Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Arbeitsgruppe Datenbanksysteme

Sommersemester 2020
28. 08. 2020

Alex El-Shaikh M.Sc.

Programmierpraktikum

Tag 1

In diesem Aufgabenblock des Programmierpraktikums sollen die Inhalte der Vorlesungen *Objektorientierte Programmierung* und *Algorithmen und Datenstrukturen* wiederholt und vertieft werden. Dazu werden in Aufgabe 1.1 anhand der Implementierung von Listen bekannte Konzepte wie Klassen, Interfaces, Zugriffsstrukturen und die Java Streams API thematisiert. Anschließend werden in Aufgabe 1.2 anhand der Implementierung einer Gitterstruktur u.a. Iteratoren und Unit Tests wiederholt. Schließlich wird in Aufgabe 1.3 das sogenannte *Decorator Pattern* durch Implementierung von Bloom Filtern behandelt.

Aufgabe 1.1: Doppelt-verkettete Liste

Eine verkettete Liste ist eine Datenstruktur, welche Elemente in einer Ordnung verwaltet, die nicht zwangsläufig der physischen Position der Elemente im Speicher entspricht. Die Reihenfolge der Elemente wird anhand von Zeigern, welche zu jedem Element gespeichert werden, festgelegt. Eine einfach-verkettete Liste besitzt für jedes Element einen Zeiger, der auf das nächste Element in der Liste verweist. Traditionell besitzt die Listendatenstruktur einen Verweis auf das erste Element der Liste (`head`), von welchem aus durch das Verfolgen der Zeiger jedes andere Element in linearer Zeit erreicht werden kann.

In dieser Aufgabe sollen Sie eine doppelt-verkettete Liste umsetzen. Diese besitzt nicht nur einen Zeiger auf das nächste Element der Liste, sondern einen weiteren Zeiger, welcher auf das vorherige Element verweist. Zusätzlich wird in der Datenstruktur ein Verweis auf das letzte Element der Liste (`tail`) vorgehalten. In Abbildung 1 ist der Aufbau einer doppelt-verketteten

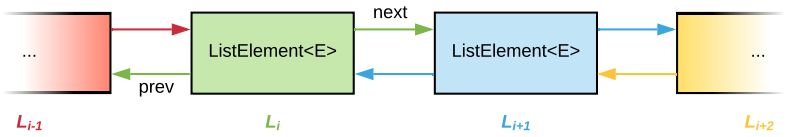

Abbildung 1: Doppelt-verkettete Liste

Liste dargestellt. Ein beliebiges Listenelement L_i verwaltet Daten und zusätzlich zwei Zeiger: prev und next. Der Zeiger prev verweist auf das vorherige Element der Liste (L_{i-1}) und der Zeiger next auf das nächste Element (L_{i+1}).

a) Erstellen Sie eine Klasse `DoubleLinkedList<E>`, welche das Interface `SimpleList<E>` implementiert und eine doppelt-verkettete Liste realisiert. Verwenden Sie dazu eine innere `ListElement<E>` Klasse, welche Daten und Zeiger verwaltet.

b) Legen Sie eine Klasse `PerformanceTestForList` an, in welcher Sie zwei `ArrayList` und zwei `DoubleLinkedList` Instanzen erstellen, um Daten vom Typ `Integer` zu speichern. Fügen Sie

jeweils 100.000 Elemente in die Datenstrukturen wie folgt ein:

- In die erste `ArrayList` durch anhängen an das Ende der Liste.
- In die zweite `ArrayList` durch anhängen an den Anfang der Liste.
- In die erste `DoubleLinkedList` durch den `addFirst` Befehl.
- In die zweite `DoubleLinkedList` durch den `addLast` Befehl.

Führen Sie dann auf die von Ihnen befüllten Datenstrukturen jeweils 100.000 Mal die `get(int n)` Methode aus. Verwenden Sie für jeden Aufruf von get eine neue zufällige (gültige) Zahl, indem Sie die Methode `nextInt(int bound)` der `Random` Klasse verwenden.

Messen Sie jeweils die Dauer zum Einfügen aller 100.000 Elemente und zum Lesen von 100.000 zufälligen Elementen. Geben Sie die Ergebnisse auf der Konsole aus. Erklären Sie ihrem Tutor, wie es zu den Ergebnissen kommt und welche Komponenten Ihrer Implementierung eine Rolle dabei spielen.

Nutzen Sie abschließend den `Iterator` Ihrer Listenimplementierung, um mit Hilfe der Stream-API aus den ersten 100 Elemente einer `DoubleLinkedList` Instanz diejenigen auszugeben, welche durch 10 teilbar sind.

Aufgabe 1.2: Grid

Ein Grid (oder auch deutsch *Gitter*) ist eine Datenstruktur mit n Zeilen und m Spalten, in welcher $n * m$ Elemente gespeichert werden können. Ein Zugriff auf ein Element im Grid erfolgt über die Angabe der Zeilnummer und Spaltennummer. So wird beispielsweise in Abbildung 2 bei

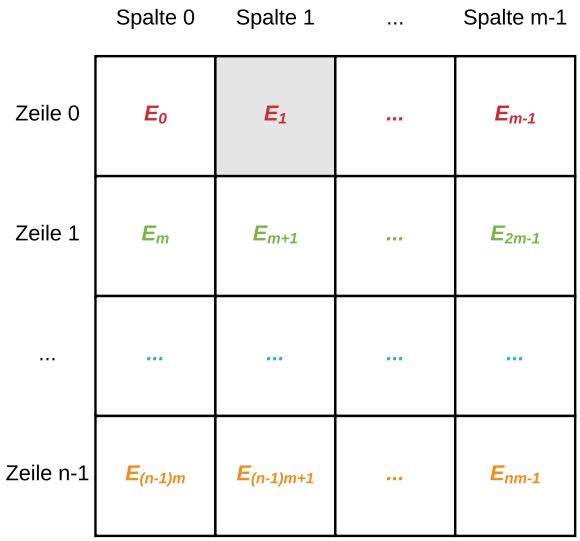

Abbildung 2: Grid

einem Zugriff auf (Zeile 0, Spalte 1) das Element E_1 zurückgegeben. Zur Umsetzung eines 2-dimensionalen Gitters gibt es unterschiedliche Strategien. Beispielsweise kann man die zwei-dimensionale Struktur direkt als Array von Arrays abbilden. Alternativ rechnet man die Koordinaten um, sodass diese auf eine 1-dimensionale Datenstruktur abbilden. Die Umrechnung können Sie aus den Indizes der Elemente E in Abbildung 2 entnehmen. So würde das Element

in (Zeile $n-1$, Spalte 1) über die Position $(n-1)*m+1$ in einer 1-dimensionalen Datenstruktur erreicht werden.

a) Erstellen Sie eine Klasse `GridImpl<E>`, welche das gegegebene Interface `Grid` implementiert. Bei Anfragen auf nicht belegten Feldern soll `null` zurückgegeben werden. Der `rowIterator` soll das gesamte Grid zeilenweise ausgeben. In Abbildung 2 wäre dies die Reihenfolge E_0, E_1, E_2, ..., E_{m-1}, E_m, E_{m+1}, ..., E_{nm-1}. Analog soll `columnIterator` das gesamte Grid spaltenweise ausgeben. In Abbildung 2 wäre dies die Reihenfolge E_0, E_m, ..., $E_{(n-1)m}$, E_1, E_{m+1}, ..., E_{nm-1}. Zur Realisierung der `iterator` Methode dürfen Sie entweder `rowIterator` oder `columnIterator` wiederverwenden.

b) Erstellen Sie JUnit Tests, welche Ihre Implementierung anhand eines `Grid<String>` Objektes testen. Ihre Tests müssen mindestens die folgenden Fälle abdecken:

- Arbeiten mit einem nicht-symmetrischen Grid (z.b. $3*4$ Elemente)
- `insert`, `get` und `remove` von gültigen Positionen mit und ohne null Elementen
- `insert`, `get` und `remove` von ungültigen Positionen (z.b. Zeile 5, Spalte 5 bei einem $3*4$ Grid)
- Korrekte Ausgabe von `rowIterator` und `columnIterator`

Aufgabe 1.3: Bloom Filter

Ein Bloom Filter ist eine probabilistische Datenstruktur, mit welcher effizient untersucht werden kann, ob ein Element ***nicht*** in einer Datenstruktur liegt. Der Bloom Filter besteht aus einem Bit-Array fester Größe und k Hashfunktionen, welche auf die Positionen des Bit-Arrays abbilden.

Wird ein Element in den Bloom Filter eingefügt, wird jede der k Hash-Funktionen auf das Element angewandt. Die Ergebnisse dieser Operation sind k Positionen im Bit-Array. Die Bits dieser Positionen werden auf 1 gesetzt. In Abbildung 3 wird beispielsweise das Element e in

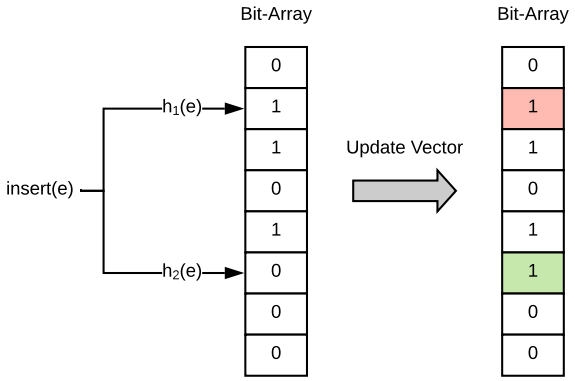

Abbildung 3: BloomFilter

ein BloomFilter mit Arraygröße 8 und 2 Hashfunktionen eingefügt. Die Funktionen geben die Positionen 1 und 5 zurück. Entsprechend müssen die Bits auf diesen Positionen auf 1 gesetzt werden. Da das Bit auf Position 1 bereits gesetzt ist, bleibt es unverändert.

Wird ein Element im Bloom Filter gesucht, werden die k Hashfunktionen auf dieses Element

angewandt und die entsprechenden Positionen im Bit-Array untersucht. Ist mindestens ein Bit 0, so kann mit Sicherheit gesagt werden, dass das Element nicht in den Bloom Filter eingefügt wurde. Sind alle Bits 1, kann aufgrund von möglichen Kollisionen, nicht gesagt werden, ob das Element tatsächlich im Bloom Filter liegt (vgl. Abbildung 3, Position 1). Durch diese Eigenschaft, und die Tatsache, dass der Bloom Filter die eigentlichen Elemente nicht abspeichert, wird der Bloom Filter in Kombination mit anderen Datenstrukturen verwendet, auf welche man in diesen Fällen zurückgreift.

a) Vervollständigen Sie die Klasse `BloomFilter<E>`, welche das Interface `IBloomFilter<E>` implementiert. Übergeben Sie im Konstruktor die Anzahl an Bits sowie eine `Collection` mit den k Hash-Funktionen. Nutzen Sie für das Bit-Array eine Instanz der Klasse `BitSet`. Beachten Sie, dass `BitSet.Size()` die gesamte Anzahl an Bits zurück liefert, die für die gegebene Instanz verwendet werden und nicht die von Ihnen angegebene Anzahl an Bits [1].

b) Erstellen Sie eine Klasse `BloomFilterCollection<E>`, welche das Interface `Collection<E>` aus der package `java.util` implementiert. Übergeben Sie im Konstruktor eine Instanz `IBloomFilter<E>` und eine Instanz von `Collection<E>`. Implementieren Sie die Methoden `add`, `addAll`, `contains` und `containsAll` so, dass der `IBloomFilter<E>` verwendet wird und nur bei Bedarf auf die übergebene `Collection<E>` zugegriffen wird. Adaptieren Sie die Methoden `remove`, `removeAll` und `retainsAll` so, dass der `BloomFilter` neu aufgebaut wird. Delegieren Sie die restlichen Methoden direkt an die übergebenene `Collection<E>`. Erklären Sie Ihrem Tutor das Zusammenspiel zwischen `Collection<E>` und `IBloomFilter<E>` in den jeweiligen Fällen.

c) Vervollständigen Sie die Klasse `ColumnImprint`, welche einen `BloomFilter<Double>` erweitert. Ein Column Imprint teilt die abzubildene Domäne in n Buckets mit gleicher Größe `bucketSize` auf (siehe Abbildung 4). Alle Punkte eines Buckets werden nun auf dasselbe Bit des Bit-Arrays abgebildet. Anders als beim Bloom Filter, müssen hier vorab das Minimum min und das Maximum max der Domäne bekannt sein. Werte die außerhalb dieses Intervalls liegen können nicht eingefügt werden.

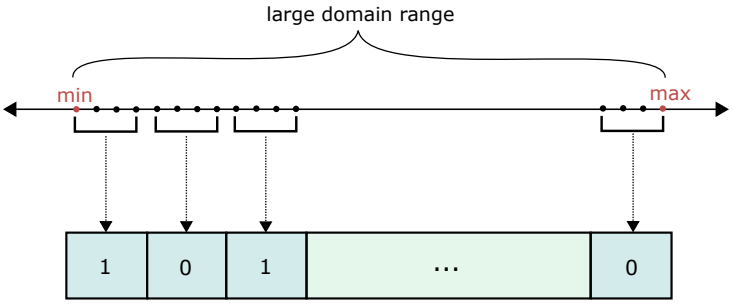

Abbildung 4: Column Imprint mit `bucketSize=4`

[1] `https://docs.oracle.com/javase/7/docs/api/java/util/BitSet.html#size()`

Weiterhin wird eine einzige Hash-Funktion verwendet, welche wie folgt definiert ist:

$$\texttt{hash(e)} = \begin{cases} 0, & \texttt{e} \in [\texttt{min, min + bucketSize}) \\ \texttt{n} - 1, & \texttt{e} \in [\texttt{max - bucketSize, max}) \\ \left\lfloor \frac{\texttt{e - min}}{\texttt{bucketSize}} \right\rfloor, & \texttt{e} \in (\texttt{min + bucketSize, max - bucketSize}) \end{cases}$$

d) Um beliebige Werte einfügen zu können, verallgemeinern Sie die Hash-Funktion in `ColumnImprint`, so dass alle Werte in $(-\infty$, `min`$]$ auf das erste Bit und alle Werte in `[max - bucketSize,` $\infty)$ auf das n-te Bit abgebildet werden. Erstellen Sie die Klasse `BloomFilterCollectionTest`, in welcher Sie zwei Varianten einer `BloomFilterCollection` testen:

- Erstellen Sie eine `BloomFilterCollection` mit Angabe eines `BloomFilter` und eine weitere mit einem `ColumnImprint` mit gleicher Anzahl an Bits/Buckets.
- Fügen Sie mindestens 200000 Datenwerte in die Collections ein und fragen sie diese ab. Variieren Sie die Parameter (`min` und `max`) für den `ColumnImprint`.
- Erklären Sie Ihrem Tutor, unter welchen Umständen sich eher ein `BloomFilter` lohnt als ein `ColumnImprint` und umgekehrt.

Hinweis: Beachten Sie, dass die Implementierungen von `HashFunction` **Werte im Bereich** `[Integer.MIN_VALUE, Integer.MAX_VALUE]` **liefern können und diese von Ihnen geeignet auf den jeweiligen Addressbereich der Hashtabelle mit Hilfe der** `hashMod` **Methode abgebildet werden müssen.**