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>
```

5.9 Unendliche Datenstrukturen

Wie können wir die Menge aller natürlichen Zahlen darstellen?

1. Versuch:

```
let rec all_nats n = n :: all_nats (n+1);;
```

`all_nats` berechnet eine unendliche Liste.

Sie versucht es zumindest:

die unendliche Berechnung kommt nie zum Ende :-(

2. Versuch:

Wir benutzen Abstraktionen :-)

Dazu definieren wir zunächst einen Datentyp:

```
type 'a inflist = Lazy of 'a * (unit -> 'a inflist)
```

Dieser Datentyp kann keine endlich großen Daten repräsentieren, denn es fehlt der Konstruktor für das Ende der Liste :-?

Das zweite Argument für `Lazy` (Rest der Liste) ist vom Typ:

```
(unit -> 'a inflist)
```

Es ist also eine Funktion, die, angewendet auf `(): unit` , den Rest der Liste ergibt :-)

Das nutzen wir aus, um unendliche Listen zu repräsentieren:

```
# let rec all_nats n = Lazy (n, fun () -> all_nats (n+1));;
val all_nats : int -> int inflist = <fun>
```

Aufrufe der Funktion `all_nats` terminieren:

```
# let nats = all_nats 0;;
val nats : 'a inflist = Lazy (0, <fun>)
```

Um auf die Komponenten von `inflist`'s zuzugreifen definieren wir:

```
# let head (Lazy (x,f)) = x;;
val head : 'a inflist -> 'a = <fun>
# let tail (Lazy (_,f)) = f();;
val tail : 'a inflist -> 'a inflist = <fun>
let rec nth = function (0, Lazy (x,_)) -> x
                     | (n, Lazy (_,f)) -> nth (n-1, f());;
```

Achtung:

Der Rest der Liste in `tail` und `nth` muss auf den Wert `()` angewendet werden. Erst dadurch wird das nächste Element (und die Funktion, die den weiteren Rest der Liste darstellt) erzeugt.

```
# head nats;;
- : int = 0
# tail nats;;
- : int inflist = Lazy (1,<fun>)
# head (tail (tail (tail nats)));;
- : int = 3
```

`()` ist der einzige Wert vom Typ `unit`.

Er wird häufig zusammen mit Abstraktionen für die Verzögerung von Berechnungen verwendet.

Extrahieren einer endlichen Teilliste:

```
# let rec first_n = function
            (0,_)            -> []
          | (n, Lazy (x,f)) -> x :: first_n (n-1,f());;

# first_n (10,nats);;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Funktionale auf unendlichen Listen:

```
let rec inf_map h (Lazy (x,f)) =
   Lazy (h x, fun () -> inf_map h (f()));;
let rec inf_filter p (Lazy (x,f)) =
   if p x then Lazy (x, fun () -> inf_filter p (f()))
   else inf_filter p (f());;
```

Dann erhalten wir etwa:

```
val inf_map : ('a -> 'b)
              -> 'a inflist -> 'b inflist
val inf_filter : ('a -> bool)
              -> 'a inflist -> 'a inflist
```

Jetzt können wir z.B. die unendliche Liste aller geraden Zahlen oder aller Quadratzahlen berechnen:

```
# first_n (10,inf_filter (fun x -> x mod 2=0) nats);;
- : int list = [0; 2; 4; 6; 8; 10; 12; 14; 16; 18]
# first_n (7,inf_map (fun x -> x*x) nats);;
- : int list = [0; 1; 4; 9; 16; 25; 36]
```

So können wir auch die Liste aller Primzahlen berechnen

(Sieb des Eratosthenes):

```
# let all_primes () =
     let rec sieve (Lazy (n,f)) =
        Lazy (n, (fun () -> sieve (inf_filter
                                     (fun x -> x mod n<>0)
                                     (f())))
     in sieve (all_nats 2);;

# first_n (10, all_primes());;
- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29]
```

6 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 :-(

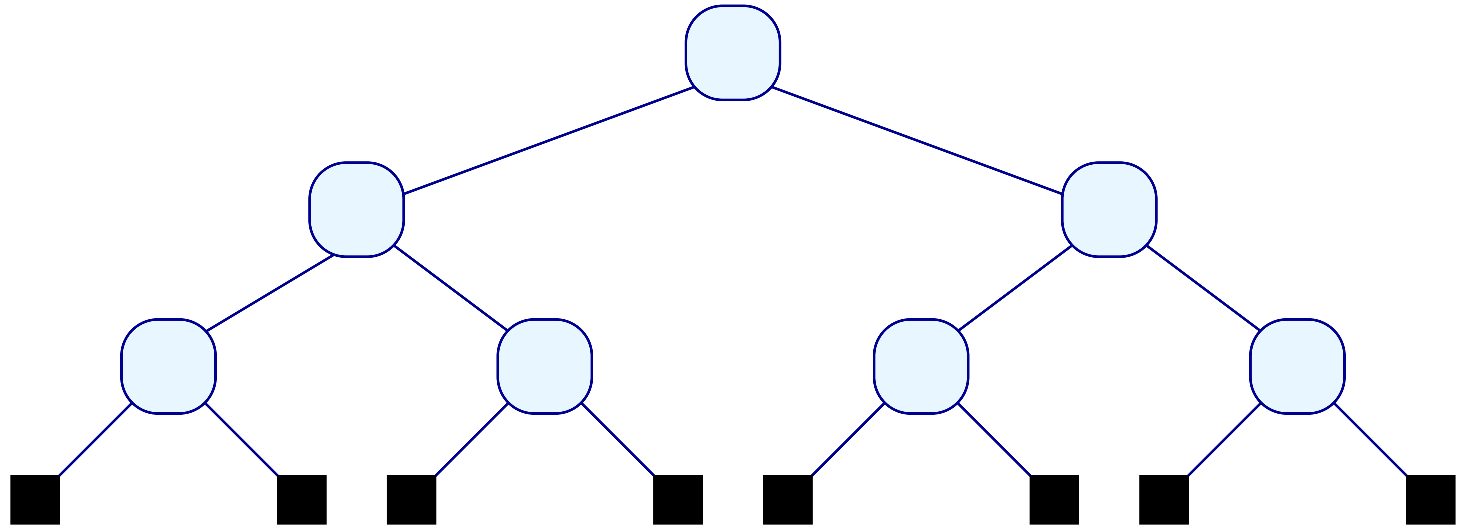
Gesucht:

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
```

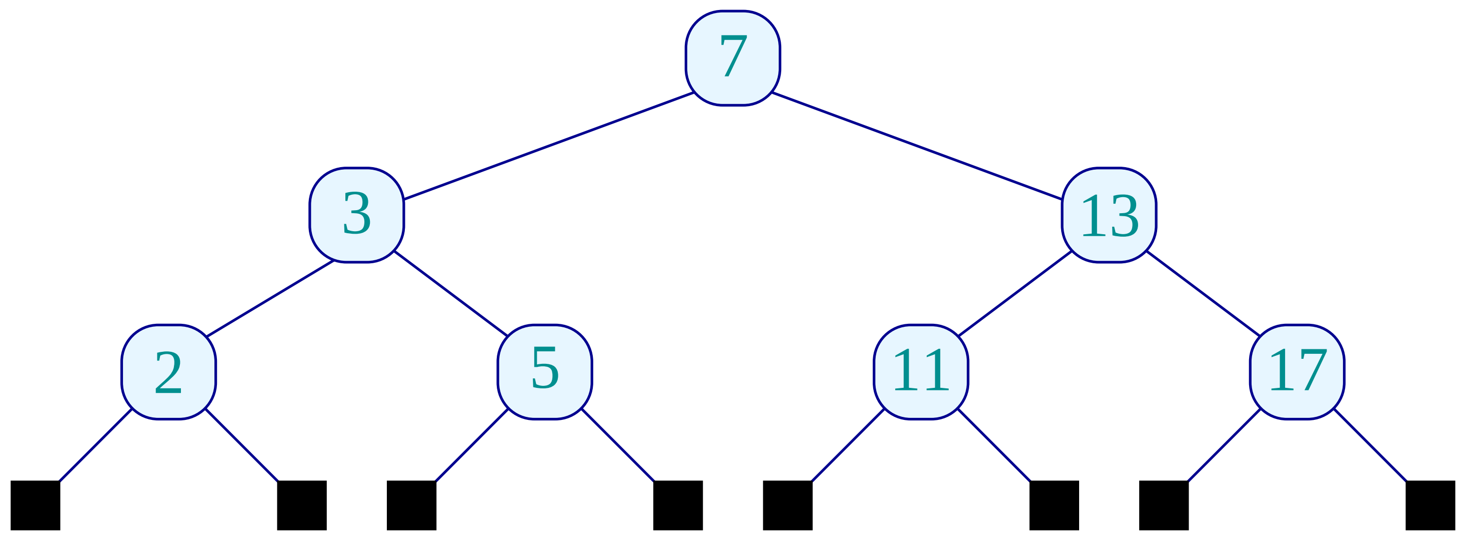
1. Idee:

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 $k = \log(n)$:-)

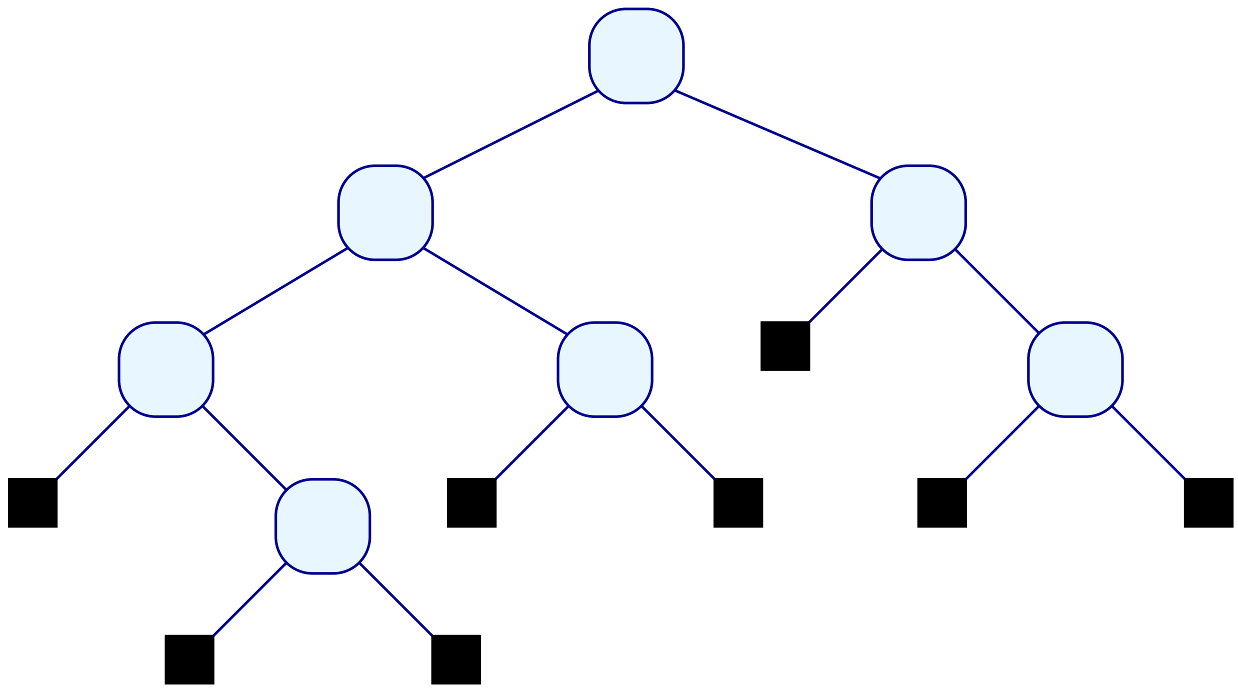
Diskussion:

- 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 ???

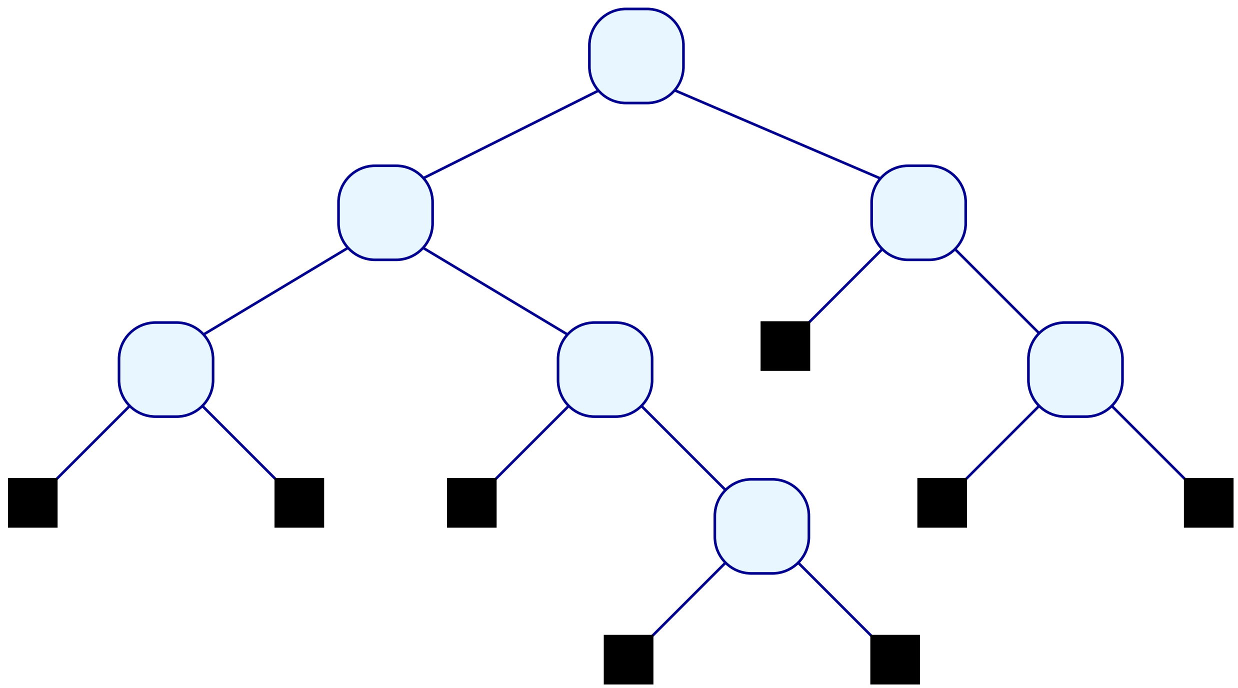
2. Idee:

- 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 ...

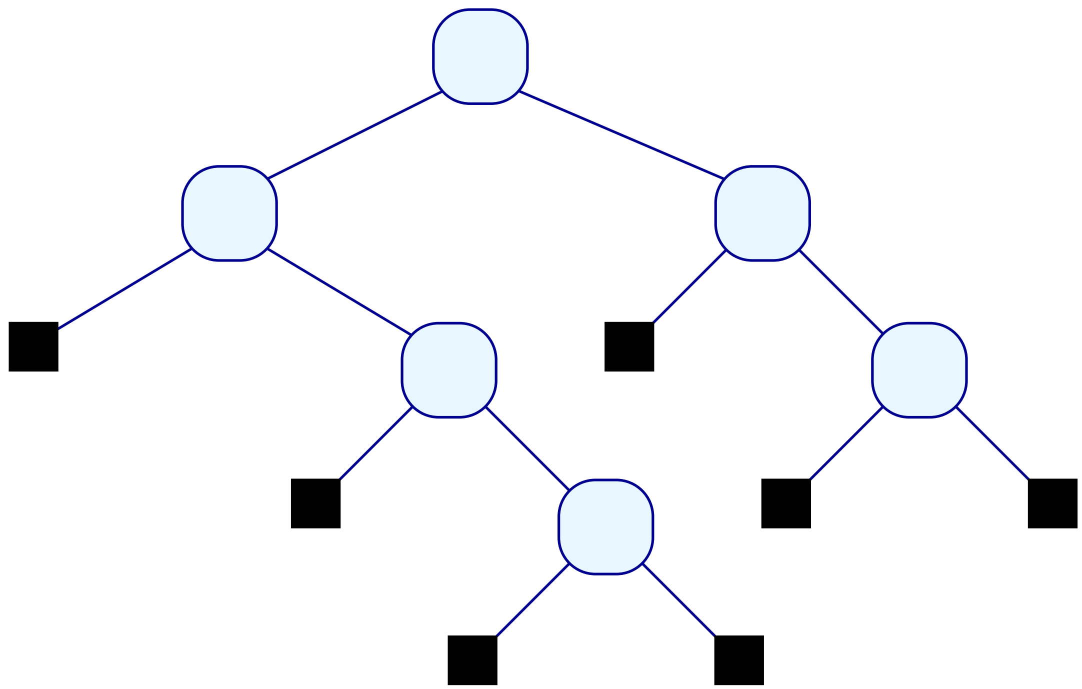
Ein AVL-Baum:



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

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

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

Beweis: Wir zeigen nur (1) :-)

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

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

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

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

:-))

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

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

$\boxed{k > 2:}$ Gelte die Behauptung bereits für $k-1$ und $k-2$ ...

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

:-)

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

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

$\boxed{k > 2 :}$ Gelte die Behauptung bereits für $k-1$ und $k-2$ ...

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

:-)

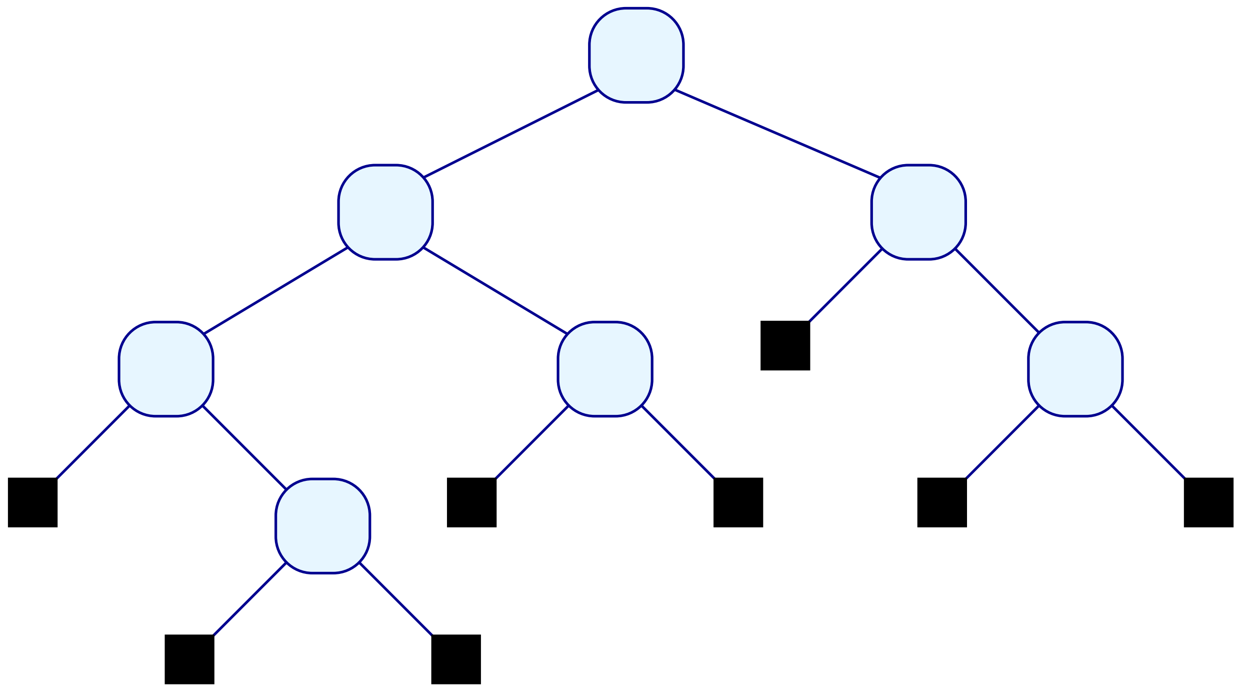
$$
\begin{aligned}
\mathsf{fib}(k) &= \mathsf{fib}(k-1) + \mathsf{fib}(k-2) \\
&\geq A^{k-2} + A^{k-3} \\
&= A^{k-3} \cdot (A+1) \\
&= A^{k-3} \cdot A^2 \\
&= A^{k-1}
\end{aligned}
$$

:-))

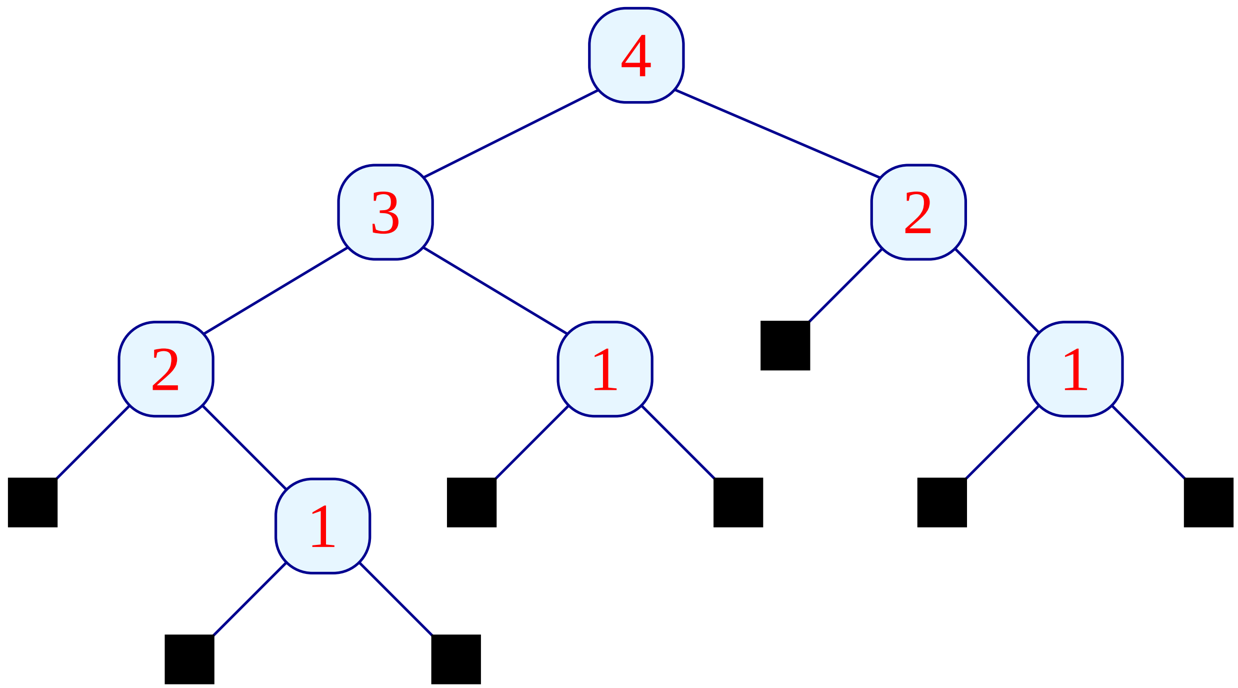
2. Idee (Fortsetzung)

- 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:

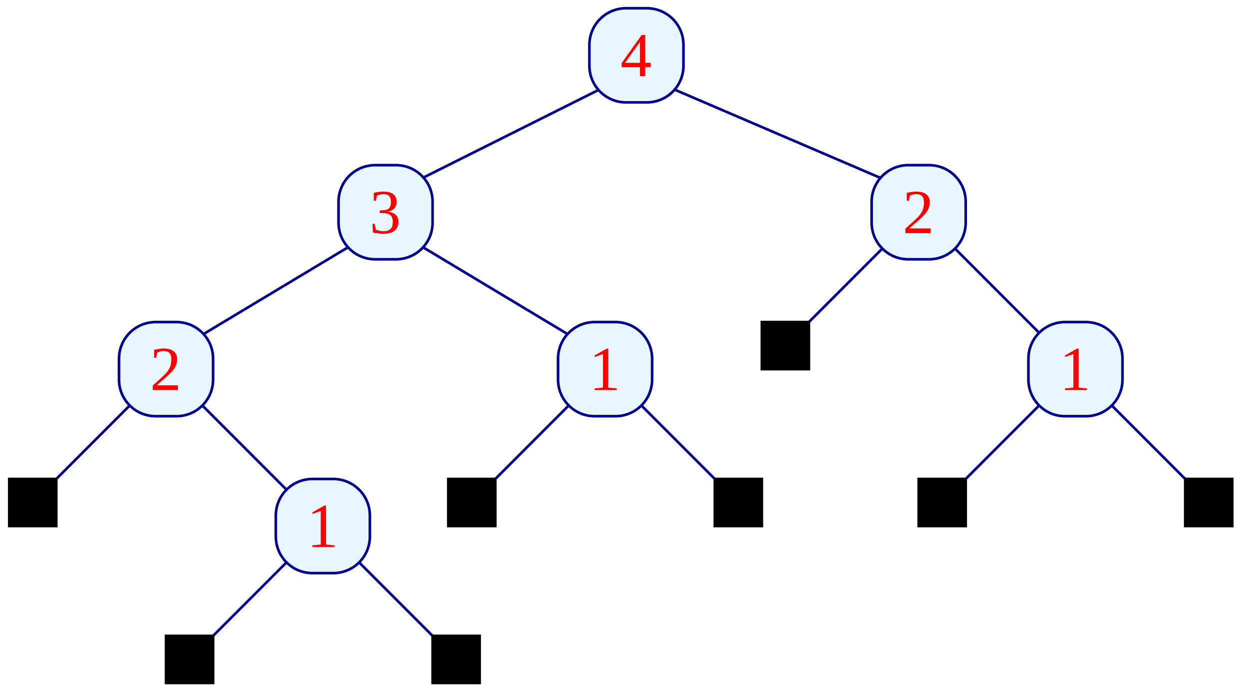


3. Idee:

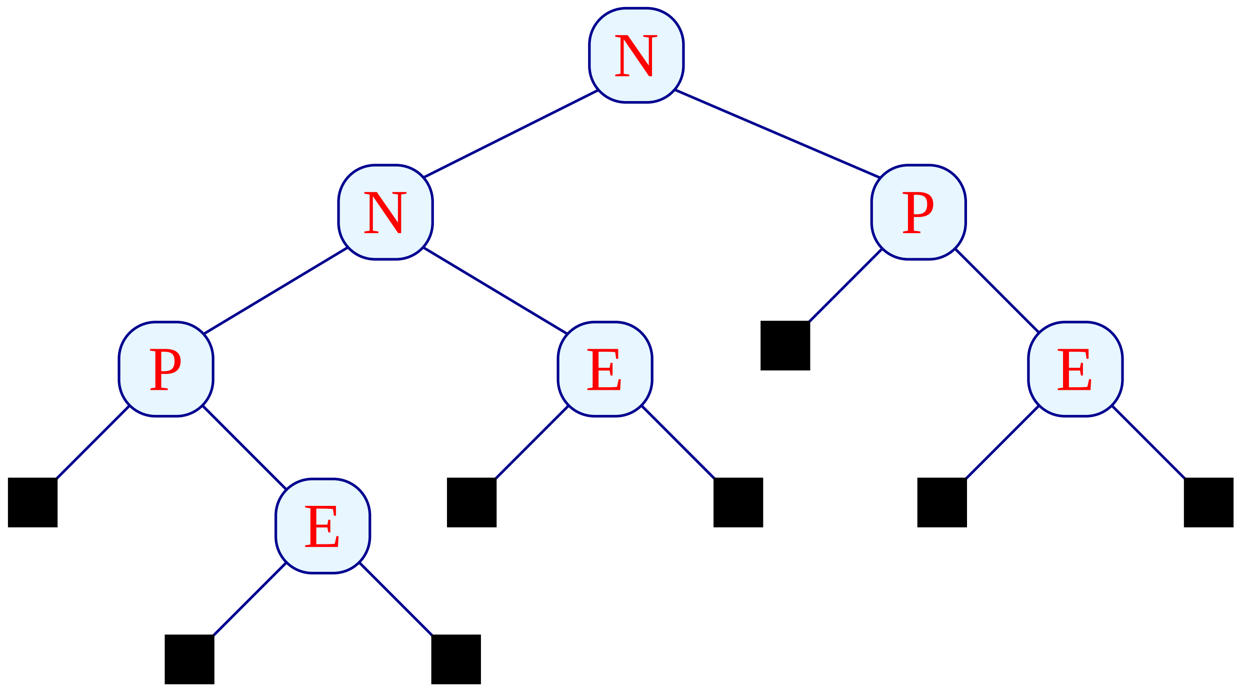
- Anstelle der absolutenTiefen speichern wir an jedem Knoten nur, ob die Differenz der Tiefen der Teilbäume negativ, positiv oder ob sie gleich sind !!!
- Als Datentyp definieren wir deshalb:

```
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.