3.6 Polymorphe Datentypen

Man kann sich auch selbst poloymorphe Datentypen definieren:

```
type 'a tree = Leaf of 'a
             | Node of ('a tree * 'a tree)
```

→ `tree` heißt Typkonstruktor, weil er aus einem anderen Typ (seinem Parameter `'a`) einen neuen Typ erzeugt.

→ Auf der rechten Seite dürfen nur die Typvariablen vorkommen, die auf der linken Seite als Argument für den Typkonstruktor stehen.

→ Die Anwendung der Konstruktoren auf Daten instanziiert die Typvariable(n):

```
# Leaf 1;;
- : int tree = Leaf 1
# Node (Leaf ('a',true), Leaf ('b',false));;
- : (char * bool) tree = Node (Leaf ('a', true),
                               Leaf ('b', false))
```

Funktionen auf polymorphen Datentypen sind typischerweise wieder polymorph ...

```
let rec  size = function
       Leaf _     -> 1
    |  Node(t,t') -> size t + size t'

let rec flatten = function
       Leaf x     -> [x]
     | Node(t,t') -> flatten t @ flatten t'

let flatten1 t = let rec doit = function
                    (Leaf x, xs) -> x :: xs
                 | (Node(t,t'), xs) -> let xs = doit (t',xs)
                                       in  doit (t,xs)
   in doit (t,[])
...
```

```
...
val size : 'a tree -> int = <fun>
val flatten : 'a tree -> 'a list = <fun>
val flatten1 : 'a tree -> 'a list = <fun>

# let t = Node(Node(Leaf 1,Leaf 5),Leaf 3);;
val t : int tree = Node (Node (Leaf 1, Leaf 5), Leaf 3)

# size t;;
- : int = 3
# flatten t;;
val : int list = [1;5;3]
# flatten1 t;;
val : int list = [1;5;3]
```

3.7 Anwendung: Queues

Gesucht:

Datenstruktur `'a queue`, die die folgenden Operationen unterstützt:

```
enqueue : 'a -> 'a queue -> 'a queue
dequeue : 'a queue  -> 'a option * 'a queue
is_empty : 'a queue -> bool
queue_of_list : 'a list -> 'a queue
list_of_queue : 'a queue -> 'a list
```

1. Idee:

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial :-)

1. Idee:

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

  Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial :-)

- Entnehmen heißt Zugreifen auf das erste Element der Liste:

```
let dequeue = function
      []    -> (None, [])
    | x::xs -> (Some x, xs)
```

1. Idee:

- Repräsentiere die Schlange als eine Liste:

```
type 'a queue = 'a list
```

Die Funktionen `is_empty`, `queue_of_list`, `list_of_queue` sind dann trivial :-)

- Entnehmen heißt Zugreifen auf das erste Element der Liste:

```
let dequeue = function
      []    -> (None, [])
    | x::xs -> (Some x, xs)
```

- Einfügen bedeutet hinten anhängen:

```
let enqueue x xs = xs @ [x]
```

Diskussion:

- Der Operator `@` konkateniert zwei Listen.
- Die Implementierung ist sehr einfach :-)
- Entnehmen ist sehr billig :-)
- Einfügen dagegen kostet so viele rekursive Aufrufe von `@` wie die Schlange lang ist :-(
- Geht das nicht besser ??

2. Idee:

- Repräsentiere die Schlange als zwei Listen !!!

```
type 'a queue = Queue of 'a list * 'a list
let is_empty = function
               Queue ([],[]) -> true
             | _             -> false
let queue_of_list list = Queue (list,[])
let list_of_queue = function
               Queue (first,[])   -> first
             | Queue (first,last) ->
                      first @ List.rev last
```

- Die zweite Liste repräsentiert das Ende der Liste und ist deshalb in umgedrehter Anordnung ...

2. Idee (Fortsetzung):

- Einfügen erfolgt deshalb in der zweiten Liste:

```
let enqueue x (Queue (first,last)) =
                  Queue (first, x::last)
```

2. Idee (Fortsetzung):

- Einfügen erfolgt deshalb in der zweiten Liste:

```
let enqueue x (Queue (first,last)) =
              Queue (first, x::last)
```

- Entnahme bezieht sich dagegen auf die erste Liste :-)

  Ist diese aber leer, wird auf die zweite zugegriffen ...

```
let dequeue = function
    Queue ([],last) -> (match List.rev last
                       with [] -> (None, Queue ([],[]))
                       | x::xs -> (Some x, Queue (xs,[])))
  | Queue (x::xs,last) -> (Some x, Queue (xs,last))
```

Diskussion:

- Jetzt ist Einfügen billig :-)
- Entnehmen dagegen kann so teuer sein, wie die Anzahl der Elemente in der zweiten Liste :-(
- Gerechnet aber auf jede Einfügung, fallen nur konstante Zusatzkosten an !!!

$\Longrightarrow$ amortisierte Kostenanalyse

3.8 Namenlose Funktionen

Wie wir gesehen haben, sind Funktionen Daten. Daten, z.B. `[1;2;3]` können verwendet werden, ohne ihnen einen Namen zu geben. Das geht auch für Funktionen:

```
# fun x y z -> x+y+z;;
- : int -> int -> int -> int = <fun>
```

- `fun` leitet eine Abstraktion ein.
  Der Name kommt aus dem λ-Kalkül.
- `->` hat die Funktion von `=` in Funktionsdefinitionen.
- Rekursive Funktionen können so nicht definiert werden, denn ohne Namen kann eine Funktion nicht in ihrem Rumpf vorkommen :-)

Alonzo Church, 1903–1995

- Um Pattern Matching zu benutzen, kann man `match ... with` für das entsprechende Argument einsetzen.
- Bei einem einzigen Argument bietet sich `function` an ...

```
# function None   -> 0
         | Some x -> x*x+1;;
- : int option -> int = <fun>
```

Namenlose Funktionen werden verwendet, wenn sie nur einmal im Programm vorkommen. Oft sind sie Argument für Funktionale:

```
# map (fun x -> x*x) [1;2;3];;
- : int list = [1; 4; 9]
```

Oft werden sie auch benutzt, um eine Funktion als Ergebnis zurückzuliefern:

```
# let make_undefined () = fun x -> None;;
val make_undefined : unit -> 'a -> 'b option = <fun>
# let def_one (x,y) = fun x' -> if x=x' then Some y
                                 else None;;
val def_one : 'a * 'b -> 'a -> 'b option = <fun>
```

4 Größere Anwendung: Balancierte Bäume

Erinnerung: Sortiertes Array:

2	3	5	7	11	13	17

Eigenschaften:

- Sortierverfahren gestatten Initialisierung mit $\approx n \cdot \log(n)$ vielen Vergleichen :-)

  // $n ==$ Größe des Arrays

- Binäre Suche erlaubt Auffinden eines Elements mit $\approx \log(n)$ vielen Vergleichen :-)

- Arrays unterstützen weder Einfügen noch Löschen einzelner Elemente :-(

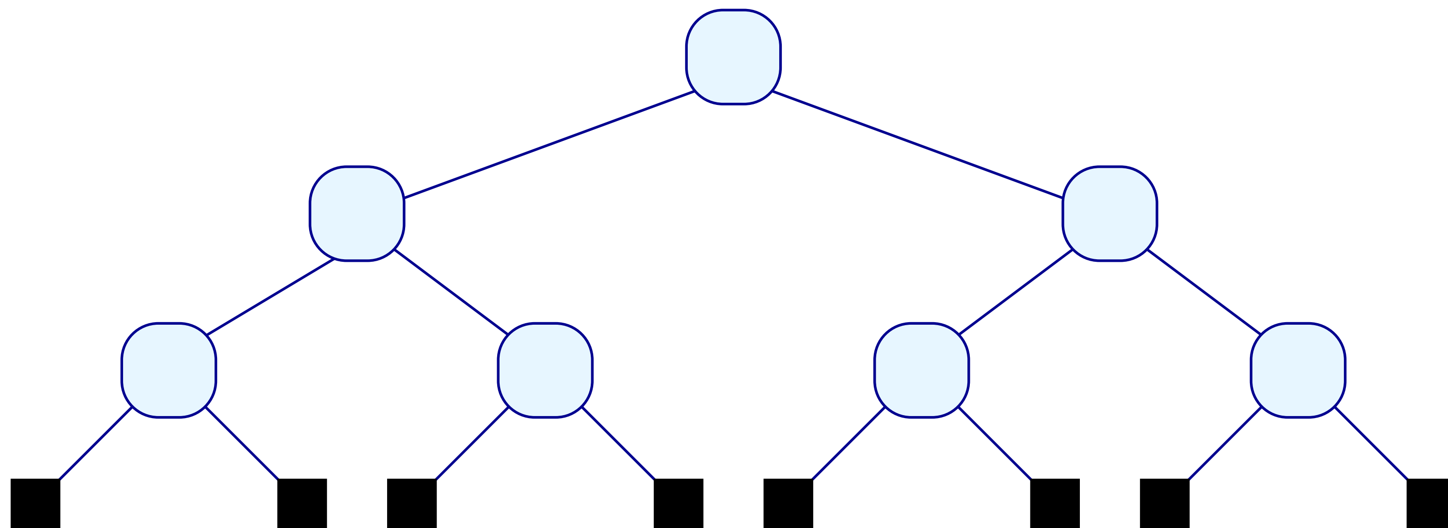
Gesucht:

Datenstruktur `'a d`, die dynamisch eine Folge von Elementen sortiert hält, d.h. die die Operationen unterstützt:

```
insert :         'a -> 'a d -> 'a d
delete :         'a -> 'a d -> 'a d
extract_min :          'a d -> 'a option * 'a d
extract_max :          'a d -> 'a option * 'a d
extract : 'a * 'a -> 'a d -> 'a list * 'a d
list_of_d :            'a d -> 'a list
d_of_list :       'a list -> 'a d
```

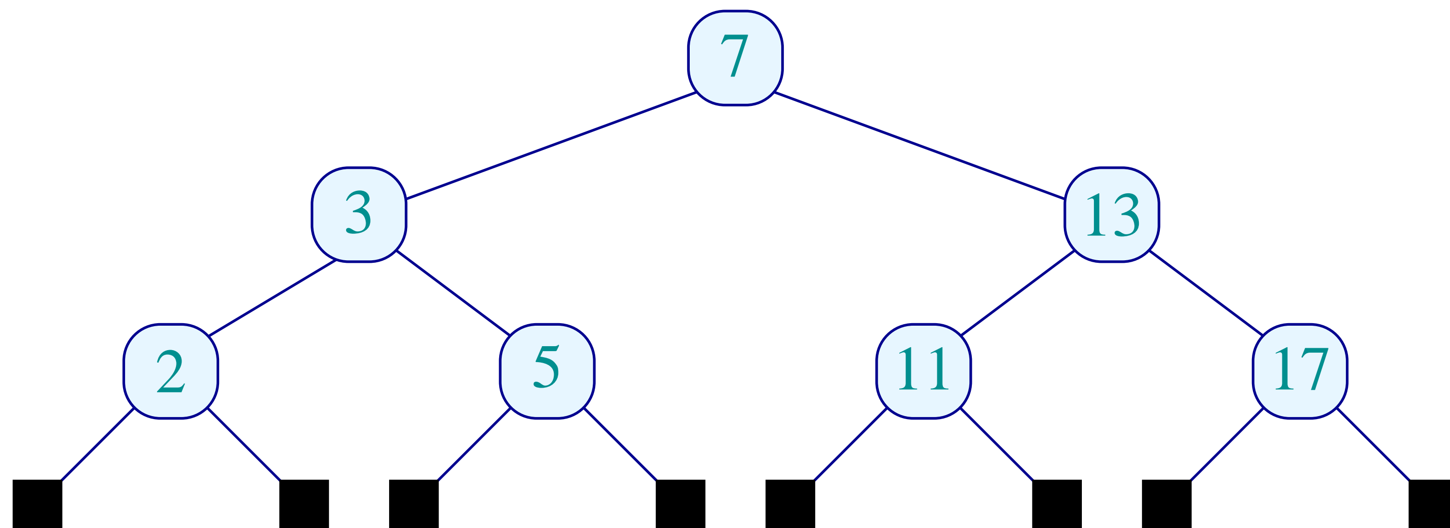
1. Idee:

Benutze balancierte Bäume ...



1. Idee:

Benutze balancierte Bäume ...



Diskussion:

- Wir speichern unsere Daten in den inneren Knoten :-)
- Ein Binärbaum mit n Blättern hat $n - 1$ innere Knoten :-)
- Zum Auffinden eines Elements müssen wir mit allen Elementen auf einem Pfad vergleichen ...
- Die Tiefe eines Baums ist die maximale Anzahl innerer Knoten auf einem Pfad von der Wurzel zu einem Blatt.
- Ein vollständiger balancierter Binärbaum mit $n = 2^k$ Blättern hat Tiefe $k = \log(n)$:-)

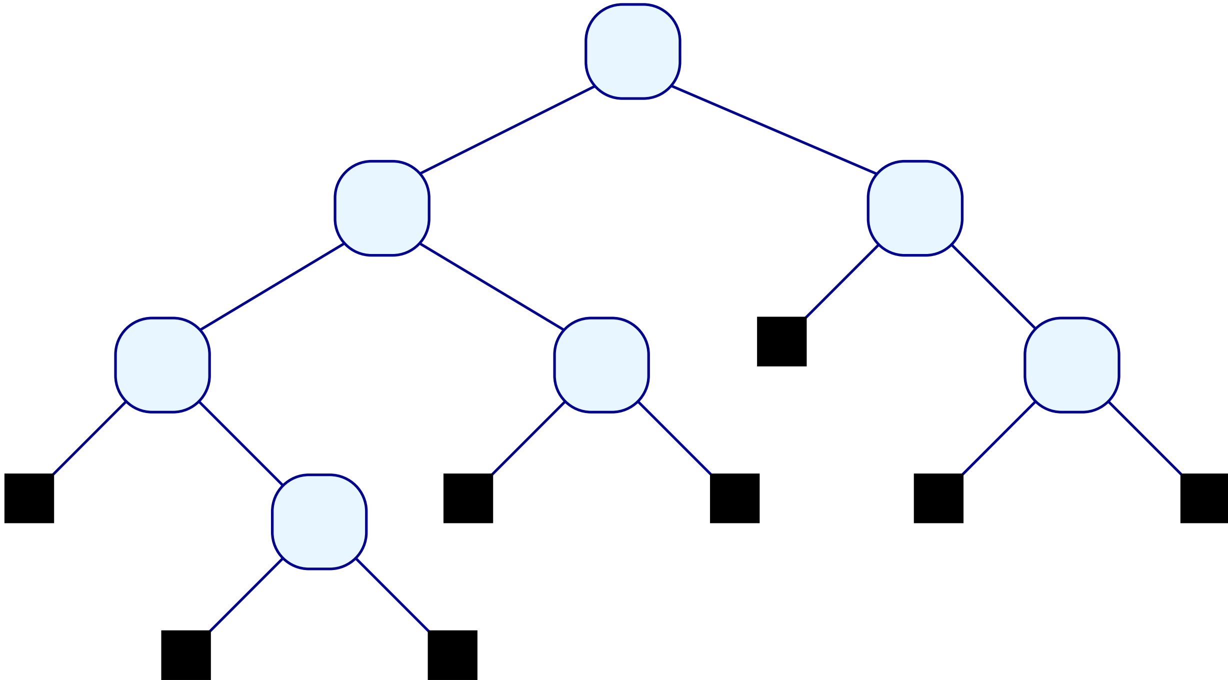
Diskussion:

- Wir speichern unsere Daten in den inneren Knoten :-)
- Ein Binärbaum mit n Blättern hat $n-1$ innere Knoten :-)
- Zum Auffinden eines Elements müssen wir mit allen Elementen auf einem Pfad vergleichen ...
- Die Tiefe eines Baums ist die maximale Anzahl innerer Knoten auf einem Pfad von der Wurzel zu einem Blatt.
- Ein vollständiger balancierter Binärbaum mit $n = 2^k$ Blättern hat Tiefe $k = \log(n)$:-)
- Wie fügen wir aber weitere Elemente ein ??
- Wie können wir Elemente löschen ???

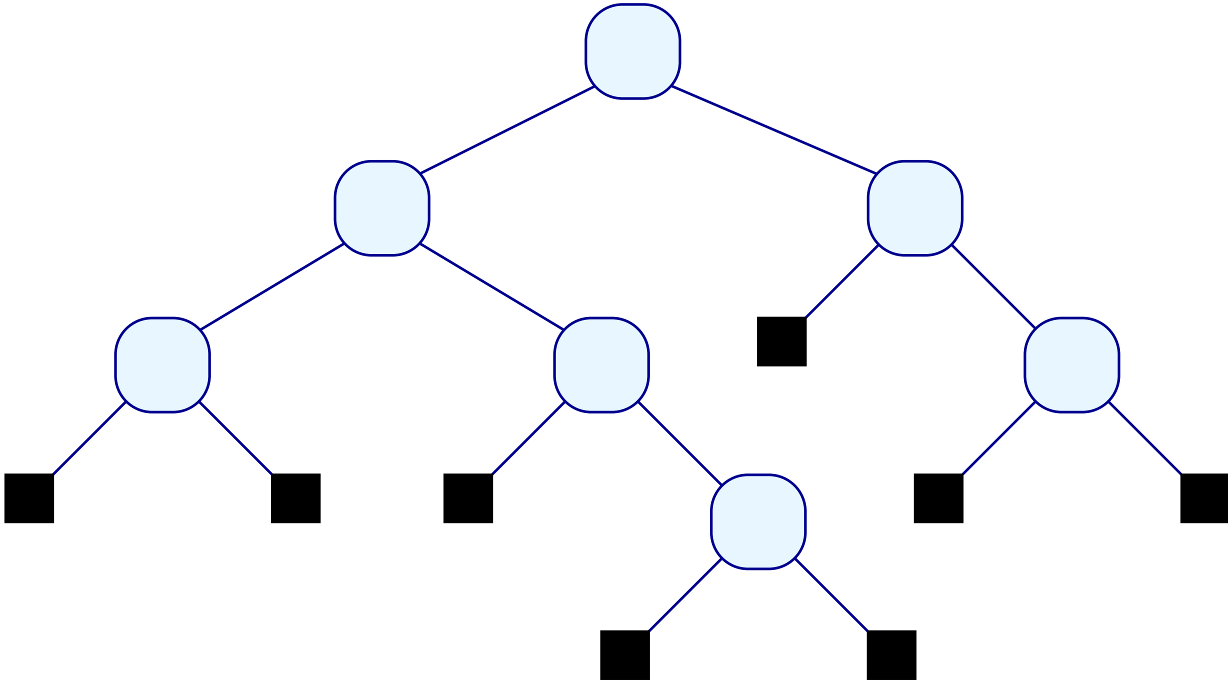
2. Idee:

- Statt balancierter Bäume benutzen wir fast balancierte Bäume ...
- An jedem Knoten soll die Tiefe des rechten und linken Teilbaums ungefähr gleich sein :-)
- Ein AVL-Baum ist ein Binärbaum, bei dem an jedem inneren Knoten die Tiefen des rechten und linken Teilbaums maximal um 1 differieren ...

Ein AVL-Baum:



Ein AVL-Baum:



Kein AVL-Baum:

G.M. Adelson-Velskij, 1922

E.M. Landis, Moskau, 1921-1997

Wir vergewissern uns:

(1) Jeder AVL-Baum der Tiefe $k > 0$ hat mindestens

$$\mathsf{fib}(k) \quad \geq \quad A^{k-1}$$

Knoten für $A = \frac{\sqrt{5}+1}{2}$ // goldener Schnitt :-)

Wir vergewissern uns:

(1) Jeder AVL-Baum der Tiefe $k > 0$ hat mindestens

$$\mathsf{fib}(k) \quad \geq \quad A^{k-1}$$

Knoten für $A = \frac{\sqrt{5}+1}{2}$ // goldener Schnitt :-)

(2) Jeder AVL-Baum mit $n > 0$ inneren Knoten hat Tiefe maximal

$$\frac{1}{\log(A)} \cdot \log(n) + 1$$

Wir vergewissern uns:

(1) Jeder AVL-Baum der Tiefe $k > 0$ hat mindestens

$$\mathsf{fib}(k) \quad \geq \quad A^{k-1}$$

Knoten für $\quad A = \frac{\sqrt{5}+1}{2} \qquad //$ goldener Schnitt :-)

(2) Jeder AVL-Baum mit $n > 0$ inneren Knoten hat Tiefe maximal

$$\frac{1}{\log(A)} \cdot \log(n) + 1$$

Beweis: Wir zeigen nur (1) :-)

Sei $N(k)$ die minimale Anzahl der inneren Knoten eines AVL-Baums der Tiefe k .

Induktion nach der Tiefe $k > 0$:-)

$\boxed{k = 1 :}$ $\quad N(1) = 1 = \mathsf{fib}(1) = A^0$:-)

$\boxed{k = 2 :}$ $\quad N(2) = 2 = \mathsf{fib}(2) \geq A^1$:-)

:-))

$k = 1:$ $N(1) = 1 = \text{fib}(1) = A^0$:-)

$k = 2:$ $N(2) = 2 = \text{fib}(2) \geq A^1$:-)

$k > 2:$ Gelte die Behauptung bereits für $k-1$ und $k-2$ ...

$$
\begin{aligned}
\Longrightarrow \quad N(k) &= N(k-1) + N(k-2) + 1 \\
&\geq \text{fib}(k-1) + \text{fib}(k-2) \\
&= \text{fib}(k)
\end{aligned}
$$

:-)

$k = 1:$ $\quad N(1) = 1 = \text{fib}(1) = A^0$:-)

$k = 2:$ $\quad N(2) = 2 = \text{fib}(2) \geq A^1$:-)

$k > 2:$ $\quad$ Gelte die Behauptung bereits für $k-1$ und $k-2$ ...

$$\Longrightarrow \quad \begin{aligned} N(k) &= N(k-1) + N(k-2) + 1 \\ &\geq \text{fib}(k-1) + \text{fib}(k-2) \\ &= \text{fib}(k) \end{aligned}$$

:-)

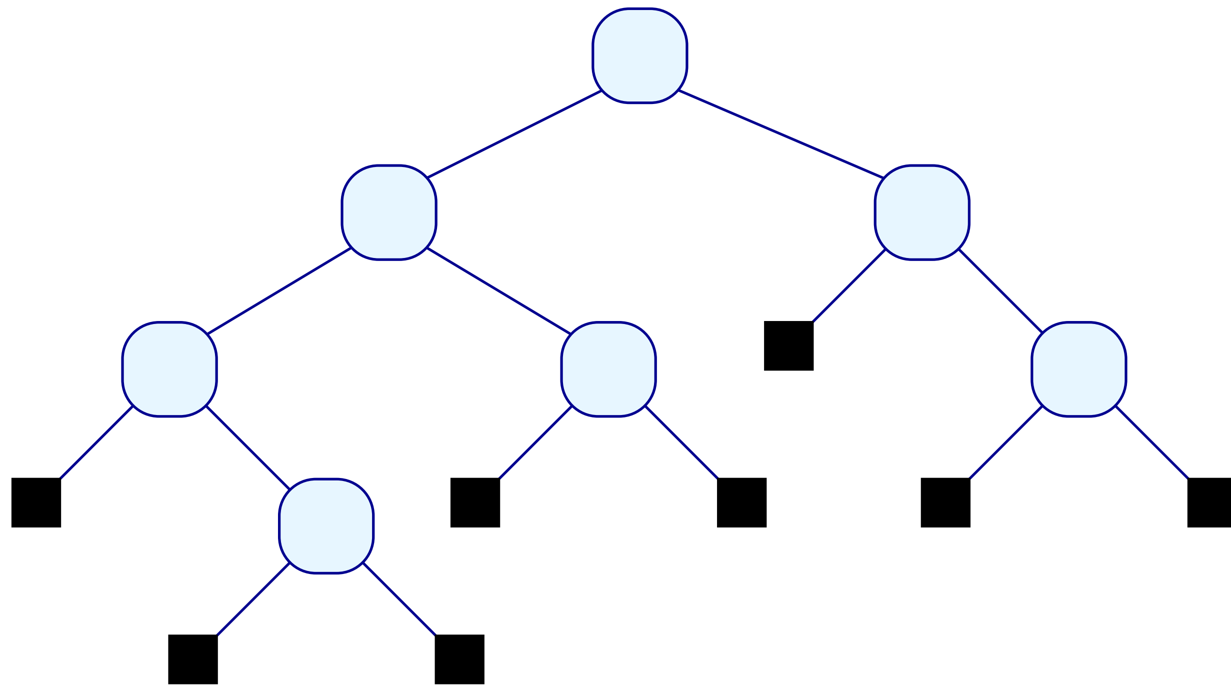
$$\begin{aligned} \text{fib}(k) &= \text{fib}(k-1) + \text{fib}(k-2) \\ &\geq A^{k-2} + A^{k-3} \\ &= A^{k-3} \cdot (A+1) \\ &= A^{k-3} \cdot A^2 \\ &= A^{k-1} \end{aligned}$$

:-))

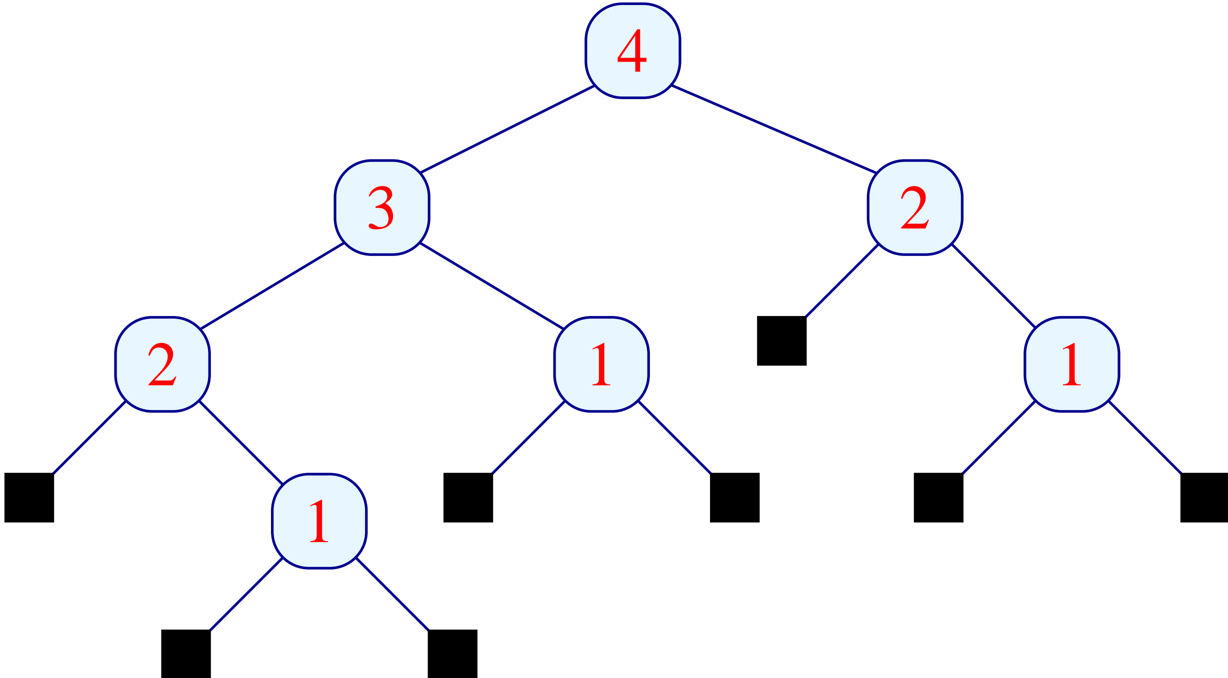
2. Idee (Fortsetzung)

- Fügen wir ein weiteres Element ein, könnte die AVL-Eigenschaft verloren gehen :-(
- Entfernen wir ein Element ein, könnte die AVL-Eigenschaft verloren gehen :-(
- Dann müssen wir den Baum so umbauen, dass die AVL-Eigenschaft wieder hergestellt wird :-)
- Dazu müssen wir allerdings an jedem inneren Knoten wissen, wie tief die linken bzw. rechten Teilbäume sind ...

Repräsentation:



Repräsentation:

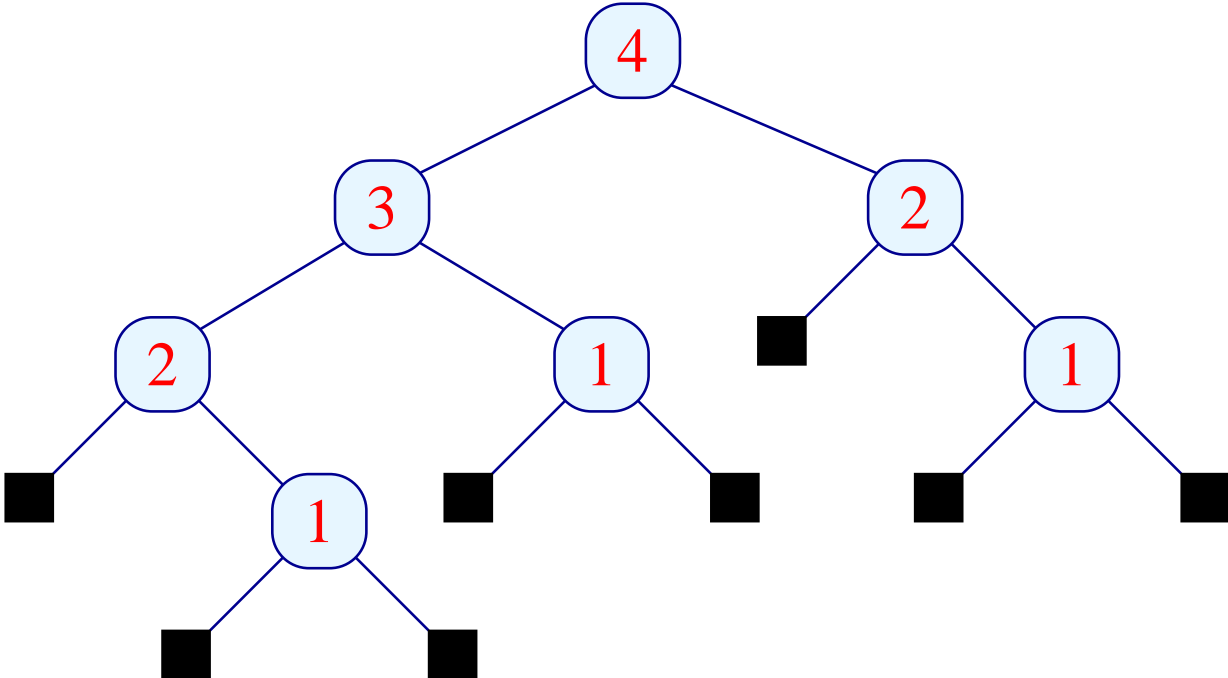


3. Idee:

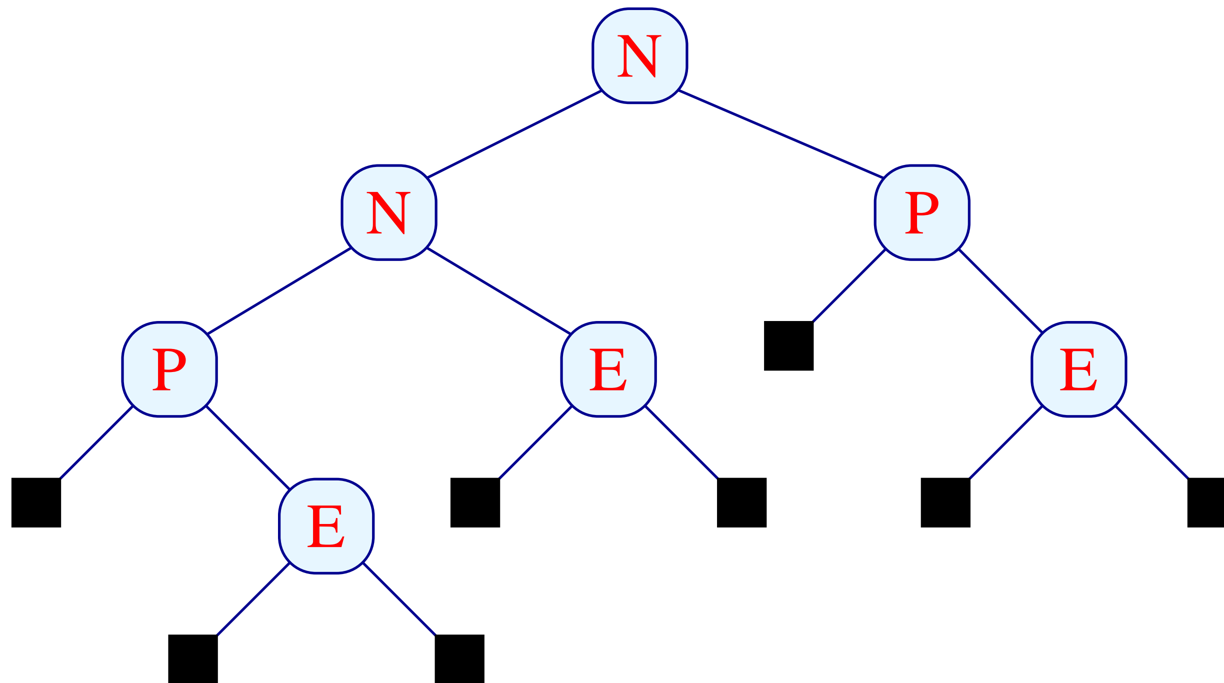
- Anstelle der absolutenTiefen speichern wir an jedem Knoten nur, ob die Differenz der Tiefen der Teilbäume negativ, positiv oder ob sie gleich sind !!!
- Als Datentyp definieren wir deshalb:

```
type 'a avl = Null
            | Neg of 'a avl * 'a * 'a avl
            | Pos of 'a avl * 'a * 'a avl
            | Eq  of 'a avl * 'a * 'a avl
```

Repräsentation:



Repräsentation:



Einfügen:

- Ist der Baum ein Blatt, erzeugen wir einen neuen inneren Knoten mit zwei neuen leeren Blättern.
- Ist der Baum nicht-leer, vergleichen wir den einzufügenden Wert mit dem Wert an der Wurzel.

  $\longrightarrow$ Ist er größer, fügen wir rechts ein.

  $\longrightarrow$ Ist er kleiner, fügen wir links ein.

- Achtung: Einfügen kann die Tiefe erhöhen und damit die AVL-Eigenschaft zerstören !
- Das müssen wir reparieren ...

```
let rec insert x avl = match avl
    with Null             -> (Eq (Null,x,Null), true)
       | Eq (left,y,right) -> if x < y then
                   let (left,inc) = insert x left
                   in if inc then (Neg (left,y,right), true)
                      else         (Eq (left,y,right), false)
              else let (right,inc) = insert x right
                   in if inc then (Pos (left,y,right), true)
                      else        (Eq  (left,y,right), false)
        ...
```

```
let rec insert x avl = match avl
    with Null              -> (Eq (Null,x,Null), true)
       | Eq (left,y,right) -> if x < y then
                    let (left,inc) = insert x left
                    in if inc then (Neg (left,y,right), true)
                       else         (Eq (left,y,right), false)
                 else let (right,inc) = insert x right
                    in if inc then (Pos (left,y,right), true)
                       else        (Eq  (left,y,right), false)
         ...
```

- Die Funktion `insert` liefert außer dem neuen AVL-Baum die Information, ob das Ergebnis tiefer ist als das Argument :-)
- Erhöht sich die Tiefe nicht, braucht die Markierung der Wurzel nicht geändert werden.

```
| Neg (left,y,right) -> if x < y then
        let (left,inc) = insert x left
        in if inc then let (avl,_) = rotateRight (left,y,right)
                       in (avl,false)
           else        (Neg (left,y,right), false)
  else let (right,inc) = insert x right
        in if inc then (Eq (left,y,right), false)
           else       (Neg (left,y,right), false)
| Pos (left,y,right) -> if x < y then
        let (left,inc) = insert x left
        in if inc then (Eq (left,y,right), false)
           else       (Pos (left,y,right), false)
  else let (right,inc) = insert x right
        in if inc then let (avl,_) = rotateLeft (left,y,right)
                       in (avl,false)
           else     (Pos (left,y,right), false);;
```

Kommentar:

- Einfügen in den flacheren Teilbaum erhöht die Gesamttiefe nie :-)

  Gegebenenfalls werden aber beide Teilbäume gleich tief.

- Einfügen in den tieferen Teilbaum kann dazu führen, dass der Tiefenunterschied auf 2 anwächst :-(

  Dann rotieren wir Knoten an der Wurzel, um die Differenz auszugleichen ...