8. Datenstrukturen

8.1 Stack

- Abstrakte Klasse
  - Signaturen der Methoden
  - Vor- und Nachbedingungen

```
abstract class Stack {
  abstract void createStack();
  abstract void push (Object
                        data);
  abstract void pop();
  abstract Object top();
  abstract boolean isempty();
  abstract boolean isfull();
}
```

- Implementierung mit Arrays
  - begrenzter Stack

```
class StackA extends Stack {
private Object s[] = new
                  Object[500];
private int n;
public createStack() {n = 0;}
public push(Object data) {
s[n] = data;
n++;
}
public void pop() { n-- ; }
public Object top() {
  return s[n-1];}
public boolean isempty() {
  return n==0;}
public boolean isfull() {
  return n==500;}
}
```

  - wann dürfen `push`, `pop` und `top` aufgerufen werden?

Anwendung der Klasse `Stack`

- ```
  class Surf {
  StackA urlStack = new StackA();
  void muell() {
  urlStack.createStack();
  urlStack.push("http://www.javasoft.com");
  urlStack.push("http://www.ibr.cs.tu-bs.de");
  }
  ...
  void go(String servername) {
       if (!urlStack.isfull()) {
              urlStack.push(servername);
              showDocument(servername);}
  }
  void goBack() {
       if(!urlStack.isempty()) {
              servername = (String) urlStack.top();
              urlStack.pop();
              showDocument(servername);}
  }
  }
  ```
- Klasse `java.util.Stack`
  - stellt unter anderem Methoden `push, pop, empty, search` zur Verfügung
  - entspricht nicht der Definition unserer abstrakten Klasse `Stack`

8.2 Liste

- Da die Größe nicht beschränkt sein soll, kommen Arrays nicht in Frage
- Für die Realisierung werden Datenstrukturen mit Zeigern gewählt
- Einfach verkettete Liste
  - Jedes Element der Liste verfügt über einen Zeiger auf das nächste Element der Liste: `next`
  - Jedes Element der Liste verfügt zusätzlich über ein Feld mit den jeweiligen Daten des Elements: `data`
  - `next`-Zeiger des letzten Elements der Liste ist `null`, um Ende der Liste anzuzeigen. Andere Möglichkeiten?

- Jeder Knoten der Liste wird als Exemplar einer Klasse ListNode realisiert
  - ```
    class ListNode {
     private ListNode next;     // Nachfolger
     private Object data;       // Referenz auf Daten
    }
    ```
  - Datenteil könnte z.B. die Information für den Stammbaum enthalten
    - ⇨ wie würde das aussehen?

Methoden für die Klasse `ListNode`

```
Class ListNode {
  ...

  public ListNode(ListNode next, Object data) {
   this.next = next; this.data = data;
  }

  public ListNode getNext() {return next;}
  public ListNode getData() {return data;}

  public void setNext(ListNode next) {
   this.next = next;
  }

  public void setData(Object data) {
   this.data = data;
  }
  }
```


Einfach verkettete Liste

```
class LinkedList {
 private ListNode first;
 public LinkedList() {      //leere Liste
   first = null;
 }
 public void setFirstNode(ListNode n) {
   first = n;
   first.next = null;
 }
 public ListNode getFirstNode() {
   if (first != null) return first;
       else return null;
 }
 public ListNode getNextNode(ListNode n) {
    if (n == null) return null;
      else if (n.getNext() != null) return n.getNext();
        else return null;
  }
```

- Durchlaufen einer einfach verketteten Liste?

Einfache verkettete Listen II

❑ Aufbau einer Liste aus verketteten Elementen

●

```
first = null;
while (...) {
 n = new ListNode();
 n.data = ...
 n.next = first;
 first = n;
}
```

● Wie sieht Verzeigerung nach einem Durchlauf aus?

● ... und wie sieht's nach dem zweiten Durchlauf aus?

Anfügen am Ende einer Liste

❑ Angenommen, `first` zeigt auf den Anfang einer Liste. Stellen Sie die Situation graphisch dar:

●

```
n = first;
while (n.next != null) n = n.next;
```

⇨ worauf zeigt `n` jetzt?

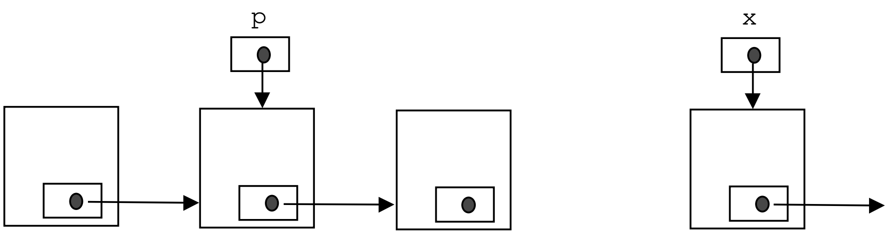
● wie fügt man nun ein Element hinten ein?

● wie sieht die graphische Darstellung aus?

● und kann man `first == null` so abhandeln?

Einfügen in verkettete Listen

- ❏ `x` soll *nach* dem Listenelement eingefügt werden, auf das `p` zeigt.

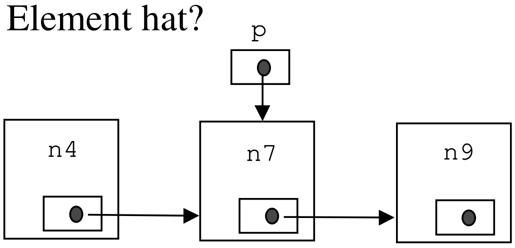


  - welche Operationen sind erforderlich?

  - Anwendung:
    - ⇨ sukzessives Einfügen in eine sortierte verkettete Liste an die richtige Stelle ("Sortieren durch Einfügen", insertion sort)

Einfügen vor einem Listenelement

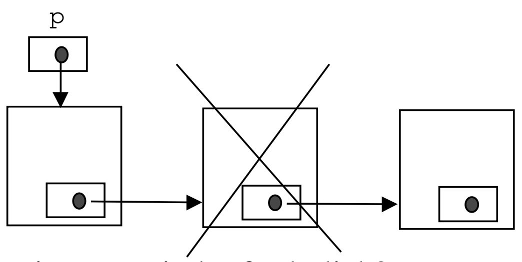
- ❏ Wie kann man `x` *vor* `p` einfügen, wenn man keinen Verweis auf das vorangehende Element hat?



  - wie sieht der schrittweise Ablauf aus?

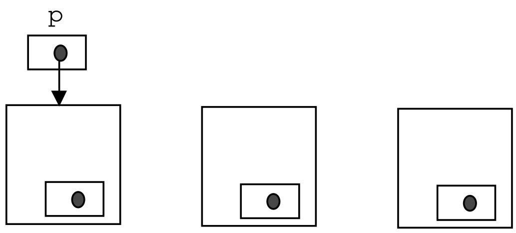
Entfernen eines Listenelements

❑ Es soll das Element nach dem Element auf das `p` verweist gelöscht werden



- welche Anweisungen sind erforderlich?

- wie kann das graphisch dargestellt werden?



Weitere Listentypen

❑ doppelt verkettete Liste:
- neben den Informationsfeldern existieren die Zeiger `next` und `prev`
- Vorteil: Einfügen und Löschen einfacher
- graphische Darstellung?

❑ zyklische Liste:
- Vorteil: Sonderbehandlung z.B. beim Löschen am Listenende entfällt
- Nachteil: zwei Arten von Listenelementen: Listenkopf und "echte" Listenelemente

❑ Wie läßt sich mit Listen ein Stack implementieren? Vorteile bzw. Nachteile gegenüber Implementierung mit Arrays?

Das Josephus-Problem

- Flavious Josephus (37 n.Chr. - 100 n.Chr.)
  - Jüdischer Geschichtsschreiber
  - Belagerung von Masada
    - ⇨ Um der Sklaverei zu entgehen, vereinbarten die 40 Belagerten von Masada, sich im Kreis aufzustellen. Dann sollte jeder siebte getötet werden, bis nur noch einer übrig blieb, der Selbstmord begehen sollte.
    - ⇨ Flavious Josephus hat sich so aufgestellt, daß er als letzter übrig blieb.
- Verallgemeinerung
  - „Abzählreim“: *m* Personen, jeder *n*-te scheidet aus dem Kreis aus.
  - Wie sieht das am Beispiel *m*=8 und *n*=3 aus?

  - Welche Permutation wird geliefert und wer ist der Gewinner?
  - Wie läßt sich das mit Listen implementieren?

8.3 Queue

- Abstrakte Klasse

```
abstract class Queue {
    abstract void enqueue(Object data);
    abstract void dequeue();
    abstract Object front();
    abstract boolean isempty();
  }
```

- Jedes Element der Queue wird als Exemplar einer Klasse QueueNode realisiert

```
class QueueNode {
    private QueueNode next;
    private Object data;
  }
```

Queue

- Implementierung

```
class Rqueue extends Queue {
    private QueueNode p, q;
    public boolean isempty() {return q==null;}
    public void enqueue(Object data) {








  }
  public Object front() {return q.data;}
  public void dequeue() {


    }
  }
```

  - Wie hoch ist der Aufwand für die Methode `enqueue`?

Queue

- Verwendung eines zusätzlichen Zeigers auf das letzte Element

```
class Queue {
  private QueueNode first, last;
  ...
  public void enqueue (Object data) {
     private QueueNode p;
    if (first == null){
          first = new QueueNode();
          first.next = null;
          first.data = data;
          last = first;
      else {
          p = new QueueNode();
          p.data = data;
          last.next = p;
          last = p;
          last.next = null;}
      return first;
    }
```

8.4 Mengen und Mehrfachmengen

- Sehr viele Datenstrukturen dienen der Repräsentation von Mengen (Sets) und Mehrfachmengen (Bags)
  - Ein Bag ist eine Menge, in der ein Element mehrfach vorkommen kann. Statt dem Element-Test gibt es eine Funktion, die sagt, wie oft ein Element in der Menge vorkommt.
- Die Auswahl einer Datenstruktur für Sets und Bags hängt von vielen Faktoren ab, beispielsweise
  - Ordnung: eine Ordnungsrelation auf den Elementen kann zum Beschleunigen von Operationen (z.B. Suche) ausgenutzt werden
  - Größe: kleine Mengen lassen sich effizienter realisieren als große
  - Beschränkte Mengen: haben maximal n Elemente. Hier kann man anders vorgehen als bei unbeschränkten Mengen
  - Schlüssel: ein Schlüssel ist eine ausgezeichnete Komponente eines Datenobjektes, der das Objekt eindeutig identifiziert. Damit wird der Objektinhalt aus Datenstruktursicht belanglos.
  - Operationen: welche Operationen werden wirklich benötigt?
  - Schlüsselstruktur: kann manchmal zur Verbesserung der Verfahren ausgenutzt werden
  - vorausichtliche Häufigkeit der einzelnen Operationen: wenn diese bekannt sind, kann die Datenstruktur darauf hin optimiert werden.

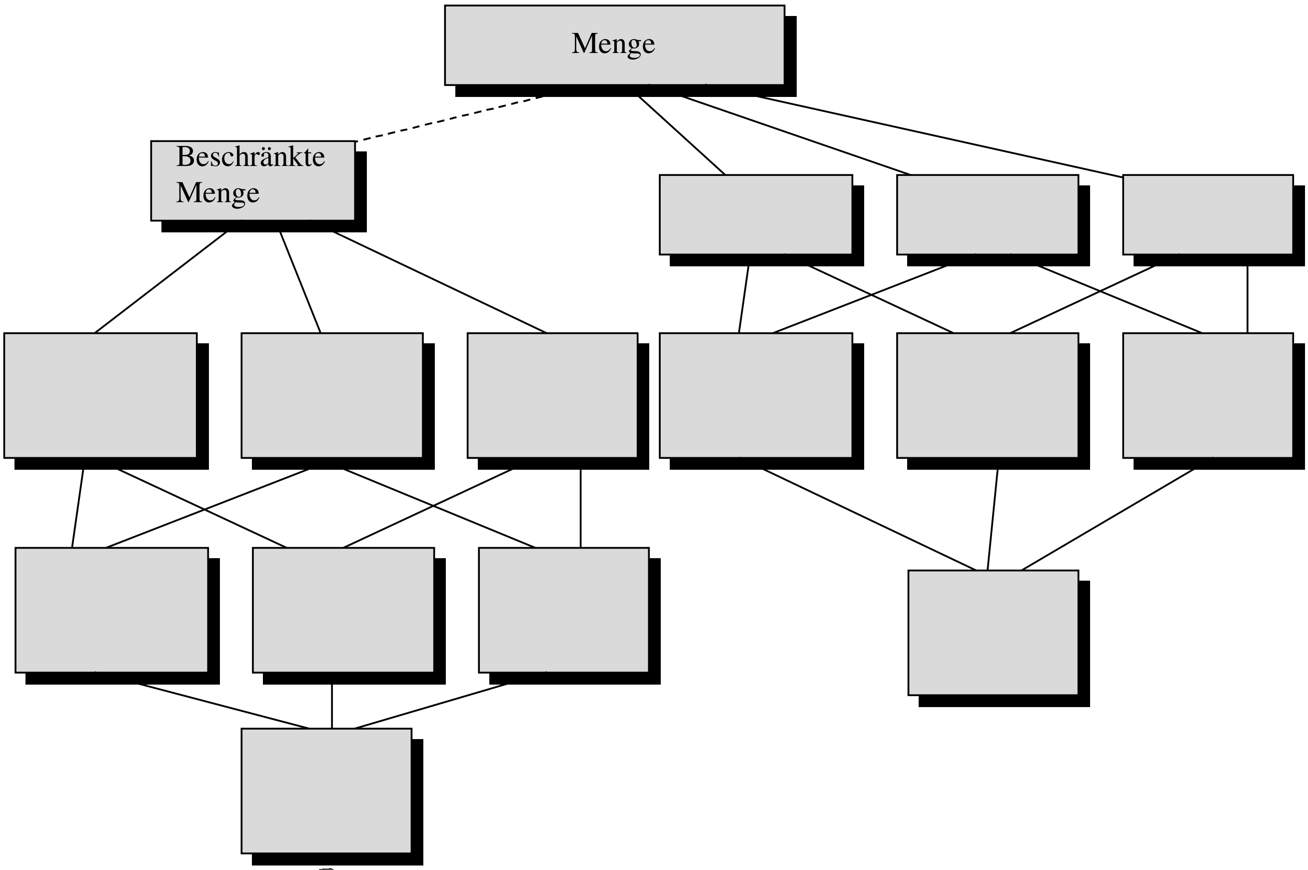
ADT für Bags

- Signatur

```
init:
create:
add:
insert:
delete:
member:
is_n_elem:
union:
intersect:
diff:
empty:
```

- Geben Sie definierende Gleichungen für die Operationen an.

Eine Taxonomie der Mengen



Mengen als Arrays

- Außer dem Array braucht man einen Pegelzähler, der die Anzahl der Einträge angibt.
- 2 Möglichkeiten
  - Elemente werden unsortiert gespeichert. Welchen Aufwand haben die Operationen?
    - ⇨ Init , create , add , insert , delete
member , is_n_elem , union , intersect
diff , empty
  - Elemente werden sortiert gespeichert (geordneter Schlüssel erforderlich). Wie sieht's hier mit dem Aufwand der Operationen aus?
    - ⇨ Init , create , add , insert , delete
member , is_n_elem , union , intersect
diff , empty

- Nachteil einer Array-basierten Implementierung?

Mengen als Listen

- Listen können ebenfalls zur Implementierung von Mengen verwendet werden
- Wie bildet man die Vereinigung zweier Mengen S und T mit unsortierten Listen?

- Hilft es, sortierte Listen zu verwenden?

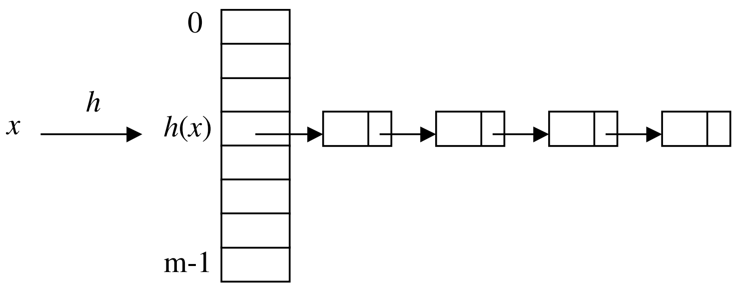
Bitvektoren

- Kleine, endliche Mengen von Elementen lassen sich effizient als Bitvektoren implementieren
  - Sei $G = [n..m]$ die Grundmenge.
  - Jede Teilmenge $M \subseteq G$ läßt sich durch ihre charakteristische Funktion $c : G \rightarrow B$ darstellen
  - Diese Funktion kann durch ein boolsches Array implementiert werden

    ➩ wie wird dann $x \in M$ dargestellt?
- Die Mengenoperationen lassen sich folgendermaßen implementieren
  - `init`
  - `insert(x,c)`
  - `delete(x,c)`
  - `member(x,c)`
  - `empty(c)`
  - `union(c,d)`
  - `intersect(c,d)`
- Wie hoch ist der Aufwand für die Operationen?

Hashverfahren

- Hashverfahren sind für sehr große Mengen geeignet
  - Beispiel: Wörterbuch
    - ⇨ Es kann nicht für jedes Wort eine Position in einem Array bereitgestellt werden, wie dies bei Bitvektoren der Fall ist
- Hashfunktion
  - Schlüsselmenge G
  - $h: G \rightarrow [0..m-1]$ bildet die Schlüssel auf Indizes ab
    - ⇨ es sind unendlich große Schlüsselmengen G zulässig
  - h ist in der Regel nicht injektiv, d.h. verschiedene Schlüssel können auf den gleichen Index abgebildet werden: $h(x) = h(y)$
  - Ist h injektiv, so heißt h eine perfekte Hashfunktion



Hashtabelle

- Der Bitvektor stellt nun eine Tabelle dar
  - `Object hashtable[] = new Object[m]`
  - Wo wird das Element x gespeichert?
  - Lastfaktor α der Tabelle
    - ⇨ Bei n Elementen: $\alpha = n / m$
  - Tabelle sollte möglichst gleichmäßig belegt sein
- Anforderung
  - Hashfunktion muß sehr schnell berechenbar sein
- Beispiel Zeichenketten
  - Grundmenge G = {MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG, FREITAG, SAMSTAG, SONNTAG}
  - $B = \{1, ..., 22\}$
  - $h: G \rightarrow B$ $\quad h(k) = f(1.$ Buchstabe von $k) + f(2.$Buchstabe von $k) - 12$
  - Berechnen Sie die Werte der Hashfunktion für die oben angegebene Grundmenge
- Beispiel Zahlen
  - G = N
  - $h: N \rightarrow [0..m-1]$
  - $h(x) = a * x \bmod m$ mit teilerfremdem a und m.

Berechnung der Kollisionswahrscheinlichkeit

- Annahme:
  - Es werden n Elemente $x_1, ..., x_n$ eingefügt. Jeder Hashwert ist gleichwahrscheinlich.
- Kollisionswahrscheinlichkeit $K(n)$ beim Einfügen von n Elementen:
  - K(1) = 0
  - K(2) =
  - Wahrscheinlichkeit für eine Kollision bei n+1 Einträgen?

- Beispiele:

n	m=100	m=365
5	0.0965	0.0271
10	0.3718	0.1169
23	0.9357	0.5072
50	0.9999	0.9703
101	1.0000	0.9999

Verkettetes Hashen

- Einfügen in die Kollisionsliste
  - ```
    ListNode hashtable = new ListNode[m];
    ListNode q;
    int i;
    ...
    i = h(x);
    q = hashtable[i];
    q.next = list.insert(q,x);
    ```
  - Wie hoch ist der Aufwand?
- Suchen nach x
  - ```
    ListNode p;
    p = hashtable[h(x)];
    while ((p!=null)&&(p.data!=x))
      p = p.next;
    ```
  - Wie hoch ist hier der Aufwand?
- Aufwand im Durchschnitt?
  - Im Durchschnitt hat jede Kollisionsliste die Länge α
- Wie löscht man ein Element aus einer Hashtabelle?

Offenes Hashen

- Elemente werden direkt in die Hashtabelle eingetragen.
  - Länge der Tabelle legt maximale Anzahl der speicherbaren Elemente fest
- Kollisionsauflösung
  - Verwendung eines zusätzlichen Arrays
    `boolean status[] = new boolean[m];`
  - Wenn Ausweichpositionen bereits belegt sind, müssen neue berechnet werden
    $c: I \times G \rightarrow I$
  -
```
k0 = h(x);
k = k0;
i=0;
while(status[k] && (i<m)) {
 i++;
 k = (k0+c(i,x)) % m;
}
if (!status[k]) {
 hashtable[k] = x;
 status[k] = true;
}
else
 Ueberlauf();
```


Suchen und Löschen bei offenem Hashen

- Suchen

```
k0 = h(x);
k = k0; i = 0;



found = hashtable[k]=x;
```

- Löschen
  - Es sind Einträge in der 1. Ausweichposition und in der 2. Ausweichposition vorhanden. Der Eintrag an der 1. Ausweichposition wird gelöscht. Was hat dies für Folgen?

Berechnung von Ausweichpositionen

- Man unterscheidet Primärkollisionen und Sekundärkollisionen
- Lineares Sondieren
  - $c(i,x) = i$. Die Elemente werden am nächsten freien Platz eingetragen
  - Problem?
  - Wie groß kann α werden?
- Quadratisches Sondieren
  - $c(i,x)=i^2$. Die Abstände zwischen den Einträgen werden immer größer.
  - Wie sieht es hier mit α aus?
- Doppeltes Hashen
  - $c(i,x) = i * g(x)$, wobei g eine weitere Hashfunktion ist mit $0 < g(x) < m$
  - Was passiert bei $g(x) = 0$ oder $g(x) = m$?

Suchbäume

- Bäume sind die Alternative zu Hashverfahren, wenn die Datenmenge sich dynamisch stark ändert
- Aufwand für Einfügen/Löschen/Suchen: O(ln n)
  - Im Worst-Case allerdings O(n)
- Bäume können folgendermaßen klassifiziert werden
  - Suchbaum/Entscheidungsbaum
    - ⇨ Ausgehend von der Wurzel wird ein Knoten mit einer bestimmten Eigenschaft gesucht. An jedem Knoten wird anhand des Inhalts entschieden, wie weitergesucht wird.
  - Gerichteter/ungerichteter Baum
    - ⇨ Enthalten entweder Verzweigungen von Knoten zu ihren Söhnen oder Verzweigungen von Knoten zu Vätern. Ungerichtete Bäume haben keine Vorzugsrichtung
  - geordneter/ungeordneter Baum
    - ⇨ Die Reihenfolge der Söhne spielt bei geordneten Bäumen eine Rolle, bei ungeordneten nicht.
  - Mehrwegbaum/Binärbaum
    - ⇨ In Binärbäumen hat jeder Knoten höchstens zwei Söhne, in Mehrwegbäumen beliebig viele.

Klassifikation von Bäumen

- Blattbaum/natürlicher Baum
  - ➩ In Blattbäumen haben nur die Blätter vollständige Knoteninformation. Die inneren Knoten enthalten nur Information zum Steuern der Suche. In einem natürlichen Baum verfügen alle Knoten über vollständige Information.
- Ausgewogener/nicht ausgewogener Baum
  - ➩ In ausgewogenen Bäumen unterscheiden sich für jeden Knoten die Höhe der Teilbäume höchstens um 1.
- Optimaler Suchbaum
  - ➩ Wenn man für n Schlüssel d_i ihre Zugriffswahrscheinlichkeit p_i kennt, kann man mit dem Verfahren von Huffmann einen Suchbaum konstruieren, so daß der Suchaufwand minimal wird.
    Ist t_i die Tiefe des Knotens d_i in diesem Baum, so bedeutet die Minimalitätseigenschaft, daß der Erwartungswert der Weglänge beim Suchen minimal wird $\sum_{i=1}^{n} p_i \cdot t_i = \min$

    Im Spezialfall gleicher Zugriffswahrscheinlichkeiten ergeben sich ausgeglichene Bäume.

Binäre Suchbäume

- ❏ Die Grundmenge G der Schlüssel ist geordnet
- ❏ Invariante für binäre Suchbäume
  - alle Schlüssel im linken Unterbaum sind kleiner, alle Schlüssel im rechten Unterbaum sind größer als der Wurzelschlüssel.
  - Dies gilt rekursiv für alle Unterbäume
- ❏ Aufbau
  - ```
    class BinBNode {
     private BinBNode rightson;
     private BinBNode leftson;
     private Object data;
    }
    ```
- ❏ Suchbaumeigenschaft teilt die Menge aller Schlüsselwerte im Baum *M* in zwei Teilmengen: $M_l = \{x \in M \mid x < v\}, M_r = \{x \in M \mid x > v\}$ mit Wurzelschlüssel *v*.

  - Wo steht der kleinste bzw. der größte Schlüssel in einem binärem Suchbaum ?
    ```

Operationen auf binären Suchbäumen

❑ Suchen

- zu Schlüssel `x` wird der entsprechende Knoten oder `null` zurück gegeben
- ```
  btree search(int x, btree t) {
    while ((t != null) && (t.data!=x)) {
           if (x < t.data) {t = t.leftson;}
           else {t = t.rightson;}
    return t;
  ```

❑ Einfügen

- `newp` zeigt auf den neuen Baumknoten; Nachfolger sind `null`
- ```
  insert(btree newp, btree t) {
    if (t == null) {
      t = newp;  }  //... Set rest of the fields
    else if (newp.key < t.leftson.key)
      insert(newp, t.leftson);
           else insert(newp, t.rightson);
  }
  ```
- funktioniert diese Lösung in Java?

Operationen auf binären Suchbäumen

❑ Löschen

- Ein Knoten muß so aus dem Suchbaum entfernt werden, daß der Rest ein Suchbaum bleibt.
  - ➩ Bei Blättern ist das einfach.
    Wie geht's?
  - ➩ Bei inneren Knoten mit nur einem Nachfolger ist es ebenfalls einfach.
    ... Und wie geht's hier?
  - ➩ Bei inneren Knoten mit zwei Nachfolgern ist es komplizierter
    Wodurch wird hier ersetzt?

Durchschnittliche Suchkomplexität

- Gegeben sei ein binärer Baum mit n Knoten. Gesucht ist der Aufwand zum Suchen eines Schlüssels im Baum
  - Im besten Fall ist der Baum voll ausgeglichen. Wieviel Suchschritte werden also höchstens benötigt?
  - Wie sieht der schlimmste Fall aus? Wie entsteht er?
  - Interessant ist auch der durchschnittliche Fall. Annahme: alle Knoten sind gleichwahrscheinlich. Der linke Unterbaum habe i Knoten (der rechte dann also n-i-1 Knoten). Die Wahrscheinlichkeiten zum Finden sind wie folgt:
    - ➪ an der Wurzel: $1/n$
    - ➪ im linken Unterbaum: i/n
    - ➪ im rechten Unterbaum: $(n-i-1)/n$
  - Dann ergibt sich für die Zahl der Suchschritte $K(n)$
    - ➪ $K(0) = 0$, $K(1) = 1$
      $K(n) = ?$

AVL-Bäume

- Durch Balancieren können unbalancierte Bäume stärker ausgeglichen werden.
  - Wie lohnenswert ist eine bessere Balancierung?

- Definition eines AVL-Baums
  - Sei $h(e)$ die Höhe des Teilbaums mit Wurzel e. Das Verfahren von Adelson/Velski/Landin stellt folgende Bedingung an die Balancierung eines Baums:
    - ➪ $|h(e.leftson) - h(e.rightson)| <= 1$

- Beispiele

Rotationsoperationen

- Rotationsoperationen können die Balance an einem Knoten verbessern. Sie gruppieren Unterbäume so um, daß die Suchbaumeigenschaft erhalten bleibt, aber die Höhenunterschiede geringer werden.
- Man unterscheidet Einfach- und Doppelrotation:

Implementierung der Rotation

- Links-Einfachrotation an Knoten *k*
  - ```
    p = k.leftson;
    k.leftson = p.rightson;
    p.rightson = k;
    ```
- Links-Doppelrotation an Knoten *k*

- Wie sieht Code für die Rechts-Einfachrotation und die Rechts-Doppelrotaton aus?
- Zeigen Sie, daß auch Doppelrotationen die Suchbaumeigenschaft invariant lassen.
    ```

Einfügen in AVL-Bäume

- Jeder Knoten *e* bekommt ein zusätzliches Feld
  - `hdiff: [-1 ... 1]`
    mit $e.hdiff = h(e.leftson) - h(e.rightson)$
- Das Einfügen geschieht zunächst wie in gewöhnlichen binären Suchbäumen. Bei der Rückkehr aus der Rekursion können sich die folgenden Fälle ergeben
  - Vor dem Einfügen
    - ⇨ $h(e.leftson) < h(e.rightson)$, $e.hdiff = -1$
  - Nach dem Einfügen

  - Aktion?
  - Vor dem Einfügen
    - ⇨ $h(e.leftson) < h(e.rightson)$, $e.hdiff = 0$
  - Nach dem Einfügen

  - Aktion?
  - Vor dem Einfügen
    - ⇨ $h(e.leftson) < h(e.rightson)$, $e.hdiff = 1$
  - Nach dem Einfügen

  - Aktion?

Komplexitätsanalyse von AVL-Bäumen

- Können AVL-Bäume zu linearen Listen degenerieren?
  -
  - Wieviele Knoten hat ein AVL-Baum einer Höhe *h* mindestens?

Komplexitätsanalyse von AVL-Bäumen

- Korollar
  - Die Höhe h eines AVL-Baums mit n Knoten ist höchstens O(ln n).
- Beweis
  - Folgendes gilt:

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \approx \frac{1{,}618^n}{2{,}236}$$

$$denn\ \lim_{n\to\infty}\left(\frac{1-\sqrt{5}}{2}\right)^n = 0$$

und deshalb

$$n \geq F_{h+3} - 1 \approx \frac{1{,}618^{h+3}}{2{,}236} - 1$$

also

$$h \leq \log_{1.618}(2{,}236(n+4)) - 1$$

$$\approx 1{,}44 \cdot \ln(n)$$

  - Die Höhe und damit die Anzahl der Suchschritte ist also im schlimmsten Fall ca. 1,44*ln n. Im durchschnittlichen Fall ist das Suchen besser.
  - Bei gewöhnlichen Binärbäumen ist der durchschnittliche Fall bereits 1,44*ln n.

Rot-Schwarz-Bäume

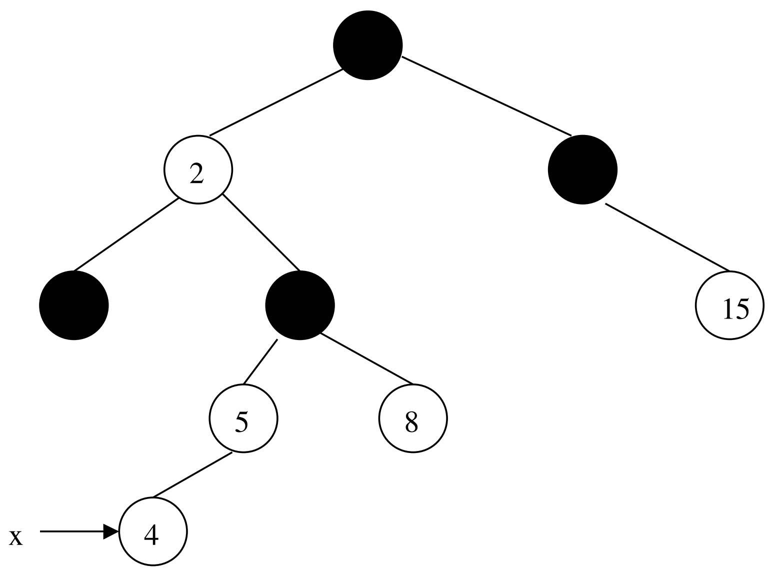
- Alternative zu AVL-Bäumen
- Alle Knoten werden entweder rot oder schwarz eingefärbt. Dabei gelten die folgenden Regeln
  - Jeder Knoten ist entweder schwarz oder rot
  - Wenn ein Knoten rot ist, ist er entweder ein Blatt oder hat zwei schwarze Söhne
  - für jeden Knoten k gilt: alle Pfade von k zu einem Blatt enthalten dieselbe Zahl von schwarzen Knoten
- Die Pfade von der Wurzel zu den Blättern in Rot-Schwarz-Bäumen sind so gestaltet, daß keiner mehr als doppelt so lang ist als irgendein anderer Pfad im Baum.
- Es gilt
  - Rot-Schwarz-Bäume haben stets eine Höhe O(ln n)
- Begründung?

Operationen auf Rot-Schwarz-Bäumen

- Einfügen
  - Der neue Knoten wird wie in einem gewöhnlichen Suchbaum eingefügt und rot gefärbt.
  - Was ist zu tun, wenn der Vater schwarz ist?
  - Was ist zu tun, wenn der Vater rot ist?

Beispiel Einfügen

- Das Beispiel zeigt einen Rot-Schwarz Baum, in den der Knoten x eingefügt wurde.
  - Welche Schritte sind durchzuführen, um wieder einen regulären Rot-Schwarz Baum zu erzeugen?



Mehrwegbäume, Tries

- Tries (sprich „try“) sind Mehrwegbäume, wobei die Kanten mit den Elementen eines Alphabets markiert sind. Sie eignen sich besonders zur kompakten Speicherung von Wörterbüchern.
- Bei Verwendung des Alphabets A={0, 1} spricht man von binären Tries
- Aufbau
  - ```
    class TrieNode {
      private TrieNode[] sons = new TrieNode[n] ;
      private boolean flag;
    }
    ```
  - Das Flag dient dazu Wortenden zu markieren

Beispiel

- Speicherung einer Menge von Worten in einem Trie:
  - {herz, herr, herzog}
  - Wie sieht der Trie aus?
    ```

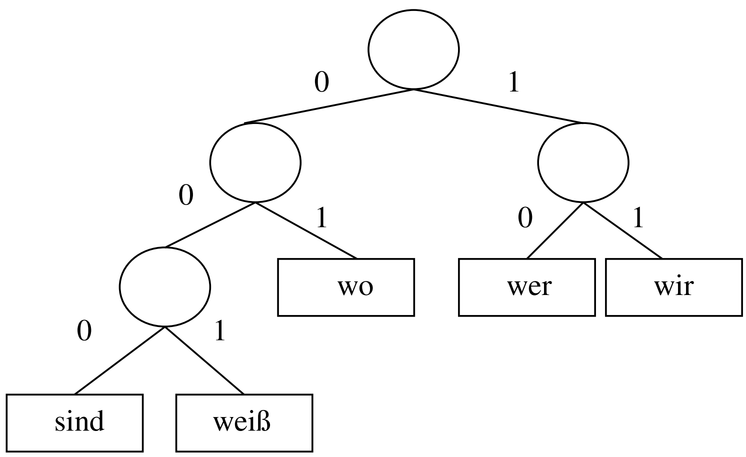
Einfügen von w in einen Trie

- TrieNodeInsert(TrieNode p, int i, int word[])

```
TrieNodeInsert(TrieNode p, int i, int word[])
{
    if (p==null)
      p = newTrieNode();
    if (i>=word.length)
      p.flag = true;
    else
      p.sons[word[i]] =
          insert(p.sons[word[i]], i+1, word);
    return p;
}
```


Anwendung: Komprimierung

- Binäre Tries sind für die Komprimierung von besonderem Interesse
- Die eigentlichen Wörter sind an den Blättern plaziert. Gesucht wird nach Wörtern über dem binären Alphabet {0,1}.



- Ein Code ist dann gut, wenn häufig vorkommende Wörter besonders kurze Codes haben
- Wie wird „wer weiß wo wir sind“ kodiert?

B-Bäume

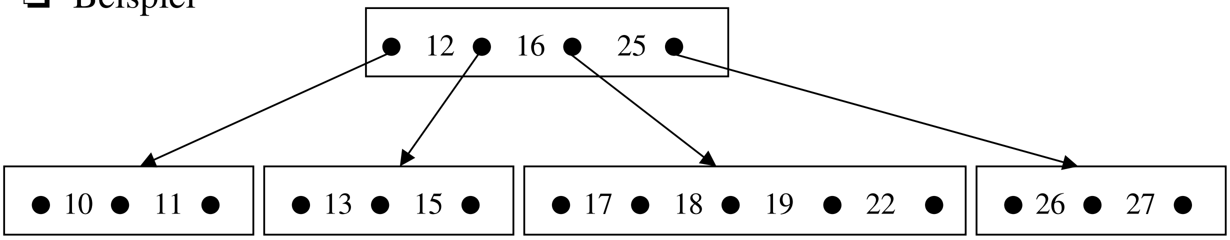
- ❑ Mehrwegbäume zum Speichern großer Datenmengen auf externen Speichern
  - Die zuvor diskutierten Bäume haben stets angenommen, daß sich die Daten im Hauptspeicher des Systems befinden.
  - Bei großen Datenmengen werden die Daten aber auf Bedarf aus dem Hintergrundspeicher in den Hauptspeicher geladen.
- ❑ Beispiel
  - Studentendatei mit den Attributen Matrikelnummer, Name, Fach und Semester
  - Satzschlüssel Matrikelnummer identifiziert die Sätze der Datei
    - ➩ es wird angenommen, daß die Schlüssel ganzzahlig sind
  - Bei Speicherung auf einer Diskette benötigt man Spur und Sektor, um einzelne Blöcke adressieren zu können. Jeder Block enthält mehrere Sätze.
  - Indextabelle
    - ➩ Information über die Blockadresse, unter der ein durch einen Satzschlüssel identifizierter Satz gespeichert ist.
    - ➩ Ist der Index sehr groß, so kann er nicht mehr komplett im Hauptspeicher gehalten werden, d.h. Teile des Indexes sind selbst auf dem Hintergrundspeicher
    - ➩ Index kann als B-Baum gehalten werden

B-Bäume

- ❑ B-Baum der Ordnung m
  - Alle Blätter haben die gleiche Tiefe
  - Jeder Knoten mit Ausnahme der Wurzel und der Blätter hat mindestens $\left\lceil \frac{m}{2} \right\rceil$ Söhne
  - Die Wurzel hat mindestens 2 Söhne
  - Jeder Knoten hat höchstens m Söhne
  - Jeder Knoten mit i Söhnen hat i-1 Schlüssel
- ❑ Beispiel

B-Bäume

- Aufbau eines Eintrags
  - ```
    class Btree {
     public final static int depth = 100;
     int l;                             // Sohnzahl
     int[] s = new int[depth];          //Schlüssel
     Btree[] p = new Btree[depth];      //Söhne
    }
    ```
- Beispiel



  - Folgende Werte sollen nacheinander eingefügt werden: 14, 24, 23, 21, 20

Operationen auf B-Bäumen

- Suchen
  - Beginn an der Wurzel
  - Vergleich, ob Schlüssel *x* im aktuellen Knoten gespeichert ist, ansonsten Bestimmen des nächsten Knotens
- Einfügen
  - Knoten (d.h. Seite) für *x* suchen
  - falls noch nicht voll, einsortieren von *x* an der passenden Stelle
  - falls voll, aufteilen der Seite in zwei neue Seiten. Trennender Schlüsselwert muß in der Vaterseite aufgenommen werden
  - falls innere Seite überläuft: Aufteilen, Erweitern der Vaterseite usw.
    - ⇨ Alle Seiten (evtl. bis auf die Wurzel) haben eine Auslastung von 50% oder mehr
    ```

Operationen auf B-Bäumen

- Löschen
  - suche Datenseite für *x*
  - lösche Eintrag für *x*
  - falls Auslastung der Seite noch größer oder gleich 50% ist: ok.
  - Falls Unterlauf: Verschmelzen der Seite mit Nachbarseite; Entfernen des entsprechenden Eintrags auf der Vaterseite
  - falls Vaterseite unterläuft: Verschmelzen mit der Nachbarseite
    - ⇨ der Baum kann sich bis auf eine Wurzelseite reduzieren
  - Löschen Sie aus dem entstandenen Baum den Wert 17

Dynamisches Hashen

- Alternative zu B-Bäumen; kombiniert Hashtabellen mit Suchbäumen

- Hashtabelle wird in Seiten aufgeteilt. In jeder Seite wird offenes Hashen verwendet.

- Zusätzlich gibt es einen binären Suchbaum, um die passende Seite zu lokalisieren. An jedem Blatt eines Suchbaums hängt eine Seite mit einer Hashtabelle

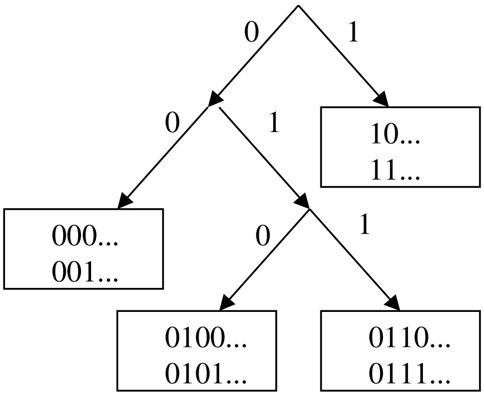
- Hashwerte werden als Bitstrings interpretiert. Das nächste Bit sagt, ob man im Suchbaum links oder rechts gehen muß, um die passende Seite zu finden.

- Hängt die Seite am Blatt auf Ebene k, so werden die ersten k-1 Bits des Hashwertes zum Lokalisieren der Seite verwendet.

- Der Suchbaum wird auf einer extra Seite gespeichert.

Dynamisches Hashing

- Beispiel



- Suchen
  - Berechne Hashwert als Bitstring
  - Bestimme Blatt im Suchbaum anhand des Bitstrings
  - Suche mit offenem Hashen in der entsprechenden Seite

Dynamisches Hashing

- Einfügen
  - Berechne Hashwert als Bitstring
  - Bestimme Blatt im Suchbaum anhand des Bitstrings
  - Füge Datensatz mit offenem Hashen in der entsprechenden Seite ein
  - falls Seite voll: Aufspalten. Eine Seite der Ebene k wird anhand des k-ten Bits der Schlüssel auf zwei neue Seiten der Ebene k=1 aufgeteilt.
  - Der Suchbaum wird entsprechend um 2 Blätter erweitert, die als Söhne an das alte Blatt angehängt werden
  - Alle Datensätze mit Bit k=0 kommen in die linke neue Seite. Die mit Bit k=1 in die rechte neue Seite.
- Löschen
  - analog.
  - Falls zwei benachbarte Seiten auf eine Seite passen, werden sie verschmolzen und die beiden entsprechenden Blätter im Suchbaum gelöscht.
- Beispiel
  - Fügen Sie den Datensatz mit Schlüssel $h(x)$=0011010 ein. Nehmen Sie an, daß es zu einem Seitenüberlauf kommt.