**Projekt 1: Aktienverwaltung**  Gruppe:

Bajrektarevic Minea

Liu Yueting

**Protokoll: Aktienverwaltung mit HashTable**

### **1. Beschreibung der Datenstruktur**

Die Aktienverwaltung erfolgt mithilfe einer **HashTable**, um Aktien und deren Kursdaten effizient zu speichern und abzurufen. Eine HashTable bietet eine schnelle **O(1) Suchzeit im Durchschnitt**, indem sie einen Schlüssel (WKN) auf eine Speicheradresse innerhalb eines Arrays abbildet.

Jede Aktie enthält folgende Informationen:

* **Name** (string) → Firmenname
* **WKN** (string) → Wertpapierkennnummer, Primärschlüssel
* **Ticker** (string) → Kurzsymbol der Aktie
* **Kursdaten** (vector) → Historische Börsenkurse

Die Kursdaten (Kurs) enthalten:

* Datum (string)
* Schlusskurs (Close) (double)
* Handelsvolumen (Volume) (long)
* Eröffnungskurs (Open) (double)
* Tageshöchstkurs (High) (double)
* Tagestiefstkurs (Low) (double)

Jede Aktie wird anhand der WKN in der HashTable gespeichert.

### **2. Hashfunktion**

Die Hashfunktion verwendet die **djb2**-Methode, eine bewährte Hashfunktion für Zeichenketten:

size\_t HashFunction::operator()(const std::string& key) {  
 size\_t hash = 1031;  
 for (char c : key) {  
 hash = ((hash << 5) - hash) + c; // hash \* 331+ c  
 }  
 return hash % tableSize; // Modulo-Operation für Begrenzung der Tabelle  
}

Die Hashfunktion nimmt die WKN als Eingabe und gibt einen Index innerhalb der Tabelle zurück.

### **3. Kollisionserkennung & -lösung**

Da Hashfunktionen unterschiedliche Schlüssel auf denselben Index abbilden können, wird **quadratische Sondierung** als Kollisionslösung verwendet:

size\_t i = 0;  
while (status[index] == SlotStatus::OCCUPIED) {  
 index = (originalIndex + i \* i) % tableSize;  
 i++;  
}

* Falls der berechnete Index bereits belegt ist, wird die Suche mit i²-Schritten fortgesetzt.
* Dies verhindert Primärclusterbildung und verbessert die Verteilung.

### **4. Verwaltung der Kursdaten**

Kursdaten werden beim **Import aus einer CSV-Datei** geladen und in einem Vektor (std::vector<Kurs>) innerhalb der Aktienstruktur gespeichert.

Die **CSV-Datei** hat folgendes Format:

Date,Close,Volume,Open,High,Low  
03/14/2025,213.49,60107580,211.25,213.95,209.58  
03/13/2025,209.68,61368330,215.95,216.83,208.42

Beim **Import** werden die Werte extrahiert und konvertiert:

if (std::getline(ss, token, ',')) {  
 k.close = std::stod(token);  
}

Die importierten Kurse werden in kursdaten.push\_back(k); eingefügt.

### **5. Löschalgorithmus**

Löschen erfolgt durch **lazy deletion**, indem der Slot-Status auf DELETED gesetzt wird:

size\_t index = hashFunction(wkn);  
while (status[index] != SlotStatus::EMPTY) {  
 if (status[index] == SlotStatus::OCCUPIED && table[index]->wkn == wkn) {  
 delete table[index];  
 table[index] = nullptr;  
 status[index] = SlotStatus::DELETED;  
 return true;  
 }  
 index = (index + 1) % tableSize;  
}  
return false;

Dadurch können zukünftige Einträge die DELETED-Slots wiederverwenden.

### **6. Aufwandsabschätzung (Best-/Worst-/Average-Case)**

#### **6.1. Einfügen (add())**

|  |  |
| --- | --- |
| **Fall** | **Aufwand** |
| Best-Case | O(1) - Direktes Einfügen ohne Kollision |
| Average-Case | O(1) - Quadratische Sondierung selten notwendig |
| Worst-Case | O(n) - Wenn die HashTable voll ist |

#### **6.2. Suchen (search())**

|  |  |
| --- | --- |
| **Fall** | **Aufwand** |
| Best-Case | O(1) - Direktes Auffinden am berechneten Index |
| Average-Case | O(1) - Wenige Kollisionsprüfungen nötig |
| Worst-Case | O(n) - Bei vollständiger Belegung und ungünstigen Kollisionen |

#### **6.3. Löschen (remove())**

|  |  |
| --- | --- |
| **Fall** | **Aufwand** |
| Best-Case | O(1) - Direkte Entfernung aus HashTable |
| Average-Case | O(1) - Wenige Sondierungen notwendig |
| Worst-Case | O(n) - Wenn viele Kollisionen bestehen |

### **7. Berechnung der Load Factor (Ladungsrate)**

Die Load Factor berechnet sich als:

Beispiel:

* tableSize = 1031
* OCCUPIED Slots = 2
* Ladungsrate=2/1031

Ein niedriger Load Factor zeigt eine **effiziente Verteilung**.

### **8. Serialisierung & Fazit**

Die HashTable unterstützt **Serialisierung**, indem sie Aktien- und Kursdaten in einer CSV-Datei speichert und wieder lädt. Die Speicherung erfolgt im CSV-Format, während das Laden die gespeicherten Daten wieder in die HashTable überträgt.

* Die HashTable implementiert eine **effiziente Verwaltung** von Aktien und Kursdaten.
* Quadratische Sondierung reduziert Clusterbildung.
* **Lazy Deletion** optimiert das Löschen von Elementen.
* Die **Laufzeiten für Suchen & Einfügen** sind im Durchschnitt **O(1)**.
* Die **djb2-Hashfunktion** ermöglicht eine schnelle und gleichmäßige Verteilung.

Das System kann schnell **große Mengen an Aktienkursen** speichern und abrufen. Falls die Anzahl an Daten wächst, kann eine **dynamische Rehashing-Strategie** hinzugefügt werden.