Berechnung von Hashwerten

Für Grunddatentypen ergibt sich der Hashwert gemäß einer Konvention, etwa Unicode oder ASCII bei `char`.

Für ein Objekt mit den Komponenten $a_0, \ldots, a_{n-1}$ (etwa Character einer Zeichenkette oder Instanzvariablen eines Objekts) ergibt sich der Hashwert zu

$$h = m^{n-1}h_0 + m^{n-2}mh_1 + \cdots + mh_{n-2} + h_{n-1}$$

Oder mit Hornerschema:

$$h = (m(\ldots(m(mh_0 + h_1) + h_2) + \ldots h_{n-2}) + h_{n-2}$$

m ist der Hashmultiplikator. Er kann unterschiedlich gewählt werden:

Java verwendet bei Zeichenketten $m = 31$.

Cormen et al (Standardwerk über Algorithmik) empfiehlt bei Zeichenketten $m = 256$.

Horstmann verwendet für ein selbstdefiniertes Objekt $m = 29$.

Berechnung von Hashwerten

NB: Überschreibt man die `equals`-Methode, so sollte man nicht vergessen, auch `hashCode` umzudefinieren, da sonst gleiche Objekte verschiedene Hashwerte haben könnten.

Binärbäume

Sind die Elemente einer Menge / Schlüssel einer Abbildung angeordnet (in Java `Comparable`, `Comparator`), so können zur Realisierung der Datenstruktur Binäre Suchbäume (BST) verwendet werden.

Erinnerung: Binärbäume in OCAML:

```
type 'a bintree = Empty |
         Build of 'a * 'a bintree * 'a bintree
```

Binärbäume in Java

```
abstract class Tree { . . . }

class Empty extends Tree {Empty() {} . . . }

class Build extends Tree {
    private Object data;
    private Tree left;
    private Tree right;

    Build(Object data, Tree left, Tree right){
        this.data = data;this.left = left;this.right=right;
    } . . .}
```

Dies verallgemeinert sich in natürlicher Weise auf andere rekursive Datentypen: eine Unterklasse für jeden Konstruktor (Konstruktor hier im Sinne von OCAML!). Eine Instanzvariable für jedes Argument eines solchen Konstruktors.

Terminologie

Der Baum `Empty` heißt *leer*.

Ein Baum der nichtleer ist, also ein `Build`, hat eine Wurzelbeschriftung (`data`), einen linken Teilbaum (`left`) und einen rechten Teilbaum (`right`).

Die `data`-Felder aller von der Wurzel aus erreichbaren `Build`-Objekten heißen *Knoten*.

Binärer Suchbaum

Definition: Sei t binärer Baum mit angeordneten Knoten.

Der Baum t heißt *binärer Suchbaum* (*binary search tree, BST*), wenn t leer ist, oder nichtleer mit Wurzelbeschriftung a, linkem Teilbaum l und rechtem Teilbaum r und

- Für jeden Knoten x von l gilt $x < a$, (in manchen Anwendungen auch $x \leq a$).
- Für jeden Knoten y von r gilt $a < y$.
- l und r sind selbst wiederum binäre Suchbäume.

Suchen in binärem Suchbaum

```
let rec enthaltenBST(x,t) = match t with
        Empty -> false
|       Build(a,l,r) -> x=a ||
                  (if x < a then enthaltenBST(x,l)
                            else enthaltenBST(x,r))
```

Wie implementiert man `enthaltenBST` in Java

Es gibt drei verschiedene Möglichkeiten, solch einen rekursiven Algorithmus objektorientiert zu implementieren.

1. Man sieht `enthaltenBST` als abstrakte Methode in `Tree` vor und implementiert sie in `Empty` und `Build` separat, ggf. durch rekursiven Aufruf.
2. Man verwendet `instanceof` zur Unterscheidung der Fälle.
3. Man verwendet das *Visitor pattern*.

Lösung mit abstrakter Methode

In `Tree`:

```
public abstract Boolean enthaltenBST1(Comparable x);
```

In `Empty`:

```
public Boolean enthaltenBST1(Comparable x) {
    return Boolean.FALSE;}
```

In `Build`:

```
public Boolean enthaltenBST1(Comparable x) {
    int cmp = x.compareTo(data);
    if (cmp == 0) return Boolean.TRUE;
    else if (cmp < 0) return left.enthaltenBST1(x);
    else return right.enthaltenBST1(x);}
```

Vorteil: einfach und “objektorientiert”

Nachteil: Teile der Definition von `enthaltenBST1` sind auf mehrere Klassen verteilt. Klassen werden immer größer je mehr Funktionen es gibt.

Lösung mit `instanceof`

In `Tree`:

```
public Boolean enthaltenBST2(Comparable x) {
    if (this instanceof Empty)
        return Boolean.FALSE;
    else /* this instanceof Build */ {
        Build node = (Build)this;
        int cmp = x.compareTo(node.data);
        if (cmp == 0) return Boolean.TRUE;
        else if (cmp < 0) return node.left.enthaltenBST2(x);
        else return node.right.enthaltenBST2(x);
    }
}
```

Nachteile: Nicht “objektorientiert”. Es wird nicht vom Compiler geprüft, ob alle Fälle abgedeckt sind.

Lösung mit Visitor Pattern

Idee: das Implementierungsmuster von `enthaltenBST1` wird nur ein einziges Mal verwendet, um eine Methode

```
Object accept(TreeVisitor v);
```

welche einen “Besucher” auf die Instanzvariablen des jeweiligen Objekts anwendet, zu implementieren.

Der “Besucher” stellt für jede Unterklasse (hier `Empty` und `Build`) jeweils eine “Besuchsmethode” bereit.

Für jede gewünschte Funktion (enthaltenBST, toString, insert, . . .) ist jeweils ein eigener “Besucher” zu implementieren, der die erforderlichen Besuchsmethoden entsprechend implementiert.

Konkret

```
interface TreeVisitor {
    Object visitEmpty();
    Object visitBuild(Object data,
                      Tree left,
                      Tree right);
}
```

In `Tree`:

```
    public abstract Object accept(TreeVisitor v);
```

In `Empty`:

```
public Object accept(TreeVisitor v) {return v.visitEmpty();}
```

In `Build`:

```
public Object accept(TreeVisitor v) {
        return v.visitBuild(data, left, right);}
```

`enthaltenBST` mit Visitor

```
class EnthaltenVisitor implements TreeVisitor {
    Comparable x;
    EnthaltenVisitor(Comparable x) {this.x = x;}

    public Object visitEmpty() {return Boolean.FALSE;}
    public Object visitBuild(Object data, Tree left, Tree right) {
        int cmp = x.compareTo(data);
        if (cmp == 0)
            return Boolean.TRUE;
        else if (cmp < 0)
            return (Boolean)left.accept(this);
        else
            return (Boolean)right.accept(this);}}
```

In `Tree` (oder sonstwo):

```
public static Boolean enthaltenBST3(Tree t, Comparable x) {
   return (Boolean)t.accept(new EnthaltenVisitor(x));}
```

Ein Besucher für `toString`

```
class ToStringVisitor implements TreeVisitor {

    public Object visitEmpty() {return "L()";}

    public Object visitBuild(Object data, Tree left, Tree right) {
        return "N(" + data + ", " +
            (String)left.accept(this) + ", " +
            (String)right.accept(this) + ")";}}
```

Konvertieren eines Baumes `t`:

```
(String)t.accept(new ToStringVisitor())
```

Die Typecasts (nach `String` bzw. `Boolean`) sind unschön. Alternativ kann man das Ergebnis zu einer Instanzvariablen des Visitors machen und nach "Besuch" abrufen. Am besten wartet man auf Java 1.5. und verwendet polymorphe Typvariablen.

Einfügen in BST in OCAML

```
let rec insert x t = match t with
  Empty -> Build(x,Empty,Empty)
| Build(a,l,r) -> if x<=a then Build(a,insert x l,r)
                          else Build(a,l,insert x r)
```

Einfügen in BST in Java

```
class InsertVisitor implements TreeVisitor {
    Comparable x;
    InsertVisitor(Comparable x) {this.x = x;}

    public Object visitEmpty(){
        return new Build(x, new Empty(), new Empty());
    }

    public Object visitBuild(Object data, Tree left, Tree right) {
        int cmp = x.compareTo(data);
        if (cmp == 0)
          return new Build(x, left, right);
        else if (cmp < 0)
          return new Build(data,(Tree)left.accept(this),right);
        else
          return new Build(data,left,(Tree)right.accept(this));
    }}
```

Imperatives Einfügen

Will man den neuen Knoten imperativ in den vorhandenen Baum einhängen, so benötigt man eine verallgemeinerte Besucher-Schnittstelle:

```
interface TreeVisitorI {
    Object visitEmpty(Empty it);
    Object visitBuild(Build it,Object data,Tree left,Tree right);}
```

und entsprechende `accept`-Methoden (overloading!):

In `Empty`:

```
    public Object accept(TreeVisitorI v) {return v.visitEmpty(th
```

In `Build`:

```
    public Object accept(TreeVisitorI v) {
        return v.visitBuild(this, data, left, right);}
```

Der Besucher bekommt also jeweils den besuchten Knoten selbst zu sehen und nicht nur seine Instanzvariablen. Dadurch kann er den besuchten Knoten imperativ verändern (z.B.: durch Aufrufen von set-Methoden).

Imperatives Einfügen

```
class InsertVisitorI implements TreeVisitorI {
    Comparable x;
    InsertVisitorI(Comparable x) {this.x = x;}

    public Object visitEmpty(Empty it)
             {return new Build(x, new Empty(), new Empty());}
    public Object visitBuild(Build it, Object data, Tree left,
        int cmp = x.compareTo(data);              [   Tree right) {
        if (cmp == 0)                              ^ Zeile zu kurz
            {it.setData(x);}
        else if (cmp < 0)
            it.setLeft((Tree)left.accept(this));
        else
            it.setRight((Tree)right.accept(this));
        return it;}}
```

Die Definition der set-Methoden in `Build` ist zu ergänzen.

Verwendung

Ist `t` ein `Tree` und soll `x` eingefügt werden, so schreibt man

```
Tree t_neu = t.accept(new InsertVisitorI(x));
```

Dann enthält `t_neu` den Baum `t` nachdem `x` eingefügt wurde. Den Baum `t` selbst darf man nach dem Aufruf von `t.accept(new InsertVisitorI(x));` nicht mehr verwenden. Er enthält nämlich ein unspezifiziertes Zwischenergebnis.

Bei der *funktionalen* Version `InsertVisitor` bleibt `t` unverändert.

Wenn man sich daran stört, so kann man eine Wrapperklasse analog zu `LinkedList` schreiben. Dies erlaubt eine Formulierung von `Insert` ohne Rückgabewert.

Entfernen aus BST

```
let rec remove x t = match t with
   Empty -> Empty
|  Build(a,l,r) -> if x<a then Build(a,remove x l,r)
                   else if x>a then Build(a,l,remove x r)
                   else if l=Empty then r
                   else if r=Empty then l
                   else let m,l' = remove_max l in
                   Build(m,l',r)
```

Hilfsfunktion `remove_max`

```
let rec remove_max t = match t with
  Build(a,l,Empty) -> a,l
| Build(a,l,r) -> let m,r' = remove_max r in
                  (m,Build(a,l,r'))
```

Als imperativer Besucher: RemoveMax

```
class RemoveMaxVisitor implements TreeVisitorI {
    Comparable themax;
    RemoveMaxVisitor() {themax=null;}
    public Object getResult() {return themax;}
    public Object visitEmpty(Empty it){return it;}
    public Object visitBuild(Build it, Object data, Tree left,
        if (right instanceof Empty) {                [ Tree right){
            themax = (Comparable)data;
            return left;
        } else {
            it.setRight((Tree)right.accept(this));
            return it;
        }}}
```

Als imperativer Besucher: Remove

```
class RemoveVisitor implements TreeVisitorI {
    Comparable x;
    RemoveVisitor(Comparable x) {this.x = x;}
    public Object visitEmpty(Empty it){return it;}
    public Object visitBuild(Build it, Object data, Tree left,
        int cmp = x.compareTo(data);                [ Tree right){
        if (cmp < 0) {
            it.setLeft((Tree)left.accept(this));
            return it;
        } else if (cmp > 0) {
            it.setRight((Tree)right.accept(this));
            return it;
        } else /* cmp == 0 */  . . .
```

Als imperativer Besucher: Remove, Forts.

```
. . . /* cmp == 0 */
if (left instanceof Empty) {
    return right;
} else if (right instanceof Empty) {
    return left;
} else {
    RemoveMaxVisitor v = new RemoveMaxVisitor();
    Tree left1 = (Tree)left.accept(v);
    it.setLeft((Tree)left.accept(v));
    it.setData(v.getResult());
    return it;}}
```

NB: Für die Verwendung gilt sinngemäß das auf Folie 346 Gesagte.

Balancierung

Die Laufzeit der definierten Operationen ist jeweils proportional zur *Höhe* des Baumes (längster Pfad von Wurzel zu einem Blatt).

Fügt man in BST Elemente in sortierter (auf- oder abwärts) Reihenfolge ein, so entartet der Baum zur Liste: die Höhe ist proportional zur Anzahl der Knoten.

Ein Baum ist ausgeglichen, wenn die Höhe proportional[a]

zum *Logarithmus* der Höhe ist. Dies ist der Fall, wenn linker und rechter Teilbaum *jeweils* in etwa gleich groß sind.

Unser Ziel ist es, durch Rebalancierungsoperationen die Ausgeglichenheit sicherzustellen unabhängig von der Reihenfolge der Einfügeoperationen.

---

[a] Man muss das geeignet formalisieren, sonst ergeben sich Fragen wie “Ist 11 proportional zu 13?”. Stichwort: Ausgeglichenheit ist eine Eigenschaft einer (i.a. unendlichen) Menge von BST.

Prä-AVL-Baum

Zu diesem Zweck führen wir in jedem Knoten noch einen Integerwert mit, den *Balancefaktor, BF*, d.h. wir betrachten Bäume mit Einträgen aus

```
class AVL_Entry implements Comparable {
    public Comparable data;
    public int bf;
    public int getBF(){return bf;}
    public void setBF(int x){bf=x;}
    public int compareTo(Object x) {return data.compareTo(x);}}
```

Ein solcher Baum t ist *Prä-AVL-Baum*, wenn gilt:

- t ist binärer Suchbaum,
- t ist `Empty`, oder
- t ist `Build` mit linkem Teilbaum l, rechtem Teilbaum r, Beschriftung d und $\mathit{Höhe}(r) - \mathit{Höhe}(l) = d$ `.getBF()`.

Der Balancefaktor gibt also die Höhendifferenz der beiden Teilbäume an.

AVL-Baum

Ein Prä-AVL-Baum ist AVL-Baum (nach den Erfindern Adelson-Vel'skij und Landis), wenn alle Balancefaktoren $-1, 0,$ der 1 sind.

Sei $h(n)$ die maximale Höhe eines AVL-Baumes mit n Knoten. Sei $B(h)$ die minimale Zahl von Knoten eines AVL-Baums der Höhe h. Es ist $h(n) = B^{-1}(n)$ und es gilt:

$B(0) = 0$ (Höhe 0 bedeutet: genau ein leerer Teilbaum; keine Knoten.)

$B(1) = 1$ (Höhe 1, genau ein Knoten.)

$$B(h) = 1 + \min(\underbrace{B(h-1) + B(h-2)}_{\text{BF}=1,-1}, \underbrace{2 \cdot B(h-1)}_{\text{BF}=0}), \text{ wenn } h > 1$$

Nachdem $B(h)$ monoton ist, gilt sogar $B(h) = 1 + B(h-1) + B(h-2)$.

Sei F_h die h-te *Fibonaccizahl*. Durch Einsetzen bestätigt man $B(h) = F_{h+1} - 1$. Es folgt $h(n) = O(\log(n))$.

AVL-Bäume sind also gut ausgeglichen.

Rotation

Durch naives Einfügen oder Löschen kann die AVL-Eigenschaft verlorengehen. Zur Wiederherstellung hat man die *Rotationen* zur Verfügung.

Sei t ein Prä-AVL-Baum mit Wurzel-BF 2 und AVL Teilbäumen. Sei weiter der BF des rechten Teilbaums 1 oder 0. Dann ist `rotate_left`(t) ein AVL-Baum.

```
let rotate_left = function
        Build((2,a),u,Build((1,b),v,w)) ->
             (-1, Build((0,b),Build((0,a),u,v),w))
  |     Build((2,a),u,Build((0,b),v,w)) ->
             (0,Build((-1,b),Build((1,a),u,v),w))
```

Wir schreiben `Build((bf,x),l,r)` für den Baum mit BF `bf`, Beschriftung `x`, linkem Teilbaum `l`, rechtem Teilbaum `r`.

Rechtsrotation

Sei t ein Prä-AVL-Baum mit Wurzel-BF -2 und AVL Teilbäumen. Sei weiter der BF des linken Teilbaums -1 oder 0. Dann ist `rotate_right`(t) ein AVL-Baum.

```
let rotate_right = function
        Build((-2,a),Build((-1,b),u,v),w) ->
                (-1, Build((0,b),u,Build((0,a),v,w)))
  |     Build((-2,a),Build((0,b),u,v),w) ->
                (0, Build((1,b),u,Build((-1,a),v,w)))
```

Doppel-Links-Rotation

Sei t ein Prä-AVL-Baum mit Wurzel-BF 2 und AVL Teilbäumen. Sei weiter der BF des rechten Teilbaums -1. Dann ist `rotate_double_left`(t) ein AVL-Baum.

```
let rotate_double_left = function
  | Build((2,a),t1,Build((-1,s),Build((1,b),t2,t3),t4)) ->
      (-1,Build((0,b),Build((-1,a),t1,t2),Build((0,s),t3,t4)))
  | Build((2,a),t1,Build((-1,s),Build((0,b),t2,t3),t4)) ->
      (-1, Build((0,b),Build((0,a),t1,t2),Build((0,s),t3,t4)))
  | Build((2,a),t1,Build((-1,s),Build((-1,b),t2,t3),t4)) ->
      (-1,Build((0,b),Build((0,a),t1,t2),Build((1,s),t3,t4)))
```

Doppel-Rechts-Rotation

Sei t ein Prä-AVL-Baum mit Wurzel-BF -2 und AVL Teilbäumen. Sei weiter der BF des linken Teilbaums 1. Dann ist `rotate_double_left`(t) ein AVL-Baum.

```
let rotate_double_right = function
 Build((-2,a),Build((1,s),t1,Build((-1,b),t2,t3)),t4) ->
    (-1, Build((0,b),Build((0,s),t1,t2),Build((1,a),t3,t4)))
|Build((-2,a),Build((1,s),t1,Build((0,b),t2,t3)),t4) ->
    (-1, Build((0,b),Build((0,s),t1,t2),Build((0,a),t3,t4)))
|Build((-2,a),Build((1,s),t1,Build((1,b),t2,t3)),t4) ->
    (-1, Build((0,b),Build((-1,s),t1,t2),Build((0,a),t3,t4)))
```

Ausgleichen von Prä-AVL-Bäumen

Sei $t = \texttt{Build}((b, x), l, r)$ ein Prä-AVL-Baum dessen Wurzel BF $b = 2$ oder $b = -2$ besitzt und dessen linker und rechter Teilbaum l bzw. r AVL-Bäume sind.

Wir behaupten: durch wenige Rotationen lässt sich t in einen AVL-Baum überführen.

Es sind folgende Fälle zu betrachten:

Fälle

- $t = \texttt{Build}((2, x), l, \texttt{Build}((1, y), m, r))$. Durch Linksrotation wird dies in $\texttt{Build}((0, y), \texttt{Build}((0, x), l, m), r)$ überführt.
- $t = \texttt{Build}((2, x), l, \texttt{Build}((0, y), m, r))$. Durch Linksrotation wird dies in $\texttt{Build}((-1, y), \texttt{Build}((1, x), l, m), r)$ überführt.
- $t = \texttt{Build}((2, x), t_1, \texttt{Build}((-1, y), \texttt{Build}((-1, z), t_2, t_3), t_4))$. Durch Doppelrotation wird dies in $\texttt{Build}((0, z), \texttt{Build}((0, x), t_1, t_2), \texttt{Build}((1, y), t_3, t_4))$ überführt.
- $t = \texttt{Build}((2, x), t_1, \texttt{Build}((-1, y), \texttt{Build}((1, z), t_2, t_3), t_4))$. wird zu $\texttt{Build}((0, z), \texttt{Build}((-1, x), t_1, t_2), \texttt{Build}((0, y), t_3, t_4))$
- $t = \texttt{Build}((2, x), t_1, \texttt{Build}((-1, y), \texttt{Build}((0, z), t_2, t_3), t_4))$. wird zu $\texttt{Build}((0, z), \texttt{Build}((0, x), t_1, t_2), \texttt{Build}((0, y), t_3, t_4))$

Die Höhe reduziert sich dabei um 1 in allen außer dem zweiten Fall. Der Fall BF=-2 ist symmetrisch.

Allgemeine Rotation

Wir sind nun in der Lage, eine Funktion

```
rotate : (int*'a) bintree -> int * (int*'a) bintree
```

zu schreiben, die diese Rotationen falls nötig vornimmt, d.h. `rotate` t ist für AVL-Bäume und Prä-AVL-Bäume, die höchstens in der Wurzel BF 2 oder -2 haben, definiert. Für das Ergebnis (m, t') gilt:

- t' ist AVL-Baum
- t' entsteht aus t durch höchstens zwei Rotationen (eine einfache oder eine Doppel Rotation)
- $m = \mathsf{hoehe}(t') - \mathsf{hoehe}(t)$ wobei $m \in \{0, -1\}$.

Einfügen in AVL-Baum

Wir schreiben eine Funktion

```
insert_avl : 'a -> (int*'a) bintree -> int * (int*'a) bintree
```

so, dass `insert_avl` $x\ t\ = m, t'$ impliziert

- Falls t AVL-Baum ist, so auch t'
- Falls t AVL-Baum ist, so entsteht t' aus t durch Einfügen von x an geeigneter Stelle und Rotationen
- $m = \mathsf{hoehe}(t') - \mathsf{hoehe}(t)$, hierbei $m \in \{0, 1\}$.

Implementierung

Es sei x in t einzufügen.

Ist t leer, so ist das Ergebnis $(1, \texttt{Build}((0, x), \texttt{Empty}, \texttt{Empty}))$.

Jetzt sei $t = \texttt{Build}((b, y), l, r)$ und (o.B.d.A.) $x \geq y$. Wir bestimmen rekursiv

$$(m_1, r') = \texttt{insert\_avl}\ x\ r$$

und bilden

$$t' := \texttt{Build}((b + m_1, y), l, r')$$

Der Baum t' enthält die Knoten von t, sowie x und ist ein Prä-AVL-Baum, dessen Kinder sogar AVL-Bäume sind.

Durch `rotate` können wir daraus das gesuchte Ergebnis machen:

$$(m_2, t'') = \texttt{rotate}\ t'$$

Implementierung

Für die auch verlangte Höhendifferenz setzen wir:

$$m_3 := \texttt{if}\ \ m_1 = 1 \wedge (b = 0 \vee (b = 1 \wedge m_2 = 0))\ \ \texttt{then}\ \ 1\ \ \texttt{else}\ \ 0$$

Wir setzen jetzt $\texttt{insert\_avl}\ x\ t = (m_3, t'')$. Man beachte, dass bei $b = -1$ die Höhe nicht verändert wird und bei $b = 1$ rotiert wird (beides nur, wenn $m_1 = 1$).

Entfernen des größten Eintrags

Wir schreiben eine Funktion

```
remove_max_avl : (int*'a) bintree -> int * 'a  * (int*'a) bintree
```

so, dass `remove_max_avl` $t = m, y, t'$ impliziert

- Falls t AVL-Baum ist, so auch t'
- Falls t AVL-Baum ist, so entsteht t' aus t durch Entfernen des größten Eintrags y (steht ganz rechts unten).
- $m = \mathsf{hoehe}(t') - \mathsf{hoehe}(t)$, hierbei $m \in \{0, -1\}$.

Implementierung

Wir wollen `remove_max_avl` t bestimmen.

Ist t leer, so ist das Ergebnis undefiniert (*exception*, o.ä).

Ist $t = \texttt{Build}((b, x), l, \texttt{Empty})$, so ist das Ergebnis $(-1, x, l)$.

Jetzt sei $t = \texttt{Build}((b, y), l, r)$ und $r \neq \texttt{Empty}$. Wir bestimmen rekursiv

$$(m_1, y, r') = \texttt{remove\_max\_avl}\ r$$

Beachte: $m_1 \leq 0$. Wir bilden

$$t' := \texttt{Build}((b + m_1, x), l, r')$$

Durch `rotate` können wir daraus das gesuchte Ergebnis machen:

$$(m_2, t'') = \texttt{rotate}\ t'$$

Für die auch verlangte Höhendifferenz setzen wir:

$$m_3 := \texttt{if}\ \ m_1 = -1 \wedge (b = 1 \vee b = -1 \wedge m_2 = -1)\ \ \texttt{then}\ \ -1\ \ \texttt{else}\ \ 0$$

Implementierung

Wir setzen jetzt `remove_max_avl` $x\ t = (m_3, y, t'')$.

Man beachte: bei $b = 1$ wird nicht rotiert, aber die Höhe verringert sich. Bei $b = -1$ wird rotiert und die Höhe reduziert sich auch.

Löschen

Wir schreiben eine Funktion

```
delete_avl : 'a -> (int*'a) bintree -> int * (int*'a) bintree
```

so, dass `delete_avl` $x\ t\ = m, t'$ impliziert

- Falls t AVL-Baum ist, so auch t'
- Falls t AVL-Baum ist, so entsteht t' aus t durch Entfernen des Eintrags x (falls vorhanden).
- $m = \mathsf{hoehe}(t') - \mathsf{hoehe}(t)$

Die Implementierung erfolgt analog zu `remove_max`.

Die, möglicherweise imperative, Implementierung aller Operationen auf AVL-Bäumen in Java verbleibt als Übung.

Implementierung von `Set` und `Map`

Die Schnittstellen `Set` und `Map` können nun durch AVL-Bäume effizient implementiert werden.

Bei `Map` sind die Beschriftungen Schlüssel-Wert Paare. Für die Ordnung und insbesondere die Gleichheit ist aber nur der Schlüssel maßgeblich:

```
class MapEntry implements Comparable {
    Comparable key;
    Object value;
    public int compareTo(Object x) {
        return key.compareTo(((MapEntry)x).key);
    }
}
```

Zusammenfassung

- Die Schnittstellen `Set` und `Map` deklarieren Methoden zur Verwaltung von endlichen Mengen und Abbildungen.
- Diese Schnittstellen lassen sich sowohl durch Hashtabellen, als auch durch binäre Suchbäume implementieren.
- AVL-Bäume sind besondere binäre Suchbäume, die der zusätzlichen Invariante $\mathit{Höhe} = O(\log(\mathit{Knotenzahl}))$ genügen. Dadurch laufen die Operationen Suchen, Einfügen, Löschen in logarithmischer Zeit.
- Hashtabellen haben sehr gute mittlere Laufzeit, die Laufzeit hängt aber entscheidend von der Kollisionshäufigkeit ab und kann im ungünstigen Fall linear (in der Zahl der Einträge) werden.
- Binäre Suchbäume erlauben die Ausgabe der Einträge in sortierter Reihenfolge (Ausgeben der Beschriftungen in symmetrischer Ordnung)
- Das Visitor Pattern erlaubt die objektorientierte Implementierung rekursiver Operationen auf baumartigen Datenstrukturen.

(a, b)-Bäume

Interner Suchbaum: Information steht in den Knoten.

Externer Suchbaum: Information steht nur in den Blättern; Knoten tragen Verwaltungsinformation.

Ein (a, b)-Baum, wobei $b \geq 2a - 1$, ist ein externer Suchbaum, dessen interne Knoten außer der Wurzel einen Rang (Kinderzahl) zwischen a und b (einschließlich) haben.

Die Wurzel hat mindestens 2 und höchstens b Kinder.

Jeder interne Knoten vom Rang N enthält $N - 1$ aufsteigend geordnete Zahlen $k_1, \ldots, k_{N-1}$ als Verwaltungsinformation: Beim Suchen vergleicht man den Schlüssel k mit diesen k_i. Es gilt: ist k überhaupt unterhalb des Knotens zu finden, so unterhalb des i-ten Kindes, falls $k_{i-1} \leq k < k_i$.

Beim Einfügen und Entfernen von Einträgen muss man all diesen Bedingungen Rechnung tragen.

Einfügen in (a, b)-Baum

Bestimme durch Suche die Blattposition, an die der neue Eintrag gehört.

Falls noch nicht zuviele Blätter (also $< b$) vorhanden, erzeuge ein neues und aktualisiere die Verwaltungsinformation.

Andernfalls teile den Knoten in zwei (fast) gleichgroße auf. (Die Bedingung $b \geq 2a - 1$ wird hier gebraucht.)

Führt dies zu Überlauf im Elternknoten, so teile man diesen ebenso auf, etc. bis ggf. zur Wurzel.

Das Löschen erfolgt analog durch Verschmelzen.

B-Bäume

Ein B-Baum ist ein (a, b) Baum mit $b = 2a - 1$.

In den Anwendungen ist a dann relativ groß, z.B. 512. Dadurch reduziert sich die Höhe enorm gegenüber einem Binärbaum.

Das ist sinnvoll, wenn soviele Daten zu verwalten sind, dass die Knoten nicht im Hauptspeicher Platz finden, sondern auf Festplatten gespeichert werden. Ein Plattenzugriff dauert sehr lange (bis zu 10000 Mal solange wie ein Hauptspeicherzugriff) kann aber gleich eine ganze "Seite" auslesen, also z.B.: einen ganzen Knoten eines B-Baumes.

Interessant ist aber auch der Spezialfall $a = 2$. Hier ist $b = 3$ und die Knoten haben 2 oder 3 Kinder. Solche Bäume kann man wiederum als Binärbäume codieren und erhält so die *Rot-Schwarz-Bäume*, siehe Eff. Alg.