**Protokoll AlgoDat Übung 1**

**Aktien-Hashtabelle**

**Übersicht und Abarbeitung/Auflistung der verlangten Infos**

**Programmiersprache:** Java, da wir bereits ein wenig Erfahrung in Java haben und wir wissen, dass man gute Libraries – etwa zum Parsen von CSV-Dateien – findet.

**IDE:** Eclipse, da wir damit schon öfter gearbeitet haben und wir die Git-Unterstützung zur Versionierung und zum Code sharen benötigten.

**Hash-Verfahren:** Die Divisions-Rest-Methode, die wir auf wikipedia gefunden haben und sich vor allem für Strings eignet.

**Hash-Tabellengröße:** 4001. Zunächst haben wir mit 10007 gearbeitet, weil der Speicherplatz irrelevant gering ist im Vergleich zum Speicherplatz, den das Programm benötigt. Doch für nur maximal 1000 Werte erschien es uns das dann doch etwas zu groß.

**Serialisierung:** Ist das kaskadierte Speichern eines Objekts, das heißt, falls es einen Verweis zu einem anderen Objekt hat, wird dieses ebenfalls mitgespeichert (sofern angegeben).

Daher scheint uns, dass das ser-Format das Geeignetste zum Abspeichern Tabelle ist.

**Aufwandsabschätzung:** Da wir sowohl beim Einfügen als auch beim Löschen eines Eintrags eine Suche durchführen (müssen), und danach der Aufwand konstant (= O(1)) ist, ist lediglich der Aufwand der Suche interessant. Zumindest wenn wir die Stringlänge, die linearen Aufwand in der Hash-Funktion hat, außen vor lassen.

Der Average-Case ist abhängig vom Füllgrad α = Einträge/Tabellengröße

α = 1/4 bei 1000 Einträgen 1/(1-α) = 1.33333…. Zugriffe

Der Best-Case ist immer O(1) und der Worst-Case ist immer O(n), da ich entweder meinen Eintrag sofort finde oder maximal n Kollisionen auftreten.

**Operationen zählen:** Hier sind wir vom Average-Case bei einer Füllung von 1000 ausgegangen.

Die Hash-Funktion hat (in Abhängigkeit zur Stringlänge) 1 + n Operationen

Die Suche (ohne Hashen) hat im Durchschnitt 8 (= Math.round(7.6666666)) Operationen

Das Einfügen (ohne Hashen und Suchen) hat im Durchschnitt 6 (=Math.round(4.3333333)) Operationen

Das Löschen (ohne Hashen und Suchen) hat 4 Operationen, falls es den Eintrag gibt

**Beschreibung**

**ADS --------> myHashtable ---------> Entry ---------> Share**

**Menü Hash-Tabelle Array-Element Aktie**

Im Grunde genommen enthält bei uns die Klasse *myHashtable* ein *Entry*-Array, welches 4001 (eine nicht-Mersenne-Primzahl) lang ist.

In einem solchen *Entry* befinden sich Name, Kürzel, WKN, eine Flag, die anzeigt, ob dieser Eintrag gelöscht worden ist, als auch ein *Share*-Objekt, welches beim Konstruieren eines *Entrys* vorerst auf NULL gesetzt wird.

Ein *Share*-Objekt besitzt sieben Arrays zur Abspeicherung aller Spalten einer CSV-Datei, die dem angegebenen Format entspricht.

Wichtig ist, dass wir zwei *myHashtables* gleichzeitig führen – einen für die Namen der Aktien und einen für die Kürzel. Allerdings besitzen beide denselben Pointer (an verschiedenen Indezes) für einen Eintrag, wodurch einige Prozeduren (zB Löschen) wesentlich erleichtert werden.

Des Weiteren machen wir uns die Zugriffsbeschränkungen einer objektorientierten Programmiersprache zunutze: Über das Menü (= die *ADS*-Klasse), kann man nur auf die *myHashtable*-Klasse zugreifen, die mit ihren Funktionen wiederum public-Funktionen der *Entry*-Klasse aufruft, die dann schlussendlich über public-Funktionen der *Share*-Klasse auf die importierten Daten zugreifen.

Der Löschalgorithmus ist dadurch sehr einfach: Es wird in einer der beiden *myHashtables* gesucht, ob der Eintrag vorhanden ist. Falls ja, dann wird die *deleteShare()*-Funktion von *Entry* aufgerufen, die genau die *deletedFlag* auf true und das *Share*-Objekt auf NULL setzt. Der Garbage-Collector gibt für uns den von der *Share* allokierten Speicher frei und da beide *myHashtables* denselben *Entry*-Pointer besitzen, sieht die andere bei der nächsten Suche, dass dort die *deletedFlag* gesetzt ist.

**Der Divisions-Rest-Algorithmus**

Ist zwar auf <http://de.wikipedia.org/wiki/Divisionsrestmethode> sehr gut erklärt, dennoch werden wir ihn kurz ansprechen:

Der Eingabe-String durchläuft folgende Schleife, wobei das erste Zeichen beim ersten Durchlauf das „aktuelle Ergebnis“ ist:

Man multipliziert das aktuelle Ergebnis mit der Alphabetlänge #∑ addiert das nächste Zeichen und nimmt die Summe modulo der Tabellengröße, die eine nicht-Mersenne-Primzahl sein sollte, da diese weniger Kollisionen aufweisen, als andere Zahlen.

Durch die Multiplikation mit #∑ werden alle Bits frei gemacht, die von den Zeichen belegt werden können. Da im nächsten Schritt genau diese wieder mit dem nächsten Zeichen befüllt werden, werden Kollisionen mit ähnlichen Wörtern bestmöglich vermieden.

Das Einzige, worauf man achten muss, ist, dass das maximale Zwischenergebnis in die Reichweite des gewählten Datentyps passt – in unserem Fall Integer.