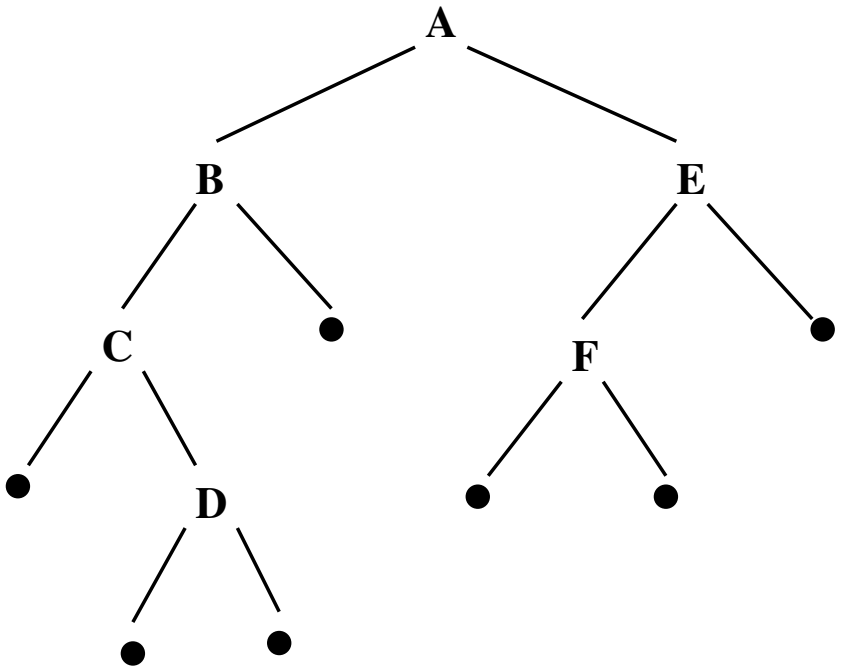
# 10 Rekursive Datenstrukturen II: Bäume

## 10.1 Binärbäume

Die Menge der Binärbäume über einer Menge $X$ von Elementen ist induktiv definiert durch

- Der *leere Binärbaum* ist ein Binärbaum.
- Ein *Knoten* ist ein Binärbaum, falls er zwei Binärbäume enthält und ein Element $x \in X$. Die Binärbäume 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*.

### 10.1.1 Definition: Leerer Baum

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

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

### 10.1.2 Definition: Knoten und Binärbaum

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

Ein Binärbaum ist dann ein gemischter, rekursiver Datentyp:

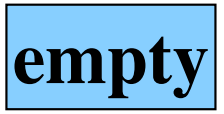
```
; Ein Binärbaum ist eins der folgenden
; - ein leerer Baum
; - ein Knoten
; Name: btree
```

## Beispiele

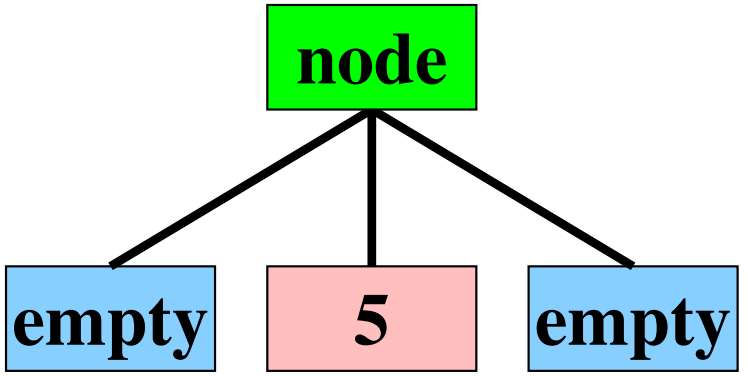
```
(define make-leaf
  (lambda (x) (make-node the-empty-tree x the-empty-tree)))
(define t0 the-empty-tree)
(define t1 (make-leaf 5))
(define t2 (make-node t1 6 (make-leaf 10)))
(define t3 (make-node t2 12 (make-leaf 20)))
```

## Grafische Darstellung

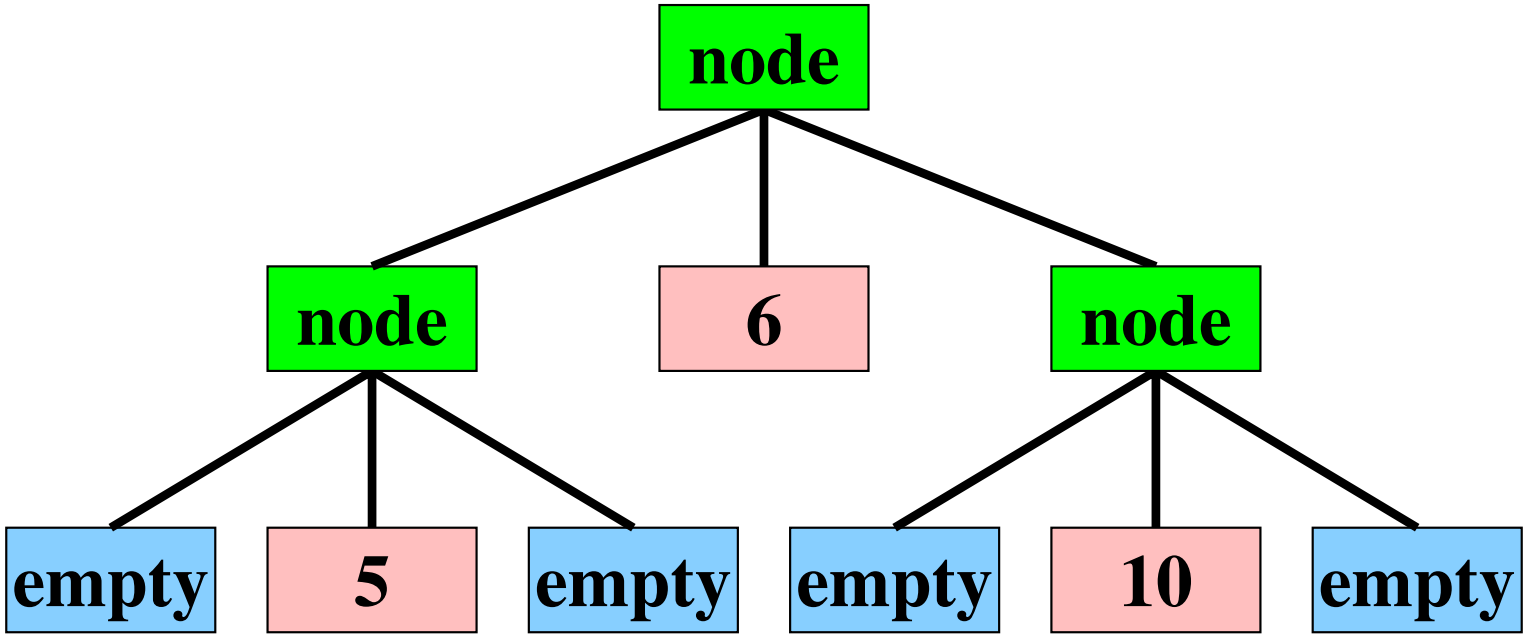
`t0`



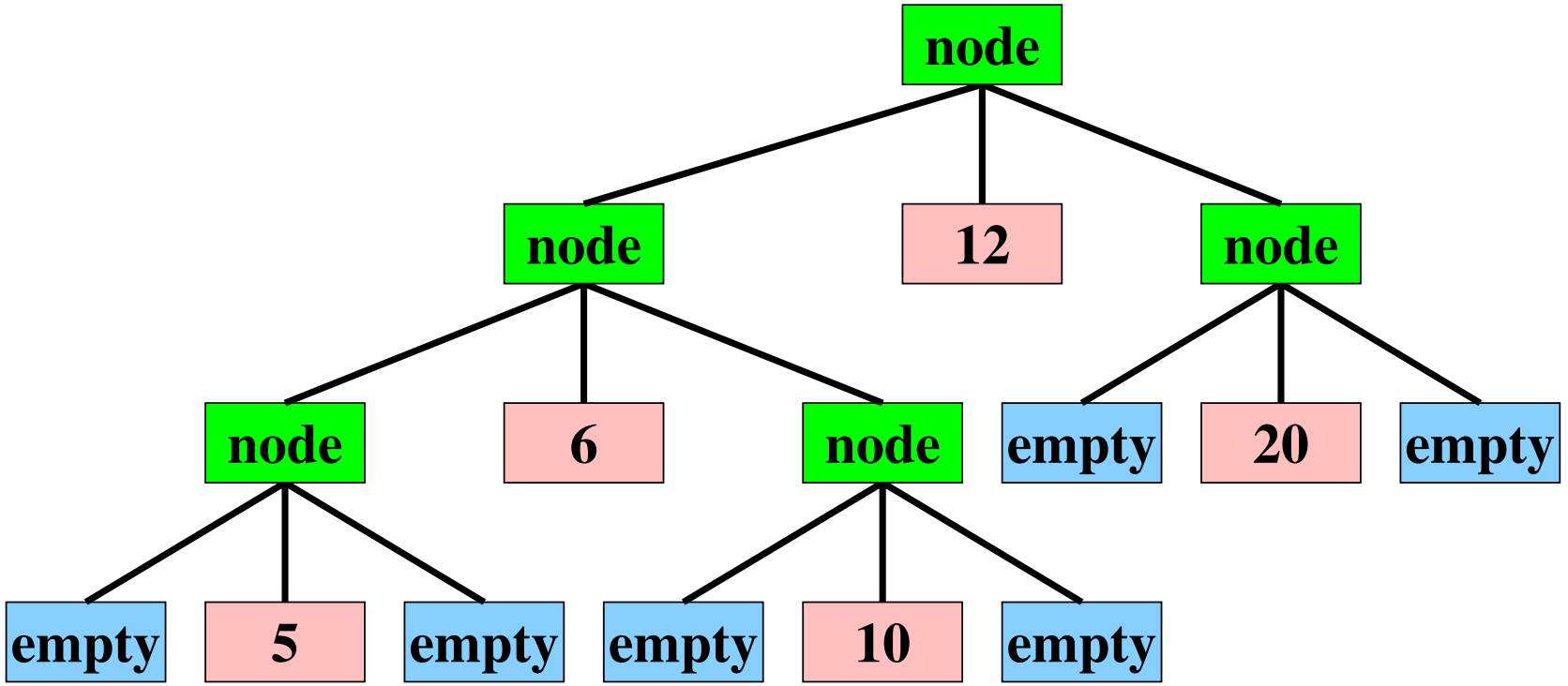

`t1`



t2



t3


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

## 10.2 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`.

**Signatur:** `btree-depth : btree -> nat`

**Erklärung:** Berechnet die Tiefe eines Binärbaums

**Beispiel:**

```
(btree-depth the-empty-tree)
; Ergebnis: 0
(btree-depth (make-node (make-leaf 1) 0 the-empty-tree))
; Ergebnis: 2
```

**Schablone:**

```
(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))))))))
```

## 10.3 Binärbaume als parametrischer Datentyp

- Die Konstruktoren und Selektoren von btree haben polymorphe Verträge.

  ```
  ; the-empty-tree : btree(A)
  ; make-node : btree(A) A btree(A) -> btree(A)
  ```

- Auch die Tiefe eines Baums hängt nicht von den Elementen ab, die im Baum gespeichert sind.

  ```
  ; btree-depth : btree(A) -> nat
  ```

## 10.4 Anzahl der Knoten eines Binärbaums

**Signatur:** `btree-size : btree(A) -> number`

**Erklärung:** `(btree-size t)` bestimmt die Anzahl der Knoten `(make-node ...)` im Baum `t`.

**Beispiel:**

```
(btree-size the-empty-tree)
; Ergebnis: 0
(btree-size (make-node (make-leaf 1) 0 the-empty-tree))
; Ergebnis: 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)))))))
```

## 10.5 Suchbäume

- Ein *Suchbaum* ist eine Datenstruktur, die das *Suchproblem* löst.
- **Das Suchproblem**

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

  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 $S \setminus \{y\}$

### 10.5.1 Binärer Suchbaum

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

### 10.5.2 Suchbaum als Datenstruktur

```
; Ein Suchbaum ist ein Wert
;  (make-search-tree eq le t)
; wobei
;  eq : A A -> boolean
;   eine Prozedur ist, die zwei Elemente auf Gleichheit testet
;  le : A A -> boolean
;   eine Prozedur ist, die zwei Elemente auf kleiner oder gleich testet
;  t : btree(A)
;   ein Binärbaum ist, der die Suchbaumeigenschaft besitzt.
(define-record-procedures search-tree
  make-search-tree search-tree?
  (search-tree-label-equal-proc
   search-tree-label-leq-proc
   search-tree-tree))
```

### 10.5.3 Konstruktor des leeren Suchbaums

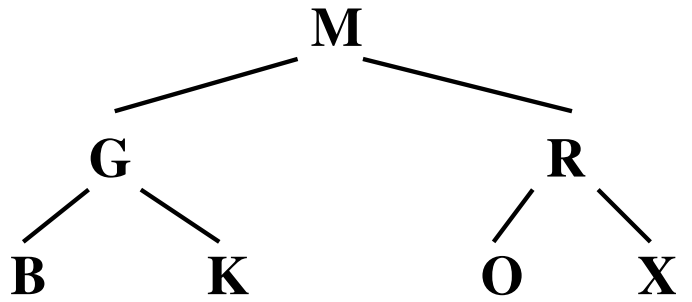
```
; leeren Suchbaum konstruieren
; make-empty-search-tree : (A A -> boolean) (A A -> boolean)
;                          -> search-tree(A)
(define make-empty-search-tree
  (lambda (eq le)
    (make-search-tree eq le the-empty-tree)))
```

```
; feststellen, ob Element im Suchbaum vorhanden ist
; search-tree-member? : A search-tree(A) -> boolean
(define search-tree-member?
  (lambda (elem st)
    (let ((eq? (search-tree-label-equal-proc st))
          (le? (search-tree-label-leq-proc st)))
      (letrec ((member?
                (lambda (t)
                  (cond
                   ((empty-tree? t)
                    #f)
                   ((node? t)
                    (cond
                     ((eq? elem (node-label t))
                      #t)
                     ((le? elem (node-label t))
                      (member? (node-left t)))
                     (else
                      (member? (node-right t))))))))))
        (member? (search-tree-tree st))))))
```

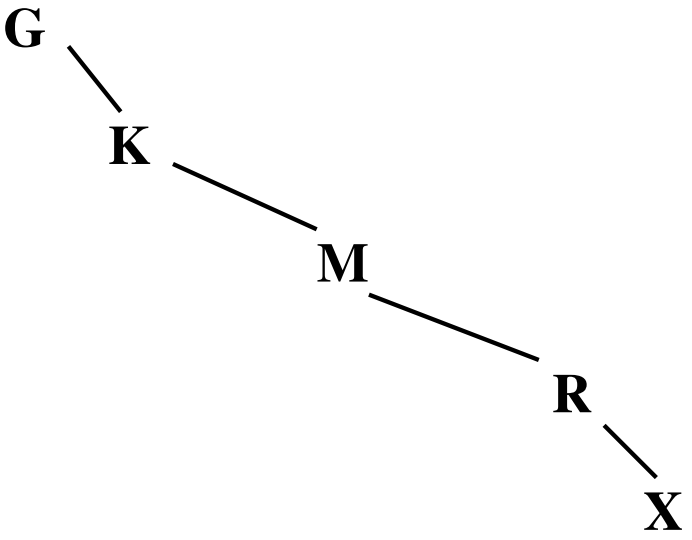
```
; Einfügen eines neuen Elements in einem Binärbaum
; search-tree-insert : A search-tree(A) -> search-tree(A)
(define search-tree-insert
  (lambda (elem st)
    (let ((eq? (search-tree-label-equal-proc st))
          (le? (search-tree-label-leq-proc st)))
      (letrec ((insert
                (lambda (t)
                  (cond
                   ((empty-tree? t)
                    (make-leaf elem))
                   ((node? t)
                    (let ((lab (node-label t)))
                     (cond
                      ((eq? elem lab)
                       t)
                      ((le? elem lab)
                       (make-node (insert (node-left t)) lab (node-right t))
                      (else
                       (make-node (node-right t) lab (insert (node-right t))))))))))))
        (make-search-tree eq? le? (insert (search-tree-tree st)))))))
```

### 10.5.4 Gute und schlechte Suchbäume

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




- Schlechter *entarteter* Suchbaum: lineare Suchzeit

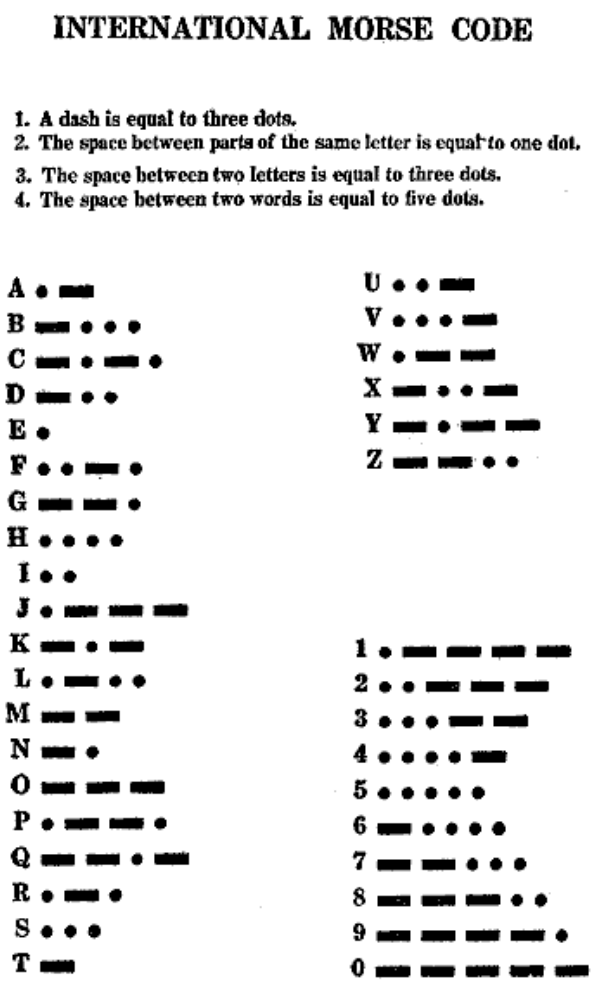


## 10.6 Huffman-Bäume

- Anwendung von Binärbäumen
- Aufgabe: Platz-effiziente Kompression von Textdaten
- Standardcodierungen von Textdaten
  - ISO-8859-1: 8 Bit pro Zeichen
  - UTF-16: 16 Bit pro Zeichen

  Codierungen mit *fester Länge*: Zeichen $\mapsto$ Bitfolge fester Länge
- Beobachtung: In Texten kommen Zeichen mit unterschiedlichen Häufigkeiten vor, trotzdem verbrauchen alle gleich viel Platz.
- Ziel: Codierung mit *variabler Länge*, so dass häufige Zeichen wenig Platz benötigen.

## 10.6.1 Beispiel: Morse-Code



INTERNATIONAL MORSE CODE

1. A dash is equal to three dots.
2. The space between parts of the same letter is equal to one dot.
3. The space between two letters is equal to three dots.
4. The space between two words is equal to five dots.

A • —
B — • • •
C — • — •
D — • •
E •
F • • — •
G — — •
H • • • •
I • •
J • — — —
K — • —
L • — • •
M — —
N — •
O — — —
P • — — •
Q — — • —
R • — •
S • • •
T —

U • • —
V • • • —
W • — —
X — • • —
Y — • — —
Z — — • •

1 • — — — —
2 • • — — —
3 • • • — —
4 • • • • —
5 • • • • •
6 — • • • •
7 — — • • •
8 — — — • •
9 — — — — •
0 — — — — —

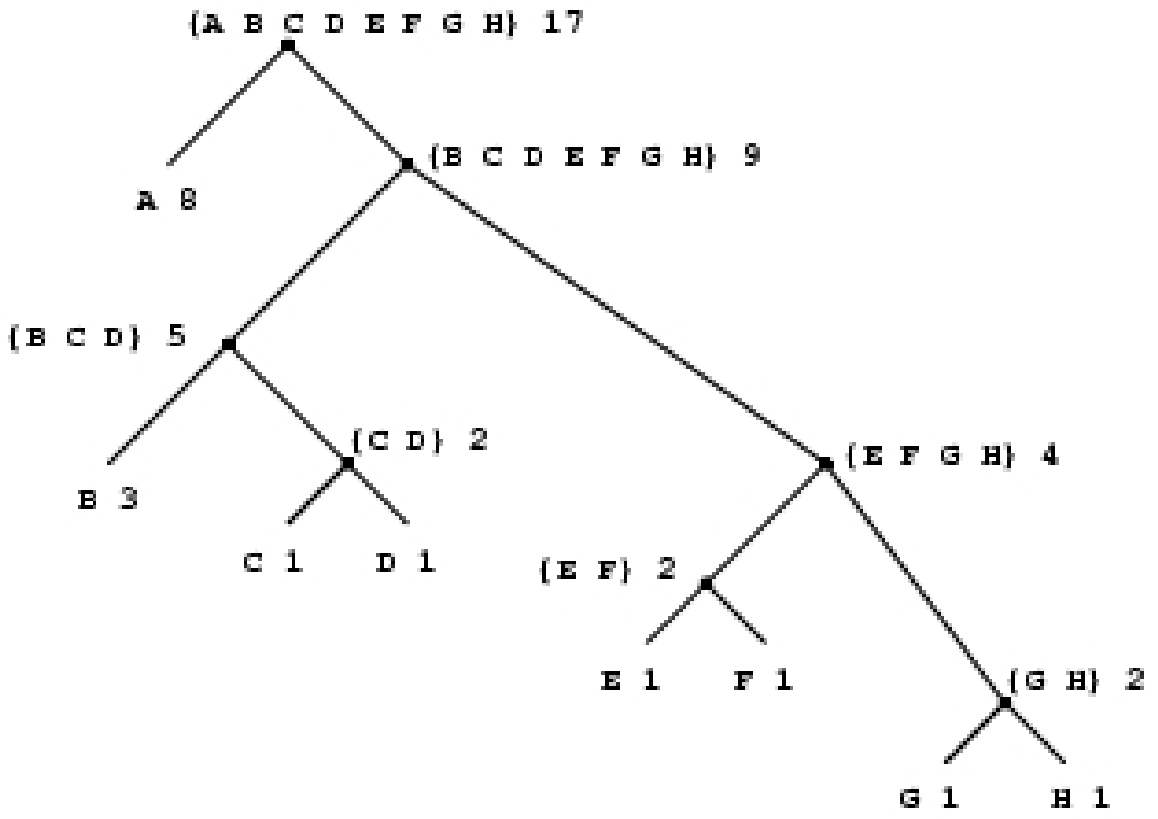
### 10.6.2 Präfix-Codierung

- Der Code eines Zeichens ist niemals Präfix des Codes eines anderen Zeichens.
- Bsp: Wenn 'E' mit '0' codiert wird, darf keine andere Codierung mit '0' beginnen.
- Problemstellung:
  Gegeben ein Zeichenvorrat mit Häufigkeiten für jedes Zeichen.
  Konstruiere eine Präfix-Codierung, die im Mittel die kürzeste Codierung liefert.

### 10.6.3 Huffman-Codierung

- Repräsentiert durch einen Binärbaum (den *Huffman-Baum*)
- Jedes Blatt repräsentiert ein Zeichen (mit Häufigkeit)
- Jeder innere Knoten repräsentiert eine Menge von Zeichen (mit Gesamthäufigkeit)
- Der Wurzelknoten repräsentiert den gesamten Zeichenvorrat
- Codierung eines Zeichens durch den Weg von der Wurzel zum Zeichen:
  - Suche das Blatt mit dem Zeichen beginnend von der Wurzel
  - Schreibe dabei '0'/'1' für jede Auswahl eines linken/rechten Teilbaums
- Decodierung eines Codes:
  - Beginne bei der Wurzel
  - Gehe zum linken/rechten Teilbaum über, je nachdem ob nächstes Bit '0'/'1' ist
  - Bei Ankunft am Blatt: Zeichen identifiziert

### 10.6.4 Beispiel: Huffman-Baum




Aus: Abelson and Sussman, Structure and Interpretation of Computer Programs, MIT Press

### 10.6.5 Implementierung von Huffman-Bäumen — Blätter

```
; Ein Huffman-Blatt ist ein Wert
;  (make-huffman-leaf s w)
; wobei s : string
; und   w : number (das Gewicht bzw die Häufigkeit von s)
(define-record-procedures huffman-leaf
  make-huffman-leaf huffman-leaf?
  (huffman-leaf-name huffman-leaf-weight))
```

### 10.6.6 Implementierung von Huffman-Bäumen — Knoten

```
; Ein Huffman-Knoten ist ein Wert
;  (make-huffman-node sl w l r)
; wobei sl : list(string)
;        w : number (kumulative Häufigkeit von sl)
;        l, r : huffman-tree
(define-record-procedures huffman-node
  really-make-huffman-node huffman-node
  (huffman-node-names huffman-node-weight
   huffman-node-left huffman-node-right))

; Ein Huffman-Baum ist eins der folgenden
; - ein Huffman-Blatt
; - ein Huffman-Knoten
; Name: huffman-tree
```

### 10.6.7 Erzeugung eines Huffman-Knotens

Die Information an einem inneren Knoten eines Huffman-Baums ergibt sich direkt aus seinen Teilbäumen, daher

```
; Huffman-Knoten aus zwei Huffman-Bäumen konstruieren
; make-huffman-node : huffman-tree huffman-tree -> huffman-node
(define make-huffman-node
  (lambda (l r)
    (really-make-huffman-node
     (append (huffman-tree-names l)
             (huffman-tree-names r))
     (+ (huffman-tree-weight l)
        (huffman-tree-weight r))
     l r)))
```

Nach Topdown-Design verbleibt zu definieren

```
; huffman-tree-names : huffman-tree -> list(string)
; huffman-tree-weight : huffman-tree -> number
```

### 10.6.8 Liste der Namen in einem Huffman-Baum

```
; huffman-tree-names : huffman-tree -> list(string)
(define huffman-tree-names
  (lambda (ht)
    (cond
     ((huffman-leaf? ht)
      (list (huffman-leaf-name ht)))
     ((huffman-node? ht)
      (huffman-node-names ht)))))
```

### 10.6.9 Gewicht eines Huffman-Baums

```
; huffman-tree-weight : huffman-tree -> number
(define huffman-tree-weight
  (lambda (ht)
    (cond
     ((huffman-leaf? ht)
      (huffman-leaf-weight ht))
     ((huffman-node? ht)
      (huffman-node-weight ht)))))
```

### 10.6.10 Decodierung eines Huffman-Codes

```
; Ein Bit ist entweder 0 oder 1 (Name: bit)

; Huffman-codierte Bitfolge decodieren
; huffman-decode : list(bit) huffman-tree -> list(string)
(define huffman-decode
  (lambda (bits ht)
    ...))
```

- Neuigkeit in dieser Prozedur
  - `bits` ist Liste
  - `ht` ist Huffman-Baum
- Beide müssen zerlegt werden!
- Die Steuerung obliegt den `bits`, daher verwende
  - das Muster für gemischte Datentypen für `ht` und
  - das Muster für Listenfunktionen für `bits`

### 10.6.11 Decodierung eines Huffman-Codes, II

Zunächst nur das erste Zeichen decodieren

```
; Erstes Zeichen einer Huffman-codierten Bitfolge decodieren
; huffman-decode-1 : list(bit) huffman-tree -> list(string)
(define huffman-decode-1
  (lambda (bits ht)
    (cond
     ((huffman-leaf? ht)
      (list (huffman-leaf-name ht)))
     ((huffman-node? ht)
      (cond
       ((empty? bits)
        empty)
       ((pair? bits)
        (if (= (first bits) 0)
            (huffman-decode-1 (rest bits) (huffman-node-left ht))
            (huffman-decode-1 (rest bits) (huffman-node-right ht))))))))))
```

```
; Huffman-codierte Bitfolge decodieren
; huffman-decode : list(bit) huffman-tree -> list(string)
(define huffman-decode
  (lambda (bits ht-root)
    (letrec ((decode-1
              (lambda (bits ht)
                (cond
                 ((huffman-leaf? ht)
                  (make-pair (huffman-leaf-name ht)
                             (decode-1 bits ht-root)))
                 ((huffman-node? ht)
                  (cond
                   ((empty? bits)
                    empty)
                   ((pair? bits)
                    (if (= (first bits) 0)
                        (decode-1 (rest bits) (huffman-node-left ht))
                        (decode-1 (rest bits) (huffman-node-right ht))))))))))
      (decode-1 bits ht-root)))
```

### 10.6.12 Huffman-Codieren

```
; huffman-encode : list(string) huffman-tree -> list(bit)
(define huffman-encode
  (lambda (names ht)
    (cond
     ((empty? names)
      empty)
     ((pair? names)
      (append (huffman-encode-name (first names) ht)
              (huffman-encode (rest names) ht))))))
```

Verbleibt zu definieren

```
; huffman-encode-name : string huffman-tree -> list(bit)
```

### 10.6.13 Huffman-Codieren eines Zeichens

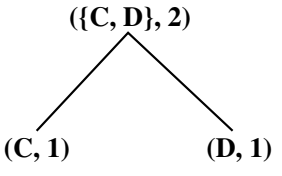
```
; huffman-encode-name : string huffman-tree -> list(bit)
(define huffman-encode-name
  (lambda (n ht)
    (letrec ((encode
              (lambda (ht bits)
                (cond
                 ((huffman-leaf? ht)
                  (reverse bits))
                 ((huffman-node? ht)
                  (let ((left (huffman-node-left ht))
                        (right (huffman-node-right ht)))
                    (if (member n (huffman-tree-names left))
                        (encode left (make-pair 0 bits))
                        (encode right (make-pair 1 bits)))))))))
      (encode ht empty))))
```

### 10.6.14 Aufbau eines Huffman-Baums

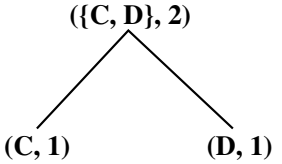
- Der Aufbau eines Huffman-Baums geschieht *bottom-up*, das heißt von den Blättern zur Wurzel.
- Vorbereitung: jeder Name wird mit seiner Häufigkeit in ein Huffman-Blatt verpackt.
- Eingabe ist die (nicht-leere) Liste dieser Blätter.
- Solange noch mindestens zwei Bäume in der Liste sind
  - Entferne die beiden Bäume mit niedrigstem Gewicht
  - Füge sie mit `make-huffman-node` zusammen
  - Lege den neuen Baum zurück in die Liste
- Der gesuchte Huffman-Baum ist das erste (einzige) Element der Liste.

### 10.6.15 Beispiel für den Aufbau

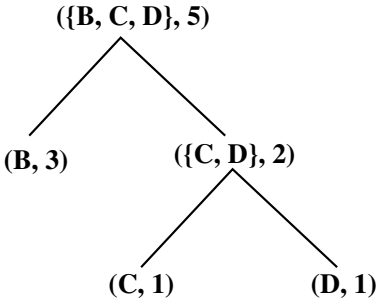
- Beginne mit $(B, 3)$, $(C, 1)$, $(D, 1)$.
- Neuer Baum aus $(C, 1)$ und $(D, 1)$:




- Weiter mit $(B, 3)$,




- Neuer Baum daraus:




- Ist auch Ergebnis

## 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 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.
- Ein Huffman-Baum ist ein Binärbaum, der eine Zeichencodierung mit variabler Länge durch den Weg von der Wurzel zu einem Zeichen am Blatt repräsentiert. Die Zeichen sind so an den Blättern des Baums verteilt, dass häufige Zeichen kurze Codeworte besitzen. Diese Codierung ist optimal, d.h., sie liefert im Mittel die kürzeste Codierung einer Zeichenfolge.