

# Informatik I

## 22. Binäre Bäume

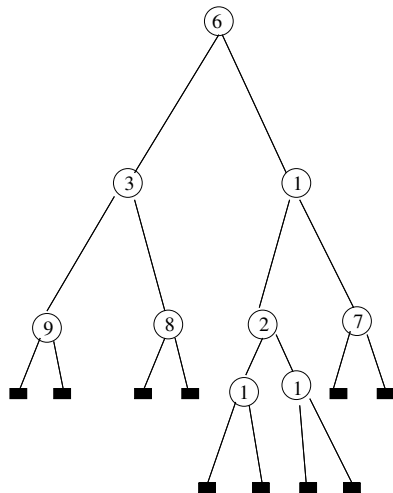
Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

3. Februar 2011

Nach verlinkten Listen und einigen eingebauten Typen werden wir jetzt eine weitere Datenstruktur betrachten: **binäre Bäume**. Diese dienen wie verlinkte Listen, Python-Listen und Tupel dazu, andere Elemente aufzubewahren.

# Beispiel



Informatik I

Jan-Georg  
Smaus


Die Klasse  
Tree

Suchbäume


Abstrakte  
Datentypen

Zusammen-  
fassung


# Erklärungen zu binären Bäumen

- Analog zur leeren verlinkten Liste gibt es den **leeren Baum**, im Bild so dargestellt: .
- Ansonsten besteht ein binärer Baum aus einer **Knotenmarkierung (Label)** und einem **linken** und einem **rechten** Teilbaum.

# Erklärungen zu binären Bäumen

- Analog zur leeren verlinkten Liste gibt es den **leeren Baum**, im Bild so dargestellt: .
- Ansonsten besteht ein binärer Baum aus einer **Knotenmarkierung (Label)** und einem **linken** und einem **rechten** Teilbaum.
- Man könnte dies auch verallgemeinern zu **ternären** etc. Bäumen.
- Was ist ein “unärer” Baum?

# Erklärungen zu binären Bäumen

- Analog zur leeren verlinkten Liste gibt es den **leeren Baum**, im Bild so dargestellt: .
- Ansonsten besteht ein binärer Baum aus einer **Knotenmarkierung (Label)** und einem **linken** und einem **rechten** Teilbaum.
- Man könnte dies auch verallgemeinern zu **ternären** etc. Bäumen.
- Was ist ein “unärer” Baum? Eine verlinkte Liste! Ein binärer Baum ist also sozusagen eine verlinkte Liste, bei der jeder Knoten zwei Nachfolger hat statt nur einen.

## Die Klasse

### Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte

Datentypen

Zusammen-

fassung

# Die Klasse Tree

# Die Klasse Tree

## Konstruktor und Attribute

```
trees.py
```

```
class Tree:  
    def __init__(self, left, label, right):  
        self.label = label  
        self.left = left  
        self.right = right
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung



# Die Klasse Tree

## Konstruktor und Attribute

```
trees.py
```

```
class Tree:  
    def __init__(self, left, label, right):  
        self.label = label  
        self.left = left  
        self.right = right
```

- Das Attribut `label` entspricht dem `data`-Attribut bei verlinkten Listen.
- Die `left`- und `right`-Attribute entsprechen dem `next`-Attribut bei verlinkten Listen. D.h. `left` und `right` sind wiederum Bäume.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

# Die Klasse Tree

## Konstruktor und Attribute

```
trees.py
```

```
class Tree:  
    def __init__(self, left, label, right):  
        self.label = label  
        self.left = left  
        self.right = right
```

- Das Attribut `label` entspricht dem `data`-Attribut bei verlinkten Listen.
- Die `left`- und `right`-Attribute entsprechen dem `next`-Attribut bei verlinkten Listen. D.h. `left` und `right` sind wiederum Bäume.
- Der **Konstruktor** ist so konzipiert, dass aus einer **Markierung** und **zwei Bäumen** ein nichtleerer Baum konstruiert wird. Er entspricht somit der **Methode** `cons` für verlinkte Listen.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

# Repräsentation des leeren Baums

```
trees.py
```

```
class Tree:
    def __init__(self, left, label, right):
        self.label = label
        self.left = left
        self.right = right
```

- Der **leere Baum** wird analog zu verlinkten Listen durch einen Knoten repräsentiert, bei dem alle Attribute None sind. Somit konstruiert man ihn durch den Aufruf `Tree(None, None, None)`.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

```
trees.py
```

```
class Tree:  
    def __init__(self, left, label, right):  
        self.label = label  
        self.left = left  
        self.right = right
```

- Es soll für einen Knoten  $x$  immer gelten:

$$x.\text{left} = \text{None} \Leftrightarrow x.\text{right} = \text{None},$$
$$x.\text{left} = \text{None} \wedge x.\text{right} = \text{None} \Rightarrow x.\text{label} = \text{None}$$

und

$$x.\text{left} = \text{None} \wedge x.\text{right} = \text{None} \Leftrightarrow x \text{ ist leerer Baum.}$$

# None in nichtleeren Bäumen

trees.py

```
class Tree:
    def __init__(self, left, label, right):
        self.label = label
        self.left = left
        self.right = right
empty_tree = Tree(None, None, None)
```

- Beachte: wir erlauben  $x.\text{label} = \text{None}$  auch für einen nichtleeren Baum (analog zu verlinkten Listen).

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

# None in nichtleeren Bäumen

trees.py

```
class Tree:
    def __init__(self, left, label, right):
        self.label = label
        self.left = left
        self.right = right
empty_tree = Tree(None, None, None)
```

- Beachte: wir erlauben  $x.\text{label} = \text{None}$  auch für einen nichtleeren Baum (analog zu verlinkten Listen).
- Analog zu verlinkten Listen führen wir eine **globale Variable** `empty_tree` ein.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

```
trees.py
```

```
class Tree:  
    def __init__(self, left, label, right):  
        self.label = label  
        self.left = left  
        self.right = right  
    def is_empty(self):  
        return (self.left is None and  
                self.right is None)
```

- Wir werden im Folgenden diese Methode verwenden, um Leerheit zu testen, und nicht einen expliziten Zugriff auf die Attribute `left` und/oder `right`.

# Konstruktion eines Baums

## Python-Interpreter

```
>>> t1 = Tree(empty_tree, 1, empty_tree)
>>> t2 = Tree(t1, 2, empty_tree)
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

**Konstruktion**

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

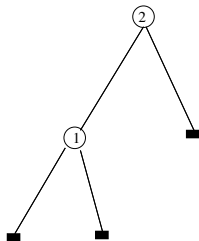


# Konstruktion eines Baums

## Python-Interpreter

```
>>> t1 = Tree(empty_tree, 1, empty_tree)
>>> t2 = Tree(t1, 2, empty_tree)
```

t2 sollte jetzt so aussehen:



Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

**Konstruktion**

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

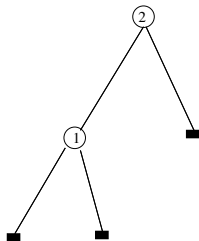
Zusammen-  
fassung

# Konstruktion eines Baums

## Python-Interpreter

```
>>> t1 = Tree(empty_tree, 1, empty_tree)
>>> t2 = Tree(t1, 2, empty_tree)
```

t2 sollte jetzt so aussehen:



Aber um uns im Weiteren die konstruierten Bäume auch vernünftig ansehen zu können, brauchen wir ein `prettyprint`.

- Eine Methode zu schreiben, um einen Baum schön darzustellen, ist gar nicht so einfach.

- Eine Methode zu schreiben, um einen Baum schön darzustellen, ist gar nicht so einfach.
  - Die Methode soll **rekursiv** sein.
  - Die Methode soll nur einfache Textausgabe benutzen, d.h. wenn eine Zeile einmal ausgedruckt ist, kann sie nicht mehr verändert werden.

- Eine Methode zu schreiben, um einen Baum schön darzustellen, ist gar nicht so einfach.
  - Die Methode soll **rekursiv** sein.
  - Die Methode soll nur einfache Textausgabe benutzen, d.h. wenn eine Zeile einmal ausgedruckt ist, kann sie nicht mehr verändert werden.
- Damit scheidet eine **vertikale** Darstellung, wo die **Wurzel** des Baums oben (oder unten) ist, aus.
- Eine **horizontale** Darstellung, wo die Wurzel links ist, bietet sich an.

# Horizontale Darstellung

Ein Baum `Tree(left, label, right)` soll folgendermaßen dargestellt werden:

```
n(right_print,
   label,
   left_print)
```

wobei `left_print` und `right_print` die Darstellungen des linken und rechten Teilbaums sind (Rekursion!).

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

**prettyprint**

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

# Horizontale Darstellung

Ein Baum `Tree(left, label, right)` soll folgendermaßen dargestellt werden:

```
n(right_print,
   label,
   left_print)
```

wobei `left_print` und `right_print` die Darstellungen des linken und rechten Teilbaums sind (Rekursion!).

Da eine Drehung gegen den Uhrzeigersinn aus “oben” “links” macht, und aus “links” “unten”, und aus “rechts” “oben”, müssen wir den **rechten** Teilbaum zuerst (“zuoberst”) ausgeben.

## Python-Interpreter

```
>>> t2.prettyprint()  
n(empty_tree,  
  2,  
  n(empty_tree,  
    1,  
    empty_tree))
```



# Befestigungspunkt

Man stelle sich vor, dass auf dem “n” für den jeweiligen Teilbaum ein **Befestigungspunkt** liegt:

```
n(right_print,  
   label,  
   left_print)
```

- Der **rote** Punkt ist der Befestigungspunkt für den Hauptbaum, die **blauen** Punkte sind die Befestigungspunkte für die Teilbäume.
- Die Methode `prettyprint` muss ihren Befestigungspunkt kennen, da die Unterbäume und das Label relativ dazu platziert werden sollen.

trees.py

```
def prettyprint(self):  
    self._prettyprint("")  
    print()
```

Nur auf der äußersten Ebene soll nach Ausgabe eines Baums ein Zeilenumbruch erfolgen. Deshalb werden wir eine Hilfsmethode `_prettyprint` haben, die eben keinen Zeilenumbruch am Ende der Ausgabe setzt.

## trees.py

```
def _prettyprint(self, indent):
    if self.is_empty():
        print("empty_tree", end="")
    else:
        print("n(", end="")
        newindent = indent + "  "
        self.right._prettyprint(newindent)
        print(",")
        print(newindent + str(self.label) + ",")
        print(newindent, end="")
        self.left._prettyprint(newindent)
        print(")", end="")
```

# Größeres Beispiel

## Python-Interpreter

```
>>> t1 = Tree(empty_tree, 1, empty_tree)
>>> t2 = Tree(t1, 2, empty_tree)
>>> t5 = Tree(empty_tree, 5, empty_tree)
>>> t7 = Tree(empty_tree, 7, t5)
>>> t4 = Tree(t7, 4, t2)
>>> t4.prettyprint()
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

**prettyprint**

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

# Größeres Beispiel II

## Python-Interpreter

```
n(n(empty_tree,  
    2,  
    n(empty_tree,  
        1,  
        empty_tree))),  
4,  
n(n(empty_tree,  
    5,  
    empty_tree),  
7,  
empty_tree))
```

Nicht superhübsch, aber einigermaßen erkennbar.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

**prettyprint**

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

Unter der **Tiefe** eines Baums versteht man die Anzahl der Ebenen im Baum. Formal ist die Tiefe induktiv definiert:

- Der leere Baum hat die Tiefe 0.
- Seien  $t_1$  bzw.  $t_2$  zwei Bäume mit Tiefe  $d_1$  bzw.  $d_2$  und  $l$  ein beliebiges Objekt. Dann hat der Baum konstruiert durch  $\text{Tree}(t_1, l, t_2)$  die Tiefe

$$\max(d_1, d_2) + 1$$

Aus der induktiven Definition ergibt sich leicht eine rekursive Methode ...

# Tiefe eines Baums: Methode

trees.py

```
def depth(self):
    if self.is_empty():
        return 0
    else:
        return 1 + max(self.left.depth(),
                       self.right.depth())
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

# Verwendung von depth

## Python-Interpreter

```
>>> t4.prettyprint()
n(n(empty_tree,
    2,
    n(empty_tree,
        1,
        empty_tree))),
4,
n(n(empty_tree,
    5,
    empty_tree),
7,
empty_tree))
>>> t4.depth()
3
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

Knoten zählen

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung



# Knoten zählen

Eine weitere interessante Frage ist: wie viele (echte) Knoten hat ein Baum? Nachdem wir `depth` definiert haben, ist das nun nicht schwierig:

```
trees.py
```

```
def node_count(self):
    if self.is_empty():
        return 0
    else:
        return (1 + self.left.node_count() +
                self.right.node_count())
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute

Der leere Baum

Konstruktion

prettyprint

Tiefe

**Knoten zählen**

Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

# Verwendung von node\_count

## Python-Interpreter

```
>>> t4.prettyprint()
n(n(empty_tree,
    2,
    n(empty_tree,
        1,
        empty_tree))),
4,
n(n(empty_tree,
    5,
    empty_tree),
7,
empty_tree))
>>> t4.node_count()
5
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Konstruktor und  
Attribute  
Der leere Baum  
Konstruktion  
prettyprint  
Tiefe  
**Knoten zählen**  
Verbesserungen

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

Man kann die Wartbarkeit des Codes erhöhen, indem man Methoden schreibt, um auf die Attribute einer Klasse zuzugreifen, und in der Definition aller anderen Methoden nur diese Zugriffsmethoden verwendet anstatt direktem Zugriff.

Allerdings kommt es zu Komplikationen, wenn der Name einer solchen Zugriffsmethode identisch mit dem Namen des entsprechenden Attributs ist.

# Suchbäume

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

## Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

- **Suchbäume** sind binäre Bäume mit zwei zusätzlichen Eigenschaften:
  - Die Knotenmarkierungen sind Objekte, auf denen eine **strikte totale Ordnung** definiert ist (wir nennen sie einfach  $<$ ).
  - Gegeben ein Baum  $\text{Tree}(s, l, t)$ , sind alle in  $s$  vorkommenden Markierungen  $< l$ , und alle in  $t$  vorkommenden Markierungen  $> l$ .

- **Suchbäume** sind binäre Bäume mit zwei zusätzlichen Eigenschaften:
  - Die Knotenmarkierungen sind Objekte, auf denen eine **strikte totale Ordnung** definiert ist (wir nennen sie einfach  $<$ ).
  - Gegeben ein Baum  $\text{Tree}(s, l, t)$ , sind alle in  $s$  vorkommenden Markierungen  $< l$ , und alle in  $t$  vorkommenden Markierungen  $> l$ .
- Was könnte das für einen Nutzen haben?

- **Suchbäume** sind binäre Bäume mit zwei zusätzlichen Eigenschaften:
  - Die Knotenmarkierungen sind Objekte, auf denen eine **strikte totale Ordnung** definiert ist (wir nennen sie einfach  $<$ ).
  - Gegeben ein Baum  $\text{Tree}(s, l, t)$ , sind alle in  $s$  vorkommenden Markierungen  $< l$ , und alle in  $t$  vorkommenden Markierungen  $> l$ .
- Was könnte das für einen Nutzen haben?  
Leichtes Auffinden einer Markierung.
- Wir sagen auch: ein Baum hat die **Suchbaumeigenschaft**.

- **Suchbäume** sind binäre Bäume mit zwei zusätzlichen Eigenschaften:
  - Die Knotenmarkierungen sind Objekte, auf denen eine **strikte totale Ordnung** definiert ist (wir nennen sie einfach  $<$ ).
  - Gegeben ein Baum  $\text{Tree}(s, l, t)$ , sind alle in  $s$  vorkommenden Markierungen  $< l$ , und alle in  $t$  vorkommenden Markierungen  $> l$ .
- Was könnte das für einen Nutzen haben?  
Leichtes Auffinden einer Markierung.
- Wir sagen auch: ein Baum hat die **Suchbaumeigenschaft**.
- Eigentlich sollten wir eine **eigene Klasse** für Suchbäume definieren, aber das wäre schon fortgeschrittene objektorientierte Programmierung.



# Gegenbeispiel

Der oben konstruierte Baum t4 ist kein Suchbaum:

## Python-Interpreter

```
>>> t4.prettyprint()
n(n(empty_tree,
    2,
    n(empty_tree,
        1,
        empty_tree))),
4,
n(n(empty_tree,
    5,
    empty_tree),
7,
empty_tree))
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

**Beispiele**

Suchen

Element

einfügen

Balanciertheit

Korrektheit von  
st.insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

## Python-Interpreter

```
>>> s1 = Tree(empty_tree, 1, empty_tree)
>>> s4 = Tree(empty_tree, 4, empty_tree)
>>> s2 = Tree(s1, 2, s4)
>>> s7 = Tree(empty_tree, 7, empty_tree)
>>> s5 = Tree(s2, 5, s7)
>>> s5.prettyprint()
```

## Python-Interpreter

```
n(n(n(empty_tree,
      7,
      empty_tree),
    5,
    n(n(empty_tree,
        4,
        empty_tree),
      2,
      n(empty_tree,
        1,
        empty_tree))))
```

Die offensichtlichste Operation auf Suchbäumen ist das **Suchen**, d.h.: gegeben ein Suchbaum und ein Element, kommt dieses Element als Knotenmarkierung vor?

Die offensichtlichste Operation auf Suchbäumen ist das **Suchen**, d.h.: gegeben ein Suchbaum und ein Element, kommt dieses Element als Knotenmarkierung vor?

```
trees.py
```

```
def st_member(self, element):
    if self.is_empty():
        return False
    elif self.label == element:
        return True
    elif element < self.label:
        return self.left.st_member(element)
    else:
        return self.right.st_member(element)
```

# Verwendung von st\_member

## Python-Interpreter

```
>>> s5.st_member(2)
True
>>> s5.st_member(8)
False
>>> t4.st_member(2)
False
```

Was bemerken wir?

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

**Suchen**

Element  
einfügen

Balanciertheit

Korrektheit von  
st.insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Verwendung von `st_member`

## Python-Interpreter

```
>>> s5.st_member(2)
True
>>> s5.st_member(8)
False
>>> t4.st_member(2)
False
```

Was bemerken wir?

Die letzte Antwort ist "falsch". Aber `t4` ist auch kein Suchbaum!

# Element einfügen

- Wir haben gesehen: `st_member` antwortet nur dann korrekt, wenn es auf einen Suchbaum angewendet wird. Jede andere Anwendung wäre ein Missbrauch.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

**Element  
einfügen**

Balanciertheit

Korrektheit von  
`st_insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung



# Element einfügen

- Wir haben gesehen: `st_member` antwortet nur dann korrekt, wenn es auf einen Suchbaum angewendet wird. Jede andere Anwendung wäre ein Missbrauch.
- Suchbäume mittels des `Tree`-Konstruktors zu konstruieren, ist umständlich und fehlerträchtig.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
`Tree`

Suchbäume

Beispiele

Suchen

**Element  
einfügen**

Balanciertheit

Korrektheit von  
`st_insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Element einfügen

- Wir haben gesehen: `st_member` antwortet nur dann korrekt, wenn es auf einen Suchbaum angewendet wird. Jede andere Anwendung wäre ein Missbrauch.
- Suchbäume mittels des `Tree`-Konstruktors zu konstruieren, ist umständlich und fehlerträchtig.
- Binäre Bäume, insbesondere Suchbäume, dienen dazu, andere Elemente aufzubewahren. Daher interessieren die Benutzerin im Wesentlichen drei Operationen:
  - **Einfügen** eines Elements in einen Suchbaum.
  - **Entfernen** eines Elements aus einem Suchbaum.
  - Test, ob ein Element in einem Suchbaum vorhanden ist (`st_member`).
- Wir wenden uns jetzt dem Einfügen zu.

## trees.py

```
def st_insert(self, element):
    if self.is_empty():
        return Tree(empty_tree, element, empty_tree)
    elif self.label == element:
        return self
    elif element < self.label:
        new_left = self.left.st_insert(element)
        return Tree(new_left, self.label, self.right)
    else:
        new_right = self.right.st_insert(element)
        return Tree(self.left, self.label, new_right)
```

## trees.py

```
def st_insert(self, element):
    if self.is_empty():
        return Tree(empty_tree, element, empty_tree)
    elif self.label == element:
        return self
    elif element < self.label:
        new_left = self.left.st_insert(element)
        return Tree(new_left, self.label, self.right)
    else:
        new_right = self.right.st_insert(element)
        return Tree(self.left, self.label, new_right)
```

- Duplikate erlauben wir nicht (Suchbaumeigenschaft!).
- Es wird entweder ein neuer Einknotenbaum konstruiert, oder das neue Element wird in den linken oder rechten Teilbaum eingefügt.

# Verwendung von st\_insert

Ein schöner Baum

## Python-Interpreter

```
>>> t = empty_tree
>>> t = t.st_insert(4)
>>> t = t.st_insert(2)
>>> t = t.st_insert(6)
>>> t = t.st_insert(1)
>>> t = t.st_insert(3)
>>> t = t.st_insert(5)
>>> t = t.st_insert(7)
>>> t.prettyprint()
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
st\_insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Verwendung von st\_insert

Ein schöner Baum II

## Python-Interpreter

```
n(n(n(empty_tree,
      7,
      empty_tree),
    6,
    n(empty_tree,
      5,
      empty_tree))),
  4,
  n(n(empty_tree,
      3,
      empty_tree),
    2,
    n(empty_tree,
      1,
      empty_tree)))
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
st\_insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Verwendung von st\_insert

Ein weniger schöner Baum

## Python-Interpreter

```
>>> t = empty_tree
>>> t = t.st_insert(1)
>>> t = t.st_insert(2)
>>> t = t.st_insert(3)
>>> t = t.st_insert(4)
>>> t = t.st_insert(5)
>>> t = t.st_insert(6)
>>> t = t.st_insert(7)
>>> t.prettyprint()
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
st\_insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Verwendung von st\_insert

## Ein weniger schöner Baum II

### Python-Interpreter

```
n(n(n(n(n(n(n(empty_tree,
                7,
                empty_tree),
                6,
                empty_tree),
                5,
                empty_tree),
                4,
                empty_tree),
                3,
                empty_tree),
                2,
                empty_tree),
                1,
                empty_tree)
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
st\_insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung



# Schöne Bäume?

Was kann man zu den vorigen Beispielen sagen?

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element

einfügen

**Balanciertheit**

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

Was kann man zu den vorigen Beispielen sagen?

- Im ersten Beispiel haben wir die Elemente “wild durcheinander” eingefügt, und das Ergebnis war ein **perfekt balancierter** Baum: die Tiefe ist exakt  $\log(n + 1)$ , wobei  $n$  die Knotenzahl ist.

Was kann man zu den vorigen Beispielen sagen?

- Im ersten Beispiel haben wir die Elemente “wild durcheinander” eingefügt, und das Ergebnis war ein **perfekt balancierter** Baum: die Tiefe ist exakt  $\log(n + 1)$ , wobei  $n$  die Knotenzahl ist.
- Im zweiten Beispiel haben wir die Elemente “sortiert” eingefügt, und das Ergebnis war ein zur verlinkten Liste **entarteter** (degenerierter) Baum.

# Schöne Bäume?

Was kann man zu den vorigen Beispielen sagen?

- Im ersten Beispiel haben wir die Elemente “wild durcheinander” eingefügt, und das Ergebnis war ein **perfekt balancierter** Baum: die Tiefe ist exakt  $\log(n + 1)$ , wobei  $n$  die Knotenzahl ist.
- Im zweiten Beispiel haben wir die Elemente “sortiert” eingefügt, und das Ergebnis war ein zur verlinkten Liste **entarteter** (degenerierter) Baum.
- In der Praxis werden die Bäume meistens irgendwo zwischen diesen Extremen liegen.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element

einfügen

**Balanciertheit**

Korrektheit von  
st.insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

Was kann man zu den vorigen Beispielen sagen?

- Im ersten Beispiel haben wir die Elemente “wild durcheinander” eingefügt, und das Ergebnis war ein **perfekt balancierter** Baum: die Tiefe ist exakt  $\log(n + 1)$ , wobei  $n$  die Knotenzahl ist.
- Im zweiten Beispiel haben wir die Elemente “sortiert” eingefügt, und das Ergebnis war ein zur verlinkten Liste **entarteter** (degenerierter) Baum.
- In der Praxis werden die Bäume meistens irgendwo zwischen diesen Extremen liegen.
- Fortgeschrittene Implementierungen von `st_member` **balancieren** den Baum, um ein bestimmtes Maß an Balanciertheit zu garantieren.





## Python-Interpreter

```
>>> t5 = Tree(empty_tree, 5, empty_tree)
>>> t2 = Tree(empty_tree, 2, empty_tree)
>>> t7 = Tree(t5, 7, t2)
>>> t9 = Tree(t7, 9, t2)
>>> t9.prettyprint()
```



# Nochmals Schabernack II

## Python-Interpreter

```
n(n(empty_tree,
    2,
    empty_tree),
    9,
    n(n(empty_tree,
        2,
        empty_tree),
        7,
        n(empty_tree,
            5,
            empty_tree)))
```

Die 2 kommt zweimal vor, da t2 zweimal bei der Konstruktion verwendet wurde. Scheint hier nicht so schlimm zu sein.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element

einfügen

Balanciertheit

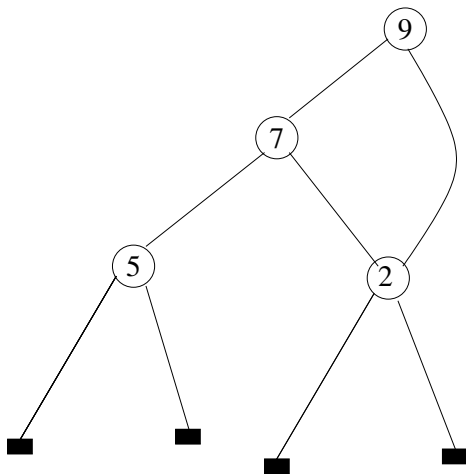
Korrektheit von  
st.insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Illustration



Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element

einfügen

Balanciertheit

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

Wir sehen, dass sich der Beweis der Korrektheit von `st_insert` schwierig gestalten könnte, aber wir müssen aufpassen, die Dinge nicht unnötig zu verkomplizieren ...

# Unabhängige Bäume

Intuitiv würde man wünschen, dass im durch  $\text{Tree}(s, l, t)$  die Teilbäume  $s$  und  $t$  “unabhängig” sind. Wir definieren:

## Definition

Zwei binäre Bäume  $s$  und  $t$  sind **voneinander unabhängig**, wenn für alle Folgen  $b_1, \dots, b_m$  und  $c_1, \dots, c_n$ , wobei  $n, m \geq 0$  und  $b_i, c_j \in \{\text{left}, \text{right}\}$ , gilt:

Wenn  $s.b_1 \dots b_m$  und  $t.c_1 \dots c_n$  beide existieren und es gilt

```
not s.b1.....b_m.is_empty()
s.b1.....b_m is not None
not t.c1.....c_n.is_empty()
t.c1.....c_n is not None
```

dann sind  $s.b_1 \dots b_m$  und  $t.c_1 \dots c_n$  nicht (im Sinne der Python-Semantik) **identisch**.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
st.insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Ein unabhängiger Baum

## Definition

Ein binärer Baum  $t$  ist **unabhängig**, wenn für alle Folgen

$b_1, \dots, b_m$  und  $c_1, \dots, c_n$ , wobei  $n, m \geq 0$  und

$b_i, c_j \in \{\text{left}, \text{right}\}$ , gilt:

Wenn  $t.b_1 \dots b_m$  und  $t.c_1 \dots c_n$  beide existieren,  $b_1, \dots, b_m$

und  $c_1, \dots, c_n$  nicht identisch sind, und es gilt

```
not s.b1.....b_m.is_empty()
```

```
s.b1.....b_m is not None
```

```
not t.c1.....c_n.is_empty()
```

```
t.c1.....c_n is not None
```

dann sind  $s.b_1 \dots b_m$  und  $t.c_1 \dots c_n$  nicht **identisch**.

# Ein unabhängiger Baum

## Definition

Ein binärer Baum  $t$  ist **unabhängig**, wenn für alle Folgen

$b_1, \dots, b_m$  und  $c_1, \dots, c_n$ , wobei  $n, m \geq 0$  und

$b_i, c_j \in \{\text{left}, \text{right}\}$ , gilt:

Wenn  $t.b_1 \dots b_m$  und  $t.c_1 \dots c_n$  beide existieren,  $b_1, \dots, b_m$

und  $c_1, \dots, c_n$  nicht identisch sind, und es gilt

```
not s.b1.....b_m.is_empty()
```

```
s.b1.....b_m is not None
```

```
not t.c1.....c_n.is_empty()
```

```
t.c1.....c_n is not None
```

dann sind  $s.b_1 \dots b_m$  und  $t.c_1 \dots c_n$  nicht **identisch**.

Beachte: der Einknotenbaum

`Tree(empty_tree, l, empty_tree)` ist unabhängig!

# Der Konstruktor und Unabhängigkeit

## Satz (Unabhängigkeit)

*Seien  $s$  und  $t$  zwei unabhängige, und voneinander unabhängige, Bäume. Dann konstruiert  $\text{Tree}(s, l, t)$  einen unabhängigen Baum.*

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element

einfügen

Balanciertheit

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Der Konstruktor und Unabhängigkeit

## Satz (Unabhängigkeit)

*Seien  $s$  und  $t$  zwei unabhängige, und voneinander unabhängige, Bäume. Dann konstruiert  $\text{Tree}(s, l, t)$  einen unabhängigen Baum.*

## Beweis.

Sei  $r$  der durch  $\text{Tree}(s, l, t)$  konstruierte Baum. Dann ist  $r$  ein "frisches" Objekt, d.h., es ist garantiert nicht mit irgendeinem vorher existierenden Objekt identisch.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element

einfügen

Balanciertheit

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung



# Der Konstruktor und Unabhängigkeit

## Satz (Unabhängigkeit)

*Seien  $s$  und  $t$  zwei unabhängige, und voneinander unabhängige, Bäume. Dann konstruiert  $\text{Tree}(s, l, t)$  einen unabhängigen Baum.*

## Beweis.

Sei  $r$  der durch  $\text{Tree}(s, l, t)$  konstruierte Baum. Dann ist  $r$  ein "frisches" Objekt, d.h., es ist garantiert nicht mit irgendeinem vorher existierenden Objekt identisch.

Für jede Folge  $b_1, \dots, b_m$ , wobei  $m \geq 0$  und  $b_i \in \{\text{left}, \text{right}\}$ , gilt:

- $r.\text{left}.b_1 \dots b_m$  ist identisch zu  $s.b_1 \dots b_m$ , und
- $r.\text{right}.b_1 \dots b_m$  ist identisch zu  $t.b_1 \dots b_m$ .

Da  $s$  und  $t$  unabhängig und voneinander unabhängig sind, folgt, dass  $r$  unabhängig ist. □

## Satz (Endlichkeit)

Sei  $t$  ein unabhängiger Baum. Dann gibt es keine **unendliche** Folge  $b_1, \dots$  mit  $b_i \in \{\text{left}, \text{right}\}$ , so dass für alle  $i$  gilt:  $t.b_1 \dots b_i$  existiert.

## Beweis.

Wenn es eine solche unendliche Folge gäbe, müsste es wegen der Unabhängigkeit von  $t$  unendlich viele explizit konstruierte Objekte geben, was nicht der Fall ist. □

# Wohlfundiertheit (Erinnerung 8. Kapitel)

## Definition

Eine Relation  $R \subseteq A \times A$  heißt **wohlfundiert**, falls  $R$  irreflexiv ist und jedes nichtleere  $B \subseteq A$  mindestens ein minimales Element bzgl.  $R$  besitzt.

## Definition

Sei  $R \subseteq A \times A$  ein Relation. Eine **unendliche absteigende Kette bzgl.  $R$**  ist eine unendliche Folge  $a_1, a_2, \dots$ , mit  $a_{i+1} R a_i$ .

Beispiel:  $<$  auf  $\mathbb{Z}$ . Dann ist  $0, -1, -2, \dots$  eine unendliche absteigende Kette, denn  $\dots - 2 < -1 < 0$ .

## Satz (Wohlfundiertheit und Ketten)

*Eine Relation  $R \subseteq A \times A$  ist wohlfundiert gdw. keine unendliche absteigende Kette bzgl.  $R$  existiert.*

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
st.insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Wohlfundiertheit bei Bäumen

Wir sprachen schon oft von **Teilbäumen**, sollten aber nochmal klarstellen:

## Definition

Seien  $s$  und  $t$  Bäume und  $r$  der durch `Tree( $s$ ,  $l$ ,  $t$ )` konstruierte Baum. Dann sind  $s$  und  $t$  **Teilbäume** von  $r$ .

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Wohlfundiertheit bei Bäumen

Wir sprachen schon oft von **Teilbäumen**, sollten aber nochmal klarstellen:

## Definition

Seien  $s$  und  $t$  Bäume und  $r$  der durch  $\text{Tree}(s, l, t)$  konstruierte Baum. Dann sind  $s$  und  $t$  **Teilbäume** von  $r$ .

## Satz (Wohlfundiertheit)

*Die Relation "Teilbaum von" ist wohlfundiert.*

## Beweis.

Folgt aus dem Satz von der Endlichkeit und dem Satz von der Wohlfundiertheit und den Ketten. □

# Wozu wohlfundierte Relationen?

Wozu ist eine wohlfundierte Relation nützlich?

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element

einfügen

Balanciertheit

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Wozu wohlfundierte Relationen?

Wozu ist eine wohlfundierte Relation nützlich?

Sie erlaubt uns, den Beweis der Korrektheit von `st_insert` mit Induktion zu führen.

Man bezeichnet Induktion auf Bäumen (aber auch z.B. auf verlinkten Listen) als **strukturelle Induktion**.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von

`st_insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

## Satz

*Sei  $t$  ein unabhängiger Baum mit Suchbaumeigenschaft und  $z$  eine beliebiges Objekt des Typs der Markierungen von  $t$ .*

*Dann enthält der durch  $t.st\_insert(z)$  konstruierte Baum das Element  $z$  und ansonsten die gleichen Elemente wie  $t$ , und er ist ebenfalls ein unabhängiger Baum mit Suchbaumeigenschaft.*

## Beweis.

Strukturelle Induktion ...



# st\_insert (Erinnerung)

trees.py

```
def st_insert(self, element):
    if self.is_empty():
        return Tree(empty_tree, element, empty_tree)
    elif self.label == element:
        return self
    elif element < self.label:
        new_left = self.left.st_insert(element)
        return Tree(new_left, self.label, self.right)
    else:
        new_right = self.right.st_insert(element)
        return Tree(self.left, self.label, new_right)
```

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element

einfügen

Balanciertheit

Korrektheit von  
st\_insert

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Induktionsbasis: $t$ is leerer Baum

## Beweis.

Der durch  $t.st\_insert(z)$  konstruierte Baum ist der durch  $Tree(empty\_tree, z, empty\_tree)$  konstruierte Baum, und dieser enthält  $z$  und ansonsten die gleichen Elemente wie der leere Baum, er ist unabhängig, und er erfüllt die Suchbaumeigenschaft.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Induktionsbasis: $t.\text{label} = z$

## Beweis.

Der durch  $t.\text{st\_insert}(z)$  konstruierte Baum ist  $t$  selbst, und dieser enthält  $z$  und ansonsten die gleichen Elemente wie  $t$ , er ist unabhängig, und er erfüllt die Suchbaumeigenschaft.

# Induktionsschritt: $z < t.\text{label}$

Gleiche Elemente

## Beweis.

Der durch  $t.\text{st\_insert}(z)$  konstruierte Baum ist der durch  $\text{Tree}(nl, t.\text{label}, t.\text{right})$  konstruierte Baum, wobei  $nl$  wiederum der durch  $t.\text{left}.\text{st\_insert}(z)$  konstruierte Baum ist.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element

einfügen

Balanciertheit

Korrektheit von  
**st\_insert**

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Induktionsschritt: $z < t.\text{label}$

Gleiche Elemente

## Beweis.

Der durch  $t.\text{st\_insert}(z)$  konstruierte Baum ist der durch  $\text{Tree}(nl, t.\text{label}, t.\text{right})$  konstruierte Baum, wobei  $nl$  wiederum der durch  $t.\text{left}.\text{st\_insert}(z)$  konstruierte Baum ist.

Nach Induktionshypothese ist  $nl$  unabhängig, enthält  $z$  und ansonsten die gleichen Elemente wie  $t.\text{left}$ , und er erfüllt die Suchbaumeigenschaft.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von

`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Induktionsschritt: $z < t.\text{label}$

Gleiche Elemente

## Beweis.

Der durch  $t.\text{st\_insert}(z)$  konstruierte Baum ist der durch  $\text{Tree}(nl, t.\text{label}, t.\text{right})$  konstruierte Baum, wobei  $nl$  wiederum der durch  $t.\text{left}.\text{st\_insert}(z)$  konstruierte Baum ist.

Nach Induktionshypothese ist  $nl$  unabhängig, enthält  $z$  und ansonsten die gleichen Elemente wie  $t.\text{left}$ , und er erfüllt die Suchbaumeigenschaft.

Somit enthält  $t.\text{st\_insert}(z)$  das Element  $z$  und ansonsten die gleichen Elemente wie  $t$ .

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von

`st.insert`

Entfernen eines

Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Induktionsschritt: $z < t.\text{label}$

## Unabhängigkeit

### Beweis.

Der durch  $t.\text{st\_insert}(z)$  konstruierte Baum ist der durch  $\text{Tree}(nl, t.\text{label}, t.\text{right})$  konstruierte Baum, wobei  $nl$  wiederum der durch  $t.\text{left}.\text{st\_insert}(z)$  konstruierte Baum ist.

Nach Induktionshypothese ist  $nl$  unabhängig, enthält  $z$  und ansonsten die gleichen Elemente wie  $t.\text{left}$ , und er erfüllt die Suchbaumeigenschaft.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Induktionsschritt: $z < t.\text{label}$

## Unabhängigkeit

### Beweis.

Der durch  $t.\text{st\_insert}(z)$  konstruierte Baum ist der durch  $\text{Tree}(nl, t.\text{label}, t.\text{right})$  konstruierte Baum, wobei  $nl$  wiederum der durch  $t.\text{left}.\text{st\_insert}(z)$  konstruierte Baum ist.

Nach Induktionshypothese ist  $nl$  unabhängig, enthält  $z$  und ansonsten die gleichen Elemente wie  $t.\text{left}$ , und er erfüllt die Suchbaumeigenschaft.

Da der Aufruf  $t.\text{left}.\text{st\_insert}(z)$  an keiner Stelle ein existierendes `left`- oder `right`-Attribut ändert, sondern allenfalls ein "frisches" Objekt konstruiert, und da außerdem  $t.\text{left}$  und  $t.\text{right}$  voneinander unabhängig sind, folgt, dass  $nl$  und  $t.\text{right}$  voneinander unabhängig sind und somit nach dem Satz von der Unabhängigkeit  $t.\text{st\_insert}(z)$  unabhängig ist.



# Induktionsschritt: $z < t.\text{label}$

## Suchbaumeigenschaft

### Beweis.

Der durch  $t.\text{st\_insert}(z)$  konstruierte Baum ist der durch  $\text{Tree}(nl, t.\text{label}, t.\text{right})$  konstruierte Baum, wobei  $nl$  wiederum der durch  $t.\text{left}.\text{st\_insert}(z)$  konstruierte Baum ist.

Nach Induktionshypothese ist  $nl$  unabhängig, enthält  $z$  und ansonsten die gleichen Elemente wie  $t.\text{left}$ , und er erfüllt die Suchbaumeigenschaft.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
`st_insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Induktionsschritt: $z < t.\text{label}$

## Suchbaumeigenschaft

### Beweis.

Der durch  $t.\text{st\_insert}(z)$  konstruierte Baum ist der durch  $\text{Tree}(nl, t.\text{label}, t.\text{right})$  konstruierte Baum, wobei  $nl$  wiederum der durch  $t.\text{left}.\text{st\_insert}(z)$  konstruierte Baum ist.

Nach Induktionshypothese ist  $nl$  unabhängig, enthält  $z$  und ansonsten die gleichen Elemente wie  $t.\text{left}$ , und er erfüllt die Suchbaumeigenschaft.

Weiter gilt nach Voraussetzung  $d < t.\text{label} < e$  für alle  $d$  in  $t.\text{left}$  und  $e$  in  $t.\text{right}$ .

Somit folgt, dass  $d < t.\text{label} < e$  für alle  $d$  in  $nl$  und  $e$  in  $t.\text{right}$  gilt, d.h.,  $t.\text{st\_insert}(z)$  erfüllt die Suchbaumeigenschaft.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
`st_insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

Induktionsschritt:  $z > t.label$  (else)

Beweis.

Symmetrisch zum vorigen Fall.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
`st.insert`

Entfernen eines  
Elements

Abstrakte  
Datentypen

Zusammen-  
fassung

# Entfernen eines Elements

Das Entfernen eines Elements ist ein bisschen kompliziert und wird hier nicht behandelt.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Beispiele

Suchen

Element  
einfügen

Balanciertheit

Korrektheit von  
st.insert

**Entfernen eines  
Elements**

Abstrakte  
Datentypen

Zusammen-  
fassung

# Abstrakte Datentypen

# Abstrakte Datentypen

- Oben hieß es: Bei Suchbäumen interessieren die Benutzerin im Wesentlichen drei Operationen:
  - **Einfügen** eines Elements in einen Suchbaum.
  - **Entfernen** eines Elements aus einem Suchbaum.
  - Test, ob ein Element in einem Suchbaum **vorhanden** ist.

(Man könnte diese Auflistung evtl. noch erweitern, z.B. um einen **Leerheitstest** oder eine Methode, die die Anzahl der Elemente zurückgibt.)

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

# Abstrakte Datentypen

- Oben hieß es: Bei Suchbäumen interessieren die Benutzerin im Wesentlichen drei Operationen:
  - **Einfügen** eines Elements in einen Suchbaum.
  - **Entfernen** eines Elements aus einem Suchbaum.
  - Test, ob ein Element in einem Suchbaum **vorhanden** ist.

(Man könnte diese Auflistung evtl. noch erweitern, z.B. um einen **Leerheitstest** oder eine Methode, die die Anzahl der Elemente zurückgibt.)

- **Abstrakter** kann es so sein, dass die Benutzerin sagt:

*Ich brauche irgendeinen Typ, den ich zum Aufbewahren beliebiger Elemente verwenden kann. D.h. die oben genannten Operationen müssen zur Verfügung stehen. **Details** interessieren mich nicht!*

Man spricht dann von einem **abstrakten Datentyp**.

# Der abstrakte Datentyp Searchable

Genauer würde die Benutzerin z.B. Folgendes festlegen: es soll eine Klasse `Searchable` mit drei Methoden `insert`, `delete`, `member` und einer Konstanten (globalen Variablen) `empty` geben, die folgende Bedingungen erfüllt:

$$(\forall x t) \text{empty.member}(x) = \text{False} \quad (1)$$

$$(\forall x t) t.\text{insert}(x).\text{member}(x) = \text{True} \quad (2)$$

$$(\forall x t) t.\text{delete}(x).\text{member}(x) = \text{False} \quad (3)$$

$$\begin{aligned} (\forall x y t) t.\text{member}(x) = \text{False} \wedge x \neq y &\Rightarrow \\ t.\text{insert}(y).\text{member}(x) = \text{False} &\quad (4) \end{aligned}$$



# Der abstrakte Datentyp Searchable

Genauer würde die Benutzerin z.B. Folgendes festlegen: es soll eine Klasse `Searchable` mit drei Methoden `insert`, `delete`, `member` und einer Konstanten (globalen Variablen) `empty` geben, die folgende Bedingungen erfüllt:

$$(\forall x t) \text{empty.member}(x) = \text{False} \quad (1)$$

$$(\forall x t) t.\text{insert}(x).\text{member}(x) = \text{True} \quad (2)$$

$$(\forall x t) t.\text{delete}(x).\text{member}(x) = \text{False} \quad (3)$$

$$\begin{aligned} (\forall x y t) t.\text{member}(x) = \text{False} \wedge x \neq y &\Rightarrow \\ t.\text{insert}(y).\text{member}(x) = \text{False} &\quad (4) \end{aligned}$$

Vielleicht gibt es dann einen Programmierer, der sagt: ich kann diesen abstrakten Datentyp **implementieren**.

# Möglichkeiten der Implementierung

- Verlinkte Liste
- Suchbaum wie hier präsentiert (`delete` fehlt!)
- Suchbaum, der in irgendeiner Weise balanciert ist.
- ...

In jedem Fall muss die Implementierung die oben genannten Bedingungen erfüllen.

# Status des Konzepts

Wir haben jetzt eine grobe Vorstellung, was ein abstrakter Datentyp ist, aber es ist noch nicht ganz klar, welchen "Status" das Konzept hat.

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

**Abstrakte  
Datentypen**

Zusammen-  
fassung

# Status des Konzepts

Wir haben jetzt eine grobe Vorstellung, was ein abstrakter Datentyp ist, aber es ist noch nicht ganz klar, welchen “Status” das Konzept hat.

Spektrum:

- Die Benutzerin benutzt eine hoffentlich verständliche ad-hoc Syntax, um die Bedingungen aufzuschreiben, und der Programmierer behauptet nach scharfem Hinsehen, dass seine Implementierung die Bedingungen erfüllt.
- Die Syntax zum Aufschreiben der Bedingungen ist exakt definiert und ein Teil der Programmierumgebung, und innerhalb der Umgebung wird **maschinell** bewiesen, dass eine **konkrete** Klasse eine Implementierung des abstrakten Datentyps ist.

In jedem Fall sind abstrakte Datentypen ein nützliches Konzept zur Strukturierung der Programmierarbeit.

# Zusammenfassung

Informatik I

Jan-Georg  
Smaus

Die Klasse  
Tree

Suchbäume

Abstrakte  
Datentypen

Zusammen-  
fassung

- Wir haben **binäre Bäume** kennengelernt. Diese sind eine Datenstruktur ähnlich wie verlinkte Listen, aber jeder Knoten hat nun zwei Nachfolger.
- Die definierten Methoden sind: `is_empty`, `prettyprint`, `depth`, `node_count`.

- Wir haben **binäre Bäume** kennengelernt. Diese sind eine Datenstruktur ähnlich wie verlinkte Listen, aber jeder Knoten hat nun zwei Nachfolger.
- Die definierten Methoden sind: `is_empty`, `prettyprint`, `depth`, `node_count`.
- Weiter haben wir **Suchbäume** kennengelernt, einen Spezialfall, bei dem die Elemente im Baum in einer bestimmten Weise sortiert sind.
- Die definierten Methoden sind: `st_member` und `st_insert`.

- Wir haben **binäre Bäume** kennengelernt. Diese sind eine Datenstruktur ähnlich wie verlinkte Listen, aber jeder Knoten hat nun zwei Nachfolger.
- Die definierten Methoden sind: `is_empty`, `prettyprint`, `depth`, `node_count`.
- Weiter haben wir **Suchbäume** kennengelernt, einen Spezialfall, bei dem die Elemente im Baum in einer bestimmten Weise sortiert sind.
- Die definierten Methoden sind: `st_member` und `st_insert`.
- Wir haben die Korrektheit von `st_insert` mittels **struktureller Induktion** bewiesen.



- Wir haben **binäre Bäume** kennengelernt. Diese sind eine Datenstruktur ähnlich wie verlinkte Listen, aber jeder Knoten hat nun zwei Nachfolger.
- Die definierten Methoden sind: `is_empty`, `prettyprint`, `depth`, `node_count`.
- Weiter haben wir **Suchbäume** kennengelernt, einen Spezialfall, bei dem die Elemente im Baum in einer bestimmten Weise sortiert sind.
- Die definierten Methoden sind: `st_member` und `st_insert`.
- Wir haben die Korrektheit von `st_insert` mittels **struktureller Induktion** bewiesen.
- Wir haben gelernt, was ein **abstrakter Datentyp** ist.