Protokoll:

# Beschreibung der Datenstruktur und Hashfunktion:

Alles auf der Klasse Stockmanager aufbauend, welche alle Funktionen public hat, damit die main function darauf zugreifen kann. Wir haben außerdem eine private contstante, welche die maximale Anzahl der Stocks beschrenkt, numStocks ist die Anzahl der aktuell gespeicherten Stocks. 2 Haschtabellen: Symbols und Namen der Stocks-> um nach den Stocks später zu suchen.

2 structs:

Stocks, die einzelne Aktie, wobei name, wkn und symbol gespeichert werden, sowie die history in form eines Vektors.

Stockdata, da werden die historischen Daten abgespeichert.

Main:

Ein Menu, für die Usereingabe, wo über den Stockmanager die Funktionen abgerufen werden.

Die einzelnen Funktionen, rufen jeweils die Hashfunktionen auf, um jeweils die Daten aufzurufen und einzulesen.

Die Hashfunktion `customHash` wird verwendet, um eine Hash-Position für einen gegebenen String (Symbol oder Name) zu berechnen. Diese Funktion summiert die ASCII-Werte der Zeichen im String und multipliziert sie mit ihrer Position im String, um die Reihenfolge zu berücksichtigen. Das Ergebnis wird dann durch die maximale Anzahl von Aktien (`MAX\_STOCKS`) moduliert, um sicherzustellen, dass der Hash-Wert im gültigen Bereich liegt.

# Aufwandsabschätzung:

1. `customHash(const string& word)`: Die Laufzeit dieser Funktion beträgt O(n), wobei n die Länge des Eingabestrings ist. Da die Funktion jedes Zeichen im String einmal durchläuft, ist die Laufzeit linear zur Länge des Strings.

2. `addStock(const string& name, const string& wkn, const string& symbol)`: Die Laufzeit beträgt im schlimmsten Fall O(m), wobei m die maximale Anzahl von Aktien (`MAX\_STOCKS`) ist. Dies liegt daran, dass die Funktion im schlimmsten Fall alle Plätze in der Hashtabelle durchlaufen muss, um einen freien Platz zu finden.

3. `deleteStock()`: Die Laufzeit beträgt im schlimmsten Fall ebenfalls O(m), da sie ähnlich wie `addStock` funktioniert und im schlimmsten Fall alle Plätze in der Hashtabelle durchläuft, um die zu löschende Aktie zu finden.

4. `importStockData()`: Die Laufzeit beträgt O(m + n), wobei m die maximale Anzahl von Aktien (`MAX\_STOCKS`) und n die Anzahl der Zeilen in der Datei ist. Dies liegt daran, dass die Funktion im schlimmsten Fall alle Plätze in der Hashtabelle durchlaufen muss, um die zu importierende Aktie zu finden (O(m)), und dann jede Zeile in der Datei einmal durchläuft, um die Daten zu importieren (O(n)).

5. `searchStock()`: Im schlimmsten Fall beträgt die Laufzeit O(2m), da die Funktion zuerst in der Hashtabelle nach dem Namen und dann nach dem Symbol sucht. Jede Suche könnte im schlimmsten Fall alle Plätze in der Hashtabelle durchlaufen müssen, um die gesuchte Aktie zu finden.

6. `plotStock()`: Die Laufzeit beträgt O(m + n), wobei m die maximale Anzahl von Aktien (`MAX\_STOCKS`) und n die Anzahl der Datenpunkte in der Historie der Aktie ist. Dies liegt daran, dass die Funktion im schlimmsten Fall alle Plätze in der Hashtabelle durchlaufen muss, um die zu plottende Aktie zu finden (O(m)), und dann jeden Datenpunkt in der Historie der Aktie einmal durchläuft, um den Plot zu erstellen (O(n)).

7. `saveToFile()`: Die Laufzeit beträgt O(m \* n), wobei m die maximale Anzahl von Aktien (`MAX\_STOCKS`) und n die durchschnittliche Anzahl von Datenpunkten in der Historie jeder Aktie ist. Die Funktion durchläuft jeden Platz in der Hashtabelle (O(m)) und für jede nicht leere oder gelöschte Aktie durchläuft sie jeden Datenpunkt in ihrer Historie, um die Daten in die Datei zu schreiben (O(n)).

8. `loadFromFile()`: Die Laufzeit beträgt ebenfalls O(m \* n), wobei m die maximale Anzahl von Aktien (`MAX\_STOCKS`) und n die Anzahl der Zeilen in der Datei ist. Die Funktion durchläuft jede Zeile in der Datei einmal (O(n)) und für jede Zeile, die eine Aktie repräsentiert, durchläuft sie im schlimmsten Fall alle Plätze in der Hashtabelle, um einen freien Platz für die Aktie zu finden (O(m)).