IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(12 Hashverfahren: Verkettung der Überläufer)

Prof. Dr. Susanne Albers

Möglichkeiten der Kollisionsbehandlung

Kollisionsbehandlung:

- Die Behandlung von Kollisionen erfolgt bei verschiedenen Verfahren unterschiedlich.
- Ein Datensatz mit Schlüssel *s* ist ein Überläufer, wenn der Behälter *h*(*s*) schon durch einen anderen Satz belegt ist.
- Wie kann mit Überläufern verfahren werden?
  1. Behälter werden durch verkettete Listen realisiert. Überläufer werden in diesen Listen abgespeichert.
  Chaining (Hashing mit Verkettung der Überläufer)
  2. Überläufer werden in noch freien anderen Behältern abgespeichert. Diese werden beim Speichern und Suchen durch sogenanntes Sondieren gefunden.
  Open Addressing (Offene Hashverfahren)

Verkettung der Überläufer (1)

- Die Hash-Tabelle ist ein Array (Länge *m*) von Listen. Jeder Behälter wird durch eine
  Liste realisiert.

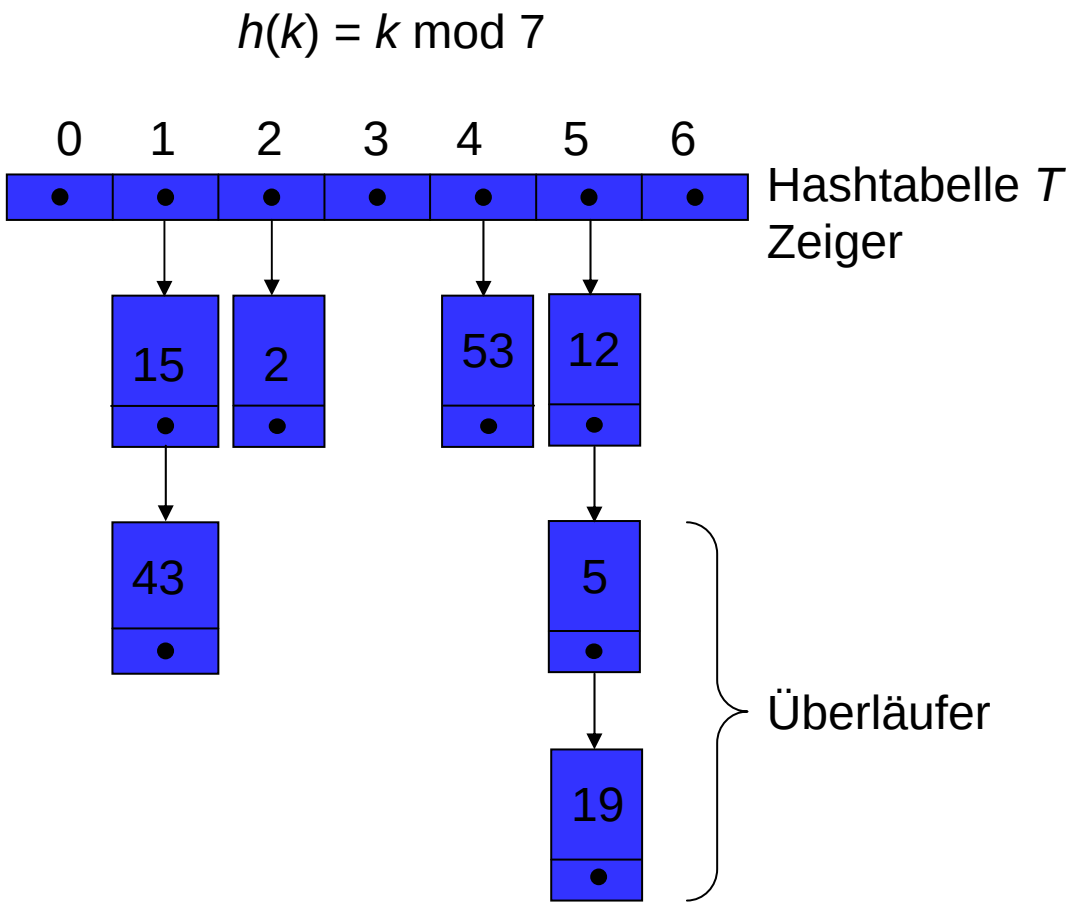
```
class hashTable {
    Liste [] ht; // ein Listen-Array
    hashTable (int m){ // Konstruktor
        ht = new Liste[m];
        for (int i = 0; i < m; i++)
            ht[i] = new Liste(); // Listen-Erzeugung
    }
    ...
}
```

- Zwei verschiedene Möglichkeiten der Listen-Anlage:

  1. Hash-Tabelle enthält nur Listen-Köpfe, Datensätze sind in Listen: Direkte Verkettung
  2. Hash-Tabelle enthält pro Behälter maximal einen Datensatz sowie einen Listen-Kopf. Überläufer kommen in die Liste: Separate Verkettung

Hashing mit Verkettung der Überläufer

Schlüssel werden in Überlauflisten gespeichert



Diese Art der Verkettung wird auch als direkte Verkettung bezeichnet.

Verkettung der Überläufer

Suchen nach Schlüssel k

- Berechne $h(k)$ und Überlaufliste $T[h(k)]$
- Suche nach k in der Überlaufliste

Einfügen eines Schlüssels k

- Suchen nach k (erfolglos)
- Einfügen in die Überlaufliste

Entfernen eines Schlüssels k

- Suchen nach k (erfolgreich)
- Entfernen aus Überlaufliste

→ Reine Listenoperationen

Implementierung in Java

```
class TableEntry {
    private Object key,value;
}

abstract class HashTable {
    private TableEntry[] tableEntry;
    private int capacity;

    //Konstruktor
    HashTable (int capacity) {
        this.capacity = capacity;
        tableEntry = new TableEntry [capacity];
        for (int i = 0; i <= capacity-1; i++)
            tableEntry[i] = null;
    }
    // die Hashfunktion
    protected abstract int h (Object key);

    // fuege Element mit Schluessel key und Wert value ein (falls nicht vorhanden)
    public abstract void insert (Object key Object value);

    // entferne Element mit Schluessel key (falls vorhanden)
    public abstract void delete (Object key);

    // suche Element mit Schluessel key
    public abstract Object search (Object key);
} // class hashTable
```

Implementierung in Java

```
class ChainedTableEntry extends TableEntry {
    // Konstruktor
    ChainedTableEntry(Object key, Object value) {
        super(key, value);
        this.next = null;
    }
    private ChainedTableEntry next;
}

class ChainedHashTable extends HashTable {
    // die Hashfunktion
    public int h(Object key) {
        return key.hashCode() % capacity ;
    }

// suche key in der Hashtabelle
public Object search (Object key) {
    ChainedTableEntry p;
    p = (ChainedTableEntry) tableEntry[h(key)];

    // Gehe die Liste durch bis Ende erreicht oder key gefunden
    while (p != null && !p.key.equals(key)) {
        p = p.next;
    }

    // Gib Ergebnis zurueck
    if (p != null)
        return p.value;
    else return null;
}
```

Implementierung in Java

```
/* fuege ein Element mit Schluessel key und Wert value ein (falls
   nicht vorhanden) */
public void insert (Object key, Object value) {
    ChainedTableEntry entry = new ChainedTableEntry(key, value);

    // Hole den Tabelleneintrag fuer key
    int k = h (key);
    ChainedTableEntry p;
    p = (ChainedTableEntry) tableEntry [k];

    if (p == null){
        tableEntry[k] = entry;
        return ;
    }
    // Suche nach key
    while (!p.key.equals(key) && p.next != null) {
        p = p.next;
    }

    // Fuege das Element ein (falls nicht vorhanden)
    if (!p.key.equals(key))
        p.next = entry;
}
```

Implementierung in Java

```
// entferne das Element mit Schluessel key (falls vorhanden)
public void delete (Object key) {
    int k = h (key);
    ChainedTableEntry p;
    p = (ChainedTableEntry) TableEntry [k];
    TableEntry[k] = recDelete(p, key);
}

// entferne das Element mit Schluessel key rekursiv (falls vorhanden)
public ChainedTableEntry recDelete (ChainedTableEntry p, Object key) {
    /* recDelete gibt einen Zeiger auf den Beginn der Liste, auf die p zeigt,
       zurueck, in der key entfernt wurde */
    if (p == null)
        return null;
    if (p.key.equals(key))
        return p.getNext();
    // ansonsten:
    p.next = recDelete(p.next, key);
    return p;
}

public void printTable () {...}
} // class ChainedHashTable
```

Test-Programm

```
public class ChainedHashingTest {
    public static void main(String args[]){
        Integer[] t= new Integer[args.length];

        for (int i = 0; i < args.length; i++)
            t[i] = Integer.valueOf(args[i]);

        ChainedHashTable h = new ChainedHashTable(7);
        for (int i = 0; i <= t.length - 1; i++)
            h.insert(t[i], null);

        h.printTable ();
        h.delete(t[0]); h.delete(t[1]);
        h.delete(t[6]); h.printTable();
    }
}
```

Aufruf:

```
java ChainedHashingTest 12 53 5 15 2 19 43
```

Ausgabe:

0: -|
1: 15 -> 43 -|
2: 2 -|
3: -|
4: 53 -|
5: 12 -> 5 -> 19 -|
6: -|

0: -|
1: 15 -|
2: 2 -|
3: -|
4: -|
5: 5 -> 19 -|
6: -|

Analyse der direkten Verkettung

Uniform-Hashing Annahme:

- alle Hashadressen werden mit gleicher Wahrscheinlichkeit gewählt, d.h.:

  $Pr(h(k_i) = j) = 1/m$

- unabhängig von Operation zu Operation

Mittlere Kettenlänge bei n Einträgen:

$n/m = \alpha$

Definition

C'_n = Erwartungswert für die Anzahl betrachteter Einträge bei erfolgloser Suche

C_n = Erwartungswert für die Anzahl betrachteter Einträge bei erfolgreicher Suche

Analyse

$$C'_n = \alpha$$

$$C_n \approx 1 + \frac{\alpha}{2}$$

Verkettung der Überläufer

Vorteile:

+ C_n und C'_n niedrig
+ $\alpha > 1$ möglich
+ echte Entfernungen
+ für Sekundärspeicher geeignet

Effizienz der Suche

α	C_n (erfolgreich)	C'_n (erfolglos)
0.50	1.250	0.50
0.90	1.450	0.90
0.95	1.457	0.95
1.00	1.500	1.00
2.00	2.000	2.00
3.00	2.500	3.00

Nachteile

- Zusätzlicher Speicherplatz für Zeiger
- Überläufer außerhalb der Hashtabelle

Zusammenfassung

Analyse des Hashings mit Verkettung der Überläufer:

- worst case: h(s) liefert immer den gleichen Wert, alle Datensätze sind in einer Liste. Verhalten wie bei linearen Liste.
- average case:
  – Erfolgreiche Suche & Entfernen: Aufwand in Datenzugriffen $\approx 1 + 0.5 \times BF$
  – Erfolglose Suche & Einfügen: Aufwand $\approx$ BF

  Das gilt für direkte Verkettung, bei separater Verkettung ist der Aufwand jeweils etwas höher.
- best case: Die Suche hat sofort Erfolg. Aufwand $\in O(1)$.