6 Hashverfahren

6.1 Problemstellung

Effiziente Implementierung eines abstrakten Datentyps für Wörterbuchoperationen auf einer Menge von Datensätzen mit Suchschlüsseln $S \subset \mathcal{U}$ (Universum)

search (k) *insert (k)* *delete (k)*

Grundidee (idealisiert)

- Ablegen der Datensätze in Array (der *Hashtabelle*) einer festen Größe m
- Berechnung des Orts (Index in Hashtabelle) des Datensatzes d aus seinem Schlüssel

$\Rightarrow$ keine Vergleiche (im Idealfall)

$\Rightarrow$ alle Operationen in konstanter Zeit

 © 2002 Peter Thiemann

Beispiele

Compiler Symboltabelle

```
i     int    0x87C50FA4
j     int    0x87C50FA8
x     double 0x87C50FAC
name  String 0x87C50FB2
```

Umgebungsvariablen Liste von Name/Wert-Paaren

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

ausführbare Programme Suchpfad

```
PATH=/usr/local/fptools/bin:/usr/local/cvs/cvs-1.11.auth/bin/Linux-unknown:
     /usr/local/cvs/gcvs/bin/Linux-unknown/bin:/usr/local/cvsstat/bin:
     /usr/local/fptools/bin:/usr/local/cvs/cvs-1.11.auth/bin/Linux-unknown:
     /usr/local/cvs/gcvs/bin/Linux-unknown/bin:/usr/local/cvsstat/bin: ...
```

 © 2002 Peter Thiemann

Probleme von Hashverfahren

1. Größe der Hashtabelle $m \ll |\mathcal{U}|$
   Nur eine kleine Teilmenge $S \subset \mathcal{U}$ aller möglichen Schlüssel kommt vor

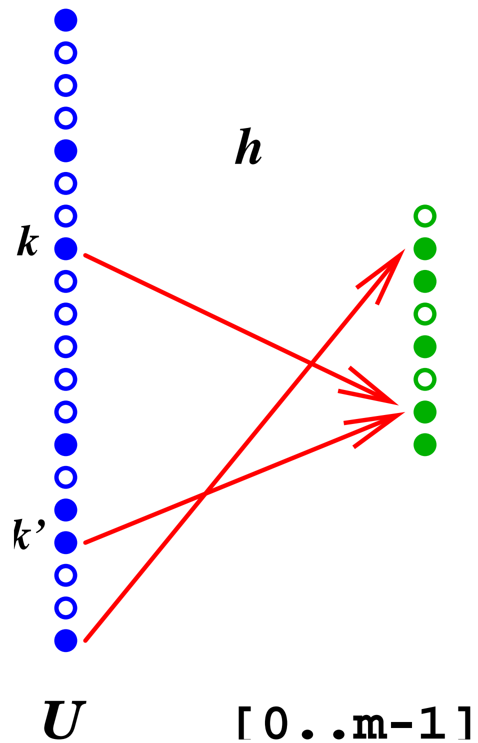
2. Berechnung der Adresse eines Datensatzes
   - Index hängt von m ab ($\Rightarrow$ rechne $\pmod m$)
   - Schlüssel sind keine ganzen Zahlen
   - Benötige Abbildung, die $\mathcal{U}$ möglichst gleichmäßig auf $[-2^{31}, 2^{31} - 1]$ verteilt

     In Java liefert `hashCode` eine solche Abbildung

```
public class Object {
  ⋮
  public int hashCode() { ... }
    ⋮
  }
```

 © 2002 Peter Thiemann

6.2 Terminologie



$\mathcal{U}$ **Universum**: Menge aus der die Schlüssel stammen

$S \subseteq \mathcal{U}$ Menge der gespeicherten Schlüssel

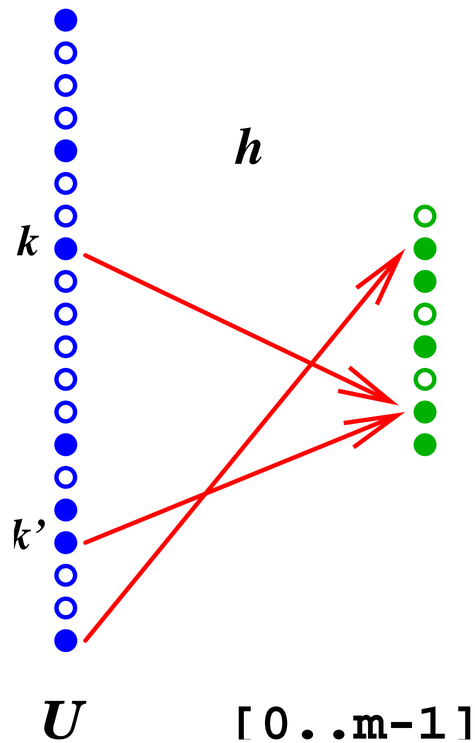
$[0..m-1]$ Indexbereich der Hashtabelle; m Größe der Hashtabelle

$h : \mathcal{U} \rightarrow [0..m-1]$ **Hashfunktion**

$h(k)$ **Hashadresse**

 © 2002 Peter Thiemann

Terminologie II



$k \neq k'$ **aber** $h(k) = h(k')$

- **Adresskollision** liegt vor
- k und k' sind **Synonyme** bzgl. h
- unvermeidlich: wenn $m < |\mathcal{U}|$, dann kann h nicht injektiv sein!

Beispiel für $\mathcal{U}$ Menge der Bezeichner in Java mit Länge ≤ 40 (nur 8-bit Zeichen)

$\Rightarrow$ $|\mathcal{U}| = 116 \cdot 126^{39} \approx 9.5 \cdot 10^{83}$

 © 2002 Peter Thiemann

Terminologie III

Hashverfahren besteht aus

1. einer möglichst guten Hashfunktion
2. einer Strategie zur Auflösung der Kollisionen

 © 2002 Peter Thiemann

Java Framework für Hashverfahren

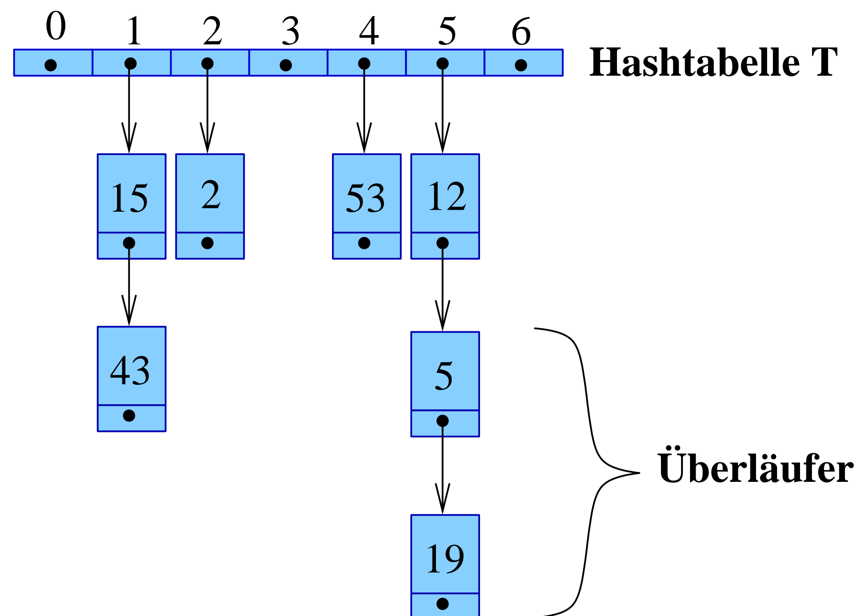
```
class TableEntry { Object key, value; }

abstract class HashTable {
    protected TableEntry[] table;
    protected int capacity;
    // Konstruktor
    HashTable (int capacity) {
        this.capacity = capacity;
        table = new TableEntry [capacity];
    }
    // die Hashfunktion
    protected int h (Object key) {
        return key.hashCode () % capacity;
    }
    // suche Element  mit Schlüssel key
    public abstract Object search (Object key);
    // füge Element mit Schlüssel key und Wert value ein, falls noch nicht vorhanden
    public abstract void insert (Object key, Object value);
    // entferne Element mit Schlüssel key, falls vorhanden
    public abstract void delete (Object key);
}
```

 © 2002 Peter Thiemann

6.3 Geschlossene Hashverfahren

- Kollisionsbehandlung durch verkettete Listen (**Überlauflisten**)
- jeder Eintrag der Hashtabelle ist lineare Liste von Datensätzen mit gleicher Hashadresse (*direkte Verkettung*)

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



© 2002 Peter Thiemann

Implementierung der Operationen

`search(k)`

- Berechne `h(k)`
- Sequentielle Suche nach k in Liste `T[h(k)]`

`insert(k)`

- `search(k)` (erfolglos)
- Einfügen von `k` in Liste `T[h(k)]`

`delete(k)`

- `search(k)` (erfolgreich)
- Entfernen von `k` aus Liste `T[h(k)]`

 © 2002 Peter Thiemann

Implementierung in Java

```
class ChainedTableEntry extends TableEntry { ChainedTableEntry next; }

class ChainedHashTable extends HashTable {
    ChainedHashTable (int capacity) {
        super (capacity);
    }

    // suche key in der Hashtabelle
    public Object search (Object key) {
        for (ChainedTableEntry p = (ChainedTableEntry) table [h(key)];
             p != null; p = p.next)
            if (p.key.equals(key))
                return p.value;

        return null;
    }
```

 © 2002 Peter Thiemann

```
// füge key mit Wert value in Hashtabelle ein
public void insert (Object key, Object value) {
    ChainedTableEntry entry = new ChainedTableEntry();
    entry.key = key;
    entry.value = value;

    // Hole den Tabelleneintrag für key
    int k = h (key);
    ChainedTableEntry p = (ChainedTableEntry) table [k];
    if (p == null) {
        table[k] = entry;
        return;
    }
    // Suche nach key
    while (! p.key.equals(key) && p.next != null)
        p = p.next;

    // Füge Eintrag ein, falls nicht vorhanden
    if (! p.key.equals(key))
        p.next = entry;
}
```

 © 2002 Peter Thiemann

```
    public void delete (Object key) {
        int k = h (key);
        ChainedTableEntry p = (ChainedTableEntry) table [k];
        table[k] = recDelete(p, key);
    }

    // entferne Element mit Schluessel key, falls vorhanden
    public ChainedTableEntry recDelete (ChainedTableEntry p, Object key) {
        //  recDelete gibt einen Zeiger auf den Beginn der Liste, auf die p
        // zeigt, zurueck, in der key entfernt wurde )
        if (p == null)
            return null;
        else if (p.key.equals(key))
            return p.next;
        else {
            p.next = recDelete(p.next, key);
            return p;
        }
    }
}
```

 © 2002 Peter Thiemann

Analyse: Grundbegriffe

Annahmen (*Uniform-Hashing*)

- bei zufälliger Wahl des Schlüssels k treten alle Hashadressen mit gleicher Wahrscheinlichkeit auf, d.h.

$$P(h(k) = j) = \frac{1}{m}$$

- unabhängig von Operation zu Operation

Definition: Belegungsgrad einer Hashtabelle Sei $n = |S|$, die Anzahl der Einträge in der Hashtabelle

$$\textbf{Belegungsgrad} \qquad \alpha = \frac{\text{Anzahl der Einträge}}{\text{Größe der Hashtabelle}} = \frac{n}{m}$$

Definition: Aufwand für Suche in Hashtabelle

C'_n Erwartungswert für die Anzahl betrachteter Einträge bei **erfolgloser Suche**

C_n Erwartungswert für die Anzahl betrachteter Einträge bei **erfolgreicher Suche**

 © 2002 Peter Thiemann

Analyse: Direkte Verkettung der Überläufer

- Nach Einfügen von n zufälligen Elementen ist der Erwartungswert für die Länge einer Überlaufliste gerade

$$\frac{n}{m} = \alpha$$

$\Rightarrow$

$$\begin{aligned} C'_n &= \tfrac{n}{m} = \alpha && \text{(erfolglose Suche)} \\ C_n &= \tfrac{1}{n} \textstyle\sum_{j=1}^{n} 1 + \tfrac{j-1}{m} = 1 + \tfrac{n-1}{2m} \approx 1 + \tfrac{\alpha}{2} && \text{(erfolgreiche Suche)} \end{aligned}$$

- Übersicht

α	C_n (erfolgreich)	C'_n (erfolglos)
0.50	1.250	0.50
0.90	1.450	0.90
0.95	1.475	0.95
1.00	1.500	1.00
2.00	2.000	2.00
3.00	2.500	3.00

 © 2002 Peter Thiemann

Diskussion: Verkettung der Überläufer

Vorteile:

- C_n und C'_n niedrig
- $\alpha > 1$ möglich
- echte Entfernungen
- für Sekundärspeicher geeignet

Nachteile:

- Zusätzlicher Speicherplatz für Zeiger
- Überläufer außerhalb der Tabelle

 © 2002 Peter Thiemann

6.4 Wahl der Hashfunktion

Kriterien

- leichte und schnelle Berechenbarkeit (Erwartung: $O(1)$)
- gleichmäßige Verteilung von $h(k)$

6.4.1 Divisions-Rest-Methode: $h(k) = k \bmod m$

Beispiele:

- m gerade, letztes Bit drückt Sachverhalt aus (z.B. 0 = weiblich, 1 = männlich)
  $\Rightarrow h(k)$ gerade $\Leftrightarrow k$ gerade
- $m = 2^i$
  $\Rightarrow h(k)$ liefert i niedrigsten Dualziffern von k

Regel: Wähle m prim mit $m \not| \, r^i \pm j$, $0 \leq j \leq r-1$, wobei r Basis der Darstellung

 © 2002 Peter Thiemann

6.4.2 Multiplikative Methode

- Wähle irrationale Zahl $\theta > 0$
- Setze $h(k) = m(k\theta - \lfloor k\theta \rfloor)$
- Operation $x - \lfloor x \rfloor =: x \bmod 1$ entspricht Abschneiden des ganzzahligen Anteils
- Wahl von m unkritisch, z.B. $m = 2^i$

$\Rightarrow$ Berechnung mit einer Ganzzahlmultiplikation und Verschieben möglich

$\Rightarrow$ Gute Verteilung

Beispiel:

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

$k = 123456 \qquad m = 10000$

$$\begin{aligned} h(k) &= \lfloor 10000(123456 \cdot 0.61803\ldots \bmod 1) \rfloor \\ &= \lfloor 10000(76300,0041151\ldots \bmod 1) \rfloor \\ &= \lfloor 41.151\ldots \rfloor = 41 \end{aligned}$$

 © 2002 Peter Thiemann

6.4.3 Universelles Hasing

Problem: zu jeder Hashfunktion h gibt es $S \subseteq \mathcal{U}$ mit vielen Kollisionen

Idee des universellen Hashing: Wähle Hashfunktion $h : \mathcal{U} \to [0, m-1]$ zufällig bei Konstruktion der Hashtabelle aus einer endlichen Menge $\mathcal{H}$ von Hashfunktionen

Definition: $\mathcal{H}$ heißt **universell**, wenn für beliebige $x \neq y \in \mathcal{U}$ gilt:

$$\frac{|\{h \in \mathcal{H} \mid h(x) = h(y)\}|}{|H|} \leq \frac{1}{m}$$

Folgerung: $x \neq y \in \mathcal{U}$ beliebig, $\mathcal{H}$ universell, $h \in \mathcal{H}$ zufällig

$$P_{\mathcal{H}}\left(h(x) = h(y)\right) \leq \frac{1}{m}$$

 © 2002 Peter Thiemann

Definition Kollisionszählfunktion

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

Erweiterung von δ auf Mengen $S \subseteq \mathcal{U}$ und $\mathcal{G} \subseteq \mathcal{H}$:

$$\begin{aligned} \delta(x,S,h) &= \sum_{k \in S} \delta(x,k,h) \\ \delta(x,y,\mathcal{G}) &= \sum_{h \in \mathcal{G}} \delta(x,y,h) \end{aligned}$$

Folgerung: $\mathcal{H}$ ist universell, genau dann wenn für alle $x, y \in \mathcal{U}$ gilt

$$\delta(x,y,\mathcal{H}) \leq \frac{|\mathcal{H}|}{m}$$

 © 2002 Peter Thiemann

Beispiel: Universelles Hashing Hashtabelle T der Größe 3, $|\mathcal{U}| = 5$

Betrachte die Menge $\mathcal{H}$ mit folgenden 20 Funktionen:

$$\begin{array}{cccc}
x+0 & 2x+0 & 3x+0 & 4x+0 \\
x+1 & 2x+1 & 3x+1 & 4x+1 \\
x+2 & 2x+2 & 3x+2 & 4x+2 \\
x+3 & 2x+3 & 3x+3 & 4x+3 \\
x+4 & 2x+4 & 3x+4 & 4x+4
\end{array}$$

(jeweils $\pmod 5 \pmod 3$)

Die Schlüssel 1 und 4 kollidieren genau für folgende Funktionen:

$$\begin{array}{rcl}
(1 \cdot 1 + 0) \bmod 5 \bmod 3 = & 1 & = (1 \cdot 4 + 0) \bmod 5 \bmod 3 \\
(1 \cdot 1 + 4) \bmod 5 \bmod 3 = & 0 & = (1 \cdot 4 + 4) \bmod 5 \bmod 3 \\
(4 \cdot 1 + 0) \bmod 5 \bmod 3 = & 1 & = (4 \cdot 4 + 0) \bmod 5 \bmod 3 \\
(4 \cdot 1 + 4) \bmod 5 \bmod 3 = & 0 & = (4 \cdot 4 + 4) \bmod 5 \bmod 3
\end{array}$$

© 2002 Peter Thiemann

Eine universelle Klasse von Hashfunktionen

Annahmen

- Sei p Primzahl, $\mathcal{U} = \{0, \ldots, p-1\}$, $p > m$
- Für $1 \leq a < p$ und $0 \leq b < p$ definiere

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

Satz Die Menge

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

ist eine universelle Klasse von Hashfunktionen.

Folgerung Erhalte gute Hashfunktion durch

- Wähle Primzahl $p \geq |\mathcal{U}|$ und setze $m = 2^i$
- Wähle zufällig $h_{a,b} \in \mathcal{H}$ aus.

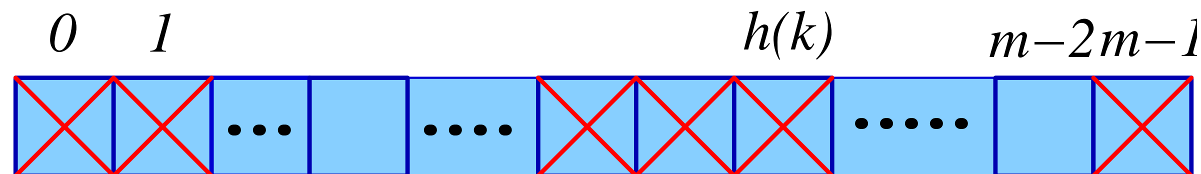
 © 2002 Peter Thiemann

6.5 Offene Hashverfahren

Ansatz

- **Nie** mehr als m Elemente in Hashtabelle speicherbar
- Wenn $T[h(k)]$ bereits belegt, suche anderen Platz für k nach fester Strategie
- **Problem:** Entfernen von Elementen ⇒ durch Markierung "entfernt"

Beispiel (lineares Sondieren)



- Falls $T[h(k)]$ belegt, versuche $T[h(k) - 1 \bmod m]$
- Falls $T[h(k) - 1 \bmod m]$ belegt, versuche $T[h(k) - 2 \bmod m]$
- Falls $T[h(k) - 2 \bmod m]$ belegt, versuche $T[h(k) - 3 \bmod m]$
- usw.

 © 2002 Peter Thiemann

6.5.1 Sondierungsfolgen

Allgemein: für $j = 0, \ldots, m-1$ betrachte der Reihe nach

$$h(k) - s(j,k) \bmod m$$

Dabei ist $s(j,k)$ die *Sondierungsfolge*.

Beispiele für Sondierungsfolgen

- lineares Sondieren
  $s(j,k) = j$
- quadratisches Sondieren
  $s(j,k) = (-1)^j + \lceil j/2 \rceil^2$
- Double Hashing; $h' \neq h$ ist *sekundäre Hashfunktion*
  $s(j,k) = jh'(k)$

 © 2002 Peter Thiemann

Anforderung an Sondierungsfolgen

Die Abbildung

$$j \mapsto s(j,k)$$

sollte für jedes k eine Permutation von $\{0,\ldots,m-1\}$ sein.

Beispiel: Für quadratisches Sondieren mit $m = 7$ ist das der Fall

j	0	1	2	3	4	5	6
$s(j,k)$	0	-1	1	-4	4	-9	9
$s(j,k) \bmod m$	0	6	1	3	4	5	2

Allgemein: quadratisches Sondieren liefert eine Permutation, falls m ist Primzahl der Form $4i+3$.

 © 2002 Peter Thiemann

6.5.2 Java Implementierung

```
class OpenHashTable extends HashTable {
    private int [] tag;

    static final int EMPTY     = 0;   // Frei )
    static final int OCCUPIED  = 1;   // Belegt )
    static final int DELETED   = 2;   // Entfernt )

    //  Konstruktor
    OpenHashTable (int capacity) {
        super(capacity);
        tag = new  int [capacity];      // alle Einträge == 0 == EMPTY
    }

    //  Funktion s für Sondierungsfolge
    protected int s (int j, Object key) {
        // quadratisches Sondieren
        int f = ((j + 1) / 2);
        if (j % 2 == 0)
            return  f * f;
        else
            return -f * f;
    }
```

 © 2002 Peter Thiemann

```
public int searchIndex (Object key) {
    // sucht in der Hashtabelle nach Index für Schlüssel key
    int h0 = h(key);
    int firstDeleted = -1;
    int i = h0;
    int j = 1;   // nächster Index der Sondierungsfolge

    while (true) {
        if (tag[i] == EMPTY)
            if (firstDeleted < 0)
                return i;
            else
                return firstDeleted;
        if (tag[i] == DELETED && firstDeleted < 0)
            firstDeleted = i;
        if (tag[i] == OCCUPIED && key.equals (table[i].key))
            return i;
        i = (h0 - s(j++, key)) % capacity;
        if (i < 0)
            i += capacity;
    }
}
```

 © 2002 Peter Thiemann

```
    public Object search (Object key) {
        // durchsucht Hashtabelle nach Eintrag mit Schlüssel key
        int i = searchIndex (key);
        if (tag[i] == OCCUPIED)
            return table[i].value;
        return null;
    }

    public void insert (Object key, Object value) {
        // fügt einen Eintrag mit Schlüssel key und Wert value ein
        int i = searchIndex (key);
        table[i] = new TableEntry();
        table[i].key = key; table[i].value = value;
        tag[i] = OCCUPIED;
    }

    public void delete (Object key) {
        // entfernt Eintrag mit Schlüssel key aus der Hashtabelle
        int i = searchIndex(key);
        if (tag[i] == OCCUPIED)
            tag[i] = DELETED;
    }
}
```

 © 2002 Peter Thiemann

6.5.3 Lineares Sondieren

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

Sondierungsfolge für k

$$h(k), h(k)-1, \ldots, 0, m-1, \ldots, h(k)+1$$

Problem: primäre Häufung ("primary clustering")

0	1	2	3	4	5	6
			5	53	12	

$$P(\text{nächstes Objekt landet an Position 2}) = \frac{4}{7}$$

$$P(\text{nächstes Objekt landet an Position 1}) = \frac{1}{7}$$

$\Rightarrow$ Lange Ketten werden mit größerer Wahrscheinlichkeit verlängert als kurze.

© 2002 Peter Thiemann

Effizienz des linearen Sondierens

Erfolgreiche Suche:

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

Erfolglose Suche:

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

α	C_n (erfolgreich)	C_n' (erfolglos)
0.50	1.5	2.5
0.90	5.5	50.5
0.95	10.5	200.5
1.00	—	—

$\Rightarrow$ Effizienz des linearen Sondierens verschlechtert sich drastisch, sobald sich der Belegungsfaktor α dem Wert 1 nähert.

 © 2002 Peter Thiemann

6.5.4 Quadratisches Sondieren

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

Sondierungsfolge für k:

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

Permutation, falls $m = 4l + 3$ eine Primzahl ist.

Problem: sekundäre Häufung, d.h. zwei **Synonyme** k und k' durchlaufen **stets dieselbe Sondierungsfolge**.

 © 2002 Peter Thiemann

Effizienz des quadratischen Sondierens

Erfolgreiche Suche:

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

Erfolglose Suche:

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

α	C_n (erfolgreich)	C'_n (erfolglos)
0.50	1.44	2.19
0.90	2.85	11.40
0.95	3.52	22.05
1.00	—	—

 © 2002 Peter Thiemann

6.5.5 Uniformes Sondieren

$$s(j,k) = \pi_k(j)$$

- π_k eine der $m!$ Permutationen von $\{0,\ldots,m-1\}$
- hängt nur von k ab
- wenn k gleichverteilt, dann soll auch π_k gleichverteilt sein

$$C_n \approx \frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right) \qquad C'_n \leq \frac{1}{1-\alpha}$$

α	C_n (erfolgreich)	C'_n (erfolglos)
0.50	1.39	2
0.90	2.56	10
0.95	3.15	20
1.00	—	—

 © 2002 Peter Thiemann

6.5.6 Zufälliges Sondieren

Realisierung von uniformem Sondieren sehr aufwendig.

Alternative: Zufälliges Sondieren

$$s(j,k) = \text{von } k \text{ abhängige Folge von Zufallszahlen}$$

$s(j,k) = s(j',k)$ möglich, aber unwahrscheinlich

 © 2002 Peter Thiemann

6.5.7 Double Hashing

Idee: Wähle zweite Hashfunktion h'

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

Sondierungsfolge für k:

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

Forderung: Sondierungsfolge soll **Permutation** der Hashadressen entsprechen.

Notwendige Voraussetzungen dafür:

- $h'(k) \neq 0$
- $h'(k)$ teilerfremd zu m

Beispiel: $h'(k) = 1 + (k \bmod (m - 2))$ (falls m Primzahl)

 © 2002 Peter Thiemann

Beispiel: Double Hashing

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

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

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

Double Hashing ist ...

- genauso effizient wie uniformes Sondieren.
- leichter zu implementieren.

 © 2002 Peter Thiemann

6.5.8 Verbesserung der erfolgreichen Suche — Motivation

Hashtabelle der Größe 11, Double Hashing mit

$$\begin{aligned} h(k) &= k \bmod 11 \qquad \text{und} \\ h'(k) &= 1 + (k \bmod (11-2)) = 1 + (k \bmod 9) \end{aligned}$$

Bereits eingefügt: 22, 10, 37, 47, 17

Einfügen von 6: $h(6) = 6,\ h'(6) = 1 + 6 = 7$

0	1	2	3	4	5	6	7	8	9	10
22			47	37		17				10

Einfügen von 30: $h(30) = 8,\ h'(30) = 1 + 3 = 4$

0	1	2	3	4	5	6	7	8	9	10
22			47	37		6		17		10

 © 2002 Peter Thiemann

Verbesserung der erfolgreichen Suche

Situation beim Einfügen von k an Stelle i:

$T[i]$ bereits belegt durch k', d.h.: $i = h(k) - s(j,k) = h(k') - s(j',k')$

Standardstrategie: füge k an Stelle $i' = h(k) - s(j+1,k)$ ein

Neue Strategie: Suche freien Platz für k oder k'

Dafür zwei Möglichkeiten

(M1) k' bleibt in $T[i]$ $\Rightarrow$ betrachte Position $i_1 = h(k) - s(j+1,k)$ für k

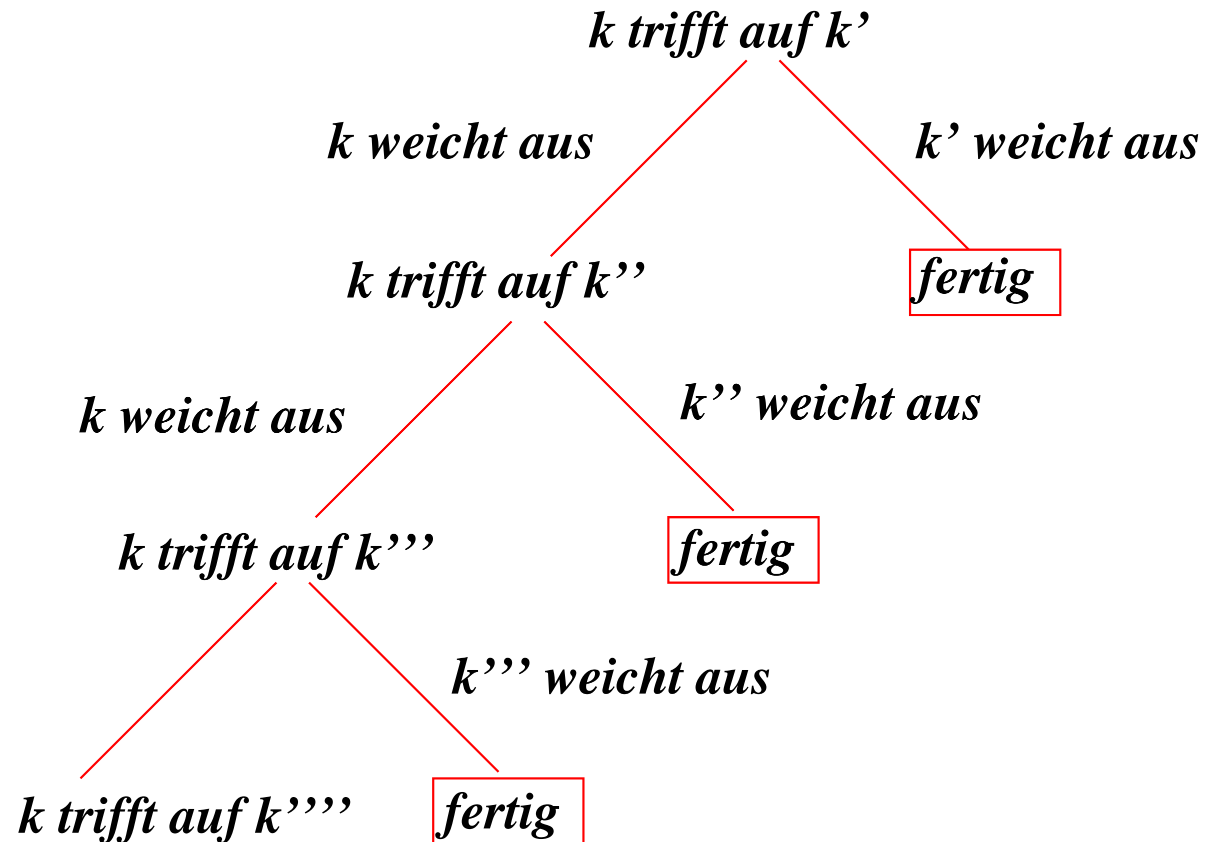
(M2) k verdrängt k' $\Rightarrow$ betrachte Position $i_2 = h(k') - s(j'+1,k')$ für k'

Verfahren:

- Falls $T[i_1]$ leer $\Rightarrow T[i_1] = k$
- Falls $T[i_2]$ leer $\Rightarrow T[i] = k$ und $T[i_2] = k'$
- Falls beide belegt, weiter mit (M1) oder (M2)

 © 2002 Peter Thiemann

Brent's Verfahren: verfolge nur (M1)



Binärbaum Sondieren: verfolge (M1) und (M2)

 © 2002 Peter Thiemann

Problem mit Brent/Binärbaum Sondieren

Angenommen k' wird durch k verdrängt

Was ist der nächste Platz in Sondierungsfolge für k'?

Ausweichen von k' einfach, wenn für alle $1 \leq j \leq m-1$ gilt:

$$s(j,k') - s(j-1,k') = s(1,k')$$

Das gilt beispielsweise für lineares Sondieren und Double Hashing.

$$\begin{aligned} C_n^{\textit{Brent}} &\approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} + \cdots < 2.5 \\ C'_n &\approx \frac{1}{1-\alpha} \end{aligned}$$

$$C_n^{\textit{Binärbaum}} \approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} + \cdots < 2.2$$

 © 2002 Peter Thiemann

Beispiel

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

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

Schlüsselfolge: 12, 53, 5, 15, 2, 19

0	1	2	3	4	5	6
				53	12	

$h(5) = 5$ belegt mit $k' = 12$

Betrachte:

- $h'(5) = 1 \Rightarrow h(5) - 1 \cdot h'(5) = 5 - 1 = 4$ belegt!
- $h'(12) = 3 \Rightarrow h(12) - 1 \cdot h'(12) = 5 - 3 = 2$ frei
- $\Rightarrow$ 5 verdrängt 12 von seinem Platz

 © 2002 Peter Thiemann

6.5.9 Verbesserung der erfolglosen Suche (Ordered Hashing)

Idee: Invariante

Alle Schlüssel in der Sondierungsfolge vor k sind kleiner als k

- Bei Suche nach k:
  treffe auf $k' > k$ in Sondierungsfolge: $\Rightarrow$ Suche erfolglos!
- Einfügen:
  kleinere Schlüssel verdrängen größere Schlüssel
- Probleme:
  - Verdrängungsprozess kann "Kettenreaktion" auslösen
  - k' von k verdrängt: Position von k' in Sondierungsfolge?

$\Rightarrow$ Für alle $1 \leq j \leq m$ muss gelten:

$$s(j,k) - s(j-1,k) = s(1,k)$$

 © 2002 Peter Thiemann

Ordered Hashing: Suche

```
search (k)
  i = h(k);
  while (table[i] nicht frei && table[i].key < k)
    i = (i - s(1,k)) % m;
  if (table[i] belegt && table[i].key == k)
    return table[i].value;
  else
    return null;
```

 © 2002 Peter Thiemann

Ordered Hashing: Einfügen

```
insert (k)
  i = h(k);
  while (table[i] nicht frei && table[i].key != k) {
    if (k < table[i].key)
      if (table[i] entfernt)
        break;
      else      // k verdrängt table[i].key
        swap (table[i].key, k);
    i = (i - s(1,k)) mod m;
  }
  if (table[i] ist nicht belegt)
    trage k in table[i].key ein
```

 © 2002 Peter Thiemann