Wiederholung

Baum:

- Gerichteter Graph, der die folgenden drei Bedingungen erfüllt:
  - Es gibt einen Knoten, der nicht Endknoten einer Kante ist. (Dieser Knoten heißt Wurzel des Baums.)
  - Jeder andere Knoten ist Endknoten genau einer Kante.
  - Der Graph ist zusammenhängend: Zu jedem Knoten existiert ein Weg von der Wurzel aus.

$\leadsto$ Bäume sind zyklenfrei.

Rekursive Definition:

- Baum = Wurzelknoten + disjunkte Menge von Kindbäumen.

Wiederholung

(Ausgangs-)Grad eines Knotens = Anzahl seiner Kinder.

- Knoten mit Grad 0 heißen Blätter.
- Knoten mit Grad ≥ 1 heißen innere Knoten.

Der Grad eines Baums ist der maximale Grad seiner Knoten.

Wiederholung

Suchbäume

- Binärbaum: Jeder Knoten hat höchstens zwei Kinder ($\leadsto$ linker/rechter Unterbaum).
- Knoten haben Markierungen (aus irgendeiner total geordneten Menge).
- Für jeden Knoten gilt:
  - Alle Markierungen im linken Unterbaum haben kleinere Markierungen.
  - Alle Markierungen im rechten Unterbaum haben größere Markierungen.

Suchbäume: Einfügen

Einfügen einer Zahl n in einen Suchbaum:

- falls n gleich dem Wert des Knotens ist: nichts zu tun.
- falls n kleiner als der Wert des Knotens ist:
  - falls es keinen linken Kindknoten gibt: erzeuge linken Kindknoten mit Wert n.
  - sonst rufe Prozedur rekursiv für linken Kindknoten auf.
- falls n größer als der Wert des Knotens ist:
  - falls es keinen rechten Kindknoten gibt: erzeuge rechten Kindknoten mit Wert n.
  - sonst rufe Prozedur rekursiv für rechten Kindknoten auf.

Suchbäume: Einfügen

```
public class Knoten {
  Knoten kindL;
  Knoten kindR;
  int wert;

  Knoten(int n) {
    wert = n;
  }

  public void fuegeEin(int n) {
    ...
  }
}
```

```
public void fuegeEin(int n) {
  if (n < wert) {
    if (kindL == null) {
      kindL = new Knoten(n);
    } else {
      kindL.fuegeEin(n);
    }
  } else if (n > wert) {
    if (kindR == null) {
      kindR = new Knoten(n);
    } else {
      kindR.fuegeEin(n);
    }
  }
}
```


Suchbäume: Einfügen

Einfügen + Ausgeben (Infix-Darstellung) = Sortieren

(im wesentlichen äquivalent zu *Quicksort*)

Suchbäume: Löschen

Löschen einer Zahl *n* in einem Suchbaum.
Die einfachen Fälle:

n kommt im Suchbaum nicht vor: nichts zu tun.

n kommt in einem Blatt vor: lösche das Blatt.

n kommt in einem inneren Knoten vor,
der keinen linken Kindknoten hat:
ersetze den Teilbaum ab diesem Knoten
durch seinen rechten Unterbaum.

n kommt in einem inneren Knoten vor,
der keinen rechten Kindknoten hat:
ersetze den Teilbaum ab diesem Knoten
durch seinen linken Unterbaum.

Suchbäume: Löschen

Löschen einer Zahl *n* in einem Suchbaum.
Der schwierige Fall:

n kommt in einem inneren Knoten vor,
der einen linken *und* einen rechten Kindknoten hat:

Suche den Vorgängerknoten,
d. h., den Knoten mit der nächstkleineren Markierung.

Ersetze den Knoten *n* durch den Vorgängerknoten.

Lösche den Vorgängerknoten an seiner bisherigen Position.
(Einfach, denn dieser hat keinen rechten Kindknoten!)

Suchbäume: Komplexität

Zeitbedarf für Einfügen, Suchen, Löschen: $O(\text{Tiefe})$.

Wie tief ist ein Baum mit n Knoten?

Im günstigsten Fall: $\lceil \log_2(n+1) \rceil - 1$, d. h., $O(\log n)$.

Im Durchschnitt (falls der Baum durch wiederholtes Einfügen von n zufällig erzeugten Werten entstanden ist): $O(\log n)$.

Im schlechtesten Fall: $n - 1$, d. h., $O(n)$.

Der schlechteste Fall (degenerierter Baum $\approx$ Liste) tritt unter anderem dann auf, wenn eine bereits sortierte Folge von Elementen nacheinander eingefügt wird.
(Problem: dieser Fall ist praktisch relevant.)

Balancierte Bäume

Ausweg: Versuche, den Baum so aufzubauen, daß er *balanciert* („einigermaßen ausgeglichen“) ist.

Was bedeutet das?

Mehrere Möglichkeiten, z. B.:

Für jeden Knoten müssen der linke und der rechte Unterbaum ungefähr gleich viele Knoten besitzen.

Für jeden Knoten müssen der linke und der rechte Unterbaum ungefähr gleich tief sein.

Balancierte Bäume: AVL-Bäume

Beispiel: AVL-Bäume (Adelson-Velskii und Landis)

Ein binärer Baum ist ein AVL-Baum, falls er folgende Bedingung erfüllt:

für jeden Knoten unterscheidet sich die Tiefe des linken und des rechten Unterbaums höchstens um 1.

(Ein nicht-vorhandener Unterbaum wird als leerer Unterbaum betrachtet; Tiefe $= -1$.)

Balancierte Bäume: AVL-Bäume

Problem: Beim Einfügen oder Löschen kann die AVL-Eigenschaft verletzt werden.

$\rightsquigarrow$ Der Baum muß dann wieder balanciert werden.

Technik: Rotieren von Teilbäumen.

Balancierte Bäume: (a, b)-Bäume

Beispiel: (a, b)-Bäume.

a, b natürliche Zahlen, $2 \leq a$, $2a - 1 \leq b$.

Ein Baum ist ein (a, b)-Baum, falls er folgende Bedingungen erfüllt:

- alle Blätter haben die gleiche Tiefe,
- alle inneren Knoten haben Grad $\leq b$,
- alle inneren Knoten außer der Wurzel haben Grad $\geq a$,
- falls die Wurzel kein Blatt ist, hat sie Grad ≥ 2.

Balancierte Bäume: (a, b)-Bäume

Im Gegensatz zu den bisher betrachteten Suchbäumen werden in (a, b)-Bäumen die eigentlichen Daten nur in den Blättern abgespeichert (in aufsteigender Reihenfolge von links nach rechts).

Die inneren Knoten enthalten Hilfsinformationen:

- Die Markierung eines inneren Knoten mit Grad n besteht aus $n - 1$ Elementen $k_1 < \ldots < k_{n-1}$.
- Die Blätter im i-ten Unterbaum des Knotens haben Werte größer oder gleich k_{i-1} und kleiner als k_i.

Balancierte Bäume: (a, b)-Bäume

Einfügen in (a, b)-Bäume:

- Suche untersten inneren Knoten, an den das neue Blatt gehört.
- Trage neues Blatt ein. $\rightsquigarrow$ Grad wächst um 1.
- Falls neuer Grad $\leq b$: ok.
- Sonst: Spalte Knoten auf in zwei Knoten.
  $\rightsquigarrow$ Grad des darüberliegenden Knoten wächst um 1.
  $\rightsquigarrow$ Gegebenenfalls: zur Wurzel hin propagieren.
- Falls der Wurzelknoten selbst zu groß wird:
  Spalte Wurzelknoten auf; führe neuen Wurzelknoten darüber ein.
  (Baum wächst nach oben!)

Balancierte Bäume: (a, b)-Bäume

Löschen in (a, b)-Bäumen:

- Ähnlich wie Einfügen.
- Falls Grad zu klein wird: Knoten verschmelzen.

Erweiterbare Arrays

Erweiterbare Arrays

Erweiterbare Arrays (in Java: `Vector`):

Eigenschaften:

Zugriff auf Daten über Index (wie bei normalen Arrays),

aber unbegrenzt:
es können jederzeit weitere Elemente am Ende angehängt werden.

Erweiterbare Arrays

Idee: Objekte der enthalten Referenz auf ein normales Array.

```
public class EArray {
  Object[] array;
  int fuellstand;
}
```

Methoden:

`Object elementAt(int n)`:
liefert `array[n]`.

`void setElementAt(Object o, int n)`:
setzt `array[n] = o`.

Erweiterbare Arrays

Methoden:

`void addElement(Object o)`:
inkrementiere `fuellstand`;
falls kein Platz mehr in `array` ist:
`neu = new Object[m]; // m > array.length`
kopiere die Werte aus `array` nach `neu`,
setze `array = neu`.
setze `array[fuellstand] = o`.

Erweiterbare Arrays

Wie groß sollte das neue Array gewählt werden,
wenn das alte zu klein ist?

`m = array.length + const`
(Arraygrößen wachsen linear):

k Aufrufe von `addElement` führen zu $O(k^2)$ Kopieroperationen.

`m = array.length * const`
(Arraygrößen wachsen exponentiell):

k Aufrufe von `addElement` führen zu $O(k)$ Kopieroperationen.

Erweiterbare Arrays

Wenn man das neue Array jeweils doppelt (oder c-mal mit $c > 1$) so groß wählt wie das alte, dann gilt:

Ein Aufruf von `addElement` benötigt bestenfalls $O(1)$ Zeit.

Ein Aufruf von `addElement` benötigt schlimmstenfalls $O(n)$ Zeit, wobei n der momentane Füllstand ist.

Bei k Aufrufen benötigt im Durchschnitt („amortisiert“) jeder Aufruf $O(1)$ Zeit (garantiert!).