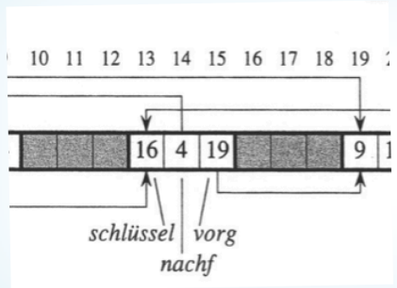
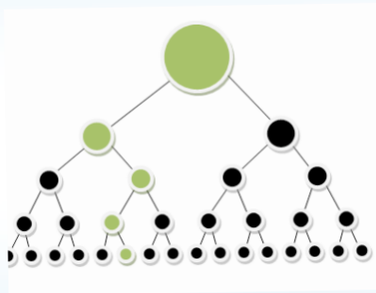





Kapitel 4: Dynamische Datenstrukturen

Algorithmen und Datenstrukturen
WS 2024/25

Prof. Dr. Sándor Fekete

Wie verwalten wir dynamische Mengen von Objekten?

Waschkorb

4.1 Grundoperationen

Aufgabenstellung:

- ***Verwalten einer Menge S von Objekten***
- ***Ausführen von verschiedenen Operationen (s.u.)***

Im Folgenden:

S Menge von Objekten

k Wert eines Elements (“Schlüssel”)

x Zeiger auf Element

NIL spezieller, “leerer” Zeiger

<u>SEARCH(S,k):</u> **“Suche in S nach k”**

Durchsuche die Menge S nach einem Element von Wert k.

Ausgabe: Zeiger x, falls x existent
NIL, falls kein Element Wert k hat.

4.1 Grundoperationen

<u>INSERT(S,x):</u> **“Füge x in S ein”**

Erweitere S um das Element, das unter der Adresse x steht.

4.1 Grundoperationen

DELETE(S,x): **“Entferne x aus S”**

Lösche das unter der Adresse x stehende Element aus der Menge S.

4.1 Grundoperationen

Langsam:

- ***O(n):*** ***lineare Zeit***

  Alle Objekte anschauen

Sehr schnell:

- ***O(1):*** ***konstante Zeit***

  Immer gleich schnell, egal wie groß S ist.

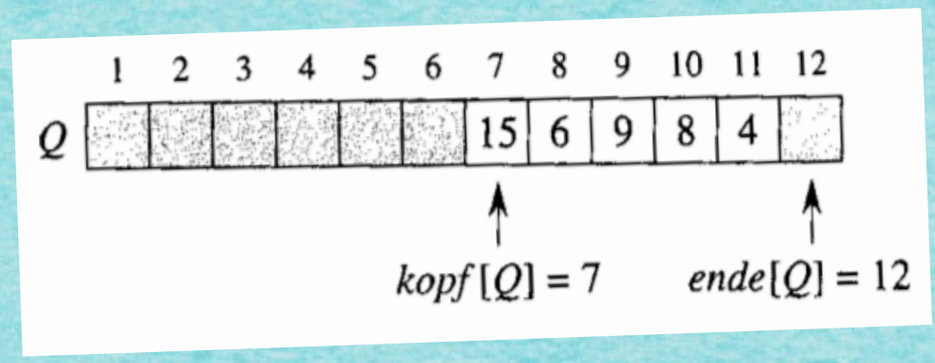
Schnell:

- ***O(log n):*** ***logarithmische Zeit***

  Wiederholtes Halbieren

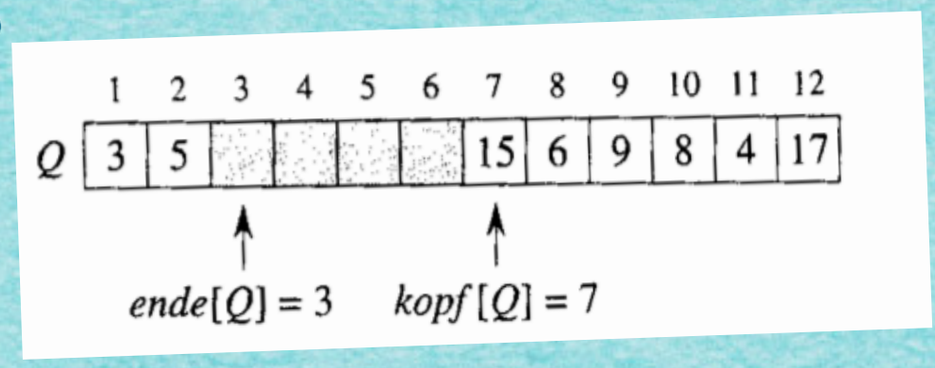
4.2 Stapel und Warteschlange

OUT

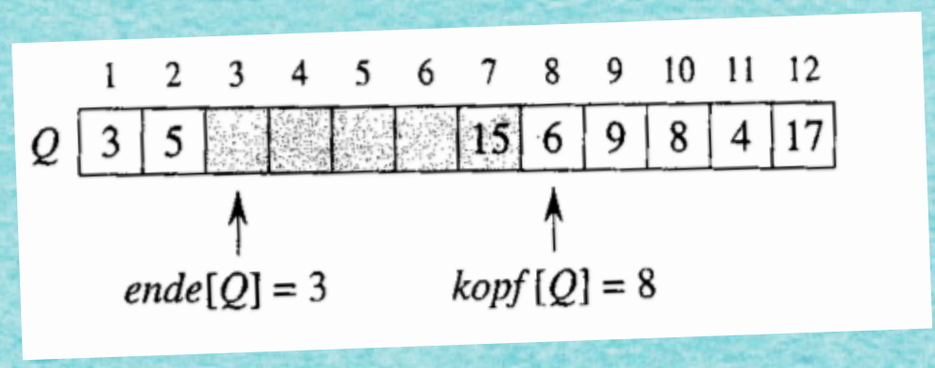


IN

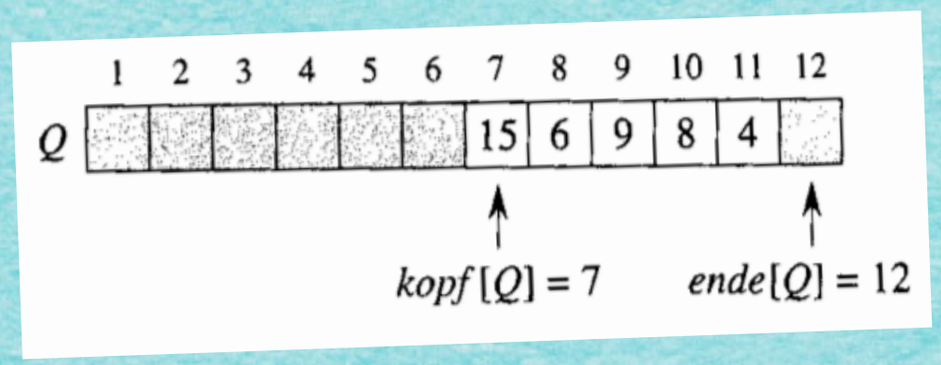
Enqueue: 17, 3, 5

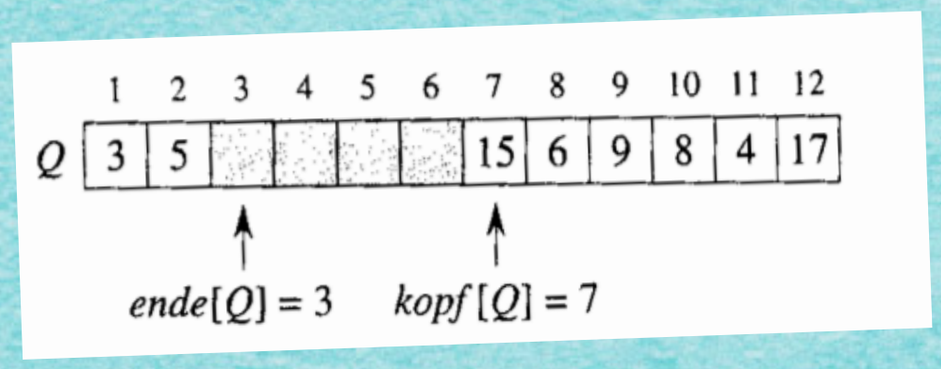


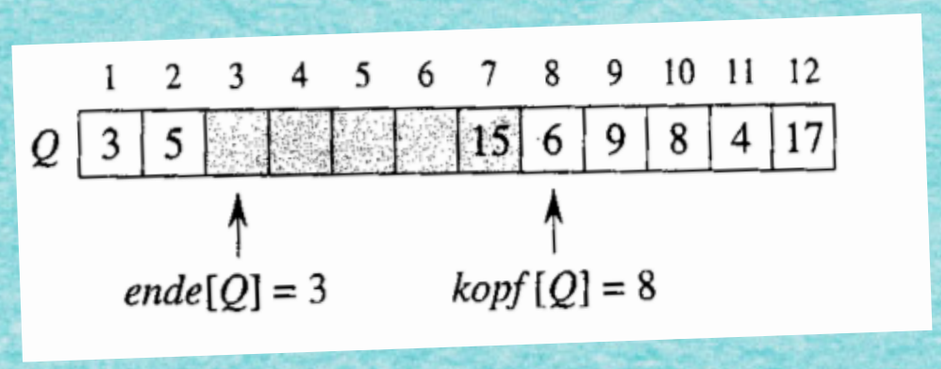
Dequeue:



Warteschlange auf Array umgesetzt



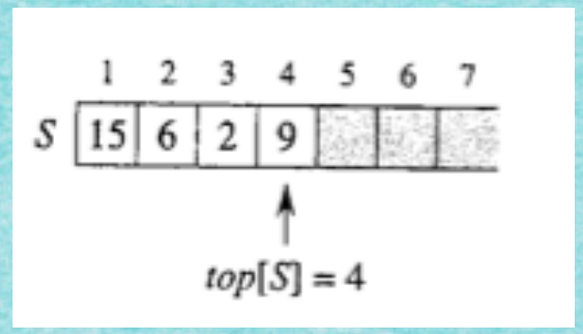





```
ENQUEUE(Q, x)
1  Q[ende[Q]] ← x
2  if ende[Q] = länge[Q]
3     then ende[Q] ← 1
4     else ende[Q] ← ende[Q] + 1
```

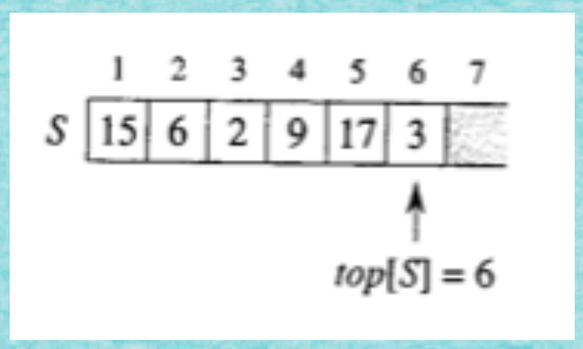
```
DEQUEUE(Q)
1  x ← Q[kopf[Q]]
2  if kopf[Q] = länge[Q]
3     then kopf[Q] ← 1
4     else kopf[Q] ← kopf[Q] + 1
5  return x
```

Stack auf Array umgesetzt



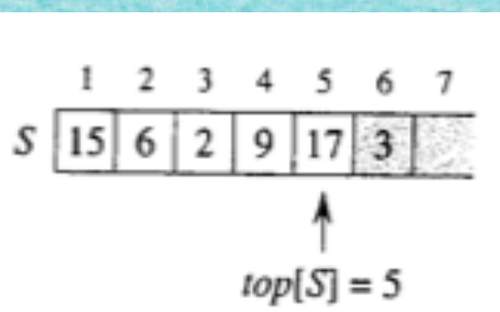

```
STACK-EMPTY(S)
1  if top[S] = 0
2       then return WAHR
3       else  return FALSCH
```

Push: 17, 3

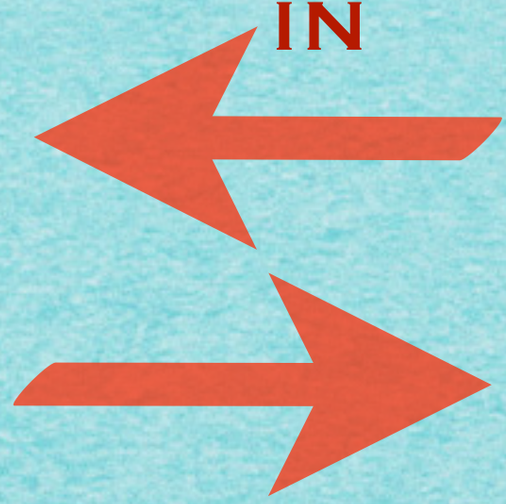



```
PUSH(S, x)
1  top[S] ← top[S] + 1
2  S[top[S]] ← x
```

Pop




```
POP(S)
1  if STACK-EMPTY(S)
2       then error "Unterlauf"
3       else  top[S] ← top[S] − 1
4             return S[top[S] + 1]
```

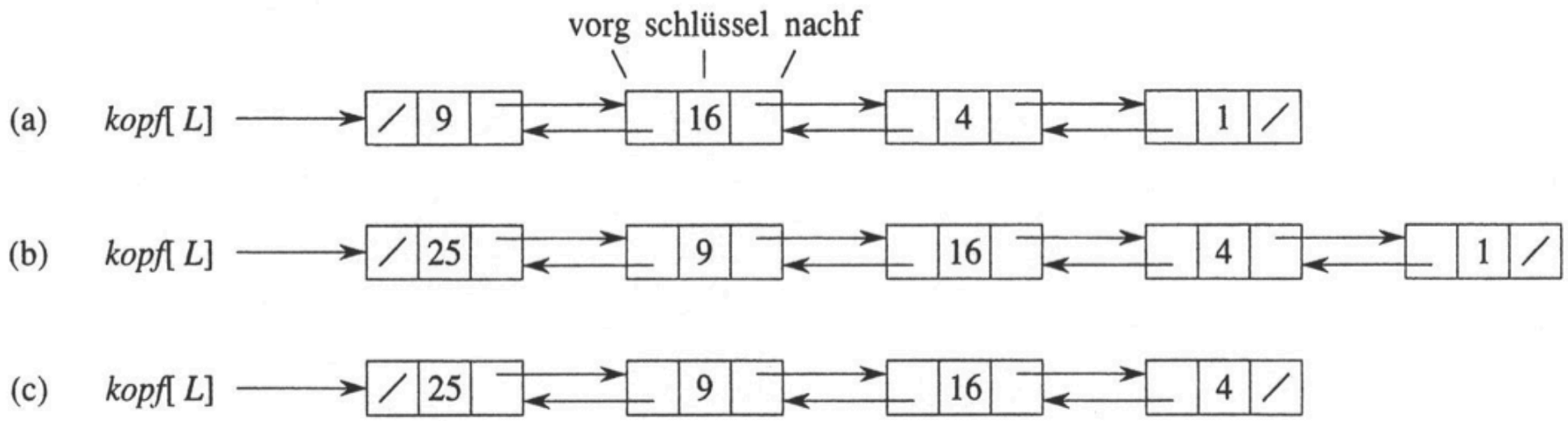


4.3 Verkettete Listen

Idee:

Ordne Objekte nicht explizit in aufeinanderfolgenden Speicherzellen an, sondern gib jeweils Vorgänger und Nachfolger an.

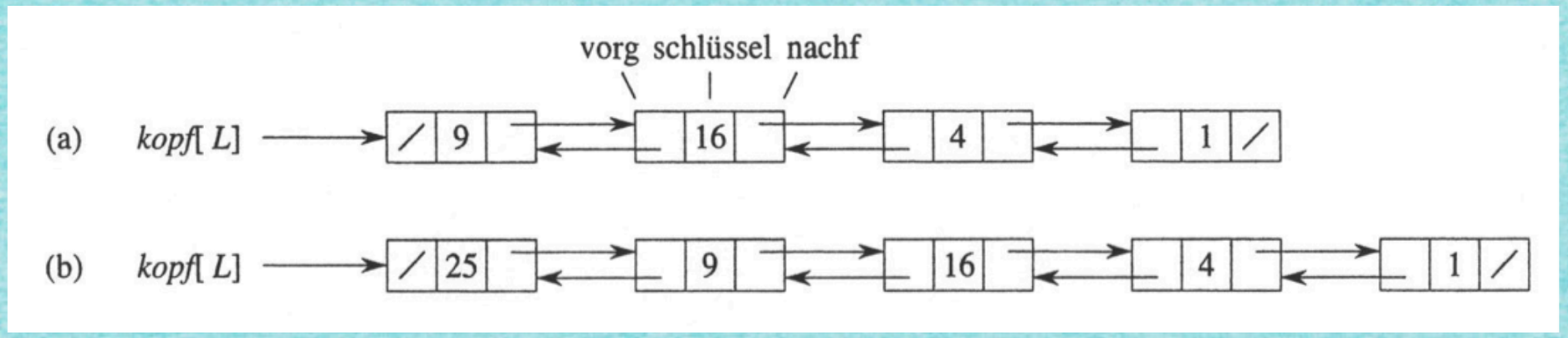
Struktur einer doppelt verketteten Liste



- **Füge vorne das Element mit Schlüssel 25 ein.**

- **Finde ein Element mit Schlüssel 1 und lösche es.**

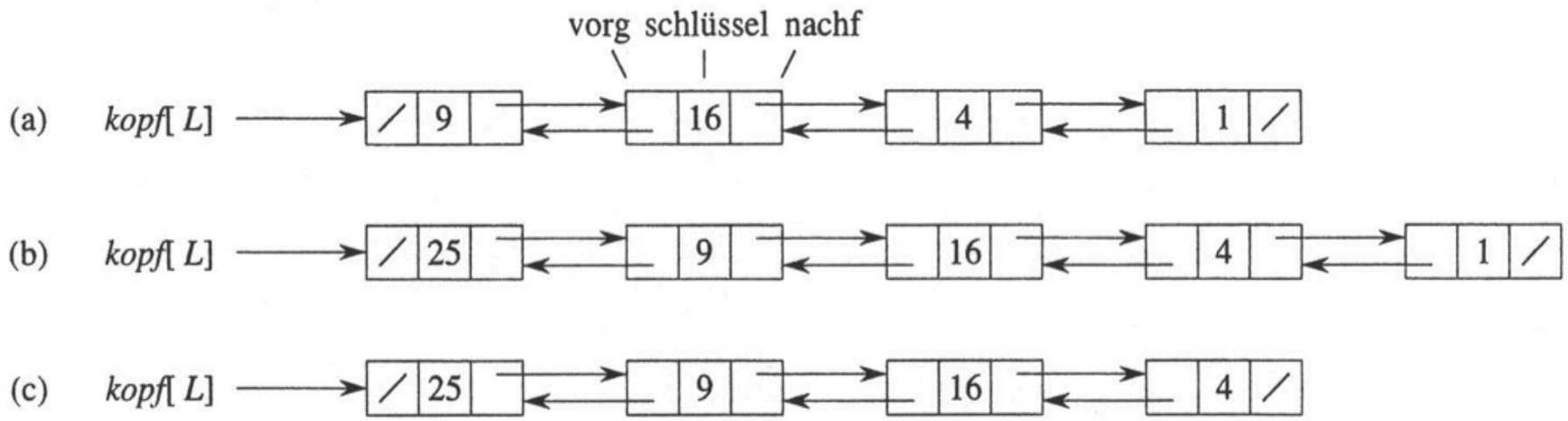
Einfügen in eine doppelt verkettete Liste




```
LIST-INSERT(L, x)
1  nachf[x] ← kopf[L]
2  if kopf[L] ≠ NIL
3      then vorg[kopf[L]] ← x
4  kopf[L] ← x
5  vorg[x] ← NIL
```

<u>Laufzeit:</u> O(1)

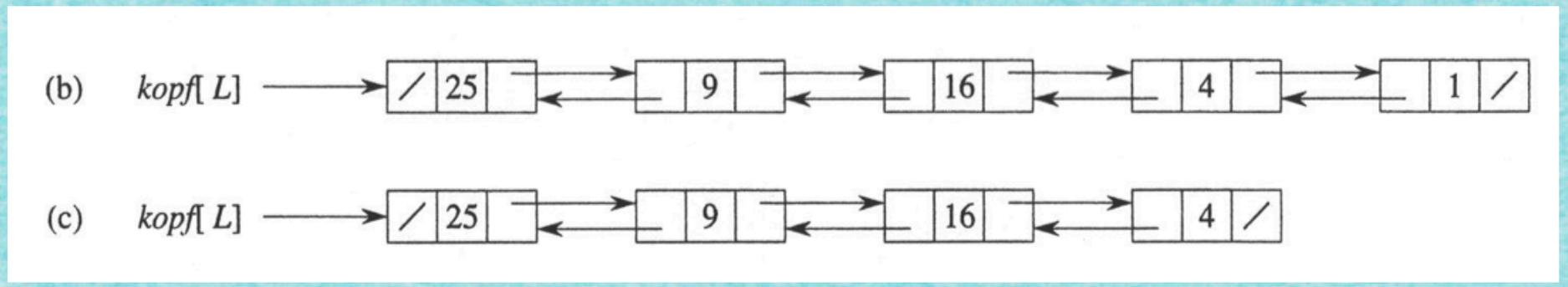
Struktur einer doppelt verketteten Liste



- **Füge vorne das Element mit Schlüssel 25 ein.**

- **Finde ein Element mit Schlüssel 1 und lösche es.**

Löschen aus einer doppelt verketteten Liste



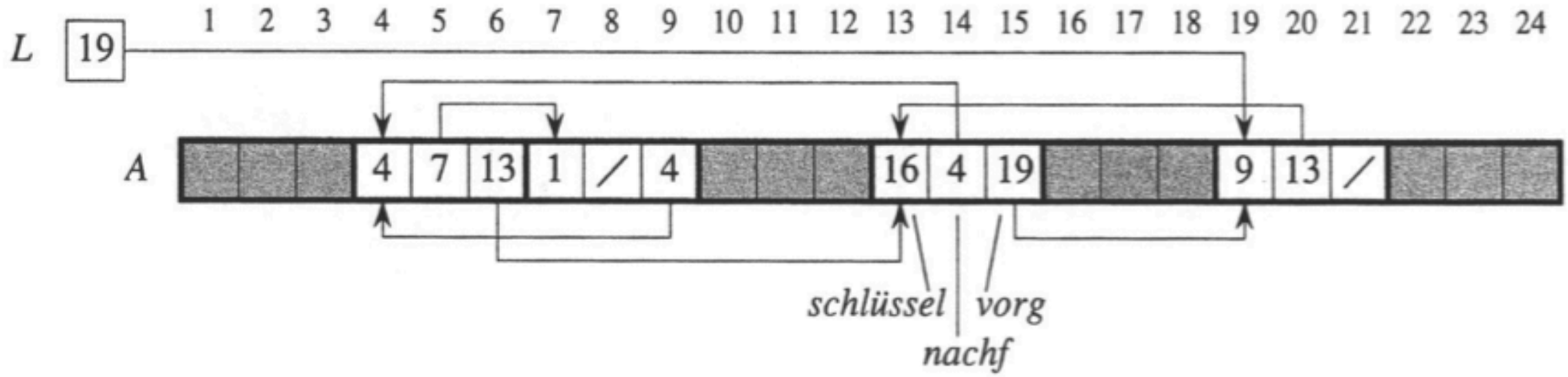
Laufzeit: O(n)

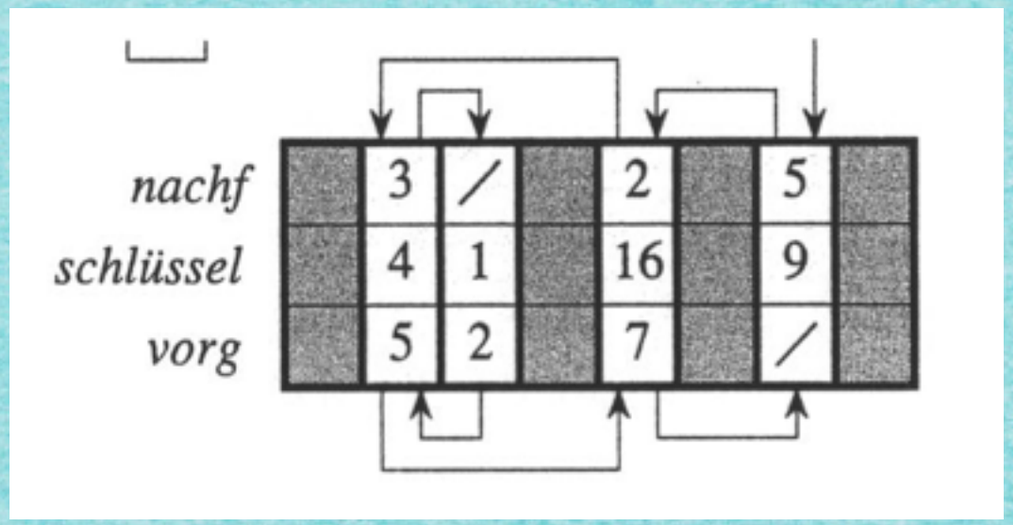
```
LIST-SEARCH(L, k)
1  x ← kopf[L]
2  while x ≠ NIL und schlüssel[x] ≠ k
3        do x ← nachf[x]
4  return x
```

Laufzeit: O(1)

```
LIST-DELETE(L, x)
1  if vorg[x] ≠ NIL
2     then nachf[vorg[x]] ← nachf[x]
3     else kopf[L] ← nachf[x]
4  if nachf[x] ≠ NIL
5     then vorg[nachf[x]] ← vorg[x]
```

Speicherung kann irgendwo erfolgen!





4.4 Binäre Suche

<u>Aufgabenstellung:</u>

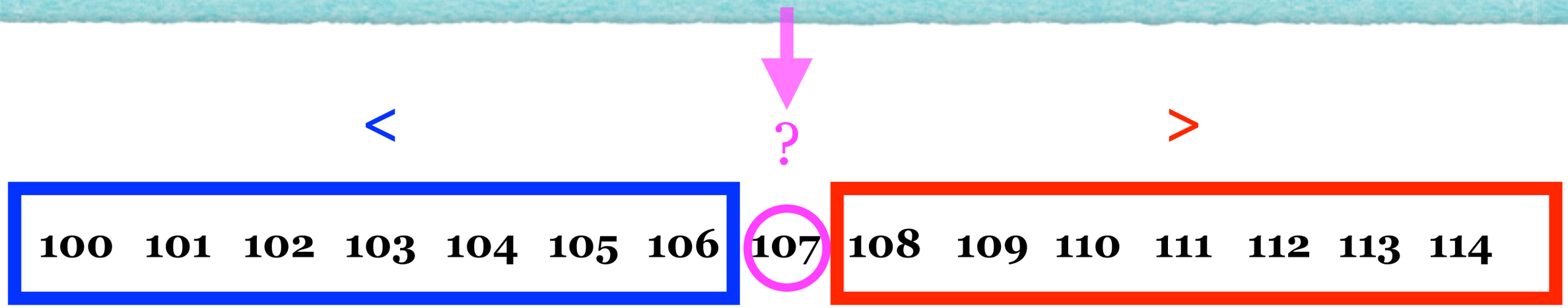
- ***Rate eine Zahl zwischen 100 und 114!***

100 101 102 103 104 105 106 107 108 109 110 111 112 113 114

4.4 Binäre Suche

<u>Aufgabenstellung:</u>

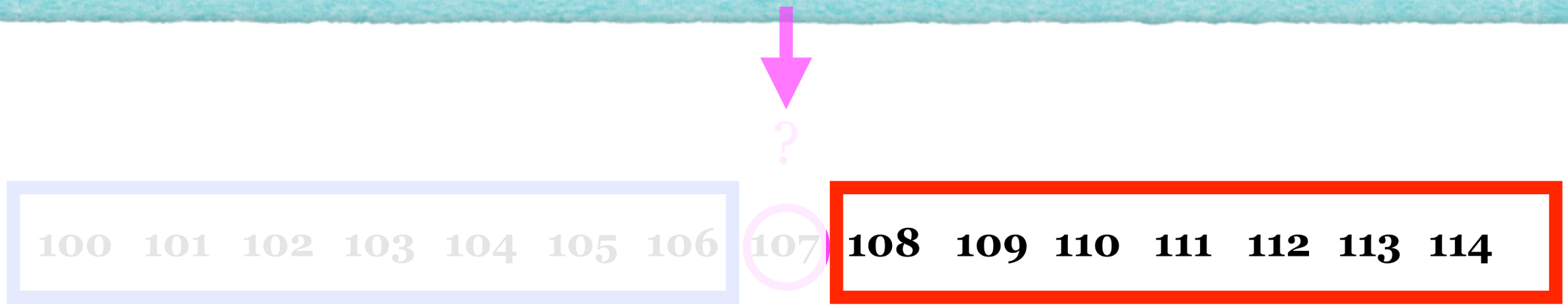
- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

<u>Aufgabenstellung:</u>

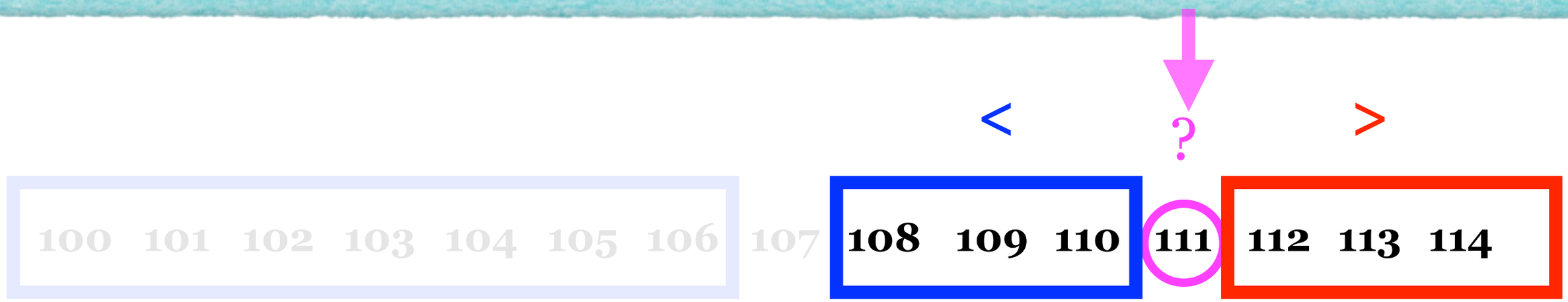
- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

<u>Aufgabenstellung:</u>

- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

<u>Aufgabenstellung:</u>

- ***Rate eine Zahl zwischen 100 und 114!***

4.4 Binäre Suche

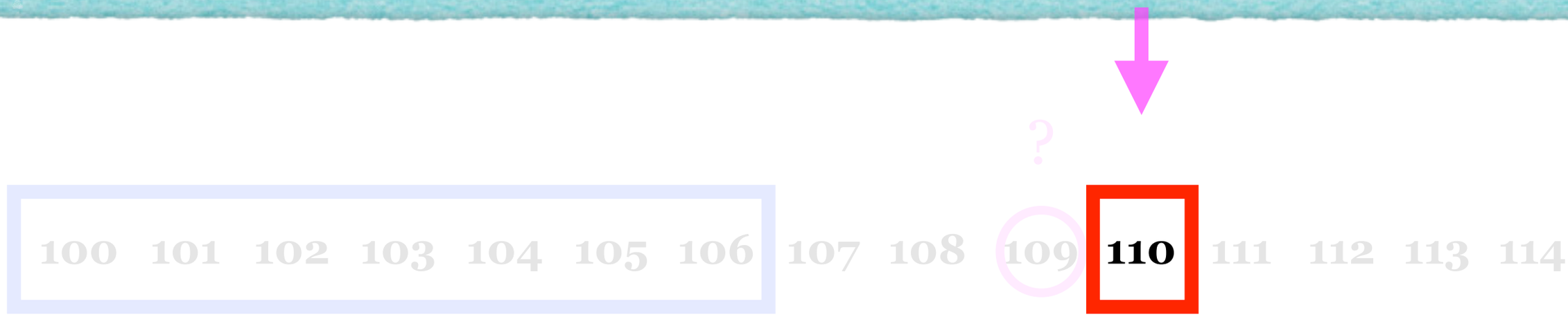
<u>Aufgabenstellung:</u>

- ***Rate eine Zahl zwischen 100 und 114!***

4.4 Binäre Suche

<u>Aufgabenstellung:</u>

- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

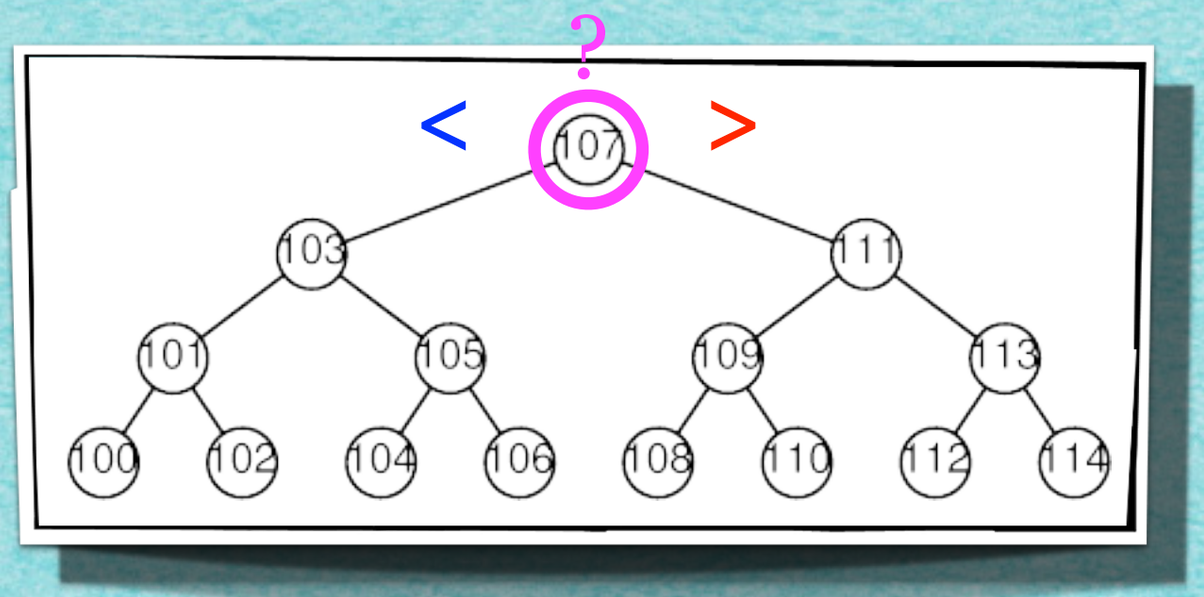
<u>Aufgabenstellung:</u>

- ***Rate eine Zahl zwischen 100 und 114!***

4.4 Binäre Suche

<u>Aufgabenstellung:</u>

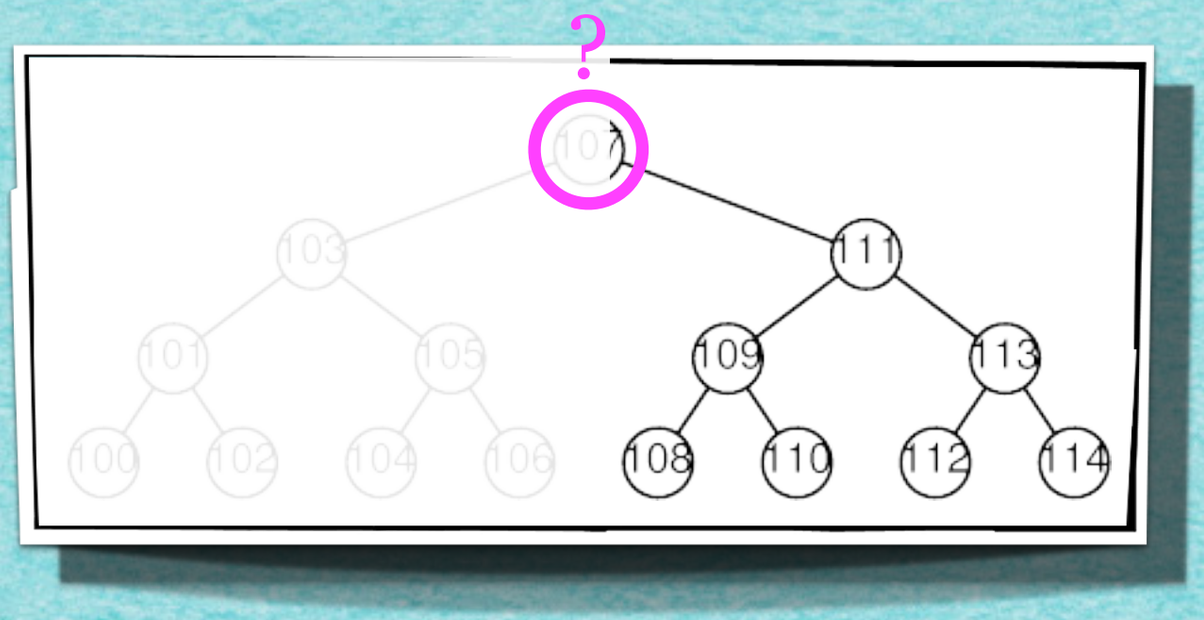
- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

<u>Aufgabenstellung:</u>

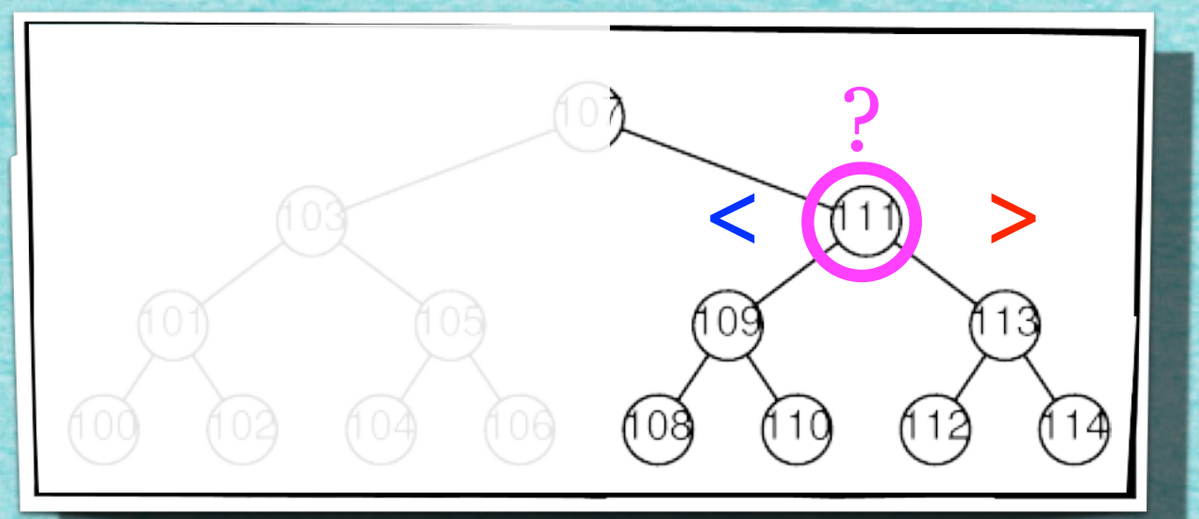
- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

<u>Aufgabenstellung:</u>

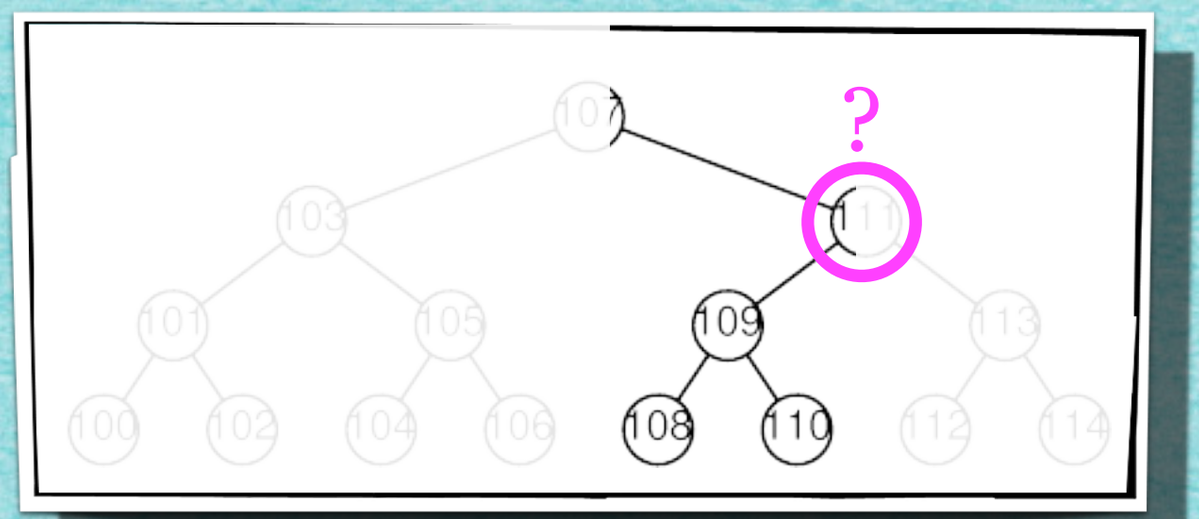
- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

<u>Aufgabenstellung:</u>

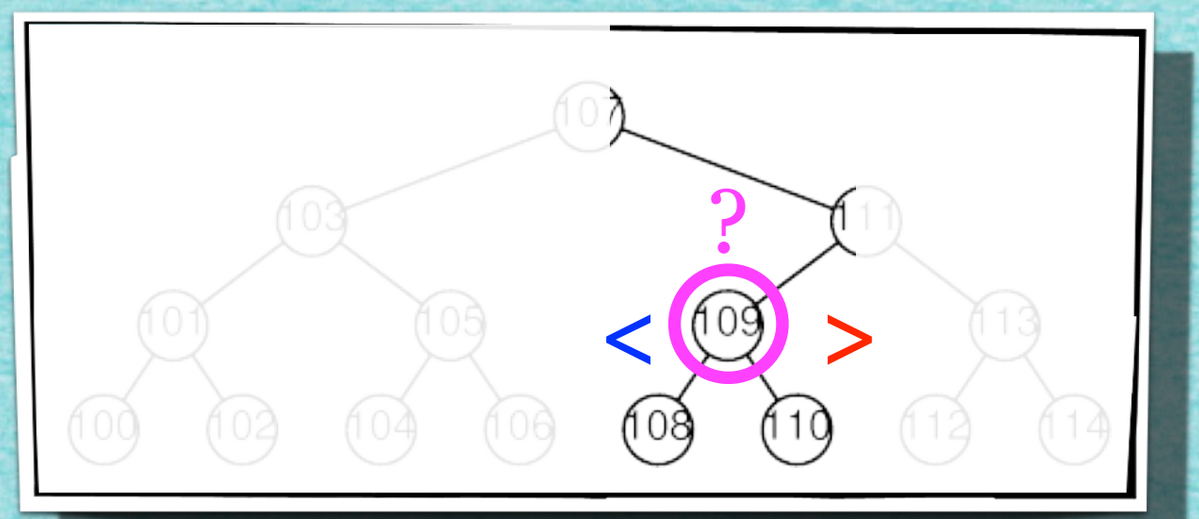
- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

<u>Aufgabenstellung:</u>

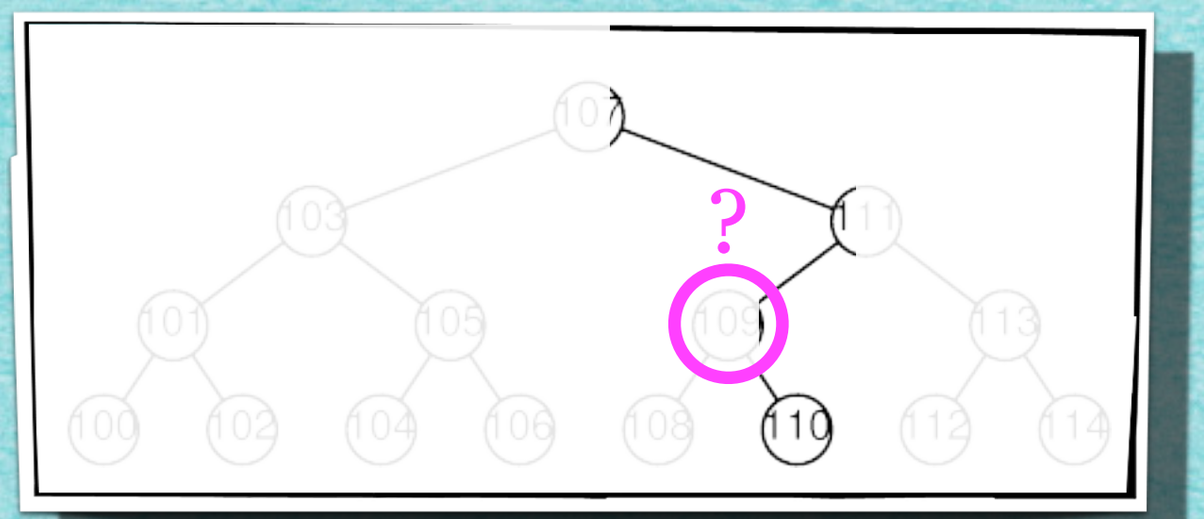
- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

<u>Aufgabenstellung:</u>

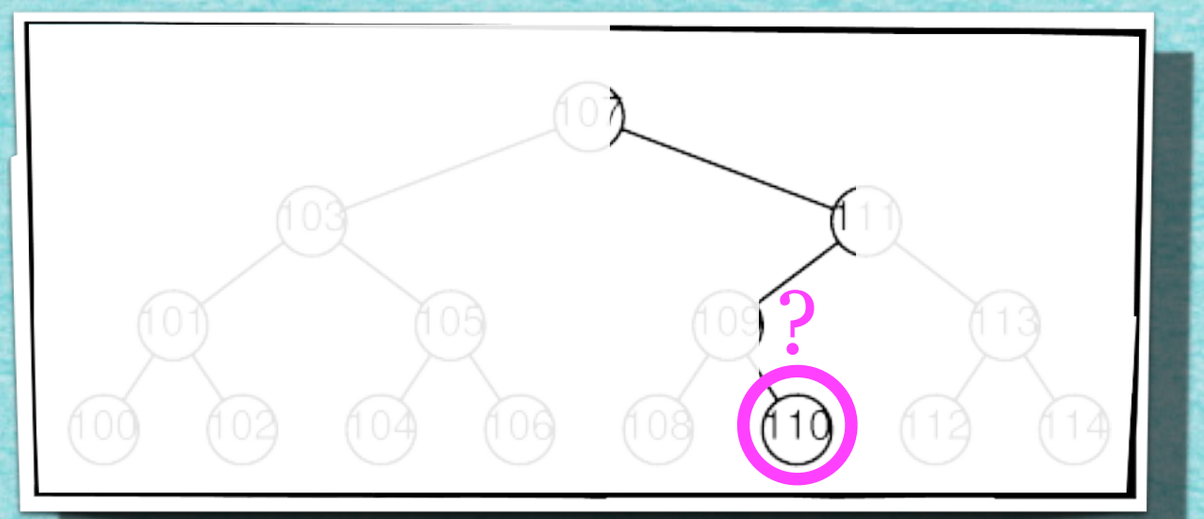
- ***Rate eine Zahl zwischen 100 und 114!***



4.4 Binäre Suche

<u>Aufgabenstellung:</u>

- ***Rate eine Zahl zwischen 100 und 114!***



Algorithmus 4.1

INPUT: Sortierter Array mit Einträgen S[I], Suchwert WERT,
linke Randposition LINKS, rechte Randposition RECHTS,

OUTPUT: Position von WERT zwischen Arraypositionen LINKS und RECHTS, falls existent

BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1. WHILE (LINKS≤RECHTS) DO {
    1.1. MITTE:= $\left\lfloor \frac{\text{LINKS} + \text{RECHTS}}{2} \right\rfloor$
    1.2. IF (S[MITTE]=WERT) THEN
        1.2.1. RETURN MITTE
    1.3. ELSEIF (S[MITTE]>WERT) THEN
        1.3.1. RECHTS:=MITTE-1
    1.4. ELSEIF
        1.4.1. LINKS:=MITTE+1
    }
2. RETURN "WERT nicht gefunden!"

```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=42

		LINKS											RECHTS	
i		1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]		1	2	5	6	13	17	28	33	42	47	52	64	89	96

```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=42

		LINKS					MITTE						RECHTS
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	1	2	5	6	13	17	28	33	42	47	52	64	89	96

```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=42

	i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
								MITTE	LINKS						RECHTS
S[i]		1	2	5	6	13	17	28	33	42	47	52	64	89	96

```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=42

								LINKS			MITTE			RECHTS
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	1	2	5	6	13	17	28	33	42	47	52	64	89	96

```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=42

								LINKS		RECHTS	MITTE			
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	1	2	5	6	13	17	28	33	42	47	52	64	89	96

```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=42

								LINKS	MITTE	RECHTS				
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	1	2	5	6	13	17	28	33	42	47	52	64	89	96

RETURN 9

```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=7

		LINKS											RECHTS
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	1	2	5	6	13	17	28	33	42	47	52	64	89	96

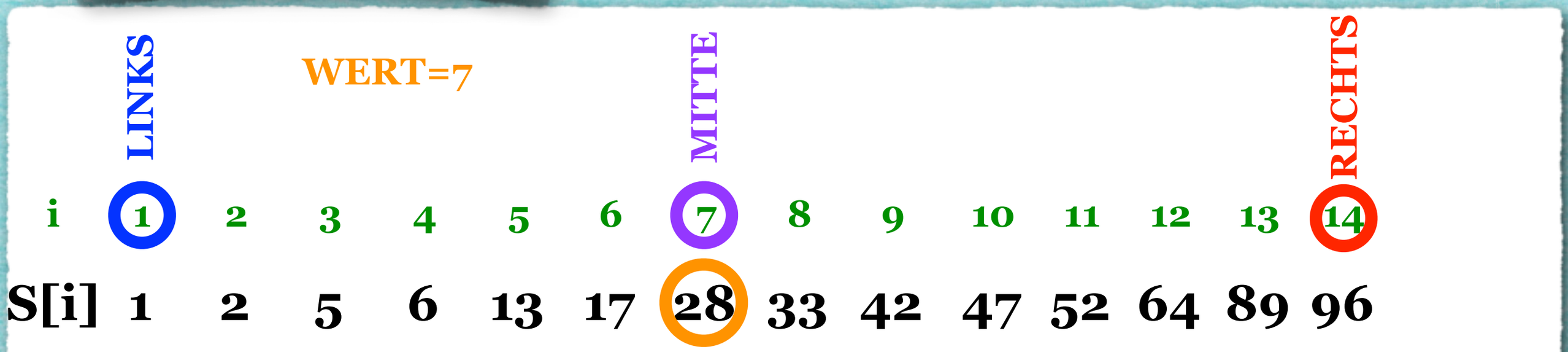
```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***



```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=7

	LINKS					RECHTS	MITTE						
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	1	2	5	6	13	17	28	33	42	47	52	64	89	96

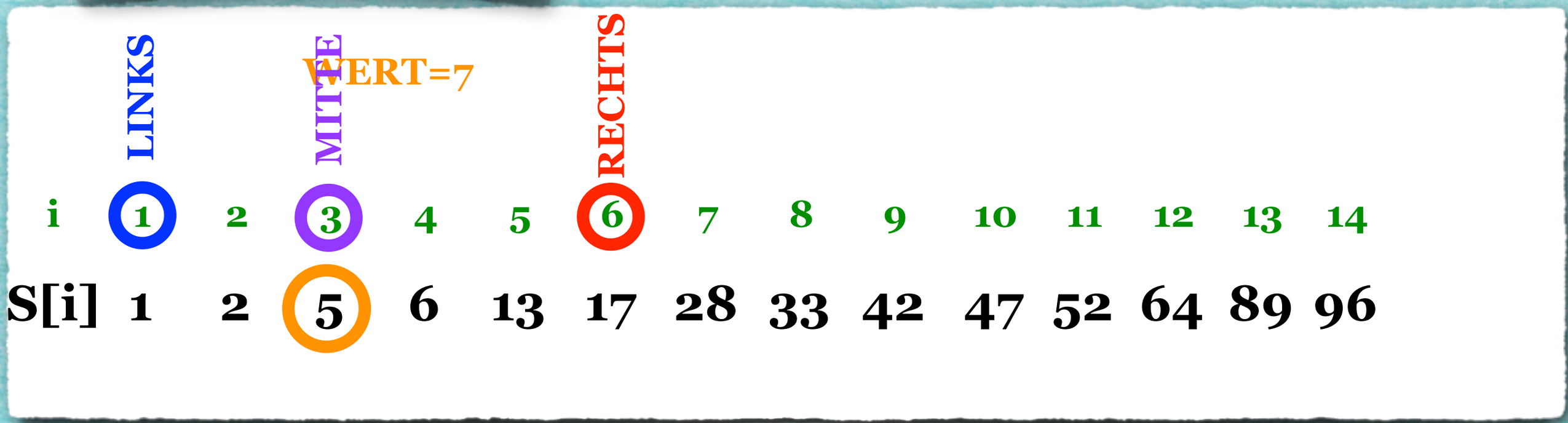
```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***



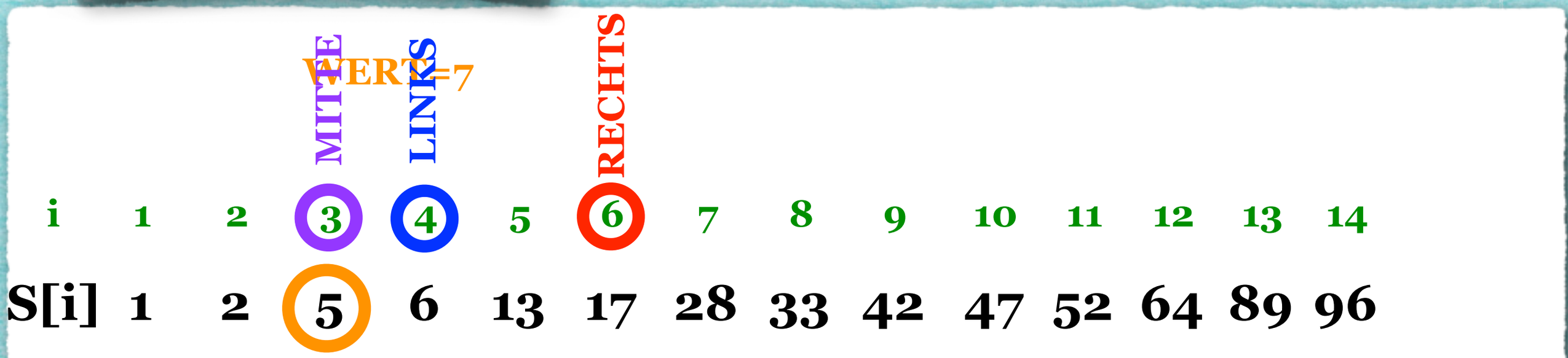
```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***



```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=7

	LINKS	MITTE	RECHTS										
i	4	5	6	1	2	3	7	8	9	10	11	12	13	14
S[i]	6	13	17	1	2	5	28	33	42	47	52	64	89	96

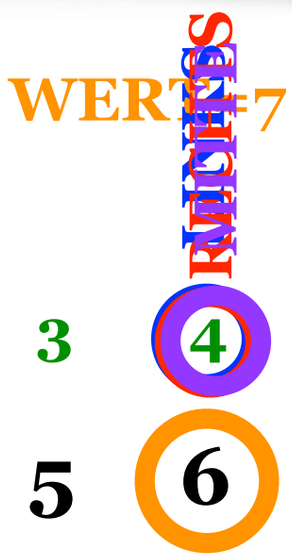
```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	1	2	5	6	13	17	28	33	42	47	52	64	89	96

```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=7

	RECHTES		LINKS										
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	1	2	5	6	13	17	28	33	42	47	52	64	89	96

```
BINÄRESUCHE(S,WERT,LINKS,RECHTS)

1.  WHILE (LINKS≤RECHTS) DO {
        1.1.  MITTE:= ⌊(LINKS + RECHTS) / 2⌋
        1.2.  IF (S[MITTE]=WERT) THEN
            1.2.1. RETURN MITTE
        1.3.  ELSEIF (S[MITTE]>WERT) THEN
            1.3.1. RECHTS:=MITTE-1
        1.4.  ELSEIF
            1.4.1. LINKS:=MITTE+1
    }
2.  RETURN "WERT nicht gefunden!"
```

Binäre Suche

Aufgabenstellung:

- ***Finde eine gesuchte Zahl in der gegebenen sortierten Menge!***

WERT=7

RECHTES

LINKS

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
S[i]	1	2	5	6	13	17	28	33	42	47	52	64	89	96

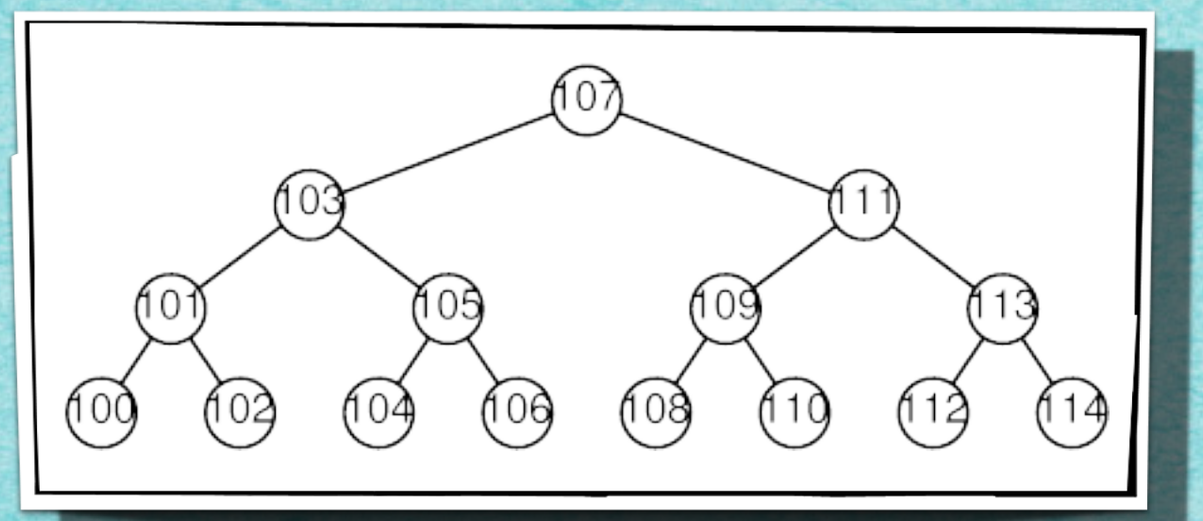
WERT nicht gefunden!

Satz 4.2

Die binäre Suche terminiert in O(log(RECHTS-LINKS)) Schritten (für RECHTS>LINKS).

Beweis:

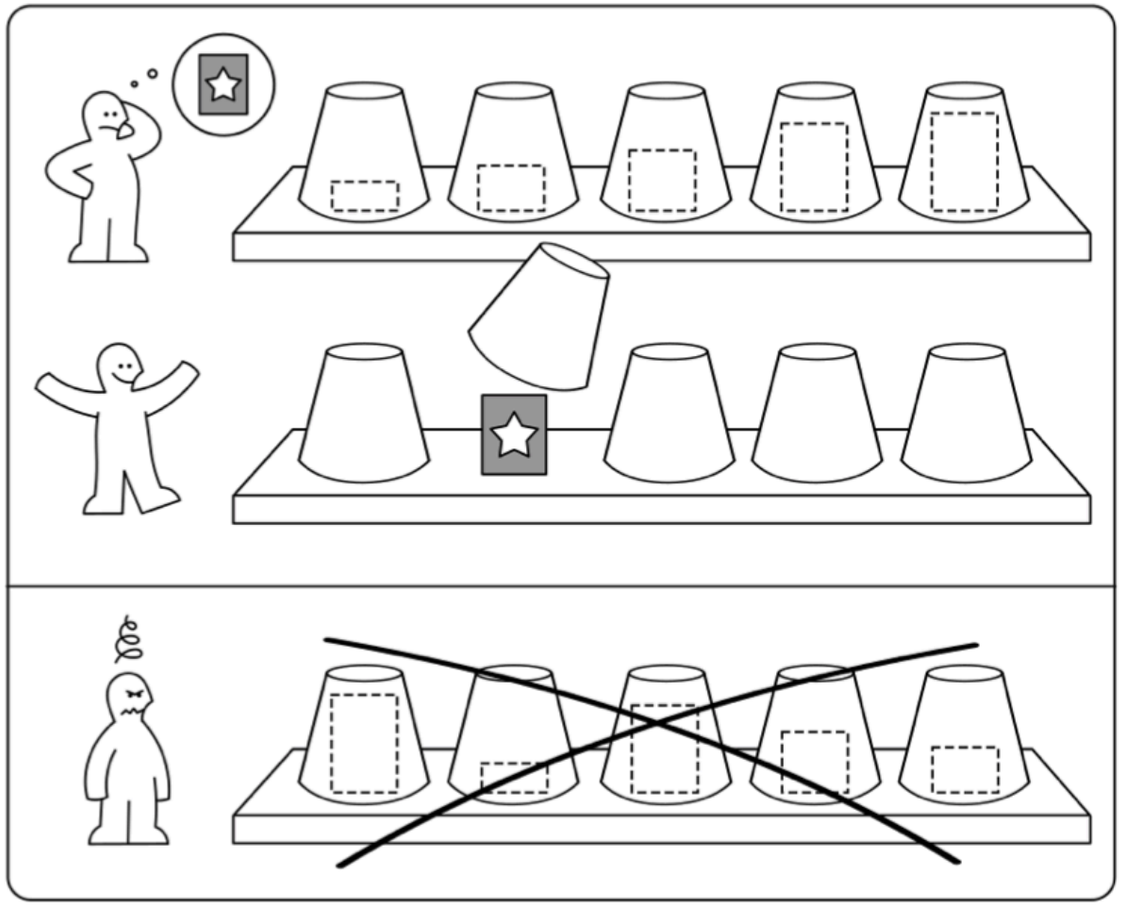
Selbst!



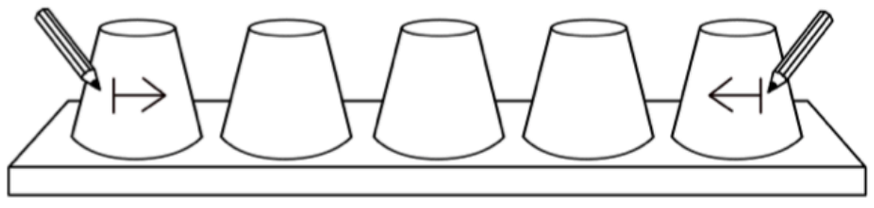
BINÄRY SEARCH

idea-instructions.com/binary-search/
v1.1, CC by-nc-sa 4.0

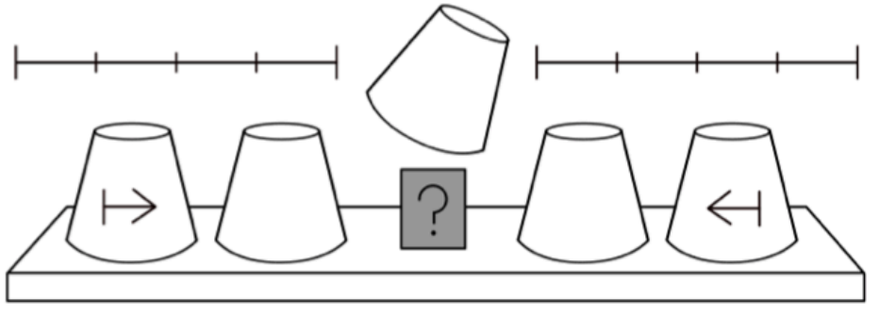
IDEA



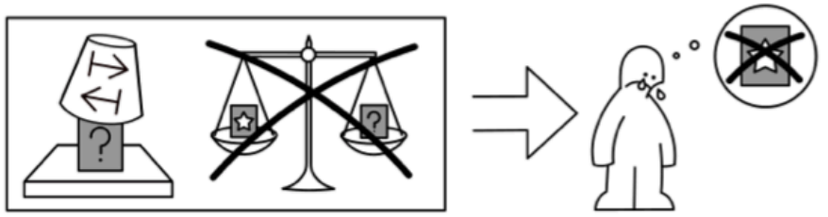
1



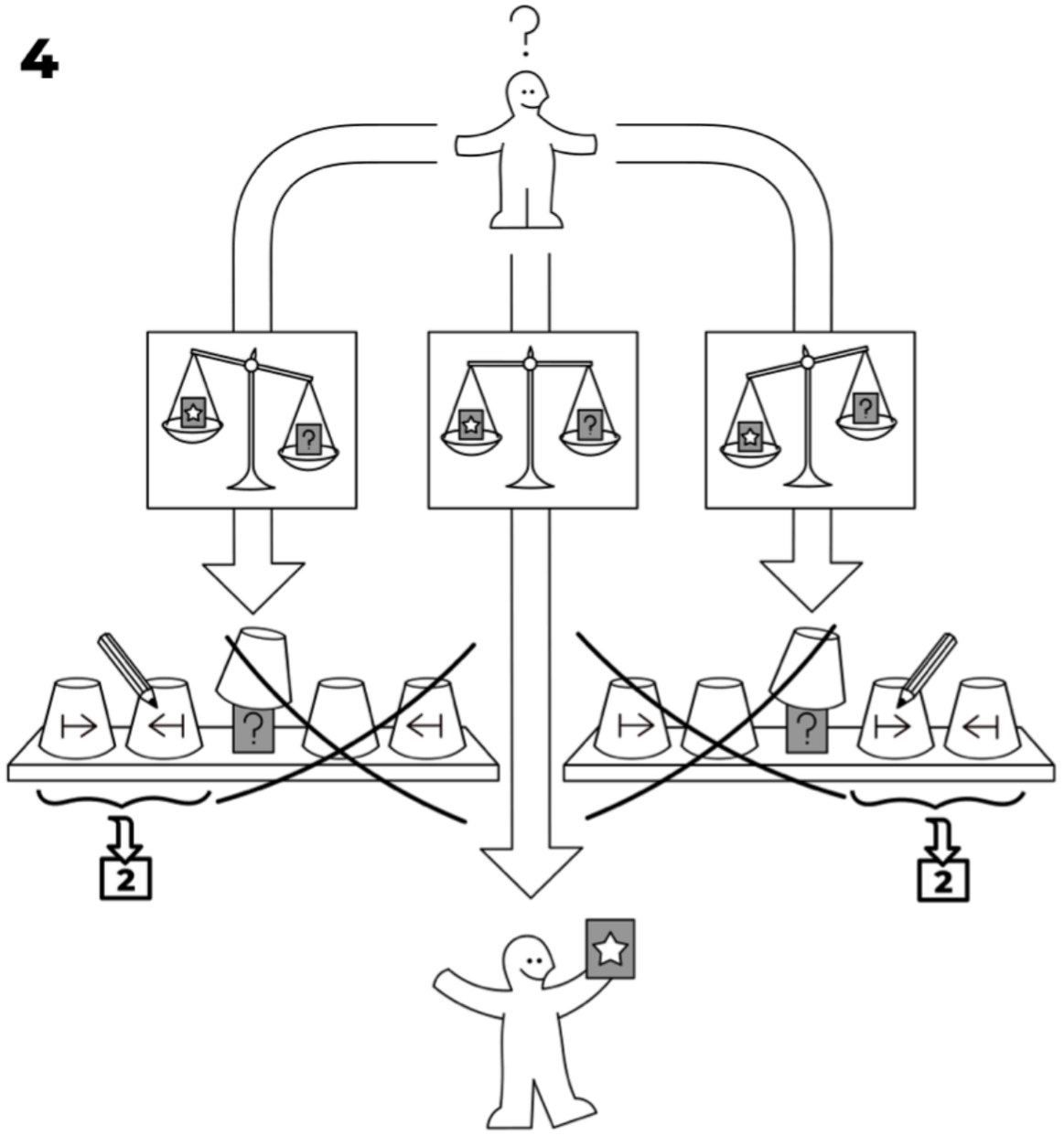
2



3



4

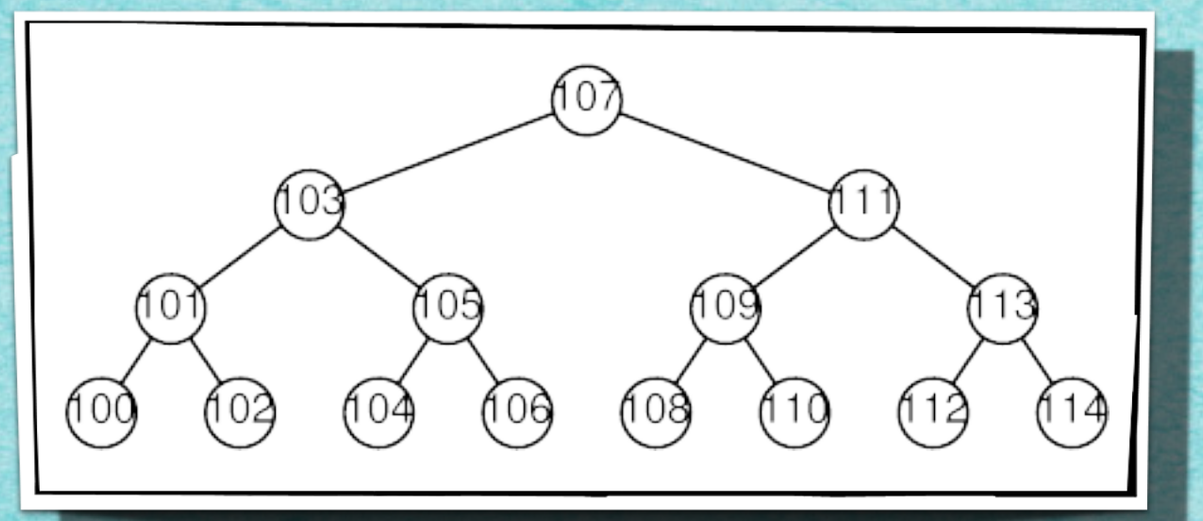


Satz 4.2

Die binäre Suche terminiert in O(log(RECHTS-LINKS)) Schritten (für RECHTS>LINKS).

Beweis:

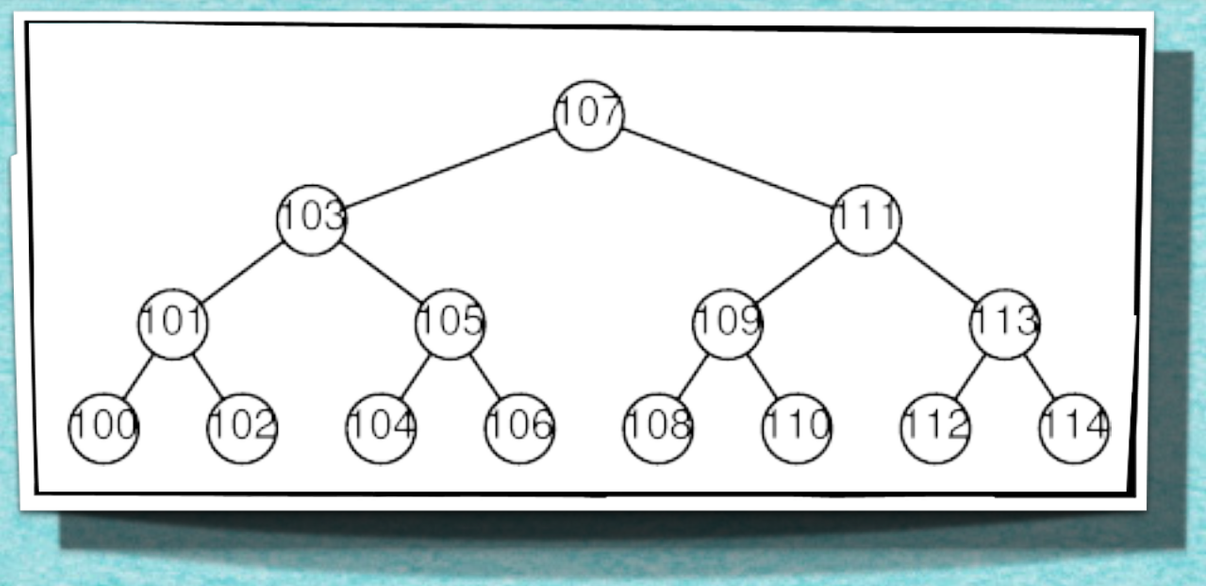
Selbst!



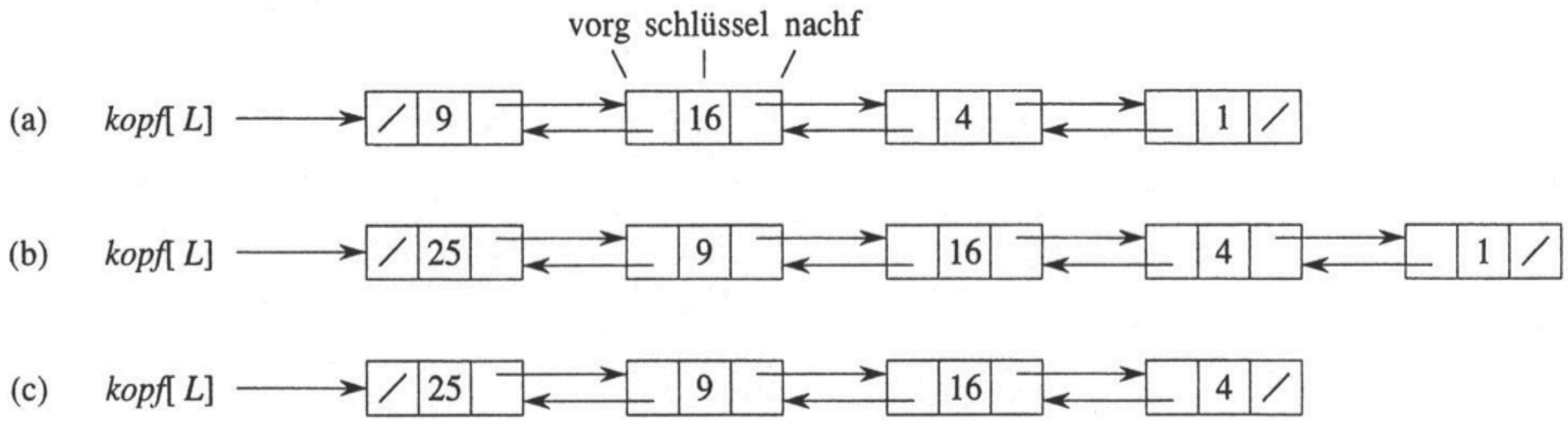
4.5 Binäre Suchbäume

Ideen:

- ***Strukturiere Daten wie im möglichen Ablauf einer binären Suche!***
- ***Erziele logarithmische Zeiten!***



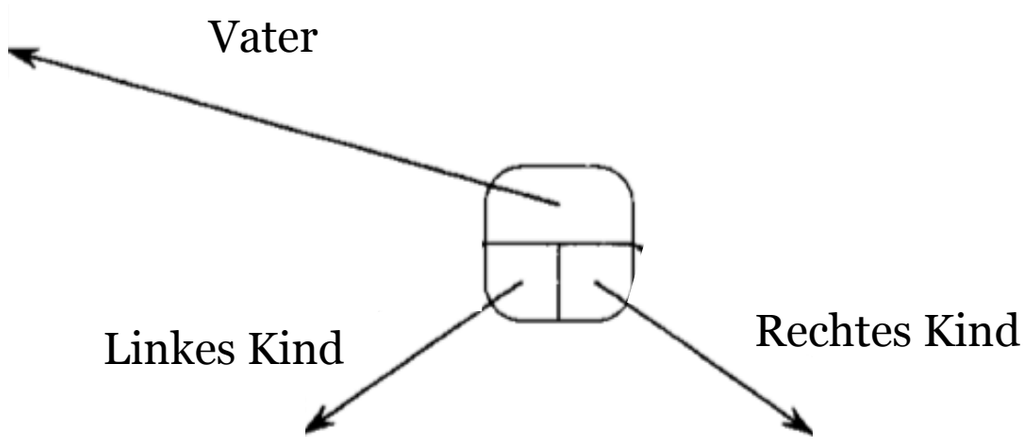
Struktur einer doppelt verketteten Liste



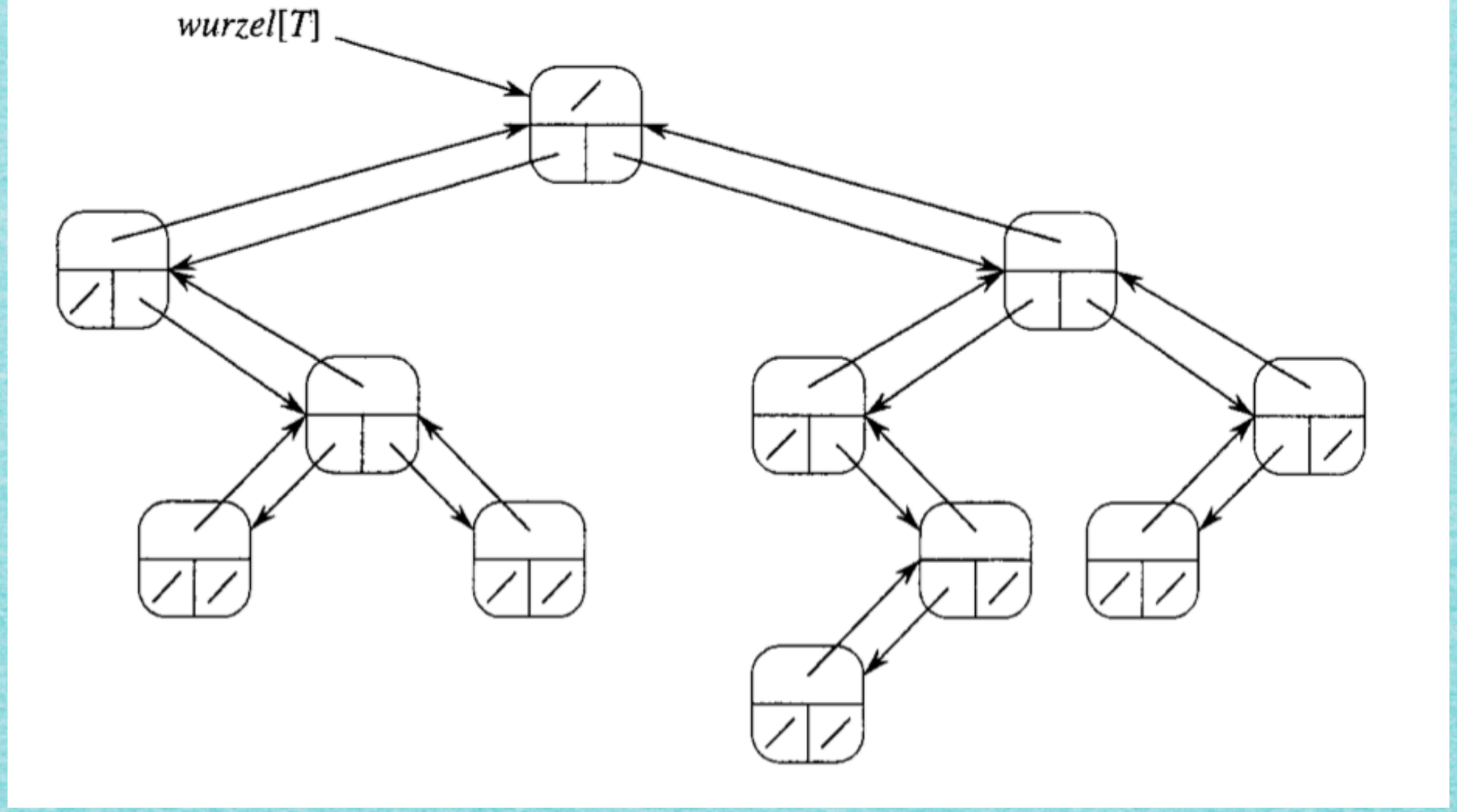
- **Füge vorne das Element mit Schlüssel 25 ein.**

- **Finde ein Element mit Schlüssel 1 und lösche es.**

Binärer Suchbaum

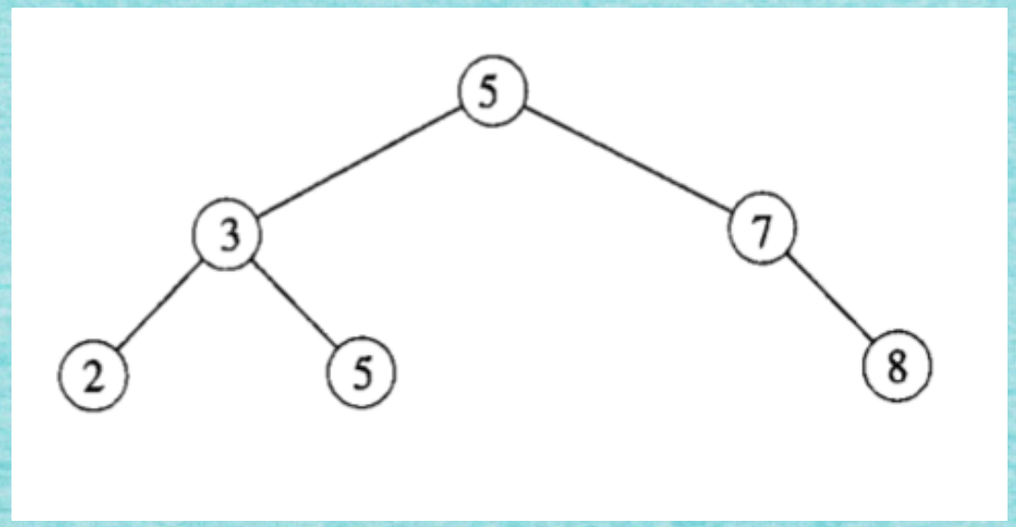


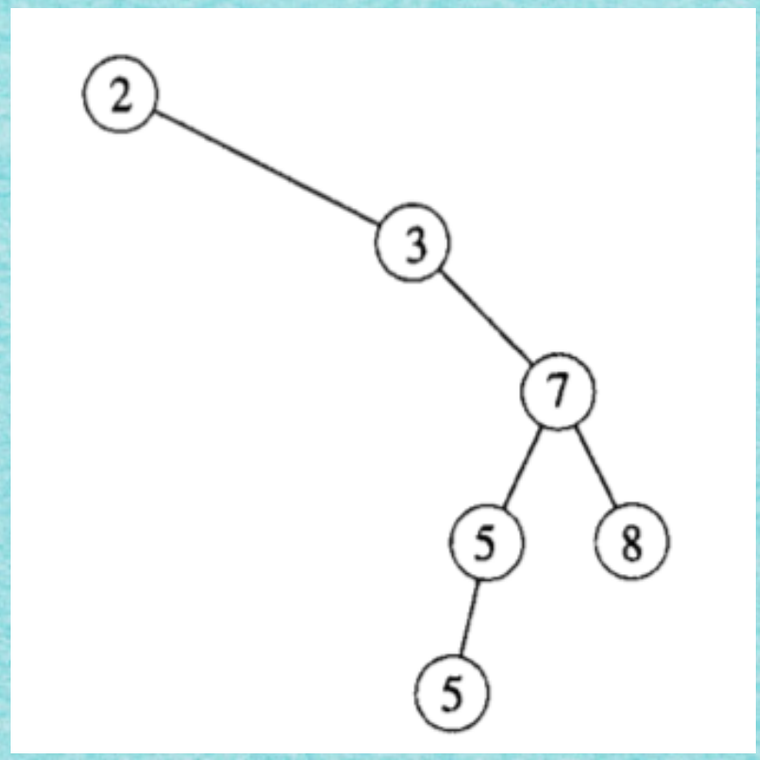
Binärer Suchbaum



Außerdem wichtig: Struktur der Schlüsselwerte!

Ordnungsstruktur





Linker Teilbaum: Kleinere (bzw. nicht größere) Zahlen
Rechter Teilbaum: Größere Zahlen

Definition 4.3

(1) Ein *gerichteter Graph* D=(V,A) besteht aus einer endlichen Menge V von Knoten v und einer endlichen Menge von gerichteten Kanten, a=(v,w). (v ist Vorgänger von w.)

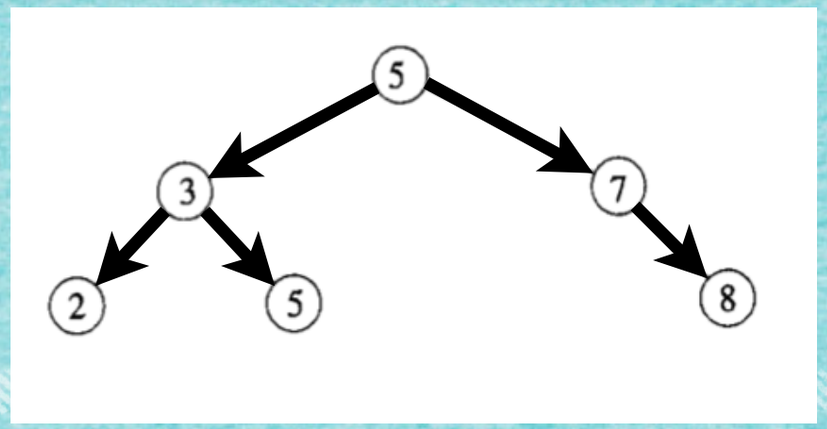
(2) Ein *gerichteter Baum* B=(V,T) hat folgende Eigenschaften:

(i) Es gibt einen eindeutigen Knoten w ohne Vorgänger.

(ii) Jeder Knoten außer w ist auf einem eindeutigen Weg von w aus erreichbar; das heißt insbesondere, dass v einen eindeutigen Vorgänger (Vaterknoten) hat.

(3) Die *Höhe* eines gerichteten Baumes ist die maximale Länge eines gerichteten Weges von der Wurzel.

(4) Ein *binärer Baum* ist ein gerichteter Baum, in dem jeder Knoten höchstens zwei Nachfolger ("Kindknoten") hat. Einer ist der "linke" l[v], der andere der "rechte", r[v].



Definition 4.3

(1) Ein *gerichteter Graph* D=(V,A) besteht aus einer endlichen Menge V von Knoten v und einer endlichen Menge von gerichteten Kanten, a=(v,w). (v ist Vorgänger von w.)

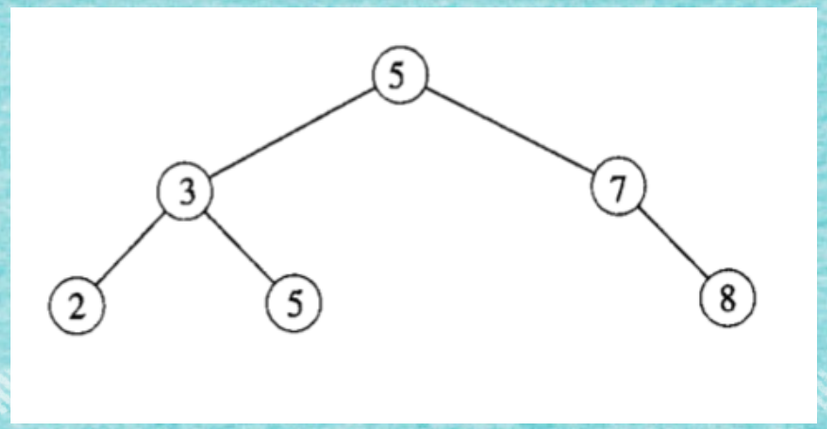
(2) Ein *gerichteter Baum* B=(V,T) hat folgende Eigenschaften:

(i) Es gibt einen eindeutigen Knoten w ohne Vorgänger.

(ii) Jeder Knoten außer w ist auf einem eindeutigen Weg von w aus erreichbar; das heißt insbesondere, dass v einen eindeutigen Vorgänger (Vaterknoten) hat.

(3) Die *Höhe* eines gerichteten Baumes ist die maximale Länge eines gerichteten Weges von der Wurzel.

(4) Ein *binärer Baum* ist ein gerichteter Baum, in dem jeder Knoten höchstens zwei Nachfolger ("Kindknoten") hat. Einer ist der "linke" l[v], der andere der "rechte", r[v].



Definition 4.3 (Forts.)

(5) Ein binärer Baum heißt *voll*, wenn jeder Knoten zwei oder keinen Kindknoten hat.

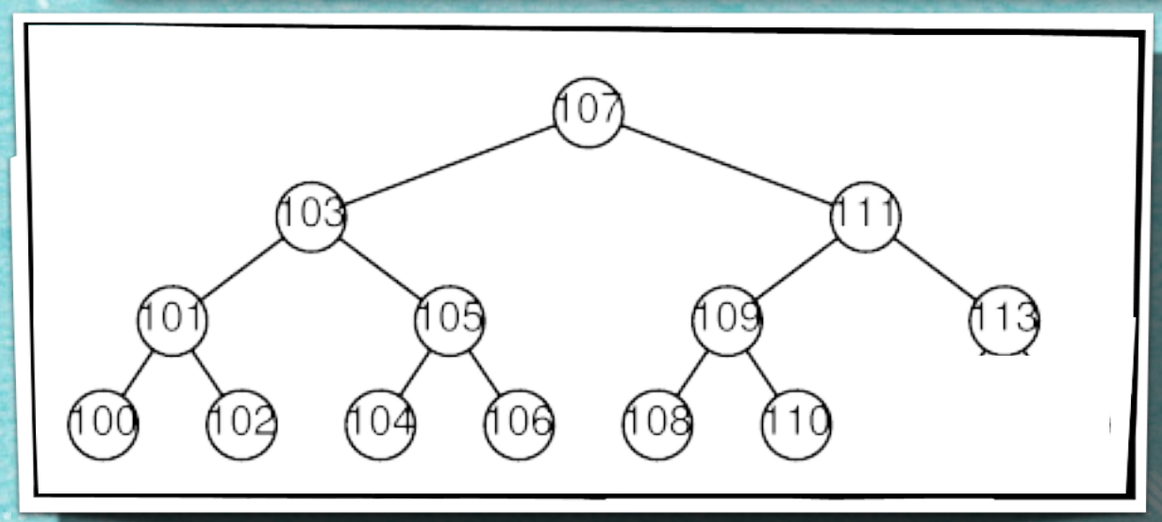
(6) Ein binärer Baum heißt *vollständig*, wenn zusätzlich alle Blätter gleichen Abstand zur Wurzel haben.

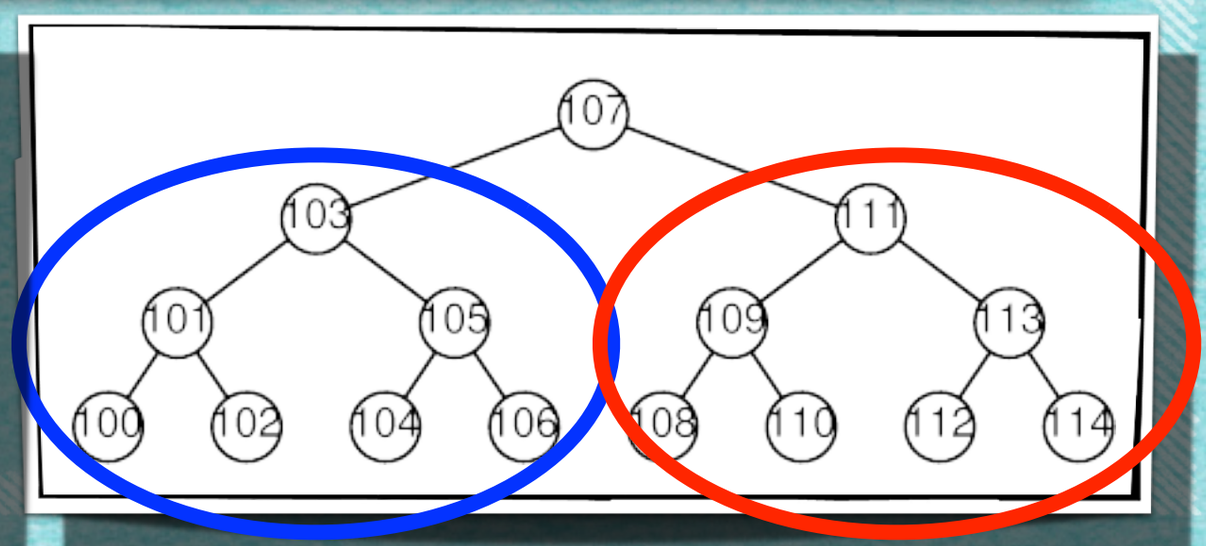
(7) Ein Knoten ohne Kindknoten heißt *Blatt*.

(8) Der *Teilbaum eines Knotens v* ist durch die Menge der von v erreichbaren Knoten und der dabei verwendeten Kanten definiert; der linke Teilbaum ist der Teilbaum von l[v].

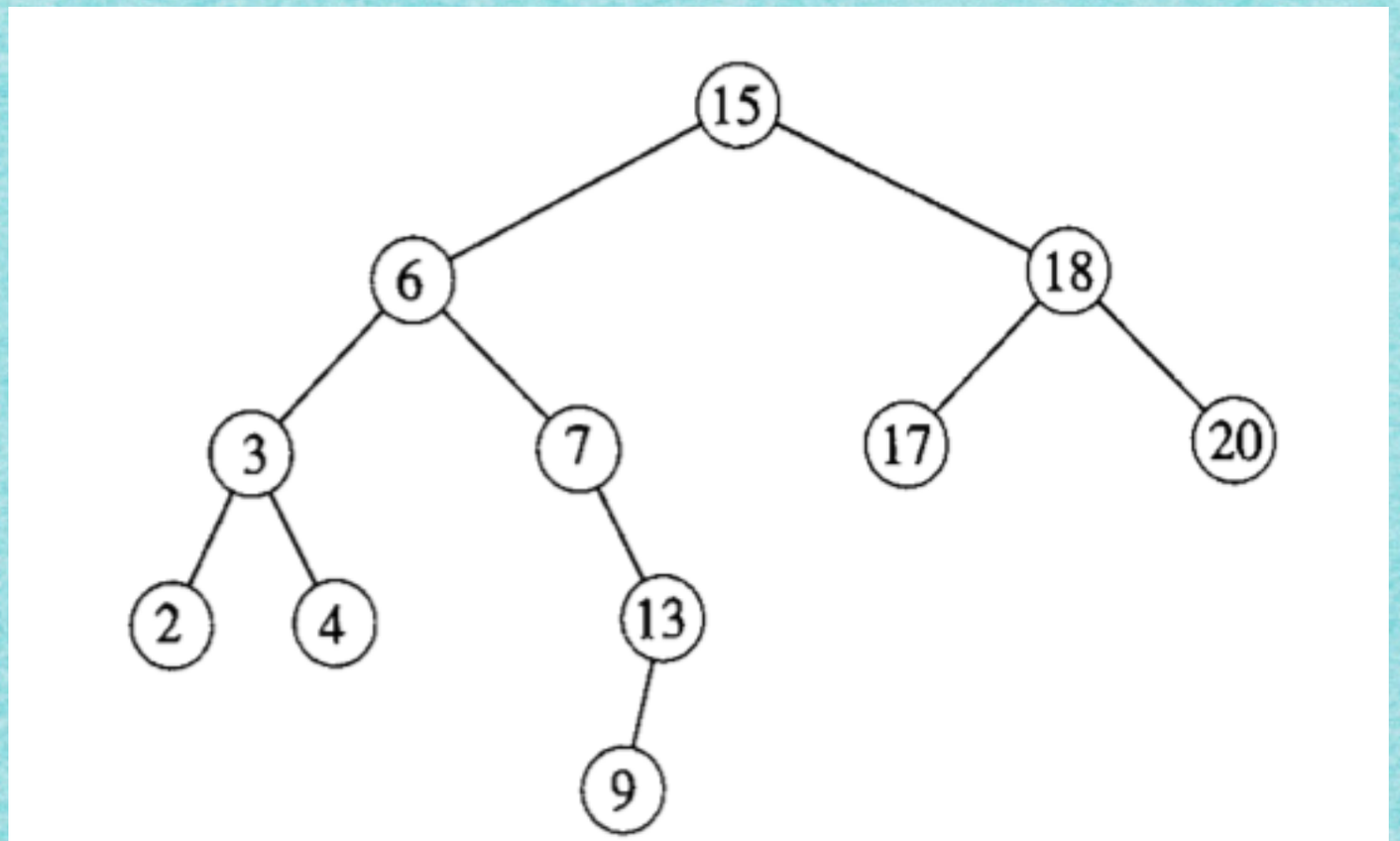
(9) In einem *binären Suchbaum* hat jeder Knoten v einen Schlüsselwert S[v],und es gilt:

- **S[u]≤S[v] für Knoten u im linken Teilbaum von v**
- **S[u]>S[v] für Knoten u im rechten Teilbaum von v**





Minimum und Maximum



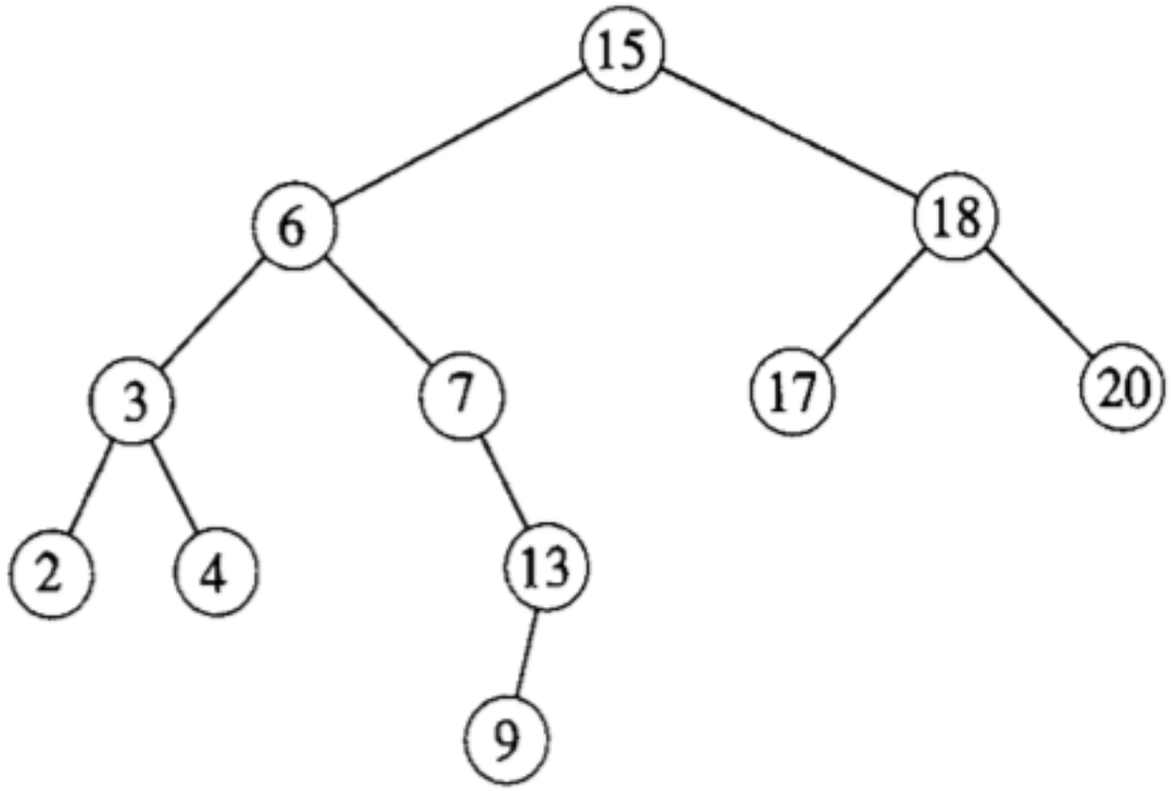
TREE-MINIMUM(x)

```
1  while links[x] ≠ NIL
2        do x ← links[x]
3  return x
```

TREE-MAXIMUM(x)

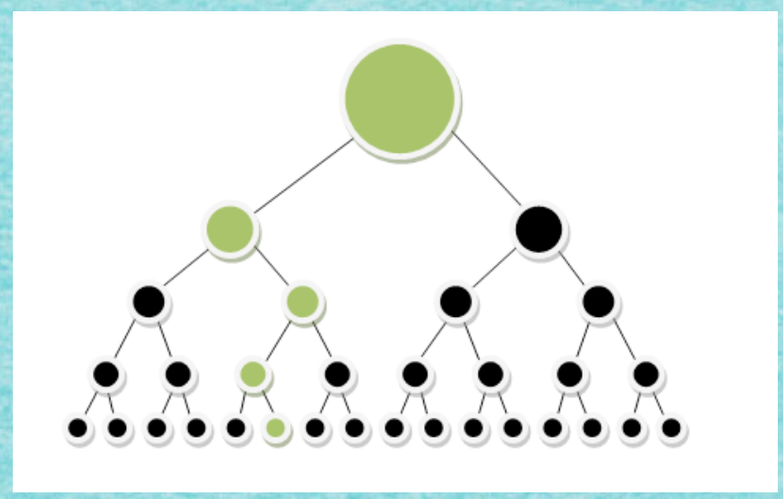
```
1  while rechts[x] ≠ NIL
2        do x ← rechts[x]
3  return x
```

Suche im Suchbaum



ITERATIVE-TREE-SEARCH(x, k)

```
1  while x ≠ NIL und k ≠ schlüssel[x]
2        do if k < schlüssel[x]
3              then x ← links[x]
4              else  x ← rechts[x]
5  return x
```

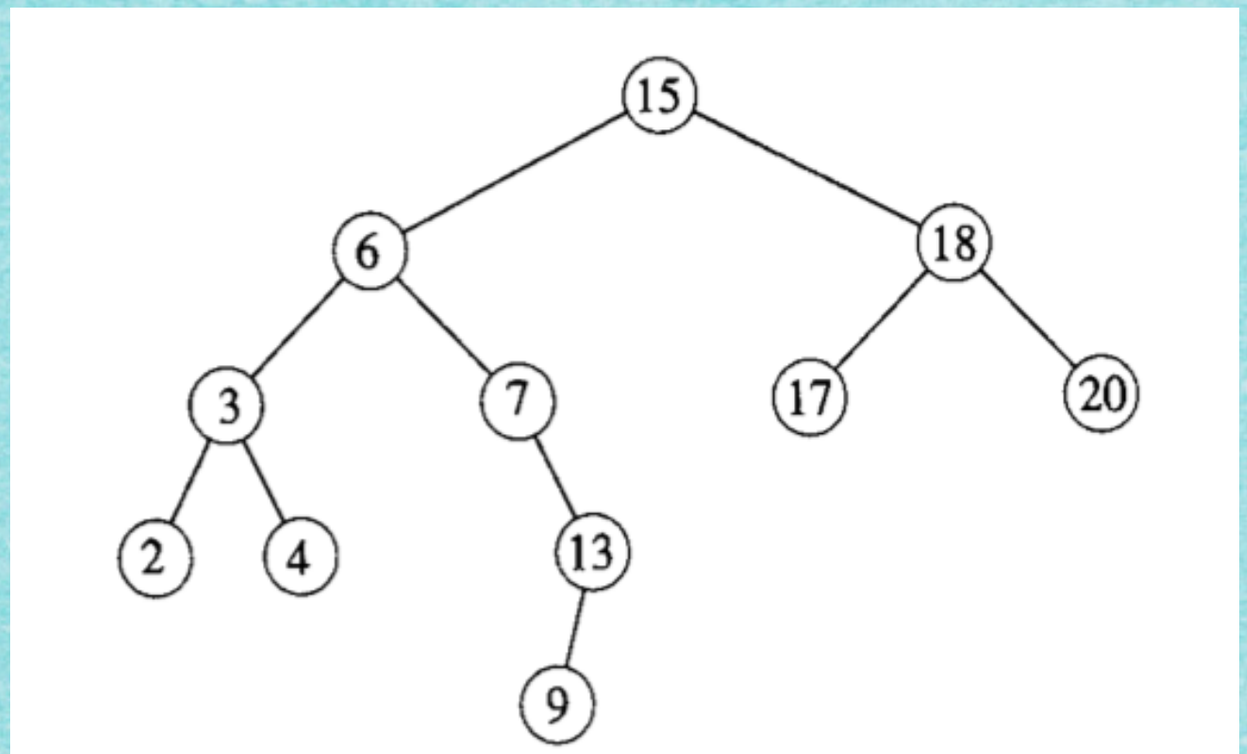


<u>SUCCESSOR(S,x):</u>
“Finde das nächstgrößere Element”

Für ein in x stehendes Element in S, bestimme ein Element von nächstgrößerem Wert in S.

Ausgabe: Zeiger y auf Element
NIL, falls x Maximum von S angibt

Nachfolger im Suchbaum



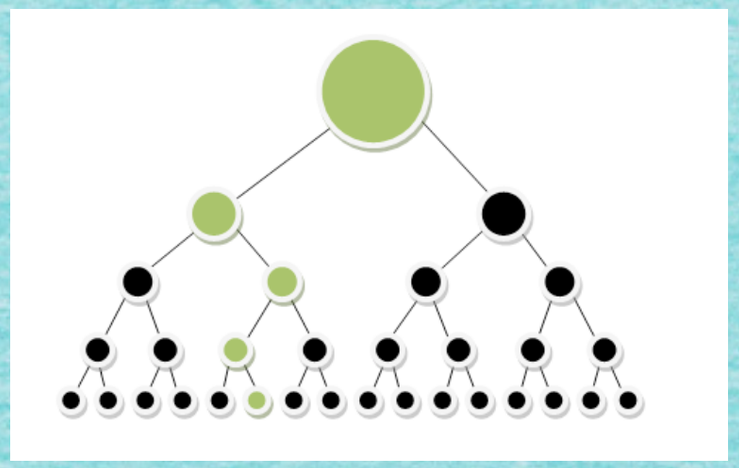

```
TREE-SUCCESSOR(x)
1  if rechts[x] ≠ NIL
2     then return TREE-MINIMUM(rechts[x])
3  y ← p[x]
4  while y ≠ NIL und x = rechts[y]
5        do x ← y
6           y ← p[y]
7  return y
```

Satz 4.4

Suchen, Minimum, Maximum, Nachfolger, Vorgänger können in einem binären Suchbaum der Höhe h in Zeit O(h) durchlaufen werden.

Beweis:

Klar, der Baum wird nur einmal vertikal durchlaufen!

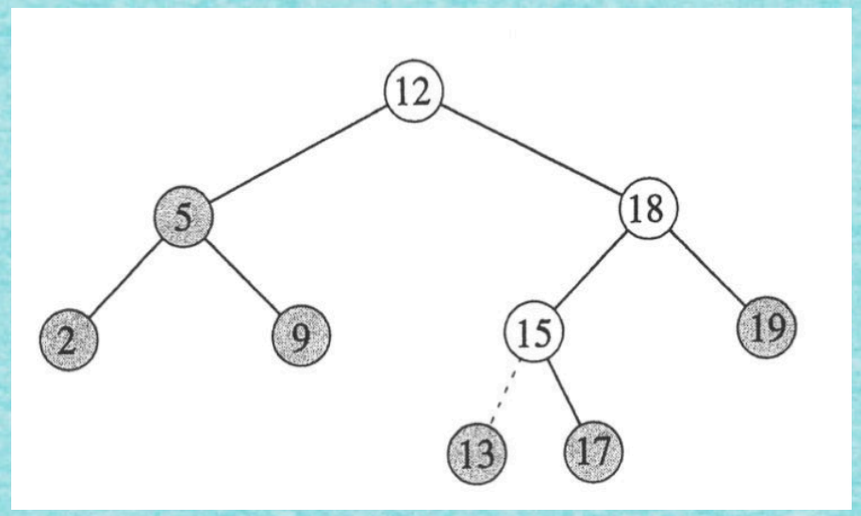


4.1 Grundoperationen

<u>INSERT(S,x):</u> **“Füge x in S ein”**

Erweitere S um das Element, das unter der Adresse x steht.

Einfügen im Suchbaum

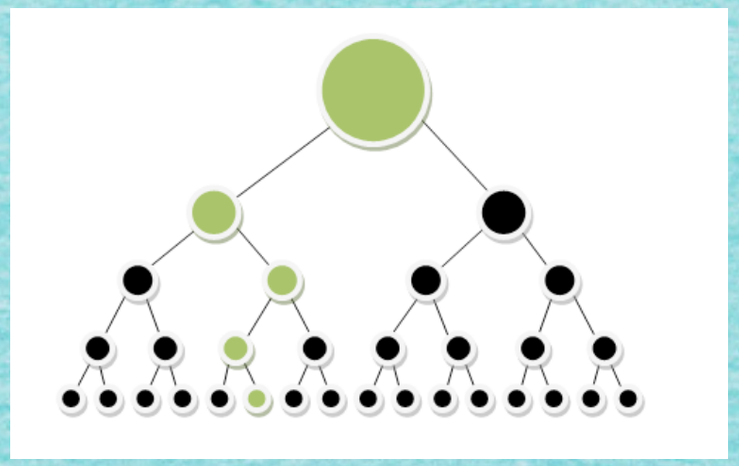


TREE-INSERT(T, z)

```
2   x ← wurzel[T]
3   while x ≠ NIL
4         do y ← x
5            if schlüssel[z] < schlüssel[x]
6               then x ← links[x]
7               else  x ← rechts[x]
8   p[z] ← y
9   if y = NIL
10     then wurzel[T] ← z
11     else  if schlüssel[z] < schlüssel[y]
12              then links[y] ← z
13              else  rechts[y] ← z
```

Satz 4.5
Einfügen benötigt O(h) für einen binären Suchbaum der Höhe h.

Beweis:
Klar, der Baum wird nur vertikal abwärts durchlaufen!



Mehr demnächst!

s.fekete@tu-bs.de