Teil VIII

Algorithmik II

16. Rechnen mit Braun Bäumen — Fortsetzung

```
type Tree ⟨'elem⟩
    | Leaf
    | Node of 'elem ∗ Tree ⟨'elem⟩ ∗ Tree ⟨'elem⟩
let rec size = function
    | Leaf          → 0
    | Node (_, l, r) → 1 + size l + size r
```

Invariante: für jeden Knoten *Node* (a, l, r) gilt, dass der linke Teilbaum l genau so viele Elemente wie der rechte Teilbaum r enthält oder ein Element mehr.

$$0 \leqslant size\ l - size\ r \leqslant 1$$

☞ Die Struktur eines Braun Baums ist durch die Anzahl der Elemente eindeutig festgelegt.

16. Größe eines Braun Baums

Wie schnell können wir die Größe eines Braun Baums bestimmen? Die Funktion *size* benötigt lineare Laufzeit. *Surely, we can do better!*

Idee: Invariante ausnutzen.

```
let rec size = function
    | Leaf          → 0
    | Node (_, l, r) → let n = size r
                       in  1 + 2 ∗ n + (0 oder 1)
```

☞ Wie können wir rausbekommen, ob der linke Teilbaum größer ist?

16. Größe eines Braun Baums

Idee: Wir versuchen auf das $(n+1)$-te Element zuzugreifen (an Position n, da Nummerierung ab 0):

```
let rec size = function
    | Leaf          → 0
    | Node (_, l, r) → let n = size r
                       in  1 + 2 ∗ n + (match nth (l, n) with
                                        | None   → 0
                                        | Some _ → 1)
```

Gelingt der Zugriff, dann ist der linke Teilbaum größer; schlägt der Zugriff fehl, dann sind beide Teilbäume gleich groß.

Laufzeit: logarithmisch viele Zugriffe, jeder Zugriff hat logarithmische Laufzeit, also $(\lg n)^2$.

16. Konstruktion von Braun Bäumen

Wie schnell können wir einen Braun Baum konstruieren?

Gesucht: Umkehrfunktion von *to-list*.

$$\mathit{to\text{-}list} \ll \mathit{from\text{-}list} = \mathit{id}$$
$$\mathit{from\text{-}list} \ll \mathit{to\text{-}list} = \mathit{id}$$

☞ Da die Struktur eines Braun Baums durch die Anzahl der Elemente eindeutig festgelegt ist, hat *to-list* tatsächlich eine Inverse.

16. Konstruktion: Möglichkeit 1

Wir verwenden das *Struktur Entwurfsmuster* für Listen und fügen nacheinander die Listenelemente mit *cons* in einen anfangs leeren Braun Baum ein.

```
let rec from-list = function
  | []      → Leaf
  | x :: xs → cons x (from-list xs)
```

Laufzeit: cons hat eine logarithmische Laufzeit; somit ist die Gesamtlaufzeit linear-logarithmisch, $n \lg n$.

16. Konstruktion: Möglichkeit 2

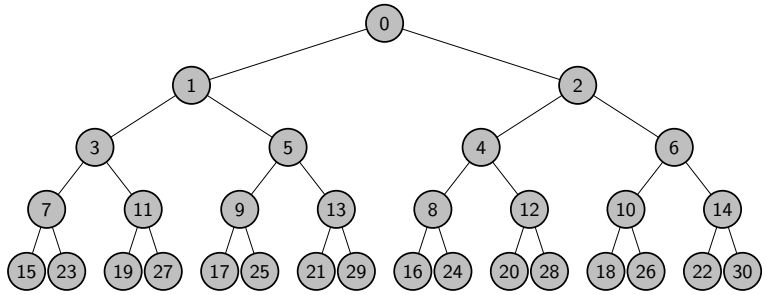
Wir verwenden das *allgemeine Leibniz Entwurfsmuster* für Listen und kehren *to-list* um: aus *zip* wird *unzip* ...

```
let rec to-list = function
  | Leaf           → []
  | Node (x, l, r) → x :: zip (to-list l, to-list r)
let rec from-list = function
  | []      → Leaf
  | x :: xs → let (xs1, xs2) = unzip xs
              Node (x, from-list xs1, from-list xs2)
```

Laufzeit: lineare Laufzeit pro Rekursionsebene; logarithmische Rekursionstiefe; somit ist die Gesamtlaufzeit linear-logarithmisch, $n \lg n$.

16. Konstruktion: Möglichkeit 3

Idee: wir konstruieren den Baum ebenenweise von unten nach oben („bottom-up“ statt „top-down“). Laufendes Beispiel: *from-list* $[0\,..\,30]$.



16. Konstruktion: Möglichkeit 3

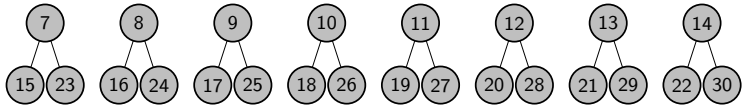
Erster Schritt: Wir teilen die Listenelemente ebenenweise auf.

0
1 2
3 4 5 6
7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

☞ Die Anzahl der Elemente verdoppelt sich von einer Ebene zur nächsten.

16. Konstruktion: Möglichkeit 3

Konstruktion der zweituntersten Ebene:

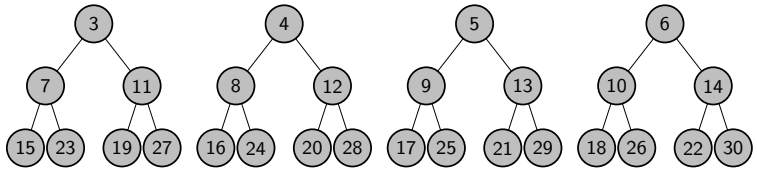


☞ Wurzeln der linken Teilbäume: 15 . . . 22; Wurzeln der rechten Teilbäume: 23 . . . 30.

15 16 17 18 19 20 21 22 ‖ 23 24 25 26 27 28 29 30

16. Konstruktion: Möglichkeit 3

Konstruktion der drittuntersten Ebene:

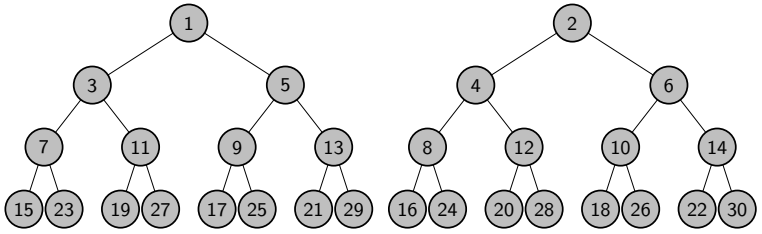


☞ Wurzeln der linken Teilbäume: 7 . . . 10; Wurzeln der rechten Teilbäume: 11 . . . 14.

7 8 9 10 ‖ 11 12 13 14

16. Konstruktion: Möglichkeit 3

Konstruktion der viertuntersten Ebene:



☞ Wurzeln der linken Teilbäume: 3 . . . 4; Wurzeln der rechten Teilbäume: 5 . . . 6.

3 4 ‖ 5 6

16. Konstruktion: Möglichkeit 3

Die Funktion *build k* überführt eine Liste von Elementen in eine Liste von k Braun Bäumen.

```
let single a = Node (a, Leaf, Leaf)

// val build : Nat → 'a list → (Tree ⟨'a⟩) list
let rec build k list =
  if k ⩾ length list
  then map single list @ [for i in 1 .. k − length list → Leaf]
  else let (xs, rest) = split-at k list
       let (ls, rs)   = split-at k (build (2 ∗ k) rest)
       map3 (fun x l r → Node (x, l, r)) xs ls rs

let from-list list = head (build 1 list)
```

☞ Sind nicht genügend Elemente für die unterste Ebene vorhanden, füllen wir mit $k - \mathit{length\ list}$ Blättern auf. (Was machen *map* und *map3*?)

Laufzeit: pro Ebene proportional zur Anzahl der Knoten; somit insgesamt linear!

16. Konstruktion: „top-down“ versus „bottom-up“

	top-down	bottom-up
Aufteilung	Liste von Elementen *unzip*	Liste von Teilbäumen *halve*