7.4 Traversieren von Bäumen

für Konstruktion/Auswertung,
Ein/Ausgabe,
Umwandlung zwischen linearer und nichtlinearer Repräsentation,
. . . . .

Traversieren =

„mit jedem Knoten wird etwas gemacht“ oder

„alle Knoten werden linear angeordnet“ oder

„alle Knoten werden in bestimmter Reihenfolge manipuliert“

(vgl. Traversieren von Listen !)

```
data Tree t = E | N(Tree t)t(Tree t)
```

• **ohne** bestimmte Reihenfolge: z.B.

```
instance Functor Tree where
  fmap f E = E
  fmap f(N l x r) = N(fmap f l)(f x)(fmap f r)
```

• **Tiefendurchlauf** (Tiefensuche, depth-first traversal)

```
preorder, inorder, postorder :: Tree t -> [t]
```

• **Breitendurchlauf** (Breitensuche, breadth-first traversal)

Tiefendurchlauf in **Präordnung** (preorder traversal) für Binärbäume:

erst *Wurzel*, dann linker Teilbaum, dann rechter Teilbaum

```
preorder E = []
preorder(N l x r) = [x] ++ preorder l ++ preorder r
```

Entsprechend **Inordnung** (inorder traversal),

erst linker Teilbaum, dann *Wurzel*, dann rechter Teilbaum

und **Postordnung** (postorder traversal):

erst linker Teilbaum, dann rechter Teilbaum, dann *Wurzel*

Beispiel **Operatorbaum:**

Präordnung	liefert Präfix- (Polnische) Notation
Inordnung	liefert Infix-Notation (ohne Klammern!)
Postordnung	liefert Postfix- (umgekehrte Polnische) Notation

Externe Iteratoren für Baumtraversierung mit Tiefendurchlauf:

```
class LinkedList ...
...
Iterator iterator(){.....}


}
```

```
class LinkedTree ...
...
Iterator preorder() {.....}
Iterator inorder()  {.....}
Iterator postorder(){.....}
}
```

Interne Iteratoren:

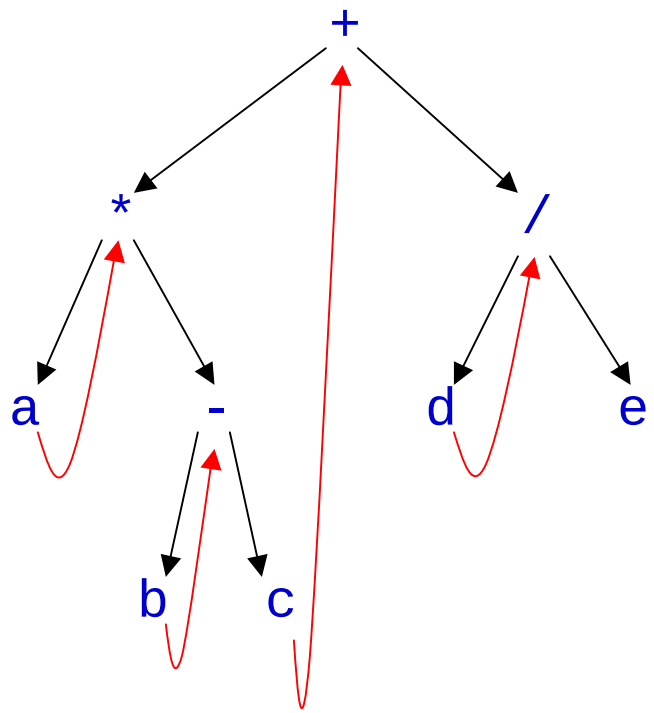
```
...
void iterate(Op op){.....}
```

```
...
void preorder(Op op) {....}
void inorder(Op op)  {....}
void postorder(Op op){....}
```

Typische Implementierung mit **nichtlinearer Rekursion**

Iteratives Traversieren (Tiefendurchlauf)

- bei linearer Präfix-Darstellung: trivial; sonst
- mittels **Keller:** nicht sofort behandelte Abzweigungen werden gekellert;
- mit aufwendigerer Repräsentation, z.B. **gefädelte Darstellung:**



Beachte: Tiefendurchlauf *nicht* anwendbar auf *unendliche Bäume* !

Breitendurchlauf eines Mehrweg*waldes* :

„erst alle Wurzeln des Waldes (Ebene 0),
dann Breitensuche auf Ebene 1“

```
data Tree t = N t [Tree t]

root(N r s) = r
subt(N r s) = s

list :: [Tree t] -> [t]

list[] = []
list forest = roots ++ list branches
              where roots = map root forest
                    branches = concat(map subt forest)
```

Typischer Aufruf: `list[mytree]`

Iterativer Breitendurchlauf:

- bei ebenenweiser Darstellung: trivial;

- sonst: unter Verwendung einer **Schlange**
zum Speichern der nicht sofort behandelten Abzweigungen:

```
Baum in leere Schlange einfügen.
Wiederholen: Baum aus Schlange entnehmen;
             Wurzel manipulieren;
             Teilbäume an Schlange anhängen;
bis Schlange leer.
```

7.5 Beispiel Übersetzung von Programmiersprachen

1. **Lexikalische Analyse:**
   Quellenprogramm in interne Darstellung umwandeln.
   Symboltabelle aufbauen.

2. **Syntaktische Analyse:**
   Quellenprogramm auf syntaktiche Korrektheit prüfen.
   Syntaxbaum aufbauen.

3. **Semantische Analyse:**
   Syntaxbaum traversieren
   zwecks statischer Typprüfung

4. **Codeerzeugung:**
   Syntaxbaum traversieren
   zwecks Erzeugung der Instruktionsfolge des Objektprogramms