1. Beschreibung der implementierten Datenstruktur und Algorithmen

Die **HashTableImpl-Klasse** implementiert eine Hashtabelle-Datenstruktur (HT). In meinem Anwendungsfall werden nur Strings in der HT gespeichert und verwaltet. Dabei wird Double Hashing-Algorithmus verwendet, um Kollisionen zu behandeln und eine bestimmte Position in der Tabelle zu finden. Ich habe mich auf diese Formel orientiert: $hi(k) = (h(k) - s(k) * i) \mod m$, wobei eine Sondierungsfunktion, die eine Permutation der Indizes in der HB bildet [1], so war: $s(k,m') = 1 + k \mod m'$ [2].
Als ich Schwierigkeiten und mein Programm hatte, habe ich eine Beispiel-Implementation angeschaut und mit meinem Code verglichen [3]. Als Ergebnis hab ich die zweite Hashfunktion daraus übernommen und es hat mein Problem gelöst. Ansonsten habe ich mich an der Formel orientiert.

a) **Konstruktor -** initialisiert die HT mit einer angegebenen Anzahl von Slots (m).

b) Die Klasse hat folgende **Attributen:**
private final String[] hashTableSlots speichert Elemente in der HT. Jeder Eintrag im Array repräsentiert einen Slot in der HT. Der Wert des Eintrags kann entweder ein String sein oder null, wenn der Slot leer ist.
private final int hashTableSize (**m** in der Formel) speichert die ursprüngliche Größe (die Anzahl der Slots) der HT, die vom Benutzer angegeben wurde. Eine meist effiziente ***hashTableSize*** wird berechnet (Anzahl der eindeutigen Einträge in der CSV-Datei*1.2 [vgl. 4]) und dem Benutzer vorgeschlagen. Obwohl es empfohlen wird, **m** in Primzahl umzuwandeln, habe ich bemerkt, dass wenn ich das nicht tue, habe ich weniger Kollisionen beim Insert (277636 im Vergleich zu 282249), daher habe ich m als normale Zahl gelassen.
int primeTableSize (**m'** in der Formel) speichert nächstkleinere zur Tabellengröße (m) Primzahl mithilfe isPrime()-Methode [5]. Wird in der Double Hashing Formel verwendet, um die Wahrscheinlichkeit von Kollisionen zu verringern. Der Wert wird während der Initialisierung der HT berechnet.
private int elementsCounterInHashTable (**j** in der Formel) speichert die Anzahl der aktuellen Einträge in der HT. Es wird aktualisiert, wenn neue Elemente hinzugefügt oder vorhandene Elemente gelöscht werden.
int collisionCounter speichert die Anzahl der Kollisionen, die während der insert-Operation in der HT aufgetreten sind und dient als Messgröße für die Effizienz

c) und folgenden **Methoden:**
getCurrentAmountOfEntries() gibt die aktuelle Anzahl der Einträge in der HT zurück.
getRateOfFullfilment()-Methode berechnet den Belegungsgrad der HT und gibt ihn aus.
insert(String entryString) sucht mithilfe von Double-Hashing einen freien Platz in der Tabelle und fügt den String hinzu.
search(String searchedString) findet mithilfe von Double-Hashing die Position des gesuchten Strings und gibt ihn zurück.
delete(String deletedString) sucht nach dem String und entfernt ihn aus der Tabelle.

getCollisionCount() gibt die Anzahl der Kollisionen zurück, die während dem Insert in der HT aufgetreten sind und damit dient als Messgröße für die Effizienz.
printHashTable() gibt den Inhalt der HT aus.

2. Beschreibung der Laufzeitkomplexität für die einzelnen Operationen in Hashtabelle und deren Vergleich mit den Operationen in der doppelt verkettete Liste

Die Operationen wie Einfügen und Löschen **in der Hashtabelle** werden mit einer durchschnittlichen konstanten Laufzeitkomplexität von O(1) durchgeführt [6]. Der Index wird berechnet und das Element wird an der Stelle effizient entfernt/hinzugefügt. Daher bleibt die Laufzeit unabhängig von der Anzahl der Elemente in der Hashtabelle konstant.

In Bezug auf die Laufzeit sind die Operationen Einfügen und Löschen **in einer doppelt verketteten Liste** ähnlich. Die Verweise der Knoten werden jeweils aktualisiert, um den neuen Knoten einzufügen/ zu löschen. Die Operationen haben aber eine konstante Laufzeitkomplexität von O(1) und die Laufzeit bleibt konstant, unabhängig von der Anzahl der Elemente [7].

Interessant wird es bei der Suchen-Operation. Nämlich dann versteht man, dass die Hashtabelle mit Double Hashing effizienter ist. Der Grund dafür ist, dass die Suche **in einer doppelt verketteten Liste** eine lineare Laufzeitkomplexität von O(n) hat [7], wobei n die Anzahl der Elemente ist. Da die Liste von Anfang bis Ende durchlaufen werden muss, um das gesuchte Element zu finden, steigt die Laufzeit linear mit der Anzahl der Elemente an. Im Vergleich dazu, hat die **Hash-Tabelle mit Double Hashing** eine Laufzeitkomplexität für die Suche, genau wie für Hinzufügen und Löschen - O(1) [6]. Es ist ja dabei möglich, direkt auf den Index des gesuchten Elements direkt zuzugreifen, so bleibt die Laufzeit unabhängig von der Anzahl der Elemente konstant.

Quellen:

1. https://de.wikipedia.org/wiki/Doppel-Hashing
2. https://moodle.htw-berlin.de/pluginfile.php/1737310/mod_resource/content/4/ALGO-Exercise-5-Guide.html#double-hashing
3. https://www.geeksforgeeks.org/java-program-to-implement-hash-tables-with-double-hashing/
4. https://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16-8.html
5. https://www.geeksforgeeks.org/java-program-to-check-if-a-number-is-prime-or-not/
6. https://medium.com/nerd-for-tech/the-magic-of-hash-tables-a-quick-deep-dive-into-o-1-1295199fcd05
7. https://www.oreilly.com/library/view/php-7-data/9781786463890/c5319c42-c462-43a1-b33d-d683f3ef7e35.xhtml