Informatik I: Einführung in die Programmierung

10. Bäume

Albert-Ludwigs-Universität Freiburg

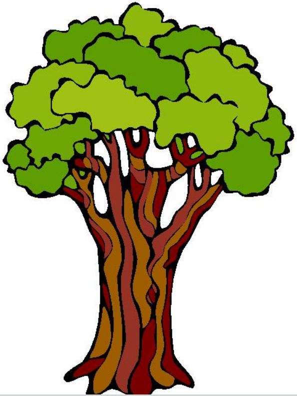
Prof. Dr. Peter Thiemann

26. November 2025

Der Baum

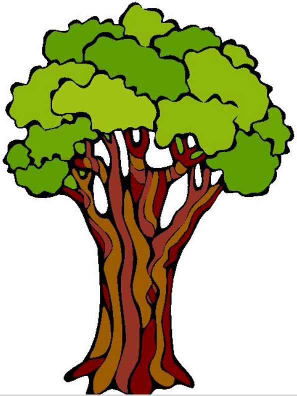
Bäume in der Informatik

- Bäume sind in der Informatik allgegenwärtig.



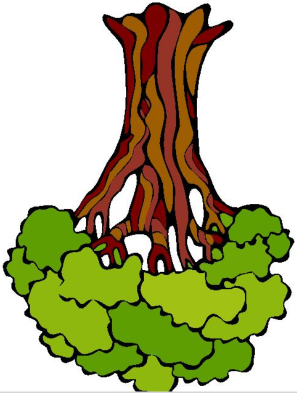
Bäume in der Informatik

- Bäume sind in der Informatik allgegenwärtig.
- Gezeichnet werden sie meistens mit der Wurzel nach oben!



Bäume in der Informatik

- Bäume sind in der Informatik allgegenwärtig.
- Gezeichnet werden sie meistens mit der Wurzel nach oben!



Definition

Bäume in der Informatik - Definition

Induktive Definition

- Gegeben eine Menge K von Knoten.

Bäume in der Informatik - Definition

Induktive Definition

- Gegeben eine Menge K von Knoten.
- Der leere Baum $\square$ ist ein Baum (ohne Knoten).

Bäume in der Informatik - Definition

Induktive Definition

- Gegeben eine Menge K von Knoten.
- Der leere Baum $\square$ ist ein Baum (ohne Knoten).
- Wenn $t_1, \ldots, t_n$, $n \geq 0$ disjunkte Bäume sind und k ein Knoten, der nicht in $t_1, \ldots, t_n$ vorkommt, dann ist auch die Struktur bestehend aus der Wurzel k mit zugeordneten Teilbäumen $t_1, \ldots, t_n$ ein Baum.

- Gegeben eine Menge K von Knoten.
- Der leere Baum $\square$ ist ein Baum (ohne Knoten).
- Wenn $t_1, \ldots, t_n$, $n \geq 0$ disjunkte Bäume sind und k ein Knoten, der nicht in $t_1, \ldots, t_n$ vorkommt, dann ist auch die Struktur bestehend aus der Wurzel k mit zugeordneten Teilbäumen $t_1, \ldots, t_n$ ein Baum.
- Nichts sonst ist ein Baum.

Bäume in der Informatik - Definition

Induktive Definition

- Gegeben eine Menge K von Knoten.
- Der leere Baum $\square$ ist ein Baum (ohne Knoten).
- Wenn $t_1, \dots, t_n$, $n \geq 0$ disjunkte Bäume sind und k ein Knoten, der nicht in $t_1, \dots, t_n$ vorkommt, dann ist auch die Struktur bestehend aus der Wurzel k mit zugeordneten Teilbäumen $t_1, \dots, t_n$ ein Baum.
- Nichts sonst ist ein Baum.
- Bildlich:

Bäume in der Informatik - Definition

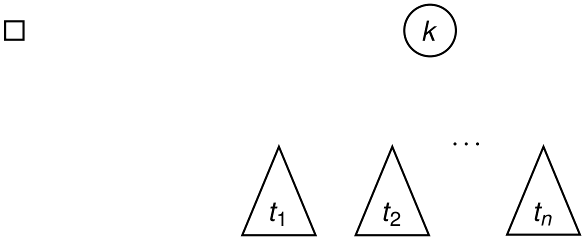
Induktive Definition

- Gegeben eine Menge K von Knoten.
- Der leere Baum $\square$ ist ein Baum (ohne Knoten).
- Wenn $t_1, \ldots, t_n$, $n \geq 0$ disjunkte Bäume sind und k ein Knoten, der nicht in $t_1, \ldots, t_n$ vorkommt, dann ist auch die Struktur bestehend aus der Wurzel k mit zugeordneten Teilbäumen $t_1, \ldots, t_n$ ein Baum.
- Nichts sonst ist ein Baum.
- Bildlich: $\square$

Bäume in der Informatik - Definition

Induktive Definition

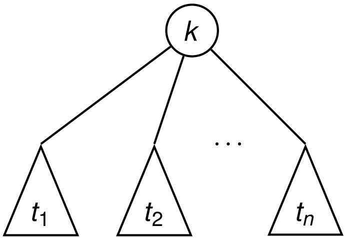
- Gegeben eine Menge K von Knoten.
- Der leere Baum $\square$ ist ein Baum (ohne Knoten).
- Wenn $t_1, \dots, t_n$, $n \geq 0$ disjunkte Bäume sind und k ein Knoten, der nicht in $t_1, \dots, t_n$ vorkommt, dann ist auch die Struktur bestehend aus der Wurzel k mit zugeordneten Teilbäumen $t_1, \dots, t_n$ ein Baum.
- Nichts sonst ist ein Baum.
- Bildlich:



Bäume in der Informatik - Definition

Induktive Definition

- Gegeben eine Menge K von Knoten.
- Der leere Baum $\square$ ist ein Baum (ohne Knoten).
- Wenn $t_1, \ldots, t_n$, $n \geq 0$ disjunkte Bäume sind und k ein Knoten, der nicht in $t_1, \ldots, t_n$ vorkommt, dann ist auch die Struktur bestehend aus der Wurzel k mit zugeordneten Teilbäumen $t_1, \ldots, t_n$ ein Baum.
- Nichts sonst ist ein Baum.
- Bildlich:



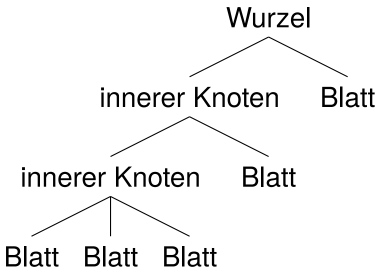
Terminologie

Terminologie I

- Baumknoten, denen keine Teilbäume zugeordnet sind, heißen Blätter.

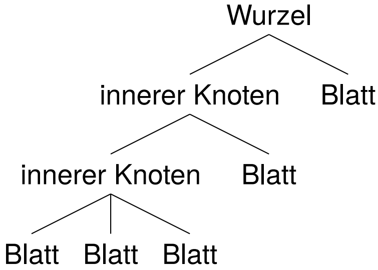
Terminologie I

- Baumknoten, denen keine Teilbäume zugeordnet sind, heißen Blätter.
- Baumknoten, die keine Blätter sind, heißen innere Knoten.



Terminologie I

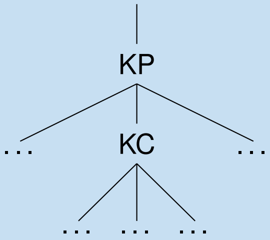
- Baumknoten, denen keine Teilbäume zugeordnet sind, heißen Blätter.
- Baumknoten, die keine Blätter sind, heißen innere Knoten.



- Die Wurzel kann also ein Blatt sein (keine weiteren Teilbäume) oder ein innerer Knoten.

Terminologie II

Eltern und Kinder



Wenn KP ein Knoten und KC die Wurzel eines zugeordneten Teilbaums ist, dann gilt:

- KP ist Elternknoten von KC (höchstens einer),
- Der Elternknoten von KP, dessen Elternknoten usw. sind Vorgänger von KC.
- KC ist Kind von KP.
- Kinder von KC, deren Kinder, usw. sind Nachfolger von KP.

Terminologie III

Markierte Bäume

- Bäume sind oft markiert. Die Markierung weist jedem Knoten eine Marke zu.
- Formal: Wenn K die Knotenmenge eines Baums ist und M eine Menge von Marken, dann ist die Markierung eine Abbildung $\mu : K \to M$.

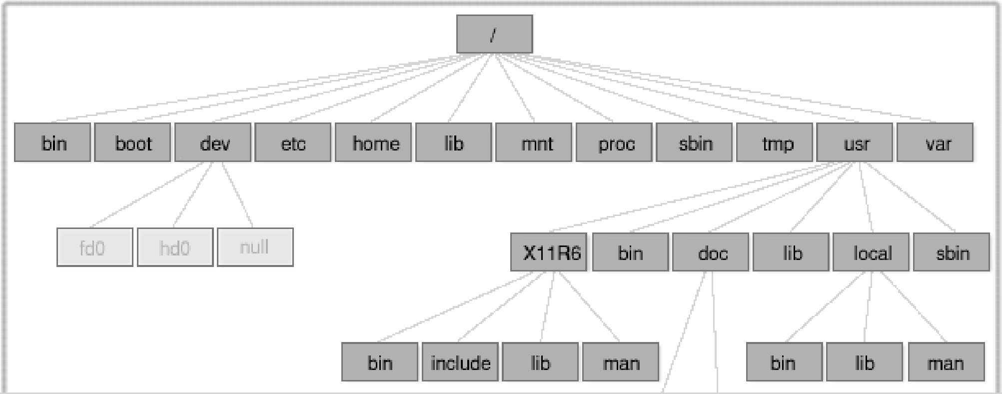
Beispiele

Beispiel: Verzeichnisbaum

In vielen Betriebssystemen ist die Verzeichnisstruktur im Wesentlichen baumartig.
Knotenmarkierung: Dateiname

Beispiel: Verzeichnisbaum

In vielen Betriebssystemen ist die Verzeichnisstruktur im Wesentlichen baumartig.
Knotenmarkierung: Dateiname



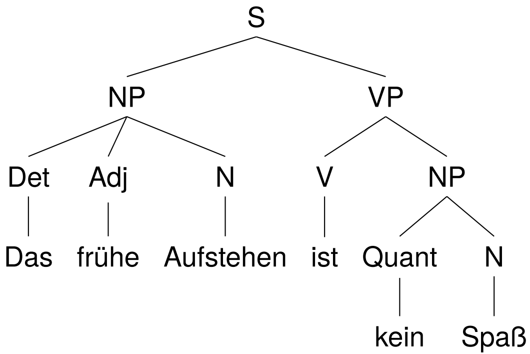
Der Baum
Definition
Terminologie
Beispiele
Binärbäume
Suchbäume
Zusammenfassung

Beispiel: Syntaxbaum

Wenn die Struktur einer Sprache mit Hilfe einer formalen Grammatik spezifiziert ist, dann kann der Satzaufbau durch Syntaxbäume beschrieben werden.

Beispiel: Syntaxbaum

Wenn die Struktur einer Sprache mit Hilfe einer formalen Grammatik spezifiziert ist, dann kann der Satzaufbau durch Syntaxbäume beschrieben werden.



Beispiel: Ausdrucksbaum

- Bäume können Ausdrücke so darstellen, dass ihre Auswertung eindeutig durchführbar ist, ohne dass Klammern notwendig sind.

Beispiel: Ausdrucksbaum

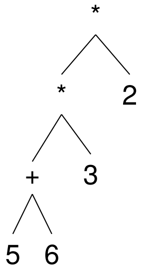
- Bäume können Ausdrücke so darstellen, dass ihre Auswertung eindeutig durchführbar ist, ohne dass Klammern notwendig sind.
- Beispiel: $(5+6)*3*2$

Beispiel: Ausdrucksbaum

- Bäume können Ausdrücke so darstellen, dass ihre Auswertung eindeutig durchführbar ist, ohne dass Klammern notwendig sind.
- Beispiel: $(5+6)*3*2$
- Entspricht: $((5+6)*3)*2$

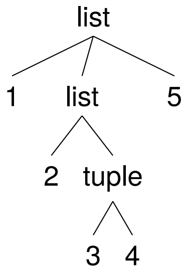
Beispiel: Ausdrucksbaum

- Bäume können Ausdrücke so darstellen, dass ihre Auswertung eindeutig durchführbar ist, ohne dass Klammern notwendig sind.
- Beispiel: $(5+6)*3*2$
- Entspricht: $((5+6)*3)*2$
- Baumdarstellung (AST, abstract syntax tree) — Operatoren als Markierung innerer Knoten, Zahlen als Markierung der Blätter:



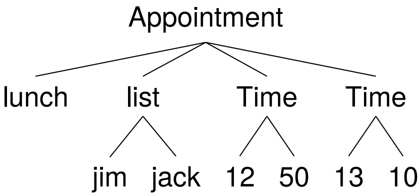
Beispiel: Listen und Tupel als Bäume

- Jede Liste und jedes Tupel kann als Baum angesehen werden, bei dem der Typ die Knotenmarkierung ist und die Elemente die Teilbäume sind.
- Beispiel: `[1, [2, (3, 4)], 5]`



Beispiel: Appointments als Bäume

- Jede Objektstruktur kann als Baum angesehen werden, bei dem der Typ die Knotenmarkierung ist und die Attribute die Teilbäume.
- Oft werden noch die Attributnamen an die Kanten geschrieben.



Binärbäume

Der Binärbaum

- Der Binärbaum ist ein Spezialfall eines Baumes.
- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Für viele Anwendungsfälle angemessen.
- Funktionen über solchen Bäumen sind induktiv definierbar.

Repräsentation

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Der leere Baum wird durch `None` repräsentiert.

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Der leere Baum wird durch `None` repräsentiert.
- Jeder andere Knoten wird durch ein `Node`-Objekt repräsentiert.

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Der leere Baum wird durch `None` repräsentiert.
- Jeder andere Knoten wird durch ein `Node`-Objekt repräsentiert.
  - Das Attribut `mark` enthält die Markierung.

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Der leere Baum wird durch `None` repräsentiert.
- Jeder andere Knoten wird durch ein `Node`-Objekt repräsentiert.
  - Das Attribut `mark` enthält die Markierung.
  - Das Attribut `left` enthält den linken Teilbaum.

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Der leere Baum wird durch `None` repräsentiert.
- Jeder andere Knoten wird durch ein `Node`-Objekt repräsentiert.
  - Das Attribut `mark` enthält die Markierung.
  - Das Attribut `left` enthält den linken Teilbaum.
  - Das Attribut `right` enthält den rechten Teilbaum.

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Der leere Baum wird durch `None` repräsentiert.
- Jeder andere Knoten wird durch ein `Node`-Objekt repräsentiert.
  - Das Attribut `mark` enthält die Markierung.
  - Das Attribut `left` enthält den linken Teilbaum.
  - Das Attribut `right` enthält den rechten Teilbaum.
- Alternative: `BTree = None | Node` abgekürzt durch `Optional[Node]`

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Der leere Baum wird durch `None` repräsentiert.
- Jeder andere Knoten wird durch ein `Node`-Objekt repräsentiert.
  - Das Attribut `mark` enthält die Markierung.
  - Das Attribut `left` enthält den linken Teilbaum.
  - Das Attribut `right` enthält den rechten Teilbaum.
- Alternative: `BTree = None | Node` abgekürzt durch `Optional[Node]`
- Beispiele für Binärbäume

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Der leere Baum wird durch `None` repräsentiert.
- Jeder andere Knoten wird durch ein `Node`-Objekt repräsentiert.
  - Das Attribut `mark` enthält die Markierung.
  - Das Attribut `left` enthält den linken Teilbaum.
  - Das Attribut `right` enthält den rechten Teilbaum.
- Alternative: `BTree = None | Node` abgekürzt durch `Optional[Node]`
- Beispiele für Binärbäume
  - Ein einziger Knoten mit der Markierung 8:
    `Node (8, None, None)`

Binärbäume durch Objekte repräsentieren

- Ein Binärbaum ist entweder leer oder besteht aus einem (Wurzel-) Knoten und zwei Teilbäumen.
- Der leere Baum wird durch `None` repräsentiert.
- Jeder andere Knoten wird durch ein `Node`-Objekt repräsentiert.
  - Das Attribut `mark` enthält die Markierung.
  - Das Attribut `left` enthält den linken Teilbaum.
  - Das Attribut `right` enthält den rechten Teilbaum.
- Alternative: `BTree = None | Node` abgekürzt durch `Optional[Node]`
- Beispiele für Binärbäume
  - Ein einziger Knoten mit der Markierung 8:
    `Node (8, None, None)`
  - Wurzel '+', linker Teilbaum mit Blatt 5, rechter Teilbaum mit Blatt 6:
    `Node('+', Node(5, None, None), Node(6, None, None))`

Klasse für Baumknoten

```
from typing import Optional
@dataclass
class Node[T]:
    mark : T
    left : 'BTree[T]' = None
    right: 'BTree[T]' = None
type BTree[T] = Optional[Node[T]]
```

Bemerkung zu den Typannotationen

- `Node[T]`: Typ einer *generischen Klasse*
  `T` ist der Typ der Markierung des Baums
- `Optional[t]`: entweder `t` oder `None` (aber nichts anderes)
- Der Typ `Node` existiert erst **nach** Ausführung der `class`-Anweisung. Python ersetzt den String `'Node[T]'` in der Typannotation rückwirkend durch den Typ `Node[T]`.

Klasse für Baumknoten

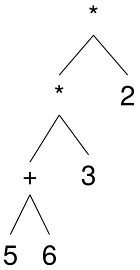
```
from typing import Optional
@dataclass
class Node[T]:
    mark : T
    left : 'BTree[T]' = None
    right: 'BTree[T]' = None
type BTree[T] = Optional[Node[T]]
```

Bemerkung zur Initialisierung

- `left : ... = None` initialisiert das Attribute `left`, falls es beim Aufruf von `Node` nicht angegeben wird.
- `Node` kann mit 1, 2 oder 3 Argumenten aufgerufen werden; die fehlenden werden durch `None` ersetzt.

Beispiel

Beispiel: Der Ausdrucksbaum



Darstellung mit `Node` Objekten:

```
b = Node('*', Node('*', Node('+', Node(5),
                                  Node(6)),
                        Node(3)),
              Node(2))
```

Funktionen auf Bäumen

Drucken von Bäumen

Aufgabe

Transformiere einen Baum mit beliebiger Markierung in einen String.

Drucken von Bäumen

Aufgabe

Transformiere einen Baum mit beliebiger Markierung in einen String.

Signatur

```
def tree_str[T](tree : BTree[T]) -> str:
```

Drucken von Bäumen

Präzisierung

- Jeder Knoten des Baums muss in einen String transformiert werden.

Drucken von Bäumen

Präzisierung

- Jeder Knoten des Baums muss in einen String transformiert werden.
- `BTree[T] = Optional[Node[T]] = None | Node[T]` ist ein Uniontyp (Alternative).

Drucken von Bäumen

Präzisierung

- Jeder Knoten des Baums muss in einen String transformiert werden.
- `BTree[T] = Optional[Node[T]] = None | Node[T]` ist ein Uniontyp (Alternative).

⇒ Pattern matching

Drucken von Bäumen

Präzisierung

- Jeder Knoten des Baums muss in einen String transformiert werden.
- `BTree[T] = Optional[Node[T]] = None | Node[T]` ist ein Uniontyp (Alternative).

⇒ Pattern matching

- Zusätzliches Problem:
  `Node`-Objekte enthalten selbst Attribute vom Typ `BTree[T]`.

Drucken von Bäumen

Präzisierung

- Jeder Knoten des Baums muss in einen String transformiert werden.
- `BTree[T] = Optional[Node[T]] = None | Node[T]` ist ein Uniontyp (Alternative).

⇒ Pattern matching

- Zusätzliches Problem:
  `Node`-Objekte enthalten selbst Attribute vom Typ `BTree[T]`.
- Abhilfe **Wunschdenken**:
  nehme an, dass `tree_str` auf den Teilbäumen schon das Problem löst!

Drucken von Bäumen

Präzisierung

- Jeder Knoten des Baums muss in einen String transformiert werden.
- `BTree[T] = Optional[Node[T]] = None | Node[T]` ist ein Uniontyp (Alternative).

⇒ Pattern matching

- Zusätzliches Problem:
  `Node`-Objekte enthalten selbst Attribute vom Typ `BTree[T]`.
- Abhilfe **Wunschdenken**:
  nehme an, dass `tree_str` auf den Teilbäumen schon das Problem löst!
- D.h. verwende die Funktion in ihrer eigenen Definition (Rekursion)!

Drucken von Bäumen

Funktionsgerüst (Alternative)

```
def tree_str[T](tree : BTree[T]) -> str:
    match tree:
        case None:
            return "fill in"
        case Node (mark, left, right):
            return "fill in"
```

Drucken von Bäumen

Funktionsgerüst (Alternative + Induktion)

```
def tree_str[T](tree : BTree[T]) -> str:
    match tree:
        case None:
            return "fill in"
        case Node (mark, left, right):
            l_str = tree_str(left)    # assume tree_str on left
            r_str = tree_str(right)   # assume tree_str on right
            return "fill in"
```

- `Node` Objekte enthalten selbst wieder `Node` Objekte (oder `None`) in den Attributen `left` und `right`.

Drucken von Bäumen

Funktionsgerüst (Alternative + Induktion)

```
def tree_str[T](tree : BTree[T]) -> str:
    match tree:
        case None:
            return "fill in"
        case Node (mark, left, right):
            l_str = tree_str(left)     # assume tree_str on left
            r_str = tree_str(right)    # assume tree_str on right
            return "fill in"
```

- `Node` Objekte enthalten selbst wieder `Node` Objekte (oder `None`) in den Attributen `left` und `right`.
- Zum Ausdrucken eines `Node` Objekts müssen auch die `Node` Objekte in den Attributen ausgedruckt werden.

Drucken von Bäumen

Funktionsgerüst (Alternative + Induktion)

```
def tree_str[T](tree : BTree[T]) -> str:
    match tree:
        case None:
            return "fill in"
        case Node (mark, left, right):
            l_str = tree_str(left)     # assume tree_str on left
            r_str = tree_str(right)    # assume tree_str on right
            return "fill in"
```

- `Node` Objekte enthalten selbst wieder `Node` Objekte (oder `None`) in den Attributen `left` und `right`.
- Zum Ausdrucken eines `Node` Objekts müssen auch die `Node` Objekte in den Attributen ausgedruckt werden.
- `tree_str` ist rekursiv, denn es wird in seiner eigenen Definition aufgerufen!

Drucken von Bäumen erfolgt rekursiv

- Die rekursiven Aufrufe `tree_str (left)` und `tree_str (right)` erfolgen nur auf den Kindern des Knotens.
- Ergibt sich zwangsläufig aus der induktiven Definition!
- Rekursive Aufrufe auf den Teilbäumen sind Teil des Funktionsgerüsts, sobald eine baumartige Struktur bearbeitet werden soll.

- Die Alternative "`case None`" ergibt sich zwangsläufig aus dem Typ `tree:Optional[Node]`: `tree` ist entweder `None` oder eine `Node`-Instanz.

- Alle Funktionen auf Binärbäumen verwenden dieses Gerüst.

Drucken von Bäumen

Funktionsdefinition

```
def tree_str[T](tree : BTree[T]) -> str:
    match tree:
        case None:
            return "None"
        case Node (mark, left, right):
            l_str = tree_str(left)     # assume tree_str on left
            r_str = tree_str(right)    # assume tree_str on right
            return ("Node("
                + repr(m) + ", "
                + l_str + ", "
                + r_str + ")")
```

Baumeigenschaften

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

induktiv definiert

- Die Tiefe eines Knotens k (Abstand zur Wurzel) ist

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

induktiv definiert

- Die Tiefe eines Knotens *k* (Abstand zur Wurzel) ist
  - 0, falls *k* die Wurzel ist,

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

induktiv definiert

- Die Tiefe eines Knotens k (Abstand zur Wurzel) ist
  - 0, falls k die Wurzel ist,
  - $t + 1$, wenn t die Tiefe des Elternknotens ist.

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

induktiv definiert

- Die Tiefe eines Knotens k (Abstand zur Wurzel) ist
  - 0, falls k die Wurzel ist,
  - $t + 1$, wenn t die Tiefe des Elternknotens ist.
- Die Höhe eines Baumes ist die maximale Tiefe über alle Blätter:

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

induktiv definiert

- Die Tiefe eines Knotens k (Abstand zur Wurzel) ist
  - 0, falls k die Wurzel ist,
  - $t + 1$, wenn t die Tiefe des Elternknotens ist.
- Die Höhe eines Baumes ist die maximale Tiefe über alle Blätter:
  - -1 für den leeren Baum,

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

induktiv definiert

- Die Tiefe eines Knotens k (Abstand zur Wurzel) ist
  - 0, falls k die Wurzel ist,
  - $t + 1$, wenn t die Tiefe des Elternknotens ist.
- Die Höhe eines Baumes ist die maximale Tiefe über alle Blätter:
  - -1 für den leeren Baum,
  - $h + 1$, wenn h die maximale Höhe aller der Wurzel zugeordneten Teilbäume ist.

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

induktiv definiert

- Die Tiefe eines Knotens k (Abstand zur Wurzel) ist
  - 0, falls k die Wurzel ist,
  - $t+1$, wenn t die Tiefe des Elternknotens ist.
- Die Höhe eines Baumes ist die maximale Tiefe über alle Blätter:
  - -1 für den leeren Baum,
  - $h+1$, wenn h die maximale Höhe aller der Wurzel zugeordneten Teilbäume ist.
- Die Größe eines Baumes ist die Anzahl seiner Knoten.

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

induktiv definiert

- Die Tiefe eines Knotens k (Abstand zur Wurzel) ist
  - 0, falls k die Wurzel ist,
  - $t+1$, wenn t die Tiefe des Elternknotens ist.
- Die Höhe eines Baumes ist die maximale Tiefe über alle Blätter:
  - -1 für den leeren Baum,
  - $h+1$, wenn h die maximale Höhe aller der Wurzel zugeordneten Teilbäume ist.
- Die Größe eines Baumes ist die Anzahl seiner Knoten.
  - 0 für den leeren Baum,

Tiefe von Knoten, Höhe und Größe von (Binär-)Bäumen

induktiv definiert

- Die Tiefe eines Knotens k (Abstand zur Wurzel) ist
  - 0, falls k die Wurzel ist,
  - $t+1$, wenn t die Tiefe des Elternknotens ist.
- Die Höhe eines Baumes ist die maximale Tiefe über alle Blätter:
  - -1 für den leeren Baum,
  - $h+1$, wenn h die maximale Höhe aller der Wurzel zugeordneten Teilbäume ist.
- Die Größe eines Baumes ist die Anzahl seiner Knoten.
  - 0 für den leeren Baum,
  - $s+1$, wenn s die Summe der Größen der Teilbäume ist.

Induktive Definition von Höhe und Größe von Binärbäumen

$$
\mathit{height}(\mathit{tree}) = \begin{cases} -1, & \text{if } \mathit{tree} \text{ is empty} \\ 1 + \max(\quad \mathit{height}(\mathit{tree.left}), & \\ \qquad\qquad\quad \mathit{height}(\mathit{tree.right})), & \text{otherwise.} \end{cases}
$$

Induktive Definition von Höhe und Größe von Binärbäumen

$$
\mathit{height}(\mathit{tree}) = \begin{cases} -1, & \text{if } \mathit{tree} \text{ is empty} \\ 1 + \max(\quad \mathit{height}(\mathit{tree.left}), & \\ \qquad\qquad \mathit{height}(\mathit{tree.right})), & \text{otherwise.} \end{cases}
$$

$$
\mathit{size}(\mathit{tree}) = \begin{cases} 0, & \text{if } \mathit{tree} \text{ is empty;} \\ 1 \quad + \mathit{size}(\mathit{tree.left}) & \\ \qquad + \mathit{size}(\mathit{tree.right})), & \text{otherwise.} \end{cases}
$$

Funktionen für Höhe und Größe

```
def height[T](tree : BTree[T]) -> int:
    match tree:
        case None:
            return -1
        case Node (m, l, r):
            return(max(height(l), height(r)) + 1)

def size[T](tree : BTree[T]) -> int:
    match tree:
        case None:
            return 0
        case Node (m, l, r):
            return(size(l) + size(r) + 1)
```


),

Traversierung

Traversierung von Bäumen

- Oft sollen alle Knoten eines Baumes besucht und bearbeitet werden.

Traversierung von Bäumen

- Oft sollen alle Knoten eines Baumes besucht und bearbeitet werden.
- 3 Vorgehensweisen (Traversierungen) sind üblich:

Traversierung von Bäumen

- Oft sollen alle Knoten eines Baumes besucht und bearbeitet werden.
- 3 Vorgehensweisen (Traversierungen) sind üblich:
  - Pre-Order (Hauptreihenfolge): Bearbeite zuerst den Knoten selbst, dann besuche den linken, danach den rechten Teilbaum

Traversierung von Bäumen

- Oft sollen alle Knoten eines Baumes besucht und bearbeitet werden.
- 3 Vorgehensweisen (Traversierungen) sind üblich:
  - Pre-Order (Hauptreihenfolge): Bearbeite zuerst den Knoten selbst, dann besuche den linken, danach den rechten Teilbaum
  - Post-Order (Nebenreihenfolge): Besuche zuerst den linken, danach den rechten Teilbaum, zum Schluss bearbeite den Knoten selbst

Traversierung von Bäumen

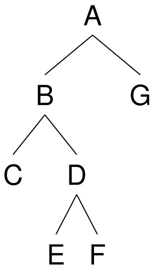
- Oft sollen alle Knoten eines Baumes besucht und bearbeitet werden.
- 3 Vorgehensweisen (Traversierungen) sind üblich:
    - Pre-Order (Hauptreihenfolge): Bearbeite zuerst den Knoten selbst, dann besuche den linken, danach den rechten Teilbaum
    - Post-Order (Nebenreihenfolge): Besuche zuerst den linken, danach den rechten Teilbaum, zum Schluss bearbeite den Knoten selbst
    - In-Order (symmetrische Reihenfolge): Besuche zuerst den linken Teilbaum, dann bearbeite den Knoten selbst, danach besuche den rechten Teilbaum

Traversierung von Bäumen

- Oft sollen alle Knoten eines Baumes besucht und bearbeitet werden.
- 3 Vorgehensweisen (Traversierungen) sind üblich:
  - Pre-Order (Hauptreihenfolge): Bearbeite zuerst den Knoten selbst, dann besuche den linken, danach den rechten Teilbaum
  - Post-Order (Nebenreihenfolge): Besuche zuerst den linken, danach den rechten Teilbaum, zum Schluss bearbeite den Knoten selbst
  - In-Order (symmetrische Reihenfolge): Besuche zuerst den linken Teilbaum, dann bearbeite den Knoten selbst, danach besuche den rechten Teilbaum
- Manchmal auch Reverse In-Order (anti-symmetrische Reihenfolge): Rechter Teilbaum, Knoten, dann linker Teilbaum

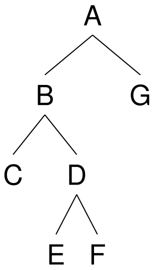
Pre-Order Ausgabe eines Baums

- Gebe den Baum *pre-order* aus



Pre-Order Ausgabe eines Baums

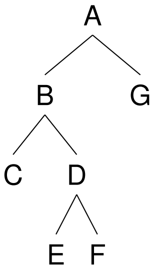
- Gebe den Baum *pre-order* aus



- Ausgabe: A

Pre-Order Ausgabe eines Baums

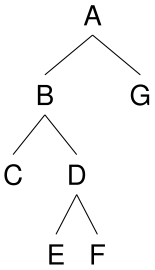
- Gebe den Baum *pre-order* aus



- Ausgabe: A

Pre-Order Ausgabe eines Baums

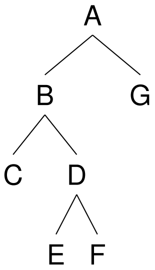
- Gebe den Baum *pre-order* aus



- Ausgabe: A B

Pre-Order Ausgabe eines Baums

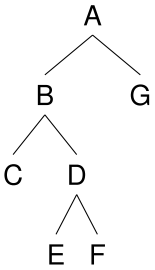
- Gebe den Baum *pre-order* aus



- Ausgabe: A B C

Pre-Order Ausgabe eines Baums

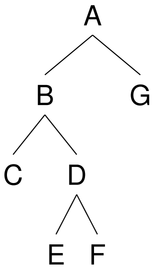
- Gebe den Baum *pre-order* aus



- Ausgabe: A B C D

Pre-Order Ausgabe eines Baums

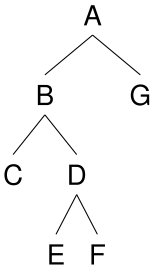
- Gebe den Baum *pre-order* aus



- Ausgabe: A B C D E

Pre-Order Ausgabe eines Baums

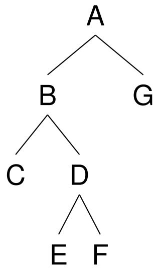
- Gebe den Baum *pre-order* aus



- Ausgabe: A B C D E F

Pre-Order Ausgabe eines Baums

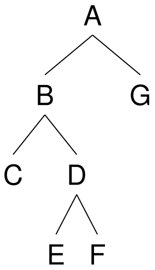
- Gebe den Baum *pre-order* aus



- Ausgabe: A B C D E F G

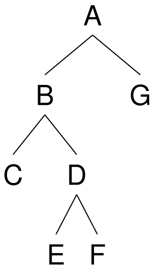
Post-Order Ausgabe eines Baums

- Gebe Baum *post-order* aus



Post-Order Ausgabe eines Baums

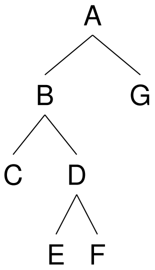
- Gebe Baum *post-order* aus



- Ausgabe: C

Post-Order Ausgabe eines Baums

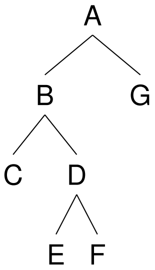
- Gebe Baum *post-order* aus



- Ausgabe: C

Post-Order Ausgabe eines Baums

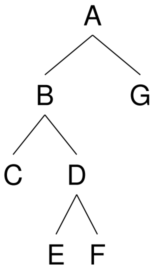
- Gebe Baum *post-order* aus



- Ausgabe: C E

Post-Order Ausgabe eines Baums

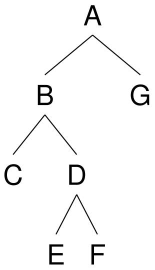
- Gebe Baum *post-order* aus



- Ausgabe: C E F

Post-Order Ausgabe eines Baums

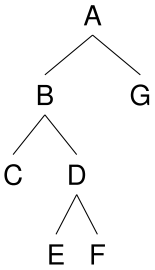
- Gebe Baum *post-order* aus



- Ausgabe: C E F D

Post-Order Ausgabe eines Baums

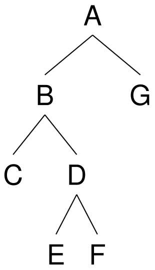
- Gebe Baum *post-order* aus



- Ausgabe: C E F D B

Post-Order Ausgabe eines Baums

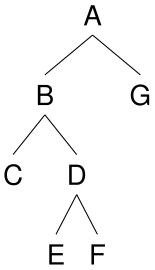
- Gebe Baum *post-order* aus



- Ausgabe: C E F D B G

Post-Order Ausgabe eines Baums

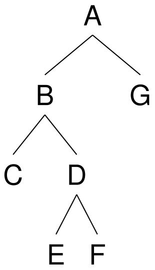
- Gebe Baum *post-order* aus



- Ausgabe: C E F D B G A

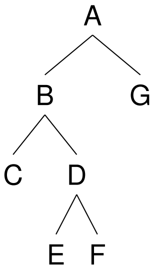
In-Order Ausgabe eines Baums

- Gebe Baum *in-order* aus.



In-Order Ausgabe eines Baums

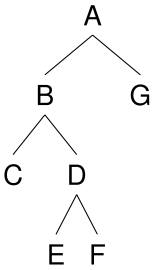
- Gebe Baum *in-order* aus.



- Ausgabe: C

In-Order Ausgabe eines Baums

- Gebe Baum *in-order* aus.

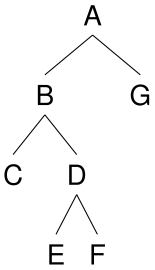


- Ausgabe: C

Der Baum
Binärbäume
Repräsentation
Beispiel
Funktionen auf Bäumen
Baumeigenschaften
Traversierung
Suchbäume
Zusammenfassung

In-Order Ausgabe eines Baums

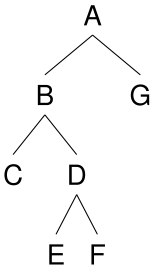
- Gebe Baum *in-order* aus.



- Ausgabe: C B

In-Order Ausgabe eines Baums

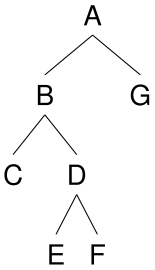
- Gebe Baum *in-order* aus.



- Ausgabe: C B E

In-Order Ausgabe eines Baums

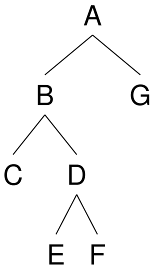
- Gebe Baum *in-order* aus.



- Ausgabe: C B E D

In-Order Ausgabe eines Baums

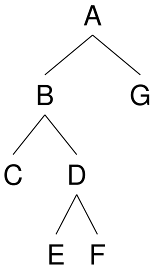
- Gebe Baum *in-order* aus.



- Ausgabe: C B E D F

In-Order Ausgabe eines Baums

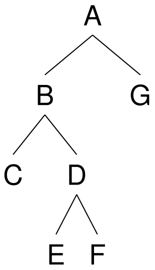
- Gebe Baum *in-order* aus.



- Ausgabe: C B E D F A

In-Order Ausgabe eines Baums

- Gebe Baum *in-order* aus.



- Ausgabe: C B E D F A G

Knotenliste in post-order

```
def postorder[T](tree : BTree[T]) -> list[T]:
    match tree:
        case None:
            return []
        case Node (m, l, r):
            return postorder(l) + postorder(r) + [m]

tree : Node[int|str] = Node('*', Node('+', Node(6), Node(5)),
                                 Node(1))
postorder(tree)
```

Knotenliste in post-order

```
def postorder[T](tree : BTree[T]) -> list[T]:
    match tree:
        case None:
            return []
        case Node (m, l, r):
            return postorder(l) + postorder(r) + [m]

tree : Node[int|str] = Node('*', Node('+', Node(6), Node(5)),
                              Node(1))
postorder(tree)
```

Die *post-order* Ausgabe eines Ausdrucks heißt auch umgekehrt polnische oder Postfix-Notation (HP-Taschenrechner, Programmiersprachen *Forth* und *PostScript*)

UPN

HP-35

Von Holger Weihe - Eigenes Werk, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=146664`

Forth

```
: DECADE 10 0 DO
         I .
         LOOP ;
```

PostScript

```
newpath
100 200 moveto
200 250 lineto
100 300 lineto
2 setlinewidth
stroke
```

Ergebnis: 

Suchbäume

Definition

Suchbäume

- *Suchbäume* dienen dazu, Objekte schnell aufzufinden.

Suchbäume

- *Suchbäume* dienen dazu, Objekte schnell aufzufinden.
- Ein Suchbaum ist ein binärer Baum, bei dem jeder Baumknoten k die Suchbaumeigenschaft erfüllt:

Suchbäume

- *Suchbäume* dienen dazu, Objekte schnell aufzufinden.
- Ein Suchbaum ist ein binärer Baum, bei dem jeder Baumknoten k die Suchbaumeigenschaft erfüllt:
  - Alle Markierungen im linken Teilbaum sind *kleiner* als die Markierung von k, alle Markierungen im rechten Teilbaum sind *größer*.

Suchbäume

- *Suchbäume* dienen dazu, Objekte schnell aufzufinden.
- Ein Suchbaum ist ein binärer Baum, bei dem jeder Baumknoten k die Suchbaumeigenschaft erfüllt:
  - Alle Markierungen im linken Teilbaum sind *kleiner* als die Markierung von k, alle Markierungen im rechten Teilbaum sind *größer*.
- Suchen nach einem Objekt m beginnend beim Knoten k:
  Vergleiche m mit Markierung des aktuellen Knotens k,

Suchbäume

- *Suchbäume* dienen dazu, Objekte schnell aufzufinden.
- Ein Suchbaum ist ein binärer Baum, bei dem jeder Baumknoten k die Suchbaumeigenschaft erfüllt:
  - Alle Markierungen im linken Teilbaum sind *kleiner* als die Markierung von k, alle Markierungen im rechten Teilbaum sind *größer*.
- Suchen nach einem Objekt m beginnend beim Knoten k:
  Vergleiche m mit Markierung des aktuellen Knotens k,
  - wenn gleich, stoppe und gebe True zurück,

Suchbäume

- *Suchbäume* dienen dazu, Objekte schnell aufzufinden.
- Ein Suchbaum ist ein binärer Baum, bei dem jeder Baumknoten k die Suchbaumeigenschaft erfüllt:
  - Alle Markierungen im linken Teilbaum sind *kleiner* als die Markierung von k, alle Markierungen im rechten Teilbaum sind *größer*.
- Suchen nach einem Objekt m beginnend beim Knoten k:
  Vergleiche m mit Markierung des aktuellen Knotens k,
  - wenn gleich, stoppe und gebe True zurück,
  - wenn m kleiner ist, suche im linken Teilbaum,

Suchbäume

- *Suchbäume* dienen dazu, Objekte schnell aufzufinden.
- Ein Suchbaum ist ein binärer Baum, bei dem jeder Baumknoten k die Suchbaumeigenschaft erfüllt:
  - Alle Markierungen im linken Teilbaum sind *kleiner* als die Markierung von k, alle Markierungen im rechten Teilbaum sind *größer*.
- Suchen nach einem Objekt m beginnend beim Knoten k:
  Vergleiche m mit Markierung des aktuellen Knotens k,
  - wenn gleich, stoppe und gebe True zurück,
  - wenn m kleiner ist, suche im linken Teilbaum,
  - wenn m größer ist, such im rechten Teilbaum.

Suchbäume

- *Suchbäume* dienen dazu, Objekte schnell aufzufinden.
- Ein Suchbaum ist ein binärer Baum, bei dem jeder Baumknoten k die Suchbaumeigenschaft erfüllt:
  - Alle Markierungen im linken Teilbaum sind *kleiner* als die Markierung von k, alle Markierungen im rechten Teilbaum sind *größer*.
- Suchen nach einem Objekt m beginnend beim Knoten k:
  Vergleiche m mit Markierung des aktuellen Knotens k,
  - wenn gleich, stoppe und gebe True zurück,
  - wenn m kleiner ist, suche im linken Teilbaum,
  - wenn m größer ist, such im rechten Teilbaum.
- Suchzeit ist proportional zur Höhe des Baums!
  Im besten Fall *logarithmisch in der Größe des Baums*.

Höhe und Größe eines Binärbaums

Lemma

Ist $h = h(t)$ die Höhe eines Binärbaums, so gilt für seine Größe $s(t) \leq 2^{h+1} - 1$.

Höhe und Größe eines Binärbaums

Lemma

Ist $h = h(t)$ die Höhe eines Binärbaums, so gilt für seine Größe $s(t) \leq 2^{h+1} - 1$.

Beweis (Induktion über den Baum t)

IA: Ist der Baum t leer, so ist seine Höhe -1 und seine Größe $0 \leq 2^0 - 1 = 0$.

IS: Besteht ein Baum t aus einem Knoten und zwei Teilbäumen l und r mit Höhen $h(l)$ und $h(r)$, so gilt nach IV $s(l) \leq 2^{h(l)+1} - 1$ und $s(r) \leq 2^{h(r)+1} - 1$.

Wegen $s(t) = 1 + s(l) + s(r)$ und $h(t) = 1 + \max(h(l), h(r))$ gilt

- $s(t) = 1 + s(l) + s(r) \leq 1 + (2^{h(l)+1} - 1) + (2^{h(r)+1} - 1) \leq 2 \cdot 2^{\max(h(l)+1, h(r)+1)} - 1 = 2^{h(t)+1} - 1$

Suche

Suche im Suchbaum

```
def search[T : (int, float, str)](tree: BTree[T], item: T) -> bool:
    match tree:
        case None:
            return False
        case Node(m, l, r) if m > item:
            return search(l, item)
        case Node(m, l, r) if m < item:
            return search(r, item)
        case _:     # m == item
            return True

# smaller values left, bigger values in right subtree
nums = Node(10, Node(5, Node(1), None),
                Node(15, Node(12), Node(20)))
print(search(nums, 12))
```

Aufbau

Aufbauen eines Suchbaums

- Aufruf `insert(tree, item)` für das Einsortieren von `item` in `tree`

Aufbauen eines Suchbaums

- Aufruf `insert(tree, item)` für das Einsortieren von `item` in `tree`
- Ist `tree` leer, so wird der Knoten `Node(item)` zurückgegeben.

Aufbauen eines Suchbaums

- Aufruf `insert(tree, item)` für das Einsortieren von `item` in `tree`
- Ist `tree` leer, so wird der Knoten `Node(item)` zurückgegeben.
- Wenn die Markierung `tree.mark` größer als `item` ist, wird `item` in den linken Teilbaum eingesetzt und der Baum rekonstruiert (das erhält die Suchbaumeigenschaft!).

Aufbauen eines Suchbaums

- Aufruf `insert(tree, item)` für das Einsortieren von `item` in `tree`
- Ist `tree` leer, so wird der Knoten `Node(item)` zurückgegeben.
- Wenn die Markierung `tree.mark` größer als `item` ist, wird `item` in den linken Teilbaum eingesetzt und der Baum rekonstruiert (das erhält die Suchbaumeigenschaft!).
- Falls `tree.mark` kleiner als `item` ist, entsprechend.

Aufbauen eines Suchbaums

- Aufruf `insert(tree, item)` für das Einsortieren von `item` in `tree`
- Ist `tree` leer, so wird der Knoten `Node(item)` zurückgegeben.
- Wenn die Markierung `tree.mark` größer als `item` ist, wird `item` in den linken Teilbaum eingesetzt und der Baum rekonstruiert (das erhält die Suchbaumeigenschaft!).
- Falls `tree.mark` kleiner als `item` ist, entsprechend.
- Falls `tree.mark == item` ist nichts zu tun!

Suchbaumaufbau

```
def insert[T : (str, int, float)](
        tree: BTree[T], item: T
        ) -> Node[T]:
    match tree:
        case None:
            return Node(item)
        case Node(m, l, r) if item < m:
            return Node(m, insert(l, item), r)
        case Node(m, l, r) if m < item:
            return Node(m, l, insert(r, item))
        case _:  # m == item
            return tree
```

Suchbaumaufbau

```
def insertall[T : (str, int, float)](
              tree : BTree[T],
              lst  : list[T]
             ) -> BTree[T]:
    for key in lst:
        tree = insert(tree, key)
    return tree

bst = insertall(None, [10, 15, 20, 12, 5, 1])
```

Zusammenfassung

Zusammenfassung

- Der Baum ist eine Struktur, die in der Informatik allgegenwärtig ist.
- Operationen über Bäumen lassen sich einfach als rekursive Funktionen implementieren.
- In einem Binärbaum besitzt jeder Knoten genau zwei Teilbäume.
- Es gibt drei Hauptarten der Traversierung von Binärbäumen: pre-order, post-order, in-order.
- Suchbäume sind Binärbäume, die die Suchbaumeigenschaft besitzen, d.h. im linken Teilbaum sind nur kleinere, im rechten nur größere Markierungen als an der Wurzel.
- Das Suchen und Einfügen kann durch einfache rekursive Funktionen realisiert werden. Sortierte Ausgabe ist auch sehr einfach!