Programmierung 1

Vorlesung 11

Livestream beginnt um 14:15 Uhr

Bäume
Teil 2

Programmierung 1

Reine Bäume

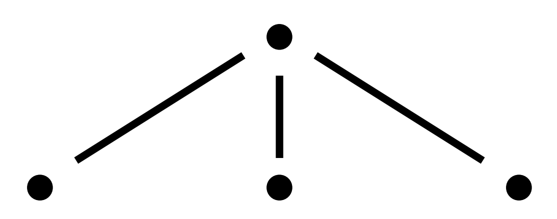
- **Idee:** "Ein Baum ist die Liste seiner Unterbäume."
- **In ML:**

```
datatype tree = T of tree list
```

- **Beispiele:**

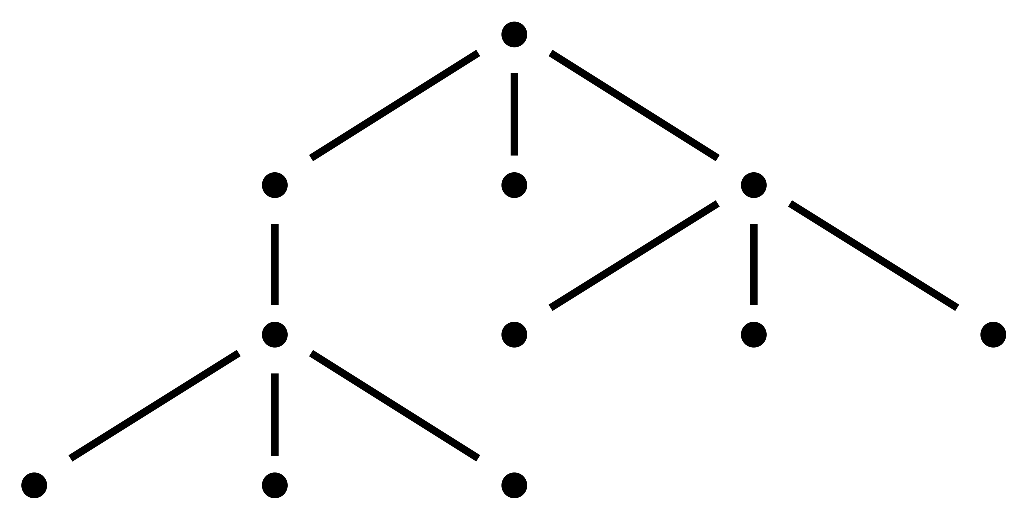
$t1 = T[]$

der atomare Baum



$t2 = T[t1, t1, t1]$

zusammengesetzter Baum



$t3 = T[T[t2], t1, t2]$

zusammengesetzter Baum

Anzahl der Knoten in einem Baum

- **Idee:** Wir berechnen den Wert für einen Baum aus einer **rekursiv** berechneten **Liste** der Werte der Unterbäume

- **Idee:** Benutze map, um die Prozedur rekursiv auf die Unterbäume anzuwenden

- **Idee:** Benutze Faltung, um aus der Liste der Werte der Unterbäume den Wert für den Baum zu berechnen.

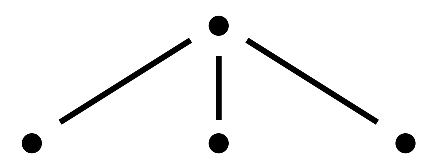
```
fun size (T ts) = foldl op+ 1 (map size ts)
```

$val\ size : tree \rightarrow int$

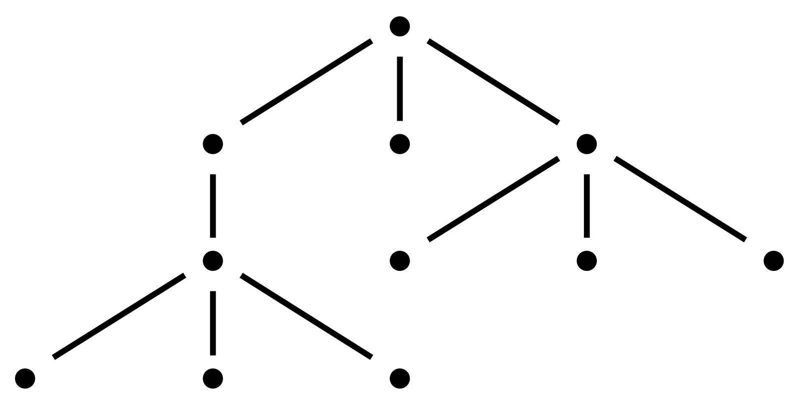
Test auf Teilbaum

```
fun subtree t (T ts) = (t = T ts) orelse
                       List.exists (subtree t) ts
```

val subtree : tree → tree → bool



$t2 = T[t1, t1, t1]$



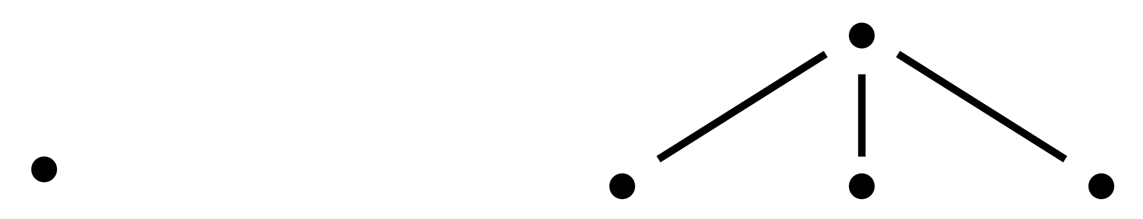
$t3 = T[T[t2], t1, t2]$

```
subtree t2 t3 = subtree t2 (T[T[t2],t1,t2])
 = false orelse exists (subtree t2) [T[t2],t1,t2]
 = exists (subtree t2) [T[t2],t1,t2]
 = subtree t2 (T[t2]) orelse exists (subtree t2) [t1,t2]
 = (false orelse exists (subtree t2) [t2])
                      orelse exists (subtree t2) [t1,t2]
 = exists (subtree t2) [t2]
                      orelse exists (subtree t2) [t1,t2]
 = (subtree t2 t2 orelse exists (subtree t2) nil)
                      orelse exists (subtree t2) [t1,t2]
 = (true          orelse …) orelse …
 = true
```

Anzahl der Auftreten eines Teilbaums

```
fun count t (T ts) = if (t= T ts) then 1
                     else foldl op+ 0 (map (count t) ts)
```

$val\ count : tree \to tree \to int$



$t1 = T[]$ $t2 = T[t1, t1, t1]$

```
count t1 t2 = count t1 (T[t1,t1,t1])
 = if t1=T[t1,t1,t1] then 1
      else foldl op+ 0 (map (count t1) [t1,t1,t1])
 = foldl op+ 0 (map (count t1) [t1,t1,t1])
 = foldl op+ 0 (map (count t1) [T[],t1,t1])
 = foldl op+ 0 ((count t1) T[])::(map (count t1) [t1,t1])
 = foldl op+ 0 (if t1=T[] then 1 else ...)
                          ::(map (count t1) [t1,t1])
 = foldl op+ 0 (1::(map (count t1) [t1,t1]))
 = …
 = foldl op+ 0 [1,1,1] = 3
```

Lexikalische Ordnung auf Listen

Die Positionen zweier anzuordnender Listen werden solange paarweise verglichen, bis die **erste Position** erreicht ist, in der **keine Gleichheit** vorliegt.
Diese Position entscheidet dann über die Anordnung der Listen.
Eine Liste, die auf allen Positionen mit einer **längeren Liste** übereinstimmt ist kleiner als die längere Liste.

```
fun lex (compare : 'a * 'a -> order) p = case p of
      (nil, _::_) => LESS
    | (nil, nil)  => EQUAL
    | (_::_, nil) => GREATER
    | (x::xr, y::yr) => case compare(x,y) of
          EQUAL => lex compare (xr,yr)
        |   s   => s
```

$lex : (\alpha * \alpha \to order) \to \alpha\ list * \alpha\ list \to order$

Lexikalische Ordnung auf Bäumen

Die Unterbäume zweier anzuordnender Bäume werden solange paarweise verglichen, bis die **erste Position** erreicht ist, bei der **keine Gleichheit** vorliegt.
Diese Position entscheidet dann über die Anordnung der Bäume.
Ein Baum, der auf allen Positionen mit einem Baum mit **mehr Unterbäumen** übereinstimmt ist kleiner.

```
fun compareTree (T nil, T nil) = EQUAL
  | compareTree (T nil, _) = LESS
  | compareTree (_, T nil) = GREATER
  | compareTree (T(t::ts),T(t'::ts')) =
                case compareTree (t,t') of
                  EQUAL => compareTree (T ts, T ts')
                | v => v
```

*val compareTree : tree * tree → order*

Beispiele

- compareTree(T[T[]], T[T[],T[]]) = LESS
- compareTree(T[T[]], T[T[]]) = EQUAL
- compareTree(T[], T[T[], T[]]) = LESS
- compareTree(T[T[], T[T[],T[]]], T[T[T[],T[]], T[]]) = LESS

```
fun compareTree (T nil, T nil) = EQUAL
  | compareTree (T nil, _) = LESS
  | compareTree (_, T nil) = GREATER
  | compareTree (T(t::ts),T(t'::ts')) =
                        case compareTree (t,t') of
                          EQUAL => compareTree (T ts, T ts')
                        | v => v
```

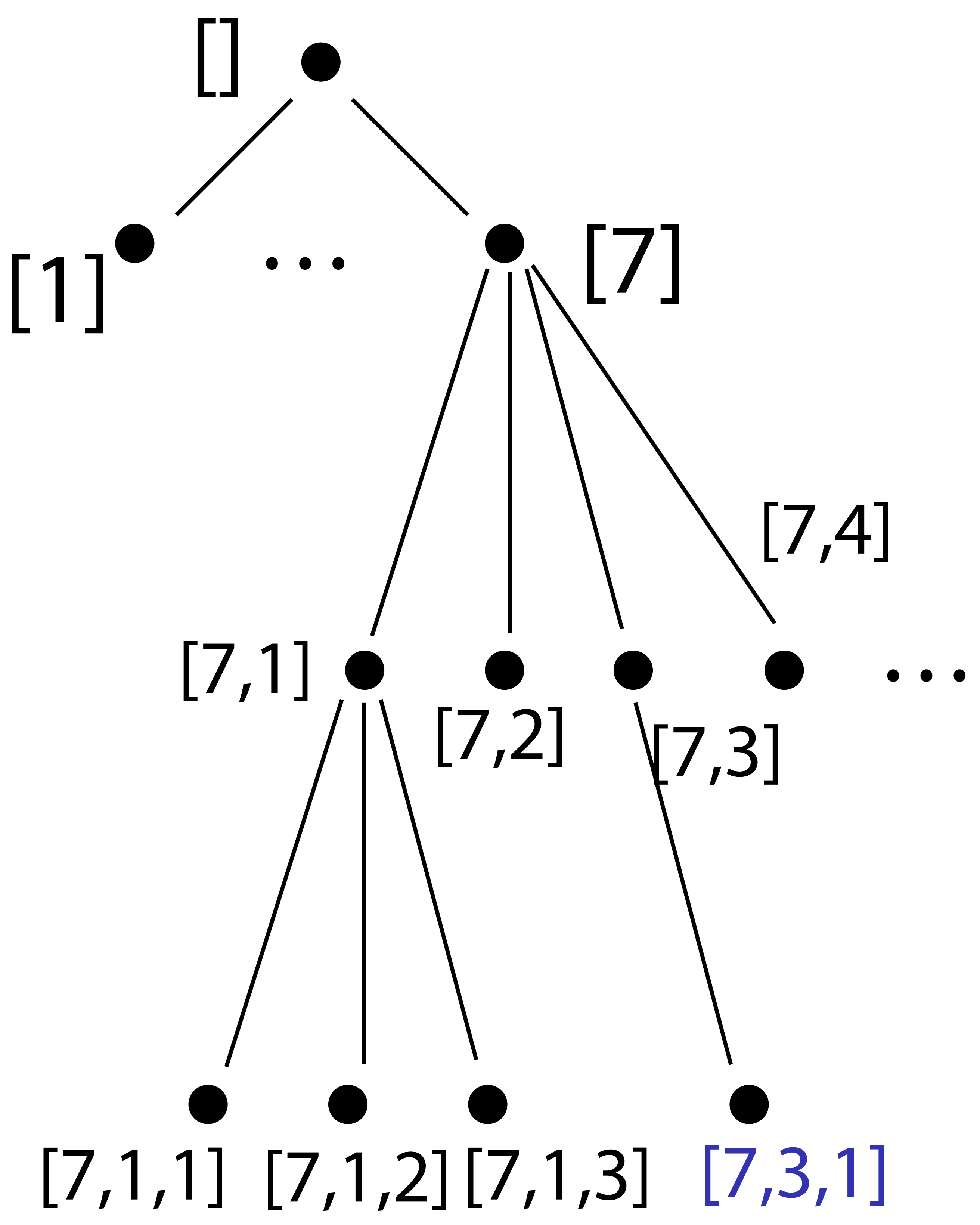
*val compareTree : tree * tree → order*

Frage

Welcher der folgenden Bäume ist der lexikalisch kleinste?

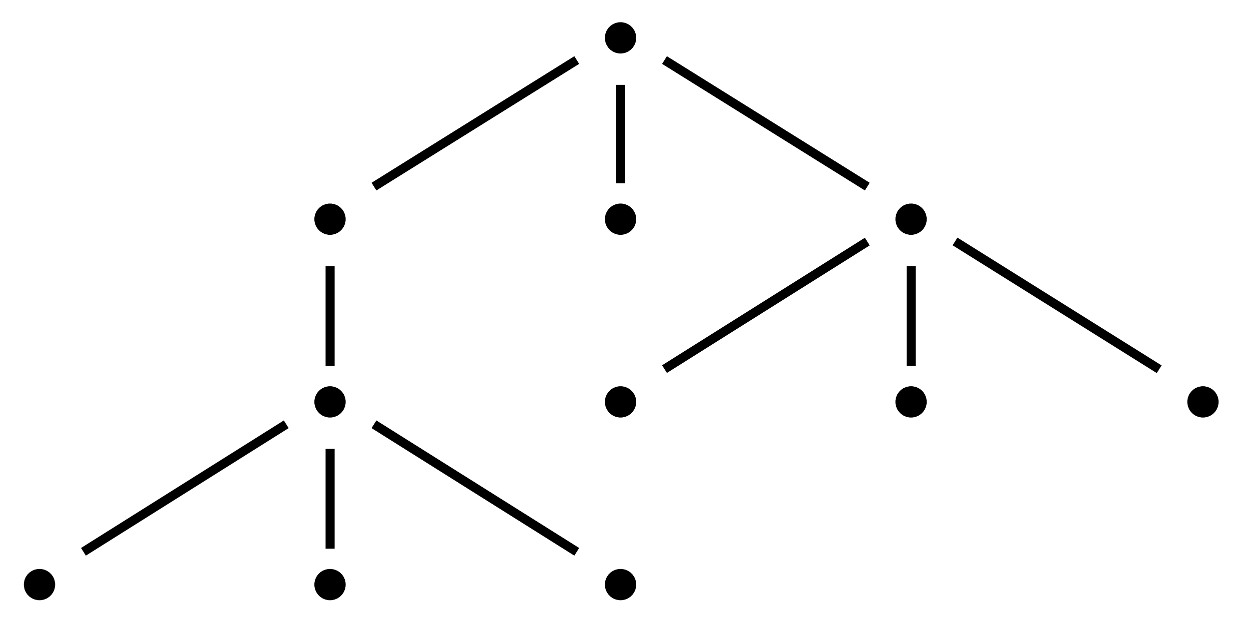
- T[T[T[]],T[T[T[],T[]]]] ✗
- T[T[T[],T[]],T[T[T[],T[]]]] ✗
- T[T[T[]],T[T[T[]],T[T[]]]] ✓
- T[T[T[T[]]]] ✗

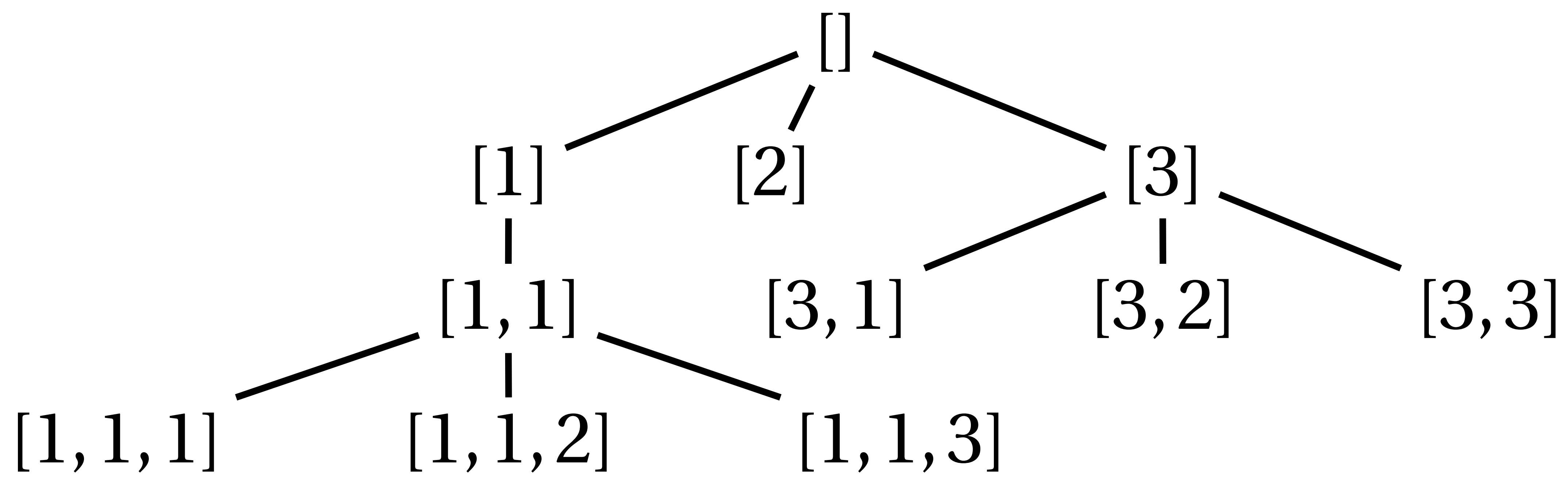
Adressen



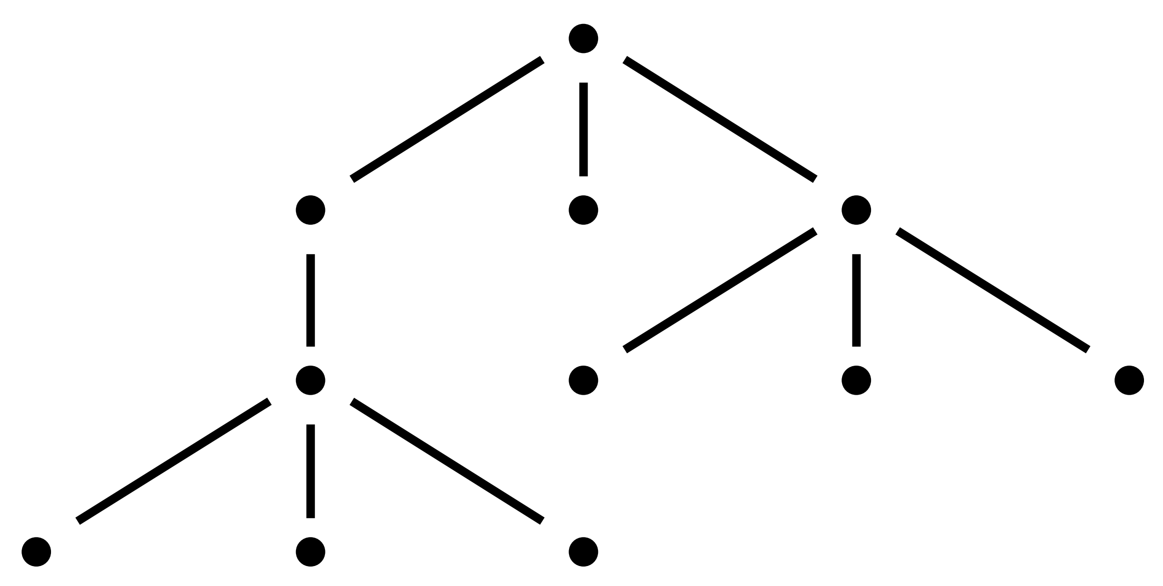
Adressen

- Die **Adresse eines Knotens** ist eine Liste ganzer Zahlen, die besagt, wie man von der Wurzel zum Knoten gelangt.
- Die Wurzel hat immer die Adresse [].
- Die Knoten werden von links nach rechts, beginnend mit 1, nummeriert.





Durch Adresse bestimmter Teilbaum



$$t3 = T[T[t2], t1, t2]$$

```
fun ast t nil = t
  | ast t (n::a) = ast (dst t n) a
```

val ast : tree → int list → tree `(* Subscript *)`

```
ast t3 [1,1,3]
```

T[] : tree

```
ast t3 [1,1]
```

T[T[], T[], T[]] : tree

Gültige Adressen

- Eine **Adresse** *a* heißt für einen Baum *t* **gültig,** wenn ast für *a* und *t* einen Teilbaum liefert.

- Ansonsten heißt die Adresse **ungültig.**

```
ast t3 [1,2]
```

! Uncaught exception: Subscript

- **Proposition:** Zwei Bäume sind genau dann **gleich**, wenn sie die **gleichen gültigen Adressen** haben.

Frage

Welche der folgenden Adressen sind gültige Adressen im Baum T[T[],T[T[],T[T[]]]]?

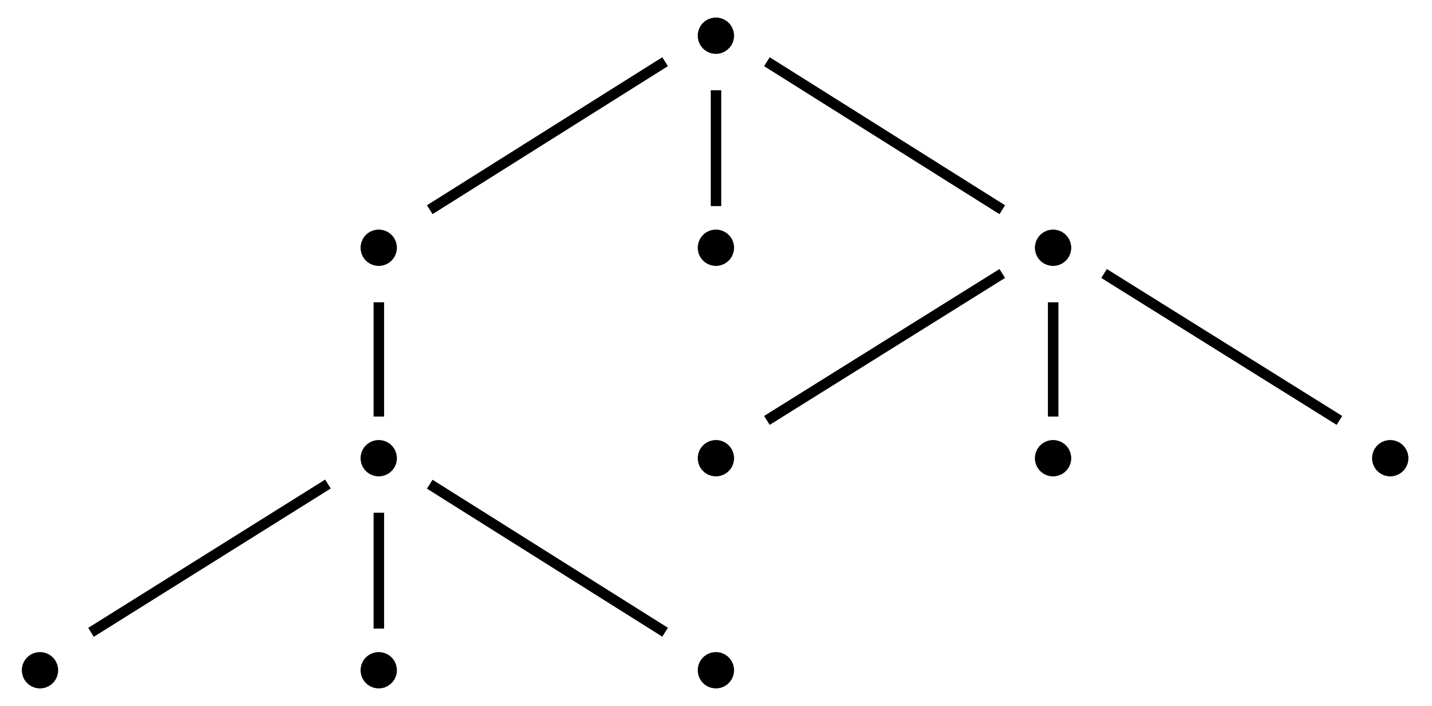
- [1,1] ✗
- [2,1] ✓
- [2,2,1] ✓
- [2,2,0] ✗

Drei Möglichkeiten zur Beschreibung von Bäumen

1. Mithilfe eines Ausdrucks des Typs tree.
   T[T[T[T[],T[],T[]] ,T[], T[T[],T[],T[]]]

2. Mithilfe der Menge seiner gültigen Adressen.
   [], [1], [1,1], [1,1,1], [1,1,2], [1,1,3], [2], [3], [3,1], [3,2],[3,3]

3. Mithilfe seiner grafischen Darstellung.



Präfix

- Eine Liste xs heißt **Präfix** einer Liste ys,
  wenn es eine Liste zs gibt mit $xs @ zs = ys$
- Eine Liste xs heißt **echtes Präfix** einer Liste ys,
  wenn es eine nichtleere Liste zs gibt mit $xs @ zs = ys$

- **Proposition:** Für jeden Baum t gilt:
  1. [] ist eine **gültige Adresse** für t.
  2. Wenn a eine gültige Adresse für t ist,
     dann ist **jedes Präfix** von a eine **gültige Adresse** für t.
  3. Wenn $a @ [n]$ eine gültige Adresse für t ist und $1 \leq k \leq n$,
     dann ist $a @ [k]$ eine gültige Adresse für t.

Nachfolger und Vorgänger

Sei ein Baum t gegeben und seien
a und a' gültige Adressen für t. Wir sagen, dass

- der durch a' bezeichnete Knoten der ***n*-te Nachfolger** des durch a bezeichneten Knoten ist, wenn $a' = a@[n]$ gilt.
- der durch a' bezeichnete Knoten ein **Nachfolger** des durch a bezeichneten Knoten ist, wenn eine Zahl n mit $a' = a@[n]$ existiert.
- der durch a bezeichnete Knoten der **Vorgänger** des durch a' bezeichneten Knoten ist, wenn eine Zahl n mit $a@[n] = a'$ existiert.

Proposition: Sei t ein Baum. Dann haben alle Knoten außer der Wurzel genau einen Vorgänger, die Wurzel hat keinen Vorgänger.

Unter- und übergeordnet

Sei ein Baum t gegeben und seien
a und a' gültige Adressen für t. Wir sagen, dass

- der durch a' bezeichnete Knoten dem durch a bezeichneten Knoten **untergeordnet** ist, wenn eine Liste ns mit $a' = a@ns$ existiert.
- der durch a bezeichnete Knoten dem durch a′ bezeichneten Knoten **übergeordnet** ist, wenn eine Liste ns mit $a@ns = a'$ existiert.

Größe und Tiefe

- Die **Größe eines Baums** ist die Anzahl seiner Knoten.

```
fun size (T ts) = foldl op+ 1 (map size ts)
```

val size : tree → int

```
size t3
```

11 : int

- Die **Tiefe eines Baums** ist die maximale Länge der für ihn gültigen Adressen.

```
fun depth (T ts) = 1 + foldl Int.max ~1 (map depth ts)
```

val depth : tree → int

```
depth t3
```

3 : int

Faltung

$$fold : (\alpha\ list \rightarrow \alpha) \rightarrow tree \rightarrow \alpha$$

Schrittprozedur   Baum

- Die **Schrittprozedur** bestimmt den **Wert eines Baums** aus den **Werten der Unterbäume**

```
fun fold f (T ts) = f (map (fold f) ts)
```

val fold : (α list → α) → tree → α

Beispiele

$$fold : (\alpha\ list \to \alpha) \to tree \to \alpha$$

- Die **Größe eines Baums**

```
fun size (T ts) = foldl op+ 1 (map size ts)
```

val size : tree → int

```
val size = fold (foldl op+ 1)
```

- Die **Tiefe eines Baums**

```
fun depth (T ts) = 1 + foldl Int.max ˜1 (map depth ts)
```

val depth : tree → int

```
val depth = fold (fn xs => 1 + foldl Int.max ~1 xs)
```

Faltung

```
fun fold f (T ts) = f (map (fold f) ts)
```

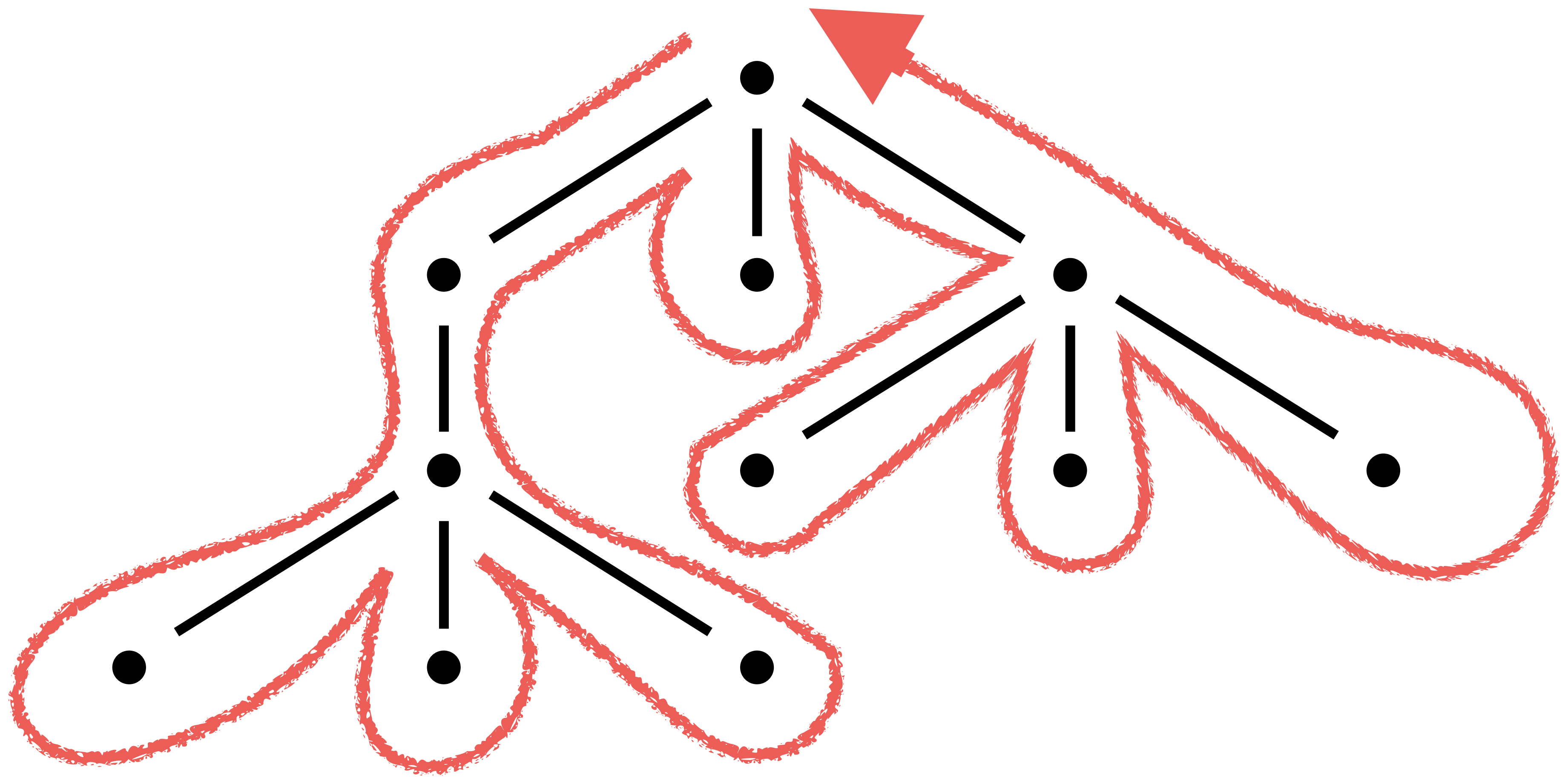
val fold : (α list → α) → tree → α

- **Beispiel:** Die **Größe eines Baums**

```
val size = fold (foldl op+ 1)

size (T[T[],T[]])
= fold (foldl op+ 1) (T[T[],T[]])
= (foldl op+ 1) (map (fold (foldl op+ 1)) [T[],T[]])
= (foldl op+ 1) (((fold (foldl op+ 1)) (T[]))
                      ::((map (fold (foldl op+ 1)) [T []])))
= (foldl op+ 1) (((foldl op+ 1) (map (fold (foldl op+ 1)) nil))
                      ::((map (fold (foldl op+ 1)) [T []])))
= (foldl op+ 1) (((foldl op+ 1) nil)
                      ::((map (fold (foldl op+ 1)) [T []])))
= (foldl op+ 1) (1::((map (fold (foldl op+ 1)) [T []])))
= (foldl op+ 1) [1,1]
= 3
```

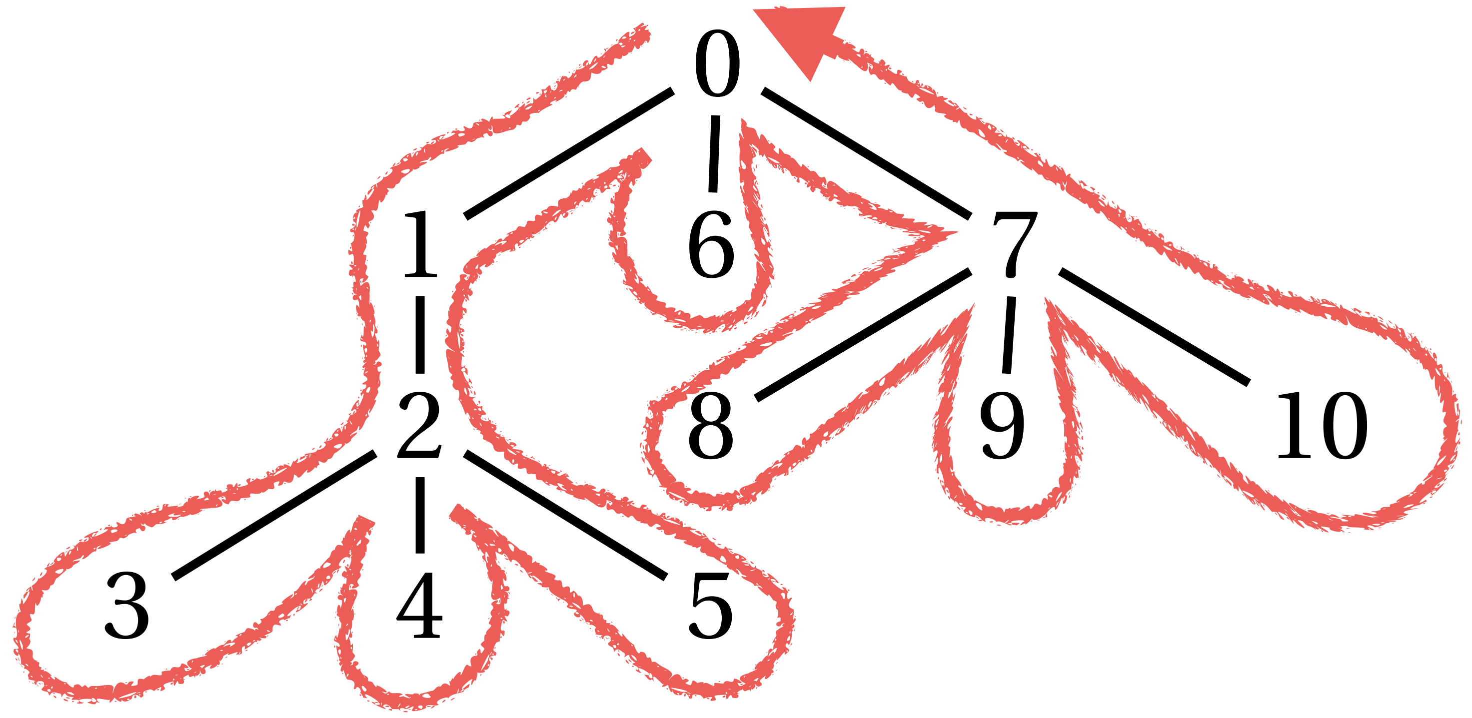
Standardtour



1. Die Tour **beginnt** und **endet** an der Wurzel.
2. Die Tour **folgt** den **Kanten** des Baums.
   Jede Kante des Baums wird genau **zweimal** durchlaufen,
   zuerst von **oben nach unten,** später von **unten nach oben**.
3. Unterbäume werden von **links nach rechts** besucht.

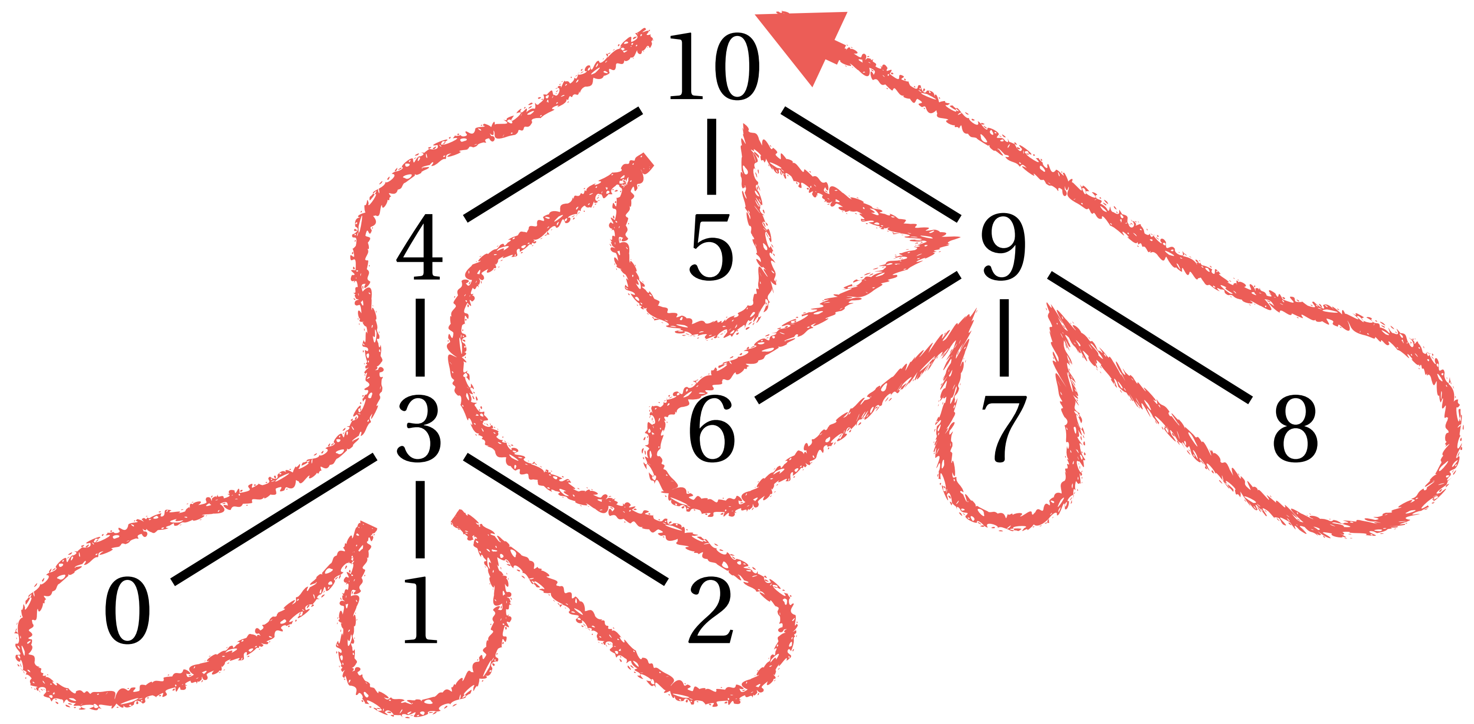
Pränummerierung vs. Postnummerierung

- **Pränummerierung** induziert **Präordnung**



Knoten wird beim **ersten Besuch** nummeriert.

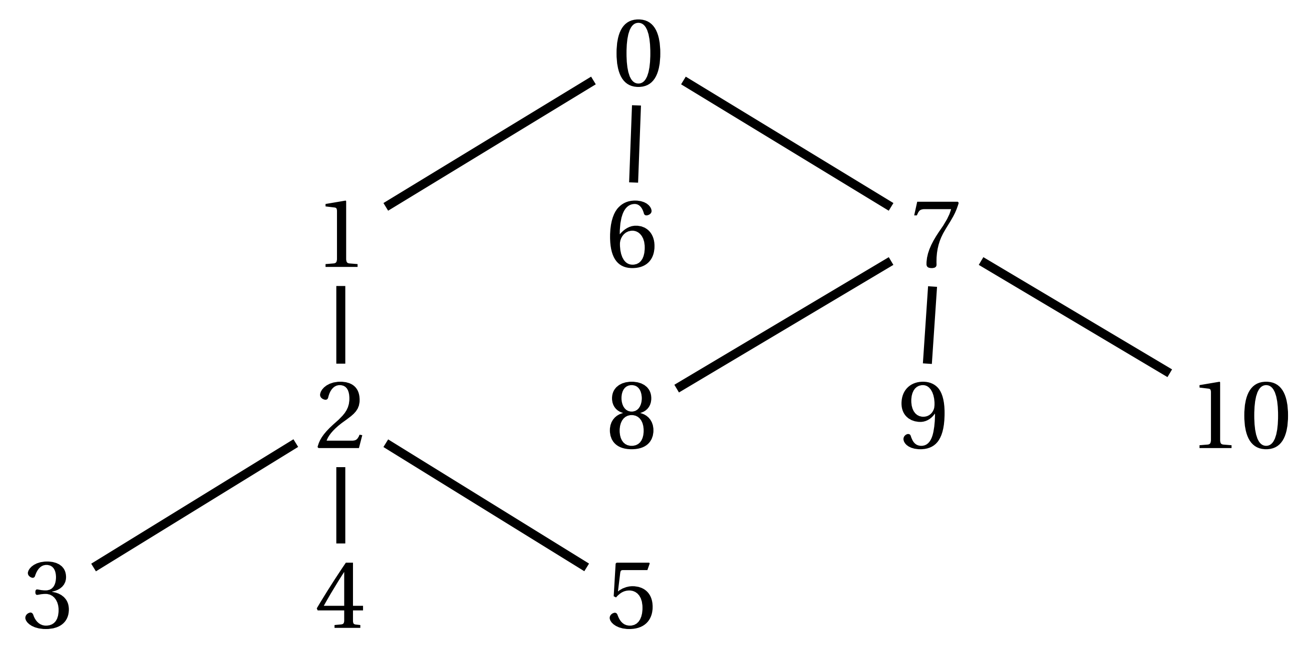
- **Postnummerierung** induziert **Postordnung**



Knoten wird beim **letzten Besuch** nummeriert.

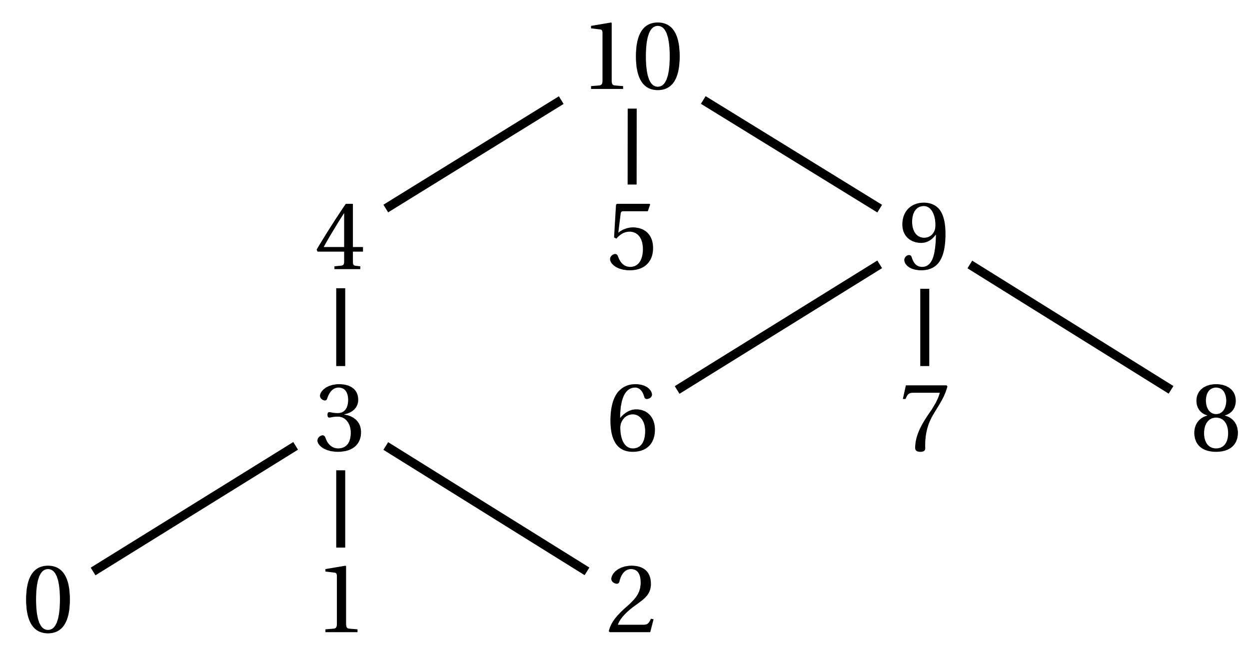
Pränummerierung vs. Postnummerierung

- **Pränummerierung**
  induziert **Präordnung**



Beginne mit der **Wurzel**
dann von **oben nach unten**
schrittweise von **links nach rechts**

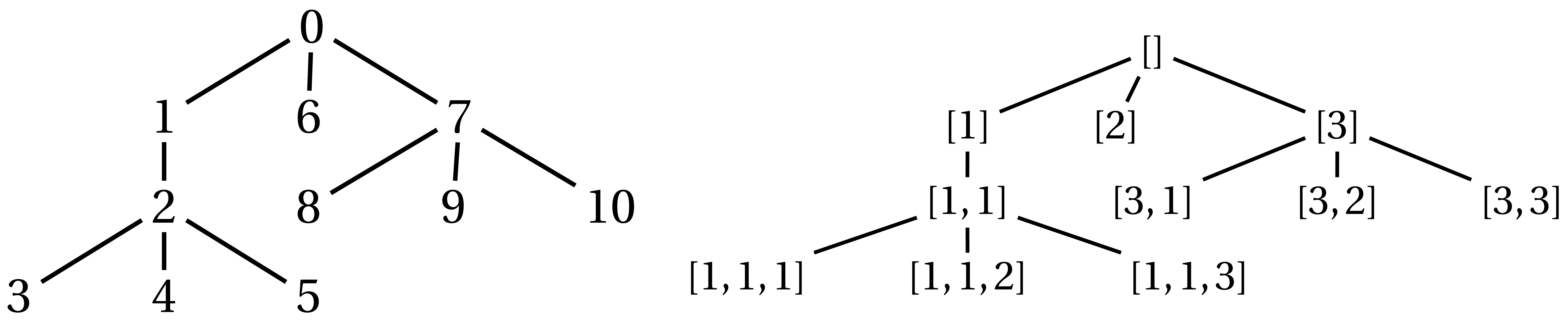
- **Postnummerierung**
  induziert **Postordnung**



Beginne mit dem **Blatt links unten**
dann von **unten nach oben**
schrittweise von **links nach rechts**

Pränummerierung

- **Pränummerierung**
  entspricht der **lexikalischen Ordnung** der Adressen



Beginne mit der **Wurzel**
dann von **oben nach unten**
schrittweise von **links nach rechts**

Frage

Welcher Teilbaum von T[T[T[]],T[]] hat Nummer 3 gemäß Pränummerierung?

- T[] ✓
- T[T[]] ✗
- T[T[T[]],T[]] ✗
- keiner ✗

Teilbaumzugriff mit Pränummern

- **Auflisten aller Teilbäume** in **Präordnung**

```
fun presubtrees (T ts) =
foldl (fn (t,tr) => tr @ (presubtrees t)) [T ts] ts
```

- **Zugriff auf Teilbaum** mit **Pränummer**

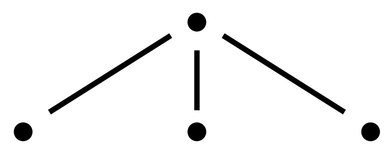
```
fun prest t k = List.nth(presubtrees t,k)
```

oder (besser:)

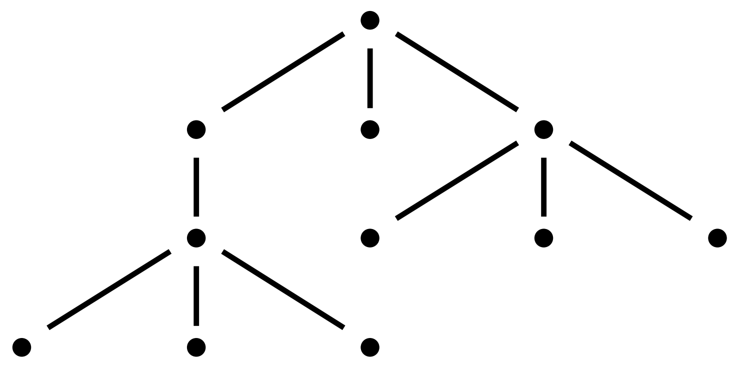
```
fun prest t =
let
  fun prel nil k = raise Subscript
    | prel (t::tr) 0 = t
    | prel ((T ts)::tr) k = prel (ts@tr) (k-1)
in
  prel [t]
end
```

Beispiel

$t1 = T[]$

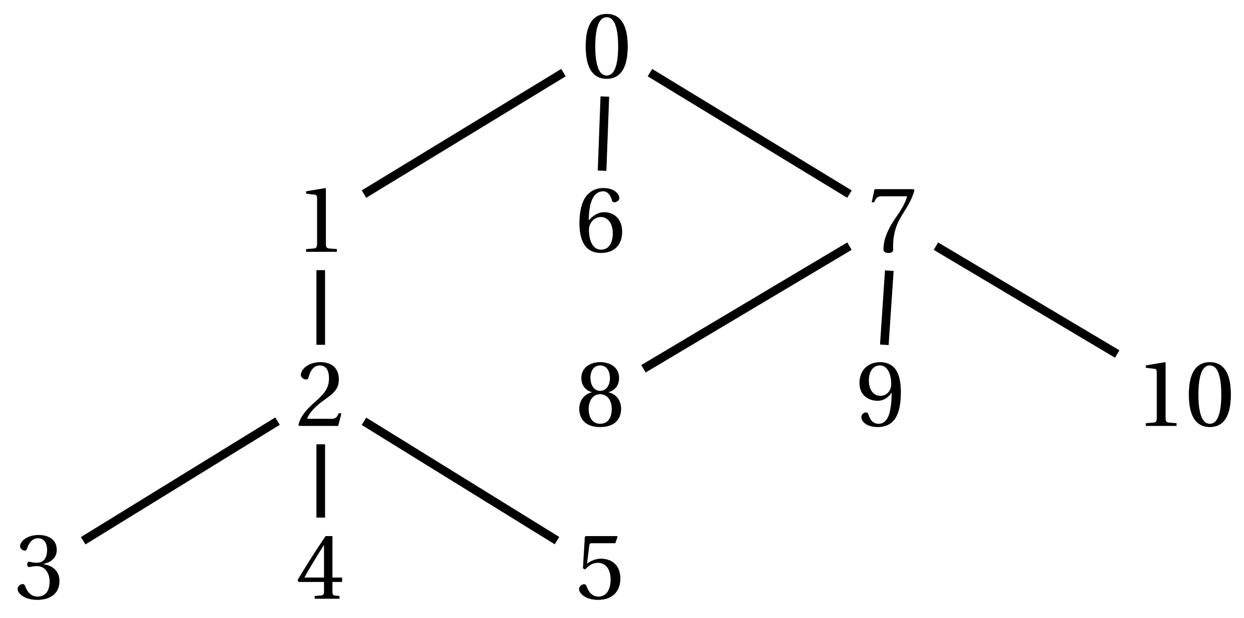


$t2 = T[t1, t1, t1$



$t3 = T[T[t2], t1, t2]$

```
fun prest t =
let
  fun prel nil k = raise Subscript
    | prel (t::tr) 0 = t
    | prel ((T ts)::tr) k = pre (ts@tr) (k-1)
in
  prel [t]
end
```




```
prest t3 3
= prel [t3] 3
= prel [(T t2), t1, t2] 2
= prel [t2, t1, t2] 1
= prel [t1, t1, t1, t1, t2] 0
= t1
```

www.prog1.saarland