Alaa Nakshbandi

PROTOKOLL – ALGOS: Programmieraufgabe 1

Allgemeines:

Das Programm wurde mit C++ geschrieben und hat 3 Klassen: stock, stockData, hashTable. Hashtabellen sind Arrays vom Typ stock. Stocks beinhalten ein Array namens data, der die Länge 30 hat, weil zu jeder Aktie die Einträge der letzten 30 Tagen gespeichert werden sollen.

- **Stock:**

```
class stock {
    string name;
    string wkn;
    string symbol;
    stockData data[MAX_DAYS]; // MAX_DAYS = 30
}
```

- **StockData:**

```
class stockData {
    std::string date;
    double open;
    double high;
    double low;
    double close;
    double adjClose;
    int volume;
}
```

- **HashTable:**

```
class hashTable {
    stock *table[TABLE_SIZE];
}
```

1. Hashfunktion

Die folgende Hashfunktion der Klasse „hashTable“ erhält einen Namen als Eingabe und generiert einen Hashwert für den Namen. Der Name wird allgemein genommen, damit die Hashfunktion sowohl auf Aktiennamen als auch auf Kürzel verwendet werden kann.

Zunächst werden alle Buchstaben des Namens in Kleinbuchstaben umgewandelt, um eine case-insensitive Suche zu ermöglichen. Damit können User die Aktien immer finden, unabhängig davon, ob sie beim Einfügen mit Groß- oder Kleinbuchstaben gespeichert wurden. Dann wird für jeden Buchstaben im Namen der ASCII-Wert des Buchstabens zum Hashwert addiert. Der Hashwert wird dann mit dem ASCII-Wert multipliziert und modulo der Größe der Hash-Tabelle berechnet, um den endgültigen Hashwert zu erzeugen.

Die Größe der Hashtabelle ist in define.h als 1301 definiert. Es sollen in der Tabelle maximal 1000 Aktien gespeichert werden können. Die optimale Größe wäre 1,3x die max. Anzahl an Werten, und die erste Primzahl, die in diesem Bereich liegt, ist 1301.

```
unsigned int hashTable::hash(string name){
// hash function that generates hash value for the given stock
    unsigned int hashValue = 0;

    // turn to lower case before hashing to make it case insensitive:
    string result = name;
    transform(result.begin(), result.end(), result.begin(), ::tolower);


    int length = size(result);
    for (int i = 0; i < length; ++i) {
        hashValue += result[i];
        hashValue = (hashValue * result[i]) % TABLE_SIZE;
    }
    return hashValue;
}
```

2. **Kollisionserkennung:**

In der Funktion „insertToTable“ wird eine Kollisionserkennung durch quadratische Probing-Technik implementiert. Wenn der Hash-Index bereits besetzt ist, wird die Schleife durchlaufen, bis ein leerer Index gefunden wird. Dabei wird die Funktion hi(x) = h(x) + i^2 mod m verwendet, wobei h(x) der ursprüngliche Hash-Index ist, i in der for-Schleife quadriert wird und m die Größe der Hash-Tabelle ist. Wenn ein leerer Index gefunden wird, wird der Wert in der Tabelle an dieser Position auf den neuen Wert gesetzt. Wenn kein leerer Index gefunden wird, gibt die Funktion „false“ zurück, um anzuzeigen, dass das Einfügen nicht erfolgreich war.

```
bool hashTable::insertToTable(string nameORsymbol, stock& stock1){
    // turn to lower case before hashing to make it case insensitive:
    string result = nameORsymbol;
    transform(result.begin(), result.end(), result.begin(), ::tolower);

    // hash the stock by its name and get its index
    int index = hash(result);

    // i *= i because we are using quadratic probing
    for (int i = 0; i < TABLE_SIZE; i *= i) {
    // hi(x) = h(x) + i^2 mod m ; wobei i in der for() quadriert wird und m
       die groesse der hashtabelle entspricht
    int test = (index + i) % TABLE_SIZE;
    // we can use null indexes bc they're empty or deleted indexes bc they
       are also now empty and available to use
    if (table[test] == NULL || table[test] == DELETED_NODE){
            table[test] = &stock1;
            return true;
        }
    }
    return false;
}
```

3. **Verwaltung der Kursdaten:**

Funktion hashTable::import()

Importiert Daten aus einer CSV-Datei für eine bestimmte Aktie in die Tabellen. Zu Beginn wird der Benutzer aufgefordert, den Namen der Aktie einzugeben. Der eingegebene Name wird in

Kleinbuchstaben umgewandelt, dann wird überprüft, ob die Aktie bereits in der Datenstruktur vorhanden ist, indem die Funktion checkStockExists() aufgerufen wird. Wenn die Aktie nicht gefunden wird, wird eine Fehlermeldung ausgegeben und die Funktion wird beendet. Wenn die Aktie gefunden wird, wird der Pfad und Dateiname der entsprechenden CSV-Datei erstellt und die Datei geöffnet. Wenn die Datei erfolgreich geöffnet wird, wird zuerst die Anzahl der Zeilen in der Datei gezählt, um später die letzten 30 Tage Daten zu lesen. Nachdem die Anzahl der Zeilen gezählt wurde, wird die Dateizeigerposition zurückgesetzt und die Daten der letzten 30 Tage werden in eine Datenstruktur stockData geladen. Dabei wird jede Zeile der CSV-Datei in stockData gespeichert und in das Datenfeld data der Aktie-Struktur geschrieben. Wenn das Ende der CSV-Datei erreicht ist oder 30 Tage Daten geladen wurden, wird die Schleife beendet.

Funktionen hashTable::searchThroughName() und hashTable::searchThroughSymbol()

Nachdem die Daten importiert werden, können diese Funktionen auf das Array data zugreifen und holen von dort vom index 29 die Daten des aktuellen Kurseintrags.

Funktion hashTable::plot()

Diese Funktion sucht nach einer bestimmten Aktie in der Hash-Tabelle und gibt dann die Schließwerte für die letzten 30 Tage aus. Wenn die Aktie nicht existiert oder keine Daten für die letzten 30 Tage importiert wurden, gibt die Funktion eine Fehlermeldung aus.

Grafik: Die x-Achse zeigt das Datum der letzten 30 Tage an, beginnend mit dem ältesten Datum am linken Rand der Grafik. Die y-Achse zeigt den Schließwert der Aktie an. Jeder Schließwert wird als horizontaler Balken aus Sternen (*) dargestellt. Die Länge des Balkens hängt von der Höhe des Schließwerts ab, wobei jeder Stern für eine Wertsteigerung von 10 steht. Der Schließwert wird rechts neben dem Balken angezeigt.

Funktionen hashTable::save() und hashTable::load()

Diese Funktionen exportieren bzw. importieren die Tabellen. Die Format entspricht <Index>,<Aktienname>,<WKN>,<Kürzel>,<Datum>,<open>,<high>,<…>

Jede Aktie wird in derselben Zeile mit ihrem Datenarray gespeichert. Eine neue Zeile entspricht eine neue Aktie.

4. Löschalgorithmus

Zum Löschen von Aktien gibt es 2 Löschfunktionen, die die Aktien aus beiden Tabellen löschen. Die Funktionen hashTable::deleteStockByName(string name) und hashTable:: deleteStockBySymbol(string symbol) nehmen den Namen- und Kürzel-Input vom User und verwenden sie in der Funktion. Die Inputs werden zu Kleinbuchstaben umgewandelt und gehasht. Danach wird mit dem Index, der daraus entsteht, mittels quadratischer Sondierung in den Tabellen gesucht. Die Suche erfolgt mit der Funktion checkStockExists(), diese returniert einen Pointer auf die Aktie wenn sie gefunden wird bzw. einen NULL-Pointer wenn nicht. Returniert sie einen Pointer auf eine Aktie, so wird von der Tabelle an dieser Stelle der Name bzw. der Kürzel geholt und mit dem User-Input verglichen. Wenn diese gleich sind, so

wurde die Aktie gefunden und kann gelöscht werden. Zum Löschen wird die Tabelle an der entsprechenden Stelle auf DELETED_NODE gesetzt und der Pointer zur Aktie wird deleted.

5. Aufwandabschätzung:

Aufwand zum Einfügen, Suchen und Löschen in einer einfach verketteten Liste:

- Einfügen: O(1)
- Suchen: abhängig vom Füllgrad (n/m)
    - Füllgrad < 1: Suchaufwand O(1)
    - Füllgrad > 1: Suchaufwand O(N)

Aufwand zum Einfügen, Suchen und Löschen in einem normalen Array:

- Best Case: O(1)
- Worst Case O(N)
- Average Case: O(N/2) = O(N)

Aufwand zum Einfügen, Suchen und Löschen in die Hashtabelle:

Einfügen:

Vor dem Einfügen soll gehasht werden. Das erfolgt mit der folgenden Schleife in der Methode hashTable::hash(string name):

```
for (int i = 0; i < length; ++i) {
    hashValue += result[i];
    hashValue = (hashValue * result[i]) % TABLE_SIZE;
}
```

Diese Methode berechnet den Hash-Wert für einen gegebenen String. Diese Schleife hat eine Zeitkomplexität von O(|name|), wobei |name| die Länge des Strings name ist.

Danach wird die Aktie in die Tabelle am generierten Index eingefügt. Dies erfolgt mit der Methode hashTable::insertToTable. Sie fügt einen gegebenen Aktien-Pointer in die Hashtabelle ein, indem sie die Hashtabelle mit quadratischer Sondierung durchläuft, um einen leeren Platz für die Einfügung zu finden. Der Aufwand beim Einfügen in die Hashtabelle in diesem Programm hängt von der Füllgrad ab und liegt bei O(1/(1-α)).

```
for (int i = 0; i < TABLE_SIZE; i++) {
    int test = (index + i*i) % TABLE_SIZE;
    if (table[test] == NULL || table[test] == DELETED_NODE){
        table[test] = &stock1;
        return true;
    }
}
```

Insgesamt hat das Einfügen also einen Aufwand von O(|name| + 1/(1-α)).

Suchen & Löschen:

Die Suche und das Löschen haben dieselbe Laufzeit wie das Einfügen, weil der gesuchte/zu löschende Input zuerst gehasht werden soll, und dann mittels quadratischer Sondierung in der Tabelle gesucht werden soll.

Daten Importieren:

1. Einlesen der Datei:

Dies hängt von der Größe der Datei ab und kann je nach Dateigröße und Geschwindigkeit des Speichers variieren. Wir schätzen hier eine O(n) Zeitkomplexität, wobei n die Anzahl der Zeilen in der Datei ist.

2. Schleifen über Zeilen und Spalten:

Für jede Zeile und jede Spalte müssen die Daten in die entsprechenden Variablen gespeichert werden. Da wir eine feste Anzahl von Spalten und Zeilen haben, beträgt die Zeitkomplexität O(1).

3. Überprüfung, ob Aktie in Tabelle existiert:

Dies wird mit der checkStockExists-Funktion durchgeführt. Die Zeitkomplexität dieser Funktion ist O(1), wenn die Aktie in der Tabelle existiert, und O(n), wenn die Aktie nicht in der Tabelle existiert.

Zusammenfassend ergibt sich folgende grobe Aufwandsschätzung für die import-Funktion: O(n) + O(1) + O(n) + O(1) = O(n)