7 Rekursive Datenstrukturen II: Bäume

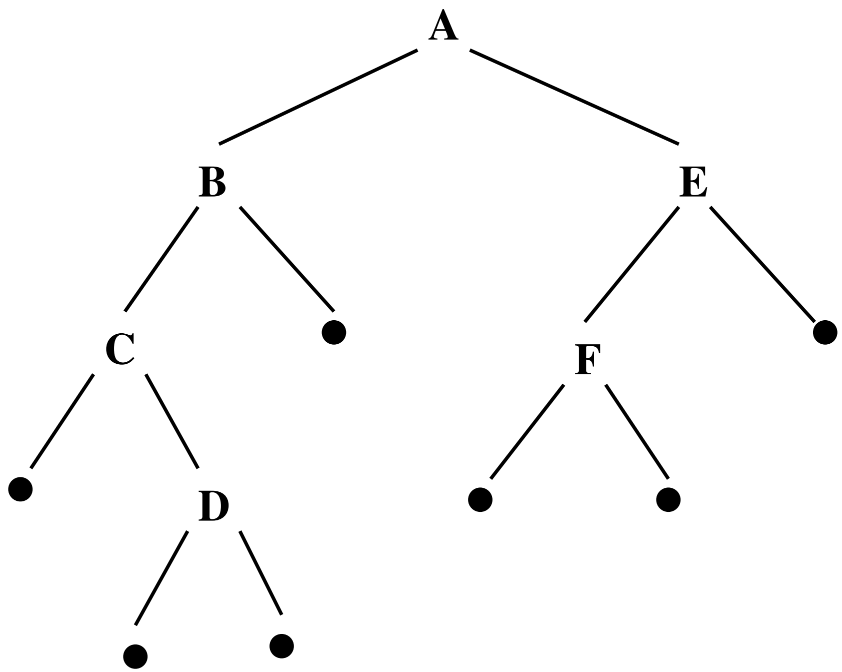
7.1 Binärbäume

Die Menge der Binärbäume über einer Menge X ist induktiv definiert durch

- Der *leere Binärbaum* ist ein Binärbaum.
- Falls $x \in X$ ist sowie l und r Binärbaume sind, dann ist ein *Knoten* bestehend aus x, l und r ebenfalls ein Binärbaum.

  Die Binärbäume l und r heißen *linker bzw. rechter Teilbaum* und das Element x ist die *Markierung des Knotens*.
- Nichts sonst ist ein Binärbaum.

Beispiel



- Der oberste Knoten ist die *Wurzel* des Baums.
- Knoten, deren Teilbäume beide leer sind, sind *Blätter*; alle anderen Knoten sind *innere Knoten*.

7.1.1 Definition: Leerer Binärbaum

```
; Ein leerer Binärbaum ist ein Wert
;  (make-empty-tree)
(define-record-procedures empty-tree
  make-empty-tree empty-tree?
  ())

; Der leere Binärbaum
; the-empty-tree : empty-tree
(define the-empty-tree (make-empty-tree))
```

7.1.2 Definition: Knoten und Binärbaum

```
; Ein Knoten ist ein Wert
;  (make-node l b r)
; wobei b : value eine Markierung ist und l und r Binärbäume.
(define-record-procedures node
  make-node node?
  (node-left node-label node-right))
```

Ein Binärbaum ist ein gemischter, rekursiver Datentyp:

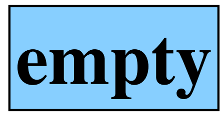
```
; Ein Binärbaum ist eins der folgenden
; - ein leerer Baum
; - ein Knoten
(define-contract btree (mixed empty-tree node))
```

Beispiele

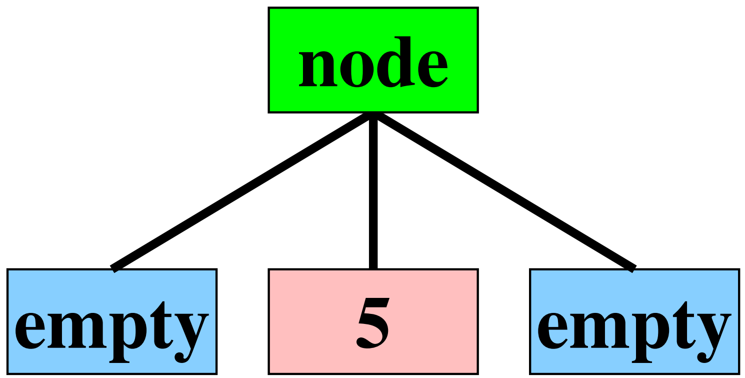
```
(define make-leaf
  (lambda (x) (make-node the-empty-tree x the-empty-tree)))
(define t0 the-empty-tree)
(define t1 (make-leaf 5))
(define t2 (make-node t1 6 (make-leaf 10)))
(define t3 (make-node t2 12 (make-leaf 20)))
```

Grafische Darstellung

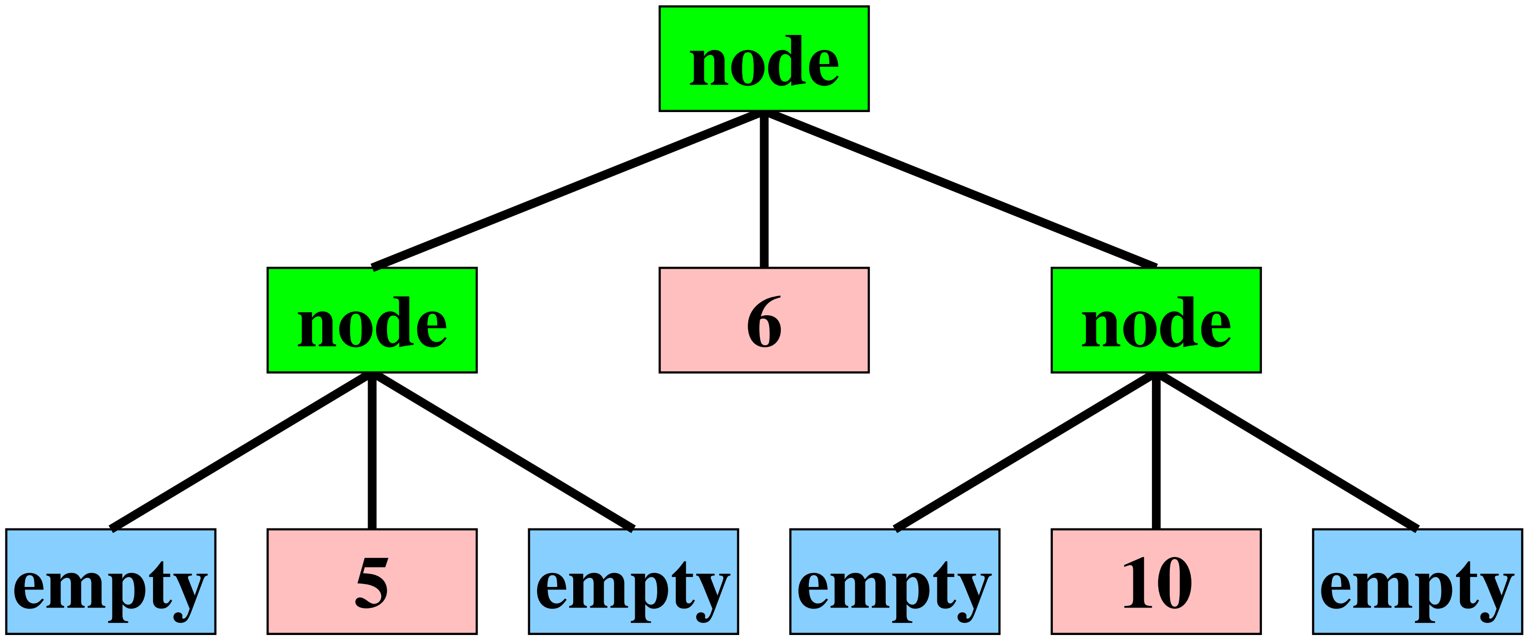
`t0`



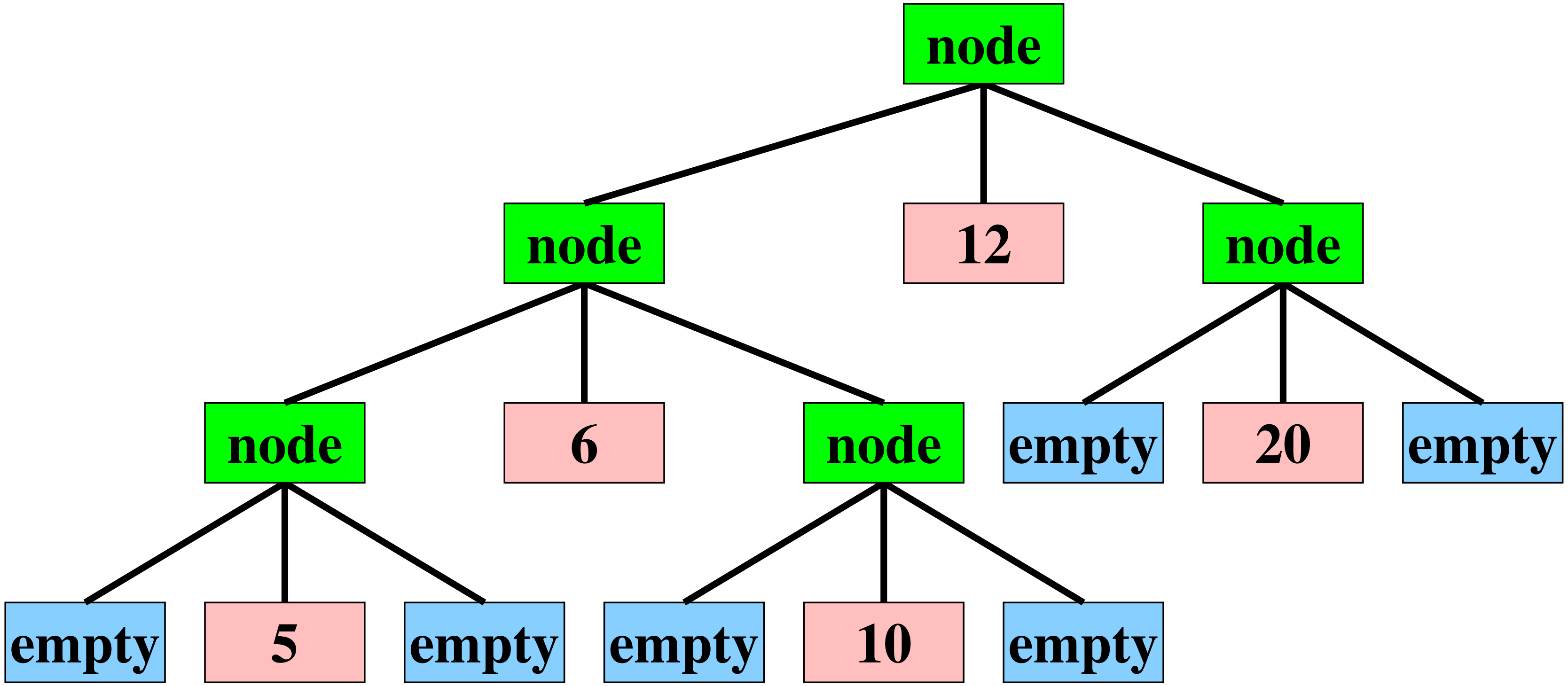
`t1`



t2



t3


node
node
12
node
node
6
node
empty
20
empty
empty
5
empty
empty
10
empty

7.1.3 Tiefe eines Binärbaums

Die Tiefe eines Binärbaums `t` ist die maximale Anzahl von Knoten von der Wurzel bis zu einem `empty-tree` im Baum `t`.

```
; Berechnet die Tiefe eines Binärbaums
(: btree-depth (btree -> natural))

; Tests
(check-expect (btree-depth the-empty-tree)
              0)
(check-expect (btree-depth (make-node (make-leaf 1) 0 the-empty-tree))
              2)
```

Schablone für Prozeduren, die Binärbäume verarbeiten:

```
(define btree-depth
  (lambda (t)
    (cond
      ((empty-tree? t) ...)
      ((node? t)
       (... (btree-depth (node-left t))
        ... (node-label t) ...
        ... (btree-depth (node-right t))
        ...)))))
```

Definition:

```
(define btree-depth
  (lambda (t)
    (cond
      ((empty-tree? t)
       0)
      ((node? t)
       (+ 1
          (max (btree-depth (node-left t))
               (btree-depth (node-right t))))))))
```

7.1.4 Anzahl der Knoten eines Binärbaums

```
; bestimmt die Anzahl der Knoten im Binärbaum
(: btree-size (btree -> natural))

; Tests
(check-expect (btree-size the-empty-tree)
              0)

(check-expect (btree-size (make-node (make-leaf 1) 0 the-empty-tree))
              2)
```

Schablone:

```
(define btree-size
  (lambda (t)
    (cond
      ((empty-tree? t)
       ...)
      ((node? t)
       (... (btree-size (node-left t))
        ... (node-label t) ...
        ... (btree-size (node-right t))
        ...)))))
```

Definition:

```
(define btree-size
  (lambda (t)
    (cond
      ((empty-tree? t)
       0)
      ((node? t)
       (+ 1
          (btree-size (node-left t))
          (btree-size (node-right t)))))))
```

7.2 Suchbäume

7.2.1 Das Suchproblem

Grundmenge M mit einer totalen Ordnung $\leq \subseteq M \times M$

Gegeben: Suchmenge $S \subseteq M$, $x \in M$

Gewünschte Operationen

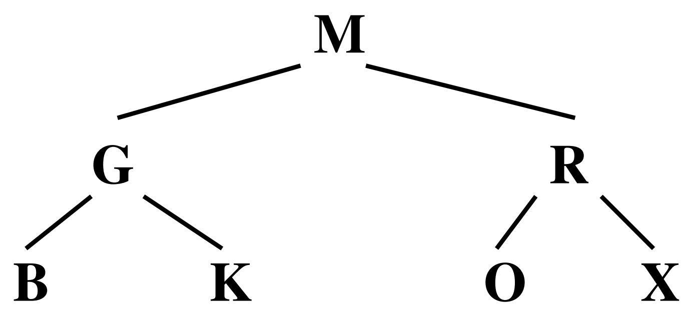
- Suche eines Elements: $x \in S$?
- Vergrößern der Suchmenge $S \cup \{y\}$
- Verkleinern der Suchmenge $S \setminus \{y\}$

7.2.2 Binärer Suchbaum

- Sei M Grundmenge mit totaler Ordnung $\leq$.
- Ein binärer Suchbaum ist ein Binärbaum `btree`, bei dem für jeden Knoten (`make-node l x r`) das Element `x`$\in M$ ist und die *Suchbaumeigenschaft* gilt. Das heißt,
  - alle Elemente im linken Teilbaum `l` sind **kleiner als** `x` sind und
  - alle Elemente im rechten Teilbaum `r` sind **größer als** `x`.
- **Zunächst für** $M =$ `real`

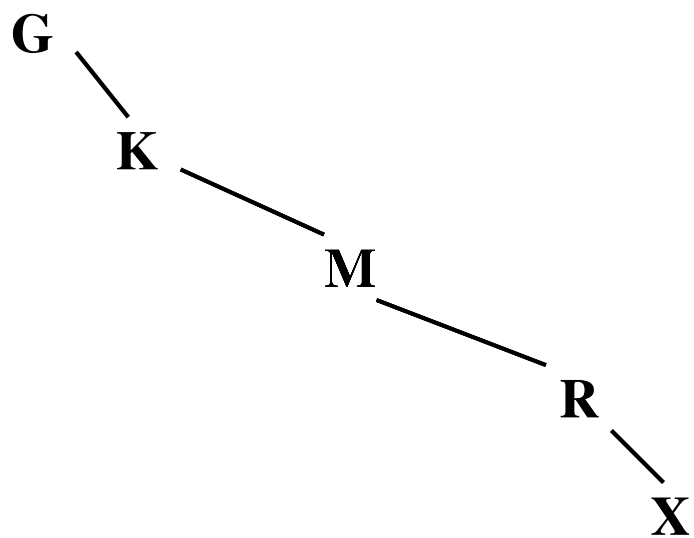
7.2.3 Warum binäre Suchbäume?

Guter Suchbaum: logarithmische Suchzeit (in der Anzahl der Elemente)



Dieser Suchbaum ist *balanciert*, d.h., für jeden Knoten unterscheiden sich die Tiefen des linken und rechten Teilbaums um höchstens eins.

Vorsicht! Suchbaum kann zur Liste *entarteten*: lineare Suchzeit



7.2.4 Suchen eines Elements

```
; Suchen eines Elements in einem binären Suchbaum
(: btree-member? (btree real -> boolean)))

; Tests
(check-expect (btree-member? empty-tree 42)
              #f)

(check-expect (btree-member? (make-leaf 99) 42)
              #f)

(check-expect (btree-member? (make-leaf 99) 99)
              #t)
```

Schablone

```
(define btree-member?
  (lambda (t y)
    (cond
     ((empty-tree? t)
      ...)
     ((node? t)
      ... (node-elem t)
      ... (btree-member? (node-left t) y)
      ... (btree-member? (node-right t) y)))))
```

Definition

```
(define btree-member?
  (lambda (t y)
    (cond
     ((empty-tree? t)
      #f)
     ((node? t)
      (let ((x (node-elem t)))
        (cond
         ((= y x)
          #t)
         ((< y x)
          (btree-member? (node-left t) y))
         ((< x y)
          (btree-member? (node-right t) y)))))))))
```

7.2.5 Einfügen eines Elements

```
; Einfügen eines Elements in einen binären Suchbaum
(: btree-insert (btree real -> btree)))

; Tests
(check-expect (btree-insert empty-tree 42)
              (make-leaf 42))

(check-expect (btree-insert (make-leaf 99) 42)
              (make-node (make-leaf 42) 99 the-empty-tree))

(check-expect (btree-insert (make-leaf 99) 99)
              (make-leaf 99))
```

Schablone

```
(define btree-insert
  (lambda (t y)
    (cond
     ((empty-tree? t)
      ...)
     ((node? t)
      ... (node-elem t)
      ... (btree-insert (node-left t) y)
      ... (btree-insert (node-right t) y)))))
```

Schablone für binären Suchbaum

```
(define btree-insert
  (lambda (t y)
    (cond
     ((empty-tree? t)
      ...)
     ((node? t)
      (let ((x (node-elem t)))
        (cond
         ((= y x)
          ...)
         ((< y x)
          ... (btree-insert (node-left t) y))
         ((< x y)
          ... (btree-insert (node-right t) y))))))))
```

Definition

```
(define btree-insert
  (lambda (t y)
    (cond
     ((empty-tree? t)
      (make-leaf y))
     ((node? t)
      (let ((x (node-elem t)))
        (cond
         ((= y x)
          t)
         ((< y x)
          (make-node (btree-insert (node-left t) y) x (node-right t)))
         ((< x y)
          (make-node (node-left t) x (btree-insert (node-right t) y)))))))))
```

7.2.6 Löschen eines Elements

```
; Löschen eines Elements in einem binären Suchbaum
(: btree-delete (btree real -> btree))

; Tests
(check-expect (btree-delete (make-leaf 55) 55)
              (make-empty-tree))
(check-expect (btree-delete (make-node (make-leaf 55) 66 (make-leaf 77)) 55)
              (make-node (make-empty-tree) 66 (make-leaf 77)))
(check-expect (btree-delete (make-node (make-leaf 55) 66 (make-leaf 77)) 66)
              (make-node (make-leaf 55) 77 (make-empty-tree)))
```

Definition nach Schablone und Wunschdenken

```
(define btree-delete
  (lambda (t y)
    (cond
      ((empty-tree? t)
       t)
      ((node? t)
       (let ((x (node-elem t)))
         (cond
           ((= y x)
            (btree-combine (node-left t) (node-right t)))
           ((< y x)
            (make-node (btree-delete (node-left t) y) x (node-right t)))
           ((> y x)
            (make-node (node-left t) x (btree-delete (node-right t) y))))))))
```

```
; Hilfsprozedur:  Zusammensetzen zweier binärer Suchbäume
; wobei alle Schlüssel des linken Baums kleiner sind als die des rechten
(: btree-combine (btree btree -> btree)))
```

```
; Hilfsprozedur: Entferne das minimale Element aus nicht-leerem Suchbaum
; ergibt Liste mit minimalem Element und verkleinertem Suchbaum
(: btree-remove-minimum (btree -> (list %value)))

; Tests
(check-expect (btree-remove-minimum (make-leaf 42))
              (list 42 (make-empty-tree)))
(check-expect (btree-remove-minimum
               (make-node (make-empty-tree) 42 (make-leaf 99)))
              (list 42 (make-leaf 99)))
(check-expect (btree-remove-minimum
               (make-node (make-node (make-empty-tree) 42 (make-leaf 99))
                          100
                          (make-leaf 150)))
              (list 42 (make-node (make-leaf 99) 100 (make-leaf 150))))
```

Zusammenfassung

- Ein Binärbaum ist entweder leer oder er besteht aus einem linken Teilbaum, einem Datenwert und einem rechten Teilbaum.
- Die Größe eines Binärbaums ist die Anzahl der Datenwerte im Baum.
- Die Tiefe eines Binärbaums ist die maximale Schachtelungstiefe der Baumknoten.
- In einem binären Suchbaum gilt für jeden Knoten die Suchbaumeigenschaft: Der Datenwert am Knoten ist größer als alle Elemente im linken Teilbaum und kleiner als alle Elemente im rechten Teilbaum.
- In einem balancierten binären Suchbaum können die Operationen Auffinden, Einfügen und Löschen mit logarithmischem Aufwand in der Anzahl der Elemente im Suchbaum durchgeführt werden.