**Protokoll Aufgabe 1**

**Funktionen:**

**hashFunc():**

Man nehme die Summe der ASCII-Werte der Kürzel. Man nehme an, das somit die kleinst mögliche Summe 195 (AAA) ist, und die größtmögliche 360 (ZZZZ). Damit kann man nun eine annäherungsweise normalverteile Formel für diesen Zahlenbereich aufstellen:

(Summe – 195) \* HASH\_TABLE\_SIZE / (360 – 195)

Mit dieser Formel werden alle Werte von 195 – 360 gleichmäßig auf die Hashtabelle verteilt. Damit aber bei Werten, die kleiner oder größer sind, die Formel nicht aus dem “erlauben” Zahlenbereich (0 – HASH\_TABLE\_SIZE) herausfällt, muss die ganze Formel noch zum Schluss Modulo gerechnet werden. Damit sieht die finale Hashfunktion so aus:

**((Summe – 195) \* HASH\_TABLE\_SIZE / (360 – 195)) % HASH\_TABLE\_SIZE**

**Kollisionserkennung:**

Mit zwei quadratischen Sondierungsfunktionen quadraticProbingForName() und quadraticProbingForKuerzel() wird die Kollisionserkennung vorgenommen. Diese wurden nochmal unterteilt für adding und searching, da diese in beiden fällen andere Daten(typen) zurückliefern. Man könnte hier auch dieselbe Funktion überladen, jedoch haben wir uns in der Implementierung für separate Implementierungen bzw. Formelnamen entschieden.

**Verwaltung der Kursdaten:**

Es können insgesamt theoretisch 2002 Aktien gespeichert werden (Hashtabelle aber ausgelegt für 1000), welche drei Strings enthalten: Name, WKN und das Kürzel. Die Kursdaten selbst sind in einem weiteren struct-Array mit der Länge 30 gespeichert. Der jeweilige Zugriff über den Namen erfolgt über eine separate Hashtabelle aus mapNameToKuerzel structs.

**Löschalgorithmus:**

Die clearHeap()-Funktion löscht linear Elemente, wenn die Werte ungleich NULL sind.

**Aufwandsabschätzung:**

Alle Operationen bis auf das Löschen der Aktie stimmen in unserer Implementierung mit dem ideal überein. Der Füllgrad erfüllt die Mindestanforderungen, um bei maximal 1000 Aktien einen Füllgrad von 50% nicht zu überschreiten. Bei dieser Größe benötigt man bei unserer Datenstruktur mindestens eine Operation und maximal 1000.

Im Vergleich dazu sind die Zugriffe, das Auslesen und die Löschung einer Aktie in einem Array deutlich performanter mit jeweils O(1). Eine Verkettete Liste weist hingegen nur beim auslesen und löschen einen O(1) Aufwand auf, wenn der Optimalfall eintritt. Ansonsten ist das Einfügen grundsätzlich linear, genauso wie die durchschnittlichen und die am wenigsten performanten Fälle des Einlesens und Löschens.

**Aufwandsabschätzung Übersicht:**

**FALSCH**

|  |  |  |  |
| --- | --- | --- | --- |
| **Einfügen** | **Hashmap** | **Array** | **Verkettete Liste** |
| Best Case | **O(1)** | **O(1)** | **O(n)** |
| Avg Case | **O(1)** | **O(1)** | **O(n)** |
| Worst Case | **O(n)** | **O(1)** | **O(n)** |
| **Löschen** |  |  |  |
| Best Case | **O(1)** | **O(1)** | **O(1)** |
| Avg Case | **O(1)** | **O(1)** | **O(n)** |
| Worst Case | **O(n)** | **O(1)** | **O(n)** |
| **Auslesen** |  |  |  |
| Best Case | **O(1)** | **O(1)** | **O(1)** |
| Avg Case | **O(1)** | **O(1)** | **O(n)** |
| Worst Case | **O(n)** | **O(1)** | **O(n)** |