Kapitel 4

Bäume

4.1 Bäume, Datenstrukturen und Algorithmen

Zunächst führen wir Graphen ein. Die einfachste Vorstellung ist, dass ein Graph gegeben ist als

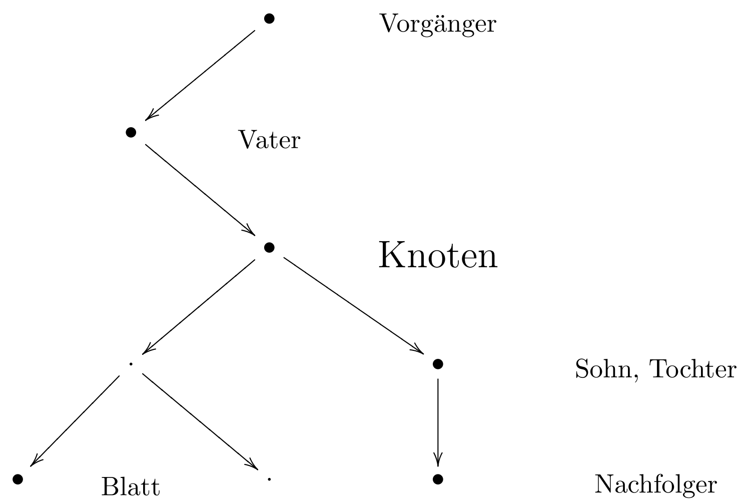
- eine Menge von Knoten und
- eine Menge von zugehörigen (gerichteten oder ungerichtete) Kanten zwischen den Knoten.

Einige Begriffe für ungerichtete Graphen sind:

Schlingen	Kanten mit gleichem Anfangs- und Endknoten
Wege	Kantenfolgen $(A, B), (B, C), \ldots$
Kreise	Kantenfolgen $(A, B), (B, C), \ldots, (Z, A)$
Erreichbarkeit	A ist von B aus erreichbar, wenn es einen Weg von A nach B gibt
zusammenhängend	Wenn alle Knoten von jedem Knoten aus erreichbar sind.
markierter Graph	Knoten bzw. Kanten haben Markierungen

Ein *Baum* ist ein (gerichteter oder ungerichteter) Graph, der zusammenhängend ist, ohne Kreise, und mit ausgezeichnetem Knoten (Wurzel), der keine Vorgänger hat. Ein Blatt ist ein Knoten ohne Nachfolger, ein innerer Knoten ein Knoten mit Vorgänger und Nachfolgern, d.h. weder Wurzel noch Blatt

In Zeichnungen wird meist die Wurzel oben hingezeichnet, die Blätter sind unten.



Einige wichtige Begriffe für Bäume:

geordneter Baum	Es gibt eine Links-Rechts-Ordnung auf den Töchtern
markierter Baum	Die Knoten (oder auch Kanten) haben Markierung.
Rand des Baumes	Liste der Blattmarkierungen eines geordneten Baumes
binärer Baum	Jeder Knoten ist Blatt oder hat genau zwei Töchter
Höhe (Tiefe)	maximale Länge eines Weges von der Wurzel zu einem Blatt
balancierter (binärer) geordneter Baum	hat unter (binären) Bäumen mit gleichem Rand kleinste Tiefe
Grad eines Knoten	Anzahl der Töchter
Grad eines Baumes	maximaler Grad eines Knoten

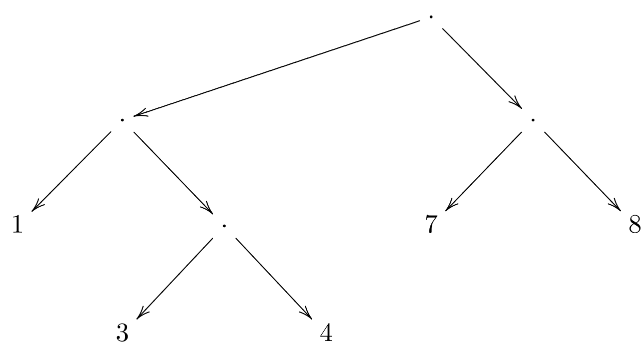
Wir betrachten zunächst Bäume mit folgenden Eigenschaften:

- Daten (Markierungen) sind an den Blättern des Baumes
- Die Daten sind von gleichem Typ
- Jeder (innere) Knoten hat genau zwei Tochterknoten
- es gibt einen linken und rechten Tochterknoten (geordnet)

Wir stellen binäre, geordnete Bäume, deren Blätter markiert sind, mit folgender Datenstruktur dar:

```
data Binbaum a = Bblatt  a | Bknoten (Binbaum a) (Binbaum a)
```

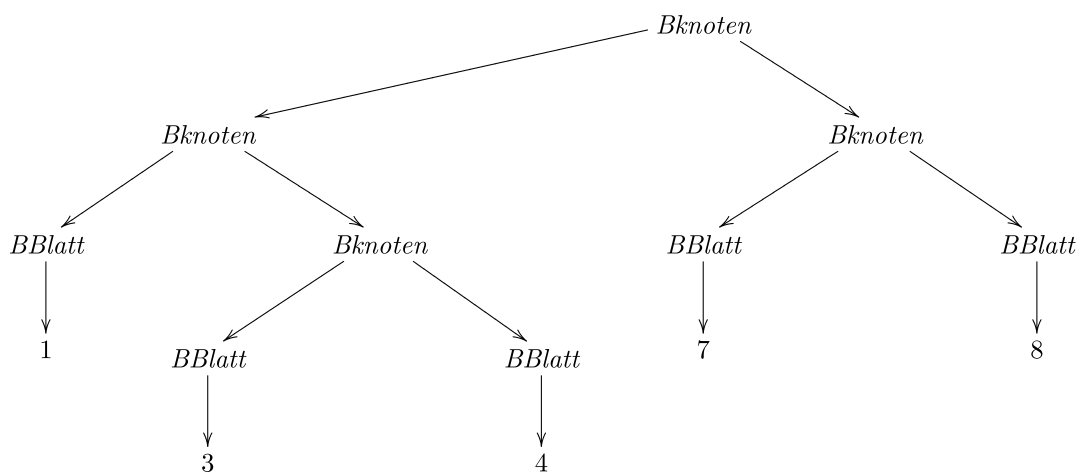
Beispiel 4.1.1 *Der folgende binäre Baum*



hat eine Darstellung als

```
Bknoten (Bknoten (Bblatt 1)
                 (Bknoten (Bblatt 3) (Bblatt 4)))
        (Bknoten (Bblatt 7) (Bblatt 8))
```

Der Syntaxbaum ist:



Einige Verarbeitungsfunktionen sind:
Berechnet die Liste der Markierungen aller Blätter eines Baumes:

```
b_rand (Bblatt x)      = [x]
b_rand (Bknoten bl br) = (b_rand bl)  ++ (b_rand br)
```

Diese Funktion testet, ob eine gegebene Markierung im Baum in einem Blatt vorkommt:

```
b_in x (Bblatt y)      = (x == y)
b_in x (Bknoten bl br) = b_in x bl  ||  b_in x br
```

Die Funktion `b_map` wendet eine Funktion auf alle Elemente des Baumes an, das Resultat ist der Baum aller Blatt-Resultate.

```
b_map f (Bblatt x)      = Bblatt (f x)
b_map f (Bknoten bl br) = Bknoten (b_map f bl)  (b_map f br)
```

Berechnung der Größe eines Baumes:

```
b_size (Bblatt x)      = 1
b_size (Bknoten bl br) = 1 + (b_size bl) + (b_size br)
```

Berechnung der Tiefe eines Baumes:

```
b_depth (Bblatt x) = 0
b_depth (Bknoten bl br) = 1 + max  (b_depth bl) (b_depth br)
```

Berechnung der Anzahl der Blätter eines Baumes:

```
b_blattnr (Bblatt x)      = 1
b_blattnr (Bknoten bl br) = (b_blattnr bl) + (b_blattnr br)
```

Berechnung der Summe aller Blätter eines Baumes, falls die Blätter mit Zahlen markiert sind:

```
b_sum (Bblatt x) = x
b_sum (Bknoten bl br) =
                  (b_sum bl) + (b_sum br)
```

Eine Funktion zum testweisen Erzeugung großer Bäume:

```
b_mkbt_test 0 k = Bblatt k
b_mkbt_test n k = Bknoten (b_mkbt_test (n-1) (k+1))
                          (b_mkbt_test (n-1) (2*k))

b_mkbt_testll [x]    = Bblatt x
b_mkbt_testll (x:xs) = Bknoten (Bblatt x) (b_mkbt_testll xs)

b_mkbt_testlr [x]    = Bblatt x
b_mkbt_testlr (x:xs) = Bknoten (b_mkbt_testlr xs) (Bblatt x)
```

Ein fold das die divide-and-conquer Methode verwendet, ist:

```
foldb :: (a -> a -> a)  -> Binbaum a -> a
foldb op (Bblatt x)    = x
foldb op (Bknoten x y) = (foldb op x) ‘op‘ (foldb op y)
```

Es ist geeignet für binäre Bäume, es ist sinnvoll für einen assoziativen Operator, es gibt keinen Initialwert für leere Bäume; und der Typ ist leicht eingeschränkt. Vorteil ist, dass es im Prinzip parallelisierbar ist.

Ein etwas schnelleres (aber sequentielles) fold über binäre Bäume kann man so definieren:

```
foldbt :: (a -> b -> b) -> b -> Binbaum a -> b
foldbt op a (Bblatt x)    = op x a
foldbt op a (Bknoten x y) = (foldbt op (foldbt op a y) x)
```

`foldbt` mit optimiertem Stackverbrauch:

```
foldbt’ :: (a -> b -> b) -> b -> Binbaum a -> b
foldbt’ op a (Bblatt x)    = op x a
foldbt’ op a (Bknoten x y) =
           (((foldbt’ op) $! (foldbt’ op a y)) x)
```

Effizientere Version von `b_rand` und `b_sum` sind:

```
b_rand_eff  = foldbt (:) []
b_sum_eff   = foldbt’ (+) 0
```

Um zu begründen, warum `foldbt` relativ schnell ist, betrachte den Zwischenausdruck, der aus einem Baum `tr` mit der Struktur $(((1,2),3),(4,5))$ entsteht, wenn man `foldbt (+) 0 tr` auswertet.: (Wir verwenden hier eine etwas vereinfachte Notation)

```
foldbt (+) 0 (((1,2),3),(4 ,5))
-->  foldbt (+) (foldbt (+) 0 (4 ,5))    ((1,2),3)
-->  foldbt (+) (foldbt (+) (foldbt (+) 0 (5)) (4)    ((1,2),3))
-->  foldbt (+) (foldbt (+) (5+0) (4)    ((1,2),3))
-->  foldbt (+) (4+ (5+0))   ((1,2),3)
-->  foldbt (+) (foldbt (+) (4+ (5+0))   (3))  (1,2)
-->  foldbt (+) (3+ (4+ (5+0)))   (1,2)
-->  foldbt (+) (foldbt (+) (3+ (4+ (5+0))) (2)  (1))
-->  foldbt (+)  (2+ (3+ (4+ (5+0))))  (1)
-->  1+ (2+ (3+ (4+ (5+0))))
```

D.h. Wenn ein binärer Baum tr mit Randliste $[a_1, \ldots, a_n]$ gegeben ist, dann entspricht `foldbt f a tr` dem Ausdruck `f a_1 (f a_2 (... (f a_n a) ...s))`. D.h. es entspricht einem `foldr (++) []`, das z.B.als Definition für `concat` schneller als `foldl (++) []`.

Wir zeigen beispielhaft eine Analyse der Funktion `b_rand:`, wobei wir nur volle binäre Bäume betrachten.

Für diese Bäume gilt:
#(innere Knoten) + 1 = #(Blätter) = 2^{Tiefe}

Für die Anzahl der Reduktionen von `b_rand baum` bei n Blättern gilt:

$$\begin{array}{lcll}
\tau(n) & = & n/2 + 2 + 2 * \tau(n/2) & \\
 & = & n/2 + 2 + 2 * (n/4 + 2 + 2 * \tau(n/4)) & \\
 & = & n/2 + 2 + n/2 + 2 * 2 + 4 * \tau(n/4)) & \\
\ldots & = & \ldots & \\
 & = & n/2 * log_2(n) + 2 * n & \text{Tiefe} = log_2(n)
\end{array}$$

Hinzu kommen noch n Reduktionen für die Blätter .

Da wir diese Analyse mit der Statistik (des Interpreters Hugs) vergleichen können, nehmen wir noch folgende Funktion `is_list` hinzu und werten `is_list (b_rand testbaum_n)` aus.

```
is_list []     = True
is_list (_:xs) = is_list xs
```

Deshalb kommen noch `(is_list lst)` = `length lst` Reduktionen dazu. Die folgende Tabelle zeigt die Anzahl der Reduktionen von `is_list (b_rand testbaum_n)` allgemein und für einige ausgewählte Werte.

Tiefe	#Blätter	#berechnet	#tatsächliche
m	2^m	$2^m + 2^{m-1} * m + 3 * 2^m$	$2^m + 2^{m-1} * m + 3 * 2^m + 14$
	n	$n + n/2 * log_2(n) + 3 * n$	$n + n/2 * log_2(n) + 3 * n + 14$
10	1024	9216	9230
12	4096	40960	40974
13	8192	86016	86030
14	16384	180224	180238

Wir machen auch eine Analyse von `b_rand_eff`, um vergleichen zu können, ebenfalls nur für volle binäre Bäume. Zur Erinnerung nochmal die Reduktion von `foldbt`:

```
     foldbt (:) [] (Bknoten lb rb)
→   foldbt (:) (foldbt (:) []  rb) lb
```

Die Anzahl der Reduktionen bei n Blättern kann man wie folgt abzählen: pro `Bknoten` wird ein foldbt-Ausdruck eingesetzt. Dies erfordert $n - 1$ Reduktionen zu foldbt-Ausdrücken.

Pro Blatt wird die Reduktion `(foldbt (:) rand (Bblatt a))` $\rightarrow$ `a : rand` ausgeführt, d.h. pro Blatt ein Reduktionsschritt.

Die Gesamtanzahl der Reduktionen ist somit in etwa $2 * n$.

Folgende Tabelle zeigt die theoretischen und die praktisch ermittelten Werte des Ausdrucks `is_list (b_rand_eff testbaum_n)`

Tiefe	#Blätter	#berechnet	#tatsächliche	#*Red*(`b_rand`)
m	2^m	$2^m + 2 * 2^m$	$2^m + 2 * 2^m + 15$	
	n	$3n$	$3n + 15$	
10	1024	3072	3087	9230
12	4096	12288	12303	40974
13	8192	24576	24591	86030
14	16384	49152	49167	180238

Man sieht, dass `foldbt` tatsächlich schneller ist als normales rekursives Programmieren. Der Grund ist der gleiche wie bei der Beschleunigung des mit `foldl` programmierten `concat` durch das mit `foldr` programmierte. Was man auch sieht, ist dass lineare Terme (hier $3n$) die logarithmische Verbesserung $n/2 * (log)_2(n)$ etwas dämpfen. Erst bei sehr großen Bäumen sind die Effekte deutlich sichtbar.

4.1.1 Haskell-Library zu Suchbäumen

Data.Map implementiert balancierte Suchbäume in Haskell:

Eine „Map“ ist eine Menge von (Schlüssel,Wert) - Paaren, so dass pro Schlüssel nur ein Paar vorkommt. Dies kann man auch als Funktion von der Menge der Schlüssel in die Menge der möglichen Daten verstehen (*map*), wobei ein fehlender Schlüssel so interpretiert, dass die Funktion dort undefiniert ist.

Vordefinierte Funktionen auf Map-Objekten sind:

singleton	erzeugt Suchbaum mit einem (Schlüssel, Wert)-Paar
insert	fügt ein Element ein.
delete	löscht ein Element zu gegebenem Schlüssel.
lookup	Findet eine Element zu gegebenem Schlüssel
adjust	ändert Wert zu gegebenem Schlüssel

4.1.2 Allgemeine Bäume

Man kann weitere Funktionen auf Bäumen definieren und auch etwas allgemeinere Bäume als Datenstruktur verwenden. Zum Beispiel kann man dann die folgende Typ- und Konstruktordefinition verwenden:

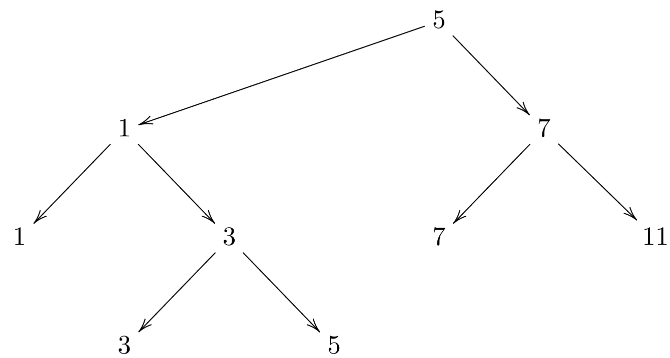
```
data Nbaum a = Nblatt a | Nknoten [Nbaum a]
```

Ein Beispiel für die angepasste Randfunktion ist:

```
nbaumrand :: Nbaum a -> [a]
nbaumrand   (Nblatt x) = [x]
nbaumrand   (Nknoten xs)  = concatMap nbaumrand xs
```

4.1.3 Suchbaum

Die Implementierung einer sortierten Liste als Baum hat Effizienzvorteile beim Zugriff: Jeder Knoten enthält als Markierung den größten Schlüssel des linken Teilbaumes mit den kleineren Werten.




```
---   Suchbaum

data Satz a = Satz Int  a     ---  Schluessel  +  Daten

data Suchbaum a =
   Sblatt  Int a | Sknoten (Suchbaum a) Int (Suchbaum a)
                 | Suchbaumleer

{-  Verwendung:    Suchbaum (Satz a)
     intial:   Suchbaumleer     -}
einfuegeDbS (Satz x sx)  Suchbaumleer = Sblatt x (Satz x sx)
einfuegeDbS (Satz x sx)  (Sblatt  k satzk) =
       if x < k
       then Sknoten (Sblatt x (Satz x sx)) x (Sblatt k satzk)
       else if x == k then error " schon eingetragen"
       else Sknoten  (Sblatt k satzk) k (Sblatt x (Satz x sx))
einfuegeDbS (Satz x sx)  (Sknoten l k r) =
        if x < k then Sknoten (einfuegeDbS (Satz x sx)  l) k r
        else if x == k then  error " schon eingetragen"
        else Sknoten l k (einfuegeDbS (Satz x sx)  r)
```

Das elementweise Einfügen sortiert die Eingabe kann also auch als Sortierverfahren verwendet werden.

Der Nachteil dieses Verfahrens liegt darin, dass es in manchen Fällen passieren kann, dass der entstehende Baum unbalanciert ist, d.h. er kann lineare Tiefe haben. Das bewirkt eine Verschlechterung der Laufzeit des Zugriffs von $O(log(n))$ auf $O(n)$.

Man kann Abhilfe schaffen durch ein Balancieren des Baumes durch zwischengeschaltete Rotationen. Die Laufzeit des Aufbaus des Baumes ist, wenn Balance mitberücksichtigt wird: $O(n * log(n))$

```
istinDbS (Satz x y) Suchbaumleer = False
istinDbS (Satz x y) (Sblatt k _)   =  (x == k)
istinDbS (Satz x y) (Sknoten  bl k br) =
           if x < k then  istinDbS (Satz x y) bl
           else if x == k then True
           else  istinDbS (Satz x y) br

erzeugeDbS [] = Suchbaumleer
erzeugeDbS (x:xs) = einfuegeDbS (Satz x x)  (erzeugeDbS xs)
druckeDbS sb = blaetterSB

foldSB :: (a -> b -> b) -> b -> Suchbaum a -> b
foldSB op a Suchbaumleer = a
foldSB op a (Sblatt _ x) = op x a
foldSB op a (Sknoten x _ y) = (foldSB op (foldSB op a y) x)

blaetterSB = foldSB (:) []
```