2.4.1 Allgemeine Einführung

Prinzipielle Idee von Hashverfahren:

Der Speicher sei aufgeteilt in eine Folge von m Speicherzellen (*buckets*) $B_0, B_1, \ldots, B_{m-1}$, die jeweils einen oder mehrere (b = *'bucketsize'*) Datensätze (Schlüssel) aufnehmen können. 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:

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

Eine Suchoperation teilt sich dann auf in:

- Auswerten der Hashfunktion um die Adresse der Speicherzelle zu finden
  ➞ *konstanter* Zeitaufwand
- Zugriff auf die entsprechende Speicherzelle
  ➞ im Idealfall: *konstanter* Zeitaufwand

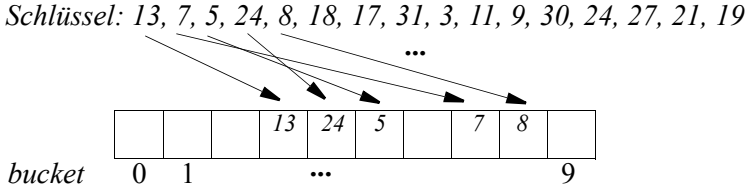
➞ im Idealfall: Gesamtzeitaufwand für eine Suchoperation: *konstant*.

Bemerkung: 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 im Gegenteil nur unter ganz bestimmten Voraussetzungen, z.B. bei statischen oder sehr kleinen Schlüsselmengen, erfüllt werden.
Damit ist auch unsere Zeitabschätzung für den Idealfall 'in Gefahr'.

Beispiel:

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

sei: $m = 10$



Für *Zeichenketten*: Benutze die 'ORD-Funktion' (ORD ('a') = 1, ORD ('b') = 2, . . .) zur Abbildung auf ganzzahlige Werte, z.B.:

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

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

Erwünschte Eigenschaften für solche Hashfunktionen:

- *Surjektivität*, d.h. alle Speicherzellen werden erfaßt (keine leeren Speicherbereiche)
- die Schlüssel sollten *gleichmäßig* über alle Speicherzellen verteilt werden. (*ideales Hashverfahren*)
- der Wert der Hashfunktion sollte *effizient* zu berechnen sein.

Im allgemeinen, wie auch in obigem Beispiel gilt: $card(K) \gg m$.

Weiteres Beispiel: *Personaldatei*

mögliche Schlüssel sind alle Nachnamen bis zur Länge 10 ➞ $card(K) = 26^{10}$

aber: die Menge der zu erfassenden Personen nur z.B. 10 000 ➞ $m = 10000$

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

➞ keine *injektive* Abbildung möglich!

Prinzipielles Problem des Ansatzes von **Hashverfahren**:

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

➞ negative Auswirkungen auf die Performanz, da Kollisionen die Suchzeit verlängern

➞ die Hashfunktion sollte nicht einfach zufällig ausgewählt werden, sondern auf die Minimierung solcher Kollisionen ausgerichtet sein!

Daß man zur Minimierung von Kollisionen selbst bei kleiner Schlüsselmenge K und großem Adressraum (großem m) nicht einfach irgendeine zufällige Funktion $h : K \to \{0, 1, \ldots, m-1\}$ wählen darf, zeigt das folgende

Geburtstagsparadoxon:

Wir wählen 23 Personen zufällig aus ($card(K) = 23$). Dann sind auch deren Geburtstage zufällig, d.h. die 'Hashfunktion' h, die jeder Person ihren Geburtstag zuordnet, ist eine zufällige Adressfunktion mit $m = 365$.

Nach wahrscheinlichkeitstheoretischen Überlegungen gilt:

$$W(\text{die Geburtstage der 23 Personen sind verschieden}) \approx 0{,}493 \quad .$$

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

Bemerkung: Auch bei nach obigen Kriterien 'guten' Hashfunktionen können Kollisionen nicht grundsätzlich vermieden werden, sofern die tatsächlich auftretenden Schlüssel nicht von vorneherein (statisch) bekannt sind.
➞ geeignete Strategien zur Behandlung von Kollisionen sind nötig.

Folgerung:

Bei der Definition eines Hashverfahrens sind zwei unabhängige Teilprobleme zu lösen:
- Bestimmung einer 'guten', kollisionsminimierenden Hashfunktion
- Angabe einer geeigneten Kollisionsstrategie

In den folgenden Untersuchungen gehen wir zunächst auf Hashverfahren **im Hauptspeicher** ein. Hierbei geht man von einer Kapazität der Speicherzellen von einem Eintrag ($b = 1$) aus.

2.4.2 Hashfunktionen

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

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

Divisionsmethode:

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

(*Modifikation*: $h(k) = (\text{ORD}^*(k)) \text{ MOD } m$, falls k ein nichtnumerischer Schlüssel und $\text{ORD}^*(k)$ 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 ➞ *ebenfalls 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.

Middle-Square-Methode:

Allgemeiner Ansatz: Der Schlüssel k wird quadriert, und man nimmt als Hashadresse $h(k)$ die "mittleren" Ziffern der Repräsentation von k^2.

Sei z.B. $m = 10^3$ und seien die Schlüssel 10-stellige Zahlen.

Für einen Schlüssel $k = a_1 a_2 \ldots a_{10}$ berechne $k^2 = b_1 b_2 \ldots \boldsymbol{b_9 b_{10} b_{11}} \ldots b_{20}$.

Als Wert der Hashfunktion verwenden wir: $h(k) = b_9 b_{10} b_{11}$.

Vorteil: $b_9 b_{10} b_{11}$ hängt von allen Ziffern $a_1 a_2 \ldots a_{10}$ von k ab
➞ bessere Streuung aufeinanderfolgender Schlüssel.

aber: dieses Verfahren versagt, falls viele 'Nullen' am Anfang oder am Ende von k stehen.

Beispiel:

Sei $m = 100$

k	$h_{\text{Div.}}(k)$	k^2	$h_{\text{M-Sq.}}(k)$
550	50	30**25**00	25
551	51	30**36**01	36
552	52	30**47**04	47
553	53	30**58**09	58

2.4.3 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

Bei der Behandlung von Kollisionen gibt es zwei prinzipielle Ansätze: offene Hashverfahren und geschlossene Hashverfahren, die im folgenden beide behandelt werden.

Offene Hashverfahren:

Hierbei geht man davon aus, daß jeder Hashadresse beliebig viele Schlüssel zugeordnet werden können. Diese werden in geeigneter Form (mittels einer *overflow area*) organisiert. Kollisionen werden somit durch ein 'erlaubtes Überlaufen' einer Speicherzelle behandelt.

Vorteile: die Gesamtanzahl der Schlüssel ist nicht beschränkt.

Probleme: zusätzlicher Speicherbedarf; hoher Suchaufwand im schlechtesten Fall.

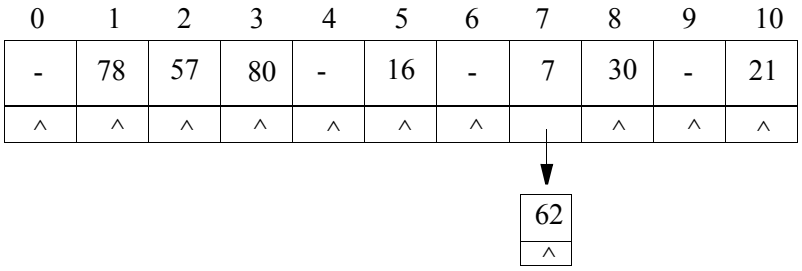
Direkte Verkettung (*'separate chaining'*):

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

Sei: **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
    }
```

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



Methode für das Suchen eines Objektes mit Schlüssel *k*:

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

Bei n Schlüsseln und m Adressen enthält jede der Listen im Mittel n/m Elemente
➞ das Hashing verbessert somit den Aufwand der sequentiellen Suche um den Faktor m.

Einschränkung für die Cardinalität m der Menge der Hashadressen:
m zu klein : sehr lange Listen von Objekten ➞ schlechte Zugriffszeit
m zu groß : viele leere Listen und damit unnötig belegter Speicherplatz.
➞ gute Wahl von m: $m \approx n$.

Gute Hashfunktion ➞ 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 und Platzbedarf: $O(n+m)$.

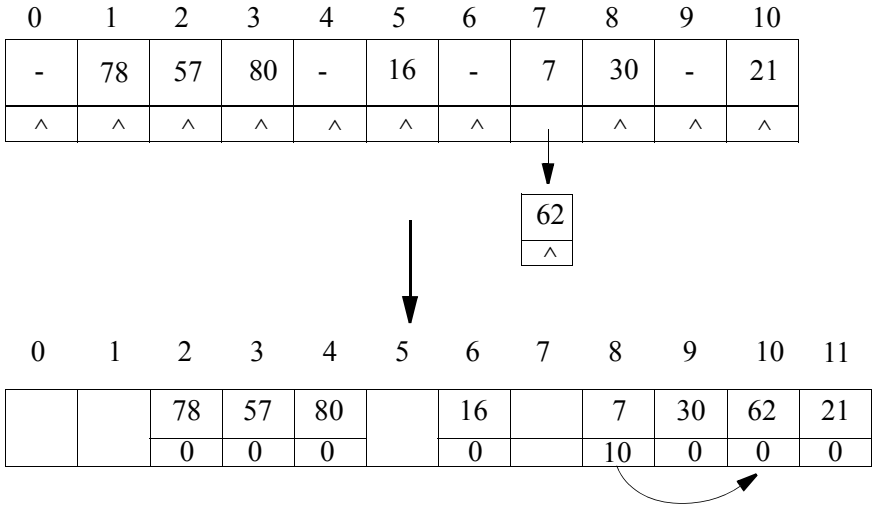
Geschlossene Hashverfahren:

Jede der Speicherzellen 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: • die Gesamtanzahl der Schlüssel ist auf $m \cdot b$ beschränkt;
• in der Regel große Effizienzprobleme, falls die Hashtabelle 'fast voll' ist.

Direkte Verkettung mit Verschmelzen:

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



Sei: **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}
```

Vorgehensweise beim Einfügen eines Schlüssels *k*:

- berechne die Hashadresse $h(k) + 1$
- wenn diese Adresse noch nicht belegt ist, so besetze sie mit *k*,
  im anderen Fall suche eine andere freie Adresse und speichere *k* dort.

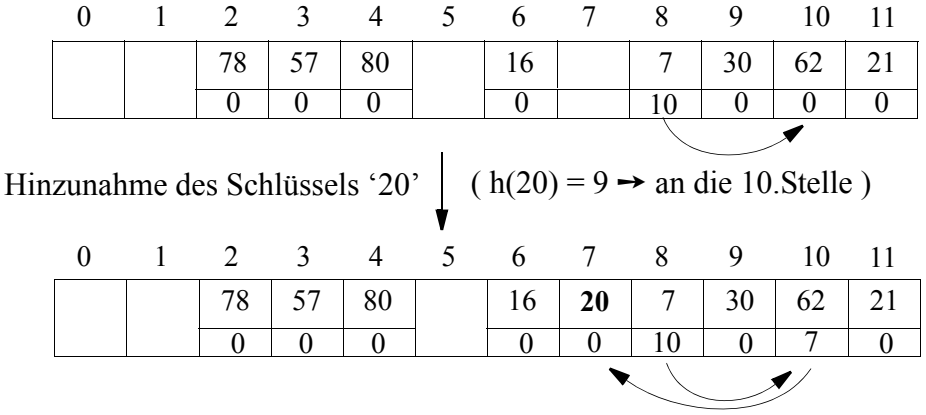
Methode für das Einfügen eines Schlüssels:

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

```
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
      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;}
```

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

Beispiel



Verschiedene Listen werden miteinander verschmolzen.

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

➞ ungünstig für die Suchzeit

Offene Adressierung (*rehashing*):

Eine explizite Verzeigerung von Elementen wird vermieden.
Stattdessen: die *Adressberechnung* wird für überlaufende Elemente *iteriert*.

Lineares Sondieren ('linear probing'):

Betrachte ausgehend von $h(k)$ jeweils das Folgeelement:

$h(k)$, $h(k) + 1$, $h(k) + 2$, ..., m - 1, 0, 1, ... $h(k) - 1$ (*zyklische Sondierungsfolge*)

Für jeden Schlüssel k werden die Elemente der Hashtabelle 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

➞ r bezeichnet die Anzahl besetzter Adressen. ('voll': $r == m - 1$)

```
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
            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;
      }
}
```

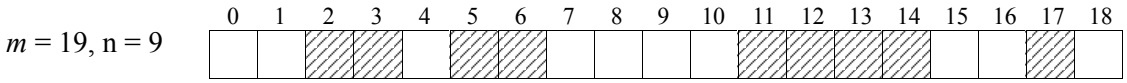
Lineares Sondieren ist gut, falls die Hashtabelle nicht zu voll ist.

Ist die Hashtabelle dagegen annähernd voll

➞ lange Ketten besetzter Positionen wachsen mit höherer Wahrscheinlichkeit als kurze.

➞ ausgeprägte Tendenz zur Kettenbildung ➞ ungünstiges Suchverhalten.

Beispiel:



k ➞ Position 15 $\Leftrightarrow 11 \le h(k) \le 15$
k ➞ 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.

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 ... m-1 auftreten.
Für einen einzufügenden Schlüssel k werden nacheinander die entsprechenden Zellen der Hashtabelle inspiziert (sondiert), bis eine freie Zelle gefunden ist.
Diesselbe Strategie muß bei Suchoperationen angewandt werden.

Im Beispiel des linearen Sondierens gilt: $h_i(k) = (h(k) + i) \text{ MOD } m$, $0 \le i \le m-1$.

Problem: Einträge dürfen nicht einfach gelöscht werden, da Kollisionspfade anderer Elemente unterbrochen werden können!
Stattdessen werden entfernte Elemente nur als “entfernt” markiert.
➞ ungünstig für stark dynamische Anwendungen.

2.4.4 Externe Hashverfahren

Um sehr große Datenmengen zu organisieren, reicht der Hauptspeicher nicht mehr aus
- ➞ Organisation der Datensätze auf dem Sekundärspeicher
- ➞ externe Verfahren notwendig!

Wesentliche Eigenschaft: Bucketgröße $b > 1$

Ziel: Anzahl der Sekundärspeicherzugriffe (Zugriffe auf Buckets) gering halten
- ➞ gute Hashfunktion sehr wichtig!

In Abhängigkeit von der Kollisionsbehandlung unterscheiden wir:
- Verfahren mit Directory (= geschlossene Hashverfahren) und
- Verfahren ohne Directory (= offene Hashverfahren)

Wir beschränken uns hier auf ein Beispielverfahren ohne Directory.
(vgl. auch die Vorlesung: '*Index- und Speicherungsstrukturen für Datenbanksysteme*')

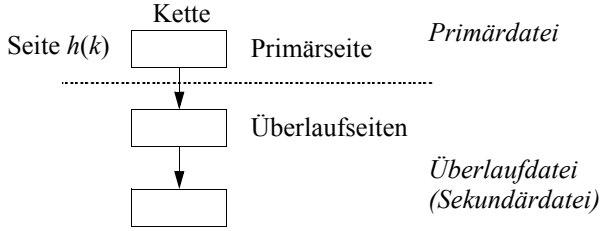
Das lineare Hashing (Litwin, 1980)

Zwei grundlegende Eigenschaften des linearen Hashing:
- die Größe der Datei (Hashtabelle) ist abhängig von der Anzahl der Schlüssel (*dynamisch*)
- Überlaufsätze einzelner Buckets ➞ **getrennte Verkettung** in einem *Überlaufbereich*

Die *Überlaufstrategie der getrennten Verkettung*:

➞ zwei verschiedene Seitentypen:
- *Primärseiten* ➞ *Primärdatei*
  - ➞ Datenseiten (Buckets), deren Adressen von der Hashfunktion berechnet werden.
- *Überlaufseiten* ➞ *Überlaufdatei (Sekundärdatei)*
  - ➞ speichern die *Überlaufsätze* der Primärdatei.

Es gilt:
(i) Überlaufseiten und Primärseiten sind verschieden.
(ii) jede Überlaufseite ist genau einer Hashadresse (einem Bucket) zugeordnet.
(iii) die Überlaufseiten derselben Primärseite werden verkettet.



➞ die Suche nach einem Überlaufsatz benötigt also mindestens 2 Seitenzugriffe (Primärseite + 1 Überlaufseite).
Entstehen lange Ketten von Überlaufseiten, so degenerieren die Suchzeiten.

Bis ca. 1978 ging man davon aus, daß
- die Größe der Primärdatei (Hashtabelle) *statisch* vorgegeben ist
- Überläufe nur durch Einführung von *Überlaufsätzen* gelöst werden können.

➞ schnelle Degenerierung der Suchzeiten, wenn die Primärseiten voll werden.

Jetzt: stattdessen eine *dynamische Erweiterung der Primärdatei*:
- wenn die Primärseiten zu voll werden ➞ Erweiterung der Datei
- neue Seiten werden über eine *neue Hashfunktion* adressiert

➞ man benötigt eine Folge von Hashfunktionen $h_0, h_1, h_2, \ldots$

Das Verfahren:

- zu Beginn ➞ kleine Primärdatei (z.B. $m = 5$); Hashfunktion: $h_0: K \to \{0, 1, 2, 3, 4\}$.
  - die ersten b Schlüssel ➞ sicher in Primärseiten.
  - zunehmende Schlüsselanzahl ➞ Wahrscheinlichkeit für Überlaufsätze steigt.

- *Idee*: Vergrößerung (**Expansion**) der Primärdatei um jeweils ein Bucket, falls eine vorgegebene Auslastung der Primärseiten erreicht ist. (**Kontrollfunktion**)

  Definition: **Belegungsfaktor**

$$bf = \frac{\text{Anzahl der Schlüssel}}{\text{Anzahl Schlüssel, die in Primärseiten passen}} = \frac{\text{Anzahl der Schlüssel}}{b \cdot \text{Anzahl der Primärseiten}}.$$

  Es wird vorgeschlagen: Expansion der Datei, falls $bf > 0{,}8$.

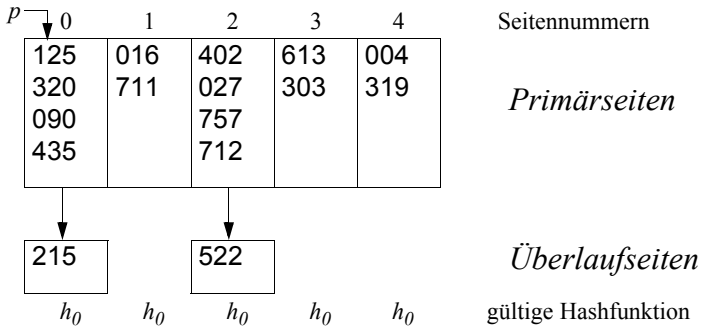
- die *Expansion der Primärdatei*
  - jeweils *eine* Primärseite wird *in zwei* Seiten aufgespalten.
  - der Split der Primärseiten erfolgt *zyklisch* in einer *fest vorgegebenen Reihenfolge*.
  - ein *Expansionszeiger p* zeigt auf die als nächstes zu splittende Seite.
  - für gesplittete Seiten gilt eine *neue Hashfunktion* ($h_i \to h_{i+1}$)
  - nach einem Seitensplit ➞ Setzen des Expansionszeigers auf die nächste Seite.
  - sind alle Seiten gesplittet ➞ die Datei wurde verdoppelt

    ➞ die neue Hashfunktion gilt für die gesamte Datei; das Verfahren beginnt von neuem.

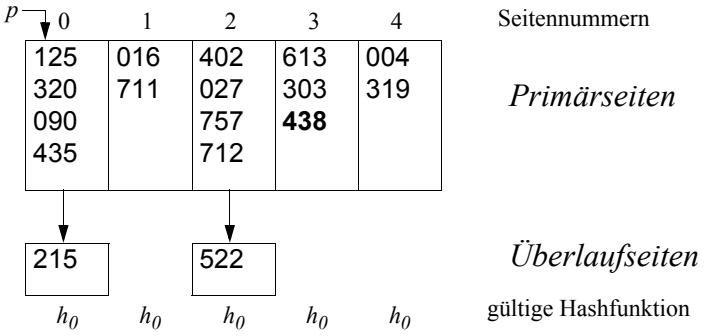
Beispiel:

Ausgangssituation:

- Datei mit 16 Datensätzen auf 5 Primärseiten (Buckets) der Seitengröße $b = 4$.
- Folge von Hashfunktionen: $h_0(k) = k \text{ MOD } 5$, $h_1(k) = k \text{ MOD } 10$, ...
- Aktuelle Hashfunktion: h_0.
- Belegungsfaktor $bf = \frac{16}{20} = 0{,}8$
- Schwellenwert für den Belegungsfaktor = 0,8
- Expansionszeiger p zeigt auf Seite 0.

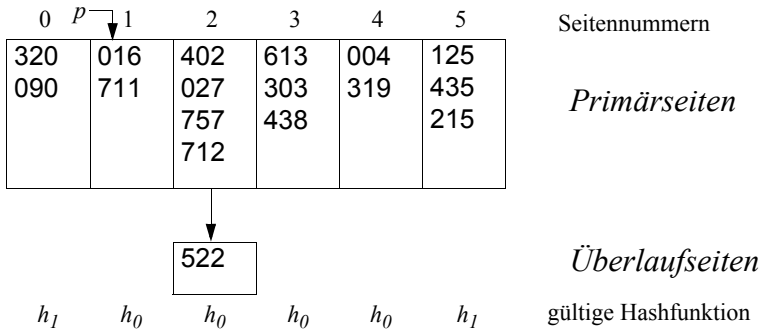


Einfügen eines neuen Datensatzes mit Schlüssel 438:



- Der Belegungsfaktor übersteigt den Schwellenwert: $bf = \frac{17}{20} = 0{,}85 > 0{,}8$

  ➞ Expansion durch Split der Seite 0 auf die Seiten 0 und 5.



Welche Datensätze von Seite 0 nach Seite 5 umgespeichert werden, bestimmt die Hashfunktion h_1:

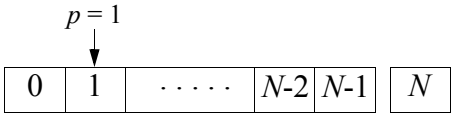
➞ Sätze mit $h_1(k) = 5$ werden umgespeichert, Sätze mit $h_1(k) = 0$ bleiben

Anschließend wird der Expansionszeiger p um 1 heraufgesetzt.

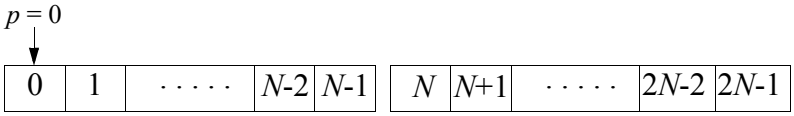
Das Prinzip der Expansion durch Splits:

Ausgangssituation:

nach dem ersten Split:



nach der Verdoppelung der Datei:



Anforderungen an die Hashfunktionen

Die Folge von Hashfunktionen $\{h_i\}$, $i \geq 0$, muß die folgenden Bedingungen erfüllen:

Bereichsbedingung:

$$h_L: K \rightarrow \{0, 1, \ldots, N \cdot 2^L - 1\}\ , L \geq 0.$$

Splitbedingung:

$$h_{L+1}(k) = h_L(k) \quad \text{oder} \quad h_{L+1}(k) = h_L(k) + N \cdot 2^L, \quad L \geq 0$$

Der *Level L* gibt dabei an, wie oft sich die Datei bereits vollständig verdoppelt hat.

Beispiel für eine mögliche Folge von Hashfunktionen:

$h_L(k) = k$ MOD $(N \cdot 2^L)$ erfüllt die Bereichs- und die Splitbedingung.

Zusätzlich erwünschte Eigenschaft:

Für $p = 0$ (nach einer Verdoppelung) besitzen alle Buckets die gleiche Wahrscheinlichkeit, einen neu eingefügten Datensatz aufzunehmen. (*Ideale Hashfunktion*)

Wichtige Eigenschaften des linearen Hashing

- die Splitreihenfolge der Seiten ist fest vorgegebenen.
  ➞ Seiten, die überlaufen werden erst mit einer gewissen Verzögerung behandelt.
- der Adreßraum wächst **linear** an und ist gerade so groß wie nötig.
- unter der Voraussetzung einer guten Hashfunktion ist der Prozentsatz der Überlaufsätze gering.
- gute Suchzeiten für gleichverteilte Schlüssel.

Das lineare Hashing mit partiellen Erweiterungen (Larson, 1980)

Beobachtung: Hashverfahren sind am effizientesten, wenn die Schlüssel möglichst gleichmäßig auf die Seiten der Datei verteilt sind.

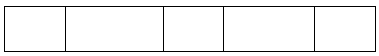
Problem: Während jeder Verdoppelung der Datei gilt:
Der durchschnittliche Belegungsfaktor bereits gesplitteter Seiten ist nur halb so hoch wie der durchschnittliche Belegungsfaktor noch nicht gesplitteter Seiten.
➞ keine gleichmäßige Verteilung der Datensätze auf die Buckets.

Verbesserung: Einführung ***partieller Expansionen:***
➞ das Verdoppeln der Datei erfolgt nicht in einem, sondern in mehreren Durchläufen
➞ Serie von $n_0 \geq 2$ partiellen Expansionen.

Vorgehensweise des **linearen hashing mit partiellen Erweiterungen*:*** (für $n_0 = 2$)

Ausgangssituation:

Datei mit *2N* Seiten



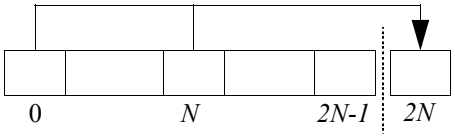
logisch unterteilt in *N* Paare (j, j+*N*) für j = 0, 1, ... ,*N*-1

1. partielle Expansion

- nach Einfügen von Schlüsseln wird aufgrund der Kontrollfunktion eine Expansion nötig.
  ➞ Expansion der Datei um die Seite *2N*:

Etwa $1/3$ der Sätze aus den Seiten 0 und N werden nach Seite $2N$ umgespeichert.

Datei mit $2N+1$ Seiten

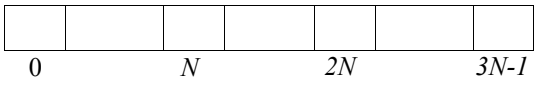


logisch unterteilt in die verbleibenden $N-1$ Paare $(j, j+N)$ für $j = 1, \ldots, N-1$
sowie das Tripel $(0, N, 2N)$

- bei weiteren Einfügungen wird, wenn es die Kontrollfunktion verlangt, für $j = 1, 2, \ldots, N-1$ jeweils das Paar $(j, j+N)$ um die Seite $j+2N$ expandiert.

  → nach N Schritten: die Datei ist von $2N$ auf $3N$ Seiten (auf das 1,5-fache) angewachsen:

Datei mit $3N$ Seiten



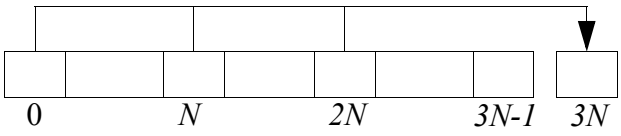
logisch unterteilt in N Tripel $(j, j+N, j+2N)$ für $j = 0, 1, \ldots, N-1$

2. Partielle Expansion:

- Zunächst Expansion der Datei um die Seite $3N$:

  Etwa je $1/4$ der Sätze aus den Seiten 0, N und $2N$ werden umgespeichert auf die Seite $3N$.

Datei mit $3N+1$ Seiten



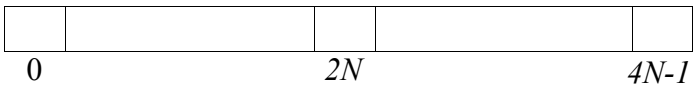
logisch unterteilt in $N-1$ Tripel $(j, j+N, j+2N)$ für $j = 1, 2, \ldots, N-1$
und ein Quadrupel $(0, N, 2N, 3N)$

- nachfolgend werden für $j = 1, 2, \ldots, N-1$ die Tripel $(j, j+N, j+2N)$ jeweils um die Seite $j+3N$ expandiert, wobei ca. $1/4$ der Sätze aus den Seiten $j, j+N, j+2N$ umgespeichert werden.

Resultat:

- die Datei hat sich auf $4N$ Seiten verdoppelt:

Datei mit $4N$ Seiten



→ dies ist die neue Ausgangssituation für die nächste Verdoppelung der Datei.

Analyse der partiellen Expansionen

Für das lineare Hashing gilt während der 1. bzw. 2. partiellen Expansion:

➞ Der durchschnittliche Belegungsfaktor einer bereits gesplitteten Seite beträgt $2/3$ bzw. $3/4$ des durchschnittlichen Belegungsfaktors einer noch nicht gesplitteten Seite.
(*verglichen mit dem Faktor* $1/2$ *ohne partielle Expansionen (* $n_0 = 1$*)*)

Kontrollfunktion

Larson schlägt vor, als Kontrollfunktion die *Speicherplatzausnutzung* zu wählen.

$$\text{Speicherplatzausnutzung} = \frac{\text{Anzahl der Schlüssel}}{\text{Gesamtkapazität von Primär- und Überlaufseiten}}$$

Expansionsregel

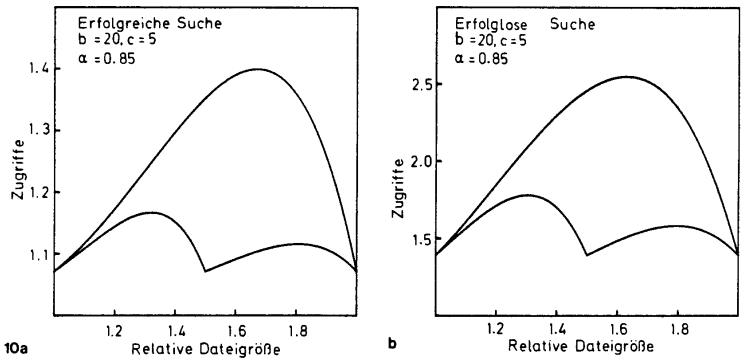
Bei jeder Einfügung wird die Speicherplatzausnutzung überprüft. Falls ihr Wert größer ist als ein vorgegebener Schwellenwert α, $0 < \alpha < 1$
➞ die Datei wird um eine weitere Seite expandiert.

➞ die Speicherplatzausnutzung liegt annähernd konstant bei α, falls nur Einfügungen auftreten.

Leistungsverhalten

Sei *b* die Kapazität einer Primärseite und *c* die Kapazität einer Überlaufseite.

Graphik ➞ Anzahl benötigter Seitenzugriffe bei erfolgreicher bzw. bei erfolgloser Suche für $n_0 = 1$ und $n_0 = 2$.



Beobachtungen:

- die Effizienz ist dann am größten, wenn die Belegung aller Ketten nahezu gleich ist.
- zyklisches Verhalten ➞ Länge eines Zyklus = 2 · Länge des vorangegangenen Zyklus.

Vergleich: durchschnittliche Anzahl benötigter Zugriffe für $n_0 = 1$, $n_0 = 2$ und $n_0 = 3$ mit den zugrundeliegenden Parametern $b = 20$, $c = 5$, $\alpha = 0{,}85$.

	$n_0 = 1$	$n_0 = 2$	$n_0 = 3$
Erfolgreiche Suche	1,27	1,12	1,09
Erfolglose Suche	2,12	1,58	1,48
Einfügen	3,57	3,21	3,31
Entfernen	4,04	3,53	3,56

Folgerung:

$n_0 = 2$ partielle Expansionen stellen den besten Kompromiß dar zwischen

- Effizienz des Suchens,
- Effizienz der Update-Operationen (Einfügen und Entfernen) und
- Komplexität des Programmcodes