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

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

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	$h_{4,4}(1) = h_{4,4}(4)$

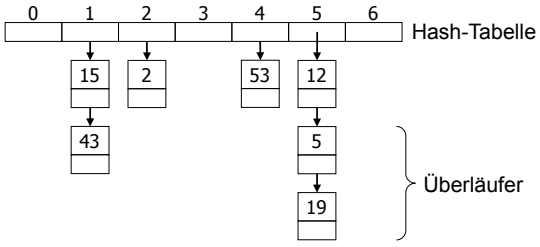
Möglichkeiten der Kollisionsbehandlung

- 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* mod 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:

$$C_n' = \alpha$$

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

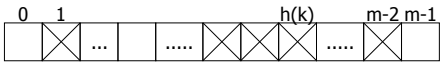
  - Vergleiche Aufwand beim linearen Suchen
  - Bemerkung: wenn $n \in O(m)$ [z. B. $n \le m$], dann ist

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

- 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: Unterbringung der Überläufer an freien („offenen") Plätzen in Hash-Tabelle
  - falls $T[\, h(k)\,]$ belegt, suche einen anderen Platz für k nach fester Regel
- Beispiel: Betrachte Eintrag mit nächst-kleinerem Index (mod m)



- 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 Platz:

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



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

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

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

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

Lineares Sondieren

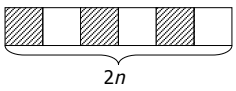
- *s*(*j*, *k*) = *j*
- Sondierungsfolge für *k*: *h*(*k*), *h*(*k*)-1, …, 0, m-1, …, *h*(*k*)+1
- Problem: primäre Häufung (*"primary clustering"*)
- Beispiel:

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

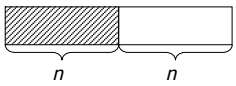
Effizienz des linearen Sondierens

- 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 =



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

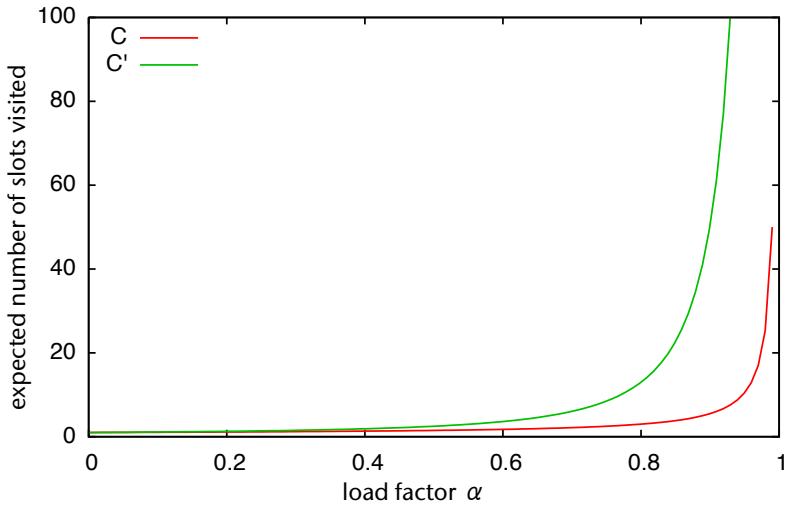
- Allgemein gilt für das lineare Sondieren (o. Bew.):
  - Erfolgreiche Suche:

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

  - 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, …

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

$$h'(k) \neq 0 \ \wedge \ h'(k) \not| m$$

- 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)

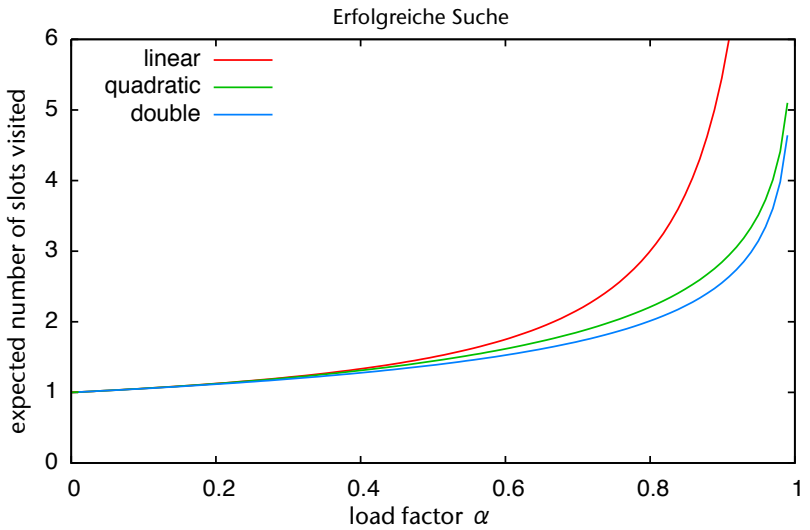
Analyse der mittleren Kosten für *Random-Hashing*

- 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, …
- 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 $= \frac{n}{m} \cdot \frac{n}{m} = \alpha^2$

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

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

Vergleich



Verbesserung der erfolgreichen Suche

- Beobachtung:

  Kollision beim Einfügen bedeutet k trifft in $T[i]$ auf k_{alt}, d.h.

  $$i = h(k) - s(j, k) = h(k_{alt}) - s(j', k_{alt})$$

- Problem:
  - Die Sondierungsfolge

    $$h(k) - s(1, k), h(k) - s(2, k), \ldots$$

    könnte lang werden
  - Dieselbe Folge muss man später bei der Suche nach k wieder durchlaufen
  - Evtl. wäre die Folge

    $$h(k_{alt}) - s(j'+1, k_{alt}), h(k_{alt}) - s(j'+2, k_{alt}), \ldots$$

    viel schneller auf einen leeren Slot gestoßen!

Brent's Verfahren

- Idee: suche freien Platz für k oder k_{alt}
- 2 Strategien:
  - (S1) k_{alt} bleibt in $T[i]$: betrachte neue Position für k

    $$h(k) - s(j+1, k) \text{ für } k$$

  - (S2) k verdrängt k_{alt} : betrachte neue Position für k_{alt}

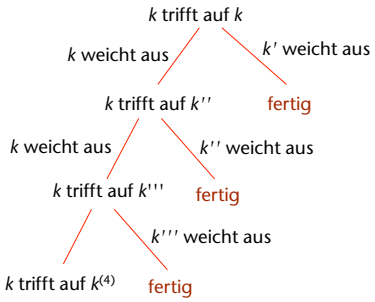
    $$h(k_{alt}) - s(j'+1, k_{alt}) \text{ für } k_{alt}$$

- Das allgemeine Verfahren: solange (S1) und (S2) auf einen belegten Slot stoßen, verfolge (S1) oder (S2) weiter
- Verwende Double-Hashing, d.h., wenn man in Sondierungsfolge bei Slot i steht, ist der nächste Slot (in der Sondierungsfolge) bei i - h'(k) mod m

Der Algorithmus zu Brent's Verfahren

- Falls (S2) einen freien Slot für k_{alt} liefert → verfolge (S2) ; sonst verfolge (S1) und wiederhole bei nächster Sondierung.

```
i = h( k )
while T[i] belegt:
    k_alt = T[i]
    i1 = (i-h'(k)) % m
    i2 = (i-h'(k_alt)) % m
    if T[i1] belegt and T[i2] frei:
        # verdränge k_alt
        T[i] = k
        k = k_alt
        i = i2
    else:
        # leave k_alt in its slot
        i = i1
T[i] = k
```



- Analyse (o. Bew.):
  - Erwartete Kosten für erfolglose Suche bleiben unverändert:

$$C'_n \approx \frac{1}{1-\alpha}$$

  - Erwartete Kosten für erfolgreiche Suche :

$$C_n \approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} + \cdots \quad < 2.5$$

Beispiel

- Hash-Funktionen: $h(k) = k \bmod 7$
  $h'(k) = 1 + (k \bmod 5)$
- Schlüsselfolge: 12, 53, 5, 15, 2, 15

0	1	2	3	4	5	6
				53	12	

- Nächster Key: $h(5) = 5 \rightarrow$ belegt $\rightarrow k_{alt} = 12$
- Betrachte also:

  $5 - h'(k) = 5 - h'(5) = 4$ belegt

  und $5 - h'(k_{alt}) = 5 - h'(12) = 3$ frei

  → 5 verdrängt 12 von seinem Platz