Baumtraversierungen

- Allg.: häufig müssen alle Knoten eines Baumes besucht werden, um bestimmte Operationen auf ihnen durchführen zu können
- Operation in Visitor-Klasse verpacken, z.B.

```
class PrintNode(object):
    def __init__( self, param ):
        . . .
    def visit( self, treenode ):
        print treenode.getItem()
```

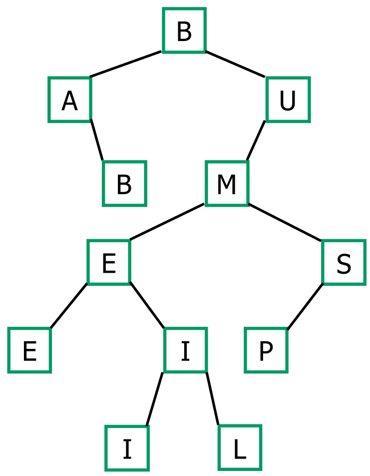
  - kann dann der Traversierungsmethode als Parameter übergeben werden
- Traversierungsarten
  - es gibt viele Möglichkeiten, einen Baum abzuwandern
  - Unterschiede in der Reihenfolge, in der die Knoten besucht werden

Preorder

- Reihenfolge:
  1. Besuche die Wurzel, führe Operation an Wurzel durch
  2. Traversiere den linken Teilbaum (in Preorder Reihenfolge)
  3. Traversiere den rechten Teilbaum (in Preorder)
- Implementierung in Python:

```
class Tree (cont'd) …
    def preorder( self, visitor ):
        visitor.visit(self)
        if self.left != None:
            self.left.preorder(visitor)
        if self.right != None :
            . . .
printnodes = PrintNode(...)
tree.preorder( printnodes )
```

- Beispiel



Preorder Traversierung:

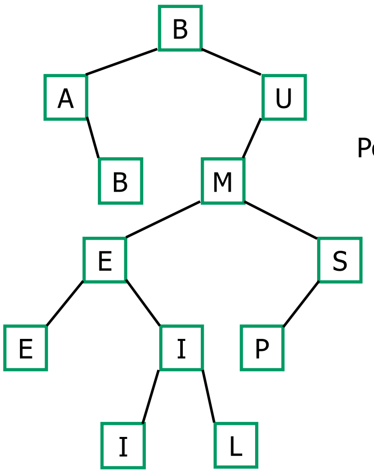
BABUMEEIILSP

Postorder

- Reihenfolge:
  - Traversiere den linken Teilbaum (in Postorder)
  - Traversiere den rechten Teilbaum (in Postorder)
  - Besuche die Wurzel ← führe Operation an Wurzel durch
- Implementierung in Python:

```
class Tree (cont'd) …
    def postorder( self, visitor ):
        if self.left:
            self.left.postorder(visitor)
        if self.right:
            self.right.postorder(visitor)
        visitor.visit(self)
```

- Beispiel



Postorder Traversierung:

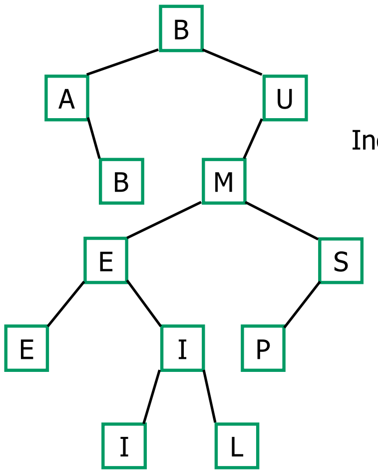
BAEILIEPSMUB

Inorder

- Reihenfolge:
  1. Traversiere den linken Teilbaum (in Inorder)
  2. Besuche die Wurzel, Operation an Wurzel
  3. Traversiere den rechten Teilbaum (in Inorder)

```
def inorder(self, visitor):
    if self.left:
        self.left.inorder(visitor)
    visitor.visit(self)
    if self.right:
        self.right.inorder(visitor)
```

- Beispiel

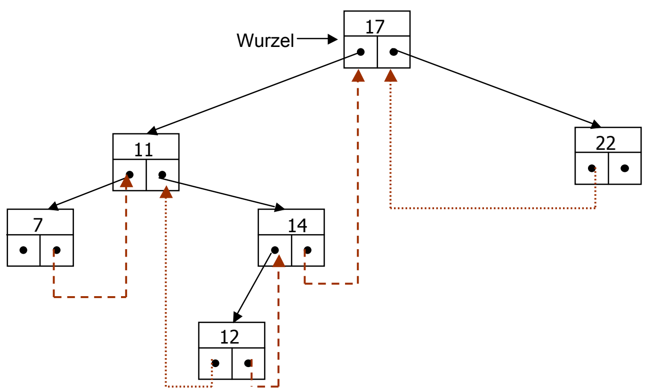


Inorder Traversierung:

ABBEEIILMPSU

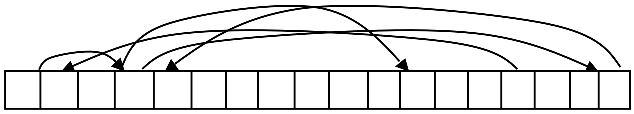
Nicht-rekursive Varianten mit *threaded trees*

- Rekursion kann vermieden werden, wenn man anstelle der Null-Referenzen sogenannte *thread pointer* auf den in-order Vorgänger bzw. den in-order Nachfolger verwendet:



Lokalität und Bäume

- Einfügen, Löschen und Umhängen von Knoten führen zu Adressfolgen, die keinerlei Lokalität aufweisen



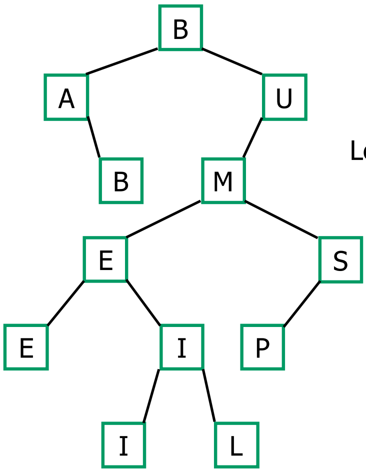
- relativ harmlos, falls sich alle Daten im Hauptspeicher befinden
  - aber: schlechte Ausnutzung des Caches (s. später)
- Katastrophe, falls Daten auf Festplatte oder Magnetband
  - siehe 2-3-4-Bäume, B-Bäume, Rot-Schwarz-Bäume

Levelorder (aka *breadth-first search*, BFS)

- Reihenfolge:
  - besuche die Knoten schichtweise
    - zuerst die Wurzel
    - dann die Wurzeln des linken und rechten Teilbaums
    - etc.
- Algorithmus
  - kann nicht rekursiv angegeben werden
  - erfordert eine Zwischenspeicherung der Knoten in einer Queue

```
def levelorder(self,visitor):
    q = Queue()
    q.enqueue(self)
    while not q.empty():
        n = q.dequeue()
        if n != None:
            visitor.visit(n)
            q.enqueue(n.left)
            q.enqueue(n.right)
```

- Beispiel



Levelorder Traversierung:

`BAUBMESEIPIL`

Exkurs: Visitor Pattern

- Aufteilung der Operationen auf Knoten und Traversierung ist einfachste Form des sog. Visitor Patterns
- Klasse **`PrintNode`** heißt Visitor, weil diese jeden Knoten "besucht"
- Methode **`preorder/postorder`** heißt Mapper , weil diese die Operation auf jeden Knoten anwenden (*mappen*), und wissen, in welcher Reihenfolge dies geschehen soll

- Vorteil von Visitor-Klasse im Gegensatz zu einer Visitor-Funktion: man kann damit die Operationen sehr einfach parametrisieren, z.B.

```
class PrintNode(object):
    def __init__(self, tolower = False):
        self.tolower = tolower
    def visit(node):
        s = str( node.getData() ) # make sure we
        if self.tolower:          # get a string
           s = s.lower()
        print s
```

```
r = root of tree
v = PrintNode()
r.preorder(v)          # print all nodes in preorder
v = PrintNode(true)
r.preorder(v)          # again, but all in lowercase
```

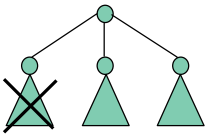

- Vorteil der Trennung in 2 Klassen:
  - man muß Traversierungsroutine nur 1x schreiben
  - kann beliebige Operationen ausführen lassen
- Beispiel: andere Operation: Knoten in Liste sammeln

```
class CollectNodes(object):
    def __init__(self):
        self.nodes = []
    def visit(self, node):
        self.nodes.append( node.getData() )
```

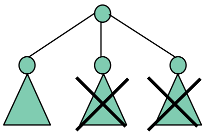
```
v = CollectNodes()
root.preorder(v)
print v.nodes
```

Feinere Steuerung der Traversierung

- Abhängig von Bedingung, möchte man manchmal…
  - in Teilbäume absteigen, abhängig von einer bestimmten Bedingung



  - Brüder überspringen



  - sofort komplett mit der Traversierung aufhören

- Lösung:
  - Visitor bestimmt obige Bedingungen
  - Return-Code von visitor.visit() in Traversal-Funktion abfragen
- Erweiterung von Visitor-Code:

```
Continue = 0            # symbolische Konstanten
Prune = 1
Skip = 2
Quit = 3
class Visitor(object):
    def visit(…):
        if … :
            return Prune/Skip/Quit
        …
        return Continue
```

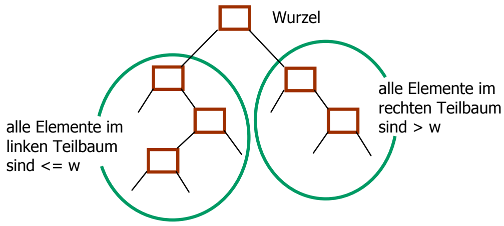
- Erweiterung des Traversierungs-Code:

```
class Tree(object):

    def preorder( self, visitor ):
        code = visitor.visit(self)
        if code == Quit or code == Skip:
            return code
        if code == Prune:
            return Continue

        if self.left:
            code = self.left.preorder(visitor)
            if code == Quit:   return Quit
            if code == Skip:   return Continue    # sic!
        if self.right:
            code = self.right.preorder(visitor)
            if code == Quit:   return Quit
        return Continue
```


Binäre Suchbäume (*binary search tree*, BST)

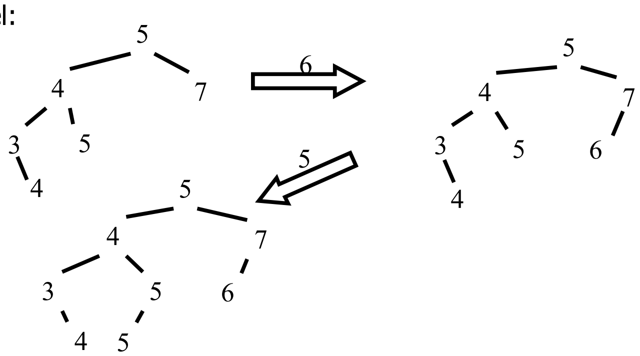
- Speichere wieder Daten als "Schlüssel + Nutzdaten"
- Sei eine Ordnungsrelation für die Schlüssel im Baum definiert
- Ziel: Binärbäume zur Speicherung von Mengen von Schlüsseln, so daß folgende Operationen effizient sind:
  - Suchen (*find*)
  - Einfügen (*insert*)
  - Entfernen (*remove, delete*)

- Definition Binärer Suchbaum:
  - ist entweder leer
  - oder hat einen Knoten (Wurzel) und zwei binäre Suchbäume, den linken und rechten Teilbaum
  - dabei sind alle Knoten des linken Teilbaumes kleiner oder gleich der Wurzel
    - alle Knoten des rechten Teilbaumes sind größer als die Wurzel
- Suchbaumeigenschaft:



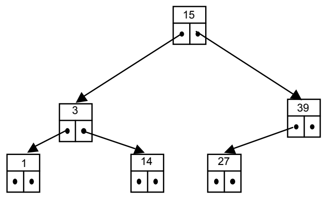
Einfügen

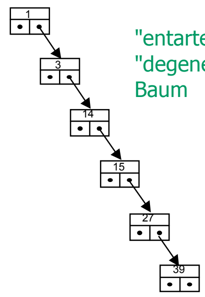
- Hinzufügen von x zum Baum B
  - wenn B leer ist, erzeuge Wurzel mit x
  - wenn x <= Wurzel, füge x rekursiv zum linken Teilbaum hinzu
  - wenn x > Wurzel, füge x rekursiv dem rechten Teilbaum hinzu
- Beispiel:



Sonderfälle

- Achtung:
  - Baum-Struktur hängt von Einfügereihenfolge im anfangs leeren Baum ab!
  - Dito für Höhe: Höhe kann, je nach Reihenfolge, zwischen n und $\lceil \log_2(n+1) \rceil$ liegen
- Resultierende Suchbäume für die Reihenfolgen 15, 39, 3, 27, 1, 14 und 1, 3, 14, 15, 27, 39:

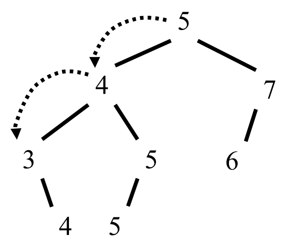




"entarteter" oder "degenerierter" Baum

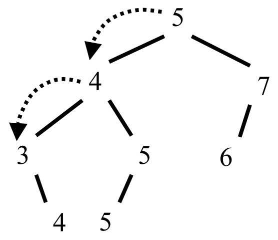
Suchen

- x im BST B suchen
  - wenn B leer ist, dann ist x nicht in B
  - wenn x = Wurzelelement gilt, haben wir x gefunden
  - wenn x < Wurzelelement, suche im linken Teilbaum
  - wenn x > Wurzelelement, suche im rechten Teilbaum
- Beispiel: suche 3 im Baum



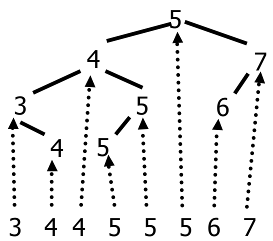
  - gibt es mehrere Knoten mit gleichem Wert , wird offenbar derjenige mit der geringsten Tiefe gefunden

- Suche kleinstes Element
  - folge dem linken Teilbaum, bis Knoten gefunden wurde, dessen linker Teilbaum leer ist



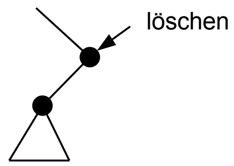
- analog: größtes Element finden

- Nachfolger in Sortierreihenfolge
  - wenn der Knoten einen rechten Teilbaum hat, ist der Nachfolger das kleinste Element des rechten Teilbaumes
  - ansonsten: Aufsteigen im Baum, bis ein Element gefunden wurde, das größer oder gleich dem aktuellen Knoten ist oder man die Wurzel erreicht hat
  - dazu sollte der Knoten evtl. eine Referenz auf seinen Vater beinhalten:
- Beispiel:

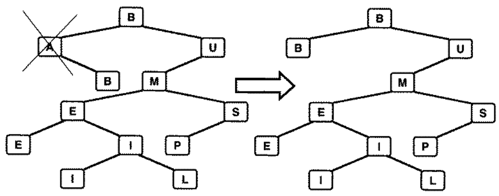


Löschen

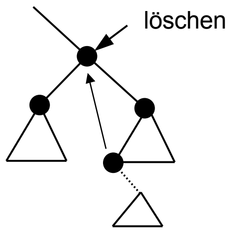
- aufwendigste Operation
  - es muß auch hier sichergestellt werden, daß der verbleibende Baum ein gültiger binärer Suchbaum ist
- 1. Fall: Knoten hat keinen Teilbaum (Blatt)
  - Knoten einfach löschen
- 2. Fall: nur ein Teilbaum
  - Teilbaum eine Etage höher schieben



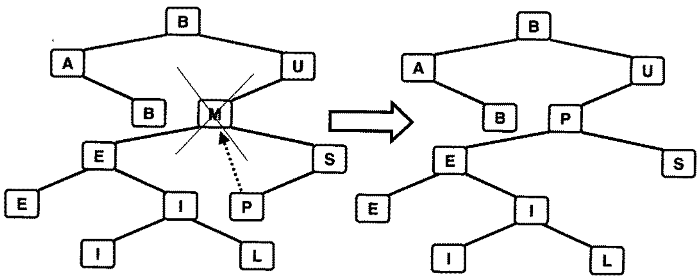
- Beispiel:



- 3. Fall: zwei Teilbäume
  - suche kleinstes Element des rechten Teilbaumes
    - kleinstes Element hat höchstens einen rechten Teilbaum
  - kopiere Inhalt (Schlüssel + Daten) dieses kleinsten Knotens in den Inhalt des zu löschenden Knotens
  - lösche den Knoten, der vorher das kleinste Element beinhaltete → Fall 2



- Beispiel



Geht so nur, falls das hochgetauschte Element P
echt kleiner als die Wurzel des rechten Teilbaumes S ist.