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 passieren, daß 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 heiße H universell, wenn gilt:

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

- Folgerung: x, $y \in U$ beliebig, H universell, $h \in H$ zufällig

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

- 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:

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

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 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 so gleichmäßig wie nur möglich in der Hash-Tabelle verteilt (auch bei *"malicious adversary"*).

Eine universelle Klasse von Hash-Funktionen

- Annahmen: $|U| = p$, mit Primzahl p und $U = \{0, \ldots, p\text{-}1\}$
  - seien $a \in \{1, \ldots, p\text{-}1\}$ und $b \in \{0, \ldots, p\text{-}1\}$, 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 $h_{4,4}(1) = h_{4,4}(4)$

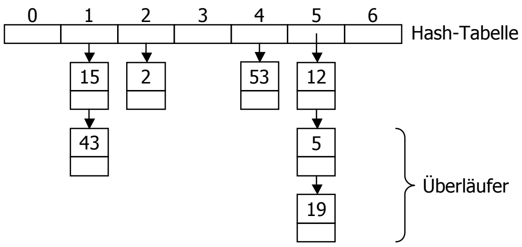
Möglichkeiten der Kollisionsbehandlung

- Die Behandlung von Kollisionen erfolgt bei verschiedenen Verfahren unterschiedlich
- Ein Datensatz mit Schlüssel *k* ist ein Überläufer, wenn der Slot *h*(*k*) schon durch einen anderen Satz 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, gefunden (Offene Hash-Verfahren)

Chaining

- Die Hash-Tabelle ist ein Array (Länge *m*) von Listen, jeder Slot wird durch eine Liste realisiert
- 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
- Reine Listenoperationen

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

  def search( key ):
    l = self.table[ h(key) ]
    for i in range( 0, len(l) ):
      if l[i].key == key:
        return l[i].value
    return None

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

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 Berechnung von h immer in $O(1)$
- Aufwand für Suchen, Einfügen, Löschen im Worst-Case immer in $O(m)$ bzw. $O(k)$
  - was uns also interessiert ist die Average-Case-Laufzeit
  - beim Löschen muß vorher Element (erfolgreich) gesucht werden
  - beim Einfügen muß vorher Element (erfolglos) gesucht werden
- Bestimme im Folgenden zwei Erwartungswerte, bezogen auf 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

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

$$C'_n = \alpha$$
$$C_n = 1 + \frac{\alpha}{2}$$

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

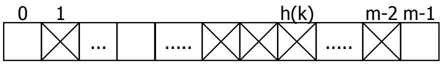
$$T(\text{erfolgloser Suche}) = \frac{n}{m} \in \frac{O(m)}{m} = O(1)$$

- Vorteile:
  - C_n und C'_n niedrig
  - $\alpha > 1$ möglich
  - für Sekundärspeicher geeignet
- Nachteile:
  - Zusätzlicher Speicherplatz für Zeiger,
  - Überläufer außerhalb der Hash-Tabelle

Offene Hash-Verfahren (*open adressing*)

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

  $(h(k) - 1) \mod m$



- allgemeiner: betrachte die Folge

  $(h(k) - j) \mod m \qquad j = 0, \ldots, m-1$
- noch allgemeiner: betrachte Sondierungsfolge

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

  für eine gegebene Funktion $s(j,k)$
- Problem: Entfernen von Keys → nur als "entfernt" markieren


```
# 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 Eintrage
    i = ( h(k) - s(j,k) ) % m
    while T[i].mark != HASH_FREE and T[i].key != k:
        j += 1
        i = ( h(k) - s(j,k) ) % m
    if T[i].key == k and T[i].mark == HASH_OCCUPIED:
        return T[i].item
    else:
        return None
```

```
# Füge Key k und Nutzdaten item in der Hash-Tabelle ein
def insert( self, k, item ):
    j = 0
    i = ( h(k) - s(j,k) ) % m
    while T[i].mark != HASH_FREE and T[i].key != k:
        j += 1
        i = ( h(k) - s(j,k) ) % m
    if T[i].key == k and T[i].mark == HASH_OCCUPIED:
        return Fehlermeldung    # Key doppelt eingefügt
    else:
        T[i].key = k
        T[i].item = item
```


Sondierungsfolgen (*probing sequences*)

- Beispiele für die Funktion *s* :

  $s(j,k) = j$ (lineares Sondieren)

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

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

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

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

  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:

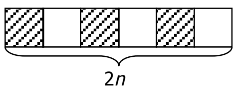
0	1	2	3	4	5	6
			5	53	12	

  - 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

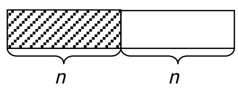
- Betrachte erfolglose Suche in zwei Extremen:
  - In beiden Fällen ist der Load-Factor = 1/2

1. Jeder 2-te Slot besetzt:
   mittlerer Aufwand =



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

2. 1 besetzter Cluster:
   mittlerer Aufwand =



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

- Sei $\alpha = \frac{n}{m}$ der Belegungsfaktor (*load factor*) mit $0 < \alpha < 1$
- Erfolgreiche Suche:

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

- Erfolglose Suche:

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

- Die Analyse [Knuth, 1962] war ein Meilenstein
- 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
- Funktion: $s(j,k) = (-1)^j \cdot \left\lceil \frac{j}{2} \right\rceil^2$
- Sondierungsfolge für k ist

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

ist Permutation, falls m Primzahl der Form $4i+3$ ist (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 zweite Hash-Funktion h'

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

- Sondierungsfolge für k: $h(k)$, $h(k)$-$h'(k)$, $h(k)$-$2h'(k)$, …
- Forderung: Sondierungsfolge muß Permutation der Hash-Adressen entsprechen
- Folgerung: $h'(k) \neq 0 \ \wedge \ h'(k) \nmid m$
- Beispiel:

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

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					

$h'(22) = 3$

0	1	2	3	4	5	6
	15				22	

$h'(1) = 2$

0	1	2	3	4	5	6
	15				22	1

$h'(29) = 5$

0	1	2	3	4	5	6
	15		29		22	1

$h'(26) = 2$

  - In diesem Beispiel genügen fast immer 1-2 Sondierschritte