**1. Datenstruktur & Hashfunktion**

Die Hashtabelle speichert Aktien über **Symbol & Name** in zwei Arrays (table und nametable). Jede Aktie enthält:

* std::string name, symbol (zur Identifikation)
* std::vector<PriceData> (letzte 30 Kurswerte mit date, open, high, low, close, volume)

**Hashfunktion (DJB2)** für Strings:

int hashFunction(const std::string& key) const

{

unsigned int hash = 5381; //ausgewählte Primzahl als Startwert

for (char c : key)

{

hash = ((hash << 5) + hash) + c; // hash = hash \* 33 + c;

}

return hash % TABLE\_SIZE; // sorgt dafür, dass der Wert in den Bereich 0 - 1008 fällt

}

**Kollisionsbehandlung**: Quadratische Sondierung:

quadraticProbing(int hash, int i) const

{

return (hash + i \* i) % TABLE\_SIZE;

}

**2. Aufwandsabschätzung**

**Vergleich Hashtabelle vs. Andere Strukturen**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Hashtabelle (Ø O(1))** | **Array (O(n))** | **Liste (O(n))** |
| Einfügen | **O(1)** (direkte Adressierung) | O(n) (suche Position) | O(1) (am Anfang) |
| Suchen | **O(1)** (direkte Hashsuche) | O(n) (lineare Suche) | O(n) (durchlaufen) |
| Löschen | **O(1)** (markieren als gelöscht) | O(n) (neu ordnen) | O(n) (Suchen + Entfernen) |

**1000 Aktien & Worst-Case Betrachtung**

* **Best-Case (keine Kollisionen)**: Alle Operationen laufen in **O(1)** ab.
* **Average-Case**: Geringe Kollisionen durch gut gewählte Hashfunktion, meist O(1).
* **Worst-Case (hohe Kollisionen, schlechtes Hashing)**:
  + Quadratische Sondierung reduziert Clusterbildung.
  + Laufzeit für Suchen & Einfügen kann auf **O(n)** steigen.

Hashtabelle bietet **schnelle Verwaltung & effiziente Suche** für Aktienkurse. Bei dieser Hashfunktion bleibt die Leistung nahe O(1).