Einführung in die Informatik

Hashtables

Hashtabellen

Wolfram Burgard
Cyrill Stachniss

Einleitung

- Wir haben bisher einige der typischen Datenstrukturen zum Speichern von Informationen kennen gelernt
  - Arrays & Vectors
  - Listen
  - Bäume
  - Mengen
- Jede der vorgestellten Datenstrukturen hat Vor- und Nachteile
- Die Applikation bestimmt, welche Technik eingesetzt wird
- In dieser Vorlesung werden wir uns mit so genannten Hashtables (Hashtabellen) beschäftigen, die Vorteile einzelner Verfahren kombinieren

Hashtabellen

- Hashtables stellen effiziente Implementierungen von Mengen dar
- Wichtige Operationen sind
    - **Einfügen (insert)**
    - **Löschen (delete)**
    - **Suchen (search/contains)**

- Ziel der Hashtabellen ist es, diese drei Operation in (durchschnittlich) konstanter Zeit zu realisieren

Idee der Hashtabellen

- Speichert man Elemente in einem Vector so muss man i.A. alle Positionen durchsuchen, um das geeignete Element zu finden
- Wenn man anhand des Elementes selbst auf die Position, an der das Element gespeichert werden soll, schließen kann, lässt sich das Verfahren optimieren
- Effizient wäre dies umzusetzen, wenn sich jedes Objekt eindeutig einem Index in einem Array zuordnen lassen würde
- Einfügen, löschen und suchen: Man muss nur an der entsprechenden Stelle nachsehen und das Element dort einfügen, löschen, oder zurückliefern (Suche)

Datensatz und Schlüssel

- Im Folgenden betrachten wir nicht mehr nur Datensätze (Einfache Referenz auf ein Objekt) sondern betrachten zusätzlich einen Schlüssel
- **Datensatz = <Schlüssel, Informationsteil>**
- Der Schlüssel realisiert eine **eindeutige Identifikation** des Datensatzes

Beispiele für solche Datensätze

- Bank:
  - Schlüssel: Kontonummer
  - Informationsteil: Name, Adresse
- Telefonbuch:
  - Schlüssel: Nachname, Vorname
  - Informationsteil: Telefonnummer

Hashfunktion (1)

- Der Schlüssel liefert die Information, **wo** der Datensatz gespeichert werden soll
- Der Programmierer muss festlegen, welcher Teil des Datensatzes als Schlüssel dient
- Problem: der Programmierer soll sich keine Gedanken über die interne Struktur der Hashtabelle machen müssen
- Die Lösung liefert eine so genannte **Hashfunktion** h

$$h : D \mapsto \{0, 1, \ldots, m-1\}$$

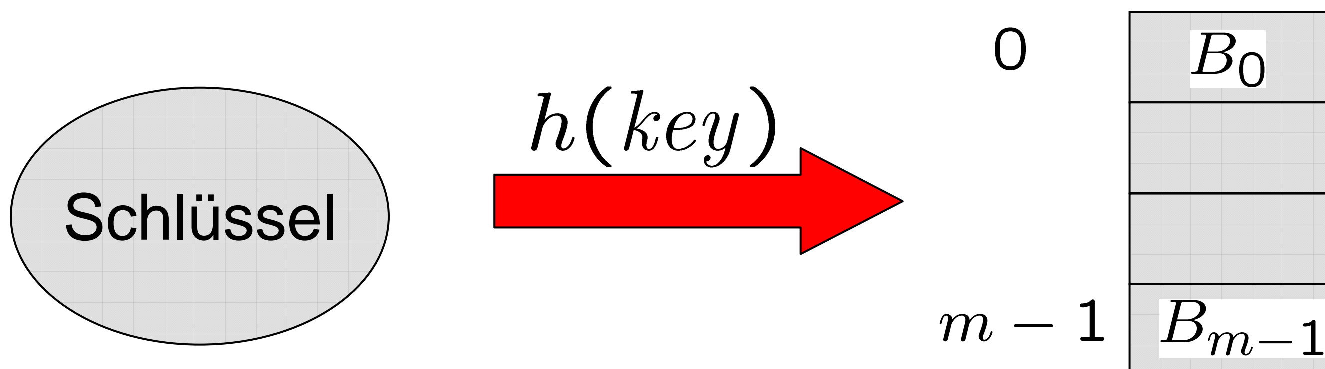
Wertebereich des Schlüssels (D)

Speicherplätze ($\{0, 1, \ldots, m-1\}$)

Hashfunktion (2)

- Wir haben eine **Menge von m Behältern** $B_0, \ldots, B_{m-1}$
- Die Behälter haben die **Adressen** $0, \ldots, m-1$
- Diese Adressen nennt man auch **Hashadressen**
- D sei der Wertebereich des Schlüssels
- h liefert eine **Abbildung von Schlüsseln auf Hashadressen**

$$h : D \mapsto \{0, 1, \ldots, m-1\}$$



Beispiel für eine Hashfunktion

- Wir möchten eine Menge von Bankkunden in einer Hashtabelle speichern
- Schlüssel: Kontonummer (int)
- Informationsteil: Name, Adresse, ...
- Uns stehen 1000 Speicherplätze zur Verfügung
- Die Hashfunktion muss also die Kontonummer auf die Zahlen 0, ..., 999 abbilden (1000 Speicherplätze)

$$h(kto) = kto \bmod m$$

- Empfehlung: m sollte ein Primzahl sein
- **Problem: Es gibt mehr Kontonummern als Speicherplätze**

Gute Hashfunktionen

- Es sollen **möglichst wenige Objekte auf eine Adresse** abgebildet werden
- Werden zwei unterschiedliche Objekte auf eine Hashadresse abgebildet, so spricht man von einer **Kollision**
- Seit M eine Menge mit n Elementen. Eine Hashfunktion heißt **perfekt für M** wenn es keine Kollision gibt ($n \leq m$)
- Gleichzeitig sollte eine gute Hashfunktion effizient berechenbar sein
- Gute Hashfunktionen für verschiedene Schlüssel zu entwerfen ist eine nicht triviale Aufgabe

Hashfunktionen in Java

- Java stellt schon in der Klasse `Object` eine geeignete Hashfunktion bereit
- Die Methode **hashCode()** liefert eine int Zahl für ein beliebiges Object zurück

```
int hashCode() {...}
```

- Um aus dem hashCode eine Hashadresse zu machen, muss man diesen auf **Zahlen zwischen 0 und m-1 beschränken**

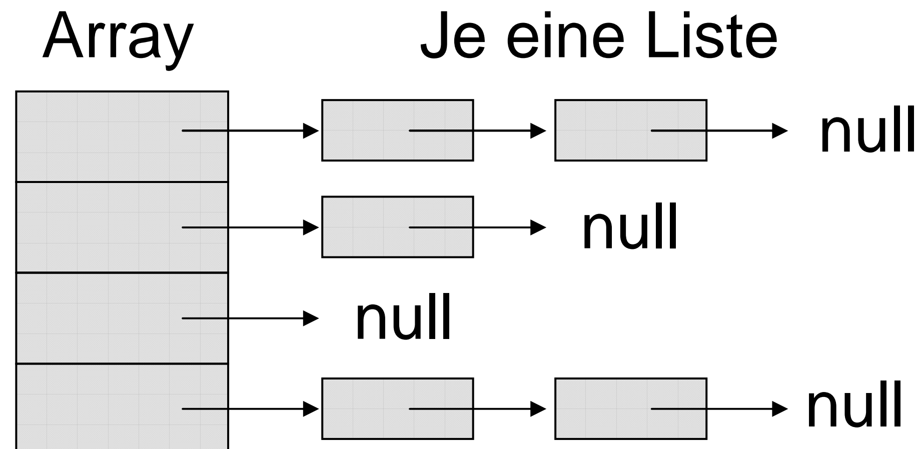
$$h(obj) = obj.\mathsf{hashCode()}\ \mathsf{mod}\ m$$

Skizze für Algorithmen

- Einfügen eines Datensatzes mit Schlüssel k:
  1. addr = h(k)
  2. Füge den Datensatz in Behälter B_{addr} ein
- Löschen eines Datensatzes mit Schlüssel k:
  1. addr = h(k)
  2. Lösche den Datensatz mit Schlüssel k aus dem Behälter B_{addr}
- Suche nach dem Datensatzes mit Schlüssel k:
  1. addr = h(k)
  2. Suche nach dem Datensatz mit Schlüssel k im Behälter B_{addr}

Wie verwaltet man die Behälter?

- Wir müssen ein feste Anzahl von Behältern speichern
- Wir müssen die Behälter **effizient über die Hashadresse (einen Index) ansprechen können**
- In jedem Behälter müssen ggf. mehrere Objekte gespeichert werden (die Anzahl ist unbekannt)
- Wir haben Datenstrukturen kennen gelernt, die dies erlauben
- Ein Array für die Behälter, jeder Behälter ist eine Liste



Realisierung der Hashtabellen-Elemente

```
public class HashtableElement {
    HashtableElement(Object key, Object info) {
        this.key = key;
        this.info = info;
    }

    void setInfo(Object info) {this.info = info;}
    Object getKey() {return this.key;}
    Object getInfo() {return this.info;}

    protected Object key;
    protected Object info;
}
```

Realisierung einer Einfachen Hashtabelle

```
class SimpleHashtable {
   SimpleHashtable(int size) {
      this.data = new HashtableElement[size];
      for (int i=0; i<size; i++)
            this.data[i] = new SingleLinkedList();
   }

   void insert(HashtableElement ds) {...}

   void delete(Object key) {...}

   HashtableElement search(Object key) {...}

   HashtableElement[] data;
}
```

Insert, Delete, and Search

```
void insert(HashtabelElement ds) {
    int addr = ds.getKey().hashCode() % data.length;
    data[addr].insert(ds);
}

void delete(Object key) {
    int addr = key.hashCode() % data.length;
    data[addr].delete(key);
}

HashtableElement search(Object key) {
    int addr = key.hashCode() % data.length;
    return data[addr].search(key);
}
```

Belegungsfaktor

- Die bisherige Implementierung erlaubt eine einfache und effiziente Realisierung
- Es wird allerdings angenommen, dass die Anzahl der zu speichernden Objekte zuvor abgeschätzt werden kann
- Werden zu viele Objekte eingefügt, so müssen bei jeder Operation ggf. lange Listen durchsucht werden
- Dies würde die Laufzeit negativ beeinflussen
- Belegungsfaktor $\alpha = \frac{n}{m}$
- n = Anzahl der Elemente in der Hashtable
- m = Anzahl der Behälter (Größe des Arrays `this.data`)

Gefahr von Kollisionen (1)

- Wie hoch ist die Wahrscheinlichkeit einer Kollision (bei Verwendung einer guten Hashfunktion)?
- Annahme: $k \in D, j \in \{0, \ldots, m-1\} : P(h(k) = j) = \frac{1}{m}$

- Wahrscheinlichkeit einer Kollision für das Aufbauen einer Hashtabelle mit n Elemente ist damit:
  P(Kollision) = 1-P(keine Kollision) $= 1 - P(1) \cdot P(2) \cdot \ldots \cdot P(n)$
  wobei $P(i)$ die Wahrscheinlichkeit ist, dass das Element i auf einen freien Behälter abgebildet wird

Gefahr von Kollisionen (2)

- Es gilt für $i < m$:

$$P(1) = 1$$
$$P(2) = \frac{(m-1)}{m}$$
$$\vdots$$
$$P(i) = \frac{(m-i+1)}{m}$$

- Somit erhalten wir

$$P(\text{Kollision}) = 1 - \prod_{i=1}^{n} P(i) = 1 - \frac{\prod_{i=1}^{n}(m-i+1)}{m^n}$$

- Beispiel (Geburtstagsparadoxon):

  m = 365

  n = 23: P(Kollision) = 50%

  n = 50: P(Kollision) = 97%

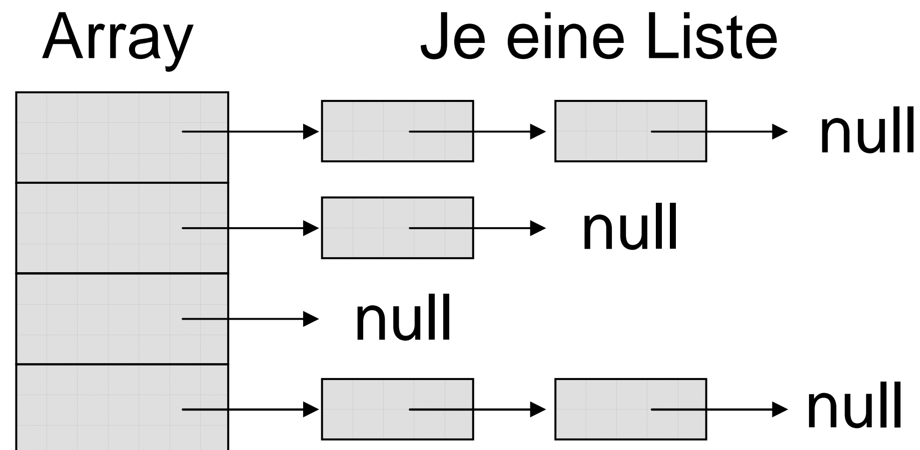
- Eine Hashtabelle muss effizient mit Kollisionen umgehen können

Verschiedene Arten von Hashtabellen

- Man unterscheidet Hashverfahren anhand Ihrer Behandlung von Kollisionen und wie die Behälter verwaltet werden
- Wir betrachten hier in der Vorlesung:
- **Hashverfahren mit Verkettung**
- **Offene (geschlossene) Hashverfahren**
  - Lineares Sondieren
  - Quadratisches Sondieren
  - Double Hashing
- **Lineares Hashing**

Hashverfahren mit Verkettung

- Hashverfahren mit Verkettung können **in den einzelnen Behältern mehrere Elemente** speichern
- Die zuvor vorgestellte Hashtabelle ist somit ein Vertreter der Hashverfahren mit Verkettung



Analyse: Hashverfahren mit Verkettung (1)

- **Im schlimmsten Fall** werden alle Elemente einem Behälter zugeordnet. In diesem Fall degeneriert die Hashtabelle zu einer Liste
- **Im Durchschnitt** ergibt sich für die **erfolglose Suche**:
  - Die Liste der entsprechenden Hashadresse muss komplett durchsucht werden
  - Bei einer Gleichverteiliung der Schlüssel auf Adressen ergibt sich für die Anzahl der Zugriffe:

$$\#\text{Zugriffe} = \frac{n}{m} = \alpha$$

  - Dies entspricht genau dem Belegungsfaktor

Analyse: Hashverfahren mit Verkettung (2)

- **Im Durchschnitt** ergibt sich für die **erfolgreiche Suche**:
- Betrachten wir den j-ten Datensatz zum Zeitpunkt des Einfügens.
- Zu diesem Zeitpunkt befanden sich in jeder Liste im Schnitt (j-1)/m Elemente
- Bei einer späteren Suche ergeben sich so 1+(j-1)/m Elemente
- Durch Summation über j und anschließendem teilen durch n ergibt sich

$$\#\text{Zugriffe} = \frac{1}{n}\sum_{j=1}^{n}\left(1 + \frac{j-1}{m}\right)$$

Analyse: Hashverfahren mit Verkettung (3)

$$
\begin{aligned}
\#\mathsf{Zugriffe} &= \frac{1}{n}\sum_{j=1}^{n}\left(1+\frac{j-1}{m}\right) \\
&= 1+\frac{n-1}{m} \\
&\approx 1+\frac{n}{2m} \\
&= 1+\frac{1}{2}\alpha
\end{aligned}
$$

- Die erfolgreiche Suche ist für $\alpha < 2$ somit teurer als die erfolglose Suche

Zusammenfassung Hashverfahren mit Verkettung

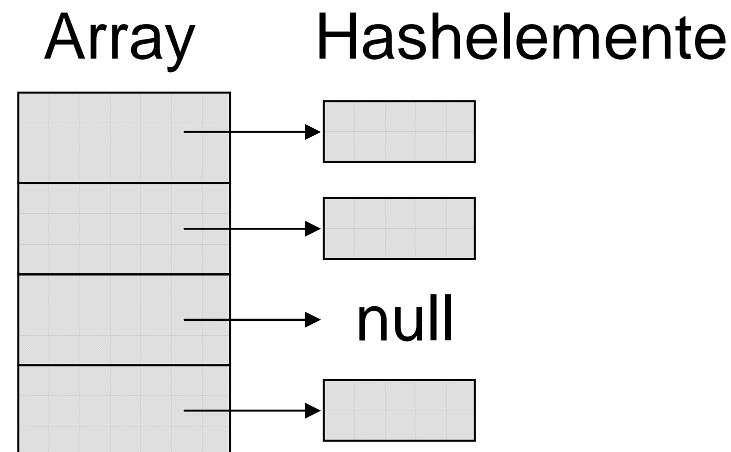
- Speichern mehrere Elemente pro Behälter
- Einfach zu realisieren
- Sie erlauben eine effiziente Realisierung einer Datenstruktur mit dem Aufwand

Verfahren	**Durchschnittlicher Aufwand**
Einfügen	$O(1)$
Suchen	$O(\alpha)$
Löschen	$O(\alpha)$

Einfügen hier ohne vorherige Existenzprüfung

Offene Hashverfahren

- Elemente, die nicht in ihrem eigentlichen Behälter gespeichert werden können nennt man Überläufer
- Die einzelnen Behältern speichern immer nur ein Element
- Überläufer werden in anderen Behältern gespeichert
- Es muss daher gelten $n \leq m$



Offene/Geschlossene Hashverfahren

- Unterscheidung bei der Namensgebung bzgl.
  - der Adressierung (offen/geschlossen)
  - der Anzahl der Elemente pro Behälter (geschlossen/offen)
- Variante A)   [siehe Seeger, Uni Marburg]
  - Offene Hashverfahren bzgl. Behälter (m. Verkettung)
  - Geschlossene Hashverfahren bzgl. Behälter (mit Sondierung)
- **Variante B) [siehe Ottmann, Uni Freiburg]**
  - Hashverfahren mit Verkettung
  - Offene Hashverfahren bzgl. Adressierung (mit Sondierung)

Sondierung

- Zentrale Frage: In welchem Behälter werden Überläufer gespeichert?
- Für jeden Schlüssel k gibt es eine Reihenfolge, in der die Speicherplätze in der Hashtabelle auf Belegung geprüft werden
- Diese Ordnung nennt man **Sondierungsfolge**
- Im Idealfall sollten die ersten m Elemente der Sondierungsfolge eine **Permutation der Hashadressen** sein
- Es gibt mehrere verschiedene Sondierungsfunktionen, u.a. lineares, quadratisches und zufälliges Sondieren (auch Double Hashing genannt)

Sondierungsfunktion

- Neben der Hashfunktion h kommt eine Sondierungsfunktion s zum Einsatz:

$$h : D \mapsto \{0, 1, \ldots, m-1\}$$

$$s : \{0, 1, \ldots, m-1\} \times D \mapsto N$$

- Aus der Sondierungsfunktion ergibt sich die Sondierungsfolge h_1, h_2, ..., h_i, ... als

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

Hashtabelle mit Sondierung: Insert

- Annahme: $n < m$

```
void insert(HashtabelElement ds) {
    int addr;
    int j=0;
    int hash = ds.getKey().hashCode();
    do {
          int sond = s(j++, ds.getKey());
          addr = (hash-sond) % this.data.length;
    } while(this.data[addr] != NULL);
    this.data[addr] = ds;
}
```

Problem: Gelöschte Elemente

- **Achtung**: werden Elemente aus der Hashtabelle entfernt, kann die Sondierungsfolge eines anderen Datensatzes unterbrochen werden
- Dadurch kann eine Suche zu früh unterbrochen werden
- Damit würde die Hashtabelle **inkonsistent**!

- Lösung: Einführen eine Belegungsmarkieung
  `boolean[] deleted; // identische Größe wie data`
- `deleted[i]` wird auf true gesetzt, wenn das Element i-te gelöscht wurde
- Dadurch wird die Sondierungsfolge nicht unterbrochen

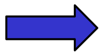
Realisierung einer Hashtabelle mit Sondierung

```
class SimpleOpenHashtable {
   SimpleOpenHashtable(int size) {
      this.data = new HashtableElement[size];
      this.deleted = new boolean[size];

      for (int i=0; i<size; i++) {
            this.data[i] = null;
            this.deleted[i] = false;
      }
   }
   ...

   HashtableElement[] data;
   boolean[] deleted;
}
```

Hashtabelle mit Sondierung: Insert (erweitert)

- Annahme: $n < m$

```
void insert(HashtabelElement ds) {
    int addr;
    int j=0;
    int hash = ds.getKey().hashCode();
    do {
        int sond = s(j++, ds.getKey());
        addr = (hash-sond) % this.data.length;
    } while(this.data[addr] != null);
    this.data[addr] = ds;
    this.deleted[addr] = false;
}
```



Hashtabelle mit Sondierung: Suchen

- Annahme: $n < m$

```
HashtableElement search(Object k) {
    int addr;
    int j=0;
    int hash = k.hashCode();
    do {
          int sond = s(j++, k);
          addr = (hash-sond) % this.data.length;
    } while(this.deleted[addr] ||
                (this.data[addr] != NULL &&
                !this.data[addr].getKey().equals(k)));
    if (this.deleted[addr] || this.data[addr] == NULL)
          return null;
    else
          return this.data[addr];
}
```

Hashtabelle mit Sondierung: Löschen

- Annahme: $n < m$

```
void delete(Object k) {
   int addr;
   int j=0;
   int hash = k.hashCode();
   do {
         int sond = s(j++, k);
         addr = (hash-sond) % this.data.length;
   } while(this.deleted[addr] ||
               (this.data[addr] != null &&
               !this.data[addr].getKey().equals(k)));
   this.data[addr] = null;
   this.deleted[addr] = true;
}
```

Lineares Sondieren

- Bisher haben wir die Sondierungsfunktion als abstrakte Funktion betrachtet
- Eine mögliche Realisierung ist lineares Sondieren:

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

- Daraus ergibt sich die Sondierungsfolge:

$$h(k), h(k)-1, h(k)-2, \ldots, 0, m-1, m-2, h(k)+1$$

Beispiel für Lineares Sondieren

- h(k) = k mod 7
- Einfügen von 78, 57, 80, 21:

57	78		80			21	data
f	f	f	f	f	f	f	deleted

- Einfügen von 29 (29 mod 7 = 1)

57	78		80		29	21	data
f	f	f	f	f	f	f	deleted

- Löschen von 57

	78		80		29	21	data
t	f	f	f	f	f	f	deleted

Primäre Clusterung

- Unter einem Cluster verstehen wir eine zusammenhängende Menge von belegten Behältern

Primäre Clusterung:

- Ein neuer Datensatz, dessen Hashadresse in einem Cluster liegt muss den Cluster linear bis zum Ende verfolgen
- Danach wird der Datensatz an das Ende des Clusters angefügt
- Damit wird der Cluster um ein Element vergrößert

Beispiel für Primäre Clusterung

- h(k) = k mod 7
- Einfügen von 78, 57, 80: 2 Cluster

57	78		80				data
f	f	f	f	f	f	f	deleted

- Einfügen von 29: Der größere Cluster vergrößert sich weiter

57	78		80			29	data
f	f	f	f	f	f	f	deleted

Quadratisches Sondieren

- Eine alternative Realisierung ist lineares Sondieren:

$$s(j,k) = (\text{ceil}(\tfrac{j}{2}))^2(-1)^j$$

- Daraus ergibt sich die Sondierungsfolge:

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

- Vorteile: Vermeidung von primären Clustern
- Quadratisches Sondieren ist effizienter als lineares Sondieren

Beispiel

- h(k) = k mod 7
- Einfügen von 78, 57, 80, 21:

21	78	57	80				data
f	f	f	f	f	f	f	deleted

- Einfügen von 16:

21	78	57	80			16	data
f	f	f	f	f	f	f	deleted

Sekundäre Clusterung

- Datensätze mit gleicher Hashadresse haben nach wie vor die gleicher Sondierungsfolge
- D.h. die Sondierungsfolge hängt nur vom Index j ab und nicht vom Schlüssel k
- Dies ist sub-optimal

Zufälliges Sondieren - Double Hashing

- **Ziel**: Vermeidung von primärer und sekundärer Clusterung
- Idealerweise sollten sich m! Sondierungsfolgen in Abhängigkeit der Schüssel ergeben
- Hashfunktionen selbst bieten die Möglichkeit nahezu „zufällig“ zu sondieren
- Lösung: Als Sondierungsfunktion wird eine Hashfunktion eingesetzt
- I.A. erzeugen Hashfunktionen aber keine Permutationen

Hashfunktion als Sondierungsfunktion

- s ist eine Hashfunktion (beachte: s ungleich h):

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

- Damit ergibt sich als Sondierungsfolge

$$h(k), (h(k) - h'(k))\%m, (h(k) - 2 \cdot h'(k))\%m, \ldots$$

- Anforderungen an die Hashfunktion zum Sondieren:
  - $h'(k)$ teilerfremd zu m
  - $h'(k) \neq 0$
- Unter diesen Bedingungen ist die Sondierungsfolge eine Permutation der Hashadressen

Beispiel

- Um eine Permutation zu erhalten haben wir gefordert:
  - $h'(k)$ teilerfremd zu m und $h'(k) \neq 0$
- Seien die Schlüssel ganze Zahlen und m eine Primzahl

$$h(k) = k \text{ mod } m$$
$$h'(k) = 1 + k \text{ mod } (m-2)$$

- Beispiel:

21	78	16	80		57		data
f	f	f	f	f	f	f	deleted

Insert(29)

21	78	16	80	29	57		data
f	f	f	f	f	f	f	deleted

- h(29) = 1 und h‘(29) = 5
- Sondierungsgfolge: 1, 3, 5, 0, 2, 4

Analyse

- Ist s eine Hashfunktion mit den beiden zuvor genannten Eigenschaften spricht man auch von **uniformem Sondieren**
- Man kann zeigen dass uniformes Sondieren **optimale Sondierungsfolgen** liefert
- Anzahl der Zugriffe für eine erfolglose Suche:

$$\#\text{Zugriffe} \approx \frac{1}{1-\alpha} \qquad \alpha < 1$$

- Anzahl der Zugriffe für eine erfolgreiche Suche:

$$\#\text{Zugriffe} \approx \frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right) \qquad \alpha < 1$$

Dynamische Hashverfahren

- Probleme bei den bisher vorgestellten Techniken
- Kleiner Belegungsfaktor ⇒ Schlechte Ausnutzung
- Großer Belegungsfaktor ⇒ Hohe Suchkosten
- Keine Unterstützung stark anwachsender Tabellen
- Für (offene) Hashverfahren mit Sondierung gilt $n \leq m$

Lösungen:

- **Globale Reorganisation** (Adaption der Tabellengrosse, Hashfunktion, und Umspeicherung von Elementen)
- Aus Effizienzgründen empfehlen sich **schrittweise, „kleine" Reorganisationen**

Lineares Hashing

- **Lineares Hashing** ist ein Vertreter der **dynamischen** Hashverfahren
- Erfunden 1980 von W. Litwin zur Verwaltung großer Datenmengen auf dem Externspeicher
- Lineares Hashing kann ebenfalls als interne Datenstruktur genutzt werden (hier beschränken wir uns auf diesen Fall)
- Wir beschränken uns weiterhin auf die Realisierung bei der jeder Behälter eine Liste darstellt

Idee des Linearen Hashings

- Initialisierung identisch zu dem anfangs vorgestellten Hashverfahren mit Verkettung

$$h : D \mapsto \{0, 1, \ldots, m-1\}$$

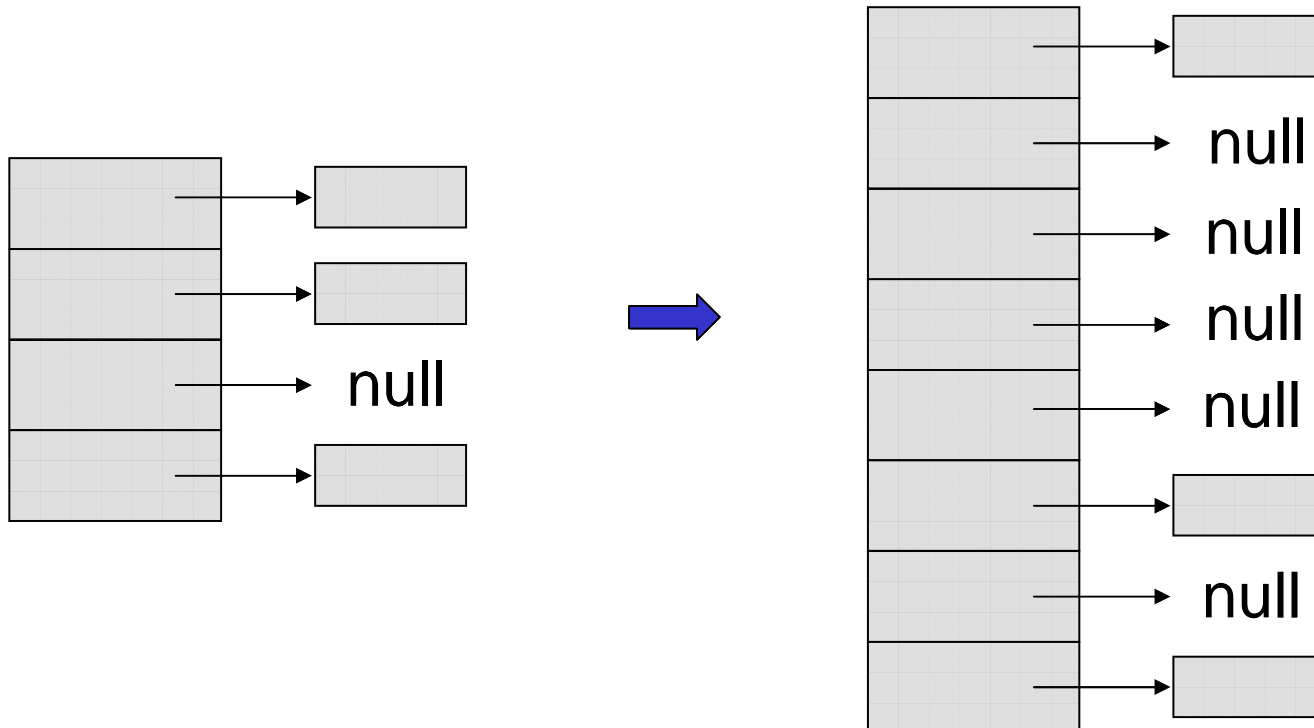
- Jeder Behälter wird durch eine Liste realisiert
- Mit steigendem Belegunsgfaktor steigt die Wahrscheinlichkeit für Kollisionen und führt somit zu einem Anwachsen der Listen

Idee:

- Falls der Belegungsfaktor zu groß wird, erweitert man die Hashtabelle schrittweise

Einfache Realisierung des Linearen Hashings

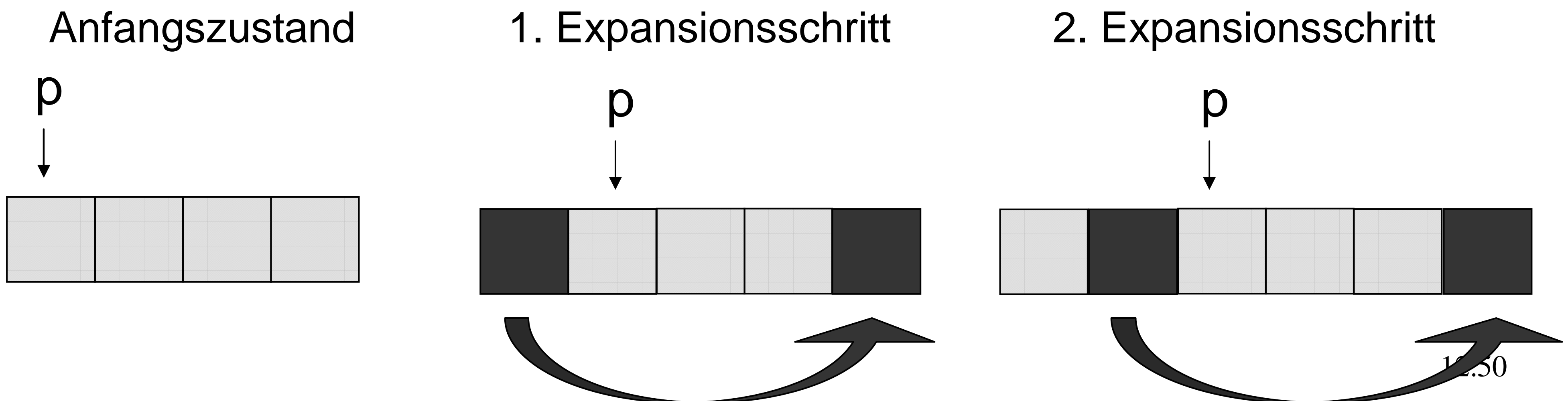
- Ist der α zu groß, erstellt man eine neue Tabelle mit der Größe 2*m und kopiert alle Elemente in die neue Tabelle



- **Problem**: Teueres rehashing aller existierender Elemente

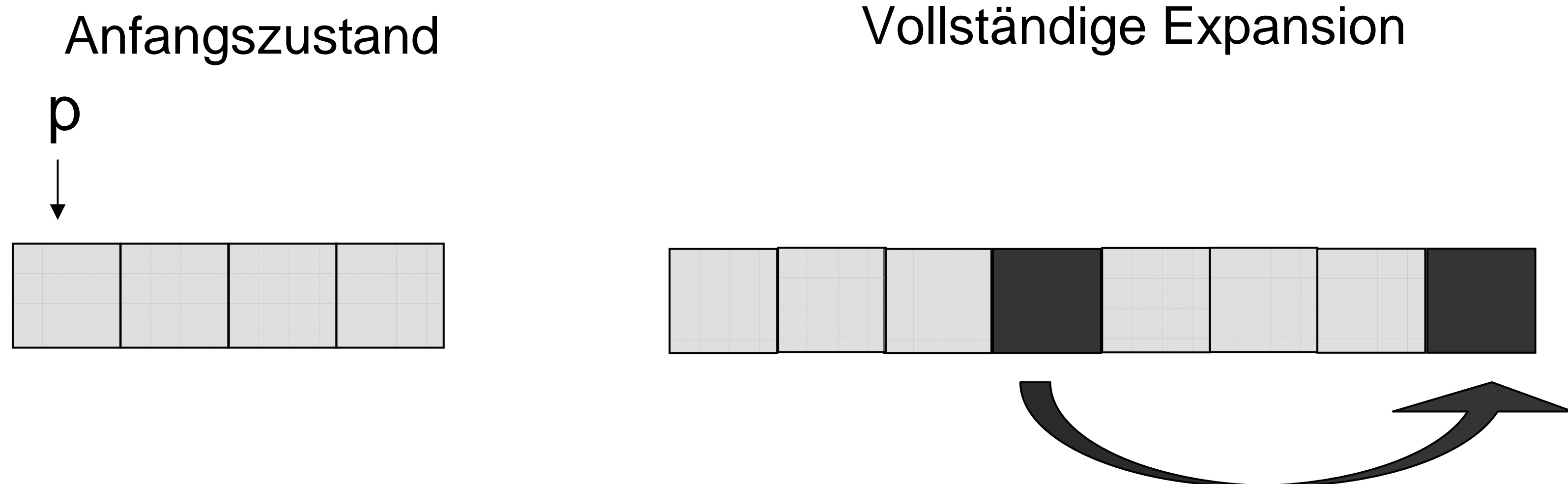
Lineares Hashing (Optimiert)

- Idee: Schrittweise Hinzunahme neuer Behälter
- In jedem Expansionsschritt wird genau ein Behälter selektiert und dessen Elemente neu verteilt
- Man fängt mit dem Behälter B_0 an und expandiert im Schritt i den Behälter B_{i-1}
- Praktisch realisiert man dies über einen Expansionszeiger p

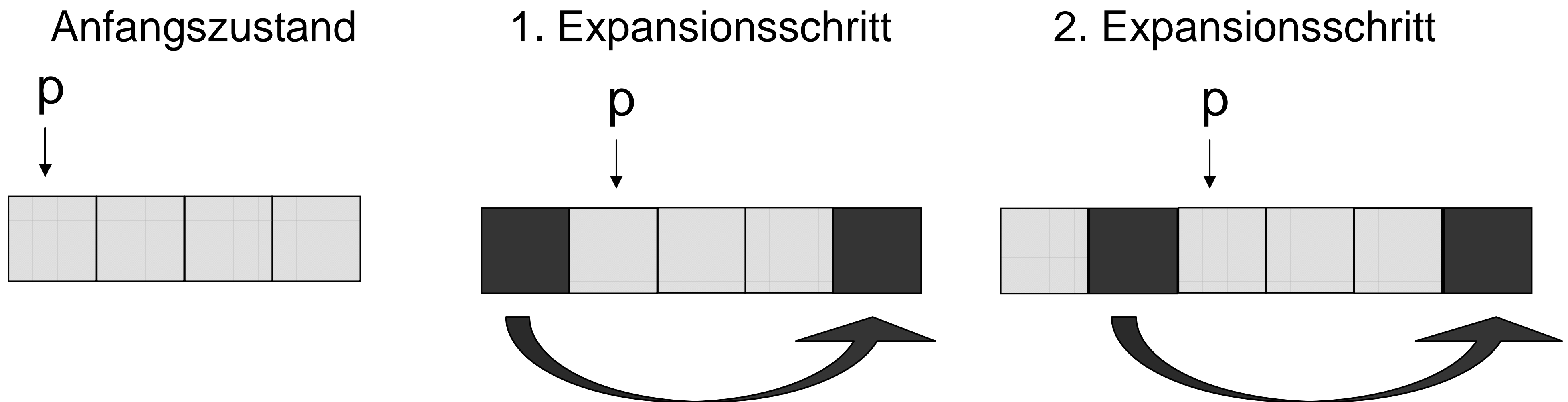


Lineares Hashing (Optimiert)

- Hat sich die Hashtabelle verdoppelt, so spricht man von einer vollständigen Expansion
- Die Anzahl der Verdopplungen bezeichnet man als Level L
- Sei die Initiale Größe der Tabelle m_0
- Die Anzahl der Hashadressen nach einer vollständigen Expansion ist somit $m_0 \cdot 2^L$



Zentrale Fragen



- Wie verteilt man die Elemente eines Behälters?
- Wie findet man nach einem Expansionsschritt die Datensätze im neuen Behälter?

Wahl der Hashfunktion(en)

- Man löst diese Probleme durch Adaption der Hashfunktion
- Dabei setzte sich die Hashfunktion h beim Linear Hashing aus bis zu zwei Hashfunktionen h_L und h_{L+1} zusammen

$$h_L : D \mapsto \{0, 1, \ldots, m_0 \cdot 2^L - 1\}$$

$$h_{L+1} : D \mapsto \{0, 1, \ldots, m_0 \cdot 2^{L+1} - 1\}$$

- Wir setzen die sog. Spalt-Eigenschaft voraus:

$$h_{L+1}(k) = \begin{array}{c} h_L(k) \\ \text{oder} \\ h_L(k) + m_0 \cdot 2^L \end{array}$$

Beispiel

- Eine Hashfunktion, die die soeben vorgestellten Eigenschaften besitzt ist

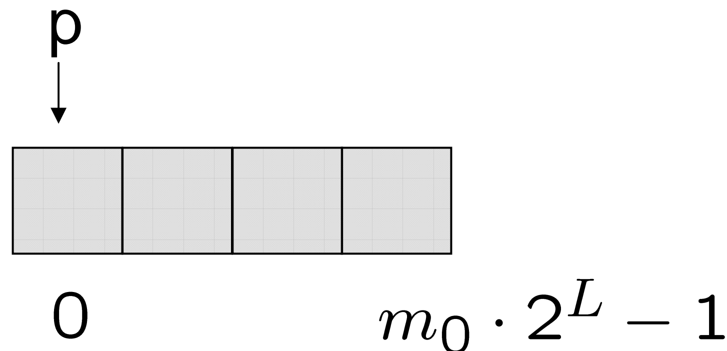
$$h_L(k) = k \text{ mod } (m_0 \cdot 2^L)$$

Beispiel:

- $m_0 = 4$
- $L = 3$
- $k = 36$
- $h_L(k) = h_3(36) = 36 \text{ mod } (4*2^3) = 36 \text{ mod } 32 = 4$
- $h_L(k) + m_0*2^L = h_3(36) + (4*2^3) = 4 + (4*2^3) = 4 + 32 = 36$
- $h_{L+1}(k) = h_4(36) = 36 \text{ mod } (4*2^4) = 36 \text{ mod } 64 = 36$

Der Expansionsschritt im Detail (1)

- Betrachten wir die Situation nach L vollständigen Expansionen
  - Es gibt also $m_0 \cdot 2^L$ Behälter
  - Der Expansionszeiger p zeigt auf das 0-te Element
  - Alle Element werden durch h_L korrekt abgebildet



- Ist nach einer Einfügeoperation der Belegungsfaktor größer als ein Schwellwert, wird ein Expansionschritt ausgeführt

Der Expansionsschritt im Detail (2)

- Im Falle einer schrittweisen Expansion wird ein **neuer Behälter angehängt** und der **Behälter B_p aufgespalten**
- Für jedes Element in B_p wird h_{L+1} ausgewertet und das Element **entweder dem neuen und alten Behälter zugeteilt**
- Aufgrund der Spalt-Eigenschaft sind nur 2 Resultate möglich
  - $h_{L+1}(k) = h_L(k) = p$
  - $h_{L+1}(k) = h_L(k) + m_0 \cdot 2^L = p + m_0 \cdot 2^L$
- Die Elemente in B_p werden daher auf B_p und $B_{p+m_0*2^L}$ aufgeteilt
- Der Zeiger p wird um 1 erhöht
- Falls $p = m_0 \cdot 2^L$, ist eine vollständige Expansion abgeschlossen, d.h. L wird um 1 erhöht und p = 0

Hashfunktion

- Mit dieser Technik lässt sich die endgültige Hashfunktion einfach konstruieren:

```
int LH_hash(k) {
    int addr = h(L, k);
    if (addr < p)
          addr = h(L+1, k);
    return addr;
}
```

- `h(L, k)` könnte z.B. wie folgt definiert sein:

$$h_L(k) = k \bmod (m_0 \cdot 2^L)$$

Expansion der Behälter-Datenstruktur

- Bisher haben wir **angenommen**, wir könnten an die Hashtabelle einfach einen neuen Behälter anhängen
- Dies ist im worst-case mit linearem Aufwand verbunden
- Man behilft sich in der Praxis mit einem „Trick“ (den beispielsweise auch die Klasse Vector nutzt)
  – Man allokiert mehr Speicher als man eigentlich benötigt
  – Ist auch dieser Speicher verbraucht, allokiert man eine neue Kollektion mit doppelt so vielen Speicherplätzen und kopiert die ursprünglichen Elemente in die neue Kollektion
- Diese Operation hat aber **linearen Aufwand:** O(n)
- Sie muss allerdings **nur alle n Schritte einmal** ausgeführt werden
- Daher spricht man von **amortisiert konstantem Aufwand**

Automatische Array Expansion

```
Class DynamicArray {
   ...
   void checkAllocation() {
       if (this.numElements != this.allocated) return;
       Object[] newdata = new Object[2*this.allocated];
       for (int i=0; i<this.allocated; i++)
              this.newdata[i] = this.data[i];
       this.data = this.newdata;
       this.allocated = this.allocated * 2;
   }

   Object[] data; // Speicherplatz für die Elemente
   int numElements;  // Anzahl der gespeicherten Elemente
   int allocated; // Anzahl der reservierten Speicherplätze
}
```

Lineares Hashing (1)

- Lineares Hashing stellt eine **effiziente Methode** da, die Größe einer Hashtabelle **dynamisch anzupassen**
- Man muss sich nicht im vorhinein festlegen, wie viele Elemente man einfügen möchte
- Ähnlich wie die Erweiterung einer Hashtabelle kann diese auch verkleinert werden, wenn der Belegungsgfaktor zu klein wird
- Das Zusammenlegen von zwei Behältern kann in konstanter Zeit erledigt werden
- Die Reduktion einer Hashtabelle funktioniert äquivalent zur Expansion

Lineares Hashing (2)

- Effizientes Suchen, Löschen, und Einfügen

Verfahren	Durchschnittlicher Aufwand
Einfügen	O(1)
Suchen	O(1)
Löschen	O(1)

- Lineares Hashing ist i.A. den anderen in der Vorlesung vorgestellten Verfahren vorzuziehen

Zusammenfassung

- Hashtabellen stellen eine Technik dar, um grundlegende **Mengenoperationen** wie Einfügen, Suchen und Löschen in **durchschnittlich konstanter Zeit** zu realisieren
- Objekte werden mittels einer **Hashfunktion adressiert**
- Jedes Objekt muss sich mittels eines **Schlüssels** (key) unterscheiden lassen
- Wird mehreren Objekten die gleiche Hashadresse zugewiesen, kommt es zu **Kollisionen**
- Hashverfahren unterscheiden sich hauptsächlich dadurch, wie sie mit Kollisionen umgehen und ob sie ihre Größe dynamisch anpassen können
- Wir haben Hashing **mit Verkettung, mit Sondierung (offenes Hashing), und lineares Hashing** betrachtet