Fortgeschrittene Algorithmen und Datenstrukturen

Texte zur Vorlesung

Johannes Waldmann
Institut für Informatik
Universität Leipzig

Wintersemester 99/00

Inhaltsverzeichnis

Kapitel 1

Vorwort

1.1 Ziel und Methode

Wir beschäftigen uns mit

- Entwurf und Analyse
- von korrekten und effizienten
- Algorithmen und Datenstrukturen.

Die Notwendigkeit dieser Untersuchungen leuchtet sicher jedem aktiven Programmierer, aber auch Programmbenutzer, unmittelbar ein.

In der Vorlesung lernen Sie, durch

- mathematisch exaktes Argumentieren, aber auch
- Lesen (!) und Schreiben von tatsächlichen Programmen,

clevere algorithmische Ideen zu verstehen.

Fundamentale Techniken dazu kennen Sie bereits aus dem Grundstudium. Wir werden das hier vertiefen und erweitern.

1.2 Inhalt

Die Gliederung der Vorlesung sieht so aus:

Prioritätswarteschlangen

Abstrakte Datentypen, balancierte binäre und binomiale Bäume, leftist und binomial heaps, amortisierte Analyse, skew und fibonacci-heaps.

Schwerpunkt liegt auf *Strukturen* (Bäumen).

Literatur: [CLR91, Wei93, Oka98]

Anwendungen: event-driven simulation, Graph-Algorithmen

Sortieren und Mischen

Untere Schranken, Beweis durch Spielstrategien, merge insertion, Turniere, Sortiernetze

Schwerpunkt liegt auf *Algorithmen*.

Da gehen wir nach dem Buch von Knuth [Knu98]. Das ist klassisch.

Suchbäume

AVL-Bäume, Splay-Trees, a-b-Bäume, rot-schwarz-Bäume, top-down-a-b, Zahlensysteme (skew binary), Braun trees.

Jetzt betonen wir wieder die *Strukturen*.

Literatur: [CLR91, Wei93, Hin98, Oka98]

Anwendungen: `FiniteMap.hs` in ghc

Verschiedenes

Erstmal sehen, wie weit wir kommen.

Möglich sind hier zum Beispiel String-Matching-Algorithmen unter Anwendung endlicher Automaten [CLR91],

und Silbentrennung in TeX.

Anwendungen: `grep`.

1.3 Dieses Skript

Im Literaturverzeichnis finden Sie sowohl

- gut ausgearbeitete Lehrbücher und Skripte als auch
- Übersichtsdarstellungen aktueller Forschungen.

Das ist mehr als genug; deswegen beschränkt sich der hier vorliegende Begleittext zur Vorlesung auf mehr oder weniger ausführlich kommentierte Verweise auf die Literatur, gewürzt mit passenden Übungsaufgaben.

Weiterhin gebe ich in manchen Abschnitten Haskell-Texte an. Sie sollten diese (obwohl sie meist ausführbar sind) weniger als Implementierung denn als *Spezifikation* von Algorithmen verstehen. Sie müssen überhaupt nicht Haskell programmieren können, aber Sie sollten üben, die Programme als Gleichungssysteme zu lesen, und diese auch korrekte umzuformen.

Auf diese Weise werden wir Eigenschaften von Algorithmen beweisen. Da Haskell streng funktional, also nebenwirkungsfrei ist, sind solche Umformungen überhaupt nichts anderes als das, was Mathematiker tagein, tagaus tun: etwas definierendes durch das Definierte, und überhaupt etwas Gleiches durch Gleiches ersetzen.

Ich werde Skript und Programme auf einer WWW-Seite veröffentlichen. Jede Skript-Seite trägt in der Kopfzeile das Datum ihrer letzten Änderung. Der Kasten neben einem Programmtext im Skript enthält den Namen des Files, in welchem das Programm zu finden ist. Die Programme können Sie unter `hugs-1.4` interpretieren oder mit `ghc-2.10` kompilieren. Zu Haskell (Sprache und Implementationen) siehe www.haskell.org.

Kapitel 2

Abstrakte Datentypen

2.1 Beispiel zur Motivation

Wir schreiben einen Sequenzer für Musik.

- Eingabe: Beschreibung einer Ereignismenge
  - Elementar-Ereignis: Ton spielen, mit gewissen Parametern (Laustärke, Dauer, Instrumenten-Nummer, usw.)
  - Steuer-Ereignis: untergeordnete Ereignisse parallel spielen, oder wiederholen.
- Ausgabe: Elementar-Ereignisse, der Zeit nach geordnet (MIDI-Daten)

Wir wünschen uns eine Datenstruktur, in der alle "derzeit aktiven" (Elementar- und Steuer-)Ereignisse stehen. Daraus suchen wir jeweils dasjenige, das als nächstes auszuführen ist, und entfernen es aus der Struktur.

Elementar-Ereignisse schicken wir an das Instrument.

War es ein Steuer-Ereignis, bedeutet "Ausführung": füge gewisse neue (zukünftige) Ereignisse in die Datenstruktur ein.

Für Ereignismengen haben wir zu implementieren:

- Ereignis einfügen
- das früheste Ereignis finden und dann entfernen

Diese Eingeschaften sind die Spezifikation des Abstrakten Datentyps (ADT) Prioritätswarteschlange (PQ).

PQs verwenden wir gern bei "event driven simulation" (weil die Events bestimmen, wie der Prozess weiter verläuft). Die Musiksteuerung ist schon ein Beispiel, ein anderes ist der Partikelsimulator von Lee Killough [Kil99], siehe dort `pool.zip`.

Aufgabe: File holen, Quelltext und Beschreibung anschauen, und beantworten: welche Operationen mit PQs werden dort gebraucht?

2.2 Definition

Definition 1 *Ein* Abstrakter Datentyp *besteht aus Signatur und Axiomen.*

Die Signatur *deklariert Funktionsnamen und -Typen.*

Die Axiome *beschreiben Eigenschaften der Funktionen.*

Abstrakt heißt: wir können den ADT benutzen, ohne seine Implementierung zu kennen.

Wir können Programme mit ADTs nicht nur *schreiben*, sondern auch *beweisen*: ihre Eigenschaften folgen aus den Axiomen.

Wir lassen offen, welches Typsystem wir zugrunde legen und in welcher (Logik-)Sprache wir die Axiome formalisieren. Das ist OK, da wir keine maschinellen Korrektheitsbeweise führen wollen. Wir wählen jeweils eine fallweise passende und ausreichende Notation.

Genaueres finden Sie in Vorlesungen (bzw. Skripten) von Dr. Hartwig (Spezifikation, Verifikation). [Har98, Har97]

2.3 Signatur - Beispiel PQ

Wir haben eine Menge E (die Ereignisse), darauf eine Ordnung $<$ (nach der Zeit). Die Menge der Ereignismengen ist M. Die benötigen Operationen sind

- singleton: $E \to M$ (Einermenge herstellen)
- union: $(M \times M) \to M$ (vereinigen)
- extract: $M \to (E \times M)$ (zerlegen in kleinstes Element und Restmenge)

Der Vollständigkeit halber ergänzen wir das um

- empty: M (leere Menge herstellen)
- null: $M \to \mathsf{Bool}$ (testen, ob Menge leer ist)
- insert: $(E \times M) \to M$ (ein Element einfügen)

In Haskell schreiben wir so:

$\boxed{\texttt{PQ}}$

```
class PQ pq where
  empty     :: Ord a => pq a
  is_empty  :: Ord a => pq a -> Bool

  singleton :: Ord a => a -> pq a
  union     :: Ord a => pq a -> pq a -> pq a

  extract   :: Ord a => pq a -> (a, pq a)
```

Wir wollen `insert` immer durch `union` implementieren

```
insert    :: (PQ pq, Ord a) => a -> pq a -> pq a
insert x p = union (singleton x) p
```
PQ

2.4 Axiome - Beispiel PQ

Die Axiome schreiben wir jetzt nicht formal auf. Der Inhalt sollte aber klar sein:

- kein eingefügtes Element geht verloren, d. h. alle kommen irgendwann wieder raus.
- es kommen nur Elemente raus, die wir eingefügt hatten, d. h. keins entsteht aus Nichts.
- was rauskommt, ist wirklich $\leq$ allen, die noch drin sind.

2.5 Anwendung

Damit können wir bereits ein Sortierprogramm schreiben. Das folgende ist die funktionale Sprache Haskell (`www.haskell.org`), aber das stört uns gar nicht, wir lesen die Zeilen einfach als Gleichungen.

```
pqsort :: Ord a => [a] -> [a]
pqsort = contents . build
```
none

```
build :: (PQ h, Ord a) => [a] -> h a
build [] = empty; build (x : xs) = insert x (build xs)

contents :: (PQ h, Ord a) => h a -> [a]
contents pq = if is_empty pq then []
              else let (t, pq') = extract pq in  t : contents pq'
```
PQ

Das Sortieren ist aber *nicht* die sinnvollste Anwendung von PQs: Wenn wir *nur* sortieren wollen, nehmen wir lieber einen anderen Algorithmus. Die PQs können nämlich *viel mehr* (zum Beispiel union), und das brauchen wir z. B. in Scheduling-Algorithmen (siehe einführendes Beispiel).

2.6 Nochmal Axiome

Der vorige Abschnitt sagte: wenn wir eine PQ-Implementierung haben, können wir damit sortieren. Wir drehen das rum: Wenn eine Struktur mit Funktionen `empty`, `insert`, `extract` die Eigenschaft hat, daß obige Funktion `sort` wirklich sortiert, dann nennen wir die Struktur PQ.

Eine Spezifikation von `sort` lautet

- `sort` bekommt eine Liste, und erzeugt eine Liste, von Elementen, die geordnet werden können.
- die Ergebnisliste ist eine Permutation der Argumentliste.
- die Ergebnisliste ist monoton wachsend geordnet.

In Haskell sieht das so aus

```
sort ::Ord a => [a] -> [a]                                              none
is_perm xs (sort xs) && is_monotone (sort xs)

span                :: (a -> Bool) -> [a] -> ([a],[a])
span p []            = ([],[])
span p xs@(x:xs')
         | p x       = let (ys,zs) = span p xs' in (x:ys,zs)
         | otherwise = ([],xs)


is_perm :: Eq a => [a] -> [a] -> Bool                                   Assert
is_perm [] ys = null ys
is_perm (x : xs) ys =
    case span (/= x) ys of
         (pre, p : post) -> is_perm xs (pre ++ post)
         (_   , []      ) -> False

is_monotone :: Ord a => [a] -> Bool
is_monotone [] = True; is_monotone [x] = True
is_monotone (x : y : ys) = x <= y && is_monotone (y : ys)
```

Übung: wie spezifizieren wir die Funktion `union`? So:

```
contents (union (build xs) (build ys)) = contents (build (xs ++ ys))   none
```

Warum verlangen wir *nicht* `union (build xs) (build ys) = build (xs ++ ys)`?

Kapitel 3

Prioritäts-Warteschlangen

Wir haben die Spezifikation von PQs gesehen, sowie zwei Anwendungen (Sortieren, Scheduling), davon eine wirlklich wichtige (Scheduling).

Jetzt suchen wir Implementationen. Sie kennen binäre Heaps.

Übung: welche der PQ-Operationen werden effizient unterstützt? Antwort: union jedenfalls nicht.

Leftist Heaps Da geht alles in $\log n$ Zeit, auch union.

Wir zeigen daran eine einfache Art des Balancierens, und führen die entsprechenden Beweise möglichst vollständig.

(Später werden die Strukturen komplizierter, und die Beweise lang. Damit wir sie dann effektiv hinschreiben können, üben wir es jetzt ausführlich.)

Binomiale Heaps Wieder alles in logarithmischer Zeit.

Wir zeigen daran eine weitere Idee, Struktur in Bäume zu bekommen: durch Vergleich mit Zahlensystemen. (Das werden wir noch öfter benutzen.)

Außerdem brauchen wir Binomialheaps, weil wir daraus Fibonacci-Heaps bauen, die für eine bestimmte Operation *noch schneller* sind.

Skew Heaps Erhalten wir aus Leftist Heaps, indem wir das Rebalancieren *einfach vergessen*.

Das geht trotzdem gut. Um das zu begründen, brauchen wir eine neue Technik: *amortisierte Analyse*. Diese ist von allgemeinem Interesse (kommt bei Fibonacci-Heaps vor, aber auch vielen anderen Strukturen, nicht nur PQs).

Fibonacci-Heaps Enstehen aus Binomialen Heaps, indem wir (wie bei Leftist $\to$ Skew) eine Operation zum Rebalancieren vergessen. Tatsächlich nicht vergessen, sondern an eine andere Stelle verschieben.

Dadurch werden die Operationen `insert, decrease-key` sehr schnell (konstante Laufzeit!). Das ist genau die Eigenschaft, die wir brauchen, wenn wir damit Prims Algorithmus für minimale aufspannende Bäume implementieren.

Beim Nachweis der amortisierten Zeitschranke tauchen Fibonacci-Zahlen auf.

3.1 Ansatz: PQs als Binärbäume

Wir treffen die wenig überraschende Entscheidung, PQs als binäre Bäume zu implementieren.

```
data Bin a = Leaf
           | Node { tag :: a, left :: Bin a, right :: Bin a
                  , siz :: Int, dep :: Int }
   deriving Show
```
BinType

Statt `Node` verwenden wir den smart constructor `node`:

```
node :: k -> Bin k -> Bin k -> Bin k
node k l r = Node { tag = k, left = l, right = r
                  , dep = 1 + max (depth l) (depth r)
                  , siz = 1 + size l + size r }
```
BinType

Ein leftist heap ist ein binärer Baum:

```
newtype Leftist k = Leftist (Bin k) deriving Show
instance (Show k) => XForm (Leftist k) where
    xform (Leftist t) = xform t
```
Leftist

Die Elemente stehen in allen inneren Knoten, und für jeden Teilbaum gilt: das minimale Element steht in der Wurzel.

```
is_heap_ordered t =
  and [ root_is_minimal s | s <- subtrees t, has_key s ]
    where root_is_minimal s =
            and [ key u <= key s | u <- subtrees s, has_key u ]
```
Tree

Beachte `and [] == True`. Beachte auch den Typ von `is_heap_ordered` (Hinweis: nächste Seite).

Daraus folgt schon die Implementierung einfacher Funktionen:

```
instance PQ.PQ Leftist where
  empty = Leftist Leaf
  is_empty (Leftist Leaf) = True; is_empty _ = False
  singleton x = Leftist ( node x Leaf Leaf )
```
Leftist

Diese Operationen sind alle in konstanter Zeit ausführbar.

Für `extract` entfernen wir die Wurzel und verschmelzen die Inhalte ihrer Kinder.

Leftist

```
extract (Leftist Leaf) = error "cannot extract anything from empty heap"
extract (Leftist n) = (key n, Leftist ( merge (left n) (right n)) )

union (Leftist x) (Leftist y) = Leftist ( merge x y )
```

Die Implementierung von `union` verschieben wir noch etwas.

3.2 Bäume, im Allgemeinen

Ein Baum besteht aus Knoten, die (eventuell) Schlüssel und Kinder haben.

Tree

```
class Tree t where
  maybe_key :: t a -> Maybe a
  children  :: t a -> [ t a ]
  construct :: Maybe a -> [ t a ] -> t a
  empty     :: t a
  is_empty  :: t a -> Bool
```

Es könnte Knoten ohne Schlüssel geben, deswegen ist das Resultat von `key` ein `Maybe`.

Tree

```
  size   :: t a -> Int
  size  t = 1 + sum     [ size  c | c <- children t ]

  depth  :: t a -> Int
  depth t = 1 + maximum ( 0 : [ depth c | c <- children t ] )
```

Bem: Wir brauchen hier die `0`, weil `maximum []` nicht definiert ist.

Bem: `size` und `depth` sind Klassenmethoden, die wir mit default-Deklarationen ausstatten. Wir gestatten es aber, daß Instanzen diese Methoden durch schnellere überschreiben. (Siehe smart constructors.)

Tree

```
key       :: Tree t => t a -> a
key = the . maybe_key

has_key :: Tree t => t a -> Bool
has_key t = exists (maybe_key t)

is_node, is_leaf :: Tree t => t a -> Bool
is_leaf t = null (children t); is_node = not . is_leaf
```

Davon können wir bereits wichtige Operationen auf Bäumen ableiten

Tree

```
rank   :: Tree t => t a -> Int
rank  t = length (children t)

subtrees :: Tree t => t a -> [ t a ]
subtrees t = t : [ s | c <- children t, s <- subtrees c ]
```

Bemerkung 1: Alle diese Funktionen folgen einem Rekursions-Schema, das kanonisch zur Definition des Typs `Bin` bzw. der Klasse `Tree` paßt. Es sind Katamorphismen. Die Theorie dazu gibts in einer Vorlesung Fortgeschrittenes Funktionales Programmieren.

BinType

```
instance Functor Bin where
    fmap f Leaf = Leaf
    fmap f t    = node (f (key t)) (fmap f (left t)) (fmap f (right t))
```

Bemerkung 2: In `subtrees` hängen wir implizit Listen aneinander. Implementieren wir das wirklich so, wird es ineffizient. Es gibt aber eine leichte Transformation, die das behebt. (Die Theorie dazu ... siehe vorhin.)

Bemerkung 3: Diese Ineffizienz stört uns jedoch hier überhaupt nicht! Wir wollen ja die Funktion `subtrees` gar nicht ausführen. Sie steht nur da, damit wir eine Bezeichnung für die Liste aller Teilbäume haben. Diese brauchen wir manchmal, wenn wir Programmtransformationen beweisen.

Übung: finde einen Zusammenhang zwischen `size t` und `length (subtrees t)`. Beweise!

3.3 Maybe

Wir haben den Typkonstruktor `Maybe` benutzt:

none

```
data Maybe a = Just a | Nothing
```

MMaybe

```
exists :: Maybe a -> Bool
exists (Just _) = True; exists Nothing = False

the :: Maybe a -> a
the (Just x) = x; the Nothing = error "the"
```

3.4 Binäre Bäume, im Besonderen

Natürlich sind binäre Bäume eine Instanz von Bäumen

BinType

```
instance Tree Bin where
  maybe_key Leaf = Nothing; maybe_key n = Just (tag n)
  children Leaf = []; children t = [ left t, right t ]
  empty = Leaf; is_empty Leaf = True; is_empty _ = False
  construct x [l, r] = Node { tag = the x, left = l, right = r }
  size Leaf = 1; size n = siz n
  depth Leaf = 1; depth n = dep n
```

Übung:

none

```
t :: Bin a  -->  1 <= size t <= 2 ** (depth t) - 1
```

Wir benötigen spezielle Maße auf binären Bäumen

BinType

```
rdepth :: Bin a -> Int
rdepth Leaf = 1; rdepth t = 1 + rdepth (right t)

ldepth :: Bin a -> Int
ldepth Leaf = 1; ldepth t = 1 + ldepth (left  t)
```

Übung: `1 <= rdepth t <= depth t` und `1 <= ldepth t <= depth t`.

Tree

```
leftmost  :: Tree t => t a -> t a
leftmost t =
    let cs = children t; c = head cs
    in  if null cs || is_empty c then t else leftmost c
```

```
rightmost :: Tree t => t a -> t a
rightmost t =
    let cs = children t; c = last cs
    in  if null cs || is_empty c then t else rightmost c
```

3.5 Zeitschranken

Übrig bleibt, `union` zu implementieren. Das soll schnell gehen. Das wollen wir dadurch erreichen, daß wir den benutzten binären Bäumen gewisse Strukturforderungen auferlegen, d. h. eine gewisse Balance erzeugen. Dabei Vorsicht: ist die Forderung zu schwach, wird die Balance zu gering (der Baum zu tief, die Operationen zu teuer); ist sie andererseits zu stark, dauert das nach jeder Operation nötige Rebalancieren zu lange.

Übung: irgendeine “zu einfache” Implementierung von union untersuchen lassen (die unbalancierte Bäume erzeugt). (Siehe nächster Abschnitt.)

Wir fragen nach dem besten nach dieser Idee erreichbaren Verhalten. Wir nehmen an, daß wir wenigstens auf einem Pfad wirklich von Wurzel zu Blatt laufen müssen. Also nehmen wir die maximale Baumtiefe als Laufzeitschranke von guten Algorithmen. Kein binärer Baum mit n Knoten hat alle Blätter weniger tief als $\log n$. Die “besten” binären Bäume sind die mit (fast) gleichviel Knoten rechts und links unterhalb der Wurzel jedes Teilbaums. Wir rechnen demnach damit, daß wir `union a b` in $O(\log \mathtt{a} + \log \mathtt{b})$ Zeit hinbekommen, also `insert a x` und `extract a` in $O(\log \mathtt{a})$.

Übungen: Welche Zeit erhalten wir damit für den Sortieralgorithmus? Wenn wir `union` in $O(1)$ schaffen würden— wie schnell könnten wir sortieren? Was sagen Sie dazu?

3.6 Listen als PQs

Die einfachste Implementierung einer PQ ist es, alle Elemente in einer geordneten Liste aufzubewahren.

ListPQ

```
instance PQ [] where
  empty = []; is_empty = null
  singleton x = [ x ]
  extract (x : xs) = (x, xs)

  union [] ys = ys; union xs [] = xs
  union (x : xs) (y : ys) =
    if x <= y then x : union  xs (y : ys) else y : union (x : xs) ys
```

Hierbei ist `union` die bekannte Operation des verschmelzens zweier geordneter Listen zu einer einzigen.

3.7 Leftist Heaps

Wir sehen in einem binären Baum am rechten Rückgrat entlang und betrachten die Liste der linken Teilbäume des Rückgrats.

Für `union` von Bäumen verschmelzen wir die beiden Rückgrat-Listen.

```
union :: Ord a => Bin a -> Bin a -> Bin a                            none
union Leaf y = y; union x Leaf = x
union x y =
    if tag x <= tag y
    then Node { tag = tag x, left = left x, right = union (right x) y }
    else Node { tag = tag y, left = left y, right = union x (right y) }
```

Wir müssen uns noch überlegen, wie wir verhindern, daß die Rückgrate zu lang werden.

Übung: was passiert, wenn wir das vergessen und mit eben hingeschriebener Implementierung `build` (oder `sort`) ausführen?

Wir möchten folgenden Trick anwenden: Wenn in einem Baum `t` das rechte Rückgrat von `right t` länger ist als das von `left t`, dann vertauschen wir diese Teilbäume.

Wir fordern

```
is_leftist_shaped, is_leftist :: Ord a => Bin a -> Bool              Leftist
is_leftist_shaped t = and [ is_biased s | s <- subtrees t ]
   where is_biased Leaf = True
         is_biased n    = rdepth (left n) >= rdepth (right n)
is_leftist t = is_leftist_shaped t && is_heap_ordered t
```

Wir haben die wichtige Eigenschaft von leftist heaps:

```
is_leftist_shaped t  -->  size t >= 2 ** (rdepth t) - 1              none
```

Beweisen (Induktion)! Andersrum gesagt, das Rückgrat ist nur logarithmisch tief, deswegen erwarten wir vernünftige Laufzeiten für die Operationen.

Übung: Wie tief kann ein leftist heap nach links gehen? Stört das?

3.8 Aufgaben zu Leftist Heaps

Hier geht es noch nicht um Algorithmen, sondern nur um Balance-Eigenschaften. Sie sollen verstehen, was es für einen Baum `t :: Bin a` bedeutet, daß `is_leftist_shaped t` gilt.

Wir definieren

```
is_left_dominated t = and [ ldepth s >= rdepth s | s <- subtrees t ]    Leftist
```

1. Zeige `is_leftist_shaped t --> is_left_dominated t`
2. Zeige (durch Gegenbeispiel), daß die Umkehrung davon *nicht* gilt.
3. Wir hatten: `is_leftist_shaped t --> rdepth t = O (log (size t))`.

   Welche Schranke für `rdepth t` erhalten wir, wenn nur `is_left_dominated t` gilt?
4. Lesen Sie in [Knu98] oder [Wei93] die dortige Definition von Leftist heaps. Schreiben Sie sie formal auf (dazu müssen Sie *null path length* definieren). Vergleichen Sie mit unserer Definition von Leftist heaps.

3.9 `union` für Leftist Heaps

Wir fügen das Vertauschen der Teilbäume in den Code für `union` ein:

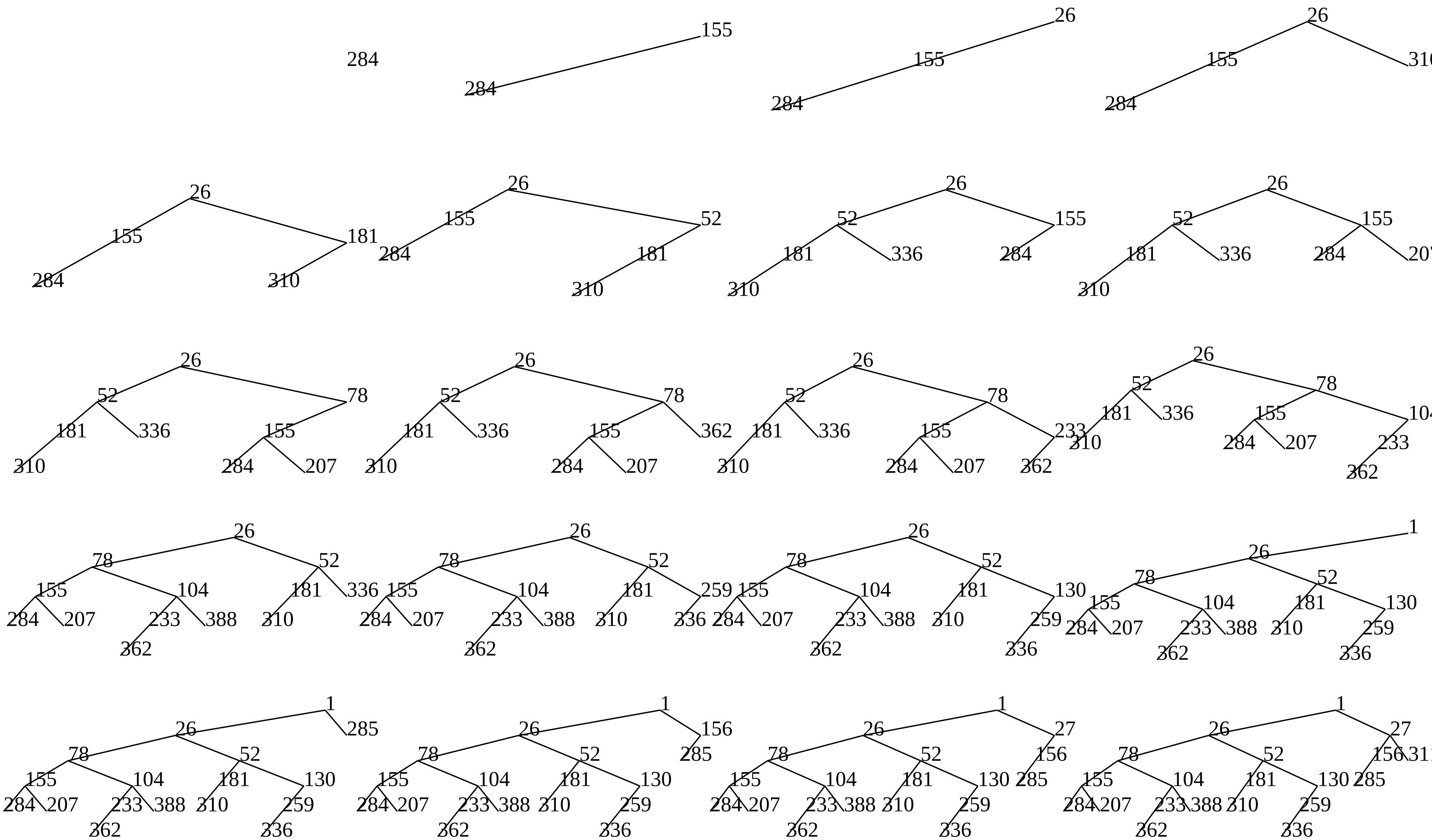
```
merge :: Ord a => Bin a -> Bin a -> Bin a                                Leftist
merge Leaf y = y; merge x Leaf = x
merge x y =
   let (m, l, r) = if key x <= key y
                   then (key x, left x, merge (right x) y)
                   else (key y, left y, merge x (right y))
   in  if  rdepth l >= rdepth r  then  node m l r  else  node m r l
```

Beweis: wir müssen zeigen

```
is_leftist x && is_leftist y --> is_leftist (union x y)                  none
```

das geht durch Induktion nach `size x + size y`.

Auf der nächsten Seite sehen wir eine Folge von Einfügungen in einen leftist heap


284
155 284
26 155 284
26 155 310 284
26 155 181 284 310
26 155 52 284 181 310
26 52 155 181 336 284 310
26 52 155 181 336 284 207 310
26 52 78 181 336 155 310 284 207
26 52 78 181 336 155 362 310 284 207
26 52 78 181 336 155 233 310 284 207 362
26 52 78 181 336 155 104 310 284 207 233 362
26 78 52 155 104 181 336 284 207 233 388 310 362
26 78 52 155 104 181 259 284 207 233 388 310 336 362
26 78 52 155 104 181 130 284 207 233 388 310 259 362 336
1 26 78 52 155 104 181 130 284 207 233 388 310 259 362 336
1 26 285 78 52 155 104 181 130 284 207 233 388 310 259 362 336
1 26 156 78 52 285 155 104 181 130 284 207 233 388 310 259 362 336
1 26 27 78 52 156 155 104 181 130 285 284 207 233 388 310 259 362 336
1 26 27 78 52 156 311 155 104 181 130 285 284 207 233 388 310 259 362 336

3.10 Binomialbäume

Wir gehen von binären Bäumen zu Mehrwegbäumen. Bei diesen hat jeder Knoten eine Liste von Kindern. (Die Listenlänge ist nicht immer Zwei. Ist sie Null, heißt der Knoten Blatt.)

```
data Multi a = Multi { tag :: a, sub :: [ Multi a ] } deriving Show    MultiType

instance Tree Multi where
  maybe_key n = Just (tag n); children n = sub n
  construct x cs = Multi { tag = the x, sub = cs }
  empty = error "there is no such thing as an empty binomial tree"
  is_empty t = False

instance Functor Multi where
  fmap f t = Multi { tag = f (tag t) , sub = map (fmap f) (sub t) }
```

Durch diese Instanzdeklaration gewinnen wir automatisch die Funktionen `rank`, `depth`, `size`, `subtrees`.

```
instance Show k => XForm (Multi k) where                                Multi
   xform = TreePaint.cmds . TreePaint.resize . MultiPaint.locate
```

Wir lassen nur *Binomialbäume* zu:

Definition: Ein Binomialbaum vom Rang d hat als Kinder Binomialbäume der Ränge $d-1, \dots, 0$ (in *dieser* Reihenfolge!)

```
is_binomial_tree_shaped t =                                             Bim
          [ rank c | c <- children t ] == [ rank t - 1, rank t - 2 .. 0 ]
  &&  and [ is_binomial_tree_shaped c | c <- children t ]
```

Übung: Dann gilt `depth t = rank t + 1`.

Diese Bäume sollen, wie üblich, heap-geordnet sein:

```
is_binomial_tree t = is_binomial_tree_shaped t && is_heap_ordered t    Bim
```

Beachte: `is_heap_ordered` ist polymorph und wird einfach nachgenutzt, weil `Bim` eine Instanz von `Tree` ist.

Übung: Zeige

```
is_binomial_tree_shaped t --> size t == 2 ** rank t                     none
```

In Binomial*bäumen* ist die Knotenanzahl also immer eine Zweierpotenz.

3.11 Binomialheaps

Um eine beliebig große Menge darzustellen, benutzen wir deswegen eine *Liste* von Binomialbäumen, und nennen diese Binomial*heap*.

```
type Bim a = Multi a
newtype BH a = BH [ Bim a ] deriving Show
instance Show k => XForm (BH k) where
    xform (BH ts) = Row (map xform ts) -- versuchsweise
```

Bim

Wir verlangen, daß die Ränge der Bäume in einem Binomialheap *streng* steigen.

Obacht! in jedem Baum: die Ränge der Kinder jedes Knotens *fallen*, in der Baumliste: die Ränge *steigen*.

Übung: Welche der folgenden Algorithmen werden dadurch vereinfacht? Welcher verkompliziert?

```
is_binomial_heap_shaped (BH ts) =
     and [ is_binomial_tree_shaped t | t <- ts ]
  &&  is_strictly_monotone  [ rank t | t <- ts ]
```

Bim

```
is_strictly_monotone :: Ord a => [a] -> Bool
is_strictly_monotone [] = True; is_strictly_monotone [x] = True
is_strictly_monotone (x : y : ys) = x < y && is_strictly_monotone (y : ys)
```

Assert

Wegen der Eindeutigkeit der Zerlegung einer Zahl in verschiedene Zweierpotenzen haben alle Binomialbäume mit gleicher Knotenzahl auch gleiche Form.

3.12 Operationen mit Binomialheaps

Das folgende ist einfach

```
instance PQ.PQ BH where
  empty = BH []; is_empty (BH ps) = null ps
  singleton x = BH [ unit x ]
     where
       unit :: a -> Bim a
       unit x = Multi { tag = x, sub = [] }
```

Bim

dabei baut `unit` einen Binomialbaum mit nur einem Knoten (= Blatt).

Das nächste schreiben wir erstmal auf, und füllen die Vorwärtsverweise später:

```
  extract (BH ps) = let (x, qs) = extract_min ps in (x, BH qs)
  union (BH ps) (BH qs) = BH (merge ps qs)
```

Bim

Dabei verschmilzt `merge` zwei (nach aufsteigendem Rang sortierte) Listen von Bäumen zu einer einzigen solchen:

```
merge :: Ord a => [ Bim a ] -> [ Bim a ] -> [ Bim a ]                Bim
merge ps [] = ps; merge [] qs = qs
merge (p : ps) (q : qs)
    | rank p < rank q  =  p : merge ps (q : qs)
    | rank p > rank q  =  q : merge (p : ps) qs
    | otherwise        =  merge [ link p q ] (merge ps qs)
```

Sind `p` und `q` gleichgroß, bauen wir daraus mit `link` einen neuen (doppelt so großen) Baum.

```
link :: Ord a => Multi a -> Multi a -> Multi a                       Multi
link a b = if tag a <= tag b
   then  Multi (tag a) (b : children a)  else  Multi (tag b) (a : children b)
```

Der minimale Schlüssel eines Binomialheaps steht in der Wurzel eines seiner Bestandteile. Diesen Teilbaum müssen wir erst finden.

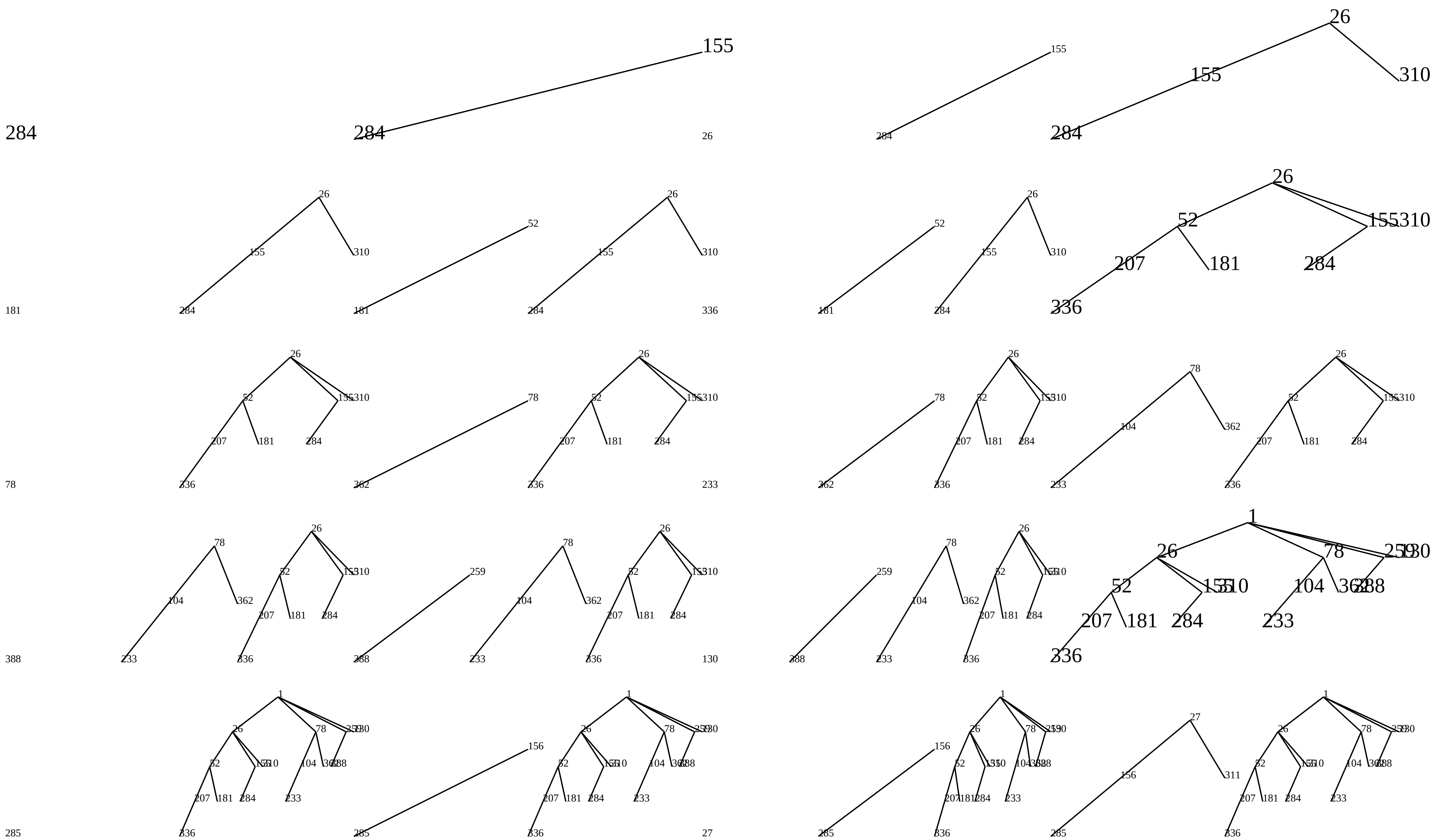
```
split_min :: Ord a => [ Multi a ] -> (Multi a, [ Multi a ])          Multi
split_min [] = error "empty heap"
split_min [ p ] = (p, [])
split_min (p : ps) =
    let (q, qs) = split_min ps
    in  if tag p < tag q then (p, ps) else (q, p : qs)
```

Wenn wir ihn haben, dann können wir seine Wurzel entfernen (denn sie ist ein minimales Element des Heaps). Danach müssen wir die Liste der Kinder dieser Wurzel mit der Liste der restlichen Bäume verschmelzen.

```
extract_min :: Ord a => [ Bim a ] -> (a, [ Bim a ])                  Bim
extract_min ps =
    let (q, qs) = split_min ps
    in  (tag q, merge (reverse (sub q)) qs)
```

Übung: Bestimmen Sie (eine obere Schranke für) die Laufzeit von `merge ps qs` abhängig von `length ps` und `length qs`. Messen Sie die Laufzeit durch die Anzahl der `(:)`-Operationen.

Auf der nächsten Seite sehen wir eine Folge von Einfügungen in einen Binomial–Heap.



284
284
155
26
284
155
284
155
26
310
181
284
155
26
310
181
52
284
155
26
310
336
181
52
284
155
26
310
336
207
52
181
284
155310
26

3.13 Zerstörender Zugriff: decrease und delete

Das `pool.zip`-Beispiel ist eine Anwendung, bei der aus einer PQ auch Elemente entfernt werden müssen.

Um das zu spezifizieren, müssen wir die rein funktionale (nebenwirkungsfreie) Welt verlassen. Warum?

Wenn ich ein Element aus einer PQ lösche, will ich es ja nicht erst suchen. Ich brauche einen Zeiger auf das Element. Den habe ich erhalten, als es eingefügt wurde. Dann müssen aber alle Operationen, die zwischen Einfügen und Löschen passieren, darauf achten, daß der Zeiger immer noch stimmt. Am besten, ich bekommen nicht den Zeiger, sondern einen Zeiger auf einen Zeiger, und dort trägt die Implementierung den jeweils richtigen Wert ein.

Ich brauche also explizites Hantieren mit Zeigern, und jede Operation hat Nebenwirkungen (andere Zeiger sind zu verbiegen). Das ist sehr unangenehm (zu spezifizieren und zu implementieren), aber scheint hier nicht anders zu gehen. (Forschungsaufgabe! Erfinde Funktionale Datenstrukturen, die das doch können.)

Im folgenden nehmen wir an, wir haben immer die passenden Zeiger bei der Hand.

Vor `delete` (Löschen eines Elements) implementieren wir `decrease_key` (Verringern eines Schlüsselwerts).

Ich schreibe als Typ `decrease_key :: PQ a -> (* a) -> a -> ()` und meine damit, daß ich eine PQ bekomme und einen Zeiger auf ein Element darin, und dieses durch ein neues mit kleinerem Schlüssel ersetzen soll. Das interessante Resultat ist natürlich nicht (), sondern die als Nebenwirkung geänderte PQ.

In heap-geordneten Bäumen läßt sich das aber leicht machen: das Element mit dem zu verringernden Schlüssel rutscht Richtung Wurzel. Ich bringe es an die passende Stelle durch Sortieren durch Einfügen auf dem Pfad von Element zu (seiner) Wurzel. Übung: die Heap-Ordnung geht nicht verloren, und die Form des Baums ändert sich überhaupt nicht. Das klappt also für leftist und binomial heaps wie von selbst.

Übung: Für leftist Heaps stimmt das nicht. Warum?

Im nächsten Abschnitt (Prims Algorithmus) sehen wir eine Anwendung von `decrease_key`.

Zunächst brauchen wir aber diese Operation nur als Hilfestellung für `delete`. Ich lösche ein Element so: zuerst verringere ich seinen Schlüssel auf $-\infty$ mit `decrease_key` (dadurch wird es zum Minimum), dann führe ich `extract` aus (dabei wird das Minimum gelöscht).

Übung: Wie lösche ich in einem binären Heap, der auf Grundkurs-Weise in einem Array steht? Geht das besser als mit der hier beschriebenen allgemeinen Methode? (Siehe `pool.zip`.)

3.14 Zeiger in Bäume

Wir bezeichnen einen Knoten (= Teilbaum) durch einen Pfad von der Wurzel. Ein Pfad ist eine Liste von Zahlen. Der Pfad `[0,2,1,0]` besagt: starte in der Wurzel, gehe zum 0. Kind (ganz links), dann zu dessen 2. (d. h. das dritte), usw.

Tree
```
type Pos = [ Int ]
```

Wir können auf eine Position zugreifen:

Tree
```
peek :: Tree t => t a -> Pos -> t a
peek t [] = t
peek t (x : xs) = peek (children t !! x) xs
```

und auch an eine Position schreiben:

Tree
```
poke :: Tree t => t a -> Pos -> t a -> t a
poke t [] s = s
poke t (x : xs) s =
    let (pre, p : post) = splitAt x (children t)
        p' = poke p xs s
    in  construct (maybe_key t) (pre ++ p' : post)

poke_key :: Tree t => t a -> Pos -> Maybe a -> t a
poke_key t p x =
    let cs = children (peek t p)
    in  poke t p (construct x cs)
```

Die Liste aller Teilbaumpositionen ist:

Tree
```
pos :: Tree t => t a -> [ Pos ]
pos t = [] : [ x : xs | (x, c) <- zip [0..] (children t) , xs <- pos c ]
```

Die Funktionen `peek t p` und `poke t p s` sind natürlich nur für `p` $\in$ `pos t` definiert.

Der Vorgänger (Vater) einer Position ist

Tree
```
parent :: Pos -> Pos
parent [] = error "empty pos has no parent"
parent xs = take (length xs - 1) xs
```

Wie gesagt, das sind alles funktionale Umschreibungen. In einer imperativen Sprache mit Zeigern sollten alles diese Operationen in konstanter Zeit ausführbar sein. Zum schnellen Auffinden von `parent p` muß jeder Baumknoten einen Zeiger auf seinen Vorgänger enthalten.

3.14.1 Schlüsselverringern in Binomialbäumen

Einen Teilbaum (= Knoten) in einem Wald bezeichnen wir durch seine Position, das heißt jetzt die Position des Baums in der Liste und die Position des Knotens im Baum. Damit wir das einfacher schreiben können, erzeugen wir aus einem Wald einen Baum, in dem wir die Wurzelliste einfach als Kinderliste einer Über-Wurzel betrachten.

```
lift :: [ Multi a ] -> Multi a
lift ts = Multi { tag = undefined, sub = ts }

unlift :: Multi a -> [ Multi a ]
unlift t = sub t

is_root, is_proper :: Pos -> Bool
is_root   p = length p == 1
is_proper p = length p >  1
```

Multi

Die Funktion `decrease_key` sieht dann im Allgemeinen so aus:

```
decrease_key_scheme :: (Tree t, Ord a)
   => (t a -> Pos -> t a) ->  t a -> Pos -> a -> t a
```

Tree

Wir wollen in einem Baum `f` den Schlüssel an der Position `p` durch einen neuen Schlüssel `x` ersetzen.

```
decrease_key_scheme rework f p x =
    let t = peek f p
    in  if x > key t
        then error "you do not decrease the key"
        else rework (poke_key f p (Just x)) p
```

Tree

Die Funktion `rebuild` erledigt das für Binomialheaps so wie eben erklärt:

```
rebuild t p =
    if is_proper p
    then let q = parent p
         in  if key (peek t q) <= key (peek t p)
             then t
             else let t'  = poke_key t  p (maybe_key (peek t q))
                      t'' = poke_key t' q (maybe_key (peek t p))
                  in rebuild t'' q
    else t
decrease_key :: Ord a =>  BH a -> Pos -> a -> BH a
decrease_key (BH ts) p x =
   BH ( unlift ( decrease_key_scheme rebuild (lift ts) p x ) )
```

Bim

Kapitel 4

Minimum Spanning Trees

4.1 Einführung

(Die Darstellung folgt Kapitel 24 von [CLR91]. Siehe auch [Knu93].)

Zu einem ungerichteten, zusammenhängenden Graphen $G = (V, E)$ mit einer Kantengewichtsfunktion $w : E \to \mathbb{R}^+$ suchen wir einen Teilgraphen $T = (V, E')$ mit $E' \subseteq E$, der ein Baum ist, und für den die Gewichtsfunktion

$$w(T) = \sum \{w(e') \mid e' \in E'\}$$

einen möglichst geringen Wert annimmt.

Anschaulich: eine Telefongesellschaft will alle Punkte aus V verbinden und dabei sowenig wie möglich Kabel verlegen. Alle Wege $\{u, v\}$, in die man ein Kabel legen könnte, bilden den Graphen E. Für diesen gibt die Gewichtsfunktion an, wie teuer das Kabellegen wäre. Der Kabelgraph T muß also ein Baum sein, der alle Punkte aus V erreicht. (Wäre es kein Baum, so enthielte er einen Kreis, aus dem man eine Kante weglassen kann, wodurch das Gewicht fällt.)

So ein Baum heißt MST (minimum spanning tree). MST-Algorithmen sind ein wichtiges Anwendungsfeld für Priority-Queues. Binomial-Heaps wurden ursprünglich genau dafür erfunden. Noch effizientere Algorithmen erhält man, wenn man nicht die tatsächlichen, sondern die amortisierte Laufzeiten von PQ-Operationen optimiert. Deswegen ist dieses Kapitel eine Überleitung zum nächsten (amortisierte Analyse), in dem wir schließlich von Binomial- zu Fibonacci-Heaps übergehen.

4.2 Grundlagen

(Diesen Abschnitt lese ich nicht vor. Wer den Beweis zu Prims Algorithmus sehen will, sollte sich nächstes Semester die Vorlesung Graphalgorithmen von Herrn Dr. Apel anhören)

Um einen MST T zu finden, brauchen wir die Kantenmenge E' von T. Die stellen wir her, indem wir schrittweise Kanten zu einer (anfangs leeren) Menge A hinzunehmen. Dabei soll immer gelten, daß $A \subseteq E'$ für irgendeinen MST $T = (V, E')$ von G. Wir dürfen also nur sichere Kanten hinzufügen:

Definitionen:

- eine Kante $\{u, v\} \notin A$ heißt *sicher* für A, wenn $A \cup \{u, v\}$ auch Teilkantenmenge eines MST von G ist.
- Ein Schnitt $(S, V \setminus S)$ eines Graphen $G = (V, E)$ ist eine Zerlegung von G.
- Eine Kante $\{u, v\} \in E$ *kreuzt* den Schnitt (S, S'), wenn $u \in S$ und $v \notin S$.
- Ein Schnitt (S, S') *beachtet* eine Kantenmenge $A \subseteq E$, wenn keine Kante aus A den Schnitt kreuzt.
- Eine *leichte* Schnittkante in einem Schnitt (S, S') ist jede Kante, die (S, S') kreuzt und unter allen diesen minimales Gewicht hat.

Satz: Für jede Kantenmenge A, die in einem MST von G enthalten ist, und jeden Schnitt $(S, V \setminus S)$, der A beachtet, gilt: jede leichte Schnittkante $e = \{u, v\}$ ist sicher für A.

Beweis: Sei $T = (V, E')$ der Ober-MST für A.

Weil T ein MST ist, enthält er eine Kante $f = \{x, y\}$, die $(S, V \setminus S)$ kreuzt. (Wenn nicht, dann würde T nicht ganz V verbinden.)

Wir bilden die Kantenmenge $E'' = E' \setminus f \cup e$. Das ist wieder ein spanning tree T'' für G. Dessen Gewicht ist kleinergleich dem von T.

Daraus folgt, daß $w(T'') = w(T)$, weil ja T selbst ein MST war. Wir haben also einen MST gefunden (nämlich T''), der $A \cup e$ enthält. Deswegen ist e eine sichere Kante für A.

4.3 Algorithmus von Prim

Wir gehen von der schon besprochene Idee aus, sichere Kanten hinzuzufügen:

```
A := emptyset
while A ist kein MST do
  e := eine sichere Kante für A
  A := A + e
```

none

Das ist korrekt, weil A durch sichere Kanten nur zulässig vergrößert wird, und das terminiert, weil es insgesamt höchstens `E` Kanten gibt.

Wir verlangen zusätzlich, daß die Kanten in A immer eine Baum bilden. Dessen Knotenmenge nennen wir V'. Natürlich ist $V' \subseteq V$.

Wir vergrößern V' dadurch, daß wir einen Knoten hinzunehmen. Und zwar nehmen wir immer eine leichte Schnittkante für $(V', V \setminus V')$.

Dazu merken wir uns für alle $u \in V \setminus V'$ das geringste Gewicht einer Kante zu einem $v \in V'$, d. h. einer leichten Kante über den Schnitt $(V', V \setminus V)$.

Wir benutzen eine Priority-Queue: Die Elemente sind Knoten in V, die Schlüssel sind diese geringsten Gewichte.

Wenn wir V' um v vergrößern, können Knoten in $V \setminus (V' \cup v)$ näher an V heranrutschen. Welche sind das? Die befinden sich jedenfalls unter den Nachbarn von v.

```
V' := emptySet
Q := build (V) -- PQ aus allen Knoten,
               -- alle Schlüsselwerte sind +infty
while V' /= V do
    (v, Q) := extract (Q); V' := V' + v
    for u in Nachbarn (v) do
        if  (u not in V')  and  (w (u, v) < key (u))
            then decrease_key (Q, u, w (u, v))
                 parent [u] = v;
        end if
    end for
end while
```

none

Als Invarianten haben wir

1. für alle v, für die `parent (v)` existiert: `key(v) = w (v, parent (v))`
2. wenn v ausgewählt wird: `{v, parent(v)}` ist eine leichte Kante bezüglich des Schnitts $(V', V \setminus V')$ (deswegen ist sie sicher)

4.4 Laufzeit von Prims Algorithmus

Die Eingabe ist $G = (V, E)$ mit $\mathtt{V} = n, \mathtt{E} = m$. Es gilt $m = O(n^2)$ und $n - 1 \leq m$ (Zusammenhang!)

Wir bauen einmal eine PQ aus n Elementen.

Wir extrahieren n-mal das Minimum.

Wir führen m-mal (nämlich für jede Kante genau einmal) `decrease-key` aus.

Mit einer vernünftigen Implementierung (z. B. Binomial-Heaps) erreichen wir eine Laufzeit von $n + (n \log n) + (m \log n) = O(m \log n)$.

Der bestimmende Anteil ist m mal $\log n$ für `decrease_key`. Das ist der Ansatz für Fibonacci-Heaps: dort schaffen wir `decrease_key` in $O(1)$, während `extract` mit (im wesentlichen) $O(\log n)$ bleibt.

Damit erreichen wir insgesamt $n + (n \log n) + m\,O(1) = O(m + n \log n)$. Das ist eine Verbesserung, wenn $m \gg n$, also der Graph recht dicht ist.

Der Zusatz (im wesentlichen) versteckt aber eine sehr wichtige Überlegung: Wir brauchen ja gar nicht jedes *einzelne* `extract` in $O(\log n)$, es reicht ja aus, wenn die Gesamtzeit *aller* n `extract` in $O(n \log n)$ geht. Das führt zu amortisierter Analyse.

Kapitel 5

Amortisierte Analyse

5.1 Einführung

Wir stellen uns eine Binärzahl der Länge n vor und inkrementieren diese. Das kostet schlimmstenfalls n Operationen, (wenn der Übertrag bis vorn durchläuft). (Operation = betrachten oder Ändern einer Binärstelle).

Nun inkrementieren wir das Resultat, usw., insgesamt m mal. (Der Einfachheit halber nehmen wir an, daß $m < 2^n$, so daß das Resultat höchstens um ein Bit länger ist als die Eingabe).

Die Gesamtkosten sind sicher $\leq n \times m$, aber erreichen wir diese Schranke wirklich? Ein Beispiel:

Zahl	Kosten
100101	
cc	2
100110	
c	1
100111	
cccc	4
101000	
c	1
101001	
cc	2
101010	
c	1
101011	
ccc	3
101100	
	14

Wir haben 7 Operationen ausgeführt, die maximalen Kosten einer Operation waren 4, aber die Summe der Kosten ist nur 14. Wir stellen fest, daß zwar teuer Operationen auftreten, aber diese immer von einigen billigen gefolgt werden.

Dies ist bei manchen Datenstrukturen der Fall: es gelingt uns nicht, teure Operationen auszuschließen, aber wir könne zeigen, daß diese so selten sind, daß die Summe über die Laufzeit mehrerer Operationen vernünftig beschränkt ist.

Solche Untersuchungen nennen wir *amortisierte Analyse*.

Definition: In einer Datenstruktur mit den Operationen $F = \{f, \ldots\}$ hat die Operation $f \in F$ die amortisierten Kosten c_f, wenn gilt: für alle Folgen von Operationen

$$S_0 \xrightarrow{f_1} S_1 \xrightarrow{f_2} \ldots \xrightarrow{f_n} S_m$$

sind die wirklichen Kosten beschränkt durch $O(c_{f_1} + c_{f_2} + \ldots + c_{f_m})$.

Wiederholung: das $O(\cdot)$ ist eine asymptotische Schranke, d. h. $g \in O(h)$, wenn $\exists a, b, c : \forall x \geq a : f(x) \leq b + cg(x)$. In diesem Fall heißt $x \geq a$: Die Schranke gilt für alle Operationsfolgen, die genügend lang sind.

In obigem Beispiel liegen die tatsächlichen Kosten eines Inkrementierens zwischen 1 und n, aber die amortisierten Kosten sind $O(1)$ (also konstant):

Wir können leicht nachrechnen, daß beim Hochzählen von 2^{n-1} auf 2^n die Anzahl der X in obigem Schema gleich 2^n ist.

5.2 Amortisation durch Kontoführung

Wir können amortisierte Kostenaussagen so beweisen: wir stellen uns neben der Datenstruktur ein Girokonto vor.

Immer, wenn wir eine Operation f ausführen wollen, zahlen wir Geld c_f auf das Konto ein.

Ein Dämon sieht vor jeder Operation nach, ob genügend Geld vorhanden ist, um die Operation f zu bezahlen, und führt diese dann aus.

Unsere Aufgabe ist es nachzuweisen, daß die c_f so gewählt sind, daß wir das Konto nie überziehen.

Das Absolutglied in der Definition des $O(\cdot)$ gestattet uns anzunehmen, daß anfangs schon $O(1)$ Geld auf dem Konto ist.

Wir wählen 2 als amortisierte Kosten eines Inkrementierens:

Zahl	tatsächliche Kosten	Einzahlung	Kontostand
			3
100101		2	5
cc	2		3
100110		2	5
c	1		4
100111		2	6
cccc	4		2
101000		2	4
c	1		3
101001		2	5
cc	2		3
101010		2	5
c	1		4
101011		2	6
ccc	3		3
101100			
	14	14	3

Das Konto war immer im Haben, und ist insgesamt ausgeglichen.

5.3 Amortisation durch Potential

Es ist umständlich, auf die Datenstruktur *und* auf das Konto aufzupassen. In vielen Fällen ist folgende Vereinfachung möglich: wir wählen die Kosten c_f so, daß der Kontostand direkt aus der Datenstruktur ablesbar ist.

Betrachte die vorige Tabelle: der Kontostand ist imme genau der Anzahl der 1 in der Binärzahl (Beispiel: anfangs: 100101, Kontostand 3).

Wir nennen das dann nicht mehr Konto, sondern *Potential*.

Für die Potentialfunktion $S \to \mathbb{R}$ (ordnet jeder Struktur eine Zahl zu) haben wir zu zeigen:

1. $\forall S : \mathsf{pot}(S) \geq 0$
2. $\forall S, S', f$: wenn $S \xrightarrow{f} S'$, dann sind die tatsächlichen Kosten dieser Operation nach oben beschränkt durch $c_f + \mathsf{pot}(S) - \mathsf{pot}(S')$ (d. h. amortisierte Kosten plus Potentialunterschied)

Anwendung: wir haben schon gesehen, daß Binomialheaps viel mit Binärzahlen zu tun haben. Das Einfügen eines neuen Elements in einen Binomialheap entspricht dem Inkrementieren einer Binärzahl.

Jetzt wissen wir, daß Inkrementieren amortisiert $O(1)$ kostet, also kostet auch Einfügen in Binomialheaps amortisiert $O(1)$. Zur Erinnerung: wirklich kostet es $O(\log n)$.

Aber Vorsicht: gilt das amortisierte $O(1)$ auch für beliebige Folgen von Einfügen und Minimum-Löschen? Nein. Wenn wir abwechselnd inkrementieren und dekrementieren, können wir gewaltig Pech haben: $1111 \to 10000 \to 1111 \to 10000 \to \ldots$.

Wir lernen daraus, daß man beim Amortisieren nie eine Operation allein betrachten kann, sondern immer für einen ganzen Satz von Operationen (nämlich alle, die in den betrachteten Operationsfolgen vorkommen dürfen) gleichzeit Kostenaussagen treffen muß.

Ein andere wichtiger Punkt: die tatsächlichen Kosten einer Operation können wir *messen*. Amortisierte Kosten *nicht*! Die können wir nur *passend festlegen*, und dann prüfen, daß das Konto immer im Haben steht. Vergleichen Sie das mit einem Restaurant-Chef. Die Kosten einer Mahlzeit kann er nicht messen, sondern nur festlegen, wobei er sich überlegen muß, was kostet der Koch, was die Saalmiete, was das Material, etc. Insbesondere kann er sagen: machen wir das Gericht billiger als jenes, und dieses teurer als ein anderes. Dadurch wird nichts falsch, es muß nur zum Schluß gutgehen. Tatsächlich gibt es manchmal verschiedene Möglichkeiten, für ein und dieselbe Datenstruktur amortisierte Kosten festzulegen. (Bestimmen wir die amortisierten Kosten durch ein Potential, haben können wir die Potentialfunktion wählen.)

5.4 Skew Heaps

Skew Heaps sind eine ganz erstaunlich einfache Implementierung von PQs: sie sind noch einfacher als Leftist Heaps.

Wir machen alles ganz genauso wie dort,

Skew

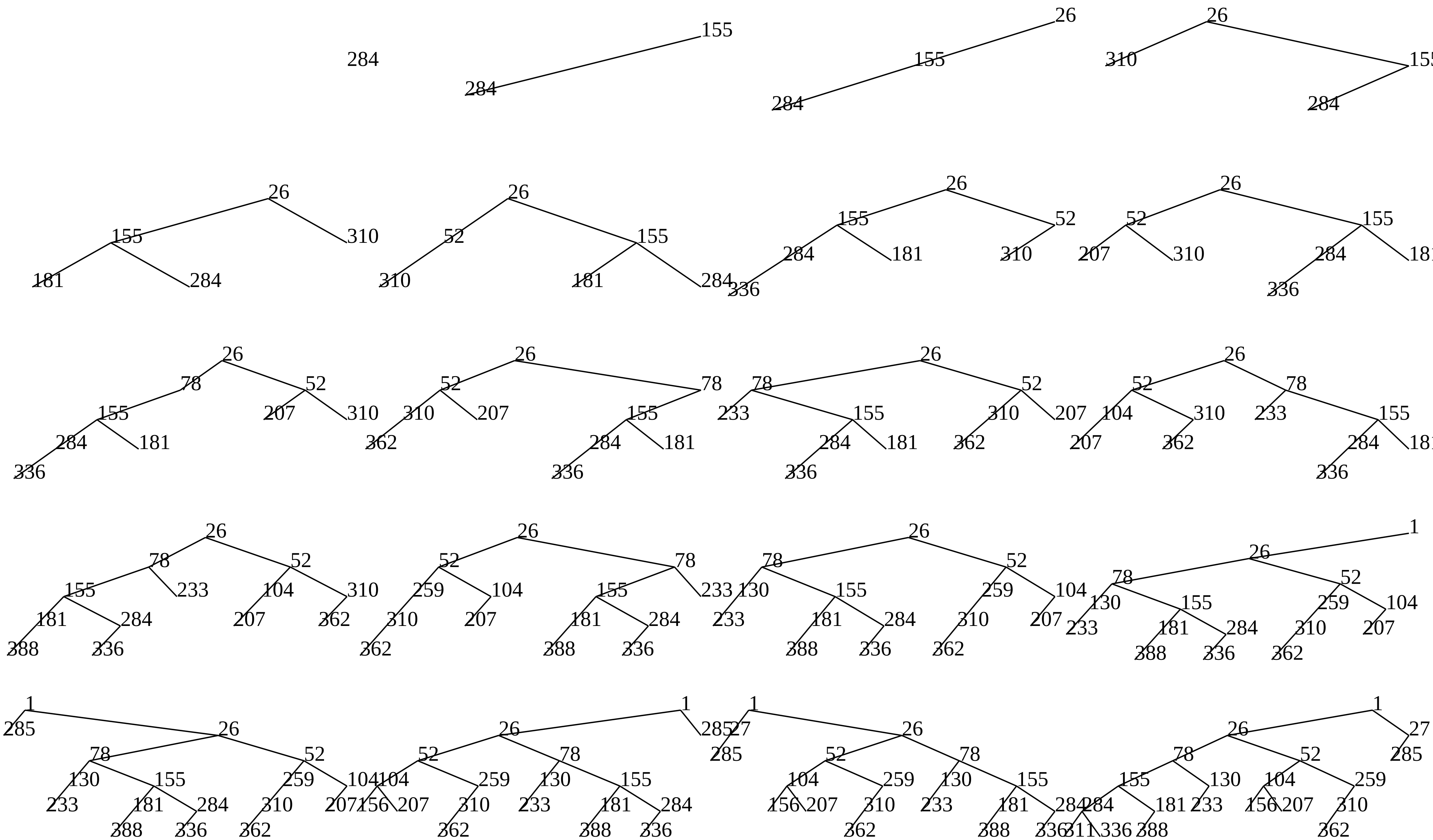
```
newtype  Skew k = Skew (Bin k) deriving Show
instance Show k => XForm (Skew k) where xform (Skew t) = xform t

instance PQ.PQ Skew where
  empty = Skew Leaf; is_empty (Skew Leaf) = True; is_empty _ = False
  singleton x = Skew ( node x empty empty )

  extract (Skew Leaf) = error "cannot extract anything from empty heap"
  extract (Skew n)    = (key n, Skew ( merge (left n) (right n) ) )
```

aber vergessen in `union` völlig, `rdepth (left t)` und `rdepth (right t)` zu vergleichen. Das führt natürlich zu grob unbalancierten Bäumen. Deswegen vertauschen wir blindlings *alle* rechten und linken Teilbäume auf dem Pfad, auf dem wir gerade arbeiten.

Skew

```
  union (Skew x) (Skew y) = Skew (merge x y)

merge Leaf y = y; merge x Leaf = x
merge x y =
   let (m, l, r) = if key x <= key y
                   then (key x, left x, merge (right x) y)
                   else (key y, left y, merge x (right y))
   in  node m r l
```

Auf der nächsten Seite sehen wir eine Folge von Einfügungen in einen Skew–Heap.


284
155 284
26 155 284
26 310 155 284
26 155 310 181 284
26 52 155 310 181 284
26 155 52 284 181 310 336
26 52 155 207 310 284 181 336
26 78 52 155 207 310 284 181 336
26 52 78 310 207 155 362 284 181 336
26 78 52 233 155 310 207 284 181 362 336
26 52 78 104 310 233 155 207 362 284 181 336
26 78 52 155 233 104 310 181 284 207 362 388 336
26 52 78 259 104 155 233 310 207 181 284 362 388 336
26 78 52 130 155 259 104 233 181 284 310 207 388 336 362
1 26 78 52 130 155 259 104 233 181 284 310 207 388 336 362
1 285 26 78 52 130 155 259 104 233 181 284 310 207 388 336 362
1 26 285 52 78 104 259 130 155 156 207 310 233 181 284 362 388 336
1 27 285 26 52 78 104 259 130 155 156 207 310 233 181 284 362 388 336
1 26 27 78 52 285 155 130 104 259 284 181 233 156 207 310 311 336 388 362

Dabei kann es schon passieren, daß wir Bäume mit langem rechten Rückgrat erhalten, aber das ist selten genug der Fall: Die amortisierte Laufzeit von `merge` ist $O(\log n)$.

Wir zeigen das durch ein geschickt gewähltes Potential. (Die Darstellung folgt Okasakis Skript, siehe seine Web-Seite. Vergleiche auch Kapitel 6.7 und 11.3 in [Wei93]).

Wir nennen einen Teilbaum *gut*, wenn sein linkes Kind nicht kleiner als das rechte ist:

```
good, bad :: Bin a -> Bool                                        Skew
good Leaf = True; good t = size (left t) >= size (right t)
bad t = not (good t)
```

Das *Potential* eines Baumes ist die Anzahl seiner schlechten Teilbäume.

```
pot :: Bin a -> Int                                               Skew
pot t = length [ s | s <- subtrees t, bad s ]
```

Übung: Welche Beziehung besteht zwischen `is_leftist_shaped t` und `pot t == 0`?

Übung: Auf dem rechten Rückgrat von `t` liegen höchstens `log (size t)` gute Knoten.

Wir haben nachzuweisen: Das Potentialgefälle reicht immer aus, um `union` zu bezahlen.

Das heißt: wenn `union x y` sehr lange dauert, weil die rechten Rückgrate lang sind, dann sind `pot x + pot y` groß genug, und `pot (union x y)` klein genug, so daß die Operation aus der Potentialdifferenz bezahlt werden kann.

Als ein Maß für die tatsächlichen Kosten von `union x y` nehmen wir die Anzahl der ausgeführten Schlüssel-Vergleiche für Teilbäume:

```
cost Leaf y = 0; cost x Leaf = 0                                    Skew
cost x y =
   let c = if tag x <= tag y
           then cost (right x) y else cost x (right y)
   in  1 + c
```

(das ist auch die Anzahl der Konstruktor-Aufrufe). Wir zeigen

```
cost x y                                                            none
  <= (pot x + pot y) - pot (union x y) + 2 * (log (size x) + log (size y))
```

Zum Beweis schreiben wir die Behauptung als

```
cost x y + pot (union x y)                                          none
  <= pot x + 2 * log (size x) + pot y + 2 * log (size y)
```

Induktionsanfang: `x = Leaf`. Dann ist `cost x y = 0`, andererseits `union x y = y`, also `pot (union x y) = pot y`.

(Symmetrisch dazu ist der Fall `y = Leaf`.)

Induktionsschritt: Weder `x` noch `y` ist `Leaf`. O.B.d.A. sein `tag x <= tag y`. Wir bezeichnen `l = left x, r = right x`.

```
cost x y = 1 + cost r y                                             none
union x y = Node (tag x) (union r y) l
```

Wir unterscheiden:

1. `good x`.

Es ist `pot (union x y) <= 1 + pot (union r y) + pot l` (wir wissen nicht, ob `union x y` gut oder schlecht ist).

Deswegen gilt nach Induktionsannahme

none

```
cost x y + pot (union x y)
<= 1 + cost r y + 1 + pot (union r y) + pot l
<= 1 + 1 + pot l + pot r + 2 * log (size r) + pot y + 2 * log (size y)
```

Wir haben `size l >= size r`, und wegen `size x = 1 + size l + size r` folgt `size x > 2 * size r`, also `log (size x) > 1 + log (size r)` und `2 * log (size x) > 2 + 2 * log (size r)`

Weiter ist `pot x = pot l + pot r` (`x` selbst ist `good`). Wir setzten deswegen die Ungleichung fort zu

```
cost x y + pot (union x y)                                        none
< pot x + 2 * log (size x) + pot y + 2 * log (size y)
```

2. `bad x`. Dann gilt `size l < size r`, und wegen `size r <= size (union r y)` gilt `good (Node (tag x) (union r y) l)`, also `good (union x y)`. Demnach ist `pot (union x y) = pot l + pot (union r y)`.

Daraus folgt nach Induktionsannahme

```
cost x y + pot (union x y)                                        none
<= 1 + cost r y + pot l + pot (union r y)
<= 1 + pot l + pot r + 2 * log (size r) + pot y + 2 * log (size y)
```

Es galt `pot x = 1 + pot l + pot r`, also

```
cost x y + pot (union x y)                                        none
<= pot x + 2 * log (size r) + pot y + 2 * log (size y)
```

Es ist sicher `size r < size x`, und daraus folgt die Behauptung.

Zusammenfassung: Warum funktioniert der Beweis?

- Ist `good x`, dann liegt es am Logarithmus (`size x > 2 * size r`)
- Ist `bad x`, dann liegt es am Potential (`pot x = 1 + ...`)

5.5 Fibonacci-Heaps

5.5.1 Einführung

(Literatur: Kurz, aber präzise: [Knu93]. Übungsaufgabe: das nachlesen. — Lang, aber umständlich: [CLR91]. Nicht nur umständlich, sondern eigentlich sogar falschrum (IMHO): die präsentieren erst die Algorithmen, und danach die Definition.)

Zur Erinnerung: Binomialheaps (-wälder) sind schon ganz schön, aber wir wollen für die MST-Anwendung das `decrease_key` noch schneller machen, am liebsten $O(1)$. Es reicht uns, wenn die Zeitschranken amortisiert sind.

Das `decrease_key` dauert $\log n$, weil die Elemente im Baum nach oben (näher zur Wurzel) rutschen, und die Pfade sind schlimmstenfalls $\log n$ lang. Wenn wir konstante Laufzeit haben wollen, dann dürfen wir den Pfad gar nicht betrachten. Stattdessen schneiden wir den Teilbaum mit dem zu verringernden Element einfach ab und stellen ihn mit in die Wurzelliste.

Dadurch verliert der Vater t des abgeschnittenen Teilbaums ein Kind. Das notieren wir: für ihn gilt jetzt `defect t = 1` (war vorher `0`). Da trotzdem alles noch wie im Binomialheap aussehen soll, müssen wir statt `rank t` die Funktion `rank t + defect t` betrachten, die wir `defect_rank t` nennen.

Betrachten wir aber die Brüder des abgeschnittenen Baums. Die sind immer noch Kinder von t, das heißt aber, daß in der Liste ihrer (Defekt–)Ränge jetzt einer fehlt (nämlich der abgeschnittene). Deswegen kommen dort nicht mehr alle Defekt-Ränge von $0 \ldots$ `rank t - 1` vor. Wir müssen die Bedingung an die Ränge der Kinder also abschwächen, damit die Idee des Abschneidens implementierbar wird.

Wohin mit den abgeschnittenen Teilbäumen? Sie landen in der Wurzelliste. Die wird dadurch immer länger. In Binomialbäumen hatte sie die Länge $O(\log n)$. Wir müssen von Zeit zu Zeit aufräumen, um die Länge wieder auf (etwa) diesen Wert zu reduzieren.

Das Aufräumen bezahlen wir aus dem angesammelten Potential. Als Potentialfunktion nehmen wir die Länge der Wurzelliste plus zweimal die Summe aller Defekt.

Damit erreichen wir folgende amortisierte Laufzeiten: `insert, union, decrease_key` in $O(1)$, `extract` in $O(\log n)$.

Bemerkung: geht `insert` und `extract` beides in amortisiert $O(1)$? Wie schnell könnten wir dann sortieren?

5.5.2 Definition

Zur Wiederholung: Ein Baum ist ein Binomialbaum, wenn in jedem Knoten die Ränge passen:

```
is_binomial_shaped t <-->                                          none
  and [ [ rank c | c <- children s ] == [ rank s - 1 , rank s - 2 .. 0 ]
      | s <- subtrees t ]
```

Das schreiben wir nochmal anders hin:

```
is_binomial_shaped t <-->                                          none
  and [ let rs = reverse [ rank c | c <- children s ]
        in  and [ rs !! k == k | k <- [ 0 .. rank s - 1 ] ]
      | s <- subtrees t ]
```

(Die Listenindizes beginnen bei `0`.)

Jetzt stellen wir uns vor, daß hier jeder Knoten Kinder verlieren darf. Die Anzahl der verlorenen Kinder heißt Defekt.

Definition: Ein Fibonacci-Baum (F-Baum) ist ein Mehrwegbaum, der in jedem Knoten noch eine Markierung (= eine Zahl) trägt. Diese heißt *Defekt* des Knotens.

```
type Fib a = Multi (a, Int)                                        Fibo
defect        :: Fib a -> Int
defect      t =  d  where  (x, d) = tag t
defect_rank   :: Fib a -> Int
defect_rank t =  defect t + rank t
```

Ein F-Baum ist heap–geordnet und erfüllt diese Rang-Bedingung:

```
is_fib_tree_shaped t =                                             Fibo
  and [ 0 <= defect s && defect s <= 1 | s <- subtrees t ]
  &&
  and [ let rs = sort [ defect_rank c | c <- children s ]
        in  and [ rs !! k >= k | k <- [ 0 .. rank s - 1 ] ]
      | s <- subtrees t ]
is_fib_tree t = is_fib_tree_shaped t && is_heap_ordered t
```

Definition: Ein Fibonacci-Heap (F-Heap, F-Wald) ist eine Liste von F-Bäumen.

```
data FH a = FH [ Fib a ]                                           Fibo
is_fib_heap (FH ts) = and [ is_fib_tree t  | t <- ts ]
```

5.5.3 Folgerungen aus der Definition

Wir benötigen die Fibonacci-Folge

Fibo
```
fib :: Int -> Int
fib 0 = 0; fib 1 = 1; fib n = fib (n-2) + fib (n-1)
```

Der Name Fibonacci-Baum stammt von einem Zusammenhang zwischen Rang und Größe (ähnlich war das bei Binomialbäumen):

none
```
is_fib_tree t --> fib (2 + rank t) <= size t
```

Beweis (Induktion nach `size t`, Induktionsschritt): Die nach aufsteigendem Defekt–Rang geordnete Liste der Kinder des F-Heaps t ist $[t_0, \dots, t_{n-1}]$ mit `n = rank t`. Es gilt $\forall k : k \leq$ `defect_rank` t_k, also $k - 1 \leq$ `rank` t_k.

Nach Induktionsannahme gilt $\forall k :$ `fib (2 + rank t_k) <= size t_k`, also `fib (1+k) <= size t_k`. Wir summieren das über alle k und addieren 1:

none
```
1 + sum [ fib (1+k) | k <- [0 .. n-1] ]
     <= 1 + sum [ size t_k | k <- [ 0 .. n-1 ]]
```

Auf der linken Seite steht genau `fib (n+2) = fib (rank t + 2)`, und rechts steht `size t`.

Folgerung: Wir wissen, daß `log (fib n)`$= \Theta(n)$ ist, also folgt `rank t = O(log (size t))`. Wir haben also eine obere Schranke für den Rang eines F-Baums.

Bemerkung 1) Natürlich ist die Implementierung von `fib` ineffizient, aber darauf kommt es hier nicht an. — Eine bessere Lösung (läuft in Linearzeit) ist

Fibo
```
fib' :: Int -> Int
fib' n = fibs !! n
  where fibs = 0 : 1 : map (uncurry (+)) (zip fibs (tail fibs))
```

Bemerkung 2) Warum wählen wir `fib 0 = 0` (und fangen nicht bei 1 an)? Dann gilt `gcd n m = gcd (fib n) (fib m)`.

Bemerkung 3) Wir schreiben F_k für `fib k`, sowie $\phi = \frac{\sqrt{5}+1}{2}$. Zeige

$$F_k = \frac{\phi^k + \phi^{-k}}{\sqrt{5}}$$

Wegen $0 < \phi^{-1} < 1$ folgt $F_k = \Theta(\phi^k)$.

5.5.4 Potentialfunktion für F-Heaps

Fibo

```
pot :: FH a -> Int
pot (FH ts) =
    length ts + 2 * sum [ defect s | t <- ts, s <- subtrees t ]
```

Die Bestandteile der Potentialfunktion wurden schon erklärt; warum nehmen wir die Defekte doppelt? Wenn wir einem defekten Knoten ein Kind abschneiden, wandert er selbst in die Wurzelliste. Das heißt: das Abschneiden kostet wirkliche Zeit, und das Verlängern der Wurzelliste kostet Potential.

5.5.5 Implementierung

Das ist klar (siehe Binomialheaps):

Fibo

```
instance PQ FH where
    empty = FH []; is_empty (FH ts) = null ts
    singleton x = FH [ unit x ]
        where unit :: a -> Fib a
              unit x = Multi { tag = (x, 0), sub = [] }
```

Zum Extrahieren des Minimums suchen wir den Baum mit minimaler Wurzel (`split_min` ist polymorph für `Multi a`). Die Kinderliste seiner Wurzel hängen wir vor die Liste der restlichen Bäume, und konsolidieren diese neue Wurzelliste.

Fibo

```
    extract (FH ps) =
        let (q, qs) = split_min ps
            (x, d)  = tag q
        in  (x, consolidate ( FH ( children q ++ qs )))
```

Beim Verschmelzen wird aber *nicht* konsolidiert! Wir benutzen es ja als Unterprogramm von `insert`, und das soll in $O(1)$ laufen.

Fibo

```
    union (FH xs) (FH ys) = FH (xs ++ ys)
```

5.5.6 Aufräumen in F-Wäldern

Wir wollen aus einer beliebigen Listen von F-Bäumen eine recht kurze herstellen. Die Idee ist die gleiche wie bei Binomialbäumen: wir verbinden jeweils zwei Bäume mit gleichem Rang zu einem einzigen. Das tun wir so oft, bis alle Ränge verschieden sind.

Wir möchten das in Linearzeit (gemessen an der Länge der Eingabeliste) schaffen. Das heißt, wir haben keine Zeit, die Bäume nach Rängen zu sortieren. bzw. wir sortieren schon, aber durch Bucketsort. Es gibt nämlich nicht allzuviele mögliche Ränge. (Warum?) Wir benutzen ein Feld, dessen Indexbereich die Ränge sind.

```
type Field a =  Array Int (Maybe (Fib a))                          Fibo
blank :: Field a
blank =  array (0, maxrank) [ (k, Nothing) | k <- [0 .. maxrank ] ]
    where maxrank = 46
```

Wir tragen die Bäume der Reihe nach ein. Immer, wenn wir eine Zelle (= einen Rang) schon besetzt finden, führen wir eine `link`-Operation aus. Schließlich geben wir alle besetzten Feld-Elemente wieder aus.

```
consolidate :: Ord a => FH a -> FH a                               Fibo
consolidate (FH ts) = FH ( out ( place ts blank ) )

out :: Field a -> [ Fib a ]
out a = [ t | Just t <- elems a ]

place :: Ord a => [ Fib a ] -> Field a -> Field a
place [] f = f
place (t : ts) f =
    case (f ! rank t) of
        Nothing -> place ts              (f // [ (rank t, Just t ) ])
        Just s  -> place (link t s : ts) (f // [ (rank t, Nothing) ])
```

Wir haben noch zu zeigen, daß bei `link t s` wirklich ein F-Baum entsteht. (Übungsaufgabe.)

Die tatsächlichen Kosten von `consolidate` sind `length ts`. Weil wir an den Defekten der Knoten gar nichts ändern, ist der Potentialunterschied also die Längendifferenz von Ein- und Ausgabeliste. Die Ausgabeliste ist $O(\log n)$ lang. (Übungsaufgabe.) Also fällt das Potential um `length ts` $- O(\log n)$. Die amortisierten Kosten sind deswegen die hier fehlenden $O(\log n)$.

5.5.7 Implementierungsdetails

In der Definition für F-Baum werden die Ränge werden zunächst sortiert. Das heißt, im Baum selbst können die Teilbäume wild durcheinander stehen. Wir hätten gar keine Zeit, dort für Ordnung zu sorgen: ein Knoten hat bis zu $\log n$ Kinder, die können wir gar nicht alle betrachten, wenn wir `decrease_key` in $O(1)$ ausführen wollen.

Damit alle Operationen schön schnell gehen, enthält jeder Baumknoten vier Zeiger:

1. auf seinen Vater
2. auf irgendein Kind (falls er welche hat)
3. auf linken Bruder (zyklisch)
4. auf rechten Bruder (zyklisch)

D. h. alle Geschwister sind untereinander durch eine doppelt verkettete zyklische Liste verbunden. Darin kann man in $O(1)$ ein Element einfügen oder löschen (wenn man einen Zeiger auf das Element hat).

Wir kommen auch von jedem Knoten zu seinem Vater und können auf diese Weise in $O(1)$ ein Kind abschneiden. Die Wurzelliste (der F-Bäume, die den F-Wald bilden) ist selbst eine doppelt verkettete zyklische Liste, also können wir den abgeschnittenen Teilbaum auch dort in $O(1)$ einfügen.

Wir müssen also bei jeder Operation einige Zeiger passend verbiegen. Ab jetzt gehen wir nicht weiter auf dieses Implementierungsdetail ein, weil das nur den Blick auf das wesentliche verstellt.

Beispielsweise haben wir oben bei `union` zwar `++` geschrieben (Listen aneinanderhängen), aber da beides doppelt verkettete Listen sind, geht das tatsächlich in konstanter Zeit.

Auch das `split_min` im `extract` geht in $O(1)$. Wir müssen dazu für den gesamten Wald ständig einen Zeiger auf den Baum mit der geringsten Wurzel bereithalten. Es ist einfach, diesen Zeiger zu aktualisieren, bei allen Operationen, für die das nötig ist. (Übung: welche sind das?)

5.5.8 Abschneiden von Teilbäumen

Die Operation `decrease_key` war ja überhaupt der Grund dafür, daß wir von Binomial– zu F-Heaps gewechselt sind. Wir müssen eine passende Funktion `rework` in das Schema zu `decrease_key` einsetzen.

Wir löschen `p` aus der Kinderliste seines Vaters `q` und kleben es hinter die Wurzelliste (`++ [me]`). Wir markieren den Vater als defekt (`construct (y, 1) ...`). War der Vater vorher noch nicht defekt (`d == 0`), dann reicht das. War er schon defekt, schneiden wir weiter.

Fibo

```
cut :: Fib a -> Pos -> Fib a
cut t0 p =
    if is_proper p
    then let q = parent p; (y, d) = the (key (peek t0 q))
             (pre, me : post) = splitAt (last p) (children (peek t0 q))

             t1 = poke t0 q (construct (Just (y, 1)) (pre ++ post))
             t2 = lift ( unlift t1 ++ [ unmark me ] )

         in  if 0 == d then t2 else cut t2 q
    else t0

unmark :: Fib a -> Fib a
unmark t = let (x, d) = tag t in  t { tag = (x, 0) }
```

Die tatsächlichen Kosten entsprechen der Anzahl der `cut`-Aufrufe. Bei jedem steigt die Länge der Wurzelliste (weil `me` dort eingefügt wird). Andererseits war `me` defekt (sonst hätten wir nicht geschnitten). Also sinkt die Anzahl der defekten Knoten um eins. (Übung: warum dürfen wir die Defekt-Markierung in `me` löschen?) Da diese im Potential doppelt zählen, gewinnen wir jedesmal eine Potentialeinheit. Genau diese brauchen wir, um die Aktion zu bezahlen. Die amortisierten Kosten sind insgesamt konstant.

Fibo

```
fdecrease_key :: Ord a =>  FH a -> Pos -> a -> FH a
fdecrease_key (FH ts) p x =
   FH ( unlift ( decrease_key_scheme cut (lift ts) p (x, 1) ) )
```

Vorsicht: `(x, 0)` geht nicht, weil der Knoten, in dem der Schlüssel verringert wird, schon defekt sein könnte. `(x, 1)` ist wahrscheinlich eine sichere Approximation.

Kapitel 6

Mengen, Wörterbücher, Suchbäume

In eine PQ stecken wir Elemente rein und holen die jeweils kleinsten wieder raus.

Wir suchen jetzt eine Struktur, in die wir Elemente einfügen, um sie zu *beliebiger* Zeit schnell wiederzufinden. Wiederfinden heißt: nach einem Schlüssel suchen.

Wir fassen Wörterbucher als Abbildungen von Schlüsselbereich `a` nach Wertebereich `b` auf.

Beispiel: Telefonbuch: Familienname $\mapsto$ Telefonnummer. Compiler: Variablenname $\mapsto$ Eigenschaftsliste (Ort der Deklaration, zugeordnete Speicherstelle, Typinformation uws.)

FiniteMap

```
class FiniteMap fm where
    empty     :: Ord a => fm a b
    is_empty  :: Ord a => fm a b -> Bool
    lookup    :: Ord a => fm a b -> a -> Maybe b
    insert    :: Ord a => fm a b -> a -> b -> fm a b
    --
    insert xs k v = union xs (singleton k v)
    delete    :: Ord a => fm a b -> a -> fm a b
    delete xs k   = error "delete not supported"
    singleton :: Ord a => a b -> fm a b
    union     :: Ord a => fm a b -> fm a b -> fm a b
    union xs ys = error "union not supported"
```

Darin haben wir schon default-Implementierungen angegeben (die dürfen (hier: sollen) bei Instantiierungen überschrieben werden): `delete` muß nicht unterstützt werden; `union` auch nicht, aber dann müssen wir uns für `insert` etwas anderes überlegen.

Aufgabe: Vergleiche das mit `FiniteMap.hs` aus dem `ghc`–Compiler. Welche zusätzlichen Operationen gibt es? Warum?

Mengen

Ein einfacher Fall von finite maps sind Mengen. Dort ist die einzige Information, die wir von den Schlüsseln brauchen, die Aussage, daß sie vorhanden sind. Wir vereinfachen die Signatur:

Set

```
class Set set where
    empty :: Ord k => set k
    is_empty :: Ord k => set k -> Bool
    insert :: Ord k => set k -> k -> set k
    lookup :: Ord k => set k -> k -> Maybe k

    build :: Ord k => [k] -> set k
    build = foldl insert empty

    contents :: Ord k => set k -> [ k ]
    --
    delete :: Ord k => set k -> k -> set k
    delete = error "delete not supported"
    union :: Ord k => set k -> set k -> set k
    union = error "union not supported"
    singleton :: Ord k => k -> set k
    insert s x = union s (singleton x)
```

Nacheinander einfügen

Set

```
nums :: [ Int ]
nums =  [111 * k `mod` 413 | k <- [10, 20 .. 200] ]

rums :: Int -> IO [ Int ]
rums n = do g <- getStdGen
            return $ take n (randomRs (0, 1000) g)

ins :: (Set set, Ord k) => [ k ] -> [ set k ]
ins xs = tail (i empty xs) where
   i h [] = [ h ]; i h (x : xs) =  h : i (insert h x) xs
```

6.1 Implementierung

Es gibt drei wesentliche Ansätze zur Implementierung von Wörterbüchern:

Arrays Wir nehmen die Schlüssel selbst als Indizes, und schreiben die Werte in die Zellen des Arrays. *Vorteil*: sauschnell. *Nachteil*: Platzverschwendung, wenn nur wenige der möglichen Schlüssel tatsächlich vorkommen.

Hashtabellen Wir wollen die Geschwindigkeit von Arrays weiter nutzen, aber Platz sparen. Wir wählen eine geeignete Array-Länge (ein bißchen größer als die maximal erwartete Anzahl der Einträge) und eine geeignete Funktion, die die Schlüssel in diesen Indexbereich abbildet. *Vorteil*: immer noch sehr schnell *Nachteil*: die Hashfunktion muß wirklich gut gewählt sein, d. h. die Schlüssel möglichst gut verteilen. Kollisionen lassen sich damit trotzdem nicht vermeiden, also müssen sie behandelt werden. Das verkompliziert die Algorithmen, wir brauchen mehr Laufzeit oder Zusatzstrukturen (Listen). Schlecht anwendbar, wenn die Verteilung und die Anzahl der Elemente im Wörterbuch vorher nicht absehbar ist. *Trotzdem* In typischen Anwendungen (Compiler) überwiegen die Vorteile! Hashing wirkt Wunder!

Suchbäume Wir ordnen die Elemente des Wöterbuchs in einem Baum an. Dabei soll jedes Element von der Wurzel aus leicht zu finden sein (d. h. nur durch Vorwärtslaufen). *Vorteil*: funktioniert immer *Nachteil*: wir brauchen mehrere Speicherzugriffe, um ein Element zu finden. Für schnellen Zugriff muß der Baum irgendwie balanciert sein, das Rebalancieren erfordert algorithmischen Aufwand (und Zeit).

Hashtabellen heißen manchmal *assoziative Arrays*. Einige Programmiersprachen (`(g)awk`,`perl`) haben das eingebaut. Damit kann man schön kurze Programme schreiben!

Aufgabe: Beispiele dafür! Quelltext der Implementierungen anschauen!

Sie kennen Hashtabellen aus dem Grundstudium, deswegen beschäftigen wir uns hier nur mit Suchbäumen.

6.2 Binäre Suchbäume

Die Idee bei binären Suchbäumen ist: alle Elemente links der Wurzel sind kleiner als das in der Wurzel, alle rechts — größer.

BinType

```
is_search_tree :: Ord k => Bin k -> Bool
is_search_tree t =
 and [    and [ l <= key n | l <- contents (left  n)  ]
       && and [ key n <  r | r <- contents (right n)  ]
     | n <- subtrees t, is_node n
     ]
contents :: Bin k -> [k]
contents n = if is_empty n then [] else
         contents (left n) ++ [ key n ] ++ contents (right n)
```

In solch einem Baum können wir einfach suchen:

BinType

```
straight_lookup n x =
  if is_empty n then Nothing
  else
    if x == key n then Just $ key n
    else if x < key n then straight_lookup (left  n) x
    else                   straight_lookup (right n) x
```

und auch einfügen:

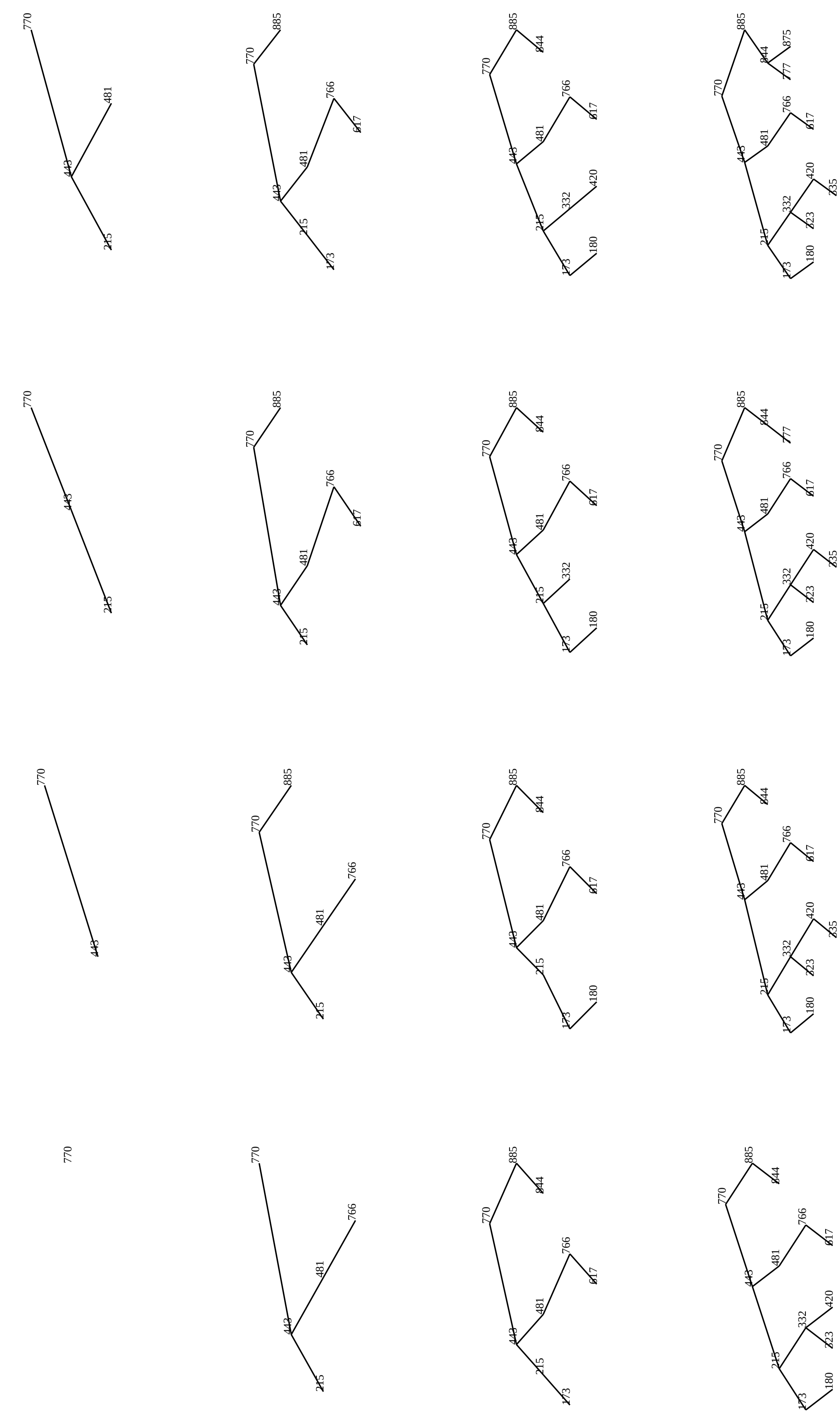
BinType

```
straight_insert :: Ord k => Bin k -> k -> Bin k
straight_insert n x =
  if is_empty n then node x empty empty
  else
    if x <= key n
    then node (key n) (straight_insert (left n) x) (right n)
    else node (key n) (left n) (straight_insert (right n) x)
```

Damit die Operationen schnell gehen, wollen wir logarithmische Tiefe. Durch das zu einfache Einfügen kann aber die Balance stark gestört werden.

UBin

```
newtype UBin k = UBin (Bin k) deriving Show
instance (Show k) => XForm (UBin k) where
    xform (UBin t) = xform t
instance Set.Set UBin where
    empty = UBin Leaf; is_empty (UBin Leaf) = True; is_empty _ = False
    insert (UBin t) x = UBin (straight_insert t x)
    lookup (UBin t) x =       straight_lookup t x
```



eine Folge von Einfügungen in einen unbalancierten Binärbaum

6.3 AVL–Bäume

Wir sollten deswegen von Zeit zu Zeit rebalancieren. Dazu gibt es verschiedene Methoden. Sie kennen bereits AVL–Bäume. Ein binärer Suchbaum ist ein (verallgemeinerter) AVL–Baum, wenn die Höhendifferenz aller Brüder beschränkt ist durch 1 (bzw. eine andere Konstante).

AVL

```
data AVL k = AVL Int (Bin k) deriving Show
instance Show k => XForm (AVL k) where xform (AVL d t) = xform t

diff :: Bin k -> Int
diff Leaf = 0; diff n = depth (left n) - depth (right n)

is_balanced :: AVL k -> Bool
is_balanced (AVL tolerance t) =
    and [ abs (diff s) <= tolerance | s <- subtrees t ]

is_avl_tree :: Ord k => AVL k -> Bool
is_avl_tree (AVL tol t) = is_balanced (AVL tol t) && is_search_tree t
```

Die wesentliche Operation auf solchen Bäumen ist das Rotieren (und Doppel–Rotieren). Das erhält die Suchbaum–Eigenschaft und ändert die Balance (richtig angewandt—zu unseren Gunsten).

Rotate

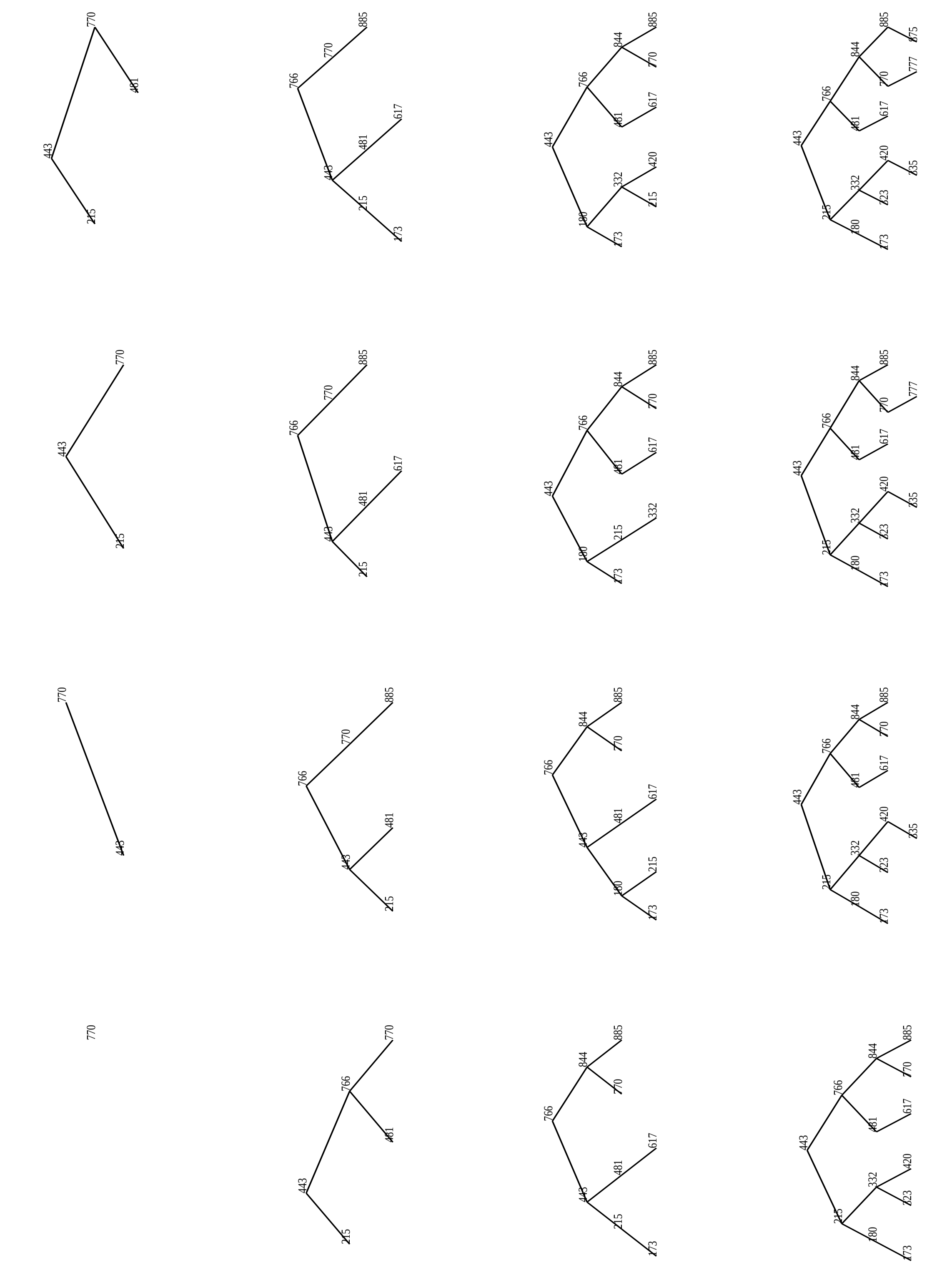
```
lotate :: Bin k -> Bin k
lotate t = node (key (right t))
                (node (key t) (left t) (left (right t)))
                (right (right t))
rotate :: Bin k -> Bin k
rotate t = node (key (left t))
                (left (left t))
                (node (key t) (right (left t)) (right t))

drotate :: Bin k -> Bin k
drotate t = rotate (node (key t) (lotate (left t)) (right t))

dlotate :: Bin k -> Bin k
dlotate t = lotate (node (key t) (left t) (rotate (right t)))
```

Wir gehen von `straight_insert` aus und rebalancieren, wenn nötig:

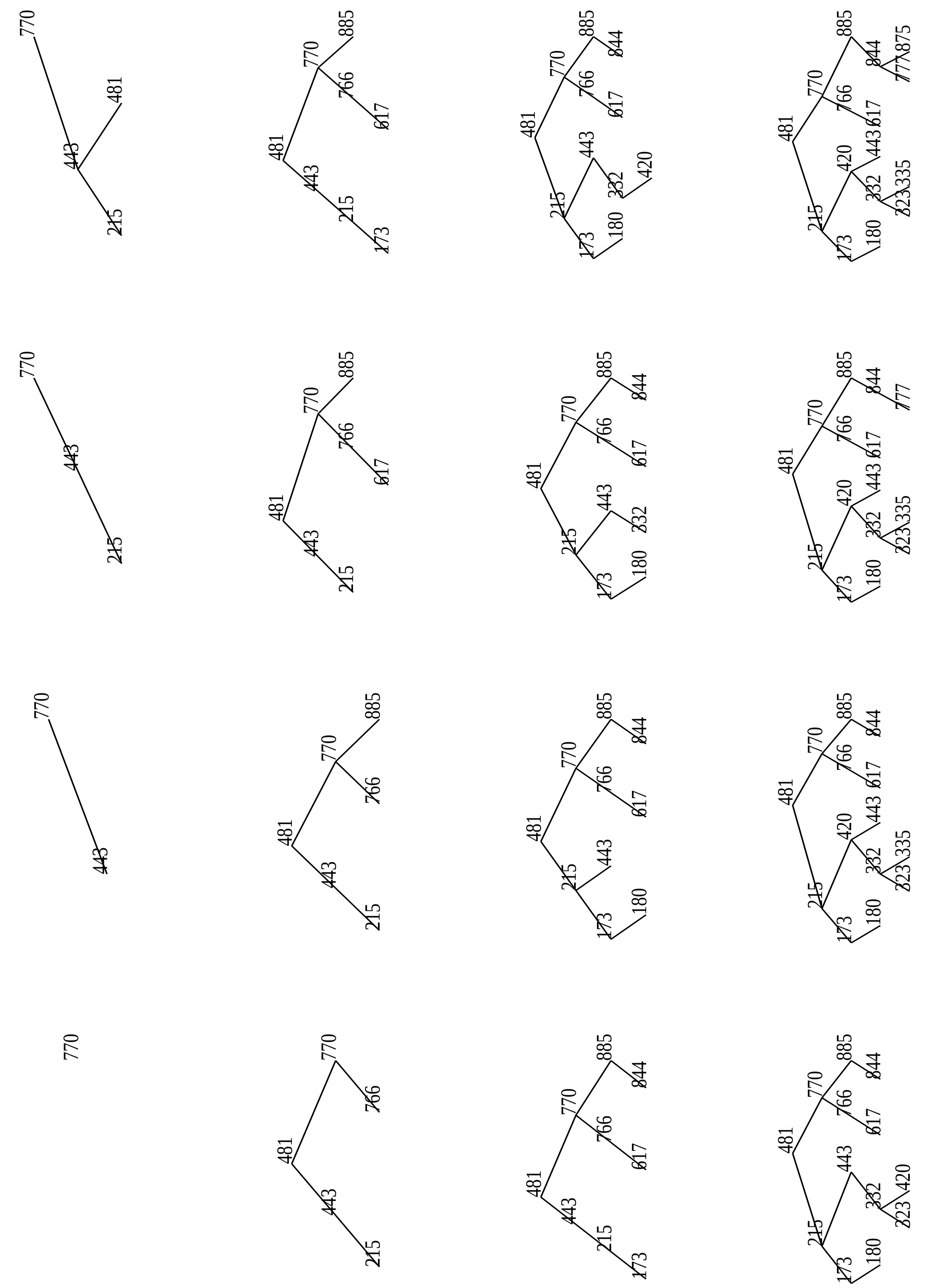
```
instance Set.Set AVL where                                             AVL
    empty = AVL 2 Leaf
    is_empty (AVL tol Leaf)   = True; is_empty _ = False
    lookup   (AVL tol t   ) x = straight_lookup t x

    insert   (AVL tol t   ) x = AVL tol ( insert t x )
      where insert Leaf x = node x empty empty
            insert t    x =
                 if x <= key t
                 then balance (node (key t) (insert (left t) x) (right t))
                 else balance (node (key t) (left t) (insert (right t) x))
            balance t = if diff t < - tol
                   then if diff (right t) < 0 then lotate t else dlotate t
                   else if tol < diff t
                   then if diff (left t ) < 0 then drotate t else rotate t
                   else t
```

Der zu rotierende Teilbaum ist der unterste (nicht der oberste!), in dem die Balance–Bedingung verletzt ist. Wiederholung: beweise, daß nach dem Einfügen eines neuen Elements eine einzige Rotation ausreicht, um die Balance wiederherzustellen.

Wie ist das beim Löschen in AVL–Bäumen?



Einfügen in Standard–AVL–Baum



Einfügen in AVL–Baum der Toleranz 2 — da die Folge „zufällig“war, sieht man hier wenig: die Schlüssel sind schon so verteilt, daß der Baum von selbst nicht sehr schief wird.

6.4 (a, b)–Bäume

Wir betrachten einen anderen Ausweg. Bei dem bleibt Vollständigkeit erhalten.

Def: ein Mehrweg-Baum ist ein (a, b)–Baum, wenn alle Blätter gleich tief sind und alle inneren Knoten höchstens b Kinder haben, sowie wenigstens a, außer der Wurzel, diese wenigstens 2.

Folgerung: die Höhe eines (a, b)–Baumes liegt zwischen $\log_b(n)$ und $\log_a(n)$. Das ist gut.

Soweit zur Form des Baums. Wir haben auch eine Forderung an den Inhalt. Das wird eine Übertragung der Suchbaumeigenschaft. Wir wollen wissen, in welches Kind wir beim Suchen abzusteigen haben. Dazu müssen wir in jedem Knoten t mit c Kindern $t_1, \ldots, t_c$ zwischen je zwei Kindern einen Schlüssel aufbewahren, insgesamt also $c-1$ Stück. Wir schreiben das als $t = (t_1, k_1, t_2, k_2, \ldots, t_{c_1}, k_{c-1}, t_c)$.

Def: ein Mehrwegbaum heißt geordnet (Mehrweg–Suchbaum), wenn für alle Knoten t und für alle i gilt: alle Schlüssel s in t_i erfüllen $k_{i-1} \leq s < k_i$, falls die enstspechenden k_i existieren.

Damit wir das besser aufschreiben können, spendieren wir schließlich noch $k_c = +\infty$. Tatsächlich werden wir den Vergleich mit $+\infty$ in einer Implementierung ersetzen durch den Test auf das Ende der Kinderliste. Wir abstrahieren hier davon.

ABType

```
data Topped k = Finite k | Infty deriving (Eq, Ord, Show)
top k = Finite k; untop (Finite k) = k

a, b :: Int; a = 2; b = 4
data AB k = Leaf  | Node [ (AB k, Topped k) ] deriving (Show)
```

AB

```
find :: Ord a => AB a -> a -> Maybe a
find Leaf k = Nothing; find (Node ps) k = findlist ps k

findlist :: Ord a => [ (AB a, Topped a) ] -> a -> Maybe a
findlist ((t, x) : txs) k =
    if      top k <  x then find t k
    else if top k == x then Just (untop x)    else findlist txs k
```

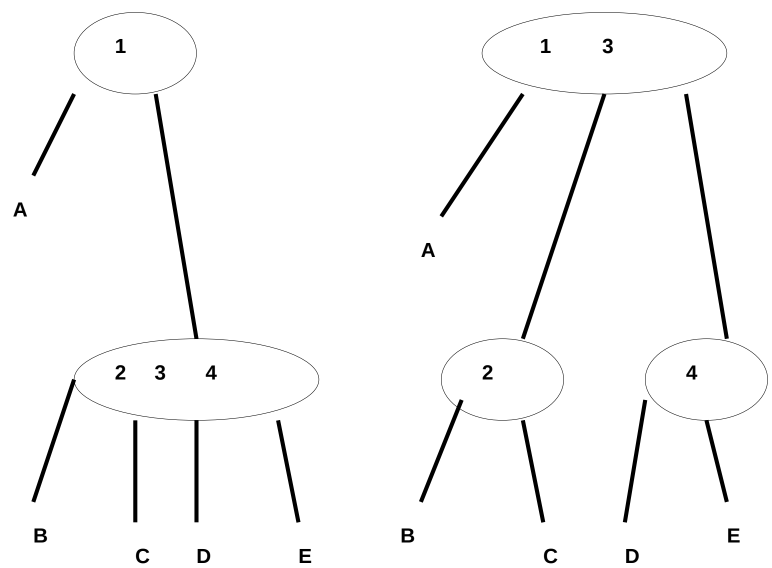
6.5 Einfügen in (a, b)–Bäume

Beim Einfügen vergrößern wir die Kinderliste eines Knotens. Das geht nur solange gut, wie die Länge unter b bleibt.

Steigt sie darüber, halbieren wir sie, und machen jeden der beiden Teile zu einem neuen Knoten. Das ist einer mehr als vorher, dieser wird nun seinerseits in seinen Vater eingefügt.

Das geht bei Bedarf immer so weiter, im schlimmsten Fall wird die Wurzel halbiert.

Das Bild zeigt das Halbieren eines Knotens mit 4 Kindern (in einem $(2, 3)$-Baum).



Durch das Halbieren müssen erlaubte Knoten entstehen, deswegen brauchen wir $\lfloor (b+1)/2 \rfloor \geq a$.

Mögliche (und tatsächlich auch benutzte Werte) sind $(2, 3)$, $(2, 4)$ — die treffen wir später noch, aber auch $(2, 100)$.

Übung: sich klarmachen, daß das immer noch ein geordneter Mehrwegbaum ist.

6.6 Weitere Operationen

Übungsaufgaben (wir später in der Vorlesung diskutiert, denke ich):

Löschen eines Elements. Sollte irgendwie invers zum Einfügen sein. Vorsicht: das einzufügende Element hat immer als Blatt begonnen. Was wir löschen wollen, ist nicht immer ein Blatt.

Vereinigen von zwei Bäumen. Vorsicht: die können verschieden tief sein, das Resultat muß natürlich überall die gleiche Tiefe haben.

6.7 Code zum Einfügen

Einfügen (erzeugt Liste von Bäumen). Erstes Argument ist Baum mit Schranke: Alle Schlüssel in `t` sind kleiner als `m`. Auch `x < m`.

```
ins :: Ord a =>                                                        AB
   (AB a, Topped a) -> Topped a -> [ (AB a, Topped a) ]
ins (Leaf, m) x = [ (Leaf, x), (Leaf, m) ]
ins (Node cs, m) x =  maybesplit (insfirst cs x) m
```

Das passende Kind suchen (das erste, für das `x < m`) und dort Schlüssel einfügen.

```
insfirst :: Ord a =>                                                   AB
   [ (AB a, Topped a) ] -> Topped a -> [ (AB a, Topped a) ]
insfirst ((t, m) : tms) x =
   if x < m then ins (t, m) x ++ tms
   else          (t, m) : insfirst tms x
```

Halbieren, falls nötig. In der Eingabe ist `m` eine Schranke für alle Elemente. Ergebnis hat Länge eins oder zwei.

```
maybesplit :: Ord a =>                                                 AB
   [ (AB a, Topped a) ] -> Topped a -> [ (AB a, Topped a) ]
maybesplit txs m =
    if length txs > b
    then let (pre, (t, x) : post) = splitAt (length txs `div` 2) txs
         in  [ (Node (pre ++ [ (t, Infty) ]), x), (Node post, m) ]
    else     [                                   (Node txs,  m) ]
```

Einfügen (erzeugt Baum)

```
insert :: Ord a => AB a -> a -> AB a                                   AB
insert t x =
    case ins (t, Infty) (top x) of
        [ (s, Infty) ]  {- Länge 1 -}  ->  s
        cs              {- länger  -}  ->  Node cs
```

Instanz machen

```
instance Set.Set AB where                                              AB
    empty = Leaf; is_empty Leaf = True; is_empty _ = False
    lookup = find; insert = insert

    contents Leaf = []
    contents (Node txs) =
        [ c | (t, x) <- txs, c <- Set.contents t ++ [ untop x | x /= Infty ] ]
```

Darstellung von (a, b)–Bäumen

Zum Zeichnen verwandeln wir einen (a, b)–Baum in einen Mehrweg–Baum:

ABPaint

```
ab2multi :: Ord k => AB k -> Multi (Topped k)
ab2multi t = a2m t Infty

a2m Leaf       top = construct (Just top) []
a2m (Node txs) top = construct (Just top) [ a2m t x | (t, x) <- txs ]
```

Das macht leider die „künstlichen“ `Infty`–Knoten sichtbar. Daraus ergibt sich aber andererseits eine

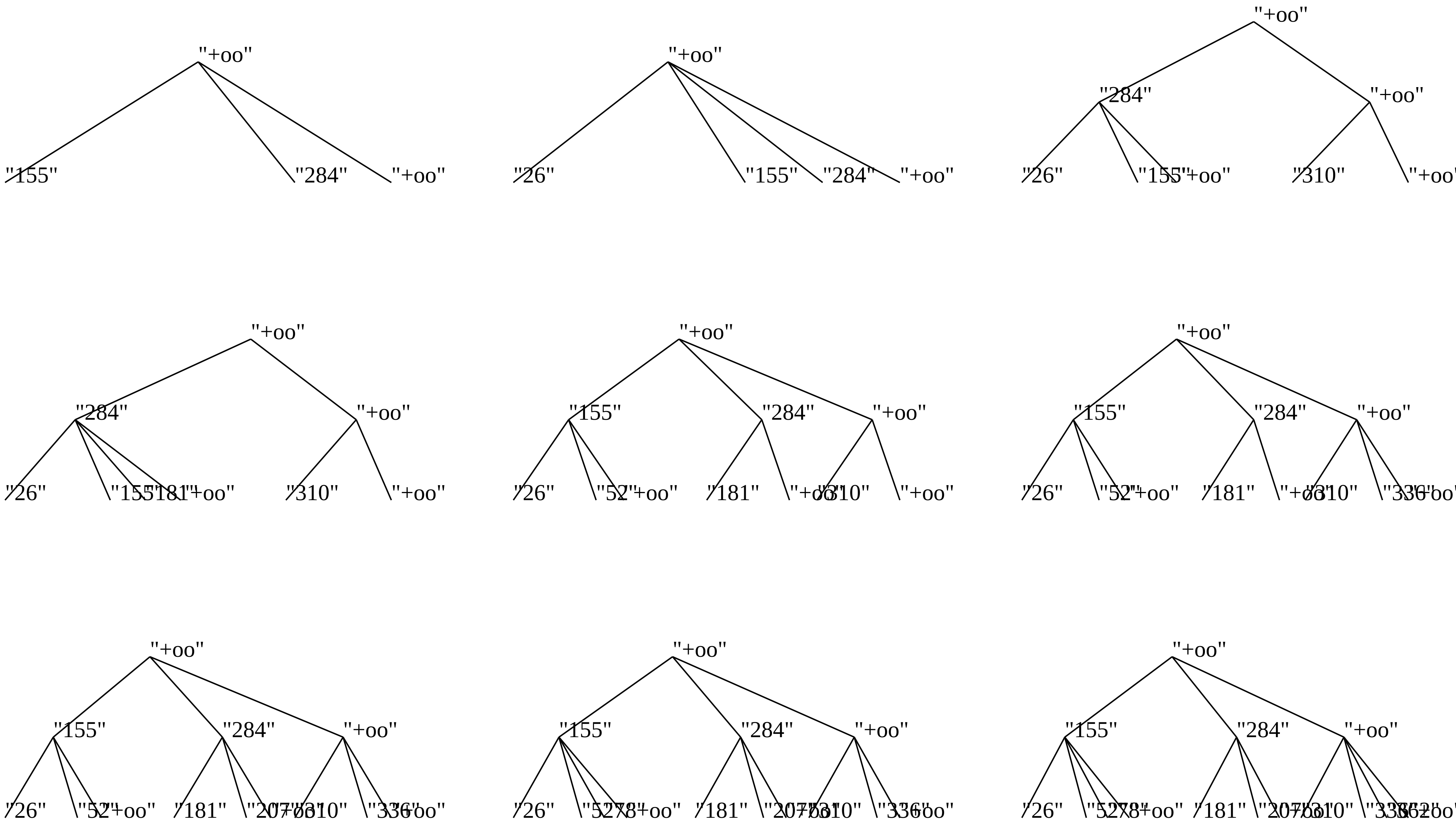
Aufgabe: welche Ordnungsbedingung gilt in den so erhaltenen Merhwegbäumen?

Aufgabe: man kann die `Infty`–Konten entfernen, indem ihre Kinder hochrutschen, welche Datenstruktur ergibt sich, welche Ordnungs– und Tiefenbedingungen, wie sehen die Algorithmen dafür aus? (Ich denke, irgendwas wie verallgemeinerte Binomialbäume.)

AB

```
instance (Ord k, Show k) => XFig.XForm (AB k) where
    xform = xform . fmap conv . ab2multi
        where conv Infty = "+oo"; conv (Finite x) = show x
```

Auf der nächsten Seite sehen wir eine Folge von Einfügungen in einen $(2, 4)$–Baum


"+oo"
"155" "284" "+oo"
"26" "155" "284" "+oo"
"284" "+oo"
"26" "310" "+oo"
"155" "284" "181" "310" "336" "52" "26" "207" "278"

6.8 Top-Down-(a, b)-Bäume

Nachteil dieses Verfahrens: beim Einfügen gehen wir erst runter (um die richtige Stelle für das neue Blatt zu finden), dann eventuell wieder hoch (wenn wir fortgesetzt halbieren müssen).

Wenn viele Leute einfügen wollen, muß jeder warten, bis der vorige komplett fertig ist. Viel besser wäre es, wenn jeder nur nach unten laufen muß, und keiner hoch. Dann kann der zweite schon anfangen, nachdem der erste nur einen Schritt ausgeführt hat.

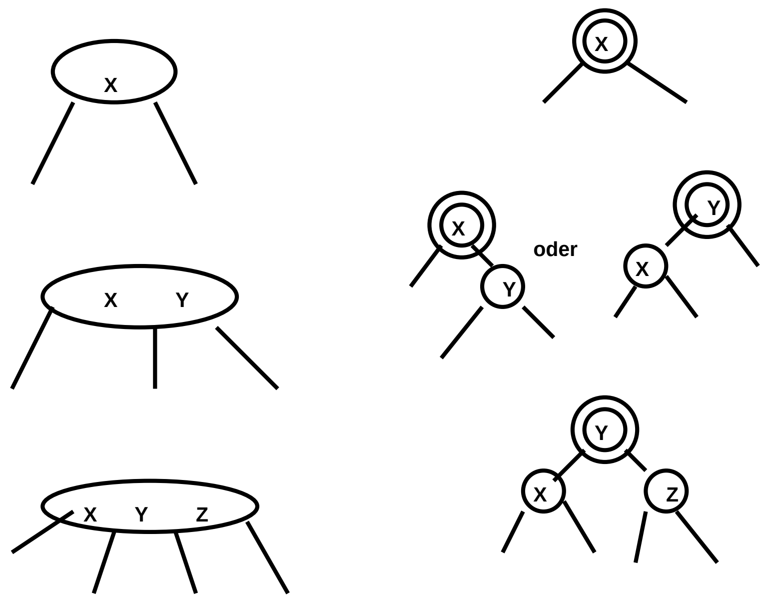
Das geht tatsächlich: wir halbieren einfach alle Knoten, die wir beim Suchen treffen, „auf Verdacht“. Damit sichern wir, daß auf dem Pfad alle Knoten $\leq \lceil b/2 \rceil \leq b - 1$ Kinder haben, also nach dem Einfügen überhaupt kein Halbieren mehr nötig ist.

Das klappt aber nur, wenn wir jeder halbierte Knoten auch zugelassen ist — also nicht in $(2, 3)$–Bäumen. Wir brauchen dazu $a \leq \lfloor b/2 \rfloor$. Das funktioniert als in $(2, 4)$-Bäumen.

Übung: Was müssen wir beim Löschen beachten?

6.9 Rot–Schwarz–Bäume

RS-Bäume sind binäre Suchbäume, die wir als Implementierung von $(2, 4)$-Bäumen auffassen.



Die doppelt umrandeten Knoten heißen *schwarz*, die anderen *rot*.

Dann haben wir folgende Strukturforderung:

Def: ein binärer Suchbaum heißt Rot–Schwarz–Baum, wenn auf allen Pfaden von der Wurzel zu Blättern

- gleichviele schwarze Knoten vorkommen
- nie zwei rote Knoten aufeinanderfolgen

Die roten betrachten wir als Hilfsknoten: wenn wir sie in ihren schwarzen Vater hochziehen, erhalten wir einen $(2, 4)$-Baum.

Um die Operationen auf RS–Bäumen zu implementieren, können wir uns jeweils vorstellen, was in der $(2, 4)$–Interpretation passiert, und das dann übertragen.

Diplomthema (ist ernst gemeint)

1) Tun Sie das! Starten Sie mit dem (a, b)–Kode aus Abschnitt 6.7, *spezialisieren* Sie diesen auf $a = 2, b = 4$, und *ändern* Sie dann *die Repräsentation* von Listen der Länge 2 bis 4 auf RS–Bäume. Beschreiben Sie die Transformationen im Einzelnen (d. h. beweisen sie die Korrektheit für jede).

2 — wäre schön) Idealerweise sollten sowohl Spezialisierung als auch Repräsentationsänderung als Annotationen im Programmtext stehen, die der Compiler dann ausführt. Denken Sie sich passende Syntax und Semantik für solche Annotationen aus.

3 — wäre noch schöner) Implementieren Sie das für `ghc`.

6.10 Allgemeines zu balancierten Suchbäumen

Erreichen wollen wir logarithmische Tiefe der Bäume. Das folgt beispielsweise aus beschränkter Höhendifferenz in jedem Knoten, siehe AVL–Bäume.

Übung: die haben Differenz höchstens eins. Man mache sich klar, daß das mit ≤ 3 oder jeder anderen Zahl auch geht. Wie sehen die Algorithmen zum Einfügen aus? Wieviel Rotationen braucht man jeweils? Wie tief können die Bäume werden d. h. welche Konstante versteckt sich dann in $O(\log n)$?

Übung: wenn wir nur konstantes Verhältnis der Höhen in jedem Knoten fordern, dann muß die Baumtiefe nicht mehr logarithmisch sein. Finde Beispiele für „schlechte" Bäume mit

```
and [ depth c >= a * depth s | s <- subtrees t, c <- children s ]
```
$\boxed{\texttt{none}}$

zum Beispiel $a = 1/2$.

Übung: was hat das mit Rot–Schwarz–Bäumen zu tun? Dort gilt doch diese halbe–Höhen–Bedingung auch. Trotzdem haben die immer logarithmische Tiefe.

Antwort: in Wirklichkeit gilt in diesen Bäumen mehr:

```
and [ shortest s >= a * depth s | s <- subtrees t ]
```
$\boxed{\texttt{none}}$

für $a = 1/2$, wobei `shortest` den *kürzesten* Pfad zu einem Blatt mißt (`depth` mißt natürlich den längsten):

```
shortest t = 1 + minimum [ shortest c | c <- children t ]
```
$\boxed{\texttt{none}}$

mit `minimum [] = 0`.

6.11 Gewichtsbalancierte Suchbäume

(Aufgabe 18–3 aus [CLR91], Seite 376f.)

Def: Ein binärer Baum ist a–balanciert, wenn

```
data WB k = WB Double (Bin k) deriving Show                          WB
instance Show k => XForm (WB k) where xform (WB a t) = xform t

is_wb_tree (WB a t) = and [ is_wb_node a s | s <- subtrees t, is_node s]
is_wb_node a s =
    dsize (left s) <= a * dsize s  &&  dsize (right s) <= a * dsize s
dsize t = fromInt (size t - 1 `div` 2)
```

Bsp: vollständige binäre Bäume sind 1/2–balanciert.

```
prop_complete = and [ is_wb_tree $ WB 0.5 (complete d) | d <- [1 .. 6] WB
```

Übung: welche anderen Bäume (also nicht mit $2^n - 1$ Knoten) sind 1/2–balanciert?

Aufgabe: Gegeben ein beliebiger Suchbaum. Wie kann man daraus in Linearzeit einen 1/2–balancierten Suchbaum machen?

Antwort: das geht nicht besser also so:

```
balance :: Ord k => Double -> [k] -> Bin k                           WB
balance a [] = empty
balance a xs =
    let c = truncate ( fromInt (length xs) * a )
        (pre, top : post) = splitAt c xs
    in  node top (balance a pre) (balance a post)

halve :: Ord k => Bin k -> Bin k
halve = balance 0.5 . contents
```

Aufgabe: Für irgendein $a \geq 1/2$: was ist die maximale Tiefe eines a–balancierten Baums? (Solange dauern ja dann die Operationen.)

Lokales Rebalancieren

Ist die a–Balance in einem Knoten gestört, können wir sie durch Rotationen wiederherstellen. (Genauso wie bei AVL–Bäumen.)

Übung: Ausprobieren für verschiedene Werte von a.

Was ist bei $a = 1/2$?

Diese Bäume heißen Braun trees (wirklich?)

Gründliches Rebalancieren

Das Einfügen erledigen wir wie gewohnt. Dadurch kann in einigen Knoten die a–Balance gestört werden. Ist das der Fall, suchen wir uns davon den höchsten, und rebalancieren den gesamten darunterhängenden Teilbaum, so daß dieser 1/2–balanciert wird.

WB

```
insert (WB a t) x = WB a $ fixup (straight_insert t x)
  where fixup s = if is_leaf s then s
                  else if is_wb_node a s then
                           if x <= key s
                           then node (key s) (fixup (left s)) (right s)
                           else node (key s) (left s) (fixup (right s))
                       else
                           halve s
instance Set.Set WB where
    empty = WB 0.75 empty
    is_empty (WB _ t) = is_empty t
    insert = insert; lookup (WB a t) = straight_lookup t
    contents (WB a t) = contents t
```

Das dauert natürlich eine Weile, aber wir zeigen, daß das nicht allzuoft passiert, und die Kosten amortisiert $\log n$ sind.

Die passenden Potentialfunktion ist

WB

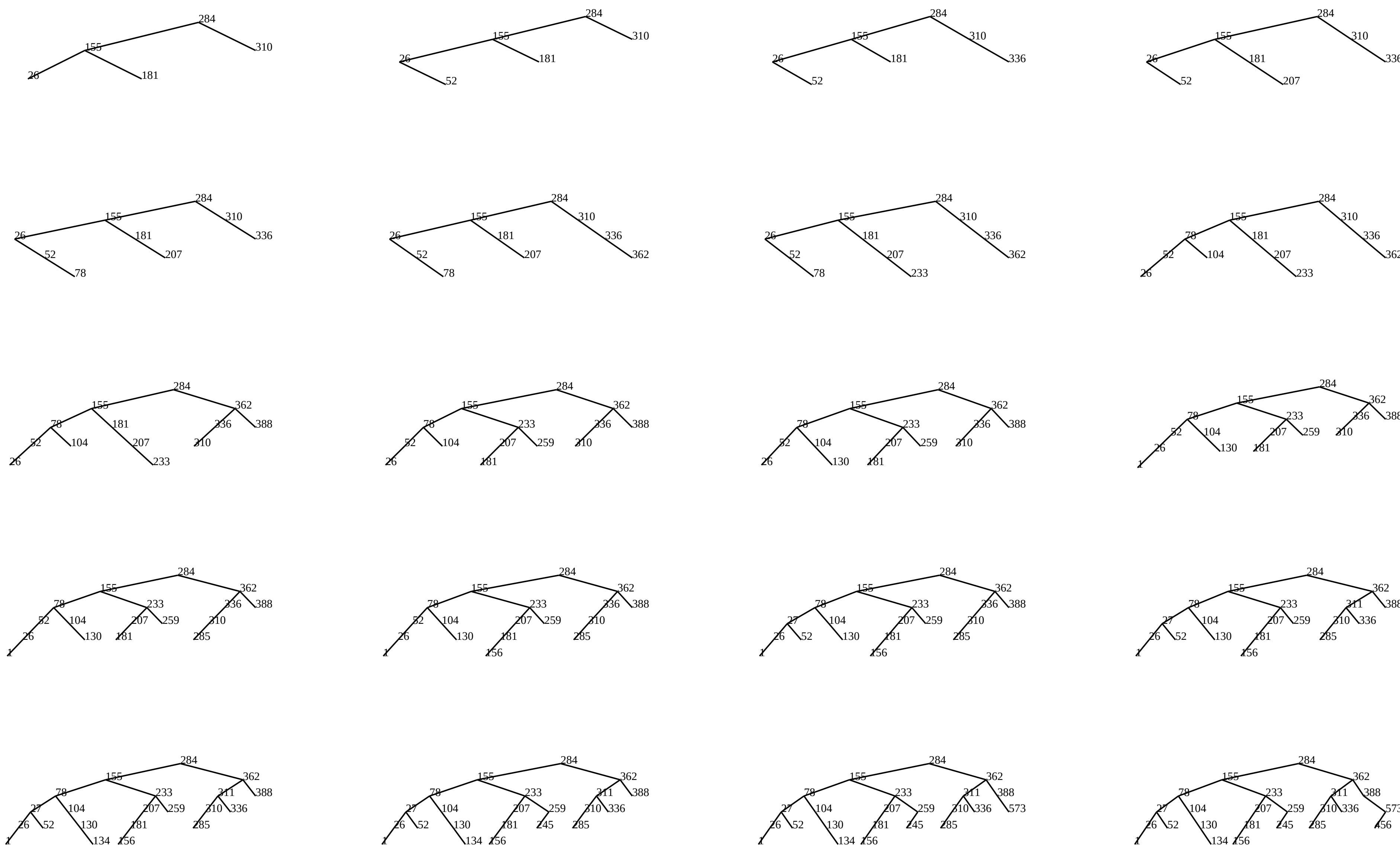
```
pot :: Bin k -> Double
pot  t = sum [ diff s | s <- subtrees t, diff s > 1 ]
diff s = if is_leaf s then 0
         else abs (dsize (left s) - dsize (right s))
```

Um wieviel kann sich das Potential durch ein Einfügen erhöhen? Um die Pfadlänge, d. h. Tiefe, d. h. $\log n$.

Wenn wir einen Teilbaum rebalancieren: Wie hoch ist sein Potential vorher? Nachher? Daraus können wir auch ablesen, welche Konstante sich in $O(\log n)$ für Einfügen verbirgt.

Übung (Ausrechnen, oder Implementieren und Messen): was ist der beste Wert für a? Ist es zu nahe bei 1/2, müssen wir zu oft rebalancieren. Ist es zu nahe bei 1, werden die Bäume zu dünn (tief).

Auf der nächsten Seite sehen wir eine Folge von Einfügungen in einen 3/4–balancierten Baum.


284 155 310 26 181
284 155 310 26 181 52
284 155 310 26 181 336 52
284 155 310 26 181 336 52 207
284 155 310 26 181 336 52 207 78
284 155 310 26 181 336 52 207 362 78
284 155 310 26 181 336 52 207 362 78 233
284 155 310 78 181 336 52 104 207 362 26 233
284 155 362 78 181 336 388 52 104 207 310 26 233
284 155 362 78 233 336 388 52 104 207 259 310 26 181
284 155 362 78 233 336 388 52 104 207 259 310 26 130 181
284 155 362 78 233 336 388 52 104 207 259 310 26 130 181 1
284 155 362 78 233 336 388 52 104 207 259 310 26 130 181 285 1
284 155 362 78 233 336 388 52 104 207 259 310 26 130 181 285 1 156
284 155 362 78 233 336 388 27 104 207 259 310 26 52 130 181 285 1 156
284 155 362 78 233 311 388 27 104 207 259 310 336 26 52 130 181 285 1 156
284 155 362 78 233 311 388 27 104 207 259 310 336 26 52 130 181 285 1 134 156
284 155 362 78 233 311 388 27 104 207 259 310 336 26 52 130 181 245 285 1 134 156
284 155 362 78 233 311 388 27 104 207 259 310 336 573 26 52 130 181 245 285 1 134 156
284 155 362 78 233 311 388 27 104 207 259 310 336 573 26 52 130 181 245 285 456 1 134 156

6.12 Splay Trees

Splay trees sind binäre Suchbäume.

```
newtype Splay a = Splay (Bin a)
```
none

Sie enthalten keinerlei Balance–Information. Die Algorithmen auf Splay trees rebalancieren „blind". Wir erhalten amortisiert logarithmische Kosten.

So etwas haben wir schonmal gesehen. Vereinfachend gesagt:

$$\frac{\text{splay}}{\text{AVL}} = \frac{\text{skew}}{\text{leftist}}$$

Die Idee bei leftist heaps war, durch Vertauschen von Teilbäumen das linke Rückgrat kurzzuhalten, weil das der Pfad im Baum ist, den wir bei der `union`–Operation benutzen. War bei skew heaps dieser Pfad sehr lang, hatte der Heap hohes Potential, das zum rebalancieren ausreichte. Danach war das Rückgrat wieder logarithmisch kurz.

Für splay trees geht das so ähnlich. Bei jedem Zugriff (Suchen und Einfügen) auf einen Knoten führen wir entlang des Zugriffspfades Rotationen aus (immer zwei, nur zuletzt eine oder keine).

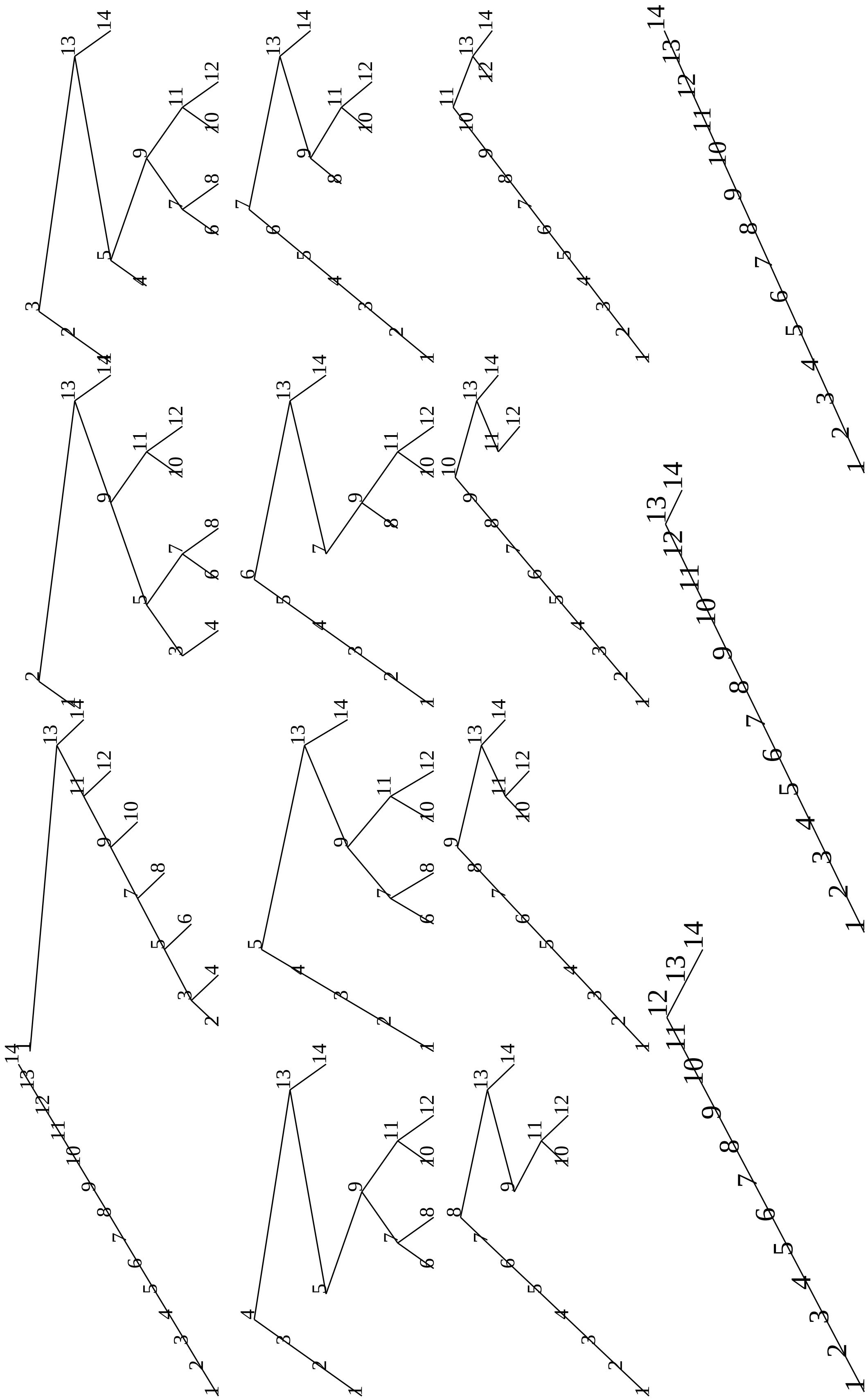
Die Wirkung dieser Rotationen ist, den zugegriffenen Knoten in die Wurzel zu bewegen und viele Pfadlängen zu verkürzen. War der Zugriffspfad lang, dauert das auch lange. Wir wählen aber eine geeignete Potentialfunktion, aus der wir genau diese Kosten bezahlen können.

```
log2 = logBase 2.0 . fromInt

pot :: Splay a -> Double
pot t = sum [ log2 (size s) | s <- subtrees t ]
```
none

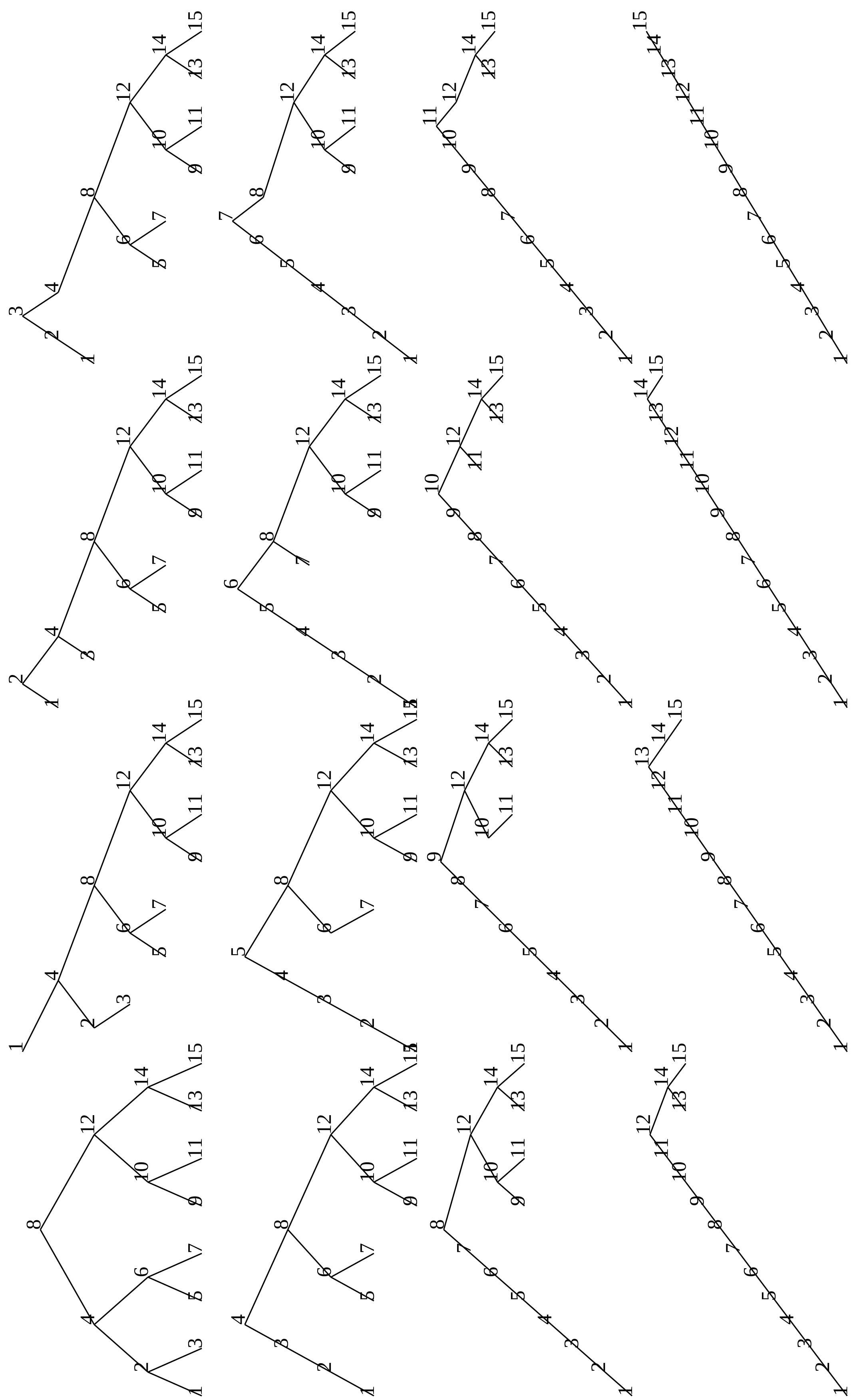
Übung: Potential eines vollständigen Baums mit dem eines schmalen Baums (Liste) vergleichen.

Die amortisierten Kosten eines `access` sind $3 * log2(sizet)$. Das müssen wir nachrechnen. An einer Stelle brauchen wir die Ungleichung zwischen arithmetischem und geometrischem Mittel:

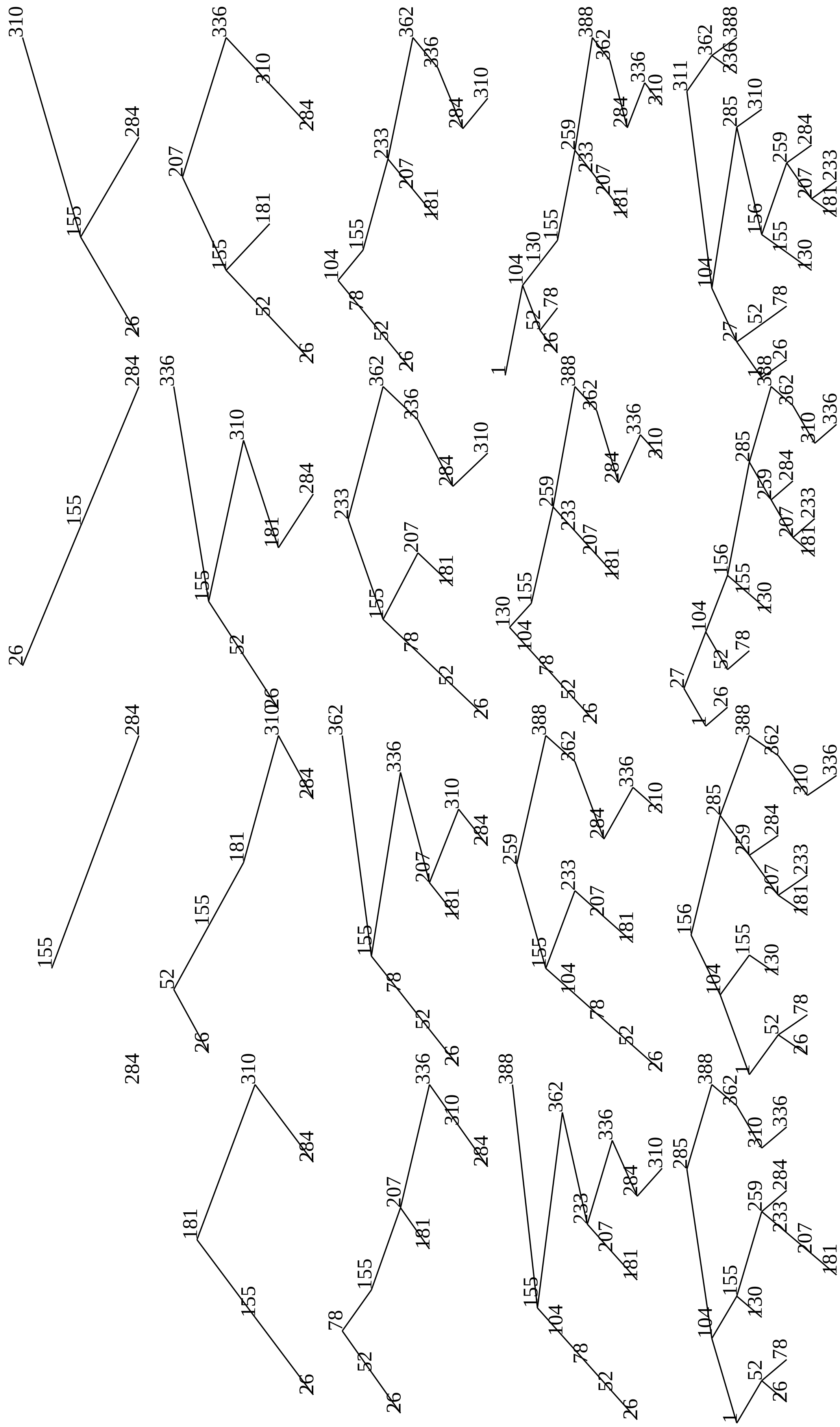
$$\log a + \log b = 2\log\sqrt{ab} \leq 2\log\frac{a+b}{2} \leq 2\log\frac{c}{2} = 2\log c - 2$$



`access` auf alle Knoten eines Splay–Trees



`access` auf alle Knoten eines Splay–Trees



Einfügen von „zufälligen“Schlüsseln in Splay–Trees

6.13 Pairing heaps

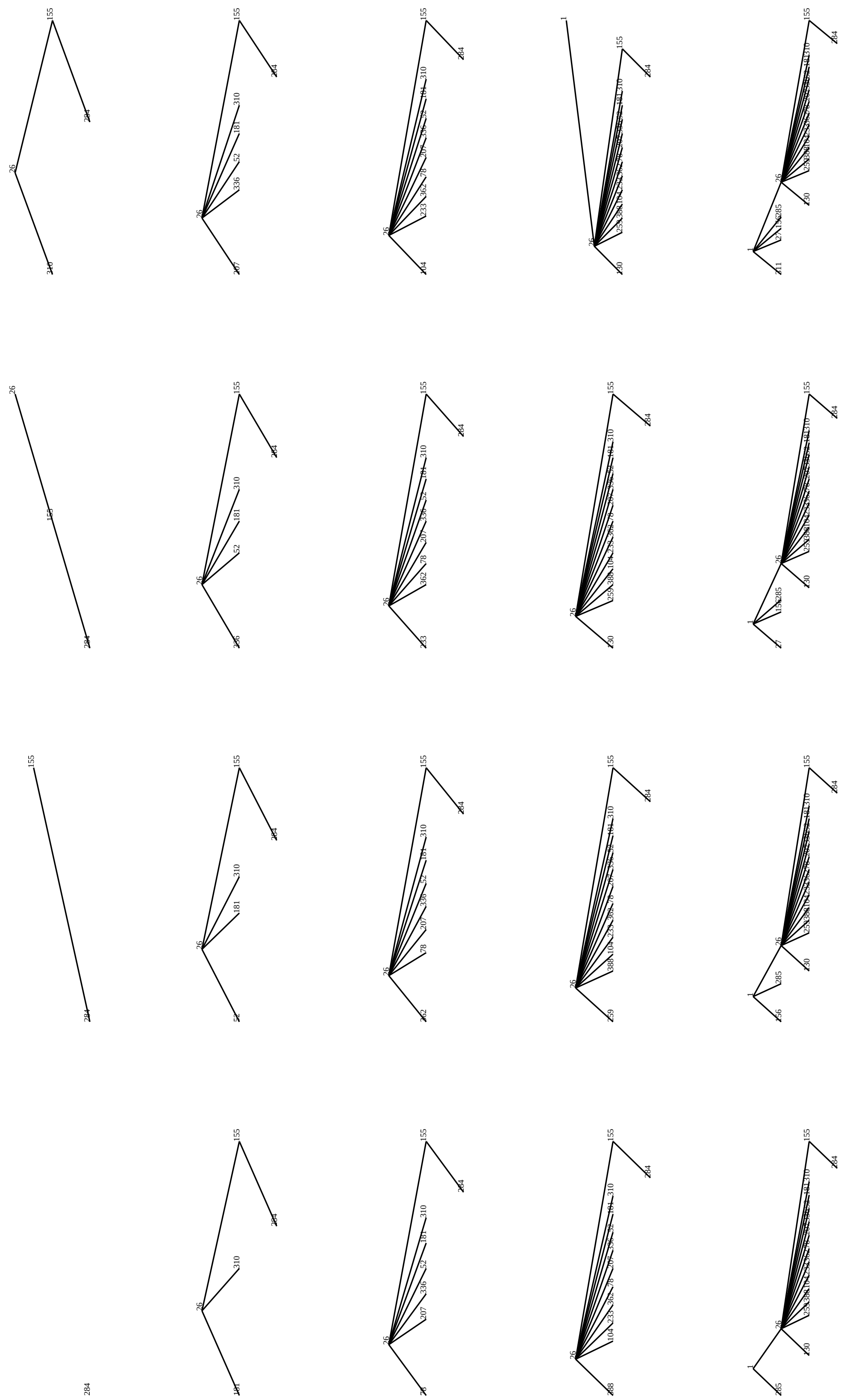
Schließlich eine Datenstruktur für Heaps, aber ähnlich zu Splay trees, für die man seit zehn Jahren vermutet, aber nicht beweisen kann, daß sie sehr schnell ist: `union, insert` in $O(1)$, `extract` in $O(\log n)$ amortisiert. Das können, wie wir gesehen und sogar bewiesen haben, Fibonacci–Heaps auch. Aber die Pairing heaps sind viel viel einfacher zu implementieren.

PH

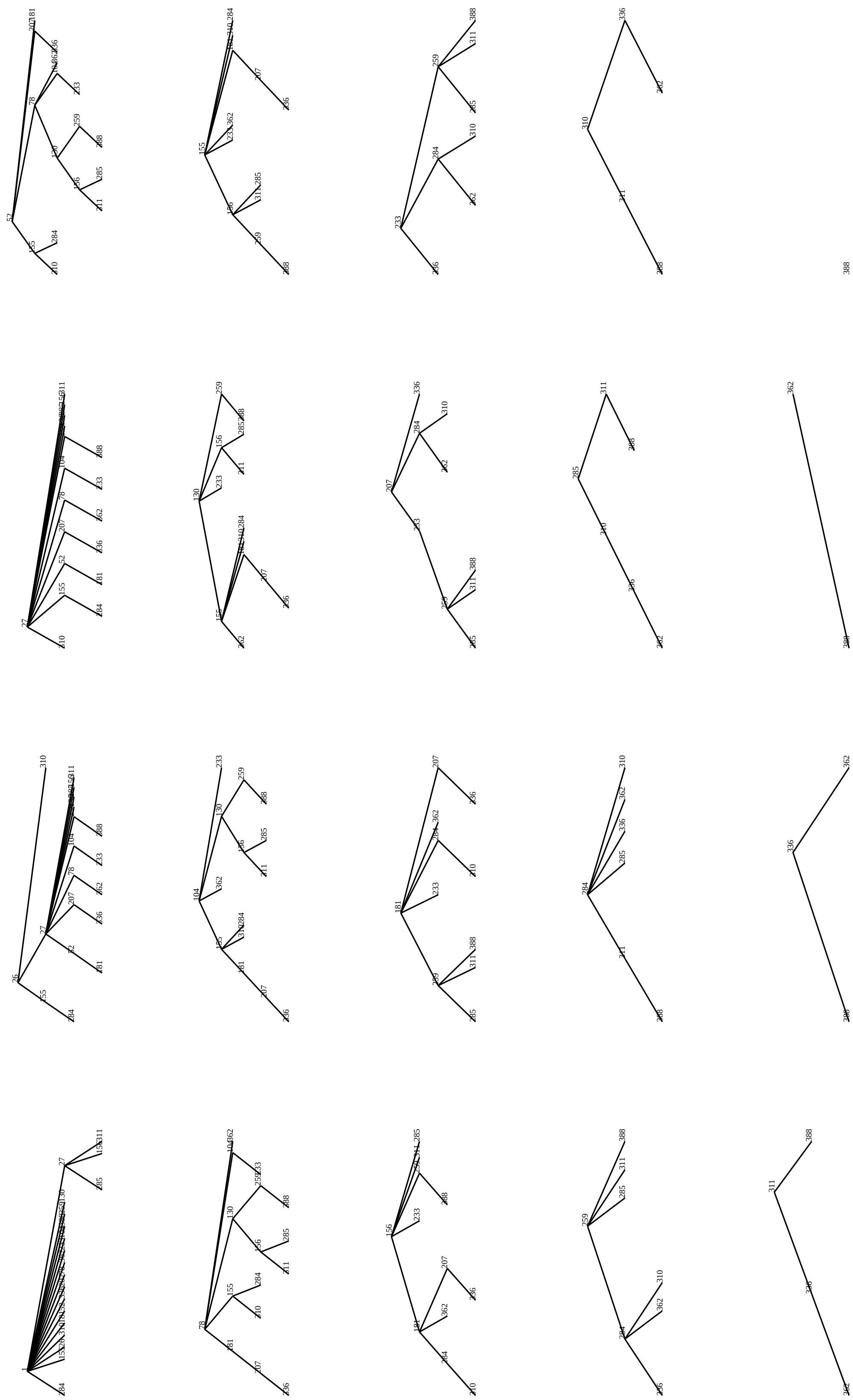
```
data PH k = Empty | PH (Multi k) deriving Show
instance Show k => XForm (PH k) where
    xform Empty = XFigType.Empty; xform (PH x) = xform x

instance PQ PH where
    empty = Empty; is_empty Empty = True; is_empty _ = False
    singleton x = PH ( Tree.construct (Just x) [] )

    union Empty y = y; union x Empty = x
    union (PH x) (PH y) = PH ( link x y )

    extract Empty = error "cannot extract from empty PH"
    extract (PH x) = (tag x, union_pairs (sub x) )

union_pairs :: Ord k => [ Multi k ] -> PH k
union_pairs [] = Empty; union_pairs [ x ] = PH x
union_pairs (x : y : zs) =
    let this = union (PH x) (PH y)
        rest = union_pairs zs
    in  union this rest
```

Übung: implementieren, messen, Potentialfunktion raten und $O(1)$ beweisen (wird man berühmt).

Vor–Übung: mit der Potentialfunktion für Splay–Heaps wenigstens amortisiert $O(\log n)$ für alles zeigen.



Einfügen einer „zufälligen“Schlüsselfolge in Pairing–Heap



Wiederholtes `extract` aus einem Pairing–Heap

6.14 Treaps: Heaps und Suchbäume in einer Person

(siehe Bemerkung in [Knu98], Abschnitt 6.2.3)

Ein Treap (= tree and heap) ist ein binärer Baum. Die Schlüssel sind Paare. Der Treap ist auf den ersten Komponenten ein Suchbaum und auf den zweiten ein Heap.

Treap

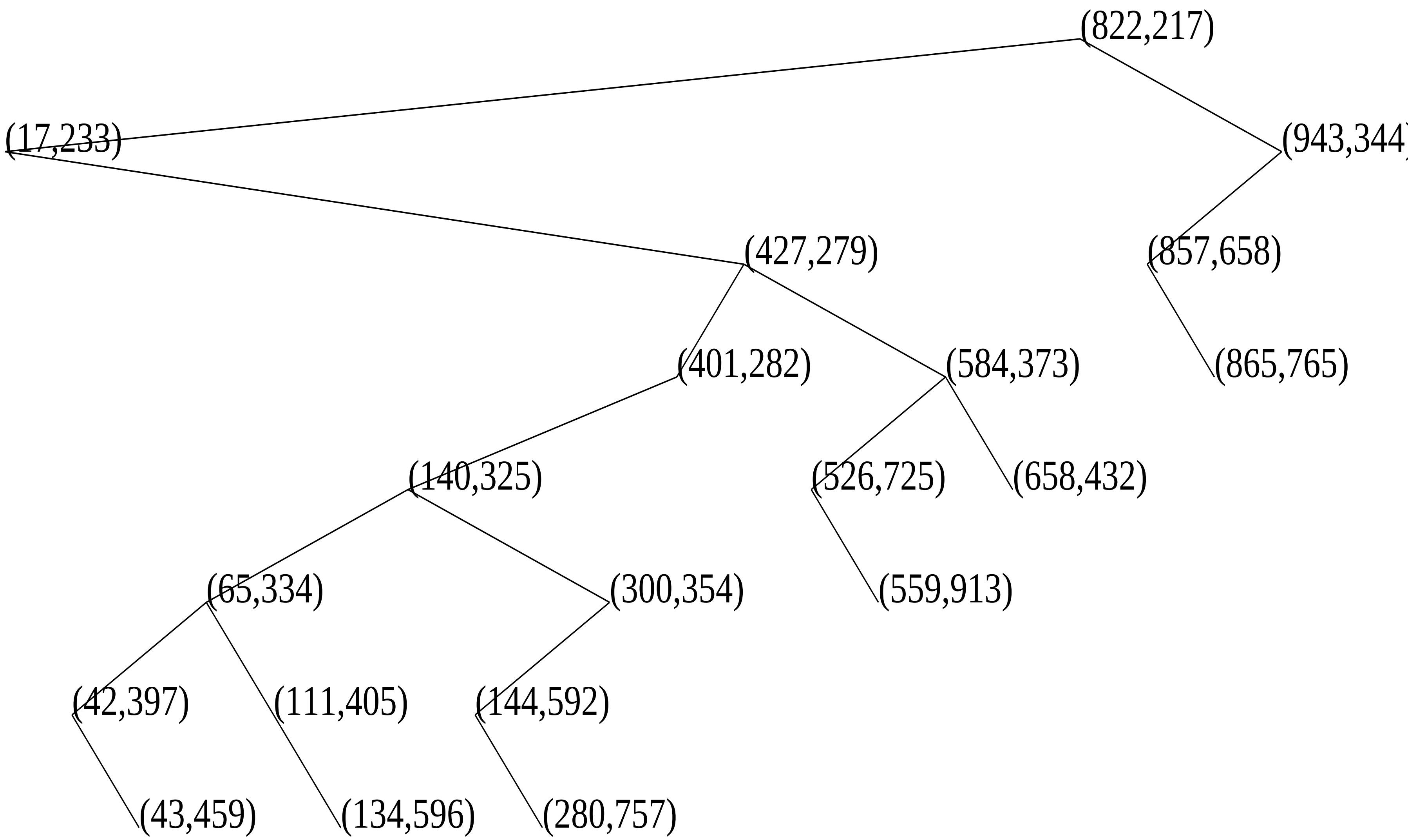
```
newtype Treap a b = Treap (Bin (a, b)) deriving Show
instance (Show a, Show b) => XForm (Treap a b) where xform (Treap t) = xform t

is_treap (Treap t) =  is_search_tree  (fmap (\ (x, y) -> x) t)
                   && is_heap_ordered (fmap (\ (x, y) -> y) t)
```

Das heißt, daß eindeutig festliegt, welcher Schlüssel in der Wurzel steht: der mit der kleinsten zweiten Komponente. Daraus folgt auch, welche Schlüssel im rechten bzw. linken Teilbaum sind: die mit größerer bzw. kleinerer erster Komponenten. Das gibt ein einfaches Verfahren, einen Treap zu bauen:

Treap

```
build :: (Ord a, Ord b) => [ (a,b) ] -> Treap a b
build [] = Treap Leaf
build xys =
    let ((tx, ty), rest) = findmin xys
        findmin [(x,y)] = ((x,y), [])
        findmin ((x,y) : rest) =
            let ((mx, my), larger) = findmin rest
            in  if y < my then ((x,y), (mx,my) : larger)
                          else ((mx,my), (x,y) : larger)
        (left, right) = partition ( \ (x, y) -> x < tx ) rest
        Treap l = build left; Treap r = build right
    in  Treap (node (tx, ty) l r)
```

Wir wollen Treaps für funktionale Implementierungen von PQs benutzen, die `delete` bzw. `decrease_key` unterstützen. Dazu benutzen wir die erste Koordinate als Adresse. Wir haben ja dort eine Suchbaumstruktur, und finden sie deswegen hoffentlich schnell. Beim Einfügen eines Schlüssels `y` in die PQ erzeugen (würfeln) wir eine Adresse `x` und fügen das Paar `(x,y)` in den Treap ein. Wir geben die Adresse `x` an den Aufrufer zurück, er muß sie sich merken. Will er an dem eingefügten `y` etwas ändern, finden wir es über die Adresse `x`. Der Witz ist, daß diese Adresse immer noch stimmt, auch wenn der Baum zwischendurch durch andere Operationen geändert wurde. Es bleibt ja dabei ein Suchbaum. Übung: implementiere das.


(822,217)
(17,233)
(943,344)
(427,279)
(857,658)
(401,282)
(584,373)
(865,765)
(140,325)
(526,725)
(658,432)
(65,334)
(300,354)
(559,913)
(42,397)
(111,405)
(144,592)
(43,459)
(134,596)
(280,757)

Ein Treap.

Kapitel 7

Sortieren und Suchen mit möglichst wenig Vergleichen

7.1 Einleitung

Wir arbeiten mit diesem Modell: Wir haben eine Eingabe $X = \{x_1, x_2, \ldots, x_n\}$ von paarweise verschiedenen Elementen aus einem Grundbereich G, und wir möchten genau die Permutation π der Indizes ausgeben, für die $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(1)}$.

Um diese zu finden, dürfen wir Fragen der Form $x_i < x_j$ stellen. Wir bemühen uns, die Anzahl der Fragen im schlechtesten Fall klein zu halten, d. h. wir suchen Algorithmen, die in der Lage sind, *jede* Eingabe der Länge n durch höchstens f solche Fragen zu sortieren. Wir nennen $S(n)$ die kleinstmögliche Zahl f, für die das gelingt.

Diese Aufgabenstellung wandeln wir auf zwei Weisen ab:

- Wir haben in der Eingabe schon gewisse Relationen $x_i < x_j$: Beispielsweise besteht die Eingabe aus zwei bereits geordneten Teilfolgen. Diese müssen wir dann nur noch zu einer einzigen verschmelzen. Haben die beiden Folgen die Längen m und n, nennen wir $M(m, n)$ die bestmögliche Fragen–Zahl zum Verschmelzen.

- Wir suchen gar nicht die vollständig geordnete Ausgabe, sondern nur das k–kleinste (oder –größte) Element von n. Beispielsweise ist das k–kleinste unter $2k + 1$ der Median, den möchten wir bei Quicksort verwenden. Wir bezeichnen mit $V_k(n)$ die kleinstmögliche erreichbare Fragen–Zahl.

7.2 Grenzen des Modells

Wir beachten, daß nur Fragen etwas kosten, und andere Operationen frei sind.

- Vorteil: wir können sehr genaue Kostenrechnungen anstellen (nicht nur bis auf Faktoren (big-Oh), sondern exakt).
- Nachteil: die für diese Kosten optimalen Algorithmen sind nicht immer praktikabel. Versteckte Kosten entstehen vorwiegend zur Speicherverwaltung (Zugriff auf, und Einfügen in, Arrays oder Listen).

Warum behandeln wir die Themen, trotz der Nachteile? Weil uns die Entwurfstechniken der Algorithmen interessieren, und die Beweistechniken für die Schranken. Das ist nämlich beides längst nicht nur auf Sortiervefahren anwendbar.

Motto: *Indeed I believe that virtually **every** important aspect of programming arises somewhere in the context of sorting and searching.*

Das sagt Donald Knuth, der es wissen muß, weil er ein dickes Buch geschrieben hat [Knu98], in dem es um nichts anderes als Sortieren und Suchen geht, und das seit 35 Jahren (!) das Standardwerk dazu ist.

Lesen Sie dieses Buch! Wenigstens den Abschnitt 5.3! Lösen Sie Übungsaufgaben daraus!

7.3 Halbordnungen, Schranken

Obere Schranken für $S(n), M(n), V_k(n)$ erhalten wir natürlich dadurch, daß wir effiziente Algorithmen hinschreiben.

Übungen: $S(4) \leq 5, S(5) \leq 7, M(2,6) \leq 6, M(2,8) \leq 6, V_3(5) \leq 6, V_4(7) \leq 10$.

Wenn wir aber trotz langen Nachdenkens keinen schnellen Algorithmus finden, brauchen wir untere Schranken. Das ist schwieriger, während früher *ein* guter Algorithmus ausreichte, geht es jetzt um Aussagen über *alle* Algorithmen (der Art: zu *jedem* Sortierverfahren für 5 Elemente gibt es eine Eingabe, für die das Verfahren mehr als 6 Vergleiche ausführt).

Hat das Verfahren bereits einige Fragen gestellt, sind also einige Relationen auf den eingegebenen Elementen bekannt. Da $\leq$ transitiv ist, haben wir damit eine Halbordnung H.

Wir möchten diese (durch weitere Fragen) zu einer totalen (linearen) Ordnung erweitern. Das heißt, aus der Menge $L(H)$ aller linearen Erweiterungen von H ist die richtige auszuwählen.

Auf jede Frage $x_i < x_j$ gibt es zwei mögliche Antworten. Das Sortierverfahren entspricht also einem binären Baum: die inneren Knoten sind die Halbordnungen, aus jedem inneren Knoten gehen zwei Kanten nach unten, die mit $x_i < x_j$ bzw. $x_i > x_j$ beschriftet sind, und die äußeren Knoten (Blätter) sind die linearen Ordnungen, d. h. die Ausgaben.

Für jeden Teilbaum mit Wurzel H (das ist die durch Fragen „über“ diesem Teilbaum bereits festgestellte Halbordnung) gilt:

$$\begin{aligned} \text{Blattzahl} &\leq 2^{\text{Tiefe}} \\ \|L(H)\| &\leq 2^{\text{Anzahl der restlichen Fragen}} \end{aligned}$$

Insbesondere ist für die völlige Unordnung H_0 (n Stück bisher unvergleichbare Elemente) an der Spitze des Baumes $\|L(H_0)\| = n!$. Das ergibt die *informationstheoretische Schranke für vergleichsbasierte Sortiervefahren*:

$$\text{Anzahl der Fragen} \geq \lceil \log_2(n!) \rceil$$

Beispiel: für $S(5) \geq \lceil \log_2(5!) \rceil = \rceil \log_2 120 \rceil = 7$. Das war knapp: von 120 ist es nicht weit bis 128. Der Entscheidungsbaum des Sortiervefahrens muß sehr gut balanciert sein.

Für $S(12)$ ist die informationstheoretische Schranke 29, aber es läßt sich durch geschicktes Absuchen aller erreichbaren Halbordungen zeigen, daß die 29 nicht erreichbar ist.

7.4 Auswahlverfahren

Wir suchen aus n Elementen das k–größte.

7.4.1 Einfache Schranken

Die informationstheoretische Schranke hilft hier nicht viel: Es sind n verschiedene Antworten möglich, also brauchen wir wenigstens $\lceil \log_2 \rceil$ Fragen. Das ist aber nur in den seltensten Fällen erreichbar.

Für $k = 1$ heißt die Aufgabe: finde das Maximum.

Wir sehen leicht, daß das mit $n - 1$ Vergleichen geht, und auch, daß es mit $< n - 1$ Vergleichen *nicht* geht. Beweis: betrachte die Anzahl der Zusammenhangskomponenten der Halbordnung. Sie sinkt durch jedem Vergleich um höchstens eins.

7.4.2 Tree Selection

Das übliche KO–Turnier für $n = 2^h$ Spieler besteht aus $2^h - 1$ Spielen (Vergleichen) und liefert natürlich den besten Spieler. Wie bestimmen wir den zweitbesten? Es ist nicht immer der Verlierer des Endspiels!

Der Zweitbeste ist aber auf jeden Fall jemand, der gegen den besten gespielt und verloren hat. Das sind genau h (im Allgemeinen $\lceil \log_2 n \rceil$) Leute. Unter diesen müssen wir den besten bestimmen. Dazu brauchen wir $h - 1$ Spiele, und schneller geht es nicht.

(Siehe extra Beweis).

Wir formulieren diese Verfahren um: äquivalent ist nämlich: nach dem Turnier, das den Besten bestimmt hat: ersetze diesen am Start durch $-\infty$, und spiele das Turnier nochmal, aber nur notwendige Spiele: d. h. Spiele aus dem vorigen Turnier werden nicht wiederholt, und gegen $-\infty$ gewinnt jeder kampflos. Offensichtlich ist der Gewinner dieses zweiten Turniers tatsächlich der zweitbeste von allen Startern.

finde t–größten von n Elementen durch tree selection:
  veranstalte KO–Turnier für alle
  für i von 2 bis t:
    nenne den Gewinner g, er scheidet aus
    ersetze g am Start durch $-\infty$
    wiederhole das Turnier (nur neue Spiele)
  der Gewinner ist der t–größte

Tree Selection ergibt eine Vergleichszahl von

$$V_t(n) \leq \text{erstes Turnier} + (t-1)\text{mal wiederholen}$$
$$V_t(n) \leq (n-1) + (t-1)(\lceil \log_2 n \rceil - 1)$$

Beispiel $V_3(5) \leq 4 + 2(\lceil \log_2 5 \rceil - 1) = 8$.

Das ist natürlich mehr als nötig, denn wegen $S(5) = 7$ hätten wir besser zunächst die gesamte Eingabe sortiert und dann einfach den dritten ausgewählt.

7.4.3 Implementierung von Tree Selection

Wir benutzen einen binären Baum, um die Turnierergebnisse aufzubewahren — mit zwei Modifikationen:

- Nachrutschen: in jedem Teilbaum steht der Teilturniersieger in der Wurzel, und in dem Teilbaum, aus dem er aufgestiegen ist, sind die dadurch nötigen Spiele (derer, die gegen ihn verloren hatten) bereits ausgeführt.
- Komprimieren: Spiele gegen $-\infty$ werden gar nicht erst aufgeschrieben. Statt $-\infty$ einzufügen, verkleinern wir den Baum.

Die wesentliche Operation ist das Verschmelzen zweier Turnierbäume. Dabei müssen die Nachrutsch–Spiele ausgetragen werden.

Pennant

```
fuse :: Ord a => Bin a -> Bin a -> Bin a
fuse l r =
     if is_empty l then r
     else if is_empty r then l
     else if key l > key r
          then node (key l) (fuse (left l) (right l)) r
          else node (key r) l (fuse (left r) (right r))
```

Mit wiederholtem `fuse` erzeugen wir das Turnier:

Pennant

```
tour :: Ord a => [a] -> Bin a
tour xs = foldbin fuse [ node x empty empty | x <- xs ]

foldbin :: (a -> a -> a) -> [a] -> a
foldbin op [x] = x; foldbin op xs = foldbin op (fb op xs)
    where fb op [] = []; fb op [x] = [x]
          fb op (x : y : zs) =  (op x y : fb op zs)
```

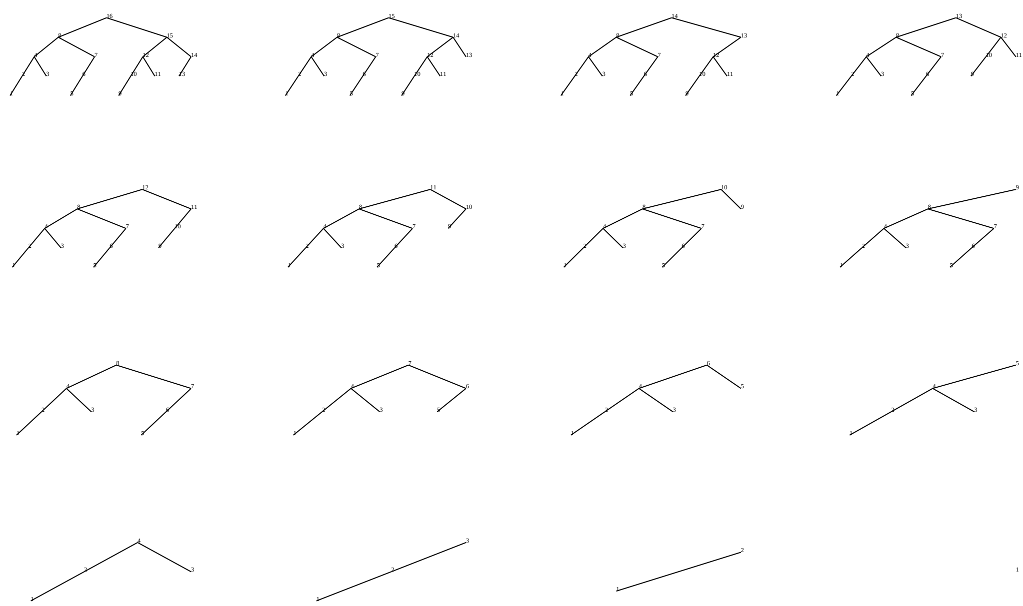
Den t–größten gewinnen wir durch t–maliges Extrahieren des Maximums.

Pennant

```
extract :: Ord a => Bin a -> (a, Bin a)
extract t = (key t, fuse (left t) (right t))

extracts :: Ord a => Bin a -> [ Bin a ]
extracts t = if is_empty t then []
             else let (top, rest) = extract t in t : extracts rest

select :: Ord a => Int -> [a] -> a
select t xs = key (extracts (tour xs) !! (t - 1))
```

Damit wir etwas sehen, bestimmen wir die gesamte Liste der Turniere. Das `select` ist dann einfach ein Zugriff auf die Wurzel des passenden Elements dieser Liste. Wegen der lazy evaluation wird nur der jeweils benötigte Präfix dieser Liste ausgewertet.

Übung: Das ist ein Sortierverfahren: wir betrachten alle Wurzeln (in umgekehrter Reihenfolge). Dieser Algorithmus führt genau dieselben Vergleiche aus wie eines der aus der Grundvorlesung bekannten Sortierverfahren. Welche? Können Sie das durch Programmtransformationen beweisen?



Fortgesetztes `extract` aus einem Turnierbaum

7.4.4 Schranke für $V_2(n)$

Satz (Schreier, Kislitsin):

$$V_2(n) \geq n - 2 + \lceil \log_2 n \rceil$$

Beweis (Knuth):

Da in jedem Spiel (Vergleich) genau einer verliert, ist die Gesamtzahl g der Spiele $= a_1 + a_2 + \ldots$, wobei $a_k :=$ Anzahl der Spieler, die wenigstens k–mal verloren haben.

Wenn wir den Zweitbesten kennen, dann steht der Beste auch fest (Übung — wenn der k–beste feststeht, hat die Halbordnung dort einen Artikulationspunkt. Hier $k = 2$).

Um den besten zu bestimmen, brauchen wir $\geq n - 1$ Spiele. Deswegen $a_1 \geq n - 1$ (bei jedem Spiel hat einer verloren).

Wir zeigen, daß sich Eingaben mit $a_2 \geq \log_2 n - 1$ nicht vermeiden lassen:

Der Sieger hat gegen p Leute gespielt (und gewonnen). Der Zweitbeste ist einer (und zwar der beste) von diesen p. Die anderen $p-1$ haben wenigstens noch einmal verloren (sonst stände der zweitbeste nicht eindeutig fest). Daraus folgt $a_2 \geq p - 1$.

Wir müssen also für jeden Algorithmus eine Eingabe finden, für die $p = \lceil \log_2 n \rceil$ ist. (Der beste muß gegen möglichst viele andere spielen, bevor wir sicher sein können, daß er es ist.)

Wir sind jetzt der „Gegner“ des Algorithmus und entscheiden erst während er die $x_i < x_j$–Fragen stellt, welche Antwor wir geben. Natürlich müssen unsere Antworten konsistent sein (d. h. früheren nicht widersprechen).

Unsere Strategie ist diese: A gewinnt gegen B, wenn

- A bisher immer gewonnen hat, aber B schon einmal verloren
- oder beide ungeschlagen sind, aber A mehr Spiele gewonnen hat als B

In allen anderen Fällen können wir nach Belieben antworten (aber müssen Konsistenz beachten).

Wir definieren $A \gg B$, falls $A = B$ oder $A \gg$ der erste (im Verlauf des Turniers), der B schlug.

Damit gilt für unsere Strategie: jeder Spieler, der seine ersten p Spiele gewonnen hat, ist $\gg$ als höchstens 2^p Spieler (ihn selbst eingeschlossen). (Beweis durch Induktion.)

Der Beste ist aber $\gg$ alle anderen, daraus folgt die Behauptung.

7.4.5 Replacement Selection

Die eben ausgetragenen Turniere waren zu groß. Wir haben nämlich nicht einfach den drittbesten bestimmt, sondern auch vorher den ersten und dann den zweiten. Die genaue Reihenfolge der beiden, die besser als der dritte sind, interessiert uns aber gar nicht. Wir könnten dort Vergleiche sparen.

Die Turniere (bis auf das letzte) müssen jeweils nur so groß sein, daß der jeweilige Gewinner auf jeden Fall *nicht* der insgesamt t–größte ist (sondern größer). D. h. er muß *mehr* als $n-t$ Leute unter sich haben. Also setzen wir ein KO–Turnier für $n-t+2$ Spieler an. Dessen Sieger ist besser als $n-t+2-1>n-t$, also besser als der t–beste. Nachdem auf diese Weise $t-1$ Sieger ausgeschieden sind, finden wir den besten der noch im Turnier befindlichen durch Einsetzen von $-\infty$.

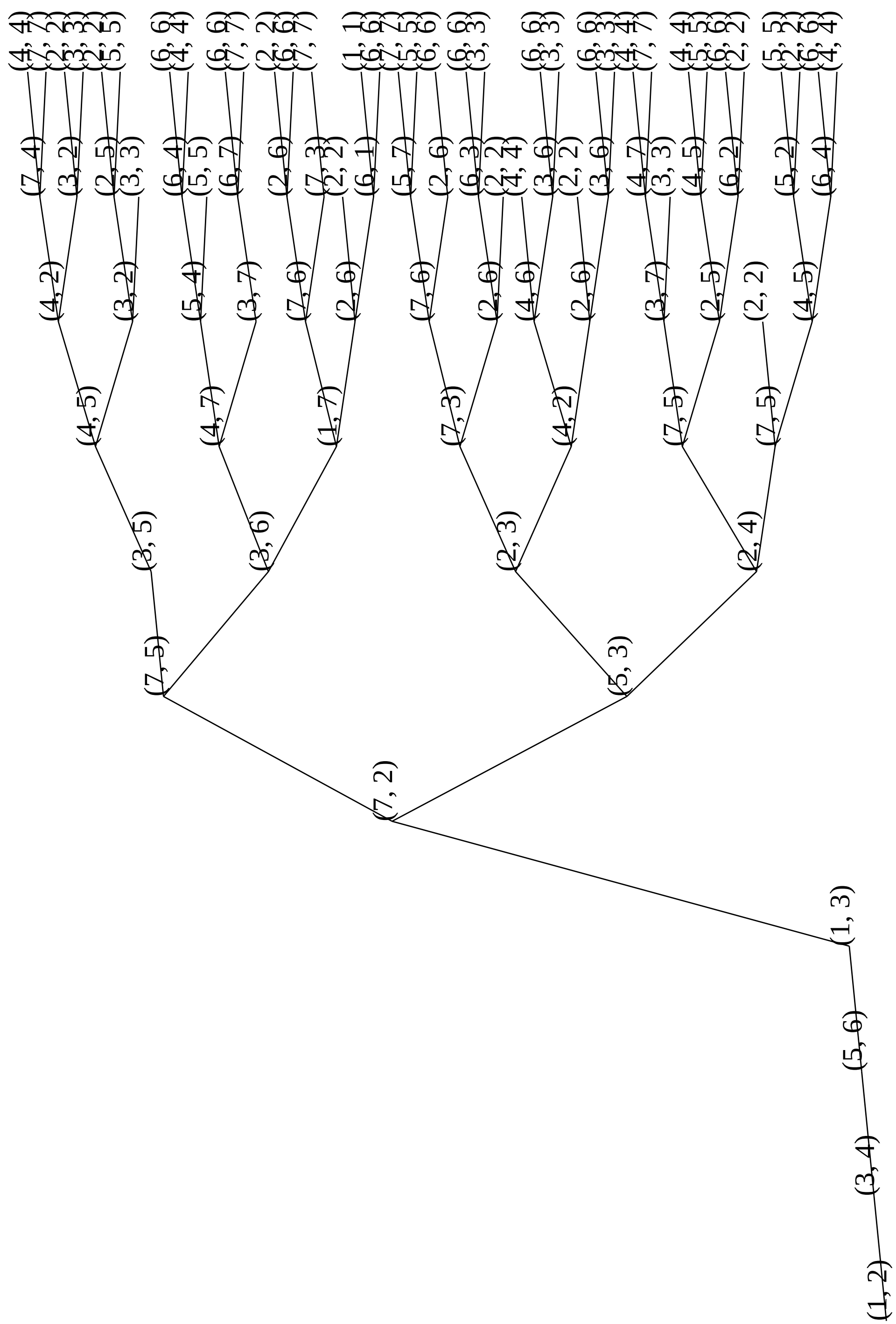
finde t–größten von n Elementen durch replacement selection:
  spiele KO–Turnier für $(n-t+2)$ Spieler
  (die anderen $(t-2)$ warten draußen)
  für i von 1 bis $t-2$:
    nenne den Gewinner g, er scheidet aus
    wähle einen der Wartenden w aus,
    ersetze g am Start durch w
    wiederhole das Turnier (nur notwendige Spiele)
  ersetze den letzten Gewinner am Start durch $-\infty$
  wiederhole das Turnier (nur notwendige Spiele)
  der Gewinner ist der t–beste.

Das ergibt eine Laufzeit von

$$V_t(n) \leq \text{erstes Turnier} + (t-2)\text{mal wiederholen} + \text{letztes mal wiederholen}$$
$$V_t(n) \leq (n-t+1) + (t-2)\lceil \log_2(n-t+2)\rceil + (\lceil \log_2(n-t+2)\rceil - 1)$$
$$V_t(n) \leq (n-t) + (t-1)\lceil \log_2(n-t+2)\rceil$$

Diese Schranke ist für $n \leq 6$ exakt. Beispiel: $V_3(5) \leq 2 + 2\log 4 = 6$, replacement selection ist ein optimales Verfahren, um den Median von 5 Elementen zu finden.

Allerdings ist $10 = V_4(7) \leq 3 + 3\cdot 3 = 12$.



Optimales Verfahren für $V_4(7) = 10$
(symmetrische Zweige sind unterdrückt)

7.4.6 Median in Linearzeit

Die Auswahl des k–besten aus $2k(+1)$ ist die Bestimmung des Medians. Das möchten wir für Quicksort benutzen. Für dessen Laufzeit auf n Elementen gilt dann

$$Q(n) \quad = \quad 2 \cdot Q(n/2) + \text{Median finden} + \text{partitionieren}$$

Das Partitionieren (um den Pivot herum) dauert sowieso $O(n)$. Wenn wir es schaffen, den Median auch in $O(n)$ zu finden, dann ist die Gesamtlaufzeit $Q(n) = O(n \log n)$.

Die bisher gezeigten Verfahren sind dazu ungeeignet, weil sie für $t = n/2$ Laufzeiten von $\Theta(n \log n)$ liefern. (Das triviale Verfahren: erst sortieren, dann mittleren wählen, geht sowieso nicht. Wir wollen es ja erst in ein Sortierverfahren einbauen!)

Das folgende rekursive Verfahren (nach Blum, Floyd, Pratt, Rivest, Tarjan) bestimmt den Median von n Elementen in $\Theta(n)$ Vergleichen. Es zerlegt die Eingabe in Gruppen von $2k+1$ Elementen, bestimmt die Mediane der Gruppen (in konstanter Zeit, da die Größe der Gruppe festliegt!), und dann den Median der Mediane (durch rekursiven Aufruf). Um diesen herum wird dann partitioniert, und im passenden Teil rekursiv selektiert.

Wir müssen noch die Parameter `k` und `cut` geeignet festlegen. Vor allem auf `k` kommt es an.

Die Laufzeit $T(n)$ für ein `select t xs` in Abhängigkeit von `n = length xs` ist:

$$\begin{aligned} T(n) &\leq \text{Mediane der Gruppen} + \text{Median der Mediane} \\ &\quad + \text{Zerlegen von } \texttt{bcs} + \text{Select im richtigen Teil} \\ T(n) &\leq C_k \cdot \frac{n}{2k+1} + T(\frac{n}{2k+1}) + \frac{k \cdot n}{2k+1} + T(\frac{n(k+1)}{2(2k+1)} + \frac{k \cdot n}{2k+1}) \end{aligned}$$

Wir haben $T(n) = O(n)$, falls die Summe der Koeffizienten vor n innerhalb der rekursiven Aufrufe kleiner als $1 - \epsilon$ ist:

$$\begin{aligned} \frac{1}{2k+1} + \frac{k+1}{2(2k+1)} + \frac{k}{2k+1} &< 1 - \epsilon \\ \frac{3/2 \cdot (k+1)}{2k+1} &< 1 - \epsilon \end{aligned}$$

Für $k = 1$ geht das also nicht, aber für die weiteren.

Warum? Der Teil der Eingabe, auf den die Funktion schließlich rekursiv angewandt wird, hat immer (etwa) die Größe $\leq 3/4 \cdot n$. Deswegen müssen die Gruppen echt mehr als 4 Elemente enthalten. Der erste Fall, für den es klappt, ist $k = 2$, also 5 Stück pro Gruppe.

Wir müssen Select auf Gruppen–Medianen ausführen und dabei die (bereits gegenüber ihrem Median angeordeten) Elemente mitführen. Die weiteren Vergleiche sollen sie natürlich nicht beeinflussen. Wir packen das alles als Zusatzinformation in einen Datentyp, und vergessen sie beim Vergleichen einfach.

Median

```
data Chain a = Chain [a] a [a] deriving Show
instance Eq  a => Eq  (Chain a) where
         Chain _ x _    ==     Chain _ y _ = x    ==     y
instance Ord a => Ord (Chain a) where
         Chain _ x _ `compare` Chain _ y _ = x `compare` y
```

Median

```
median :: Ord a => [a] -> Chain a
median xs = select (length xs `div` 2 + 1) xs

select :: Ord a => Int -> [a] -> Chain a
select t xs =
    if length xs <= cut then iselect t xs else
    let groups = chop (2 * k + 1) xs
        chains = [ imedian gr | gr <- groups ]

        Chain lowchains midchain highchains = median chains -- !!
        Chain mlow med mhigh = midchain

        as   =  [ x | Chain l m h <-  lowchains, x <- m : l ] ++ mlow
        bcs  =  [ x | Chain l m h <-  lowchains, x <- h     ]
             ++ [ x | Chain l m h <- highchains, x <- l     ]
        ds   =  [ x | Chain l m h <- highchains, x <- m : h ] ++ mhigh

        (here, there) = partition (\ x -> x < med) bcs
        low  = as ++ here; high = ds ++ there

    in  if t <= length low
        then let Chain lo mi hi = select t low
             in  Chain lo mi (hi ++ [med] ++ high)

        else if t == 1 + length low then Chain low med high

        else let Chain lo mi hi = select (t - 1 - length low) high
             in  Chain (lo ++ [med] ++ low) mi hi
```

Bemerkung zum Programm: es enthält zwei rekursive Aufrufe, der erste (!!) ist interessant — betrachte den Argumenttyp: `chains :: [Chain a]`. Das führt also zu einem `select` mit dem Resultattyp `Chain (Chain a)` usw.

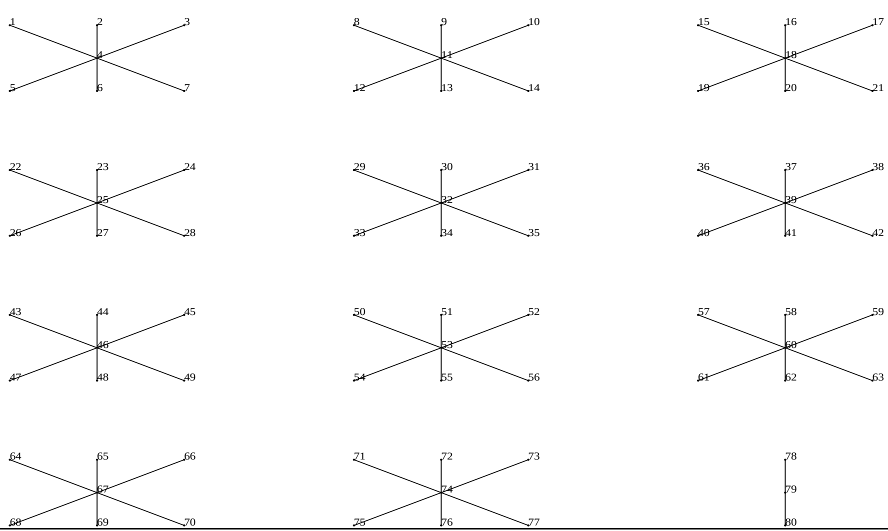
Die nicht erklärten Funktionen sind: `imedian, iselect`: Median und Selektion für kurze Argumentlisten, `chop d`: Zerlegen in Listen der Länge `d` (die letzte Teilliste evtl. kürzer).

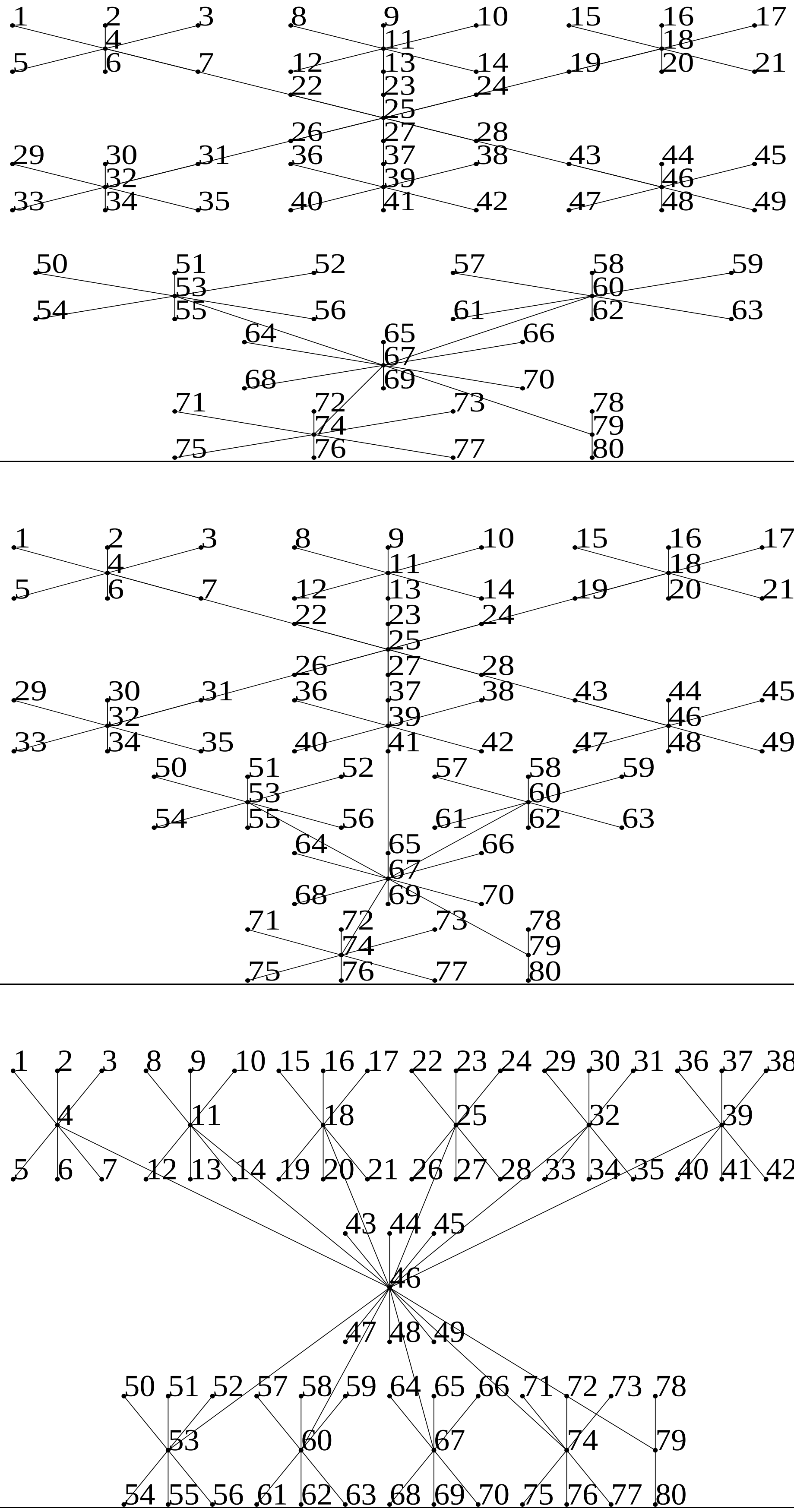
Dieser Algorithmus läßt sich noch verbessern, bis hinunter zu $T(n) = 3n$, nach einer Idee von Schönhage, Paterson, Pippenger: Bisher vergessen wir nämlich vor dem schließlichen rekursiven Aufruf wertvolle Information. (Suche das im Programmtext. Nämlich `++`.) Wir sollten uns irgendwie merken, welche Vergleiche schon stattgefunden haben. Wir betrachten Halbordnungen, die Binomialbäumen ähnlich sehen, siehe Literatur [May98].

Übung: wieviele Vergleiche führt der hier gezeigte Algorithmus wirklich aus?

Übung: Wieviel davon sind redundant? Implementierung: alle bereits gefragten Fragen aufheben und vor jeder neuen Frage nachschlagen, ob die Antwort bereits (wegen Transitivität) feststeht. Nur falls nein, dann wirklich fragen.

Übung: Diese Frage auch für andere Sortierverfahren (Mergesort, Heapsort). Welche der dabei festgestellten redundanten Fragen lassen sich (mit wieviel Aufwand) vermeiden?





Zwischenergebnisse bei `select 15 [1..80]` (ungefähr)

7.4.7 Untere Schranke für Median

(siehe [May98])

Wir sind Gegenspieler eines Medianalgorithmus für n Elemente (n ungerade) und bauen ihm eine Eingabe, für die er $\lceil 3n/2 \rceil - 2$ Vergleiche braucht.

Wir haben drei Mengen T, M, H (tief, mittel, hoch). Wir wünschen uns den Median in M. Am Anfang sind alle Elemente in M.

M soll möglichst ungeordnet bleiben: die Zusammenhangskomponenten haben höchstens Größe 2.

Während des Verfahrens geben wir Elemente aus M auf und bewegen sie nach T oder H.

Wir zählen $C :=$ Anzahl der Vergleiche (bisher), $P :=$ Anzahl der Paare (= Kanten) im aktuellen M.

Wir können so vorgehen, daß immer $C - P + 2\#M \geq 2n$. (Beweis durch Fallunterscheidung. Uns interessieren nur Fragen, bei denen beide Elemente in M sind.)

Das geht solange, bis $\#M \leq 2$ oder $\#H = (n-1)/2$ oder $\#T = (n-1)/2$.

Im ersten Fall sind wir fertig (bereits genügend Vergleiche ausgeführt): $C \geq 2n + P - 2\#M \geq 2n - 2 \cdot 2$.

Im zweiten Fall (o. B. d. A. # H = (n-1)/2) setzen wir so fort: wir zwingen den Algorithmus, das Minimum von M zu bestimmen. Dazu braucht er $\#M - 1$ Vergleiche, von diesen hat er schon P ausgeführt. Also sind noch $\#M - 1 - P$ nötig.

Insgesamt $C' \geq (2n + P - 2\#M) + (\#M - 1 - P) = 2n - \#M - 1$. Wir haben $\#M \leq n - \#H = (n+1)/2$, also $C \geq 2n - (n+1)/2 - 1 = 3n/2 - 3/2$, fertig.

Beispiel $n = 5$ ergibt $C' \geq 3 \cdot 5/2 - 2 = 5.5$. Das ist scharf.

7.5 Lösen von Rekurrenzgleichungen

Bei Kostenabschätzugen rekursiver Funktionen finden wir oft rekurrente (Un–)gleichungen. Im einfachsten Fall haben sie die Form

$$T(n) = a \cdot T(n/b) + f(n)$$

Die Lösungstheorie dazu steht ausführlich in [CLR91], wir wiederholen hier nur die Merkregel:

- Die Lösung T_0 der homogenen Gleichung (also für $f(n) = 0$) hängt nur vom Verhältnis a/b ab:
  - für $a/b < 1 - \epsilon$: $T_0(n) = \theta(\log_{b/a} n)$,
  - für $a/b \geq 1$: $T_0(n) = \theta(n^{a/b})$.
- Die Lösung der inhomogenen Gleichung (also für beliebige f) hängt vom (asymptotischen) Verhältnis von f zur homogenen Lösung T_0 ab:
  - für $f = o(T_0)$: $T(n) = \theta(T_0)$,
  - für $f = \theta(T_0)$: $T(n) = \theta(T_0 \cdot \log n)$,
  - für $f = \omega(T_0)$: $T(n) = \theta(f)$.

Das ist mit gewisser Vorsicht zu genießen, weil es nur für f gilt, die genügend glatt sind und nicht zu knapp neben $\theta(T_0)$ liegen (siehe Literatur und warnende Beispiele).

Übungsaufgaben, Teil a, aus [CLR91],

$$\begin{aligned}
T(n) &= 2T(n/2) + n^3 && (7.1)\\
T(n) &= T(9n/10) + n && (7.2)\\
T(n) &= 16T(n/4) + n^2 && (7.3)\\
T(n) &= 7T(n/3) + n^2 && (7.4)\\
T(n) &= 7T(n/2) + n^2 && (7.5)\\
T(n) &= 2T(n/4) + \sqrt{n} && (7.6)\\
T(n) &= T(n-1) + n && (7.7)\\
T(n) &= T(\sqrt{n}) + 1 && (7.8)
\end{aligned}$$

Übungsaufgaben, Teil b, aus [CLR91],

$$\begin{aligned}
T(n) &= 3T(n/2) + n \log n && (7.9)\\
T(n) &= 3T(n/3 + 5) + n/2 && (7.10)\\
T(n) &= 2T(n/2) + n/\log n && (7.11)\\
T(n) &= T(n-1) + 1/n && (7.12)\\
T(n) &= T(n-1) + \log n && (7.13)\\
T(n) &= \sqrt{n}T(\sqrt{n}) + n && (7.14)
\end{aligned}$$

Wir können es jedoch verifizieren für $f = n^c \log^d n$: Wir machen einfach den Ansatz $T = n^x \log^y n$ und bestimmen die Koeffizienten x und y (Übung. Summe einer geometrischen Reihe.)

Für unsere Analyse von Quicksort mit Medianbestimmung in Linearzeit haben wir die Gleichung

$$Q(n) = 2Q(n/2) + c \cdot n$$

mit der Lösung $Q = \Theta(n \log n)$. (Verifizieren wie eben.)

Eine leichte Variation ist die Gleichung, die sich beim Select ergibt:

$$T(n) = T(a \cdot n) + T(b \cdot n) + c \cdot n$$

Das Ergebnis hängt hier nur von $a + b$ ab. Wenn das < 1 ist, ergibt sich $\theta(n)$. Wir machen einen Ansatz $T(n) = F \cdot n$, um den Faktor F zu bestimmen:

$$\begin{aligned} T(a \cdot n) + T(b \cdot n) + c \cdot n = aFn + bFn + cn &= Fn \\ (a+b)F + c &= F \\ F &= \frac{c}{1 - (a+b)} \end{aligned}$$

Im dortigen Fall (für $a = 1/5, b = 7/10$) wird das recht knapp: $F = c \cdot 10$, aber Linearzeit ist Linearzeit.

Kapitel 8

Sortiernetze

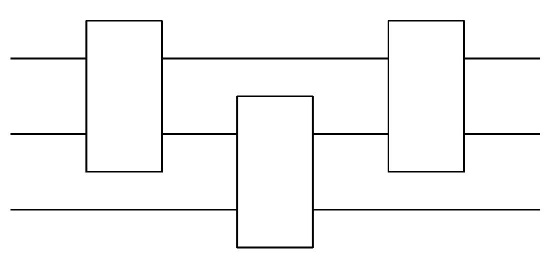
8.1 Einleitung

Wir sortieren Folgen von Zahlen, das heißt, wir suchen eine Funktion *sort* mit $sort(x_1, \dots, x_n) = (y_1, \dots, y_n) \iff y_1 \leq \dots \leq y_n$ und $(y_1, \dots, y_n)$ ist eine Permutation von $(x_1, \dots, x_n)$.

Wir führen das Sortieren aber nicht auf einem üblichen Rechner etwa mit einem PASCAL–Programm aus, sondern bauen ein Sortiernetz aus ganz einfachen Prozessoren. Jeder Prozessor kann zwei Zahlen sortieren. Er hat zwei Eingänge (e_1, e_2) und zwei Ausgänge (a_1, a_2). An a_1 erscheint $\max(e_1, e_2)$, an a_2 immer $\min(e_1, e_2)$.

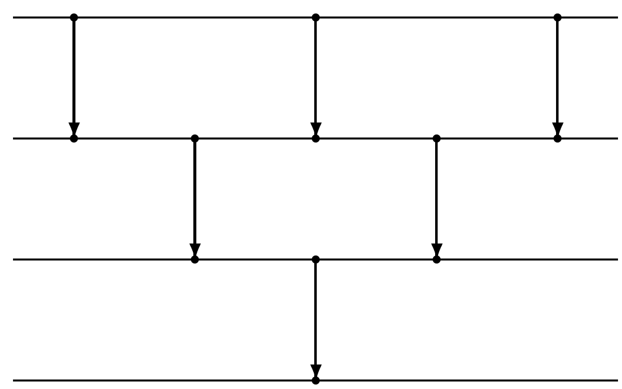
Ausgänge eines Prozessors können mit Eingängen eines anderen verbunden werden. Dabei darf keine Schleife entstehen, Verzweigungen sind ebenfalls nicht erlaubt. Die Zahlen fließen also von links nach rechts; da keine verlorengeht, liegt an den Ausgangsknoten immer eine Permutation der Eingangswerte.

Drei Zahlen lassen sich mit einem Netz aus drei Prozessoren sortieren:

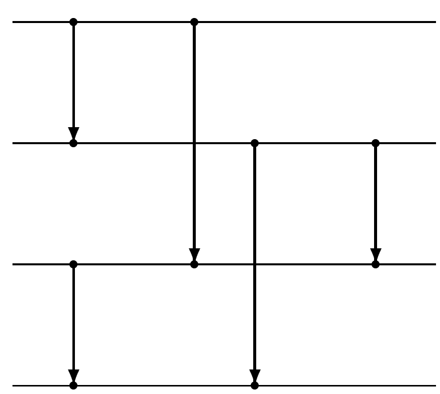


Der Beweis ist einfach: offenbar gelangt die größte Zahl ganz nach oben, die kleinste ganz nach unten. Für die dritte bleibt dann nur der mittlere Ausgang übrig.

Ein einfaches Sortiernetz für vier Eingänge ist



4 Zahlen lassen sich aber schon mit 5 Komparatoren vergleichen:



Informationstheoretische Schranken für beliebige Sortieralgorithmen gelten natürlich auch für Netze. Andererseits läßt sich nicht jedes Sortierverfahren als Netz ausdrücken. Ein Sortiernetz kann nicht während der Rechnung „weiter hinten“ Leitungen umbauen, sondern die Topologie (d. h. der Kontrollfluß des Programms) ist für alle Eingaben die gleiche.

Beispiel: wir können 5 Zahlen mit 7 Vergleichen ordnen, aber das kleinste 5–Sortiernetz braucht 8 Komparatoren.

8.2 Das 0–1–Prinzip

Satz: Wenn ein Netz alle {0,1}-Folgen richtig sortiert, dann sortiert es auch alle beliebigen Eingabefolgen.

Beweis: Sei f eine monotone Funktion: $a \leq b \Rightarrow f(a) \leq f(b)$. Wenn ein Netz x in y überführt, dann führt die Eingabe $f(x) = (f(x_1), \ldots, f(x_n))$ zur Ausgabe $f(y) = (f(y_1), \ldots, f(y_n))$. Nehmen wir an, N sortiert alle $\{0,1\}$-Folgen richtig, aber die Folge x führt zur falschen Ausgabe $y_1, \ldots, y_i > y_{i+1}, \ldots, y_n$. Wir wählen f als die Funktion, die allen Zahlen $< y_i$ die 0 zuordnet, und allen anderen Zahlen die 1. Dieses f ist monoton. Die Eingabe $f(x)$ ist eine $\{0,1\}$-Folge, sie erzeugt also $f(y) = (\ldots, f(y_i), f(y_{i+1}), \ldots) = (\ldots, 1, 0, \ldots)$. Das ist aber nicht korrekt sortiert.

8.3 Periodische Sortiernetze

Das folgend Bubblesort–ähnliche Verfahren hat Periode zwei:

```
for k := 1 to 4 do
    x1 <-> x2     x3 <-> x4     x5 <-> x6     x7
           x2 <-> x3     x4 <-> x5     x6 <-> x7
end
```
none

Dabei heißt `x3 <-> x4`: Komparator zwischen `x3` und `x4`.

Aufgabe: (für die offensichtliche Verallgemeinerung auf n Eingänge) wie oft muß die Schleife durchlaufen werden?

Etwas trickreicher ist das (Shear–Sort ??) wir ordnen die zu sortierenden Elemente in einer $w \times w$–Matrix an.

```
for k := 1 to ??? do
    sortiere alle ungeraden Zeilen nach rechts aufsteigend
        und alle geraden Zeilen nach links
    sortiere alle Spalten nach unten
```
none

Hier ist die Periodenlänge drei.

Ist das ein Sortierverfahren? Wie sieht das Resultat aus? Nach wieviel Schritten erreichen wir es?

Zu periodischen Netzen siehe Arbeiten von R. Wanka [Wan95].

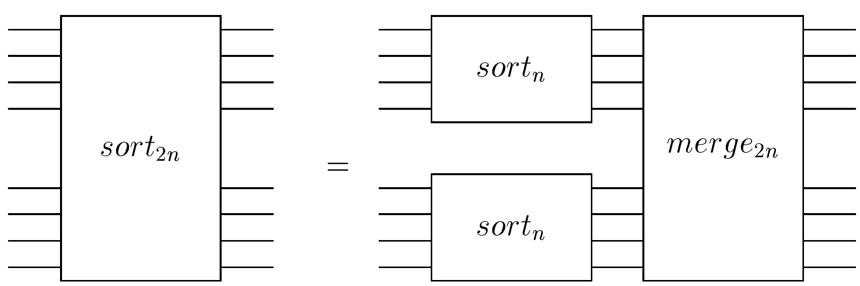
8.4 Odd–Even–Merge

Bevor wir nun beliebig viele Zahlen sortieren, betrachten wir eine damit zusammenhängende, aber einfachere Aufgabenstellung: wir mischen zwei *sortierte* Folgen zu einer einzigen: $merge(x_1, \ldots, x_n, y_1, \ldots, y_n) = (z_1, \ldots, z_{2n}) \iff x_1 \leq \ldots \leq x_n, y_1 \leq \ldots \leq y_n, z_1 \leq \ldots \leq z_{2n}$ und $(z_1, \ldots, z_{2n})$ ist eine Permutation von $(x_1, \ldots, x_n, y_1, \ldots, y_n)$.
Zum Beispiel ist $merge(2, 3, 6, 9,\ 1, 4, 5, 7) = (1, 2, 3, 4, 5, 6, 7, 9)$.

Wenn wir ein Verfahren für das Mischen haben, können wir auch sortieren: Wir halbieren die zu sortierende Folge, sortieren die Hälften und mischen die Ergebnisse. Die Rekursion endet, wenn die zu sortierenden Hälften nur noch ein Element enthalten; dann brauchen wir nicht zu mischen.

$$\begin{aligned} sort(x_1) &= (x_1) \\ sort(x_1, \ldots, x_{2n}) &= merge(sort(x_1, \ldots, x_n)\, sort(x_{n+1}, \ldots, x_{2n})) \end{aligned}$$

Als Netz sieht das so aus:



Sortieren wir $(9, 3, 2, 6, 7, 5, 4, 1)$ auf diese Weise:

$$\begin{aligned} & sort(9, 3, 2, 6, 7, 5, 4, 1) \\ = \;& merge(sort(9, 3, 2, 6), sort(7, 5, 4, 1)) \\ = \;& merge(merge(sort(9, 3), sort(2, 6)), merge(sort(7, 5), sort(4, 1))) \\ = \;& merge(merge(merge(9, 3), merge(2, 6)), \\ & \quad merge(merge(7, 5), merge(4, 1))) \\ = \;& merge(merge(3, 9,\ 2, 6), merge(5, 7,\ 1, 4)) \\ = \;& merge(2, 3, 6, 9,\ 1, 4, 5, 7) = (1, 2, 3, 4, 5, 6, 7, 9) \end{aligned}$$

Für das Mischen benutzen wir eine ähnliche Halbierungstechnik, die von Batcher vorgeschlagen wurde: um $merge(x_1, \ldots, x_n,\ y_1, \ldots, y_n) = (z_1, \ldots,\ z_{2n})$ zu erhalten, setzen wir, falls $n = 1$, einfach einen Prozessor ein, der $z = (\min(x_1, y_1), \max(x_1, y_1))$ liefert. Für $n > 1$ bestimmen wir

$$\begin{aligned} a &= merge(x_1, x_3, \ldots,\ y_1, y_3, \ldots) \\ b &= merge(x_2, x_4, \ldots,\ y_2, y_4, \ldots). \end{aligned}$$

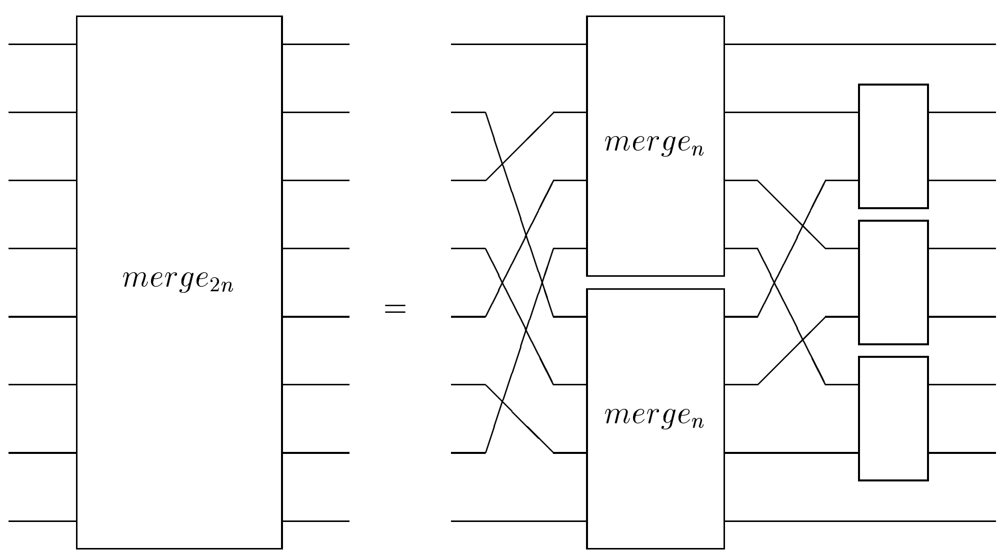
(Daher hat das Verfahren seinen Namen: Odd–Even–Merge.) Nun ist die Reißverschlußfolge $c = (a_1, b_1, a_2, b_2, \ldots, a_n, b_n)$ schon fast das gesuchte $(z_1, \ldots, z_n)$. Auf jeden Fall sind $z_1 = a_1$ und $z_{2n} = b_n$ (Übungsaufgabe!)

In unserem Beispiel: $x = (2, 3, 6, 9)$;$y = (1, 4, 5, 7)$; $a = merge(2, 6,\ 1, 5) = (1, 2, 5, 6)$; $b = merge(3, 9,\ 4, 7) = (3, 4, 7, 9)$; Das Zwischenergebnis ist $c = (1, 3, 2, 4, 5, 7, 6, 9)$. Das ist schon fast sortiert, wir müssen nur noch 3 mit 2 und 7 mit 6 vertauschen. Schlimmer wird es in keinem Fall, denn

$$z = (a_1, \min(a_2, b_1), \max(a_2, b_1), \min(a_3, b_2), \ldots, \max(a_n, b_{n-1}), b_n).$$

Zum Beweis nutzen wir das 0–1–Prinzip. Besteht nun x aus k Nullen und danach $n-k$ Einsen, y aus l Nullen und $n-l$ Einsen. Dann hat a genau $\lceil k/2 \rceil + \lceil l/2 \rceil$ Nullen (und danach Einsen), und b hat $\lfloor k/2 \rfloor + \lfloor l/2 \rfloor$ Nullen (und danach Einsen). Nun gilt aber $(\lceil k/2 \rceil + \lceil l/2 \rceil) - (\lfloor k/2 \rfloor + \lfloor l/2 \rfloor) \in \{0, 1, 2\}$. Ist die Differenz 0 oder 1, dann ist c schon sortiert; ist die Differenz 2, dann berichtigt der min–max–Tausch den Fehler.

Das Netz zum Mischen zweier sortierter Folgen ist damit:



8.5 Bitonisches Sortieren

Definition: Eine Zahlenfolge $(x_1, \ldots, x_n)$ heißt *bitonisch*, falls es einen Index m gibt, so daß $x_1 \geq \cdots \geq x_m \leq \cdots \leq x_n$.

Wir suchen eine Funktion **bisort**, die das bitonische x sortiert. Es ist klar, daß jede Teilfolge von x bitonisch ist; insbesondere die beiden

$$\begin{aligned} \mathbf{odd}(x) &= (x_1, x_3, x_5, \ldots) \\ \mathbf{even}(x) &= (x_2, x_4, x_6, \ldots) \end{aligned}$$

Wir bestimmen also

$$a = \mathbf{bisort}(\mathbf{odd}(x)), \quad b = \mathbf{bisort}(\mathbf{even}(x)).$$

Beispiel:

$$\begin{aligned} x = (7,5,1,2,4,6,8) \\ \mathbf{odd}(x) = (7,1,4,8), \quad \mathbf{even}(x) = (5,2,6) \\ a = \mathbf{bisort}(\mathbf{odd}(x)) = (1,4,7,8), \quad b = \mathbf{bisort}(\mathbf{even}(x)) = (2,5,6) \end{aligned}$$

Wir stellen fest, daß in diesem Beispiel

$$\mathbf{bisort}(x) = c = (a_1 : b_1) \circ (a_2 : b_2) \circ \cdots \circ a_{\lceil n/2 \rceil},$$

wobei der letzte Term auftrat, weil die Länge n von x ungerade war. Wir werden zeigen, daß die letzte Gleichung immer richtig ist. Nach dem 0–1–Prinzip genügt es, das für Eingaben zu zeigen, die nur aus 0 und 1 bestehen.

Wir setzen also x als eine bitonische $\{0,1\}$-Folge an:

$$x = (\underbrace{1,\ldots,1}_{k \text{ mal}}, \underbrace{0,\ldots,0}_{(l-k) \text{ mal}}, \underbrace{1,\ldots,1}_{(n-l) \text{ mal}})$$

Dann gilt

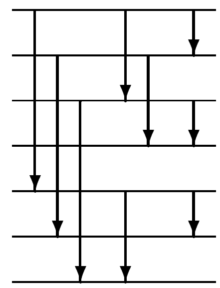
$$\begin{aligned} \mathbf{odd}(x) &= (\underbrace{1,\ldots,1}_{\lceil k/2 \rceil \text{ mal}}, \underbrace{0,\ldots,0}_{\lceil l/2 \rceil - \lceil k/2 \rceil \text{ mal}}, 1,\ldots,1) \\ a = \mathbf{bisort}(\mathbf{odd}(x)) &= (\underbrace{0,\ldots,0}_{\lceil l/2 \rceil - \lceil k/2 \rceil \text{ mal}}, 1,\ldots,1) \\ \mathbf{even}(x) &= (\underbrace{1,\ldots,1}_{\lfloor k/2 \rfloor \text{ mal}}, \underbrace{0,\ldots,0}_{\lfloor l/2 \rfloor - \lfloor k/2 \rfloor \text{ mal}}, 1,\ldots,1) \\ b = \mathbf{bisort}(\mathbf{odd}(x)) &= (\underbrace{0,\ldots,0}_{\lfloor l/2 \rfloor - \lfloor k/2 \rfloor \text{ mal}}, 1,\ldots,1) \end{aligned}$$

Es gilt jedoch

$$(\lceil l/2 \rceil - \lceil k/2 \rceil) - (\lfloor l/2 \rfloor - \lfloor k/2 \rfloor) \in \{-1, 0, +1\},$$

so daß der Fehler durch die Vertauschungen in c behoben werden kann—**bisort** ist korrekt.

Das Sortiernetz für bitonische Folgen der Länge 7 ist:



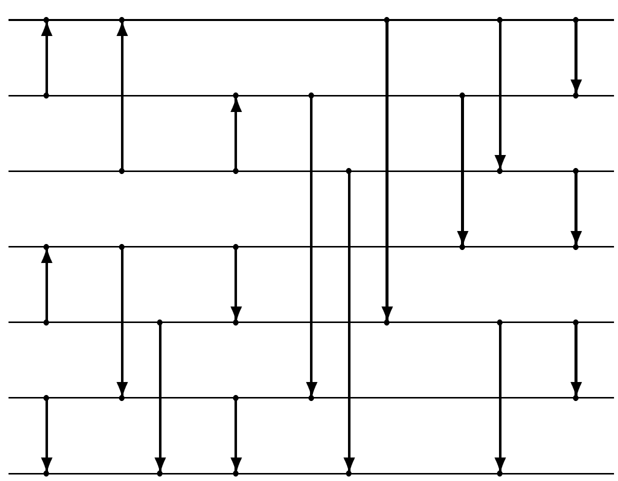
Mit **bisort** können wir **merge** für sortierte Folgen implementieren: wir ordnen die eine ab-, die andere aufsteigend an und hängen sie aneinander. Das Resultat ist bitonisch und kann sortiert werden.

$$\mathbf{merge}(x, y) = \mathbf{bisort}(\overline{x} \circ y).$$

Ausgestattet mit **merge** läßt sich nun eine beliebige Folge in bekannter Weise sortieren:

$$\mathbf{sort}(x) = \mathbf{merge}(\mathbf{sort}(x_1, \ldots, x_{\lfloor n/2 \rfloor}), \mathbf{sort}(x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n)).$$

Das Sortiernetz für beliebige 7 Eingänge ist demnach:



8.6 Optimale Netze

Sowohl bitonic als auch odd–even haben Tiefe $\theta(\log^2 n)$, Breite n, also Größe $\theta(n \log^2 n)$.

Die informationstheoretische Schranke für Sortieren ist jedoch $\Omega(n \log n)$. Gibt es ein Sortiernetz der Tiefe $\theta(\log n)$?

Man kann zeigen, daß jedes Merge–Netz für n Eingänge bereits Tiefe $\log n$ braucht. Also wird Mergesort nicht besser als bereits beschrieben.

Eine andere Variante ist Quicksort: erst Partitionieren, dann die Hälften getrennt sortieren. Es ist leider so, daß man auch zum Partitionieren $\Omega(\log n)$ Tiefe braucht.

Es war darum eine Überraschung, daß Ajtai, Komlos, Szemeredi 1983 tatsächlich ein Sortiernetz mit $\theta(\log n)$ Tiefe konstruierten.

Der entscheidende Gedanke ist die Verwendung von ϵ–Partitionierern. Das sind Partitionierer, die einen relativen Fehler von ϵ machen dürfen, z. b. $\epsilon = 1/128$. Das heißt, sie teilen die Eingabe von n Elementen in eine untere und eine obere Hälfte, aber im unteren Teil sind bis zu $\epsilon \cdot n$ Elemente falsch (und im oberen dann auch) — und wir wissen nicht, wo die falschen liegen.

Man muß zeigen

- es gibt ϵ–Partitionierer mit konstanter Tiefe
- man kann Quicksort mit 1/128–Partitionierern implementieren.

Zum ersten Punkt: wir gewinnen diese Partitionier–Netze aus speziellen Graphen: Superkonzentratoren bzw. Expandern. (Die haben wenige Kanten, aber schaffen trotzdem viele Verbindungen.) Es ist zunächst gar nicht klar, daß genüngend sparsame Expander existieren. Wir können sie uns auf zwei Arten besorgen:

- mit einem probabilistischen Argument (das aber nicht konstruktiv ist)
- durch explizite Konstruktion mit algebraischen (!) Hilfsmitteln

Zum zweiten Punkt: da die Partitionierer Fehler machen, schicken wir Teile (genauer: Ränder) der "sortierten" Hälften nochmals zurück. Das organisieren wir so, daß alle Ausreißer die Chance bekommen, an ihren eigentlichen Platz zurückzukehren. Tatsächlich dürfen wir im Netz nichts zurückschicken, sondern müssen Teile des Netzes eben nocheinmal aufmalen. Dabei soll natürlich die resultierende Netztiefe trotzdem nur $\theta(\log n)$ bleiben.

Für eine kompakte Darstellung siehe [Pip90] .

Literaturverzeichnis

[CLR91] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1991. http://theory.lcs.mit.edu/~clr/.

[Har97] Rolf Hartwig. Algebraische Spezifikation abstrakter Datentypen (Skript zur Vorlesung). http://www.informatik.uni-leipzig.de/~rhartwig/lad/skr4.html, 1997.

[Har98] Rolf Hartwig. Grundlagen der Programmverifikation (Skript zur Vorlesung). http://www.informatik.uni-leipzig.de/~rhartwig/lad/skr1.html, 1998.

[Hin98] Ralf Hinze. Numerical representations as higher order nested datatypes. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn, 1998. http://www.informatik.uni-bonn.de/~ralf/HigherOrderNested.ps.gz.

[Hin99] Ralf Hinze. Explaining binomial heaps. *Journal of Functional Programming*, 9(1):93–104, 1999. http://www.informatik.uni-bonn.de/~ralf/publications.html.

[Kil99] Lee Killough. Priority queues. http://priority-queues.webhop.org/, 1999.

[Knu93] Donald Erwin Knuth. *The Stanford Graphbase: A Platform for Combinatorial Computing*. ACM Press, New York, 1993. ftp://labrea.stanford.edu/pub/sgb/.

[Knu98] Donald Erwin Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1998. http://www-cs-staff.stanford.edu/~uno/taocp.html.

[May98] Ernst W. Mayr. Effiziente Algorithmen und Datenstrukturen (Skript zur Vorlesung). http://www14.informatik.tu-muenchen.de/skripten/ead_ws9899_html/index.h%tml, 1998.

[MT90] K. Mehlhorn and A. Tsakalidis. *Data Structures*, pages 301–342. Volume A of van Leeuwen [vL90b], 1990.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. http://www.cs.columbia.edu/~cdo/papers.html#cup98.

[Pip90] N. Pippenger. *Communication Networks*, pages 805–834. Volume A of van Leeuwen [vL90b], 1990.

[Sho92] Berry Shoemakers. *Data structures and Amortized Complexity in a Functional Setting.* PhD thesis, Eindhoven University of Technology, 1992. http://www.win.tue.nl/math/dw/pp/berry/phd.html.

[VF90] J. S. Vitter and Ph. Flajolet. *Average-Case ananlysis of Algorithms and Data Structures*, pages 431–524. Volume A of van Leeuwen [vL90b], 1990.

[vL90a] J. van Leeuwen. *Graph Algorithms*, pages 525–632. Volume A of van Leeuwen [vL90b], 1990.

[vL90b] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*. Elsevier Science Publishers B.V., 1990.

[Wan95] Rolf Wanka. Parallele Sortiernetzwerke (Skript zur Vorlesung). http://www.uni-paderborn.de/fachbereich/AG/agmadh/WWW/german/RolfsVorle%sungen/ParSort/parsort.html, 1995.

[Wei93] Mark Allen Weiss. *Data Structures and Algorithms in Ada*. Benjamin/Cummings, 1993. http://www.cs.fiu.edu/~weiss/.

Kapitel 9

Plan der Vorlesung

Vorlesung vom 15. 10. 99

- Beispiel Musik-Sequenzer, event driven simulation, als Motivation für Priority Queues
- Aufgabe: `pool.zip`
- Signatur für PQs, Begriff ADT
- Idee der Implementierung von PQs durch binäre Bäume
- Definition leftist heap, Satz: logarithmische Rechtstiefe
- Aufgabe: vergleiche leftist und `ldepth t >= rdepth t`

Vorlesung vom 22. 10. 99

- Erwärmung: leftist heaps erkennen
- Aufgaben zu leftist besprechen
- `union` für leftist heaps
- `extract` auf `union` zurückführen
- Mehrwegbäume, generische Definitionen von `size`, `depth`, `subtrees`
- Binomialbäume, Satz: `size t = 2 ** (depth t - 1)`
- Aufgabe: in Entfernung d von Wurzel sind $\binom{\texttt{depth t - 1}}{d}$ Knoten
- Binomialwälder (heaps), Eindeutigkeit der Zerlegung in Zweierpotenzen
- Operation `find_min`

Vorlesung vom 27. 10. 99

- Operation `link` für gleichgroße Binomial-Bäume
- Operationen für Binomialheaps: `merge`
- Aufgabe: Zeit für `merge`
- Lösung der Aufgabe `pool.zip`: Operation `delete`
- `delete` und `decrease-key` in heaps
- Aufgabe: `decrease_key` in leftist Heaps
- Anwendung von PQs bei Kruskal (MST) (sehr kurz und knapp)
- Amortisierte Analyse (Beispiel Inkrementieren)

Vorlesung vom 3. 11. 99

- Amortisierte Analyse (Definition, Konto, Potential)
- Skew Heaps

Vorlesung vom 10. 11. 99

- Fibonacci-Heaps

Vorlesung vom 24. 11. 99

- ADT FiniteMap (Wörterbuch)
- Implementierung als Array, als Hashtabelle
- Suchbäume
- Implementierung durch (a, b)–Bäume
- Wiederholung AVL, Zusammenhang mit (2,3)–Bäumen
- Definition Rot-Schwarz-Bäume, Zusammenhang mit (2,4)–Bäumen

---

Vorlesung vom 1. 12. 99

Gastvortrag von Dr. Ralf Hinze, Bonn

- Rot–Schwarz–Bäume
- Zahlensysteme

Vorlesung vom 8. 12. 99

- gewichtsbalancierte Bäume (lokales rebuild, globales rebuild und amortisierte Analyse)
- Splay–Trees (amortisierte Analyse)

Vorlesung vom 15. 12. 99

- Pairing–Heaps
- Treaps **Ab hier gehts nach dem Buch von Knuth. Deswegen gibts im wesentlichen kein Skript.**
- informationstheoretische Schranke für Sortieren
- Definition und einfache Beispiele für Sortiernetze
- Merge Insertion als optimales Sortierverfahren für 5 Elemente

Vorlesung vom 5. 1.

- Merge Insertion für $5, 10, 21$ Elemente
- Halbordnungen, Anzahl der Linearisierungen
- verfeinerte untere Schranken durch Entscheidungsbäume

Vorlesung vom 12. 1.

- untere Schranke für Minimum (trivial)
- untere Schranke für zweitbesten (durch Gegenspieler)
- tree selection
- replacement selection (optimal für $V_3(5)$)

Vorlesung vom 19. 1.

- untere Schranke für Median
- Median in Linearzeit (rekursiv)

Vorlesung vom 26. 1.

- Sortiernetze: Monotonie, $0 - -1$–Prinzip,
- odd–even–sort

Vorlesung vom 2. 2.

- bitonic sort
- AKS–Netz