SKIZZE

SKIZZE ZUR AUFGABE 4 AUS DER VORLESUNGSREIHE “ALOGRITHMEN UND DATENSTRUKTUREN”

TEAM 10: ANTON, MESUT UND IGOR

Aufgabenaufteilung

Alle Aufgaben wurden gemeinsam entworfen und bearbeitet.

Quellenangaben

Lehrveranstaltung und Skript

Bearbeitungszeitraum

Datum	Dauer in Stunden
20.12.15	5
22.12.15	4
Gesamt	**9**

Aktueller Stand

Die Skizze ist fertig.

Änderungen in der Skizze

-

Skizze:

Es sollen folgende Klassen in Java implementiert werden: die ADT Hashmap, TextParser und Benchmark (Package: „ADT“, Java Klassen: „ADTHashmap“, „TextParser“ und „Benchmark“). Dazu müssen auch JUnit Tests erstellt werden. (Package: „ADT/adthasmaptests“, extern als „adthasmapJUt.jar“ verfügbar). Die verwendete Datenstruktur ist ein Array. Die Dateien müssen genau diese Namen besitzen, damit sie mit den anderen Gruppen austauschbar bleiben. Es folgen nun die Beschreibungen und Signaturen der Methoden.

ADT Hashmap

Bei der Hashmap beziehungsweise dem Hashverfahren werden die Datensätze einfach in einem Feld *(mit Indexwerten von 0 bis N-1)* mit direktem Zugriff gespeichert und eine spezielle Funktion, die sogenannte Hashfunktion, ermöglicht für jeden gespeicherten Wert den direkten Zugriff auf den Datensatz. Die Hashfunktion *(h)* bestimmt somit für ein Element *(e)* die Position *h(e)* im Feld und dient für eine ausgefeilte Adressberechnung.

Funktionale Vorgabe

- Die ADT Hashmap kann nur mittels dem Parameter „*strategy*“ und „*filename*“ gestartet werden. Der Parameter „*strategy*“ kann dementsprechend L *(lineares Sondieren)*, Q *(quadratisches Sondieren)* und B *(Double-Hashing)* annehmen.
- Der dazugehörige Parameter „*filename*“ nimmt nur Textdateien im ASCII-Format an.
- Es wird eine Logdatei erstellt, worin die Anzahl der Vorkommen aller Wörter von der übergegebenen Textdatei gezählt werden.

Technische Vorgabe

- Bei der Hashmap bilden die Wörter den sogenannten „Hashkey“.
- Die Hashmap kann Wörter und gegebenenfalls deren Anzahl speichern.

Public Operationen (semantische Signatur/syntaktische Signatur)

create: size x strategy → hashmap

(Input: Integer size, Strategy strategy | Output: ADTHashmap hashmap)

Diese Methode erzeugt eine neue Hashmap mit der angegebenen Kapazität *(die gewünschte Größe der Hashmap)* und der gewünschten Strategie *(L, Q oder B)*. Die Strategie beeinflusst im Nachhinein die Funktionalität des Hashverfahrens.

size: ∅ → size

(Input: ∅ | Output: int size)

Die Methode size() liefert uns die aktuelle Anzahl der gespeicherten Schlüssel in einer Hashmap als ganze positive Zahl (inkl. 0) zurück.

isEmpty: ∅ → isEmpty

(Input: ∅ | Output: Boolean isEmpty)

Die Methode isEmpty() prüft im Allgemeinen, ob die gegebene Hashmap mit Elementen belegt ist oder nicht. Im Fall einer leeren Hashmap liefert die Methode den Wert TRUE, sonst FALSE.

contains: key → contains

(Input: String key | Output: Boolean contains)

Mit der Methode contains(key) können wir prüfen, ob eine Hashmap den angegebenen Hashkey enthält. Die Methode gibt TRUE zurück, falls ein Key enthalten ist, ansonsten FALSE. Ein leerer Key *(also „Null“)* wird nicht akzeptiert.

insert: hashmap x word → hashmap

(Input: ADTHashmap hashmap, String word | Output: ADTHashmap hashmap)

Eine Hashmap ist eine Zuordnung von einem Key zu einem Wert. Da wir in unserem Fall mit Strings arbeiten, binden wir unseren Key zu einem Wort. Für das Einfügen von Strings benutzen wir die Divisions-Rest-Methode, welche uns Zeichenketten in Integer umzuwandeln erlaubt.

Wichtig zu beachten ist die Kollision, welche entstehen kann, wenn wir zwei Wörter an der gleichen Feldadresse (Key) abbilden. In der Regel gibt es zwei Möglichkeiten Kollisionen zu verhindern: Geschlossenes Hashing (Verkettung) und offene Adressierung. Wir arbeiten mit der zweiten Methode, sodass wir auf die Verkettung nicht eingehen möchten.

Die offene Adressierung bietet die Möglichkeit, Überläufer an weiteren freien Feldplätzen abspeichern zu können. Anhand eines Beispiels möchten wir nun verdeutlichen, wie es zu einer Kollision führen kann.

Kollisionsbeispiel für insert

- Die 42 einfügen → **h(42) = 42 mod 10 = 2**
- Die 119 einfügen → **h(119) = 119 mod 10 = 9**
- Die 69 einfügen, → **h(69) = 69 mod 10 = 9 → Kollision**

Position	Eintrag
0	
1	
2	42
3	
4	
5	
6	
7	
8	
9	119

Es ist eine Kollision entstanden, welche es zu beseitigen gilt. Offene Adressierung bietet drei Varianten/Strategien an, um Kollisionen zu beheben. Bei der Methode `create()` entscheiden wir uns für eine und somit auch für Funktionalität unserer Hashmap. Dabei müssen wir den Zeiger des Feldes immer um ein von der Strategie abhängiges Inkrement erhöhen, somit hätte die 69 mit Hilfe des linearen Sondierens an der Position 0 ihren Platz gefunden.

- **Lineares Sondieren**: Bei der Suche nach einem freien Platz wird um ein konstantes Intervall verschoben. In der Regel beträgt die Größe des Intervalls 1.
- **Quadratisches Sondieren**: Falls eine Suche nach einem Platz erfolglos war, wird der im vorherigen Schritt verwendete Wert des Intervalls quadriert.
- **Doppeltes Hashing**: Eine weitere Hash-Funktion h‘ ist für die Berechnung des Intervalls zuständig.
- **Brents-Verfahren:** Im Unterschied zum Doppelten Hashing wählt der Brent-Algorithmus aus, ob der neue Eintrag (neu) oder der schon in der Tabelle vorhandene, kollidierende Eintrag (alt) verschoben wird. Falls es bei einem neuen Eintrag eines Wortes zu einer Kollision führen, gibt es drei Fälle, die von dem Algorithmus beachtet werden:

Fall 1: h‘(neu) ist frei, das Wort wird auf h‘(neu) gespeichert.
Fall 2: h‘(neu) ist belegt und h‘(alt) ist frei, dann wird das alte Wort auf h‘(alt) verschoben, und das neue Wort bekommt den Platz von dem Platz des alten Wortes.
Fall 3: h‘(neu) ist belegt und h‘(alt) ist belegt. Es erfolgt ein rekursiver Aufruf der Funktion und dabei müssen diese drei Fälle nochmals unterschieden werden.

find: hashmap x word → count

(Input: ADTHashmap hashmap, String word | Output: Integer count)

Beim Suchen ist es das Ziel, Werte anhand ihrer Keys wiederzufinden. Dabei wird der Schlüssel des Wortes berechnet (Horner-Schema) und somit die Position ermittelt. Sollte das gesuchte Wort mit an der berechneten Feldadresse gespeichertem Wort nicht übereinstimmen (Kollision), so haben wir die Möglichkeit mit dem Sondieren (siehe Insert) das gewünschte Wort wiederzufinden.
Je nach der ausgesuchten Strategie bei `create()` verwenden wir die in `insert()` beschriebenen Algorithmen und liefern nach einer erfolgreichen Suche die Anzahl, wie viele Mal das gesuchte Wort in einem Text vorkommt, zurück.

export: hashmap → file

(Input: ADTHashmap hashmap | Output: File words)

Die Methode gibt die gespeicherten Werten in der Hashtabelle in einer „.csv“ log-Datei aus. Das entspricht der Anzahl der Vorkommen aller Wörter in der Textdatei.

TextParser

Die Klasse „*TextParser*“ dient dazu, um aus einer angegebenen Datei die Wörter richtig zu zerteilen.

Funktionale Vorgabe

- Die angegebene Datei muss existieren.

Technische Vorgabe

- Der TextParser nutzt Trennzeichen **([. , ? !]+)**, damit man beim Auslesen zwischen den Wörtern unterscheiden kann.
- Die getrennten Wörter werden in einem HashSet gespeichert *(doppeltvorkommen direkt ausgeschlossen)*

Public Operationen (semantische Signatur/syntaktische Signatur):

readSourceFile: file → parsedWords

(Input: File file | Output: HashSet parsedWords)

Die Methode liest aus einer gegebenen Datei die Inhalte und zerteilt diese dann nach Wörter und speichert dies in einem HashSet *(die gängige HashSet-Klasse speichert vorkommen nur einmal)*. Nur bei einem Spezialfall *(IOException)* wird eine Fehlermeldung rausgeschmissen.

Benchmark

Benchmark ist unsere separate Klasse, die mit Hilfe von `insert()` aus einer gegebenen Textdatei die Wörter rausliest, diese in eine Hashmap importiert und dann Schritt für Schritt ausführt. Die Laufzeit der `insert()` und `find()`-Methode und die Anzahl der lesenden und schreibenden Zugriffe werden separat und automatisiert gemessen. Dabei werden verschiedene importierte Textdateien in drei Anordnungen gemessen: Lineares und quadratisches Sondieren und zuletzt Double-Hashing nach Brent.

Funktionale Vorgabe

- Erstellung einer Datei mit dem Suffix „.csv“.

Technische Vorgabe

- Besteht die Datei schon, wird sie folglich überschrieben.

Public Operationen (semantische Signatur/syntaktische Signatur):

main: ∅ → file

(Input: ∅ | Output: File benchmark)

Testen mit JUnit

- Alle obengenannten Methoden aufrufen.
- Es sollen folgende Situationen betrachtet werden: Die Kollisionsbehandlung soll bis zu seiner Grenze voll getestet werden (obiges Beispiel betrachten)
- Die fehlerhaften Situationen sollen berücksichtigt werden: z.B. der übergebene Wert bei `insert()` darf nicht NULL sein usw.