Double Hashing

oder auch

MultiHashing

Hashing/ Allgemein

- "to hash„ (englisch): *zerhacken*
- sehr gebräuchliche Funktion => gut erforscht
- umfangreiche Quellmenge („Text“) wird auf eine wesentlich kleinere Zielmenge (Hash-Werte) abbildet.
- Anwendungen:
  - Auffinden von Daten in einer Datenbank
  - digitales Signieren eines Dokumentes
  - Einweg-Verschlüsselung (Kryptologie)
  - Prüfsummen (Datenintegrität bei Störeinflüssen sicherzustellen)

Hashing/ Pro- Contra

Vorteile:

- im Idealfall sehr schnell
- Dynamisch verwendbar (Schlüsselanzahl „unbekannt“)
- Guter Kompromiss zwischen Zeit und Platzbedarf
- Wenn es keine Speicherbeschränkung gäbe, könnte man den Schlüssel als Speicheradresse verwenden => jede beliebige Suche mit nur einem Zugriff auf den Speicher ausführbar

Nachteile:

- Kollisionen
- kann bei „Problemen“ sehr langsam sein!

Hashing/ Funktionsweiße

Algorithmus berechnet für jedes einzelne Zeichen (einer Zeichenkette) einen Hashwert

⇒Hashwert wird in einer Tabelle (Hashtabelle) abgelegt

⇒Falls an der stelle kein Platz ist

⇒Kollision

⇒ Kollisionsbehandlung

⇒ kann sehr zeitaufwendig sein

⇒ Berechnung und Suche SEHR langsam

Hashing/Aufbau

- k: Schlüsselwert (key)
- h(k): Hashfunktion
- w: Menge aller möglichen Schlüssel
- s: Menge der zu speichernden Schlüssel
- β: Belegungsfaktor
- m: Größe der Hash- Tabelle

Hashing/Aufbau (2)

die Menge der Hash-Adressen (Adressraum) $\{0, \cdots, m-1\}$

die Hash-Funktion $h : k \rightarrow \{0, \cdots, m-1\}$

und der Belegungsfaktor $\beta = \frac{|s|}{m}$

Der Fall $k \neq k' \rightarrow h(k) = h(k')$ führt zu einer Kollision (dazu später genaueres)

Hashing/ Kriterien

Damit die Hashfunktion gut funktioniert, bedarf es einiger Vorraussetzungen:

- Datenreduktion

  Der Speicherbedarf des Hash- Wertes soll deutlich kleiner sein als der der Nachricht.

- Zufälligkeit

  Ähnliche Quellelemente sollen zu völlig verschiedenen Hash- Werten führen. Im Idealfall verändert das Umkippen eines Bits in der Eingabe durchschnittlich die Hälfte aller Bits im resultierenden Hash- Wert.

Hashing/ Kriterien (2)

- Eindeutigkeit

  Die Funktion muss deterministisch von der Quellmenge auf die Zielmenge abbilden. Wiederholtes Berechnen des Hash- Wertes desselben Quellelements muss also dasselbe Ergebnis liefern.

- Effizienz

  Die Funktion muss schnell berechenbar sein, ohne großen Speicherverbrauch auskommen und sollte die Quelldaten möglichst nur einmal lesen müssen.

Hashing/ Beispiel

Zeichenkette: AKEY

Tabellengröße: 101 (muss Primzahl sein!)

5 Bit Code (i-te Buchstabe im Alphabet wird durch die Binärdarstellung der Zahl i dargestellt)

A K E Y

AKEY => 00001 01011 00101 11001

00001 01011 00101 11001 entspricht in Dezimaldarstellung 44217

44217 mod 101 = 80

Errechnete Hashadresse ist somit 80

Hashing/ Probleme

Probleme beim Hashing: Kollisionen

Grund: Plätze können in der Hashtabelle schon belegt sein, und ein neuer muss gefunden werden=> Kollision muss behandelt werden

$\Rightarrow$ wenn die Tabelle schon sehr voll ist, ist ein „freier Platz zu finden“ sehr Zeitaufwendig

$\Rightarrow$ Tabelle muss recht leer sein, damit der Algorithmus schnell ist

Hashing/ „Lösungen“

Möglichkeiten zum Lösen von Kollisionsproblemen:

- Offenes Hashing
- Geschlossenes Hashing
- Lineares Sondieren
- Quadratisches Sondieren
- **Doppel- Hashing**

Double Hashing

Dynamisierte Darstellung des algorithmischen Ablaufs

Einfügen von Elementen in eine Hash-Tabelle

Die Ergebnisse der folgenden Hash-Funktion sind exemplarisch und nur zur Verdeutlichung derart gewählt.

Double Hashing

S	A	M	P	L	E

Hash h() Wert:

Rehash h‘() Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Double Hashing

	A	M	P	L	E

S - - - - - - > Hash h() Wert:

Rehash h‘() Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Double Hashing

	A	M	P	L	E

Hash h(S) Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Double Hashing

	A	M	P	L	E

Hash h(S) - > **Wert: 0**

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Double Hashing

	A	M	P	L	E

Hash h(S) **Wert: 0**

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																

Double Hashing

	A	M	P	L	E

Hash h(S) Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																

Double Hashing

	A	M	P	L	E

Hash h()

Wert:

Rehash h‘()

Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																

Double Hashing

	A	M	P	L	E

A - - - - - > Hash h() Wert:

Rehash h‘() Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																

Double Hashing

		M	P	L	E

Hash h(A) Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																

Double Hashing

		M	P	L	E

Hash h(A) - > **Wert: 1**

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																

Double Hashing

		M	P	L	E

Hash h(A) **Wert: 1**

Rehash h‘(key) Wert:

0	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															

Double Hashing

		M	P	L	E

Hash h(A) Wert: 1

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															

Double Hashing

		M	P	L	E

Hash h() Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															

Double Hashing

		M	P	L	E

M - - - - > Hash h() Wert:

Rehash h‘() Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															

Double Hashing

			P	L	E

Hash h(M) Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															

Double Hashing

			P	L	E

Hash h(M) - > **Wert: 13**

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															

Double Hashing

			P	L	E

Hash h(M) Wert: 13

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15	16
S																
	A															
													M			

Double Hashing

			P	L	E

Hash h(M) - > Wert: 13

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			

Double Hashing

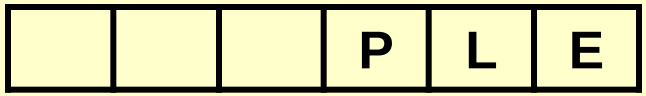
			P	L	E

Hash h() - > Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			

Double Hashing



P - - - > Hash h() Wert:

Rehash h‘() Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			

Double Hashing

				L	**E**

Hash h(P) Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			

Double Hashing

				L	E

Hash h(P) - > **Wert: 16**

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			

Double Hashing

				L	**E**

Hash h(P)

Wert: 16

Rehash h‘(key)

Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	**16**
S																
	A															
													M			
																P

Double Hashing

				L	E

Hash h(P) Wert: 16

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P

Double Hashing

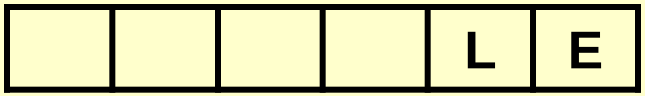
				L	E

Hash h() Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P

Double Hashing



L - - > Hash h() Wert:

Rehash h‘() Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P

Double Hashing

					E

Hash h(L) Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			

Double Hashing

					E

Hash h(L) - > **Wert: 13**

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P

Double Hashing

					E

Hash h(L)

Wert: **13**

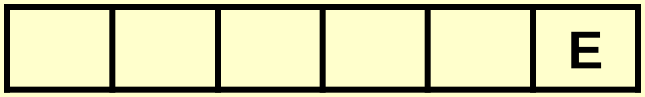
Rehash h‘(key)

Wert:

Kollision

0	1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15	16
S																
	A															
													M			
																P

Double Hashing



Hash h() Wert:

L - - > Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P

Double Hashing

					E

Hash h()

Wert:

Rehash h‘(L)

Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P

Double Hashing

					E

Hash h() Wert:

Rehash h‘(L) - > **Wert:** **12**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P

Double Hashing

					E

Hash h()

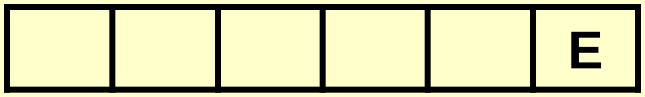
Wert:

Rehash h‘(**L**)

Wert: 12

0	1	2	3	4	5	6	7	8	9	10	11	**12**	13	14	15	16
S																
	A															
													M			
																P
												L				

Double Hashing



Hash h()

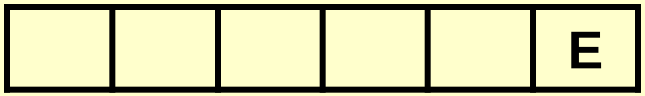
Wert:

Rehash h‘()

Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P
												L				

Double Hashing

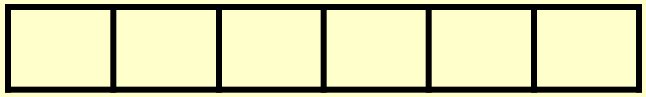


E - > Hash h() Wert:

Rehash h‘() Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
												L				

Double Hashing

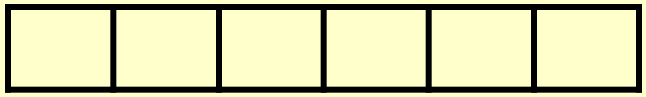


Hash h(E) Wert:

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
												L				

Double Hashing

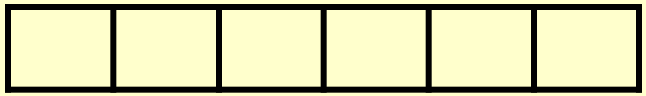


Hash h(E) - > **Wert: 5**

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
												L				

Double Hashing

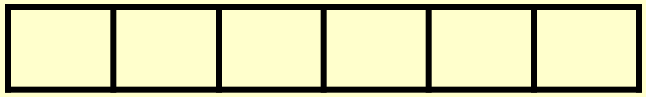


Hash h(E) Wert: 5

Rehash h‘(key) Wert:

0	1	2	3	4	**5**	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P
												L				
					E											

Double Hashing

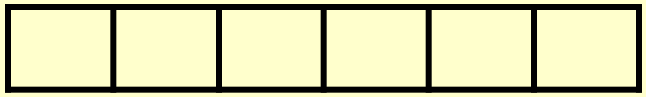


Hash h(E) Wert: 5

Rehash h‘(key) Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P
												L				
					E											

Double Hashing



Hash h()

Wert:

Rehash h‘(key)

Wert:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S																
	A															
													M			
																P
												L				
					E											

Übersicht über die Methoden und deren Aufgaben

(in der AlgoDat1 Template Implementation)

Double Hashing

Übersicht über die Methoden und deren Aufgaben

MultiHashTable()	Konstruktor
MultiHashTable(K keys[],V values[])	Konstruktor zum Füllen mit Daten
~MultiHashTable()	Destruktor
Put(K key, V value)	Element einfügen
Remove(K key)	Element löschen
Get(K key)	Element(e) ausgeben
getKeys	Alle Schlüsselwerte abrufen
getValues	Alle Werte abrufen
print	Tabelle ausgeben
toString	Table als String zurückgeben

Double Hashing

Allgemeines zur Implementation:

- Implementation der MultiHashTable hat lediglich den Anschein einer Tabelle mit Zugriff der Ordnung 1 auf Elemente.
- Intern als Liste (in Form von Vektoren) verwaltet.
- Gründe:
  - Sofern es zu Kollisionen kommt, sollen die Elemente am Ende der Tabelle hinzugefügt werden (lt. Spezifikation)
  - Es wird keine (Re)hash-Funktion verwendet, die errechnet, an welcher Stelle ein Element zu stehen hat => es muss immer die **ganze** Liste durchsucht werden, um (zusammengehörige) Elemente zu finden.
  - Diese Eigenschaften sind in den Performance-Tests ersichtlich

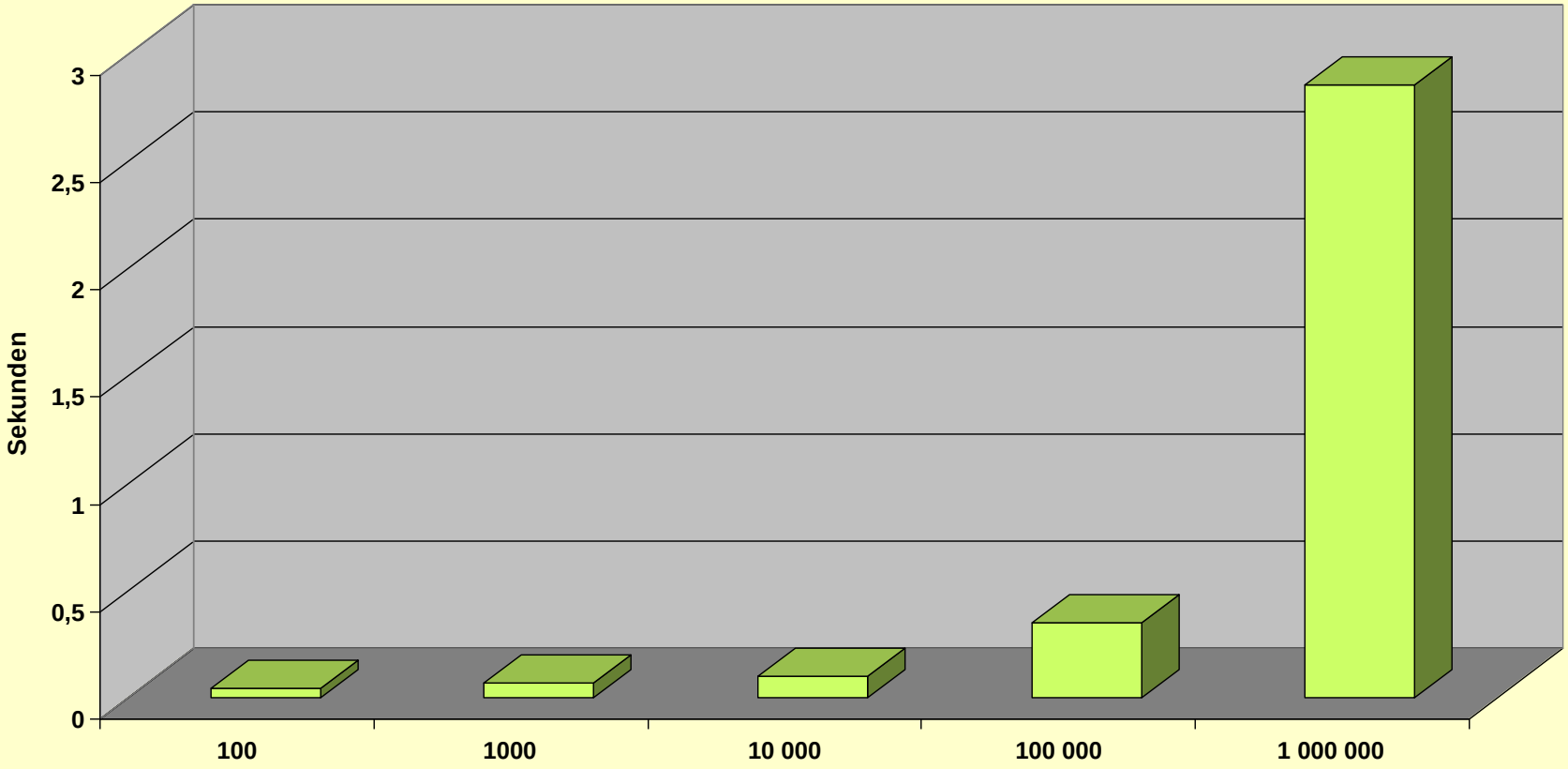
Performance Tests

„Wie schnell arbeitet die Implementation mit verschieden vielen Elementen“

Zeiten sind Durchschnittswerte von 10 Testmessungen

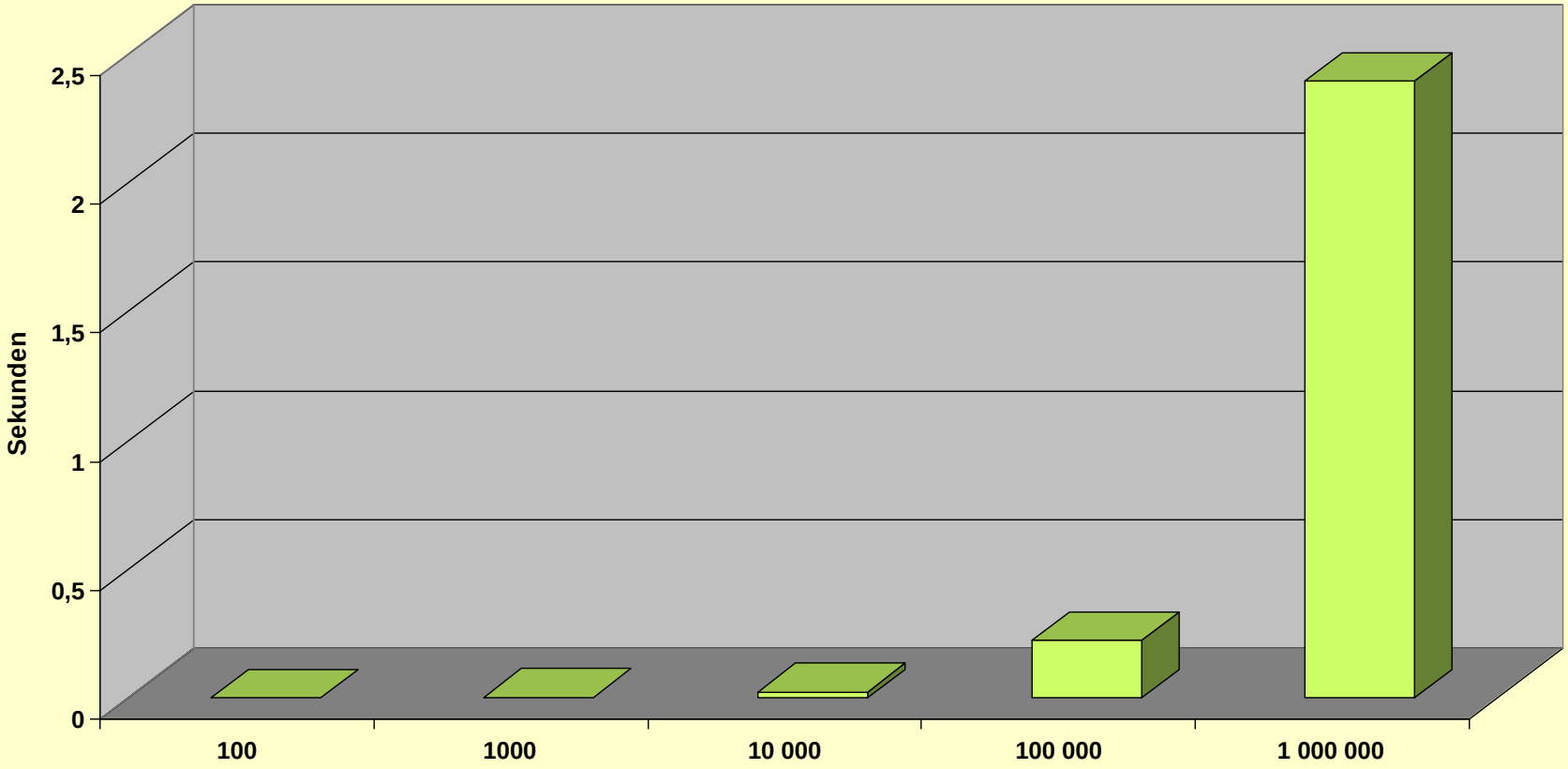
(Prozessor: Pentium 200 / 128MB RAM / Windows 2000 SP3)

Methode put (Auffüllen der Tabelle mit Werten)

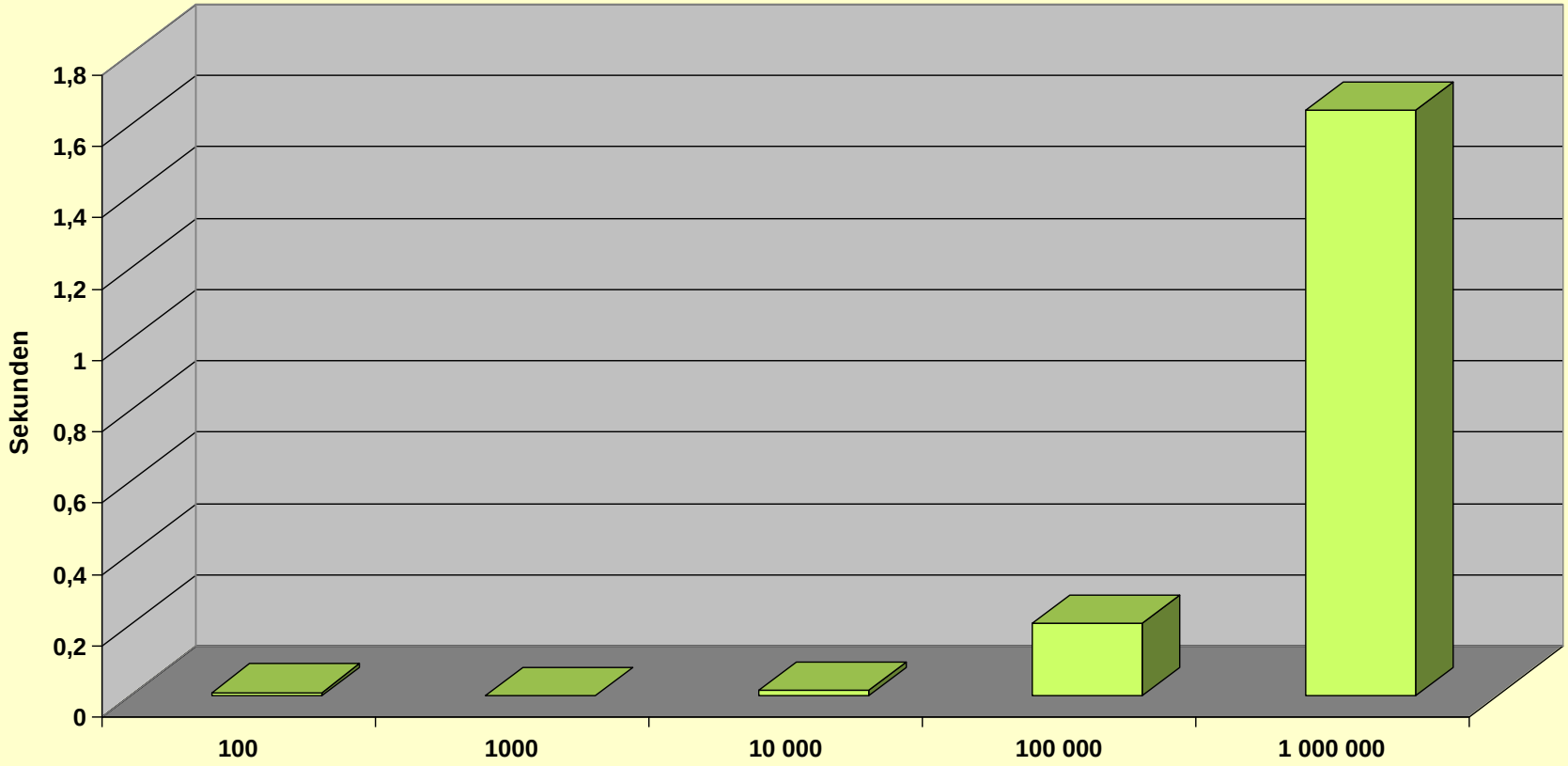


Double Hashing

Methode get (Abrufen von Wert(en) aus Table)



Methode remove (Löschen von Wert(en) aus Table)



Double Hashing

Ergebnisse in Zahlen (Table mit ...)

100 Elemente	**1 000 Elemente**	**10 000 Elemente**	**100 000 Elemente**	**1 000 000 Elemente**
put 100 Elements: 0.046s	**put** 1000 Elements: 0.071s	**put** 10000 Elements: 0.101s	**put** 100000 Elements: 0.348s	**put** 1000000 Elements: 2.855s
get 1 Element: 0s	**get** 1 Element: 0.003s	**get** 1 Element: 0.023s	**get** 1 Element: 0.224s	**get** 1 Element: 2.395s
remove 1 Element: 0.01s	**remove** 1 Element: 0s	**remove** 1 Element: 0.017s	**remove** 1 Element: 0.204s	**remove** 1 Element: 1.642s