

- First segment (XXXX): Patient's birth year and month (YYMM)
- Second segment (XXXX): Department code and admission sequence
- Third segment (XXXX): Treatment code and physician ID

For example, a key might be "9307-4521-8732" representing:

- Patient born in July 1993
- Admitted to department 45, admission 21
- Receiving treatment 87 from physician 32

Records need to be retrievable instantly in emergency situations. Storage of records should result in minimal collisions.

3 Tasks

You will be required to implement a Hashmap to store patient records. It implements chaining as a collision strategy as well as 3 hash functions where-in a composite hashing approach will be implemented.

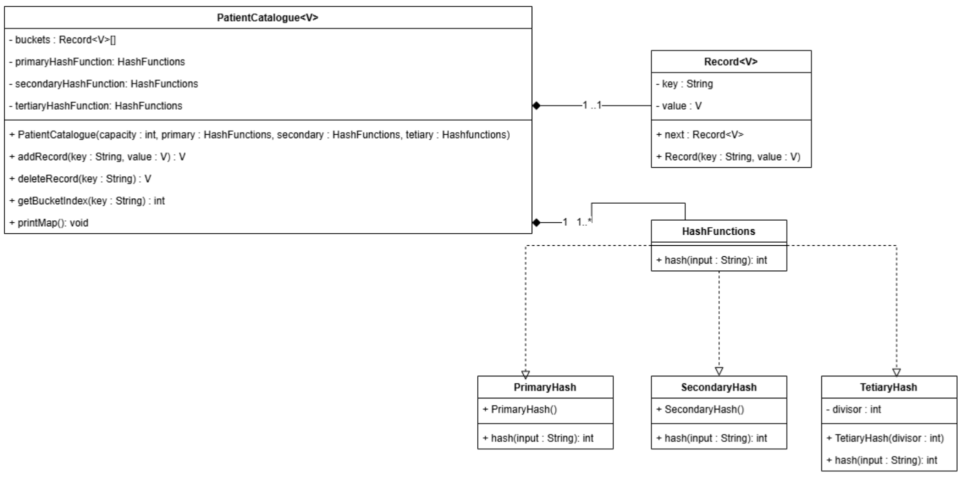
PatientCatalogue<V>
- buckets : Record<V>[]
- primaryHashFunction: HashFunctions
- secondaryHashFunction: HashFunctions
- tertiaryHashFunction: HashFunctions
+ PatientCatalogue(capacity : int, primary : HashFunctions, secondary : HashFunctions, tetiary : Hashfunctions)
+ addRecord(key : String, value : V) : V
+ deleteRecord(key : String) : V
+ getBucketIndex(key : String) : int
+ printMap(): void
1 ..1
Record<V>
- key : String
- value : V
+ next : Record<V>
+ Record(key : String, value : V)
1
1..*
HashFunctions
+ hash(input : String): int
PrimaryHash
+ PrimaryHash()
+ hash(input : String): int
SecondaryHash
+ SecondaryHash()
+ hash(input : String): int
TetiaryHash
- divisor : int
+ TetiaryHash(divisor : int)
+ hash(input : String): int

Figure 1: Binary-Tree Structure for Container Database

3.1 PatientCatalogue Class

The PatientCatalogue class implements a hash map data structure using multiple hash functions for efficient patient record storage and retrieval. It uses a chaining approach to handle collisions, where each bucket contains a linked list of records.

3.1.1 Members

Record<V>

- An inner private static class that represents nodes in the linked lists, storing key-value pairs and references to the next record in the chain.

buckets

- Array of linked lists that serve as storage containers for the records, with each bucket potentially containing multiple records.

primaryHashFunction

- The first hash function applied to keys, which implements a folding algorithm to process the input.

secondaryHashFunction

- The second hash function applied to the result of the primary hash, which uses a mid-square technique.

tertiaryHashFunction

- The third hash function applied to the result of the secondary hash, which uses division method to determine final bucket index.

3.1.2 Functions

PatientCatalogue

- Constructor that initializes the catalogue with a specified capacity and three hash functions.

getBucketIndex

- Private method that applies the three hash functions in sequence to determine the appropriate bucket index for a given key, ensuring the result stays within the bounds of the buckets array.
- for empty strings return index -1
- The Sequence of the Hash Functions is PrimaryHash -> SecondaryHash -> TertiaryHash

addRecord

- Adds a new record to the catalogue or updates an existing one, returning the previous value if the key already existed, or null otherwise. The method handles key comparison for both null and non-null keys.
- append new record entries

deleteRecord

- Removes a record with the specified key from the catalogue, handling special cases for removing the head of a list and ensuring proper linking of the remaining records. Returns the value that was associated with the key, or null if the key was not found.

printMap

- Displays the structure of the hash map, including capacity, hash functions used, and the contents of each bucket with visual representation of linked records using arrow notation.

3.2 HashFunctions Class

The HashFunctions abstract class serves as the base class for all hash function implementations used by the PatientCatalogue.

3.2.1 Functions

hash

- Abstract method that each concrete hash function must implement to convert a string input into an integer hash value, forming the basis of the hashing strategy.

3.3 PrimaryHash Class

The PrimaryHash class implements a folding hash function that processes input strings in the format "XXXX-XXXX-XXXX".

3.3.1 Functions

PrimaryHash

- Default constructor that initializes the hash function without any specific parameters.

hash

- Implements the hash method by splitting the input by hyphens, reversing the middle part, and summing all parts together. Validates that the input follows the expected format with three parts return -1 if not.

3.4 SecondaryHash Class

The SecondaryHash class implements a mid-square hash function that extracts the middle 4 digits after squaring the input.

3.4.1 Functions

SecondaryHash

- Default constructor that initializes the hash function without any specific parameters.

hash

- Implements the hash method by converting the input to a numeric value squaring it, and extracting the middle 4 digits of the squared value. Tip* starting Positon = (length - 4) / 2;
  - The third hash function applied to the result of the secondary hash, which uses division method to determine final bucket index.

3.5 TertiaryHash Class

The TertiaryHash class implements a division hash function.

3.5.1 Members

divisor

- The positive integer value used as the divisor.

3.5.2 Functions

TertiaryHash

– Constructor that initializes the divisor value.

hash

– Implements the hash method by returning the absolute value of the modulo operation with the divisor.

4 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.

5 Upload checklist

The following files should be in the root of your archive

- Main.java
- PatientCatalogue.java
- PrimaryHash.java
- SecondaryHash.java
- TertiaryHash.java
- HashFunctions.java
- Any textfiles needed by your Main

6 Allowed libraries

- No imports allowed.

7 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!**