2. Hashing

- Einführung
- Hash-Funktionen
  - Divisionsrest-Verfahren
  - Faltung
  - Mid-Square-Methode, . . .
- Behandlung von Kollisionen
  - Verkettung der Überläufer
  - Offene Hash-Verfahren: lineares Sondieren, quadratisches Sondieren, ...
- Analyse des Hashing
- Hashing auf Externspeichern
  - Bucket-Adressierung mit separaten Überlauf-Buckets
  - Analyse
- Dynamische Hash-Verfahren
  - Erweiterbares Hashing
  - Lineares Hashing

Einführung

- Gibt es bessere Strukturen für direkte Suche für Haupt- und Externspeicher ?
  - AVL-Baum: $O(\log_2 n)$ Vergleiche
  - B*-Baum: E/A-Kosten $O(\log_{k^*}(n))$, vielfach 3 Zugriffe
- Bisher:
  - Suche über Schlüsselvergleich
  - Allokation des Satzes als physischer Nachbar des "Vorgängers" oder beliebige Allokation und Verküpfung durch Zeiger
- Gestreute Speicherungsstrukturen / Hashing
  (Schlüsseltransformation, Adreßberechnungsverfahren, scatter-storage technique usw.)
  - Berechnung der Satzadresse SA(i) aus Satzschlüssel K_i --> **Schlüsseltransformation**
  - Speicherung des Satzes bei SA (i)
  - Ziele: schnelle direkte Suche + Gleichverteilung der Sätze (möglichst wenig Synonyme)

Einführung (2)

- Definition:

  S sei Menge aller möglichen Schlüsselwerte eines Satztyps (Schlüsselraum)

  A = {0,1,...,m-1} sei Intervall der ganzen Zahlen von 0 bis m-1 zur Adressierung eines Arrays bzw. einer Hash-Tabelle mit m Einträgen

  Eine Hash-Funktion $h : S \rightarrow A$

  ordnet jedem Schlüssel $s \in S$ des Satztyps eine Zahl aus A als Adresse in der Hash-Tabelle zu.
- Idealfall:
  1 Zugriff zur direkten Suche
- Problem: Kollisionen

Beispiel: m=10

h (s) = s mod 100

	Schlüssel	Daten
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Perfektes Hashing: Direkte Adressierung

- Idealfall (perfektes Hashing): keine Kollisionen
  - h ist eine injektive Funktion.
  - Für jeden Schlüssel aus S muß Speicherplatz bereitgehalten werden, d. h., die Menge aller möglichen Schlüssel ist bekannt.
- Parameter

  l = Schlüssellänge, b = Basis, m = #Speicherplätze

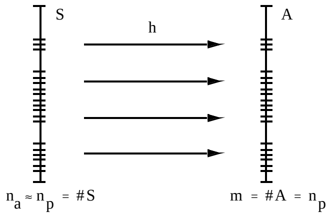
  n_p = #S = b^l mögliche Schlüssel

  n_a = #K = # vorhandene Schlüssel

  Wenn K bekannt ist und K fest bleibt, kann leicht eine injektive Abbildung

  $h: K \rightarrow \{0, . . . , m-1\}$

  z. B. wie folgt berechnet werden:



  - Die Schlüssel in K werden lexikographisch geordnet und auf ihre Ordnungsnummern abgebildet oder
  - Der Wert eines Schlüssels K_i oder eine einfache ordnungserhaltende Transformation dieses Wertes (Division/Multiplikation mit einer Konstanten) ergibt die Adresse: $A_i = h(K_i) = K_i$

Direkte Adressierung (2)

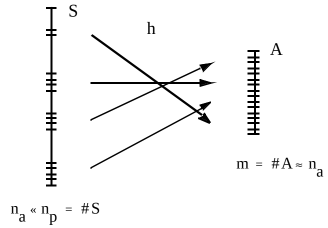
- Beispiel: Schlüsselmenge {00, . . . , 99}

	Schlüssel	Daten
00		
01	01	D01
02	02	D02
03		
04	04	D04
05	05	D05
⋮		
95		
96	96	D96
97		
98		
99	99	D99

- Eigenschaften
  - Statische Zuordnung des Speicherplatzes
  - Kosten für direkte Suche und Wartung ?
  - Reihenfolge beim sequentiellen Durchlauf ?
- Bestes Verfahren bei geeigneter Schlüsselmenge K, aber aktuelle Schlüsselmenge K ist oft nicht "dicht":
  - eine 9-stellige Sozialversicherungsnummer bei 10^5 Beschäftigten
  - Namen / Bezeichner als Schlüssel (Schlüssellänge k):

Allgemeines Hashing

- Annahmen
  - Die Menge der möglichen Schlüssel ist meist sehr viel größer als die Menge der verfügbaren Speicheradressen
  - h ist nicht injektiv



- Definitionen:
  - Zwei Schlüssel K_i, $K_j \in K$ kollidieren (bzgl. einer Hash-Funktion h) gdw. h (K_i) = h (K_j).
  - Tritt für K_i und K_j eine Kollision auf, so heißen diese Schlüssel Synonyme.
  - Die Menge der Synonyme bezüglich einer Speicheradresse A_i heißt Kollisionsklasse.
- Geburtstags-Paradoxon

k Personen auf einer Party haben gleichverteilte und stochastisch unabhängige Geburtstage.
Mit welcher Wahrscheinlichkeit p (n, k) haben mindestens 2 von k Personen am gleichen Tag (n = 365) Geburtstag?
Die Wahrscheinlichkeit, daß keine Kollision auftritt, ist

$$q(n,k) = \frac{\text{ZahldergünstigenFälle}}{\text{ZahldermöglichenFälle}} = \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \ldots \cdot \frac{n-k}{n} = \frac{(n-1) \cdot \ldots \cdot (n-k)}{n^k}$$

Es ist p (365, k) = 1 - q (365, k) > 0.5 für k

- Behandlung von Kollisionen erforderlich !

Hash-Verfahren: Einflußfaktoren

- Leistungsfähigkeit eines Hash-Verfahrens: Einflußgrößen und Parameter
  - Hash-Funktion
  - Datentyp des Schlüsselraumes: Integer, String, ...
  - Verteilung der aktuell benutzten Schlüssel
  - Belegungsgrad der Hash-Tabelle HT
  - Anzahl der Sätze, die sich auf einer Adresse speichern lassen, ohne Kollision auszulösen (Bucket-Kapazität)
  - Technik zur Kollisionsauflösung
  - ggf. Reihenfolge der Speicherung der Sätze (auf Hausadresse zuerst!)
- Belegungsfaktor der Hash-Tabelle
  - Verhältnis von aktuell belegten zur gesamten Anzahl an Speicherplätzen $\beta = n_a / m$
  - für $\beta \geq 0{,}85$ erzeugen alle Hash-Funktionen viele Kollisionen und damit hohen Zusatzaufwand
  - Hash-Tabelle ausreichend groß zu dimensionenieren (m > n_a)
- Für die Hash-Funktion h gelten **folgende Forderungen:**
  - Sie soll sich einfach und effizient berechnen lassen (konstante Kosten)
  - Sie soll eine möglichst gleichmäßige Belegung der Hash-Tabelle HT erzeugen, auch bei ungleich verteilten Schlüsseln
  - Sie soll möglichst wenige Kollisionen verursachen

Hash-Funktionen (2)

1. Divisionsrest-Verfahren (kurz: Divisions-Verfahren): h (K_i) = K_i mod q, (q ~ m)

⇒ Der entstehende Rest ergibt die relative Adresse in HT

- Beispiel:

Die Funktion nat wandle Namen in natürliche Zahlen um:
nat(Name) = ord (1. Buchstabe von Name) - ord ('A')

h (Name) = nat (Name) mod m

HT: m=10

	Schlüssel	Daten
0		
1	BOHR	D1
2	CURIE	D2
3	DIRAC	D3
4	EINSTEIN	D4
5	PLANCK	D5
6		
7	HEISENBERG	D7
8	SCHRÖDINGER	D8
9		

- Wichtigste Forderung an Divisor q:
  q = Primzahl (größte Primzahl <= m)
  - Hash-Funktion muß etwaige Regelmäßigkeiten in Schlüsselverteilung eliminieren, damit nicht ständig die gleichen Plätze in HT getroffen werden
  - Bei äquidistantem Abstand der Schlüssel $K_i + j \cdot \Delta K$, j = 0, 1, 2, ... maximiert eine Primzahl die Distanz, nach der eine Kollision auftritt. Eine Kollision ergibt sich, wenn

    K_i mod q = $(K_i + j \cdot \Delta K)$ mod q oder $j \cdot \Delta K = k \cdot q$, k = 1, 2, 3, ...
  - Eine Primzahl kann keine gemeinsamen Faktoren mit ΔK besitzen, die den Kollisionsabstand verkürzen würden

Hash-Funktionen (3)

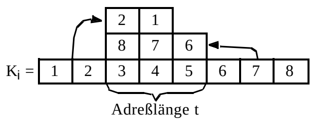
2. Faltung

- Schlüssel wird in Teile zerlegt, die bis auf das letzte die Länge einer Adresse für HT besitzen
- Schlüsselteile werden dann übereinandergefaltet und addiert.

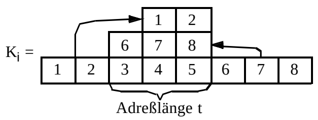
■ Variationen:

- Rand-Falten: wie beim Falten von Papier am Rand
- Shift-Falten: Teile des Schlüssels werden übereinandergeschoben
- Sonstige: z.B. EXOR-Verknüpfung bei binärer Zeichendarstellung
- Beispiel: b = 10, t = 3, m = 10^3

Rand-Falten



Shift-Falten



■ Faltung

- verkürzt lange Schlüssel auf "leicht berechenbare" Argumente, wobei alle Schlüsselteile Beitrag zur Adreßberechnung liefern
- diese Argumente können dann zur Verbesserung der Gleichverteilung mit einem weiteren Verfahren "gehasht" werden

Hash-Funktionen (4)

3. Mid-Square-Methode

- Schlüssel K_i wird quadriert. t aufeinanderfolgende Stellen werden aus der Mitte des Ergebnisses für die Adressierung ausgewählt.
- Es muß also $b^t = m$ gelten.
- mittlere Stellen lassen beste Gleichverteilung der Werte erwarten
- Beispiel für b = 2, t = 4, m = 16: $K_i = 1100100$ $K_i^2 = 10011\underbrace{1000}_{t}10000 \rightarrow h(K_i) = 1000$

4. Zufallsmethode:

- K_i dient als Saat für Zufallszahlengenerator

5. Ziffernanalyse:

- setzt Kenntnis der Schlüsselmenge K voraus. Die t Stellen mit der besten Gleichverteilung der Ziffern oder Zeichen in K werden von K_i zur Adressierung ausgewählt

Hash-Funktionen: Bewertung

■ Verhalten / Leistungsfähigkeit einer Hash-Funktion hängt von der gewählten Schlüsselmenge ab

- Deshalb lassen sie sich auch nur unzureichend theoretisch oder mit Hilfe von analytischen Modellen untersuchen
- Wenn eine Hash-Funktion gegeben ist, läßt sich immer eine Schlüsselmenge finden, bei der sie **besonders viele Kollisionen** erzeugt
- **Keine Hash-Funktion** ist immer besser als alle anderen

■ Über die Güte der verschiedenen Hash-Funktionen liegen jedoch eine Reihe von empirischen Untersuchungen vor

- Das **Divisionsrest-Verfahren** ist im Mittel am leistungsfähigsten; für bestimmte Schlüsselmengen können jedoch andere Techniken besser abschneiden
- Wenn die Schlüsselverteilung nicht bekannt ist, dann ist das Divisionsrest-Verfahren die bevorzugte Hash-Technik
- Wichtig dabei: ausreichend große Hash-Tabelle, Verwendung einer Primzahl als Divisor

Behandlung von Kollisionen

■ Zwei Ansätze, wenn $h(K_q) = h(K_p)$

- K_p wird in einem separaten Überlaufbereich (außerhalb der Hash-Tabelle) zusammen mit allen anderen Überläufern gespeichert; Verkettung der Überläufer
- Es wird für K_p ein freier Platz innerhalb der Hash-Tabelle gesucht („Sondieren“); alle Überläufer werden im Primärbereich untergebracht („offene Hash-Verfahren“)

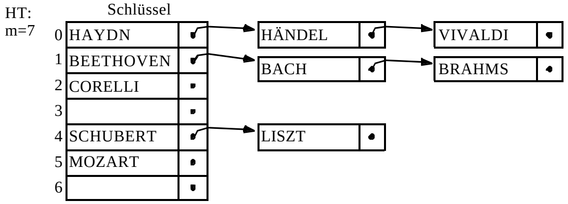
■ Methode der Kollisionsauflösung entscheidet darüber, welche Folge und wieviele relative Adressen zur Ermittlung eines freien Platzes aufgesucht werden

■ Adreßfolge bei Speicherung und Suche für Schlüssel K_p sei $h_0(K_p), h_1(K_p), h_2(K_p), \ldots$

- Bei einer Folge der Länge n treten also n-1 Kollisionen auf
- **Primärkollision:** $h(K_p) = h(K_q)$
- **Sekundärkollision:** $h_i(K_p) = h_j(K_q)$, $i \neq j$

Hash-Verfahren mit Verkettung der Überläufer (separater Überlaufbereich)

- Dynamische Speicherplatzbelegung für Synonyme
  - Alle Sätze, die nicht auf ihrer Hausadresse unterkommen, werden in einem separaten Bereich gespeichert (Überlaufbereich)
  - Verkettung der Synonyme (Überläufer) pro Hash-Klasse
  - Suchen, Einfügen und Löschen sind auf Kollisionsklasse beschränkt
  - Unterscheidung nach Primär- und Sekundärbereich: n > m ist möglich !



- Entartung zur linearen Liste prinzipiell möglich
- Nachteil: Anlegen von Überläufern, auch wenn Hash-Tabelle (Primärbereich) noch wenig belegt ist

Java-Realisierung

```
/** Einfacher Eintrag in Hash-Tabelle */
class HTEntry {
  Object key;
  Object value;
  /** Konstruktor */
  HTEntry (Object key, Object value) {
    this.key = key; this.value = value; } }

/** Abstrakte Basisklasse für Hash-Tabellen */
public abstract class HashTable {
  protected HTEntry[] table;
  /** Konstruktor */
  public HashTable (int capacity) { table = new HTEntry[capacity]; }
  /** Die Hash-Funktion */
  protected int h(Object key) {
    return (key.hashCode() & 0x7fffffff) % table.length; }
  /** Einfuegen eines Schluessel-Wert-Paares */
  public abstract boolean add(Object key, Object value);
  /** Test ob Schluessel enthalten ist */
  public abstract boolean contains(Object key);
  /** Abrufen des einem Schluessel zugehoerigen Wertes */
  public abstract Object get(Object key);
  /** Entfernen eines Eintrags */
  public abstract void remove(Object key); }
```



```
/** Eintrag in Hash-Tabelle mit Zeiger für verkettete Ueberlaufbehandlung */
class HTLinkedEntry extends HTEntry {
  HTLinkedEntry next;
  /** Konstruktor */
  HTLinkedEntry (Object key, Object value) { super(key, value); } }

/** Hash-Tabelle mit separater (verketteter) Ueberlaufbehandlung */
public class LinkedHashTable extends HashTable {

  /** Konstruktor */
  public LinkedHashTable (int capacity) { super(capacity); }

  /** Einfuegen eines Schluessel-Wert-Paares */
  public boolean add(Object key, Object value) {
    int pos = h(key);         // Adresse in Hash-Tabelle fuer Schluessel
    if (table[pos] == null)  // Eintrag frei?
      table[pos] = new HTLinkedEntry(key, value);
    else {                   // Eintrag belegt -> Suche Eintrag in Kette
      HTLinkedEntry entry = (HTLinkedEntry) table[pos];
      while((entry.next != null) && (! entry.key.equals(key)))
        entry = entry.next;
      if (entry.key.equals(key)) // Schluessel existiert schon
        entry.value = value;
      else                     // fuege neuen Eintrag am Kettenende an
        entry.next = new HTLinkedEntry(key, value); }
    return true; }
```



```
  /** Test ob Schluessel enthalten ist */
  public boolean contains(Object key) {
    HTLinkedEntry entry = (HTLinkedEntry) table[h(key)];
    while((entry != null) && (! entry.key.equals(key)))
      entry = entry.next;
    return entry != null;
  }

  /** Abrufen des einem Schluessel zugehoerigen Wertes */
  public Object get(Object key) {
    HTLinkedEntry entry = (HTLinkedEntry) table[h(key)];
    while((entry != null) && (! entry.key.equals(key)))
      entry = entry.next;
    if (entry != null)
      return entry.value;
    return null;
  }
  ...
}
```

Offene Hash-Verfahren: Lineares Sondieren

- Offene Hash-Verfahren
  - Speicherung der Synomyme (Überläufer) im Primärbereich
  - Hash-Verfahren muß in der Lage sein, eine Sondierungsfolge, d.h. eine Permutation aller Hash-Adressen, zu berechnen
- Lineares Sondieren (linear probing)

Von der Hausadresse (Hash-Funktion h) aus wird sequentiell (modulo der Hash-Tabellen-Größe) gesucht. Offensichtlich werden dabei alle Plätze in HT erreicht:

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = (h_0(K_p) + i) \bmod m \quad , i = 1, 2, \ldots$$

m=10, h (K) = K mod m

	Schlüssel
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

- Beispiel: Einfügereihenfolge 79, 28, 49, 88, 59
  - Häufung von Kollisionen durch „Klumpenbildung“

  ⇒ lange Sondierungsfolgen möglich

Java-Realisierung

- Suche in einer Hash-Tabelle bei linearem Sondieren

```
/** Hash-Tabelle mit Ueberlaufbehandlung im Primaerbereich (Sondieren) */
public class OpenHashTable extends HashTable {

  protected static final int EMPTY = 0;      // Eintrag ist leer
  protected static final int OCCUPIED = 1;   // Eintrag belegt
  protected static final int DELETED = 2;    // Eintrag geloescht

  protected int[] flag;  // Markierungsfeld; enthaelt Eintragsstatus

  /** Konstruktor */
  public OpenHashTable (int capacity) {
    super(capacity);
    flag = new int[capacity];
    for (int i=0; i<capacity; i++)  // initialisiere Markierungsfeld
      flag[i] = EMPTY;
  }

  /** (Lineares) Sondieren. Berechnet aus aktueller Position die naechste.*/
  protected int s(int pos) {
    return ++pos % table.length;
  }
```



```
  /** Abrufen des einem Schluessel zugehoerigen Wertes */
  public Object get(Object key) {
    int pos, startPos;
    startPos = pos = h(key);  // Adresse in Hash-Tabelle fuer Schluessel
    while((flag[pos] != EMPTY) && (! table[pos].key.equals(key))) {
      pos = s(pos);                        // ermittle naechste Position
      if (pos == startPos) return null;    // Eintrag nicht gefunden
    }
    if (flag[pos] == OCCUPIED)
      // Schleife verlassen, da Schluessel gefunden; Eintrag als belegt
      // markiert
      return table[pos].value;
    // Schleife verlassen, da Eintrag leer oder
    // Eintrag gefunden, jedoch als geloescht markiert
    return null;
  }
  ...
}
```


Lineares Sondieren (2)

- Aufwendiges Löschen
  - impliziert oft Verschiebungen
  - entstehende Lücken in Suchsequenzen sind aufzufüllen, da das Antreffen eines freien Platzes die Suche beendet.

m=10, h (K) = K mod m

	Schlüssel
0	49
1	88
2	59
3	
4	
5	
6	
7	
8	28
9	79

Lösche ⇒ 28

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

- Verbesserung: Modifikation der Überlauffolge

$$h_0(K_p) = h(K_p)$$

$$h_i(K_p) = (h_{i-1}(K_p) + f(i)) \bmod m \qquad \text{oder}$$

$$h_i(K_p) = (h_{i-1}(K_p) + f(i, h(K_p))) \bmod m \quad , \ i = 1, 2, \ldots$$

- Beispiele:
  - Weiterspringen um festes Inkrement c (statt nur 1): f (i) = c * i
  - Sondierung in beiden Richtungen: f (i) = c * i * $(-1)^i$

Quadratisches Sondieren

- Bestimmung der Speicheradresse

$$h_0(K_p) = h(K_p) \qquad h_i(K_p) = (h_0(K_p) + a \cdot i + b \cdot i^2) \bmod m \quad , \; i = 1, 2, \ldots$$

- m sollte Primzahl sein

- Folgender **Spezialfall** sichert Erreichbarkeit aller Plätze:

$$h_0(K_p) = h(K_p) \qquad h_i(K_p) = \left(h_0(K_p) - \left(\left\lceil \frac{i}{2} \right\rceil\right)^2 (-1)^i\right) \bmod m \qquad 1 \le i \le m-1$$

- Beispiel:
  Einfügereihenfolge 79, 28, 49, 88, 59

m=10,
h (K) = K mod m

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Weitere offene Hash-Verfahren

- Sondieren mit Zufallszahlen

Mit Hilfe eines deterministischen Pseudozufallszahlen-Generators wird die Folge der Adressen [1 .. m-1] mod m genau einmal erzeugt:

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = (h_0(K_p) + z_i) \bmod m \; , i = 1, 2 \ldots$$

- Double Hashing

Einsatz einer zweiten Funktion für die Sondierungsfolge

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = (h_0(K_p) + i \cdot h'(K_p)) \bmod m \; , \; i = 1, 2, \ldots$$

Dabei ist h’(K) so zu wählen, daß für alle Schlüssel K die resultierende Sondierungsfolge eine Permutation aller Hash-Adressen bildet

- Kettung von Synonymen
  - explizite Kettung aller Sätze einer Kollisionsklasse
  - verringert nicht die Anzahl der Kollisionen; sie verkürzt jedoch den Suchpfad beim Aufsuchen eines Synonyms.
  - Bestimmung eines freien Überlaufplatzes (Kollisionsbehandlung) mit beliebiger Methode

Analyse des Hashing

- Kostenmaße
  - β = n/m: Belegung von HT mit n Schlüsseln
  - S_n = # der Suchschritte für das Auffinden eines Schlüssels - entspricht den Kosten für erfolgreiche Suche und Löschen (ohne Reorganisation)
  - U_n = # der Suchschritte für die erfolglose Suche - das Auffinden des ersten freien Platzes entspricht den Einfügekosten

- Grenzwerte

best case:
$S_n = 1$
$U_n = 1.$

worst case:
$S_n = n$
$U_n = n+1.$

- Modell für das lineare Sondieren
  - Sobald β eine gewisse Größe überschreitet, verschlechtert sich das Zugriffsverhalten sehr stark.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

  - Je länger eine Liste ist, umso schneller wird sie noch länger werden.
  - Zwei Listen können zusammenwachsen (Platz 3 und 14), so daß durch neue Schlüssel eine Art Verdopplung der Listenlänge eintreten kann

⇒ Ergebnisse für das lineare Sondieren nach Knuth:

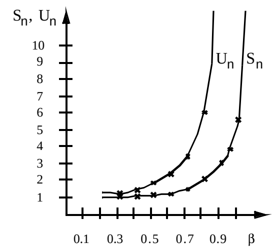
$$S_n \approx 0{,}5\left(1 + \frac{1}{1-\beta}\right) \quad \text{mit} \quad 0 \le \beta = \frac{n}{m} < 1 \qquad U_n \approx 0{,}5\left(1 + \frac{1}{(1-\beta)^2}\right)$$

Analyse des Hashing (2)

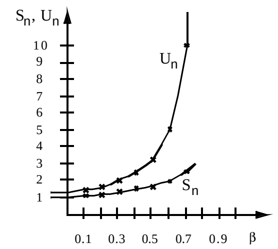
- Abschätzung für offene Hash-Verfahren mit optimierter Kollisionsbehandlung (gleichmäßige HT-Verteilung von Kollisionen)

$$S_n \sim -\frac{1}{\beta} \cdot \ln(1-\beta) \qquad U_n \sim \frac{1}{1-\beta}$$

- Anzahl der Suchschritte in HT



a) bei linearem Sondieren



a) bei “unabhängiger” Kollisionsauflösung

Analyse des Hashing (3)

- Modell für separate Überlaufbereiche
  - Annahme: n Schlüssel verteilen sich gleichförmig über die m mögl. Ketten.
  - Jede Synonymkette hat also im Mittel n/m = β Schlüssel
  - *Erfolgreiche Suche:* wenn der i-te Schlüssel K_i in HT eingefügt wird, sind in jeder Kette (i-1)/m Schlüssel. Die Suche nach K_i kostet also 1+(i-1)/m Schritte, da K_i an das jeweilige Ende einer Kette angehängt wird.
    Erwartungswert für erfolgreiche Suche: $S_n = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(1 + \frac{i-1}{m}\right) = 1 + \frac{n-1}{2 \cdot m} \approx 1 + \frac{\beta}{2}$
  - *Erfolglosen Suche:* es muß immer die ganze Kette durchlaufen werden
    U_n = 1 + 1 · WS (zu einer Hausadresse existiert 1 Überläufer)+ 2 · WS (zu Hausadresse existieren 2 Überläufer) + 3 ...

    $U_n \approx \beta - e^{-\beta}$.

β	0.5	0.75	1	1.5	2	3	4	5
S_n	1.25	1.37	1.5	1.75	2	2.5	3	3.5
U_n	1.11	1.22	1.37	1.72	2.14	3.05	4.02	5.01

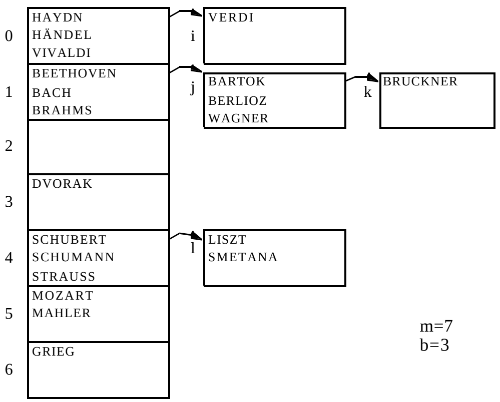
- Separate Kettung ist auch der “unabhängigen” Kollisionsauflösung überlegen
- Hashing ist i. a. sehr leistungsstark. Selbst bei starker Überbelegung (β>1) erhält man bei separater Kettung noch günstige Werte

Hashing auf Externspeichern

- Hash-Adresse bezeichnet Bucket (hier: Seite)
  - Kollisionsproblem wird entschärft, da mehr als ein Satz auf seiner Hausadresse gespeichert werden kann
  - Bucket-Kapazität b -> Primärbereich kann bis zu b*m Sätze aufnehmen !
- Überlaufbehandlung
  - Überlauf tritt erst beim (b+1)-ten Synonym auf
  - alle bekannten Verfahren sind möglich, aber lange Sondierungsfolgen im Primärbereich sollten vermieden werden
  - häufig Wahl eines separaten Überlaufbereichs mit dynamischer Zuordnung der Buckets
- Speicherungsreihenfolge im Bucket
  - ohne Ordnung (Einfügefolge)
  - nach der Sortierfolge des Schlüssels: aufwendiger, jedoch Vorteile beim Suchen (sortierte Liste!)
- Bucket-Größe meist Seitengröße (Alternative: mehrere Seiten / Spur einer Magnetplatte)
  - Zugriff auf die Hausadresse bedeutet 1 physische E/A
  - jeder Zugriff auf ein Überlauf-Bucket löst weiteren physischen E/A-Vorgang aus

Hashing auf Externspeichern (2)

- Bucket-Adressierung mit separaten Überlauf-Buckets
  - weithin eingesetztes Hash-Verfahren für Externspeicher
  - jede Kollisionsklasse hat eine separate Überlaufkette.



- Klassifikation

	Primär-Bucket	Überlauf-Bucket
inneres Bucket	0, 1, 4	j
Rand-Bucket	2, 3, 5, 6	i, k, l

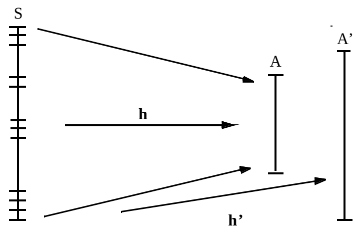
Hashing auf Externspeichern (3)

- Grundoperationen
  - direkte Suche: nur in der Bucket-Kette
  - sequentielle Suche ?
  - Einfügen: ungeordnet oder sortiert
  - Löschen: keine Reorganisation in der Bucket-Kette - leere Überlauf-Buckets werden entfernt
- Kostenmodelle sehr komplex
- Belegungsfaktor:
  $\beta = n / (b \cdot m)$
  - bezieht sich auf Primär-Buckets (**kann größer als 1 werden!)**
- **Zugriffsfaktoren**
  - Gute Annäherung an idealen Wert
  - Bei Vergleich mit Mehrwegbäumen ist zu beachten, daß Hash-Verfahren sortiert sequentielle Verarbeitung aller Sätze nicht unterstützen. Außerdem stellen sie statische Strukturen dar. Die Zahl der Primär-Buckets m läßt sich nicht dynamisch an die Zahl der zu speichernden Sätze n anpassen.

b \ β		0.5	0.75	1.0	1.25	1.5	1.75	2.0
b = 2	S_n	1.10	1.20	1.31	1.42	1.54	1.66	1.78
	U_n	1.08	1.21	1.38	1.58	1.79	2.02	2.26
b = 5	S_n	1.02	1.08	1.17	1.28	1.40	1.52	1.64
	U_n	1.04	1.17	1.39	1.64	1.90	2.15	2.40
b = 10	S_n	1.00	1.03	1.12	1.24	1.36	1.47	1.59
	U_n	1.01	1.13	1.41	1.72	1.96	2.19	2.44
b = 20	S_n	1.00	1.01	1.08	1.21	1.34	1.45	1.56
	U_n	1.00	1.08	1.44	1.81	1.99	2.17	2.45
b = 30	S_n	1.00	1.00	1.05	1.20	1.33	1.43	1.54
	U_n	1.00	1.02	1.46	1.93	2.00	2.08	2.47

Dynamische Hash-Verfahren

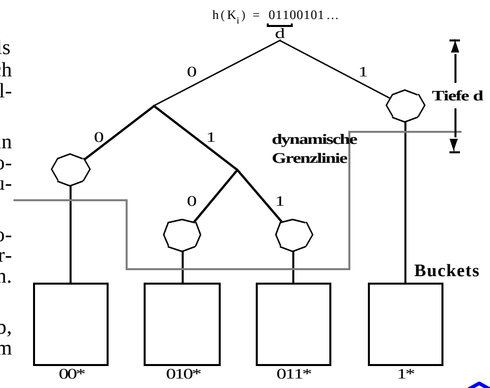
- Wachstumsproblem bei statischen Verfahren
  - Statische Allokation von Speicherbereichen: Speicherausnutzung?
  - Bei Erweiterung des Adreßraumes: Re-Hashing
    - ⇒ Alle Sätze erhalten eine **neue Adresse**



  - Probleme: Kosten, Verfügbarkeit, Adressierbarkeit
- Entwurfsziele
  - Eine im Vergleich zum statischen Hashing dynamische Struktur , die Wachstum und Schrumpfung des Hash-Bereichs (Datei) erlaubt
  - Keine Überlauftechniken
  - Zugriffsfaktor ≤ 2 für die direkte Suche
- Viele konkurrierende Ansätze
  - Extendible Hashing (Fagin et al., 1978)
  - Virtual Hashing und Linear Hashing (Litwin, 1978, 1980)
  - Dynamic Hashing (Larson, 1978)

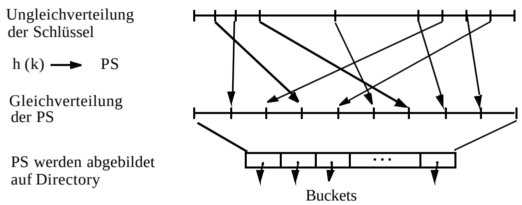
Erweiterbares Hashing

- Kombination mehrerer Ideen
  - Dynamik von B-Bäumen (Split- und Mischtechniken von Seiten) zur Konstruktion eines dynamischen Hash-Bereichs
  - Adressierungstechnik von Digitalbäumen zum Aufsuchen eines Speicherplatzes
  - Hashing: gestreute Speicherung mit möglichst gleichmäßiger Werteverteilung
- Prinzipielle Vorgehensweise
  - Die einzelnen Bits eines Schlüssels steuern der Reihe nach den Weg durch den zur Adressierung benutzten Digitalbaum $K_i = (b_0, b_1, b_2, \ldots)$.
  - Verwendung der Schlüsselwerte kann bei Ungleichverteilung zu unausgewogenem Digitalbaum führen (Digitalbäume kennen keine Höhenbalancierung)
  - Verwendung von $h(K_i)$ als sog. Pseudoschlüssel (PS) soll bessere Gleichverteilung gewährleisten.
    $h(K_i) = (b_0, b_1, b_2, \ldots)$
  - Digitalbaum-Adressierung bricht ab, sobald ein Bucket den ganzen Teilbaum aufnehmen kann



Erweiterbares Hashing (2)

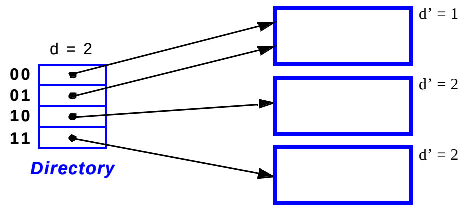
- Prinzipielle Abbildung der Pseudoschlüssel
  - Zur Adressierung eines Buckets sind d Bits erforderlich, wobei sich dafür i. a. eine dynamische Grenzlinie variierender Tiefe ergibt.
  - ausgeglichener Digitalbaum garantiert minimales d_{max}
  - Hash-Funktion soll möglichst Gleichverteilung der Pseudoschlüssel erreichen (minimale Höhe des Digitalbaumes, minimales d_{max})



- dynamisches Wachsen und Schrumpfen des Hash-Bereiches
  - Buckets werden erst bei Bedarf bereitgestellt: kein statisch dimensionierter Primärbereich, keine Überlauf-Buckets
  - nur belegte Buckets werden gespeichert
  - hohe Speicherplatzbelegung möglich

Erweiterbares Hashing (3)

- schneller Zugriff über *Directory* (Index)
  - binärer Digitalbaum der Höhe d wird durch einen Digitalbaum der Höhe 1 implementiert (entarteter Trie der Höhe 1 mit 2^d Einträgen).
  - d wird festgelegt durch den längsten Pfad im binären Digitalbaum.
  - In einem Bucket werden nur Sätze gespeichert, deren Pseudoschlüssel in den ersten d' Bits übereinstimmen (d' = lokale Tiefe).
  - d = MAX (d'): d Bits des PS werden zur Adressierung verwendet (d = globale Tiefe).
  - Directory enthält 2^d Einträge
  - alle Sätze zu einem Eintrag (d Bits) sind in einem Bucket gespeichert; wenn d' < d, können benachbarte Einträge auf dasselbe Bucket verweisen
  - max. 2 Seitenzugriffe

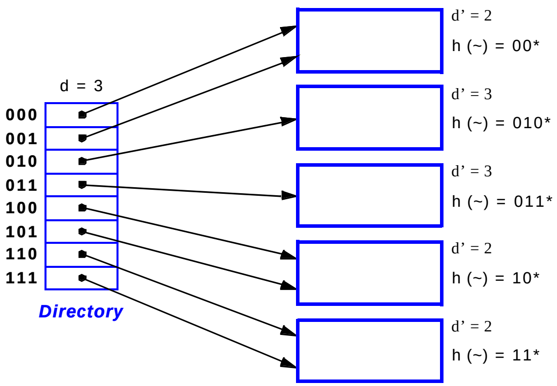


Erweiterbares Hashing: Splitting von Buckets

- Fall 1: Überlauf eines Buckets, dessen lokale Tiefe kleiner ist als globale Tiefe d

  ⇒ lokale Neuverteilung der Daten

  - Erhöhung der lokalen Tiefe
  - lokale Korrektur der Pointer im Directory

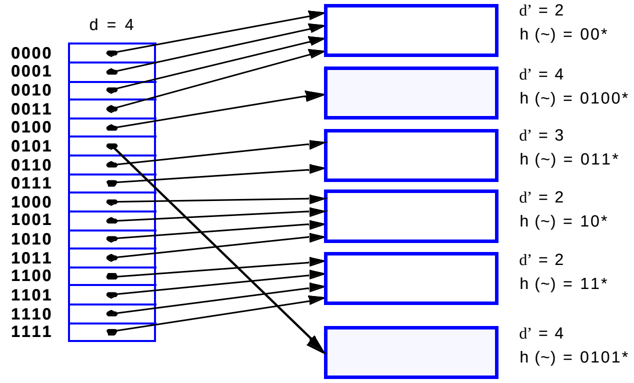


Erweiterbares Hashing: Splitting von Buckets (2)

- Fall 2: Überlauf eines Buckets, dessen lokale Tiefe gleich der globalen Tiefe ist

  ⇒ lokale Neuverteilung der Daten (Erhöhung der lokalen Tiefe)

  - Verdopplung des Directories (Erhöhung der globalen Tiefe)
  - globale Korrektur/Neuverteilung der Pointer im Directory



Lineares Hashing

- Dynamisches Wachsen/Schrumpfen des Hash-Bereiches ohne große Directories
  - inkrementelles Wachstum durch sukzessives Splitten von Buckets in fest vorgegebener Reihenfolge
  - Splitten erfolgt bei Überschreiten eines Belegungsfaktors ß (z.B. 80%)
  - Überlauf-Buckets sind notwendig
- Prinzipieller Ansatz
  - m: Ausgangsgröße des Hash-Bereiches (#Buckets)
  - sukzessives Neuanlegen einzelner Buckets am aktuellen Dateiende, falls Belegungsfaktor ß vorhandener Buckets einen Grenzwert übersteigt (Schrumpfen am aktuellen Ende bei Unterschreiten einer Mindestbelegung)
  - Adressierungsbereich verdoppelt sich bei starkem Wachstum gelegentlich, L=Anzahl vollständig erfolgter Verdoppelungen (Initialwert 0)
  - Größe des Hash-Bereiches: $m * 2^L$
  - Split-Zeiger p (Initialwert 0) zeigt auf nächstes zu splittende Bucket im Hash-Bereich mit $0 <= p < m*2^L$
  - Split führt zu neuem Bucket mit Adresse $p + m*2^L$; p wird um 1 inkrementiert $p:=p+1 \bmod (m*2^L)$
  - wenn p wieder auf 0 gesetzt wird (Verdoppelung des Hash-Bereichs beendet), wird L um 1 erhöht

Lineares Hashing (2)

- Hash-Funktion
  - Da der Hash-Bereich wächst oder schrumpft, ist Hash-Funktion an ihn anzupassen.
  - Folge von Hash-Funktionen $h_0, h_1, \ldots$ mit

    $h_j(k) \in \{0, 1, \ldots, m * 2^j - 1\}$,

    z.B. $h_j(k) = k \bmod m * 2^j$
  - i.a. gilt $h = h_L(k)$
- Adressierung: 2 Fälle möglich
  - h (k) >= p -> Satz ist in Bucket h (k)
  - h (k) < p (Bucket wurde bereits gesplittet):

    Satz ist in Bucket $h_{L+1}(k)$ (d.h. in h(k) oder $h(k) + m*2^L$)
  - gleiche Wahrscheinlichkeit für beide Fälle erwünscht

Lineares Hashing (3)

- Beispiel

m=5, L=0, p=0, Blockungsfaktor B = 4, $h_0(k) = k \bmod 5$, $h_1(k) = k \bmod 10$ ...

Schwellwert $ß_S = 0,8$

p → 0

Primärbereich:

0	1	2	3	4
105	111	512	413	144
790	076	477	243	
335		837		
995		002		

Überlaufsätze:

0	1	2	3	4
055		117		
010				
h_0	h_0	h_0	h_0	h_0

Einfügen von 888 erhöht Belegung auf 17/20=0,85 > ß -> Split-Vorgang

0	1	2	3	4	5
	111	512	413	144	
	076	477	243		
		837			
		002			
		117			
	h_0	h_0	h_0	h_0	

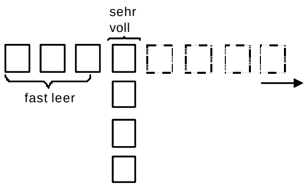
Lineares Hashing (4)

Einfügen von 244, 399, 100 erhöht Belegung auf 20/24=0,83 > ß -> Split-Vorgang

0	1	2	3	4	5
790	111	512	413	144	105
010	076	477	243		335
		837	888		995
		002			055
		117			
		h_0	h_0	h_0	

Lineares Hashing: Bewertung

- Überläufer weiterhin erforderlich
- ungünstiges Split-Verhalten / ungünstige Spleicherplatznutzung möglich (Splitten unterbelegter Seiten)



- Zugriffskosten 1 + x

Zusammenfassung

- Hashing: schnellster Ansatz zur direkten Suche
  - Schlüsseltransformation: berechnet Speicheradresse des Satzes
  - zielt auf bestmögliche Gleichverteilung der Sätze im Hash-Bereich (gestreute Speicherung)
  - anwendbar im Hauptspeicher und für Externspeicher
  - konstante Zugriffskosten O (1)
- Hashing bietet im Vergleich zu Bäumen eingeschränkte Funktionalität
  - i. a. kein sortiert sequentieller Zugriff
  - ordnungserhaltendes Hashing nur in Sonderfällen anwendbar
  - Verfahren sind vielfach statisch
- Idealfall: Direkte Adressierung (Kosten 1 für Suche/Einfügen/Löschen)
  - nur in Ausnahmefällen möglich ('dichte' Schlüsselmenge)
- Hash-Funktion
  - Standard: **Divisionsrest-Verfahren**
  - ggf. zunächst numerischer Wert aus Schlüsseln zu erzeugen
  - Verwendung einer Primzahl für Divisor (Größe der Hash-Tabelle) wichtig

Zusammenfassung (2)

- Kollisionsbehandlung
  - Verkettung der Überläufer (separater Überlaufbereich) i.a. effizienter und einfacher zu realisieren als offene Adressierung
  - ausreichend große Hash-Tabelle entscheidend für Begrenzung der Kollisionshäufigkeit, besonders bei offener Adressierung
  - Belegungsgrad $\beta \leq 0.85$ dringend zu empfehlen
- Hash-Verfahren für Externspeicher
  - reduzierte Kollisionsproblematik, da Bucket b Sätze aufnehmen kann
  - direkte Suche ~ $1 + \delta$ Seitenzugriffe
  - statische Verfahren leiden unter schlechter Speicherplatznutzung und hohen Reorganisationskosten
- Dynamische Hashing-Verfahren: reorganisationsfrei
  - Erweiterbares Hashing: 2 Seitenzugriffe
  - Lineares Hashing: kein Directory, jedoch Überlaufseiten
- Erweiterbares Hashing <u>widerlegt</u> alte „Lehrbuchmeinungen“
  - „Hash-Verfahren sind immer statisch, da sie Feld fester Größe adressieren“
  - „Digitalbäume sind nicht ausgeglichen“
  - „Auch ausgeglichene Suchbäume ermöglichen bestenfalls Zugriffskosten von O (log n)“