# Beschreibung Datenstruktur

## Hashfunktion

**public** **static** **int** **hashCode**(String str) {

**int** **result** = 0;

**for** (**int** **i** = 0; i < str.length(); i++) {

result += str.charAt(i) \* (I + 1);

}

**return** result;

}

Berechnet den Hashcode eines gegebenen Strings *str*.

Dies geschieht indem der ASCII Code des aktuellen Zeichens mit dem um 1 inkrementierten Index multipliziert wird. Folgend wird jeder der so erhaltenen Werte aufsummiert und zurückgegeben.

Für die Berechnung des Index wird der Hashcode modulo (Größe der Hashtable) berechnet um die Grenzen der Hashtable nicht zu überschreiten.

## Kollisionserkennung

Für die Kollisionserkennung wurde der Index einer gegebenen Aktie berechnet und gegen die Hashtabelle abgeglichen. Wurde an der angegeben Stelle (dem Index) bereits eine eingetragene Aktie gefunden, so wird mittels „quadratischem probing“ ein Nachfolgeindex gesucht.

Diese Suche wird so lange fortgesetzt bis ein Wert an einem Index auf „NULL“ gesetzt ist und somit der neue Aktienwert eingetragen werden kann. Ist die Hashtabelle so gut wie voll, wird der probing Algorithmus mehr Zeit in Anspruch nehmen, ist die Hashtabelle allerding voll, so wird die Suche gar nicht erst begonnen und direkt „false“ in der Einfüge-Methode „putStockByXXX“ returniert.

## Quadratisches probing

Die beiden einfachsten probing Methoden für Hashtabellen sind unter anderem das „lineare“ und das „squared“ (bzw. quadratische“) Probing. Der einfachste Fall der Implementierung ist das lineare probing und kann wie folgt implementiert werden.

(index + 1) % maxCapacity;

Wobei *index* den aktuellen Index der Hashfunktion angibt und *maxCapacity* die gesamte Größe der Hashtabelle. Der Index wird in diesem Algorithmus lediglich um den Wert eins erhöht, was in gewissen Fällen, wie zum Beispiel dem Fall, dass viele Werte hintereinander gespeichert sind, sehr inperformant werden. Mithilfe des modulo (%) Operators wird ein index overflow verhindert.

Die zweitere, etwas langsamere, aber bei sehr stark befüllten Hashtabellen schnellere Methode, ist as sogenannte „squared probing“. Bei diesem Algorithmus wird der aktuelle Index um eins erhöht und quadriert.

((index + 1) \* (index + 1)) % maxCapacity;

Dieser Algorithmus ist in vielen Implementierungen der Hashtabellen der weitaus bevorzugtere. Es ist hier notwendig, den Index um den Wert eins zu erhöhen, da sonst, im Falle des Index 0, null quadriert wird und sich am resultierenden Index nichts ändert.

## Verwaltung der Kursdaten

Die Verwaltung der Kursdaten wird über mehrere Klassen geführt. Darunter die Modellklassen (bzw. Datenklassen) „SingleStock.java“ und „StockDataset.java“. Modellklassen bestehen lediglich aus Datensätzen, „getter“ und „setter“ Methoden und führen keinerlei Kalkulationen aus.

Die Klasse „SingleStock.java“ ist vergleichbar mit einem Aktieneintrag und beinhaltet unter anderem die Abkürzung sowie den Namen der Aktie, außerdem wird eine Liste, befüllt mit allen zeilenweisen Einträgen, mitgeführt.

Die Klasse „StockDataset.java“ speichert alle Einträge, ähnlich einem Snapshot, der Aktie zu einer gegebenen Zeit. In dieser Modellklasse wird sowohl das Datum sowie die Uhrzeit als „java.util.Date“ gespeichert. Alle zugehörigen Datensätze werden als „double“ geführt. Darunter der höchste, niedrigste, sowie der Schluss- und der Öffnungswert einer Aktie.

Die Methoden „toString“, „equals“ und „hashCode“ wurden gemäß dem Java-Standard für Modellklassen standardisiert überschrieben.

Die eigentliche Verwaltung der Modellklassen der Aktien wird in der Klasse „StockHashtable.java“ geregelt. Die Implementierung besteht aus zwei Hauptarrays für die Referenzen zwischen Name und Kürzel und Kürzel und Aktieneintrag.

Wird in dieser Hashtabelle eine Aktie mithilfe des Namens gesucht, so wird der Hashcode (Index) für den gegebenen Namen berechnet und im Referenzarray für Name-Kürzel eingesetzt. Liefert das Array an der angegebenen Stelle ein konkretes Kürzel, so existiert auch ein Eintrag. Dieses Kürzel wird wiederrum gehasht (bzw. wird gehasht gespeichert) und mit diesem Index wird in der zweiten Referenztabelle mittels quadratischem Probing gesucht.

Wird für den Namen ein Eintrag gefunden, so **müssen** auch ein Eintrag für das Kürzel und der zugehörige Eintrag für die Aktie existieren.

Bei allen Operationen, die die Werte der Tabellen verändern, wird ein Zähler mitgeführt, damit die aktuelle Füllgröße und Füllrate der Hashtabelle immer bekannt sind und performant abgefragt werden können.

## Löschalgorithmus

Das Löschen einzelner Aktien wird mittels angegebenem Namen oder Kürzel durchgeführt. Im Falle von letzterem wird lediglich der zugehörige Name zum Kürzel gesucht und der Methode als Parameter übergeben.

Mithilfe des Namens wird der passende Index errechnet und die dazugehörige Aktie aus der Hashtable temporär gespeichert. Dies ist nötig da der Name mit jenem der angeforderten Aktie verglichen werden muss. Sind die Namen ident, so wurde die angeforderte Aktie gefunden und kann gelöscht werden.

Ist dies allerdings nicht der Fall, so muss genau wie beim Einfügen in die Hashtable „squared probing“ durchgeführt werden und kann dann ebenso mittels dem erhaltenen Index gelöscht werden.

Nach dem Löschen werden die übrigen Einträge innerhalb der Hashtable verschoben. Ausgangspunkt ist der errechnete Index, welcher nun leer ist

//TODO

## Aufwandsabschätzung

Im Falle der Hashtable verhält sich der Aufwand zum Einfügen, Suchen oder Löschen eines Datensatzes stets gleich. Ebenso gilt, dass das Suchen nach dem Namen sowie nach dem Kürzel in beiden Fällen einem fixen Zeitraum enspricht, woraus Folgendes resultiert:

Treten bei einer Operation dieser Typen keine Kollisionen auf, so entspricht der Aufwand für die Suche nach dem Namen **O(1)** und für die Suche nach dem Kürzel **O(2)**.

**O(1)** und **O(2)** sind als äquivalent zu betrachten.

Kommt es allerdings zu Kollisionen, beispielsweise wenn die Anzahl der Datensätze dem Füllgrad nahekommt, vergrößert sich der Aufwand zu **O(n)**.