

**Team:** 03, Sebastian Diedrich – Murat Korkmaz

**Aufgabenaufteilung:**

1. Aufgaben, für die Teammitglied 1 verantwortlich ist:

- (1) Skizze

Dateien, die komplett/zum Teil von Teammitglied 1 implementiert/bearbeitet wurden:

- (1) ADTHashmap: alle Operationen

- (2) Programm: kompletter Ablauf

- (3) Test

2. Aufgaben, für die Teammitglied 2 verantwortlich ist:

- (1) Skizze

Dateien, die komplett/zum Teil von Teammitglied 2 implementiert/bearbeitet wurden:

- (1) ADTHashmap: alle Operationen

- (2) Programm: kompletter Ablauf

- (3) Test

**Quellenangaben:** Vorlesung am 16.12.15, Folien AD\_Hash.pdf (Prof. Klauck)

**Bearbeitungszeitraum:** 28.12.15 (6h), 06.01.16 (7h), 13.01.15 (6h), 14.01.16 (3h), 15.01.16 (2h)

**Aktueller Stand:**

1. Skizze Version 2
2. Alle Funktionen implementiert und getestet. Programmablauf erfolgreich getestet.
3. Messungen durchgeführt (**mit korrigiertem Code**) und interpretiert

**Messung (neu) vom 15.01.16:** (ab Seite 2)

**Messung (alt) vom 14.01.16:** (ab Seite 3)

**Skizze:** (ab Seite 5)

## Messung (neu):

Insert: (Strategy -> L, Q, B)

Strategy /Anzahl Wörter	500	1000	2000	4000	8000	16000	32000	64000
L -> in ms	0,92	0,41	0,57	1,02	2,17	3,15	5,22	7,5
Q -> in ms	1,08	0,45	0,6	0,93	1,59	3,04	5,24	27,96
B -> in ms	1,29	0,58	0,78	0,89	1,58	3,15	5,23	7,62

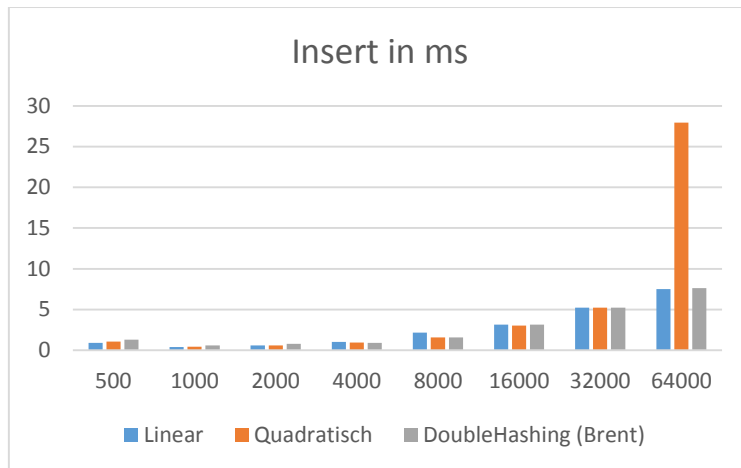


Abbildung 01

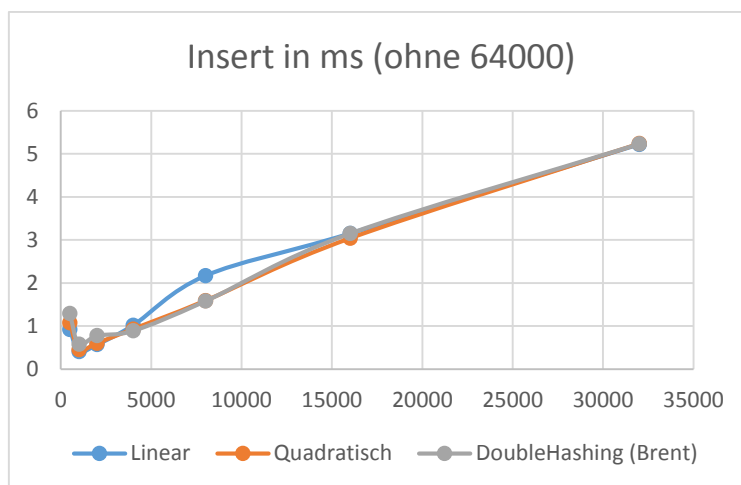


Abbildung 02

Verbesserung (Faktor)	500	1000	2000	4000	8000	16000	32000	64000
L -> in ms	9,18	47,10	143,00	305,05	584,87	1646,81	4044,75	8117,49
Q -> in ms	9,69	42,78	136,22	336,74	800,77	1740,33	4028,70	2180,27
B -> in ms	9,47	34,09	99,49	353,08	801,80	1651,44	4037,00	8035,07

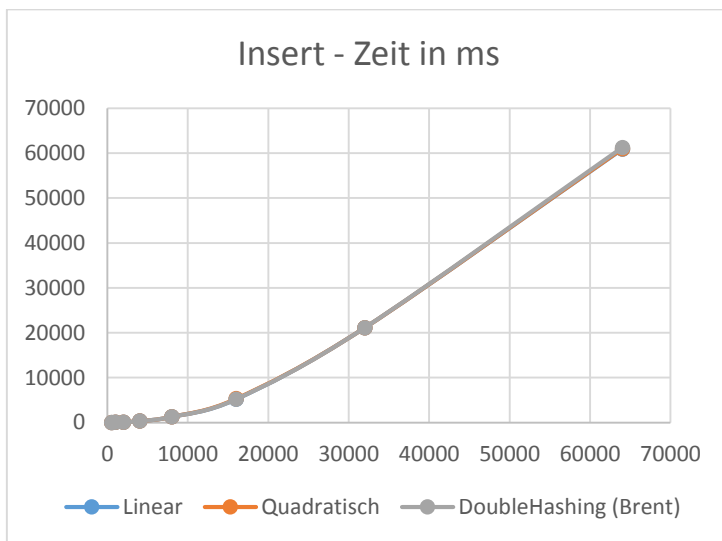
## Interpretation:

Durch die Herausnahme der Abfrage (find==0) konnte eine dramatische Verbesserung der Geschwindigkeit erreicht werden. In der Abbildung 02 zeigt sich eine nahezu lineare Steigerung (zur Veranschaulichung wurde der „Ausreißer-Wert“ von Strategy.Q mit 64000 herausgenommen). Möglich war dieses, da zuvor in der Implementierung nach Brent, eine Abfrage (ist der gefundene Schlüssel, mein Eigener) zu spät erfolgte und somit in der if-Bedingung ein find==0 erfolgen musste.

## Messungen (alt):

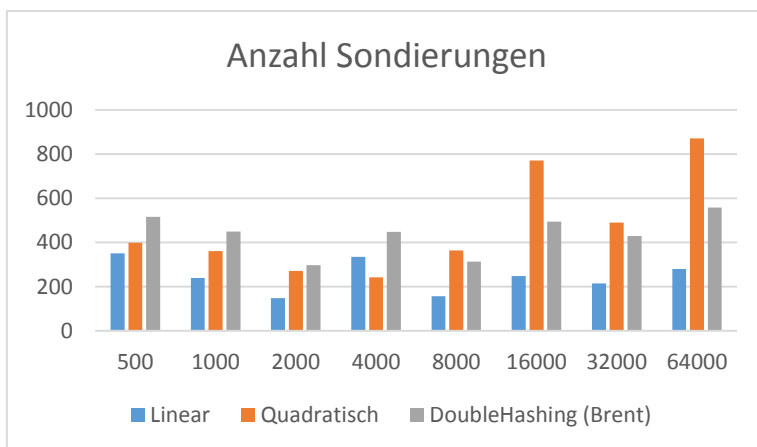
Insert: (Strategy -> L, Q, B)

Strategy /Anzahl Wörter	500	1000	2000	4000	8000	16000	32000	64000
L -> in ms	8,45	19,31	81,51	311,15	1269,17	5187,45	21113,62	60881,18
Q -> in ms	10,47	19,25	81,73	313,17	1273,23	5290,61	21110,41	60960,22
B -> in ms	12,22	19,77	77,6	314,24	1266,85	5202,04	21113,53	61227,26



Anzahl der Sondierungen in Abhängigkeit der Strategy:

Strategy /Sondierungen	500	1000	2000	4000	8000	16000	32000	64000
L	350	239	148	334	156	247	214	279
Q	398	361	271	241	364	771	490	870
B	515	449	296	447	312	494	428	558

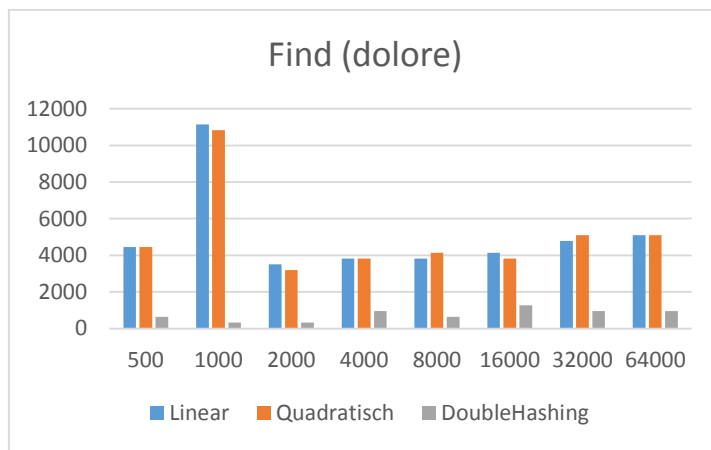


Interpretation:

Trotz starker Zunahme der Anzahl der Wörter, steigt die Anzahl der Sondierungen nur wenig. Insgesamt hat die Auswahl der Sondierungsfunktion, keine Auswirkung auf die Einfügegeschwindigkeit. Zu beachten dabei ist, dass in jedem Text maximal 500 verschiedene Wörter in unterschiedlicher Anzahl vorkommen.

*Find: (Strategy -> L, Q, B)*

Strategy /Anzahl Wörter	500	1000	2000	4000	8000	16000	32000	64000
L -> in ns	4460	11150	3504	3823	3823	4142	4779	5097
Q -> in ns	4460	10832	3186	3823	4141	3823	5097	5098
B -> in ns	637	318	319	956	637	1274	955	956



### **Interpretation:**

Es zeigt sich, dass das Double Hashing zu einem deutlich schnelleren Such-Ergebnis führt. Die Sondierungsfunktionen linear und quadratisch sind nahezu identisch in ihrer Such-Geschwindigkeit.

## Skizze Aufgabe 4:

### Aufgabe: 4.1

Ziel: ADT-HashMap implementieren

Quelle: [http://pub.informatik.haw-hamburg.de/home/pub/prof/klauck\\_christoph/AD/AD\\_Hash.pdf](http://pub.informatik.haw-hamburg.de/home/pub/prof/klauck_christoph/AD/AD_Hash.pdf)

Folgendes soll immer gelten (Invarianten):

- I) funktional
  - Das Programm wird mit folgenden Parametern gestartet:  
*strategy*: enum (L -> lineares Sondieren, Q -> quadratisches Sondieren und B -> Double-Hashing nach Brent)  
*filename*: absoluter Pfad zur Textdatei (inklusive Filename)
  - Als Dateien werden Text-Dateien im ASCII Format akzeptiert
  - Das Programm soll die Anzahl der Vorkommen aller Wörter in der Textdatei zählen und in einer log-Datei ausgeben.
- II) technisch
  - In der HashMap werden die Wörter gespeichert und ihre Anzahl
  - Die einzelnen Wörter bilden die Hashkeys, im Value steht die Anzahl

#### Objektmengen:

- word: Wort als Key in der HashMap  
(Sollten Umlaute und Sonderzeichen im Wort vorkommen, wird das Wort ignoriert)
- count: Anzahl der Vorkommnisse eines Wortes
- size: Anzahl der maximalen Speicheradressen in der HashMap  
(Bedingung: natürliche Zahl ohne 0)
- hashmap: ADT-HashMap (Key: word, Value: count)
- strategy: Auswahl des Sondierungsverfahrens

Folgende Operationen sollen bereitgestellt werden (semantische Signatur):

- *create*: eine leere HashMap (mit size und strategy) erstellen  
(size x strategy -> hashmap)  
Fehlerbehandlung: Exception wird geworfen und es wird keine HashMap erzeugt
- *insert*: Fügt ein Wort in die HashMap ein oder erhöht die Anzahl, falls Wort bereits vorhanden um 1  
(hashmap x word -> hashmap)  
Fehlerbehandlung: ignorieren (Umlaute und Sonderzeichen im Wort vorhanden)
- *find*: Gibt das Value (Anzahl) eines vorgegebenen Keys (Wort) zurück  
(hashmap x word -> count)  
Fehlerbehandlung: 0 zurückgeben

Syntaktische Vorgaben:

Dateiname: adtHashMap.jar

Klassenname: ADTHashmap

Anwendung der oben genannten Operationen:

*create*: ADTHashmap.create(size, strategy)

*insert*: <Objektname>.insert(word)

*find*: <Objektname>.find(word)

## Kollisionsbehandlung in kurzer Zusammenfassung:

### Verkettung (geschlossenes Hashing):

Trifft ein Schlüssel auf eine bereits belegte Adresse, wird dieser Schlüssel ebenfalls an diesem Speicheradresse abgelegt. Um dieses zu ermöglichen wird eine **Liste** an diesen Speicheradresse angelegt, in der sich unsortiert alle Schlüssel befinden. Wird nach einem Schlüssel gesucht, muss über die Liste iteriert werden.

### Offene Adressierung:

Jeder Schlüssel erhält einen **eigene** Speicheradresse. Sollte beim Einfügen ein Schlüssel auf eine bereits belegte Speicheradresse treffen, wird mittels Sondierungsfunktionen (s.u.) nach einer freien Speicheradresse gesucht.

## Hashfunktion:

### Beschreibung:

Mittels Hashfunktion wird aus einem Schlüssel eine Hashadresse berechnet. Die Hashadresse gibt den Index in einem Feld an, auch Hashtabelle genannt.

### Methode:

Es soll die Division-Rest-Methode als Hashfunktion für Java-Strings verwendet werden. Um Integer Überläufe zu vermeiden (max.  $2^{31}-1$ ) soll das Horner-Schema eingesetzt werden.

### Pseudocode:

i natürliche Zahl,  $h=0$ ; s Zeichenkette / Feld

1. for  $i = 0$  to  $i < \text{länge\_von}(s)$
2.  $h = (h * 128 + s[i]) \bmod m$ ;

Ergebnis: h

Quelle: [http://pub.informatik.haw-hamburg.de/home/pub/prof/klauck\\_christoph/AD/AD\\_Hash.pdf](http://pub.informatik.haw-hamburg.de/home/pub/prof/klauck_christoph/AD/AD_Hash.pdf)

### Beispiel:

s = hallo; m = 101

s[0] : h = (0 * 128 + 104) mod 101 = 3	// h
s[1] : h = (3 * 128 + 97) mod 101 = 77	// a
s[2] : h = (77 * 128 + 108) mod 101 = 66	// l
s[3] : h = (66 * 128 + 108) mod 101 = 72	// l
s[4] : h = (72 * 128 + 111) mod 101 = 35	// o

Ergebnis: Speicheradresse des Java-Strings „hallo“ ist 35.

### **Beschreibung der Sondierungsverfahren und deren Durchführung:**

Es soll bei der Implementation der Operationen (find und insert) der ADT-Hashmap die offene Adressierung verwendet werden.

#### Definitionen:

s : Sondierungsfunktion  
j : Anzahl der Versuche, eine freie Speicheradresse zu finden  
k : Schlüssel  
h : Hashfunktion  
h' : zweite Hashfunktion

#### Linear:

$$h_j(k) = (h(k) - s(j,k)) \bmod m \text{ mit } s(j,k) = j$$

#### Beispiel:

Sollte das Wort „hallo“ erneut als Schlüssel eingetragen werden, erfolgt dieses um eine Speicheradresse versetzt nach rechts.

$$h_0(k) = (35 + 0) \bmod 101 = 35$$

$$h_1(k) = (35 + 1) \bmod 101 = 36$$

#### Quadratisch:

$$h_j(k) = (h(k) - s(j,k)) \bmod m \text{ mit } s(j,k) = (-1)^j * (j/2)^2$$

#### Beispiel:

Sollte das Wort „hallo“ mehrmals als Schlüssel eingetragen werden, wird mittels der obigen quadratischen Sondierungsfunktion versucht eine freie Speicheradresse zu finden.

$$h_0(k) = (35 + 0) \bmod 101 = 35$$

$$h_1(k) = (35 + (-1)) \bmod 101 = 34$$

$$h_0(k) = (35 + 0) \bmod 101 = 35$$

$$h_1(k) = (35 + (-1)) \bmod 101 = 34$$

$$h_2(k) = (35 + 4) \bmod 101 = 39$$

#### Double-hashing mittels Brent-Verfahren (s.u.):

$$h_j(k) = (h(k) + s(j,k)) \bmod m \text{ mit } s(j,k) = j * h'(k)$$

$$h'(k) = 1 + (k \bmod m') \quad // \text{ mittels Hornerschema neu berechnen}$$

$$m' = m - 2$$

#### Beispiel:

Sollte das Wort „hallo“ mehrmals als Schlüssel eingetragen werden, wird mittels der obigen Sondierungsfunktion versucht eine freie Speicheradresse zu finden.



$h_0(k) = (35 + 0) \bmod 101 = 35$  // belegt

$h_1(k) = (35 + 36) \bmod 101 = 71$  // frei

$h_0(k) = (35 + 0) \bmod 101 = 35$  // belegt

$h_1(k) = (35 + 36) \bmod 101 = 71$  // belegt

$h_2(k) = (35 + 72) \bmod 101 = 6$  // frei

### Verbesserung der erfolgreichen Speicheradressen-Suche (Brent-Verfahren)

Um beim Einfügen von Schlüsseln schneller eine freie Speicheradresse zu finden, kann eine Breiten- oder Tiefensuche (Verfahren nach Brent) benutzt werden.

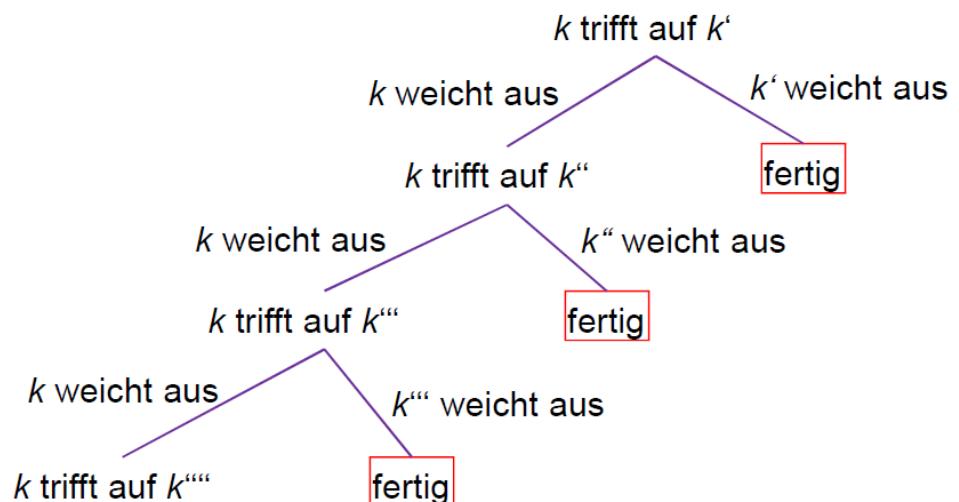
Bei der Implementation der insert Methode soll das Verfahren nach Brent angewendet werden. Dabei muss jeder Schlüssel „wissen“ wie viele bisherige Versuche (j) er gebraucht hat, eine freie Speicheradresse zu finden. Trifft ein neuer Schlüssel auf eine belegte Speicheradresse, versucht zunächst der alte Schlüssel auf eine freie Speicheradresse auszuweichen. Dafür wird die Sondierungsfunktion mit  $j = j+1$  aufgerufen.

Sollte dieses nicht zu einer freien Speicheradresse führen, weicht der neue Schlüssel mittels der Sondierungsfunktion mit  $j = j+1$  aus. Trifft der neue Schlüssel auf eine freie Speicheradresse, wird diese benutzt. Falls die Speicheradresse erneut belegt ist, läuft der oben beschriebene Ablauf von vorne ab.

M1 : Pfad vom neuen Schlüssel

### Brent's Verfahren: verfolge nur (M1)

Quelle: [http://pub.informatik.haw-hamburg.de/home/pub/prof/klauck\\_christoph/AD/AD\\_Hash.pdf](http://pub.informatik.haw-hamburg.de/home/pub/prof/klauck_christoph/AD/AD_Hash.pdf)



## Test

Für jede Sondierungsstrategie sollen folgende Tests durchgeführt werden

### 1. J-Unit Tests

- Diese sollen austauschbar sein
  - i. Name der Jar: hashmJUt.jar
- Input:
  - i. Text mit 100 Wörtern:  
<http://users.informatik.haw-hamburg.de/~klauck/AlguDat/texta.txt>
  - ii. Text mit 9790 Wörtern:  
<http://users.informatik.haw-hamburg.de/~klauck/AlguDat/textb.txt>
  - iii. Text mit 0 Wörtern:  
selbstgenerierter Text
  - iv. Text mit 100.000 Wörtern:  
selbstgenerierter Text
- Output:
  - i. ADT-HashMap mit Anzahl unterschiedlicher Wörter (Key) und deren Vorkommen im Text (Value)

## Messung von Laufzeit

1. Implementierung einer Version der ADT um die Laufzeit der Operationen von insert und find durchführen zu können

2. Vorgaben für die Messungen: Anzahl der Wörter:

Um die find-Operation vergleichbar zu machen, muss in jedem Text das Wort „dolore“ min. einmal vorkommen. Dieses Wort wird in der Messung gesucht und die Anzahl der Vorkommen zurückgegeben.

- 1) 500
- 2) 1.000
- 3) 2.000
- 4) 4.000
- 5) 8.000
- 6) 16.000
- 7) 32.000
- 8) 64.000
- 9) 128.000

3. Resultate:

<b>ADT-Hashmap</b>									
	500	1000	2000	4000	8000	16000	32000	64000	128000
Laufzeit insert (ms)									
Laufzeit find (ms)									

Aus den Ergebnissen werden Excel-Graphiken erzeugt, um die Steigung der einzelnen interpolierten Kurven vergleichen zu können.

Die daraus resultierenden Schlussfolgerungen werden zusammen mit den Graphiken in einem PDF dokumentiert.