Einführung in die Informatik 2

– Wörterbücher und Hashing –

Sven Kosub

AG Algorithmik/Theorie komplexer Systeme
Universität Konstanz

E 202 | Sven.Kosub@uni-konstanz.de | Sprechstunde: nach Vereinbarung

Sommersemester 2010

Worterbücher

Wörterbuch (engl. *dictionary*):

dynamisch veränderliche Objektmenge D aus einem Universum U möglicher Objekte mit den Grundoperationen:

- **Suchen**: Ist ein Objekt d in D enthalten? Falls „Ja“, gib es zurück
- **Löschen**: Entferne Objekt d aus D
- **Einfügen**: Füge Objekt d in D ein

Probleme mit bekannten Wörterbuchimplementierungen:

- Reihungen: dynamische Anpassungen teuer, viel ungenutzter Speicher bei wenigen Objekten
- Listen: Suchen in $O(n)$; Einfügen, Löschen zwar effizient, aber vorher immer Suche notwendig

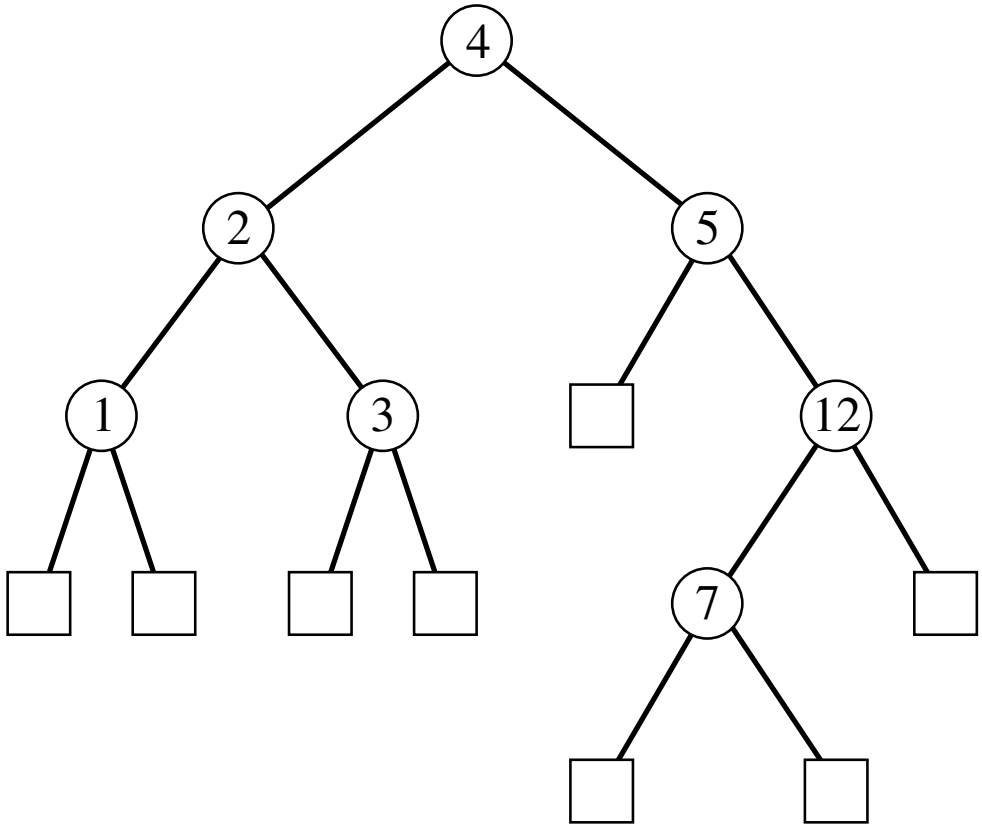
Suchbäume

Annahme: Universum U ist geordnet bezüglich $\leq_U$

(binärer) Suchbaum für Wörterbuch D:

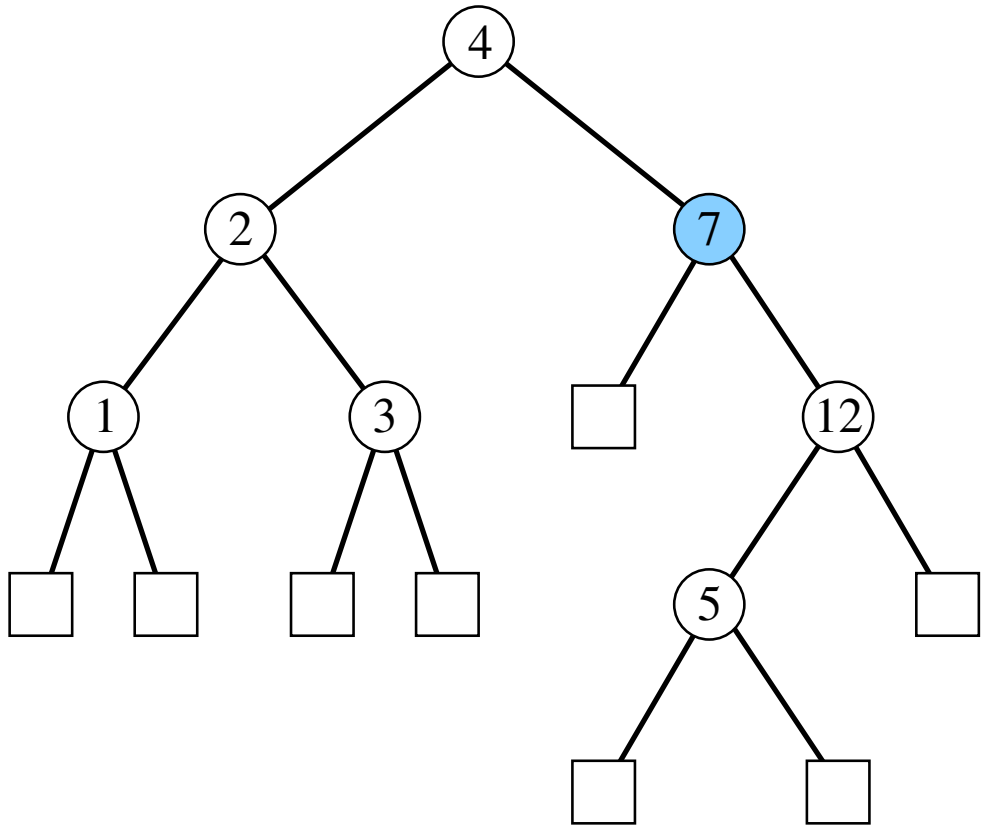
- voller Binärbaum T
- innere Knoten enthalten die Objekte von D
- in jedem inneren Knoten v gilt die Suchbaumeigenschaft:
  - Objekte im linken Teilbaum von $v \leq_U$ Objekt in v
  - Objekte im rechten Teilbaum von $v \geq_U$ Objekt in v

Suchbäume: Beispiel



Suchbaumeigenschaft für alle Knoten erfüllt

Suchbäume: Beispiel



Suchbaumeigenschaft nur für Knoten mit Inhalt 7 nicht erfüllt

Suchbäume: Implementierung in Java

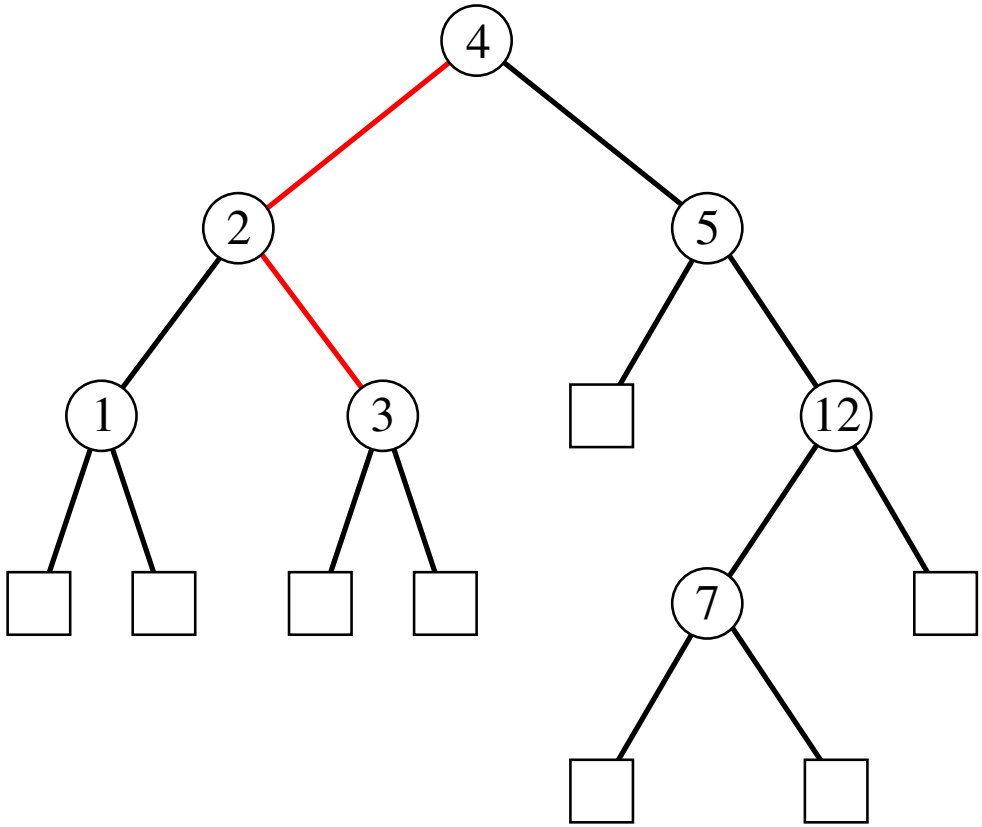
```
package searchtree;

public class Node {
  Comparable key;
  Node parent;
  Node leftChild;
  Node rightChild;
  boolean isExternal;
  // Konstruktor
  Node() {
    key=null;
    parent=null;
    leftChild=null;
    rightChild=null;
    isExternal=true;
  }
}
...
```

```
...
public class SearchTree {
  private Node root;
  // Konstruktor
  SearchTree() {
    root=new Node;
  }
}
```

Suchbäume: Suchen

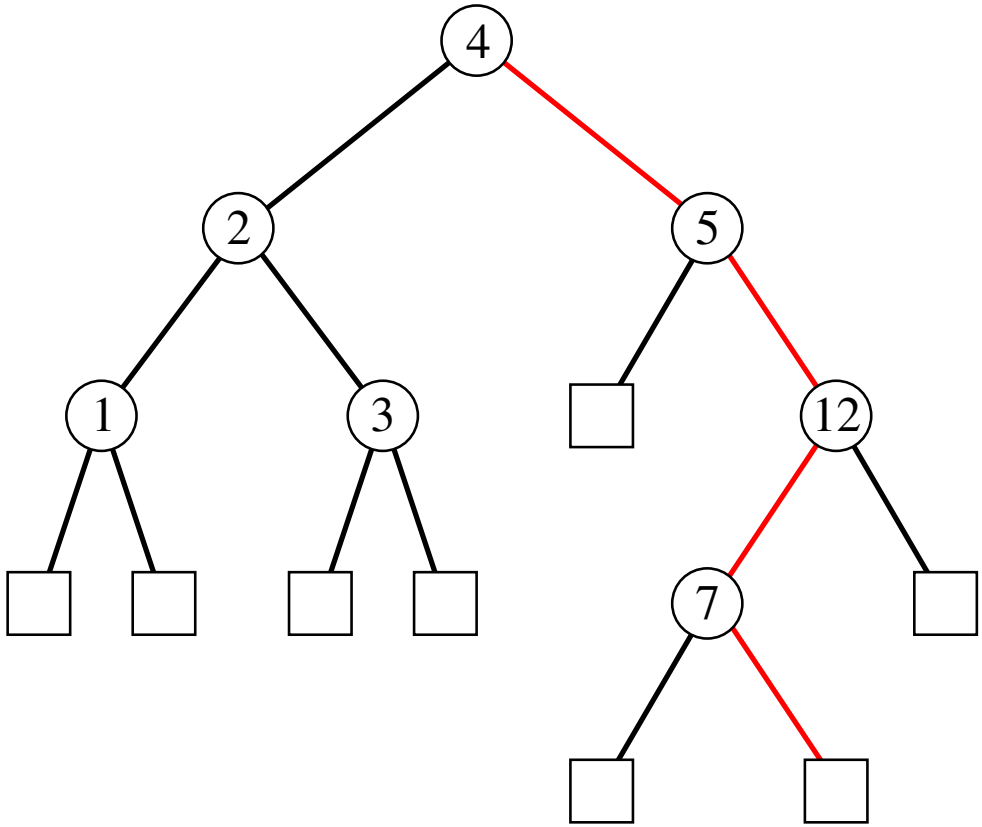
```
package searchtree;
public class SearchTree {
  :
  private Node TreeSearch(Node n, Comparable k) {
    if (n.isExternal) return(n);
    int c=n.key.compareTo(k);
    if (c==0) return(n);
    if (c<0) return(TreeSearch(n.rightChild, k));
    else return(TreeSearch(n.leftChild), k));
  }
  public Node findItem(Comparable d) {
    return(TreeSearch(root,d));
  }
  :
}
```

Suchbäume: Suchen



erfolgreiche Suche nach Schlüssel 3

Suchbäume: Suchen

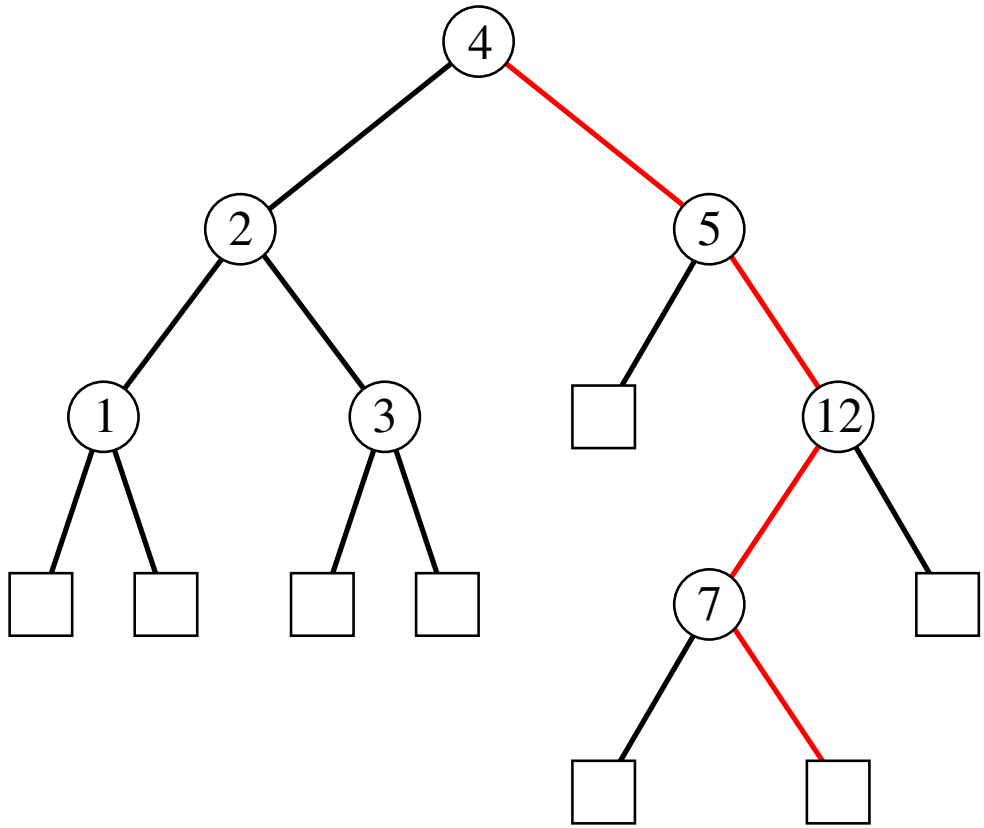


erfolglose Suche nach Schlüssel 8

Suchbäume: Einfügen

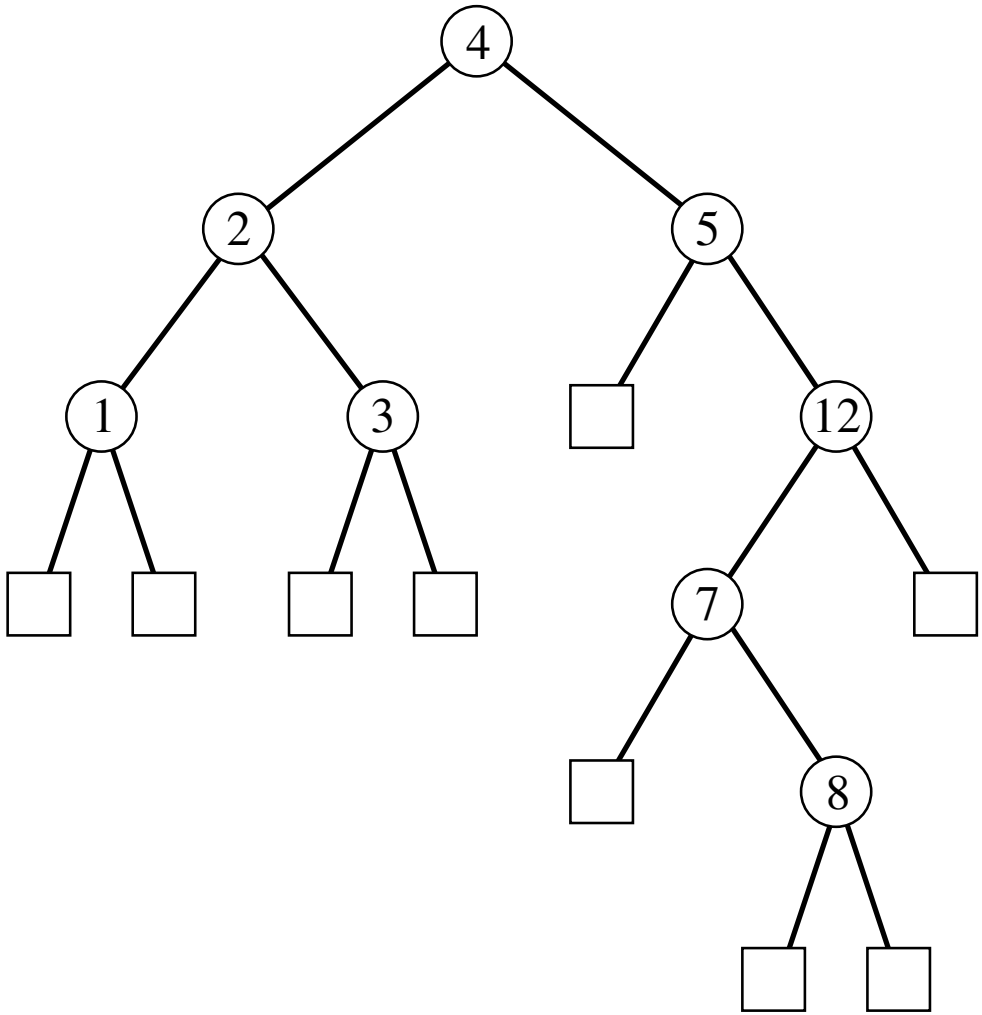
```
package searchtree;
public class SearchTree {
  :
  public void insertItem(Comparable k) {
    // Suche Blatt zum Einfügen
    Node n=TreeSearch(root, k);
    while (!n.isExternal) n=TreeSearch(n.leftChild, k);
    // Mache Blatt zu inneren Knoten mit neuem Inhalt k
    n.key=k;
    n.isExternal=false;
    n.leftChild=new Node;
    n.leftChild.parent=n;
    n.rightChild=new Node;
    n.rightChild.parent=n;
  }
  :
}
```

Suchbäume: Suchen



Finde Blatt zum Einfügen von Schlüssel 8

Suchbäume: Suchen



Füge Schlüssel 8 ein

Suchbäume: Beurteilung

- **Suchen:**
  benötigt $O(h)$ Vergleiche
- **Einfügen:**
  benötigt ebenfalls $O(h)$ Vergleiche wegen Verwendung von `TreeSearch`; Einfügen selbst ist effizient, d.h. $O(1)$
- **Löschen:**
  benötigt $O(h)$ Vergleiche wegen Restrukturierung zur Gewährleistung der Suchbaumeigenschaft (private Methode `inorderNext(Node n)`)

Fazit: bei Höhenbalancierung ist $O(\log n)$ für alle Operationen realisierbar; nach Einfügen und Löschen Ausbalancierung notwendig

Komponenten von Hashverfahren (Streuspeicherverfahren)

Hash-Funktion (engl. *hash function*)

- bildet Objekte auf natürliche Zahlen (Java-Typ `int`) ab
- jedes Objekt erhält somit Hash-Wert (engl. *hash codes*)
- Hash-Werte dienen als Schlüssel zur Identifizierung der Objekte
- Abbildung nicht notwendigerweise injektiv, d.h. zwei Objekte könnten gleichen Hash-Wert besitzen (Kollision)

Hash-Tabelle (engl. *hash table*)

- speichert Objekte unter ihrem Hash-Wert ab
- Implementierung als Reihung mit Hash-Werten als Index

Verallgemeinerung von Hashing

- Verwendung nichtnummerischer Schlüssel: Abbildung von Objekten („Schlüsselobjekte“) auf Objekte („Wert-Objekte“)
- Prinzip: Wertobjekt wird unter Schlüsselobjekt (z.B. aus Klasse `String`) als Paar

$$\langle \text{ Schlüsselobjekt, Wertobjekt } \rangle$$

in Hash-Tabelle unter Hash-Wert von Schlüsselobjekt gespeichert

- Java-Implementierung ist so organisiert

Einfachste Version einer Hash-Tabelle

- nummerische Schlüssel als Indizes für Reihung
- sinnvoll falls:
  - Hash-Funktion injektiv, d.h. verschiedene Objekte haben verschiedene Schlüssel
  - genügend Speicherplatz vorhanden, um alle *möglichen* Schlüssel als Index unterzubringen (häufig problematisch, da typischerweise nur wenige Schlüssel verwendet werden)
- Kollisionen sind praktisch nicht vermeidbar, d.h. spezielle Strategien zur Kollisionsbehandlung notwendig

Einfachste Version einer Hash-Tabelle

- Objekte der Klasse `String` sind durch Personennamen gegeben
- Hash-Funktion h sei wie folgt definiert:

$$h(s) = i \quad \Longleftrightarrow_{\text{def}} \quad \begin{array}{l}\text{Anfangsbuchstabe des Nachnamens von } s \\ \text{ist } i\text{-ter Buchstabe des Alphabets}\end{array}$$

- Beispielmenge:

Anton Wagner	W	$\mapsto$	23
Doris Bach	B	$\mapsto$	2
Doris May	M	$\mapsto$	13
Friedrich Dörig	D	$\mapsto$	4

- Reihung (Array) der Größe 26 reicht aus
- keine Kollisionen

Hash-Funktionen

- formale Hash-Funktion h:

$$h: \text{ Menge der Objekte } \rightarrow \mathbb{N} \text{ (bzw. } \mathbb{N}^k)$$

- Objekt s wird Speichplatz `HT[h(s)]` zugewiesen
- Zugriff auf Objekt in konstanter Zeit, d.h. in $O(1)$ (ohne Kollisionen)

$h(\text{„Doris Bach“}) = 2$ führt zu `HT[2]="Doris Bach"`

- Entwurfsaspekte zur Kollisionsvorbeugung:
  - Tabelle hinreichend groß wählen
  - Hash-Funktion vom gesamten Objekt abhängen lassen (d.h. bei Objekten der Klasse `String` von allen Buchstaben)

Hash-Funktionen für Zeichenketten

- $h(s) =_{\text{def}}$ Konkatenation des ASCII-Codes der Buchstaben in s

$h(\text{„INFO“}) = 494E464F_{16}$ mit

Symbol	I	N	F	O
ASCII	49_{16}	$4E_{16}$	46_{16}	$4F_{16}$

- Problem: $h(s)$ ergibt zu große Indizes
  - z.B. bei Zeichenketten bis zur Länge 20 nimmt benötigter Indexraum 2^{21}Byte bzw. 2MB in Anspruch (1 Byte pro Symbol)
- Ausweg: Reduktion von $h(s)$ durch Rechnung $\mod m$
  - Hash-Tabellen hat Kapazität für maximal m Einträge
  - Kollisionen nicht ausgeschlossen

Für $m = 83$ ist $h(\text{„INFO“}) = 48_{10}$, denn

$$494E464F_{16} \mod 83 = 1229866575_{10} \mod 83 = 48_{10} \mod 83$$

Modulo-Arithmetik

- Problem: explizite Berechnung von $h(s)$ (ohne Modulus) nicht in CPU-Arithmetik möglich (in `long`) für `s.length()` > 8
- Ausweg: berechnen $h(s)$ mit Modulus m mittels folgender Regeln:

$$\begin{aligned}(a+b) \mod m &= \big((a \mod m) + (b \mod m)\big) \mod m \\ (a \cdot b) \mod m &= \big((a \mod m) \cdot (b \mod m)\big) \mod m\end{aligned}$$

Horner-Schema:

- $z(i,s)$ sei (8-Bit-)ASCII-Wert des i-ten Symbols von $s = s_0 s_1 \ldots s_{n-1}$
- $h(s) = h_{n-1}(s)$ ergibt sich rekursiv wie folgt:

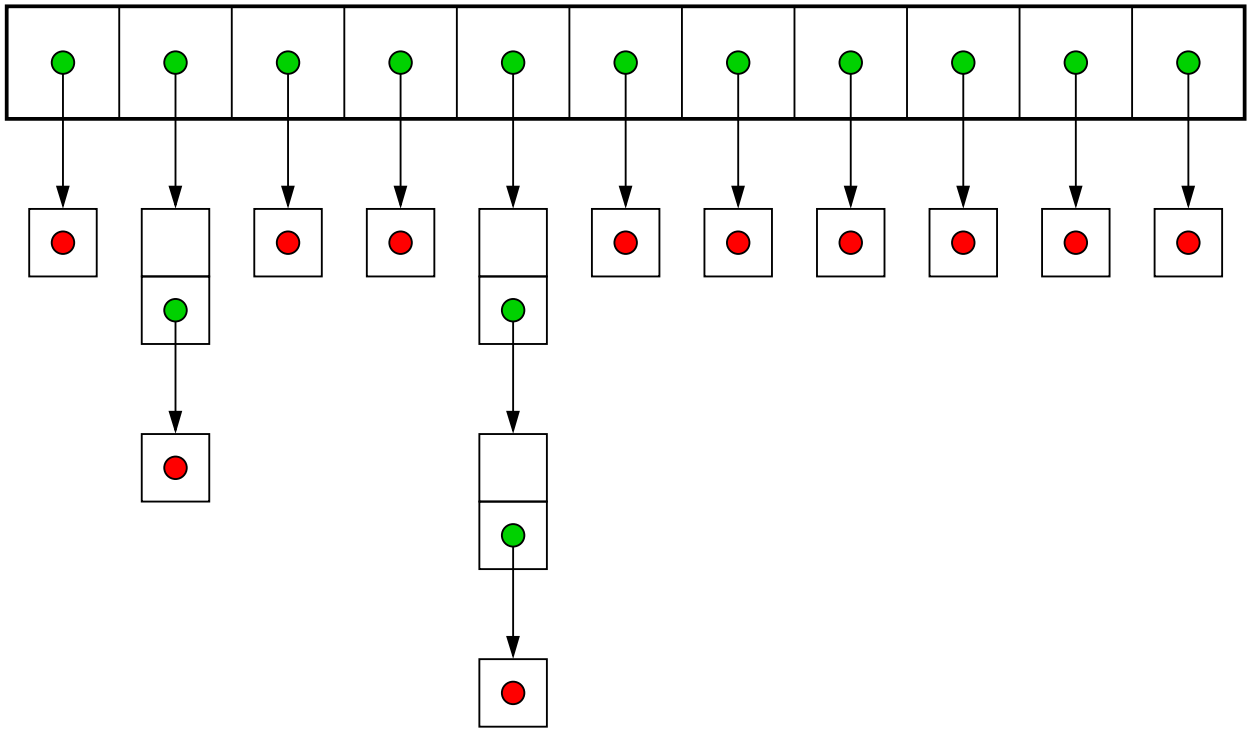
$$\begin{aligned}h_0(s) &= z(0,s) \mod m \\ h_k(s) &= \big(256 \cdot h_{k-1} + z(k,s)\big) \mod m \quad \text{für } k = 1, 2, \ldots, n-1\end{aligned}$$

Wahl des Modulus:

- schlecht: z.B. $m = 256$, da $h(s)$ nur vom letzten Buchstaben abhängt
- gut: Primzahlen

Kollisionsbehandlung: Verkettung

Verkettung (engl. chaining): Objekte werden in Listen verwaltet



- Problem: lineare Suche in lange Listen notwendig (bei hohem Füllfaktor)
- Ausweg: offene Adressierung

Kollisionsbehandlung: Offene Adressierung

Offene Adressierung (engl. open hashing):

- Einfügen: wenn `HT[h(s)]` bereits belegt, dann suche erste freie Stelle $h(s) + k$ für $k = 1, 2, \ldots$
- Suchen: solange Objekt in `HT[h(s)+k]` verschieden von s, inkrementiere k beginnend bei 0

- Problem: Häufungen in bestimmten Bereichen der Hash-Tabelle möglich (Effizienzverminderung)
- Ausweg:
  - Verwendung gut streuender Hash-Funktion (abhängig von Datencharakteristik)
  - Verwendung komplexerer Hash-Verfahren (z.B. *double hashing*, *cuckoo hashing*, *universal hashing* etc.)

Das Hashing-Konzept von Java

Hash-Funktionen:

- Klasse `Object` ist Basisklasse jeder Klasse und enthält virtuelle Methode `int hashCode()`
- durch Überschreiben der Methode in abgeleiteter Klasse Anpassung an Anwendung möglich
- Achtung: Objekte werden durch Referenz gespeichert, `hashCode()` benutzt nur diese Referenz, d.h. Kopie mit anderer Referenz ergibt anderen Rückgabewert von `hashCode()`

Hash-Tabellen:

- Paket `java.util` enthält Klasse `HashTable`
- Klasse `Hashtable` greift auf Methode `hashCode` zu und ist durch sie impliziert parametrisiert
- implementiert ist allgemeiner eine Abbildung (Interface `Map`) von einer Menge von Schlüsselobjekten in einer Menge von Wertobjekten

Das Hashing-Konzept von Java

Klasse `Hashtable` implementiert Interface `Map` mit folgenden Spezifikationen:

- `Object put(Object key, Object value)`
  - legt Objekt `value` in der Map unter `key`
  - falls schon Objekt unter `key` abgelegt war, wird dieses Objekt überschrieben
  - Ergebnis ist das bisher unter `key` abgelegte Objekt (oder `null`)
- `Object get(Object key)`
  - Ergebnis ist das unter `key` abgelegte Objekt (oder `null`)
- `Object remove(Object key)`
  - löscht das unter `key` abgelegte Objekt und gibt es zurück (oder `null`)

Das Hashing-Konzept von Java

Realisierung von Hash-Tabellen:

- zum Schlüsselobjekt wird nummerischer Hash-Wert berechnet mittels `hashCode()`
- in der Hash-Tabelle wird beim Index `hashCode()` das Schlüsselobjekt und das Wertobjekt abgelegt

Kollisionsbehandlung:

- offene Adressierung wird verwendet
- Vergleich von Schlüssel mittels virtueller Methode `equals()` eines Objektes
- beachte: nur Referenz von Schlüsselobjekten wird verwendet

Das Hashing-Konzept von Java: Beispiel 1

```
import java.util.*
public class HashtableExample{
  public static void main(String[] args){
    Hashtable table=new Hashtable();
    String s;
    Integer k;
    s="Anton Wagner";
    k=new Integer(s.hashCode());
    table.put(k,s);
    s="Doris Bach";
    k=new Integer(s.hashCode());
    table.put(k,s);
    s="Doris May";
    k=new Integer(s.hashCode());
    table.put(k,s);
    s="Friedrich Dörig";
    k=new Integer(s.hashCode());
    table.put(k,s);
    :
```

Besonderheiten:

- Wert-Objekte: Personennamen
- Schlüssel-Objekte: Hash-Werte der Namen
- `table.put` benötigt Klassenobjekte als Parameter, deshalb `Integer` statt `int`
- interne Position (Hash-Wert) in `table` ergibt sich aus `hashCode()` des Objektes s:

  es werden also zweimal Hash-Werte berechnet

Das Hashing-Konzept von Java: Beispiel 1

```
    :
    Integer key;
    String s1, s2;
    s1="Doris May";
    key=new Integer(s1.hashCode());
    Object e;
    e=table.get(key);
    if (e == null)
      System.out.println("Kein Element mit Schlüssel " + key +
                                                    " vorhanden!");
    else{
      s2=(String) e;
      System.out.println("Gefundenes Element mit Schlüssel " + key +
                                                      ": " + s2);
    }
  }
}
```

Das Hashing-Konzept von Java: Beispiel 2

```
import java.util.*
public class HashtableExample{
  public static void main(String[] args){
    Hashtable table=new Hashtable();
    String s;
    Date d;
    s="Anton Wagner";
    d=new Date(12,2,1960);
    table.put(d,s);
    s="Doris Bach";
    d=new Date(27,4,1970);
    table.put(d,s);
    s="Doris May";
    d=new Date(24,12,1973);
    table.put(d,s);
    s="Friedrich Dörig";
    d=new Date(1,1,1953);
    table.put(d,s);
    :
    :
```

Besonderheiten:

- Wert-Objekte: Personennamen
- Schlüssel-Objekte: Geburtsdaten
- Annahme: keine zwei Personen mit gleichen Geburtsdatum
- Klasse `Date` wird noch spezifiziert

Das Hashing-Konzept von Java: Beispiel 2

```
    :
    Date sd=new Date(27,4,1970);
    String rs;
    Object e;
    e=table.get(sd);
    if (e == null)
      System.out.println("Kein Element mit Geburtsdatum " + sd +
                                                    " vorhanden!");
    else {
      rs=(String) e;
      System.out.println("Gefundenes Element mit Geburtsdatum " + sd +
                                                        ": " + rs);
    }
  }
}
```

Das Hashing-Konzept von Java: Beispiel 2

```
public class Date{
  private byte day, month;
  private short year;

  // Konstruktoren
  public Date(){ day=1; month=1; }
  public Date(short y){ this(); year=y; }
  public Date(byte m, short y){ this(y); month=m; }
  public Date(byte d, byte m, short y){ this(m,y); day=d; }
  public Date(int d, int m, int y){
    day=(byte) d; month=(byte) m; year=(short) y;
  }

  // Selektoren
  public byte getDay(){ return day; }
  public byte getMonth(){ return month; }
  public short getYear(){ return year; }
  public void setDay(byte d){ day=d; }
  public void setMonth(byte m){ month=m; }
  public void setYear(short y){ year=y; }
  :
```

Das Hashing-Konzept von Java: Beispiel 2

```
  ⋮
  // Darsteller
  public String toString(){ return(day + "." + month + "." + year); }
  // Überschreibe virtuelle Methode equals aus Basisklasse Object
  public boolean equals(Object obj){
    if (!(obj instanceof Date)){ return false };
    return((((Date) obj).day == day) && (((Date) obj).month == month) &&
                                          (((Date) obj).year == year));
  }
  // Überschreibe virtuelle Methode hashCode aus Basisklasse Object
  public int hashCode(){
    return(day + month*32 + year*1024);
  }
}
```