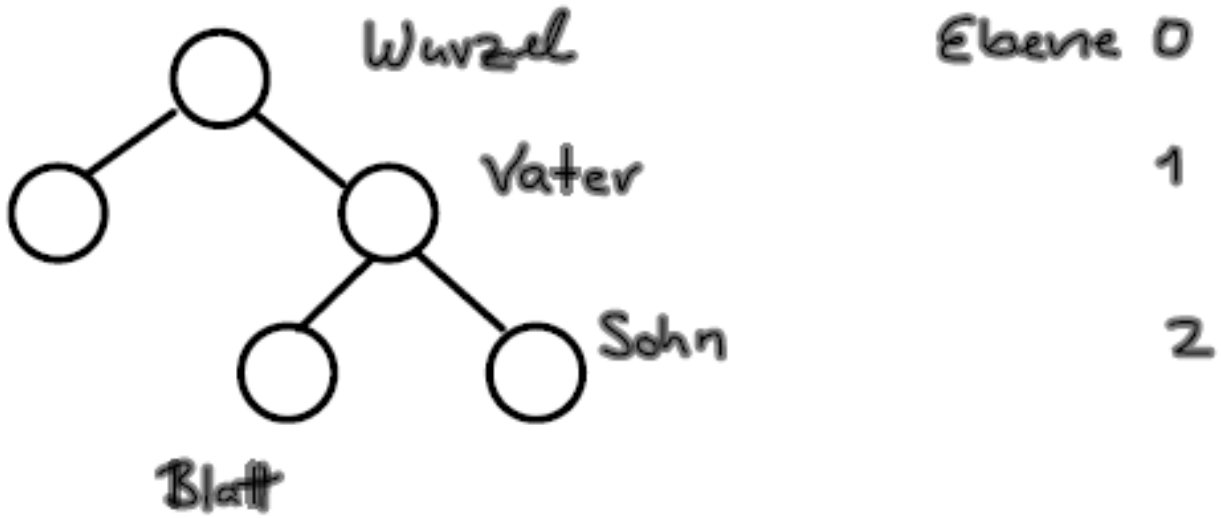
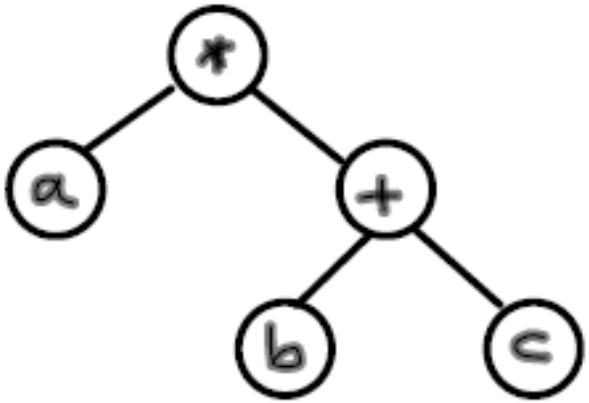
Informatik

Abstrakte Datentypen - Baum

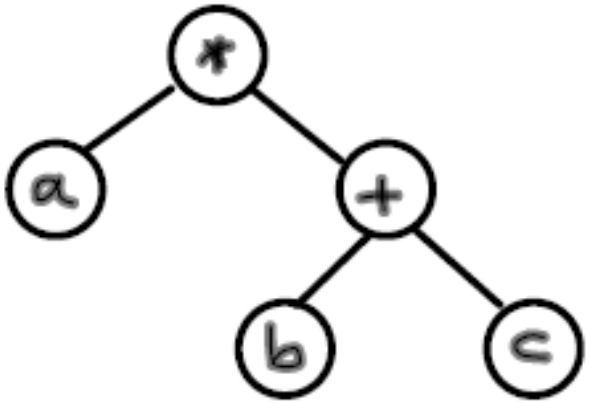
Ein *binärer Baum* ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind.



In den Knoten eines Baums können beliebige Objekte gespeichert werden.



In den Knoten eines Baums können beliebige Objekte gespeichert werden.



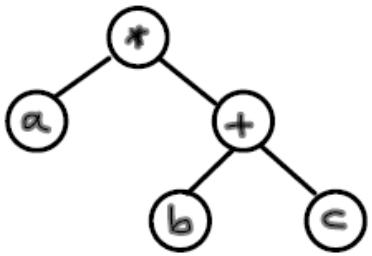
entspricht dem arithmetischen Ausdruck: $a \cdot (b + c)$

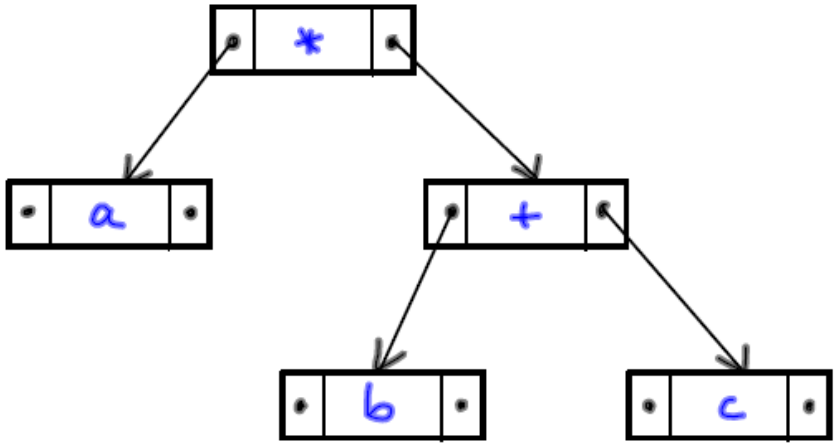
Schnittstelle des ADT Baum:

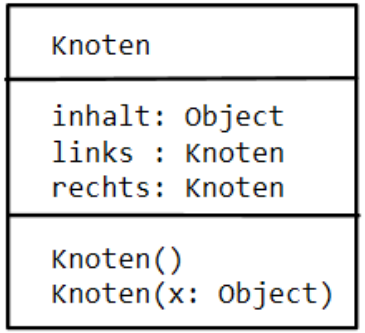
empty	:	liefert true, falls Baum leer
left	:	liefert linken Teilbaum
right	:	liefert rechten Teilbaum
value	:	liefert Wurzelelement

In der Schnittstelle sind keine Methoden für das Wachstum des Baumes vorgesehen. Wir nutzen dazu den Konstruktor.

Implementation mittels verzeigerter Knoten

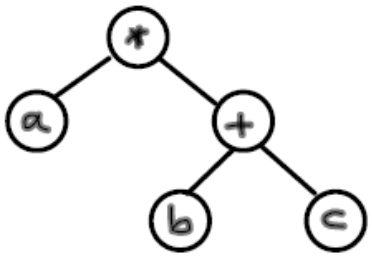


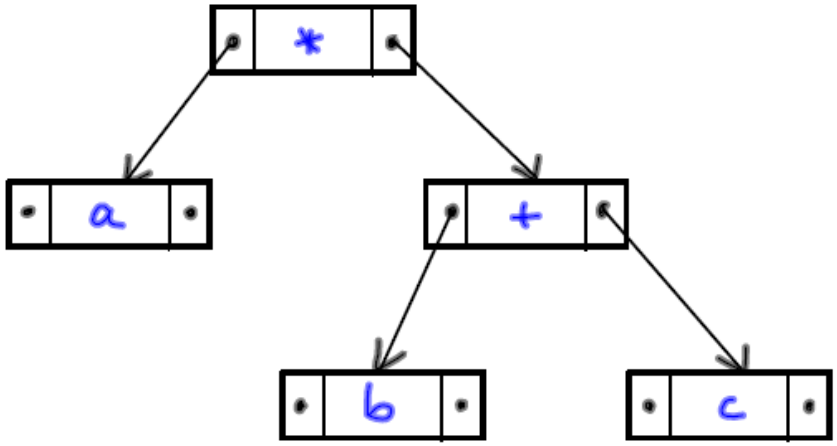


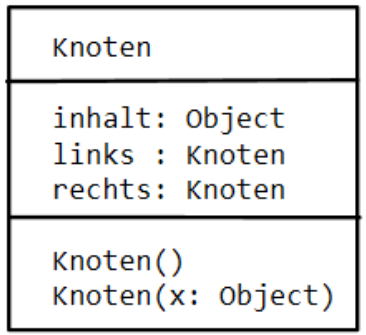



```
class Knoten:
```

Implementation mittels verzeigerter Knoten





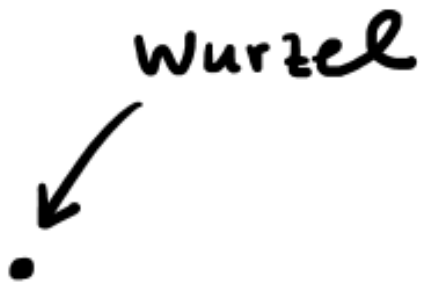



```
class Knoten:
    def __init__(self, x = None):
        self.inhalt = x
        self.links = None
        self.rechts = None
```

Der Baum hat zur internen Verwaltung nur einen Zeiger auf die wurzel.

Baum
wurzel: Knoten
Baum() Baum(x: Object) Baum(x: Object, l Baum, r Baum) empty(): boolean value(): Object left(): Baum right(): Baum

Mit `b = Baum()` soll ein leerer Baum entstehen.

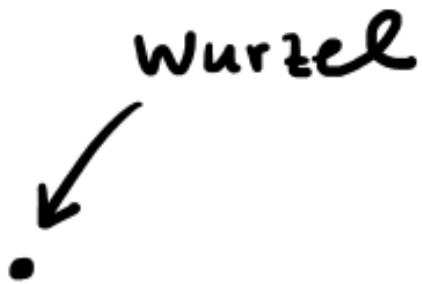



```
def __init__(self):
```

Der Baum hat zur internen Verwaltung nur einen Zeiger auf die wurzel.

Baum
wurzel: Knoten
Baum() Baum(x: Object) Baum(x: Object, l Baum, r Baum) empty(): boolean value(): Object left(): Baum right(): Baum

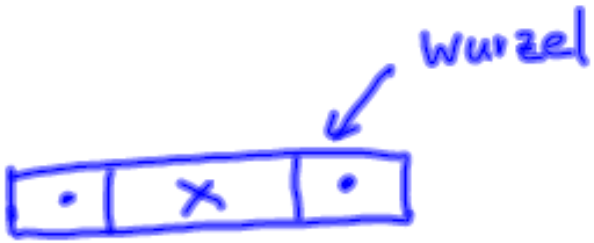
Mit `b = Baum()` soll ein leerer Baum entstehen.




```
def __init__(self):
    self.wurzel = None
```

Ein Objekt x soll in der Wurzel des Baums gespeichert werden:

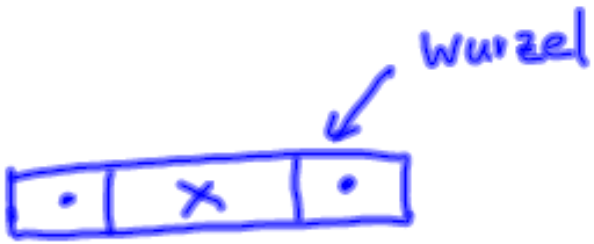
```
b = Baum(x)
```




```
def __init__(self,
```

Ein Objekt x soll in der Wurzel des Baums gespeichert werden:

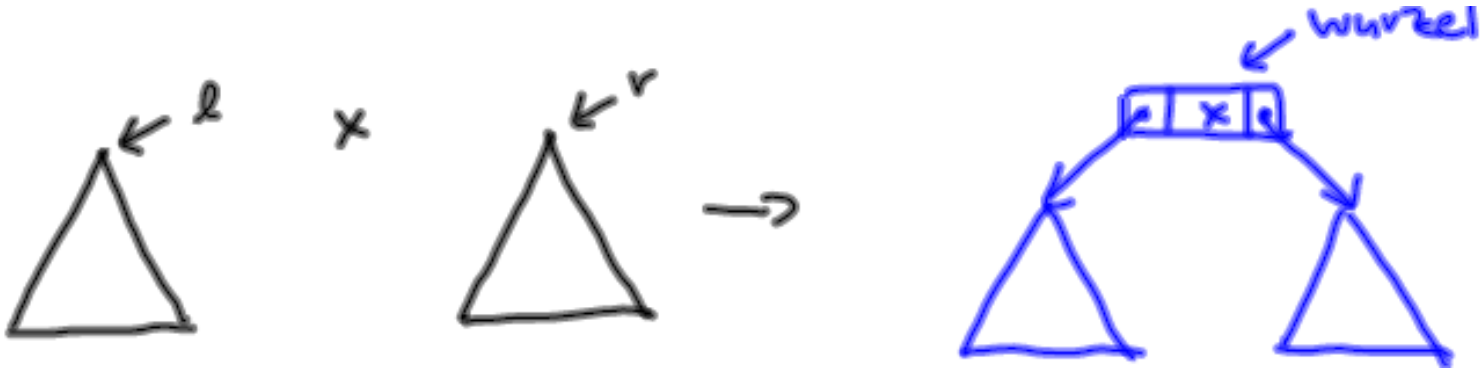
```
b = Baum(x)
```




```
def __init__(self,x = None):
    self.wurzel = None
    if x is not None:
        self.wurzel = Knoten(x)
```

Aus zwei Bäumen und einem Objekt x soll ein neuer Baum mit x in der Wurzel und den beiden Bäumen als linker und rechter Teilbaum geschaffen werden.

```
b = Baum(x, l, r)
```




```
def __init__(self,x = None,
```

Aus zwei Bäumen und einem Objekt x soll ein neuer Baum mit x in der Wurzel und den beiden Bäumen als linker und rechter Teilbaum geschaffen werden.

```
b = Baum(x, l, r)
```


```
def __init__(self,x = None,l = None,r = None):
    self.wurzel = None
    if x is not None:
        self.wurzel = Knoten(x)
    if l is not None:
        self.wurzel.links = l.wurzel
    if r is not None:
        self.wurzel.rechts = r.wurzel
```

```
class Baum:
    def __init__(self,x = None,l = None,r = None):
        self.wurzel = None
        if x is not None:
            self.wurzel = Knoten(x)
        if l is not None:
             self.wurzel.links = l.wurzel
        if r is not None:
             self.wurzel.rechts = r.wurzel

    def empty(self):
```

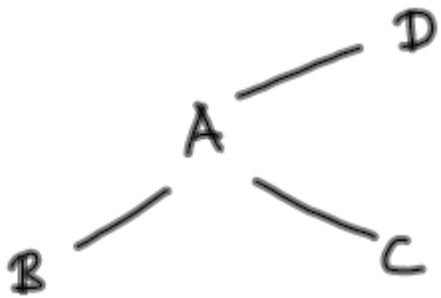
```
class Baum:
    def __init__(self,x = None,l = None,r = None):
        self.wurzel = None
        if x is not None:
            self.wurzel = Knoten(x)
        if l is not None:
             self.wurzel.links = l.wurzel
        if r is not None:
             self.wurzel.rechts = r.wurzel

    def empty(self):
        return self.wurzel is None

    def value(self):
```

```
class Baum:
    def __init__(self,x = None,l = None,r = None):
        self.wurzel = None
        if x is not None:
            self.wurzel = Knoten(x)
        if l is not None:
             self.wurzel.links = l.wurzel
        if r is not None:
             self.wurzel.rechts = r.wurzel

    def empty(self):
        return self.wurzel is None

    def value(self):
        if self.empty(): raise RuntimeError("Fehler: Baum ist leer")
        return self.wurzel.inhalt

    def left(self):
```

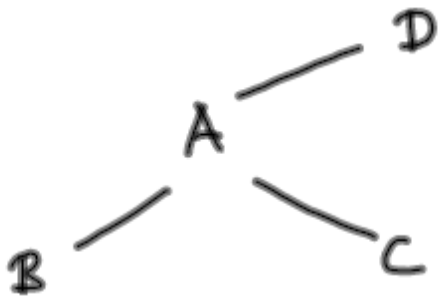
```
class Baum:
    def __init__(self,x = None,l = None,r = None):
        self.wurzel = None
        if x is not None:
            self.wurzel = Knoten(x)
        if l is not None:
             self.wurzel.links = l.wurzel
        if r is not None:
             self.wurzel.rechts = r.wurzel

    def empty(self):
        return self.wurzel is None

    def value(self):
        if self.empty(): raise RuntimeError("Fehler: Baum ist leer")
        return self.wurzel.inhalt

    def left(self):
        if self.empty(): raise RuntimeError("Fehler: Baum ist leer")
        temp = Baum()
        temp.wurzel = self.wurzel.links
        return temp

    def right(self):
```

```
class Baum:
    def __init__(self,x = None,l = None,r = None):
        self.wurzel = None
        if x is not None:
            self.wurzel = Knoten(x)
        if l is not None:
             self.wurzel.links = l.wurzel
        if r is not None:
             self.wurzel.rechts = r.wurzel

    def empty(self):
        return self.wurzel is None

    def value(self):
        if self.empty(): raise RuntimeError("Fehler: Baum ist leer")
        return self.wurzel.inhalt

    def left(self):
        if self.empty(): raise RuntimeError("Fehler: Baum ist leer")
        temp = Baum()
        temp.wurzel = self.wurzel.links
        return temp

    def right(self):
        if self.empty(): raise RuntimeError("Fehler: Baum ist leer")
        temp = Baum()
        temp.wurzel = self.wurzel.rechts
        return temp
```

Erzeuge folgenden Baum :



Erzeuge folgenden Baum :

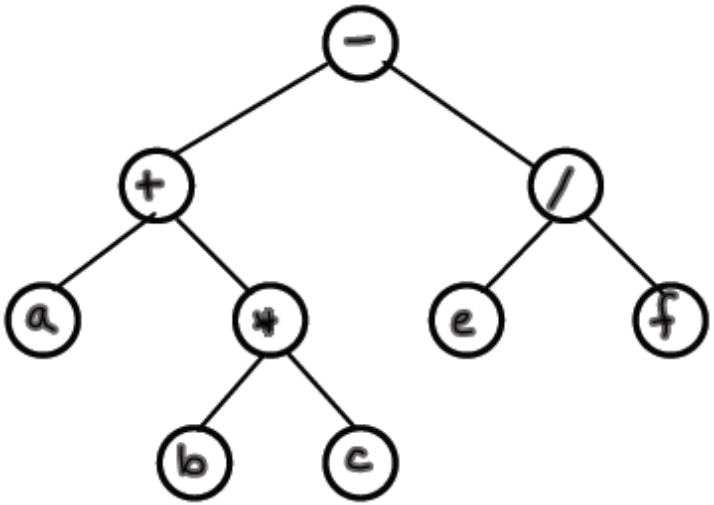



```
a = Baum('a',Baum('b'),Baum('c'))
d = Baum('d',a,None)
```

Einen Baum *traversieren* bedeutet, jeden seiner Knoten besuchen.

Verschiedene Reihenfolgen:

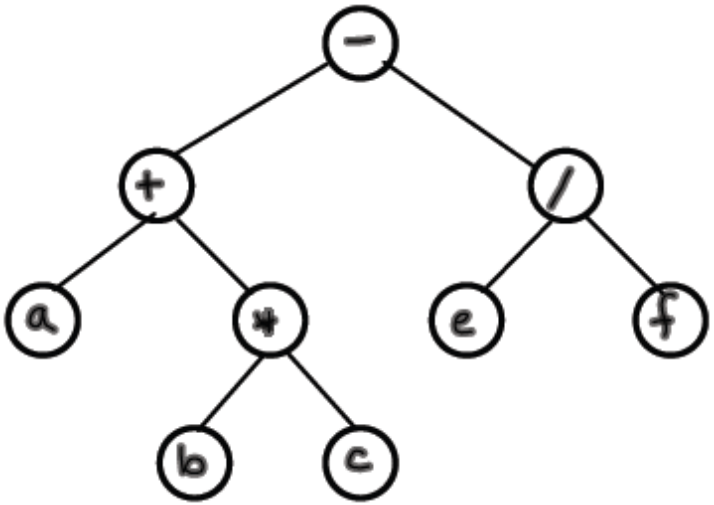
- postorder - linker Sohn, rechter Sohn, Vater
- inorder - linker Sohn, Vater, rechter Sohn
- preorder - Vater, linker Sohn, rechter Sohn



Einen Baum *traversieren* bedeutet, jeden seiner Knoten besuchen.

Verschiedene Reihenfolgen:

- postorder - linker Sohn, rechter Sohn, Vater
- inorder - linker Sohn, Vater, rechter Sohn
- preorder - Vater, linker Sohn, rechter Sohn

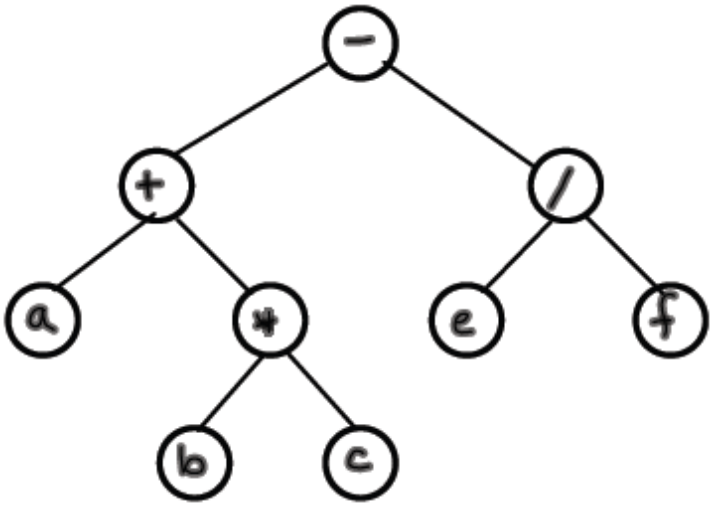


postorder:

Einen Baum *traversieren* bedeutet, jeden seiner Knoten besuchen.

Verschiedene Reihenfolgen:

- postorder - linker Sohn, rechter Sohn, Vater
- inorder - linker Sohn, Vater, rechter Sohn
- preorder - Vater, linker Sohn, rechter Sohn

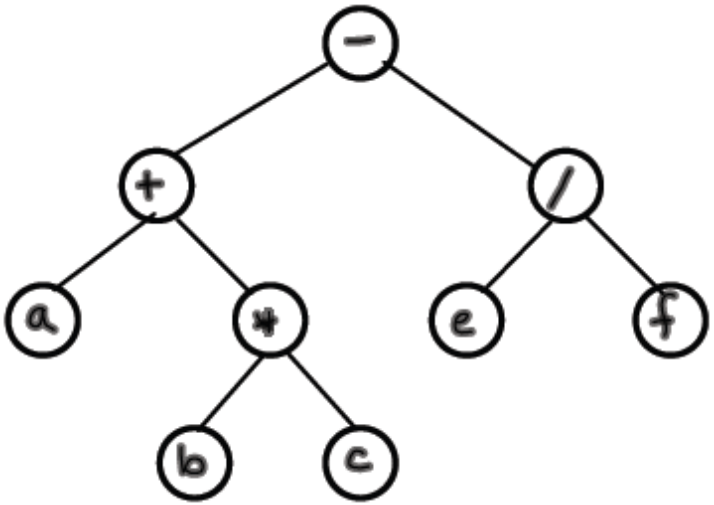


postorder: a b c * + e f / -
inorder:

Einen Baum *traversieren* bedeutet, jeden seiner Knoten besuchen.

Verschiedene Reihenfolgen:

- postorder - linker Sohn, rechter Sohn, Vater
- inorder - linker Sohn, Vater, rechter Sohn
- preorder - Vater, linker Sohn, rechter Sohn

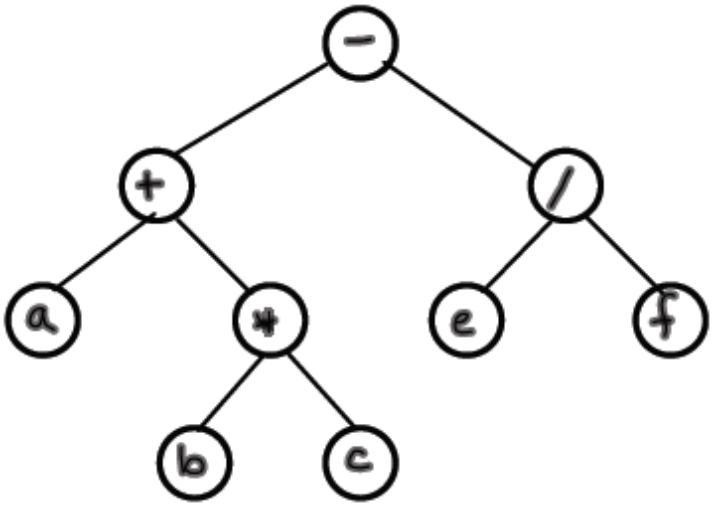


postorder: a b c * + e f / -
inorder: a + b * c - e / f
preorder:

Einen Baum *traversieren* bedeutet, jeden seiner Knoten besuchen.

Verschiedene Reihenfolgen:

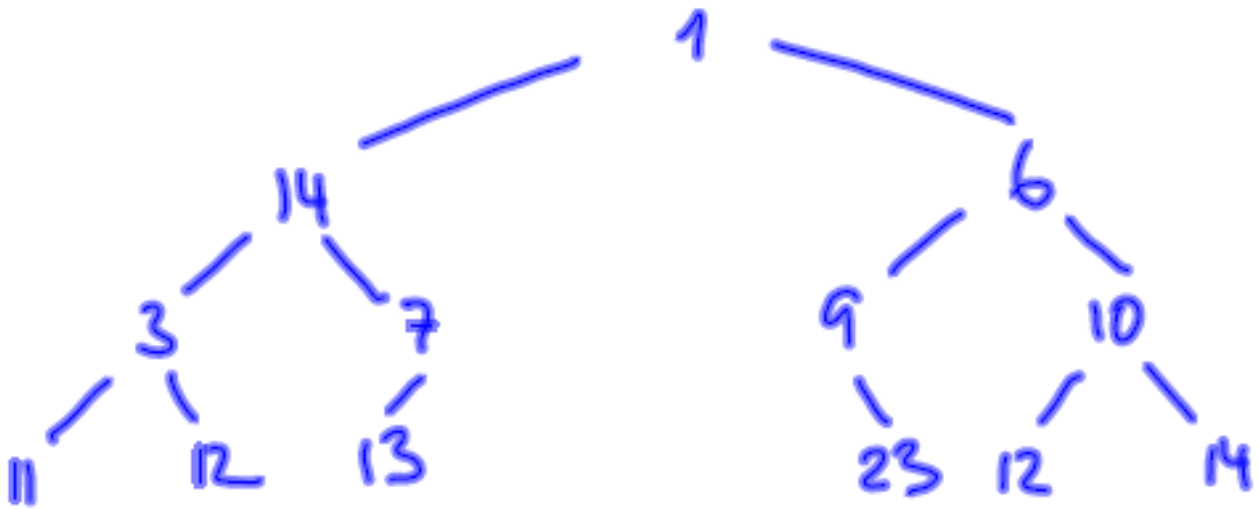
- postorder - linker Sohn, rechter Sohn, Vater
- inorder - linker Sohn, Vater, rechter Sohn
- preorder - Vater, linker Sohn, rechter Sohn



postorder: a b c * + e f / -
inorder: a + b * c - e / f
preorder: - + a * b c / e f

Gib die Reihenfolge in postorder, inorder und preorder aus

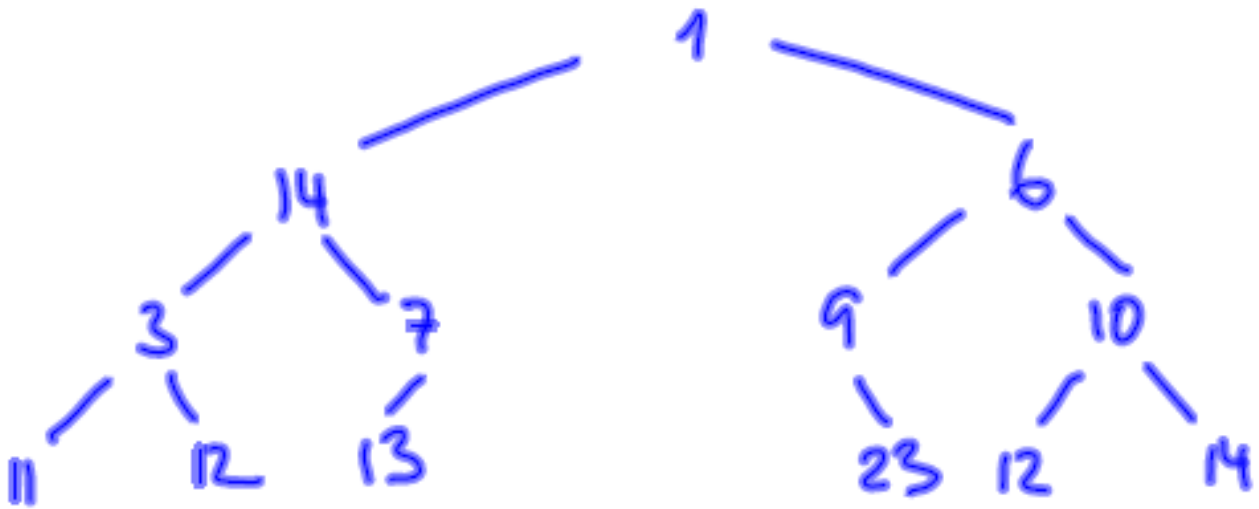
```
...14
..10
...12
.6
...23
..9
1
..7
...13
.14
...12
..3
...11
```



postorder:

Gib die Reihenfolge in postorder, inorder und preorder aus

```
...14
..10
...12
.6
...23
..9
1
..7
...13
.14
...12
..3
...11
```

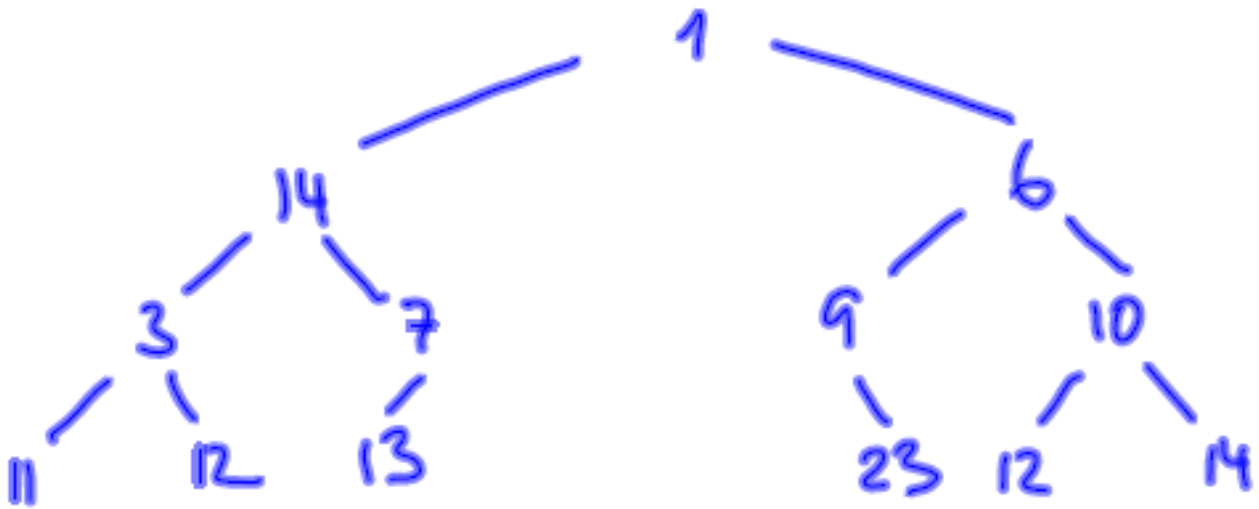


postorder: 11 12 3 13 7 14 23 9 12 14 10 6 1

inorder:

Gib die Reihenfolge in postorder, inorder und preorder aus

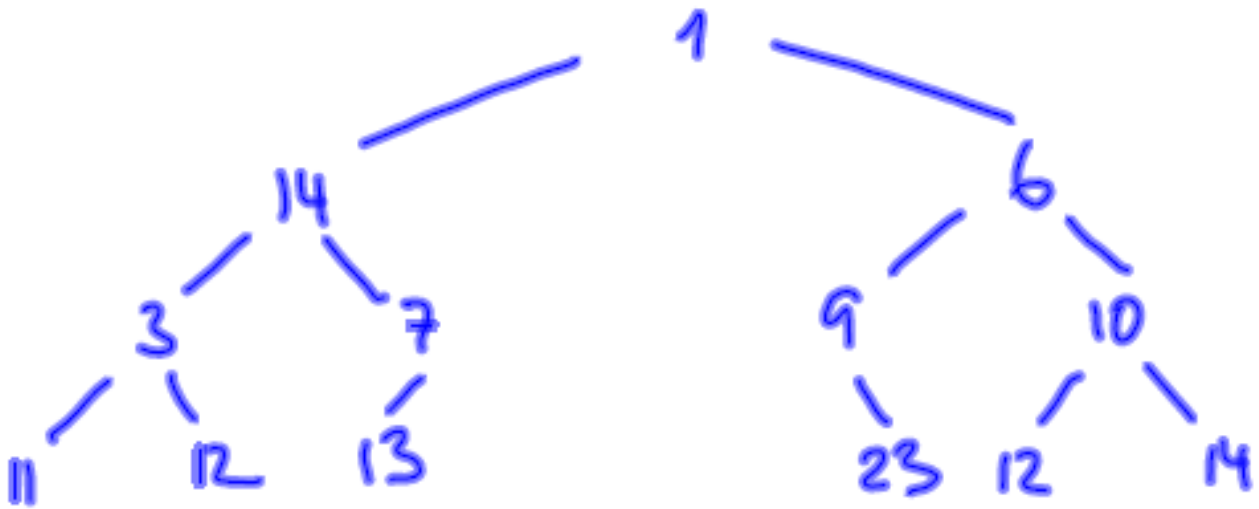
```
...14
..10
...12
.6
...23
..9
1
..7
...13
.14
...12
..3
...11
```



postorder: 11 12 3 13 7 14 23 9 12 14 10 6 1
inorder: 11 3 12 14 13 7 1 9 23 6 12 10 14
preorder:

Gib die Reihenfolge in postorder, inorder und preorder aus

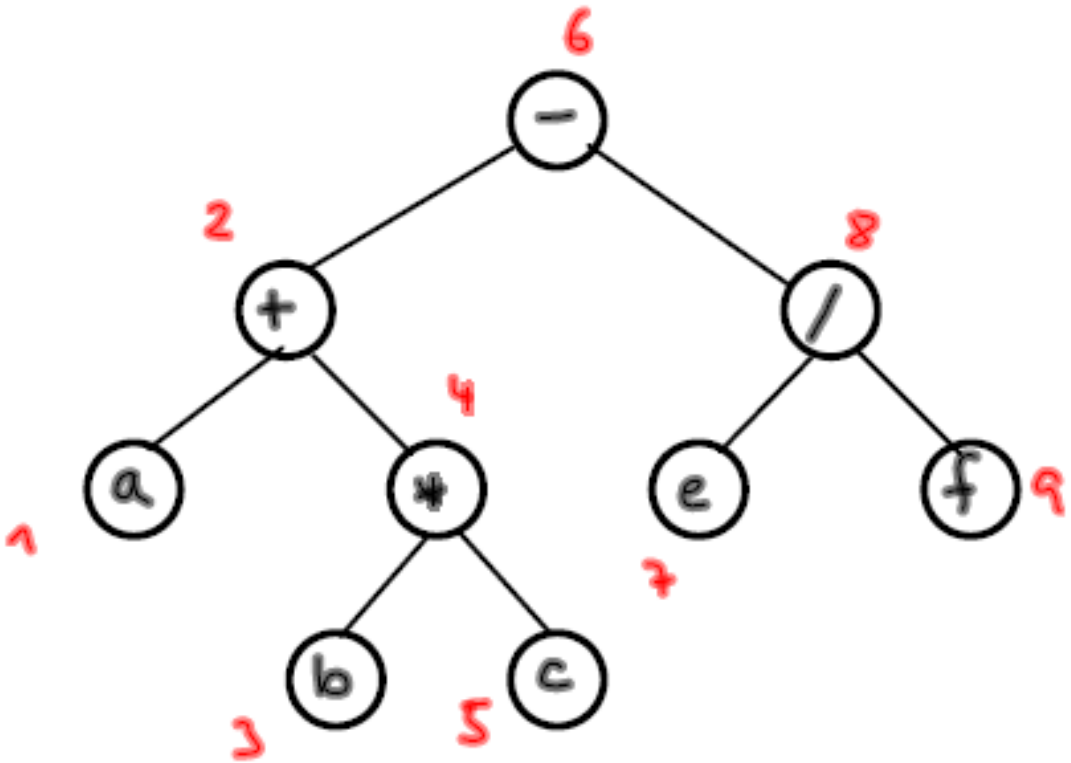
```
...14
..10
...12
.6
...23
..9
1
..7
...13
.14
...12
..3
...11
```



postorder: 11 12 3 13 7 14 23 9 12 14 10 6 1
inorder: 11 3 12 14 13 7 1 9 23 6 12 10 14
preorder: 1 14 3 11 12 7 13 6 9 23 10 12 14

Die richtige Reihenfolge zu programmieren, scheint nicht einfach zu sein.

Beispiel inorder:



Ein Baum lässt sich mittels rekursiver Methoden traversieren.

```
def inorder(b):
```

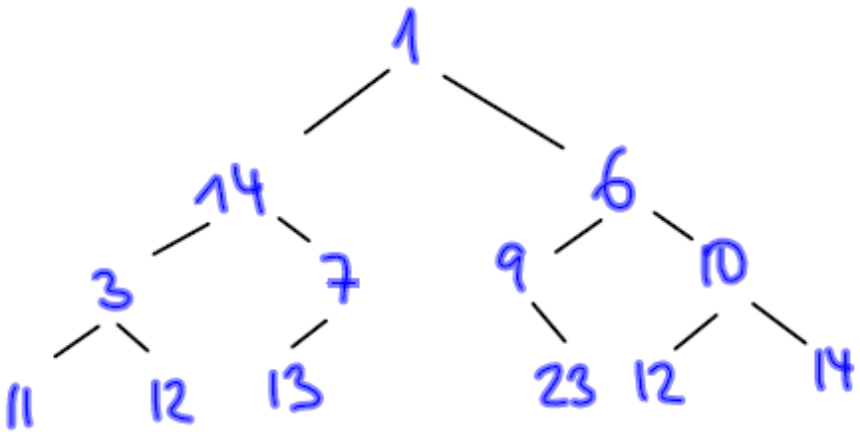
Ein Baum lässt sich mittels rekursiver Methoden traversieren.

```
def inorder(b):
    if b.empty(): return
    inorder(b.left())
    print(b.value(),end=" ")
    inorder(b.right())

def preorder(b):
```

Ein Baum lässt sich mittels rekursiver Methoden traversieren.

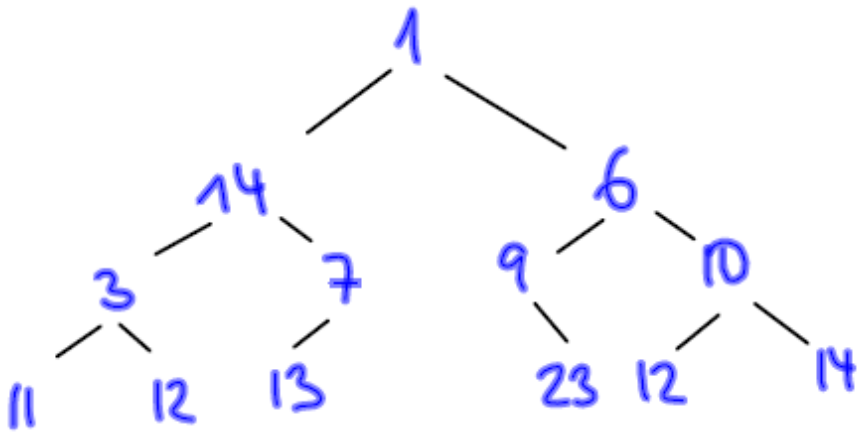
```
def inorder(b):
    if b.empty(): return
    inorder(b.left())
    print(b.value(),end=" ")
    inorder(b.right())

def preorder(b):
    if b.empty(): return
    print(b.value(),end=" ")
    preorder(b.left())
    preorder(b.right())

def postorder(b):
    if b.empty(): return
    postorder(b.left())
    postorder(b.right())
    print(b.value(),end=" ")
```

Die iterative Tiefensuche (preorder) wird mit einem Keller organisiert.
Wir laufen nach links unten und legen die rechten Bäume, die uns unterwegs begegnen, in den Keller.

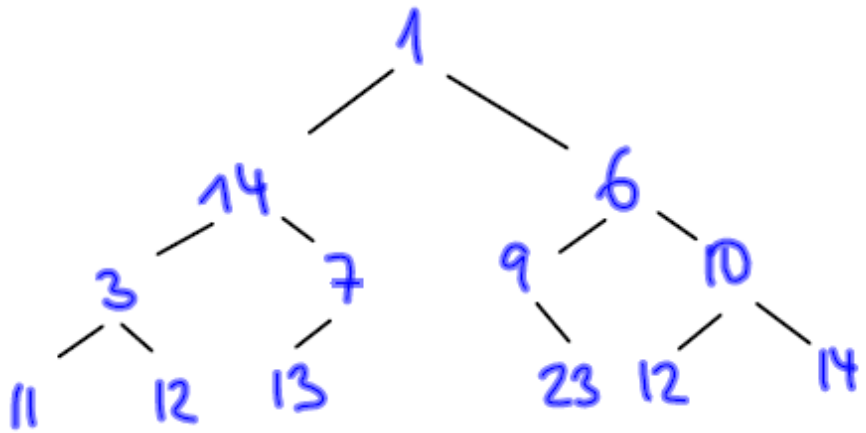


Ausgabe	Keller
	1

Die iterative Tiefensuche (preorder) wird mit einem Keller organisiert.
Wir laufen nach links unten und legen die rechten Bäume, die uns unterwegs begegnen, in den Keller.



Die iterative Tiefensuche (preorder) wird mit einem Keller organisiert.
Wir laufen nach links unten und legen die rechten Bäume, die uns unterwegs begegnen, in den Keller.



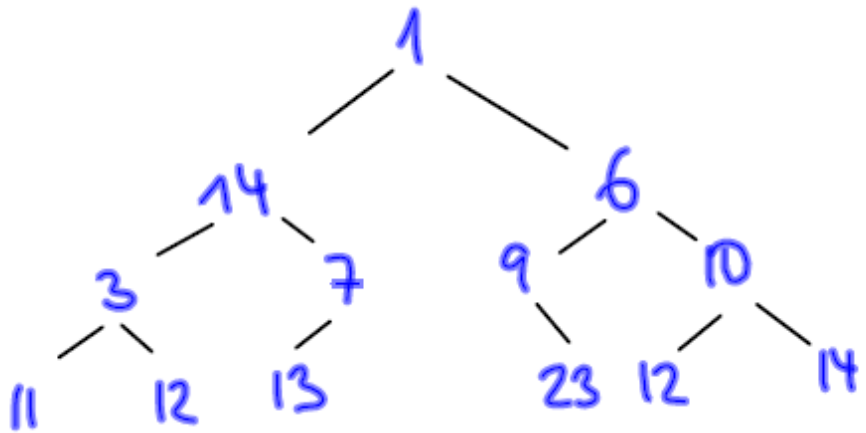
Ausgabe	Keller
	1
1	6
14	7 6
3	12 7 6
11	12 7 6
12	7 6
7	6
13	6
6	10
9	23 10
23	10
10	14
12	14
14	-

```
def tiefenSuche(baum):
```

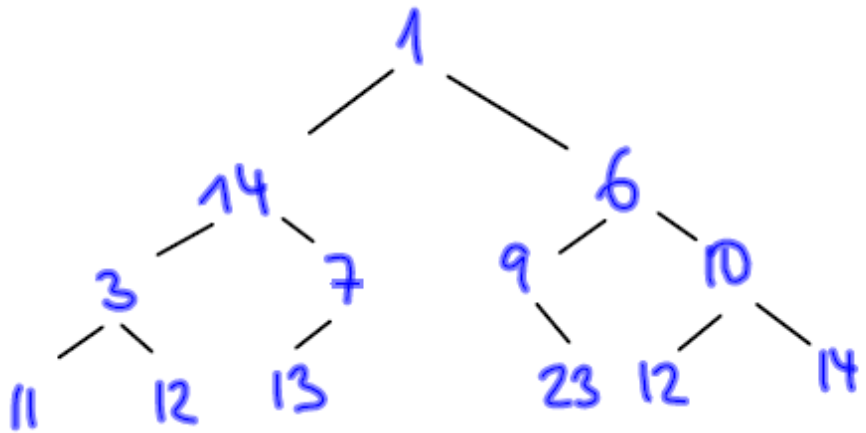
```
def tiefenSuche(baum):
    k = Keller()
    if not baum.empty():
        k.push(baum)
    while not k.empty():
        b = k.top()
        k.pop()
        while not b.empty():
            print(b.value(),end=' ')
            if not b.right().empty():
                k.push(b.right())
            b = b.left()
    print()
```

Ausgabe	Schlange
	1

Bei der Breitensuche werden die Inhalte der Knoten ebenenweise ausgegeben. Sie wird mit einer Schlange organisiert.
Wir reihen die linken und rechten Teilbäume in eine Schlange ein.



Bei der Breitensuche werden die Inhalte der Knoten ebenenweise ausgegeben. Sie wird mit einer Schlange organisiert.
Wir reihen die linken und rechten Teilbäume in eine Schlange ein.



Ausgabe	Schlange
	1
1	14 6
14	6 3 7
6	3 7 9 10
3	7 9 10 11 12
7	9 10 11 12 13
9	10 11 12 13 23
10	11 12 13 23 12 14
11	12 13 23 12 14
12	13 23 12 14
13	23 12 14
23	12 14
12	14
14	-

```
def breitenSuche(baum):
```

```
def breitenSuche(baum):
    s = Schlange()
    if not baum.empty():
        s.enq(baum)
    while not s.empty():
        b = s.front()
        s.deq()
        print(b.value(),end=' ')
        if not b.left().empty():
            s.enq(b.left())
        if not b.right().empty():
            s.enq(b.right())
```