29. Knobelaufgabe #14

Harry Hacker behauptet eine Funktion definiert zu haben, die einen Binärbaum der Größe n in logarithmischer (!) Zeit konstruiert. Seine Funktion erfüllt die Eigenschaft

$$size\ (create\ n) = n$$

wobei *size* wie folgt definiert ist:

```
let rec size = function
  | Leaf          -> 0
  | Node (l, a, r) -> size l + 1 + size r
```

Genie oder Scharlatan?

29. Mengen und endliche Abbildungen

Zurück zum *Mini Softwareprojekt:* Verwaltung von Personaldaten eines Unternehmens.

Weitere typische Aufgaben:

- Personen zum Personalstamm hinzufügen;
- Personen aus dem Personalstamm entfernen.

☞ Wir realisieren im Prinzip eine *endliche Abbildung* von Personalnummern auf Personendaten.

Ambitionierter: wir abstrahieren von unserer Anwendung und programmieren endliche Abbildungen (wir internalisieren $A \rightarrow_{\mathrm{fin}} B$).

29. Abstrakte Datentypen

- Ein Typ mit einer zugehörigen Menge von Operationen heißt *abstrakter Datentyp* (ADT).
- Zu einem abstrakten Datentyp gehört
  - eine *Schnittstelle* (engl. interface), die die verfügbaren Typen und Operationen beschreibt („was“), und
  - eine oder mehrere *Implementierungen*, die die Typen und Operationen realisieren („wie“).
- *Idee:* Implementierungsdetails werden vor den Klienten verborgen.
- *Vorteil:* Implementierungen lassen sich einfach austauschen — ein Gewinn an Modularität.
- (Good SE practice: programming against an interface.)
- konkrete versus abstrakte Datentypen:
  - ein konkreter Datentyp wird durch seine Elemente definiert;
  - ein abstrakter Datentyp wird durch seine Operationen definiert.

29. Abstrakter Datentyp: endliche Menge

Statt endlicher Abbildungen behandeln wir den konzeptionell etwas einfacheren ADT „endliche Menge“ — endliche Abbildungen im Skript.

Schnittstelle:

type $Set\ \langle 'elem$ ***when*** $'elem : comparison \rangle$

val *empty* : $Set\ \langle 'elem \rangle$
val *add* : $'elem * Set\ \langle 'elem \rangle \to Set\ \langle 'elem \rangle$
val *remove* : $'elem * Set\ \langle 'elem \rangle \to Set\ \langle 'elem \rangle$
val *is-empty* : $Set\ \langle 'elem \rangle \to Bool$
val *contains* : $'elem * Set\ \langle 'elem \rangle \to Bool$
val *from-list* : $List\ \langle 'elem \rangle \to Set\ \langle 'elem \rangle$
val *to-list* : $Set\ \langle 'elem \rangle \to List\ \langle 'elem \rangle$

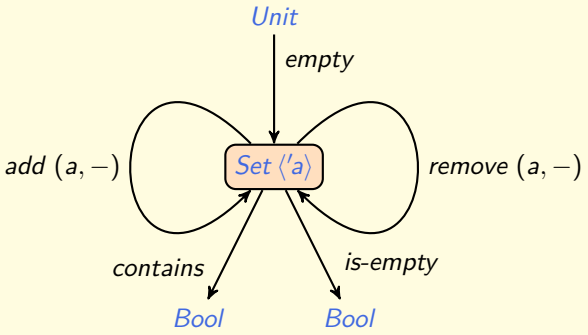
☞ Mengen sind wie Listen Containertypen; der Zusatz ***when*** $'elem : comparison$ schränkt den Elementtyp auf Typen ein, die die Vergleichsoperationen $\leqslant$, $<$ etc unterstützen.

Mathematikbrille: $\emptyset$, $\{a\} \cup A$, $A \setminus \{a\}$, $A = \emptyset$, $a \in A$.

29. Abstrakte Datentypen: Operationen

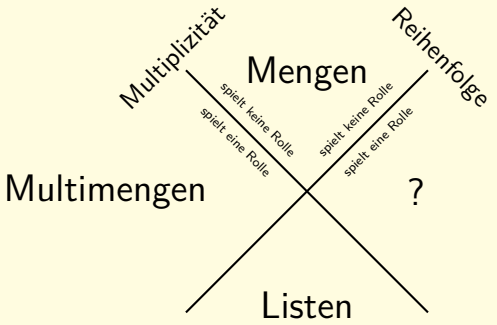
Plausibilitätsprüfung: ein ADT sollte Operationen bereitstellen

- um Elemente des ADTs zu *konstruieren*;
- um Elemente des ADTs in andere Elemente zu *transformieren*;
- um Elemente des ADTs zu *analysieren*.



29. Boom Hierarchie

- Mengen, Multimengen (engl. bags) und Listen sind Containertypen (engl. container types, collection types).
- Eine Menge ist eine ungeordnete Sammlung von Elementen; auch spielt die Multiplizität von Elementen keine Rolle.
- Eine Multimenge ist eine ungeordnete Sammlung von Elementen.
- Eine Liste ist eine geordnete Sammlung von Elementen.



29. Boom Hierarchie

- Die Mengen $\{b; o; m\}$, $\{m; o; b\}$ und $\{b; o; o; m\}$ sind gleich.
- Die Multimengen $\wr b; o; m \int$ und $\wr m; o; b \int$ sind gleich; die Multimengen $\wr b; o; m \int$ und $\wr b; o; o; m \int$ sind verschieden.
- Die Listen $[b; o; m]$, $[m; o; b]$ und $[b; o; o; m]$ sind alle verschieden.
- ACI Eigenschaften:
  - *assoziativ:* $(a + b) + c = a + (b + c)$;
  - *kommutativ:* $a + b = b + a$ (Reihenfolge spielt keine Rolle);
  - *idempotent:* $a + a = a$ (Multiplizität spielt keine Rolle).
- Mengen-, Multimengen- und Listenoperationen unterscheiden sich in den Gesetzen, die sie erfüllen:

	A	C	I
Mengenvereinigung $\cup$	ja	ja	ja
Multimengenvereinigung $\uplus$	ja	ja	nein
Listenkonkatenation @	ja	nein	nein

29. Endliche Mengen: Implementierung — Listen

Implementierung 1: wir repräsentieren endliche Mengen durch ungeordnete Listen, die möglicherweise Duplikate enthalten.

Der Containertyp *Set* wird durch einen Variantentyp mit einer einzigen Varianten implementiert (siehe auch Folie 316).

```
type Set ⟨'elem when 'elem : comparison⟩ =
    | Rep of List ⟨'elem⟩
let empty = Rep []
let add (key, Rep list) = Rep (key :: list)
```

☞ Der Konstruktor *Rep* überführt die Repräsentation einer Menge, eine Liste, in den abstrakten Typ der Menge.

29. Endliche Mengen: Implementierung — Listen

```
let remove (key, Rep list) =
  let rec del = function
   | []      → []
   | x :: xs → if key = x then del xs else x :: del xs
  in Rep (del list)
```

☞ Das zu löschende Element kann mehrfach vorkommen; wir müssen alle Vorkommen entfernen.

29. Endliche Mengen: Implementierung — Listen

```
let is-empty (Rep list) = List.is-empty list

let contains (key, Rep list) =
  let rec find = function
  | []      → false
  | x :: xs → key = x || find xs
  in find list

let from-list list = Rep list

let to-list (Rep list) = list
```

29. Endliche Mengen: Implementierung—Suchlisten

Implementierung 2: wir repräsentieren endliche Mengen durch aufsteigend geordnete Listen, die keine Duplikate enthalten.

Kurz: durch Suchlisten (das ist kein etablierter Begriff).

```
type Set ⟨'elem when 'elem : comparison⟩ =
    | Rep of List ⟨'elem⟩

let empty = Rep []

let add (key, Rep list) =
  let rec ins = function
    | []      → [key]
    | x :: xs → if    key < x   then key :: x :: xs
                elif  key = x   then key :: xs
                (* key > x *)  else  x :: ins xs
  in Rep (ins list)
```

☞ Beim Einfügen wird ein bereits vorhandenes Element „überschrieben“.

29. Endliche Mengen: Implementierung—Suchlisten


```
let remove (key, Rep list) =
  let rec del = function
    | []      → []
    | x :: xs → if    key < x   then x :: xs
                elif  key = x   then xs
                (* key > x *)  else  x :: del xs
  in Rep (del list)
```

☞ Es muss höchstens ein Element entfernt werden, da die Listen keine Duplikate enthalten.

29. Endliche Mengen: Implementierung—Suchlisten

```
let is-empty (Rep list) = List.is-empty list

let contains (key, Rep list) =
  let rec find = function
  | []      → false
  | x :: xs → key = x || key > x && find xs
  in find list

let from-list list = Rep (List.distinct (List.sort list))

let to-list (Rep list) = list
```

29. Endliche Mengen: Implementierung—Suchbäume

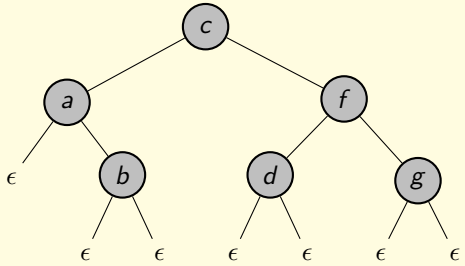
Implementierung 3: wir repräsentieren endliche Mengen durch binäre Suchbäume, die keine Duplikate enthalten.

```
type Tree ⟨'a⟩ =
    | Leaf
    | Node of Tree ⟨'a⟩ * 'a * Tree ⟨'a⟩
```

Schauen wir uns Binärbäume und binäre Suchbäume noch einmal in Ruhe an ...

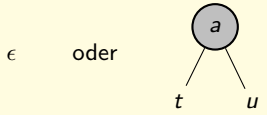
29. Binärbäume

- Der Rekursionsbaum der binären Suche ist ein *Binärbaum*.
- *Idee:* eine *Kontrollstruktur wird zu einer Datenstruktur*, so dass das Leibiz Entwurfsmuster zum Struktur Entwurfsmuster wird.



29. Binärbäume: Definition

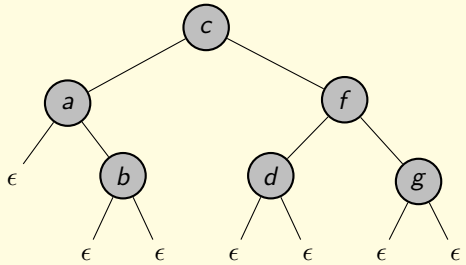
- Ein *binärer Baum* ist entweder
  - leer oder
  - ein Knoten, der aus einem linken Baum, einem Element und einem rechten Baum besteht.



- Leere Bäume heißen auch *Blätter*.
- Der linke und der rechte Baum sind *Teilbäume*.

29. Binärbäume: Begriffe

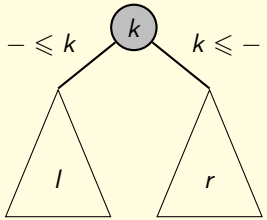
- In der Informatik wachsen Bäume typischerweise von oben nach unten; Bäume werden mit der Wurzel nach oben gezeichnet.



- Der Knoten c ist die *Wurzel*; die leeren Teilbäume sind die *Blätter*.
- Alle Knoten, mit Ausnahme der Wurzel, haben einen *Vorgänger*.
- d's Vorgänger ist f; f's Vorgänger ist c.
- Knoten können *Kinder* haben.
- f hat die Kinder d und g; d hat keine Kinder; a hat ein Kind.
- a und f sind *Geschwister*; d und g sind *Geschwister*.

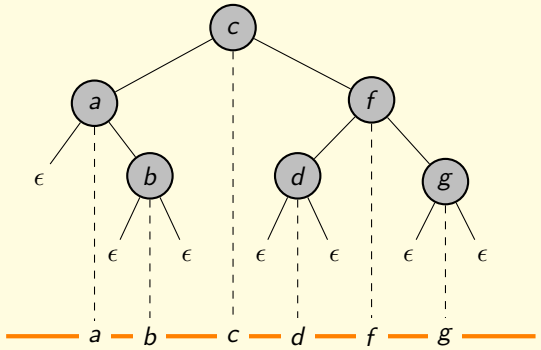
29. Binäre Suchbäume

- Ein *binärer Suchbaum* ist ein binärer Baum, so dass
  - der linke Teilbaum jedes Knotens nur Elemente enthält, die gleich oder kleiner sind als das Element in dem Knoten selbst;
  - der rechte Teilbaum jedes Knotens nur Elemente enthält, die gleich oder größer sind als das Element in dem Knoten selbst.



29. Binäre Suchbäume: Beispiel

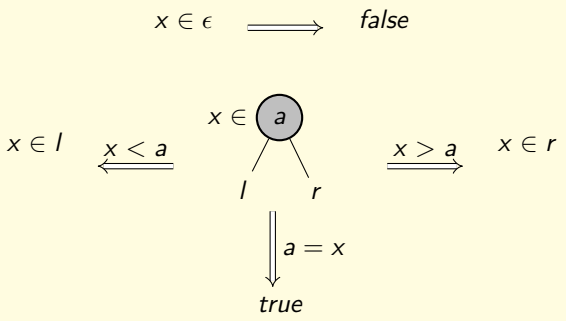
- Unser laufendes Beispiel ist ein binärer Suchbaum:



- Wenn wir die Elemente auf eine horizontale Linie projizieren, dann erhalten wir eine aufsteigend geordnete Sequenz.

29. Binäre Suchbäume: Suche

- Suchen eines Elements in einem binären Suchbaum mit einem 3-Wege Vergleich:



- Das Wurzelelement dient als Wegweiser.

29. Binäre Suchbäume: Suche

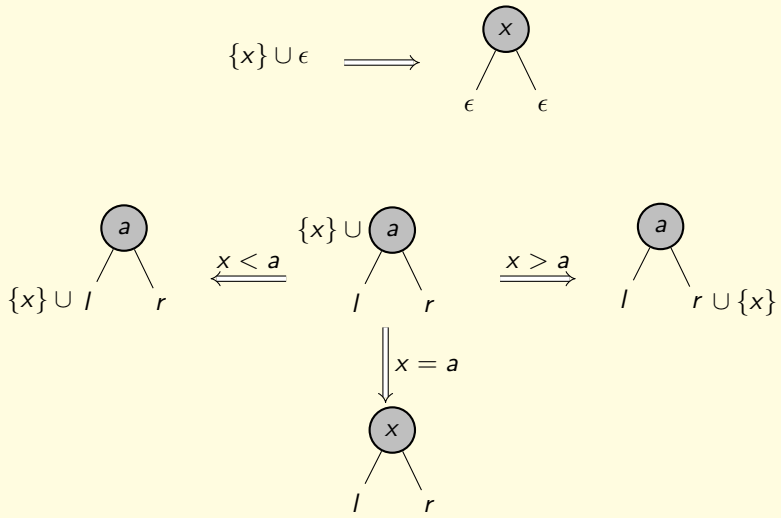
Der 3-Wege Vergleich wird mit einer geschachtelten Fallunterscheidung realisiert:

```
let contains key = function
    | Leaf           → false
    | Node (l, x, r) → if   key < x   then contains key l
                       elif key = x   then true
                       (* key > x *) else  contains key r
```

☞ *contains* ist die Mutter aller Algorithmen auf Suchbäumen.

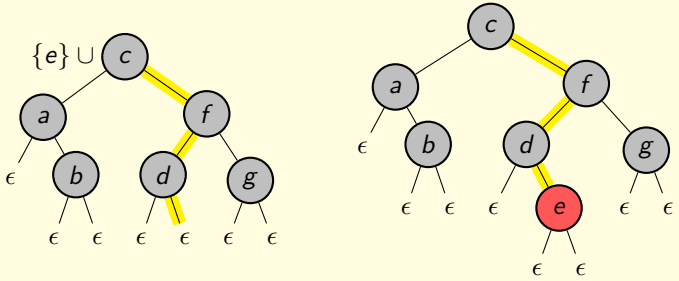
29. Binäre Suchbäume: Einfügen

Das Einfügen verwendet das gleiche Rekursionsmuster wie das Suchen.



29. Binäre Suchbäume: Einfügen — Beispiel

- Wir fügen e ein:



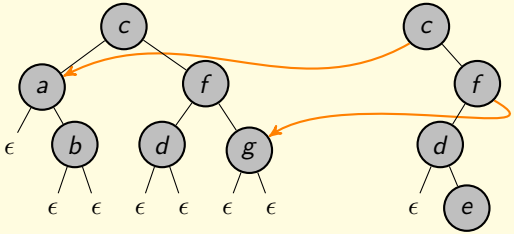
- Das neue Element e ersetzt einen leeren Teilbaum.

29. Binäre Suchbäume: Einfügen

```
let rec insert key = function
    | Leaf           → Node (Leaf, key, Leaf)
    | Node (l, x, r) → if    key < x   then Node (insert key l, x, r)
                       elif  key = x   then Node (l, key, r)
                       (* key > x *)  else  Node (l, x, insert key r)
```

29. Einfügen — „sharing“

- Die Knoten entlang des Suchpfades werden neu angelegt.



- Die Teilbäume, die nicht traversiert werden, werden „geteilt“:
  - sie sind sowohl Teilbäume im ursprünglichen Baum (Eingabe)
  - als auch im erweiterten Baum (Ausgabe).
- (im Englischen spricht man von „sharing“)

29. Endliche Mengen: Implementierung—Suchbäume

Zur Erinnerung:

```
type Tree ⟨'a⟩ =
    | Leaf
    | Node of Tree ⟨'a⟩ * 'a * Tree ⟨'a⟩
```

Implementierung der Schnittstelle:

```
type Set ⟨'elem when 'elem : comparison⟩ =
    | Rep of Tree ⟨'elem⟩

let empty = Rep Leaf

let add (key, Rep tree) =
  let rec ins = function
    | Leaf           → Node (Leaf, key, Leaf)
    | Node (l, x, r) → if   key < x   then Node (ins l, x, r)
                       elif key = x   then Node (l, key, r)
                       (* key > x *) else  Node (l, x, ins r)
  in Rep (ins tree)
```

29. Endliche Mengen: Implementierung—Suchbäume

```
let is-empty (Rep tree) =
  match tree with
  | Leaf              → true
  | Node (_, _, _)    → false

let contains (key, Rep tree) =
  let rec find = function
  | Leaf              → false
  | Node (l, x, r)    → if    key < x    then find l
                        elif  key = x    then true
                        (* key > x *)   else  find r
  in find tree

let from-list list = Rep (balanced-tree (List.distinct (List.sort list)))

let to-list (Rep tree) = inorder tree
```

☞ Löschen von Elementen zur Übung (siehe auch Skript).

☞ *balanced-tree* und *inorder* gehen wir als nächstes an ...

29. Wechsel der Repräsentation

Wir haben drei Implementierungen des ADTs „endliche Menge“ kennengelernt: Listen, Suchlisten und Suchbäume. Qual der Wahl?

Wunsch: die verschiedenen Repräsentationen einer Menge ineinander zu überführen (zu konvertieren).

Als nächstes beschäftigen wir uns mit der Konstruktion und der Linearisierung von Binärbaumen.

☞ Lässt sich ein Suchbaum in linearer Zeit konstruieren?

29. Konstruktion eines Binärbaums

Aufgabe: Konstruktion eines balancierten Suchbaums aus einer geordneten Liste.

Idee: wir orientieren uns an der Struktur eines Suchbaums:

- *Rekursionsbasis*: Ist die Liste leer, so geben wir den leeren Suchbaum zurück.
- *Rekursionsschritt*: Eine mindestens einelementige Liste teilen wir in drei Teile auf, den linken Teil, das Wurzelelement und den rechten Teil. Um die Ausgeglichenheit des Suchbaums zu gewährleisten, müssen die beiden Teillisten möglichst gleich lang sein.
- *Teilaufgabe:* halbieren einer Liste.

29. Halbierung einer Liste

Spezifikation: wir suchen *eine* Umkehrfunktion der Listenkonkatenation.

$$x \mathbin{@} y = z \quad \text{wobei} \quad (x, y) = \mathit{halve}\ z$$

☞ Die Funktion *unzip* lässt sich somit nicht verwenden.

Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

```
let rec halve (list : List ⟨'a⟩) : List ⟨'a⟩ * List ⟨'a⟩ =
  match list with
  | []      → ...
  | x :: xs → ... halve xs ...
```

29. Halbierung einer Liste

- *Rekursionsbasis*:

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{halve}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle) : \mathit{List}\ \langle'a\rangle * \mathit{List}\ \langle'a\rangle = \\ \quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\ \quad \mid [] \quad\ \to ([], []) \\ \quad \mid x :: xs \to \ \ldots \mathit{halve}\ xs \ldots\end{array}$$

- *Rekursionsschritt*: Wie können wir aus der halbierten Restliste eine halbierte Liste konstruieren?

$$\begin{array}{l}\textbf{\textit{let rec}}\ \mathit{halve}\ (\mathit{list} : \mathit{List}\ \langle'a\rangle) : \mathit{List}\ \langle'a\rangle * \mathit{List}\ \langle'a\rangle = \\ \quad \textbf{\textit{match}}\ \mathit{list}\ \textbf{\textit{with}} \\ \quad \mid [] \quad\ \to ([], []) \\ \quad \mid x :: xs \to \textbf{\textit{let}}\ (xs_1, xs_2) = \mathit{halve}\ xs\ \textbf{\textit{in}} \ldots\end{array}$$

Das Kopfelement x muss vor xs_1 gesetzt werden; eventuell muss das letzte Element von xs_1 zu xs_2 verschoben werden.
☞ Machbar, aber langsam!

29. Aufteilung einer Liste

Wir *verallgemeinern* die Aufgabe: eine Liste *list* wird in zwei Teillisten der Längen n und $len - n$ zerteilt, wobei *len* die Länge der Liste *list* ist.

☞ Halbierung ist dann ein Spezialfall mit $n = len \div 2$.

Jetzt kommt das Peano Entwurfsmuster zum Einsatz.

let rec *split* $(n : Nat, list : List\ \langle'a\rangle) : List\ \langle'a\rangle * List\ \langle'a\rangle =$
  if $n = 0$ ***then*** $\ldots$
    else $\ldots split\ (n \div 1, \ldots) \ldots$

29. Aufteilung einer Liste

- *Rekursionsbasis*: die erste Teilliste ist leer.

$$\begin{array}{l}\textbf{let rec } split\ (n : Nat, list : List\ \langle'a\rangle) : List\ \langle'a\rangle * List\ \langle'a\rangle = \\ \quad \textbf{if } n = 0 \textbf{ then } ([], list) \\ \qquad\qquad \textbf{else } \ldots split\ (n \dot{-} 1, \ldots) \ldots\end{array}$$

- *Rekursionsschritt*: Wir machen zusätzlich eine Fallunterscheidung über das Listenargument.

$$\begin{array}{l}\textbf{let rec } split\ (n : Nat, list : List\ \langle'a\rangle) : List\ \langle'a\rangle * List\ \langle'a\rangle = \\ \quad \textbf{if } n = 0 \\ \quad \textbf{then } ([], list) \\ \quad \textbf{else } \textbf{match } list \textbf{ with} \\ \qquad\quad | \ [] \qquad\ \to ([], []) \\ \qquad\quad | \ x :: xs \to \textbf{let } (xs_1, xs_2) = split\ (n \dot{-} 1, xs) \\ \qquad\qquad\qquad\quad\ \ (x :: xs_1, xs_2)\end{array}$$

29. Konstruktion eines Binärbaums

Zurück zur ursprünglichen Aufgabe: der Konstruktion eines balancierten Suchbaums aus einer geordneten Liste.

Wir verwenden das verallgemeinerte Leibniz Entwurfsmuster — wir „kämpfen“ gegen die Struktur von Listen an.

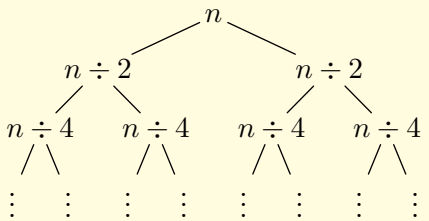
```
let rec balanced-tree (list : List ⟨'a⟩) : Tree ⟨'a⟩ =
  let n = length list
  if n = 0 then Leaf
           else let (xs₁, x :: xs₂) = split (n ÷ 2, list)
                Node (balanced-tree xs₁, x, balanced-tree xs₂)
```

☞ Der Abgleich mit dem Muster $(xs_1, x :: xs_2)$ kann nicht scheitern. (Warum?)

29. Konstruktion eines Binärbaums

☞ Welche Laufzeit hat *balanced-tree*?

Rekursive Aufrufstruktur von *balanced-tree* in Abhängigkeit von der Listenlänge:



Abschätzung:

- Höhe des Rekursionsbaumes: $\lg n$.
- Laufzeit pro Rekursionsebene: linear.
- Gesamtlaufzeit: $n \lg n$.

☞ Lässt sich die Laufzeit verbessern?

29. Linearisierung eines Binärbaums

Aufgabe: Überführung eines Binärbaums in eine Liste — ein Suchbaum soll dabei auf eine geordnete Liste abgebildet werden.

Das Struktur Entwurfsmuster für *Tree* führt fast direkt zum Ziel.

```
let rec inorder (tree : Tree ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → ...
  | Node (left, x, right) → ... inorder left ... inorder right ...
```

☞ *Zur Erinnerung:* Im Rekursionsschritt dürfen wir die Teillösungen für den linken *und* den rechten Teilbaum verwenden.

29. Linearisierung eines Binärbaums

- *Rekursionsbasis*: der leere Baum wird zur leeren Liste.

```
let rec inorder (tree : Tree ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → []
  | Node (left, x, right) → ... inorder left ... inorder right ...
```

- *Rekursionsschritt*: wir müssen zwei Listen aneinanderhängen.

```
let rec inorder (tree : Tree ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → []
  | Node (left, x, right) → inorder left @ x :: inorder right
```

☞ Die relative Reihenfolge der Elemente bleibt erhalten. Wegen der Position des Wurzelelements heißt die Funktion *inorder*.

29. Linearisierung eines Binärbaums: Laufzeit

- Wie schnell ist die Funktion *inorder*?
- Das hängt wie so oft von der Form des Binärbaums ab.
- Zunächst: wie schnell ist die Listenkonkatenation @?
- Die Funktion @ rekurriert über das erste Argument, die Laufzeit ist also proportional zur Länge der ersten Liste.
- Zurück zu *inorder*:
  - Wenn der linke Teilbaum immer leer ist: lineare Laufzeit.
  - Ist der rechte Teilbaum immer leer, dann werden nacheinander Listen der Längen 1, 2, ..., $n-2$, $n-1$ durchlaufen. Insgesamt: $1+2+\cdots+n-2+n-1=(n-1)\cdot n/2$ Schritte, also eine *quadratische* Laufzeit.

29. Linearisierung eines Binärbaums: Laufzeit

Die Laufzeit von *inorder* ist vielleicht unerwartet, auf jeden Fall ist sie unbefriedigend. Die folgende Tabelle zeigt warum.

$\lg n$	n	$n \lg n$	n^2
≈ 7	100	≈ 700	10.000
≈ 10	1.000	≈ 10.000	1.000.000
≈ 14	10.000	≈ 140.000	100.000.000
≈ 17	100.000	$\approx 1.700.000$	10.000.000.000
≈ 20	1.000.000	$\approx 20.000.000$	1.000.000.000.000

☞ Um zum Beispiel einen Baum mit zehntausend Elementen zu linearisieren — Bäume dieser Größenordnung sind nicht ungewöhnlich —, werden hundertmillionen Schritte benötigt.

29. Linearisierung eines Binärbaums

Wie können wir *inorder* verbessern?

Idee: wir *verallgemeinern* die Aufgabenstellung und programmieren eine Funktion, die gleichzeitig linearisiert *und* konkateniert.

Spezifikation:

$$\mathit{inorder\text{-}append}\ (\mathit{tree}, \mathit{list}) = \mathit{inorder}\ \mathit{tree}\ @\ \mathit{list}$$

29. Linearisierung eines Binärbaums

Mit dem Struktur Entwurfsmuster für *Tree* erhalten wir:

```
let rec inorder-append (tree : Tree ⟨'a⟩, list : List ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → ...
  | Node (left, x, right) →
    ... inorder-append (left, ...) ... inorder-append (right, ...) ...
```

29. Linearisierung eines Binärbaums

- *Rekursionsbasis*: die Liste wird zurückgegeben.

```
let rec inorder-append (tree : Tree ⟨'a⟩, list : List ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → list
  | Node (left, x, right) →
    ... inorder-append (left, ...) ... inorder-append (right, ...) ...
```

- *Rekursionsschritt*: wir müssen die rekursiven Aufrufe nur ineinander schachteln.

```
let rec inorder-append (tree : Tree ⟨'a⟩, list : List ⟨'a⟩) : List ⟨'a⟩ =
  match tree with
  | Leaf                  → list
  | Node (left, x, right) →
    inorder-append (left, x :: inorder-append (right, list))
```

29. Programmiertechnik: Rekursionsparadoxon

Fazit: Ein schwierigeres Problem muss nicht schwieriger zu lösen sein.

Die Ursache für diese scheinbar paradoxe Tatsache liegt in der Rekursion begründet: im Rekursionsschritt können wir auf Teillösungen zurückgreifen; die rekursiven Aufrufe lösen aber bereits schwierigere Teilprobleme, so dass der Schritt zur Gesamtlösung oft einfacher ist.

Im Fall von *inorder-append* zum Beispiel erledigt der rekursive Aufruf zusätzlich das Aneinanderhängen der Teillisten.

(Beim Beweisen verwendet man ähnliche Techniken: Verstärkung der Induktion; auch bekannt unter dem Namen „Inventor's paradox".)

29. Übersicht — Laufzeit

	Liste	Suchliste	Suchbaum
empty	konstant	konstant	konstant
add	konstant	linear	linear zur Höhe
remove	linear	linear	linear zur Höhe
is-empty	konstant	konstant	konstant
contains	linear	linear	linear zur Höhe
from-list	konstant	linear-logarithmisch	linear-logarithmisch
to-list	konstant	konstant	linear

☞ Die Höhe eines Binärbaums ist im schlechtesten Fall linear zur Größe!

29. Zusammenfassung

Wir haben

- verschiedene Sortieralgorithmen kennengelernt (unter anderem Sortieren durch Mischen),
- mit Rekursionsbäumen die Laufzeit von Programmen abgeschätzt,
- elementare Suchstrukturen kennengelernt (unter anderem binäre Suchbäume),
- Vorbedingungen, Nachbedingungen und Invarianten eingeführt,
- zwischen partieller und totaler Korrektheit unterschieden,
- die Korrektheit der binären Suche bewiesen,
- gesehen, dass schwierigere Probleme manchmal einfacher zu lösen sind: Rekursionsparadoxon.