# Algorithmen und Datenstrukturen (Th. Ottmann und P. Widmayer)

## Folien: Offene Hashverfahren

**Autor: Sven Schuierer**

Institut für Informatik
Georges-Köhler-Allee
Albert-Ludwigs-Universität Freiburg

# Offene Hashverfahren

Idee:

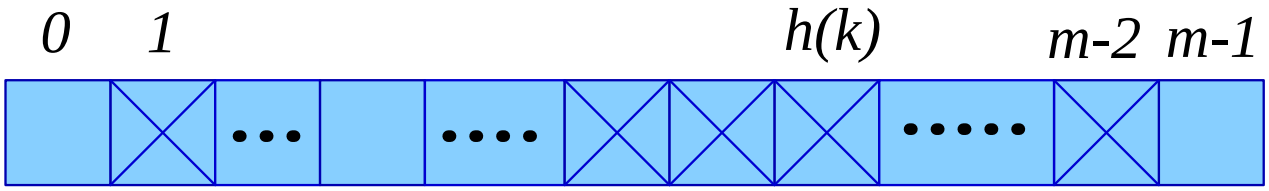
Unterbringung der Überläufer an freien (“offenen”) Plätzen in Hashtabelle

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 $s(j,k)$:

$$s(j,k) = j \qquad \text{(lineares Sondieren)}$$

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

$$s(j,k) = j \cdot h'(k) \qquad \text{(Double Hashing)}$$

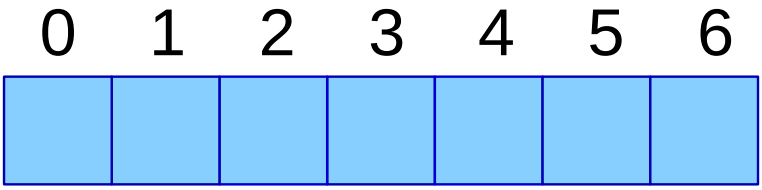
## Sondierungsfolgen

**Eigenschaften** von $s(j,k)$
Folge:

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

$\Rightarrow$ Permutation von $0, \ldots, m-1$

**Beispiel**: Quadratisches Sondieren




$h(11) = 4$

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

**Kritisch**:
Entfernen von Sätzen $\Rightarrow$ 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
     Schlüssel key und liefert den zugehörigen
     Index oder -1 zurück */
  int i = h(key);
  int j = 1; /* nächster Index der
                Sondierungsfolge */

  while (tag[i] != EMPTY && !
         key.equals(T[i].key)) {
    /* Untersuche nächsten 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
     Schlüssel key und liefert den zugehörigen
     Wert oder null zurück */
  int i = searchIndex (key);

  if (i >= 0) return T[i].value;
  if (i < 0)  return null;
}
```

# Offene Hashverfahren—Einfügen

```
public void insert (Object key, Object
  value) {
  /* fügt einen Eintrag mit Schlüssel key
     und Wert value ein */

  int j = 1; /* nächster Index der
                Sondierungs- folge */
  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 Schlüssel 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[]){
    int vec[] = { 12, 53, 5, 15, 2, 19, 43 };
    if (args.length >= 7) {
      vec = new int [args.length];
      for (int j = 0 ; j < args.length ; j++){
        vec [j] =
        Integer.valueOf(args[j]).intValue();
      }
    }
    Integer[] t = new Integer[vec.length];
    for (int i = 0; i <= vec.length - 1; i++) {
      t[i] = new Integer(vec[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();
  }
}
```

**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)
```

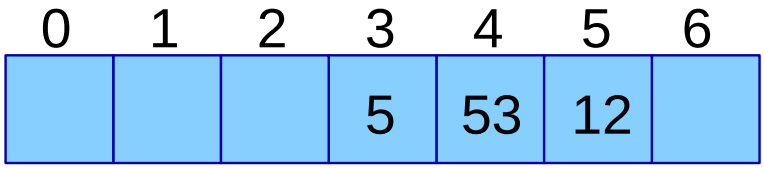
# Sondierungsfolgen—Lineares Sondieren

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

Sondierungsfolge für $k$:

$$h(k), \quad h(k)-1, \quad h(k)-2, \quad h(k)-3, \ldots$$

Problem:
primäre Häufung (“primary clustering”)




$P(\text{nächster Schlüssel bei } T_2) =$

$P(\text{nächster Schlüssel bei } T_0) =$

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

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

# Quadratisches Sondieren

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

Sondierungsfolge für $k$:

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

Permutation, falls $m = 4l + 3$, prim

Problem: sekundäre Häufung

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

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

## Uniformes Sondieren

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

$\pi_k$ eine der $m!$ Permutationen von $\{0, \ldots, m-1\}$

- hängt nur von $k$ ab
- gleichwahrscheinlich für jede Permutation

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

## Zufälliges Sondieren

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

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

# Double Hashing

Idee: Wähle zweite Hashfunktion $h'$

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

Sondierungsfolge für $k$:

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

Permutation:

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

($h'(k)$ relativ prim zu $m$)

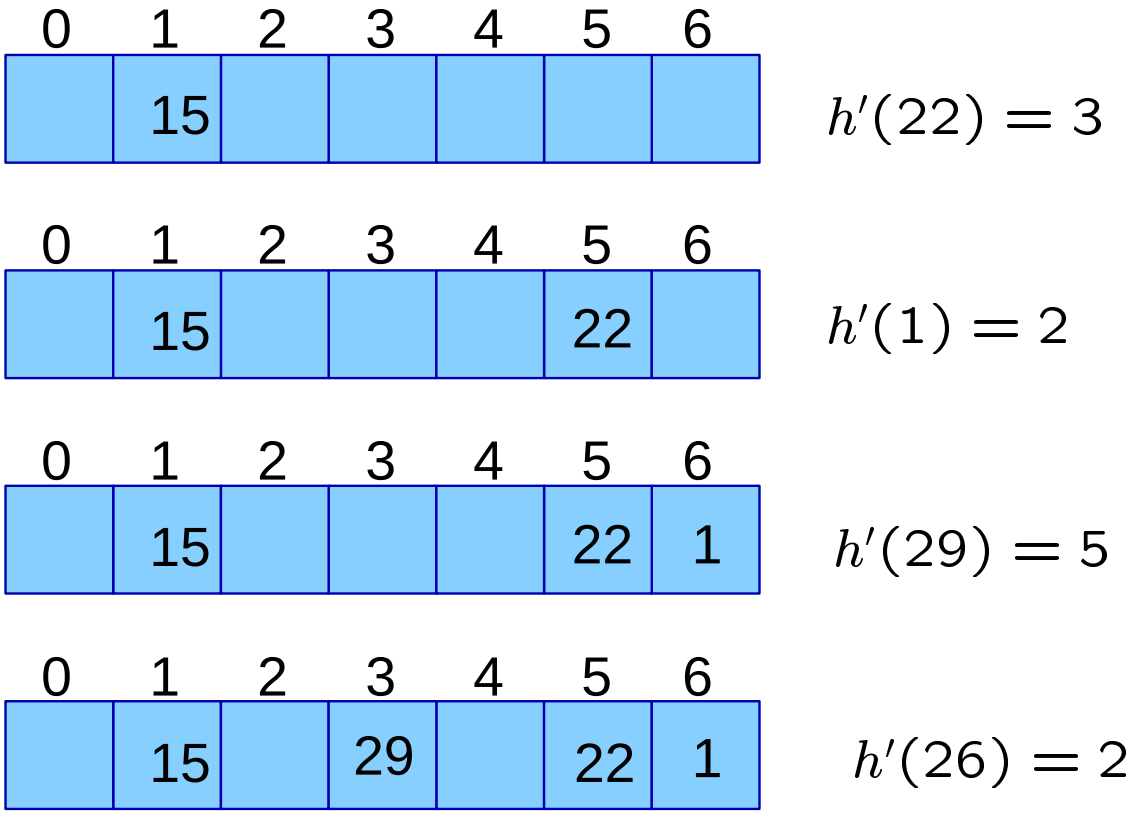
z.B. $h'(k) = 1 + (k \bmod (m-2))$

# Beispiel

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

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

Schlüsselfolge: 15, 22, 1, 29, 26



## Verbesserung der erfolgreichen Suche

**Beispiel**:

Hashtabelle der Größe 11, Double Hashing mit

$$h(k) = k \bmod 11 \qquad \text{und}$$

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

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

$h(k) - s(j+1, k)$ für $k$

(M2) $k$ verdrängt $k_{alt}$

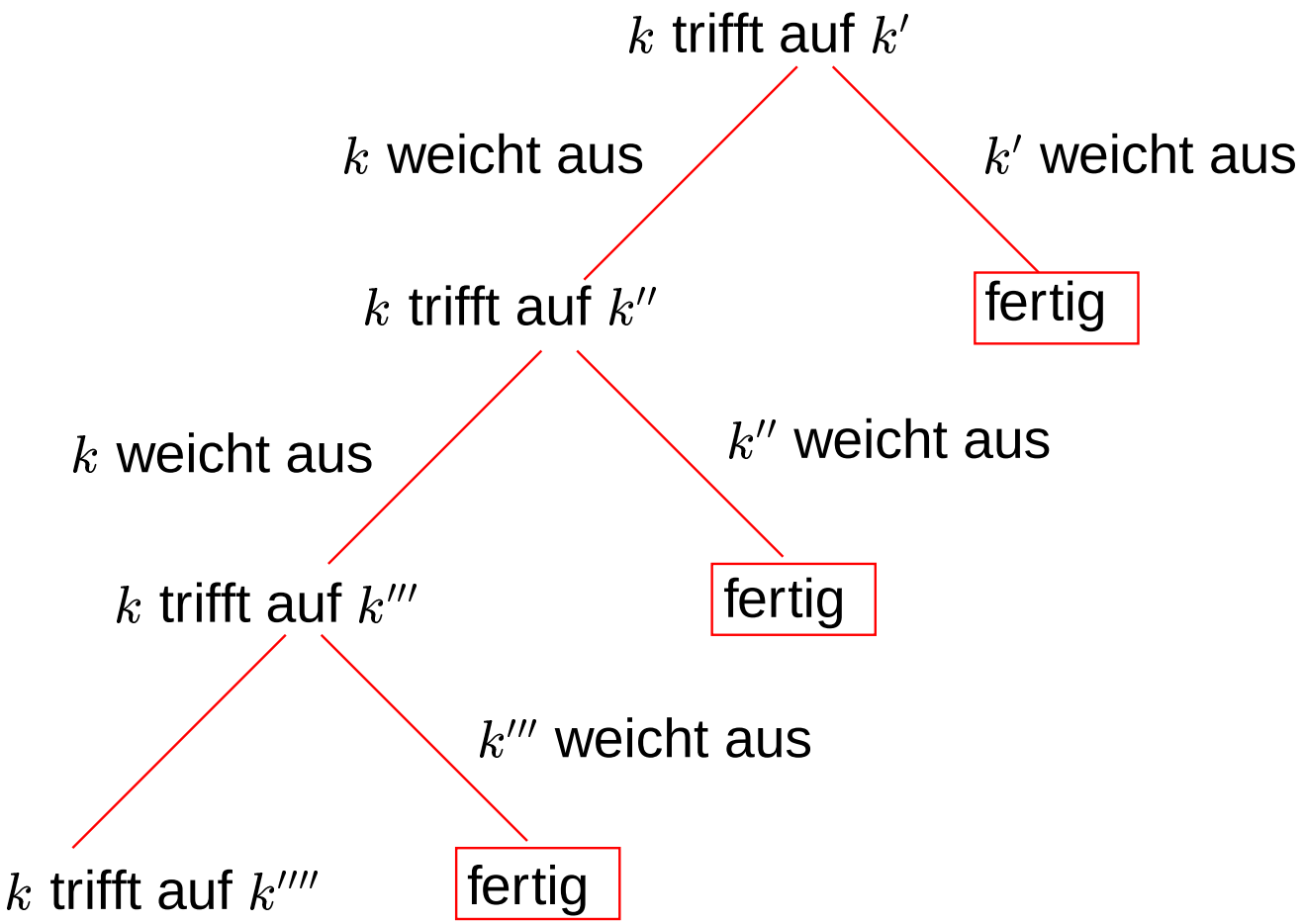
betrachte neue Position

$h(k_{alt}) - s(j'+1, k_{alt})$ 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:
$\Rightarrow$ 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 $1 \leq j \leq m-1$.

$\Rightarrow$ lineares Sondieren, double Hashing

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

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

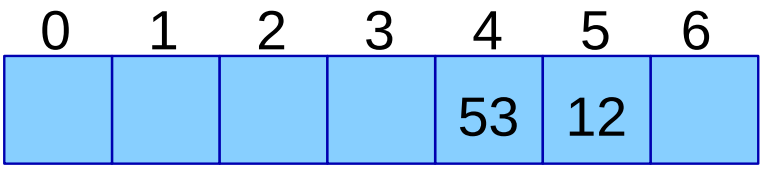
$$C_n^{Binärbaum} \approx 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} + \cdots < 2.2$$

# Beispiel

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

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

Schlüsselfolge: 12, 53, 5, 15, 2, 19




$h(5) = 5$ 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

# Verbesserung erfolglose Suche

**Suche** nach $k$:

$k' > k$ in Sondierungsfolge: $\Rightarrow$ 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?

  $\Rightarrow s(j,k) - s(j-1,k) = s(1,k)$, $1 \leq j \leq m$

# Ordered Hashing

Suchen

Input: Schlüssel $k$

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

1 Beginne bei $i \leftarrow h(k)$

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

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

end while;

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

then Suche erfolgreich

else Suche erfolglos

# Ordered Hashing

Einfügen

Input: Schlüssel $k$

1 Beginne bei $i \leftarrow h(k)$

2 while $T[i]$ nicht frei and $T[i].k \neq k$ 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 $k$

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

  end while;

3 if $T[i]$ ist nicht belegt

  then trage $k$ bei $T[i]$ ein

# 1 Dynamische Tabellen

Problem: Verwaltung einer Tabelle unter den Operationen Einfügen und Entfernen, so daß

- die Tabellengröße der Anzahl der Elemente angepaßt werden kann
- immer ein konstanter Anteil der Tabelle mit Elementen belegt ist
- die Kosten für $n$ Einfüge- oder Entferne-Operationen $O(n)$ sind.

Organisation der Tabelle: Hashtabelle, Heap, Stack, etc.

Belegungsfaktor $\alpha_T$: Anteil der Tabellenplätze von $T$, die belegt sind.

# Implementation

```
class dynamicTable {

  private int [] table;

  private int size;
  private int num;

  dynamicTable () {
    table = new int [1];
    size = 1;
    num = 0;
  }

  public void insert (int x) {
    if (num == size) {
      int[] newTable = new int[2*size];
      for (int i=0; i < size; i++)
        füge table[i] in newTable ein;
      table = newTable;
      size = 2*size;
    }
    füge x in table ein;
    num = num + 1;
  }
```

# Entfernen in Dynamischen Tabellen

Ziel: Belegungsfaktor soll nicht unter einen Schwellwert fallen.

$\Rightarrow$ Kontraktion erforderlich

Idee: kontrahiere die Tabelle, falls der Belegungsfaktor $\alpha$ unter $1/2$ fällt.

Beispiel:

$n = 2^k$ Operationen auf $T$

Erste $n/2$ Operationen Einfügen (I)

Zweite $n/2$ Operationen:

I, D, D, I, I, D, D, I, I, ...

$\Rightarrow O(n^2)$ viele Operationen

Lösung: kontrahiere die Tabelle, falls der Belegungsfaktor $\alpha$ unter $1/4$ fällt.