Informatik I

22. Binäre Bäume

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

3. Februar 2011

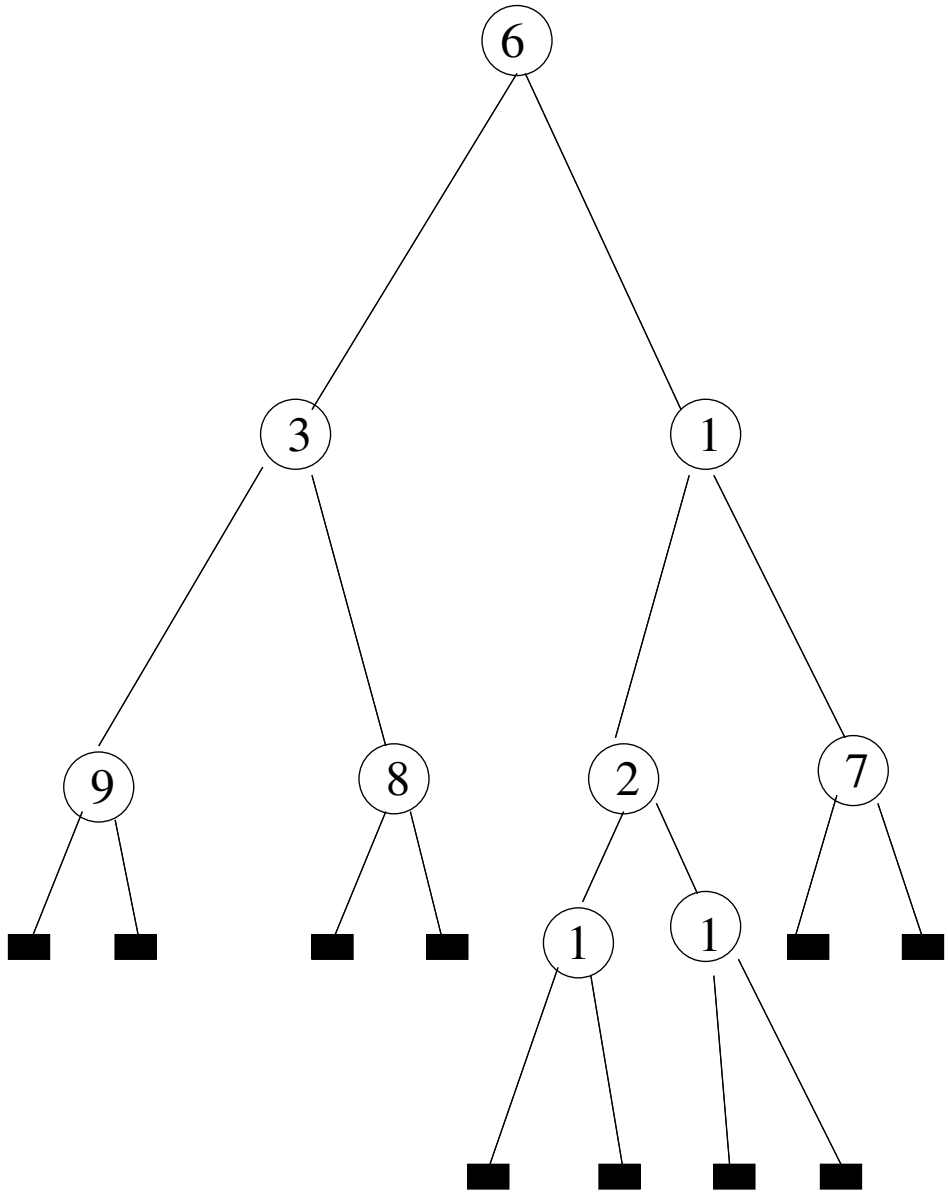
Informatik I

3. Februar 2011 — 22. Binäre Bäume

Binäre Bäume

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



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.

22.1 Die Klasse Tree

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

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.

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)`.

Charakterisierung

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:

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

und

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

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.label $=$ None auch für einen nichtleeren Baum (analog zu verlinkten Listen).
- Analog zu verlinkten Listen führen wir eine globale Variable empty_tree ein.

Testen auf Leerheit

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)
```

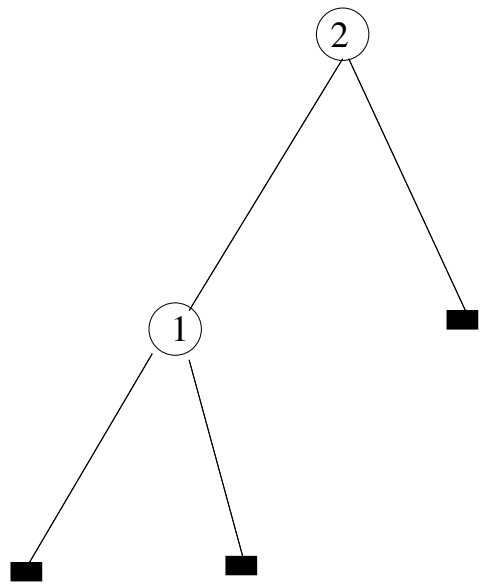
- Wir werden im Folgenden diese Methode verwenden, um Leerheit zu testen, und nicht einen expliziten Zugriff auf die Attribute `left` und/oder `right`.

Konstruktion eines Baums

Python-Interpreter

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

t2 sollte jetzt so aussehen:



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

Einen Baum ausdrucken

- 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!).

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.

Beispiel

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.

prettyprint

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.

_prettyprint

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()
```

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.

Tiefe eines Baums

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 `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())
```

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
```

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())
```

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
```

Verbesserungen

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.

22.2 Suchbäume

- Beispiele
- Suchen
- Element einfügen
- Balanciertheit
- Korrektheit von `st_insert`
- Entfernen eines Elements

Suchbäume

- 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 `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))
```

Beispiel

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()
```

Beispiel

Python-Interpreter

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

Suchen

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?

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.
- 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.

st_insert

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()
```

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)))
```

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()
```

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)
```

Schöne Bäume?

Was kann man zu den vorigen Beispielen sagen?

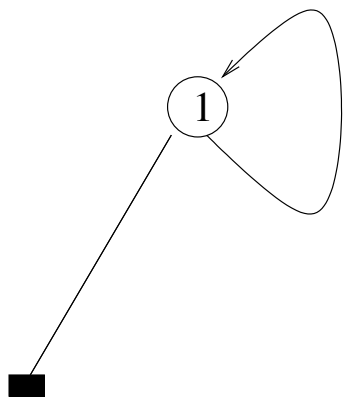
- Im ersten Beispiel haben wir die Elemente "wild durcheinander" eingefügt, und das Ergebnis war ein perfekt balancierter Baum: die Tiefe ist exakt $\log(n+1)$, wobei n die Knotenzahl ist.
- Im zweiten Beispiel haben wir die Elemente "sortiert" eingefügt, und das Ergebnis war ein zur verlinkten Liste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

Schabernack mit Bäumen

Python-Interpreter

```
>>> t = Tree(empty_tree, 1, empty_tree)
>>> t.right = t
>>> t.prettyprint()
n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(
n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(
n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n(n( ...
```

`t` ist kein richtiger binärer Baum mehr, sondern “zirkulär” (wie wir es schon mal für verlinkte Listen hatten).



Nochmals Schabernack

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()
```

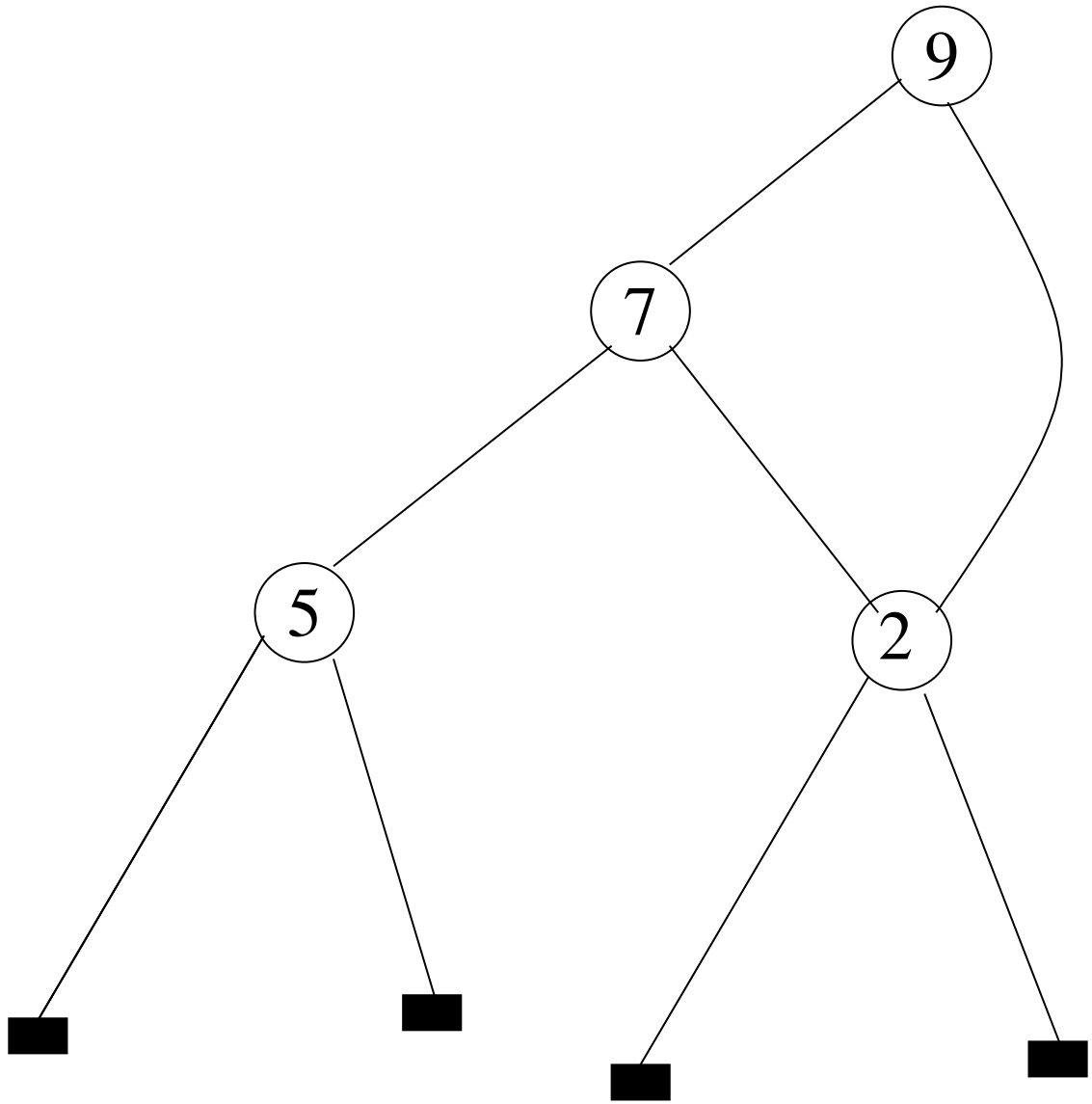
Nochmals Schabernack II

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.

Illustration



Schabernack verbieten

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 `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 $b_1, \ldots, b_m$ und $c_1, \ldots c_n$, wobei $n, m \geq 0$ und $b_i, c_j \in \{\texttt{left}, \texttt{right}\}$, gilt:
Wenn $s.b1.....b_m$ und $t.c_1.....c_n$ beide existieren und es gilt

$$\texttt{not } s.b1.....b_m\texttt{.is\_empty()}$$
$$s.b1.....b_m \texttt{ is not None}$$
$$\texttt{not } t.c_1.....c_n\texttt{.is\_empty()}$$
$$t.c_1.....c_n \texttt{ is not None}$$

dann sind $s.b1.....b_m$ und $t.c_1.....c_n$ nicht (im Sinne der Python-Semantik) identisch.

Ein unabhängiger Baum

Definition

Ein binärer Baum t ist unabhängig, wenn für alle Folgen $b_1, \ldots, b_m$ und $c_1, \ldots c_n$, wobei $n, m \geq 0$ und $b_i, c_j \in \{\texttt{left}, \texttt{right}\}$, gilt:
Wenn $t.b1.\ldots.b_m$ und $t.c_1.\ldots.c_n$ beide existieren, $b_1, \ldots, b_m$ und $c_1, \ldots c_n$ nicht identisch sind, und es gilt

$$\texttt{not } s.b1.\ldots.b_m\texttt{.is\_empty()}$$
$$s.b1.\ldots.b_m \texttt{ is not None}$$
$$\texttt{not } t.c_1.\ldots.c_n\texttt{.is\_empty()}$$
$$t.c_1.\ldots.c_n \texttt{ is not None}$$

dann sind $s.b1.\ldots.b_m$ und $t.c_1.\ldots.c_n$ 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 `Tree(`s`,` l`,` t`)` *einen unabhängigen Baum.*

Beweis.

Sei r der durch `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 $b_1, \ldots, b_m$, wobei $m \geq 0$ und $b_i \in \{\texttt{left}, \texttt{right}\}$, gilt:

- $r.\texttt{left}.b1.\ldots.b_m$ ist identisch zu $s.b1.\ldots.b_m$, und
- $r.\texttt{right}.b1.\ldots.b_m$ ist identisch zu $t.b1.\ldots.b_m$.

Da s und t unabhängig und voneinander unabhängig sind, folgt, das t unabhängig ist. □

Nur endliche Pfade

Satz (Endlichkeit)

Sei t ein unabhängiger Baum. Dann gibt es keine unendliche Folge $b_1, \ldots$ mit $b_i \in \{\texttt{left}, \texttt{right}\}$, so dass für alle i gilt: $t.b_1.\ldots.b_i$ 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 $a_1, a_2, \ldots,$ mit $a_{i+1}\ R\ a_i$.

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

Satz (Wohlfundiertheit und Ketten)

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

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.

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?

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.

Korrektheit von st_insert

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)
```

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.

Induktionsbasis: t.label $= z$

Beweis.

Der durch t.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: $z < t.\texttt{label}$

Gleiche Elemente

Beweis.

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

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

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

Induktionsschritt: $z < t.\texttt{label}$

Unabhängigkeit

Beweis.

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

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

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

Induktionsschritt: $z < t.\texttt{label}$

Suchbaumeigenschaft

Beweis.

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

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

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

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

Induktionsschritt: $z > t.\texttt{label}$ (else)

Beweis.
Symmetrisch zum vorigen Fall. □

Entfernen eines Elements

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

22.3 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.)
- 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:

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

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

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

$$(\forall x\ y\ t)\ t.\texttt{member}(x) = \texttt{False} \wedge x \neq y \Rightarrow t.\texttt{insert}(y).\texttt{member}(x) = \texttt{False} \quad (4)$$

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.

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 Programmiererumgebung, 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.

22.4 Zusammenfassung

Zusammenfassung

- 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.