**Allgemeines:**

Das Programm ist in 2 Klassen aufgeteilt, Controller und Hashtable. Unter Controller befindet sich eine Instanz der Hashtable Klasse und der grundlegende Menu Loop. Zudem werden die meisten Ausgaben und Inputs von diesen Methoden verarbeitet und weitergegeben an Methoden der Hashtable Klasse. In der Hashtable Klasse sind Methoden welche direkt mit der Hashtable arbeiten und Einträge suchen, löschen etc.. In der Hashtable Klasse befinden sich also Attribute wie die Hashtable selbst, welches ein Array aus Stock Strukturen ist, ein Dictionary, welches ebenfalls ein Array ist, jedoch aus Entry Strukturen, und ein Zähler für die Anzahl der Einträge.

Da nach Name oder Kürzel gesucht werden kann wurde das zweite Array namens Dictionary hinzugefügt. Wenn der Benutzer also nach einer Aktie sucht und das Kürzel angibt wird dieses gehashed, danach wird der Eintrag in der Hashtable gesucht und eventuell sondiert, und bei Erfolg der Index in der Hashtable zurückgeliefert. Wenn mit dem Namen gesucht wird dann kommt bei der Hashfunktion fast immer ein anderer Wert heraus und die Aktie kann nicht gefunden werden. Deswegen gibt es das Dictionary Array, wo für jede Aktie ein Eintrag gemacht wurde und mithilfe des Namen der Index errechnet wurde. Wenn der Name übereinstimmt dann kann in dem Eintrag der Index für die eigentliche Hashtable gefunden werden.

Da der Hashtable und dem Dictionary maximal 1000 Einträge eingefügt werden, wurde eine Größe von 2003 Einträge festgesetzt, da bei der quadratischen Sondierung ab einem Füllgrad von 50% eventuell kein freier Platz mehr gefunden werden kann. Außerdem ist 2003 die erste Primzahl nach 2000.

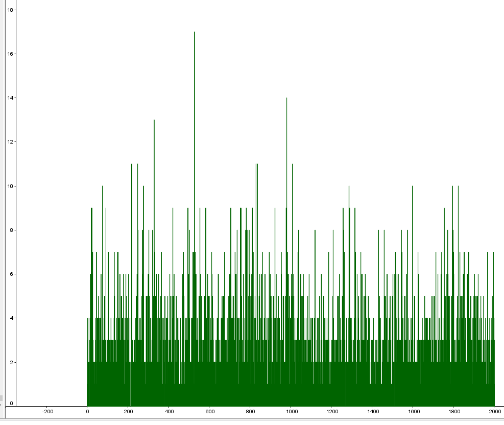
Die Hashtable und das Dictionary werden beide am Heap allokiert, da die Größe ungefähr 700 KB beträgt. Ursprünglich war geplant alles am Stack zu speichern, da die Operationen schneller wären, jedoch kommt die Größe gefährlich nahe an die maximale Stacksize unter Windows (1 MB). Man könnte natürlich auch ein Array aus Pointern machen und die einzelnen Einträge erst dann allokieren oder initialisieren, wenn sie benötigt werden. Das würde die Suche und das Sondieren etwas verändern, da auf bestimmte Attribute zugegriffen wird (filled = true/false). Beim anlegen müsste das Array quasi mit NULL-Pointern befüllt werden und dann könnte auf die Art gecheckt werden ob an der Stelle bereits ein Eintrag ist. Selbst dann würde mit wachsendem Füllgrad die Größe steigen und Probleme bereiten, also wäre wiederum eine Allokierung am Heap notwendig und da diese mal nicht alle Einträge auf einmal allokiert werden, müssten man das Löschen in einer anderen Funktion machen und man könnte nicht die gesamte Deallokierung im Destruktor der Klasse haben.

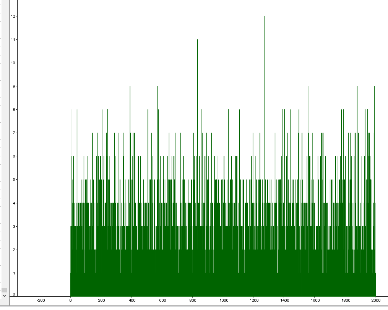
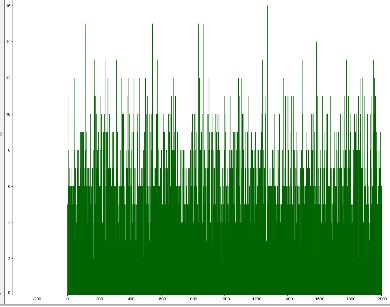
Im Code werden oftmals statt herkömmlichen int Typen die Definitionen der cstdint Library verwendet, da zB. die Hashfunktion mit Overflow arbeitet und gleiche Ergebnisse für jeden Rechner herauskommen sollen. Auch für die Sondierung wurden derartige Typen verwendet, da auch diese, auch wenn es sehr lange dauern würde, irgendwann overflowen könnte.

**Hashfunktion:**

Um eine geeignete Hashfunktion zu finden haben wir uns entschieden einen eigenen Tester zu programmieren, der aus einer Datei Kürzel oder Namen einliest und in eine andere Datei die jeweiligen Hashwerte. Damit konnten dann Durchschnittswerte und, besonders wichtig, Grafiken erstellt werden um zu sehen wie die Verteilung der Werte ist.

Als Input für die Grafiken wurden um die 5.000 Kürzel verwendet, wobei beim genaueren Testen dann auf 10.000 bis über 100.000 Kürzel erweitert wurde (keine Grafiken mehr möglich gewesen, da GeoGebra dafür verwendet wurde und bei mehr als 10.000 Werten nicht mehr funktionieren wollte).

Als erstes wurde die Hashfunktion von Java als Beispiel genommen, wobei mit verschiedenen Basen herumprobiert wurde. Hierbei gibt es eine maximale Kollision von 17 Einträgen bei 5.000 Kürzeln und einer Größe von 2003. Niedrigere Basen haben beim Testen eher schlechter abgeschnitten. Höhere Basen haben bis zu einem gewissen Grad besser abgeschnitten, jedoch kam es ab einem bestimmten Punkt zu einer gewaltigen Ansammlung an 0 Werten, wobei um ehrlich zu sein nicht ganz klar war warum es dazu kam, da bereits niedrige Basen mit Overflow gearbeitet haben und keine Problem hatten.

Obwohl die Hashfunktion auf den ersten Blick bereits eine relativ gute Verteilung hatte, war die Berechnung trotzdem etwas kompliziert. Deswegen haben wir getestet wie das Multiplizieren der einzelnen Buchstabenwerte die Werte verteilt und waren sehr überrascht. Bei 5.000 Werten war die maximal Kollision bei 12, bei 10.000 bei 16. Der Durchschnitt über 100.000 Werte Betrug 1001,76 wobei der Optimalwert 1001,5 wäre. Da die Funktion auch für das Hashen der Namen genutzt wird wurde auch mit einer Liste der Namen getestet und das Ergebnis war ähnlich, wobei bei Namen ab einer gewissen Länge und Zeichensatz eine vermehrte Anzahl von 0 Werten bemerkt wurde und deswegen die Länge der Namen auf 40 Zeichen limitiert wurde.

Der Hashtester ist bei der Abgabe dabei, wobei sowohl Input als auch Output in eine .csv Datei gemacht wird.

**Kollisionserkennung:**

Für die Behandlung von Kollisionen wurden die quadratische Sondierung verwendet, welche im Falle einer Kollision zum derzeitigen Index 1², 2², 3³ etc. hinzurechnet. Die Hashfunktion liefert also nur den Hashwert des Inputs (Kürzel oder Name) und bei der jeweiligen Operation selbst wird gecheckt ob es zu einer Kollision gekommen ist. Dieser Check könnte natürlich auch in der Hashfunktion selbst ausprogrammiert werden, jedoch werden bei manchen mehr Information benötigt als ein freier Index.

**Überblick der Funktionen:**

**ADD:**

Aktien Struktur wird angelegt und befüllt. Sowohl der Name als auch das Kürzel werden gehasht. Das Kürzel wird verwendet um den Wert in die Hashtable einzufügen und bei Erfolg, also wenn keine Aktie mit gleichem Namen gefunden wird, wird die Information an dem Index mit der neuen überschrieben. Danach wird dasselbe mit der Name gemacht, nur falls es hier zu einer Übereinstimmung kommt wird der Eintrag aus der Hashtabelle wieder gelöscht, da kein Eintrag gemacht werden kann.

**SEARCH:**

Da beim Programmierbeispiel eine quadratische Sondierung verwendet werden soll muss man dies bei der Suche beachten. Der Benutzer soll nach dem Namen oder nach dem Kürzel der Aktie suchen können.

* Wird ein Kürzel eingegeben wird dieses gehashed wodurch man den Index bekommt. Auf diesem Index wird jetzt nachgeschaut, ob der User Input (Kürzel) gleich dem Kürzel ist, das im Hashtable gespeichert ist, falls diese übereinstimmen, werden die Kurswerte ausgegeben. Falls die Kürzel aber nicht übereinstimmen und der Eintrag nicht leer ist oder deleted auf true gesetzt ist wird die Suchfunktion rekursiv aufgerufen diesmal aber mit der quadratischen Sondierung. Dies passiert so lange bis ein leerer Eintrag mit deleted auf false gefunden wird, was eine erfolglose Suche bedeutet. Hierbei wird -1 returned und eine Fehlermeldung ausgegeben.
* Wird ein Name angegeben wird dieser gehashed was den Index des Namens zurückliefert. Nun wird in einer zweiten Tabelle worin sich der Aktienname, das Aktienkürzel und der Index vom gehasheden Aktienkürzel befindet verglichen, ob der eingegebene Name gleich dem abgespeicherten Namen ist. Wenn das der Fall ist, wird der Index vom gehasheden Kürzel returned und die aktuellen Kurswerte werden ausgegeben. Falls der Name aber nicht übereinstimmen und der Eintrag nicht leer ist oder deleted auf true gesetzt ist wird die Suchfunktion rekursiv aufgerufen diesmal aber mit der quadratischen Sondierung. Falls auf einen leeren Eintrag mit deleted auf false gestoßen wird wurde nichts gefunden und es wird -1 returned und eine Fehlermeldung ausgegeben.

**DELETE:**

Da die quadratische Sondierung eine offene Adressierung ist, gibt es 2 verschiedene Lösungswege zum Löschen. Man kann nicht einfach ein Element rauslöschen da es sonst die Sondierungskette unterbrechen würde. Der erste Ansatz wäre die Umsortierung der gesamten Tabelle, wenn ein Element gelöscht wird. Dies ist sehr kompliziert und ist mit einem sehr hohen Aufwand verbunden.

Der zweite Ansatz wäre das man die Einträge auf dem Index der „gelöscht“ werden soll einfach leert und ein delete Attribut hinzufügt, das man dann auf true setzt. Somit kann man wieder ein neues Element hinzufügen. Dieser Ansatz erfordert aber ein paar Änderungen in der add und search Funktion da jetzt zusätzlich abgefragt werden muss, ob der Index frei bzw deleted auf true ist.

Ansich ist die Löschfunktion wie die Suchfunktion, nur dass nach erfolgreicher Suche der Eintrag gelöscht und das deleted Attribut eben auf true gesetzt wird. Hierbei ist anzumerken, dass sowohl der Eintrag in der Hashtable als auch im Dictionary gelöscht werden muss, wobei etwas Performance hergeschenkt wird, da die Search Funktion wiederverwendet wird, welche nur den Index des Eintrags in der Hashtable zurückliefert. Dies wäre kein Problem wenn der Benutzer nach dem Kürzel sucht, jedoch wenn nach dem Namen gesucht wird, wird zuerst der Name gehashed und der Index aus dem Dictionary gesucht und danach der Delete Funktion übergeben, wobei hierbei wiederum aus dem Namen der am Index in der Hashtabelle gefunden wird der Index des Eintrages im Dictionary gesucht wird um diesen zu löschen. Man hätte eine eigene Funktion dafür schreiben können, jedoch wurde der Einfachheit und Übersichtlichkeit halber dagegen entschieden.

**IMPORT:**

Beim Import wird wieder eine Suche nach dem Index gemacht und danach eine Datei mit dem Namen KÜRZEL.csv im import Ordner gesucht. Diese wird dann Zeile für Zeile eingelesen, wobei die Werte in einem zweidimensionalen Vektor gespeichert werden. Hierbei werden auch die Zeilen gezählt. Da nur die letzten 30 Werte gespeichert werden sollen muss die Datei mindestens 30 Datensätze beinhalten, sonst wird der Import abgebrochen. Andernfalls werden die hinteren 30 Einträge aus dem Vektor in den eigentlichen Vektor in der Hashtabelle gespeichert.

**PLOT:**

Es wird wiederum nach dem Index der Eingabe gesucht und bei erfolgreicher Suche und vorhanden sein von Daten ein Graph ausgegeben. Hierbei wird auf eine Einteilung von 15 Zeilen gesetzt, also die einzelnen Werte in 15 Gruppen eingeteilt. Um die Präzision zu erhöhen kann die Einteilung erhöht werden.

**SAVE:**

Um die Hashtabelle und das Dictionary zu speichern kann die Save Funktion verwendet werden. Hierbei werden die Inhalte in eine .csv Datei gespeichert. Ursprünglich war der Plan zur Serialisierung eine externe Library zu verwenden (boost), jedoch gab es damit ein paar Schwierigkeiten, da die Datenstruktur die wir serialisieren wollten nicht aus einer Klasse mit ein paar Integern und Strings bestand. Im Nachhinein haben wir die Library Cereal von einem Kollegen empfohlen bekommen, jedoch war da schon die Save und Load Funktion fertig programmiert.

Ansich wird bei der Save Funktion einfach nur die Hashtabelle und das Dictionary durchlaufen und für jeden Eintrag wo entweder wirklich Inhalt vorhanden ist oder einfach nur das deleted Attribut auf true gesetzt ist, eine Zeile in der Datei verwendet. Zur Trennung von Hashtabelle und Dictionary wird eine Zeile mit „START\_DICTIONARY“ eingefügt, wobei hier auch etwas anderes stehen könnte, da lediglich eine Unterscheidung zu einem Index bewerkstelligt sein muss.

**LOAD:**

Die Load Funktion ist das Gegenstück zur Save Funktion. Hierbei wird die Datei, welche von der Save Funktion erstellt wurde ausgelesen (save.csv). Aufgrund des Aufbaus der Datei werden an den jeweiligen Indizes die Werte eingetragen. Selbiges für das Dictionary. Anzumerken ist nur, dass beim Laden die vorherige Hashtabelle und das Dictionary gelöscht werden.

**Aufwandsabschätzung:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Unsere Lösung (Hashtabelle & Dictionary)** | **Array** | **Linked List** |
| **Einfügen** | Best Case: O(kürzel) + O(name) + O(1) + O(1) Average Case: O(kürzel) + O(name) + O(1) + O(1) Worst Case: O(kürzel) + O(name) + O(n) + O(n) | Best Case: O(1) Average Case: O(1) Worst Case: O(n) | Best Case: O(1) Average Case: O(1) Worst Case: O(1) |
| **Suchen** | Best Case 1: O(kürzel) + O(1) Best Case 2: O(name) + O(1) Average Case 1: O(kürzel) + O(1) Average Case 2: O(name) + O(1) Worst Case 1: O(kürzel) + O(n) Worst Case 2: O(name) + O(n) | Best Case: O(1) Average Case: O(n) Worst Case: O(n) | Best Case: O(1) Average Case: O(n) Worst Case: O(n) |
| **Löschen** | Best Case 1: O(kürzel) + O(name) + O(1) + O(1) Best Case 2: O(kürzel) + O(name) + O(1) + O(1) Average Case 1: O(kürzel) + O(name) + O(1) + O(1) Average Case 2: O(kürzel) + O(name) + O(1) + O(1) Worst Case 1: O(kürzel) + O(name) + O(n) + O(n) Worst Case 2: O(kürzel) + O(name) + O(n) + O(n) | Best Case: O(1) Average Case: O(n) Worst Case: O(n) | Best Case: O(1) Average Case: O(n) Worst Case: O(n) |

Hierbei muss angemerkt werden, dass beim Löschen bei einem Array und der Linked List der Suchvorgang miteinberechnet wurde. Außerdem kann das Einfügen nicht so gut verglichen werden, da bei uns in zwei Arrays eingefügt werden muss.