Informatik II
Hashing

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Das Datenbank-Problem "revisited"

- Lösung bisher:
  - Preprocessing: Elemente 1x sortieren, kostet $O(n \log n)$
  - Laufzeit: Suchen kostet dann nur noch Zeit $O(\log n)$
  - Nachteil: Einfügen ist teuer
- Ziel: Datenstruktur mit möglichst folgenden Eigenschaften
  - Kein Preprocessing nötig
  - Suchen und Einfügen soll in Zeit O(1) möglich sein!
  - Beliebige Keys
- Namen: *Hash-Table*, *Dictionary, Symbol-Table, assoziatives Array*
  - In Python: Datentyp **`Dictionary`**
- Hash-Tabellen / Dictionaries sind — in gewissem Sinn — Verallgemeinerungen von Arrays

```
d = {}
d["hallo"] = 12
d["Paul"] = 18
```

Beispiele für Hash-Tabellen

- Memory Management: Tabelle im Betriebssystem
- Symbol-Tabellen im Compiler: Variablen/Identifier in einem Programm

Identifier	Type	Adress
i	int	0x87C50FA4
a	float	0x87C50FA8
x	double	0x87C50FAC
name	String	0x87C50FB2

- Environment-Variablen in Unix (Variablenname, Attribut)

  EDITOR=emacs
  GROUP=mitarbeiter
  HOST=vulcano
  HOSTTYPE=sun4
  PRINTER=hp5
  MACHTYPE=sparc

- Der Ort (= Pfad) ausführbarer Programme auf der Festplatte
  PATH=~/bin:/usr/local/gnu/bin:/usr/local/bin:/usr/bin:/bin:

Prinzipieller Ansatz

- *Hashing* = Aufteilung des gesamten Key-Universums
- Die Position des Daten-Elements im Speicher ergibt sich (zunächst) durch Berechnung direkt aus dem Key
  - keine Vergleiche & konstante Zeit
- Datenstruktur = lineares Array der Größe m → Hash-Tabelle
- Hashing (engl.: to hash = zerhacken) beschreibt eine spezielle Art der Speicherung einer Menge von Keys durch Zerlegung des Key-Universums

Typische Implementierung einer Hash-Klasse

```
class TableEntry( object ):
    def __init__( self, key, value ):
        self.key = key
        self.value = value

class HashTable( object ):
    def __init__( self, capacity ):
        self.capacity = capacity
        self.table = capacity * [ None ]
        # wir nehmen hier an, daß table ein statisches Array sei,
        # mit fester Größe, wie das auch in C++/Java der Fall wäre
    # Hash-Funktion
    def h( self, key ): …
    # Füge value mit Schlüssel key ein, falls noch nicht vorhanden
    def insert( self, key, value ): …
    # Lösche Element mit key aus Tabelle, falls vorhanden
    def delete( self, key ): …
    # Suche Element mit key und liefere dessen Wert
    def search( self, key ): …
```


Prinzipielle Idee des Hashings

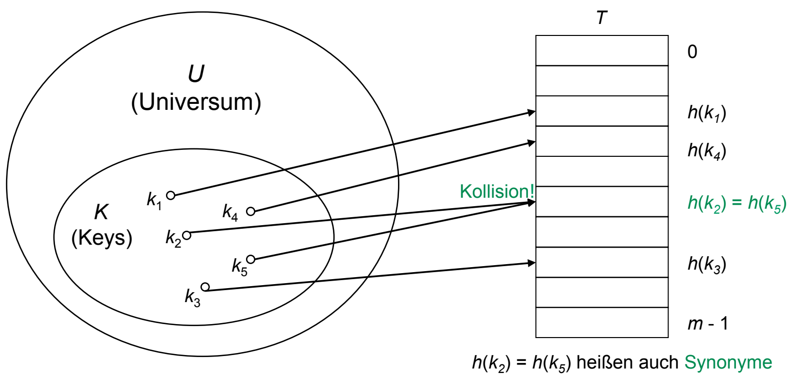
- Notation:
  - U = Universum aller möglichen Keys
  - K = Menge von Keys, die aktuell in der Hash-Tabelle gespeichert sind
  - $|K| = n$
- Beobachtung: wenn U sehr groß ist, ist ein Array für ganz U nicht praktikabel
  - Außerdem gilt im Allgemeinen: $|K| \ll |U|$
- Idee der Hash-Tabelle: benutze eine Tabelle, deren Größe in der selben Größenordnung wie $|K|$ ist
- Einträge der Hash-Tabelle nennt man Slots
- Definiere Funktion, die die Keys auf die Slots der Hash-Tabelle abbildet

- Hash-Funktion: surjektive Abbildung von U in alle Slots einer Hash-Tabelle T[0..m-1]

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

- Vergleiche Arrays: Key k wird abgebildet in den Slot $A[k]$
- Bei Hash-Tabellen: Key k wird abgebildet in den Slot $T[\,h(k)\,]$ ("*k hashes to …*")
- $h(k)$ heißt der Hash-Wert oder die Hash-Adresse des Keys k

Kollisionen und Belegungsfaktor



$h(k_2) = h(k_5)$ heißen auch Synonyme

- Normalerweise gilt |U| >> m ⇒ h kann nicht injektiv sein ⇒ (Adress-)Kollisionen sind unvermeidlich
- Belegungsfaktor (*load factor*):

$$\alpha = \frac{\#\text{ gespeicherter Keys}}{\text{Größe der Hash-Tabelle}} = \frac{|K|}{m}$$

Die beiden Bestandteile eines Hash-Verfahrens

- Hash-Verfahren :=
  1. möglichst "gute" Hash-Funktion +
  2. Strategie zur Auflösung von Adresskollisionen

- Annahme im Folgenden: Tabellengröße m ist fest

Anforderungen an gute Hash-Funktionen

- Eine Kollision tritt dann auf, wenn bei Einfügen eines Elementes mit Schlüssel k der Slot $T[\, h(k)\,]$ schon belegt ist
- Eine Hash-Funktion h heißt perfekt für eine Menge von Keys K, falls keine Kollisionen für K auftreten
- Die Hash-Funktion h kann nur dann perfekt sein, wenn $|K| = \leq m$; m.a.W.: der Belegungsfaktor der Hash-Tabelle muß $\frac{n}{m} \leq 1$ sein
- Eine Hash-Funktion ist gut, wenn:
  - für viele Schlüssel-Mengen auch bei hohem Belegungsfaktor die Anzahl der Kollisionen möglichst klein ist; und
  - diese effizient zu berechnen ist.

Beispiel einer Hash-Funktion

- U = {alle möglichen Identifiers in einer Programmiersprache}
  - M.a.W.: U = alle möglichen Funktions-, Variablen-, ..., Klassennamen
- Eine mögliche, einfache Hash-Funktion für Strings:

```
# m = size of table = const
def h( k, m ):
    s = 0
    for i in range( 0, len(k) ):
        s += ord( k[i] )    # ord = ASCII value of char
    return s % m
```

- Folgende Hash-Adressen werden generiert für m = 13:

Key k	h(k)
Test	0
Hallo	2
bar	10
Algo	10

- h wird umso besser, je größer m gewählt wird

Zur Wahrscheinlichkeit einer Kollision

- Die Anforderungen "hoher Belegungsfaktor" und "Kollisionsfreiheit" stehen offensichtlich in Konflikt zueinander
- Für eine Menge K, mit $|K|=n$, und Tabelle T, mit $|T|=m$, gilt:
  - für $n > m$ sind Konflikte unausweichlich
  - für $n \leq m$ gibt es eine (Rest-) Wahrscheinlichkeit $P(n, m)$ für das Auftreten mindestens einer Kollision
- Wie findet man eine Abschätzung für $P(n, m)$?
  - Für beliebigen Key k ist die W'keit dafür, daß $h(k) = j$, $j \in \{0,\ldots,m\text{-}1\}$: $P[\ h(k) = j\ ] = \frac{1}{m}$, falls Gleichverteilung gilt
  - Es ist $P(n, m) = 1 - \overline{P}(n, m)$, wenn $\overline{P}(n, m)$ die W'keit dafür ist, daß es beim Speichern von n Elementen in m Slots zu keinen Kollisionen kommt

- Werden n Schlüssel nacheinander auf die Slots T_0, …, T_{m-1} verteilt (bei Gleichverteilung), gilt jedes mal

$$P[h(s) = j] = \frac{1}{m}$$

- Die W'keit für keine Kollision im Schritt i ist $p_i = \frac{m-(i-1)}{m}$
- Damit ist

$$P(n,m) = 1 - p_1 \cdot p_2 \cdots p_n = 1 - \frac{m(m-1)\cdots(m-n+1)}{m^n}$$

- Beispiel ("Geburtstagsparadoxon"):

$$P(23{,}365) > 50\%$$

und

$$P(50{,}365) \approx 97\%$$

Gebräuchliche Hash-Funktionen

- Zunächst die Divisions-Methode
- Für U = Integer wird die Division-mit-Rest-Methode verwandt:

$$h(s) = (a \times s) \bmod m \ (a \neq 0, a \neq m, m \text{ Primzahl})$$

- Für Strings der Form $s = s_1 s_2 \ldots s_k$ nimmt man oft:

$$h(s) = \left(\left(\sum_{i=1}^{k} B^i s_i \right) \bmod 2^w \right) \bmod m$$

  - etwa mit B = 131 und w = Wortbreite des Rechners (w = 32 oder w = 64 ist üblich).

- (Einfache) Divisions-Methode:

$$h(k) = k \mod m$$

- Wahl von m ?
- Schlechte Beispiele:
  - m gerade → $h(k)$ gerade ⇔ k gerade
    - Ist problematisch, wenn letztes Bit eine spezielle Bedeutung hat! (z.B. 0 = weiblich, 1 = männlich)
  - $m = 2^p$ liefert die p niedrigsten Binärziffern von k, d.h., höhere Ziffern gehen gar nicht in die Hash-Adresse ein!
- Regel: Wähle m prim, wobei m keine Zahl $2^i \pm j$ teilt, wobei i und j kleine, nicht-negative Zahlen sind, d.h., wähle m prim, aber nicht "zu nahe" an einer 2-er-Potenz

Multiplikative Methode

- Satz von Vera Turán Sós [1957]:

  Sei Θ eine irrationale Zahl. Platziert man die Punkte $\Theta - \lfloor\Theta\rfloor, 2\Theta - \lfloor 2\Theta\rfloor, \ldots, n\Theta - \lfloor n\Theta\rfloor$ in das Intervall $[0,1]$, dann haben die $n+1$ Intervallteile höchstens 3 verschiedene Längen. Außerdem fällt der nächste Punkt $(n+1)\Theta - \lfloor(n+1)\Theta\rfloor$ in eines der größten (schon existierenden) Intervallteile.

- Fazit: die so gebildeten "Punkte" liegen ziemlich gleichmäßig gestreut im Intervall [0,1]
- Es gilt: von allen Zahlen θ, 0 ≤ θ ≤ 1, führt der goldene Schnitt

$$\theta^{-1} = \frac{\sqrt{5}-1}{2} \approx 0.6180339$$

zur gleichmäßigsten Verteilung.

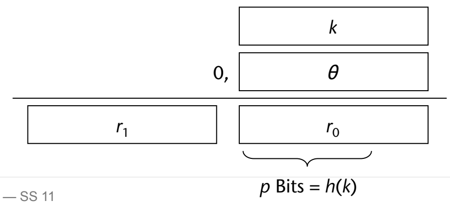
- Wähle eine Konstante $\theta, 0 < \theta < 1$
  1. Berechne $k\theta \bmod 1 := k\theta - \lfloor k\theta \rfloor$
  2. $h(k) = \lfloor m(k\theta \bmod 1) \rfloor$

- Beispiel:

$$\begin{aligned}
\theta^{-1} &= \frac{\sqrt{5}-1}{2} \approx 0.6180339 \\
k &= 123456 \\
m &= 10000 \\
h(k) &= \lfloor 10000(123456 \cdot 0.61803\ldots \bmod 1 \rfloor \\
&= \lfloor 10000(76300.0041151\ldots \bmod 1 \rfloor \\
&= \lfloor 41.151\ldots \rfloor = 41
\end{aligned}$$

Praktische Berechnung von *h* in der multiplikativen Methode

- Wahl von m (= Tabellengröße) ist hier unkritisch → wähle $m = 2^p$
- Ann.: k passe in ein einzelnes Wort, d.h., k hat w Bits
- Wähle $\theta \in [0,1)$ und setze $s = \theta \cdot 2^w$
- Dann ist $k \cdot s = k \cdot \theta \cdot 2^w = r_1 2^w + r_0$
- r_1 ist der ganzzahlige Teil von $k\theta$ (= $\lfloor k\theta \rfloor$) und r_0 ist der gebrochene Rest (= $k\theta \bmod 1 = k\theta - \lfloor k\theta \rfloor$)
- Damit kann man $h(k)$ mit Integer-Arithmetik berechnen:



Universelles Hashing

- Problem: h fest gewählt → es gibt ein $S \subseteq U$ mit vielen Kollisionen
  - Wir können nicht annehmen, daß die Keys gleichverteilt im Universum liegen (z.B. Identifier im Programm)
  - Könnte also bspw. passieren, daß die Compile-Zeit bei einigen best. Programmen sehr lange dauert, weil es sehr viele Kollisionen gibt
- Idee des universellen Hashing:
  - wähle Hash-Funktion h zufällig (→ randomisierte Datenstruktur)
- Definition: Sei H endliche Menge von Hash-Funktionen, $h \in H : U \to \{0, \ldots, m-1\}$, dann heißt H universell, wenn gilt:

$$\forall x, y \in U, x \neq y : \frac{|\{h \in H | h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

- Äquivalent: für $x, y \in U$ beliebig und $h \in H$ zufällig gilt

$$P\left[h(x) = h(y)\right] \leq \frac{1}{m}$$

- Definition: "Kollisionsindikator"

$$\delta(x, y, h) = \begin{cases} 1 & \text{falls } h(x) = h(y) \text{ und } x \neq y \\ 0 & \text{sonst} \end{cases}$$

- Erweiterung von δ auf Mengen

$$\delta(x, S, h) = \sum_{s \in S} \delta(x, s, h)$$

$$\delta(x, y, G) = \sum_{h \in G} \delta(x, y, h)$$

- Definition für "universell" nochmal mit δ formuliert:

  H ist universell $\Leftrightarrow \forall x, y \in U : \delta(x, y, H) \leq \frac{|H|}{m}$

Nutzen des Universellen Hashing

- Sei K eine Menge von Schlüsseln, die in Tabelle T gespeichert werden sollen
  - Wähle zufällig Hash-Funktion $h \in H$, diese bleibt fest für die restliche Lebensdauer der Tabelle
  - Bilde alle Schlüssel $k \in K$ mit h auf Tabelle ab und füge diese ein
- Nun soll weiterer Key x gespeichert werden
  - Vernünftig ist: Maß für Aufwand = #Kollisionen zw. x und allen $k \in K$
  - Berechne den Erwartungswert für diese Anzahl, also E[δ(x, K, h)]

$$
\begin{aligned}
E\left[\delta(x,K,h)\right] &= \frac{1}{|H|}\sum_{h\in H}\delta(x,K,h) \\
&= \frac{1}{|H|}\sum_{h\in H}\sum_{y\in K}\delta(x,y,h) \\
&= \frac{1}{|H|}\sum_{y\in K}\sum_{h\in H}\delta(x,y,h) \\
&= \frac{1}{|H|}\sum_{y\in K}\delta(x,y,H) \\
&\leq \frac{1}{|H|}\sum_{y\in K}\frac{|H|}{m} \\
&= \frac{|K|}{m}
\end{aligned}
$$

- Schlußfolgerung:

$$E\left[\delta(x, K, h)\right] \leq \frac{|K|}{m}$$

Man kann also erwarten, daß eine aus einer universellen Klasse *H* von Hash-Funktionen zufällig gewählte Funktion *h* eine beliebige, noch so "bösartig" gewählte Folge von Schlüsseln (also bei einem *"malicious adversary"*) so gleichmäßig wie nur möglich in der Hash-Tabelle verteilt.

Eine universelle Klasse von Hash-Funktionen

- Annahmen: |U| = p, mit Primzahl p und U = {0, …, p-1}
  - Seien a ∈ {1, …, p-1} und b ∈ {0, …, p-1}
  - Definiere $h_{a,b} : U \to \{0, \ldots, m-1\}$ wie folgt

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m$$

- Satz: Die Menge

$$H = \left\{ h_{a,b} \mid 1 \leq a < p, 0 \leq b < p \right\}$$

ist eine universelle Klasse von Hash-Funktionen.

Beispiel

- Hashtabelle *T* der Größe 3, |*U*| = 5
- Betrachte die 20 Funktionen (Menge *H*):

1x+0	2x+0	3x+0	4x+0
1x+1	2x+1	3x+1	4x+1
1x+2	2x+2	3x+2	4x+2
1x+3	2x+3	3x+3	4x+3
1x+4	2x+4	3x+4	4x+4

jeweils (mod 5) (mod 3), d.h. p = 5 und m = 3

- Betrachte die Schlüssel 1 und 4 :

(1*1+0) mod 5 mod 3 = 1 = (1*4+0) mod 5 mod 3 $h_{1,0}(1) = h_{1,0}(4)$

(1*1+4) mod 5 mod 3 = 0 = (1*4+4) mod 5 mod 3 $h_{1,4}(1) = h_{1,4}(4)$

(4*1+0) mod 5 mod 3 = 1 = (4*4+0) mod 5 mod 3 $h_{4,1}(1) = h_{4,1}(4)$

(4*1+4) mod 5 mod 3 = 0 = (4*4+4) mod 5 mod 3 $h_{4,4}(1) = h_{4,4}(4)$

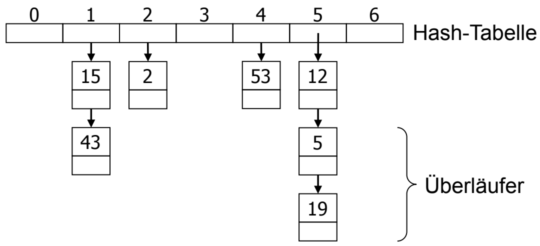
Möglichkeiten der Kollisionsbehandlung

- Die Behandlung von Kollisionen erfolgt bei verschiedenen Verfahren unterschiedlich
- Ein Key *k* ist ein Überläufer, wenn der Slot *h*(*k*) schon durch einen anderen Key (inkl. "dranhängendem" Datensatz) belegt ist
- Wie kann mit Überläufern verfahren werden?
  1. Chaining: Slots werden durch verkettete Listen realisiert, Überläufer werden in diesen Listen abgespeichert (Hashing mit Verkettung der Überläufer)
  2. Open Addressing: Überläufer werden in anderen, noch freien (*open*) Slots abgespeichert. Diese werden beim Speichern und Suchen durch ein systematisches und konsistentes Verfahren, sogenanntes Sondieren (*probing*), gefunden (Offene Hash-Verfahren)

Chaining

- Die Hash-Tabelle ist ein Array (Länge m) von Listen, jeder Slot ist der Kopf einer Liste
- Zwei verschiedene Möglichkeiten der Listen-Anlage:
  1. Hash-Tabelle enthält nur Listen-Köpfe, Datensätze sind in Listen: Direkte Verkettung
  2. Hash-Tabelle enthält pro Slot maximal einen Datensatz sowie einen Listen-Kopf, Überläufer kommen in die Liste: Separate Verkettung

Beispiel:
$h(k) = k \bmod 7$



- Suchen nach Key k:
  - Berechne $h(k)$
  - Suche nach k in der Überlaufliste $T[h(k)]$
- Einfügen eines Keys k:
  - Suchen nach k (erfolglos)
  - Einfügen in die Überlaufliste
- Entfernen eines Keys k:
  - Suchen nach k (erfolgreich)
  - Entfernen aus Überlaufliste
- Alles sind reine Listenoperationen

```
class HashTable( object ):
  def __init__( self, m ):
    self.table = m * [None]

  def insert( self, key, value ):
    e = TableEntry( key, value )
    a = self.h( key )
    self.table[a].append( e )

  def search( self, key ):
    l = self.table[ h(key) ]
    for e in l:
      if e.key == key:
        return e.value
    return None
```

Test-Programm

```
import sys
import HashTable.py
ht = HashTable( 17 )
for i in range( 0, len(sys.argv) ):
    ht.insert( sys.argv[i] )
ht.print()
for i in range( 0, len(sys.argv), 2 ):
    ht.delete( sys.argv[i] )
ht.print()
```

- Aufruf: **HashTableTest 12 53 5 15 2 19 43**
- Ausgabe:

```
0: -|                   0: -|
1: 15 -> 43 -|          1: 15 -|
2: 2 -|                 2: -|
3: -|                   3: -|
4: 53 -|                4: 53 -|
5: 15 -> 5 -> 19 -|     5: 19 -|
6: -|                   6: -|
```


Effizienz eines Hash-Verfahrens

- Aufwand für die Berechnung von h ist immer in $O(1)$
- Aufwand für Suchen, Einfügen, Löschen im worst-case ist immer in $O(m)$ bzw. $O(n)$ (m = Größe, n = Belegung der Hash-Table)
  - Was uns also interessiert ist die Average-Case-Laufzeit
  - Beim Löschen muß vorher ein Element (erfolgreich) gesucht werden
  - Beim Einfügen muß vorher ein Element (erfolglos) gesucht werden
- Bestimme im Folgenden zwei Erwartungswerte, bezogen auf eine feste Tabellengröße m:

  C_n = Erwartungswert der Anzahl der „besuchten" Einträge bei erfolgreicher Suche

  C_n' = Erwartungswert der Anzahl der „besuchten" Einträge bei erfolgloser Suche

  wobei n = Anzahl der belegten Einträge in der Tabelle

Analyse des Chaining

- Annahme: uniformes Hashing, d.h.
  - alle Hashadressen werden mit gleicher Wahrscheinlichkeit gewählt, d.h.: $P[h(k_i) = j] = \frac{1}{m}$; und
  - unabhängig von Operation zu Operation
- Mittlere Listenlänge bei n Einträgen: $\frac{n}{m} =: \alpha$
- Analyse:

$$C'_n = \alpha$$

$$C_n \approx 1 + \frac{\alpha}{2}$$

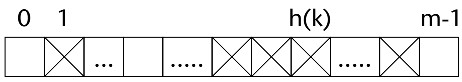
  - Vergleiche Aufwand beim linearen Suchen
  - Bemerkung: wenn $n \in O(m)$ [z. B. $n \leq m$], dann ist

$$C'_n, C_n \in O(1)$$

- Vorteile des Chainings:
  - C_n und C_n' sind niedrig
  - $\alpha > 1$ ist möglich
  - Für Sekundärspeicher geeignet
- Nachteile:
  - Zusätzlicher Speicherplatz für Zeiger,
  - Überläufer liegen außerhalb der Hash-Tabelle (Cache!)

Offene Hash-Verfahren (open adressing)

- Idee:
  - Bringe Überläufer an freien („offenen") Slots in Hash-Tabelle unter
  - Falls *T*[*h*(*k*)] belegt, suche einen anderen Slot für *k* nach fester Regel
- Beispiel:
  - Betrachte Slot mit nächst-kleinerem Index (mod *m*)
  - Falls Key gefunden → fertig, liefere dort gespeicherten Value
  - Sonst: betrachte nächstkleineren Slot, bis man auf leeren Slot stößt



- Problem: Entfernen von Keys → nur als "entfernt" markieren !

- Allgemeiner: verwende eine Sondierungsfolge (*probe sequ.*)

$$(h(k) - s(j,k)) \mod m \qquad j = 0, \ldots, m-1$$

für eine gegebene Sondierungsfunktion *s*(*j*,*k*)

- Wenn *T*[*h*(*k*)] belegt ist, suche nach einem freien Slot:

```
j = 0            # Anzahl der inspizierten Einträge
i = ( h(k) - s(j,k) ) % m
while T[i].status != SLOT_FREE:
    j += 1
    i = ( h(k) - s(j,k) ) % m
```

Der Such-Algorithmus

```
# Suche nach Key k in der Hash-Tabelle liefert Item
# oder None
# T[i].status ∈ { SLOT_FREE, SLOT_OCCUPIED, SLOT_DELETED }
def search( self, k ):
    j = 0            # Anzahl der inspizierten Einträge
    i = ( h(k) - s(j,k) ) % m
    while T[i].status != SLOT_FREE and T[i].key != k:
        j += 1
        i = ( h(k) - s(j,k) ) % m
    if T[i].key == k and T[i].status == SLOT_OCCUPIED:
        return T[i].item
    else:
        return None
```


- Erwünschte Eigenschaften von *s(j,k)*:

$$\text{Folge}\quad \begin{aligned} &(h(k) - s(0,k)) \bmod m, \\ &(h(k) - s(1,k)) \bmod m, \\ &\vdots \\ &(h(k) - s(m-2,k)) \bmod m, \\ &(h(k) - s(m-1,k)) \bmod m \end{aligned}$$

sollte eine Permutation von 0, …, m-1 liefern

Lineares Sondieren

- $s(j, k) = j$
- Sondierungsfolge für k: $h(k)$, $h(k)$-1, …, 0, m-1, …, $h(k)$+1
- Problem: primäre Häufung (*"primary clustering"*)
- Beispiel:

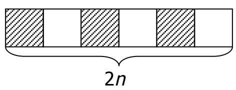
0	1	2	3	4	5	6
			5	53	12	

  - P[nächstes Objekt landet an Position 2] = 4/7
  - P[nächstes Objekt landet an Position 1] = 1/7
  - Lange Cluster werden mit größerer Wahrscheinlichkeit verlängert als kurze

Effizienz des linearen Sondierens

- Betrachte erfolglose Suche in zwei Extremen:
  - In beiden Fällen ist der Load-Factor = 1/2

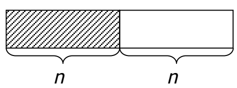
1. Jeder 2-te Slot besetzt:



   Mittlerer Aufwand =

$$1 + \frac{0 + 1 + 0 + \cdots}{2n} = 1 + \frac{1}{2}$$

2. Nur 1 besetzter Cluster:



   Mittlerer Aufwand =

$$1 + \frac{n + (n-1) + \cdots + 1 + 0 + \cdots + 0}{2n} \approx 1 + \frac{n}{4}$$

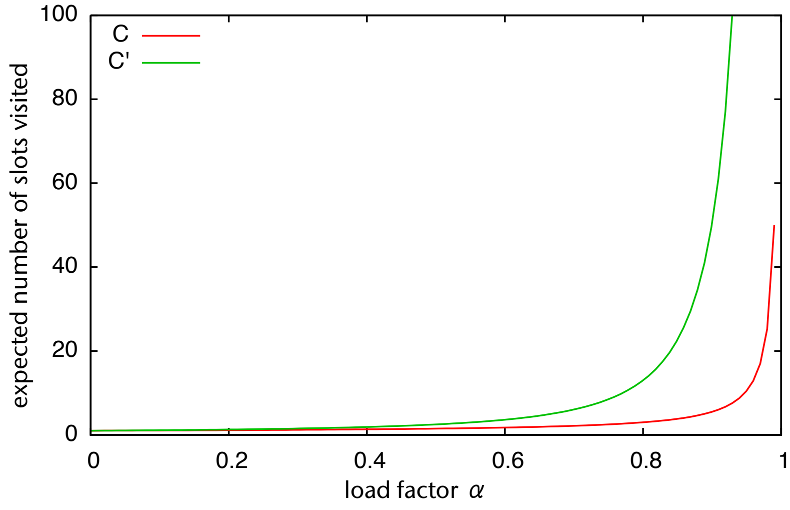
- Allgemein gilt für das lineare Sondieren (o. Bew.):
  - Erfolgreiche Suche:

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

  - Erfolglose Suche:

$$C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

  - Wobei $\alpha = \frac{n}{m}$, $0 < \alpha < 1$, der Belegungsfaktor (load factor) ist.
- Die Analyse [Knuth, 1962] muß über alle mögliche Belegungen der Hash-Tabelle gehen
- Fazit: die Effizienz des linearen Sondierens verschlechtert sich drastisch, sobald sich der Belegungsfaktor α dem Wert 1 nähert



Quadratisches Sondieren

- Idee: versuche, *primary clustering* zu vermeiden, indem durch quadratisch wachsenden Abstand um *h(k)* herum nach freiem Platz gesucht wird
- Die Sondierungsfunktion:

$$s(j,k) = (-1)^j \cdot \left\lceil \frac{j}{2} \right\rceil^2$$

- Sondierungsfolge für k ist

$$h(k),\ h(k)+1,\ h(k)-1,\ h(k)+4,\ h(k)-4,\ \ldots$$

- Satz (o. Bew.)
  Die quadratische Sondierungsfunktion lieferte eine Permutation, falls *m eine* Primzahl der Form $4i+3$ ist.

Effizienz (o. Bew.)

- Erfolgreiche Suche:

$$C_n \approx 1 - \frac{\alpha}{2} + \ln\left(\frac{1}{1-\alpha}\right)$$

- Erfolglose Suche:

$$C'_n \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

- Problem: sekundäre Häufung, d.h., zwei Synonyme k_1 und k_2 (d.h. $h(k_1) = h(k_2)$) durchlaufen stets dieselbe Sondierungsreihenfolge

Double Hashing

- Idee: Wähle eine zweite Hash-Funktion h'

$$s(j,k) = j \cdot h'(k)$$

- Sondierungsfolge für *k* ist somit

$$h(k),\ \ h(k) - h'(k),\ \ h(k) - 2h'(k),\ \ \ldots$$

- Forderung: Sondierungsfolge muß Permutation der Hash-Adressen ergeben
- Folgerung daraus:

$$h'(k) \neq 0 \ \wedge \ h'(k) \nmid m$$

- Beispiel: wähle *m* prim und

$$h'(k) = 1 + (k \mod (m-2))$$

Beispiel

- Hash-Funktionen:

$$h(k) = k \bmod 7$$
$$h'(k) = 1 + (k \bmod 5)$$

- Schlüsselfolge: 15, 22, 1, 29, 26

0	1	2	3	4	5	6	
	15						$h'(22) = 3$
	15				22		$h'(1) = 2$
	15				22	1	$h'(29) = 5$
	15		29		22	1	$h'(26) = 2$

- In diesem Beispiel genügen fast immer 1 oder 2 Sondierschritte

Analyse von Double-Hashing

- Satz: Wenn Kollisionen mit Double-Hashing aufgelöst werden, dann ist die durchschnittliche Anzahl von Sondierungsschritten in einer Tabelle der Größe *m* mit $n=\alpha m$ vielen Elementen
$$C_n = \frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right) \quad \text{bzw.} \quad C'_n = \frac{1}{1-\alpha}$$
für die erfolgreiche bzw. erfolglose Suche.
- Beweis:
  - Sehr kompliziert [Guibas & Szemeredi]
  - Idee: Zeige, daß Double-Hashing fast äquivalent zu dem (aufwendigeren) *Random-Hashing* ist
- Random-Hashing: $s(j,k)$ ist eine Folge von Pseudo-Zufallszahlen, die von *k* abhängt und jeden Slot der Hash-Tabelle gleich wahrscheinlich trifft (mehrfache Hits sind aber möglich)

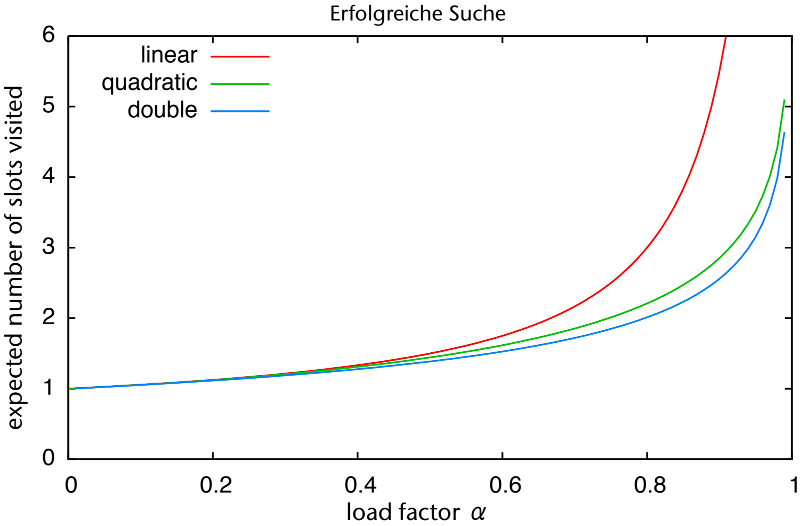
Analyse der mittleren Kosten für *Random-Hashing*

- Definiere Zufallsvariable X = Anzahl der Sondierungen bei erfolgloser Suche
- Sei $P[X \geq i]$:= Wahrscheinlichkeit, daß eine Suche *i* oder mehr Sondierungsschritte machen muß, $i = 1, 2, \ldots$
- Klar: $P[X \geq 1] = 1$
- $P[X \geq 2]$ = W'keit, daß erster (zufälliger) untersuchter Slot belegt ist $= \frac{n}{m} = \alpha$
- $P[X \geq 3]$ = W'keit, daß erster (zufällig gewählter) Slot belegt ist *und* zweiter (zufällig gewählter) Slot besetzt ist $= \frac{n}{m} \cdot \frac{n}{m} = \alpha^2$

$$
\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{\infty} i \cdot \mathrm{P}[X = i] \\
&= 1 \cdot \mathrm{P}[X = 1] + 2 \cdot \mathrm{P}[X = 2] + \cdots \\
&= \mathrm{P}[X = 1] + \\
&\quad\ \mathrm{P}[X = 2] + \mathrm{P}[X = 2] + \\
&\quad\ \mathrm{P}[X = 3] + \mathrm{P}[X = 3] + \mathrm{P}[X = 3] + \\
&\quad\ \vdots \qquad \longrightarrow \text{Additivität, da Ereignisse unabhängig} \\
&= \mathrm{P}[X = 1 \vee X = 2 \vee \ldots] + \\
&\qquad\qquad \mathrm{P}[X = 2 \vee X = 3 \vee \ldots] + \\
&\qquad\qquad\qquad \mathrm{P}[X = 3 \vee X = 4 \vee \ldots] + \cdots \\
&= \mathrm{P}[X \geq 1] + \mathrm{P}[X \geq 2] + \mathrm{P}[X \geq 3] + \cdots \\
&= 1 + \alpha + \alpha^2 + \cdots = \frac{1}{1 - \alpha}
\end{aligned}
$$

- Fazit:
  - Double-Hashing ist genauso effizient wie randomisiertes Sondieren
  - Double-Hashing ist schneller, um einen konstanten Faktor (Pseudo-Zufallszahlen sind rel. teuer)

Vergleich



Das Dictionary in Python

- Ein Dictionary ist ein assoziatives Array, bei dem Objekte mit Schlüsseln (*Keys*) indiziert werden (anstatt Integer)

```
student = {
              "Name"        : "Meier",
              "Matrikelnummer" : "123456",
              "Klausurnote"    : 4
           }
```

- Zugriff auf die Elemente erfolgt über den Index-Operator:

```
a = student["Name"]
student["Klausurnote"] = 1
```

- Wird oft (leider) *Hash* oder *Map* genannt (diese sind bloß konkrete Implementierung der allg. Datenstrukur "Dictionary")

Beispiel: "Chaos Game" mit Dictionary

```
import Image
import random
import sys

im = Image.new("RGB", (512, 512), (256, 256, 256) )
N = int( sys.argv[1] )

vertex = { "r" : (0.0,0.0),   # dictionary of coords of vertices
           "g" : (512.0,0.0),
           "b" : (256.0,443.4)
         }
x0, y0 = 0.0, 0.0                     # start at "red" vertex

for i in range( 0, N ):
    r = random.random()

    if r < 0.333:
        x1, y1 = vertex["r"]
    elif r < 0.6667:
        x1, y1 = vertex["g"]
    else:
        x1, y1 = vertex["b"]

    x0 = ( x0 + x1 ) / 2.0
    y0 = ( y0 + y1 ) / 2.0
    im.putpixel ( (int(x), int(y)), (int(x), int(y), 0) )

im.show()
```


- Programmatisches Erzeugen von Dictionaries:

```
d = dict()
d["Name"] = "Meier"
d["Matrikelnummer"] = 007
```

- Testen von Enthaltensein mit dem Keyword **`in`**:

```
if "Name" in student:
      name = student["Name"]
else:
      name = "Mustermann"
```

- Eine Schlüssel-Liste erhält man mit **`keys()`**:

```
l = student.keys()
# liefert
# ["Name","Matrikelnummer","Klausurnote"]
```

- Mit **`del`** entfernt man Elemente (= *key-value pair*):

```
del student["Klausurnote"]
```


Operationen auf Dictionaries

Operationen auf Dicitonaries	
`len(d)`	Ergibt die Anzahl der Elemente in d
`d[k]`	Ergibt Element von d mit Schlüssel k
`d[k] = x`	Setzt d[k] auf x
`x in d`	Liefert True, wenn x als Key im Dictionary d vorhanden ist
`del d[k]`	Entfernt d[k] aus d
`d.clear()`	Entfernt alle Elemente von d
`d.copy()`	Macht eine Kopie von d
`d.has_key(k)`	Ergibt 1, falls d einen Schlüssel k enthält, sonst 0
`d.items()`	Ergibt eine Liste von Schlüssel-Wert-Paaren
`d.keys()`	Ergibt eine Liste aller Schlüssel in d
`d.values()`	Ergibt eine Liste aller Objekte in d

wobei **`d`** ein Dictionary ist (d.h. **`type(d) == dict`**)