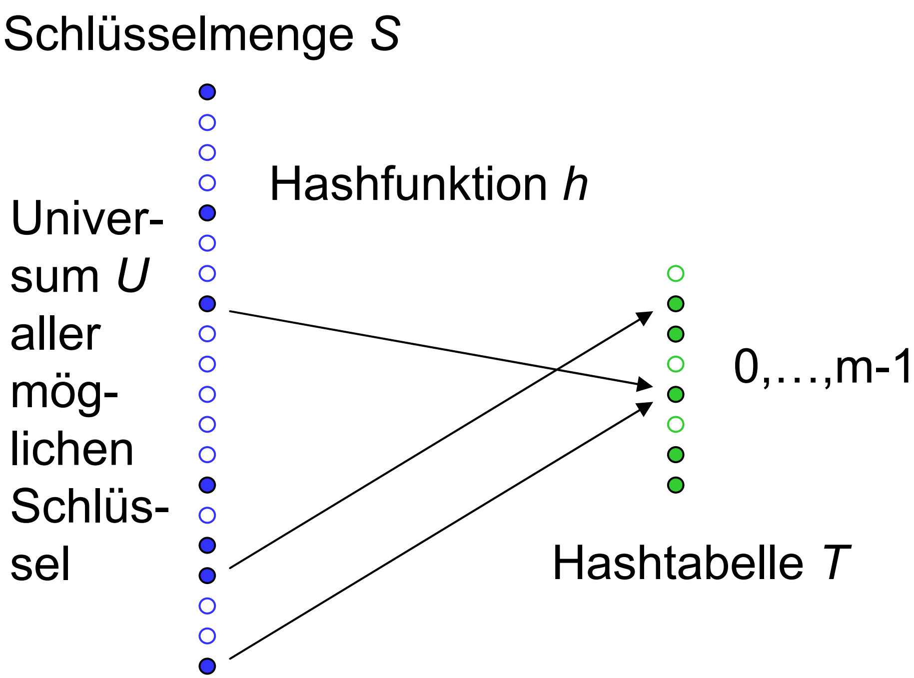
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(13 – Offenes Hashing)

Prof. Dr. Susanne Albers

Hashing: Allgemeiner Rahmen



$h(s)$ = Hashadresse

$h(s) = h(s')$ $\Leftrightarrow$ s und s' sind Synonyme bzgl. h

Adresskollision

Möglichkeiten der Kollisionsbehandlung

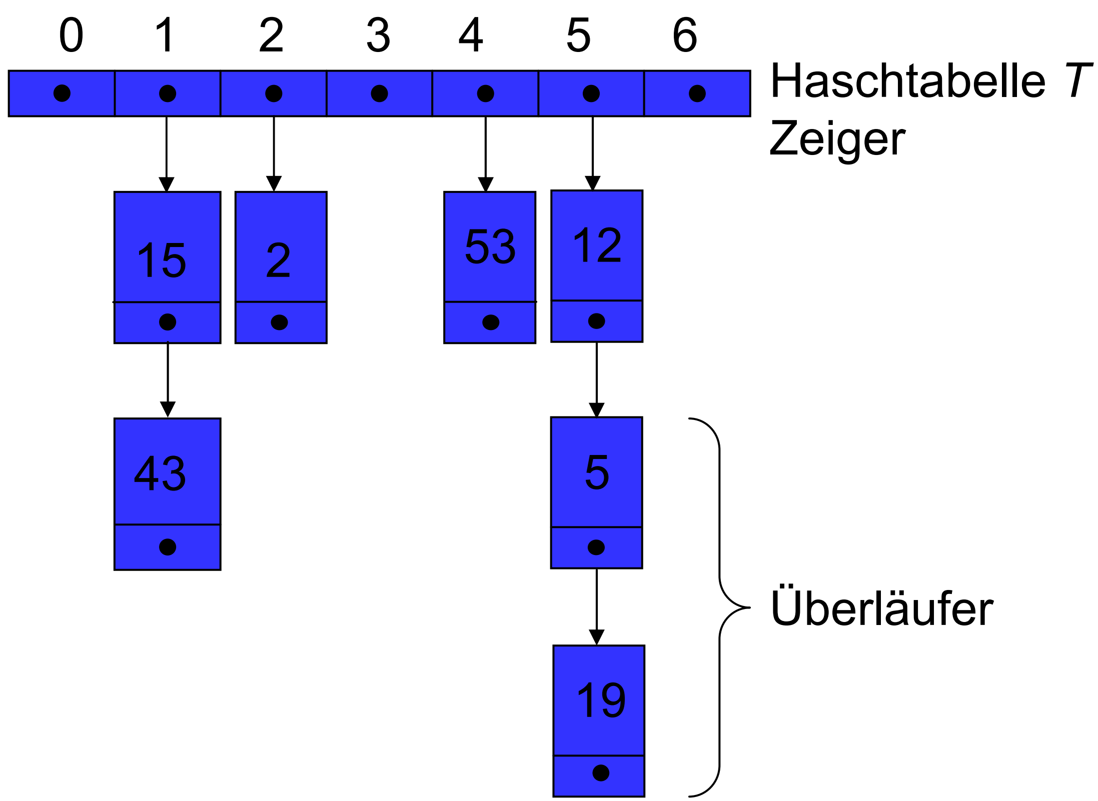
Kollisionsbehandlung:

- Die Behandlung von Kollisionen erfolgt bei verschiedenen Verfahren unterschiedlich.
- Ein Datensatz mit Schlüssel s ist ein Überläufer, wenn der Behälter $h(s)$ schon durch einen anderen Satz belegt ist.
- Wie kann mit Überläufern verfahren werden?
  1. Behälter werden durch verkettete Listen realisiert. Überläufer werden in diesen Listen abgespeichert.
  Chaining (Hashing mit Verkettung der Überläufer)
  2. Überläufer werden in noch freien anderen Behältern abgespeichert. Diese werden beim Speichern und Suchen durch sogenanntes Sondieren gefunden.
  Open Addressing (Offene Hashverfahren)

Hashing mit Verkettung der Überläufer

Schlüssel werden in Überlauflisten gespeichert

$h(k) = k \bmod 7$



Diese Art der Verkettung wird auch als direkte Verkettung bezeichnet.

Offene Hashverfahren

Idee:

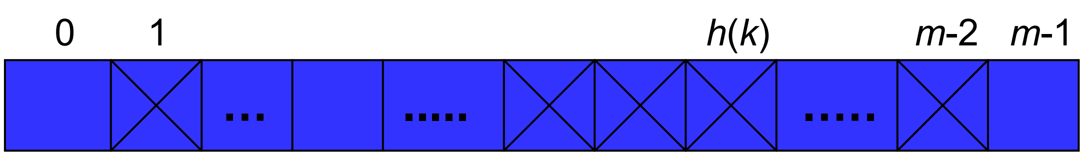
Unterbringung der Überläufer an freien (“offenen”) Plätzen in Hashtabelle

Falls $T[h(k)]$ belegt, suche anderen Platz für k nach fester Regel

Beispiel:

Betrachte Eintrag mit nächst kleinerem Index:

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



Allgemeiner:

Betrachte die Folge

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

$j = 0,\ldots,m-1$

Sondierungsfolgen

Noch allgemeiner:

Betrachte Sondierungsfolge

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

$j = 0,\ldots,m-1$, für eine gegebene Funktion $s(j,k)$

Beispiele für die Funktion

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

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

$s(j,k) = j * h'(k)$ (Double Hashing)

Sondierungsfolgen

Eigenschaften von $s(j,k)$

Folge

$(h(k) - s(0,k)) \bmod m,$
$(h(k) - s(1,k)) \bmod m,$

$(h(k) - s(m-2,k)) \bmod m,$
$(h(k) - s(m-1,k)) \bmod m$

sollte eine Permutation von $0,\ldots,m-1$ liefern.

Beispiel: Quadratisches Sondieren

0	1	2	3	4	5	6

$h(11) = 4$

$s(j,k) = -1,1,-4,4,-9,9$

Kritisch:

Entfernen von Sätzen → als entfernt markieren

(Einfügen von 4, 18, 25, Löschen 4, Suche 18, 25)

Offene Hashverfahren

```
class OpenHashTable extends HashTable {
    // in HashTable: TableEntry [] T;
    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];
        for (int i = 0; i < capacity; i++) {
            tag[i] = EMPTY;
        }
    }

    // Die Hashfunktion
    protected int h (Object key) {...}

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

Offene Hashverfahren - Suchen

```
public int searchIndex (Object key) {
    /* sucht in der Hashtabelle nach Eintrag mit Schluessel key und
       liefert den zugehoerigen Index oder -1 zurueck */
    int i = h(key);
    int j = 1;          // naechster Index der Sondierungsfolge

    while (tag[i] != EMPTY &&!key.equals(T[i].key)){
        // Naechster Eintr. in Sondierungsfolge
        i = (h(key) - s(j++, key)) % capacity;
        if (i < 0)
            i = i + capacity;
    }

    if (key.equals(T[i].key) && tag[i] == OCCUPIED)
        return i;
    else
        return -1;
}

public Object search (Object key) {
    /* sucht in der Hashtabelle nach Eintrag mit Schluessel key und liefert
       den zugehoerigen Wert oder null zurueck */
    int i = searchIndex (key);
    if (i >= 0)
        return T[i].value;
    else
        return null;
}
```

Offene Hashverfahren - Einfügen

```
public void insert (Object key, Object value) {
    // fuegt einen Eintrag mit Schluessel key und Wert value ein
    int j = 1;      // naechster Index der Sondierungsfolge
    int i = h(key);

    while (tag[i] == OCCUPIED) {
        i = (h(key) - s(j++, key)) % capacity;
        if (i < 0)
            i = i + capacity;
    }

    T[i] = new TableEntry(key, value);
    tag[i] = OCCUPIED;
}
```

Offene Hashverfahren - Entfernen

```
public void delete (Object key) {
    // entfernt Eintrag mit Schluessel key aus der Hashtabelle

    int i = searchIndex(key);

    if (i >= 0) {
        // Suche erfolgreich
        tag[i] = DELETED;
     }
}
```

Test-Programm

```
public class OpenHashingTest {
    public static void main(String args[]) {
        Integer[] t= new Integer[args.length];
        for (int i = 0; i < args.length; i++)
            t[i] = Integer.valueOf(args[i]);
        OpenHashTable h = new OpenHashTable (7);
        for (int i = 0; i <= t.length - 1; i++) {
            h.insert(t[i], null);#
            h.printTable ();
        }
        h.delete(t[0]); h.delete(t[1]);
        h.delete(t[6]); h.printTable();
    }
}
```

Aufruf:

```
java OpenHashingTest 12 53 5 15 2 19 43
```

Ausgabe (Quadratisches Sondieren):

```
[ ]  [ ]  [ ]  [ ]  [ ]  (12) [ ]
[ ]  [ ]  [ ]  [ ]  (53) (12) [ ]
[ ]  [ ]  [ ]  [ ]  (53) (12) (5)
[ ]  (15) [ ]  [ ]  (53) (12) (5)
[ ]  (15) (2)  [ ]  (53) (12) (5)
(19) (15) (2)  [ ]  (53) (12) (5)
(19) (15) (2)  (43) (53) (12) (5)

(19) (15) (2)  {43} {53} {12} (5)
```

Sondierungsfolgen - 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”)

0	1	2	3	4	5	6
			5	53	12	

Pr (nächstes Objekt landet an Position 2) = 4/7

Pr (nächstes Objekt landet an Position 1) = 1/7

Lange Ketten werden mit größerer Wahrscheinlichkeit verlängert als kurze.

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	-	-

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

Quadratisches Sondieren

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

Sondierungsfolge für *k*:

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

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

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

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	-	-

Uniformes Sondieren

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

π_k eine der m! Permutationen von {0,...,m-1}
- hängt nur von k ab
- gleichwahrscheinlich für jede Permutation

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

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

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

Zufälliges Sondieren

Realisierung von uniformem Sondieren sehr aufwändig.

Alternative:

Zufälliges Sondieren

$s(j,k)$ = von k abhängige Zufallszahl

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

Double Hashing

Idee: Wähle zweite Hashfunktion h'

$$s(j,k) = j * h'(k)$$

Sondierungsfolge für k:

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

Forderung:

Sondierungsfolge muss Permutation der Hashadressen entsprechen.

Folgerung:

$h'(k) \neq 0$ und $h'(k)$ kein Teiler von m, d.h. $h'(k)$ teilt m nicht.

Beispiel:

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

Beispiel

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

$h'(26) = 2$

In diesem Beispiel genügt fast immer einfaches Sondieren.

- Double Hashing ist genauso effizient wie uniformes Sondieren.
- Double Hashing ist leichter zu implementieren.

Verbesserung der erfolgreichen Suche - Motivation

Hashtabelle der Größe 11, Double Hashing mit

$h(k) = k \bmod 11$ und

$h'(k) = 1 + (k \bmod (11 - 2)) = 1 + (k \bmod 9)$

Bereits eingefügt: 22, 10, 37, 47, 17

Noch einzufügen: 6 und 30

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

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

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

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

Verbesserung der erfolgreichen Suche

Allgemein:

Einfügen:

- k trifft in $T[i]$ auf k_{alt}, d.h. $i = h(k) - s(j,k) = h(k_{alt}) - s(j',k_{alt})$
- k_{alt} bereits in $T[i]$ gespeichert

Idee:

Suche freien Platz für k oder k_{alt}

Zwei Möglichkeiten:

(M1) k_{alt} bleibt in $T[i]$
betrachte neue Position

$$h(k) - s(j+1,k) \text{ für } k$$

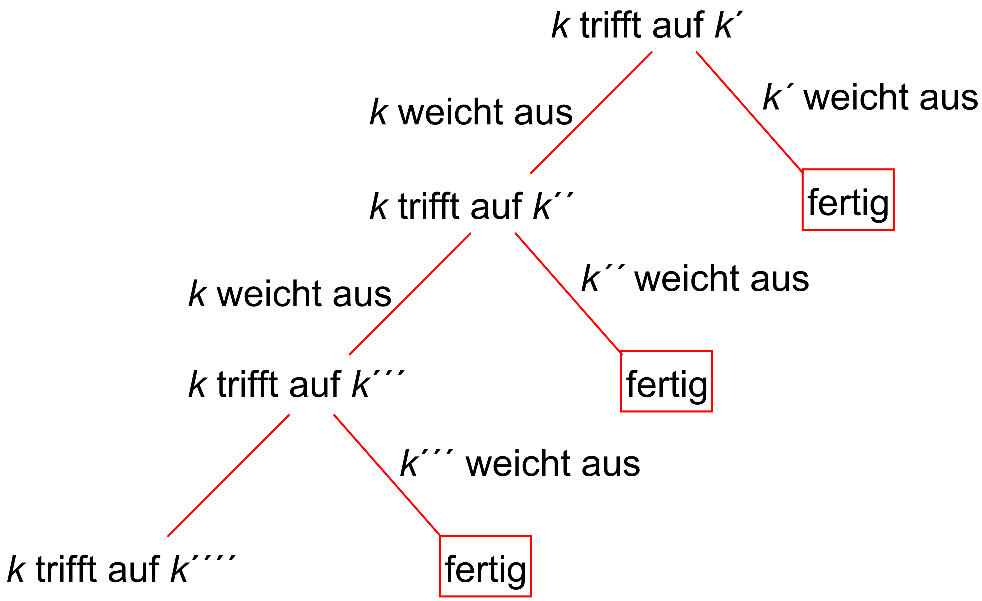
(M2) k verdrängt k_{alt}
betrachte neue Position

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

if (M1) or (M2) trifft auf einen freien Platz
then trage entsprechenden Schlüssel ein
fertig
else verfolge (M1) oder (M2) weiter

Verbesserung der erfolgreichen Suche

Brent's Verfahren: verfolge nur (M1)



Binärbaum Sondieren: verfolge (M1) und (M2)

Verbesserung der erfolgreichen Suche

Problem: k_{alt} von k verdrängt:

→ nächster Platz in Sondierungsfolge für k_{alt}?

Ausweichen von k_{alt} einfach, wenn gilt:

$$s(j, k_{alt}) - s(j-1, k_{alt}) = s(1, k_{alt})$$

für alle $1 \le j \le m-1$.

Das gilt beispielsweise für lineares Sondieren und double Hashing.

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

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

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

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 $k' = 12$

Betrachte:

h´(k) = 1 → h(5) -1 * h´(5)

→ 5 verdrängt 12 von seinem Platz

Verbesserung der erfolglosen Suche

Suche nach *k*:

k´>k in Sondierungsfolge: → Suche erfolglos

Einfügen:

kleinere Schlüssel verdrängen größere Schlüssel

Invariante:

Alle Schlüssel in der Sondierungsfolge vor *k* sind kleiner als *k*
(aber nicht notwendigerweise aufsteigend sortiert)

Probleme:

- Verdrängungsprozess kann “Kettenreaktion” auslösen
- *k´* von *k* verdrängt: Position von *k´* in Sondierungsfolge?

→ Es muss gelten:

$$s(j,k) - s(j-1,k) = s(1,k), \quad 1 \le j \le m$$

Ordered Hashing

Suchen

Input: Schlüssel k

Output: Information zu Datensatz mit Schlüssel k oder *null*

Beginne bei $i \leftarrow h(k)$

while $T[i]$ nicht frei and $T[i].k < k$ do

    $i \leftarrow (i - s(1,k)) \bmod m$

  end while;

if $T[i]$ belegt and $T[i].k = k$

    then Suche erfolgreich

    else Suche erfolglos

Ordered Hashing

Einfügen

Input: Schlüssel k

```
Beginne bei i ← h(k)
while T[i] nicht frei and T[i] .k ≠ k do
    if k < T[i].k
        then if T[i] ist entfernt
            then  exit while-loop
            else // k verdrängt T[i].k
                vertausche T[i].k mit k
    i = (i – s(1,k)) mod m
end while;
if T[i] ist nicht belegt
    then trage k bei T[i] ein
```