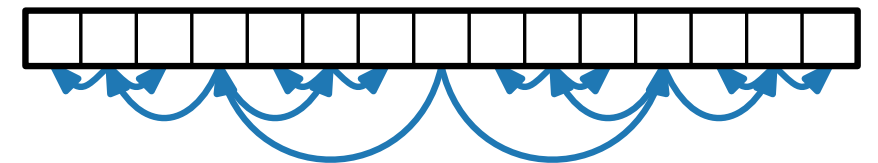
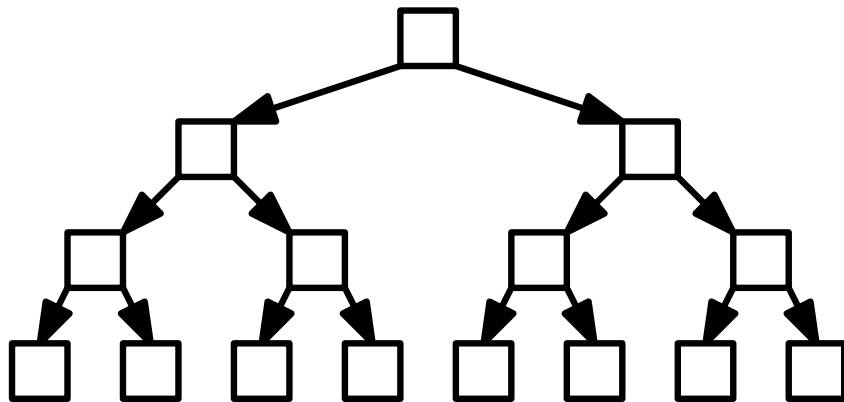


Algorithmen und Datenstrukturen

Vorlesung 13: Binäre Suchbäume



Zwischentest II: 12.12.2024, 8:15–10:00

- Zufallsexperimente, (Indikator-) Zufallsvariable, Erwartungswert
- (Randomisiertes) QuickSort
- Untere Schranke für WC-Laufzeit von vergleichsbasierten Sortierverfahren
- Linearzeit-Sortierverfahren
- Auswahlproblem (Median): randomisiert & deterministisch
- Elementare Datenstrukturen
- Hashing
- Binäre Suchbäume (Rot-Schwarz-Bäume noch nicht)

Dynamische Menge



Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M , wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation

ptr INSERT(key k , value v)
DELETE(ptr x)

ptr SEARCH(key k)

Wörter-
buch

ptr MINIMUM()

ptr MAXIMUM()

ptr PREDECESSOR(ptr x)

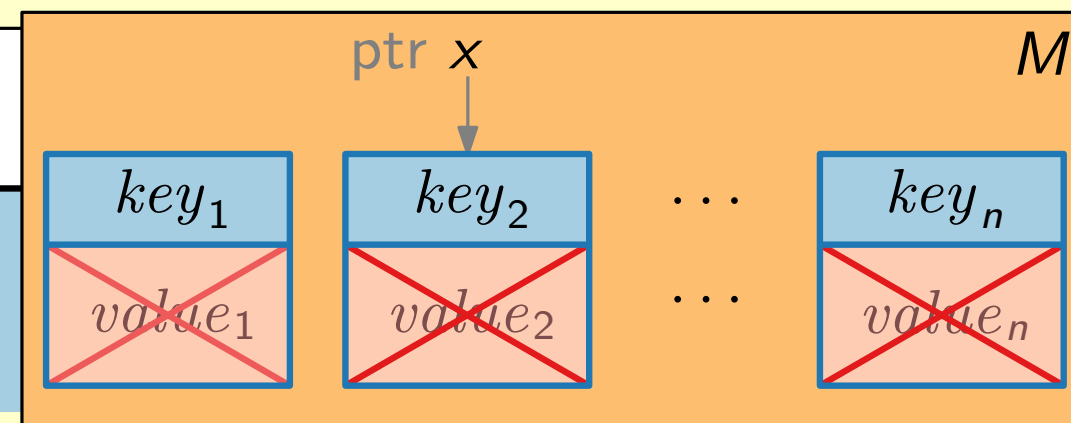
ptr SUCCESSOR(ptr x)

Pointer

Funktionalität

Veränderungen

Anfragen



Vereinfachung

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
--	--------	---------	---------	-----------

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste				

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$			

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$		

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste				

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$			

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$		

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld				

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld			$\Theta(1)$	

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld			$\Theta(1)$	$\Theta(1)$

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld		$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle				

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle			—	—

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$		—	—

* unter bestimmten Annahmen.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—

* unter bestimmten Annahmen.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP				

* unter bestimmten Annahmen.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—			

* unter bestimmten Annahmen.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$		

* unter bestimmten Annahmen.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/—$	

* unter bestimmten Annahmen.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/—$	—

* unter bestimmten Annahmen.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum				

* unter bestimmten Annahmen.

Implementierung

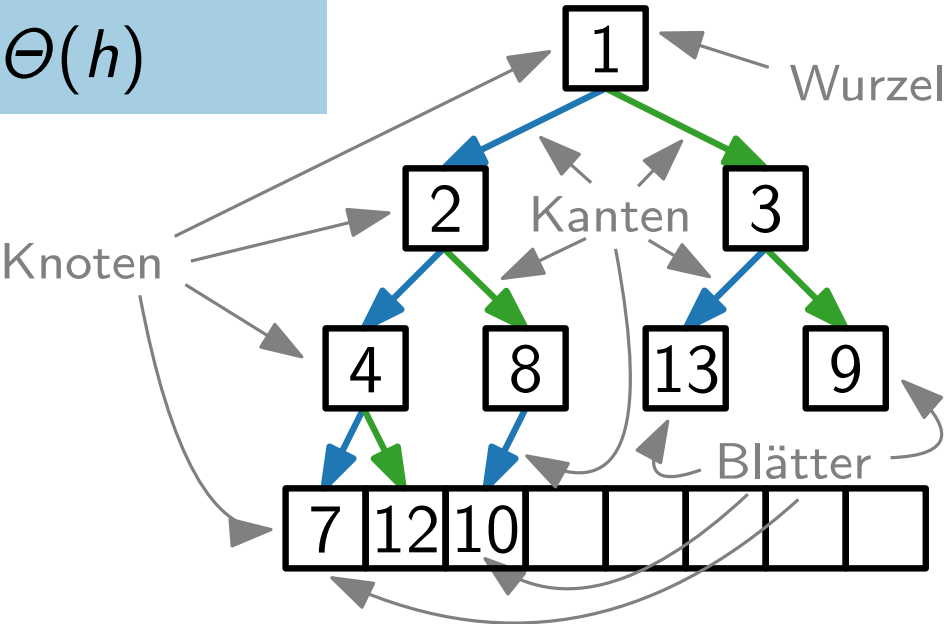
	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

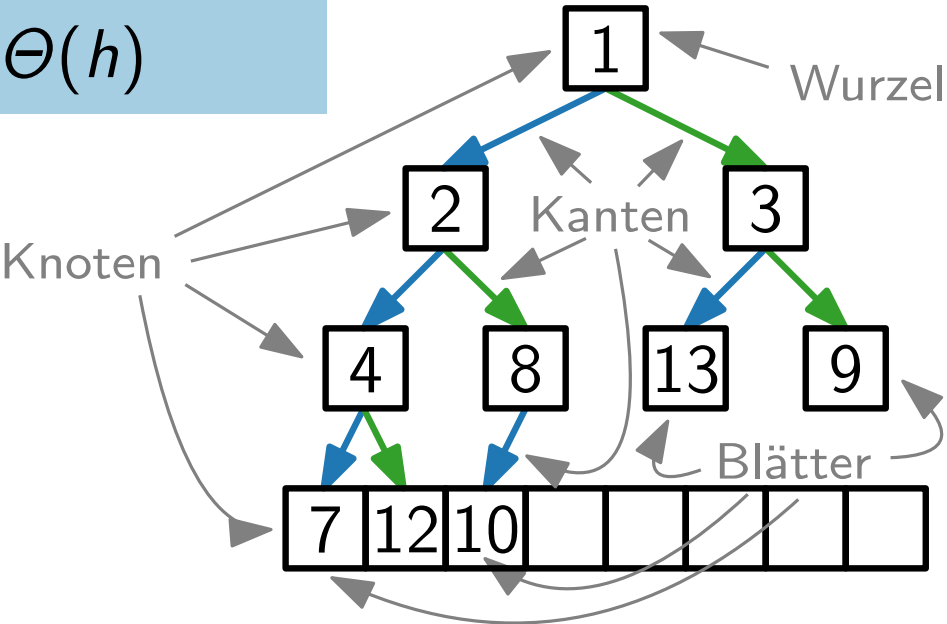
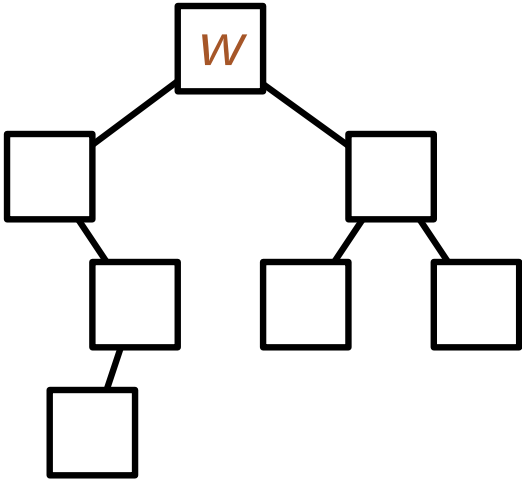
* unter bestimmten Annahmen.



Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

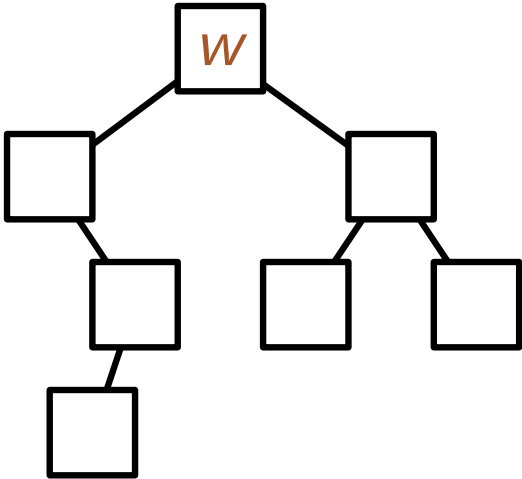
* unter bestimmten Annahmen.



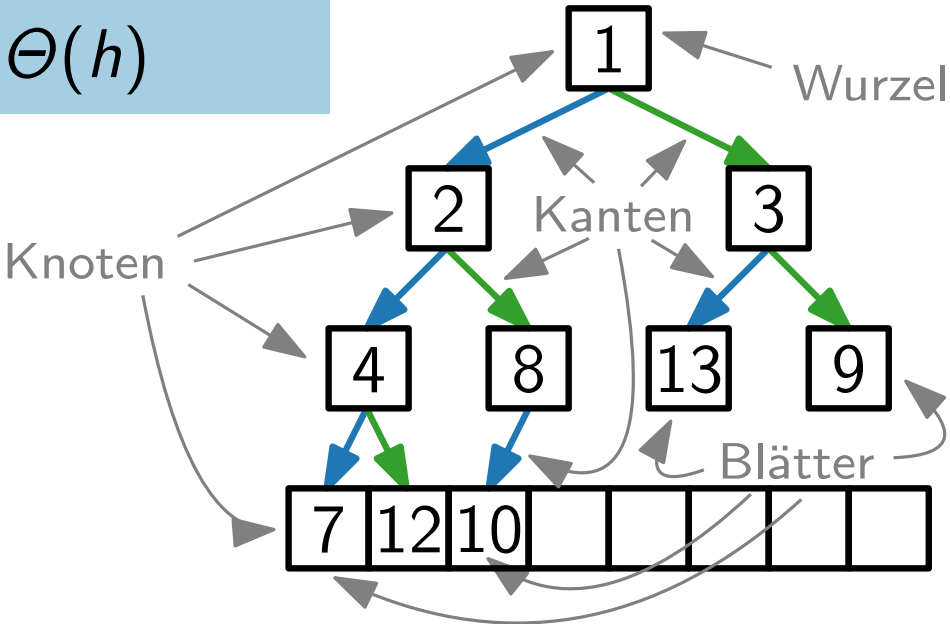
Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.



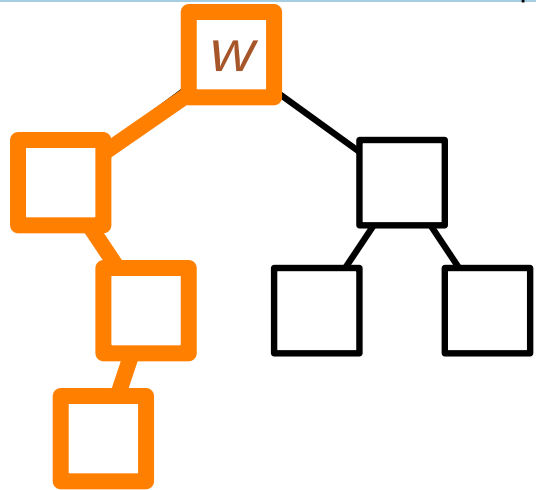
$h(T)$ = Höhe des Baums T



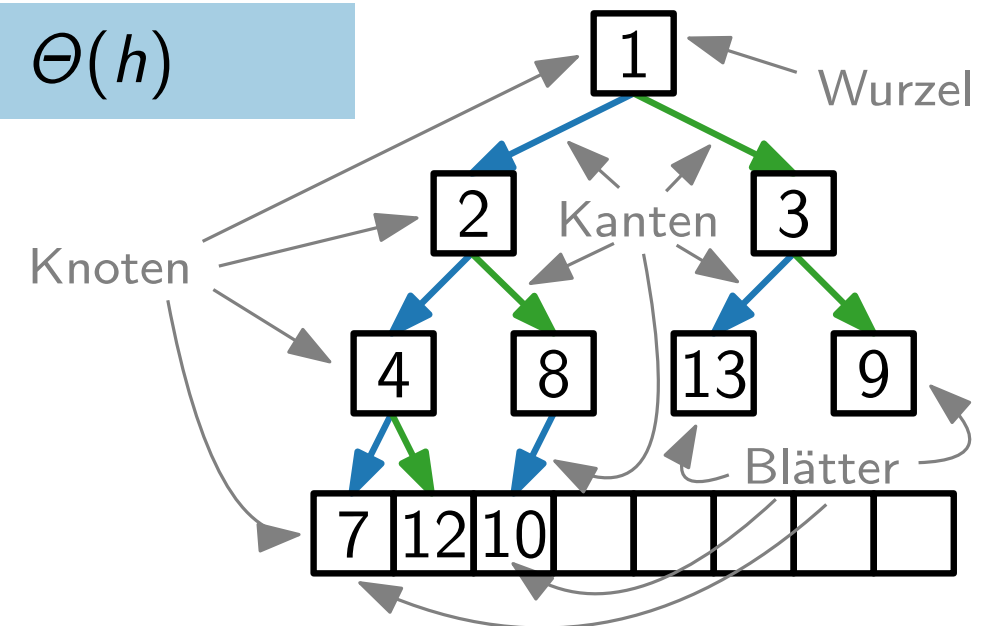
Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.



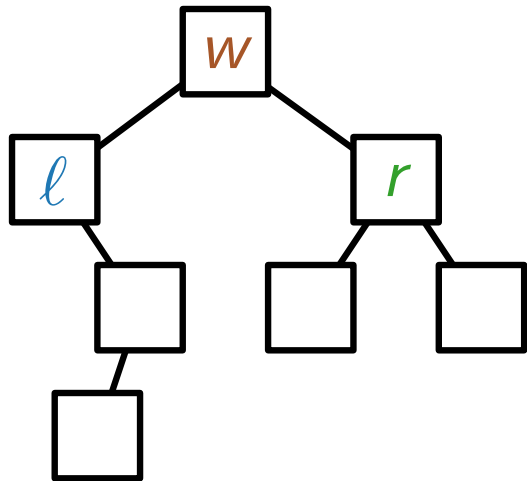
$h(T)$ = Höhe des Baums T
 = Anz. Kanten auf längstem
 Wurzel-Blatt-Pfad



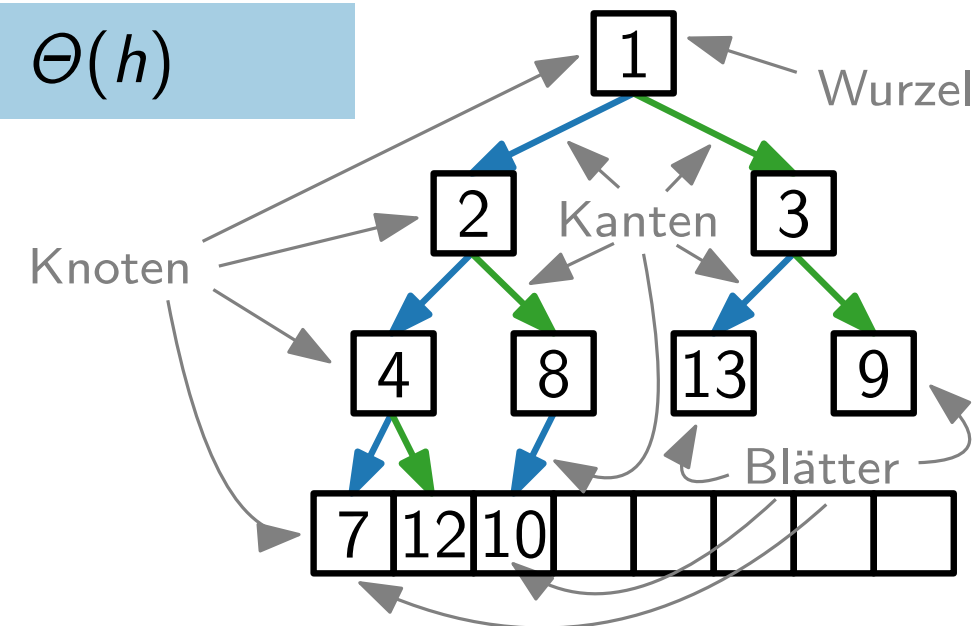
Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.



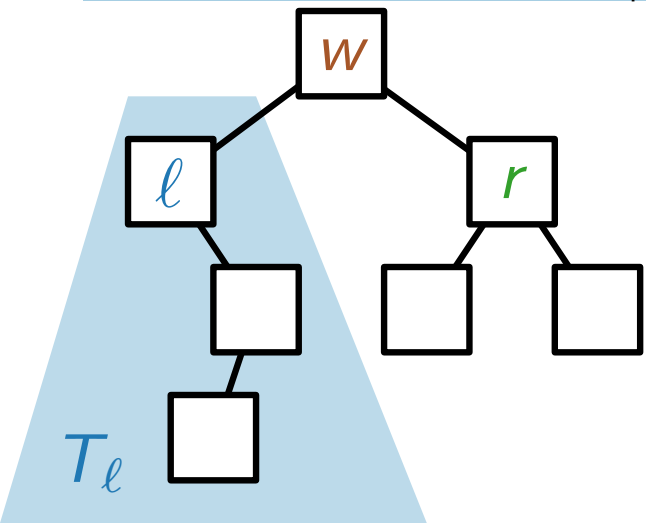
$h(T)$ = Höhe des Baums T
 = Anz. Kanten auf längstem
 Wurzel-Blatt-Pfad



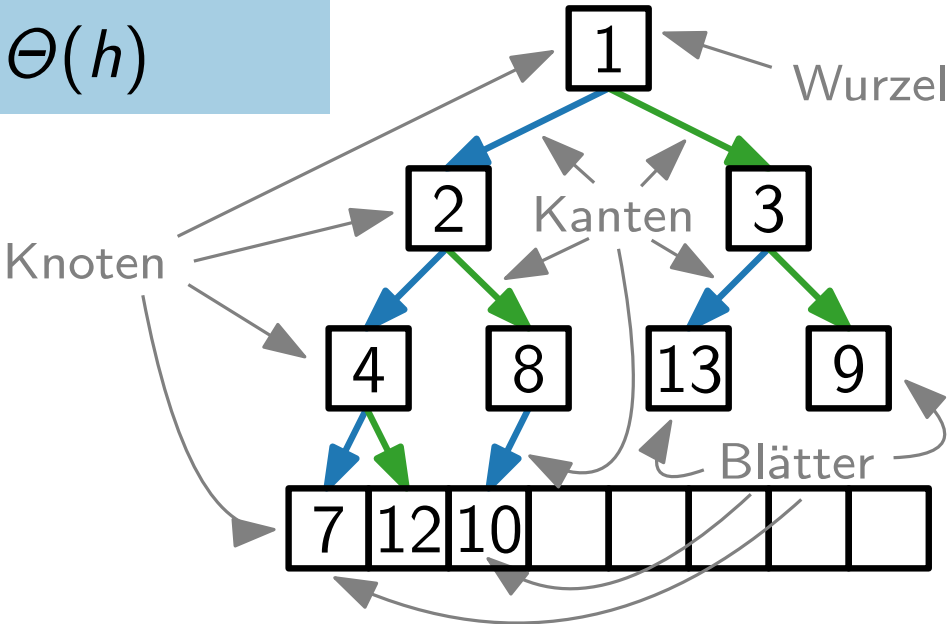
Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.



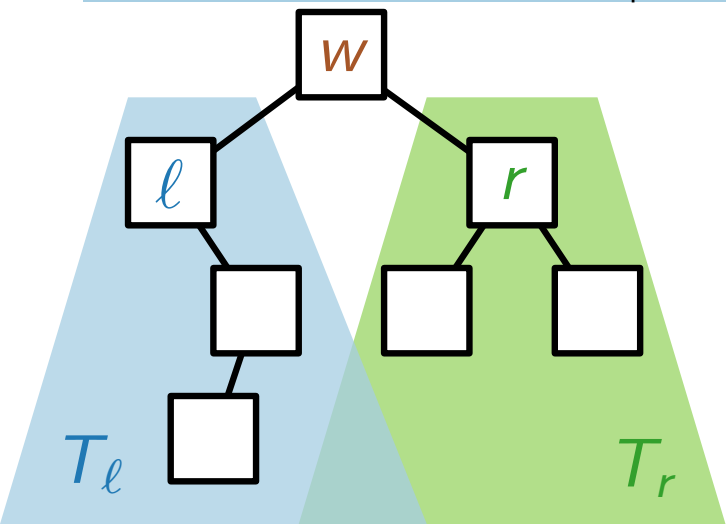
$h(T)$ = Höhe des Baums T
 = Anz. Kanten auf längstem
 Wurzel-Blatt-Pfad



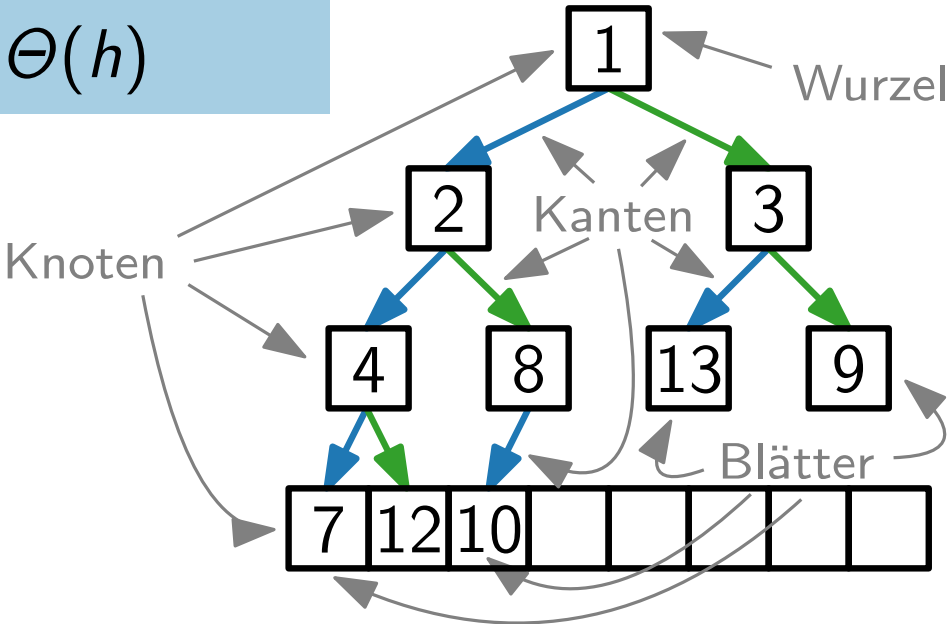
Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.



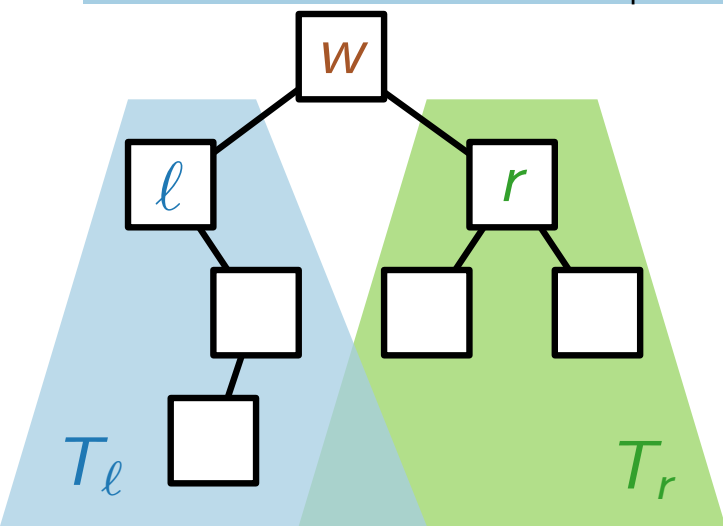
$h(T)$ = Höhe des Baums T
 = Anz. Kanten auf längstem
 Wurzel-Blatt-Pfad



Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.



$h(T)$ = Höhe des Baums T
 = Anz. Kanten auf längstem
 Wurzel-Blatt-Pfad

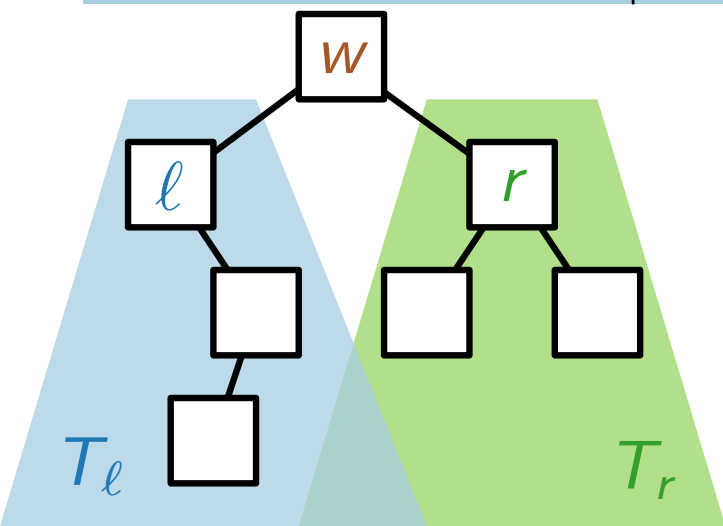
= { **Aufgabe.**
 Finden Sie die
 Rekursionsgleichung!

falls Baum = Blatt
 sonst.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/-$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.



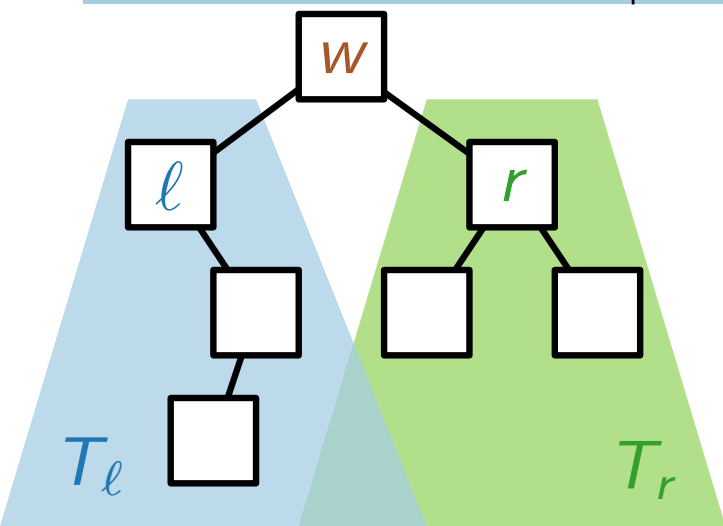
$$\begin{aligned}
 h(T) &= \text{Höhe des Baums } T \\
 &= \text{Anz. Kanten auf längstem} \\
 &\quad \text{Wurzel-Blatt-Pfad} \\
 &= \begin{cases} 0 \\ \end{cases}
 \end{aligned}$$

falls Baum = Blatt
sonst.

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/—$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.



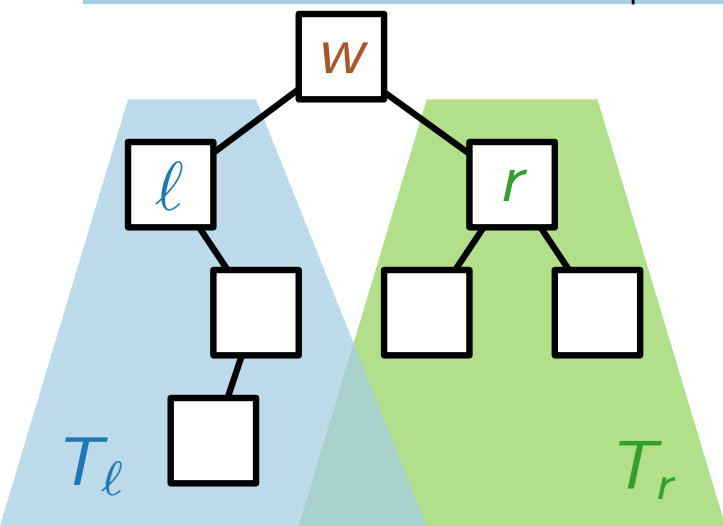
$h(T)$ = Höhe des Baums T
 = Anz. Kanten auf längstem
 Wurzel-Blatt-Pfad

$$= \begin{cases} 0 & \text{falls Baum = Blatt} \\ 1 + \max\{h(T_\ell), h(T_r)\} & \text{sonst.} \end{cases}$$

Implementierung

	SEARCH	INS/DEL	MIN/MAX	PRED/SUCC
unsortierte Liste	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sortierte Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sortiertes Feld	?	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Hashtabelle	$\Theta(1)^*$	$\Theta(1)^*$	—	—
MINHEAP	—	$\Theta(\log n)$	$\Theta(1)/—$	—
Binärer Suchbaum	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$

* unter bestimmten Annahmen.



$h(T)$ = Höhe des Baums T
 = Anz. Kanten auf längstem
 Wurzel-Blatt-Pfad

$$= \begin{cases} 0 & \text{falls Baum = Blatt} \\ 1 + \max\{h(T_\ell), h(T_r)\} & \text{sonst.} \end{cases}$$

Suche im sortierten Feld

2	3	5	6	8	9	11	12	13	14	17	19	21	24	27
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

SEARCH(21)

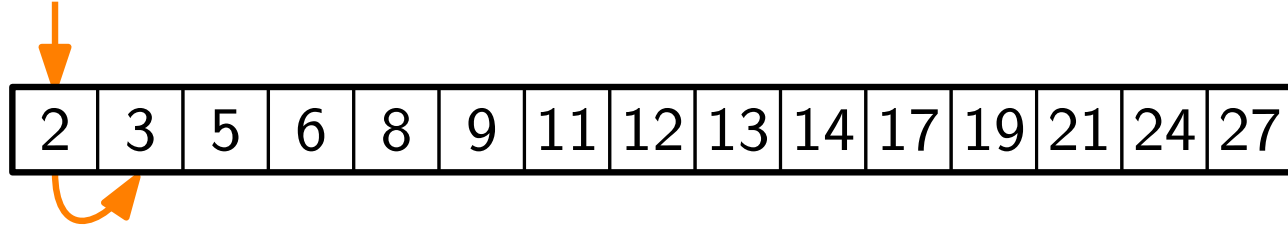
Suche im sortierten Feld



2	3	5	6	8	9	11	12	13	14	17	19	21	24	27
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

SEARCH(21)

Suche im sortierten Feld

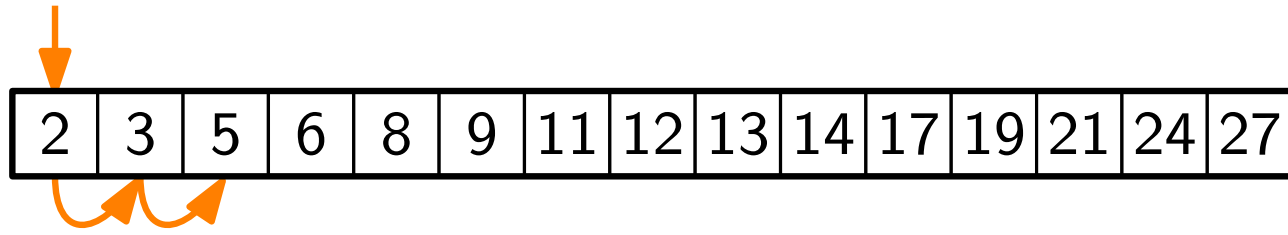


A horizontal array of 15 cells, each containing a number. An orange arrow points down to the first cell (2). A curved orange arrow starts from the bottom of the first cell and points to the bottom of the second cell (3).

2	3	5	6	8	9	11	12	13	14	17	19	21	24	27
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

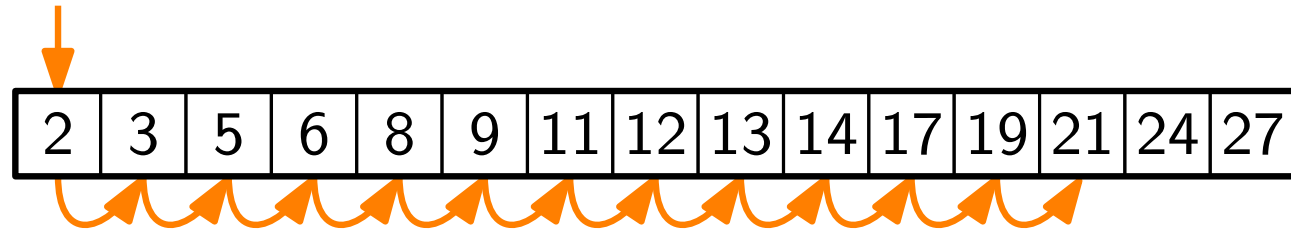
SEARCH(21)

Suche im sortierten Feld



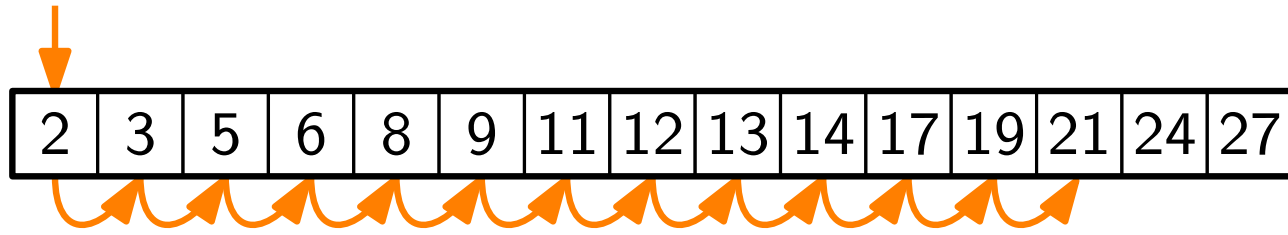
SEARCH(21)

Suche im sortierten Feld



SEARCH(21)

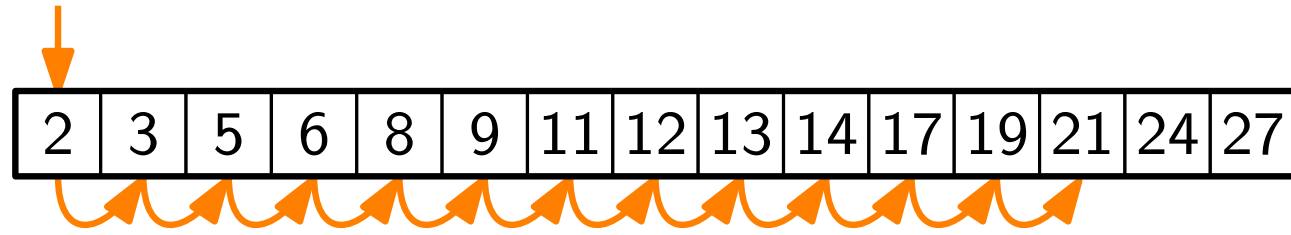
Suche im sortierten Feld



SEARCH(21)

Lineare Suche:

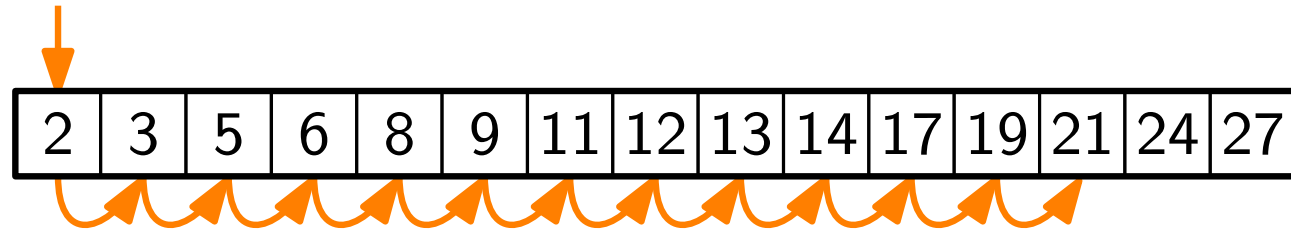
Suche im sortierten Feld



SEARCH(21)

Lineare Suche: hier 13 Schritte

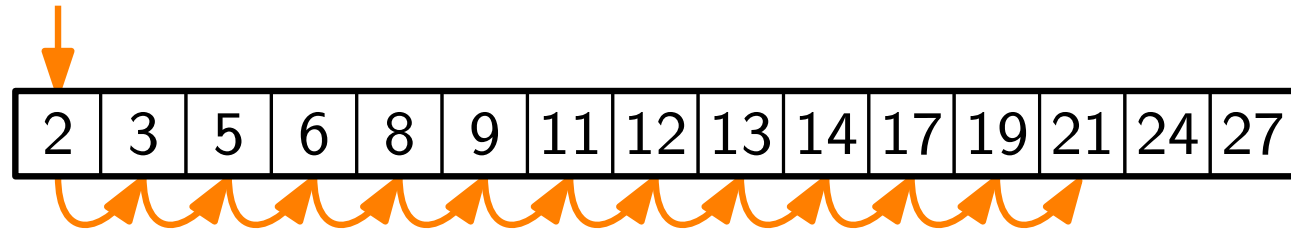
Suche im sortierten Feld



SEARCH(21)

Lineare Suche: hier im Worst Case 13 Schritte

Suche im sortierten Feld



SEARCH(21)

Lineare Suche: hier 13 im Worst Case n Schritte

Suche im sortierten Feld

2	3	5	6	8	9	11	12	13	14	17	19	21	24	27
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

SEARCH(21)

Lineare Suche: hier 13 im Worst Case n Schritte

Suche im sortierten Feld

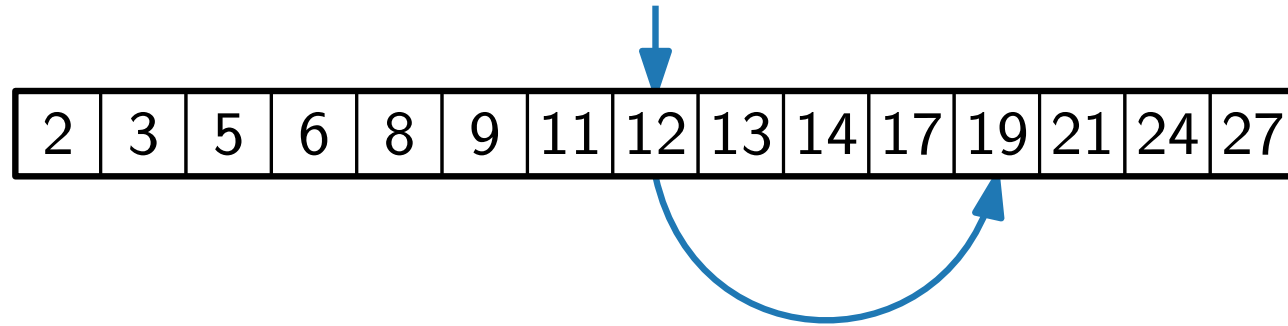


2	3	5	6	8	9	11	12	13	14	17	19	21	24	27
---	---	---	---	---	---	----	----	----	----	----	----	----	----	----

SEARCH(21)

Lineare Suche: hier 13 im Worst Case n Schritte

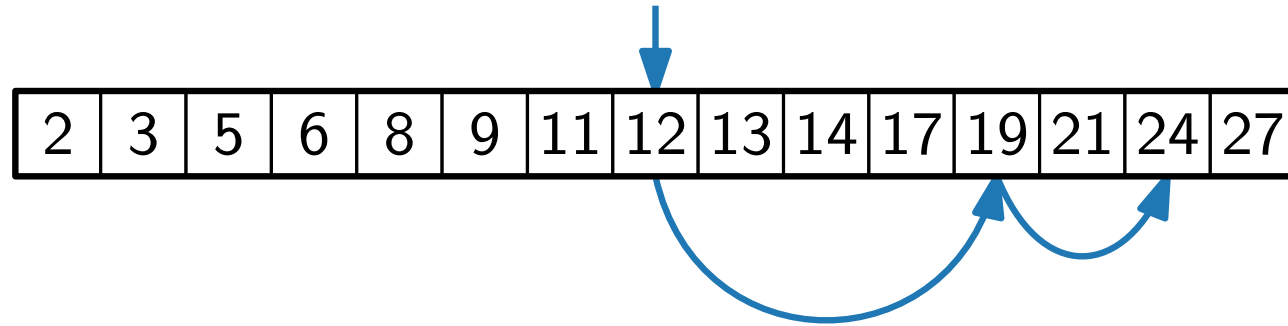
Suche im sortierten Feld



SEARCH(21)

Lineare Suche: hier 13 Schritte
im Worst Case n Schritte

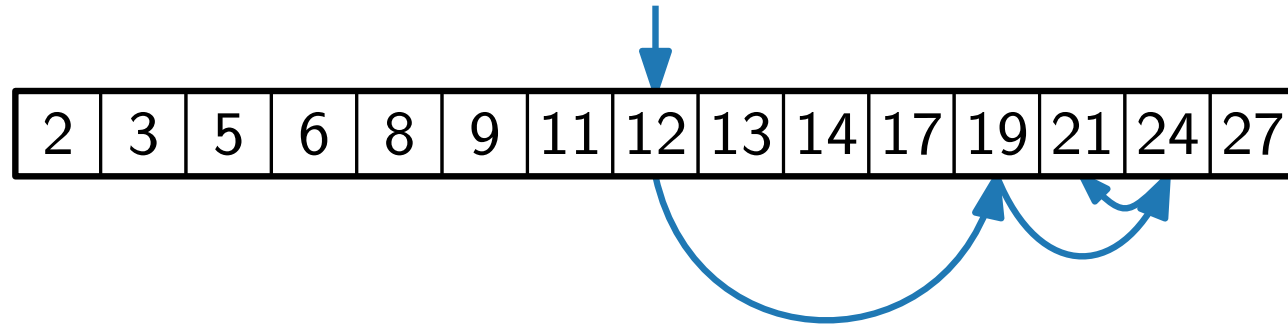
Suche im sortierten Feld



SEARCH(21)

Lineare Suche: hier 13 im Worst Case n Schritte

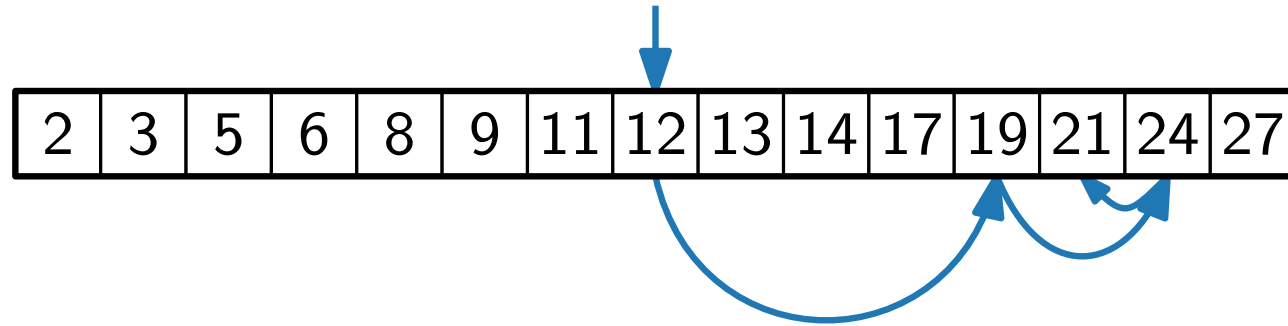
Suche im sortierten Feld



SEARCH(21)

Lineare Suche: hier 13 im Worst Case n Schritte

Suche im sortierten Feld

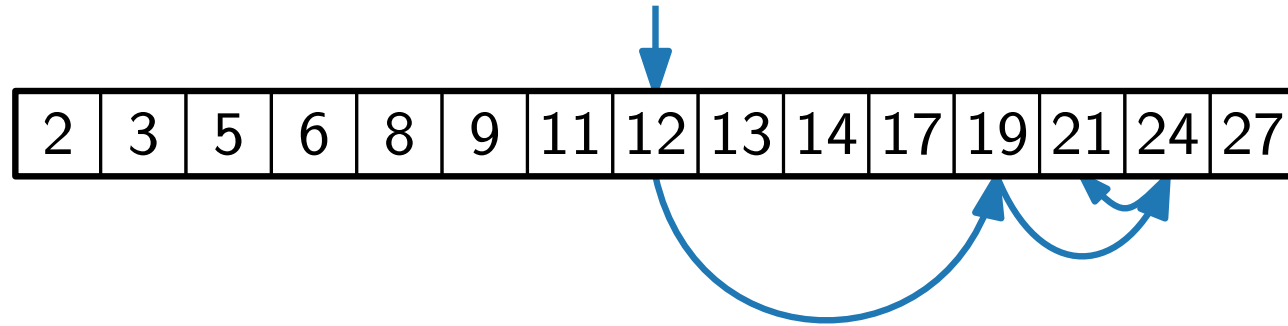


SEARCH(21)

Lineare Suche: hier 13 im Worst Case n Schritte

Binäre Suche:

Suche im sortierten Feld

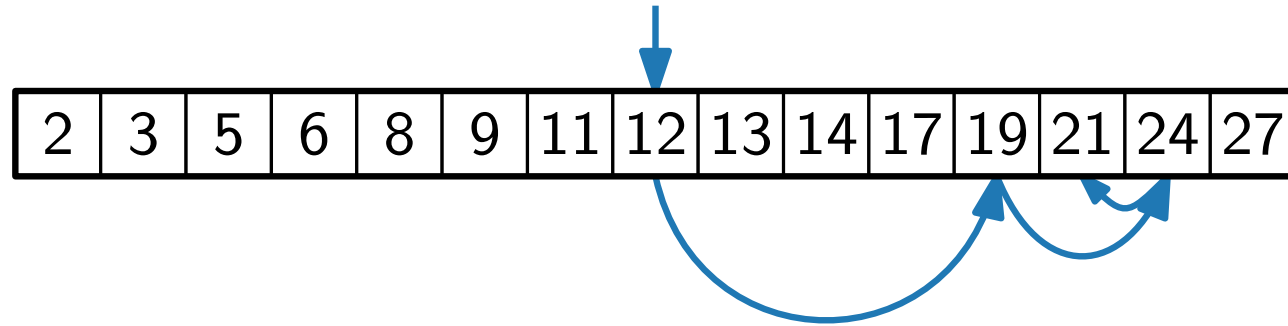


SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4		Schritte

Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

Suche im sortierten Feld

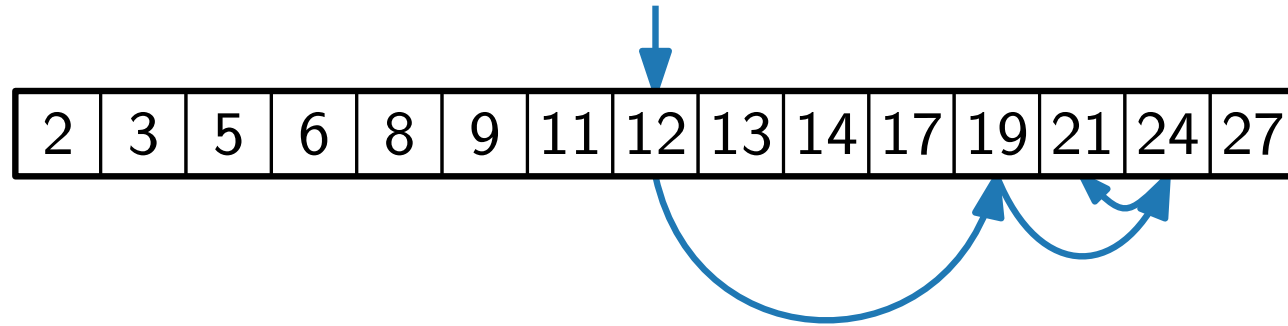


SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

Suche im sortierten Feld



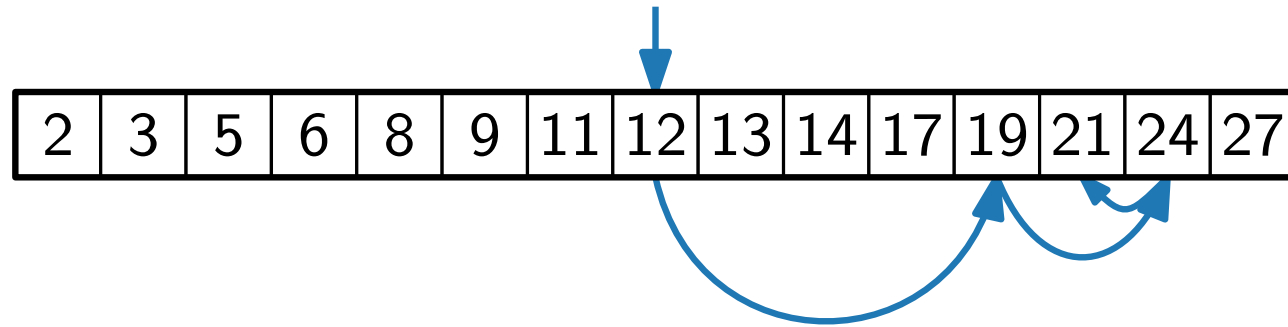
SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

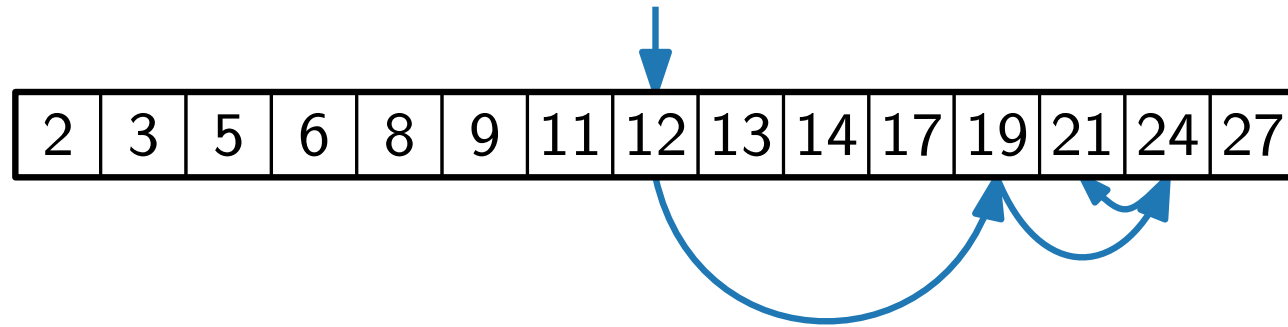
grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq$

Aufgabe.

Finden Sie die Rekursions(un)gleichung

Suche im sortierten Feld



SEARCH(21)

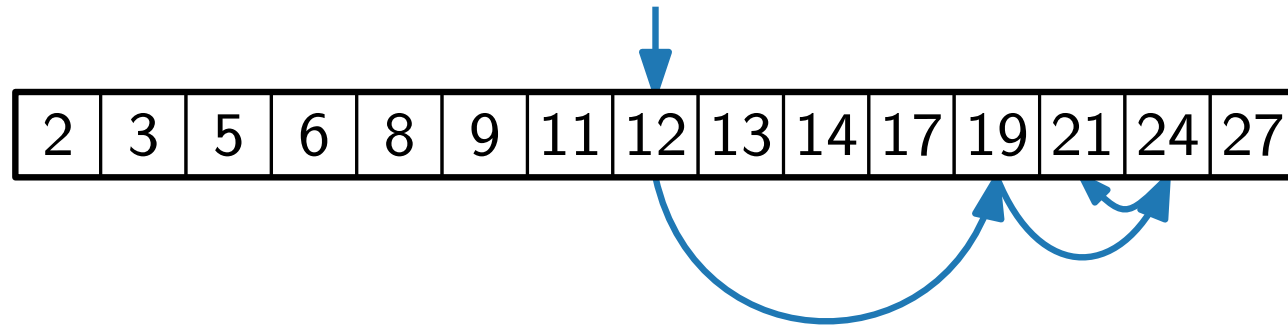
	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

Suche im sortierten Feld



SEARCH(21)

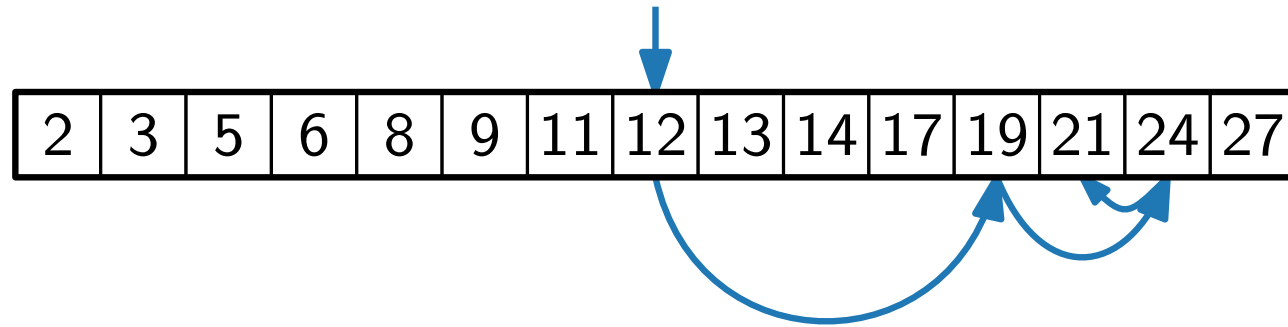
	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

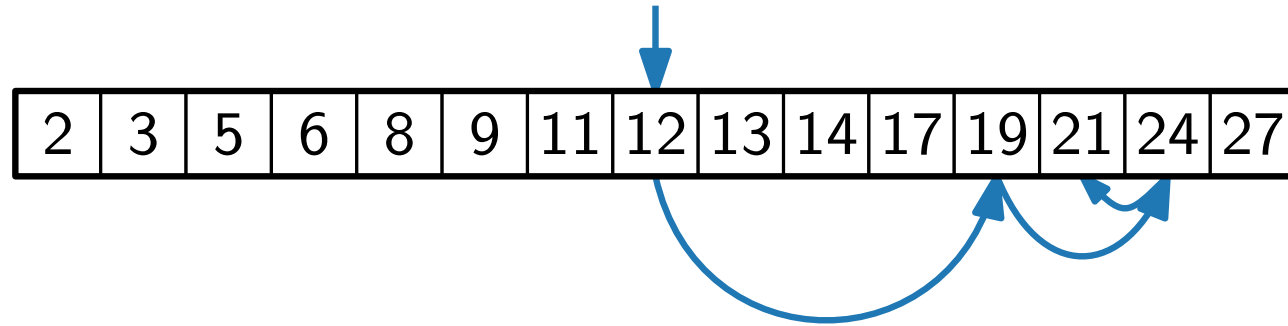
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

$\leq$ $+ 1$

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

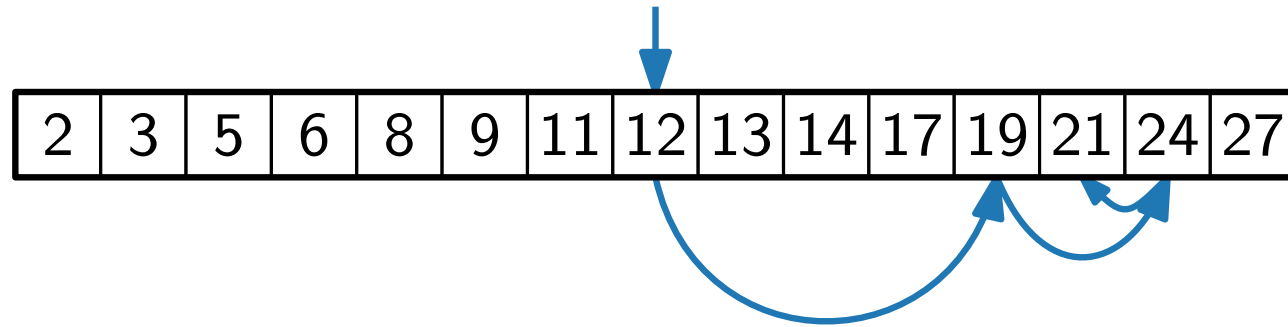
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

$$\leq T(\lfloor n/4 \rfloor) + 1 + 1$$

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

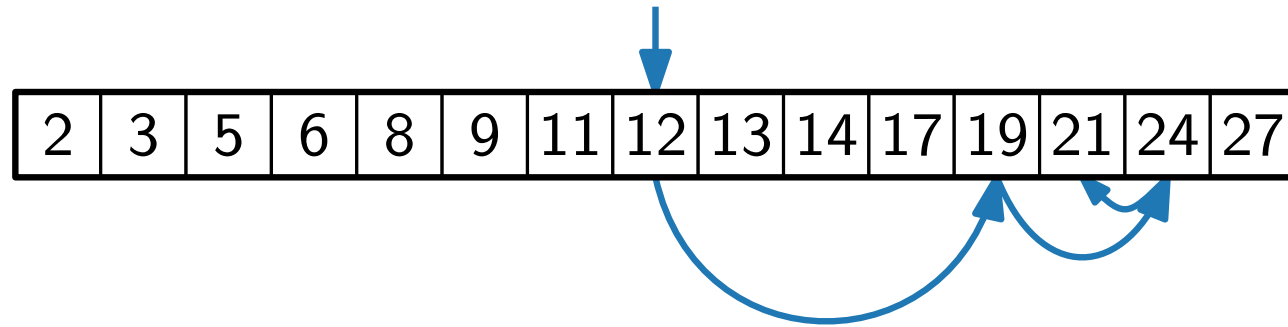
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

$$\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq$$

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

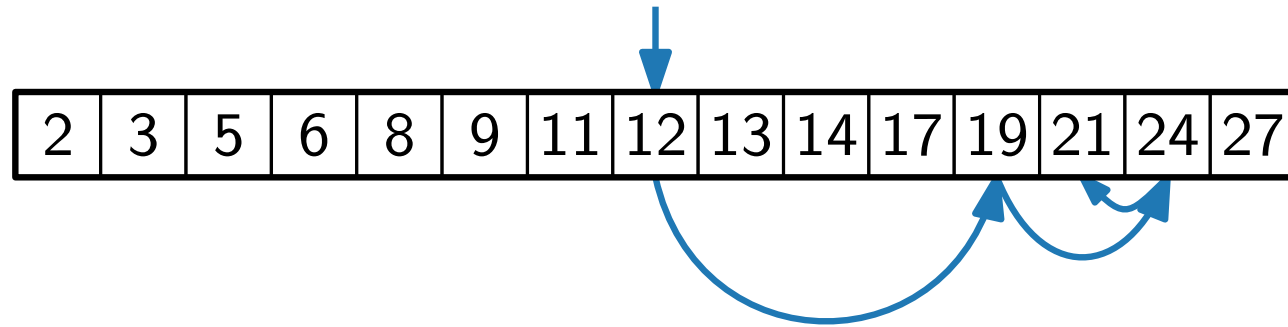
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

$$\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq T(1) + 1 + \dots + 1$$

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

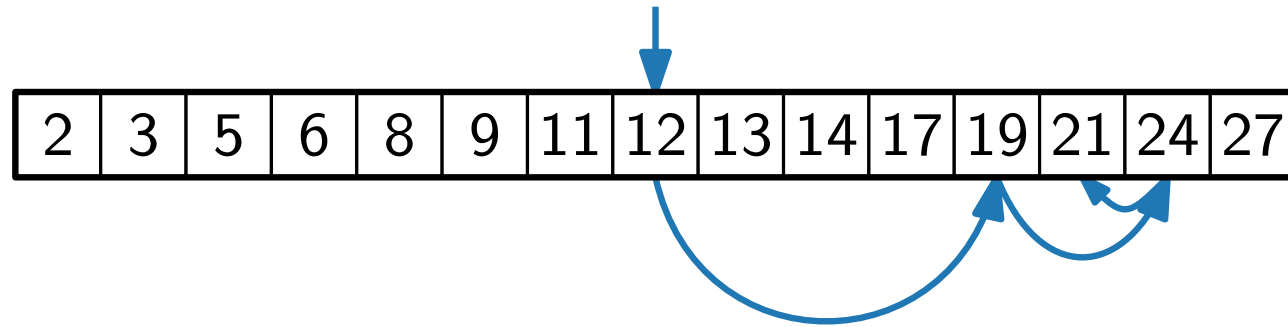
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

$$\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq T(1) + \underbrace{1 + \dots + 1}$$

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

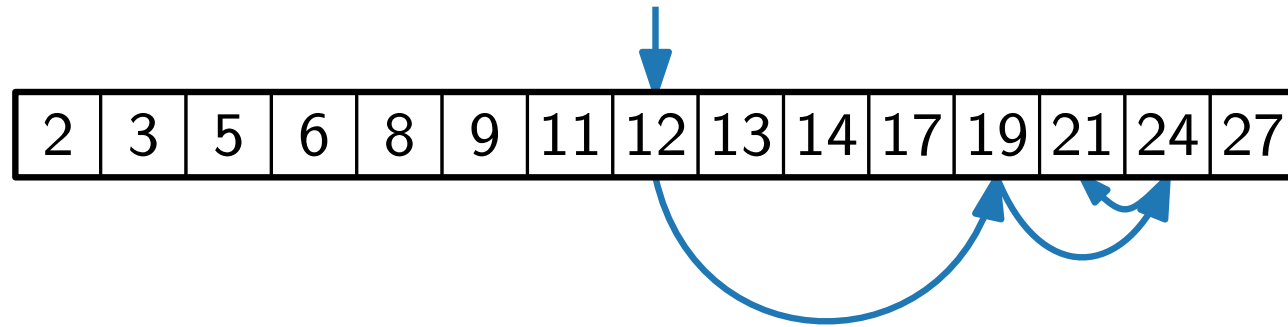
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

$$\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq T(1) + \underbrace{1 + \dots + 1}_{\lfloor \log_2 n \rfloor}$$

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

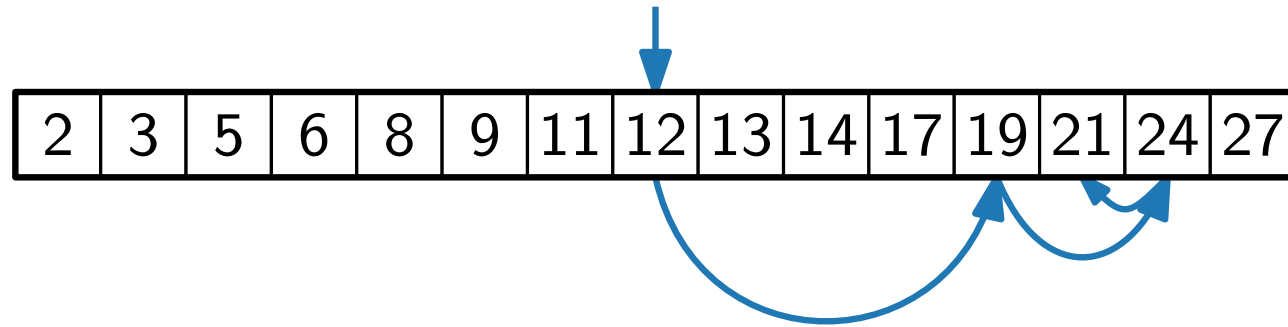
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

$$\begin{aligned}
 &\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq T(1) + \underbrace{1 + \dots + 1}_{\lfloor \log_2 n \rfloor} \\
 &= 1 + \lfloor \log_2 n \rfloor
 \end{aligned}$$

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	?	Schritte

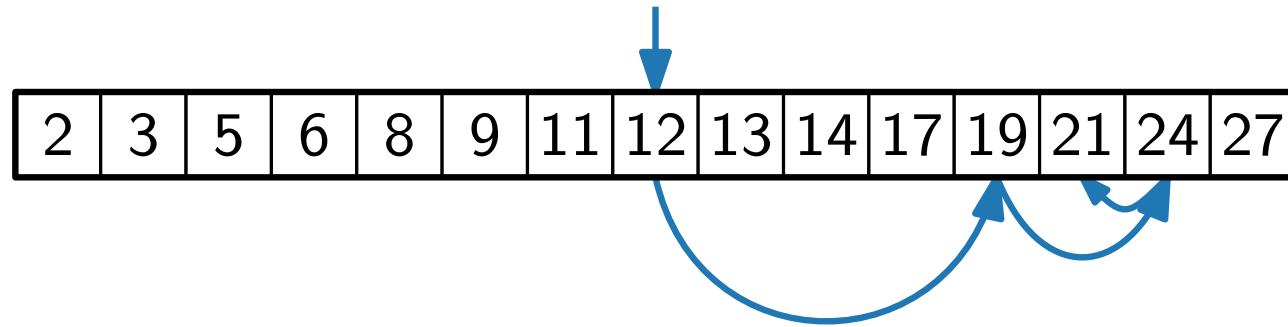
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

$$\begin{aligned}
 &\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq T(1) + \underbrace{1 + \dots + 1}_{\lfloor \log_2 n \rfloor} \\
 &= 1 + \lfloor \log_2 n \rfloor = \lceil \log_2(n+1) \rceil
 \end{aligned}$$

Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	$\lceil \log_2(n+1) \rceil$	Schritte

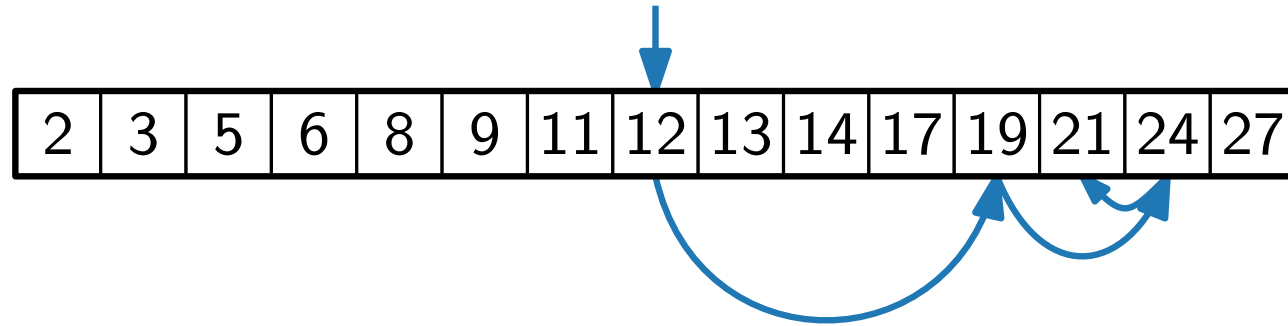
Je nach Implementierung braucht ein Schritt ein oder zwei Vergleiche (z.B. = und <).

grob: Wie oft muss ich n halbieren, bis ich bei 1 bin?

genau: $T(n) \leq T(\lfloor n/2 \rfloor) + 1$ und $T(1) = 1$

$$\begin{aligned}
 &\leq T(\lfloor n/4 \rfloor) + 1 + 1 \leq \dots \leq T(1) + \underbrace{1 + \dots + 1}_{\lfloor \log_2 n \rfloor} \\
 &= 1 + \lfloor \log_2 n \rfloor = \lceil \log_2(n+1) \rceil
 \end{aligned}$$

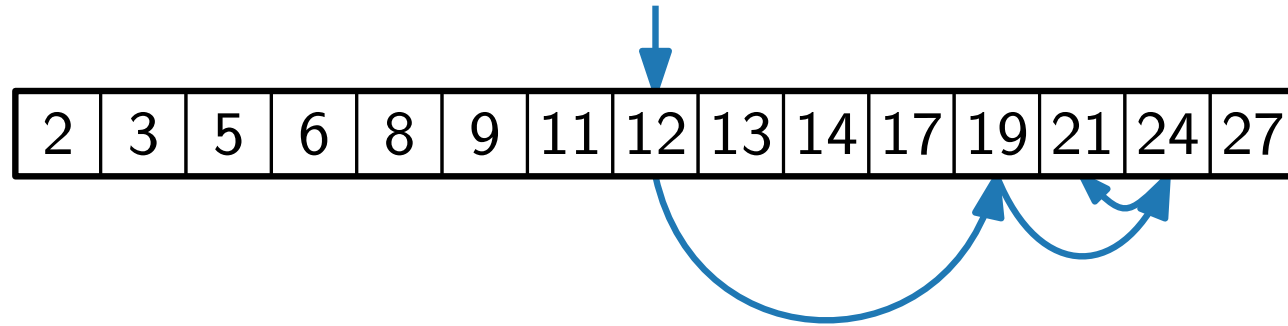
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case	
Lineare Suche:	13	n	Schritte
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte

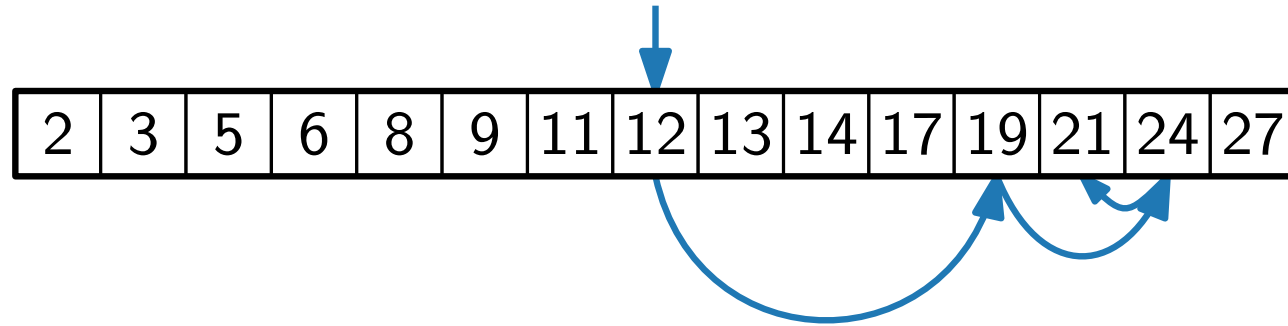
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case		
Lineare Suche:	13	n	Schritte	
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

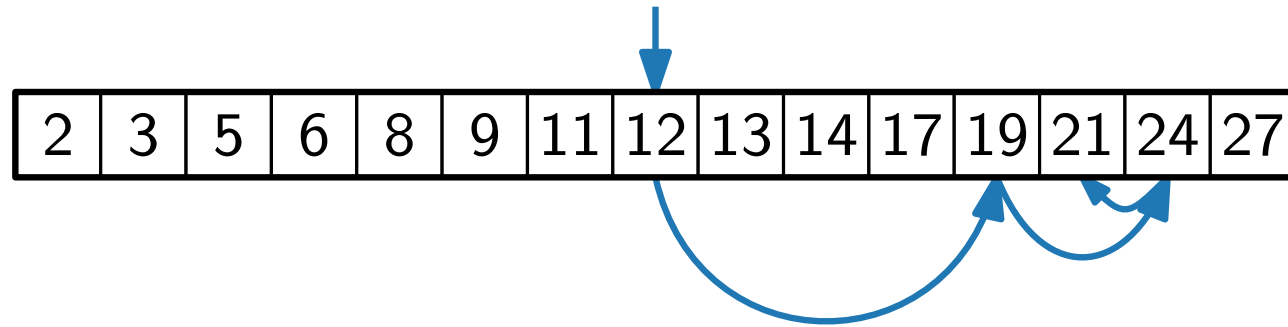
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case		
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

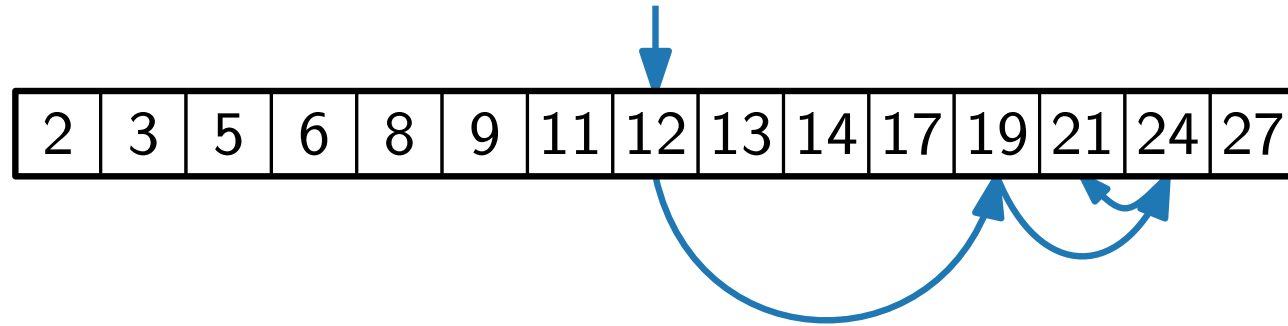
Suche im sortierten Feld



SEARCH(21)

	hier	im Worst Case		$\approx 1 \text{ Mio.}$
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

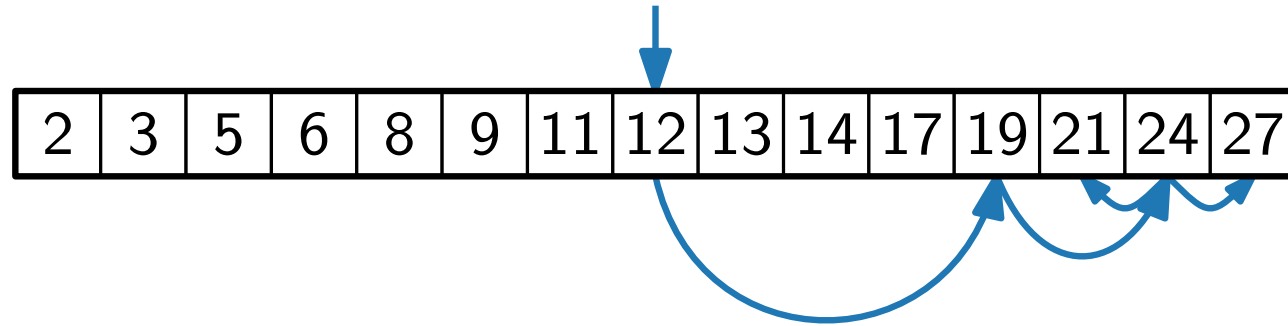
Suche im sortierten Feld



SEARCH(21)
SEARCH(27)

	hier	im Worst Case		≈ 1 Mio.
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

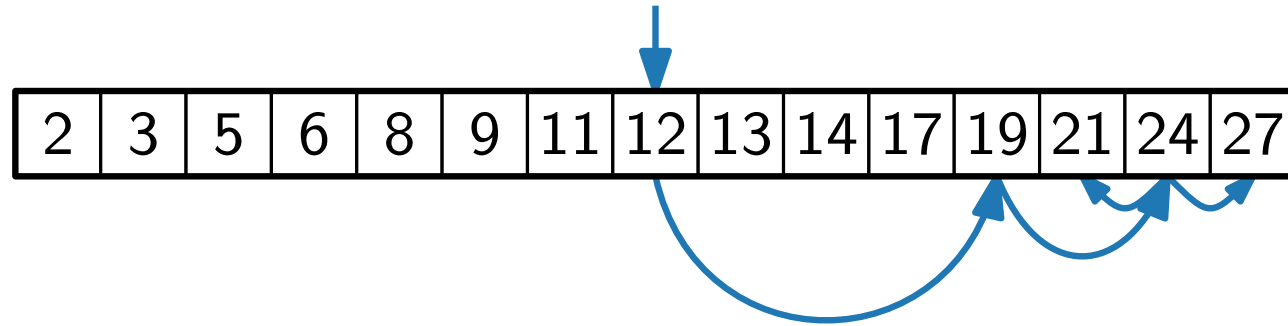
Suche im sortierten Feld



SEARCH(21)
SEARCH(27)

	hier	im Worst Case		≈ 1 Mio.
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

Suche im sortierten Feld



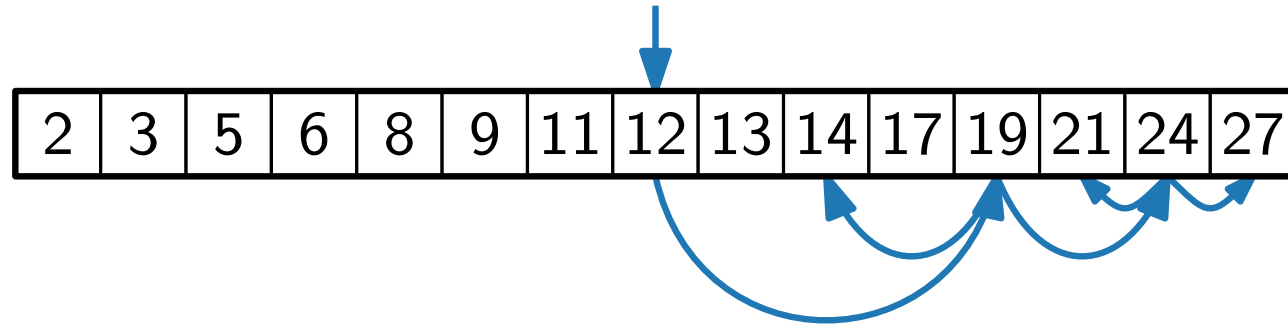
SEARCH(21)

SEARCH(27)

SEARCH(17)

	hier	im Worst Case		≈ 1 Mio.
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

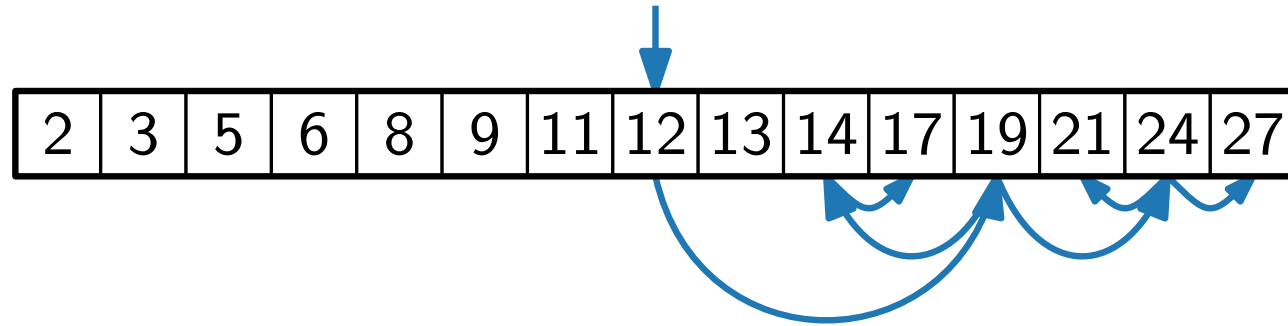
Suche im sortierten Feld



SEARCH(21)
 SEARCH(27)
 SEARCH(17)

	hier	im Worst Case		$\approx 1 \text{ Mio.}$
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

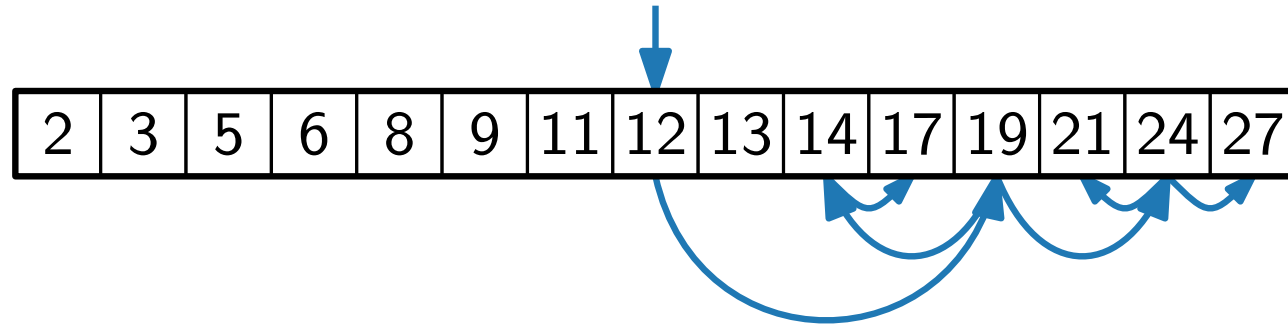
Suche im sortierten Feld



SEARCH(21)
 SEARCH(27)
 SEARCH(17)

	hier	im Worst Case		$\approx 1 \text{ Mio.}$
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

Suche im sortierten Feld



SEARCH(21)

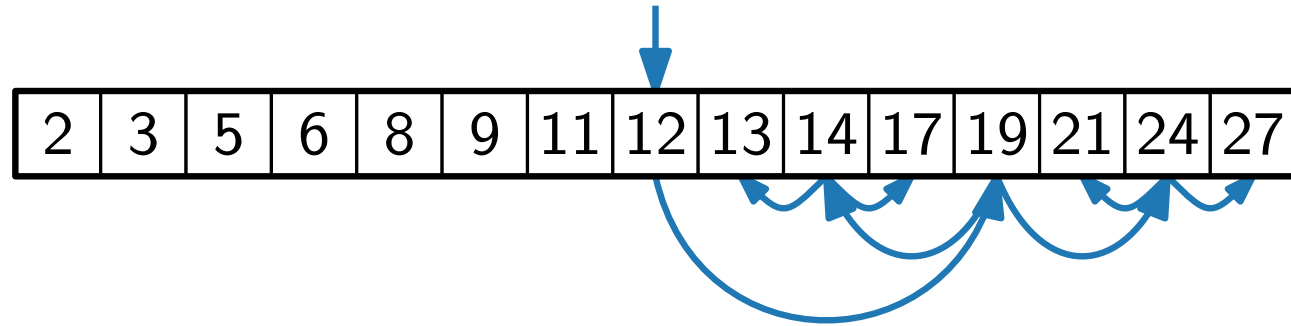
SEARCH(27)

SEARCH(17)

SEARCH(13)

	hier	im Worst Case		$\approx 1 \text{ Mio.}$
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

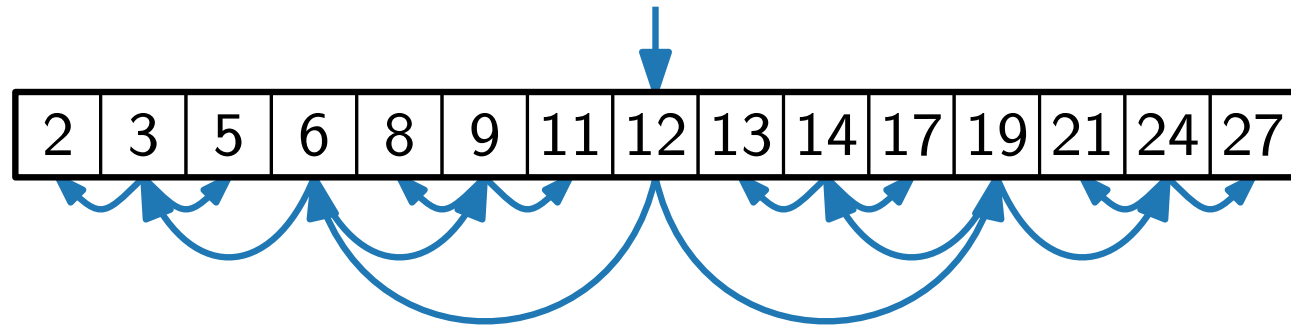
Suche im sortierten Feld



SEARCH(21)
 SEARCH(27)
 SEARCH(17)
 SEARCH(13)

	hier	im Worst Case		$\approx 1 \text{ Mio.}$
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

Suche im sortierten Feld



SEARCH(21)

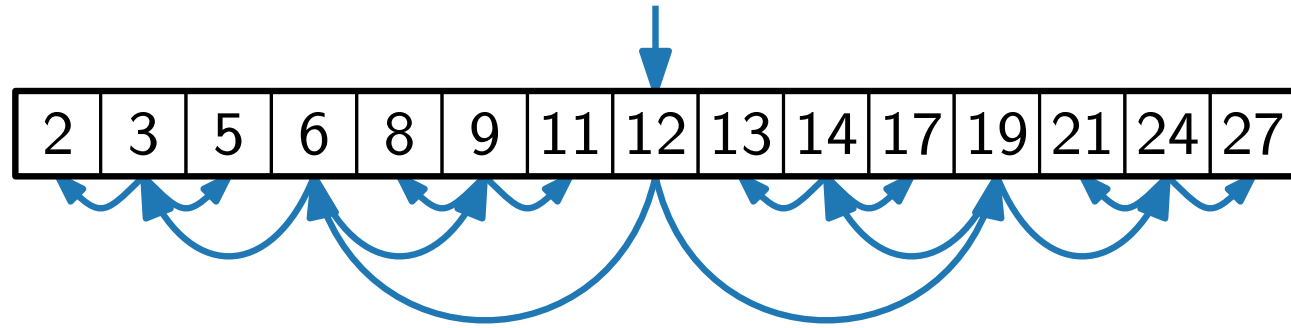
SEARCH(27)

SEARCH(17)

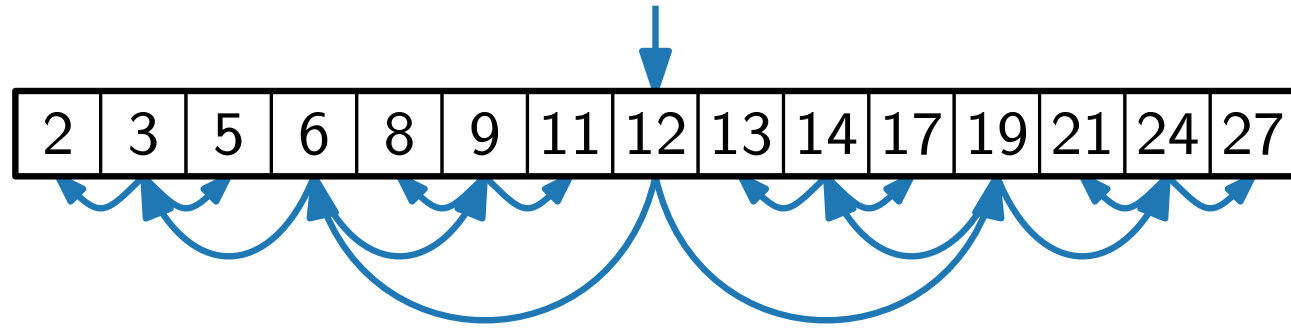
SEARCH(13)

	hier	im Worst Case		$\approx 1 \text{ Mio.}$
Lineare Suche:	13	n	Schritte	$2^{20} - 1$
Binäre Suche:	4	$\lceil \log_2(n + 1) \rceil$	Schritte	20

Suche im sortierten Feld

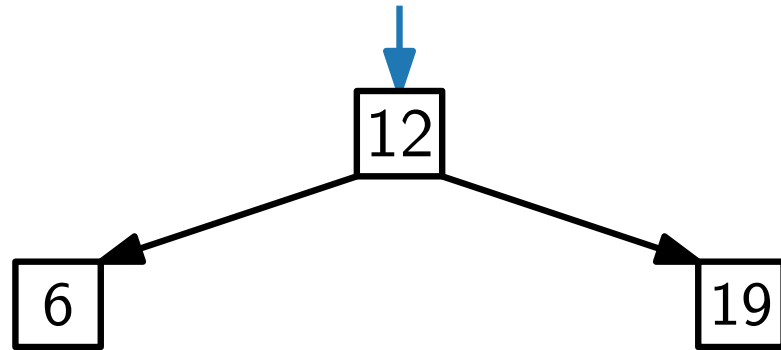
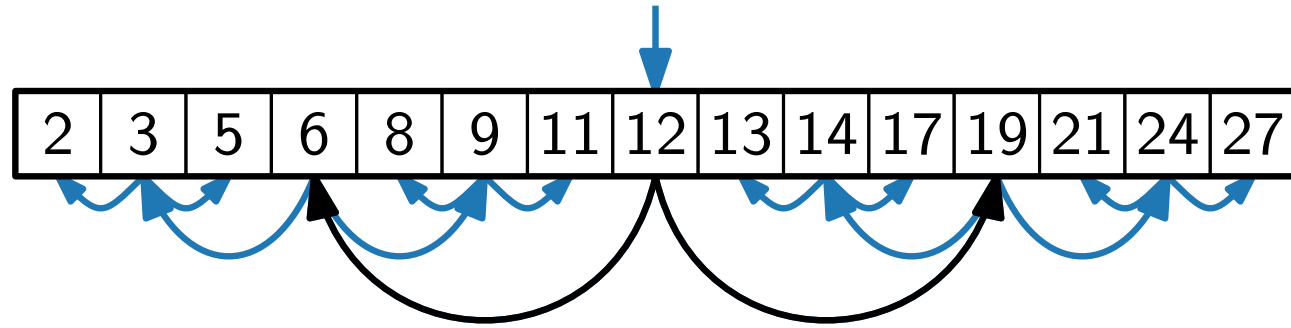


Suche im sortierten Feld

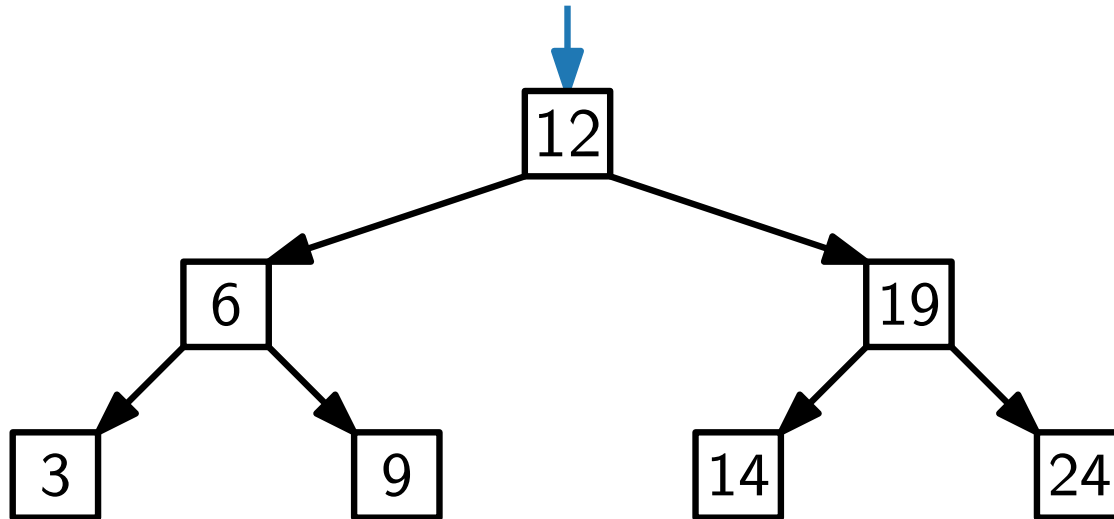
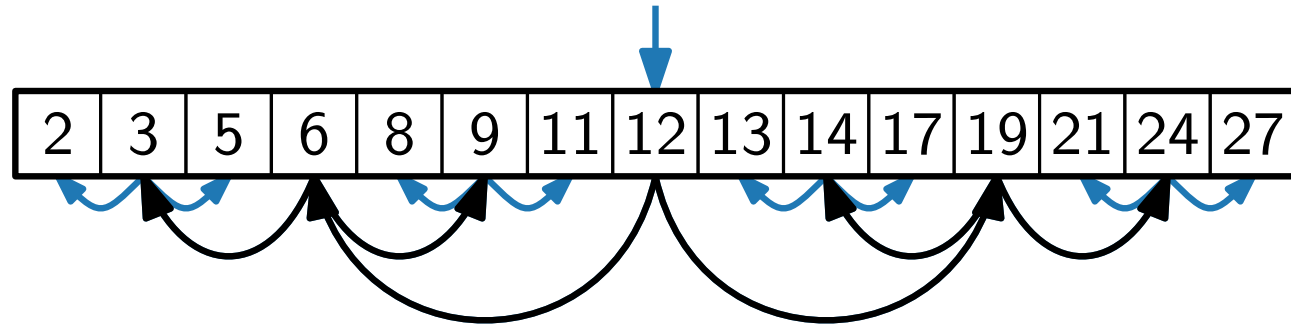


12

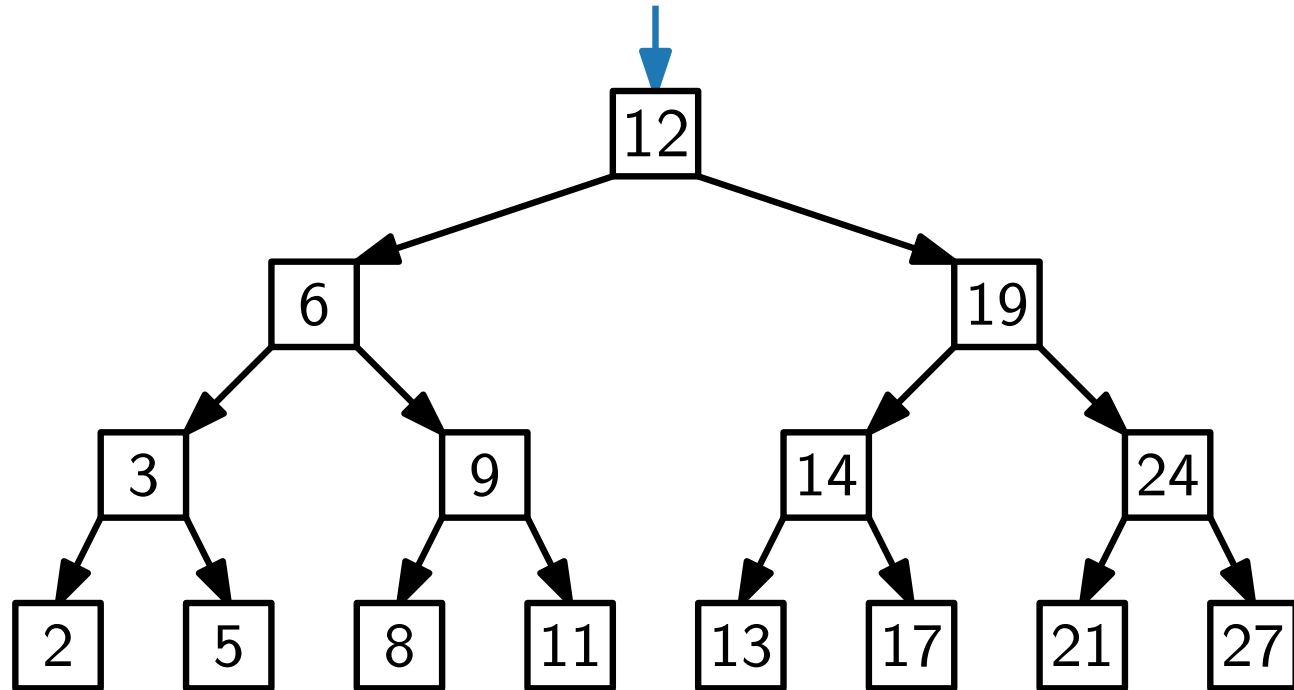
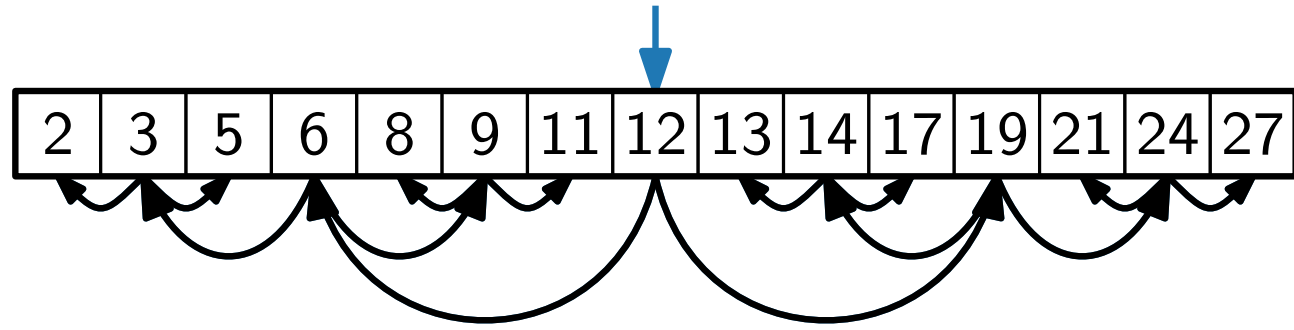
Suche im sortierten Feld



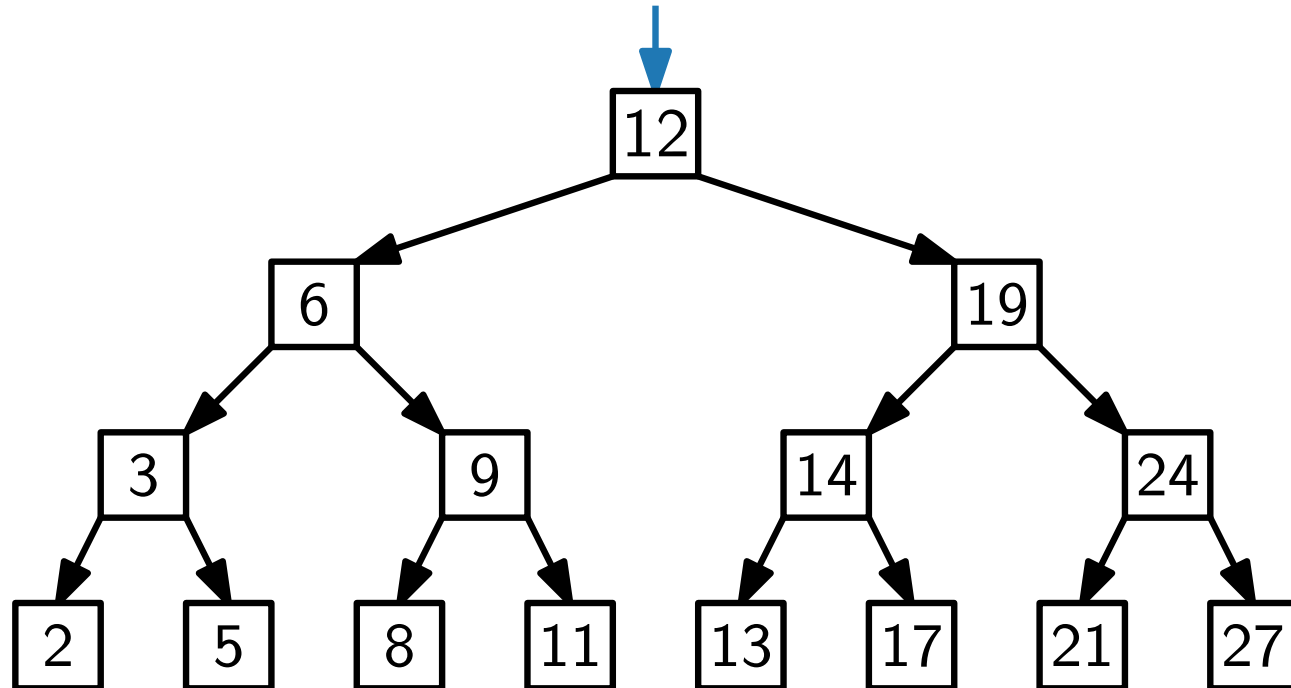
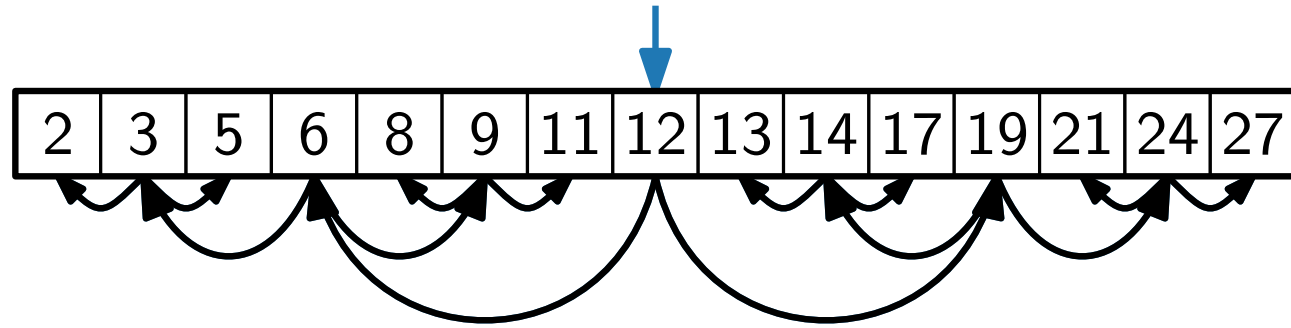
Suche im sortierten Feld



Suche im sortierten Feld

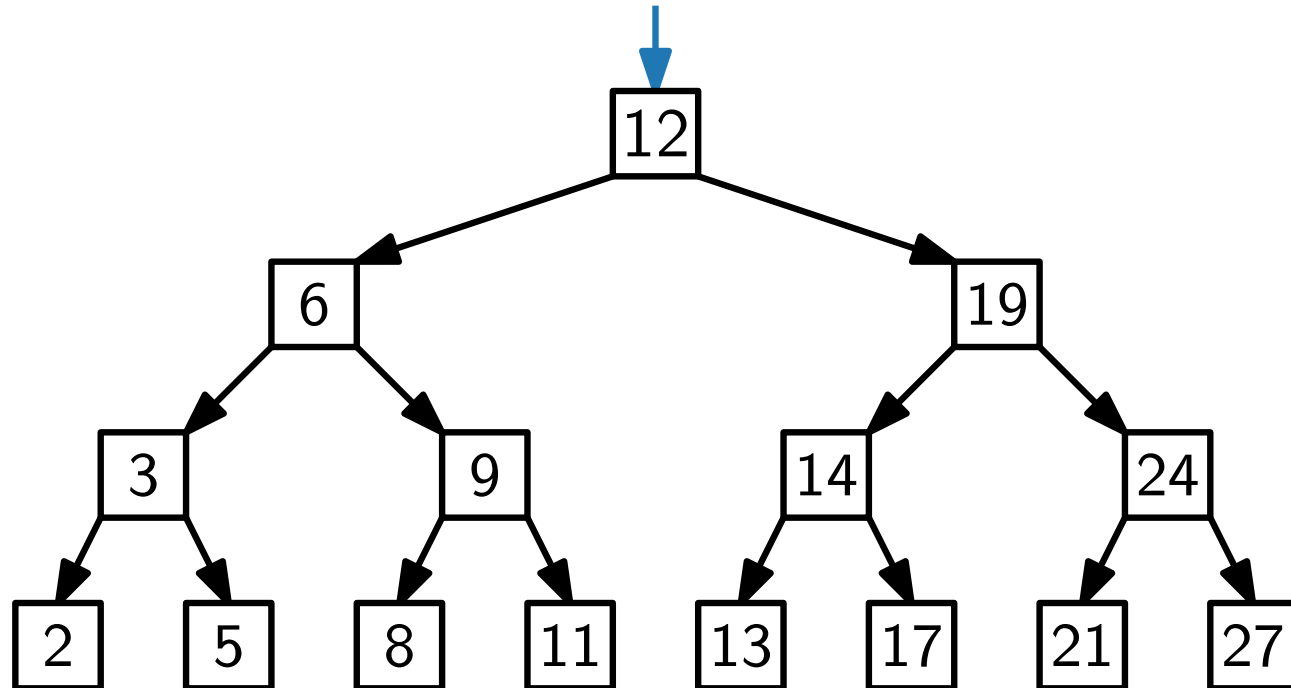
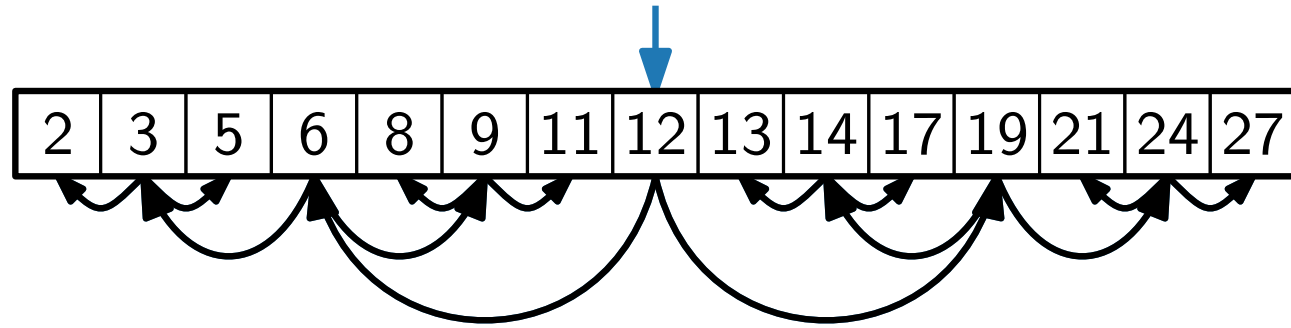


Suche im sortierten Feld



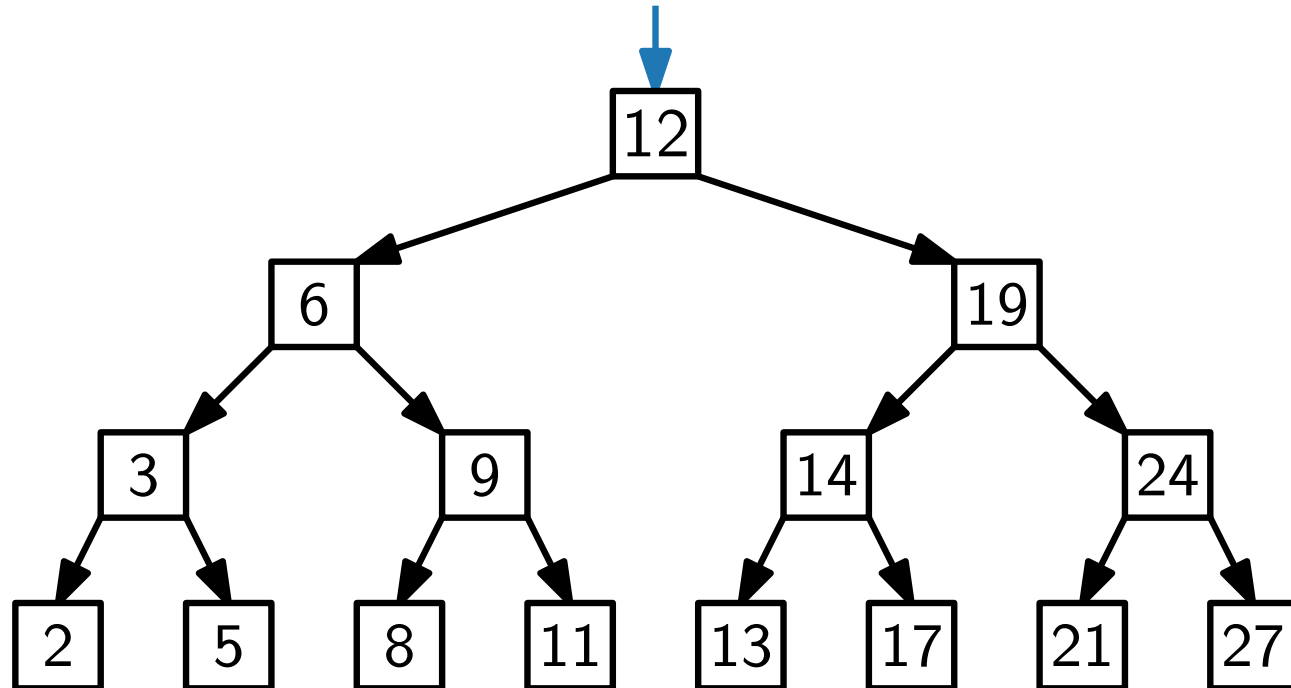
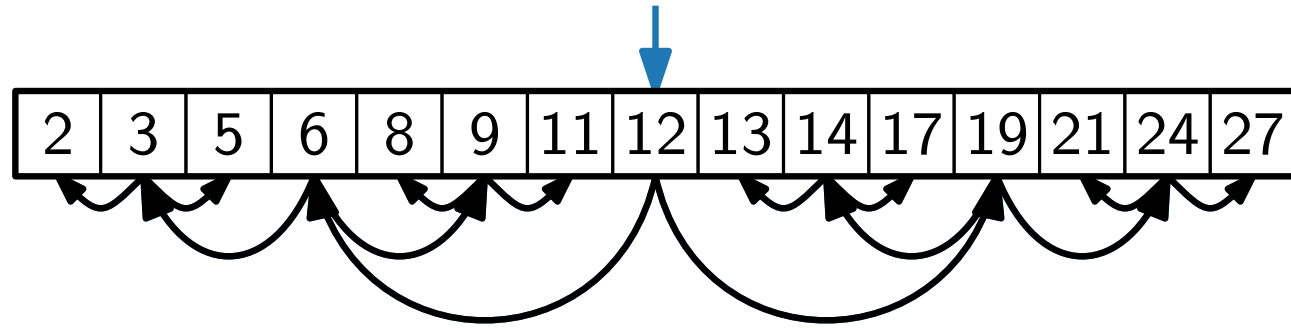
**Binärer
Suchbaum**

Suche im sortierten Feld



**Binärer
Suchbaum**

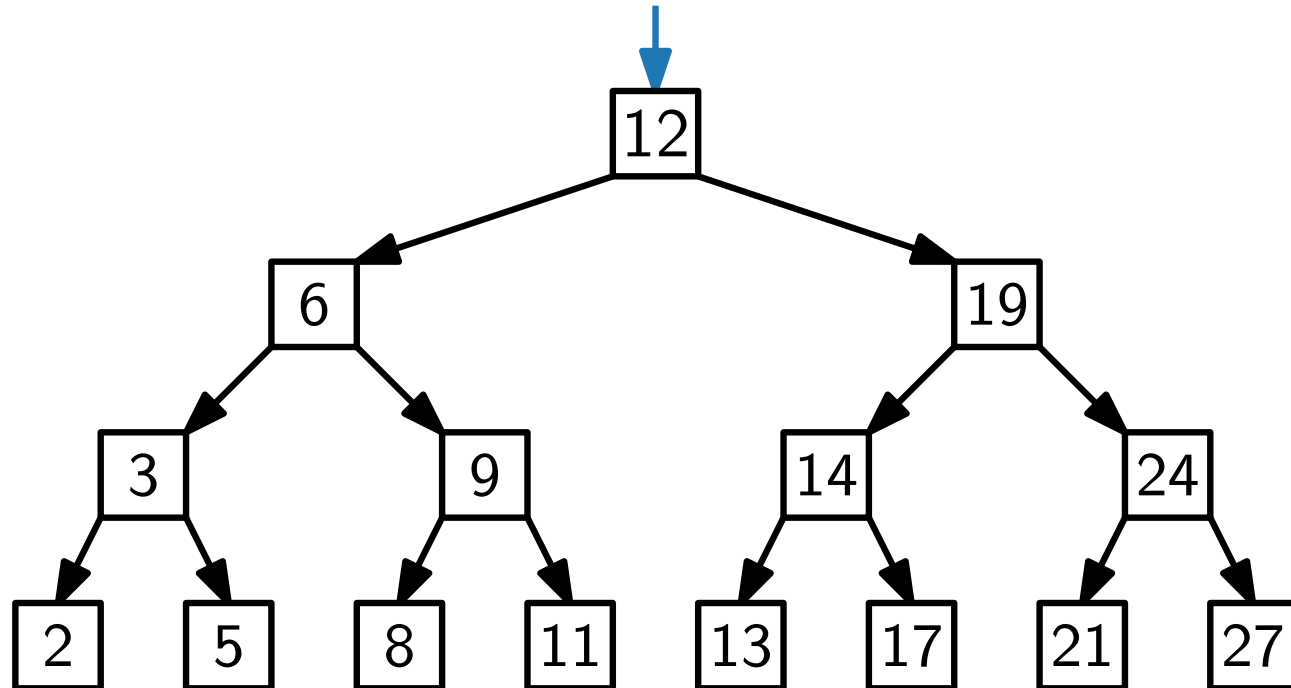
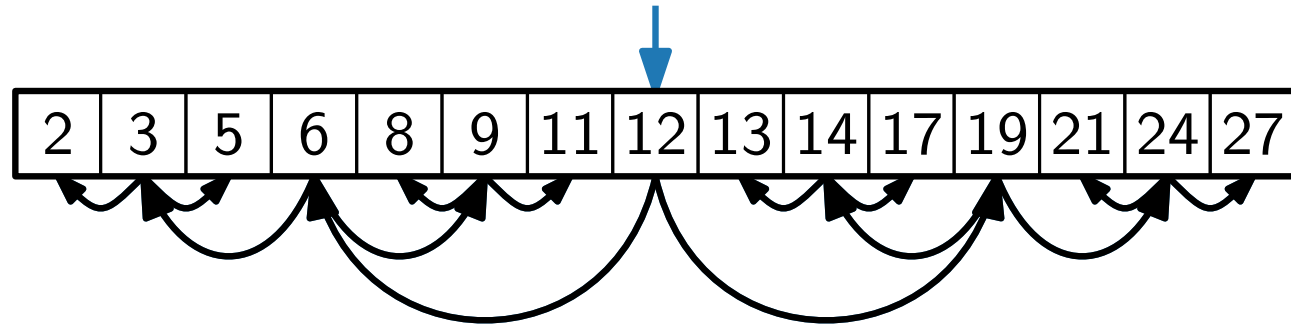
Suche im sortierten Feld



Binärer Suchbaum

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

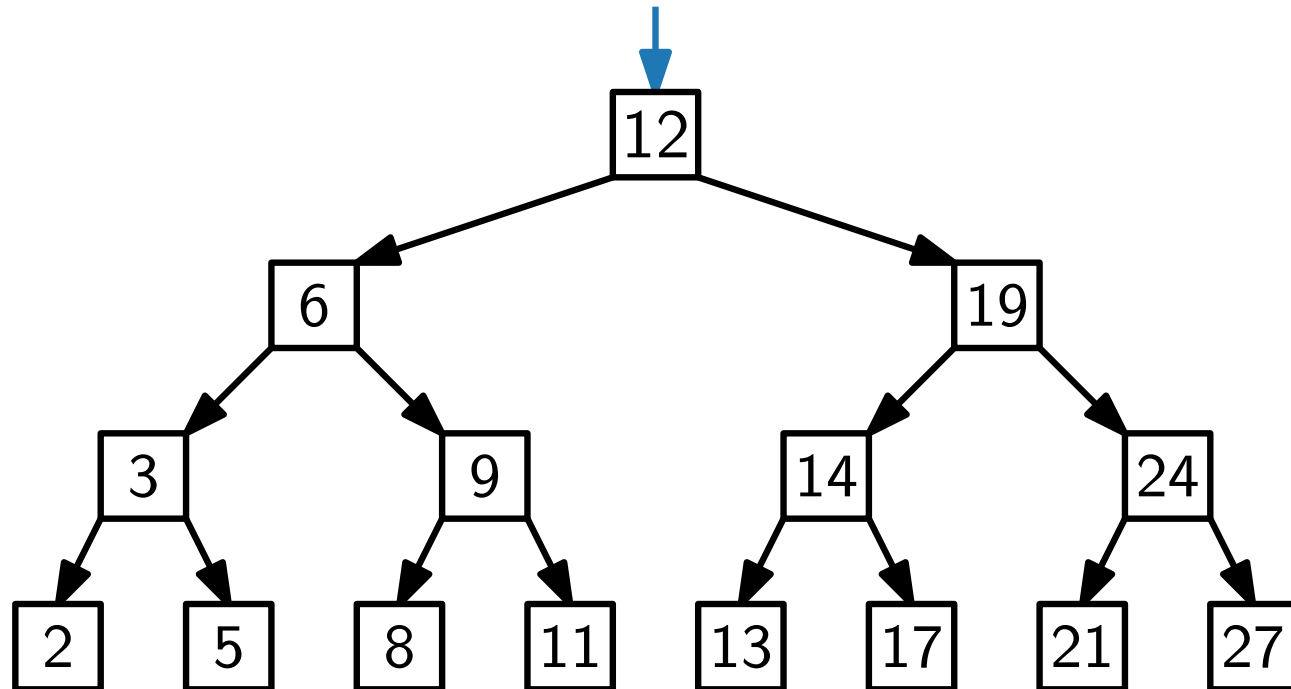
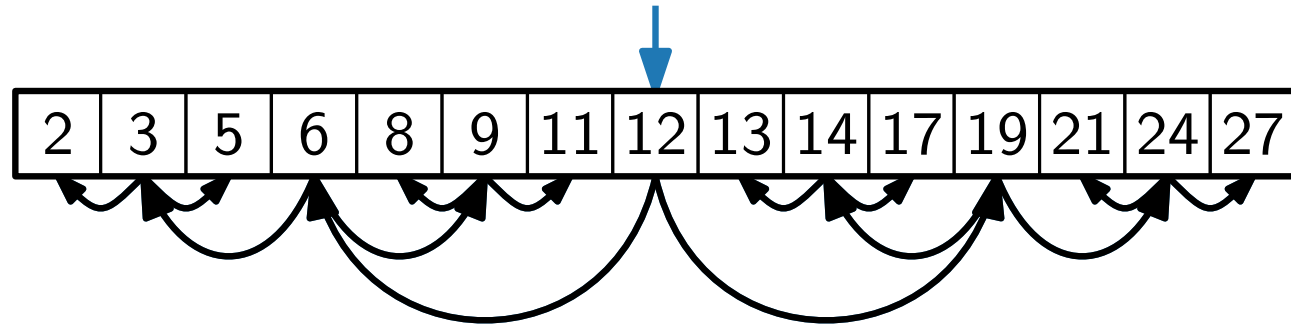
Suche im sortierten Feld



Binärer Suchbaum

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Suche im sortierten Feld

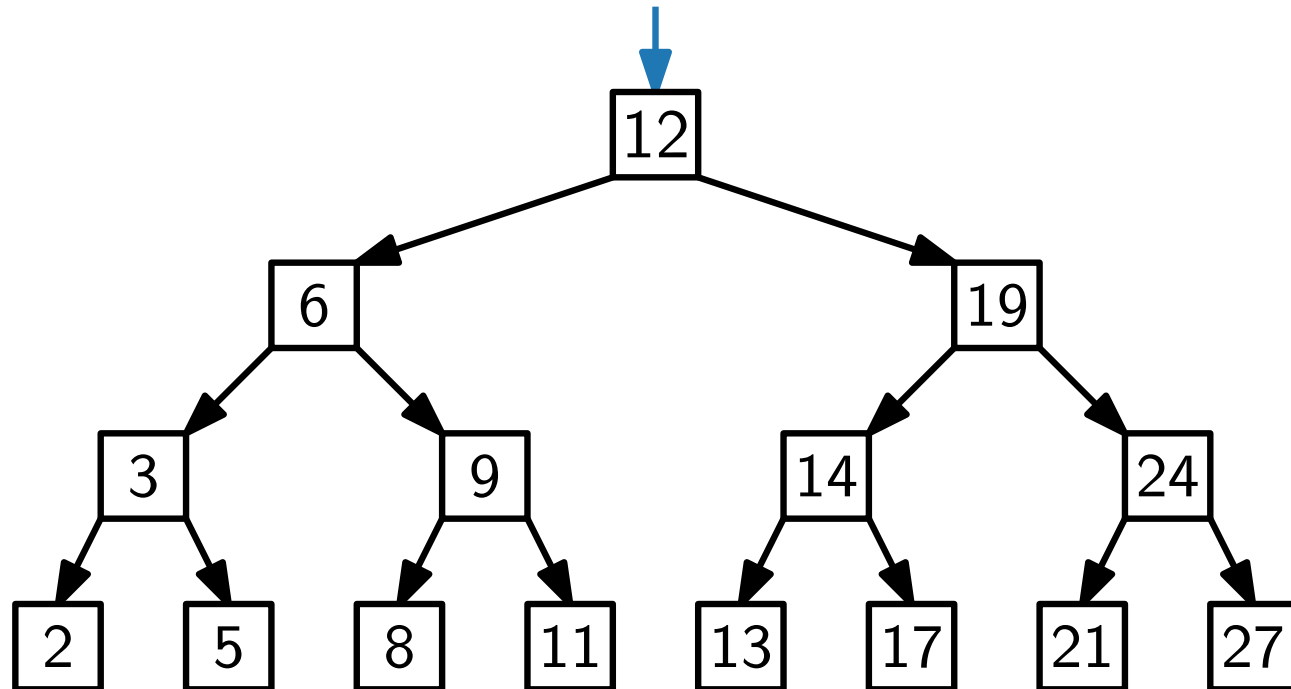
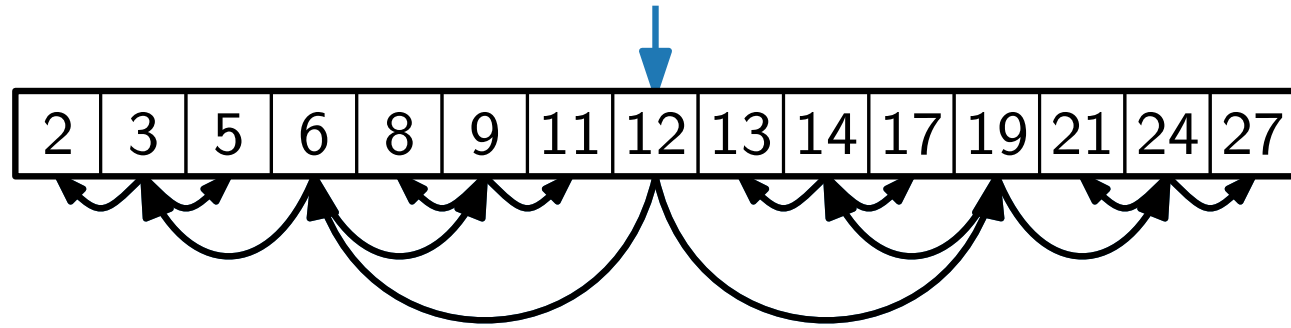


Binärer Suchbaum

Jeder Knoten hat höchstens zwei Kinder.

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Suche im sortierten Feld

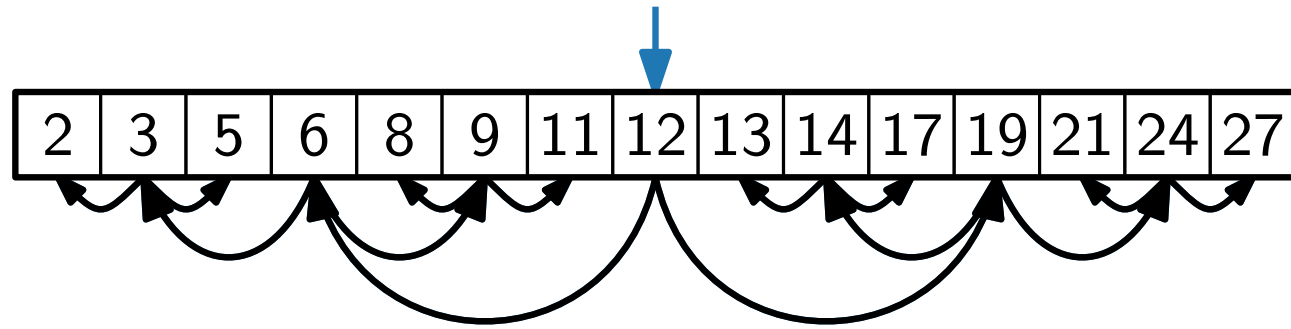


Binärer Suchbaum

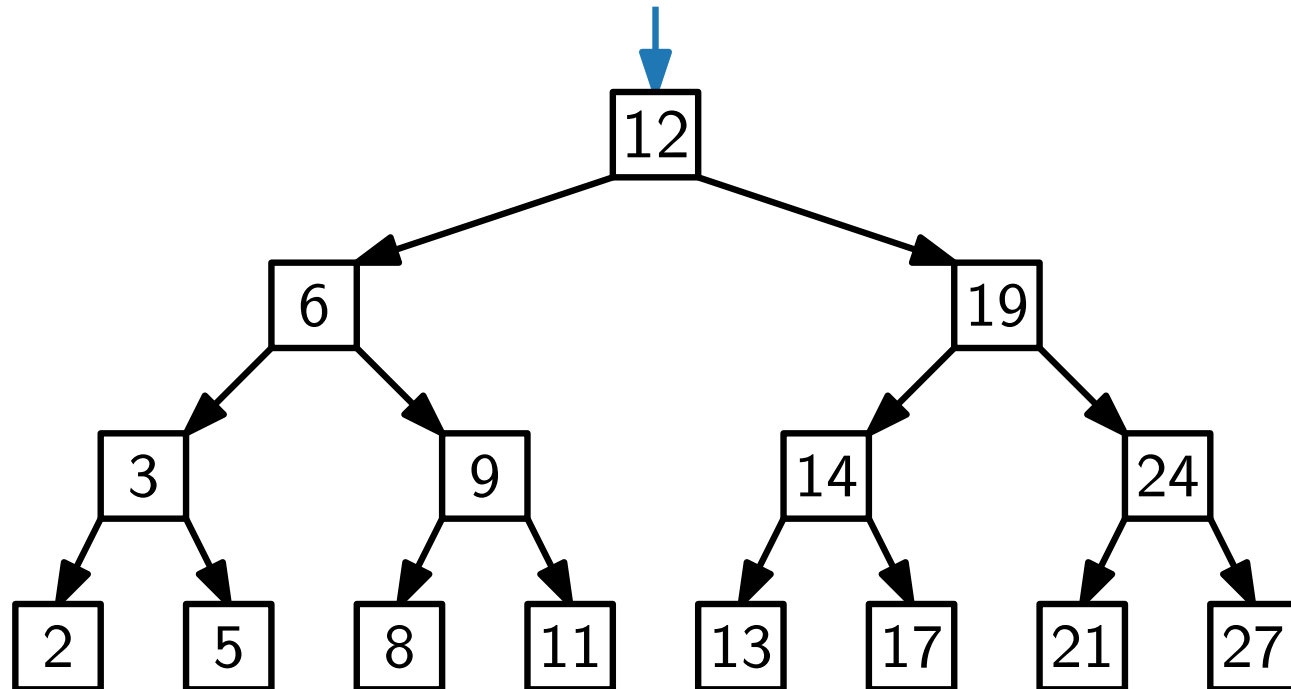
Jeder Knoten hat höchstens zwei Kinder.

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

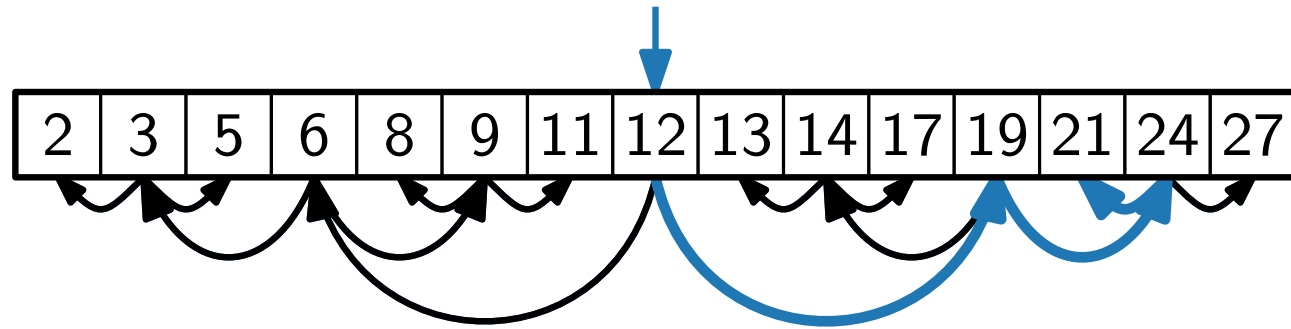


Binärer
Suchbaum

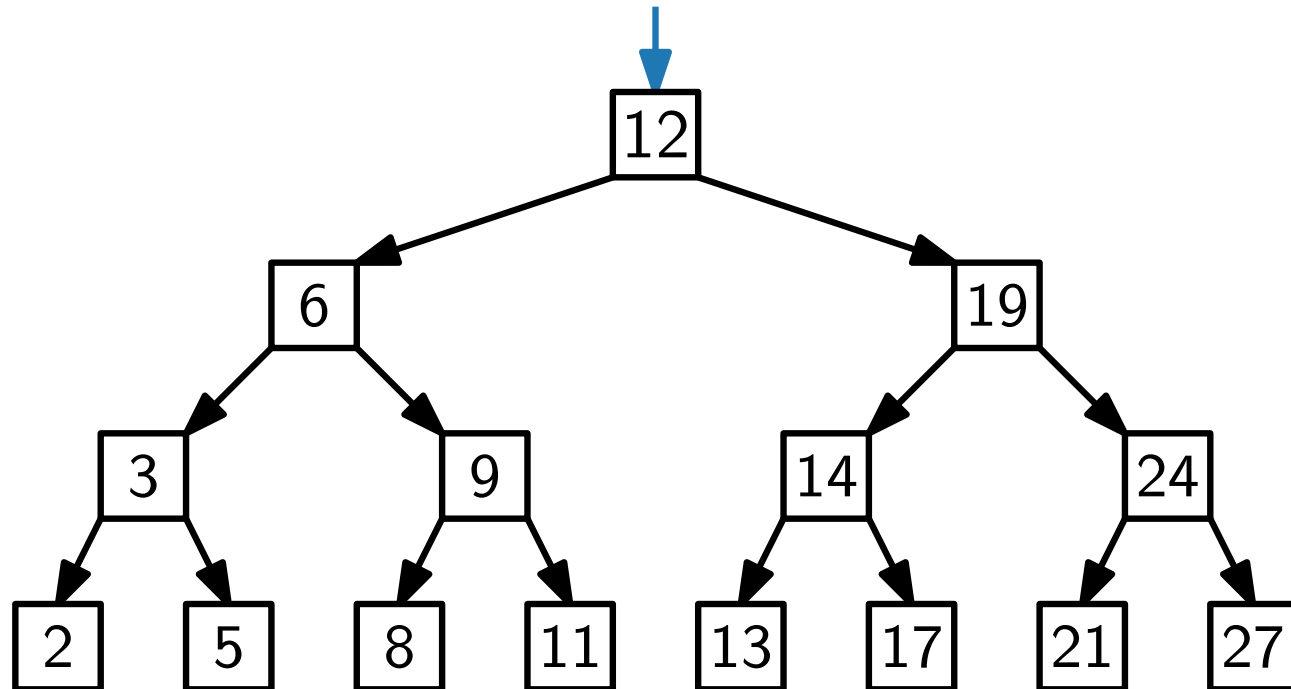
Jeder Knoten hat höchstens zwei Kinder.

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

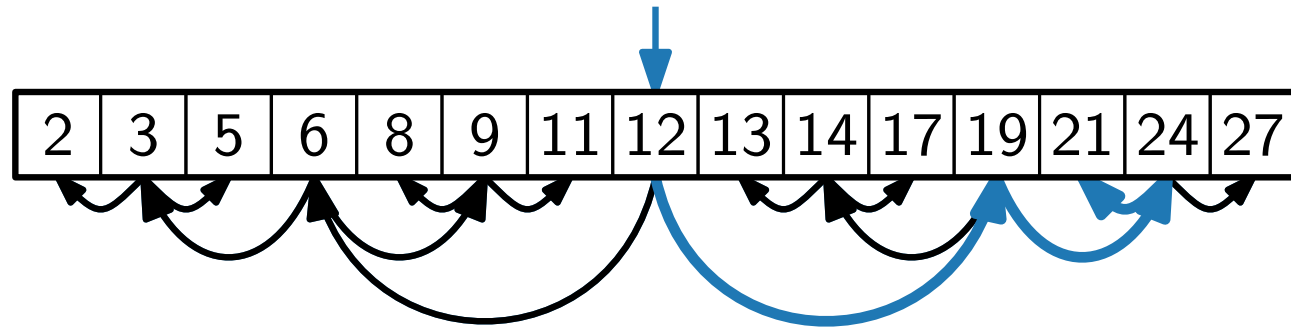


Binärer
Suchbaum

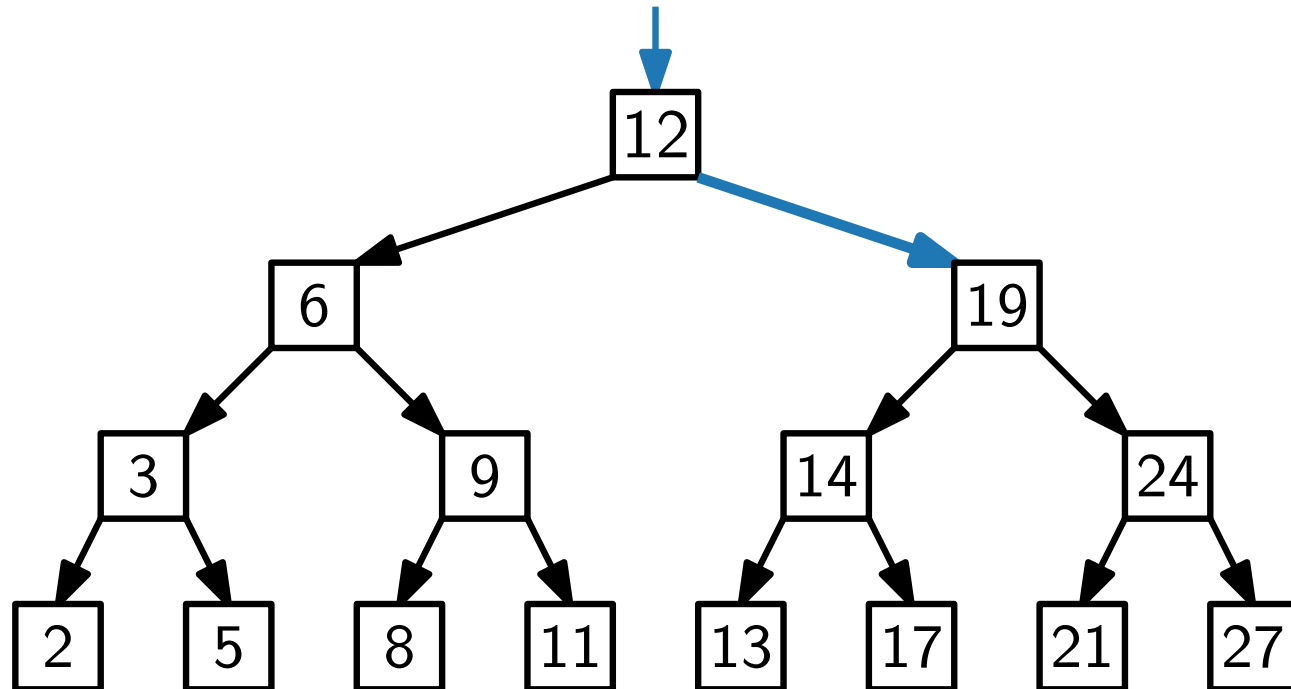
Jeder Knoten hat höchstens zwei Kinder.

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

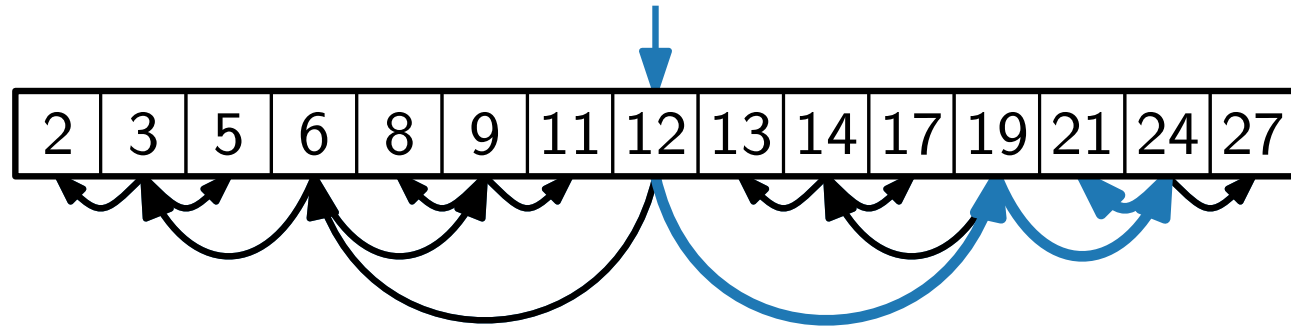


Jeder Knoten hat höchstens zwei Kinder.

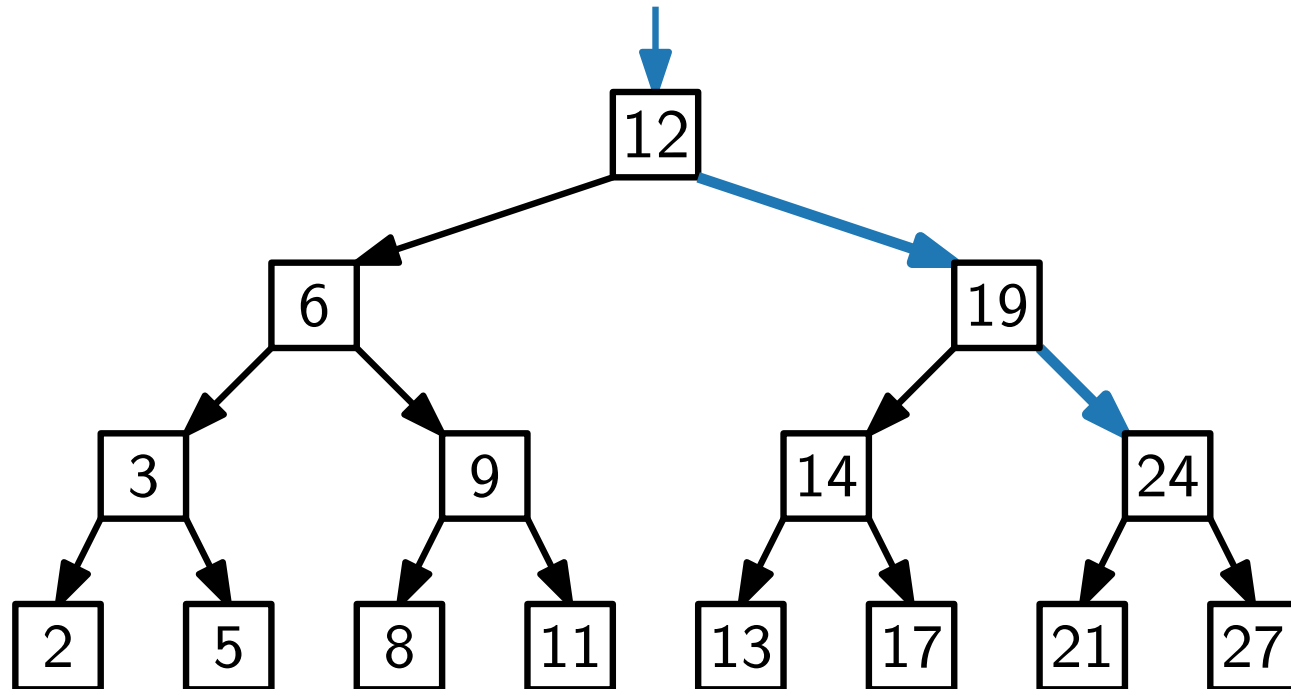
Binärer
Suchbaum

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

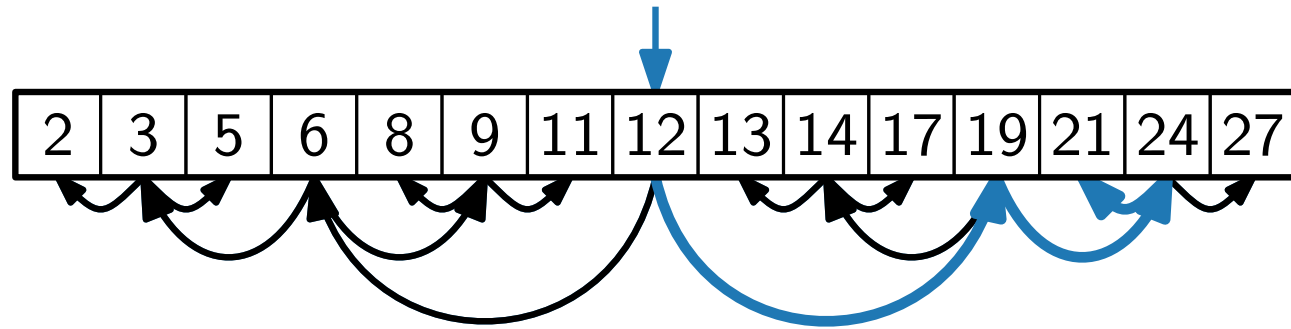


**Binärer
Suchbaum**

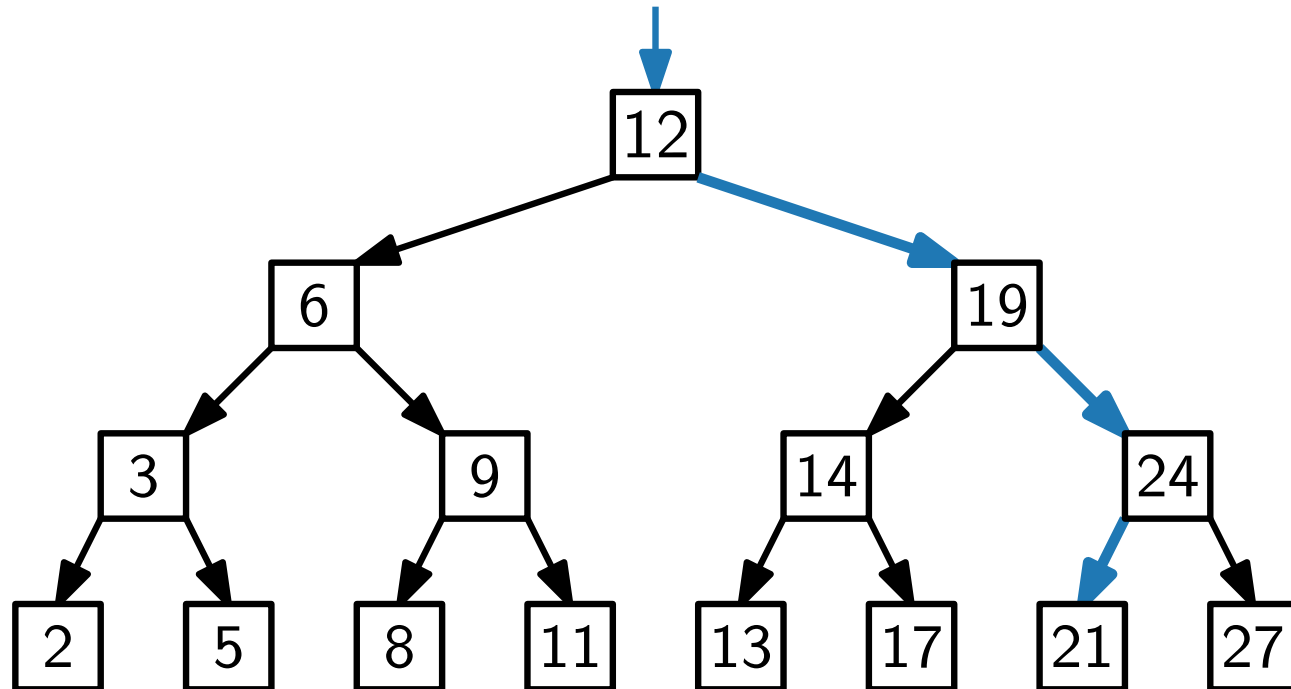
Jeder Knoten hat höchstens zwei Kinder.

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)

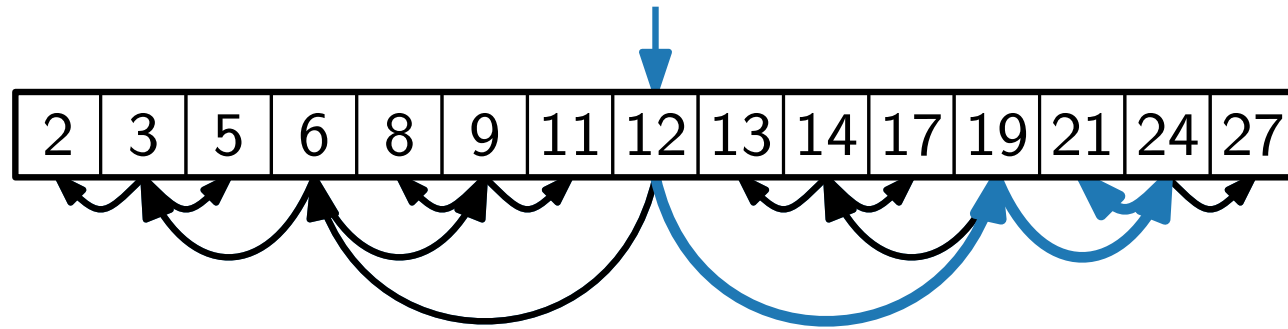


Jeder Knoten hat höchstens zwei Kinder.

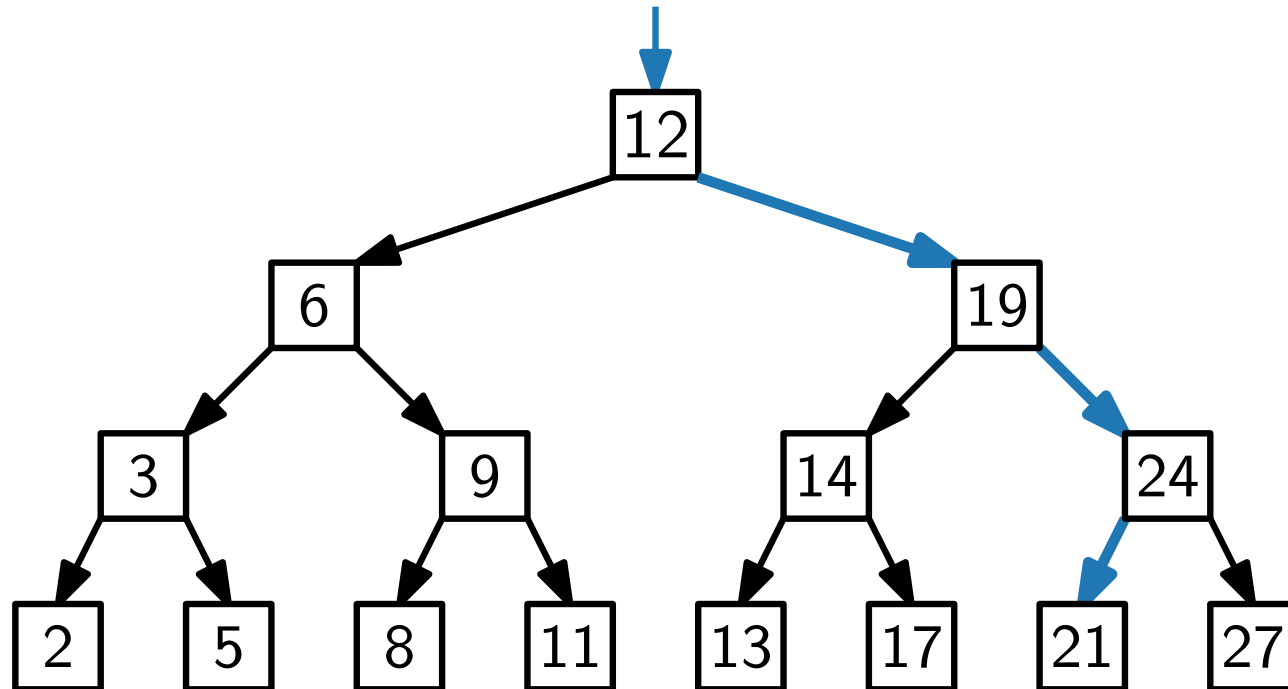
Binärer
Suchbaum

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Suche im sortierten Feld



SEARCH(21)



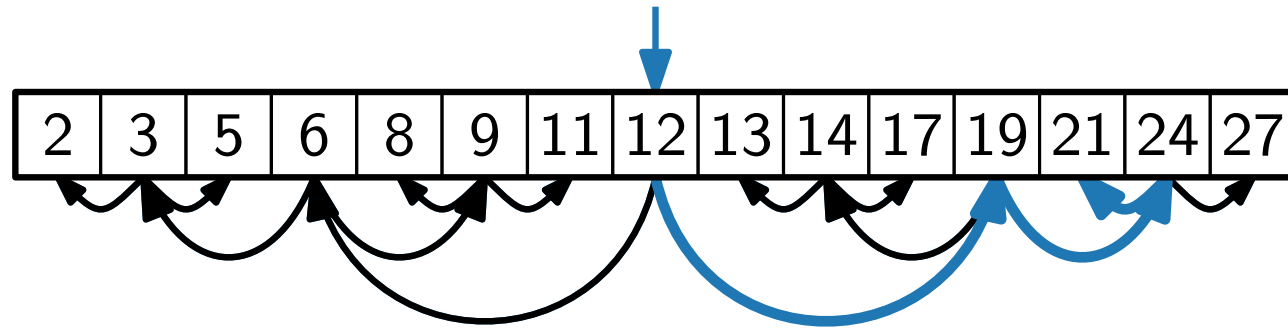
Jeder Knoten hat höchstens zwei Kinder.

**Binärer
Suchbaum**

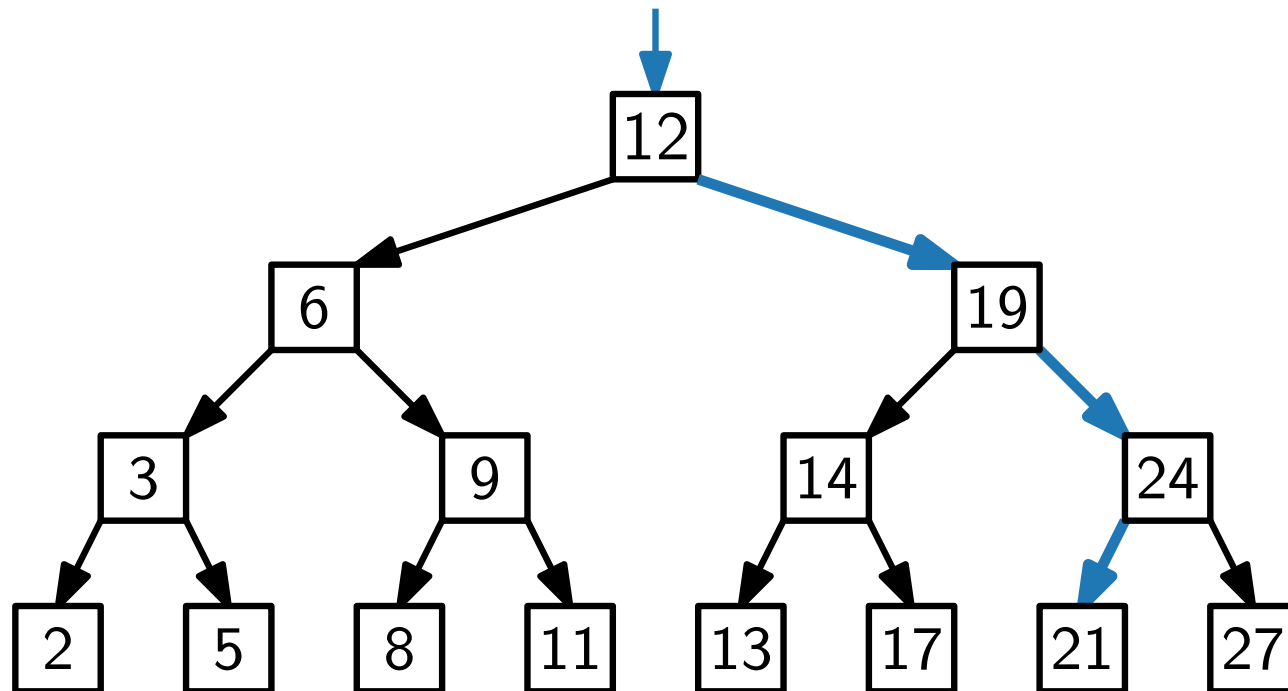
zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Binärer-Suchbaum-Eigenschaft.

Suche im sortierten Feld



SEARCH(21)



Jeder Knoten hat höchstens zwei Kinder.

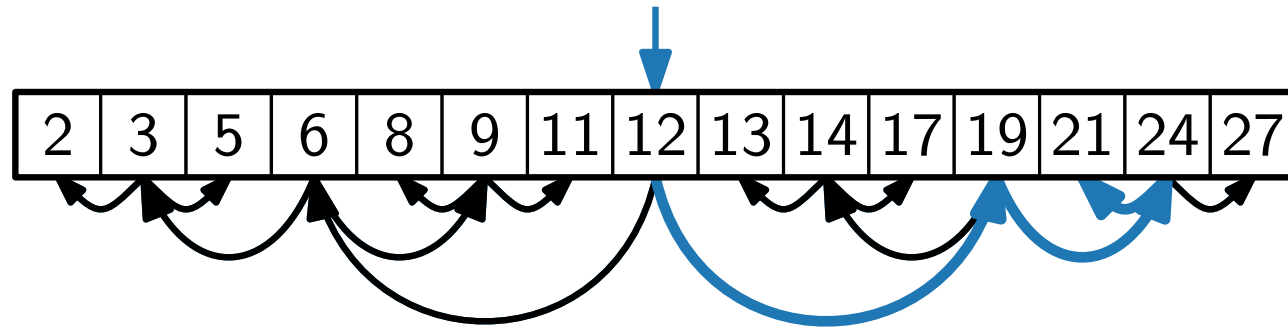
**Binärer
Suchbaum**

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

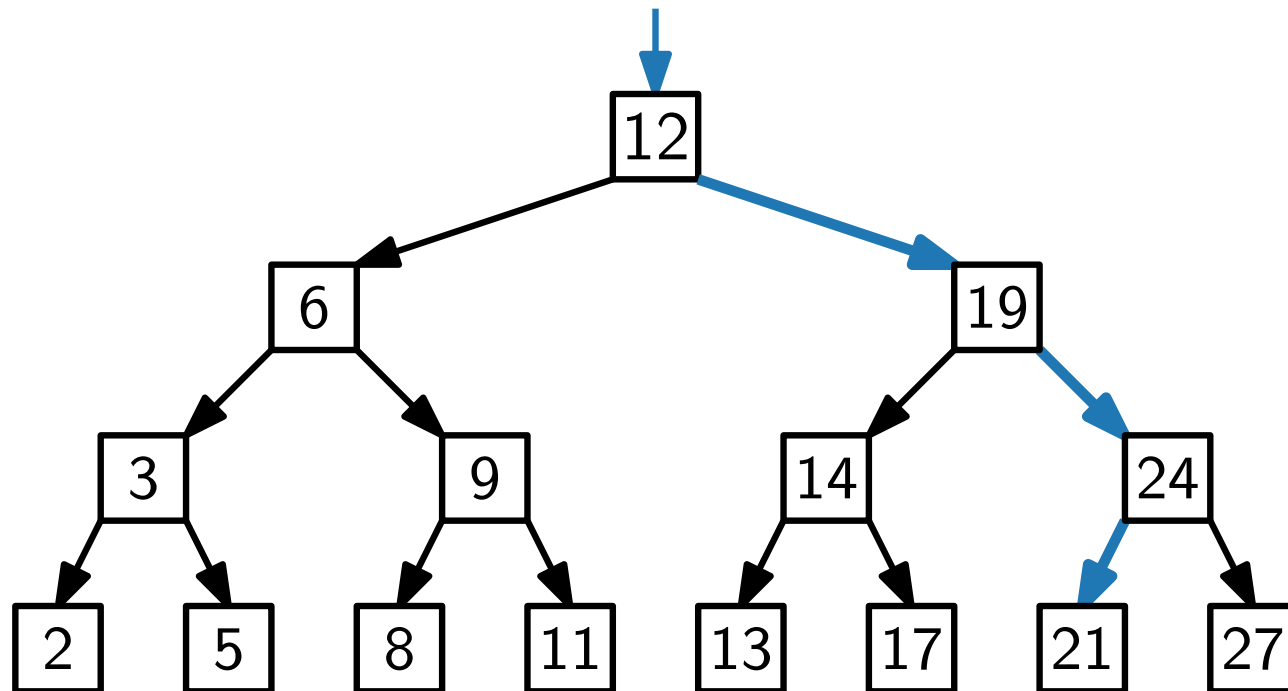
Binärer-Suchbaum-Eigenschaft.

Für jeden Knoten v gilt:

Suche im sortierten Feld



SEARCH(21)



Jeder Knoten hat höchstens zwei Kinder.

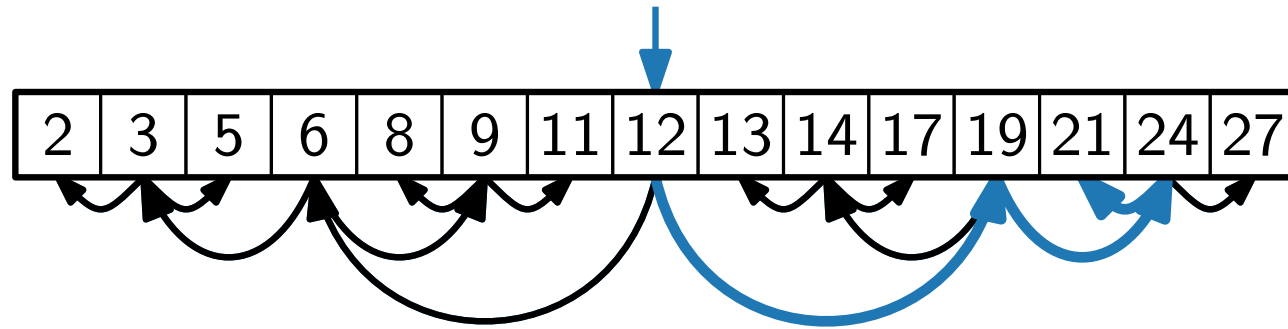
**Binärer
Suchbaum**

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

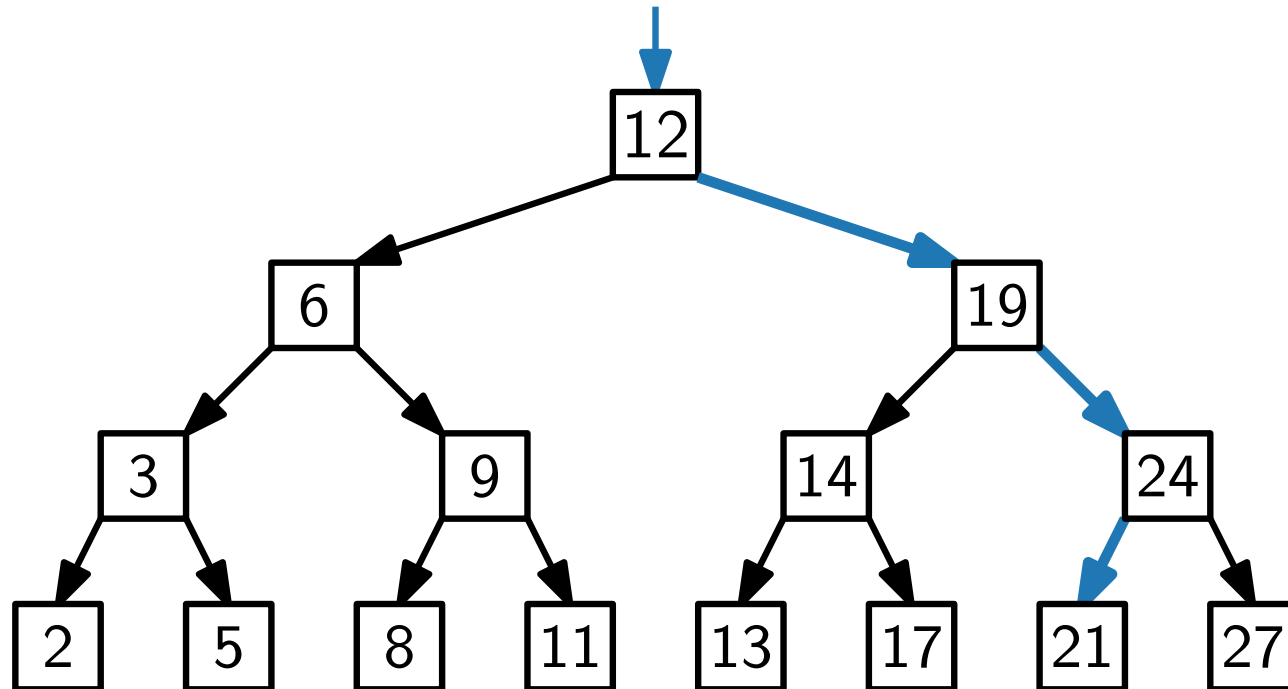
Binärer-Suchbaum-Eigenschaft.

Für jeden Knoten v gilt:
alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$

Suche im sortierten Feld



SEARCH(21)



Jeder Knoten hat höchstens zwei Kinder.

**Binärer
Suchbaum**

zusammenhängender
kreisfreier Graph
(mehr dazu in 2 Wochen)

Binärer-Suchbaum-Eigenschaft.

Für jeden Knoten v gilt:
alle Knoten im linken Teilbaum von v haben Schlüssel $\leq v.key$
rechten $\geq$

Binärer Suchbaum

Operation	Implementierung

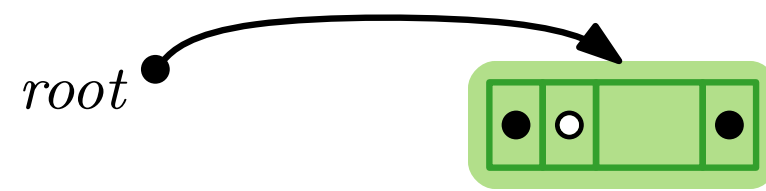
Binärer Suchbaum

Operation

BINSEARCHTREE()

Implementierung

Binärer Suchbaum

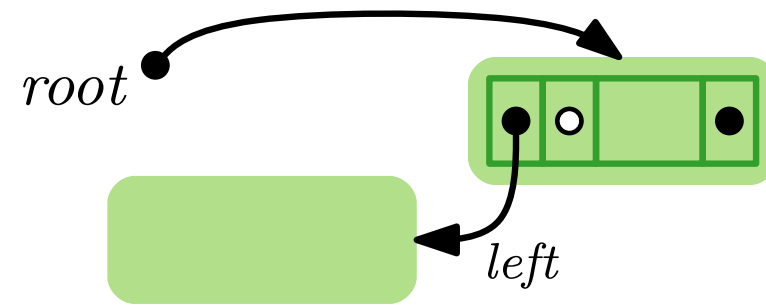


Operation

BINSEARCHTREE()

Implementierung

Binärer Suchbaum

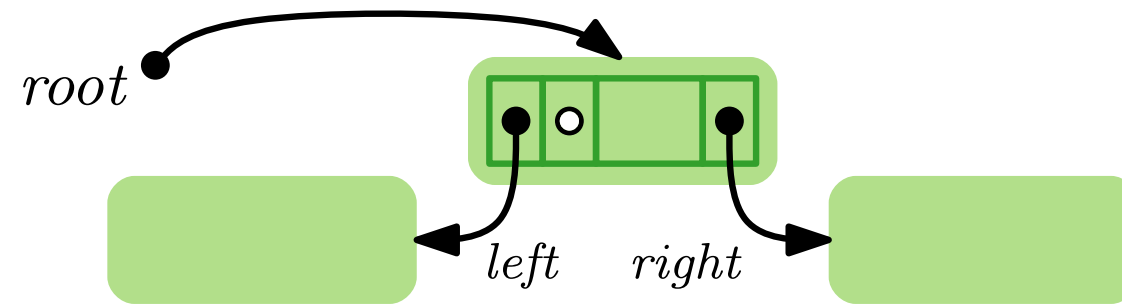


Operation

BINSEARCHTREE()

Implementierung

Binärer Suchbaum

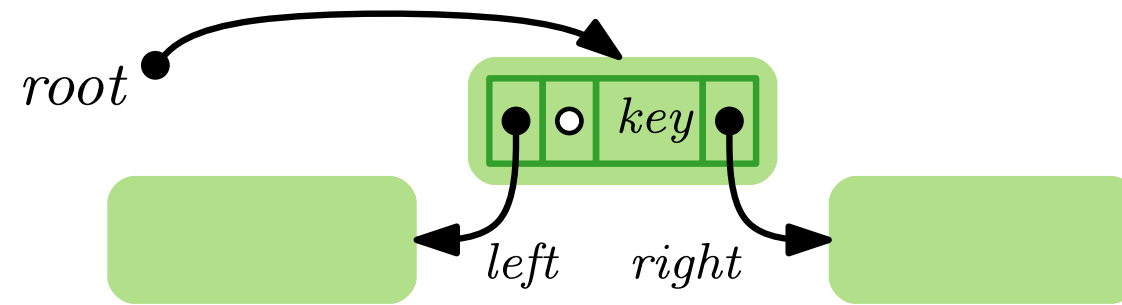


Operation

BINSEARCHTREE()

Implementierung

Binärer Suchbaum

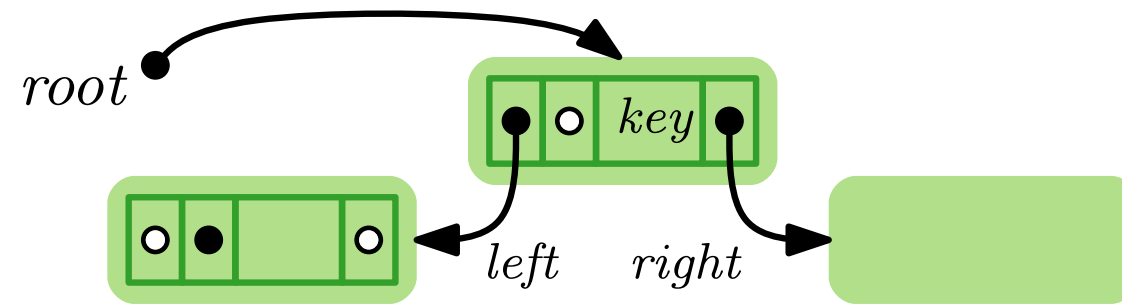


Operation

BINSEARCHTREE()

Implementierung

Binärer Suchbaum

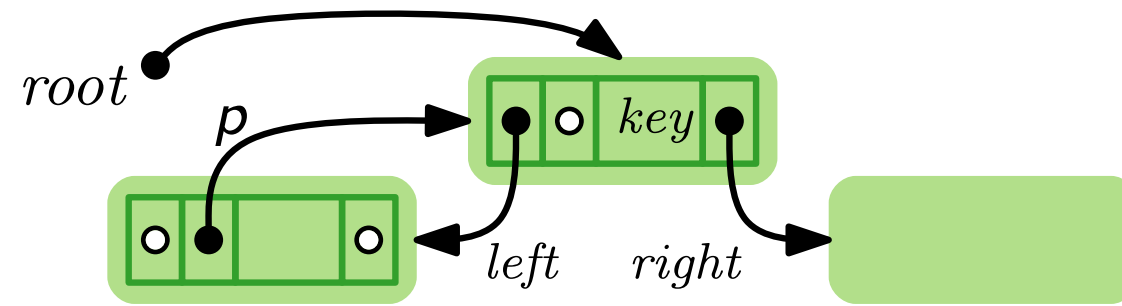


Operation

BINSEARCHTREE()

Implementierung

Binärer Suchbaum

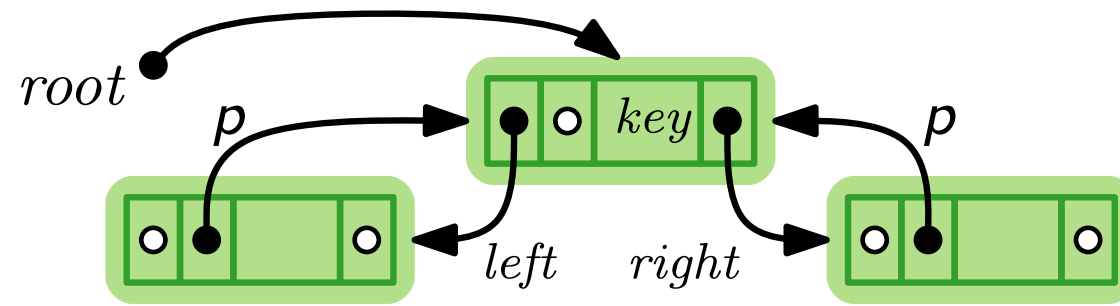


Operation

BINSEARCHTREE()

Implementierung

Binärer Suchbaum

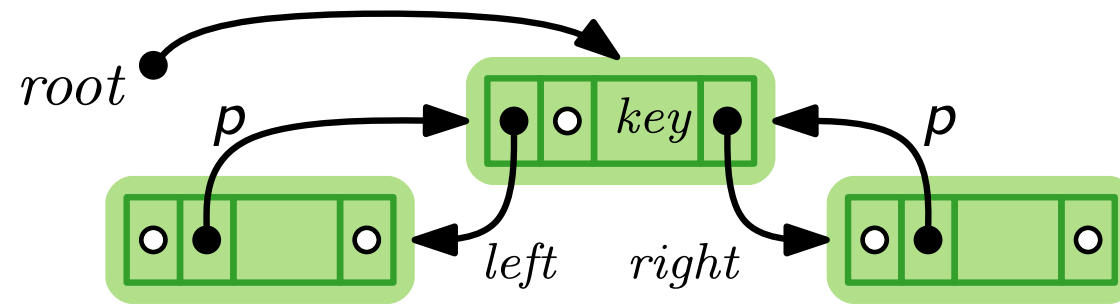


Operation

BINSEARCHTREE()

Implementierung

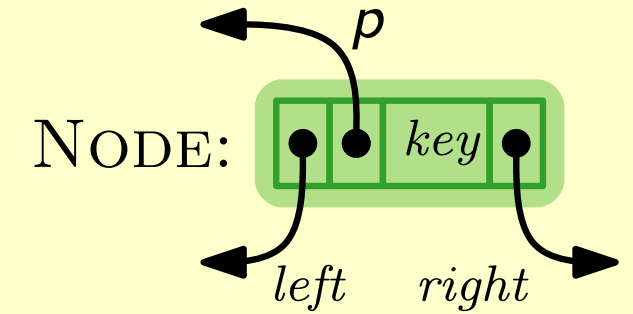
Binärer Suchbaum



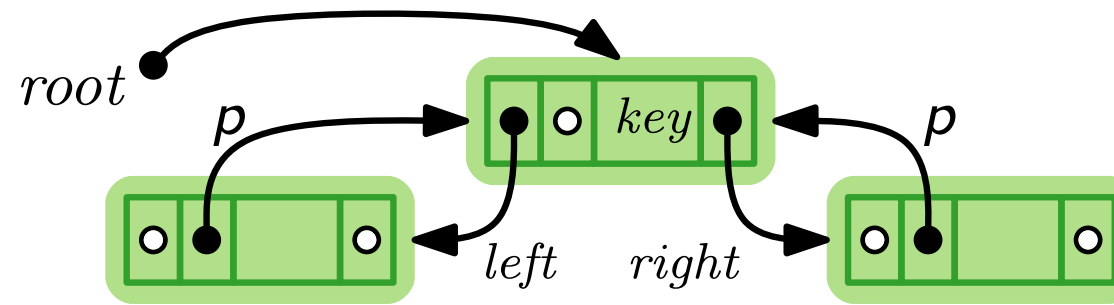
Operation

BINSEARCHTREE()

Implementierung



Binärer Suchbaum



Operation

BINSEARCHTREE()

Implementierung

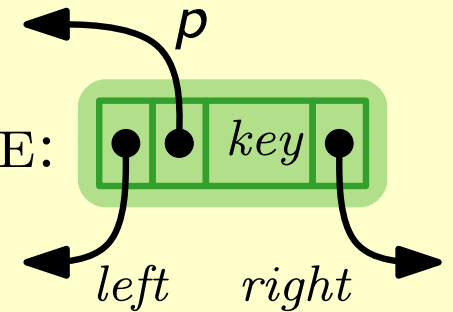
NODE(key k , NODE par)

$key = k$

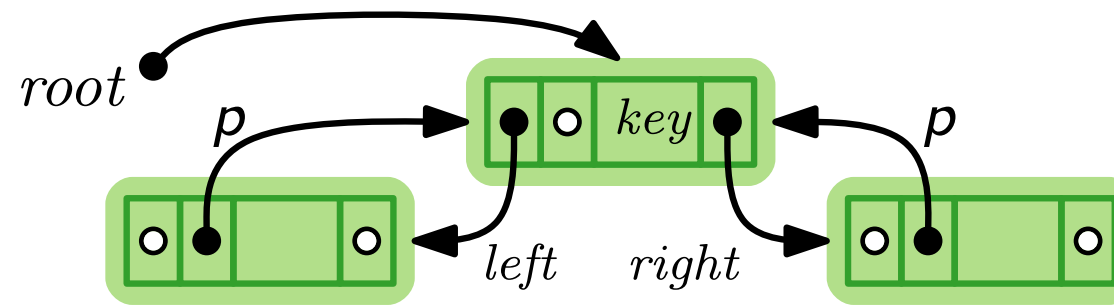
$p = par$

$right = left = nil$

NODE:



Binärer Suchbaum



Operation

BINSEARCHTREE()

Implementierung

$root = nil$

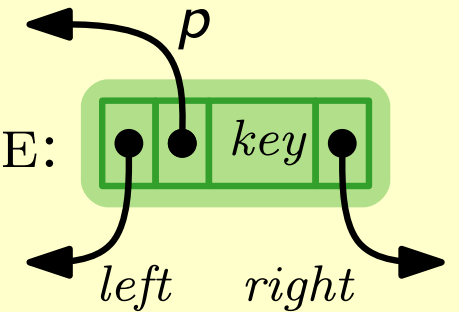
NODE(key k , NODE par)

$key = k$

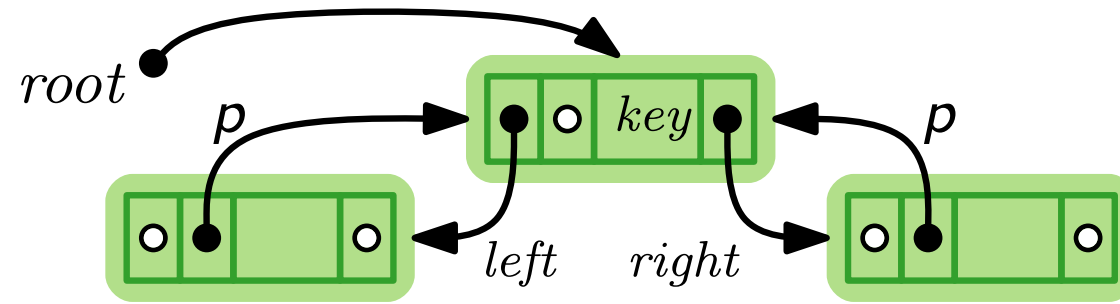
$p = par$

$right = left = nil$

NODE:



Binärer Suchbaum



Operation

Implementierung

BINSEARCHTREE()

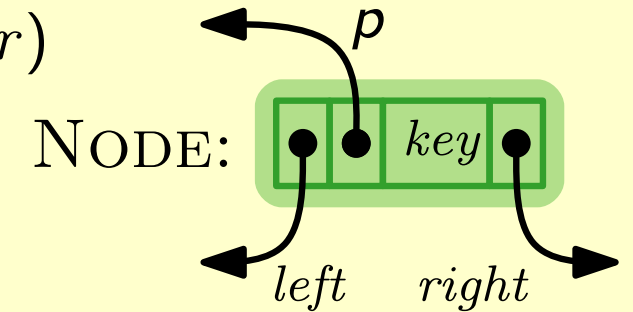
$root = nil$

NODE(key k , NODE par)

$key = k$

$p = par$

$right = left = nil$



Node SEARCH(key k)

Node INSERT(key k)

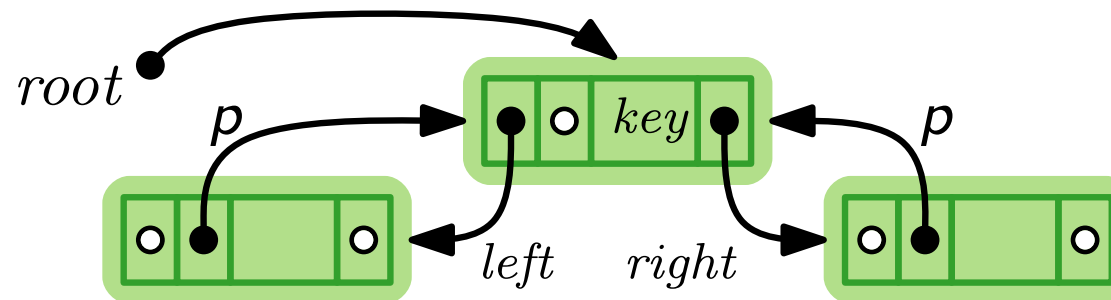
DELETE(Node x)

Node MINIMUM()

Node MAXIMUM()

Node PREDECESSOR(Node x)

Node SUCCESSOR(Node x)



Operation

Implementierung

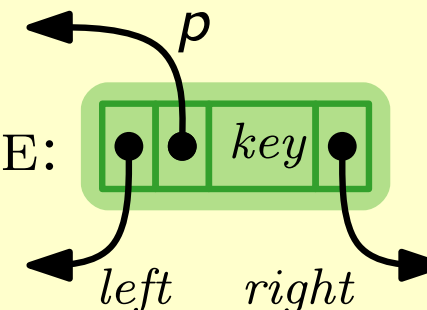
BINSEARCHTREE()

$$root = nil$$

NODE(key k , NODE par)

$$key = k$$
$$p = par$$
$$right = left = nil$$

NODE:



Node SEARCH(key k)

Node INSERT(key k)

DELETE(Node x)

Node MINIMUM()

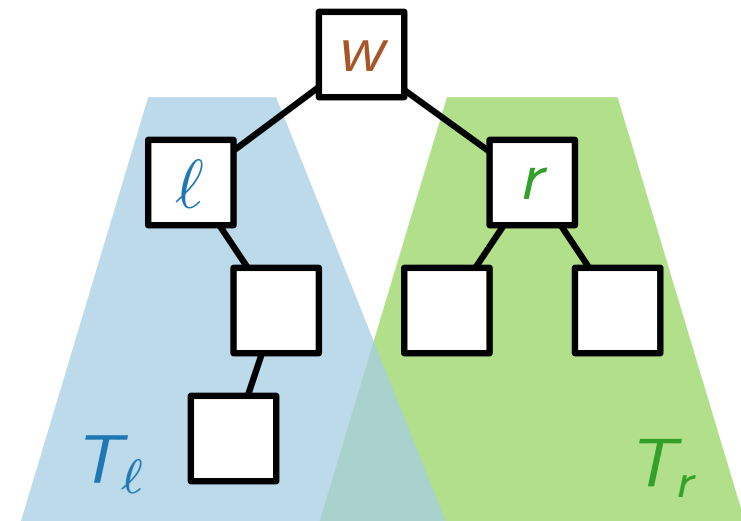
Node MAXIMUM()

Node PREDECESSOR(Node x)Node SUCCESSOR(Node x)

TODO

Inorder-Traversierung

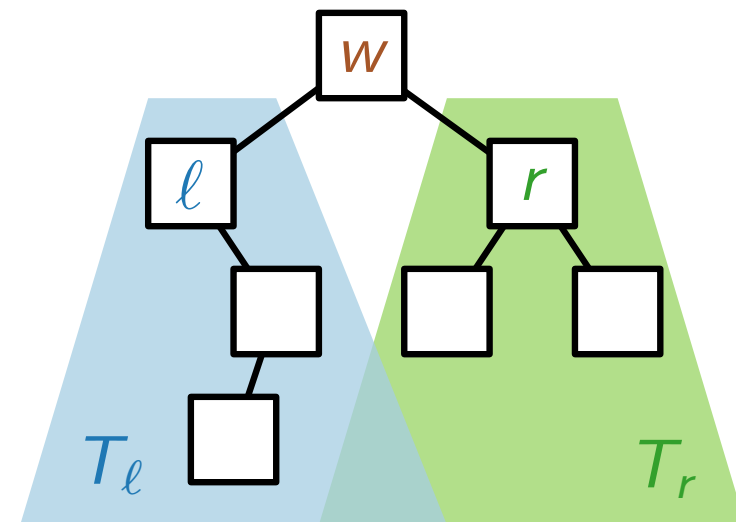
(Binäre) Bäume haben eine zur Rekursion einladende Struktur...



Inorder-Traversierung

(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

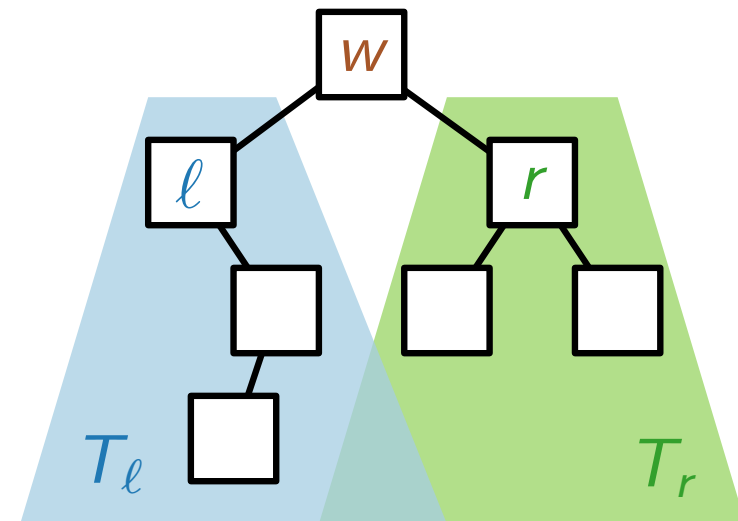


Inorder-Traversierung

(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

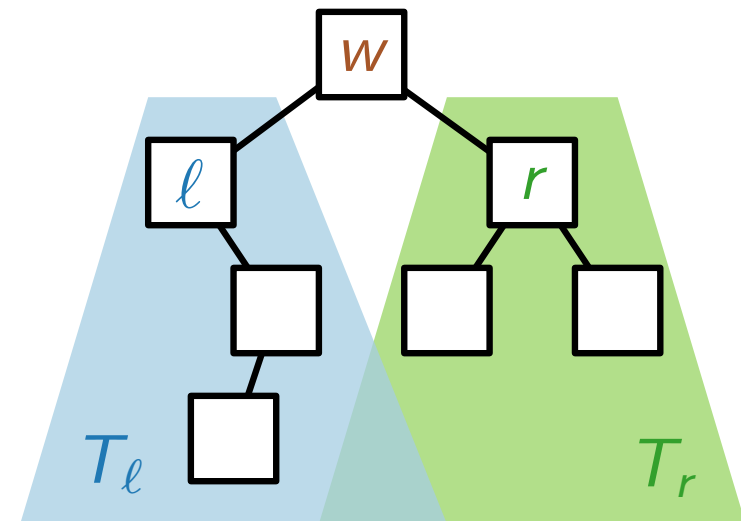


Inorder-Traversierung

(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung: 1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.



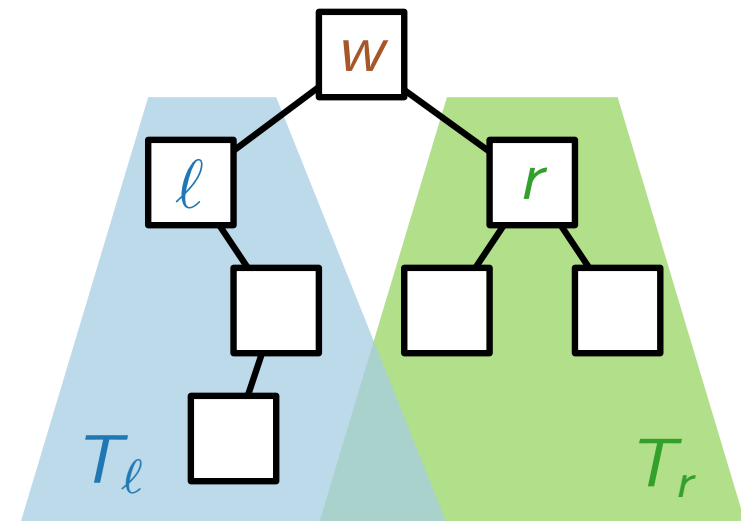
Inorder-Traversierung

(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.



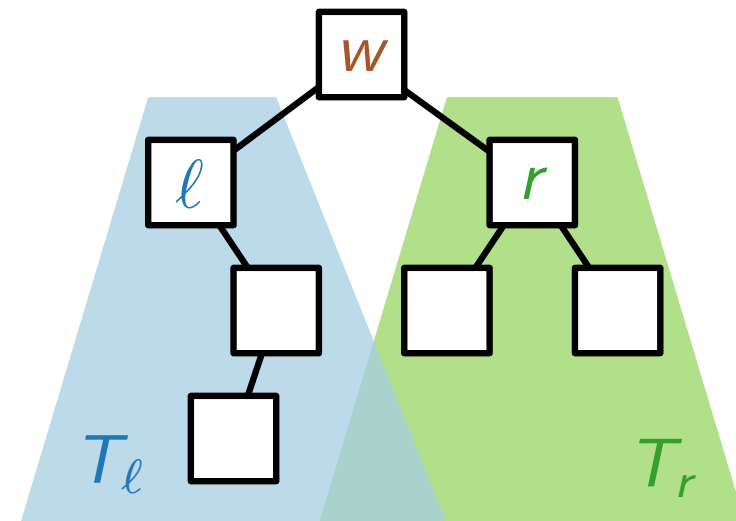
Inorder-Traversierung

(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.
3. Durchlaufe rekursiv **rechten Teilbaum** der Wurzel.



Inorder-Traversierung

(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

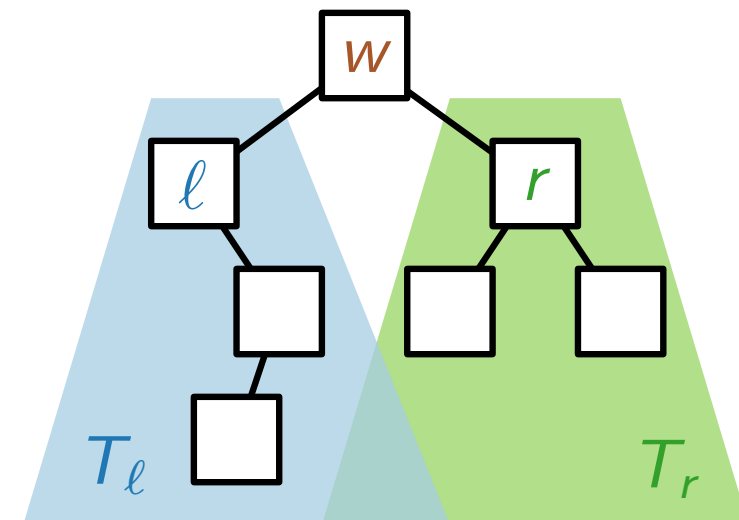
Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

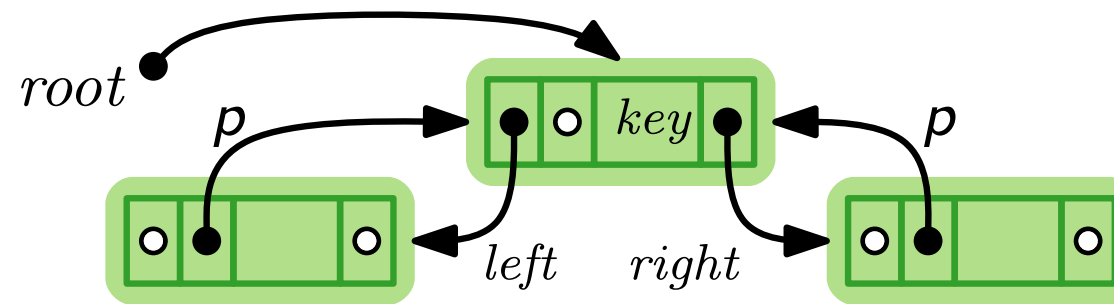
1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.
3. Durchlaufe rekursiv **rechten Teilbaum** der Wurzel.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
```



Inorder-Traversierung



(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

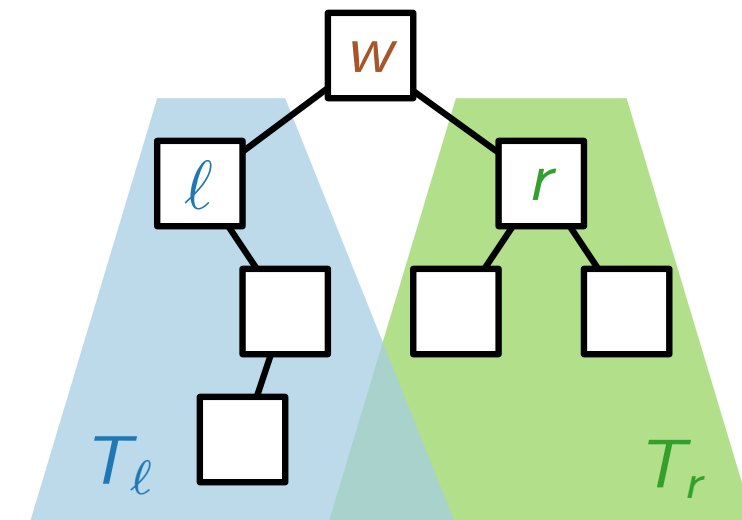
Lösung:

1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.
3. Durchlaufe rekursiv **rechten Teilbaum** der Wurzel.

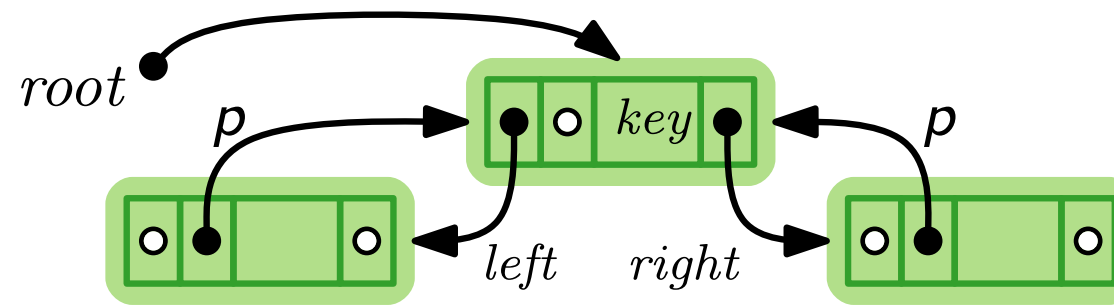
Code: INORDERTREEWALK(Node $x = \text{root}$)

Aufgabe.

Geben Sie eine rekursive Implementierung an!



Inorder-Traversierung



(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

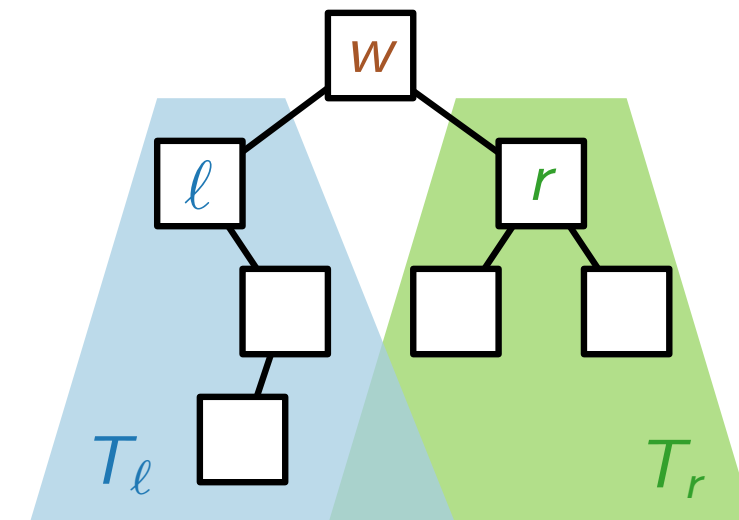
Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

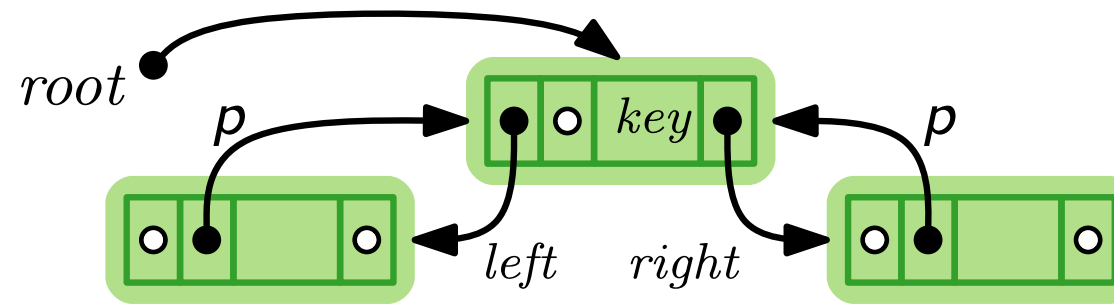
1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.
3. Durchlaufe rekursiv **rechten Teilbaum** der Wurzel.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    |
    |
    |
```



Inorder-Traversierung



(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

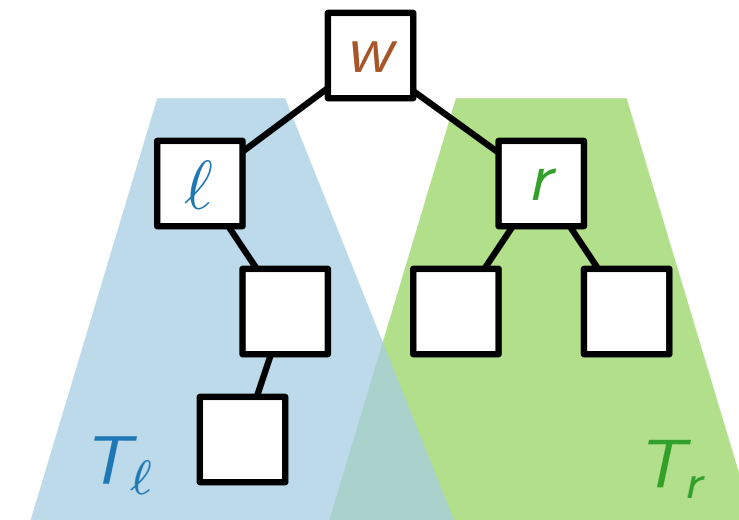
Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

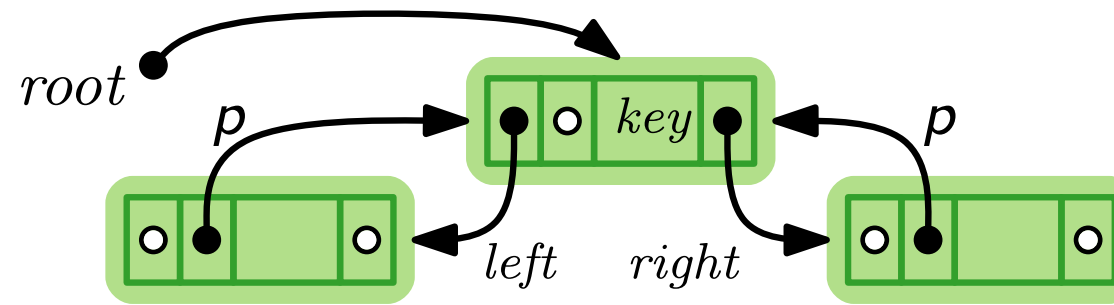
1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.
3. Durchlaufe rekursiv **rechten Teilbaum** der Wurzel.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
     $\vdash$ 
```



Inorder-Traversierung



(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

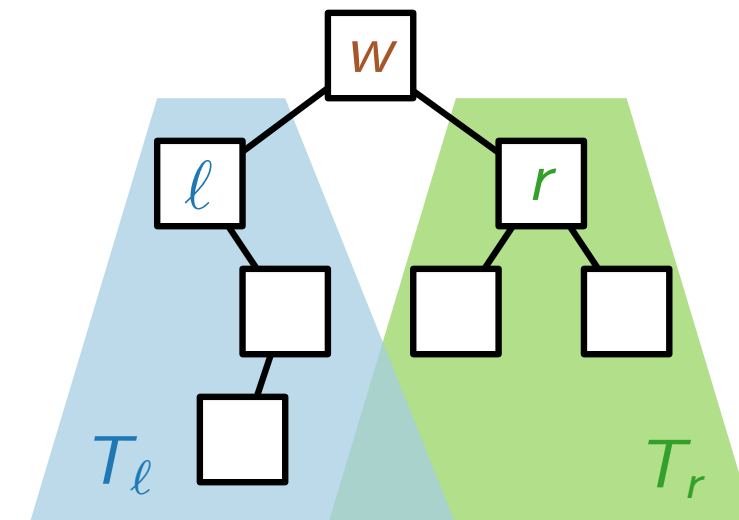
Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

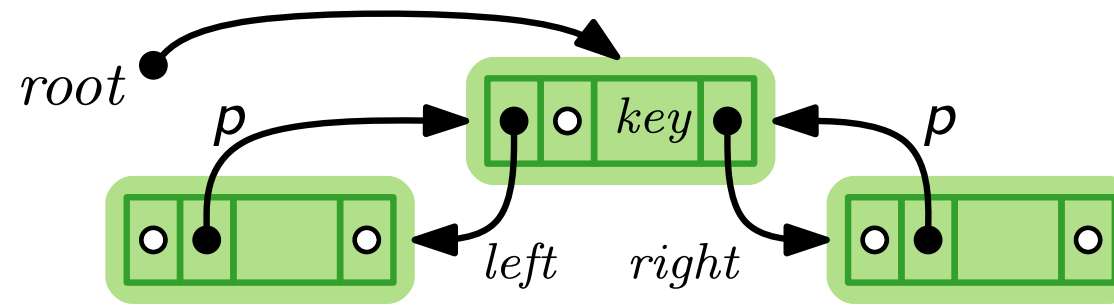
1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.
3. Durchlaufe rekursiv **rechten Teilbaum** der Wurzel.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
```



Inorder-Traversierung



(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

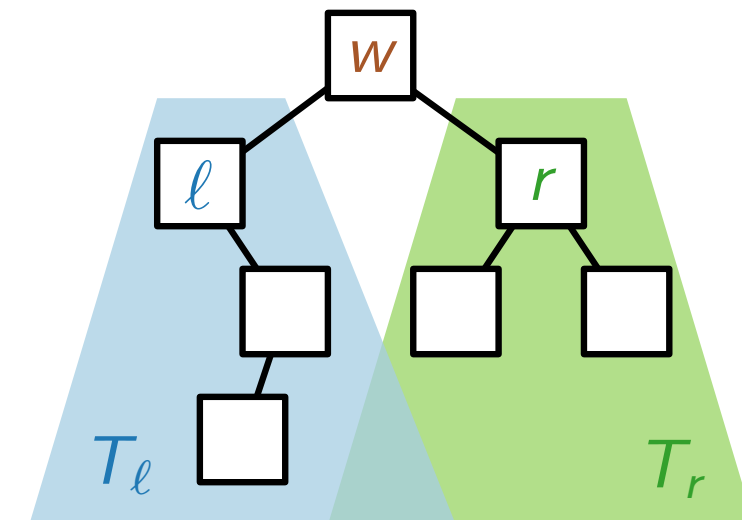
Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

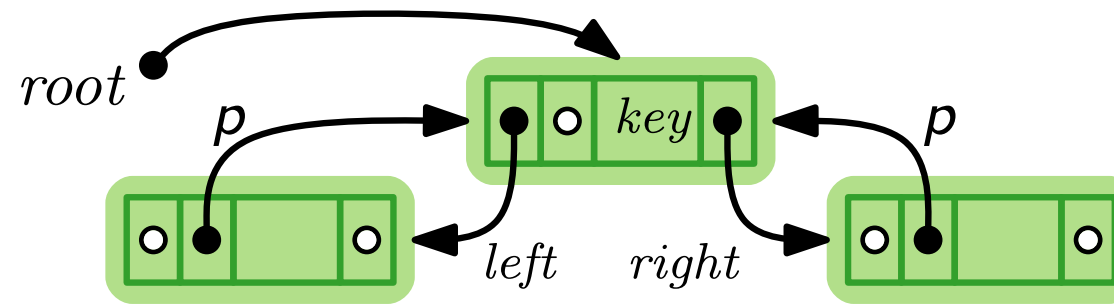
1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.
3. Durchlaufe rekursiv **rechten Teilbaum** der Wurzel.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```



Inorder-Traversierung



(Binäre) Bäume haben eine zur Rekursion einladende Struktur...

Beispiel: Gib Schlüssel eines binären Suchbaums *sortiert* aus!

Lösung:

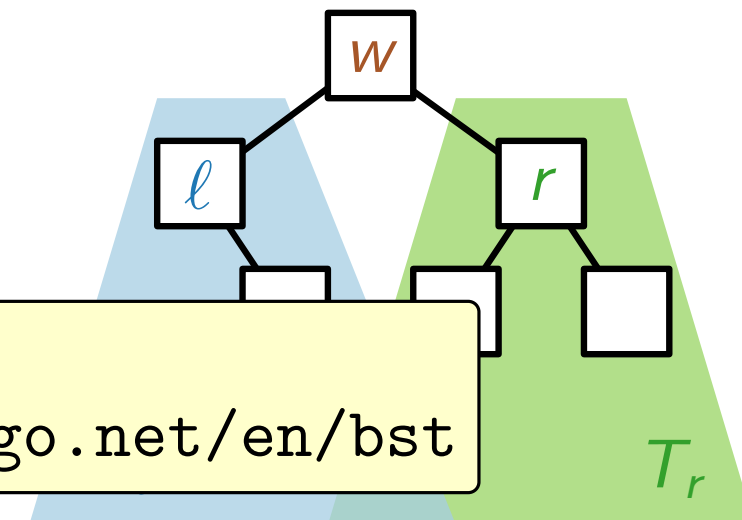
1. Durchlaufe rekursiv **linken Teilbaum** der Wurzel.
2. Gib den Schlüssel der **Wurzel** aus.
3. Durchlaufe rekursiv **rechten Teilbaum** der Wurzel.

Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ nil then
    INORDERTREEWALK(x.left)
    gib x.key aus
    INORDERTREEWALK(x.right)
```

Demo.

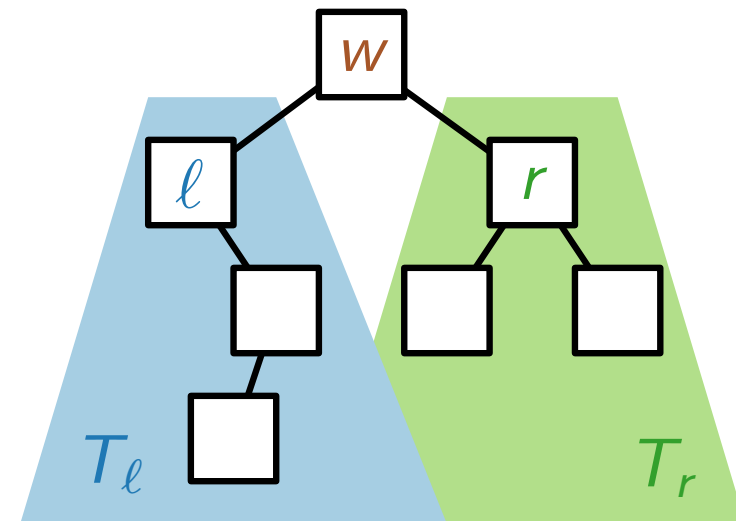
<https://visualgo.net/en/bst>



Korrektheit

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )  
  if  $x \neq \text{nil}$  then  
    INORDERTREEWALK( $x.\text{left}$ )  
    gib  $x.\text{key}$  aus  
    INORDERTREEWALK( $x.\text{right}$ )
```

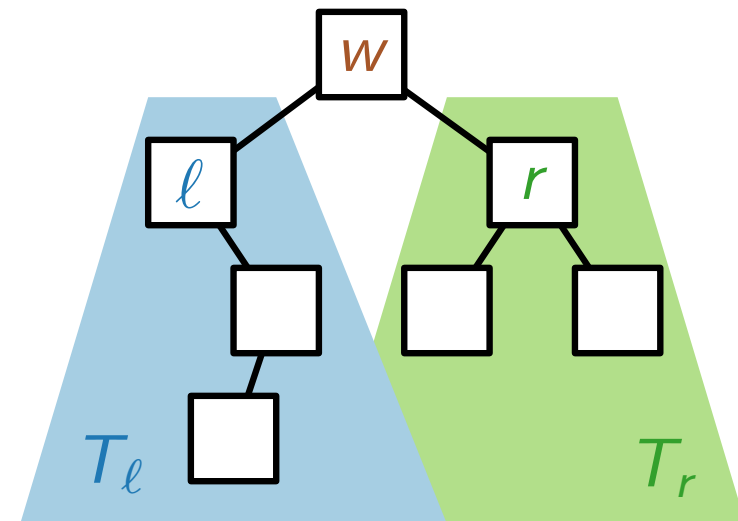


Korrektheit

Zu zeigen:

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )  
  if  $x \neq \text{nil}$  then  
    INORDERTREEWALK( $x.\text{left}$ )  
    gib  $x.\text{key}$  aus  
    INORDERTREEWALK( $x.\text{right}$ )
```

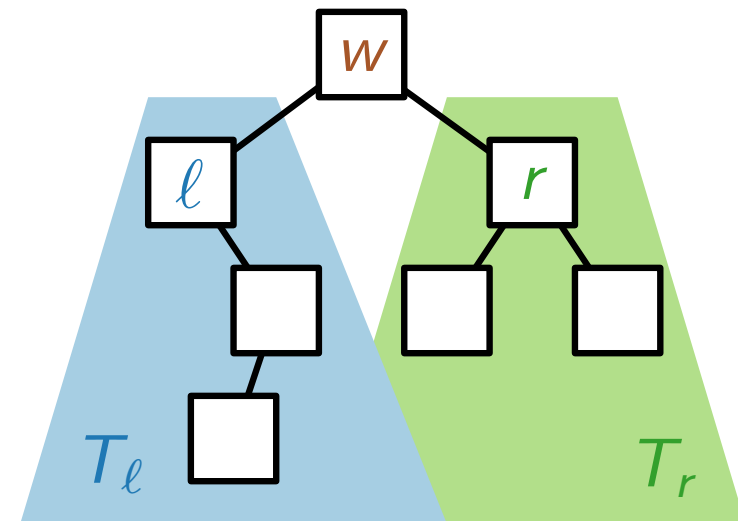


Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )  
  if  $x \neq \text{nil}$  then  
    INORDERTREEWALK( $x.\text{left}$ )  
    gib  $x.\text{key}$  aus  
    INORDERTREEWALK( $x.\text{right}$ )
```

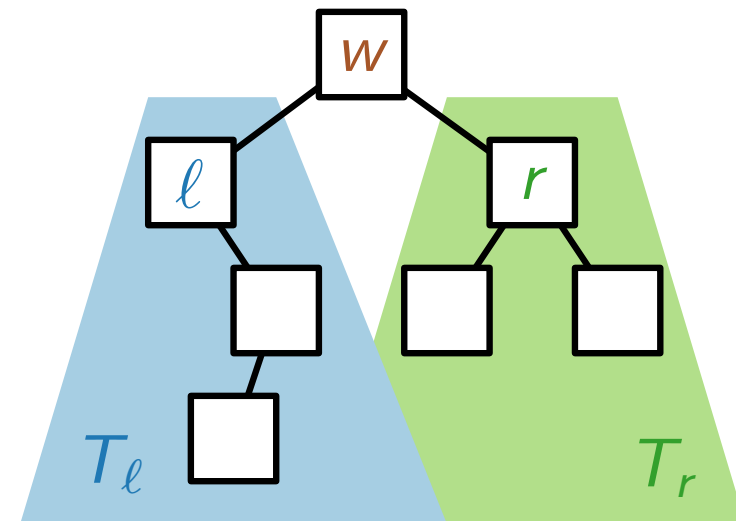


Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.
Induktion über die Baumhöhe h .

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )  
  if  $x \neq \text{nil}$  then  
    INORDERTREEWALK( $x.\text{left}$ )  
    gib  $x.\text{key}$  aus  
    INORDERTREEWALK( $x.\text{right}$ )
```



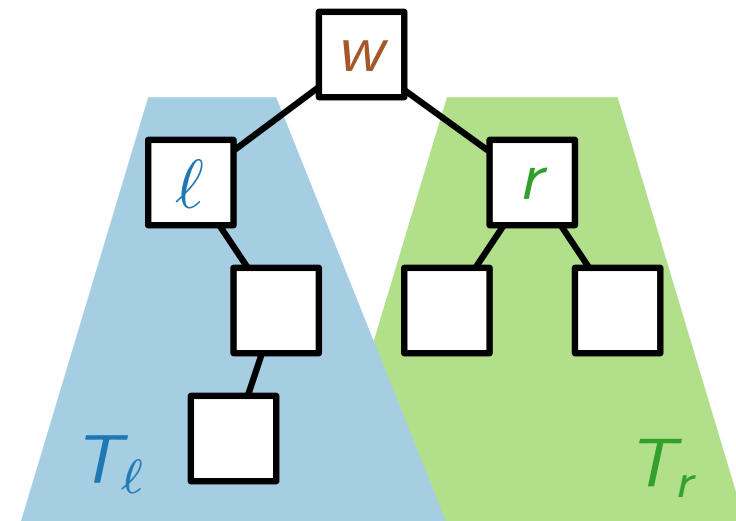
Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.
Induktion über die Baumhöhe h .

$h = -1$:

Code:

```
INORDERTREEWALK(Node  $x = root$ )  
  if  $x \neq nil$  then  
    INORDERTREEWALK( $x.left$ )  
    gib  $x.key$  aus  
    INORDERTREEWALK( $x.right$ )
```



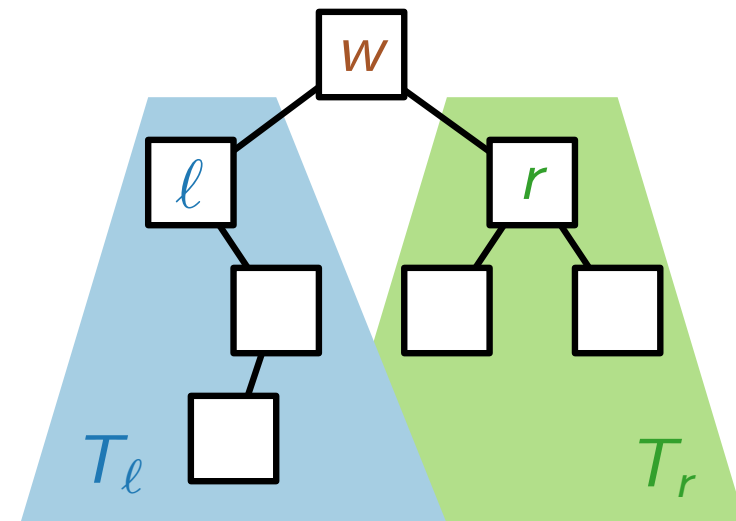
Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.
Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$

Code:

```
INORDERTREEWALK(Node  $x = root$ )  
  if  $x \neq nil$  then  
    INORDERTREEWALK( $x.left$ )  
    gib  $x.key$  aus  
    INORDERTREEWALK( $x.right$ )
```



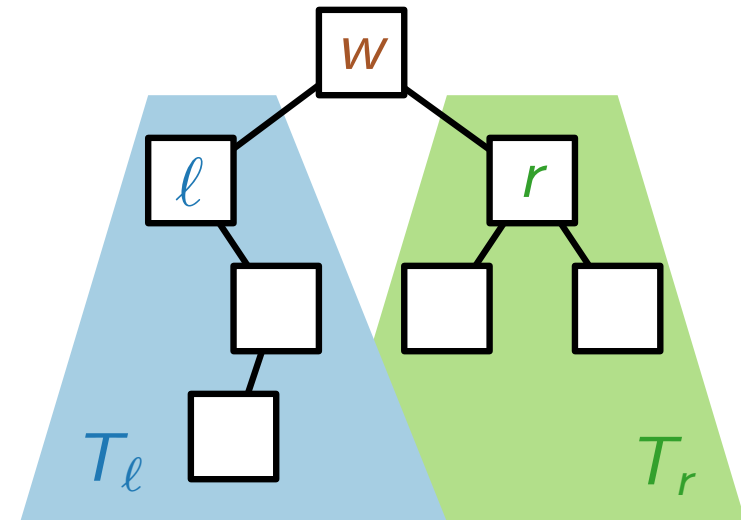
Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.
Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$ ✓

Code:

```
INORDERTREEWALK(Node  $x = root$ )  
  if  $x \neq nil$  then  
    INORDERTREEWALK( $x.left$ )  
    gib  $x.key$  aus  
    INORDERTREEWALK( $x.right$ )
```



Korrektheit

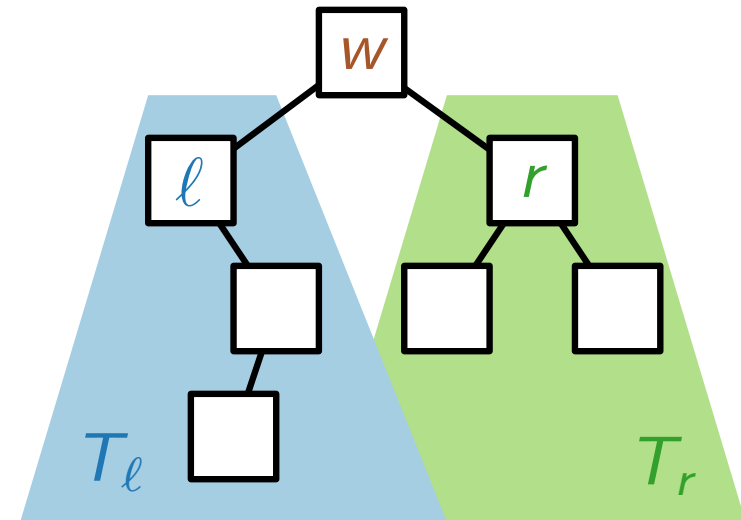
Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.
Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$ ✓

$h \geq 0$:

Code:

```
INORDERTREEWALK(Node  $x = root$ )  
  if  $x \neq nil$  then  
    INORDERTREEWALK( $x.left$ )  
    gib  $x.key$  aus  
    INORDERTREEWALK( $x.right$ )
```



Korrektheit

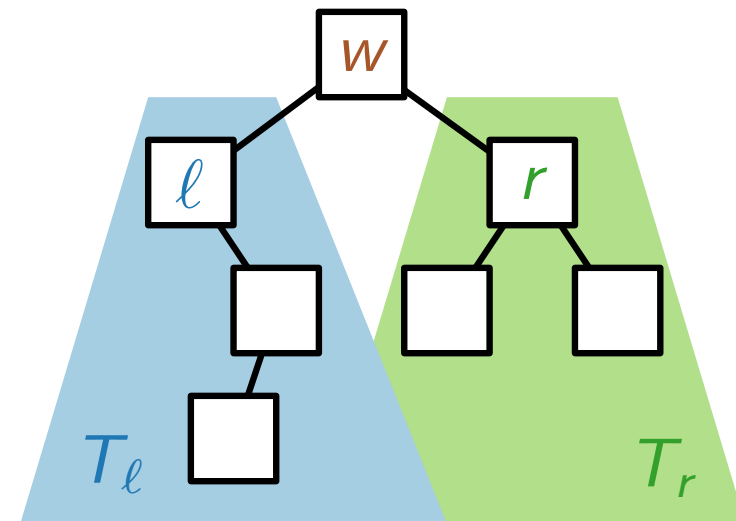
Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.
Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$ ✓

$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

Code:

```
INORDERTREEWALK(Node  $x = root$ )  
  if  $x \neq nil$  then  
    INORDERTREEWALK( $x.left$ )  
    gib  $x.key$  aus  
    INORDERTREEWALK( $x.right$ )
```



Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Induktion über die Baumhöhe h .

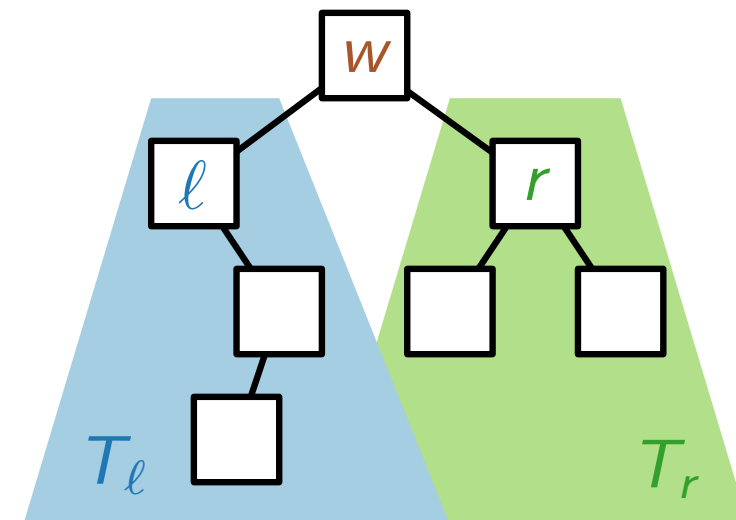
$h = -1$: Baum leer, d.h. $root = nil$ ✓

$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} linker und rechter Teilbaum der Wurzel.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```



Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$ ✓

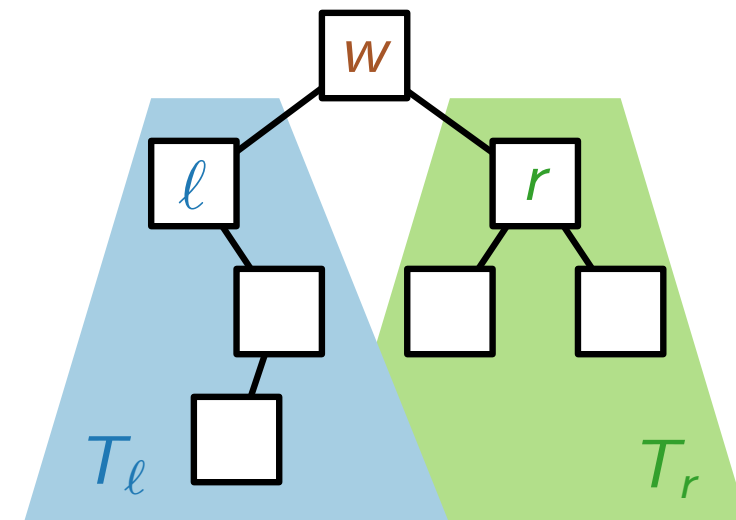
$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} linker und rechter Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```



Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$ ✓

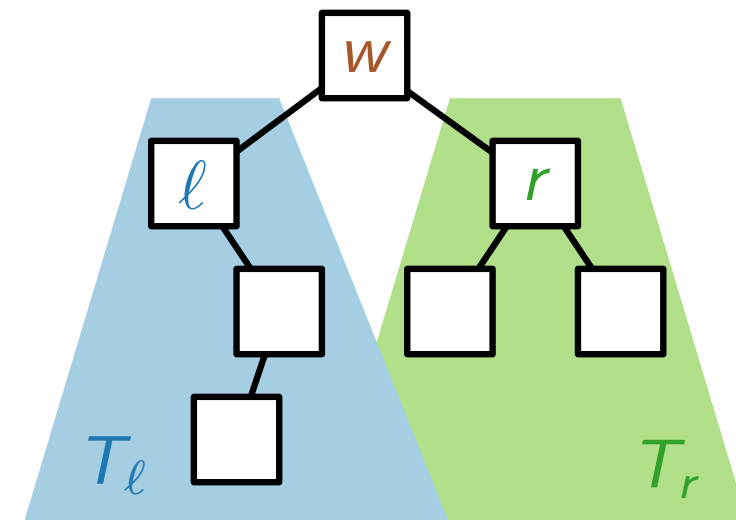
$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} linker und rechter Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. rekursive Def. der Höhe

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```



Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$ ✓

$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

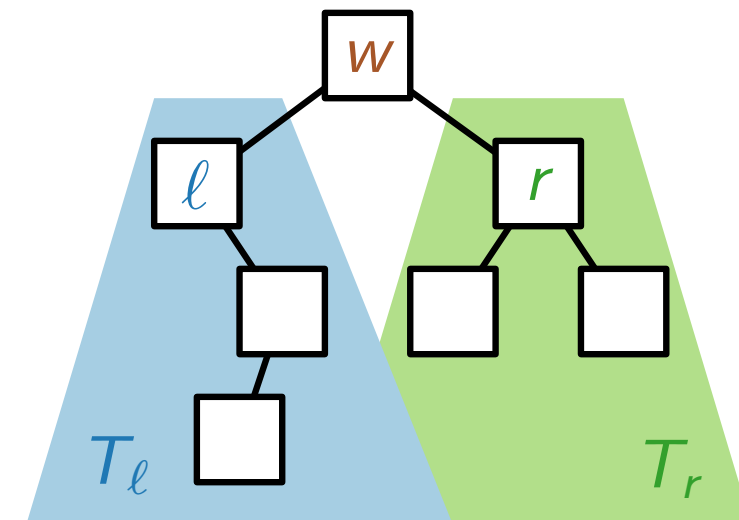
Seien T_{links} und T_{rechts} linker und rechter Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. rekursive Def. der Höhe

Also werden *ihre* Schlüssel sortiert ausgegeben.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```



Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$ ✓

$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} linker und rechter Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. rekursive Def. der Höhe

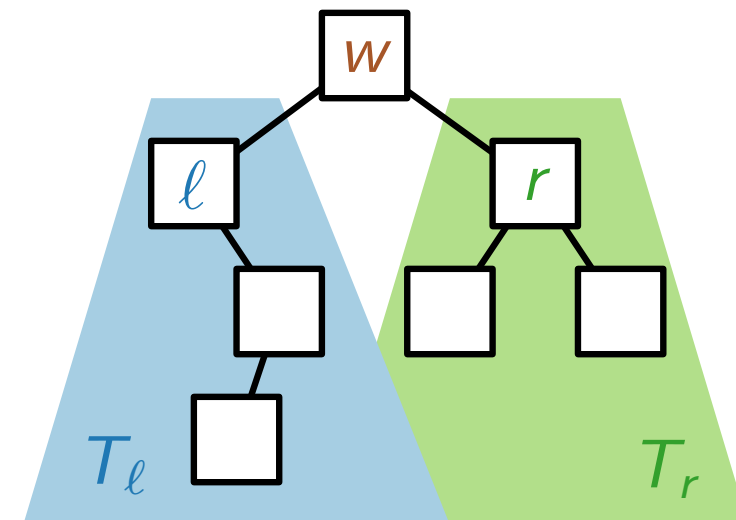
Also werden *ihre* Schlüssel sortiert ausgegeben.

$\Rightarrow$

Ausgabe (sortierte Schlüssel von T_{links} , dann $root.key$, dann sortierte Schlüssel von T_{rechts}) ist sortiert.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```



Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$ ✓

$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} linker und rechter Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. rekursive Def. der Höhe

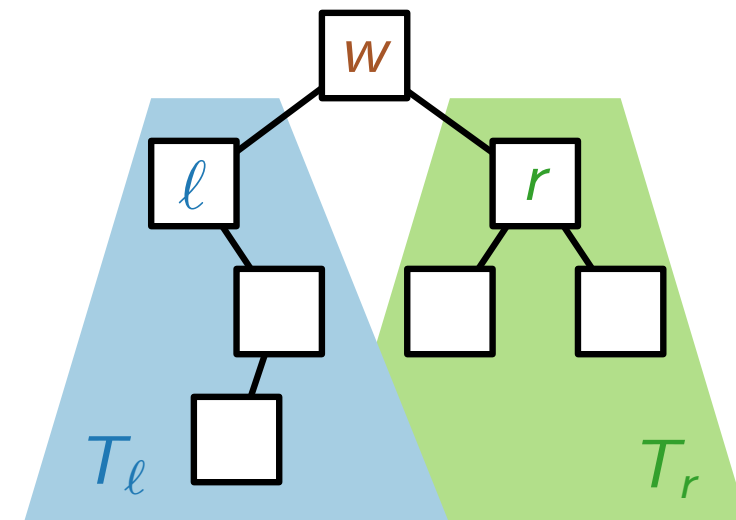
Also werden *ihre* Schlüssel sortiert ausgegeben.

Binärer-Suchbaum-Eigenschaft $\Rightarrow$

Ausgabe (sortierte Schlüssel von T_{links} , dann $root.key$, dann sortierte Schlüssel von T_{rechts}) ist sortiert.

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```



Korrektheit

Zu zeigen: Schlüssel werden in sortierter Reihenfolge ausgegeben.

Induktion über die Baumhöhe h .

$h = -1$: Baum leer, d.h. $root = nil$ ✓

$h \geq 0$: Induktionshypothese sei wahr für Bäume der Höhe $< h$.

Seien T_{links} und T_{rechts} linker und rechter Teilbaum der Wurzel.

T_{links} und T_{rechts} haben Höhe $< h$. rekursive Def. der Höhe

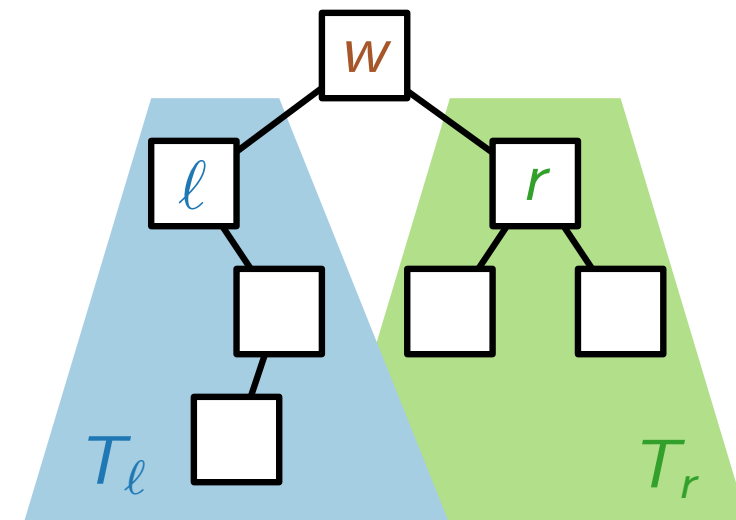
Also werden *ihre* Schlüssel sortiert ausgegeben.

Binärer-Suchbaum-Eigenschaft $\Rightarrow$

Ausgabe (sortierte Schlüssel von T_{links} , dann $root.key$, dann sortierte Schlüssel von T_{rechts}) ist sortiert. ✓

Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```

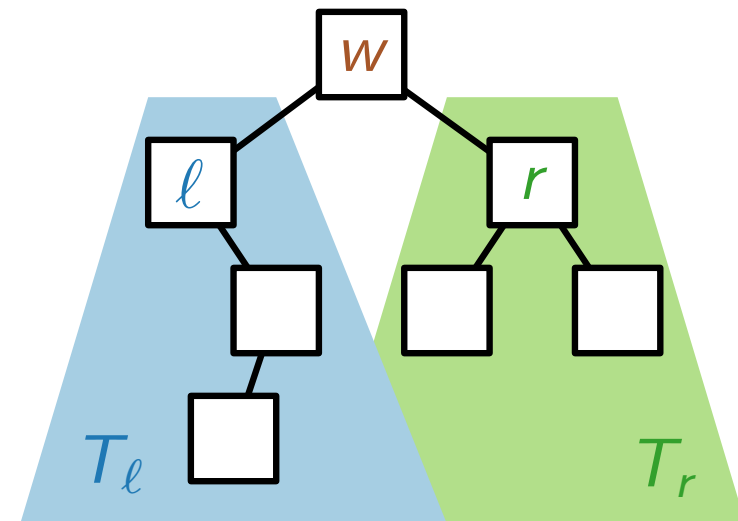


Laufzeit

$$T(n) =$$

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )  
  if  $x \neq \text{nil}$  then  
    INORDERTREEWALK( $x.\text{left}$ )  
    gib  $x.\text{key}$  aus  
    INORDERTREEWALK( $x.\text{right}$ )
```



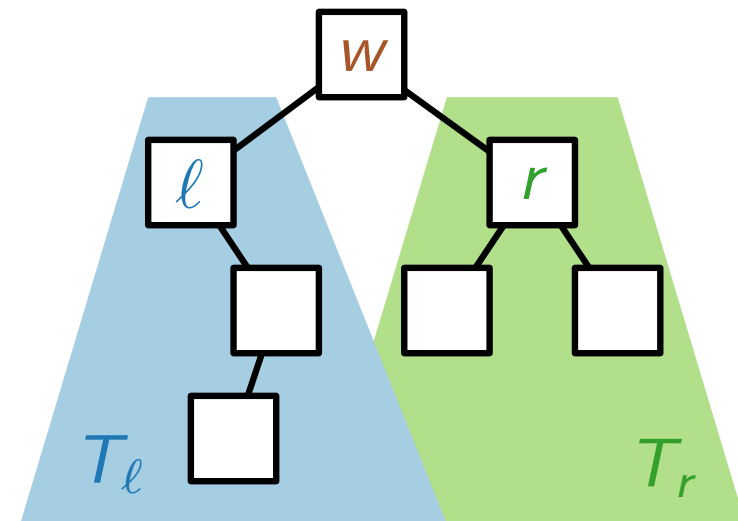
Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(\text{blue box}) + T(\text{green box}) + 1 & \text{sonst.} \end{cases}$$

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



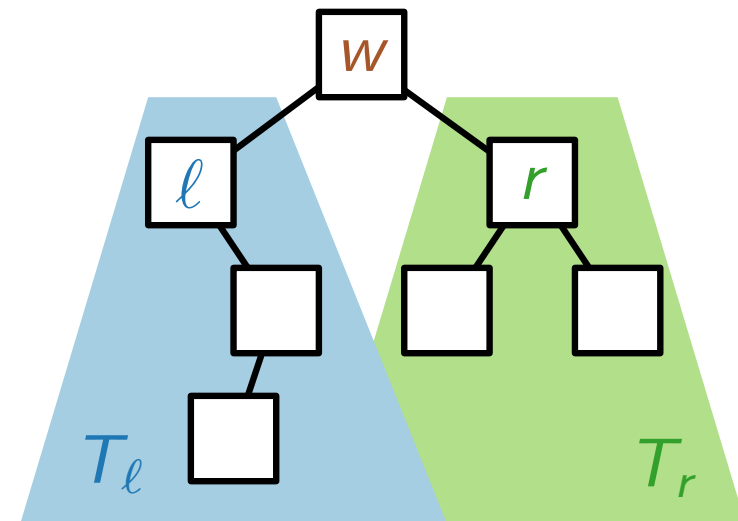
Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

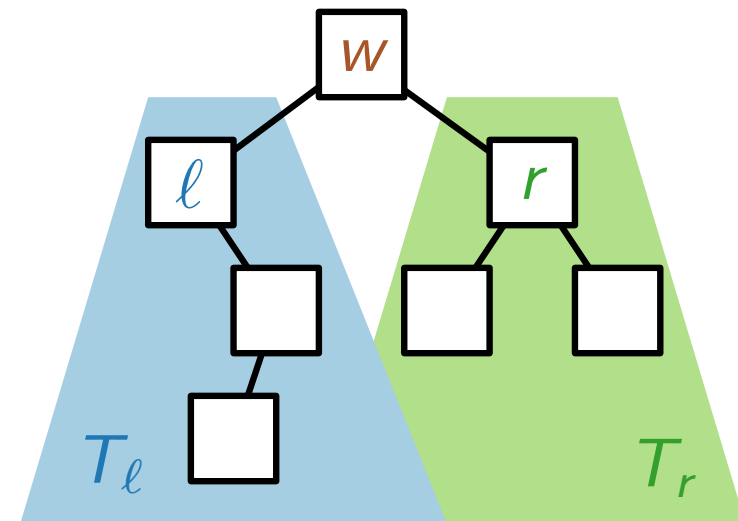
Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n$

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

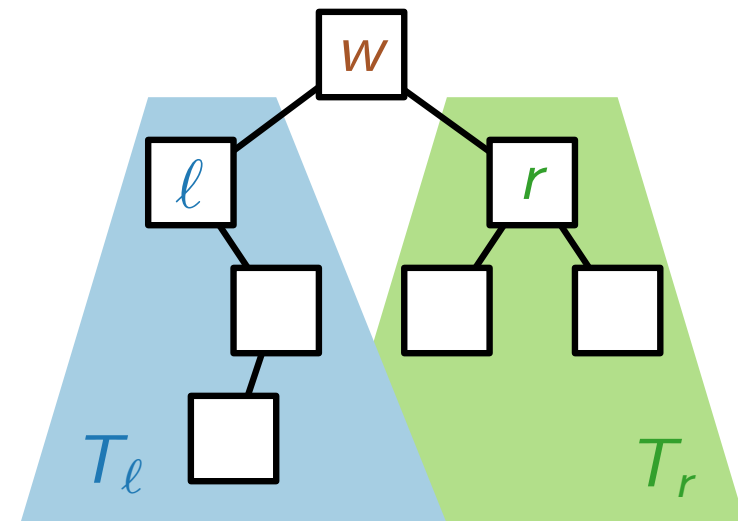
Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

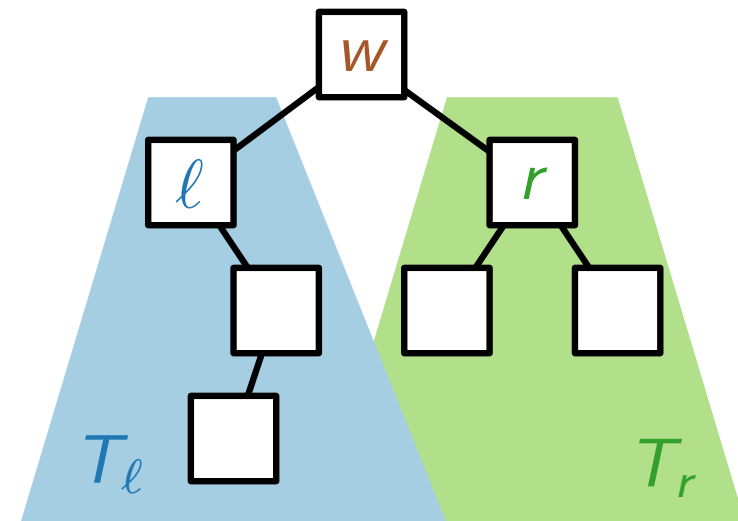
$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder:

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

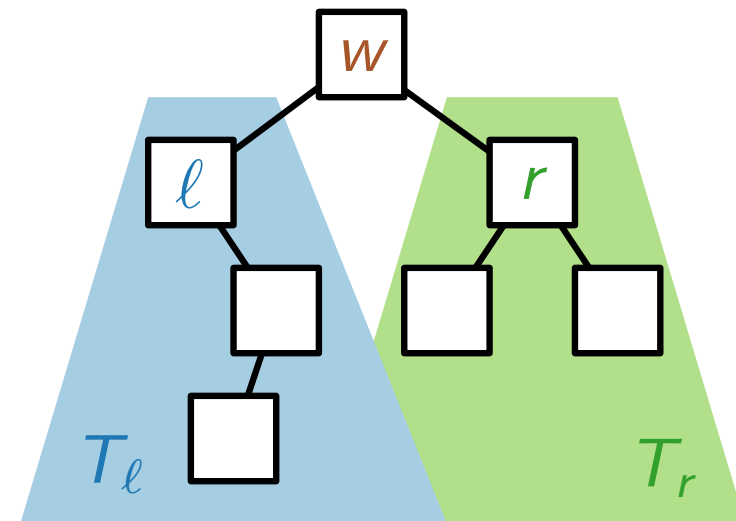
$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

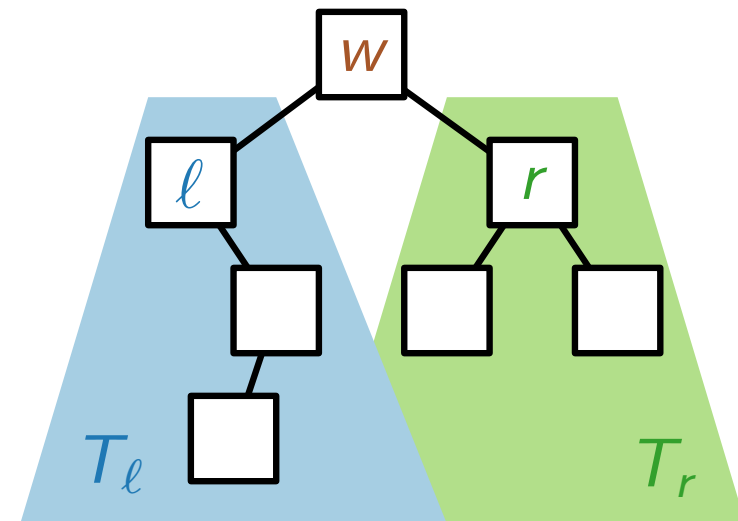
Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

Für Bäume gilt: #Kanten =

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

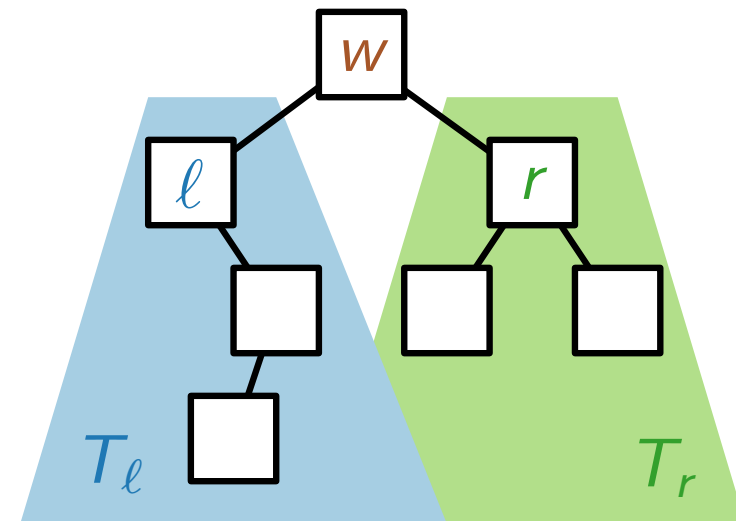
Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

Für Bäume gilt: $\# \text{Kanten} = \# \text{Knoten} - 1 = n - 1$

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

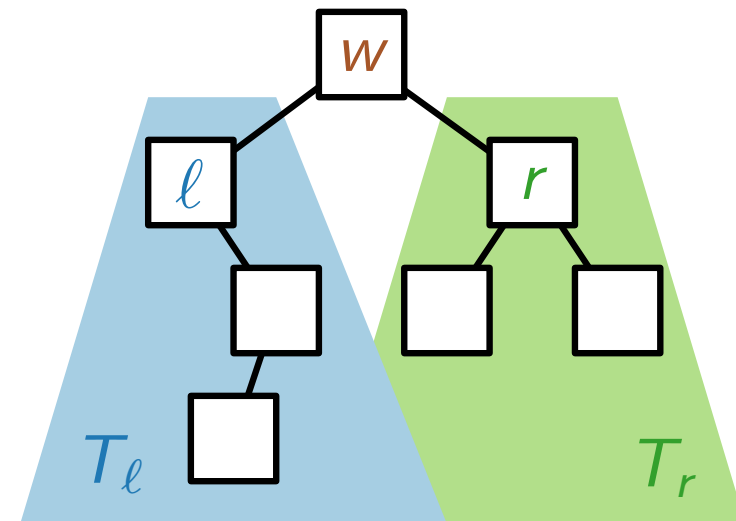
oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

Für Bäume gilt: $\# \text{Kanten} = \# \text{Knoten} - 1 = n - 1$

Beweis durch
Induktion

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

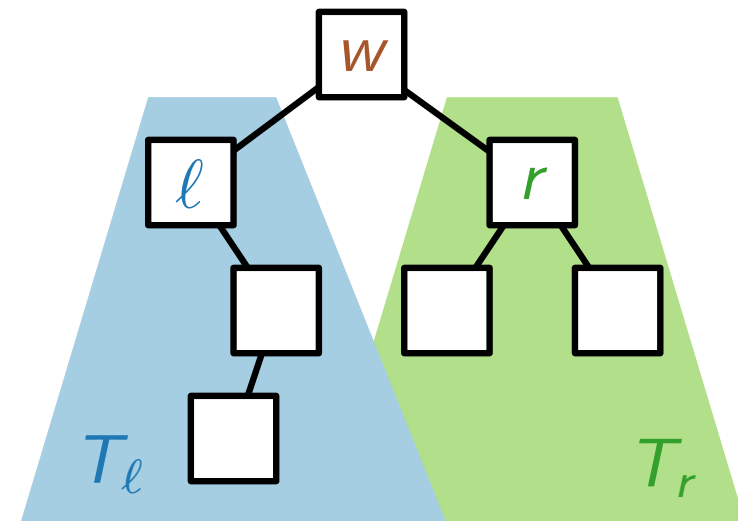
Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

Für Bäume gilt: $\# \text{Kanten} = \# \text{Knoten} - 1 = n - 1$

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

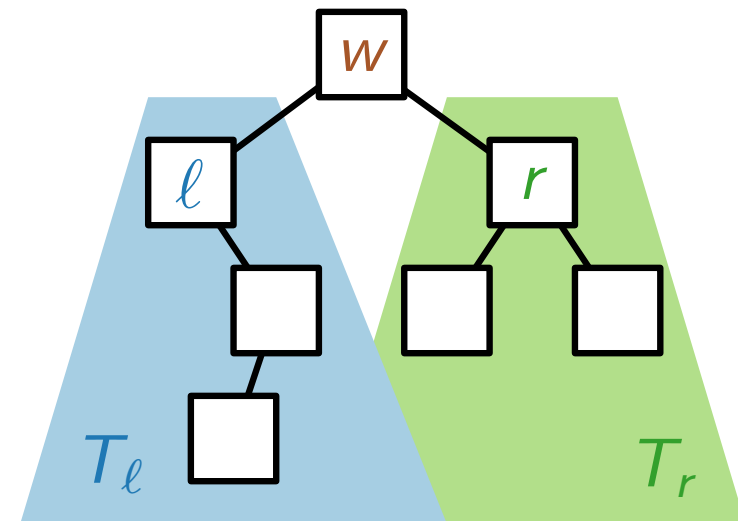
Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

Für Bäume gilt: $\# \text{Kanten} = \# \text{Knoten} - 1 = n - 1$

Code:

```
INORDERTREEWALK(Node  $x = \text{root}$ )
  if  $x \neq \text{nil}$  then
    INORDERTREEWALK( $x.\text{left}$ )
    gib  $x.\text{key}$  aus
    INORDERTREEWALK( $x.\text{right}$ )
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

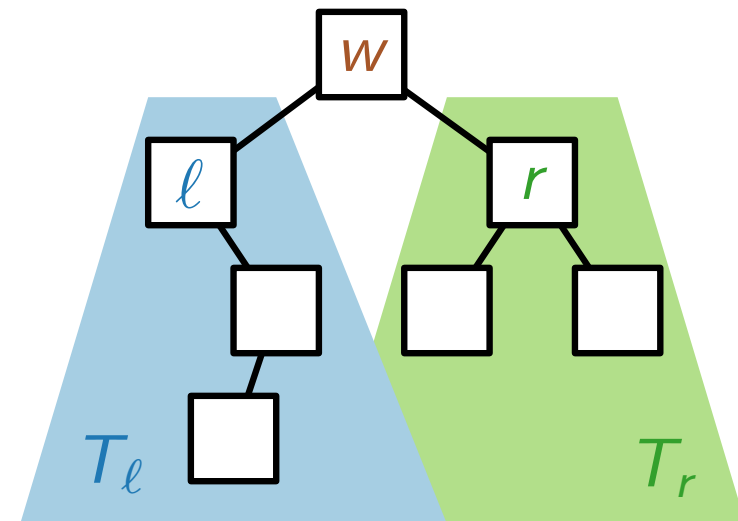
oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

Für Bäume gilt: $\# \text{Kanten} = \# \text{Knoten} - 1 = n - 1$

$$\Rightarrow T(n) = c_1 \cdot (n - 1) + c_2 \cdot n$$

Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ nil then
    INORDERTREEWALK(x.left)
    gib x.key aus
    INORDERTREEWALK(x.right)
```



Laufzeit

Anz. der Knoten im linken / rechten Teilbaum der Wurzel

$$T(n) = \begin{cases} 1 & \text{falls } n = 1, \\ T(k) + T(n - k - 1) + 1 & \text{sonst.} \end{cases}$$

Zeige (mit Substitutionsmethode) $T(n) \leq c \cdot n - 1$

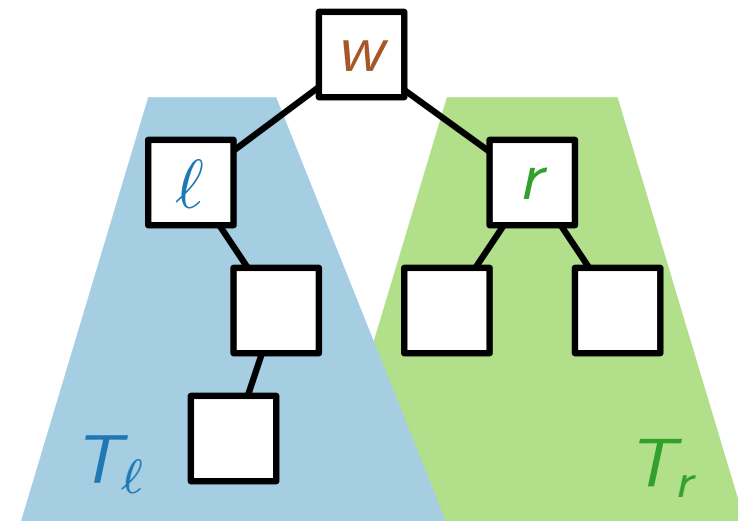
oder: Für jeden Knoten und jede Kante des Baums führt INORDERTREEWALK eine konstante Anzahl von Schritten aus.

Für Bäume gilt: $\# \text{Kanten} = \# \text{Knoten} - 1 = n - 1$

$\Rightarrow T(n) = c_1 \cdot (n - 1) + c_2 \cdot n \in \mathcal{O}(n)$.

Code:

```
INORDERTREEWALK(Node x = root)
  if x ≠ nil then
    INORDERTREEWALK(x.left)
    gib x.key aus
    INORDERTREEWALK(x.right)
```

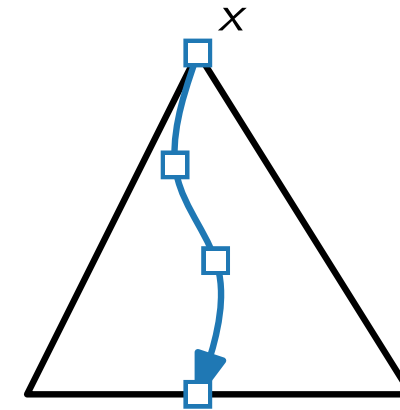


Suche

Node SEARCH(key k , Node $x = root$)

Aufgabe.

Schreiben Sie Pseudocode für die rekursive Methode



Code:

INORDERTREEWALK(Node $x = root$)

if $x \neq nil$ **then**

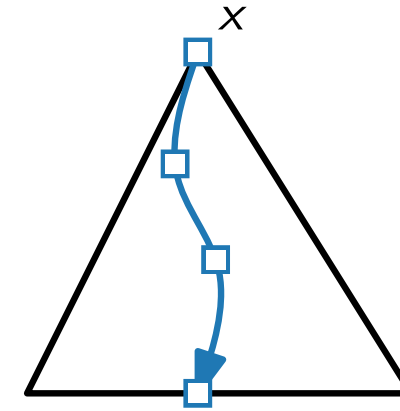
 INORDERTREEWALK($x.left$)

 gib $x.key$ aus

 INORDERTREEWALK($x.right$)

Suche

```
Node SEARCH(key  $k$ , Node  $x = root$ )
  if  $x.key == k$  then
    return  $x$ 
```

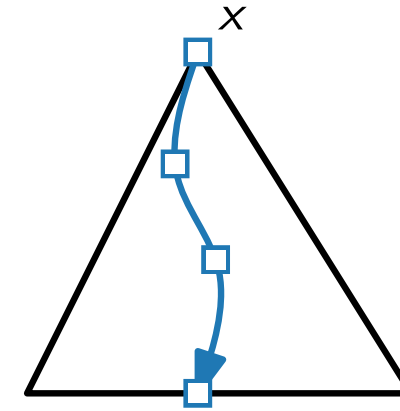


Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```

Suche

```
Node SEARCH(key  $k$ , Node  $x = root$ )
  if  $x == nil$  or  $x.key == k$  then
    └ return  $x$ 
```



Code:

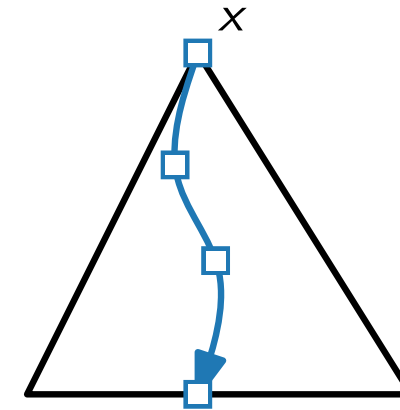
```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    └ INORDERTREEWALK( $x.left$ )
      gib  $x.key$  aus
    └ INORDERTREEWALK( $x.right$ )
```

Suche

```
Node SEARCH(key  $k$ , Node  $x = root$ )
```

```
  if  $x == nil$  or  $x.key == k$  then
    |   return  $x$ 
```

```
  if  $k < x.key$  then
    |
```



Code:

```
INORDERTREEWALK(Node  $x = root$ )
```

```
  if  $x \neq nil$  then
```

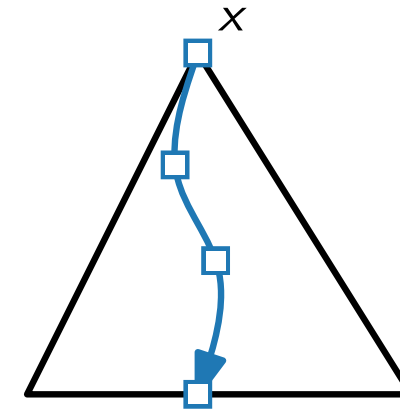
```
    | INORDERTREEWALK( $x.left$ )
```

```
    | gib  $x.key$  aus
```

```
    | INORDERTREEWALK( $x.right$ )
```

Suche

```
Node SEARCH(key  $k$ , Node  $x = root$ )
  if  $x == nil$  or  $x.key == k$  then
    return  $x$ 
  if  $k < x.key$  then
    return SEARCH( $k$ ,  $x.left$ )
```



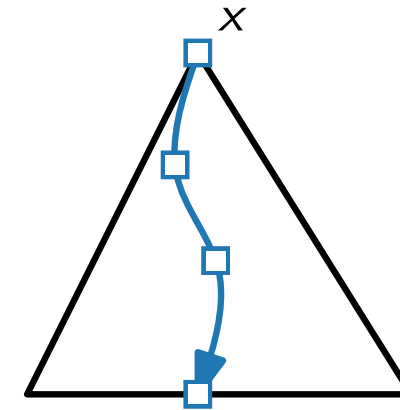
Code:

```
INORDERTREEWALK(Node  $x = root$ )
  if  $x \neq nil$  then
    INORDERTREEWALK( $x.left$ )
    gib  $x.key$  aus
    INORDERTREEWALK( $x.right$ )
```

Suche

```

Node SEARCH(key  $k$ , Node  $x = root$ )
    if  $x == nil$  or  $x.key == k$  then
        return  $x$ 
    if  $k < x.key$  then
        return SEARCH( $k$ ,  $x.left$ )
    else return SEARCH( $k$ ,  $x.right$ )
  
```



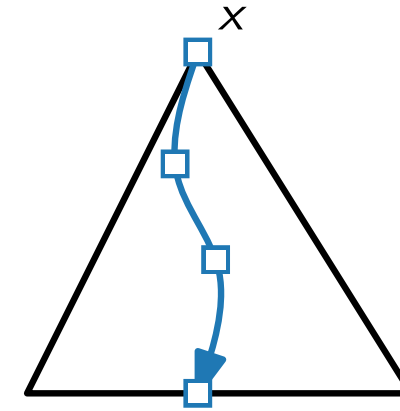
Code:

```

INORDERTREEWALK(Node  $x = root$ )
    if  $x \neq nil$  then
        INORDERTREEWALK( $x.left$ )
        gib  $x.key$  aus
        INORDERTREEWALK( $x.right$ )
  
```

Suche

```
Node SEARCH(key  $k$ , Node  $x = root$ )
    if  $x == nil$  or  $x.key == k$  then
        return  $x$ 
    if  $k < x.key$  then
        return SEARCH( $k$ ,  $x.left$ )
    else return SEARCH( $k$ ,  $x.right$ )
```



Laufzeit:

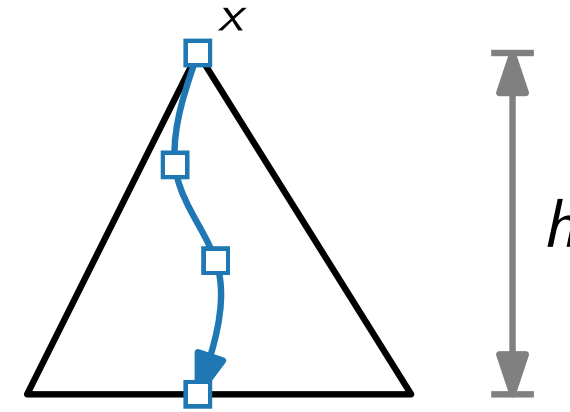
Code:

```
INORDERTREEWALK(Node  $x = root$ )
    if  $x \neq nil$  then
        INORDERTREEWALK( $x.left$ )
        gib  $x.key$  aus
        INORDERTREEWALK( $x.right$ )
```

Suche

```

Node SEARCH(key  $k$ , Node  $x = root$ )
    if  $x == nil$  or  $x.key == k$  then
        return  $x$ 
    if  $k < x.key$  then
        return SEARCH( $k$ ,  $x.left$ )
    else return SEARCH( $k$ ,  $x.right$ )
  
```



Laufzeit:

Code:

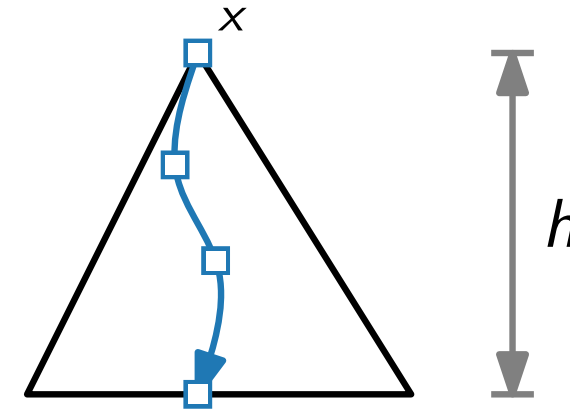
```

INORDERTREEWALK(Node  $x = root$ )
    if  $x \neq nil$  then
        INORDERTREEWALK( $x.left$ )
        gib  $x.key$  aus
        INORDERTREEWALK( $x.right$ )
  
```

Suche

```

Node SEARCH(key  $k$ , Node  $x = root$ )
    if  $x == nil$  or  $x.key == k$  then
        return  $x$ 
    if  $k < x.key$  then
        return SEARCH( $k$ ,  $x.left$ )
    else return SEARCH( $k$ ,  $x.right$ )
  
```



Laufzeit: $\mathcal{O}(h)$

Code:

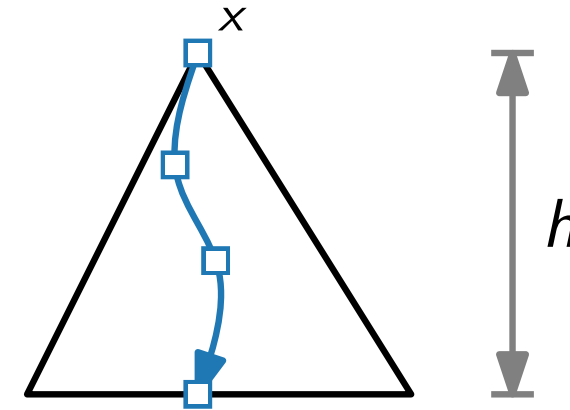
```

INORDERTREEWALK(Node  $x = root$ )
    if  $x \neq nil$  then
        INORDERTREEWALK( $x.left$ )
        gib  $x.key$  aus
        INORDERTREEWALK( $x.right$ )
  
```

Suche

rekursiv

```
Node SEARCH(key  $k$ , Node  $x = root$ )
    if  $x == nil$  or  $x.key == k$  then
        return  $x$ 
    if  $k < x.key$  then
        return SEARCH( $k$ ,  $x.left$ )
    else return SEARCH( $k$ ,  $x.right$ )
```



Laufzeit: $\mathcal{O}(h)$

Code:

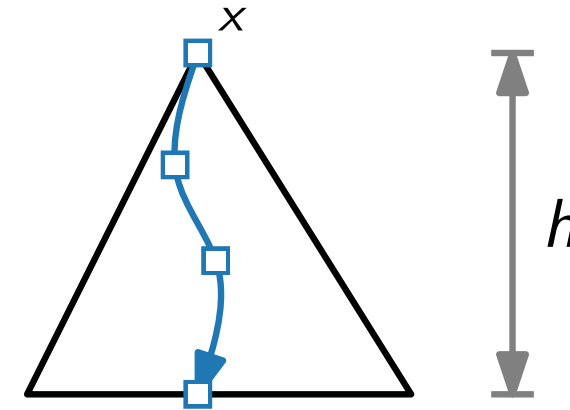
```
INORDERTREEWALK(Node  $x = root$ )
    if  $x \neq nil$  then
        INORDERTREEWALK( $x.left$ )
        gib  $x.key$  aus
        INORDERTREEWALK( $x.right$ )
```

Suche

rekursiv

```
Node SEARCH(key  $k$ , Node  $x = root$ )  
  
  if  $x == nil$  or  $x.key == k$  then  
    └ return  $x$   
  if  $k < x.key$  then  
    └ return SEARCH( $k$ ,  $x.left$ )  
  else return SEARCH( $k$ ,  $x.right$ )
```

iterativ



Laufzeit: $\mathcal{O}(h)$

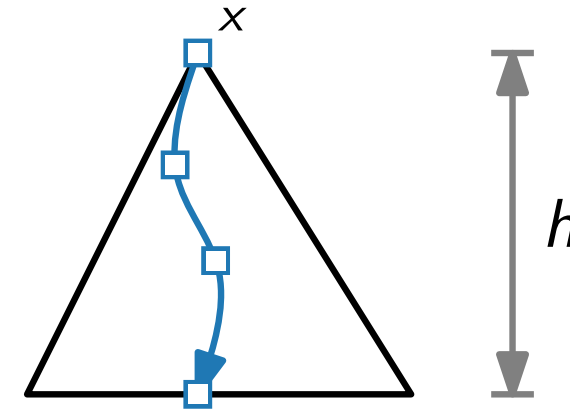
Suche

rekursiv

```
Node SEARCH(key  $k$ , Node  $x = root$ )  
  
  if  $x == nil$  or  $x.key == k$  then  
    └ return  $x$   
  if  $k < x.key$  then  
    └ return SEARCH( $k$ ,  $x.left$ )  
  else return SEARCH( $k$ ,  $x.right$ )
```

iterativ

```
while  $x \neq nil$  and  $x.key \neq k$  do  
  └  
return  $x$ 
```



Laufzeit: $\mathcal{O}(h)$

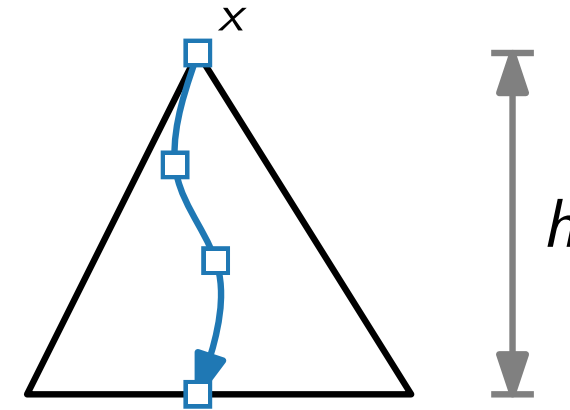
Suche

rekursiv

```
Node SEARCH(key  $k$ , Node  $x = root$ )
  if  $x == nil$  or  $x.key == k$  then
    └ return  $x$ 
  if  $k < x.key$  then
    └ return SEARCH( $k$ ,  $x.left$ )
  else return SEARCH( $k$ ,  $x.right$ )
```

iterativ

```
while  $x \neq nil$  and  $x.key \neq k$  do
  if  $k < x.key$  then
    └
  return  $x$ 
```



Laufzeit: $\mathcal{O}(h)$

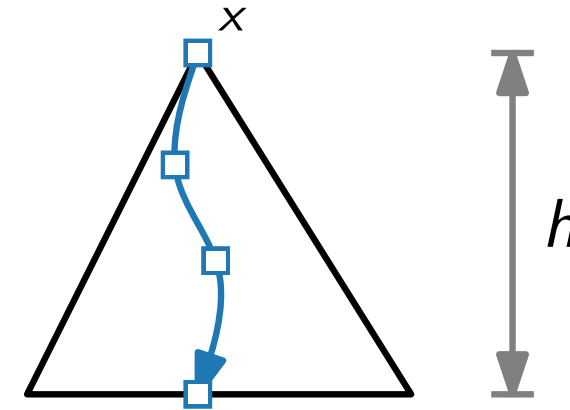
Suche

rekursiv

```
Node SEARCH(key  $k$ , Node  $x = root$ )  
  
  if  $x == nil$  or  $x.key == k$  then  
    |   return  $x$   
  if  $k < x.key$  then  
    |   return SEARCH( $k$ ,  $x.left$ )  
  else return SEARCH( $k$ ,  $x.right$ )
```

iterativ

```
while  $x \neq nil$  and  $x.key \neq k$  do  
  |   if  $k < x.key$  then  
  |   |    $x = x.left$   
  |  
  return  $x$ 
```



Laufzeit: $\mathcal{O}(h)$

Suche

rekursiv

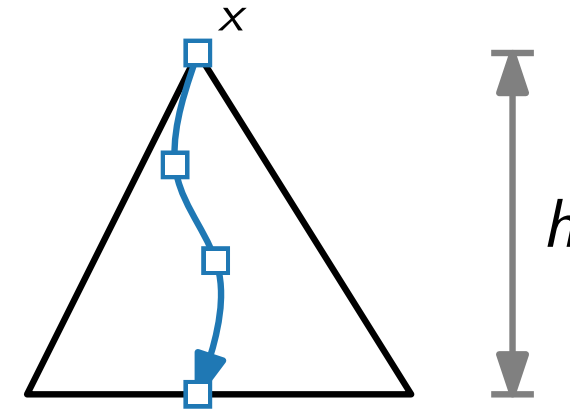
```

Node SEARCH(key  $k$ , Node  $x = root$ )
    if  $x == nil$  or  $x.key == k$  then
        return  $x$ 
    if  $k < x.key$  then
        return SEARCH( $k$ ,  $x.left$ )
    else return SEARCH( $k$ ,  $x.right$ )
  
```

iterativ

```

while  $x \neq nil$  and  $x.key \neq k$  do
    if  $k < x.key$  then
         $x = x.left$ 
    else  $x = x.right$ 
return  $x$ 
  
```



Laufzeit: $\mathcal{O}(h)$

Suche

rekursiv

```

Node SEARCH(key  $k$ , Node  $x = root$ )
  if  $x == nil$  or  $x.key == k$  then
    return  $x$ 
  if  $k < x.key$  then
    return SEARCH( $k$ ,  $x.left$ )
  else return SEARCH( $k$ ,  $x.right$ )

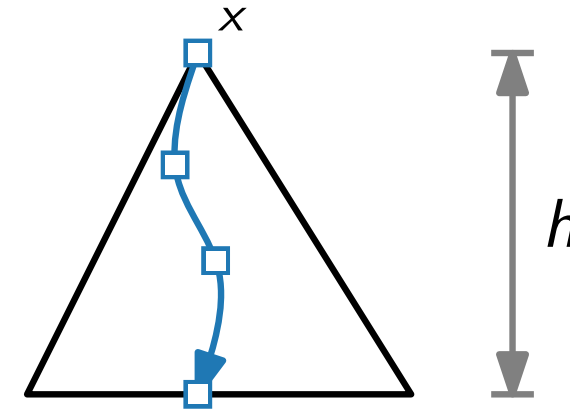
```

iterativ

```

while  $x \neq nil$  and  $x.key \neq k$  do
  if  $k < x.key$  then
     $x = x.left$ 
  else  $x = x.right$ 
return  $x$ 

```



Laufzeit: $\mathcal{O}(h)$

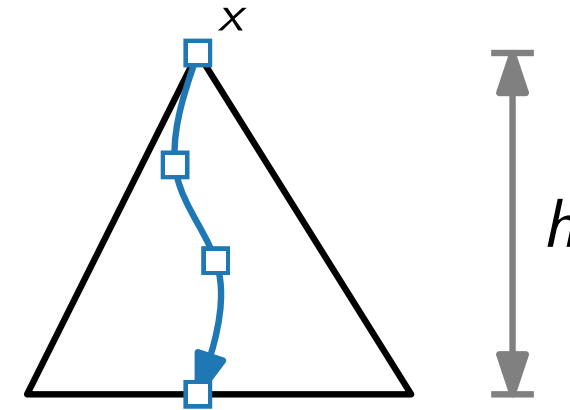
Suche

rekursiv

```
Node SEARCH(key  $k$ , Node  $x = root$ )
  if  $x == nil$  or  $x.key == k$  then
    return  $x$ 
  if  $k < x.key$  then
    return SEARCH( $k$ ,  $x.left$ )
  else return SEARCH( $k$ ,  $x.right$ )
```

iterativ

```
while  $x \neq nil$  and  $x.key \neq k$  do
  if  $k < x.key$  then
     $x = x.left$ 
  else  $x = x.right$ 
return  $x$ 
```



Laufzeit: $\mathcal{O}(h)$

Laufzeit:

Suche

rekursiv

```

Node SEARCH(key  $k$ , Node  $x = root$ )
  if  $x == nil$  or  $x.key == k$  then
    return  $x$ 
  if  $k < x.key$  then
    return SEARCH( $k$ ,  $x.left$ )
  else return SEARCH( $k$ ,  $x.right$ )

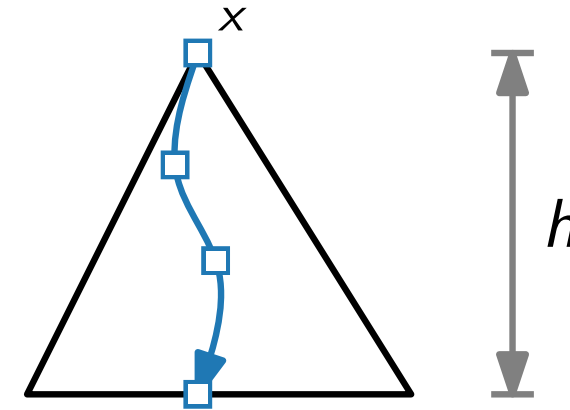
```

iterativ

```

while  $x \neq nil$  and  $x.key \neq k$  do
  if  $k < x.key$  then
     $x = x.left$ 
  else  $x = x.right$ 
return  $x$ 

```



Laufzeit: $\mathcal{O}(h)$

Laufzeit: $\mathcal{O}(h)$

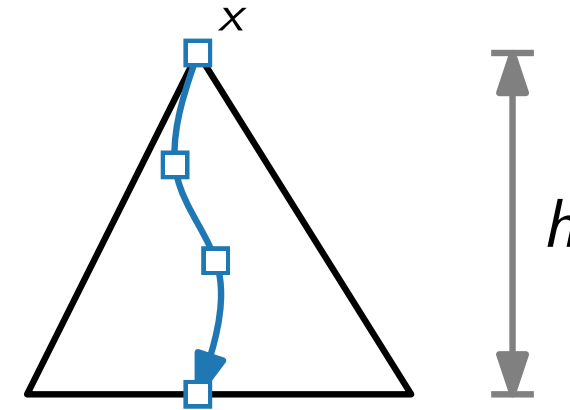
Suche

rekursiv

```
Node SEARCH(key  $k$ , Node  $x = root$ )
  if  $x == nil$  or  $x.key == k$  then
    return  $x$ 
  if  $k < x.key$  then
    return SEARCH( $k$ ,  $x.left$ )
  else return SEARCH( $k$ ,  $x.right$ )
```

iterativ

```
while  $x \neq nil$  and  $x.key \neq k$  do
  if  $k < x.key$  then
     $x = x.left$ 
  else  $x = x.right$ 
return  $x$ 
```



Laufzeit: $\mathcal{O}(h)$

Laufzeit: $\mathcal{O}(h)$

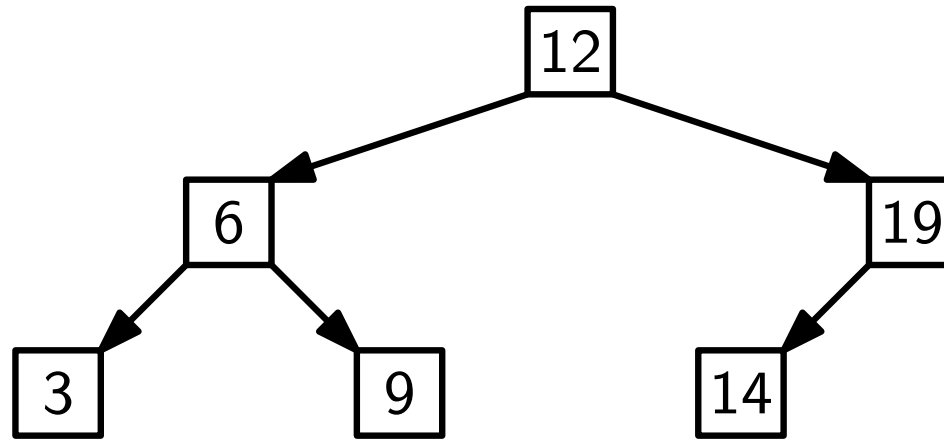
Trotzdem schneller, da keine Verwaltung der rekursiven Methodenaufrufe.

Minimum & Maximum

Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?

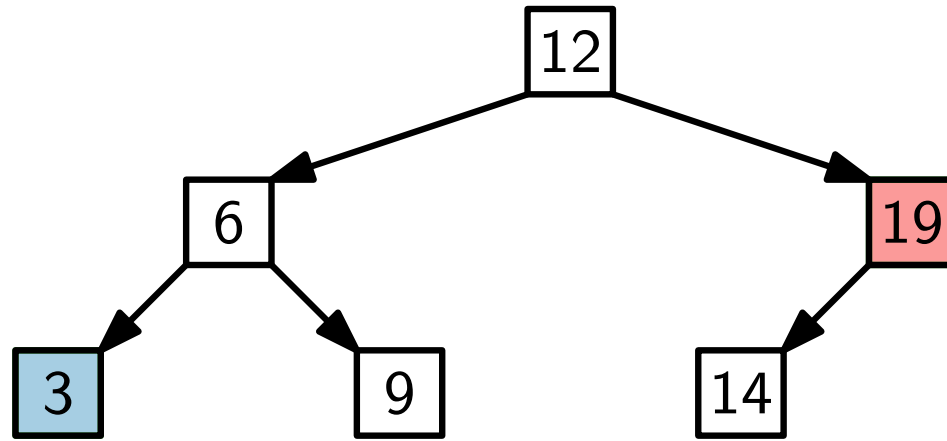
Minimum & Maximum

Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



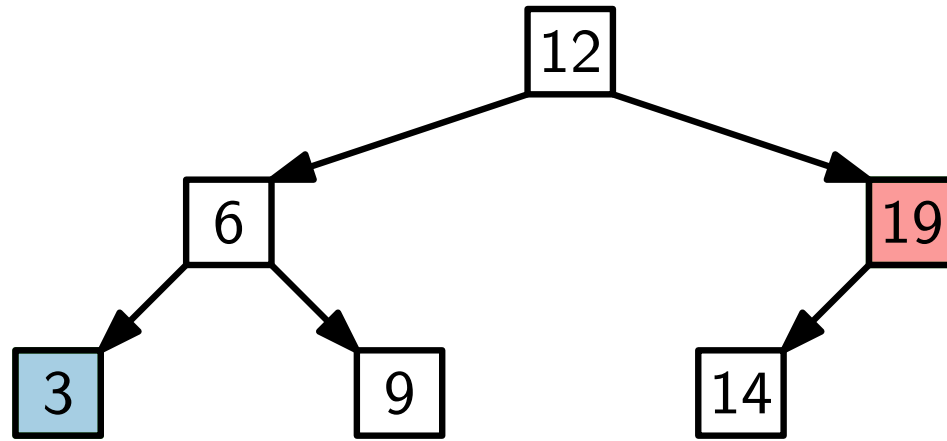
Minimum & Maximum

Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



Minimum & Maximum

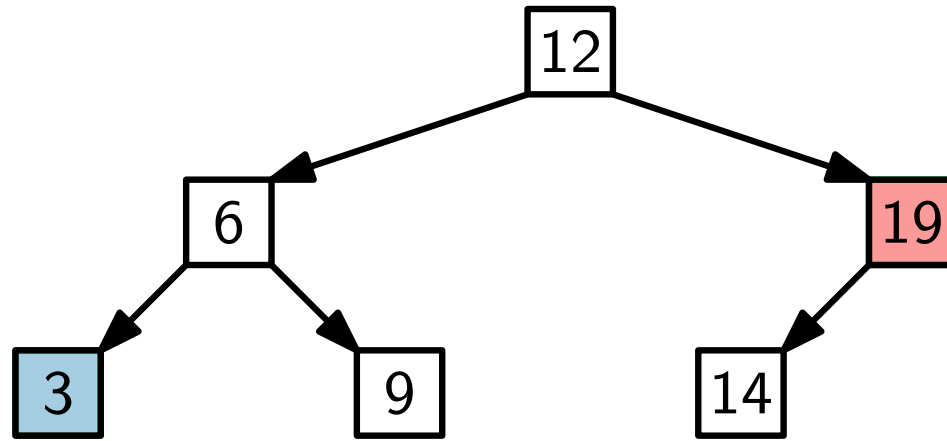
Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



Antwort: Min steht ganz links, Max ganz rechts!

Minimum & Maximum

Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



Antwort: Min steht ganz links, Max ganz rechts!

iterativ

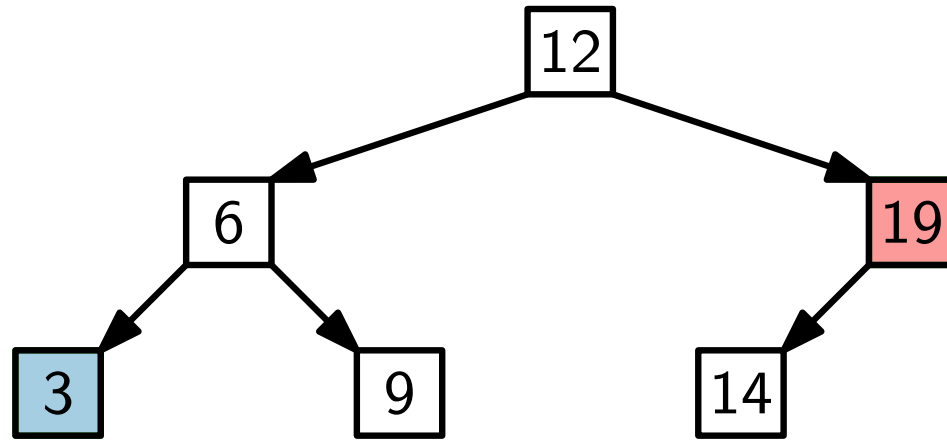
`Node MINIMUM(Node  $x = root$ )`

Aufgabe.

Schreiben Sie für binäre Suchbäume die Methode

Minimum & Maximum

Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



Antwort: Min steht ganz links, Max ganz rechts!

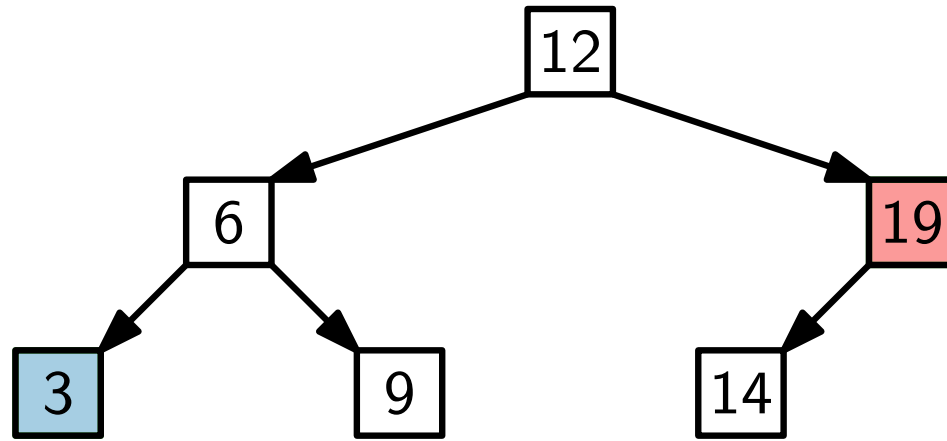
iterativ

```
Node MINIMUM(Node  $x = root$ )
```

```
while  $x.left \neq nil$  do  
   $x = x.left$   
return  $x$ 
```

Minimum & Maximum

Frage: Was folgt aus der Binärer-Suchbaum-Eigenschaft für die Position von Min und Max im Baum?



Antwort: Min steht ganz links, Max ganz rechts!

iterativ

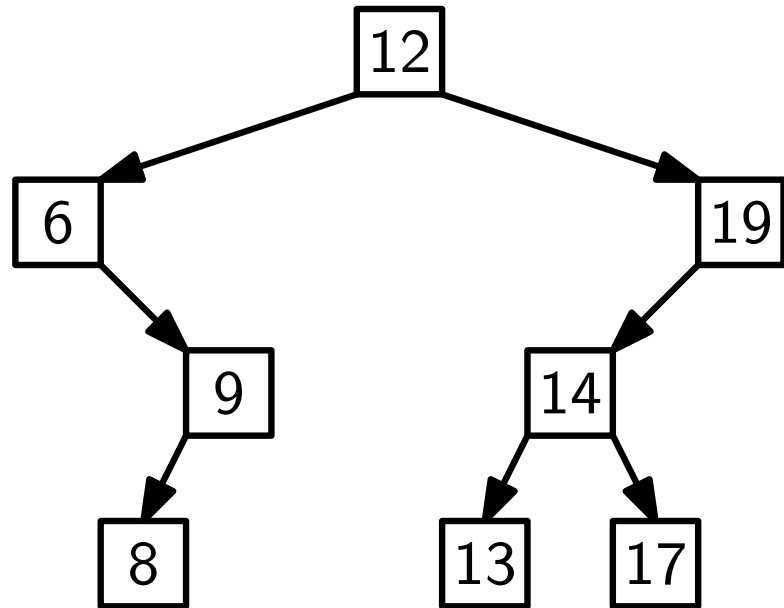
```
Node MINIMUM(Node x = root)
  if x == nil then return nil
  while x.left != nil do
    x = x.left
  return x
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Nachfolger (und Vorgänger)

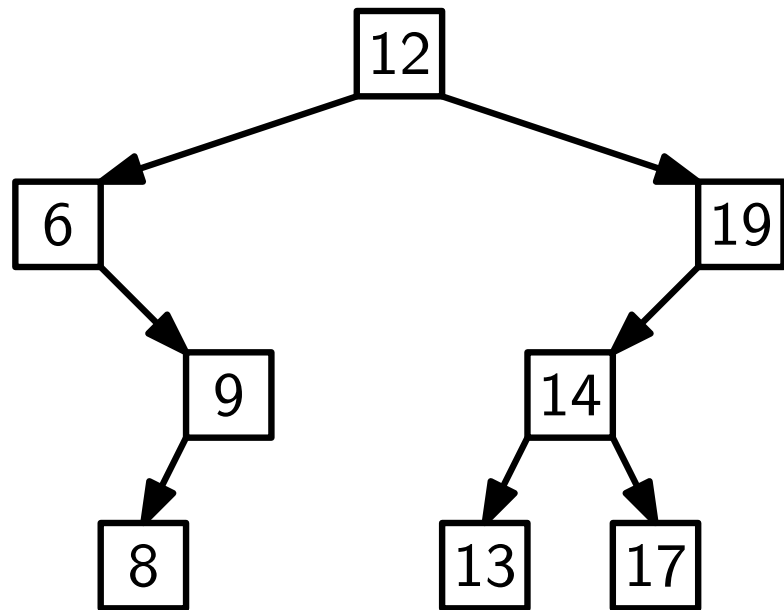
Vereinfachende Annahme: alle Schlüssel sind verschieden.



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

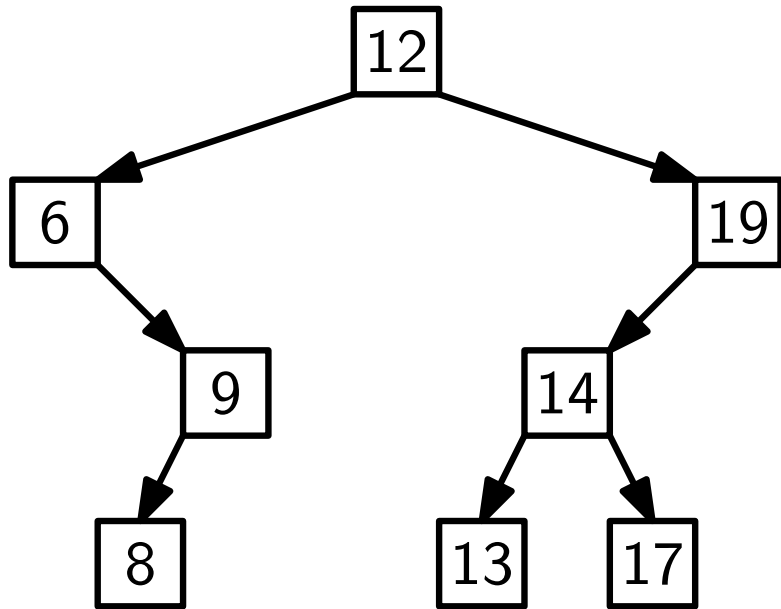
Erinnerung: $\text{NACHFOLGER}(x) =$



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

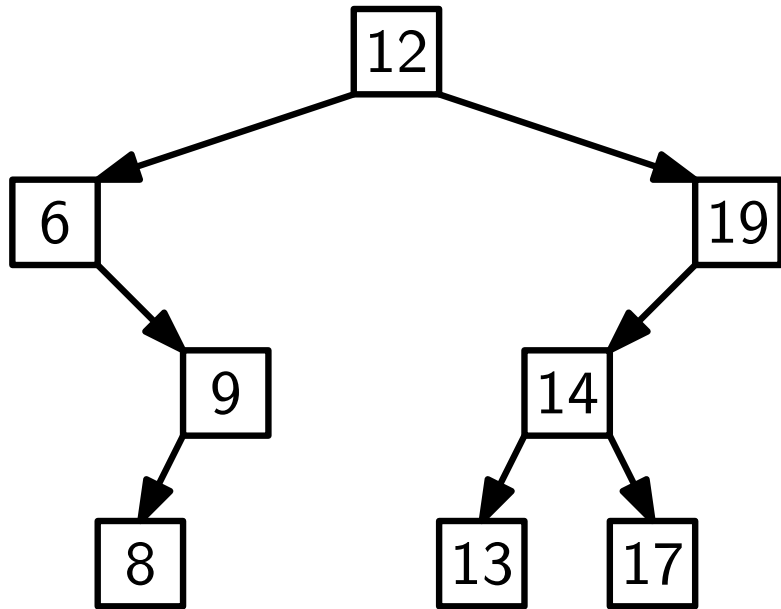
Erinnerung: $\text{NACHFOLGER}(x) =$ Knoten mit kleinstem Schlüssel
unter allen y mit $y.\text{key} > x.\text{key}$.



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

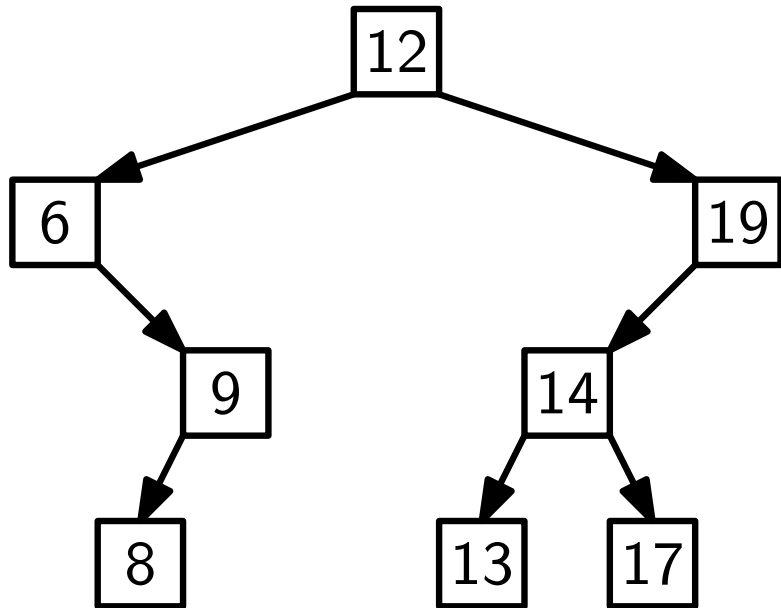
Erinnerung: $\text{NACHFOLGER}(x) =$ Knoten mit kleinstem Schlüssel
unter allen y mit $y.\text{key} > x.\text{key}$.
 $= \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

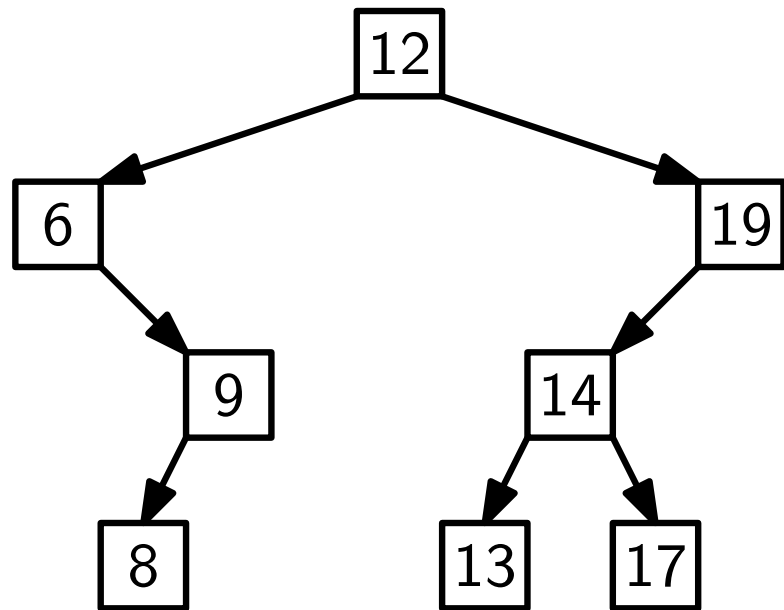
Erinnerung: $\text{NACHFOLGER}(x) =$ Knoten mit kleinstem Schlüssel
unter allen y mit $y.\text{key} > x.\text{key}$.
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) =$ Knoten mit kleinstem Schlüssel
unter allen y mit $y.\text{key} > x.\text{key}$.
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$

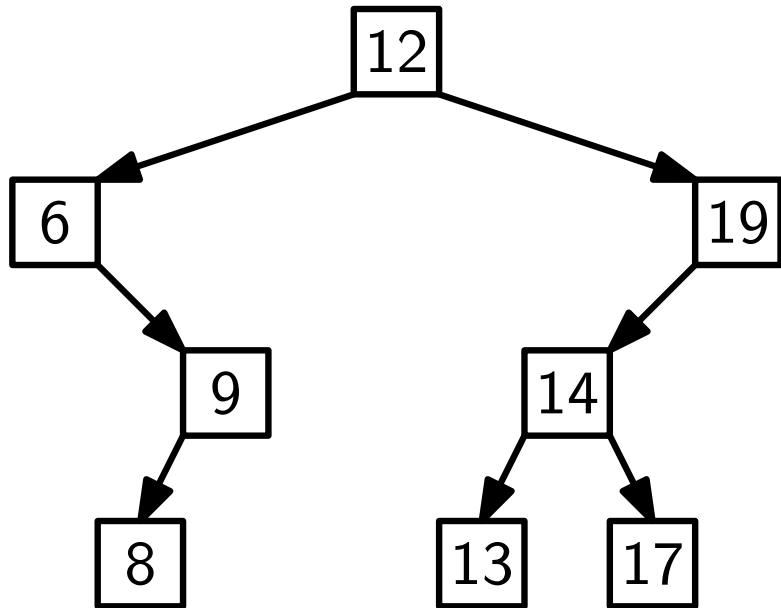


Nachfolger(19) = *nil*

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) =$ Knoten mit kleinstem Schlüssel
unter allen y mit $y.\text{key} > x.\text{key}$.
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



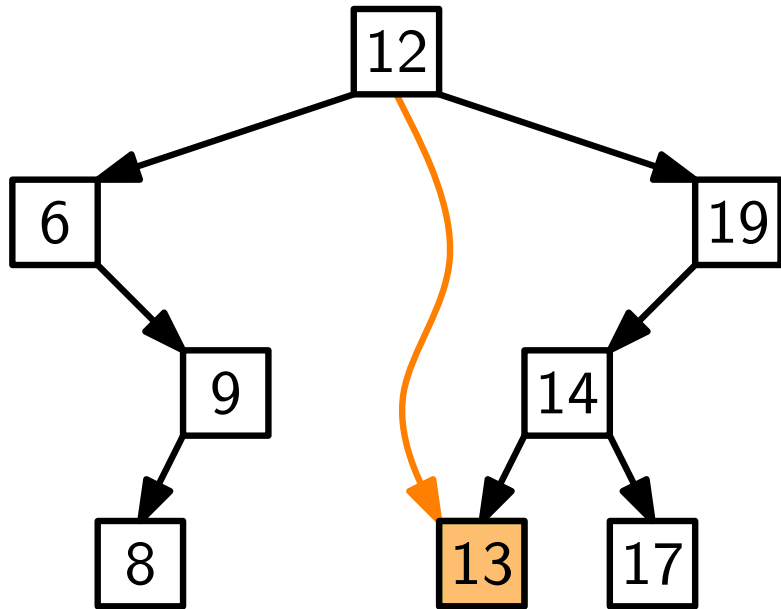
Nachfolger(19) = *nil*

Nachfolger(12) = ?

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel}$
unter allen y mit $y.\text{key} > x.\text{key}$.
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



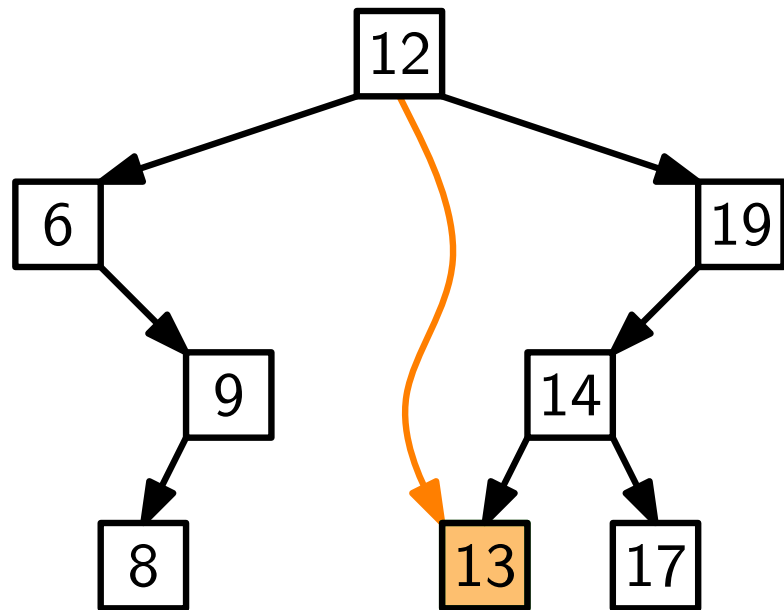
Nachfolger(19) = *nil*

Nachfolger(12) = ?

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



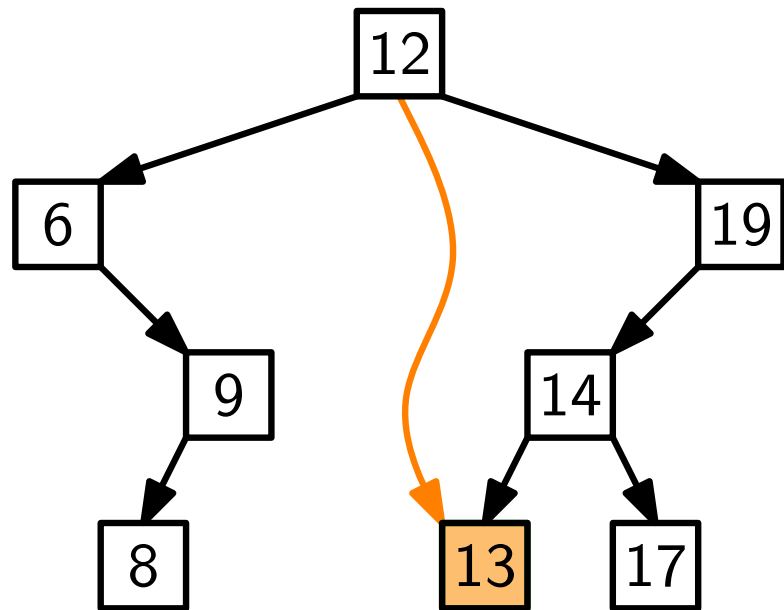
Nachfolger(19) = *nil*

Nachfolger(12) = 13 = MINIMUM(12.right)

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



Nachfolger(19) = *nil*

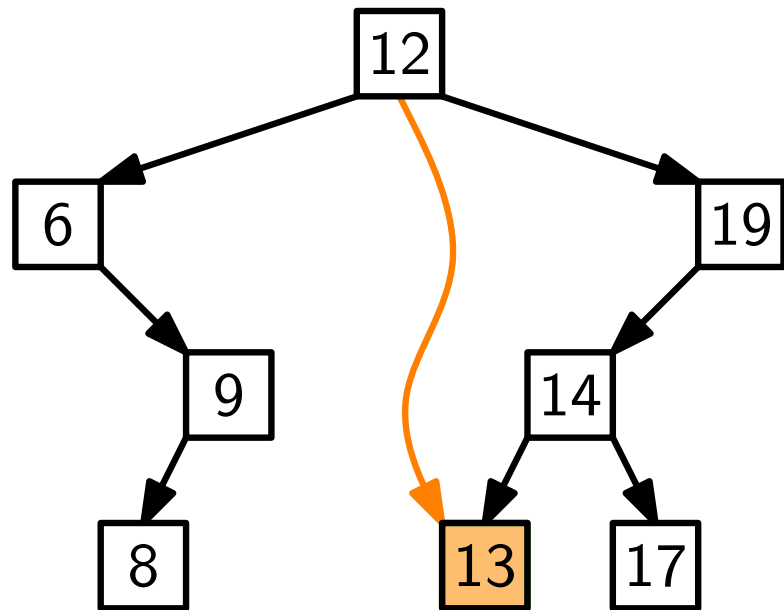
Nachfolger(12) = 13 = MINIMUM(12.right)

Nachfolger(9) = ?

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



Nachfolger(19) = *nil*

Nachfolger(12) = 13 = MINIMUM(12.right)

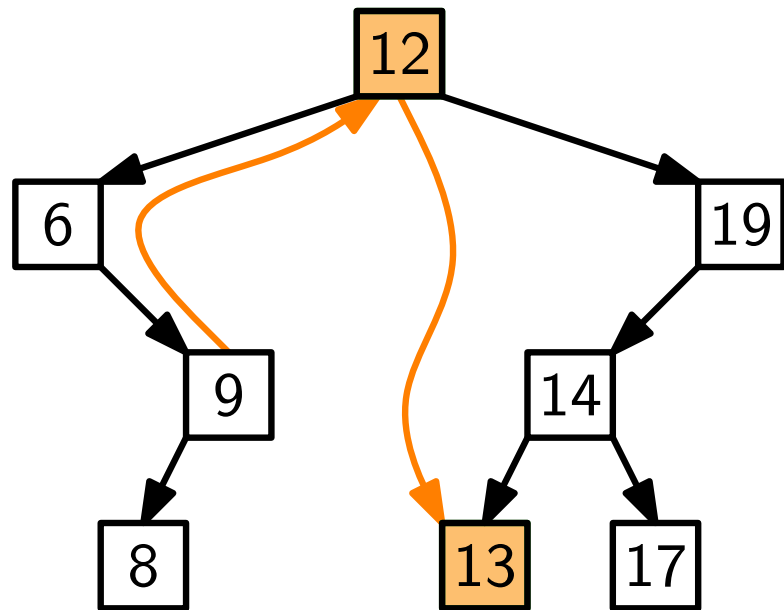
Nachfolger(9) = ?

9 hat kein rechtes Kind; 9 = MAXIMUM(12.left)

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



Nachfolger(19) = *nil*

Nachfolger(12) = 13 = MINIMUM(12.right)

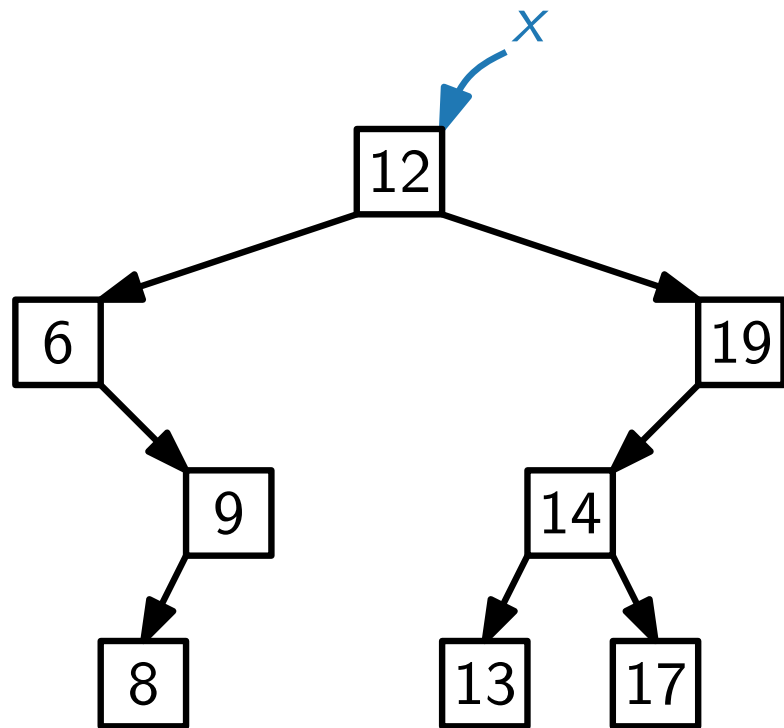
Nachfolger(9) = ?

9 hat kein rechtes Kind; 9 = MAXIMUM(12.left)

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) =$ Knoten mit kleinstem Schlüssel
unter allen y mit $y.\text{key} > x.\text{key}$.
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$

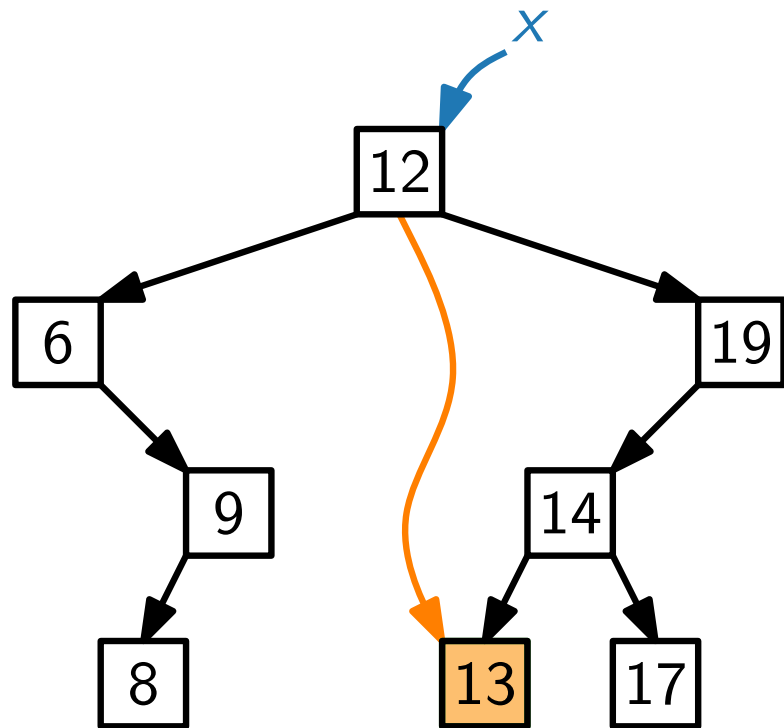


NODE SUCCESSOR(NODE x)

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) =$ Knoten mit kleinstem Schlüssel
unter allen y mit $y.\text{key} > x.\text{key}$.
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$

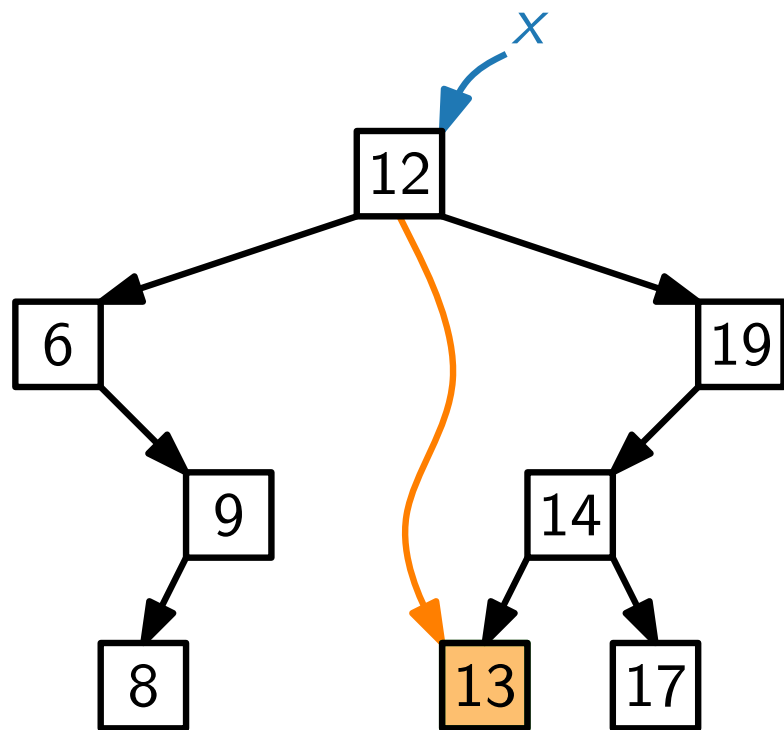


NODE SUCCESSOR(NODE x)

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$

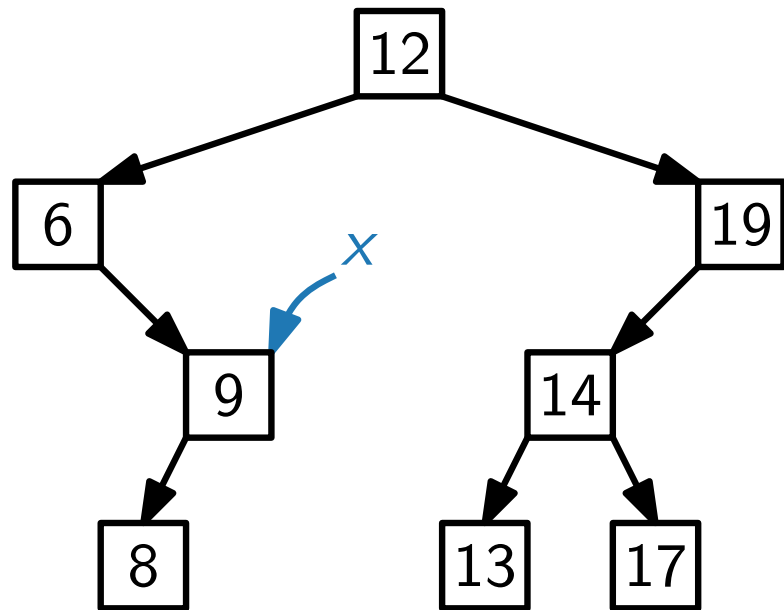


```
NODE SUCCESSOR(NODE x)  
if x.right  $\neq$  nil then  
  return MINIMUM(x.right)
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$

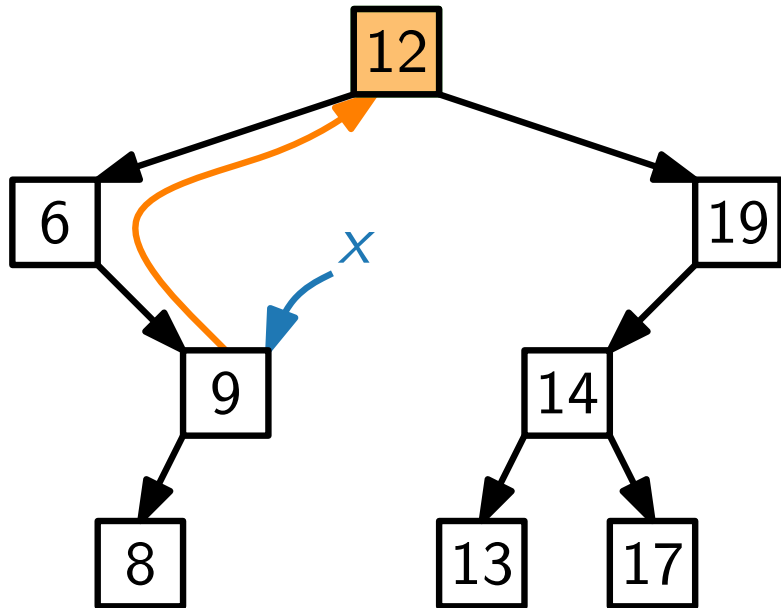


```
NODE SUCCESSOR(NODE x)  
  if x.right  $\neq$  nil then  
    return MINIMUM(x.right)
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$

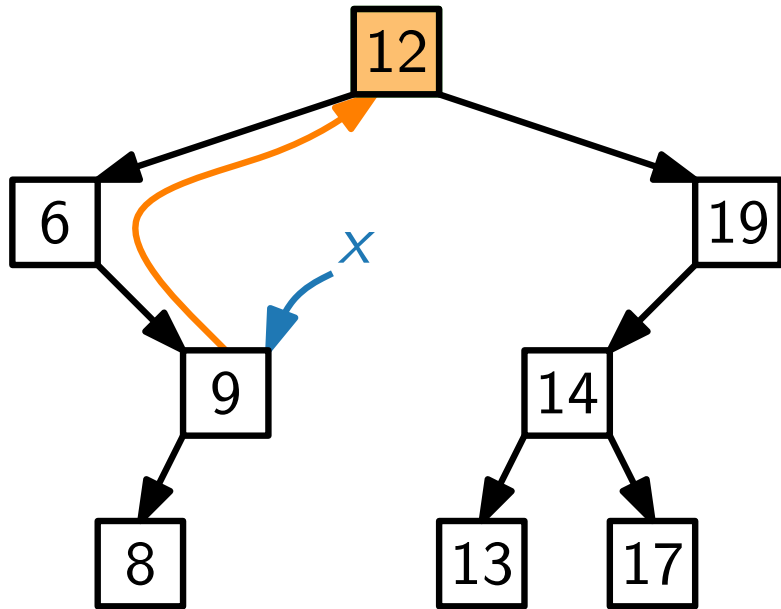


```
NODE SUCCESSOR(NODE x)  
  if x.right ≠ nil then  
    return MINIMUM(x.right)
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$

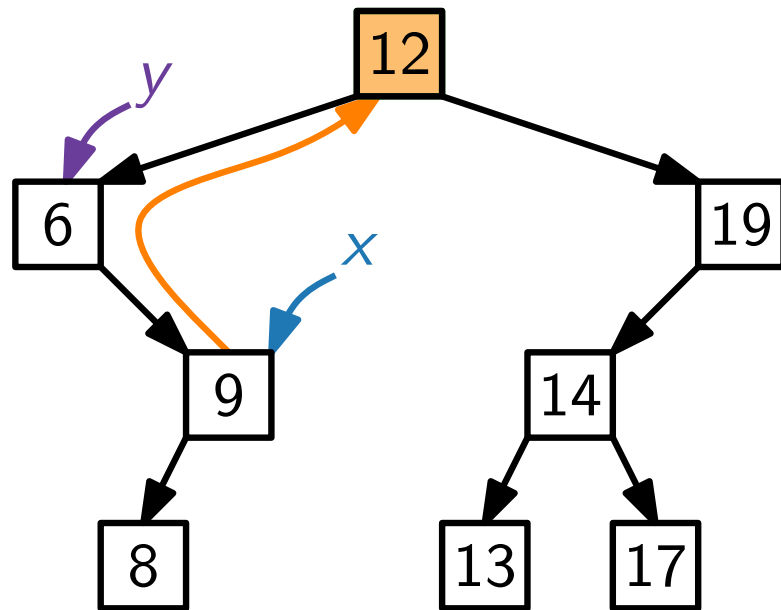


```
NODE SUCCESSOR(NODE x)  
  if x.right ≠ nil then  
    | return MINIMUM(x.right)  
  y = x.p
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$

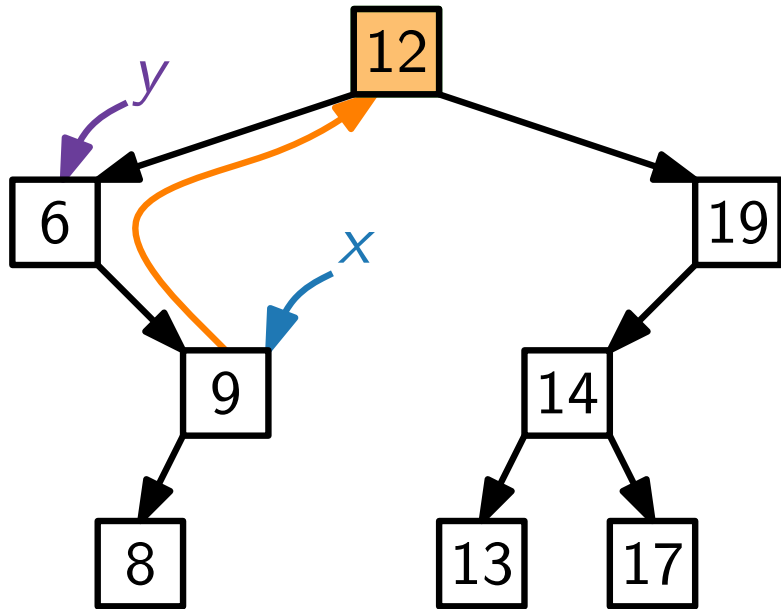


```
NODE SUCCESSOR(NODE x)  
  if x.right ≠ nil then  
    return MINIMUM(x.right)  
  y = x.p
```

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$

$$= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$$


NODE SUCCESSOR(NODE x)

if $x.\text{right} \neq \text{nil}$ **then**
 return MINIMUM($x.\text{right}$)

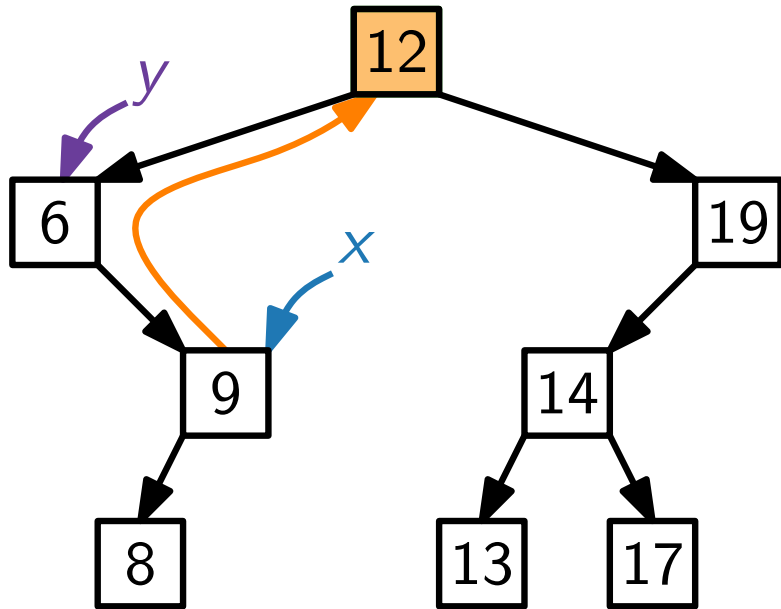
$y = x.p$

while $y \neq \text{nil}$ **and** $x == y.\text{right}$ **do**

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$

$$= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$$


NODE SUCCESSOR(NODE x)

if $x.\text{right} \neq \text{nil}$ **then**
 return MINIMUM($x.\text{right}$)

$y = x.p$

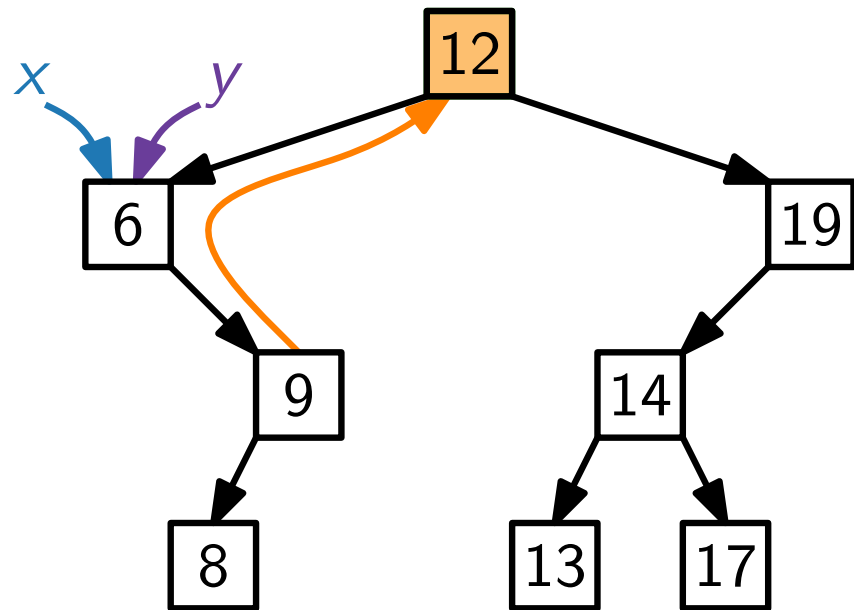
while $y \neq \text{nil}$ **and** $x == y.\text{right}$ **do**

$x = y$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$

$$= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$$


NODE SUCCESSOR(NODE x)

if $x.\text{right} \neq \text{nil}$ **then**

└ **return** MINIMUM($x.\text{right}$)

$y = x.p$

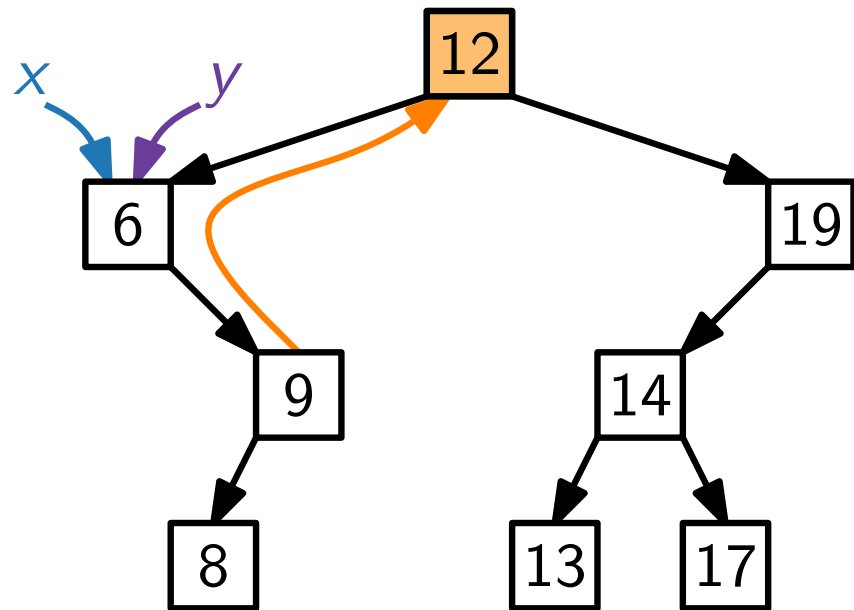
while $y \neq \text{nil}$ **and** $x == y.\text{right}$ **do**

└ $x = y$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$

$$= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$$


NODE SUCCESSOR(NODE x)

if $x.\text{right} \neq \text{nil}$ **then**
 return MINIMUM($x.\text{right}$)

$y = x.p$

while $y \neq \text{nil}$ **and** $x == y.\text{right}$ **do**

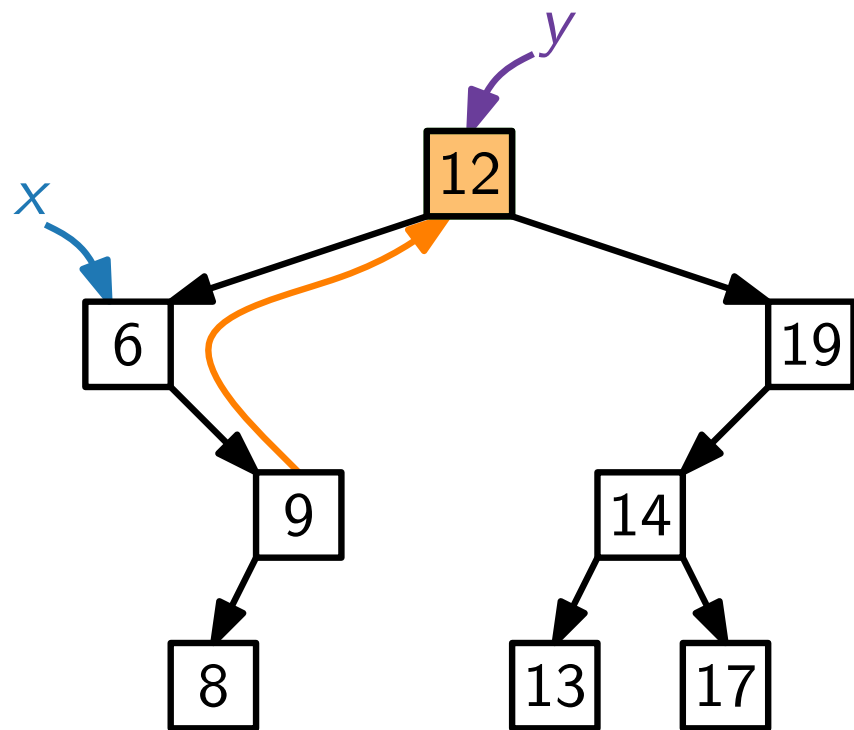
$x = y$

$y = y.p$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



NODE SUCCESSOR(NODE x)

if $x.\text{right} \neq \text{nil}$ **then**
 | **return** MINIMUM($x.\text{right}$)

$y = x.p$

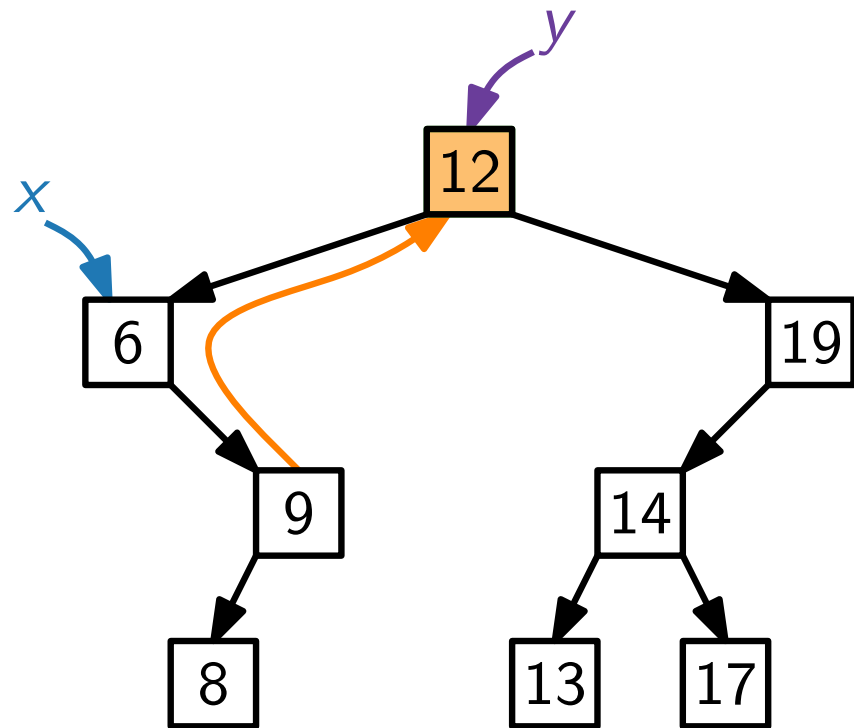
while $y \neq \text{nil}$ **and** $x == y.\text{right}$ **do**

| $x = y$
 | $y = y.p$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



NODE SUCCESSOR(NODE x)

if $x.\text{right} \neq \text{nil}$ **then**
 | **return** MINIMUM($x.\text{right}$)

$y = x.p$

while $y \neq \text{nil}$ **and** $x == y.\text{right}$ **do**

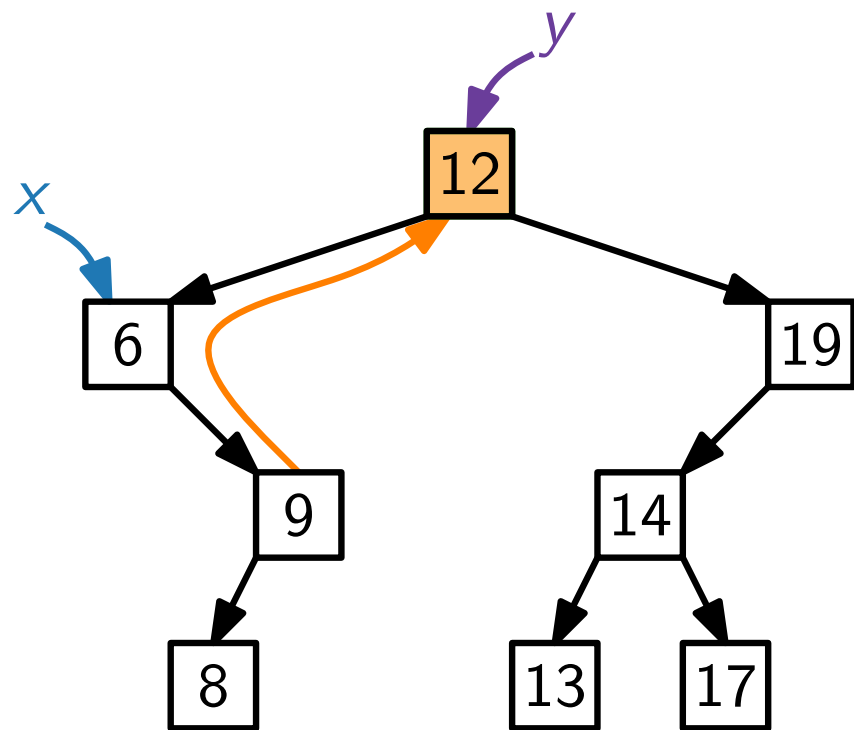
| $x = y$

| $y = y.p$

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$
 $= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$



NODE SUCCESSOR(NODE x)

if $x.\text{right} \neq \text{nil}$ **then**
 | **return** MINIMUM($x.\text{right}$)

$y = x.p$

while $y \neq \text{nil}$ **and** $x == y.\text{right}$ **do**

| $x = y$

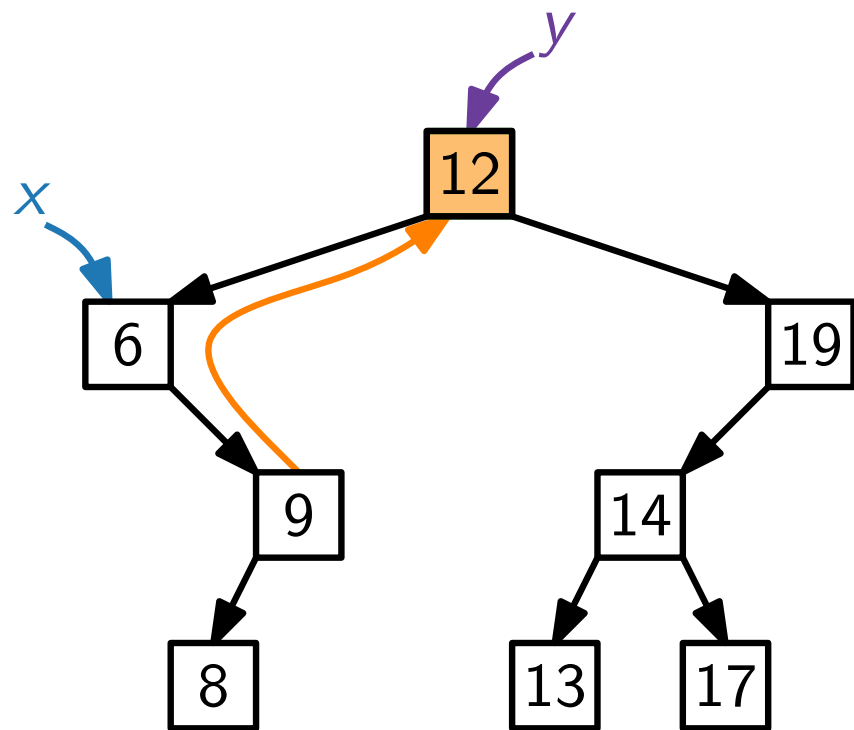
| $y = y.p$

return y

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$

$$= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$$


NODE SUCCESSOR(NODE x)

if $x.\text{right} \neq \text{nil}$ **then**
 return MINIMUM($x.\text{right}$)

$y = x.p$

while $y \neq \text{nil}$ **and** $x == y.\text{right}$ **do**

$x = y$

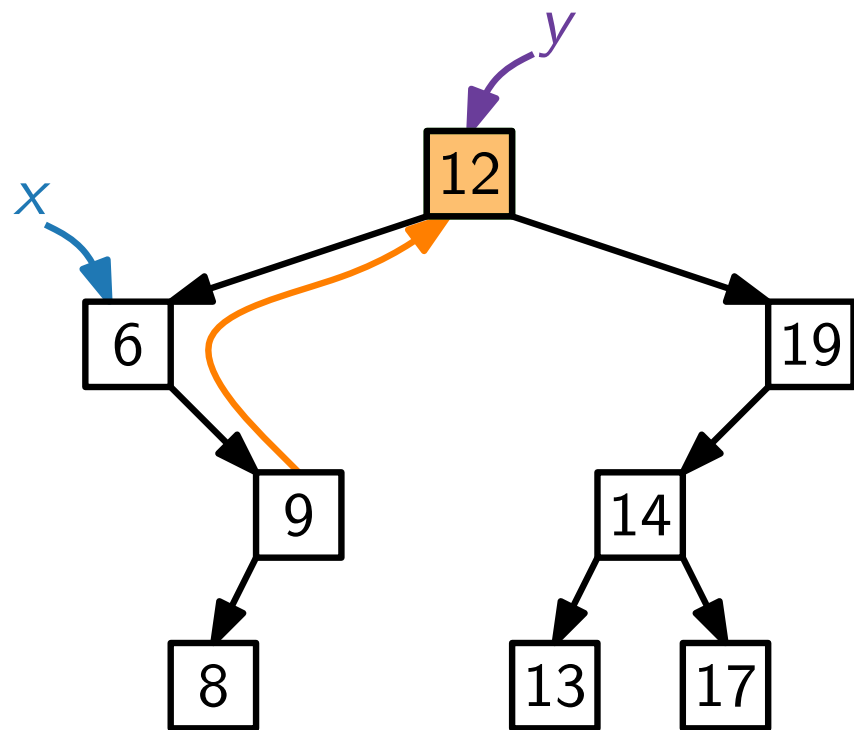
$y = y.p$

return y

Nachfolger (und Vorgänger)

Vereinfachende Annahme: alle Schlüssel sind verschieden.

Erinnerung: $\text{NACHFOLGER}(x) = \text{Knoten mit kleinstem Schlüssel unter allen } y \text{ mit } y.\text{key} > x.\text{key}.$

$$= \arg \min_y \{y.\text{key} \mid y.\text{key} > x.\text{key}\}.$$


Tipp: Probieren Sie auch
z.B. $\text{SUCCESSOR}(19)$!

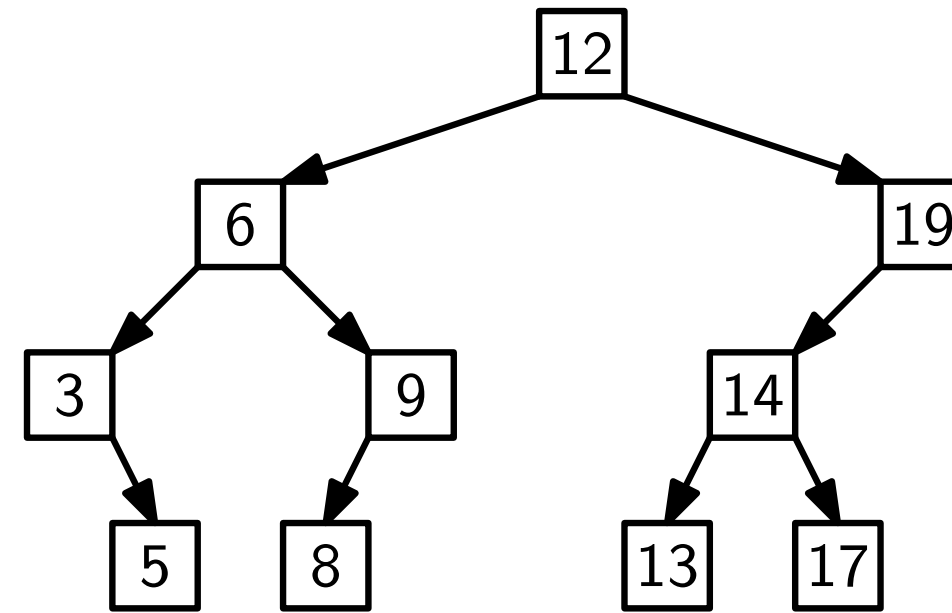
NODE SUCCESSOR(NODE x)

```

if  $x.\text{right} \neq \text{nil}$  then
  | return MINIMUM( $x.\text{right}$ )
 $y = x.p$ 
while  $y \neq \text{nil}$  and  $x == y.\text{right}$  do
  |  $x = y$ 
  |  $y = y.p$ 
return  $y$ 
  
```

Einfügen

Node INSERT(key k)



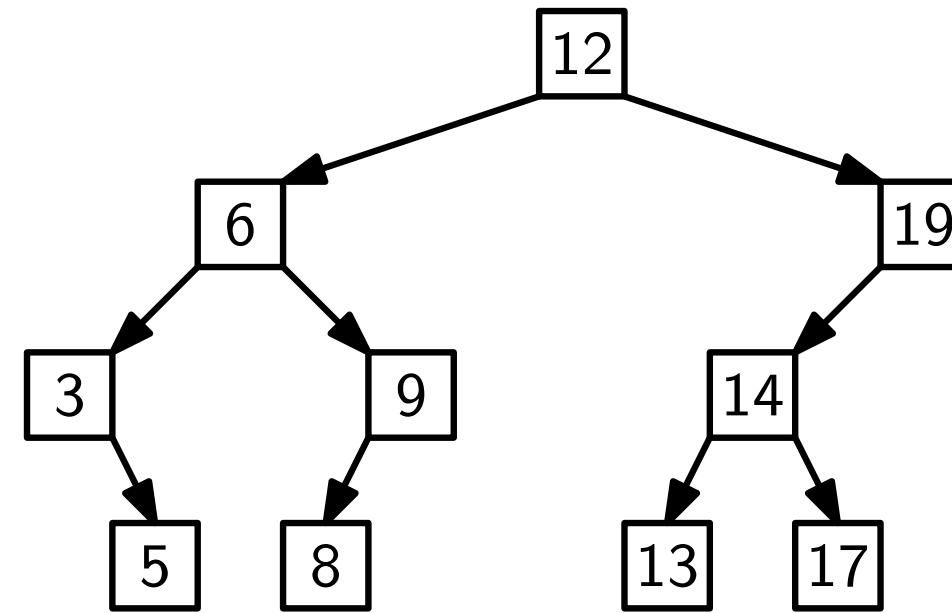
INSERT(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$



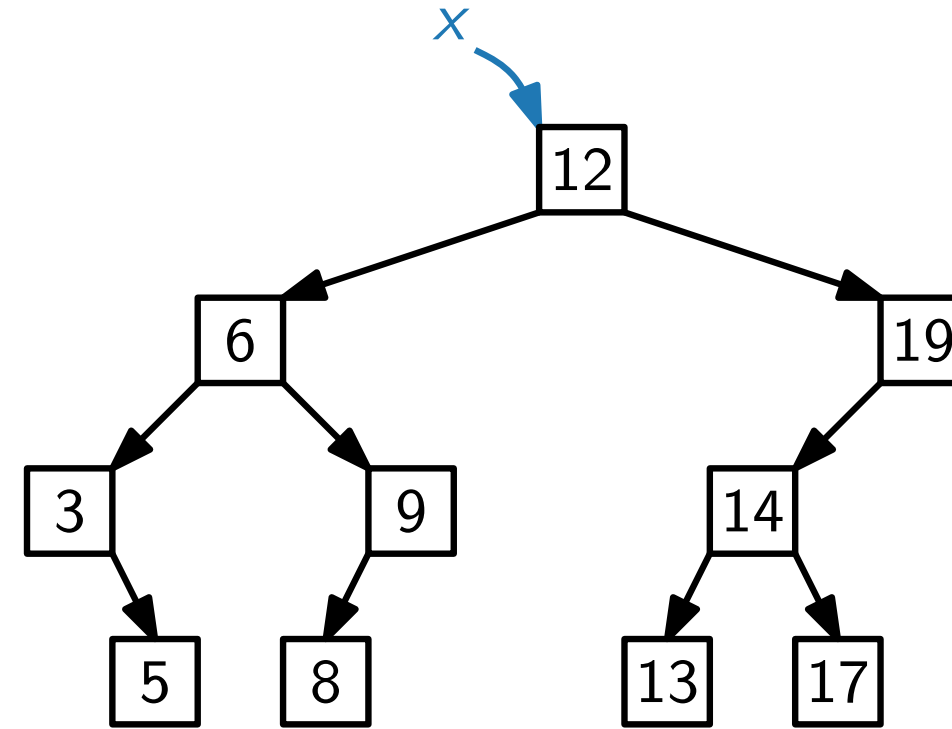
INSERT(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$



INSERT(11)

Einfügen

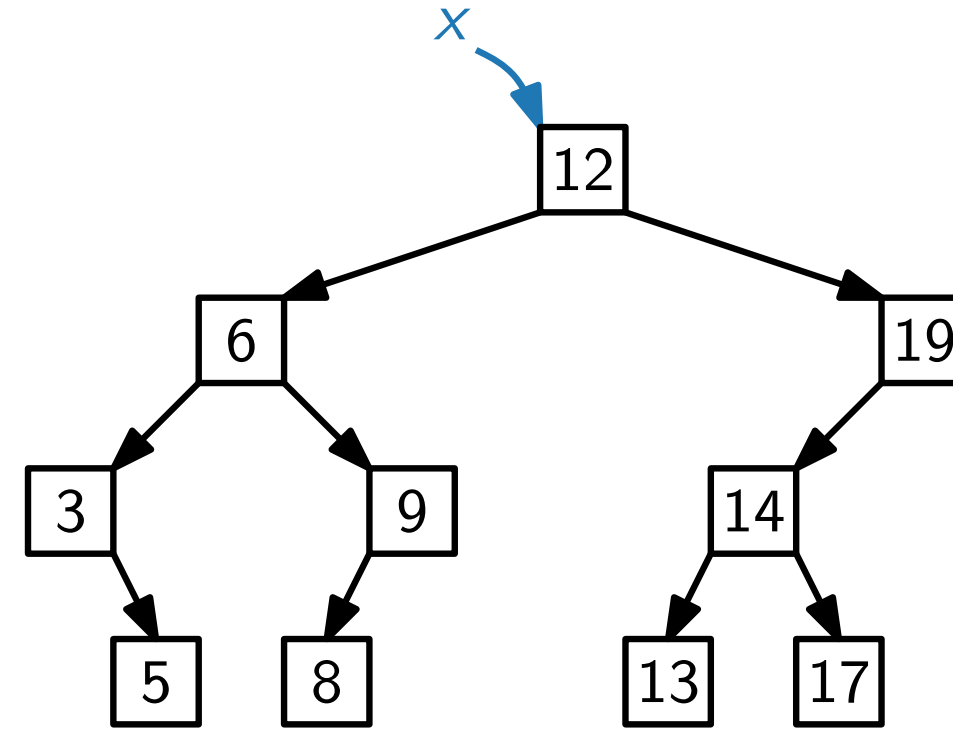
Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

|



INSERT(11)

Einfügen

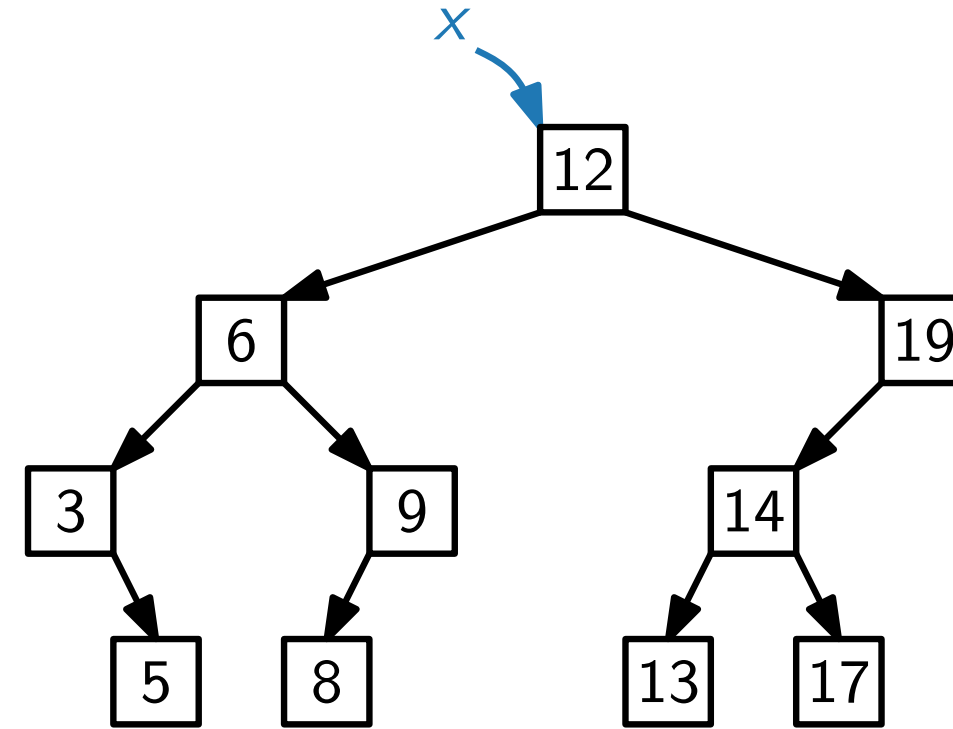
Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

$y = x$



INSERT(11)

Einfügen

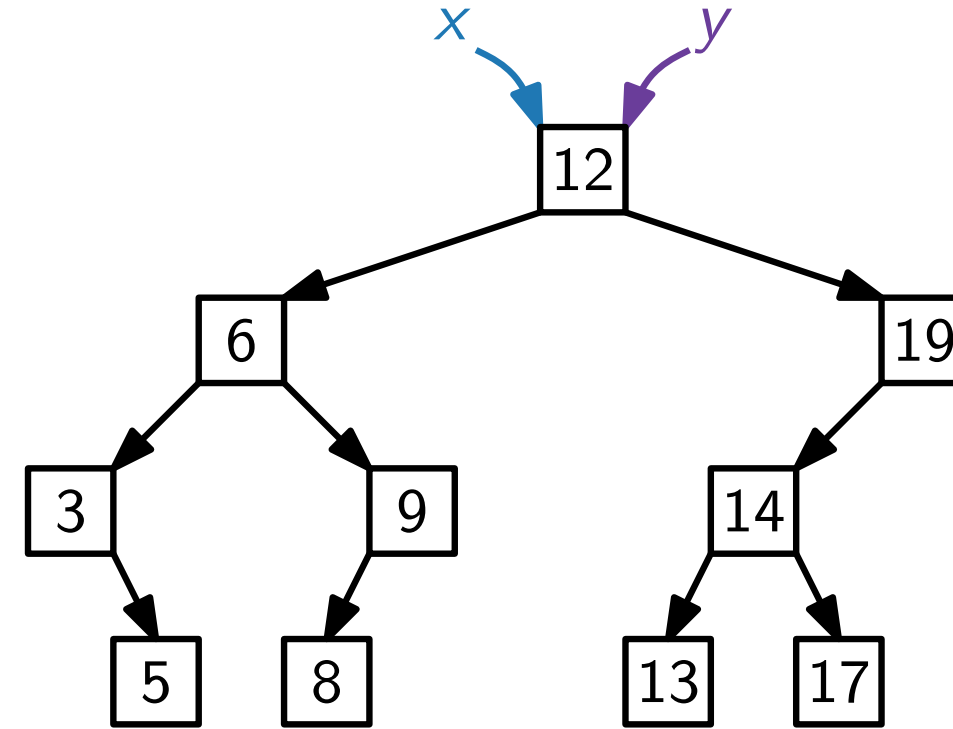
Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

$y = x$



INSERT(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

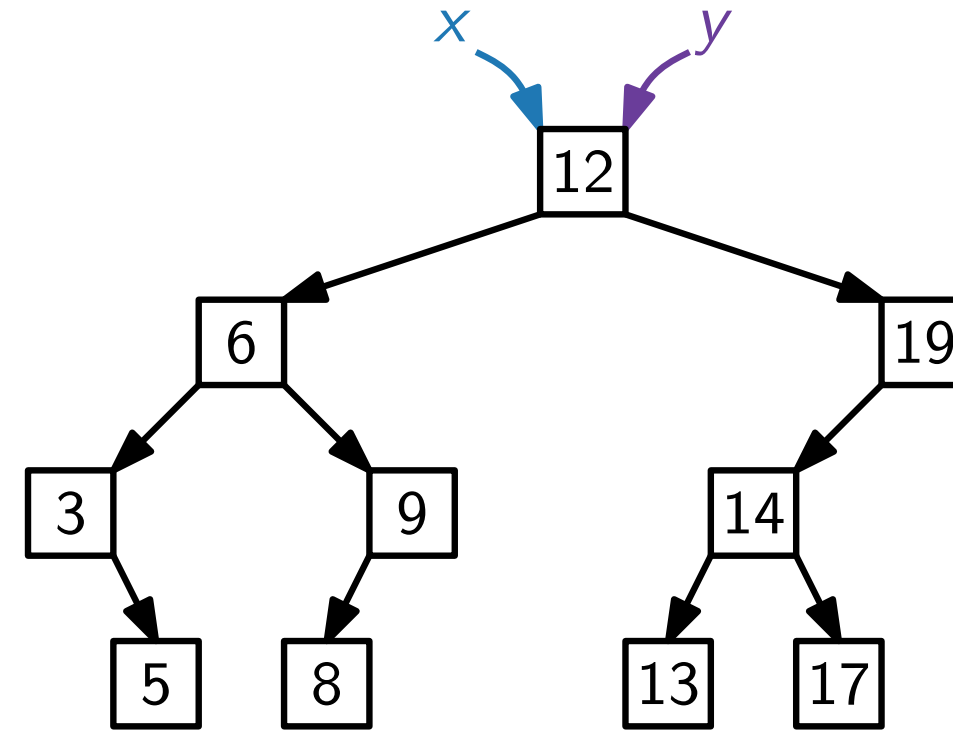
while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$



INSERT(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

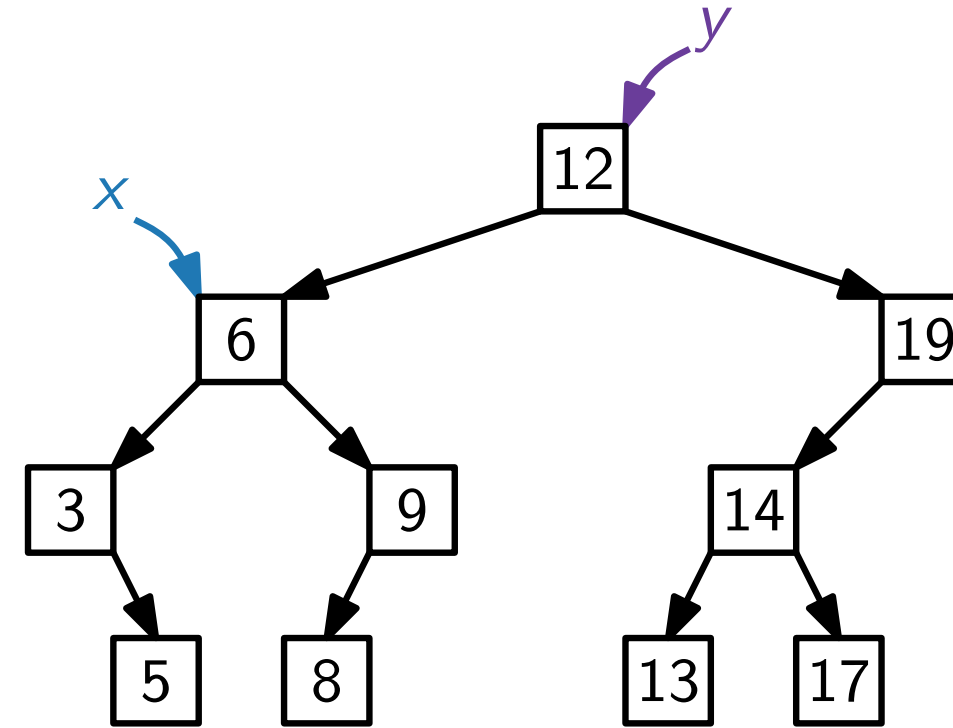
while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$



INSERT(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

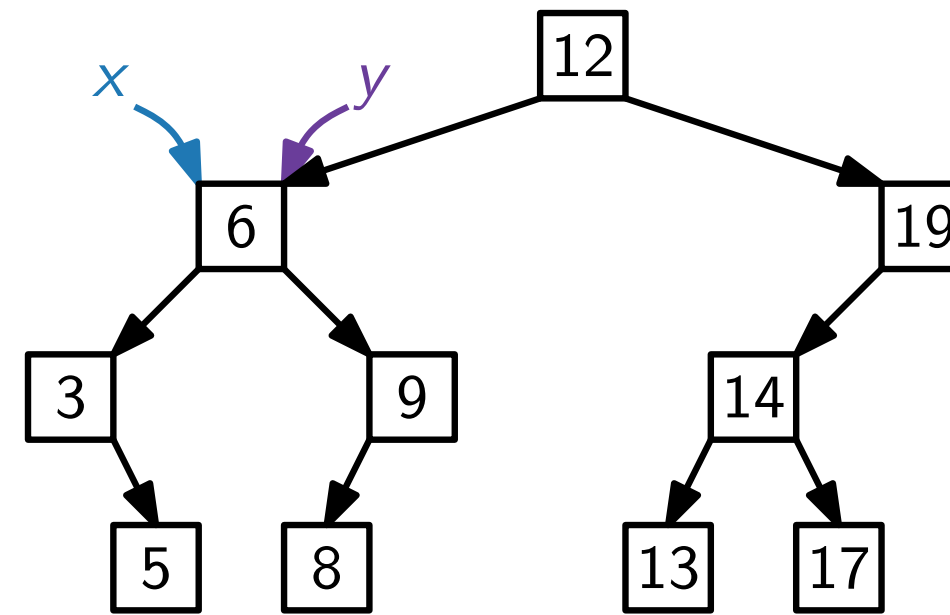
while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$



INSERT(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

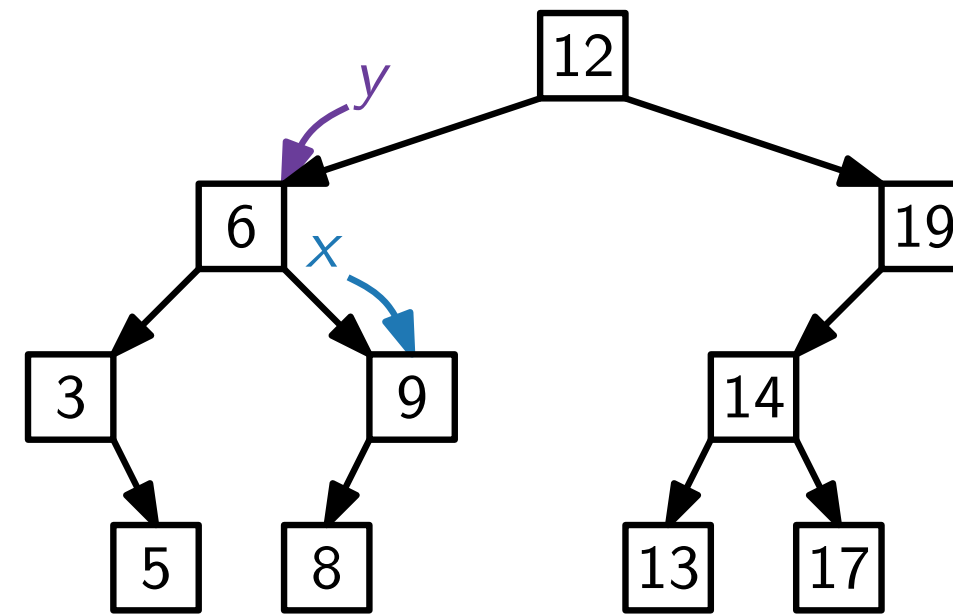
while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$



INSERT(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

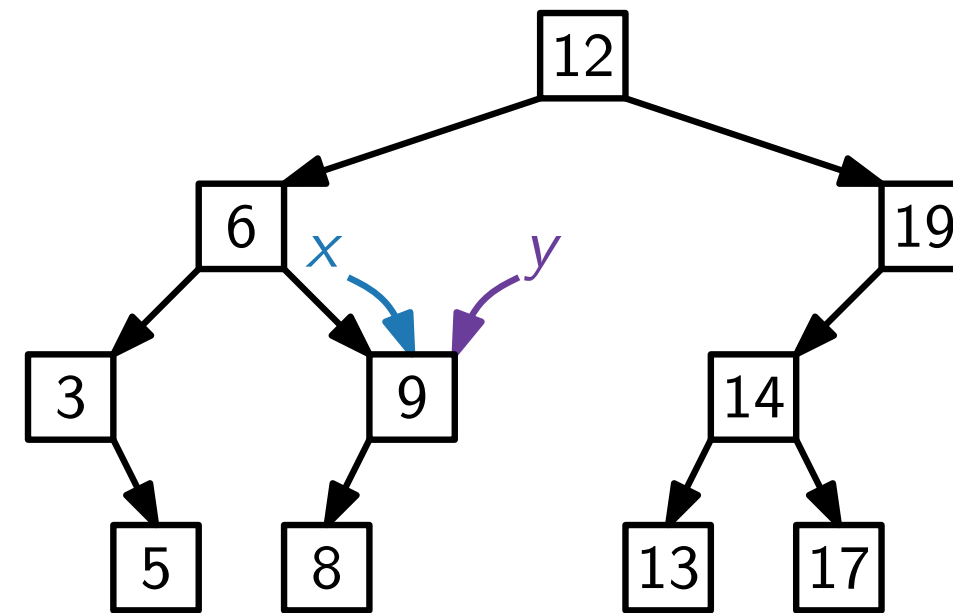
while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$



INSERT(11)

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

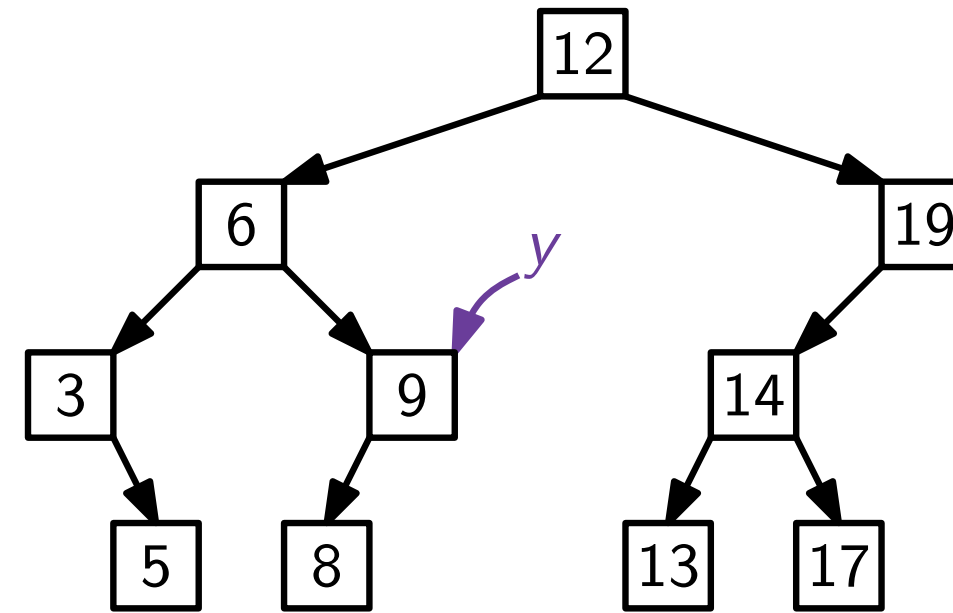
while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$



INSERT(11)

$x == nil$

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

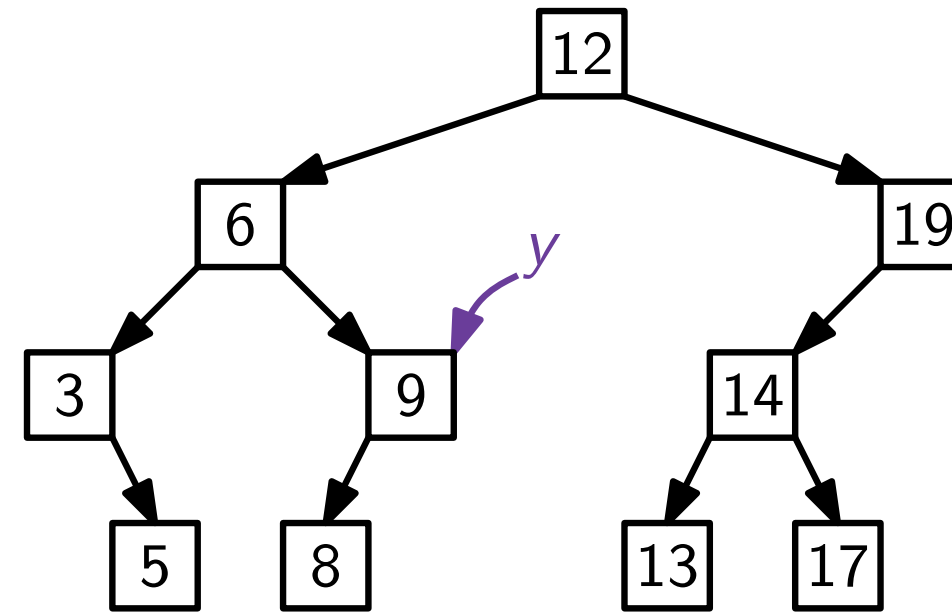
$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new Node}(k, y)$



INSERT(11)

$x == nil$

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

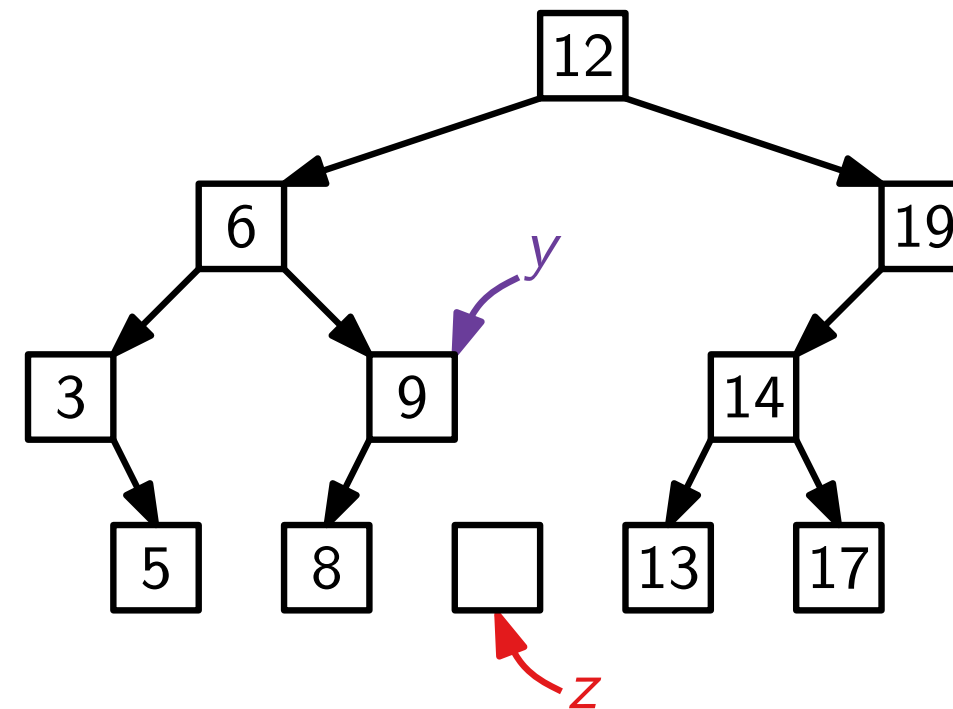
$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new Node}(k, y)$



INSERT(11)

$x == nil$

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

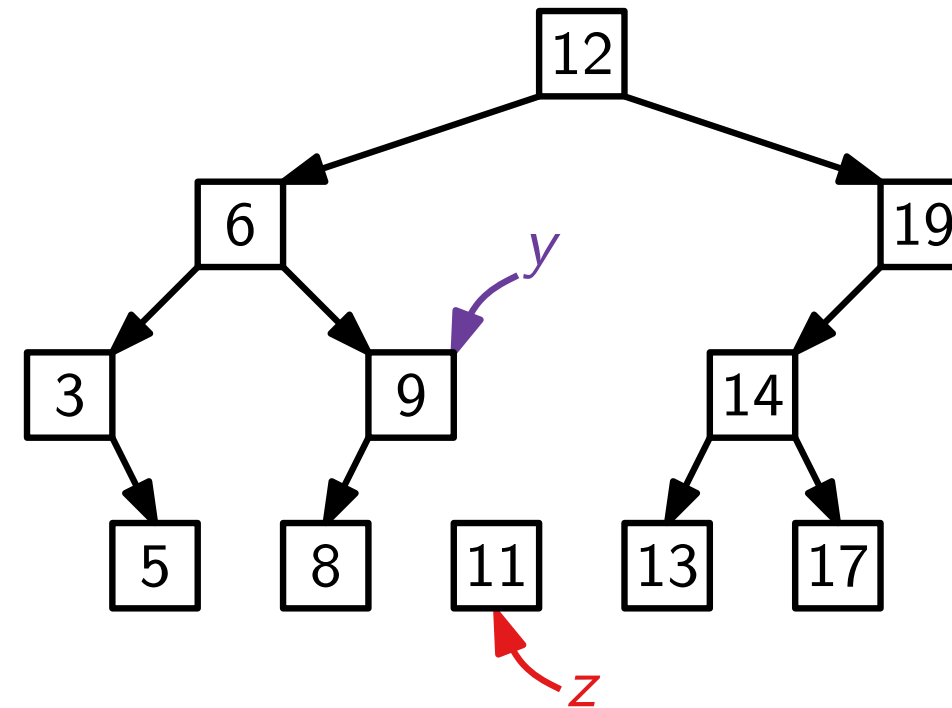
$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new Node}(k, y)$



INSERT(11)

$x == nil$

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

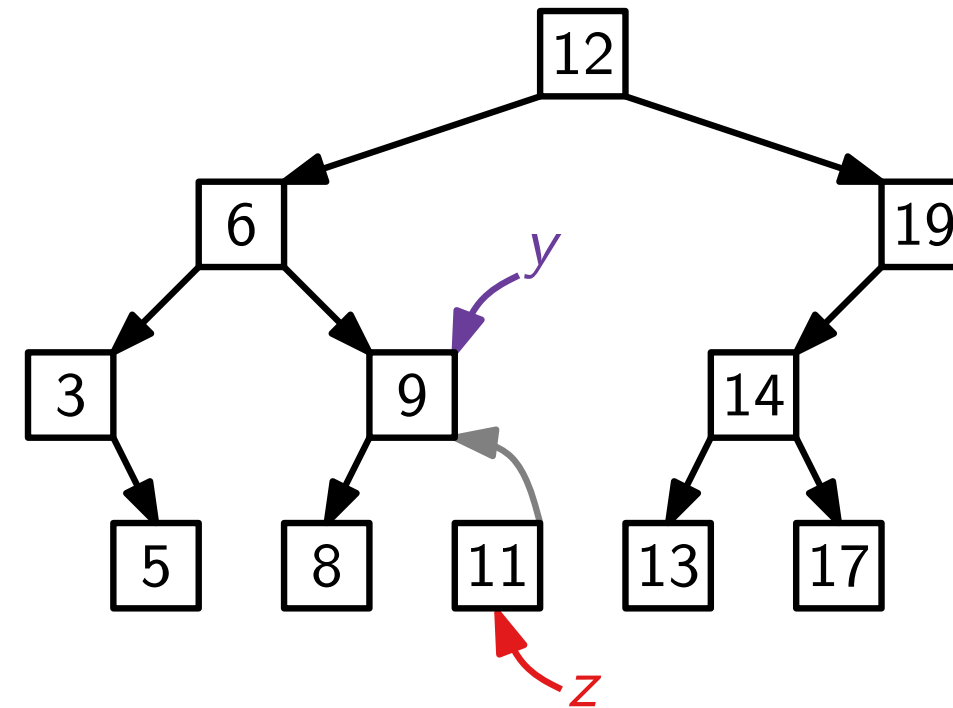
$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new Node}(k, y)$



INSERT(11)

$x == nil$

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

$y = x$

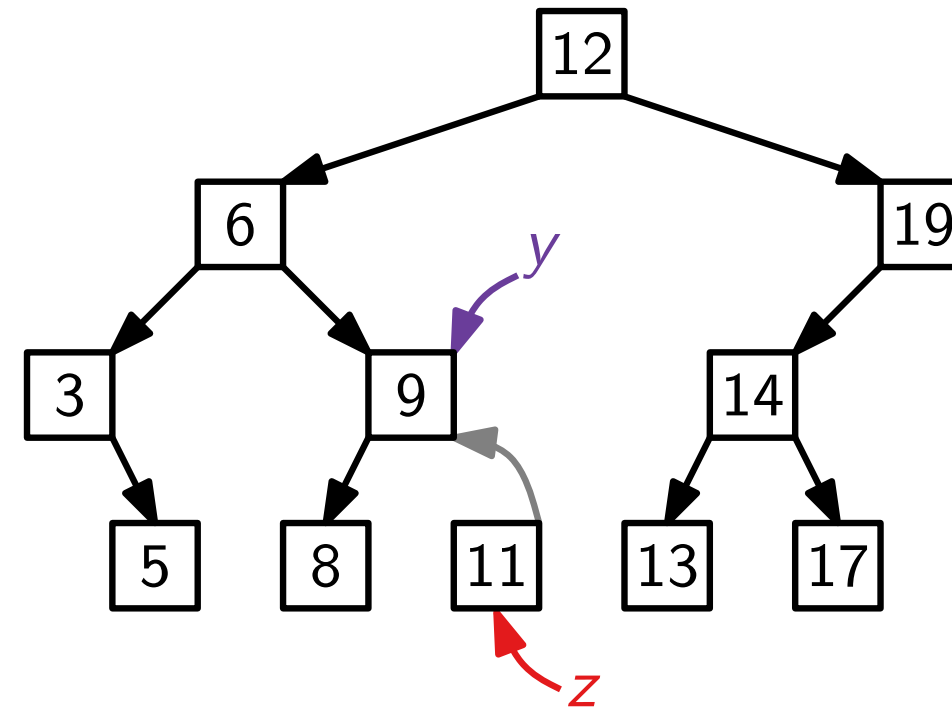
if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new Node}(k, y)$

if $y == nil$ **then** $root = z$



INSERT(11)

$x == nil$

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

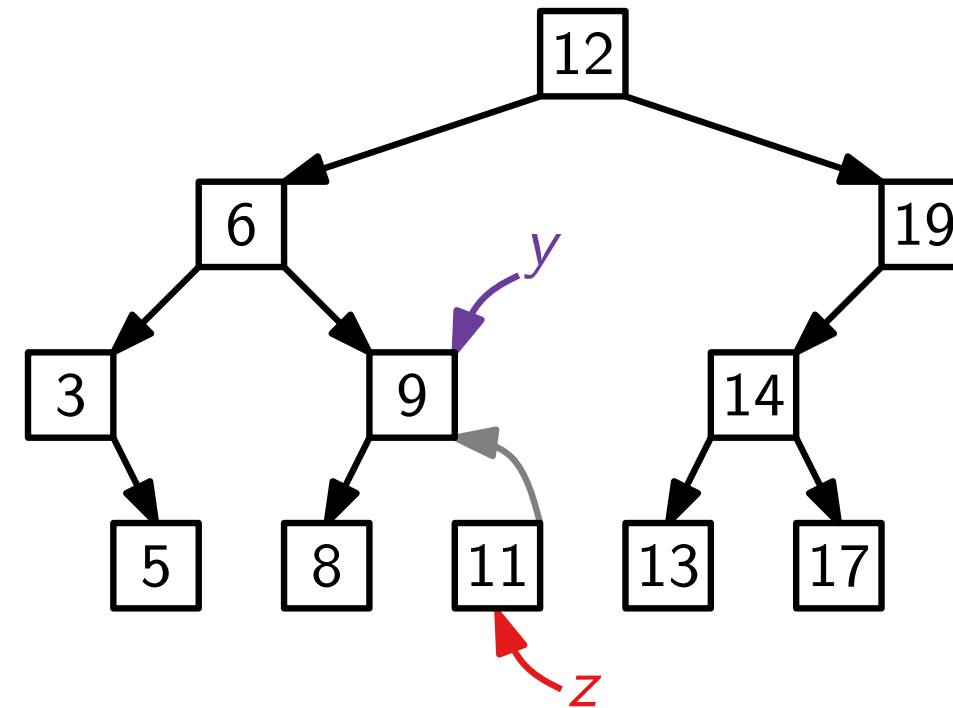
$x = x.left$

else $x = x.right$

$z = \text{new Node}(k, y)$

if $y == nil$ **then** $root = z$

else



INSERT(11)

$x == nil$

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

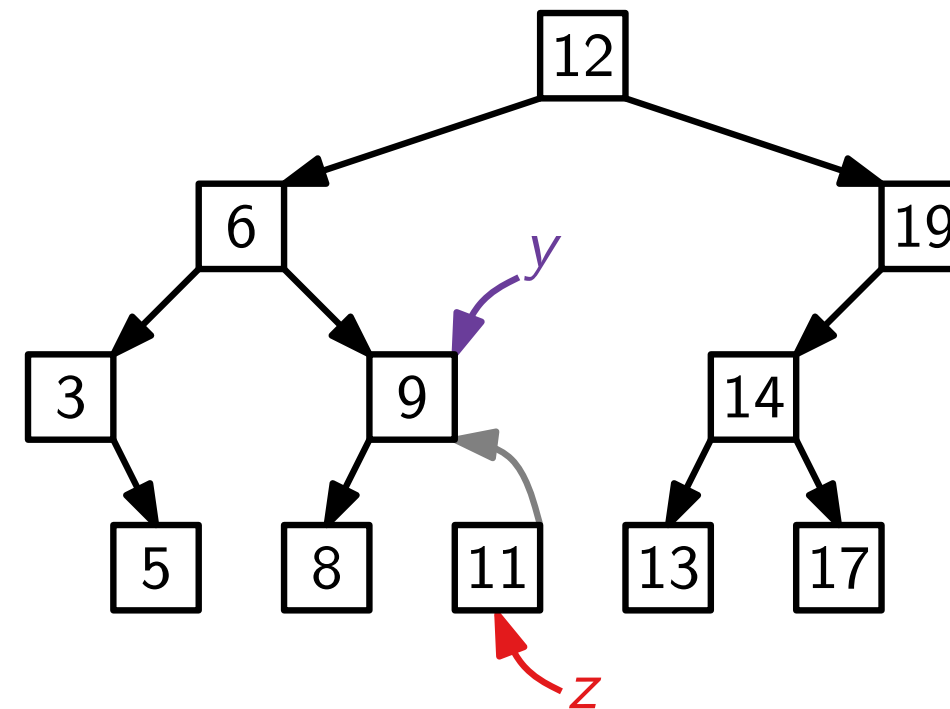
$z = \text{new Node}(k, y)$

if $y == nil$ **then** $root = z$

else

if $k < y.key$ **then** $y.left = z$

else $y.right = z$



INSERT(11)

$x == nil$

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

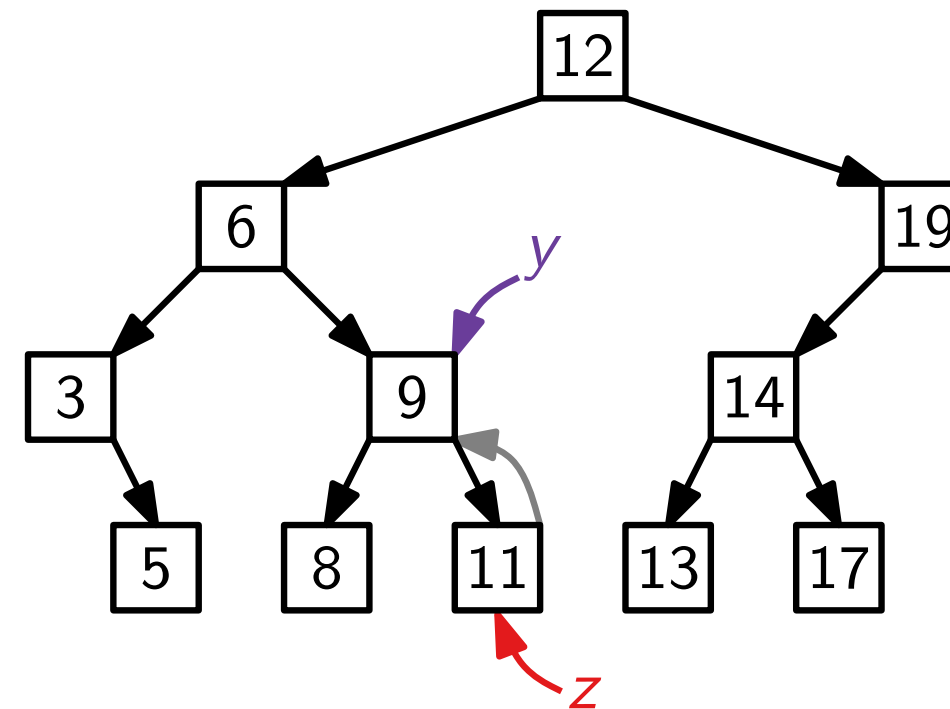
$z = \text{new Node}(k, y)$

if $y == nil$ **then** $root = z$

else

if $k < y.key$ **then** $y.left = z$

else $y.right = z$



INSERT(11)

$x == nil$

Einfügen

Node INSERT(key k)

$y = nil$

$x = root$

while $x \neq nil$ **do**

$y = x$

if $k < x.key$ **then**

$x = x.left$

else $x = x.right$

$z = \text{new Node}(k, y)$

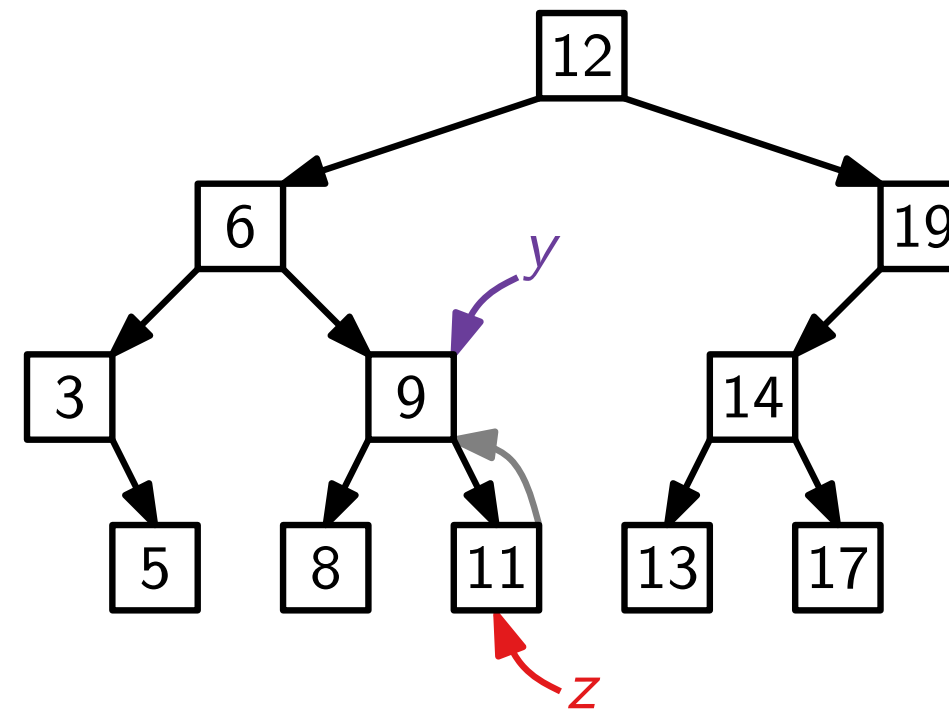
if $y == nil$ **then** $root = z$

else

if $k < y.key$ **then** $y.left = z$

else $y.right = z$

return z

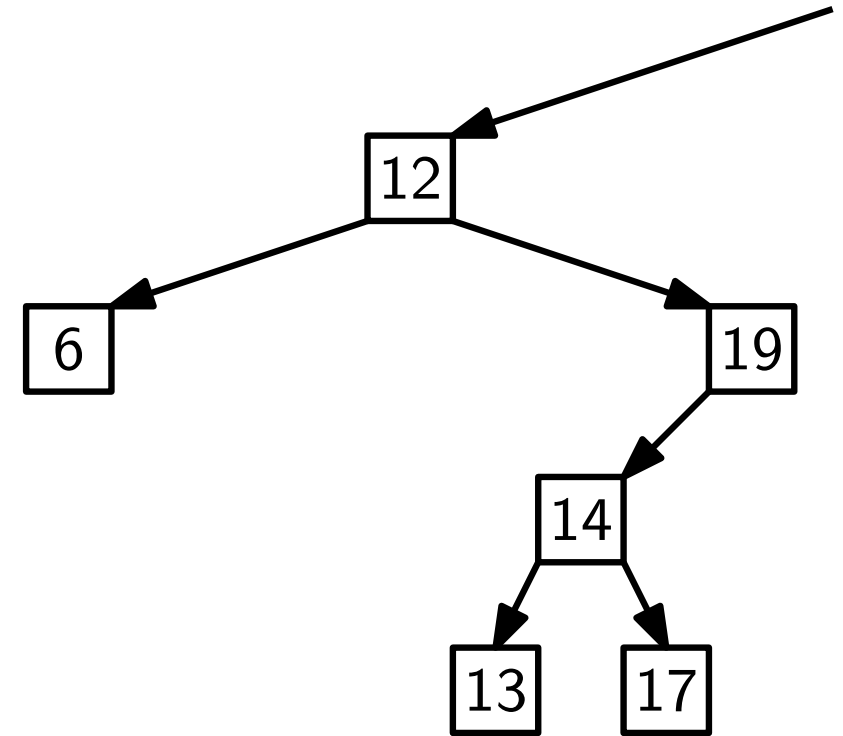


INSERT(11)

$x == nil$

Löschen

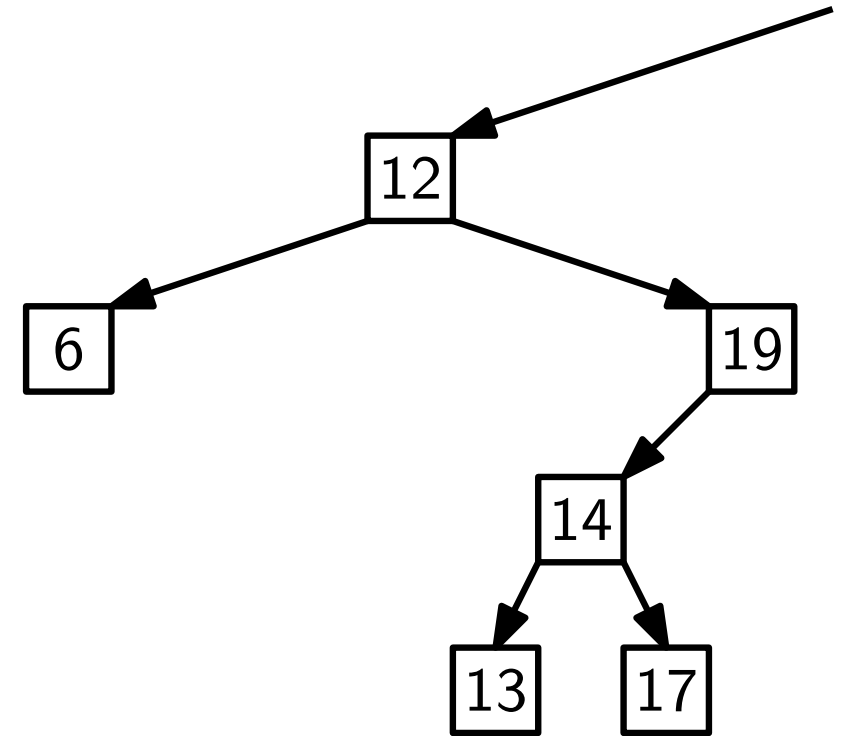
Sei **z** der zu löschende Knoten. Wir betrachten drei Fälle:



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

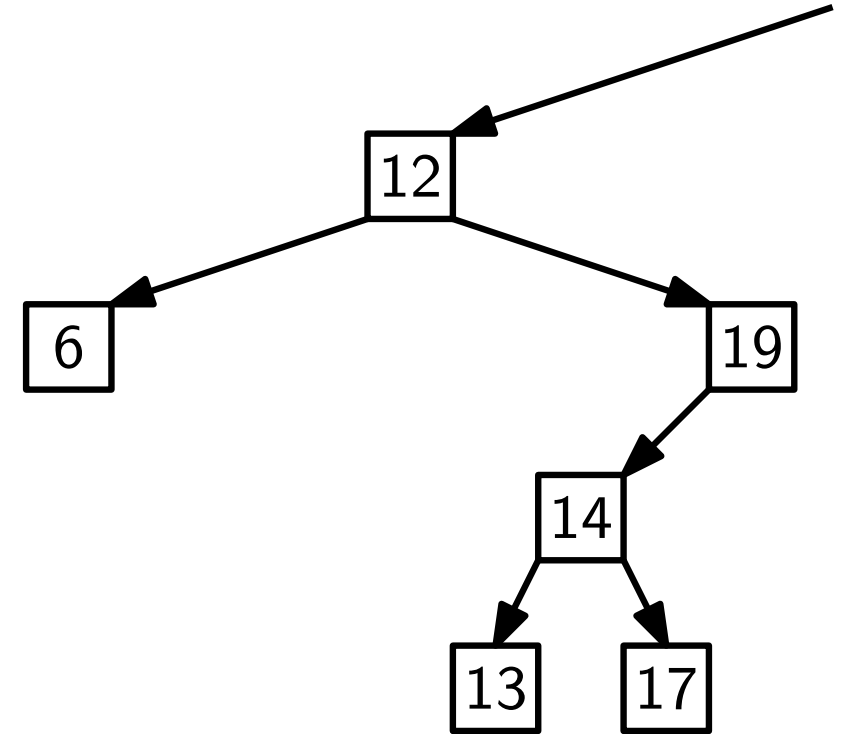


Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

2. z hat ein Kind x .



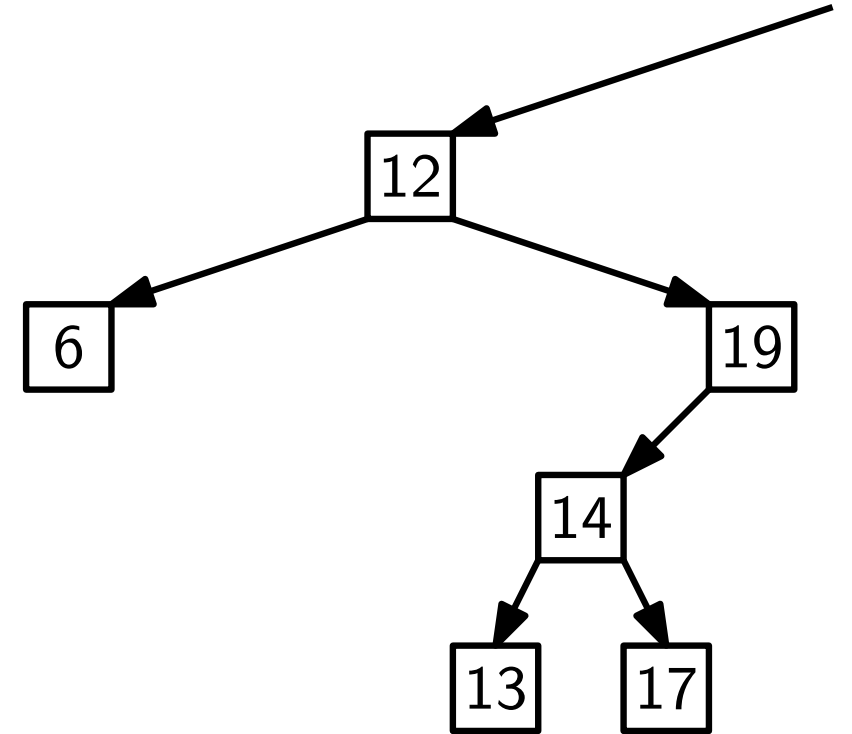
Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

2. z hat ein Kind x .

3. z hat zwei Kinder.



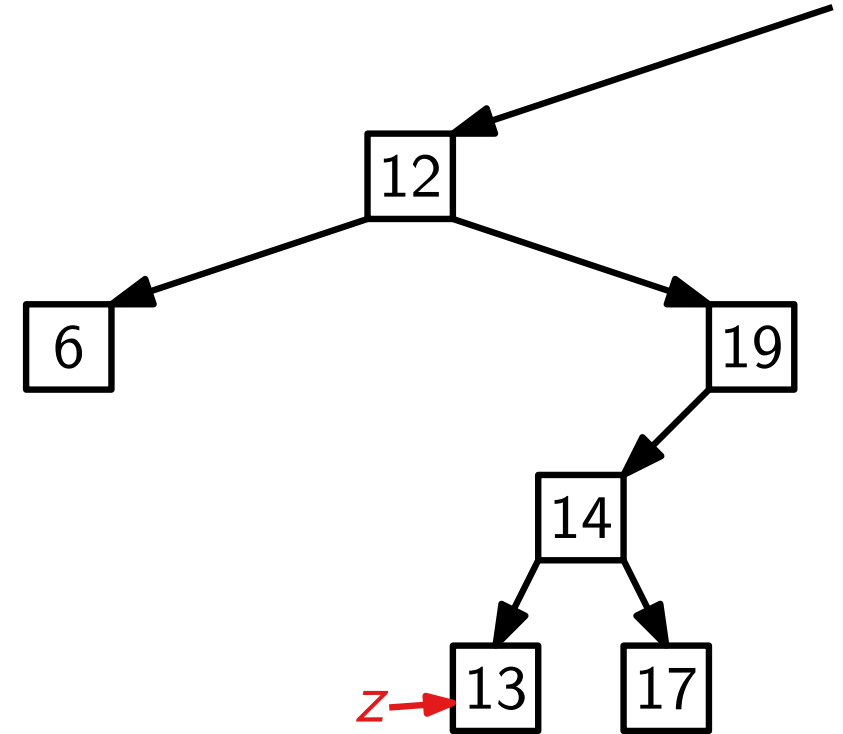
Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

2. z hat ein Kind x .

3. z hat zwei Kinder.



Löschen

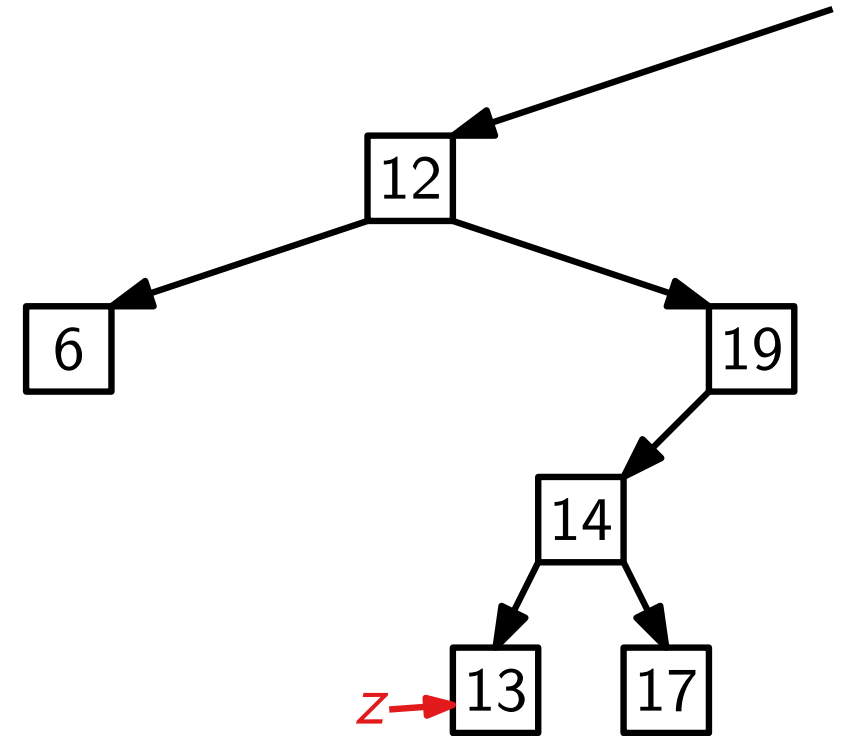
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

3. z hat zwei Kinder.



Löschen

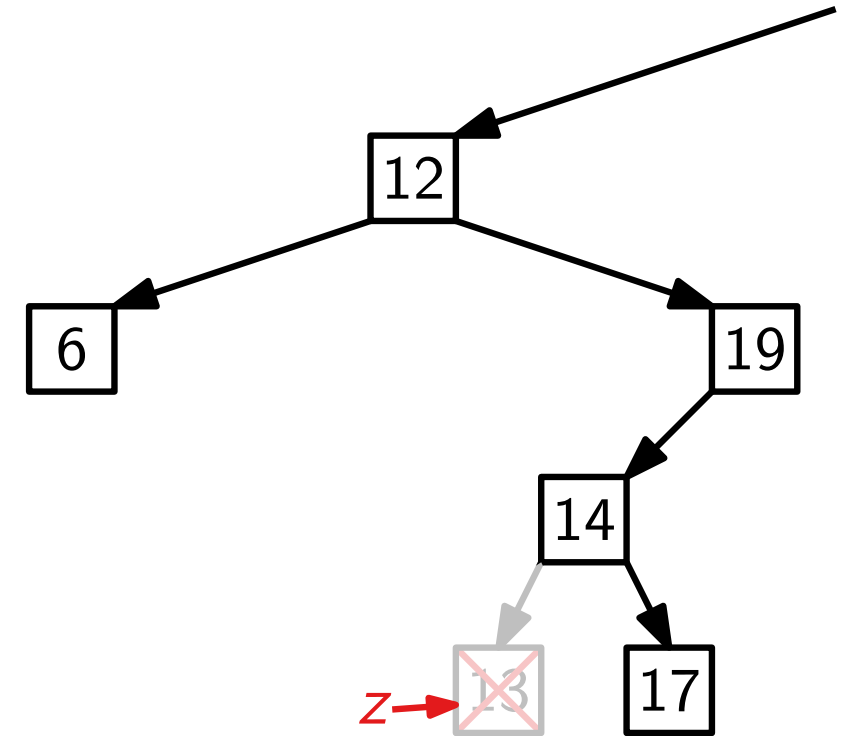
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

3. z hat zwei Kinder.



Löschen

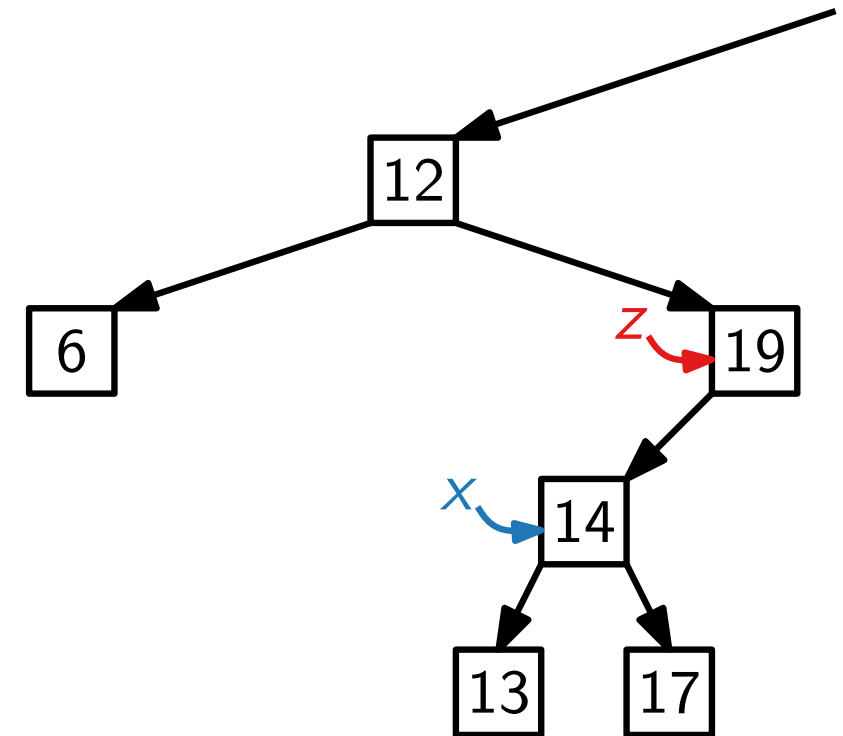
Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

3. z hat zwei Kinder.



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

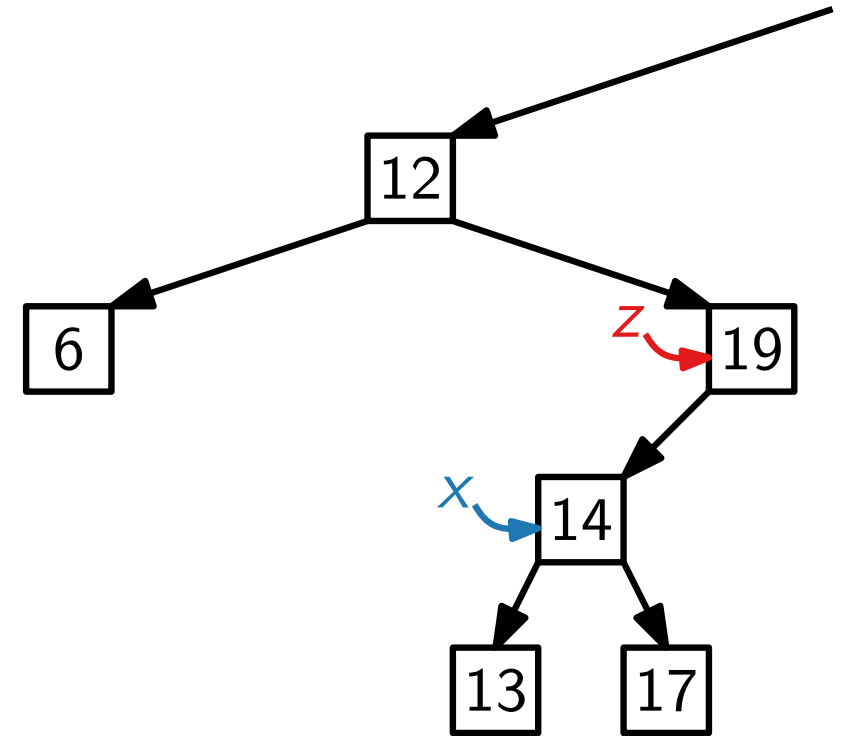
1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .
Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

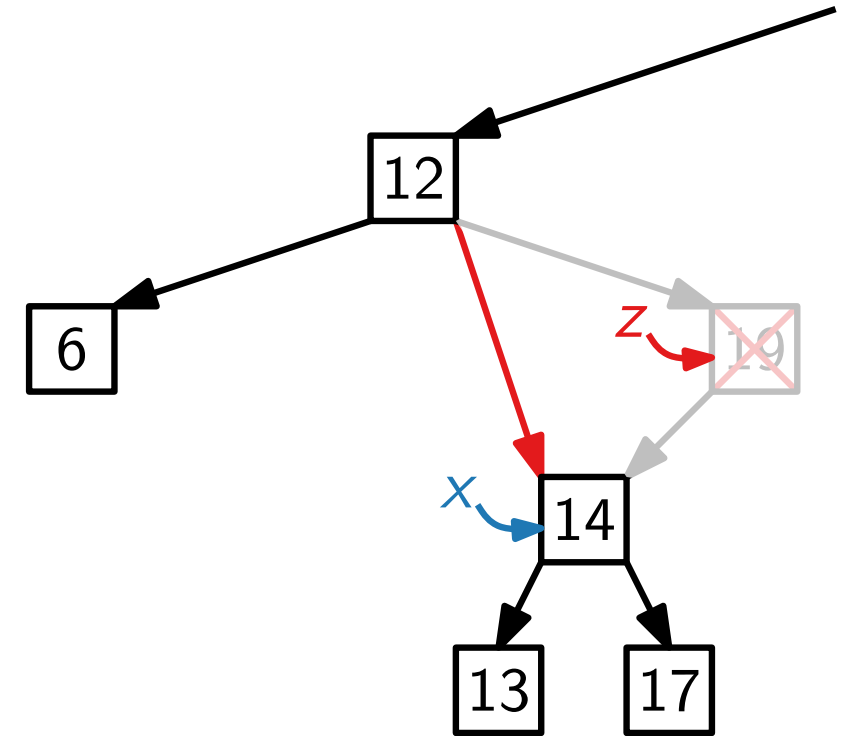
1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .
Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

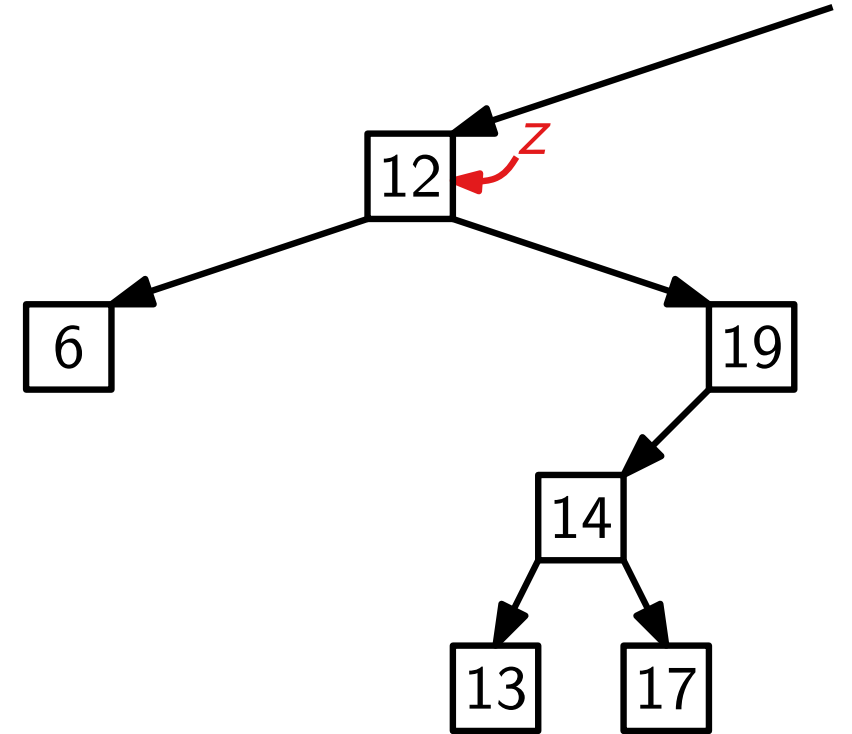
1. z hat keine Kinder.

Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .
Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

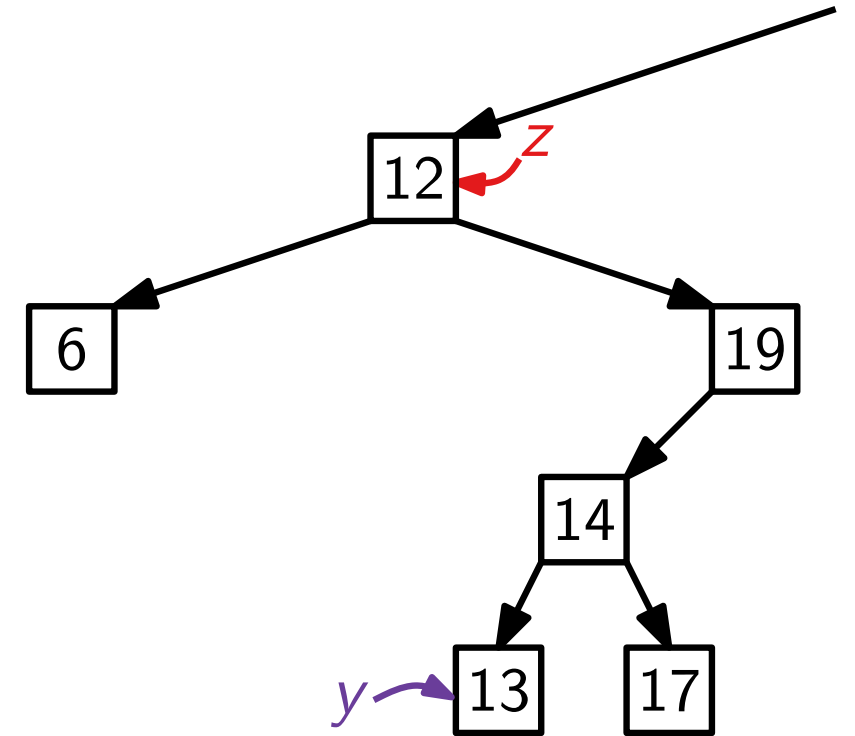
Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .
Setze $x.p = z.p$. Lösche z .

3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$ und $z.key = y.key$. Lösche y . (Fall 1 oder 2!)



Löschen

Sei z der zu löschende Knoten. Wir betrachten drei Fälle:

1. z hat keine Kinder.

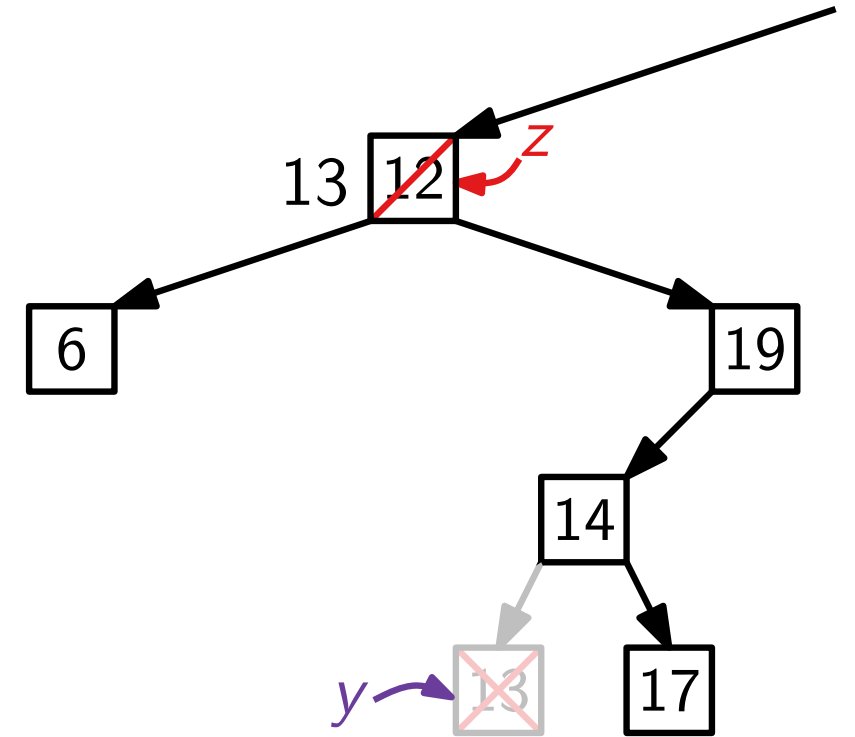
Falls z linkes Kind von $z.p$ ist,
setze $z.p.left = nil$; sonst umgekehrt. Lösche z .

2. z hat ein Kind x .

Setze den Zeiger von $z.p$, der auf z zeigt, auf x .
Setze $x.p = z.p$. Lösche z .

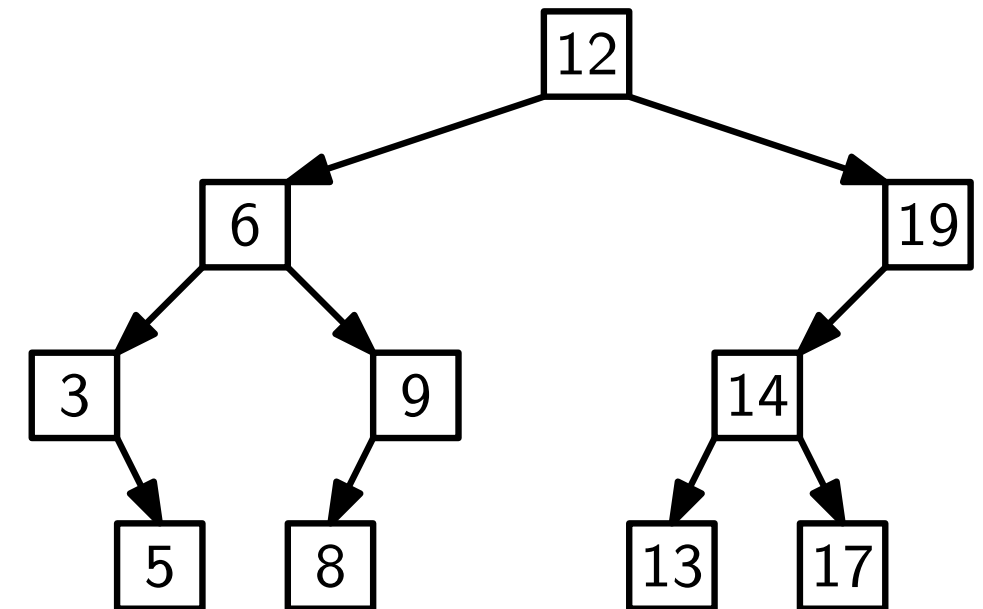
3. z hat zwei Kinder.

Setze $y = \text{SUCCESSOR}(z)$ und $z.key = y.key$. Lösche y . (Fall 1 oder 2!)



Zusammenfassung

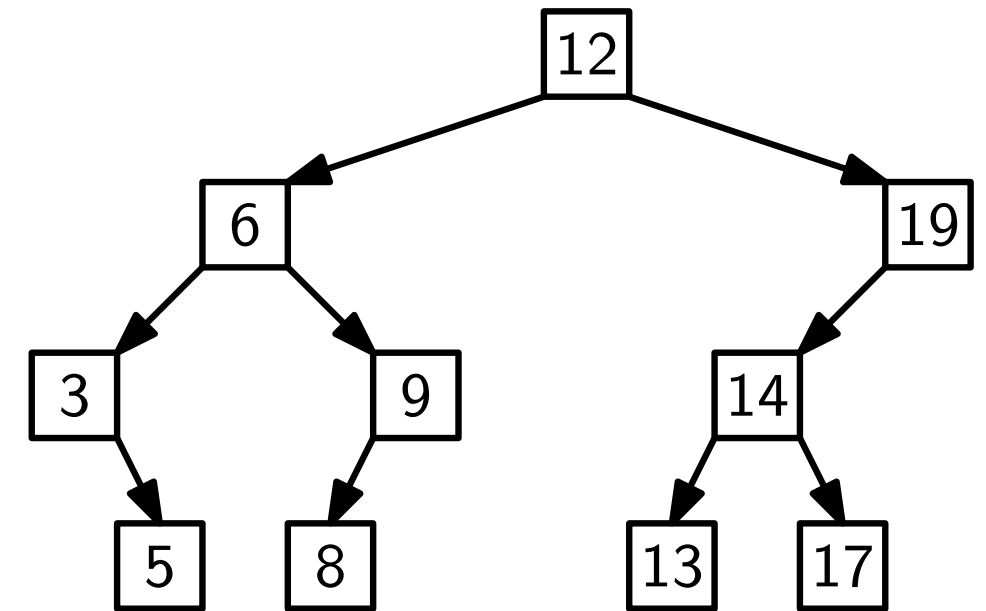
Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.



Zusammenfassung

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

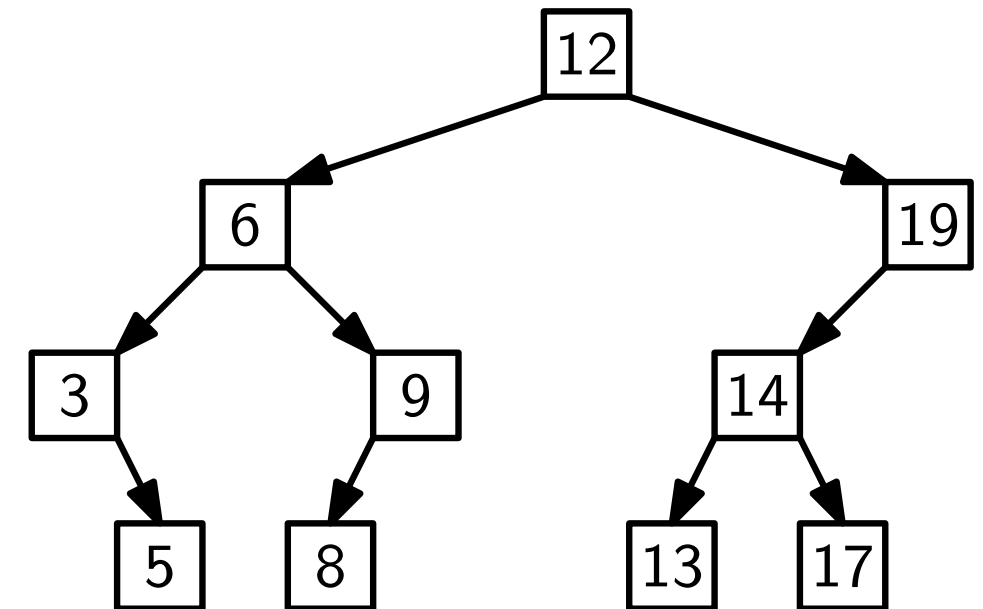
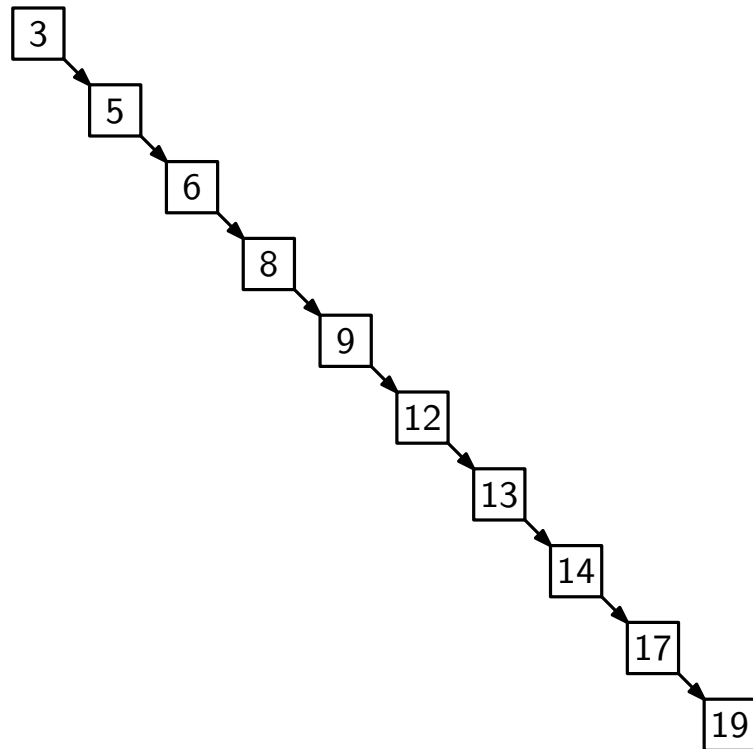
Aber:



Zusammenfassung

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

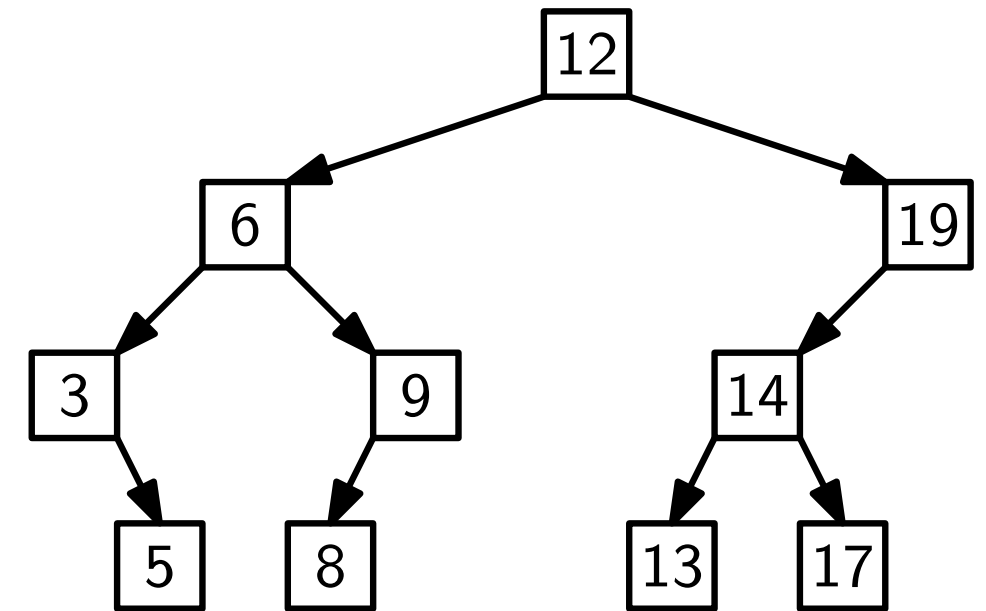
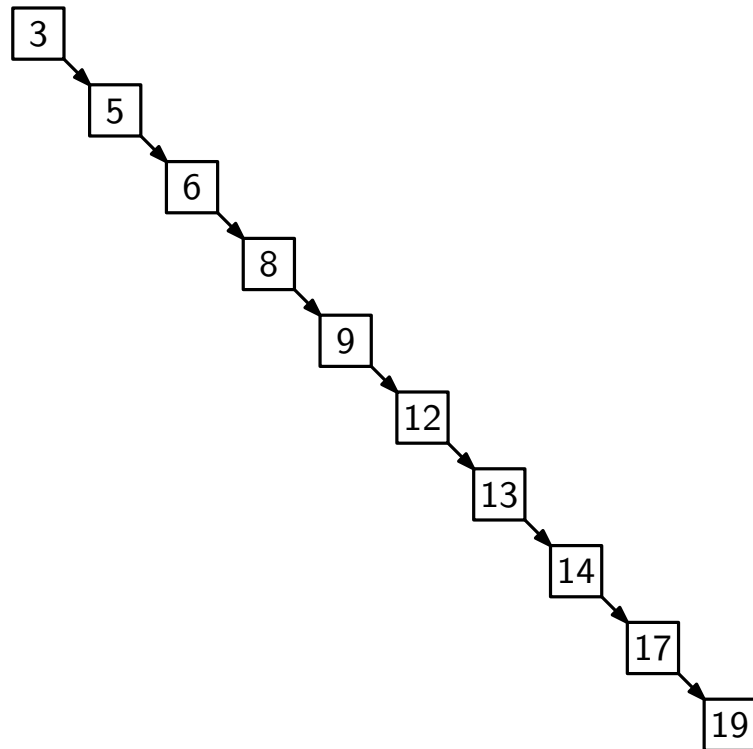
Aber:



Zusammenfassung

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

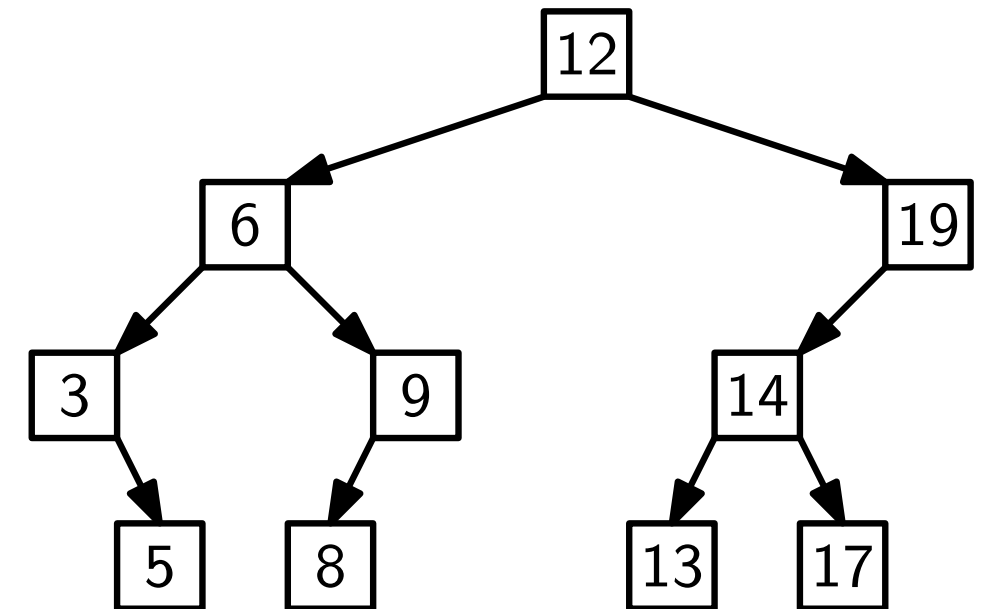
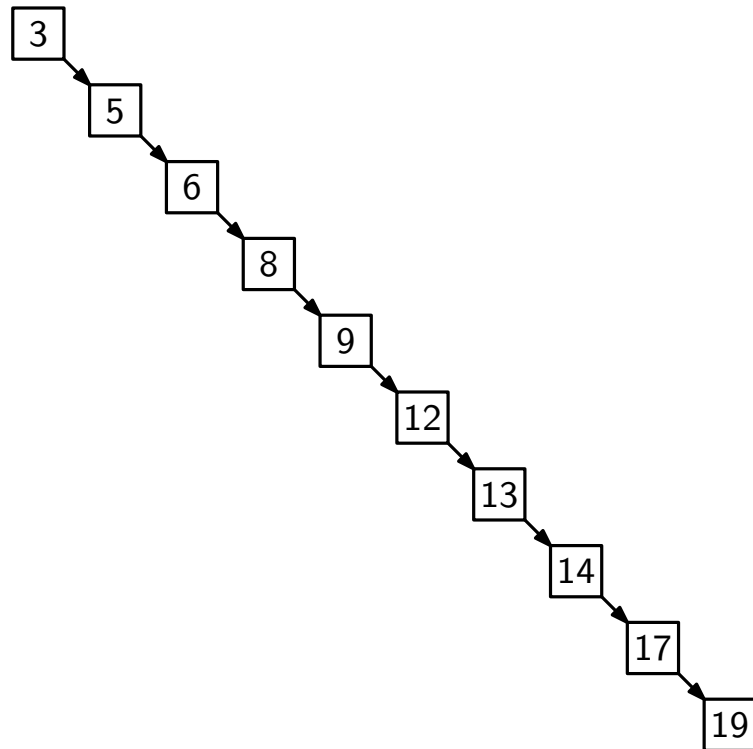


Zusammenfassung

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel:

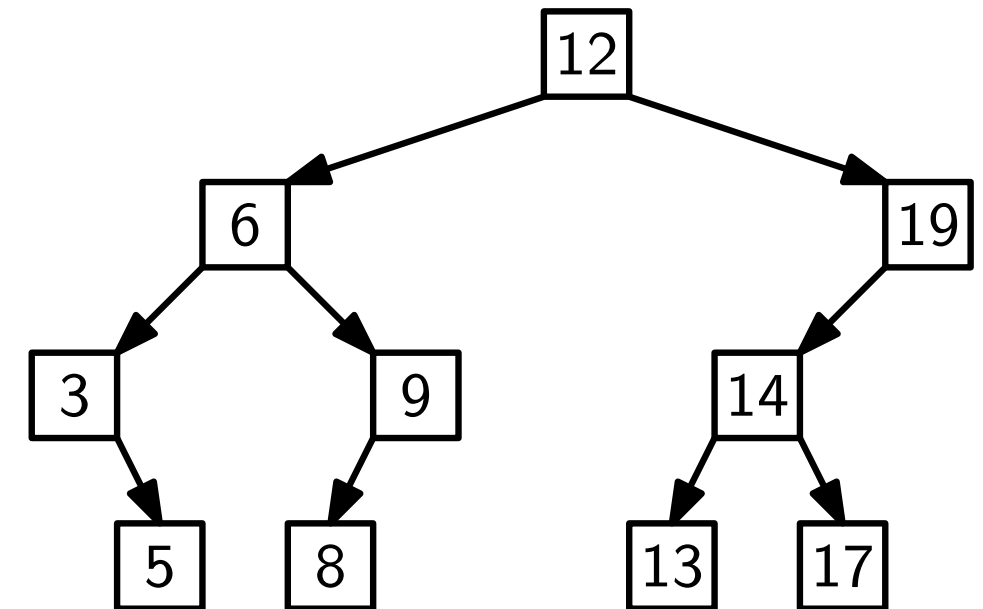
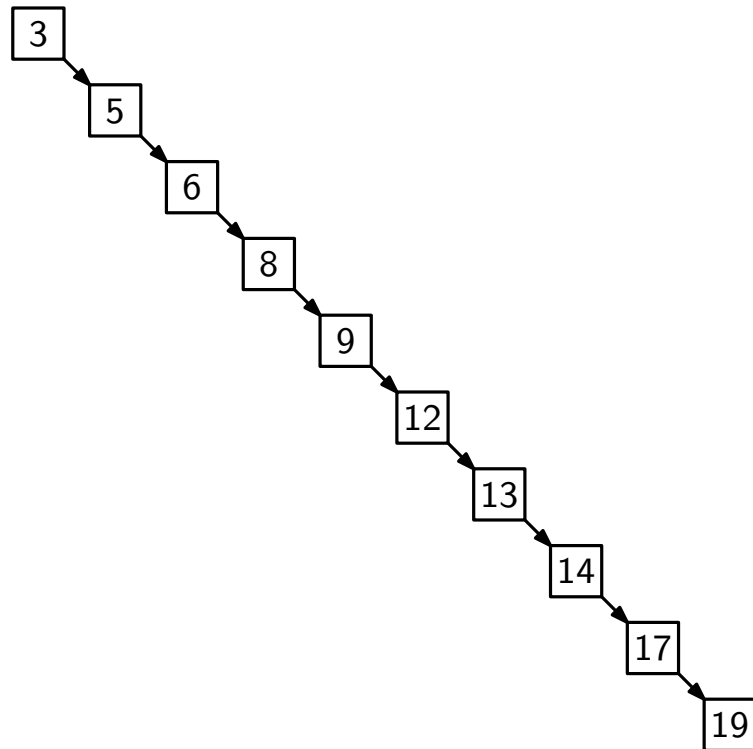


Zusammenfassung

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume **balancieren**



Zusammenfassung

Satz. Binäre Suchbäume implementieren alle dynamische-Menge-Operationen in $\mathcal{O}(h)$ Zeit, wobei h die momentane Höhe des Baums ist.

Aber: Im schlechtesten Fall gilt $h \in \Theta(n)$.

Ziel: Suchbäume **balancieren** $\Rightarrow h \in \mathcal{O}(\log n)$

