Hashverfahren I

- **Bisher:**
  Suchen mit Hilfe von **Schlüsselvergleichen**
- **Jetzt:**
  Stattdessen Adressberechnung
  → Auswertung einer Funktion (**Hash- oder Adressfunktion**)
- **Vorteil:**
  Suche erfolgt weitgehend unabhängig von den anderen Schlüsselwerten
  → in der Regel schneller
  (im Durchschnitt konstanter statt logarithmischer Aufwand)

Hashverfahren II

Prinzipielle Idee von Hashverfahren:

- Speicher ist aufgeteilt in eine Folge von m Speicherzellen (**Buckets**) $B_0, B_1, \ldots, B_{m-1}$.
- Jeder Bucket kann einen oder mehrere ($b =$ ‚bucketsize') Datensätze (**Schlüssel**) aufnehmen.
- Jeder Schlüssel wird durch die **Hashfunktion** eindeutig einem Bucket B_i zugeordnet.

- Sei K die Menge der Schlüssel, so stellt eine Hashfunktion h eine Abbildung der Schlüssel in die Menge der Bucketadressen dar:

$$h: K \rightarrow [0,1, \ldots, m-1]$$

Hashverfahren III

Prinzipielle Idee von Hashverfahren: (Fortsetzung)

- Eine Suchoperation teilt sich auf in:
  - **Auswerten der Hashfunktion**
    (finden der Adresse der Speicherzelle)
    → konstanter Zeitaufwand
  - **Zugriff auf die entsprechende Speicherzelle**
    → im Idealfall: konstanter Zeitaufwand
- Im Idealfall:
  Gesamtzeitaufwand für eine Suchoperation: konstant

Hashverfahren IV

Bemerkungen:

- Ein Hashverfahren, bei dem die obige Suchstrategie in jedem Fall zum Erfolg führt, d.h. gewährleistet, dass jeder Adresse höchstens b Schlüssel zugeordnet werden, nennt man **perfektes Hashing** (‚perfect hashing').
- Wir werden jedoch sehen, dass diese Eigenschaft in der Regel nicht erfüllt ist.
- Sie kann tatsächlich nur unter ganz bestimmten Voraussetzungen, z.B. bei statischen oder sehr kleinen Schlüsselmengen, erfüllt werden.

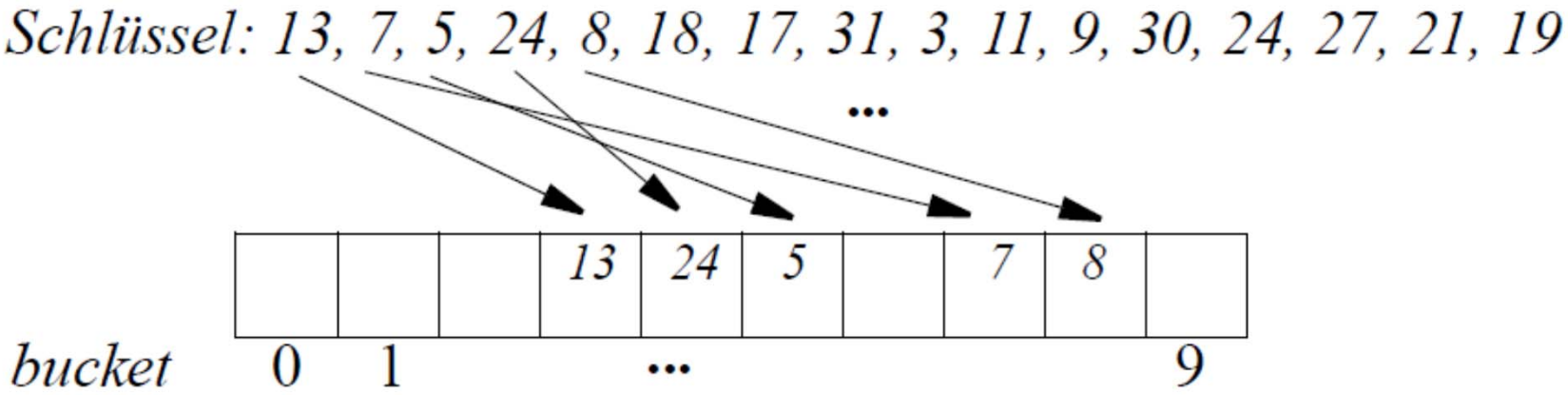
→ Zeitabschätzung für den Idealfall ist ‚in Gefahr'

Hashverfahren V

Beispiele:

Für ganzzahlige Schlüssel:
$h: K \rightarrow [0,1, \ldots, m-1]$ mit $h(k) = k\ MOD\ m$.

Sei $m = 10$



Hashverfahren VI

Für Zeichenketten:

Benutze die ORD-Funktion ($ORD(a) = 1$, $ORD(b) = 2$, …) zur Abbildung auf ganzzahlige Werte, z.B.

$$h: STRING \rightarrow \left(\sum_{i=0}^{s.length-1} (ORD(s[i]) - ORD(a) + 1) \right) MOD\ m$$

Sei $m = 15$

JAN	→ 25 MOD 15 = 10	JUL	→ 43 MOD 15 = 13
FEB	→ 13 MOD 15 = 13	AUG	→ 29 MOD 15 = 14
MAR	→ 32 MOD 15 = 2	SEP	→ 40 MOD 15 = 10
APR	→ 35 MOD 15 = 5	OKT	→ 46 MOD 15 = 1
MAI	→ 23 MOD 15 = 8	NOV	→ 51 MOD 15 = 6
JUN	→ 45 MOD 15 = 0	DEZ	→ 35 MOD 15 = 5

Hashverfahren VII

Erwünschte Eigenschaften für Hashfunktionen:

- **Surjektivität:** Alle Speicherzellen werden erfasst (keine leeren Speicherbereiche)
- Schlüssel sollten **gleichmäßig** über alle Speicherzellen verteilt werden (**ideales Hashverfahren**)
- Wert der Hashfunktion sollte **effizient** zu berechnen sein

Im Allgemeinen gilt: $card(K) \gg m$.

Hashverfahren VIII

Weiteres Beispiel: Personaldatei

- Mögliche Schlüssel: Nachnamen bis zur Länge 10
  $\rightarrow card(K) = 26^{10}$
- Aber: Menge der zu erfassenden Personen nur 10000
  $\rightarrow m = 10000$

Beobachtung:

Die Menge K aller möglichen Schlüssel ist häufig um mehrere Größenordnungen umfangreicher als die Menge der verfügbaren Adressen

→ keine *injektive* Abbildung möglich!

Hashverfahren IX

Prinzipielles Problem des Ansatzes von Hashverfahren:

- **Kollisionen**, d.h. die Hashfunktion ordnet einem Bucket mehr Schlüssel zu, als dieses fasst.
- Im allgemeinen ist die Bucketkapazität für Hashverfahren
  - für Verfahren im Hauptspeicher $b = 1$
  - für Verfahren für den Sekundärspeicher $b > 1$

→ negative Auswirkungen auf die Performanz (Kollisionen verlängern die Suchzeit)

→ Hashfunktion nicht einfach zufällig auswählen, sondern soll auf die Minimierung von Kollisionen ausgerichtet sein!

Selbst bei kleiner Schlüsselmenge K und großem Adressraum (m) ist eine zufällige Funktion $f: K \to \{0,1,\ldots,m-1\}$ gefährlich:

Hashverfahren X

Geburtstagsparadoxon:

- Zufällige Auswahl von 23 Personen ($card(K) = 23$)
- Zufällige Geburtstage, d.h. die Hashfunktion h, die jeder Person ihren Geburtstag zuordnet, ist eine zufällige Adressfunktion mit $m = 365$.
- Nach wahrscheinlichkeitstheoretischen Überlegungen gilt: W(verschiedene Geburtstage der 23 Personen) $\approx 0{,}493$

→ bereits bei einer derart geringen Anzahl von Schlüsseln ist die Wahrscheinlichkeit einer Kollision bei zufällig gewählter Adressfunktion $> 50\%$.

Hashverfahren XI

Bemerkung:

- Auch bei ‚guten' Hashfunktionen können Kollisionen nicht grundsätzlich vermieden werden, sofern die tatsächlich auftretenden Schlüssel nicht von vorneherein bekannt sind.
- Geeignete Strategien zur Behandlung von Kollisionen nötig.

Folgerung:

- Zwei unabhängige Teilprobleme zu lösen:
  - Bestimmung einer ‚guten', kollisionsminimierenden Hashfunktion
  - Angabe einer geeigneten Kollisionsstrategie

In Folgenden Untersuchungen gehen wir auf Hashverfahren im Hauptspeicher ein (→ Kapazität von einem Eintrag, $b = 1$)

Hashfunktionen

Problem:

Gesucht ist eine Hashfunktion $h: K \to [0,1, \ldots, m-1]$, die nichtzufälligen Schlüsseln scheinbar zufällige Adressen zuordnet (**Randomisierung**).

Eine möglichst uniforme Verteilung der Schlüssel auf die Hashadressen wird dabei angestrebt.

Im Folgenden sei die Schlüsselmenge K stets als **int** vorgegeben.

Hashfunktionen: Divisionsmethode

Allgemeiner Ansatz: $h(k) = k\ MOD\ m.$

(Modifikation: $h(k) = ORD'(k)\ MOD\ m$, falls k ein nichtnumerischer Schlüssel und $ORD'(k)$ eine Funktion ist, die k auf einen ganzzahligen Wert abbildet.)

Welche Werte sind für m geeignet?

$m = 2^d$: $h(k) \equiv$ letzte d Bits der Binärzahl k

→ keine zufällige Zuordnung!

geradzahliges m: $h(k)$ gerade $\Leftrightarrow$ k gerade → auch ungünstig!

m Primzahl > 2: → gut geeignet!

Aber: aufeinanderfolgende Schlüssel $k, k+1, \ldots$ werden mit hoher Wahrscheinlichkeit auf aufeinanderfolgende Adressen ($h(k+1) = h(k) + 1$) abgebildet!

Hashfunktionen: Kollisionsstrategien

Beispiel: $m = 11$ und $h(k) = k\ MOD\ 11$

k	7	16	21	30	57	62	78	80
$h(k)$	7	5	10	8	2	7	1	3

Kollision

Zwei Ansätze bei der Behandlung von Kollisionen:

- Offene Hashverfahren
- Geschlossene Hashverfahren

Hashfunktionen: Offene Hashverfahren I

- Jeder Hashadresse können beliebig viele Schlüssel zugeordnet werden.
- Diese werden in geeigneter Form (mittels einer **overflow area**) organisiert.
- Kollisionen werden somit durch ein „erlaubtes Überlaufen“ einer Speicherzelle behandelt.

Vorteile:

- Gesamtanzahl der Schlüssel ist nicht beschränkt

Probleme:

- Zusätzlicher Speicherbedarf
- Hoher Suchaufwand im schlechtesten Fall

Hashfunktionen: Offene Hashverfahren II

Direkte Verkettung („separate chaining“):

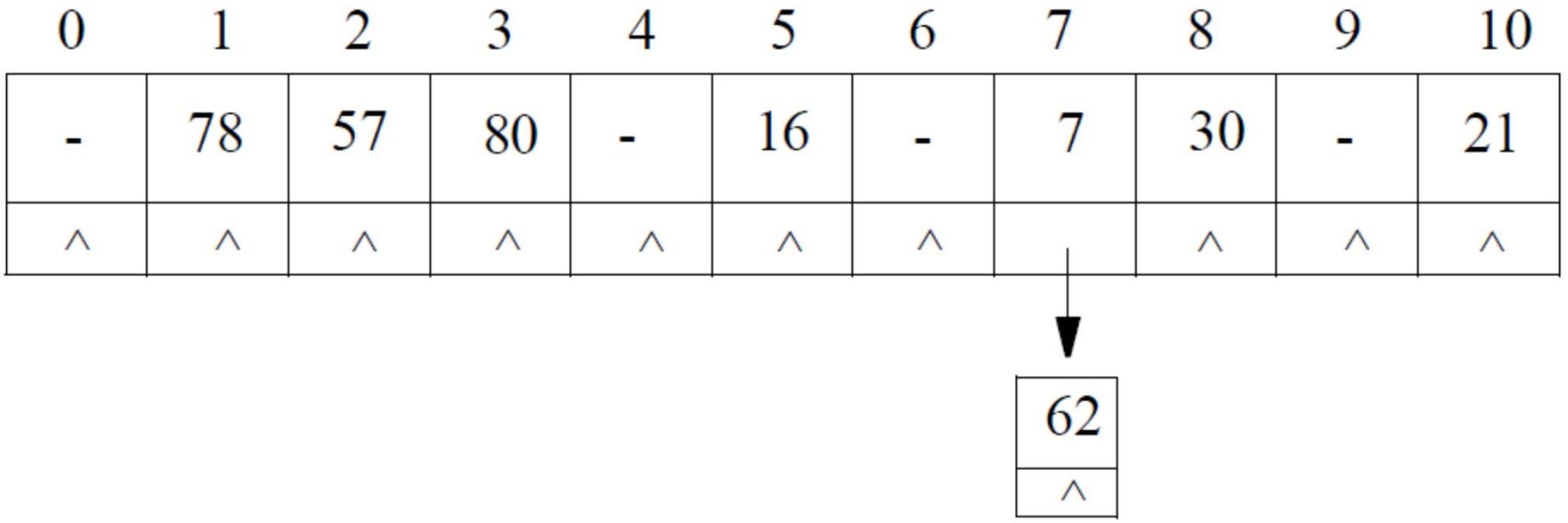
Jedem Eintrag der Hashtabelle wird eine separate Liste von Elementen zugeordnet.

```
class Entry {
   public int key;
   public Entry next;
}
class OpenHashtable {
   protected Entry[] hTable;
   protected int m;
   public OpenHashtable(int size) {
      hTable = new Entry[size];
      m = size;
   }
   // weitere Methoden
}
```

Hashfunktionen: Offene Hashverfahren III

```
OpenHashtable T = new OpenHashtable(11);
```

Beispiel:



Hashfunktionen: Offene Hashverfahren IV

Methode für das Suchen eines Objektes mit Schlüssel k:

```
public boolean search(int k) {
   Entry p;
   // h(k) sei die Hashfunktion, z.B.: h(k) = k % m;
   p = hTable[h(k)];
   while (p != null) {
      if (p.key == k)
         return true;
      else
         p = p.next;
   }
   return false;
}
```

Hashfunktionen: Offene Hashverfahren V

- Bei n Schlüsseln und m Adressen enthält jede der Listen im Mittel $\frac{n}{m}$ Elemente

  → Hashing verbessert den Aufwand der sequentiellen Suche um den Faktor m.

- Einschränkung für die Kardinalität m der Menge der Hashadressen:
  - m zu klein: sehr lange Listen von Objekten → schlechte Zugriffszeit
  - m zu groß: viele leere Listen → unnötig belegter Speicherplatz

  → gute Wahl von m: $m \approx n$.

Hashfunktionen: Offene Hashverfahren VI

Gute Hashfunktion wichtig!

→ keine der Listen wird übermäßig lang
(Ziel: alle Listen gleich lang)

→ guter durchschnittlicher Zeitaufwand: Suche, Einfügen und Entfernen in $O(1)$ Zeit (für $m = n$).

Aber:

Degenerierung im worst-case möglich, auch bei gut gewählter Hashfunktion

→ Zeitaufwand: $O(n)$ im worst-case
Platzbedarf: $O(n + m)$

Hashfunktionen: Geschlossene Hashverfahren I

- Jede Speicherzelle darf grundsätzlich nur mit der durch die Bucketkapazität b festgelegten maximalen Anzahl von Einträgen belegt werden.
- Ist eine Speicherzelle bereits maximal belegt, so ist für weitere Schlüssel mit derselben Hashadresse eine spezielle **Kollisionsstrategie** anzuwenden.

Probleme:

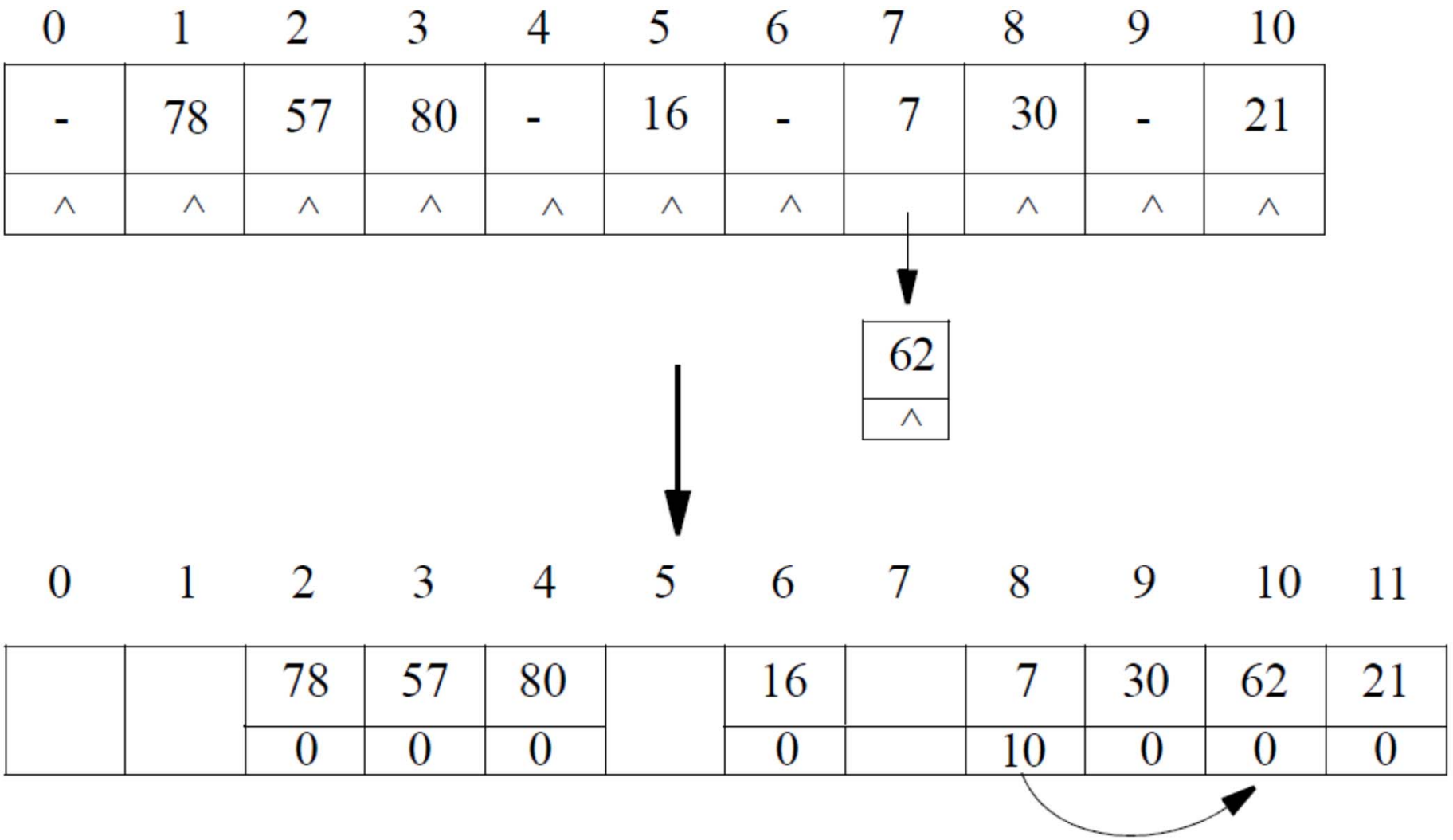
- Gesamtanzahl der Schlüssel ist auf $m \cdot b$ beschränkt
- In der Regel große Effizienzprobleme, falls die Hashtabelle „fast voll“ ist

Hashfunktionen: Geschlossene Hashverfahren II

Direkte Verkettung mit Verschmelzen:

- Wie bei direkten Verkettung, nur werden die Listen innerhalb der Hashtabelle selbst gespeichert.
- Ein Teil des benötigten Speicherplatzes wird eingespart (der für das Array von Zeigern), falls $m \approx n$.
- Speicherverwaltung wird komplexer.

Hashfunktionen: Geschlossene Hashverfahren III



Hashfunktionen: Geschlossene Hashverfahren IV


```
class Entry {
   public boolean frei;
   public int key;
   public int next; // 0 <= next <= m
}
class ClosedHashtable {
   protected Entry hTable[];
   protected int m;
   protected int r; // Anzahl besetzter Adressen
   void ClosedHashtable (int size) {
      hTable = new Entry[size+1];
      m = size;
      r = 0;
      hTable[0].frei = true;
   }
   // weitere Methoden
}
```

Hashfunktionen: Geschlossene Hashverfahren V

Vorgehensweise beim Einfügen eines Schlüssels k:

- Berechne die Hashadresse $h(k) + 1$
- Wenn diese Adresse noch nicht belegt ist: Besetze sie mit k, ansonsten suche eine andere freie Adresse und speichere k dort.

Methode für das Einfügen eines Schlüssels:

- Verwendung einer Hilfsvariablen j mit $j = m + 1$.
  → sucht die höchste freie Komponente in der Hashtabelle
- Erweiterung der Hashtabelle um eine Komponente; dabei gilt stets: `T[0]` ist frei
  → notwendig um OVERFLOW der Hashtabelle zu erkennen

Hashfunktionen: Geschlossene Hashverfahren VI


```
public boolean insert (int k) {
   int i, j;
   boolean b;
   i = h(k) + 1;
   if (!hTable[i].frei) {
      b = false;
      j = m + 1;
      while (!b) {
         if (hTable[i].key == k)
            return false; // Schlüssel vorhanden
         else if (hTable[i].next != 0)
            i = hTable[i].next;
         else
            b = true;
      } // i mit hTable[i].next == 0 und hTable[i].key != k -> k einfügen
                                  ...
```

Hashfunktionen: Geschlossene Hashverfahren VII


```
                                    ...
        do
           j = j - 1;
        while (!hTable[j].frei); // hTable[0] ist stets frei
        if (j == 0)
           return false; // overflow
        else
           hTable[i].next = j; i = j;
     } // end if
     hTable[i].frei = false;
     hTable[i].next = 0;
     hTable[i].key = k;
     return true;
}
```

Hashfunktionen: Geschlossene Hashverfahren VIII


```
ClosedHashtable T = new ClosedHashtable(11);
```

Beispiel:

0	1	2	3	4	5	6	7	8	9	10	11
		78	57	80		16		7	30	62	21
		0	0	0		0		10	0	0	0

Hinzunahme des Schlüssels ‘20’ (h(20) = 9 → an die 10.Stelle)

0	1	2	3	4	5	6	7	8	9	10	11
		78	57	80		16	**20**	7	30	62	21
		0	0	0		0	0	10	0	7	0

Hashfunktionen: Geschlossene Hashverfahren IX

Verschiedene Listen werden miteinander verschmolzen.

→ Suche ist damit häufig nicht nur auf die Elemente einer Hashadresse beschränkt.

→ ungünstig für die Suchzeit.

Hashfunktionen: Offene Adressierung (rehashing) I

- Explizite Verzweigung von Elementen wird vermieden.
- Stattdessen: **Adressberechnung** wird für überlaufende Elemente iteriert.

Lineares Sondieren („linear probing“):

- Betrachte ausgehend von $h(k)$ jeweils das Folgeelement:
$$h(k), h(k)+1, h(k)+2, \ldots, m-1, 0, 1, \ldots, h(k)-1$$
(**zyklische Sondierungsfolge**)
- Für jeden Schlüssel k: Elemente der Hashtabelle werden in dieser zyklischen Sondierungsfolge besucht, bis der Schlüssel k oder eine leere Position gefunden wird.
- Einfügen eines neuen Schlüssels erfolgt nur falls $r < m-1$
$\rightarrow r$: Anzahl besetzter Adressen („voll“: $r == m-1$)

Hashfunktionen: Offene Adressierung (rehashing) II


```
public boolean insert(int k) {
   int i;
   boolean b;
   if (r == m - 1)
      return false; // overflow
   else {
      r = r + 1;
      i = h(k);
      if (!hTable[i].frei) {
         b = false;
         while (!b) {
            if (hTable[i].key == k)
               return false; // Schlüssel vorhanden

                                 ...
```

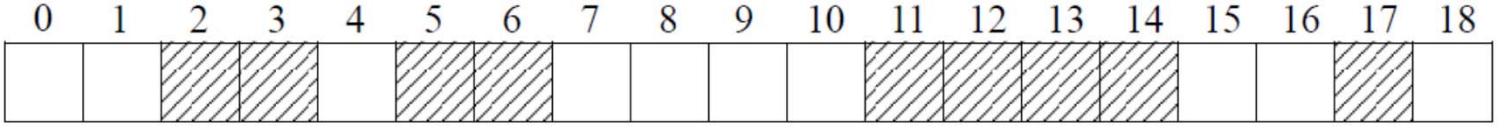
Hashfunktionen: Offene Adressierung (rehashing) III


```
                                ...
                else {
                    i = i + 1;
                    if ( i > m -1)
                        i = i - m;
                    if (hTable[i].frei)
                        b = true;
                }
            }
        } // nach der Schleife gilt hTable[i].frei == true
        hTable[i].frei = false;
        hTable[i].key = k;
        return true;
    }
}
```

Hashfunktionen: Offene Adressierung (rehashing) IV

- Lineares Sondieren ist gut, falls Hashtabelle nicht zu voll ist
- Ist die Hashtabelle dagegen annähernd voll:
  → lange Ketten besetzter Positionen wachsen mit höherer Wahrscheinlichkeit als kurze.
  → ausgeprägte Tendenz zur Kettenbildung
  → ungünstiges Suchverhalten

$m = 19$, n = 9



$k \rightarrow$ Position 15 $\Leftrightarrow 11 \leq h(k) \leq 15$
$k \rightarrow$ Position 8 $\Leftrightarrow h(k) = 8$

$\Rightarrow$ Einfügen in Position 15 ist 5 mal so wahrscheinlich wie in Position 8

allgemein: Einfügen an eine Stelle mit l besetzten Vorgängern ist $(l+1)$ mal so wahrscheinlich wie das Einfügen an eine isolierte Position.

Hashfunktionen: Offene Adressierung (rehashing) V

- Allgemein gibt man bei der offenen Adressierung eine Folge von Hashfunktionen vor: $h_0(k), h_1(k), h_2(k), \ldots, h_{m-1}(k)$.
- Diese sollte so gewählt sein, dass sämtliche Adressen $0 \ldots m-1$ auftreten.
- Für einen einzufügenden Schlüssel k werden nacheinander die entsprechenden Zellen der Hashtabelle inspiziert (sondiert), bis eine freie Zelle gefunden ist.
- Dieselbe Strategie gilt auch für Suchoperationen.

Im Beispiel des linearen Sondierens gilt:

$$h_i(k) = (h(k) + i)\ MOD\ m, \qquad 0 \leq i \leq m-1$$

Hashfunktionen: Offene Adressierung (rehashing) VI

Problem:

- Einträge dürfen nicht einfach gelöscht werden: Kollisionspfade anderer Elemente könnten unterbrochen werden!
- Stattdessen werden entfernte Elemente nur als „entfernt“ markiert
  → ungünstig für stark dynamische Anwendungen