Imperative Programmierung (IPR)

Kapitel 7: Bäume

Univ.-Prof. Dr.-Ing. habil. Gero Mühl

Lehrstuhl für Architektur von Anwendungssystemen (AVA)
Fakultät für Informatik und Elektrotechnik (IEF)
Universität Rostock

Universität Rostock

Traditio et Innovatio

Inhalte

1. Grundlagen
2. Spezifikation
3. Implementierung

Kapitel 7.1

Grundlagen

Grundlegende Definitionen

- Ein Baum ist ein grundlegender **hierarchischer Datentyp** mit großer Bedeutung für die Informatik.

Definition 1 (Baum)

Ein **Baum** *t ist entweder leer, d. h. $t = \epsilon$, oder er besteht aus einem* **Wurzelknoten** *w und einer geordneten, nicht-leeren Menge von* **Teilbäumen** *$(t_1, \ldots, t_n)$, die wieder Bäume sind: $t = (w, (t_1, \ldots, t_n))$.*

- Aufgrund der rekursiven Definition von Bäumen ist jeder Knoten eines Baums selbst wieder die Wurzel eines Baumes.

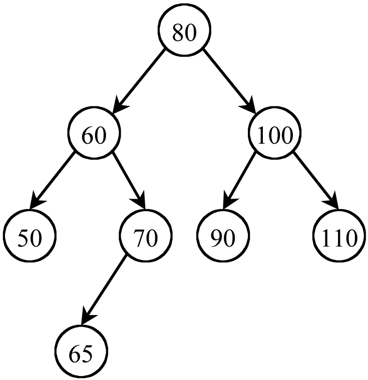
Beispiel 1 (Bäume)

- Der Baum $(0, (\epsilon, \epsilon))$ besteht aus dem einzelnen Knoten 0.
- Der Baum $(1, (2, (\epsilon, \epsilon)), (3, (\epsilon, \epsilon)))$ besteht aus dem Wurzelknoten 1 und zwei Teilbäume mit den Wurzeln 2 bzw. 3.

Beispiel Baum

- Der rechts abgebildete Baum hat die Form:

$$
\begin{array}{l}
(80, (\\
\quad (60, (\\
\quad\quad (50, (\epsilon, \epsilon)), \\
\quad\quad (70, (\\
\quad\quad\quad (65, (\epsilon, \epsilon)), \\
\quad\quad\quad \epsilon \\
\quad\quad)) \\
\quad)), \\
\quad (100, (\\
\quad\quad (90, (\epsilon, \epsilon)), \\
\quad\quad (110, (\epsilon, \epsilon)) \\
\quad)) \\
))
\end{array}
$$



Eltern/Kind-Beziehungen zwischen Knoten

Definition 2 (Kind- und Elternknoten)

Sei $t = (w, (t_1, \ldots, t_n))$ ein nicht-leerer Baum und $t_i = (c_i, (t_{i1}, \ldots, t_{ij}))$ ein nicht-leerer Teilbaum von t, dann heißt c_i **Kindknoten** *von w bzw. umgekehrt w* **Elternknoten** *von c_i. Ist ein Teilbaum von t leer, so* **fehlt** *das entsprechende Kind.*

- Ein Baum besteht also aus einer Menge von Knoten mit hierarchischen **Eltern-/Kindbeziehungen** zwischen diesen.
- Von jedem Knoten aus gelangt man durch wiederholtes Aufsteigen zum jeweils nächsthöheren Elternknoten zum Wurzelknoten.
- Analog kann man von der Wurzel durch wiederholtes Absteigen (in den jeweils richtigen Teilbaum) zu jedem Knoten des Baums gelangen.

Blätter und innere Knoten eines Baums

Definition 3 (Blätter)

Die Knoten eines Baum ohne Kinderknoten heißen **Blätter***.*

- Während Blätter also keine Kinderknoten haben, hat die Wurzel eines Baumes keinen Elternknoten.

Definition 4 (Innere Knoten)

Die Knoten eines Baumes, die keine Blätter sind, heißen **innere Knoten** *des Baumes.*

- Die inneren Knoten eines Baumes haben also immer mindestens einen Kinderknoten.
- Dies gilt auch für die Wurzel eines Baumes, sofern der Baum aus mehr als einem Knoten besteht.

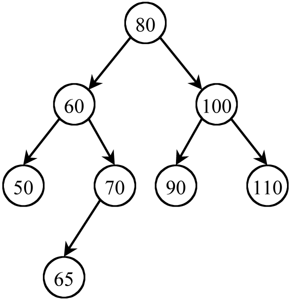
Höhe eines Baumes

Definition 5 (Höhe eines Baumes)

Die **Höhe** $h(t)$ *eines Baumes* t *beträgt:*

$$h(t) = \begin{cases} 0, & \text{falls } t = \epsilon \\ 1 + \max_{i=1}^{n} \left[h(t_i) \right], & \text{falls } t = (n, (t_1, \ldots, t_n)) \end{cases}$$

- Beispiel:
  Der rechts abgebildete Baum hat die Höhe 4.



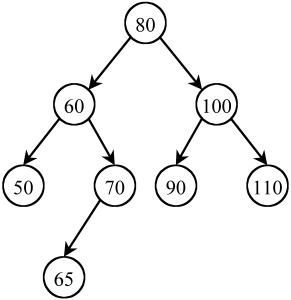
Minimale Höhe eines Baumes

Definition 6 (Minimale Höhe eines Baumes)

Die **minimale Höhe** $h_{min}(t)$ *eines Baumes* t *beträgt:*

$$h_{min}(t) = \begin{cases} 0, & \textit{falls } t = \epsilon \\ 1 + \min_{i=1}^{n} \left[h_{min}(t_i) \right], & \textit{falls } t = (n, (t_1, \ldots, t_n)) \end{cases}$$

- Beispiel:
  Der rechts abgebildete Baum hat die minimale Höhe 3.



Definition Binärbaum

Definition 7 (Binärer Baum oder auch Binärbaum)

Ein Baum ist ein **Binärbaum***, wenn er entweder leer ist oder er genau zwei Teilbäume hat und diese wieder Binärbäume sind.*

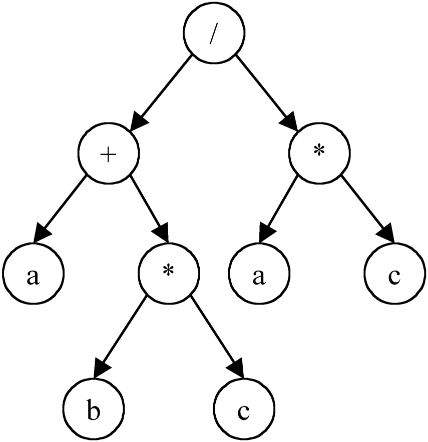
- In einem Binärbaum hat jeder Knoten höchstens zwei (existierende oder auch fehlende) Kinder.

Definition 8 (Linker und rechter Teilbaum eines Binärbaums)

In einem Binärbaum $t = (w, (t_1, t_2))$ *ist* t_1 *der* **linke Teilbaum** *und* t_2 *der* **rechte Teilbaum** *von* t*.*

Beispiel Binärbaum

- Binärbaum mit Wurzel „/“ und fünf Blättern.



Halbblätter und Teilendknoten eines Binärbaums

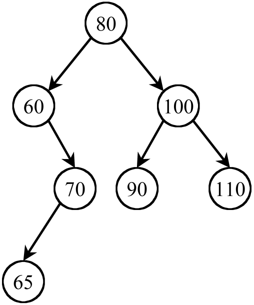
Definition 9 (Halbblätter eines Binärbaums)

In einem Binärbaum werden die Knoten mit genau einem Kindknoten **Halbblätter** *genannt.*

Definition 10 (Teilendknoten)

Die Blätter und Halbblätter eines Binärbaums werden zusammen auch als **Teilendknoten** *bezeichnet.*

- In dem rechts abgebildeten Binärbaum sind die Knoten 60 und 70 Halbblätter, während die Knoten 65, 90 und 110 Blätter sind.
- Diese fünf Knoten sind die Teilendknoten dieses Baumes.

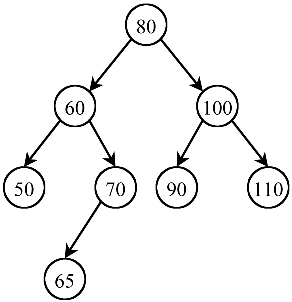


Voller Binärbaum

Definition 11 (Voller Binärbaum)

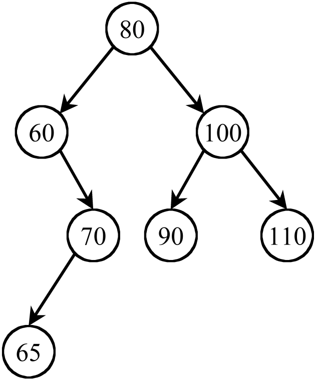
Ein Binärbaum t heißt **voll** *oder auch* **balanciert***, wenn für seine Höhe $h(t)$ und seine minimale Höhe $h_{min}(t)$ gilt: $h(t) - h_{min}(t) \leq 1$.*

- In einem vollen Binärbaum ist jede Ebene bis auf die unterste vollständig gefüllt.
- Beispiel: Der rechts abgebildete Binärbaum ist voll, da die Höhe des Baums 4 und die minimale Höhe des Baums 3 beträgt.
- Alle Ebenen, bis auf die vierte sind vollständig gefüllt.



Beispiel für nicht-vollen binären Baum

- Der unten abgebildete Binärbaum ist *nicht* voll, da die Höhe des Baums 4 und die minimale Höhe des Baums 2 ist.
- Die dritte Ebene (linkes Kind des Knotens 60 fehlt) und die vierte Ebene sind *nicht* vollständig gefüllt.



Vollständiger Binärbaum

Definition 12 (Vollständiger Binärbaum)

Ein Binärbaum ist **vollständig***, wenn seine Höhe und seine minimale Höhe übereinstimmen.*

- Offensichtlich ist jeder vollständige Binärbaum auch voll.
- Ein vollständiger Binärbaum der Höhe h hat:
  - $2^h - 1$ Knoten,
  - $2^{h-1} - 1$ innere Knoten (nicht Blatt, aber evtl. Wurzel),
  - 2^{h-1} Blätter und
  - 2^t Knoten in Tiefe t ($0 \leq t \leq h - 1$).

Beispiel 2 (Vollständiger Binärbaum)

- Ein vollständiger Binärbaum der Höhe 4, hat also 15 Knoten, von denen 7 innere Knoten und 8 Blätter sind.
- Auf den einzelnen Ebenen befinden sich 1, 2, 4 und 8 Knoten.

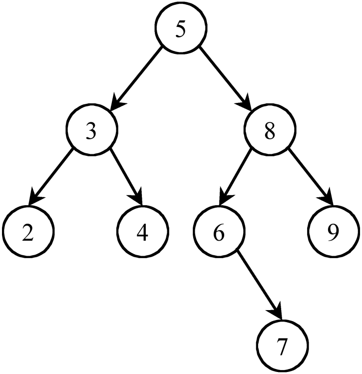
Traversieren eines Binärbaums: Hauptvarianten

- **Preorder-Traversierung**
  1. Verarbeitung der Wurzel
  2. Preorder-Traversierung des linken Teilbaums
  3. Preorder-Traversierung des rechten Teilbaums
- **Inorder-Traversierung**
  1. Inorder-Traversierung des linken Teilbaums
  2. Verarbeitung der Wurzel
  3. Inorder-Traversierung des rechten Teilbaums
- **Postorder-Traversierung**
  1. Postorder-Traversierung des linken Teilbaums
  2. Postorder-Traversierung des rechten Teilbaums
  3. Verarbeitung der Wurzel

Preorder-Traversierung eines Binärbaums

1. Verarbeitung der Wurzel
2. Preorder-Traversierung des linken Teilbaums
3. Preorder-Traversierung des rechten Teilbaums

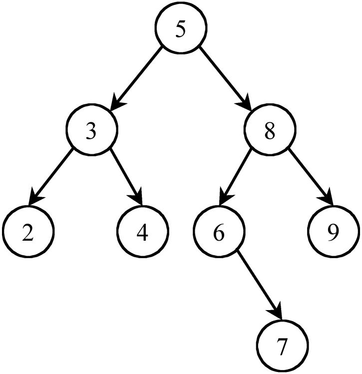
- Besuchsreihenfolge der Knoten: 5, 3, 2, 4, 8, 6, 7, 9



Inorder-Traversierung eines Binärbaums

1. Inorder-Traversierung des linken Teilbaums
2. Verarbeitung der Wurzel
3. Inorder-Traversierung des rechten Teilbaums

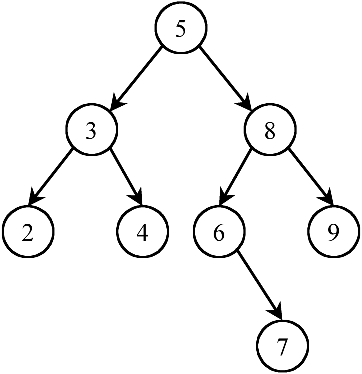
- Besuchsreihenfolge der Knoten: 2, 3, 4, 5, 6, 7, 8, 9



Postorder-Traversierung eines Binärbaums

1. Postorder-Traversierung des linken Teilbaums
2. Postorder-Traversierung des rechten Teilbaums
3. Verarbeitung der Wurzel

- Besuchsreihenfolge der Knoten: 2, 4, 3, 7, 6, 9, 8, 5



Traversieren eines Binärbaums

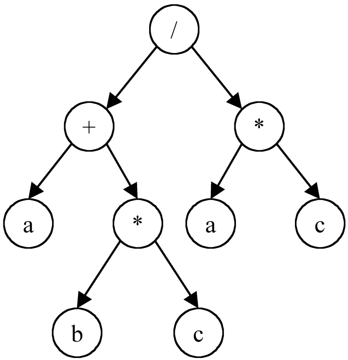
- Für Bäume arithmetischer Ausdrücke liefern die Besuchsfolgen die Präfix-, Infix- und Postfix-Schreibweise des Ausdrucks.

- Beispiel
  - Preoder-Traversierung liefert Präfix-Schreibweise:

    / + a * b c * a c
  - Inorder-Traversierung liefert Infix-Schreibweise:

    (a + b * c) / (a * c)
  - Postorder-Traversierung liefert Postfix-Schreibweise:

    a b c * + a c * /

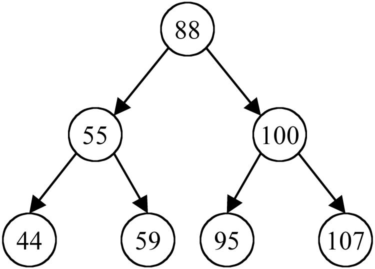


Binäre Suchbäume

Definition 13 (Binäre Suchbäume (engl. Binary Search Tree (BST)))

Ein **binärer Suchbaum** *ist ein Binärbaum, bei dem für jeden Knoten gilt, dass die Schlüssel aller Knoten im linken Teilbaum des Knotens kleiner und aller Knoten im rechten Teilbaum des Knotens größer als der Schlüssel des Knotens sind.*

- BSTs werden auch **sortierte Binärbäume** genannt.
- Die Inorder-Traversierung eines BSTs liefert die sortierte Folge seiner Schlüssel → **Tree-Sort**.
- Hier: [44, 55, 59, 88, 95, 100, 107]

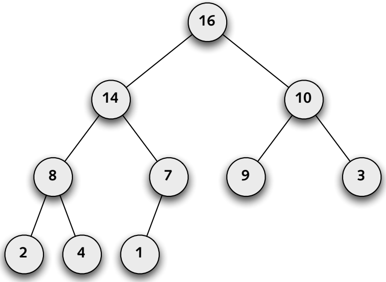


Heaps

Definition 14 (Halde (engl. Heap))

Eine **Halde** *ist ein voller Binärbaum, bei dem die Blätter linksbündig aufgefüllt sind. Zusätzlich muss der Schlüssel jedes Knotens größergleich (bzw. kleinergleich) den Schlüsseln seiner Kinderknoten sein* $\rightarrow$ **Max-Heap-Bedingung** *(bzw.* **Min-Heap-Bedingung***).*

- **Heap-Sort**
  - Fortgesetztes Entnehmen des kleinsten Elements aus einem Min-Heap liefert die aufsteigend sortierte Folge.
  - Analog führt die Verwendung eines Max-Heaps und das fortgesetzte Entnehmen des größten Elements auf die absteigend sortierten Folge.



Max-Heap

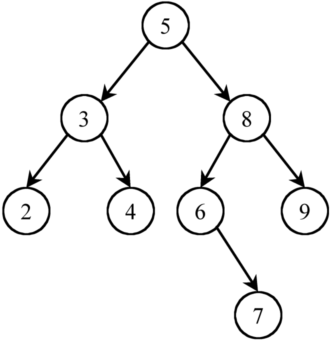
Kapitel 7.2

Spezifikation

Modellierung von Bäumen als Terme

- Wir beschränken uns im Folgenden auf binäre Bäume.
- Bäume werden analog zur rekursiven Definition als Terme modelliert:

$$\mathit{tree}(5, \mathit{tree}(3, \mathit{tree}(2, \mathit{init}, \mathit{init}), \mathit{tree}(4, \mathit{init}, \mathit{init})),$$
$$\mathit{tree}(8, \mathit{tree}(6, \mathit{init}, \mathit{tree}(7, \mathit{init}, \mathit{init})), \mathit{tree}(9, \mathit{init}, \mathit{init})))$$

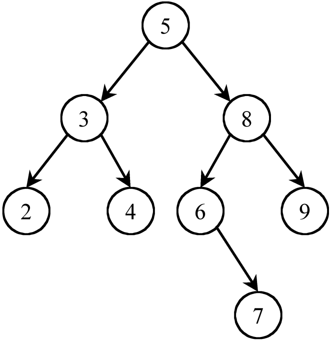


Modellierung von Bäumen als Terme

- Um Knoten in Bäumen eindeutig zu referenzieren, wird der **Pfad** zu dem jeweiligen Knoten verwendet.
- Dieser wird ausgehend von der Wurzel des Baumes gebildet.
- Der Pfad *nil* existiert für jeden nicht-leeren Baum und weist auf die Wurzel des jeweiligen Baums.
- Der Pfad $left(right(nil))$ referenziert z. B. den Knoten, der von der Wurzel aus erreicht wird, indem zunächst in den linken und in diesem dann in den rechten Teilbaum gewechselt wird.

Modellierung von Bäumen als Terme

- Beim exemplarischen Baum wird durch den Pfad *left*(*right*(*nil*)) der Knoten mit dem Schlüssel 4 referenziert.
- Der Pfad *right*(*left*(*right*(*nil*))) referenziert in diesem Baum den Knoten mit dem Schlüssel 7.



Grundlegende Funktionen eines Baums

Funktion	Beschreibung der Funktion
`init`	Erzeugt leeren Baum
`tree(n,t1,t2)`	Erzeugt neuen Baum aus einem Knoten und zwei Teilbäumen
`ltree(t)`	Liefert linken Teilbaum
`rtree(t)`	Liefert rechten Teilbaum
`getroot(t)`	Liefert Wurzelknoten
`empty(t)`	Prüft, ob Baum leer
`isin(n,t)`	Prüft, ob Knoten vorhanden
`height(t)`	Liefert Höhe des Baums
`minheight(t)`	Liefert minimale Höhe
`nodes(t)`	Liefert Anzahl der Knoten

Grundlegende Funktionen eines Baums

Funktion	Beschreibung der Funktion
`valid(p,t)`	Prüft, ob Pfad zu einem Knoten führt
`getnode(p,t)`	Liefert Knoten zum Pfad
`updnode(n,p,t)`	Aktualisiert Knoten zum Pfad
`lkupnode(n,t)`	Liefert Pfad zu Knoten
`getsubtree(p,t)`	Liefert Teilbaum zum Pfad
`delsubtree(p,t)`	Löscht Teilbaum zum Pfad
`updsubtree(u,p,t)`	Aktualisiert Teilbaum zum Pfad
`preorder(t)`	Liefert Preorder-Durchlauf als Liste
`inorder(t)`	Liefert Inorder-Durchlauf als Liste
`postorder(t)`	Liefert Postorder-Durchlauf als Liste

Grundlegende Funktionen eines Baums

Funktion	Beschreibung der Funktion
`minnode(t)`	Liefert kleinsten Knoten
`maxnode(t)`	Liefert größten Knoten
`sorted(t)`	Testet, ob Baum sortiert ist
`balanced(t)`	Testet, ob Baum balanciert ist

Grundlegende Funktionen eines Pfads

Funktion	Beschreibung der Funktion
`nil`	Liefert leeren Pfad
`left(p)`	Ergänzt Pfad um Abstieg nach links
`right(p)`	Ergänzt Pfad um Abstieg nach rechts
`parent(p)`	Entfernt den letzten Abstieg
`descend(p)`	Entfernt den ersten Abstieg
`isnil(p)`	Prüft, ob Pfad leer ist
`isleft(p)`	Prüft, ob erster Abstieg nach links geht
`isright(p)`	Prüft, ob erster Abstieg nach rechts geht
`length(p)`	Liefert die Länge des Pfads

Schrittweise Entwicklung der Spezifikation

- Die Beschreibung des Datentyps `tree` greift zurück auf boolesche Wahrheitswerte, ganze Zahlen, auf die Menge der Knoten, die in einem Baum sein können, sowie auf die Menge der Pfade:

$$[\mathbb{B}, \mathbb{Z}, \mathit{Node}, \mathit{Path}]$$

- Innerhalb von *Node* soll es ein Fehlerelement geben:

$$\mid \quad \mathit{errornode} \in \mathit{Node}$$

- Details zu *Path* folgen auf der übernächsten Folie.

Schrittweise Entwicklung der Spezifikation

- Wir definieren die Menge der Bäume:

  $[\mathit{Tree}]$

- Innerhalb von *Tree* soll es ein Fehlerelement geben:

  $\mathit{errortree} \in \mathit{Tree}$

- Bäume können mit folgenden Funktionen erzeugt werden:

  $\mathit{init} : \mathit{Tree}$
  $\mathit{tree} : \mathit{Node} \times \mathit{Tree} \times \mathit{Tree} \longrightarrow \mathit{Tree}$

- Bäume können nicht aus *errornode* oder *errortree* erzeugt werden:

  $$\forall\, n : \mathit{Node};\, l, r : \mathit{Tree} \bullet$$
  $$n = \mathit{errornode} \lor l = \mathit{errortree} \lor r = \mathit{errortree} \Rightarrow \mathit{tree}(n, l, r) = \mathit{errortree}$$

Schrittweise Entwicklung der Spezifikation

- Wir definieren die Menge der Pfade:

$$[Path]$$

- Innerhalb von *Path* soll es ein Fehlerelement geben:

$$errorpath \in Path$$

- Pfade können mit folgenden Funktionen erzeugt werden:

$$\begin{array}{l} nil : Path \\ left : Path \longrightarrow Path \\ right : Path \longrightarrow Path \end{array}$$

Schrittweise Entwicklung der Spezifikation

- Pfade können nicht aus *errorpath* erzeugt werden:

$$\mathit{left}(\mathit{errorpath}) = \mathit{errorpath}$$
$$\mathit{right}(\mathit{errorpath}) = \mathit{errorpath}$$

- Zur einfacheren Spezifikation der Eigenschaften definieren wir noch:

$$\mathit{Node}_V = \mathit{Node} \setminus \{\mathit{errornode}\}$$
$$\mathit{Path}_V = \mathit{Path} \setminus \{\mathit{errorpath}\}$$
$$\mathit{Tree}_V = \mathit{Tree} \setminus \{\mathit{errortree}\}$$

Baumoperationen: `ltree`

$$
\begin{array}{|l}
ltree : Tree \longrightarrow Tree \\
\hline
\forall\, n : Node_V;\ l, r : Tree_V \bullet \\
\qquad ltree(errortree) = errortree \\
\qquad ltree(init) = errortree \\
\qquad ltree(tree(n, l, r)) = l
\end{array}
$$

Beispiel 3 (`ltree`)

$$ltree(tree(\underbrace{0}_{n}, \underbrace{tree(2, init, init)}_{l}, \underbrace{tree(4, init, init)}_{r}))$$

$$= tree(2, init, init)$$

Baumoperationen: `rtree`

$$rtree : Tree \longrightarrow Tree$$

$$\forall\, n : Node_V;\ l, r : Tree_V \bullet$$

$$rtree(errortree) = errortree$$
$$rtree(init) = errortree$$
$$rtree(tree(n, l, r)) = r$$

Beispiel 4 (`rtree`)

$$rtree(tree(\underbrace{0}_{n}, \underbrace{tree(2, init, init)}_{l}, \underbrace{tree(4, init, init)}_{r}))$$

$$= tree(4, init, init)$$

Baumoperationen: getroot

$$getroot : Tree \longrightarrow Node$$

$$\forall\, n : Node_V;\ l, r : Tree_V \bullet$$

$$getroot(errortree) = errornode$$
$$getroot(init) = errornode$$
$$getroot(tree(n, l, r)) = n$$

Beispiel 5 (getroot)

- $getroot(tree(\underbrace{2}_{n}, \underbrace{init}_{l}, \underbrace{init}_{r})) = 2$
- $getroot(tree(\underbrace{0}_{n}, \underbrace{tree(2, init, init)}_{l}, \underbrace{tree(4, init, init)}_{r})) = 0$

Baumoperationen: `empty`

$$empty : Tree \longrightarrow \mathbb{B}$$

$$\forall\, n : Node_V;\ l, r : Tree_V \bullet$$

$$\begin{aligned} empty(errortree) &= True \\ empty(init) &= True \\ empty(tree(n, l, r)) &= False \end{aligned}$$

Beispiel 6 (empty)

$$\begin{aligned} & empty(tree(2, init, init)) \\ = {} & False \end{aligned}$$

Baumoperationen: `isin`

$$isin : Node \times Tree \longrightarrow \mathbb{B}$$

$$\forall\, n : Node;\ t : Tree \bullet$$
$$n = errornode \ \vee\ t = errortree$$
$$\Rightarrow isin(n, t) = False$$

$$\forall\, n, o : Node_V;\ l, r : Tree_V \mid n \neq o \bullet$$
$$isin(n, init) = False$$
$$isin(n, tree(n, l, r)) = True$$
$$isin(o, tree(n, l, r)) = oder(isin(o, l), isin(o, r))$$

Beispiel 7 (isin)

$$isin(3, tree(0, tree(2, init, init), tree(3, init, init)))$$
$$= oder(isin(3, tree(2, init, init)), isin(3, tree(3, init, init)))$$
$$= oder(oder(isin(3, init), isin(3, init)), True)$$
$$= oder(oder(False, False), True) = oder(False, True) = True$$

Baumoperationen: `height`

$$
\begin{array}{|l}
\quad height : Tree \longrightarrow \mathbb{Z} \\
\hline
\forall\, n : Node_V;\ l, r : Tree_V \bullet \\
\qquad height(errortree) = -1 \\
\qquad height(init) = 0 \\
\qquad height(tree(n, l, r)) = \\
\qquad\qquad 1 + \max[height(l), height(r)]
\end{array}
$$

Beispiel 8 (height)

$$
\begin{aligned}
& height(tree(\underbrace{0}_{n}, \underbrace{tree(2, init, init)}_{l}, \underbrace{init}_{r})) \\
&= 1 + \max[height(tree(2, init, init)), height(init)] \\
&= 1 + \max[1 + \max[height(init), height(init)], height(init)] \\
&= 1 + \max[1 + \max[0, 0], 0] \\
&= 1 + \max[1, 0] = 1 + 1 = 2
\end{aligned}
$$

Baumoperationen: `minheight`

$$
\begin{array}{|l}
minheight : Tree \longrightarrow \mathbb{Z} \\
\hline
\forall\, n : Node_V;\ l, r : Tree_V \bullet \\
\qquad minheight(erorrtree) = -1 \\
\qquad minheight(init) = 0 \\
\qquad minheight(tree(n, l, r)) = \\
\qquad\qquad 1 + \min[minheight(l), minheight(r)]
\end{array}
$$

Beispiel 9 (minheight)

$$minheight(tree(\underbrace{0}_{n}, \underbrace{tree(2, init, init)}_{l}, \underbrace{init}_{r}))$$

$$= 1 + \min[minheight(tree(2, init, init)), minheight(init)]$$

$$= 1 + \min[1 + min[minheight(init), minheight(init)], minheight(init)]$$

$$= 1 + \min[1 + \min[0, 0], 0]$$

$$= 1 + \min[1, 0] = 1 + 0 = 1$$

Baumoperationen: nodes

$$\begin{array}{|l} \quad nodes : Tree \longrightarrow \mathbb{Z} \\ \hline \forall\, n : Node_V;\ l, r : Tree_V \bullet \\ \qquad nodes(errortree) = -1 \\ \qquad nodes(init) = 0 \\ \qquad nodes(tree(n, l, r)) = 1 + nodes(l) + nodes(r) \end{array}$$

Beispiel 10 (nodes)

$$nodes(tree(\underbrace{0}_{n}, \underbrace{tree(2, init, init)}_{l}, \underbrace{tree(3, init, init)}_{r}))$$

$$= 1 + nodes(tree(2, init, init)) + nodes(tree(3, init, init))$$

$$= 1 + (1 + nodes(init) + nodes(init)) + (1 + nodes(init) + nodes(init))$$

$$= 1 + (1 + 0 + 0) + (1 + 0 + 0)$$

$$= 1 + 1 + 1 = 3$$

Baumoperationen: `preorder`

$$
\begin{array}{|l}
\textit{preorder} : \textit{Tree} \longrightarrow \textit{List} \\
\hline
\forall\, n : \textit{Node}_V;\ l, r : \textit{Tree}_V \bullet \\
\qquad \textit{preorder}(\textit{errortree}) = \textit{List.init} \\
\qquad \textit{preorder}(\textit{init}) = \textit{List.init} \\
\qquad \textit{preorder}(\textit{tree}(n, l, r)) = \textit{List.append}(\\
\qquad\qquad \textit{List.append}(\textit{List.insert}(n, \textit{List.init}), \textit{preorder}(l)), \\
\qquad\qquad \textit{preorder}(r))
\end{array}
$$

Beispiel 11 (preorder)

$$
\begin{aligned}
&\textit{preorder}(\textit{tree}(0, \textit{tree}(2, \textit{init}, \textit{init}), \textit{tree}(3, \textit{init}, \textit{init}))) \\
&\ldots \\
&= \textit{List.insert}(0, \textit{List.insert}(2, \textit{List.insert}(3, \textit{List.init})))
\end{aligned}
$$

Baumoperationen: inorder

$$inorder : Tree \longrightarrow List$$

$$\forall\, n : Node_V;\ l, r : Tree_V \bullet$$

$$\begin{aligned}
&inorder(errortree) = List.init\\
&inorder(init) = List.init\\
&inorder(tree(n, l, r)) = List.append(\\
&\qquad List.append(inorder(l), List.insert(n, List.init)),\\
&\qquad inorder(r))
\end{aligned}$$

Beispiel 12 (inorder)

$$inorder(tree(0, tree(2, init, init), tree(3, init, init)))$$

$$\ldots$$

$$= List.insert(2, List.insert(0, List.insert(3, List.init)))$$

Baumoperationen: `postorder`

$$postorder : Tree \longrightarrow List$$

$$\forall\, n : Node_V;\ l, r : Tree_V \bullet$$

$$postorder(errortree) = List.init$$
$$postorder(init) = List.init$$
$$postorder(tree(n, l, r)) = List.append($$
$$\quad List.append(postorder(l), postorder(r)),$$
$$\quad List.insert(n, List.init))$$

Beispiel 13 (postorder)

$$postorder(tree(0, tree(2, init, init), tree(3, init, init)))$$
$$\ldots$$
$$= List.insert(2, List.insert(3, List.insert(0, List.init)))$$

Baumoperationen: `valid`

$$valid : Path \times Tree \longrightarrow \mathbb{B}$$

$$\forall\, p : Path;\ t : Tree \bullet$$
$$p = errorpath \ \vee\ t = errortree$$
$$\Rightarrow valid(p, t) = False$$

$$\forall\, p : Path_V;\ n : Node_V;\ l, r : Tree_V \bullet$$
$$valid(p, init) = False$$
$$valid(nil, tree(n, l, r)) = True$$
$$valid(left(p), tree(n, l, r)) = valid(p, l)$$
$$valid(right(p), tree(n, l, r)) = valid(p, r)$$

Beispiel 14 (valid)

$$valid(left(\underbrace{nil}_{p}), tree(\underbrace{0}_{n}, \underbrace{tree(2, init, init)}_{l}, \underbrace{tree(3, init, init)}_{r})))$$
$$= valid(nil, tree(2, init, init)) = True$$

Baumoperationen: getnode

$$getnode : Path \times Tree \longrightarrow Node$$

$$\forall\, p : Path;\ t : Tree \bullet$$
$$p = errorpath \ \vee\ t = errortree$$
$$\Rightarrow getnode(p, t) = errornode$$

$$\forall\, p : Path_V;\ n : Node_V;\ l, r : Tree_V \bullet$$
$$getnode(p, init) = errornode$$
$$getnode(nil, tree(n, l, r)) = n$$
$$getnode(left(p), tree(n, l, r)) = getnode(p, l)$$
$$getnode(right(p), tree(n, l, r)) = getnode(p, r)$$

Beispiel 15 (getnode)

$$getnode(left(\underbrace{nil}_{p}), tree(\underbrace{0}_{n}, \underbrace{tree(2, init, init)}_{l}, \underbrace{tree(3, init, init)}_{r}))$$
$$= getnode(nil, tree(2, init, init)) = 2$$

Baumoperationen: `updnode`

$$updnode : Node \times Path \times Tree \longrightarrow Tree$$

$$\forall\, n : Node;\ p : Path;\ t : Tree \bullet$$

$$p = errorpath \lor n = errornode \lor t = errortree$$
$$\Rightarrow updnode(n, p, t) = errortree$$
$$p \neq nil \Rightarrow updnode(n, p, init) = errortree$$

$$\forall\, n, o : Node_V;\ p : Path_V;\ l, r : Tree_V \bullet$$

$$updnode(n, nil, init) = tree(n, init, init)$$

$$updnode(n, nil, tree(o, l, r)) =$$
$$tree(n, l, r)$$

$$updnode(n, left(p), tree(o, l, r)) =$$
$$tree(o, updnode(n, p, l), r)$$

$$updnode(n, right(p), tree(o, l, r)) =$$
$$tree(o, l, updnode(n, p, r))$$

Baumoperationen: `updnode`

Beispiel 16 (updnode)

$$updnode(\underbrace{4}_{n}, nil, tree(\underbrace{0}_{o}, \underbrace{tree(2, init, init)}_{l}, \underbrace{tree(3, init, init)}_{r}))$$

$$= tree(4, tree(2, init, init), tree(3, init, init))$$

$$updnode(\underbrace{4}_{n}, left(\underbrace{nil}_{p}), tree(\underbrace{0}_{o}, \underbrace{tree(2, init, init)}_{l}, \underbrace{tree(3, init, init)}_{r}))$$

$$= tree(0, updnode(\underbrace{4}_{n}, nil, tree(\underbrace{2}_{o}, \underbrace{init}_{l}, \underbrace{init}_{r})), tree(3, init, init))$$

$$= tree(0, tree(4, init, init), tree(3, init, init))$$

Baumoperationen: `lkupnode`

$$lkupnode : Node \times Tree \longrightarrow Path$$

$$\forall\, n : Node;\ t : Tree \bullet$$

$$n = errornode \ \vee\ t = errortree$$
$$\Rightarrow lkupnode(n, t) = errorpath$$

$$\forall\, p : Path_V;\ n, o : Node_V;\ l, r : Tree_V \mid n \neq o \bullet$$

$$lkupnode(o, init) = errorpath$$

$$\neg isin(o, r) \ \wedge\ \neg isin(o, l)$$
$$\Rightarrow lkupnode(o, tree(n, l, r)) = errorpath$$

$$lkupnode(o, tree(o, l, r)) = nil$$

$$isin(o, l)$$
$$\Rightarrow lkupnode(o, tree(n, l, r)) = left(lkupnode(o, l))$$

$$\neg isin(o, l) \ \wedge\ isin(o, r)$$
$$\Rightarrow lkupnode(o, tree(n, l, r)) = right(lkupnode(o, r))$$

Baumoperationen: `lkupnode`

- Die spezifizierte Version von `lkupnode` sucht bevorzugt im linken Teilbaum nach dem gesuchten Knoten.
- Nur, wenn der Knoten im linken Teilbaum nicht enthalten ist, wird im rechten Teilbaum nach diesem gesucht.
- Dies ist bedeutsam, falls Duplikate im Baum vorhanden sind.

Beispiel 17 (lkupnode)

$$\begin{aligned}&\mathit{lkupnode}(3, \mathit{tree}(0, \mathit{tree}(2, \mathit{init}, \mathit{init}), \mathit{tree}(3, \mathit{init}, \mathit{init})))\\ &= \mathit{right}(\mathit{lkupnode}(3, \mathit{tree}(3, \mathit{init}, \mathit{init})))\\ &= \mathit{right}(\mathit{nil})\end{aligned}$$

$$\mathit{isin}(3, \mathit{tree}(2, \mathit{init}, \mathit{init})) = \mathit{False}$$

$$\mathit{isin}(3, \mathit{tree}(3, \mathit{init}, \mathit{init})) = \mathit{True}$$

Baumoperationen: getsubtree

$$
\begin{array}{|l}
\textit{getsubtree} : \textit{Path} \times \textit{Tree} \longrightarrow \textit{Tree} \\
\hline
\forall\, p : \textit{Path};\ t : \textit{Tree} \bullet \\
\qquad p = \textit{errorpath} \ \vee\ t = \textit{errortree} \\
\qquad\qquad \Rightarrow \textit{getsubtree}(p, t) = \textit{errortree} \\
\\
\forall\, p : \textit{Path}_V;\ n : \textit{Node}_V;\ l, r : \textit{Tree}_V \bullet \\
\qquad \textit{getsubtree}(p, \textit{init}) = \textit{errortree} \\
\qquad \textit{getsubtree}(\textit{nil}, \textit{tree}(n, l, r)) = \textit{tree}(n, l, r) \\
\qquad \textit{getsubtree}(\textit{left}(p), \textit{tree}(n, l, r)) = \textit{getsubtree}(p, l) \\
\qquad \textit{getsubtree}(\textit{right}(p), \textit{tree}(n, l, r)) = \textit{getsubtree}(p, r)
\end{array}
$$

Beispiel 18 (getsubtree)

$$
\textit{getsubtree}(\textit{right}(\underbrace{\textit{nil}}_{p}), \textit{tree}(\underbrace{0}_{n}, \underbrace{\textit{tree}(2, \textit{init}, \textit{init})}_{l}, \underbrace{\textit{tree}(3, \textit{init}, \textit{init})}_{r})))
$$

$$
= \textit{getsubtree}(\textit{nil}, \textit{tree}(3, \textit{init}, \textit{init})) = \textit{tree}(3, \textit{init}, \textit{init})
$$

Baumoperationen: `delsubtree`

$$delsubtree : Path \times Tree \longrightarrow Tree$$

$$\forall\, p : Path;\ t : Tree \bullet$$
$$p = errorpath \ \vee\ t = errortree$$
$$\Rightarrow delsubtree(p, t) = errortree$$

$$\forall\, p : Path_V;\ n : Node_V;\ l, r : Tree_V \bullet$$
$$delsubtree(p, init) = errortree$$
$$delsubtree(nil, tree(n, l, r)) = init$$
$$delsubtree(left(p), tree(n, l, r)) = tree(n, delsubtree(p, l), r)$$
$$delsubtree(right(p), tree(n, l, r)) = tree(n, l, delsubtree(p, r))$$

Beispiel 19 (delsubtree)

$$delsubtree(right(nil), tree(0, tree(2, init, init), tree(3, init, init)))$$
$$= tree(0, tree(2, init, init), delsubtree(nil, tree(3, init, init)))$$
$$= tree(0, tree(2, init, init), init)$$

Baumoperationen: `updsubtree`

$$updsubtree : Tree \times Path \times Tree \longrightarrow Tree$$

$$\forall\, u, t : Tree;\ p : Path \bullet$$
$$p = errorpath \lor u = errortree \lor t = errortree \Rightarrow updsubtree(u, p, t) = errortree$$
$$p \neq nil \Rightarrow updsubtree(u, p, init) = errortree$$

$$\forall\, l, r, u : Tree_V;\ p : Path_V;\ n : Node_V \bullet$$
$$updsubtree(u, nil, init) = u$$
$$updsubtree(u, nil, tree(n, l, r)) = u$$
$$updsubtree(u, left(p), tree(n, l, r)) = tree(n, updsubtree(u, p, l), r)$$
$$updsubtree(u, right(p), tree(n, l, r)) = tree(n, l, updsubtree(u, p, r))$$

Baumoperationen: `updsubtree`

Beispiel 20 (updsubtree)

$$\mathit{updsubtree}(\mathit{tree}(4, \mathit{init}, \mathit{tree}(5, \mathit{init}, \mathit{init})), \mathit{right}(\mathit{nil}),$$
$$\mathit{tree}(0, \mathit{tree}(2, \mathit{init}, \mathit{init}), \mathit{tree}(3, \mathit{init}, \mathit{init})))$$

$$= \mathit{tree}(0, \mathit{tree}(2, \mathit{init}, \mathit{init}),$$
$$\mathit{updsubtree}(\mathit{tree}(4, \mathit{init}, \mathit{tree}(5, \mathit{init}, \mathit{init})), \mathit{nil}, \mathit{tree}(3, \mathit{init}, \mathit{init})))$$

$$= \mathit{tree}(0, \mathit{tree}(2, \mathit{init}, \mathit{init}), \mathit{tree}(4, \mathit{init}, \mathit{tree}(5, \mathit{init}, \mathit{init})))$$

Baumoperationen: `minnode`

$$minnode : Tree \longrightarrow \mathbb{Z}$$

$$\forall\, n : Node_V;\ l, r : Tree_V \bullet$$
$$minnode(errortree) = \infty$$
$$minnode(init) = \infty$$
$$minnode(tree(n, l, r)) = \min[n, minnode(l), minnode(r)]$$

Beispiel 21 (minnode)

$$\begin{aligned}
& minnode(tree(0, tree(2, init, init), tree(3, init, init))) \\
&= \min[0, minnode(tree(2, init, init), minnode(tree(3, init, init)))] \\
&= \min[0, \min[2, minnode(init), minnode(init)], \\
&\qquad\quad \min[3, minnode(init), minnode(init)]] \\
&= \min[0, \min[2, \infty, \infty], \min[3, \infty, \infty]] \\
&= \min[0, 2, 3] = 0
\end{aligned}$$

Baumoperationen: maxnode

$$maxnode : Tree \longrightarrow Node$$

$$\forall\, n : Node_V;\, l, r : Tree_V \bullet$$
$$maxnode(errortree) = -\infty$$
$$maxnode(init) = -\infty$$
$$maxnode(tree(n, l, r)) = \max[n, maxnode(l), maxnode(r)]$$

Beispiel 22 (maxnode)

$$\begin{aligned}
&maxnode(tree(0, tree(2, init, init), tree(3, init, init))) \\
&= \max[0, maxnode(tree(2, init, init), maxnode(tree(3, init, init)))] \\
&= \max[0, \max[2, maxnode(init), maxnode(init)], \\
&\qquad\qquad \max[3, maxnode(init), maxnode(init)]] \\
&= \max[0, \max[2, -\infty, -\infty], \max[3, -\infty, -\infty]] \\
&= \max[0, 2, 3] = 3
\end{aligned}$$

Baumoperationen: `sorted`

$$
\begin{array}{|l}
sorted : Tree \longrightarrow \mathbb{B} \\
\hline
\forall\, n : Node_V;\ l, r : Tree_V \bullet \\
\qquad sorted(errortree) = False \\
\qquad sorted(init) = True \\
\qquad sorted(tree(n, init, init)) = True \\
\qquad sorted(tree(n, l, r)) = sorted(l) \wedge sorted(r) \wedge \\
\qquad\qquad maxnode(l) \leq n \wedge minnode(r) \geq n
\end{array}
$$

Beispiel 23 (sorted)

$$
\begin{aligned}
& sorted(tree(0, tree(2, init, init), tree(3, init, init))) \\
& = sorted(tree(2, init, init)) \wedge sorted(tree(3, init, init)) \\
& \qquad \wedge maxnode(tree(2, init, init)) \leq 0 \wedge minnode(tree(3, init, init)) \geq 0 \\
& = True \wedge True \wedge False \wedge True = False
\end{aligned}
$$

Baumoperationen: `balanced`

$$balanced : Tree \longrightarrow \mathbb{B}$$

$$\forall\, t : Tree \bullet$$
$$balanced = (height(t) - minheight(t) \leq 1)$$

Beispiel 24 (sorted)

$$balanced(tree(0, tree(2, init, init), tree(3, init, init)))$$
$$= (2 - 2) \leq 1$$
$$= True$$

Achtung

- Nach obiger Spezifikation gilt:

$$balanced(errortree) = True$$

Pfadoperationen: `parent`

$$
\begin{array}{|l}
\quad parent : Path \longrightarrow Path \\
\hline
\quad \forall\, p : Path_V \mid p \neq nil \bullet \\
\qquad\qquad parent(nil) = errorpath \\
\qquad\qquad parent(errorpath) = errorpath \\[1ex]
\qquad\qquad parent(left(nil)) = nil \\
\qquad\qquad parent(right(nil)) = nil \\[1ex]
\qquad\qquad parent(left(p)) = left(parent(p)) \\
\qquad\qquad parent(right(p)) = right(parent(p))
\end{array}
$$

- Lässt den letzten Abstieg des Pfads weg.

Beispiel 25 (parent)

$$parent(left(\underbrace{right(nil)}_{p})) = left(parent(right(nil))) = left(nil)$$

Pfadoperationen: `descend`

$$
\begin{array}{|l}
\textit{descend} : \textit{Path} \longrightarrow \textit{Path} \\
\hline
\forall\, p : \textit{Path}_V \bullet \\
\qquad \textit{descend}(\textit{errorpath}) = \textit{errorpath} \\
\qquad \textit{descend}(\textit{nil}) = \textit{errorpath} \\
\\
\qquad \textit{descend}(\textit{left}(p)) = p \\
\qquad \textit{descend}(\textit{right}(p)) = p
\end{array}
$$

- Lässt den ersten Abstieg des Pfads weg.

Beispiel 26 (descend)

$$
\begin{aligned}
&\textit{descend}(\textit{left}(\underbrace{\textit{right}(\textit{nil})}_{p})) \\
&= \textit{right}(\textit{nil})
\end{aligned}
$$

Pfadoperationen: `isnil`

$$isnil : Path \longrightarrow \mathbb{B}$$

$$\forall\, p : Path_V \bullet$$

$$isnil(errorpath) = False$$

$$isnil(nil) = True$$
$$isnil(left(p)) = False$$
$$isnil(right(p)) = False$$

- Prüft, ob der Pfad leer ist.

Beispiel 27 (isnil)

$$isnil(left(right(nil)))$$
$$= False$$

Pfadoperationen: `isleft`

$$isleft : Path \longrightarrow \mathbb{B}$$

$$\begin{aligned}
&\forall\, p : Path_V \bullet \\
&\qquad isleft(errorpath) = False \\
&\qquad isleft(nil) = False \\
&\qquad isleft(left(p)) = True \\
&\qquad isleft(right(p)) = False
\end{aligned}$$

- Prüft, ob erster Abstieg nach links geht.

Beispiel 28 (isleft)

$$\begin{aligned}
&isleft(left(right(nil))) \\
&= True
\end{aligned}$$

Pfadoperationen: `isright`

$$isright : Path \longrightarrow \mathbb{B}$$

$$\forall\, p : Path_V \bullet$$
$$isright(errorpath) = False$$
$$isright(nil) = False$$
$$isright(left(p)) = False$$
$$isright(right(p)) = True$$

- Prüft, ob erster Abstieg nach rechts geht.

Beispiel 29 (isright)

$$\begin{aligned} & isright(left(right(nil))) \\ = {} & False \end{aligned}$$

Pfadoperationen: `length`

$$length : Path \longrightarrow \mathbb{Z}$$

$$\forall\, p : Path_V \bullet$$

$$length(errorpath) = -1$$

$$length(nil) = 0$$
$$length(left(p)) = 1 + length(p)$$
$$length(right(p)) = 1 + length(p)$$

- Liefert die Länge des Pfads.

Beispiel 30 (isright)

$$\begin{aligned} & length(left(right(nil))) \\ & = 1 + length(right(nil)) \\ & = 1 + 1 + length(nil) \\ & = 1 + 1 + 0 = 2 \end{aligned}$$

Ausführbare Spezifikation des Baums

```
module Path where
import Prelude hiding (Left,Right)

data Path = Errorpath | Nil | Left(Path) | Right(Path)
     deriving (Eq,Show)

nil   :: Path
left  :: Path -> Path
right :: Path -> Path

nil      = Nil
left(p)  = Left(p)
right(p) = Right(p)
```

Ausführbare Spezifikation des Baums

```
parent  :: Path -> Path
descend :: Path -> Path

parent(Nil)          = Errorpath
parent(Errorpath)    = Errorpath
parent(Left(Nil))    = Nil
parent(Right(Nil))   = Nil
parent(Left(p))      = Left(parent(p))
parent(Right(p))     = Right(parent(p))

descend(Nil)         = Errorpath
descend(Errorpath)   = Errorpath
descend(Left(p))     = p
descend(Right(p))    = p
```

Ausführbare Spezifikation des Baums

```
isnil   :: Path -> Bool
isleft  :: Path -> Bool
isright :: Path -> Bool

isnil(Errorpath) = False
isnil(p)         = if p == Nil then True else False

isleft(Nil)       = False
isleft(Errorpath) = False
isleft(Left(p))   = True
isleft(Right(p))  = False

isright(Nil)       = False
isright(Errorpath) = False
isright(Left(p))   = True
isright(Right(p))  = False
```

Ausführbare Spezifikation des Baums

```
module Tree where
import Prelude hiding (init,Left,Right)
import qualified List
import Path

type Node = Int
errornode = -1

data Tree = Errortree | Empty | Tree(Node,Tree,Tree)
     deriving (Eq,Show)

init :: Tree

init = Empty
```

Ausführbare Spezifikation des Baums

```
tree  :: (Node,Tree,Tree) -> Tree
ltree :: Tree -> Tree
rtree :: Tree -> Tree

tree(n,l,r) =
      if n == errornode then Errortree
      else if l == Errortree then Errortree
           else if r == Errortree then Errortree
                else Tree(n,l,r)

ltree(Errortree)   = Errortree
ltree(Empty)       = Errortree
ltree(Tree(n,l,r)) = l

rtree(Errortree)   = Errortree
rtree(Empty)       = Errortree
rtree(Tree(n,l,r)) = r
```

Ausführbare Spezifikation des Baums

```
getroot :: Tree -> Node
empty   :: Tree -> Bool

getroot(Errortree)    = errornode
getroot(Empty)        = errornode
getroot(Tree(n,l,r)) = n

empty(Errortree)    = True
empty(Empty)        = True
empty(Tree(n,l,r)) = False
```

Ausführbare Spezifikation des Baums

```
isin   :: (Node,Tree) -> Bool

isin(n,Empty)        = False
isin(n,Errortree)    = False
isin(n,Tree(o,l,r)) =
      if n == errornode then False
      else if n == o then True
           else if isin(n,l) then True
                else isin(n,r)
```

Ausführbare Spezifikation des Baums

```
height    :: Tree -> Int
minheight :: Tree -> Int
nodes     :: Tree -> Int

height(Errortree)   = -1
height(Empty)       = 0
height(Tree(n,l,r)) =
      1 + max (height(l)) (height(r))

minheight(Errortree)   = -1
minheight(Empty)       = 0
minheight(Tree(n,l,r)) =
      1 + min (minheight(l)) (minheight(r))

nodes(Errortree)   = -1
nodes(Empty)       = 0
nodes(Tree(v,l,r)) = 1 + nodes(l) + nodes(r)
```

Ausführbare Spezifikation des Baums

```
preorder  :: Tree -> List.List
inorder   :: Tree -> List.List
postorder :: Tree -> List.List

preorder(Errortree)    = List.init
preorder(Empty)        = List.init
preorder(Tree(n,l,r))  = List.append(
                            List.insert(n,List.init),
                            List.append(preorder(l),preorder(r)))

inorder(Errortree)     = List.init
inorder(Empty)         = List.init
inorder(Tree(n,l,r))   = List.append(
                            List.append(inorder(l),
                                        List.insert(n,List.init)),
                            inorder(r))

postorder(Errortree)   = List.init
postorder(Empty)       = List.init
postorder(Tree(n,l,r)) = List.append(
                            List.append(postorder(l),postorder(r)),
                            List.insert(n,List.init))
```

Ausführbare Spezifikation des Baums

```
valid    :: (Path,Tree) -> Bool
getnode  :: (Path,Tree) -> Node

valid(p,Errortree)             = False
valid(Errorpath,t)             = False
valid(p,Empty)                 = False
valid(Nil,Tree(n,l,r))         = True
valid(Left(p),Tree(n,l,r))     = valid(p,l)
valid(Right(p),Tree(n,l,r))    = valid(p,r)

getnode(p,Errortree)             = errornode
getnode(Errorpath,t)             = errornode
getnode(p,Empty)                 = errornode
getnode(Nil,Tree(n,l,r))         = n
getnode(Left(p),Tree(n,l,r))     = getnode(p,l)
getnode(Right(p),Tree(n,l,r))    = getnode(p,r)
```

Ausführbare Spezifikation des Baums

```
updnode  :: (Node,Path,Tree) -> Tree

updnode(n,Errorpath,t)            = Errortree
updnode(n,p,Errortree)            = Errortree

updnode(n,Nil,Tree(o,l,r))        =
      if n == errornode then Errortree
      else Tree(n,l,r)

updnode(n,Left(p),Tree(o,l,r))    =
      if n == errornode then Errortree
      else Tree(o,updnode(n,p,l),r)

updnode(n,Right(p),Tree(o,l,r)) =
      if n == errornode then Errortree
      else Tree(o,l,updnode(n,p,r))
```

Ausführbare Spezifikation des Baums

```
lkupnode :: (Node,Tree) -> Path

lkupnode(n,Errortree)    = Errorpath
lkupnode(n,Empty)        = Errorpath

lkupnode(o,Tree(n,l,r)) =
      if o == errornode then Errorpath
      else if n == o then Nil
           else if isin(o,l) then left(lkupnode(o,l))
                else if isin(o,r)
                           then right(lkupnode(o,r))
                     else Errorpath
```

Ausführbare Spezifikation des Baums

```
getsubtree :: (Path,Tree) -> Tree

getsubtree(p,Errortree)              = Errortree
getsubtree(Errorpath,t)              = Errortree
getsubtree(p,Empty)                  = Errortree

getsubtree(Nil,Tree(n,l,r))          = Tree(n,l,r)
getsubtree(Left(p),Tree(n,l,r))      = getsubtree(p,l)
getsubtree(Right(p),Tree(n,l,r))     = getsubtree(p,r)
```

Ausführbare Spezifikation des Baums

```
delsubtree :: (Path,Tree) -> Tree

delsubtree(p,Errortree)             = Errortree
delsubtree(Errorpath,t)             = Errortree
delsubtree(p,Empty)                 = Errortree

delsubtree(Nil,Tree(n,l,r))         = Empty
delsubtree(Left(p),Tree(n,l,r))     =
      Tree(n,delsubtree(p,l),r)
delsubtree(Right(p),Tree(n,l,r))    =
      Tree(n,l,delsubtree(p,r))
```

Ausführbare Spezifikation des Baums

```
updsubtree :: (Tree,Path,Tree) -> Tree

updsubtree(u,p,Errortree)          = Errortree
updsubtree(Errortree,p,t)          = Errortree
updsubtree(u,Errorpath,t)          = Errortree

updsubtree(u,p,Empty)              =
      if p == Nil then u else Errortree

updsubtree(u,Nil,Tree(n,l,r))      = u

updsubtree(u,Left(p),Tree(n,l,r))  =
      Tree(n,updsubtree(u,p,l),r)

updsubtree(u,Right(p),Tree(n,l,r)) =
      Tree(n,l,updsubtree(u,p,r))
```

Ausführbare Spezifikation des Baums

```
minnode :: Tree -> Node
maxnode :: Tree -> Node

minnode(Errortree)   = errornode
minnode(Empty)       = maxBound
minnode(Tree(n,l,r)) =
      min (min (n) (minnode(l))) (minnode(r))

maxnode(Errortree)   = errornode
maxnode(Empty)       = minBound
maxnode(Tree(n,l,r)) =
      max (max (n) (maxnode(l))) (maxnode(r))
```

Ausführbare Spezifikation des Baums

```
sorted   :: Tree -> Bool
balanced :: Tree -> Bool

sorted(Errortree)   = False
sorted(Empty)       = True
sorted(Tree(n,l,r)) =
      (sorted(l)) && (sorted(r)) &&
      (maxnode(l) <= n) && (minnode(r) >= n)

balanced(t) = (height(t) - minheight(t)) <= 1
```

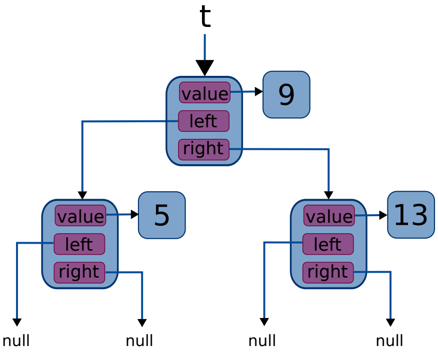
Kapitel 7.3

Implementierung

Implementierung Baum

- Umsetzung eines Baums durch eine verkette Datenstruktur: Ein Baum ist entweder leer oder er enthält Zeiger auf ein Element sowie seinen linken und seinen rechten Teilbaum.
- Die Implementierung unterscheidet also nicht zwischen Baum und Knoten.

```
struct _tree {
  element value;
  struct _tree* left;
  struct _tree* right;
};
```



Implementierung Pfad

- Umsetzung durch einen String in einem Struct.
- Der leere String `""` steht für den Pfad *nil*.
- Das vorangestellte Zeichen `l` steht für einen Abstieg *zuerst* nach links.
- Das vorangestellte Zeichen `r` für einen Abstieg *zuerst* nach rechts.
- Zum Beispiel steht der String `"rl"` für den Pfad, der bei der Wurzel beginnt und zunächst nach rechts und dann nach links absteigt.
- Bei den Operationen `left(p)` und `right(p)` wird dem String des Pfads `p` ein `l` bzw. `r` vorangestellt.

Beispiel 31 (Pfadoperationen)

- nil = `""`
- left(right(nil)) = `"lr"`
- right(left(right(nil))) = `"rlr"`

Implementierung Pfad

- Bei `parent(p)` wird das letzte Zeichen des Strings des Pfads `p` gelöscht, also der letzte Abstieg.
- Bei `descend(p)` wird das erste Zeichen des Strings des Pfads `p` gelöscht, also der erste Abstieg.
- Durch diese Vorgehensweise können alle Pfadoperationen effizient durch Stringoperationen durchgeführt werden.

Beispiel 32 (Pfadoperationen)

- right(left(right(nil))) = `"rlr"`
- parent(right(left(right(nil)))) = `"rl"`
- descend(right(left(right(nil)))) = `"lr"`

Implementierung

```
#define PATH_MAX_LENGTH 32

struct _path {
  char p_str[PATH_MAX_LENGTH];
};

typedef struct _path path;

static path PATH_ERROR_PATH = {"ERROR_PATH"};

int path_is_error_path(path p) {
  return strcmp(p.p_str, PATH_ERROR_PATH.p_str) == 0;
}
```

Implementierung nil

```
int path_length(path p) {
  if (path_is_error_path(p))
    return -1;
  else
    return strlen(p.p_str);
}

path path_nil() {
  path p;
  p.p_str[0] = 0;
  return p;
}
```

Implementierung copy und isnil

```
path path_copy(path p) {
  if (path_is_error_path(p))
    return PATH_ERROR_PATH;
  path q;
  strcpy(q.p_str, p.p_str);
  return q;
}

int path_is_nil(path p) {
  if (path_is_error_path(p))
    return 0;
  return path_length(p) == 0;
}
```

Implementierung `left` und `right`

```
void path_move_right(char* str, char c) {
  // "lr0" --> "clr0"
  for (int i = strlen(str) + 1; i > 0; i--)
    str[i] = str[i - 1];
  str[0] = c;
}

void path_move_left(char* str) {
  // "lr0" --> "r0"
  for (int i = 0; i < strlen(str); i++)
    str[i] = str[i + 1];
}
```

Implementierung left und right

```
path path_left(path p) {
  if (path_is_error_path(p))
    return PATH_ERROR_PATH;
  path_move_right(p.p_str, 'l');
  return p;
}

path path_right(path p) {
  if (path_is_error_path(p))
    return PATH_ERROR_PATH;
  path_move_right(p.p_str, 'r');
  return p;
}
```

Implementierung `parent` und `descend`

```
path path_parent(path p) {
  if (path_is_error_path(p) || path_is_nil(p))
    return PATH_ERROR_PATH;
  p.p_str[path_length(p) - 1] = 0;
  return p;
}

path path_descend(path p) {
  if (path_is_error_path(p) || path_is_nil(p))
    return PATH_ERROR_PATH;
  path_move_left(p.p_str);
  return p;
}
```

Implementierung isleft und isright

```
int path_is_left(path p) {
  if (path_is_error_path(p) || path_is_nil(p))
    return 0;
  return p.p_str[0] == 'l';
}

int path_is_right(path p) {
  if (path_is_error_path(p) || path_is_nil(p))
    return 0;
  return p.p_str[0] == 'r';
}
```

Implementierung

```
#include "path.h"
#include "../arraylist/arraylist.h"

#define TREE_ERROR_ELEMENT INT_MIN

typedef int element;

struct _tree {
  element value;
  struct _tree* left;
  struct _tree* right;
};

typedef struct _tree tree;
```

Implementierung

```c
static int tree_no_mallocs = 0;
static int tree_no_frees = 0;

void* tree_malloc(size_t size) {
  tree_no_mallocs++;
  return malloc(size);
}

void tree_free(void* ptr) {
  if (ptr != NULL)
    tree_no_frees++;
  free(ptr);
}
```

Implementierung

```
int tree_get_mallocs() { return tree_no_mallocs; }

int tree_get_frees() { return tree_no_frees; }

void tree_print_mallocs_frees() {
  printf("tree_no_mallocs␣=␣%i\n", tree_get_mallocs());
  printf("tree_no_frees␣␣␣=␣%i\n", tree_get_frees());
  if (tree_get_mallocs() > tree_get_frees())
    printf("memory␣leak␣in␣tree␣implementation!\n");
  else if (tree_get_mallocs() < tree_get_frees())
    printf("double␣free␣in␣tree␣implementation!\n");
}
```

Implementierung init und tree

```
tree* tree_init() {
  tree* n = tree_malloc(sizeof(tree));
  n->left = NULL;
  n->right = NULL;
  n->value = TREE_ERROR_ELEMENT;
  return n;
}

tree* tree_build(element e, tree* t1, tree* t2) {
  if (t1 == NULL || t2 == NULL ||
        e == TREE_ERROR_ELEMENT)
    return NULL;
  tree* t = tree_init();
  t->value = e;
  t->left = t1;
  t->right = t2;
  return t;
}
```

Implementierung empty, ltree und rtree

```
tree* tree_ltree(tree* t) {
  return tree_empty(t) ? NULL : t->left;
}

tree* tree_rtree(tree* t) {
  return tree_empty(t) ? NULL : t->right;
}
```

Implementierung getroot und empty

```
element tree_get_root(tree* t) {
  return tree_empty(t) ?
             TREE_ERROR_ELEMENT : t->value;
}

int tree_empty(tree* t) {
  return t == NULL || t->value == TREE_ERROR_ELEMENT;
}
```

Implementierung `isin`

```c
int tree_isin(element e, tree* t) {
  if (e == TREE_ERROR_ELEMENT || tree_empty(t))
    return 0;

  if (t->value == e)
    return 1;

  return tree_isin(e, t->left) ||
             tree_isin(e, t->right);
}
```

- Der rechte Teilbaum wird nur durchsucht, wenn das gesuchte Element nicht im linken Teilbaum enthalten ist.

Implementierung height

```
int tree_get_height(tree* t) {
  if (tree_empty(t))
    return 0;

  int l = tree_get_height(t->left);
  int r = tree_get_height(t->right);
  return 1 + l > r ? l : r;
}

int tree_height(tree* t) {
  return t == NULL ? -1 : tree_get_height(t);
}
```

Implementierung minheight

```
int tree_get_min_height(tree* t) {
  if (tree_empty(t))
    return 0;

  int l = tree_get_min_height(t->left);
  int r = tree_get_min_height(t->right);
  return 1 + l < r ? l : r;
}

int tree_minheight(tree* t) {
  return t == NULL ? -1 : tree_get_min_height(t);
}
```

Implementierung nodes

```
int tree_count_nodes(tree* t) {
  if (tree_empty(t))
    return 0;

  int l = tree_count_nodes(t->left);
  int r = tree_count_nodes(t->right);
  return 1 + l + r;
}

int tree_nodes(tree* t) {
  return t == NULL ? -1 : tree_count_nodes(t);
}
```

Implementierung valid

```
int tree_valid(path p, tree* t) {
  int x;
  if (path_is_error_path(p) || tree_empty(t))
    x = 0;
  else if (path_is_nil(p) &&
              t->value != TREE_ERROR_ELEMENT)
    x = 1;
  else if (path_is_left(p) && t->left != NULL &&
           t->left->value != TREE_ERROR_ELEMENT)
    x = tree_valid(path_descend(p), t->left);
  else if (path_is_right(p) && t->right != NULL &&
           t->right->value != TREE_ERROR_ELEMENT)
    x = tree_valid(path_descend(p), t->right);
  else
    x = 0;
  return x;
}
```

Implementierung getnode

```
element tree_get_node(path p, tree* t) {
  element e = TREE_ERROR_ELEMENT;
  if (path_is_error_path(p) || tree_empty(t))
    e = TREE_ERROR_ELEMENT;
  else if (path_is_nil(p) &&
                t->value != TREE_ERROR_ELEMENT)
    e = t->value;
  else if (path_is_left(p) && t->left != NULL &&
            t->left->value != TREE_ERROR_ELEMENT)
    e = tree_get_node(path_descend(p), t->left);
  else if (path_is_right(p) && t->right != NULL &&
            t->right->value != TREE_ERROR_ELEMENT)
    e = tree_get_node(path_descend(p), t->right);
  else
    e = TREE_ERROR_ELEMENT;
  return e;
}
```

Implementierung updnode

```
tree* tree_updnode(element e, path p, tree* t) {
  if (path_is_error_path(p) ||
        e == TREE_ERROR_ELEMENT || t == NULL)
    return NULL;

  if (!path_is_nil(p) && tree_empty(t))
    return NULL;

  tree* s = NULL;

  // continued on next slide
```

Implementierung updnode

```
  // continued from previous slide

  if (path_is_nil(p)) {
    if (tree_empty(t))
      s = tree_build(e, tree_init(), tree_init());
    else
      s = tree_build(e, t->left, t->right);
  } else if (path_is_left(p))
    s = tree_build(t->value,
          tree_updnode(e, path_descend(p), t->left),
          t->right);
  else // path_is_right()
    s = tree_build(t->value, t->left,
          tree_updnode(e, path_descend(p), t->right));
  tree_free(t);
  return s;
}
```

Implementierung `lkupnode`

```
path tree_lkupnode(element e, tree* t) {
  if (e == TREE_ERROR_ELEMENT || tree_empty(t))
    return PATH_ERROR_PATH;

  // node found?
  if (t->value == e)
    return path_nil();
  // node in left subtree?
  if (tree_isin(e, t->left))
    return path_left(tree_lkupnode(e, t->left));
  // node in right subtree?
  if (tree_isin(e, t->right))
    return path_right(tree_lkupnode(e, t->right));
  // node was not found!
  return PATH_ERROR_PATH;
}
```

Implementierung getsubtree

```
tree* tree_get_subtree(path p, tree* t) {
  if (path_is_error_path(p) || tree_empty(t))
    return NULL;

  if (path_is_nil(p))
    return t;

  if (path_is_left(p))
    return tree_get_subtree(path_descend(p), t->left);
  else // path_is_right(p)
    return tree_get_subtree(path_descend(p), t->right);
}
```

Implementierung delsubtree

```
tree* tree_del_subtree(path p, tree* t) {
  if (path_is_error_path(p) || tree_empty(t))
    return NULL;

  if (path_is_nil(p)) {
    tree_destroy(t);
    return tree_init();
  };

  // continued on next slide
```

Implementierung delsubtree

```
  // continued from previous slide

  tree* s = NULL;
  if (path_is_left(p))
    s = tree_build(t->value,
            tree_del_subtree(path_descend(p), t->left),
            t->right);
  else // path_is_right(p)
    s = tree_build(t->value, t->left,
            tree_del_subtree(path_descend(p), t->right));

  tree_free(t);

  return s;
}
```

Implementierung updsubtree

```
tree* tree_update_subtree(tree* u, path p, tree* t) {
  if (path_is_error_path(p) || u == NULL || t == NULL)
    return NULL;

  if (!path_is_nil(p) && tree_empty(t))
    return NULL;

  if (path_is_nil(p)) {
    tree_destroy(t);
    return u;
  }

  // continued on next slide
```

Implementierung updsubtree

```
  // continued from previous slide

  tree* s = NULL;
  if (path_is_left(p))
    s = tree_build(t->value,
                tree_update_subtree(
                    u, path_descend(p), t->left),
                t->right);
  else // path_is_right(p)
    s = tree_build(t->value, t->left,
                tree_update_subtree(
                    u, path_descend(p), t->right));

  tree_free(t);

  return s;
}
```

Implementierung destroy

```
void tree_destroy2(tree* t) {
  if (t == NULL)
    return;
  tree_destroy2(t->left);
  tree_free(t->left);
  tree_destroy2(t->right);
  tree_free(t->right);
}

void tree_destroy(tree* t) {
  tree_destroy2(t);
  tree_free(t);
}
```

Implementierung `preorder`

```
list* tree_get_preorder(tree* t) {
  list* l = list_init();
  if (!tree_empty(t)) {
    l = list_add(t->value, l);
    list* x = tree_get_preorder(t->left);
    l = list_append(l, x);
    list_destroy(x);
    x = tree_get_preorder(t->right);
    l = list_append(l, x);
    list_destroy(x);
  }
  return l;
}

list* tree_preorder(tree* t) {
  return tree_get_preorder(t);
}
```

Implementierung `inorder`

```
list* tree_get_inorder(tree* t) {
  list* l = list_init();
  if (!tree_empty(t)) {
    list* x = tree_get_inorder(t->left);
    l = list_append(l, x);
    list_destroy(x);
    l = list_add(t->value, l);
    x = tree_get_inorder(t->right);
    l = list_append(l, x);
    list_destroy(x);
  }
  return l;
}

list* tree_inorder(tree* t) {
  return tree_get_inorder(t);
}
```

Implementierung postorder

```
list* tree_get_postorder(tree* t) {
  list* l = list_init();
  if (!tree_empty(t)) {
    list* x = tree_get_postorder(t->left);
    l = list_append(l, x);
    list_destroy(x);
    x = tree_get_postorder(t->right);
    l = list_append(l, x);
    list_destroy(x);
    l = list_add(t->value, l);
  }
  return l;
}

list* tree_postorder(tree* t) {
  return tree_get_postorder(t);
}
```

Implementierung minnode

```
int tree_min_node(tree* t) {
  if (t == NULL || t->value == TREE_ERROR_ELEMENT)
    return INT_MAX;

  int min = t->value;
  int l = tree_min_node(t->left);
  if (l < min)
    min = l;
  int r = tree_min_node(t->right);
  if (r < min)
    min = r;
  return min;
}
```

Implementierung maxnode

```
int tree_max_node(tree* t) {
  if (t == NULL || t->value == TREE_ERROR_ELEMENT)
    return INT_MIN;

  int max = t->value;
  int l = tree_max_node(t->left);
  if (l > max)
    max = l;
  int r = tree_max_node(t->right);
  if (r > max)
    max = r;
  return max;
}
```

Implementierung `sorted` und `balanced`

```
int tree_sorted(tree* t) {
  if (t == NULL)
    return 0;

  if (tree_empty(t))
    return 1;

  return tree_sorted(t->left) &&
         tree_sorted(t->right) &&
         (tree_max_node(t->left) <= t->value) &&
         (t->value <= tree_min_node(t->right));
}

int tree_balanced(tree* t) {
  return (tree_height(t) - tree_minheight(t)) <= 1;
}
```

Implementierung print

```
void tree_print(tree* t) {
  if (t == NULL) {
    printf("errortree");
    return;
  }
  if (tree_empty(t)) {
    printf("init");
    return;
  }
  printf("tree(%i,", t->value);
  tree_print(t->left);
  printf(",");
  tree_print(t->right);
  printf(")");
}
```

Implementierung main

```
int main() {
  tree* t1 =
      tree_build(7,
          tree_build(3, tree_init(), tree_init()),
          tree_build(10, tree_init(), tree_init()));
  tree* t2 =
      tree_build(8,
          tree_build(4, tree_init(), tree_init()),
          tree_build(11, tree_init(), tree_init()));
  tree* t = tree_build(9, t1, t2);
  tree_println(t);

  int c = tree_count_nodes(t);
  printf("number␣of␣tree␣nodes:␣%i\n", c);

  // continued on next slide
```

Implementierung `main`

```
  // continued from previous slide

  b = tree_isin(4, t);
  printf("tree_isin(4,␣t)␣=␣%i\n", b);

  b = tree_isin(11, t);
  printf("tree_isin(11,␣t)␣=␣%i\n", b);

  b = tree_isin(12, t);
  printf("tree_isin(12,␣t)␣=␣%i\n", b);

  b = tree_isin(12, NULL);
  printf("tree_isin(12,␣NULL)␣=␣%i\n", b);

  tree_destroy(t);
}
```

Exemplarische Fragen zur Lernkontrolle

1. Erläutern Sie die Definition eines Baums!
2. Wozu dient ein Baum?
3. Welche Operation bietet ein Baum typischerweise an?
4. Spezifizieren Sie alle grundlegenden Operationen des Baums!
5. Erläutern Sie die Preorder-, die Inorder sowie die Postorder-Traversierung eines Binärbaums!
6. Erläutern Sie die Implementierung eines Baums als verkettete Datenstruktur!
7. Definieren Sie einen binären Suchbaum sowie einen Heap!
8. Wann ist ein Baum balanciert und wann ist er vollständig?
9. Welche Höhe hat ein balancierter Baum?
10. Wie viele Knoten enthält ein vollständiger Binärbaum der Höhe h?

Vielen Dank für Ihre Aufmerksamkeit!

Univ.-Prof. Dr.-Ing. Gero Mühl

gero.muehl@uni-rostock.de
https://www.ava.uni-rostock.de