- Satz: Ein maximal vollständiger binärer Baum der Höhe h enthält 2^{h-1} Blätter und 2^h-1 Knoten und $2^{h-1}-1$ inneren Knoten.

- Beweis:

1. Induktionsanfang: h= 1

   Der Baum besteht nur aus der Wurzel, die auch das einzige Blatt ist:

   $2^{1-1} = 2^0 = 1$ Blatt

   $2^1-1 = 2 - 1 = 1$ Knoten

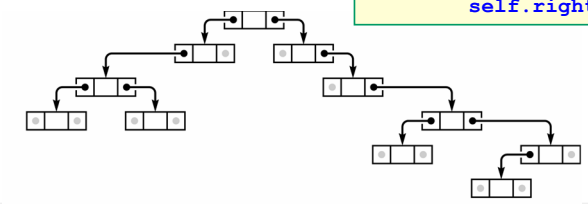
2. Induktionsschritt: h → h' = h + 1

Höhe h	Höhe h' = h + 1
2^{h-1} Blätter	$2 \cdot 2^{h-1} = 2^h = 2^{h'-1}$ Blätter → Beh.
2^h-1 Knoten	2^h-1 innere Knoten + 2^h Blätter = $2^{h+1}-1 = 2^{h'}-1$

Implementierung eines Baumes (in Python)

- Ein Knoten hat (mind.) 3 Instanzvariablen:
  - Eine Referenz zu item (*payload data*)
  - Eine Referenz zum linken Unterbaum
  - Eine Referenz zum rechten Unterbaum

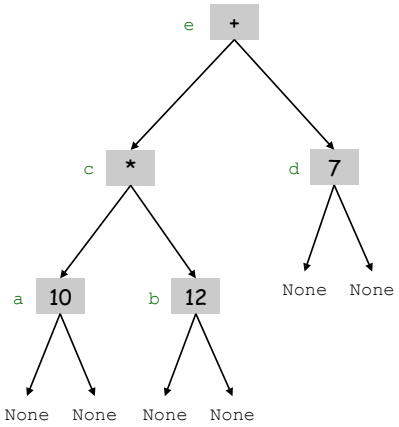
```
class Tree:
    def __init__( self, item,
                  left = None,
                  right = None):
        self.item = item
        self.left = left
        self.right = right
```



Anwendung: Parse-Tree von Ausdrücken

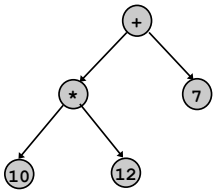
- Parse-Tree = Abstrakte Repräsentation von Ausdrücken
- Anwendung: Compiler, Computerlinguistik, ...
- Beispiel für den Aufbau:

```
a = Tree(10)
b = Tree(12)
c = Tree("*",a,b)
d = Tree(7)
e = Tree("+",c,d)
```



Auswertung eines Ausdruckes mit Hilfe von Parse-Trees

- Auswertung eines Parse-Tree:
  - Wenn Knoten ein Integer ist → Wert = Integer
  - Sonst, werte rekursiv beide Unterbäume aus und gebe die Summe oder das Produkt aus (je nach Operator, der im Knoten gespeichert ist)




```
((10 * 12) + (7)) = 127
```

```
class Tree:
    def eval(self):
        if self.item == "+" :
            return self.left.eval() + self.right.eval()
        elif self.item == "*" :
            return self.left.eval() * self.right.eval()
        else
            return self.item
```

Baumtraversierungen: Preorder-Traversierung

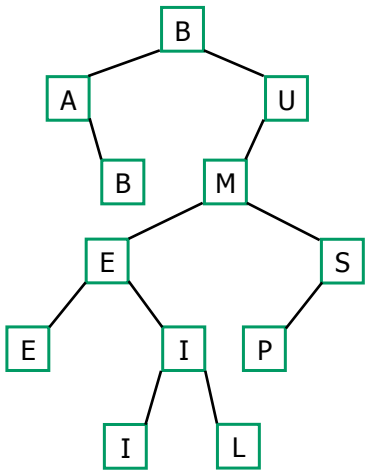
- Reihenfolge:
  1. Besuche die Wurzel, führe Operation an Wurzel durch
  2. Traversiere den linken Teilbaum (in Preorder-Reihenfolge)
  3. Traversiere den rechten Teilbaum (in Preorder)
- Implementierung :

```
class Tree (cont'd) …

    def preorder( self, params ):
        do something with node self
        if self.left != None:
            self.left.preorder(params)
        if self.right != None :
            self.right.preorder(params)

tree.preorder( params )
```


- Beispiel



Preorder Traversierung:

BABUMEEIILSP

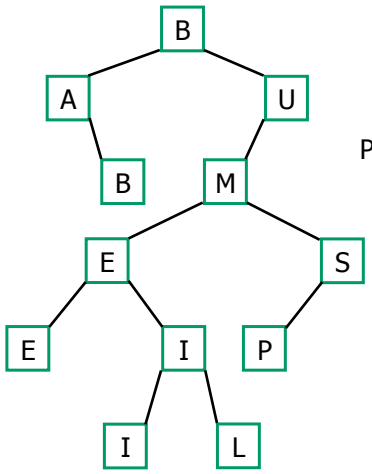
Postorder-Traversierung

- Reihenfolge:
  - Traversiere den linken Teilbaum (in Postorder)
  - Traversiere den rechten Teilbaum (in Postorder)
  - "Besuche" den Knoten selbst = führe die Operation durch
- Implementierung :

```
class Tree (cont'd) …

    def postorder( self, params):
        if self.left:
            self.left.postorder(params)
        if self.right:
            self.right.postorder(params)
        do something with node self
```


- Beispiel:



Postorder Traversierung:

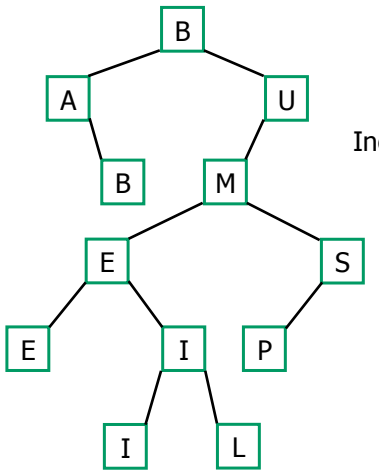
BAEILIEPSMUB

Inorder-Traversierung

- Reihenfolge:
  1. Traversiere den linken Teilbaum (in Inorder)
  2. "Besuche" den Knoten, Operation darauf durchführen
  3. Traversiere den rechten Teilbaum (in Inorder)

```
def inorder(self, params):
    if self.left:
        self.left.inorder(params)
    do something with node self
    if self.right:
        self.right.inorder(params)
```


- Beispiel

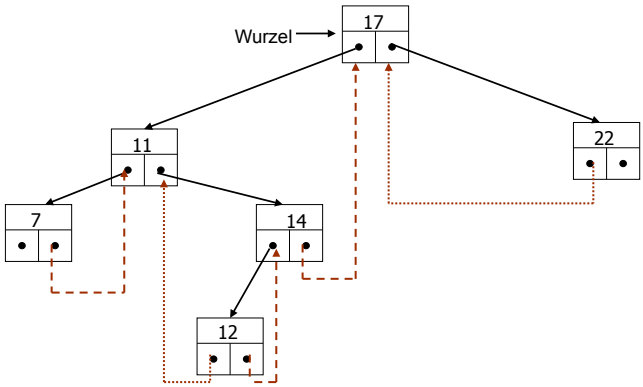


Inorder Traversierung:

ABBEEIILMPSU

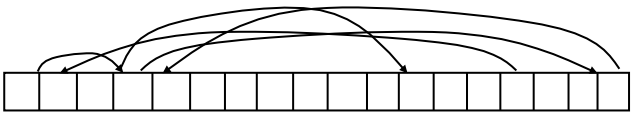
Nicht-rekursive Varianten mit *threaded trees*

- 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



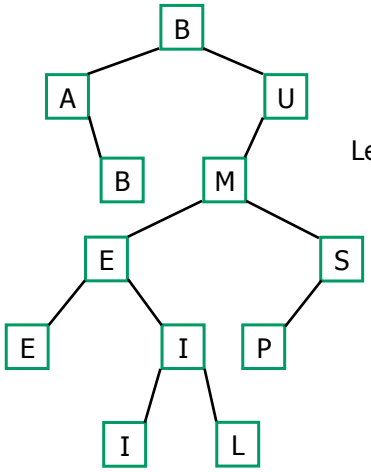
- relativ harmlos, falls sich alle Daten im Hauptspeicher befinden
  - aber: schlechte Ausnutzung des Caches
- Katastrophe, falls Daten auf Festplatte oder Magnetband
  - siehe 2-3-4-Bäume, B-Bäume, Rot-Schwarz-Bäume

Levelorder-Traversierung (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, params):
    q = Queue()
    q.enqueue(self)
    while not q.empty():
        n = q.dequeue()
        if n != None:
            do something with node n
            q.enqueue(n.left)
            q.enqueue(n.right)
```


- Beispiel



Levelorder Traversierung:

BAUBMESEIPIL

Exkurs: Visitor Pattern

- Oftmals muß man verschiedene Operationen (= ***do something with node*** im Code) auf dem Baum ausführen
- Bisherige (naive) Implementierung würde jedesmal eine neue **`preorder`**-Methode benötigen
- Auslagerung der Operationen außerhalb der Traversierungs-methoden ist die einfachste Form des sog. Visitor Patterns
- Klasse **`DoSomethingWithANode`** heißt Visitor, weil diese jeden Knoten "besucht"
- Methode **`preorder/postorder`** heißt Mapper , weil diese die Operation (**`DoSomethingWithANode.visit`**) auf jeden Knoten anwenden (mappen), und wissen, in welcher Reihenfolge dies geschehen soll

Beispiel

- Ansatz: diese Operation in eine sog. Visitor-Klasse verpacken, z.B.

```
class DoSmethingWithANode(object):
    def __init__( self, params ):
        . . .
    def visit( self, treenode ):
        do something with treenode.item
```

Beispiel

- Instanz davon kann dann der allgemeinen Traversierungsmethode als Parameter übergeben werden:

```
class Tree (object):
    ...
    def preorder( self, visitor ):
        visitor.visit(self)
        if self.left != None:
            self.left.preorder(visitor)
        if self.right != None :
            self.right.preorder(visitor)

doIt = DoSmethingWithANode( params )
tree.preorder( doIt )
```


- 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):
        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(false)
r.preorder(v)          # print all nodes in preorder
v = PrintNode(true)
r.preorder(v)          # again, but all in lowercase
```

- Vorteil der Trennung in Visitor-Klasse und Baum-Traversierungsmethode:
  - man muß Traversierungsroutine nur 1x schreiben
  - man kann trotzdem beliebige Operationen ausführen lassen
- Beispiel: andere Operation, z.B. alle Knoten in eine 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
```