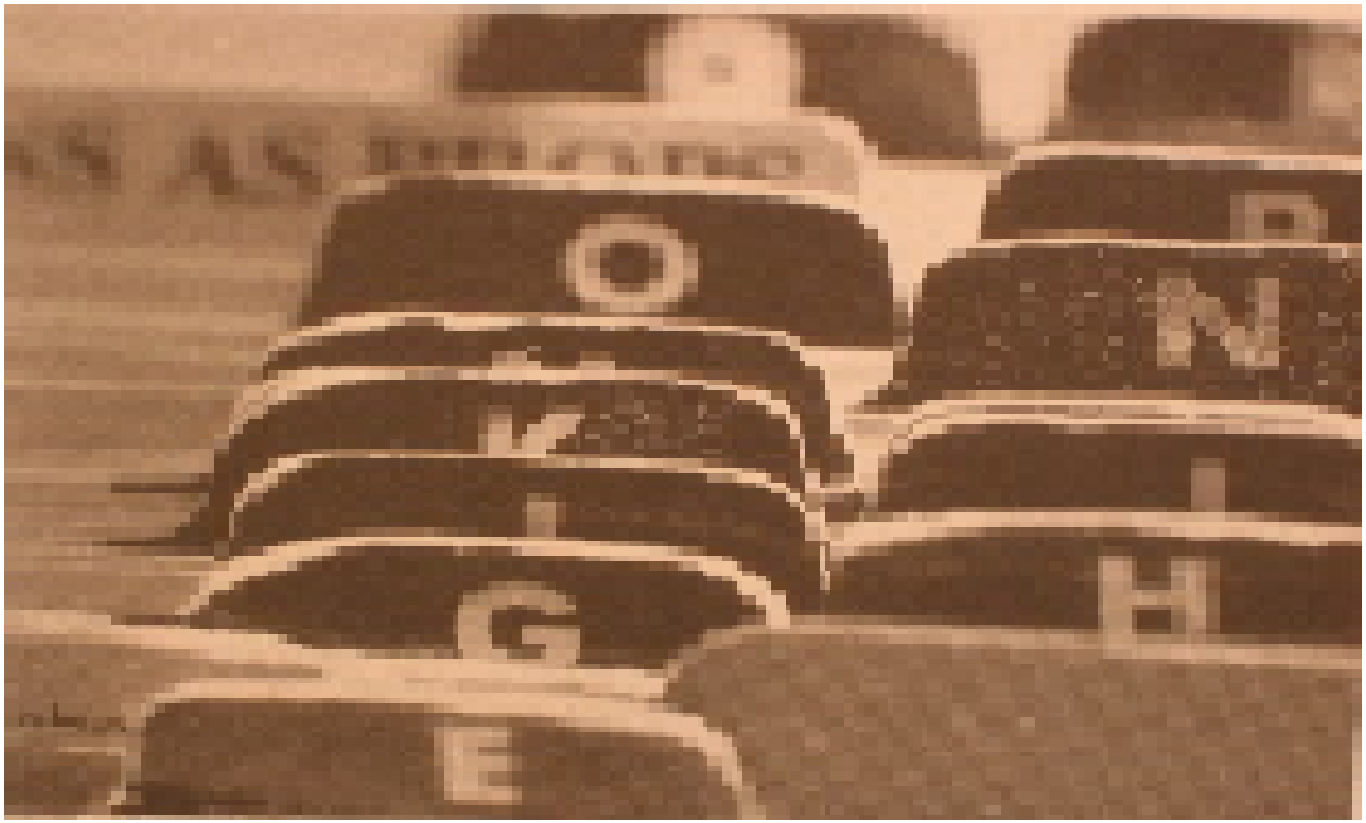
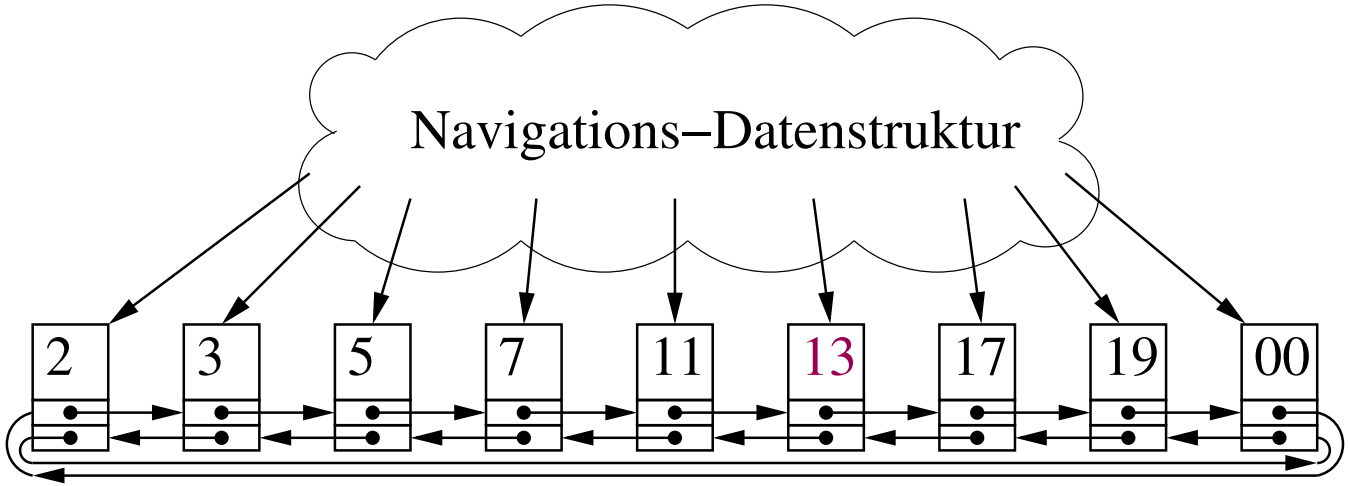
Sortierte Folgen



Sortierte Folgen:

$\langle e_1, \ldots, e_n \rangle$ mit $e_1 \leq \cdots \leq e_n$
„kennzeichnende“ Funktion:

$$M.\text{locate}(k) := \textbf{addressof } \min\{e \in M : e \geq k\}$$



Annahme: Dummy-Element mit Schlüssel ∞
Achtung: In Abbildungen sieht ∞ wie 00 aus

Statisch: Sortiertes Feld mit binärer Suche

```
// Find min{i ∈ 1..n+1 : a[i] ≥ k}
Function locate(a[1..n], k : Element)
    (ℓ, r) := (0, n+1)                    // Assume a[0] = −∞, a[n+1] = ∞
    while ℓ+1 < r do
        invariant 0 ≤ ℓ < r ≤ n+1   and   a[ℓ] < k ≤ a[r]
        m := ⌊(r+ℓ)/2⌋                                  // ℓ < m < r
        if k ≤ a[m] then r := m else ℓ := m
    return r
```

Übung: Müssen die Sentinels ∞ / $-\infty$ tatsächlich vorhanden sein?

Übung: Variante von binärer Suche:
bestimme ℓ, r so dass $a[\ell..r-1] = [k,\ldots,k]$, $a[\ell-1] < k$ und $a[r] > k$

Statisch: Sortiertes Feld mit binärer Suche

```
// Find min{i ∈ 1..n+1 : a[i] ≥ k}
Function locate(a[1..n], k : Element)
    (ℓ, r) := (0, n+1)                    // Assume a[0] = −∞, a[n+1] = ∞
    while ℓ+1 < r do
        invariant 0 ≤ ℓ < r ≤ n+1  and  a[ℓ] < k ≤ a[r]
        m := ⌊(r+ℓ)/2⌋                                  // ℓ < m < r
        if k ≤ a[m] then r := m else ℓ := m
    return r
```

Zeit: $O(\log n)$

Beweisidee: $r - \ell$ „halbiert“ sich in jedem Schritt

Binäre Suche – Beispiel: $k = 15$

Function locate($a[1..n], k$: Element) // $\min\{i \in 1..n+1 : a[i] \geq k\}$
  $(\ell, r) := (0, n+1)$ // Assume $a[0] = -\infty$, $a[n+1] = \infty$
  while $\ell + 1 < r$ **do**
    invariant $0 \leq \ell < r \leq n+1$ and $a[\ell] < k \leq a[r]$
    $m := \lfloor (r+\ell)/2 \rfloor$ // $\ell < m < r$
    if $k \leq a[m]$ **then** $r := m$ **else** $\ell := m$
  return r

Indizes:	[0,	1,	2,	3,	4,	5,	6,	7,	8,	9]
Einträge:	$[-\infty$,	2,	3,	5,	7,	11,	13,	17,	19,	$\infty]$
	$[-\infty$,	2,	3,	5,	7,	11,	13,	17,	19,	$\infty]$
	$[-\infty$,	2,	3,	5,	7,	11,	13,	17,	19,	$\infty]$
	$[-\infty$,	2,	3,	5,	7,	11,	13,	17,	19,	$\infty]$

Dynamische Sortierte Folgen – Grundoperationen

insert, remove, update, locate $\qquad O(\log n)$

$(M.\text{locate}(k) := \min\{e \in M : e \geq k\})$

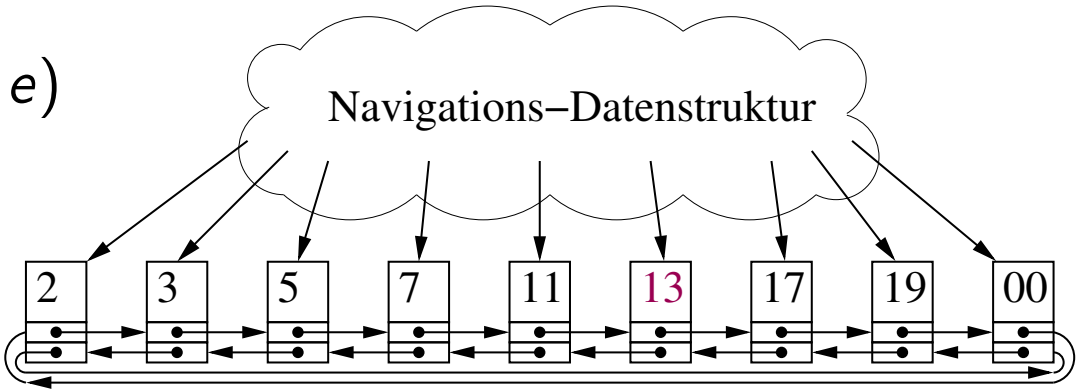
Mehr Operationen

$\langle \text{min}, \ldots, a, \ldots, b, \ldots, \text{max} \rangle$

min: Erstes Listenelement — Zeit $O(1)$

max: Letztes Listenelement — Zeit $O(1)$

```
rangeSearch(a, b)                                   // O(log n + |result|)
    result:= ⟨⟩
    h:= locate(a)
    while h → e ≤ b do
        result.pushBack(h → e)
        h:= h →next
    return result
```



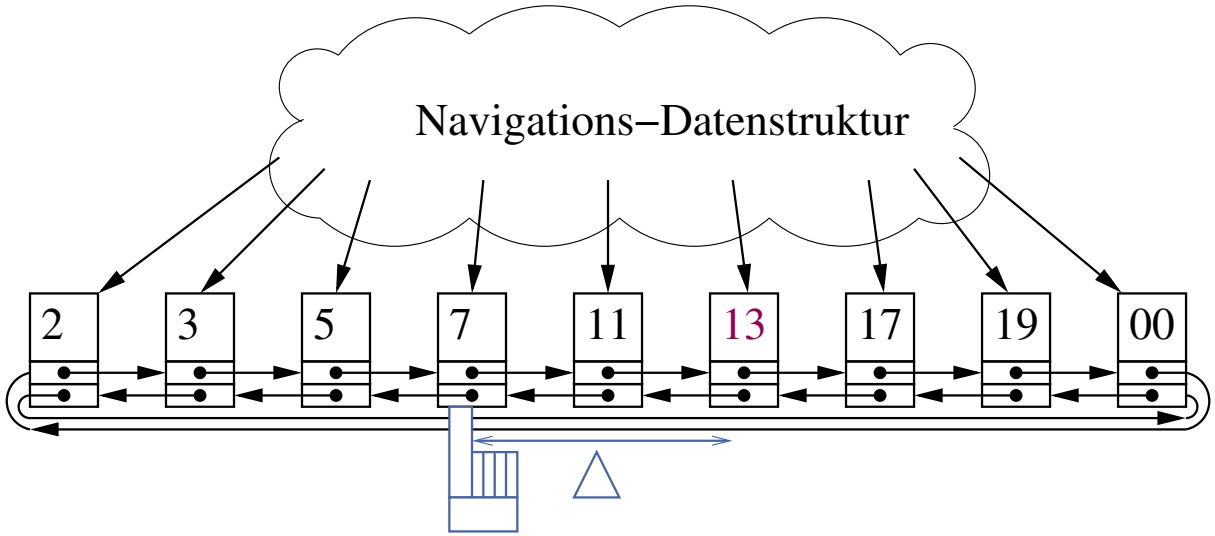
Noch mehr Operationen

- (re)**build**: Navigationsstruktur für sortierte Liste aufbauen $O(n)$
- $\langle w,\ldots,x\rangle$.**concat**$(\langle y,\ldots,z\rangle) = \langle w,\ldots,x,y,\ldots,z\rangle$ $O(\log n)$
- $\langle w,\ldots,x,y,\ldots,z\rangle$.**split**$(y) = (\langle w,\ldots,x\rangle,\langle y,\ldots,z\rangle)$ $O(\log n)$

Zählen: rank, select, rangeSize $O(\log n)$

Fingersuche: $\Delta =$ Abstand zu Fingerinfo

zusätzlicher Parameter für insert, remove, locate,... $O(\log n) \to \log \Delta$



Abgrenzung

Hash-Tabelle: nur insert, remove, find. Kein locate, rangeQuery

Sortiertes Feld: nur bulk-Updates. Aber:
Hybrid-Datenstruktur oder $\log \frac{n}{M}$ geometrisch wachsende statische Datenstrukturen

Prioritätsliste: nur insert, deleteMin, (decreaseKey, remove). Dafür: schnelles merge

Sortierte Folgen allgemein: die eierlegende Wollmilchdatenstruktur.
„Etwas“ langsamer als speziellere Datenstrukturen

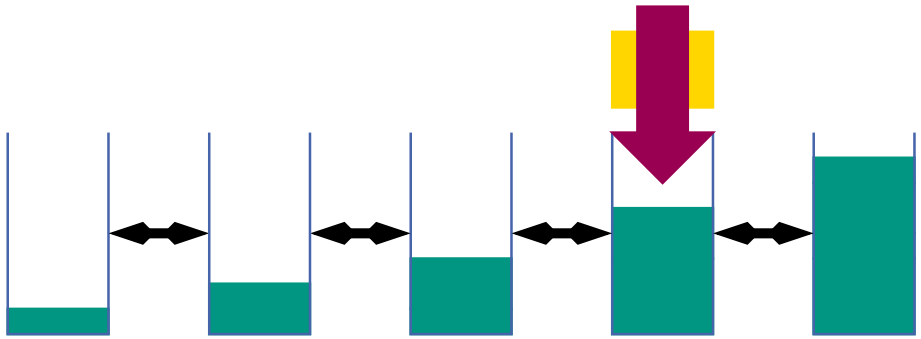
Sortierte Folgen – Anwendungen

- Best-First Heuristiken
- Alg. Geometrie: Sweepline-Datenstrukturen
- Datenbankindex
- ...

Anwendungsbeispiel: Best Fit Bin Packing

Procedure binPacking(s)
  B : SortedSequence // used bins sorted by free capacity
  foreach $e \in s$ by decreasing element size // sort
    if $\neg\exists b \in B : \text{free}(b) \geq e$ **then** B.insert(new bin)
    locate $b \in B$ with smallest $\text{free}(b) \geq e$
    insert e into bin b

Zeit: $O(|s| \log |s|)$
Qualität: „gut“. Details: nicht hier



Binäre Suchbäume

Blätter: Elemente

einer sortierten Folge.

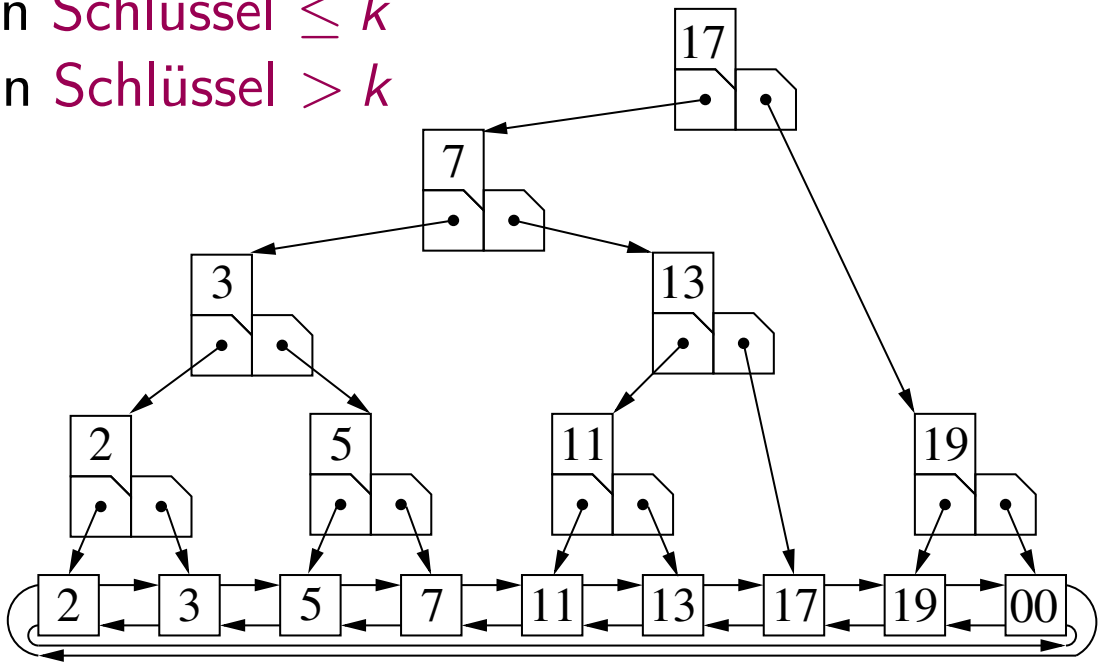
Innere Knoten $v = (k, \ell, r)$,

(Spalt-Schlüssel, linker Teilbaum, rechter Teilbaum).

Invariante:

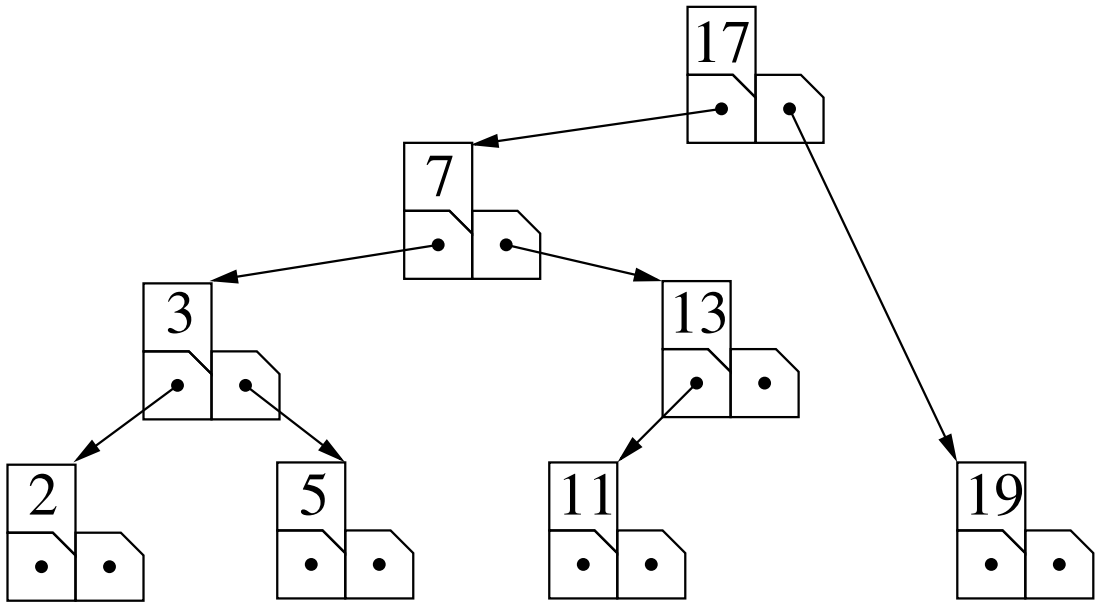
über ℓ erreichbare Blätter haben Schlüssel $\leq k$

über r erreichbare Blätter haben Schlüssel $> k$



Varianten, Bemerkungen

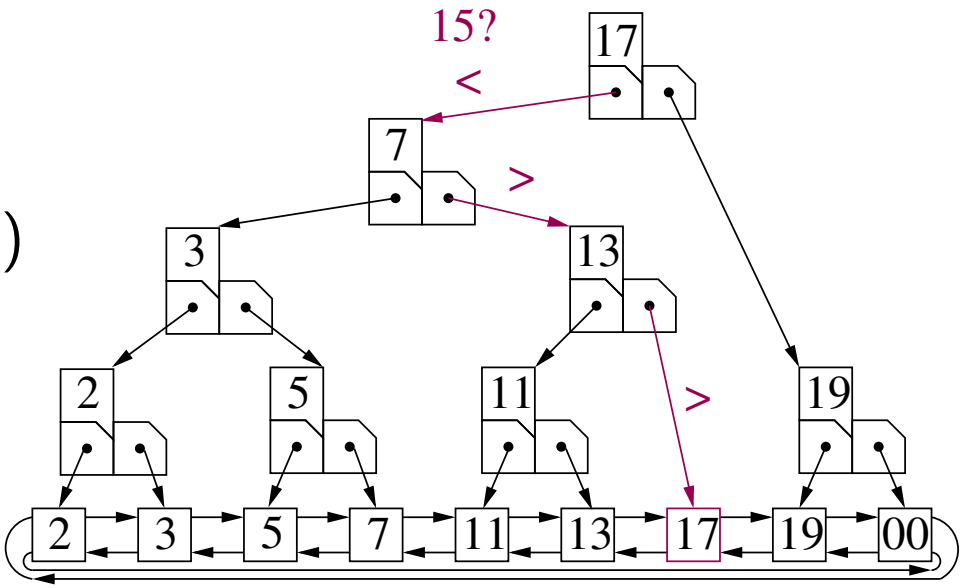
- Dummy-Element im Prinzip verzichtbar
- Oft speichern auch innere Knoten Elemente
- „Suchbaum“ wird oft als Synomym für „sortierte Folge“ verwendet. (Aber das vermischt (eine) Implementierung mit der Schnittstelle)



locate(k)

Idee: Benutze Spaltschlüssel x als Wegweiser.

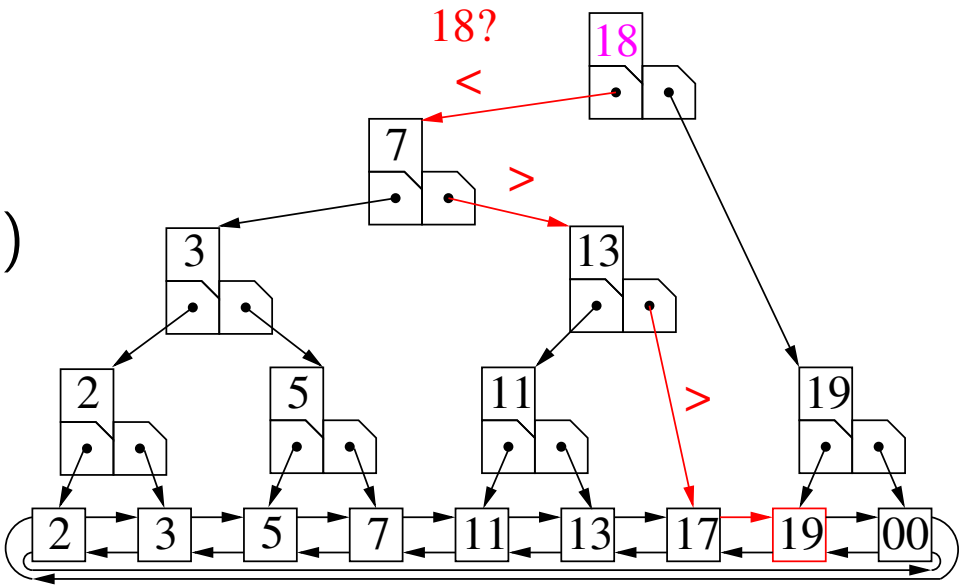
Function locate(k, x)
    if x is a leaf **then**
        if $k \leq x$ **then return** x
        else return $x \rightarrow$next
    if $k \leq x$ **then**
        return locate($k, x \rightarrow$left)
    else
        return locate($k, x \rightarrow$right)



locate(k) – anderes Beispiel

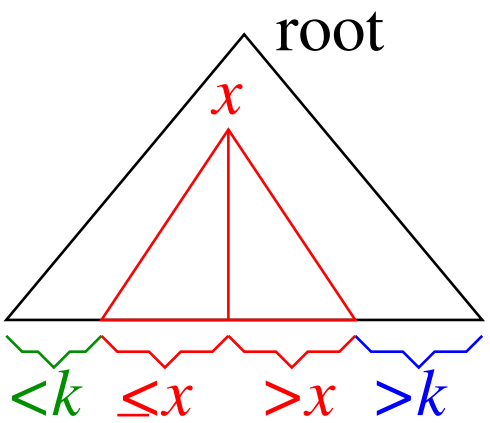
Idee: Benutze Spaltschlüssel x als Wegweiser.

Function locate(k, x)
  if x is a leaf **then**
    if $k \leq x$ **then return** x
    else return $x \rightarrow$next
  if $k \leq x$ **then**
    return locate($k, x \rightarrow$left)
  else
    return locate($k, x \rightarrow$right)



Invariante von locate(k)

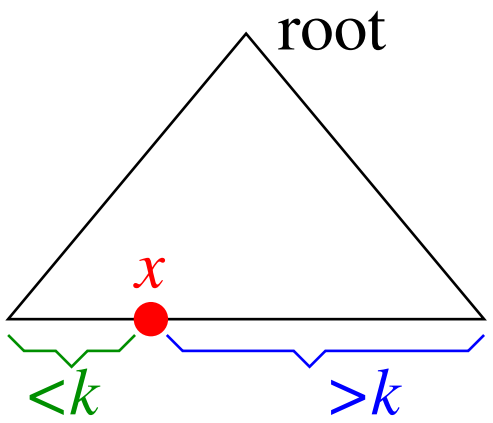
Function locate(k, x)
  if x is a leaf **then**
    if $k \leq x$ **then return** x
    else return $x \rightarrow$next
  if $k \leq x$ **then**
    return locate($k, x \rightarrow$left)
  else
    return locate($k, x \rightarrow$right)



Invariante: Sei X die Menge aller von x erreichbaren Listenelemente.
Listenelemente links von X sind $< k$
Listenelemente rechts von X sind $> k$

Ergebnisberechnung von locate(k)

```
Function locate(k, x)
    if x is a leaf then
        if k ≤ x then return x
        else return x →next
    if k ≤ x then
        return locate(k, x →left)
    else
        return locate(k, x →right)
```



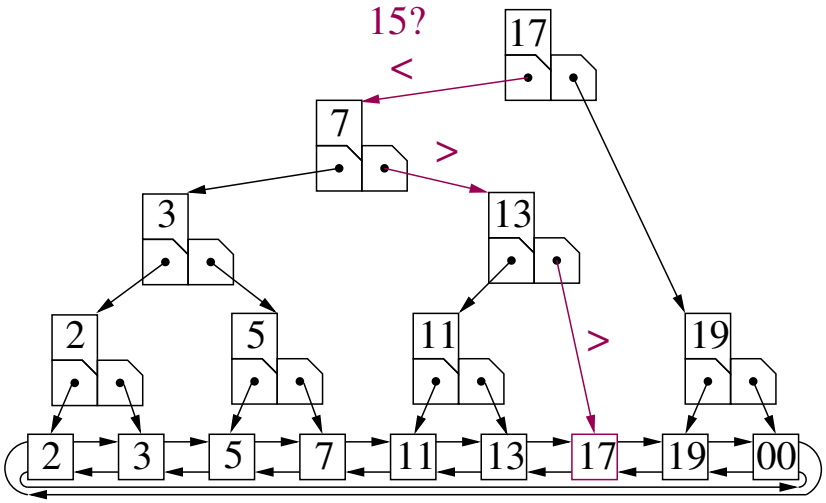
Fall $k = x$: return x — Bingo!

Fall $k < x$: return x — links ist es auch nicht

Fall $k > x$: return $x \rightarrow$next — nächstes ist $> k$ und k gibt es nicht

Laufzeit von locate(k)

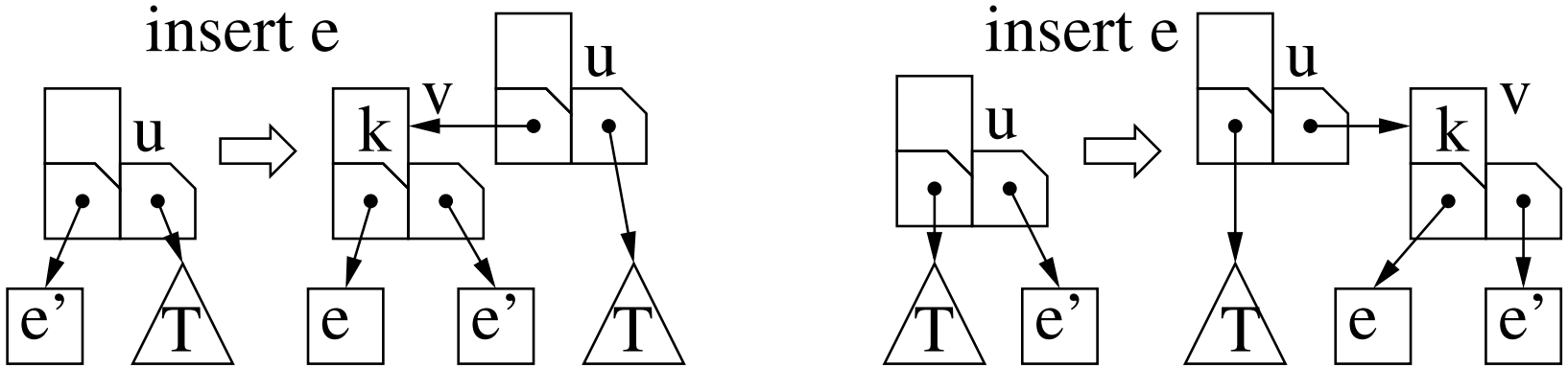
Function locate(k, x)
  if x is a leaf **then**
    if $k \leq x$ **then return** x
    else return $x \rightarrow$next
  if $k \leq x$ **then**
    return locate($k, x \rightarrow$left)
  else
    return locate($k, x \rightarrow$right)



Laufzeit: O(Höhe).
Bester Fall: perfekt balanciert, d. h. Tiefe $= \lfloor \log n \rfloor$
Schlechtester Fall: Höhe n

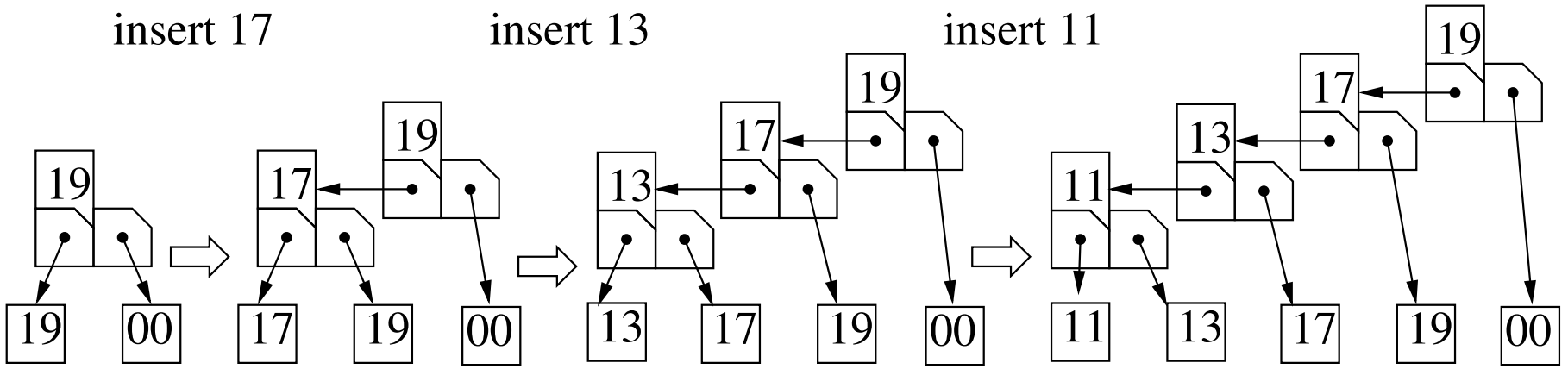
Naives Einfügen

Zunächst wie locate(e). Sei e' gefundenes Element, u der Elterknoten



k=key(e)

Beispiel



Problem: Der Baum wird beliebig unbalanciert.

$\rightsquigarrow$ langsam